Departamento de Ciencias da Computación
e Tecnoloxías da Información

Facultade de Informática
UNIVERSIDADE DA CORUÑA

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

Aplicación multiplataforma para gestionar finanzas personales y cuentas conjuntas

Estudiante: Pablo Pérez Rodríguez
Director/a/es/as: Miguel Ángel Rodríguez Luaces
Alejandro Cortiñas Álvarez

A Coruña, 20 de junio de 2019

A mi familia, que siempre ha estado a mi lado, hasta estando lejos.

Agradecimientos

A mi familia, dándome todas las posibilidades aunque por momentos ni yo las quisiese.

A Yessica, que sin su apoyo diario habría sido imposible, y a la que ya se puede considerar familia.

A mis amigos, que han hecho estos años mucho más llevaderos.

Y a mis tutores, Álex y Miguel, porque me han motivado continuamente a mejorar y porque han contribuido en todas las facetas, mucho más allá de sus responsabilidades.

Resumen

El objetivo de este trabajo de fin de grado es desarrollar una aplicación que permita a sus usuarios introducir sus gastos e ingresos del día a día, ofreciéndoles información interesante en tiempo real sobre el estado de sus cuentas, como estadísticas de gasto, balances o listados de movimientos. También dará soporte para gastos compartidos, permitiendo a los usuarios de un grupo introducir pagos y transferencias indicando qué miembros participaron para posteriormente saldar deudas sin tener que hacer los cálculos a mano.

Para alcanzar este objetivo fue necesario en primer lugar realizar un análisis previo con el fin de conocer más en detalle los objetivos del proyecto, establecer una serie de requisitos y estudiar la viabilidad de los puntos clave que formarían parte del producto final. A continuación se llevó a cabo el diseño, desarrollo y prueba de las funcionalidades de la aplicación, y por último se terminó desplegando el proyecto en un servidor remoto, haciéndolo accesible desde cualquier sitio a través de Internet.

En el desarrollo se emplearon por una parte Java y Spring para producir un servidor que albergase toda la lógica de la aplicación y asegurase que los datos de un usuario sólo fuesen accedidos por él, y que ofreciese un servicio REST para ejecutar las operaciones disponibles. Por otro lado, se desarrolló un cliente Angular que, utilizando el servicio REST ofrecido, permitiese al usuario llevar a cabo las funciones disponibles. El cliente está basado en los principios de las *Progressive Web Apps*, por lo que permite trabajar sin conexión a Internet, recibir notificaciones sobre eventos importantes o instalar la aplicación en cualquier dispositivo.

El trabajo de fin de grado se gestionó siguiendo una metodología iterativa e incremental para el desarrollo software que dividió el todo el proceso en cinco iteraciones. Cada iteración comenzó con una reunión de planificación y que sirvió también como revisión de la iteración previa, durante cada una se llevaron a cabo las todas las tareas de análisis, diseño, implementación y pruebas. Al término de las mismas se generó un incremento con las funcionalidades desarrolladas en cada iteración y todas las que formaron parte de los incrementos previos.

Abstract

The objective of this end-of-degree project is to develop an application that allows users to introduce expenses and incomes in their day-to-day, offering them relevant information in real time about their accounts, such as expense statistics, balances or movements listings. It will also provide functionalities to record shared expenses, allowing users of a group to introduce payments and transfers providing information about which members participated in order to settle debts afterwards without having to perform calculations manually.

In order to achieve this goal, it was necessary first of all to perform an early analysis in order to understand the project's goals in detail, establish some requisites and study the viability of the key points that would be part of the final product. Next, the functionalities of the application were designed, developed and tested, and finally, the project was deployed to a remote server in order to make it reachable from anywhere through Internet.

In the development, Java and Spring where used to produce a server that would contain all the logic in the application and ensured the data from a user would only be accessed by them, and that would offer a REST service to execute the avaiable operations. On the other hand, an Angular client was developed that uses the REST service offered to allow users perform the available functionalities. The client is based on the *Progressive Web Apps* principles, so it allows to work offline, receive notifications about important events or install the application in any device.

The end-of-degree work was managed following an iterative and incremental methodology for software development which divided the whole process into five iterations. Each iteration started with a planning meeting which was also used as a revision of the previous iteration, and in every one of them, the tasks of analysis, design, implementation and testing were conducted. After each of them, an increment was generated containing all the functionalities developed in each iteration, and all the ones that were part of the previous increments.

Palabras clave:

- Angular
- Spring
- *Progressive Web App*
- Gestión de finanzas
- *Offline*
- Notificaciones *push*
- Servicio REST

Keywords:

- Angular
- Spring
- Progressive Web App
- Accounting
- Offline
- Push notifications
- REST service

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1 Motivación

En un panorama donde el uso del móvil está cada vez más extendido, se dispone en todo momento de un dispositivo preparado para llevar una contabilidad de ingresos y gastos de forma rápida y sencilla. Aún así, por ahora, éste es un campo en el que aún se pueden introducir mejoras.

Este proyecto presenta una oportunidad de desarrollar una solución que no se ofrece hasta el momento. Si bien es cierto que ya existen aplicaciones para la gestión de finanzas personales o grupales, no hay ninguna de código abierto que proporcione al usuario un mínimo de funcionalidades (sincronización en la nube, uso móvil...) que le permitan considerarla sobre las alternativas no abiertas. Y entre este tipo de soluciones, nos encontramos siempre con aplicaciones que se ofrecen con planes de pago para disfrutar de todas las funcionalidades, y que suelen contener anuncios.

El proyecto propuesto también presenta una serie de desafíos al desarrollador. Pese a haber sido diseñado como una aplicación web, ha sido concebido con una serie de características, como una interfaz muy agradable y fácil de usar desde dispositivos móviles, o funcionalidad sin conexión, que son más propios de aplicaciones nativas que web, pero que se han considerado como necesarios para que la solución ofreciese al usuario la mejor experiencia de uso posible.

El objetivo principal del trabajo es poner a disposición de cualquier usuario una herramienta que le permita gestionar sus gastos e ingresos personales desde el dispositivo que elija, ofreciendo una experiencia agradable, y de fácil uso. Además, se complementará con las funcionalidades de cuentas en grupos, que estarán conectadas con las individuales, dotando al usuario de un control total sobre sus gastos.

1.2 Objetivos

Para poder ofrecer todo esto, se tendrán que cumplir los siguientes objetivos:

- Disponer de una **aplicación de finanzas** que permita al usuario registrar sus ingresos y gastos, ofreciéndole información relevante como listados de movimientos, estadísticas y categorías de gasto. También permitirá gestionar cuentas conjuntas, facilitando el cálculo de deudas entre los miembros de un grupo. Se conectarán los gastos grupales con los individuales para ofrecer al usuario información más detallada del estado de sus cuentas.
- Proporcionar un sistema de **registro de usuarios**, a través del cual se podrán crear cuentas en la aplicación para que los datos de cada usuario sean accesibles sólo por él. Además, este sistema proporcionará una forma de invitar a otros usuarios a unirse a grupos para gestionar pagos conjuntos.
- **Complementar las funcionalidades** de la aplicación con añadidos que incrementen el valor de ésta. Podrá ser usada desde múltiples tipos de dispositivos, permitirá almacenar operaciones en entornos sin conexión y ofrecerá una experiencia de uso intuitiva y agradable, pudiendo realizar operaciones de forma rápida de modo que no supongan una pérdida de tiempo para el usuario.
- Desarrollar para ello una **aplicación web** que podrá ser usada en cualquier navegador. Además, basándose en el paradigma de las ***Progressive Web Apps***, permitirá ser instalada y usada sin una conexión estable a Internet.
- Producir una **solución segura y fiable**, esto es, que asegure al usuario que no se producirán pérdidas de información en sus movimientos y que los cálculos que se obtengan serán correctos y coherentes con los datos almacenados.
- Asegurar un **producto de calidad**, que no tenga *bugs* conocidos y ofrezca una experiencia de uso sin errores. Para ello, se tendrán que llevar a cabo pruebas en todo el proceso de desarrollo.

Capítulo 2

Fundamentos tecnológicos

2.1 Estado del arte

Teniendo en cuenta los objetivos detallados en el capítulo inicial, hemos buscado soluciones existentes que cubran de alguna manera las necesidades expresadas, con el objetivo tener una perspectiva de qué ofrecen y cómo complementarlo.

Una de las herramientas de gestión de finanzas personales más descargadas es *CoinKeeper* [5]. Esta aplicación permite al usuario registrar y clasificar sus gastos de forma fácil e intuitiva para mostrarle información relevante del estado de sus cuentas. La aplicación basa su funcionalidad en tres elementos: fuentes de ingreso, como *sueldo* o *beca*; monederos, como *cuenta bancaria* o *efectivo*; y categorías de gasto, como *entretenimiento*, *transporte*, *compra*... En la Figura 2.1 se puede observar cómo se almacena un gasto en esta aplicación, arrastrando un monedero sobre una categoría de gasto. El proceso es el mismo para realizar un ingreso y una transferencia, arrastrando una fuente de ingreso sobre un monedero, y un monedero sobre otro, respectivamente. Esta aplicación, pese a ser gratuita, tiene diversas funcionalidades que sólo están accesibles para aquellos usuarios que paguen una cuota mensual, por lo que en la versión básica, entre otras limitaciones, hay un máximo de categorías que puede crear un usuario, lo que reduce sus posibilidades de uso.

En cuanto a la gestión de cuentas grupales, hay varias aplicaciones que nos permiten ir registrando los pagos de cada miembro de un grupo para después saber quién le debe dinero a quién. Dos de las aplicaciones más populares son SplitWise [6] y SettleUp [2]. Aunque visualmente sean diferentes, la idea detrás de ellas es la misma, almacenar gastos que afecten a grupos de personas. Son especialmente útiles para, por ejemplo, llevar las cuentas de un viaje en grupo o de un piso compartido, almacenando cada integrante los gastos que ha pagado y que no sólo le repercuten a él (unos billetes de avión, una reserva de un apartamento, el recibo del gas, etc.) y de esta forma llevar la cuenta de cuánto gasta cada uno, como se puede ver en la Figura 2.2. Este tipo de aplicaciones permite, en cualquier momento, establecer las deudas

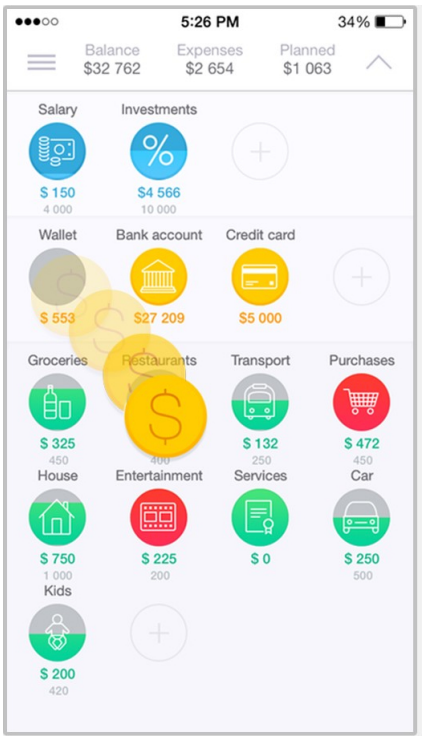

Figura 2.1: Almacenamiento de un gasto en CoinKeeper [1]

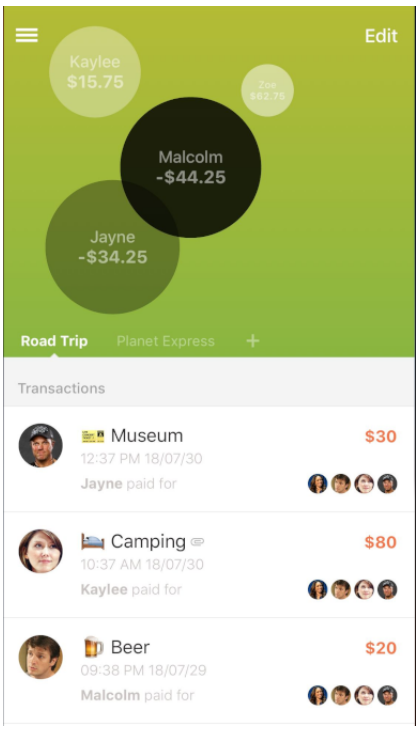

Figura 2.2: Vista general en SettleUp [2]

entre los miembros de la forma más eficiente, ahorrando el tedioso proceso de calcularlas a mano. Al igual que pasaba con CoinKeeper, pese a ser gratuitas, se complementan con pagos que aumentan sus funcionalidades.

En cuanto a la tecnología que emplean, contrariamente a los objetivos de este proyecto, todas las soluciones analizadas dan soporte al uso en dispositivos móviles a través de aplicaciones nativas. Frente a una alternativa web, tienen las desventajas de que el usuario estará obligado a descargar la aplicación a la hora de probarla y que tendrá que descargarla sí o sí en todos aquellos dispositivos en los que la quiera usar. Esto es algo en lo que el estándar *PWA* detallado en la Sección 7.3 destaca.

Concluyendo este estudio, hemos detectado que ninguna de las aplicaciones existentes da soporte completo al objetivo marcado para este proyecto, ya que se centran en las finanzas personales o colectivas por separado, no ambas a la vez. Además, estas soluciones no son de código abierto y contienen planes de pago y anuncios.

Aún así, han servido de inspiración para la realización de este proyecto, extrayendo grandes ideas de cada una de las aplicaciones. De *CoinKeeper* nos hemos quedado con su facilidad de uso y flexibilidad, adaptando el modelo de fuentes de ingreso, monederos y categorías de gasto a este proyecto e intentando desarrollar una forma de realizar movimientos como la que se ilustra en la Figura 2.1. Por otra parte, de *SettleUp* hemos adaptado su algoritmo de cálculo de deudas, explicado en detalle en la Sección 6.1.1.

2.2 Tecnologías utilizadas

En esta sección se enumerarán las tecnologías que han formado parte del desarrollo de este proyecto, junto con una descripción muy breve de la aportación de cada una:

- **Spring**. *Framework* Java que facilita el desarrollo de aplicaciones empresariales, desde su creación hasta su despliegue [7].
- **Hibernate**. Mapeador objeto-relacional para Java que ha facilitado las operaciones que implicaron un acceso a base de datos [8].
- **Lombok**. Librería para Java que, a través de anotaciones, genera código como *getters*, *setters* o *constructores*, reduciendo el tiempo de codificación y generando código más limpio [9].
- **Angular**. Framework para desarrollar aplicaciones de lado cliente [10].
- **Angular Material**. Librería desarrollada por Angular con componentes que siguen las directrices de diseño material [11] y que permite crear de forma más fácil interfaces atractivas y optimizadas para dispositivos móviles [12].

- **RxJS**. Librería JavaScript que permite, a través de un enfoque funcional y el uso de los patrones Observador e Iterador, desarrollar funciones asíncronas, mejorando el funcionamiento ofrecido por Angular [13].
- **TypeScript**. Superconjunto de JavaScript al que añade, entre otras cosas, tipado estático. Es el lenguaje de programación usado por Angular [14].
- **IndexedDB**. Sistema de almacenamiento persistente en el navegador [15].
- **Dexie**. Librería que trabaja sobre *IndexedDb* y facilita las diferentes operaciones de almacenamiento [16].
- **Push API**. API que permite mostrar notificaciones nativas desde aplicaciones web aun cuando el navegador está cerrado.[17].
- **Firebase FCM**. API multiplataforma que facilita el envío y recepción de notificaciones push, y que siguen el estándar Push API [18]. Ha sido utilizado tanto en cliente como en servidor.

Capítulo 3

Metodología y planificación

El objetivo de este capítulo es presentar las características principales de la metodología llevada a cabo para producir este proyecto y la adaptación concreta que se ha realizado, indicando las herramientas que han facilitado la aplicación de la metodología. Se comentará también cómo ha sido planificado el proyecto, y se hará una descripción detallada de los costes que supuso.

3.1 Metodología de desarrollo

Para desarrollar este proyecto, se ha considerado que el tipo de metodología que mejor se adaptaba sería una metodología incremental, caracterizada por dividir el ciclo de vida del proyecto en varias iteraciones que se suceden en el tiempo. Cada iteración es en sí un pequeño proyecto que cuenta con las fases típicas de un ciclo de desarrollo tradicional (análisis, diseño, implementación y pruebas) [19].

Este tipo de metodologías ofrecen una rápida adaptación a los cambios y el refinamiento continuo de los objetivos, derivado de dividir el trabajo en iteraciones [20]. Buscan producir continuos incrementos de producto funcionales que ofrecer al cliente para obtener *feedback* de forma rápida, lo que permite detectar errores mucho antes que las metodologías clásicas.

El trabajo a realizar en cada iteración es planificado antes de que ésta comience, pero después de que termine la iteración previa. Por esto, es más sencillo estimar y planificar en este tipo de metodologías que en las clásicas, donde se tiene que planificar por adelantado muchas más tareas, y por lo tanto también se reduce el riesgo del proyecto.

Por todos estos motivos se ha considerado que las metodologías incrementales se adaptaban bien a este proyecto. Para su puesta en práctica, se ha decidido adaptar *Scrum*, un marco de trabajo caracterizado por ser ligero y simple de entender, y que tiene como bases el empirismo y desarrollo iterativo [21]. Debido a que *Scrum* se concibe para equipos con siete miembros por lo menos [20], y en el desarrollo de este proyecto solo participan tres, no ha

sido posible implementar todas las directrices que propone, llevando a cabo una adaptación de la metodología a las características del proyecto.

Para ilustrar esta adaptación, se comentarán los principales elementos que forman *Scrum* y cómo han sido aplicados individualmente, dividiéndolos en tres grupos:

3.1.1 Equipo Scrum

Grupo auto-organizado y que cuenta con todas las habilidades necesarias para llevar a cabo el trabajo sin depender de personas ajenas al proyecto. Está formado por tres roles:

- ***Product Owner***, el responsable de maximizar el valor del producto, para lo cual identifica los requisitos del proyecto en la Pila del Producto y se encarga de priorizarlos para la maximización del beneficio [21].
- ***Scrum Master***: se dedica a facilitar la aplicación de Scrum y su principal objetivo es maximizar la productividad del Equipo Scrum a través de un buen uso de la metodología [21].
- **Equipo de Desarrollo**, formado por las personas encargadas de producir incrementos a partir de los requisitos establecidos por el *Product Owner* [21].

Para la realización de este proyecto, la figura del *Product Owner* fue ejercida por el autor y los directores, ya que las decisiones han sido tomadas de forma conjunta tomando todos responsabilidades en la obtención y priorización de requisitos. No existió la figura del *Scrum Master*, y el Equipo de Desarrollo tuvo un único miembro, el autor del trabajo.

3.1.2 Artefactos Scrum

Son una serie de elementos que forman parte del ciclo de vida de Scrum, representan el trabajo o el valor en diferentes formas y buscan que todos los miembros del equipo compartan la misma percepción de la información clave. Son los siguientes:

- **Pila del Producto.** Lista ordenada de todo aquello que podría ser interesante incluir en el producto, normalmente representado en forma de historias de usuario. Es la única fuente de requisitos, está ordenada por prioridad de unos elementos frente a otros, está en continuo cambio (ya que la prioridad y la concepción de cada requisito puede ir cambiando según se avanza en el proceso, y pueden aparecer nuevos requisitos en cualquier momento) y es utilizada y modificada en las reuniones de planificación y de revisión por el *Product Owner* [21].
- **Pila del Sprint**. Es el conjunto de elementos de la pila del producto que han sido seleccionados para un *sprint* en su reunión de planificación [21].

- **Incremento**: Es el resultado de un *sprint*, y contiene las funcionalidades de los elementos de la Pila del Producto completados durante ese *sprint* más las de todos los anteriores.

A la hora de valorar los artefactos, Scrum añade la definición de «Terminado» para compartir entre todos los miembros del equipo una serie de criterios para catalogar una historia de usuario como finalizada. En nuestro caso, para considerar una historia totalmente «Terminada» ésta tuvo que ser implementada y haber superado todas sus pruebas de unidad e integración.

Todos estos elementos formaron parte del desarrollo de este proyecto, detallándose la Pila del Producto en la Sección 4.1.4, las diferentes pilas del sprint e incrementos, en la Sección 3.2.3, y las pruebas realizadas, en la Sección 6.2.

3.1.3 Eventos

Diferentes reuniones de tiempo limitado que tienen lugar a lo largo de las iteraciones y que permiten la continua adaptación del proceso. Son:

- ***Sprints***. Las iteraciones de Scrum, en las que se ha dividido el ciclo de vida del proyecto. En total, cinco *sprints* de tres semanas cada uno, durante los cuales se han ido produciendo incrementos de producto «Terminado» y cuyo detallado se lleva a cabo en las Secciones 3.2.2 y 3.2.3.

- **Reunión de planificación del Sprint y Revisión del Sprint.** Aunque en Scrum se conciben por separado, durante el desarrollo del proyecto se han llevado a cabo conjuntamente, ya que la una precede a la otra y así se redujo el número de reuniones. La reunión de planificación se realiza siempre antes de cada *sprint* y es donde se planifica el trabajo que se va a llevar a cabo durante el mismo [19]. En estas reuniones no sólo se determinó qué funcionalidades iban a formar parte del *sprint*, sino que además se diseñó cómo iban a ser realizadas dichas funcionalidades y el resultado esperado al término de cada una de ellas. Por otra parte, la Revisión del Sprint se lleva a cabo cuando éste termina, y en ella se repasa el trabajo hecho y se ejecuta una demostración del incremento finalizado, buscando detectar errores y posibles mejoras a implementar el los siguientes *sprints* [19]. Estas dos reuniones han tenido lugar siempre al finalizar un sprint (y por tanto comenzar el siguiente), y su duración combinada fue habitualmente de aproximadamente una hora y media.

- **Scrum Diario**. Es una reunión diaria entre los miembros del Equipo de Desarrollo en la que cada miembro comenta qué ha realizado en el último día, su plan de trabajo para el

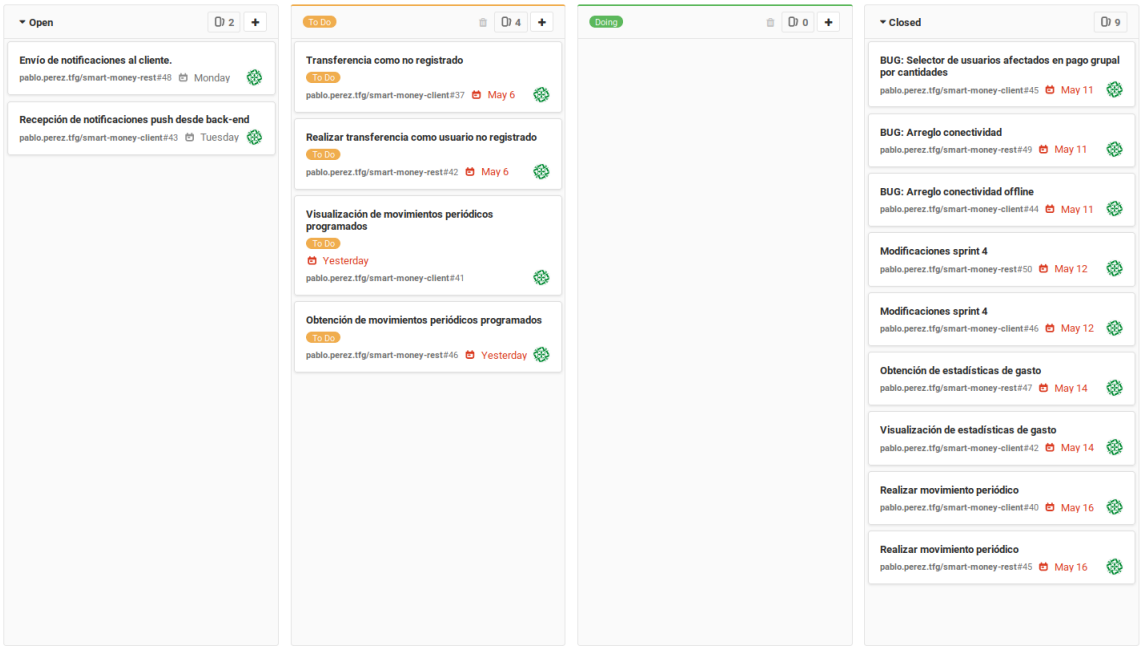

Figura 3.1: Tablero de *issues* en GitLab

siguiente día, y si ha encontrado algún obstáculo. Estas reuniones no se han producido en este proyecto al estar el Equipo de Desarrollo compuesto por un único integrante.

- **Retrospectiva del Sprint**. Reunión realizada al término de la Revisión del Sprint, participando todo el Equipo Scrum, y en la que se valora cómo ha trabajado el propio Equipo y qué mejoras se pueden implementar. Estas reuniones tampoco se han llevado a cabo en este proyecto.

3.1.4 GitLab

A la hora de la puesta en práctica de Scrum, la herramienta que más ha contribuido en todas las facetas ha sido GitLab, una herramienta que, además de sus funcionalidades como repositorio de código gestionado con Git, cuenta con ayudas para el desarrollo ágil, como sistema de *issue-traking*, gestión de tareas dentro de cada *issue*, soporte para hitos, tratamiento de la documentación a través de wikis, etc. [22].

Para el control de este trabajo se crearon dos proyectos, uno para el cliente y otro para el servidor, dentro del mismo grupo para así poder gestionar las *issues* conjuntamente desde una vista general de grupo. De esta forma, los hitos eran compartidos por ambos proyectos, y las propias *issues*, pese a pertenecer a cada proyecto individual, podían ser gestionadas desde la misma vista. Cada *issue* generada en este proyecto ha representado o bien una historia de usuario (en cuyo caso se han creado dos, una para el cliente y otra para el servidor), o bien

una tarea específica que no representaba ninguna historia ni estaba contenida dentro de una, como por ejemplo, una migración de base de datos o el arreglo de un *bug*.

Estas *issues* han sido controladas desde el tablero que proporciona GitLab para ello, una especie de tablero Kanban [23] en el que las historias de usuario se muestran como tarjetas agrupadas en varias columnas que representan el estado en el que se encuentra cada una. El tablero utilizado constó de cuatro columnas, representando *issues* abiertas, pendientes, en curso y cerradas. Las abiertas han sido aquellas creadas pero que aún no habían sido detalladas lo suficiente y/o planificadas; pendientes, aquellas que sí lo han sido pero que no se está trabajando en ellas; al establecer una como en curso se indicó que se estaba trabajando en ella; y cerrándolas, se las ha considerado como «terminadas». El tablero de la Figura 3.1 muestra, de forma visual, el estado de cada *issue*, lo que supone mucha información del estado del *sprint* en curso.

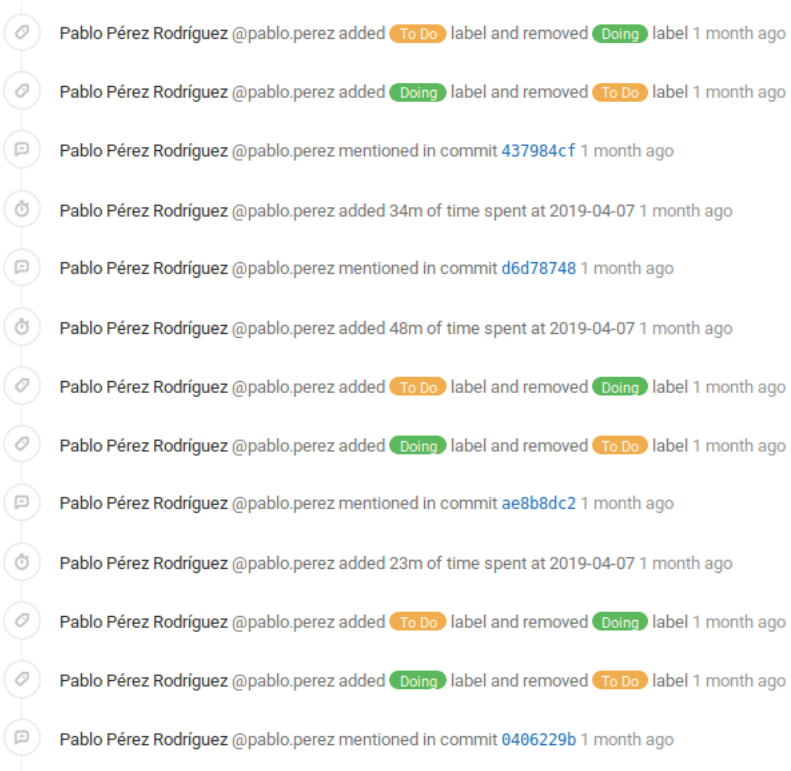

Figura 3.2: Detalle de una *issue* en GitLab

Las *issues* de GitLab también pueden ser referenciadas por *commits* en git, lo que permite establecer una trazabilidad exacta de qué partes de código implementaron determinada historia de usuario. La Figura 3.2 representa toda la información que queda almacenada de una *issue*, entre la que se pueden ver las menciones en *commits*. Otra funcionalidad utilizada ha sido la imputación de tiempos por *issue*, y que se puede observar también en la Figura 3.2. Así

se ha podido llevar un control detallado del tiempo empleado en cada historia, lo que ha permitido comparar las planificaciones con la realidad e ir refinando las estimaciones sucesivas.

3.1.5 GitFlow

El flujo de trabajo llevado a cabo a lo largo del proyecto se ha basado en *GitFlow*, un modelo para trabajar con *Git* que, en lugar de utilizar una única rama (*master*) en la que se subirán todos los cambios, crea varias ramas con cometidos distintos. El trabajo realizado en cada funcionalidad de la aplicación irá en su propia rama, y cuando esté finalizada, se llevará a *develop*, donde se juntará con todas las funcionalidades previas [24]. Cuando se acerque una fecha de entrega se creará una rama *release* partiendo de *develop* que contendrá todas las funcionalidades de la nueva entrega y sobre la que sólo se podrán hacer correcciones e incluir documentación. Una vez que esté lista para ser publicada, se hará un *merge* a *master*, indicando el lanzamiento de una nueva versión [24].

Este modelo de trabajo permite tener una clara separación sobre en qué momento se han implementado qué funcionalidades, permitiendo detectar fácilmente qué *commits* introdujeron errores. Además, el ciclo de trabajo que propone a través de ramas *release* combina muy bien con metodologías iterativas, facilitando las tareas de lanzamiento de incrementos. Pese a estar pensado para grupos de trabajo de varias personas, se han podido aprovechar algunas de las ventajas que ofrece, y ha supuesto un enfoque más profesional del trabajo.

3.1.6 Herramientas de soporte a la metodología

Para facilitar las diferentes tareas relacionadas con el proyecto, se han utilizado, además de los ya mencionados *Git* y *GitLab*, las siguientes herramientas:

- *IDEs*, usados para el desarrollo. Se ha utilizado ***Eclipse*** para el desarrollo Java, y ***Visual Studio Code*** para el desarrollo Angular.
- Navegadores, que permitieron la ejecución del proyecto. **Google Chrome y Firefox**, en sus versiones de escritorio y móvil; además, la versión de escritorio de Google Chrome incluye unas herramientas de desarrollador que han permitido *debugging*, inspección de HTML y CSS y simulación de un dispositivo móvil.
- **Maven**. Herramienta software para la gestión y construcción de proyectos Java utilizada para el *back-end* de la aplicación [25].
- **JUnit**. *Framework* Java para la automatización de pruebas [26].
- **JaCoCo**. *Framework* Java que permite analizar la cobertura del código implementado [27].

- **Npm**. Gestor de paquetes para JavaScript que facilitó la inclusión de librerías en el cliente [28].
- Programas para la generación de *mockups*. Se ha elegido ***Pencil*** que además es *software* libre [29].
- Terminal y shell, que han permitido lanzar los servidores, trabajar con Git y desplegar las versiones. La *shell* escogida ha sido ***zsh***, que contiene añadidos de autocompletado y facilita el trabajo con Git.
- Programas para diseño de diagramas, utilizando ***Draw.io***, una alternativa web sin coste [30], y ***MagicDraw***, con la licencia de estudiant [31].

3.2 Planificación y seguimiento

En esta sección se comentan las estimaciones y planificaciones realizadas inicialmente, comparándolas con el resultado final, y se enumeran los recursos utilizados para desempeñar dichas tareas.

3.2.1 Recursos

Los recursos empleados para llevar a cabo este proyecto se dividen en tres grandes grupos, recursos humanos, recursos materiales y recursos *software*.

Recursos humanos

Para la realización de este proyecto se ha contado con un equipo de tres personas que, como se menciona en la sección previa, ejercen los roles de *Product Owner* y Equipo de Desarrollo. A la hora de trabajar, han ejercido tres funciones: la de *Product Owner*, llevada a cabo por los tres integrantes; la de analista, realizando labores de planificación de trabajo y diseño de cómo se implementarían funcionalidades; y la de desarrollador, implementando el trabajo planificado.

El Cuadro 3.1 indica los salarios por hora trabajada de cada miembro del equipo en función del rol desempeñado.

Recursos materiales

Se contó tanto con recursos propios como en la nube:

- **Ordenador portátil personal** *Acer Aspire*. En él se llevó a cabo el desarrollo de todo el proyecto y sirvió también para llevar a cabo las demostraciones de incrementos en las reuniones de revisión de cada *sprint*.

- **Móvil personal** *Huawei P9*. Sirvió para probar de una forma más real el proyecto desarrollado en su versión móvil.

- **Servidor Heroku** [32], en el que se alojó el *back-end* del proyecto. Cuenta con planes gratis y de pago, en el momento de escribir esta memoria se sigue esperando a que nos concedan el plan gratuito para estudiantes. Mientras tanto, se está usando el plan sin coste, que tiene la desventaja de que tras un periodo de inactividad de 30 minutos, el servidor se apaga, teniendo que llevar a cabo todo el proceso de levantamiento al llegarle una nueva petición.

- **Firebase Hosting** [33], en el que se alojó el *front-end* del proyecto. También se optó por el plan gratuito ofrecido, que no es tan reducido como el de *Heroku*.

Recursos software

Todos los mencionados en la Sección 3.1.6, que al ser soluciones abiertas o contar con licencias de estudiante, no han supuesto ningún coste.

3.2.2 Planificación inicial

Previo al comienzo del desarrollo, se realizó un trabajo de análisis preliminar para sentar las bases del proyecto en el que se llevaron a cabo las siguientes tareas:

- **Estudio de la tecnología**, para conocer las capacidades y los límites de las alternativas disponibles. Entre otras cosas, se compararon las características de los principales *frameworks* de JavaScript (Angular, React y Vue), se trataron de determinar los límites existentes en el marco web para la realización de una interfaz que se sintiese casi nativa en móviles (arrastrar elementos por la pantalla, velocidad, fluidez, *Progressive Web App*...) y se analizaron las posibilidades para el futuro despliegue del proyecto. Para complementar este análisis, una vez que se tomó la decisión de utilizar Angular por sus facilidades para el desarrollo de aplicaciones web centradas en dispositivos móviles y madurez de la tecnología, se pasó al estudio en profundidad del *framework* para facilitar las estimaciones posteriores.

- Realización de una **Pila de Producto inicial**, a través de reuniones del Equipo Scrum, con una serie de historias de usuario iniciales que se irían refinando con el avance del proyecto.

- Desarrollo de unos **prototipos**, basándose en la Pila de Producto inicial que permitieron comprender mejor el ámbito en el que se iba a trabajar.

- Diseño de un **modelo de datos** para el almacenamiento de la información requerida por la aplicación. Se complementó con el escenario incluido en el Apéndice B con el que se comprobó la validez del modelo para llevar a cabo todos los cálculos necesarios, y que sirvió para posteriormente probar que la aplicación había sido implementada correctamente.

- Diseño de la **interfaz del servicio REST** que se ofrecería.

En el Capítulo 4 se detallan en profundidad estas labores y el resultado obtenido.

Para terminar este estudio se llevó a cabo una planificación inicial que resultó en el diagrama de Gantt de la Figura 3.3. Como se puede ver, se decidió utilizar *sprints* de tres semanas, buscando un compromiso entre cantidad de trabajo y *feedback* constante; por una parte, iteraciones más largas resultarían en incrementos mayores, pero con el riesgo de no evaluar a tiempo el trabajo realizado, y por otra parte, iteraciones más cortas resultarían en un *feedback* muy rápido, pero sin suficiente material que evaluar.

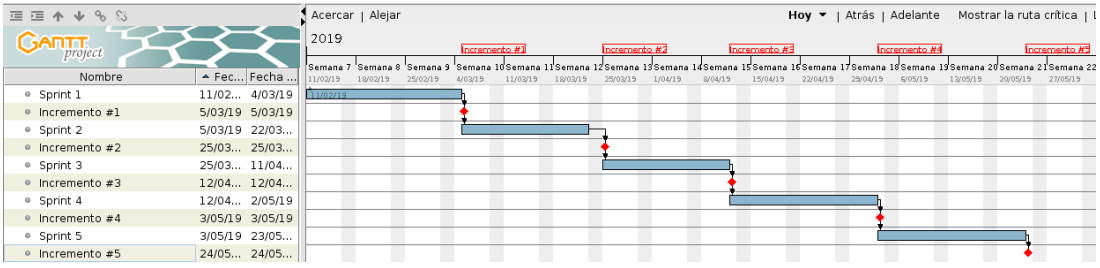

Figura 3.3: Diagrama de Gantt de la planificación proyecto

Así, el desarrollo se llevaría a cabo en cinco *sprints* de tres semanas cada uno, dejando cerca de un mes al finalizar esta etapa para la redacción de la memoria final. Dentro de cada *sprint*, se planificó que el trabajo se dividiría como sigue:

- 4 horas al día, 5 días a la semana en trabajo de desarrollo realizado por el autor.

- 1 hora al día, 3 días a la semana, en trabajo de análisis realizado por el autor.

- 1,5 horas en cada *sprint* por cada miembro del equipo Scrum, representando las funciones de *Product Owner* y analista en las reuniones que se llevarían a cabo.

Esto resulta en **66** horas de trabajo por *sprint* del autor, sumando **330** horas trabajadas, a las que se le deben añadir 20 del estudio inicial, resultando en una estimación de **350** horas de trabajo por parte del alumno.

3.2.3 Seguimiento de la planificación

Ahora se comentará la planificación individual de cada *sprint* con sus tareas correspondientes, y se comparará con el trabajo que finalmente se llevó a cabo en cada uno, ilustrando las desviaciones sufridas sobre la planificación inicial y sus motivos. Cabe aclarar que las planificaciones de cada *sprint* fueron realizadas, de acuerdo con Scrum, en la reunión de planificación previa a cada uno, por lo que a la hora de planificar un *sprint* se conocía el resultado del anterior y se pudo actuar en consecuencia.

Sprint 1

En este *sprint* se creó el esqueleto del proyecto al que se le irían añadiendo funcionalidades, y se empezaron a desarrollar las más básicas que posteriormente permitirían efectuar operaciones más complejas. Las tareas inicialmente planificadas par este *sprint* son las siguientes:

- Generación de un esqueleto para el proyecto Java (con el generador de Spring Boot) y de un esqueleto para el proyecto Angular, a través de su CLI.
- Implementación de las operaciones de registro en la aplicación, inicio y cierre de sesión (historias 1, 2 y 3).
- Desarrollo de los *slots* que gestionan las cuentas individuales, y de las operaciones individuales básicas —ingreso, pago y transferencia, que se corresponden con las historias 4, 5, 6, 7, 8 y 9.
- Desarrollo de la interfaz básica que permita arrastrar *slots* para realizar todas las operaciones y sobre la que se basa la interfaz del proyecto.

La planificación de esta iteración resultó muy acertada, ya que se pudieron llevar a cabo todas las tareas previstas y todas las historias de usuario fueron completadas en las tres semanas del *sprint*. En la reunión de revisión posterior se detecta que las animaciones al arrastrar los elementos por la interfaz no son del todo precisos y dan la impresión de fallar, por lo que se añade al trabajo futuro la mejora de las mismas. También se estudia una mejor implementación de la gestión de sesiones de usuario, dando la posibilidad de recordar la sesión para no introducirla cada vez que se abre una nueva pestaña.

Sprint 2

Se planificó incluir las operaciones básicas para trabajar con grupos, traduciéndose en las siguientes tareas:

- Gestión de grupos: crear un grupo, añadir y eliminar miembros, y editar ajustes del grupo (historias 10, 12, 11 y 20).
- Realización de pagos y transacciones en un grupo, y cálculo de las deudas que se generan, materializados en las historias 13, 14, 15, 16 y 17.

El trabajo a realizar en este *sprint* fue subestimado en su planificación y como consecuencia, dos historias quedaron sin implementar (17 y 20). Esto se debió principalmente a que no se tuvieron en cuenta diversos factores de complejidad asociados a la gestión de grupos y la seguridad que ello conlleva, y por lo tanto su desarrollo llevó más de lo previsto.

Sprint 3

El tercer *sprint* tuvo como objetivo complementar las funcionalidades existentes con otras que aporten valor añadido:

- Envío de transacciones entre miembros de un grupo, planificada para el *sprint* anterior pero no llevada a cabo a tiempo.
- Visualización de los movimientos realizados, tanto propios como ajenos (historia 18) y sus detalles (historia 19).
- Mejora del control sobre un grupo, permitiendo cambiar categorías o monederos por defecto, traducidos en las historias 20, 21 y 22.
- Representación de más información en la vista principal, como el estado de cada *slot* y de cada grupo (historias 23 y 24).

En este *sprint* se llevó a cabo el trabajo planificado para la anterior iteración pero no terminado, además de una serie de modificaciones sobre el trabajo previo a las que se dio prioridad alta (mejor gestión de invitaciones, arreglo de un problema con las transferencias y mejora de las animaciones de arrastre).

Debido a la mala estimación de un trabajo de una asignatura, se tuvieron que reducir las horas de desarrollo en este *sprint*, haciendo algo más de 30 horas en lugar de las 40 planificadas, provocando que no se implementasen las historias de usuario 23 y 24.

Sprint 4

El incremento generado al término de este *sprint* otorgó al usuario de un mayor control sobre los estados de sus cuentas y grupos desde la vista principal, y se dotó a la aplicación de funcionalidad *PWA*, incluyendo las siguientes tareas:

- Realización de la última tarea de la iteración previa que no se completó en su *sprint*.
- Inclusión de capacidades de *Progressive Web App*, incluyendo funcionalidad *offline* (historia 25) y la capacidad de ser instalada en cualquier dispositivo.
- Obtención de detalles, edición y eliminación de un *slot* (historias 26 y 27).
- Modificación y eliminación de un movimiento, representados por las historias 29 y 28.

Para compensar la falta de trabajo del *sprint* anterior, en éste se dedicaron unas 10 horas adicionales al desarrollo, permitiendo implementar las dos historias pendientes del *sprint* 3 y todo el trabajo planificado para este *sprint*, que supuso 3 horas extraordinarias de trabajo como analista por la complejidad de la funcionalidad sin conexión.

Además, durante el trabajo de análisis de esta iteración, al recopilar información de *PWAs*, se observa que éste tipo de aplicaciones web permiten el envío de notificaciones nativas, por lo que, además de las tareas descritas, se incluye una adicional de estudio de viabilidad de este tipo de notificaciones, para considerar su inclusión en el siguiente incremento.

Sprint 5

Esta iteración añade funcionalidades que complementan la aplicación con gran valor añadido, como estadísticas, y se determina que se implementarán notificaciones debido al resultado favorable del estudio realizado. Además, en la reunión de planificación de este *sprint* se detectan una serie de *bugs* al operar sin conexión a los que se les da prioridad máxima, por lo que se planifican las siguientes tareas:

- Arreglo de *bugs* al trabajar sin conexión.
- Obtención y representación de estadísticas relevantes de gasto detalladas en la historia 30.
- Inclusión de movimientos periódicos en el tiempo (historias 31 y 32).
- Envío de notificaciones al usuario cuando haya movimientos en un grupo o tenga lugar un movimiento periódico (historia 33).

Para completar este *sprint* se tuvieron que emplear unas 8 horas extras de trabajo de desarrollo, y aproximadamente 2 de trabajo de análisis, con respecto a la planificación inicial de horas trabajadas por iteración, para poder terminar todas las tareas.

Como conclusión se puede extraer que pese a que la planificación inicial se vio afectada en un par de ocasiones, mayormente se compensaron unas desviaciones con otras (por ejemplo,

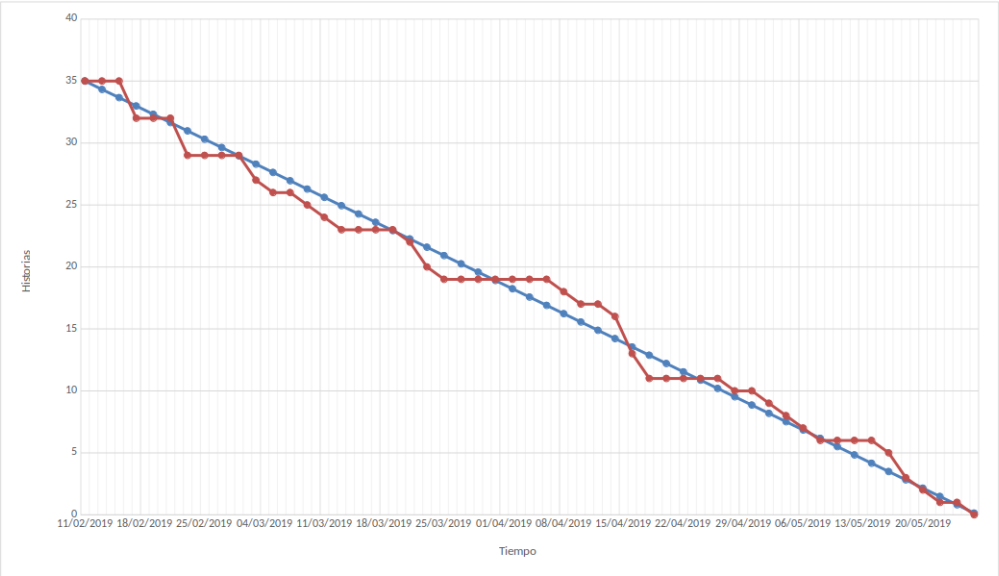

Figura 3.4: *Burdonw chart* de la progresión del proyecto

la falta de trabajo del tercer *sprint* fue recuperada con más trabajo en el cuarto). Estas desviaciones en el tiempo se pueden ver de forma gráfica en la Figura 3.4, en la que se muestra un un *Burndown chart* en el que se compara las historias de usuario que sería ideal que quedasen por hacer en cada momento (en azul) y las historias de usuario que de verdad quedan por llevar a cabo (rojo).

En ella se representa el avance en el trabajo a lo largo del proyecto, y se puede observar cómo se empezó a buen ritmo, pero a la altura del tercer *sprint* hubo varios días en los que no se terminó ninguna historia. También se ve cómo inmediatamente después se avanza de forma muy rápida (*sprint* 4) y cómo a pocos días de la entrega final había un retraso que fue arreglado con horas extras. En total, se realizaron un total de 13 horas adicionales a las inicialmente planificadas para poder completar las funcionalidades seleccionadas a tiempo.

3.2.4 Evaluación de costes

Para contabilizar el coste total del proyecto no se van a tener en cuenta los costes materiales, ya que como se mencionó en la Sección 3.2.1, no hay costes de licencias ni de equipamiento adicional al ya existente.

En cuanto a los costes de recursos humanos, la planificación inicial de tiempo realizada al comienzo de esta sección resultó en la planificación de costes del Cuadro 3.2. Por otro lado, el Cuadro 3.3 muestra el coste real del proyecto al contabilizar las horas extras que se tuvieron que realizar. Los costes se han calculado en base al salario por hora de cada miembro del equipo, en función del rol que desempeñó —Cuadro 3.1— y teniendo en cuenta que una hora

Miembro	Jefe de proyecto	Analista	Desarrollador
Miguel Luaces	30 €	26 €	-
Alejandro Cortiñas	30 €	26 €	-
Pablo Pérez	20 €	16 €	12 €

Cuadro 3.1: Salario/hora de cada miembro por rol

Miembro	Horas Jefe de Proyecto	Horas analista	Horas desarrollador	Coste
Miguel Luaces	5	2,5	0	215 €
Alejandro Cortiñas	2,5	5	0	205 €
Pablo Pérez	2,5	50	300	4 450 €
Totales	10	57,5	300	**4 870 €**

Cuadro 3.2: Costes planificados del proyecto

extraordinaria tiene un precio de 1,5 veces la hora normal.

En el total, no se ve una gran desviación, ya que normalmente se compensaron las tareas sobreestimadas con las subestimadas, y las horas no realizadas en una iteración con horas extras en otra; aún así, podemos observar que en el último incremento se tuvo que trabajar más allá de lo planificado para entregar el producto, y esto se debió a que al haber más trabajo realizado aparecieron más *bugs* y a que se incluyó una nueva funcionalidad no contemplada al inicio del proyecto.

Miembro	Horas Jefe de Proyecto	Horas analista	Horas desarrollador	Coste
Miguel Luaces	5	2,5	0	215 €
Alejandro Cortiñas	2,5	5	0	205 €
Pablo Pérez	2,5	55	308	4 714 €
Totales	10	62,5	308	**5 134 €**

Cuadro 3.3: Costes reales del proyecto

Capítulo 4

Análisis

4.1 Requisitos

En esta sección se describirán los requisitos de la aplicación desarrollada, tanto funcionales como no funcionales. Primero, se hará una enumeración con un nivel de abstracción alto de dichos requisitos, para posteriormente definirlos de forma más detallada utilizando historias de usuario.

4.1.1 Requisitos funcionales

Los requisitos establecidos al comienzo del proyecto son los siguientes:

- **Autenticación de usuarios**: Un usuario podrá registrarse en la aplicación y usar las credenciales proporcionadas para iniciar sesión posteriormente.
- **Sistema basado en *slots* y movimientos**: Son los dos componentes básicos; un *slot* representa fuentes de ingreso, monederos, categorías de gasto y usuarios no registrados, y un movimiento es cualquier operación de *slot* a *slot* —ingresos, pagos y transferencias.
- **Seguimiento de cuentas individuales**: La aplicación permitirá al usuario almacenar sus ingresos y gastos individuales.
- **Seguimiento de gastos grupales**: Se ofrecerá la posibilidad de formar grupos de usuarios en los que se podrán almacenar los gastos compartidos por sus miembros. Se permitirá saldar las cuentas de forma individual y conjuntamente, y podrán participar usuarios sin cuenta en la aplicación.
- **Se mostrará información relevante** de la situación de cuentas, tanto individuales como grupales, incluyendo listados de movimientos, estadísticas, balances de *slots* y deudas de cada grupo.

- **Movimientos periódicos**: Se dotará al usuario de la posibilidad de efectuar cualquier movimiento de forma repetida en el tiempo (diariamente, mensualmente, etc.).

Durante el ciclo de desarrollo se vio que la incorporación de notificaciones sería viable y un añadido de valor para el usuario, por lo que se incluyó el siguiente requisito funcional:

- Aviso mediante **notificaciones** de que un evento importante ha tenido lugar: una operación en un grupo, o un movimiento periódico efectuado.

4.1.2 Requisitos no funcionales

En cuanto a los requisitos no funcionales, se establecieron los siguientes, que permanecieron invariantes a lo largo del proyecto:

- **Facilidad de uso**: Se deben poder almacenar las operaciones de un modo rápido e intuitivo, a través de gestos de arrastre entre *slots*, como se describe gráficamente en la Figura 2.1.
- **Intefaz atractiva**: Se intentará que la interfaz proporcionada por la aplicación sea lo más amigable e intuitiva posible, lo que además favorecerá la facilidad de uso de la aplicación.
- **Aplicación multiplataforma**: Se podrá operar desde cualquier dispositivo a través de una interfaz pensada para móviles pero que se adapte correctamente a los dispositivos de escritorio. Los datos serán almacenados en la nube para disponer de ellos en cualquier lugar.
- **Funcionalidad *offline***, permitiendo al usuario almacenar cualquier operación en ausencia de conexión. Esto incluye, por una parte, la carga de las páginas con la información más reciente que se tenga, y por otra parte, el almacenamiento temporal de las operaciones para poder ser efectuadas al volver la conexión.
- **Seguridad**: La aplicación debe contener algún sistema de autenticación de usuarios que asegure que ningún otro usuario puede acceder a sus datos.

4.1.3 Actores

Se han distinguido únicamente dos actores:

- **Usuario no autenticado**, al que sólo se le da la opción de acceder a las pantallas de registro e inicio de sesión.
- **Usuario autenticado**, que solo tendrá acceso a sus cuentas individuales y a los grupos de los que sea miembro.

4.1.4 Pila del Producto

A la hora de trabajar, se tradujeron todos estos requisitos en varias historias de usuario, que representan una especificación más detallada. Como resultado de la aplicación de Scrum, el detalle de cada historia de usuario fue refinándose en las sucesivas reuniones, hasta que la historia fuese seleccionada para un *sprint*, momento en el que alcanza su máximo nivel de detalle. Cabe aclarar que las historias aquí descritas representan la Pila del Producto al término del proyecto y no como fue concebida al principio, por lo tanto el nivel de detalle de cada historia es el suficiente para que pueda ser implementada.

1. Como usuario no autenticado quiero crear una nueva cuenta en la aplicación. Debe estar asociada a un nombre de usuario y correo electrónico únicos, y con una contraseña de al menos ocho caracteres.

2. Como usuario no autenticado quiero poder iniciar sesión con mi nombre de usuario y contraseña y, si son correctos, acceder a la vista principal de la aplicación. En caso contrario, quiero ver un mensaje de error. Se dará también la opción de recordar las credenciales, de forma que la próxima vez que acceda no tenga que introducirlas.

3. Como usuario autenticado quiero poder cerrar sesión, teniendo que introducir mis credenciales nuevamente si quiero volver a acceder a la aplicación.

4. Como usuario autenticado quiero crear una fuente de ingreso con un nombre y una descripción.

5. Como usuario autenticado quiero crear un monedero con un nombre, una descripción, un gasto máximo planeado mensual, y un balance opcional que por defecto estará a 0.

6. Como usuario autenticado quiero crear una categoría de gasto con un nombre, una descripción y un máximo de gasto planeado mensual.

7. Como usuario autenticado quiero realizar un ingreso, de una fuente de ingreso a un monedero, indicando una cantidad, un concepto, una descripción opcional y una fecha opcional que por defecto será la actual.

8. Como usuario autenticado quiero realizar una transferencia entre monederos indicando una cantidad, un concepto, una descripción opcional y una fecha opcional, que por defecto será la actual.

9. Como usuario autenticado quiero realizar un pago de un monedero a una categoría de gasto indicando una cantidad, un concepto, una descripción opcional y una fecha opcional, que por defecto será la actual.

10. Como usuario autenticado quiero crear un grupo, indicando un nombre y un monedero por defecto para transferencias ajenas.

11. Como usuario autenticado quiero invitar a otros usuarios a unirse a un grupo al que pertenezco a través de un enlace de invitación, que tendrá una fecha de expiración.

12. Como usuario autenticado quiero crear un usuario no registrado dentro de un grupo al que pertenezco, dándole un nombre y una descripción.

13. Como usuario autenticado quiero realizar un pago dentro de un grupo al que pertenezco, de uno de mis monederos a una de mis categorías, indicando una cantidad, un concepto, una descripción opcional, una fecha opcional que por defecto será la actual y una periodicidad, también opcional. Además, quiero poder seleccionar a qué usuarios le afecta el pago (por defecto a todos), y qué cantidad le afecta a cada uno.

14. Como usuario autenticado quiero realizar un pago dentro de un grupo al que pertenezco, en nombre de un usuario no registrado, indicando una cantidad, un concepto, una descripción opcional, una fecha opcional que por defecto será la actual y una periodicidad, también opcional. Además, quiero poder seleccionar a qué usuarios le afecta el pago (por defecto a todos), y qué cantidad le afecta a cada uno.

15. Como usuario autenticado quiero visualizar las deudas pendientes, esto es, quién le debe dinero a quién, y cuánto.

16. Como usuario autenticado quiero indicar que todas las deudas pendientes han sido saldadas, quedando todos los miembros a cero.

17. Como usuario autenticado quiero enviar una transacción a uno de los miembros del grupo, ya sea un usuario registrado en la aplicación o uno que no está registrado, desde uno de mis monederos, indicando una cantidad, un concepto, un destinatario, una descripción opcional, una fecha opcional que por defecto será la actual y una periodicidad, también opcional.

18. Como usuario autenticado quiero ver una lista con los últimos movimientos, separando los individuales de los grupales. En mis movimientos individuales aparecerán todas las operaciones de mis cuentas y los movimientos grupales que me han afectado; se mostrará, para cada uno, el tipo de operación, la cantidad, el concepto y la fecha. En los movimientos de cada grupo aparecerán aquellos realizados en ese grupo, mostrando el tipo de operación, la cantidad, el usuario que lo realizó y la fecha.

19. Como usuario autenticado quiero ver el detalle de un movimiento que aparezca en una lista de movimientos. De mis movimientos individuales quiero ver, además de los datos

ya listados, su descripción y el *slot* de origen y de destino de la operación. En los movimientos de un grupo, quiero ver el origen y el destino del movimiento, que pueden ser *slots* o usuarios, la descripción, los usuarios que participaron y la cantidad que afectó a cada usuario.

20. Como usuario autenticado quiero editar los detalles de un grupo al que pertenezco, pudiendo modificar el nombre del grupo y los miembros que pertenecen a dicho grupo, siempre y cuando no tengan deudas pendientes, en cuyo caso se mostrará un error.

21. Como usuario autenticado quiero modificar la categoría por defecto para un grupo, a la que irán los pagos realizados por otros miembros del grupo y en los que haya participado, de entre una lista con todas mis categorías.

22. Como usuario autenticado quiero modificar el monedero por defecto para un grupo, al que irán las transferencias que me envíen miembros del grupo, de entre una lista con todos mis monederos.

23. Como usuario autenticado quiero ver el balance de cada uno de mis *slots* de forma rápida al lado de cada uno.

24. Como usuario autenticado quiero obtener información del estado de cuentas de cada uno de mis grupos de forma rápida al acceder a la vista de grupo.

25. Como usuario autenticado quiero poder operar cuando no tenga conexión. Esto incluye acceder a la aplicación sin conexión, algo sólo disponible si se solicitó guardar la sesión y ésta no se cerró previamente; cargar las diferentes secciones de la aplicación, con información lo más reciente posible; y almacenar cambios y que éstos se efectúen cuando se reanude la conexión.

26. Como usuario autenticado quiero visualizar información de cada uno de mis *slots*, tanto individuales como grupales. Para cada uno se mostrará su nombre, su descripción y una lista con los últimos movimientos procedentes de ese *slot*. Además, para cada tipo de *slot* particular quiero ver:

    Fuente de ingreso: cantidad ingresada en el último mes, y la cantidad ingresada en total por esa fuente de ingreso.

    Monedero: balance, gasto en el último mes, objetivo de gasto mensual y gasto total realizado desde él. Además, si es el monedero por defecto de algún grupo, se indicará también.

    Categoría de gasto: gasto en la categoría en los últimos 30 días, objetivo de gasto mensual y gasto total. Además, si es la categoría por defecto de algún grupo, se mostrará también.

Usuario no registrado: su gasto en los últimos 30 días, y el gasto total.

27. Como usuario autenticado quiero eliminar un *slot* del que sea dueño, posteriormente a que se me solicite confirmación de que deseo hacerlo.

28. Como usuario autenticado quiero eliminar un movimiento individual o de uno de mis grupos. Primero se me solicitará confirmación, y si confirmo la operación, se eliminará el movimiento. El único caso en el que no se podrá eliminar un movimiento será cuando se trata de un movimiento grupal visto desde la vista de mis movimientos individuales.

29. Como usuario autenticado quiero modificar los datos de un movimiento. Se distinguen tres casos:

    Movimiento individual. Se podrá modificar su origen y su destino, la cantidad, el concepto, la descripción y la fecha.

    Movimiento grupal. Se podrá modificar el origen si el usuario es el autor del movimiento, el destino, la cantidad, concepto, descripción, fecha, los usuarios implicados y cuánto les afectó a cada uno.

    Movimiento grupal - parte individual. Desde la vista de movimientos individuales, se podrá editar la parte que afecta al usuario, pudiendo modificar destino, concepto, descripción y fecha, pero no el importe.

30. Como usuario autenticado quiero poder consultar estadísticas que muestren información de mis últimos gastos. También quiero seleccionar el periodo de información que se muestra. Se diferencian dos casos:

    Estadísticas individuales. Se mostrará un histórico de mis gastos individuales (incluyendo la parte que me haya afectado de los movimientos grupales), un desglose de gasto por categoría, y un desglose de gasto por monedero origen.

    Estadísticas de grupo. Se mostrará un histórico del gasto realizado en el grupo y un desglose de gasto por miembro.

31. Como usuario autenticado quiero indicar, a la hora de realizar un movimiento, si este va a ser periódico o no. En caso de ser periódico, podré indicar cuatro periodicidades: diaria, semanal (pudiendo elegir el día de la semana que se efectuará), mensual (eligiendo el día del mes) y anual (eligiendo día y mes).

32. Como usuario autenticado quiero eliminar un movimiento periódico programado.

33. Como usuario autenticado quiero que se me envíen notificaciones nativas cuando se efectúa algún movimiento que me afecte en uno de mis grupos, o cuando un movimiento periódico se efectúe.

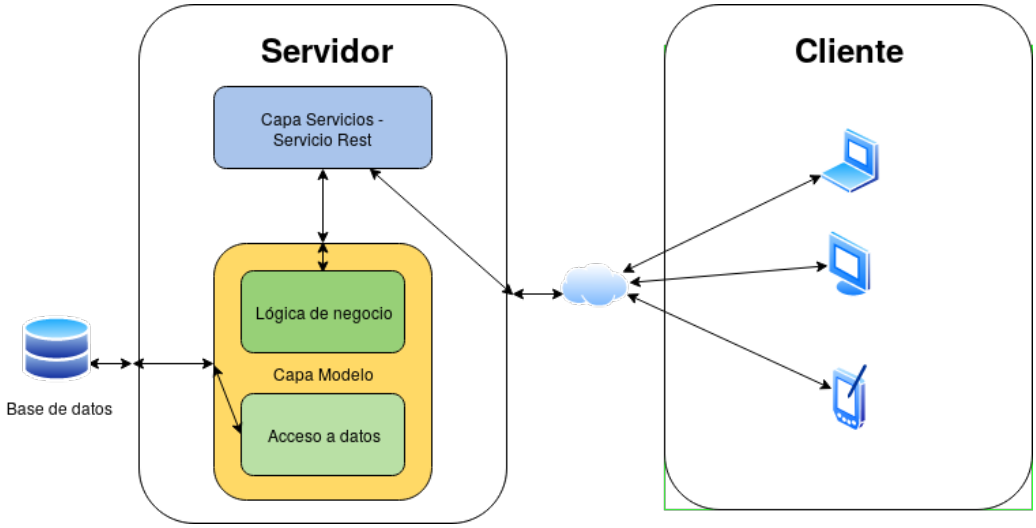

Figura 4.1: Arquitectura general del proyecto

4.2 Arquitectura del sistema

En esta sección se tratará la arquitectura del sistema con un nivel de abstracción bastante alto, centrándose en la descomposición de alto nivel de los componentes del sistema, y no en las tecnologías que les dan soporte. Esta estructura se puede observar en la Figura 4.1. Como se puede ver, la arquitectura está dividida principalmente en dos componentes, cliente y servidor, también conocidos como *front-end* y *back-end*, que se encargan de la interacción con los usuarios y el procesado y almacenamiento de los datos, respectivamente. Esta arquitectura favorece la modularidad y posibilita, entre otras cosas, la existencia de múltiples clientes para un mismo *back-end* (cliente Android, iOS, web...).

4.2.1 Servidor

El servidor alberga la lógica de la aplicación, es el encargado de persistir y recuperar la información de la base de datos, y ofrece una serie de servicios al cliente que le permiten realizar las operaciones requeridas, a través de una API REST.

La estructura del *back-end* está organizada por capas, como se aprecia en la Figura 4.1. Este tipo de estructura, ampliamente utilizada, divide el servidor en varias capas, ejerciendo cada una un rol específico [34]. Las capas inferiores en el esquema proporcionan un servicio a las situadas sobre ellas, ignorando éstas por completo cómo se implementa [35]. Los componentes de una capa sólo tratan con la lógica que corresponde a esa capa, cumpliendo con el principio de diseño de la Separación de Intereses [34]. Además, una capa sólo se comunica con las capas adyacentes, derivando en una arquitectura muy desacoplada que provoca que cambios en una capa no afecten componentes de otras capas.

La división en capas del servidor de este proyecto se puede observar en la Figura 4.1. Se puede ver que consta de dos capas principales, capa modelo y capa servicios. La capa modelo está a su vez subdividida en otras dos, la capa de la lógica de negocio, y la capa de acceso a datos. Se encargan de implementar la lógica de las operaciones especificadas y de comunicarse con la base de datos, respectivamente. Así, la capa de acceso a datos provee a la capa superior de las operaciones necesarias para leer/almacenar datos, sin ésta preocuparse de cómo se llevan a cabo. En conjunto, la capa modelo expone un servicio que permite llevar a cabo todas las operaciones disponibles.

Por otro lado se encuentra la capa de servicios, que expone un conjunto de puntos de acceso (*endpoints*) que pueden ser invocados externamente. A través de ellos, recibe las peticiones procedentes del cliente y delega en el servicio ofrecido por la capa Modelo para llevar a cabo las solicitudes [35]. Esta capa, por lo tanto, expone un servicio al cliente que puede ser invocado remotamente, optando por un servicio REST para este proyecto. REST (*Representational State Transfer*) es un estilo arquitectónico que agrupa una serie de directrices para el diseño de servicios web, basado en HTTP, a través del cual se envían peticiones a una URI utilizando los métodos HTTP disponibles (en este caso, GET, POST, PUT y DELETE) [36]. La interfaz expuesta por este servicio se detalla en la Sección 4.5.

4.2.2 Cliente

El cliente es el encargado de la presentación de una interfaz gráfica al usuario y de responder a sus interacciones, para lo cual envía información al servidor (usando la interfaz REST que éste le proporciona) y actualiza la información presentada según corresponda. Del servidor sólo conoce la interfaz que éste ofrece, produciendo una arquitectura con un acoplamiento muy bajo [34]. El cliente de este proyecto será desarrollado con tecnologías web (explicado más en detalle en la Sección 5.1.3), pero cabe destacar que debido a la estructura elegida, se podrían añadir múltiples clientes para múltiples plataformas (Windows, iOS, Android, etc.) sin modificar la estructura existente.

4.3 Interfaz de usuario

En esta sección se quiere hacer una ilustración de alto nivel de la interfaz de usuario de la aplicación, describiendo la estructura general de pantallas y la navegación entre ellas.

Se llevó a cabo un diseño detallado, a través de prototipos, de las pantallas que formarían parte de la aplicación. El objetivo de este trabajo no ha sido sólo facilitar una futura implementación, sino que además, ha servido para detectar requisitos que no se habían considerado, y para refinar los ya existentes.

A la hora del diseño, se optó por un *mobile-first design*, una alternativa que prioriza el

diseño móvil, relegando los diseños de tableta o escritorio a un paso posterior [37]. De esta forma, la interfaz de la aplicación estará optimizada para estos dispositivos, mejorando el resultado con respecto a diseños centrados en el escritorio y que son después portados a móvil. Esta decisión se tomó teniendo en cuenta, por un lado, la creciente tendencia al acceso a Internet desde dispositivos móviles, siendo ya la mayoría del tráfico *online* procedente de uno de estos dispositivos [38]; y por otro lado, la concepción de la aplicación la orienta a un uso muy móvil. Es por esto que los prototipos que se muestran son sólo para dispositivos móviles.

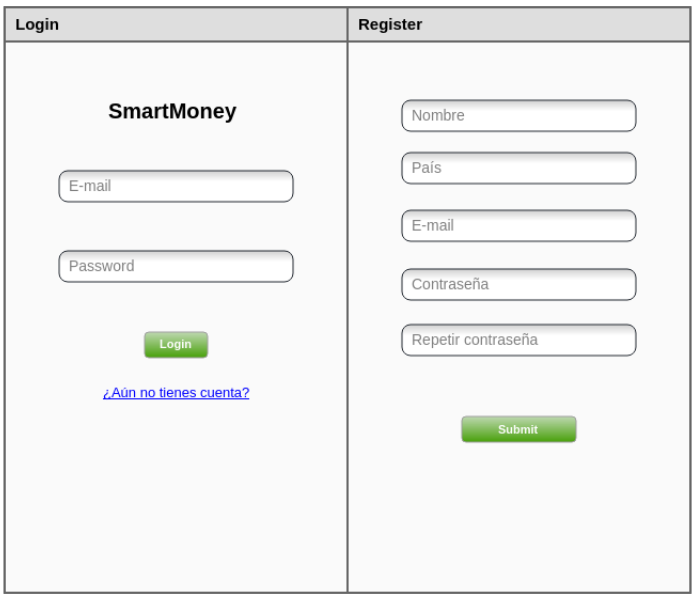

Figura 4.2: Pantallas de inicio de sesión y registro

La vista principal de la aplicación es la que se muestra en las Figuras 4.3 (vista individual) y 4.4 (vista de grupo). En la vista individual sólo se muestran los *slots* del usuario, agrupados según su función: primero las fuentes de ingreso, después los monederos y por último las categorías de gasto. Al final de cada grupo se incluye un botón para crear un *slot* del grupo correspondiente (pantallas 4.5).

Para los grupos, la interfaz es prácticamente la misma, exceptuando que no se muestran fuentes de ingreso (no se van a realizar ingresos en un grupo, es algo individual) y en su lugar se muestra una tabla que, de un vistazo, da información sobre la situación individual de cada miembro en el grupo (si debe dinero, le deben dinero o está a cero) como se muestra en la Figura 4.4. Además, en la sección de monederos se añaden el monedero del grupo y los usuarios no registrados a los ya existentes en la vista individual.

Para navegar entre ambas vistas, se proporcionan unas pestañas para rápidamente cambiar entre la pantalla individual y las de los diferentes grupos. En las Figuras 4.3 y 4.4 también

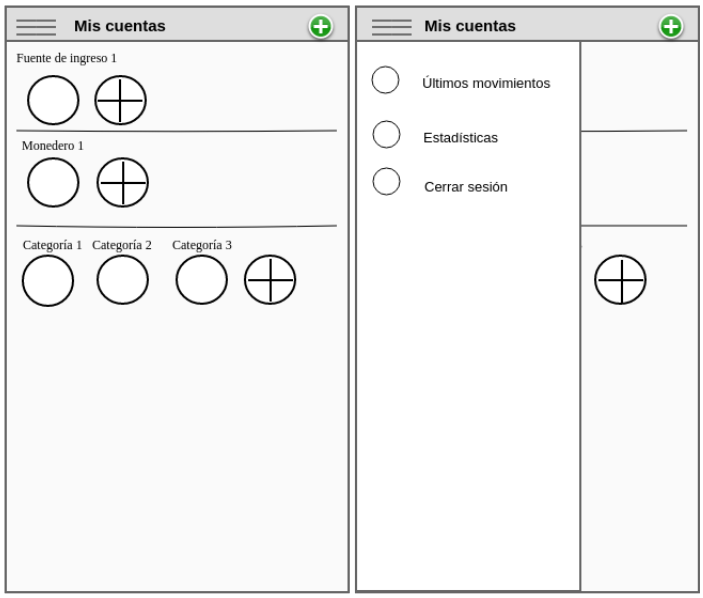

Figura 4.3: Pantalla principal y panel lateral

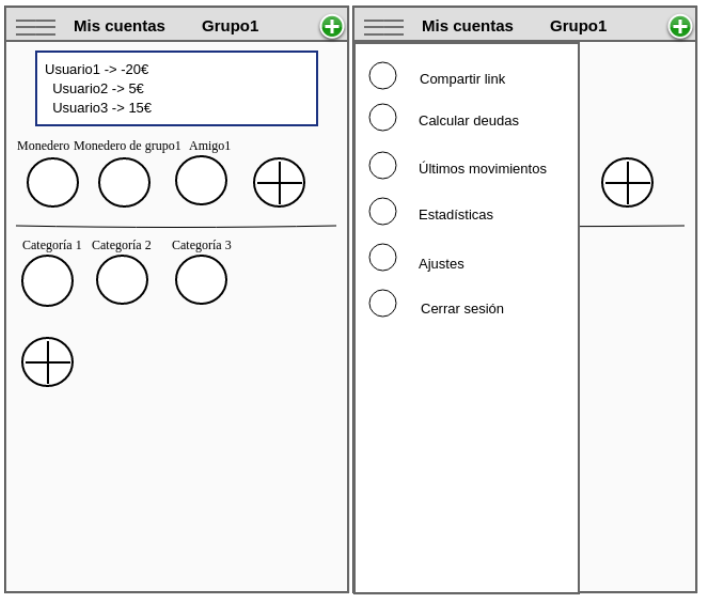

Figura 4.4: Página principal de grupo y panel lateral

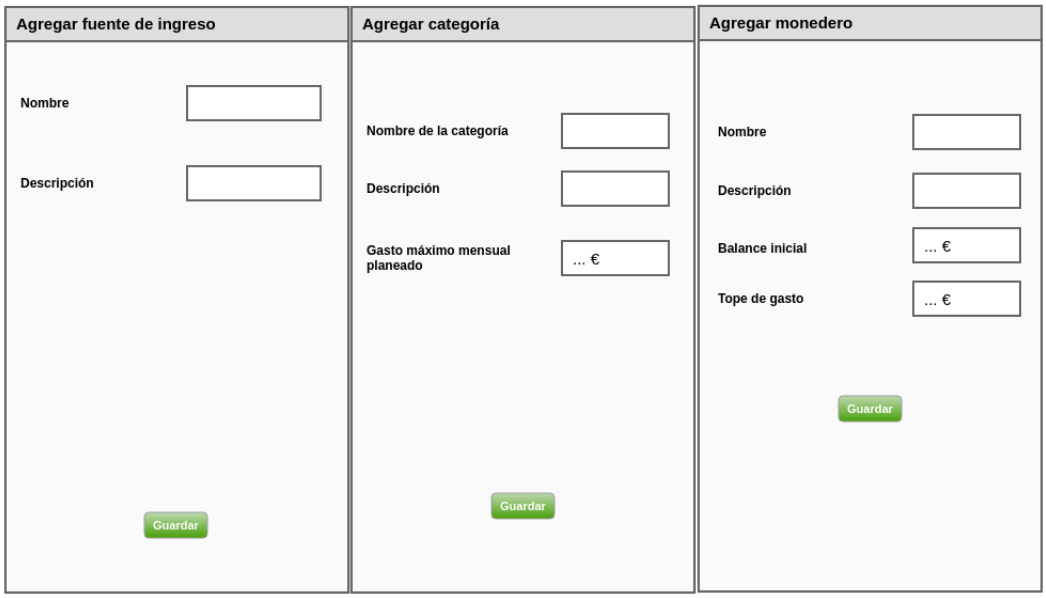

Figura 4.5: Pantallas de creación de *slots* individuales

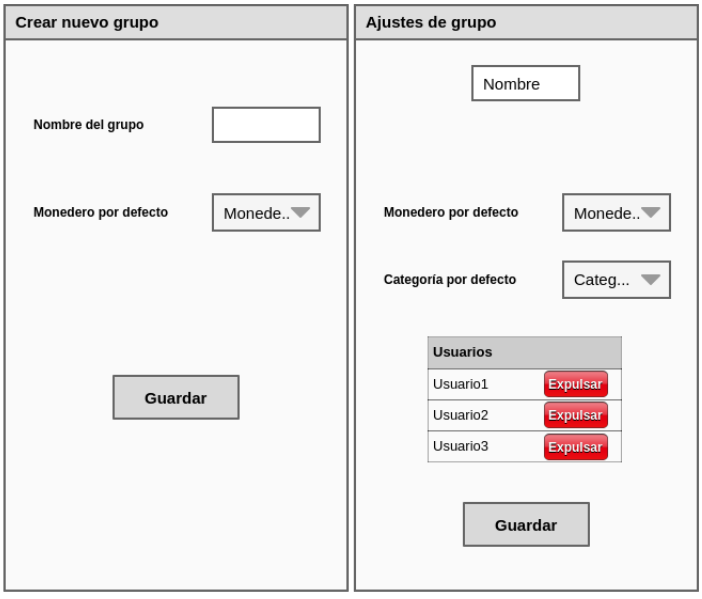

Figura 4.6: Creación y edición de grupo

Figura 4.7: Detalle de una categoría, un ingreso y un monedero, respectivamente

se muestra el menú lateral con las opciones más importantes disponibles. Para crear un grupo se pulsará en el botón de la parte superior derecha visible desde ambas vistas, y se accederá a la primera pantalla de la Figura 4.6, en la que se indicará el nombre del grupo y el monedero por defecto para las transferencias de ese grupo. En la segunda pantalla de la Figura 4.6 se editarán los ajustes de un grupo, permitiendo modificar el nombre, el monedero y la categoría por defecto, o expulsar a algún miembro. Esta pantalla es accesible desde el menú lateral de la vista de grupo (Figura 4.4).

Para acceder al detalle de cada *slot* (pantallas 4.7 basta con pulsar en él desde la vista principal. En estas pantallas se muestran detalles de la actividad en los últimos 30 días y la actividad total del *slot*, el objetivo de gasto si es una categoría, y el balance de los monederos. Además, se incluye para todos un listado de los últimos movimientos que tengan como origen ese *slot*.

Para realizar un movimiento se arrastrará el *slot* origen sobre el de destino, accediendo a las pantallas 4.8 (movimiento individual y grupal, respectivamente) en las que se indicará el importe de la operación, un concepto, una descripción, la fecha (por defecto, la actual) y si es periódico, en cuyo caso se indicará también la periodicidad. Además, para un movimiento grupal, se podrá seleccionar quién ha participado en él, y la cantidad que le ha afectado.

Desde el panel lateral descrito previamente se tiene acceso a las pantallas de listado de movimientos, en las que se listarán en orden cronológico inverso, mostrando el concepto, el tipo de operación (ingreso, pago, transferencia) y el importe del ingreso para los movimientos

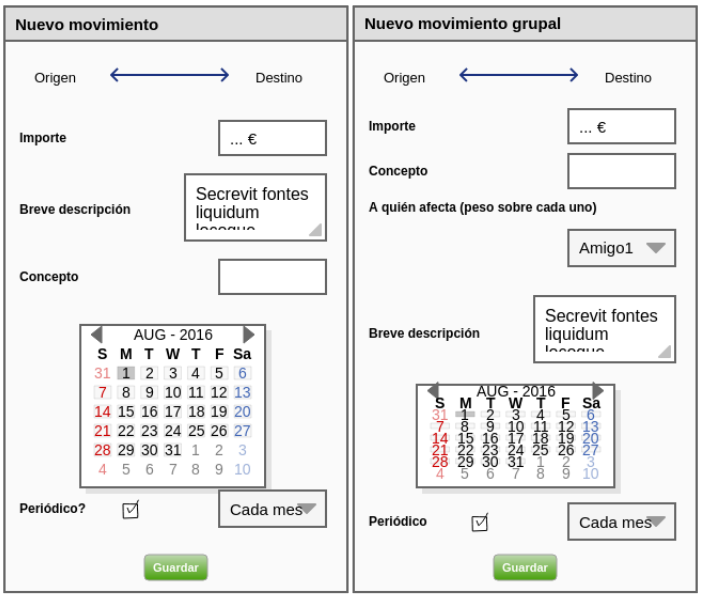

Figura 4.8: Creación de un movimiento individual y uno grupal

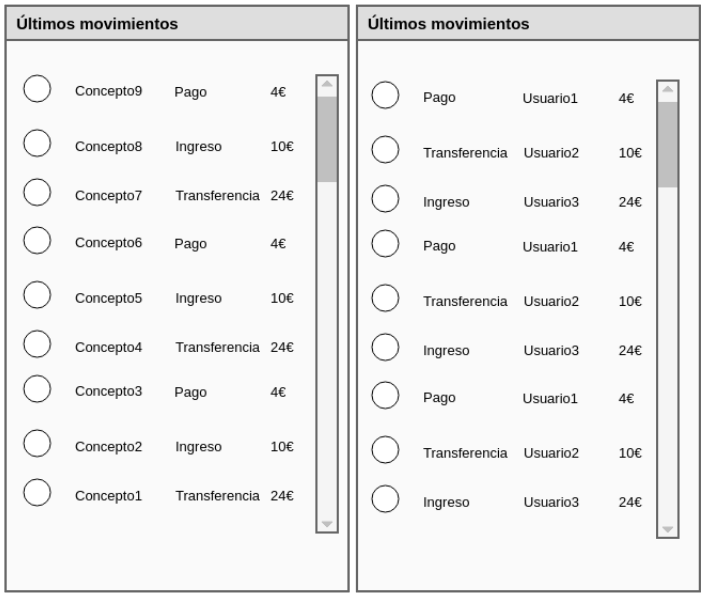

Figura 4.9: Listado de movimientos individuales y grupales

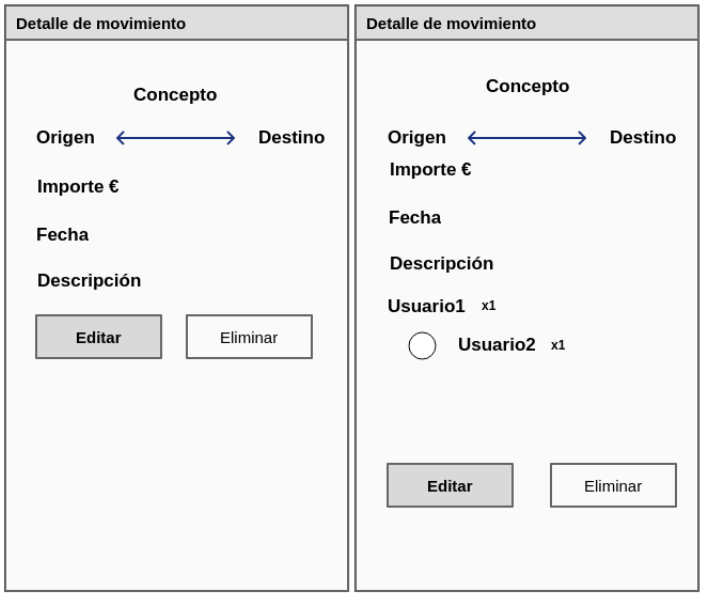

Figura 4.10: Detalle de un movimiento individual y uno grupal

individuales; para los grupales, se verá el tipo de movimiento, el usuario que lo realizó, y el importe (Figura 4.9). Pulsando cada uno de los movimientos se accederá al detalle del mismo, en el que se listará toda la información relevante de un movimiento. Para los movimientos grupales, además, se mostrarán los usuarios que participaron en él y la cantidad que afectó a cada uno. También se podrán editar los detalles, y eliminar el movimiento (Figura 4.10).

Las estadísticas de gasto, tanto grupales como individuales, y la obtención de las deudas pendientes en un grupo (Figura 4.11), serán también accesibles desde los paneles laterales. En el caso de las deudas, sólo estarán disponibles desde el panel lateral en la vista de un grupo.

4.4 Modelo conceptual de datos

En esta etapa también se diseñó el modelo de datos de la base de datos (Figura 4.12) que facilitaría la posterior implementación de la aplicación, y que fue complementado con el escenario propuesto en el Apéndice B, con el que se pretendió comprobar que la información modelada sería suficiente para llevar a cabo todos los cálculos posteriores. Del modelo diseñado cabe destacar lo siguiente:

- Como se puede apreciar, se establece una relación de herencia entre los diferentes tipos de *slots* (Categoría, Fuente_ingreso, Monedero y Usuario_no_registrado) y **Slot**. La superclase alberga información común a todas, y cada subclase, la información específica de ella.

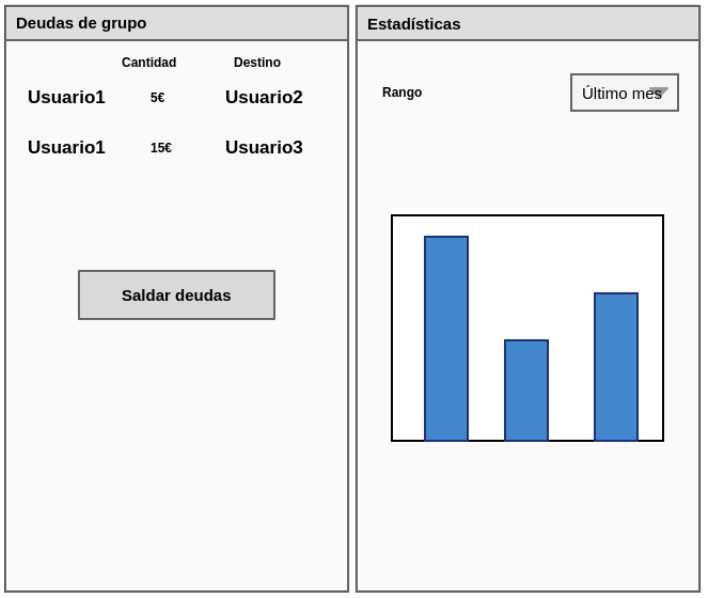

Figura 4.11: Cálculo de deudas y estadísticas

- **Usuario** representa a un usuario registrado en la aplicación, y guarda los datos para que pueda iniciar sesión (nombre de usuario y contraseña), junto con otros de interés, como su correo o su nombre. Para indicar que un Slot pertenece a un Usuario, se establece una relación entre ambas, con cardinalidad *0..1* - *, indicando que un Usuario puede tener múltiples Slots, pero que un Slot sólo pertenece a uno o a ninguno, como en el caso de los usuarios no registrados. Esta relación es heredada por las subclases de Slot.

- En **Movimiento**, se almacenará el importe, concepto, descripción, fecha y el tipo de operación que representa (ingreso, transferencia o pago). Tiene un único Slot como origen y otro como destino, de forma que no hay que especificar varios tipos de movimientos para los distintos tipos de operaciones, que tendrían distintos orígenes y destinos, sino que se establece una relación con la superclase, y será trabajo posterior asegurar que, por ejemplo, no se puede realizar un pago con una fuente de ingreso como origen. También se relaciona Movimiento con Grupo para indicar que el movimiento fue realizado dentro del grupo, pudiendo no relacionarlo con ningún grupo cuando el movimiento es individual.

- La información de **movimientos periódicos** se almacena en una clase independiente. Esta clase tiene toda la información de un movimiento, para poder instanciar un Movimiento cuando sea necesario, y además incluye datos para saber cuándo se tiene que efectuar. La periodicidad representará la frecuencia con la que se llevará a cabo el

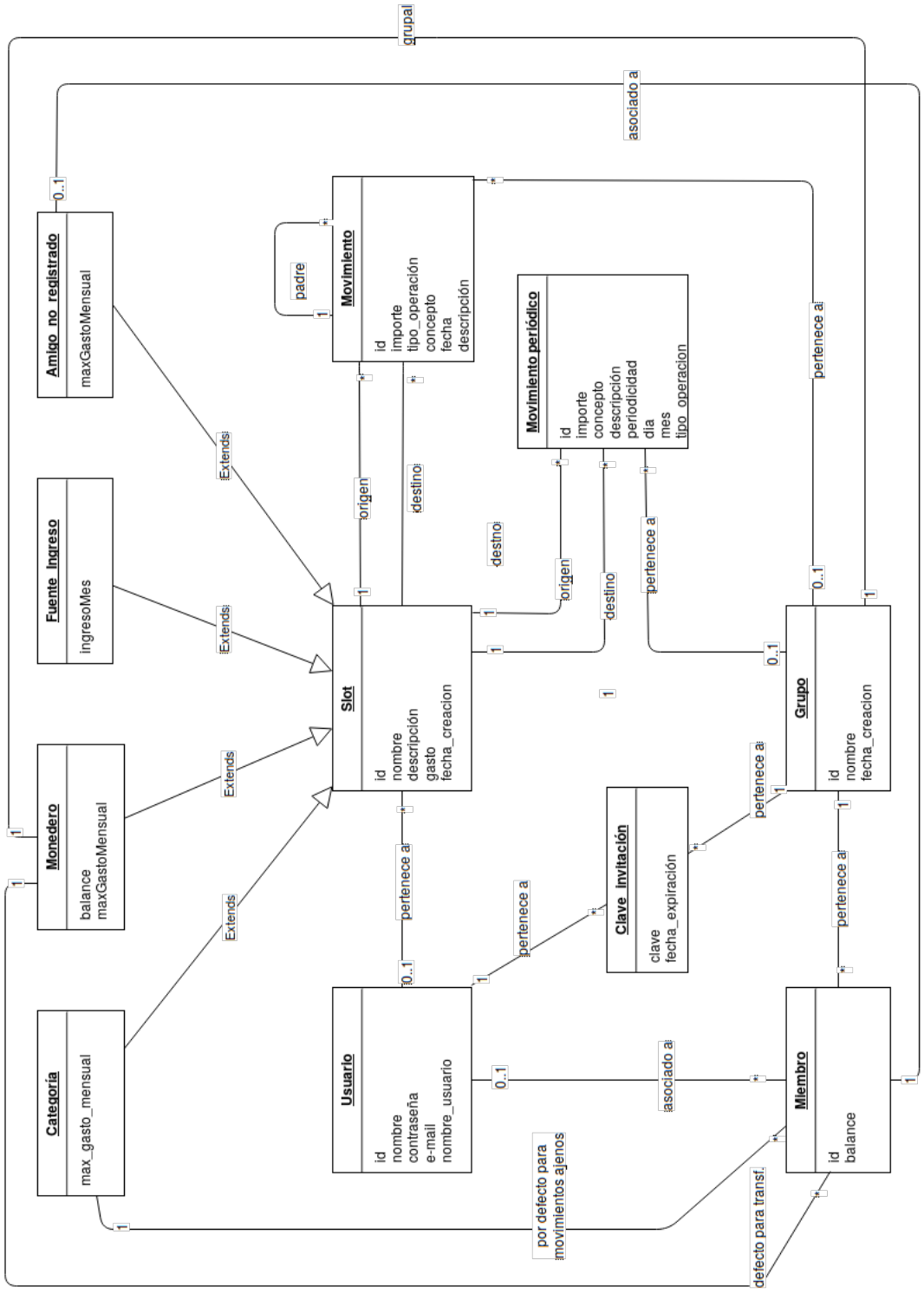

Figura 4.12: Modelo de datos

movimiento, y día y mes complementarán este dato.

- La clase **Grupo** albergará el nombre y la fecha de creación del mismo, y tendrá una relación 1 a 1 con monedero que representará el monedero asociado a ese grupo. Este monedero no tendrá una funcionalidad específica para el usuario final, pero actuará por debajo en los movimientos de grupo, de forma que todo movimiento asociado a un grupo pasará por este monedero. Por ejemplo, al realizar un pago de 40€ en el que participan dos miembros, se efectuará un movimiento del monedero origen al monedero de grupo de 40€, y dos del monedero de grupo a las respectivas categorías de gasto especificadas. Esta separación permitirá imputar de forma más fácil los gastos individuales asociados a uno grupal, y facilitará los cálculos de deudas posteriores —un movimiento hacia el monedero grupal es «positivo», en el sentido de que suma su importe al balance del miembro que lo realizó; un movimiento desde el monedero grupal es «negativo», restando su importe al balance del miembro.

- Para establecer las relaciones entre usuarios y grupos se utiliza la clase **Miembro**, que almacena el balance en cada instante del miembro en el grupo. Relaciona un único grupo con un único usuario, o con ninguno, en cuyo casó estará relacionado con un usuario no registrado. Se ha decidido que la relación esté mapeada por una entidad adicional, en lugar de una sola relación, para poder almacenar en esta entidad un monedero y una categoría por defecto para un usuario en un grupo. La categoría por defecto será aquella a la que vayan los movimientos que realicen otros miembros del grupo y le afecten al usuario; el monedero será el que se use como destino de las transferencias que le hagan al usuario en ese grupo. Así, se permite al usuario controlar individualmente esto, en lugar de una configuración global menos flexible.

- Por último, la entidad **Clave_invitación** representa una invitación, que irá asociada al usuario que la genera y al grupo destino. Su identificador será la propia clave, y se almacenará a mayores la fecha de expiración.

4.5 Interfaz REST

Por último, se diseñó la interfaz REST que expondrá el servidor para realizar las operaciones especificadas. Se separarán por contexto para facilitar su comprensión:

1. **Operaciones de usuario**

   (a) **POST /api/auth/login**: permite iniciar sesión a un usuario con las credenciales del cuerpo de la petición.

   (b) **POST /api/auth/register**: permite registrarse a un usuario

(c) **POST /api/auth/ping**: expuesto para comprobar desde cliente que el servidor es accesible, a la hora de comprobar la conexión

2. **Operaciones individuales**

(a) **POST /api/income**: permite la creación de una fuente de ingreso.

(b) **GET /api/income/username**: devuelve las fuentes de ingreso del usuario «username».

(c) **GET /api/income**: devuelve los detalles de una fuente de ingreso.

(d) **PUT /api/income**: permite modificar los detalles de una fuente de ingreso.

(e) **POST /api/wallet**: permite la creación de un monedero.

(f) **GET /api/wallet/username**: devuelve los monederos del usuario «username».

(g) **GET /api/wallet**: devuelve los detalles de un monedero.

(h) **PUT /api/wallet**: permite modificar los detalles de un monedero.

(i) **POST /api/category**: permite la creación de una categoría.

(j) **GET /api/category/username**: devuelve las categorías del usuario «username».

(k) **GET /api/category**: devuelve los detalles de una categoría.

(l) **PUT /api/category**: permite modificar los detalles de una categoría.

(m) **DELETE /api/slot**: elimina un slot individual.

(n) **POST /api/deposit**: lleva a cabo un ingreso.

(o) **POST /api/payment**: lleva a cabo un pago.

(p) **POST /api/transfer**: lleva a cabo una transferencia.

(q) **GET /api/movement**: devuelve el detalle de un movimiento individual.

(r) **POST /api/movement**: modifica los datos de un movimiento individual.

(s) **DELETE /api/movement**: elimina un movimiento individual.

(t) **GET /api/movements/username**: devuelve los movimientos del usuario «username», paginados y en orden cronológico inverso.

(u) **GET /api/stats/historicExpense**: devuelve el histórico de gasto de un usuario en un rango.

(v) **GET /api/stats/expenseByCategory**: devuelve el gasto agrupado por categoría de un usuario en un rango.

(w) **GET /api/stats/expenseByWallet**: devuelve el gasto agrupado por monedero origen de un usuario en un rango.

(x) **GET /api/periodicMovement/**: devuelve una lista con los movimientos periódicos de un usuario.

(y) **POST /api/periodicMovement/**: crea un movimiento periódico.

(z) **DELETE /api/periodicMovement/**: elimina un movimiento periódico de un usuario.

3. **Operaciones grupales**

(a) **POST /api/group**: crea un grupo.

(b) **GET /api/group**: devuelve los grupos de los que es miembro un usuario.

(c) **POST /api/group/invite**: devuelve una clave de invitación para un grupo.

(d) **POST /api/group/add**: añade un usuario a un grupo.

(e) **POST /api/group/edit**: modifica los detalles de un grupo.

(f) **POST /api/group/wallet/edit**: modifica el monedero por defecto para un grupo.

(g) **POST /api/group/category/edit**: modifica la categoría por defecto para un grupo.

(h) **POST /api/group/unRegistered**: crea un usuario no registrado asociado a un grupo.

(i) **GET /api/group/unRegistered**: obtiene los usuarios no registrados de un grupo.

(j) **GET /api/group/unRegistered/name**: devuelve los detalles de un usuario no registrado.

(k) **PUT /api/group/unRegistered**: modifica un usuario no registrado.

(l) **DELETE /api/group/unRegistered**: elimina un usuario no registrado.

(m) **POST /api/group/payment**: crea un pago en grupo.

(n) **POST /api/group/transfer**: crea una transferencia entre miembros de un grupo.

(o) **GET /api/group/wallet**: devuelve los monederos individuales más el grupal de un usuario en un grupo.

(p) **GET /api/group/members**: obtiene los miembros de un grupo.

(q) **DELETE /api/group/members**: elimina a un miembro de un grupo.

(r) **GET /api/group/debts**: devuelve las deudas pendientes en un grupo.

(s) **POST /api/group/debts**: salda todas las deudas pendientes de un grupo.

(t) **GET /api/group/movements**: devuelve los movimientos de un grupo, paginados, en orden cronológico inverso.

(u) **DELETE /api/group/movement**: elimina un movimiento grupal.

(v) **GET /api/group/stats/expenseByUser**: devuelve las estadísticas de gasto de un grupo agrupadas por miembro a lo largo de un rango.

(w) **GET /api/group/stats/historicExpense**: devuelve el histórico de gasto de un grupo en un rango.

(x) **GET /api/group/periodicMovement**: obtiene los movimientos periódicos asociados a un grupo.

(y) **DELETE /api/group/periodicMovement**: elimina un movimiento periódico grupal.

Capítulo 5

Diseño

5.1 Arquitectura tecnológica del sistema

Esta sección busca complementar el diseño de la arquitectura realizado en el Capítulo 4 (Figura 4.1), a través de un diseño más completo, que detalle las tecnologías elegidas para el desarrollo y cómo éstas se interrelacionan..

5.1.1 Servidor

En el lado servidor, la tecnología escogida para el desarrollo ha sido Java EE versión 8, sirviéndose de gran cantidad de las funcionalidades que ofrece Sring, un *framework* para el desarrollo de aplicaciones empresariales en Java [7]. En cuanto a la arquitectura, se utilizaron diversos componentes para implementar una estructura en capas a partir de la diseñada en el capítulo anterior y cuyo resultado se puede ver en la Figura 5.1.

La capa modelo estará formada por los siguientes componentes, repositorios y servicios:

- Los **repositorios** son clases que actúan de intermediarios entre el almacenamiento de los datos y la aplicación, poniendo en práctica el patrón DAO y por lo tanto, encapsulando el acceso a datos [39]. La ventaja de usar este patrón es que si en algún momento se quisiese realizar un cambio en la base de datos que afectase a cómo los datos son extraídos, las capas superiores no se verían afectadas, facilitando el proceso.

- Los **servicios** implementarán la lógica de negocio haciendo uso de los repositorios para las operaciones de lectura y escritura de datos persistentes y ofrecerán una interfaz a las capas superiores para llevar a cabo las operaciones de la aplicación.

Por debajo, esta capa hará uso de Hibernate, un *framework* que actúa como mapeador objeto-relacional y que permitirá facilitar las labores de programación de la capa de persistencia utilizando objetos persistentes en lugar de trabajar de forma directa con las tablas de base de datos [8].

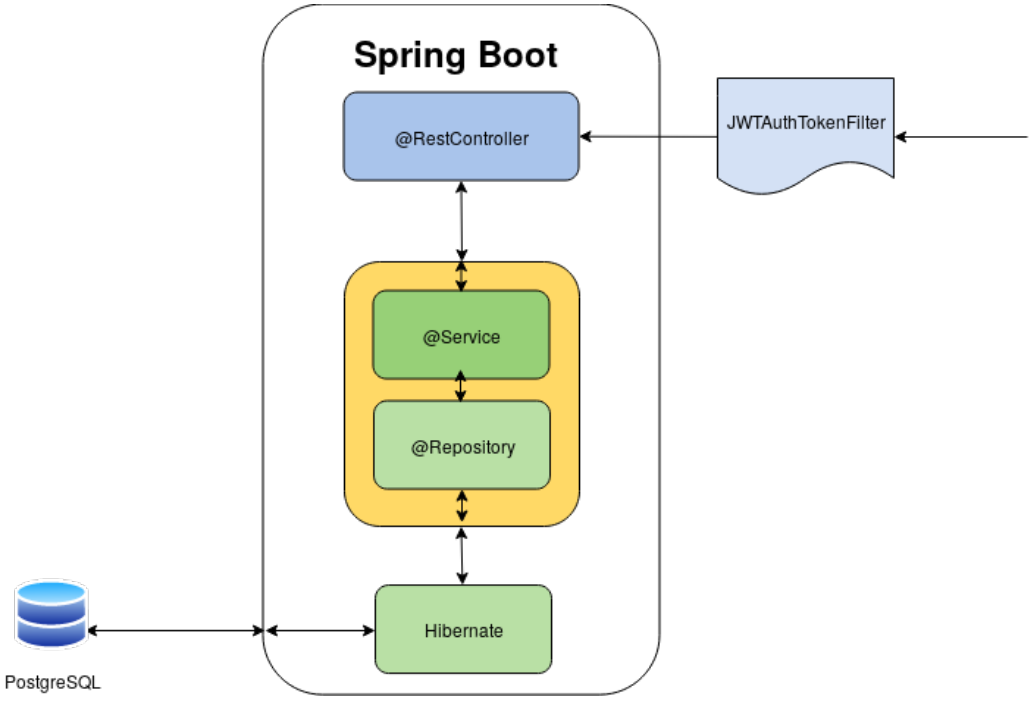

Figura 5.1: Estructura del servidor

Por encima de ella se encuentra la capa servicios, que expone un servicio REST al cliente, a través del cual devolverá, en formato JSON, la información solicitada [40]. También gracias a las anotaciones de Spring, se mapeará cada combinación de URL y método HTTP del API REST a un método, que se encargará de llevar a cabo la operación correspondiente y responder a la solicitud. Todas las peticiones que lleguen al servidor serán interceptadas por la clase *JWTAuthTokenFilter*, cuya funcionalidad detallada se explica en la Sección 5.4.

5.1.2 Base de datos

A la hora de elegir la base de datos, se ha decidido optar por dos aproximaciones:

- **PostgreSQL**: una base de datos relacional *open source* ampliamente utilizada y que se usará para almacenar todos los datos que use la aplicación.
- **H2**: una base de datos relacional que puede actuar en memoria, esto es, almacena los datos en memoria en lugar de en disco (como hace PostgreSQL), permitiendo accesos mucho más rápidos pero con el coste de tener toda una base de datos en memoria [41]. Será usada para ejecutar las pruebas.

De esta forma, a la hora de ejecutar las pruebas no se requiere un servidor de base de datos instalado. Por otra parte, a la hora de desplegar la aplicación sí será necesario, ya que

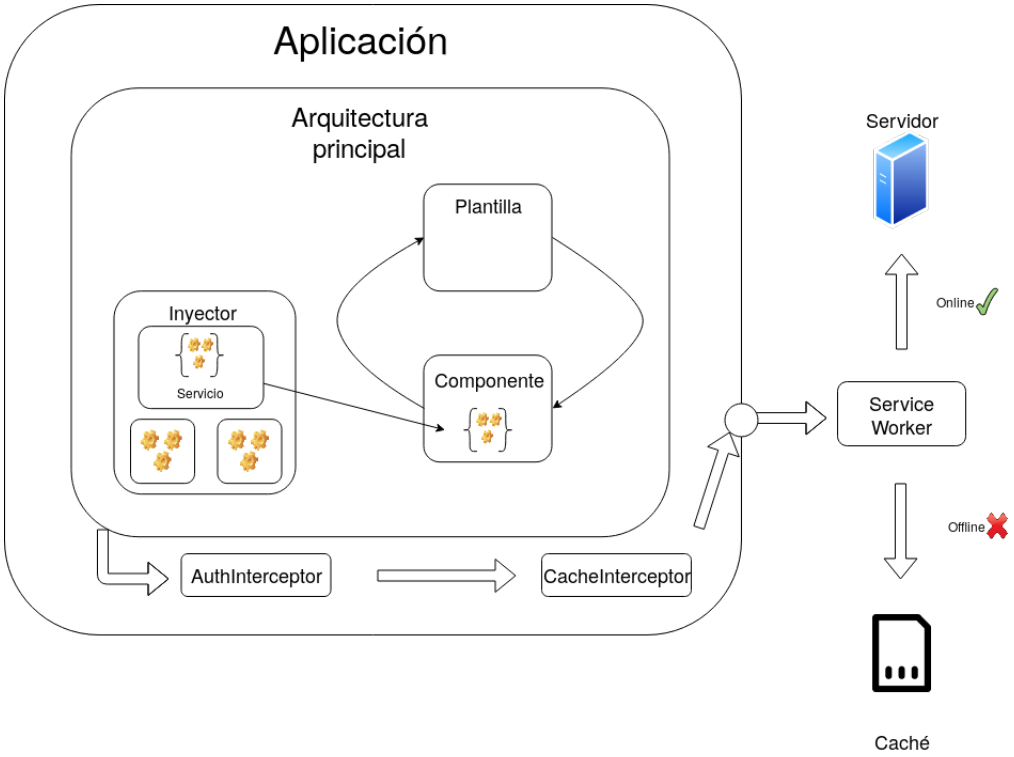

Figura 5.2: Estructura del cliente, adaptada de [3]

no sería una alternativa asequible tener toda la base de datos en memoria, principalmente por los costes que conlleva.

5.1.3 Cliente

Para el diseño del cliente se ha tenido en cuenta el patrón MVC (*Model View Controller*), usado para desacoplar la interfaz del usuario (vista), los datos (modelo) y la lógica de la aplicación (controlador), y conseguir a su vez separación de intereses [42].

Angular da soporte al patrón MVC explicado a través de tres elementos —servicios, plantillas y componentes— cuya función específica se puede observar en la Figura 5.2, en la que además se detalla la toda la estructura del cliente. El modelo, que contiene la lógica de la aplicación y se encarga de comunicarse con el *back-end* estaría representado por los servicios. Estos servicios son inyectados en los diferentes componentes para que puedan efectuar su labor de controladores, comunicando modelo y vista, para lo cual reaccionan a las acciones del usuario, las validan y le pasan los cambios al modelo. Por último, las plantillas corresponden a la vista, y se encargan de presentar la información al usuario.

Los servicios, por lo tanto, encierran la lógica para comunicar cliente y back-end. Para ello, envían peticiones al servicio REST ofrecido por el servidor, y transmiten las respuestas a

los componentes. Estas peticiones, antes ser enviadas, pasan por dos *HttpInterceptors*, esto es, dos componentes que interceptan todas las peticiones salientes, pudiendo modificarlas antes de transmitirlas al siguiente punto [43]. En nuestro caso, cada *interceptor* sirve a un propósito distinto:

1. **AuthInterceptor**. Modifica todas las peticiones que recibe, incluyéndoles una cabecera con el token de seguridad correspondiente para ser aceptadas por el *back-end*, como se explicará en la Sección 5.4.

2. **CacheInterceptor**. Es el encargado de almacenar una serie de peticiones cuando no se tiene conexión con el *back-end* para reintentar su envío una vez que se vuelva a tener conexión. Para que una petición sea almacenada tiene que cumplir las siguientes condiciones:

   (a) No puede haber conexión con el servidor.

   (b) El método de la petición tiene que ser POST, PUT o DELETE, ya que son los métodos que llevan a cabo operaciones que producen cambios en la aplicación (como un pago). Del cacheado de los métodos GET se encarga el Service Worker, explicado más adelante.

   (c) No puede ser una petición de inicio de sesión o registro, por motivos de seguridad, ni puede ser una petición para obtener un token de invitación, puesto que no habría forma de hacérselo llegar posteriormente cuando la conexión se reanudase.

   Si la petición cumple todas las condiciones, será almacenada y, una vez se detecte que se vuelve a tener conexión, será enviada. Una vez que la solicitud haya sido respondida, será eliminada de la caché.

Una vez que las peticiones salen del último interceptor, son interceptadas por el *Service Worker*. Los *service workers* son *scripts* que se ejecutan en segundo plano en el navegador, independientemente de las pestañas, y funcionan como un *proxy*, interceptando todas las peticiones realizadas por la aplicación. Se encargan del cacheado de solicitudes GET, permitiendo cargar una página de forma rápida cuando la conexión es inestable, o incluso cuando no existe conexión [44]. A la hora de implementar un *service worker*, se puede configurar la estrategia de cacheado. La implementación de Angular ofrece dos posibilidades: *performance* y *freshness*. La diferencia entre ambas se basa en cuándo se devuelve un recurso de caché; la primera opta por acudir a la caché primero, y si se encuentra el recurso, devolverlo sin llevar a cabo ninguna petición; la segunda, por el contrario, envía siempre la petición y sólo acude a la caché cuando no obtiene respuesta [45]. Aunque la primera opción optimice la velocidad de respuesta en

muchos casos, ésta no tiene por qué devolver datos actualizados, por lo que se descartó para esta aplicación, que debe trabajar siempre con los últimos datos disponibles.

La existencia de un *service worker* también es condición necesaria para que una aplicación web pueda ser PWA, y posibilita la recepción de notificaciones nativas, temas que se tratarán en las Secciones 7.3 y 7.4. Como condición, para poder registrar un *service worker*, la página tiene que ser servida a través de HTTPS, por motivos de seguridad [44].

5.2 Diseño de la aplicación

En esta sección se detallará la estructura interna de los componentes que forman parte de las arquitecturas explicadas en la Sección 5.1.

5.2.1 Servidor

El diagrama de paquetes del *back-end* se detalla en la Figura 5.3. Se pueden destacar dos paquetes principales:

- **src/test/java**: En él se incluyen los archivos de pruebas, sobre los que se hablará en detalle en la Sección 6.2.
- **src/main/java**: Aquí se encuentran las clases Java que, conjuntamente, ofrecen toda la funcionalidad del lado servidor. Tiene, a su vez, la siguiente estructura jerárquica de paquetes:

  Raíz (es.udc.fic.tfg.pablo): En él se encuentra el *entry point* de la aplicación Java, y una clase que se encarga de ejecutar tareas en segundo plano, explicadas en detalle en la Sección 6.1.5.

  model: Paquete donde se almacenan todas las entidades de la aplicación.

  repository: Incluye todos los DAOs del proyecto (Sección 5.2.1).

  security: Paquetes encargados de ejecutar controles de seguridad, autenticar usuarios, etc. Su función detallada está explicada en la Sección 5.4.

  service: Servicios que expone el modelo. Los componentes que incluye se describen en la Sección 5.2.1.

  utils: Aquí se alojan aquellas clases que proveen funcionalidades compartidas por más de un paquete.

  controller: En este paquete se encuentran los controladores REST, que exponen métodos accesibles desde fuera del servidor, y todas las clases requeridas para ello, descritas en la Sección 5.2.1

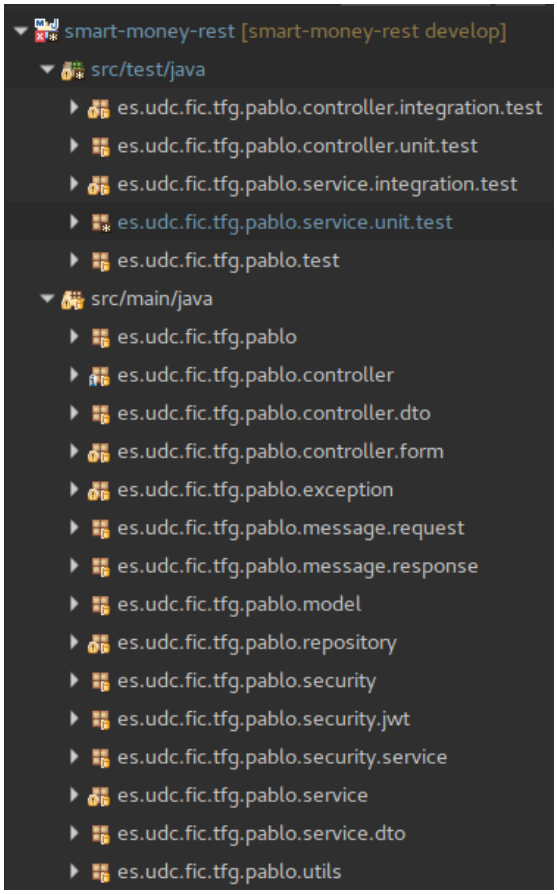

Figura 5.3: Esquema de paquetes del lado servidor.

Estos dos principales paquetes se apoyan cada uno en su directorio *resources* correspondiente, en el que se incluyen archivos de configuración que detallan, por ejemplo, las cadenas de conexión a su correspondiente base de datos, la configuración de las mismas, o propiedades usadas a lo largo del proyecto. A continuación, se tratarán más en detalle algunos de los paquetes explicados.

Entidades

Son las clases persistentes que representan las entidades diseñadas en la Sección 4.4 y que contienen, además de los atributos especificados, *getters* y *setters* para todos los atributos, constructores, y un *equals* y un *hashcode*. Para especificar que son clases persistentes y que sean mapeadas a base de datos, cada una de éstas clases se anotará con @Entity. Además, se utilizará la anotación de *Lombok*[9] @Data, que automáticamente incluye *getters*, *setters*, un constructor con todos los parámetros, *equals* y *hashcode* sin tener que escribirlos a mano y sin que aparezcan en el código, lo que hace que las clases queden mucho más limpias.

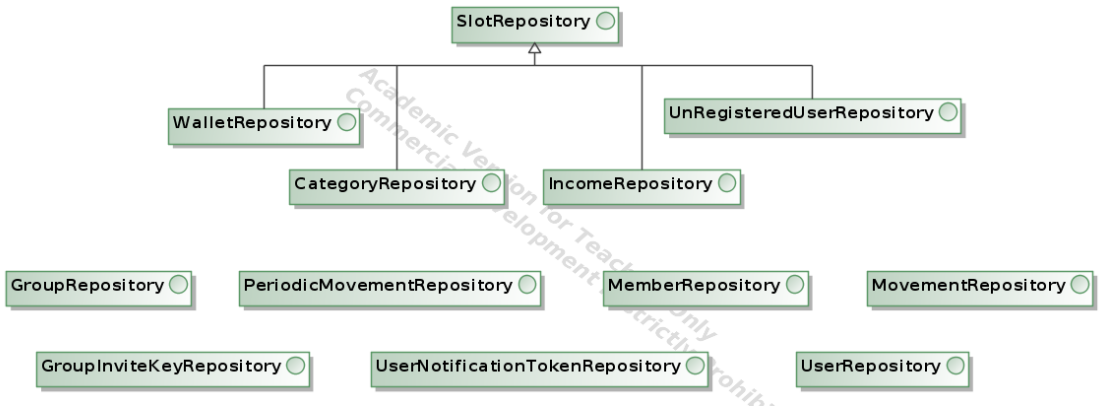

Figura 5.4: Diagrama básico de los repositorios

A la hora de elegir la estrategia de herencia para las relaciones entre *Slot* y sus subclases se optó, de entre las cuatro principales opciones ofrecidas por Java e Hibernate, por la estrategia ***Joined***. Ha sido escogida porque es la más correcta en cuanto a nivel de diseño (minimiza la información redundante), a pesar de no ser la más eficiente a la hora de recuperar los datos [46, 47]. En esta sección no se entrará a detallar cada una de las entidades ya que su función, relaciones y atributos son los mismos que los explicados en el capítulo de Análisis.

Repositorios (DAOs)

Se incluye uno por cada entidad, como se puede ver en la Figura 5.4. Todos se anotan con @Repository permitiendo de forma fácil su inyección en otras clases, y heredan de *JPARepository*, directa o indirectamente, una interfaz que provee Spring y que proporciona a cada interfaz que extienda de ella una serie de métodos CRUD, reduciendo el trabajo del programador.

Así, solo se tendrán que incluir aquellos métodos específicos a una clase que no contenga ya *JPARepository*. Habrá repositorios que no tendrán ningún método adicional a los proporcionados por el repositorio de Spring si su entidad no requiere de lógica específica a la hora de ser obtenida.

Servicios

La capa de servicios del modelo es donde se encuentra la mayor parte de la lógica de *back-end* e implementa los requisitos especificados en el Análisis. Análogamente a lo que se comentaba con los repositorios, cada servicio se marca con @Service. Al final del capítulo se incluye una tabla que relaciona los métodos de todos los servicios con las pantallas de la aplicación y las URLs del servicio REST.

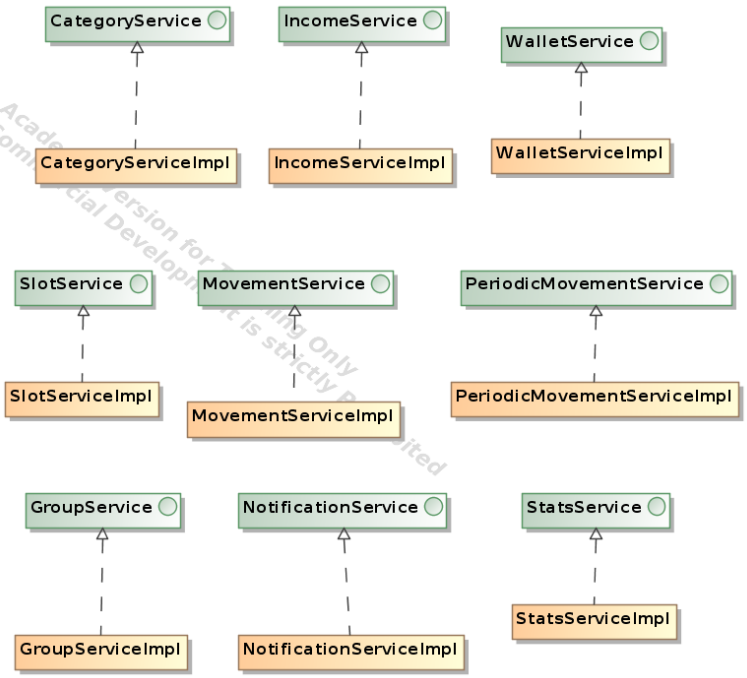

Figura 5.5: Diagrama básico de los servicios

En cuanto a su estructura, disponible en la Figura 5.5, se ve como para cada servicio hay una interfaz implementada por una clase, encargada de la lógica que se esconde detrás de la interfaz expuesta. De esta forma, sólo se expone la firma de los métodos, quedando la implementación oculta y pudiendo ser cambiada en cualquier momento sin llevar a cabo modificaciones en las clases que la utilicen, cumpliendo así con el *patrón interfaz* [48].

La división de esta capa en sus servicios individuales ha atendido a cuestiones conceptuales. De esta forma:

- Se tiene un servicio por cada tipo de *slot*, que se encarga de sus operaciones de creación, modificación u obtención.
- Aquellas operaciones comunes a todos los *slots* se encuentran en el *SlotService*.
- *MovementService* engloba la lógica de los diferentes movimientos, tanto grupales como individuales.
- Para acentuar la separación existente entre movimientos periódicos y sus instanciaciones, existe un servicio sólo para ellos.
- *StatsService* contiene las operaciones que devuelven estadísticas individuales y grupales.

- Las operaciones asociadas a un grupo, como su creación y modificación, gestión de miembros o cálculo de deudas conforman el *GroupService*.

- La gestión de suscripciones a notificaciones y el envío de las mismas se hace a través del *NotificationService*.

Cada servicio usará aquellos repositorios que requiera para sus operaciones y, en algunos casos, hará uso de otros servicios. Por ejemplo, se inyecta el servicio de notificaciones en *MovementService* para que al realizar un movimiento grupal se envíe una notificación a los usuarios correspondientes.

Esta capa también hace uso de DTOs (*Data Transfer Objects*) cuando la información que transmite lo requiere. Por ejemplo, las estadísticas, al no existir ninguna entidad que las represente como tal, ya que son cálculos derivados de la información almacenada, tienen que ser transportadas en un DTO a la capa superior; lo mismo pasa con el cálculo de deudas, donde se usa un DTO para transmitir el resultado de los cálculos realizados.

Controladores

Esta capa es la encargada de exponer una serie de servicios a través de una API REST. Para ello, se mapea cada URL a un método de un controlador (anotado con @RestController para facilitar esta tarea). Cada método hará uso del servicio expuesto por la capa inferior para llevar a cabo sus funciones, no teniendo más lógica que el mapeado de los objetos devueltos por los métodos del servicio del modelo a DTOs en los que se encapsulará toda la información requerida por la petición, evitando que se realicen múltiples peticiones. De esta forma, actúa de capa intermedia entre el servicio del modelo y el cliente, ejecutando los métodos requeridos por la petición entrante y «traduciendo» los objetos devueltos a DTOs que contienen toda la información necesaria.

Para hacer el mapeado de URLs del servicio REST a métodos del controlador se usarán las anotaciones @GetMapping (HTTP GET), @PostMapping (HTTP POST), @PutMapping (HTTP PUT) y @DeleteMapping (HTTP DELETE), a los que se indica a través de un parámetro la URL asociada. Además, se puede anotar un controlador con @RequestMapping para que todas las peticiones que cumplan un patrón sean asignadas a esa clase. Así, a la hora de mapear la URL 1a, se anotará un controlador como *@RequestMapping("/api/auth")* y un método de ese controlador con *@PostMapping("/login")*. Para ver la correspondencia entre URLs del servicio REST, métodos de los controladores y métodos de los servicios, acudir al Cuadro 5.1.

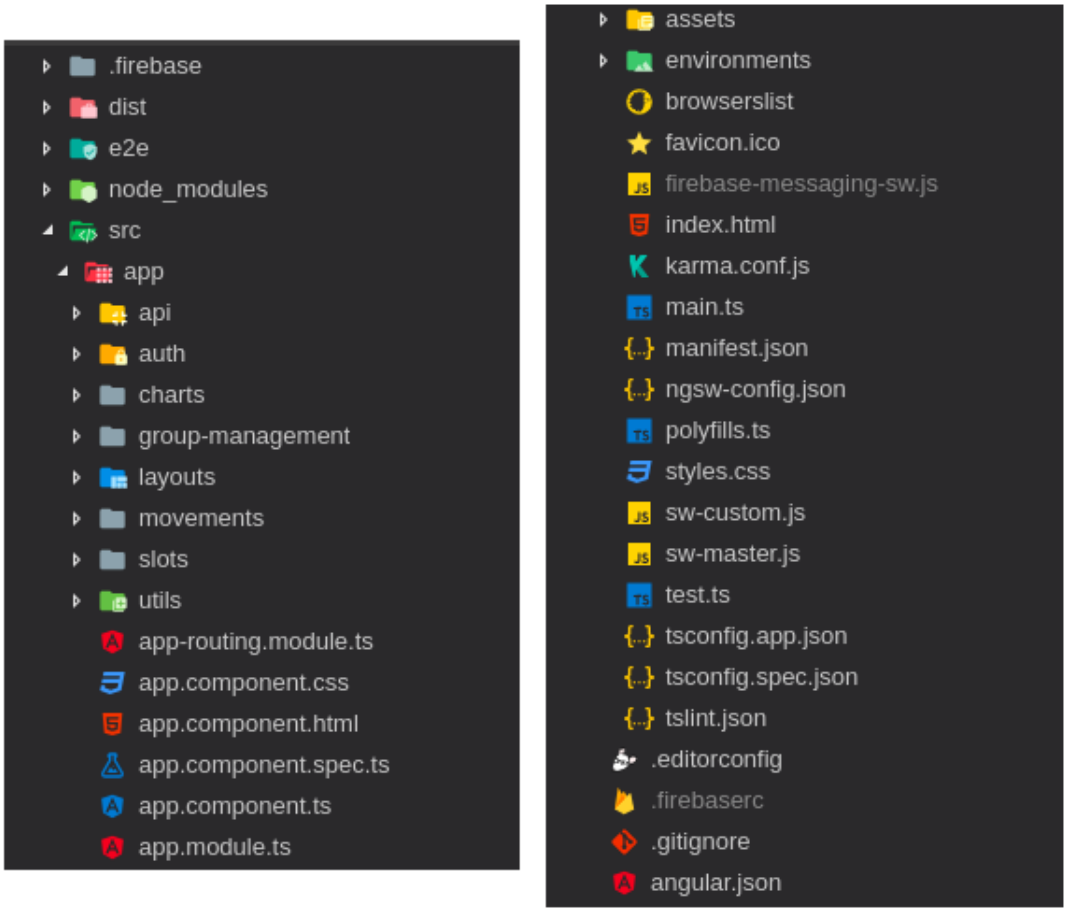

Figura 5.6: Esquema de paquetes del lado cliente.

5.2.2 Cliente

Internamente, la estructura de paquetes del cliente está disponible en la Figura 5.6. En el directorio raíz se encuentran una serie de archivos de configuración (*.gitignore*, *.editorconfig* ...) y los subdirectorios de más alto nivel:

- ***.firebase***: Directorio de archivos de configuración para el despliegue en *Firebase*, el cual se explica en detalle en la Sección 7.1
- ***dist***: Aquí se encuentra la aplicación Angular compilada. Los pasos para generarla se explican también en la Sección 7.1.
- ***e2e***: Archivos de configuración para las pruebas.
- ***node_modules***: Directorio con las diferentes librerías de *npm* utilizadas por la aplicación.
- ***src***: Directorio que aloja los principales componentes de la aplicación:
  - ***assets***: Aquí se almacenan los iconos de la aplicación.
  - ***environments***: Incluye archivos de configuración..
  - ***app***: Carpeta donde se aloja todo el código de la aplicación. Se incluyen todos los componentes y servicios, divididos en subdirectorios de acuerdo a su funcionalidad:
    - ***api***: Aloja utilidades para la comunicación con el *backend*, principalmente *DTOs*, y servicios compartidos por varios grupos de componentes.
    - ***auth***: Engloba las operaciones de gestión de usuarios.
    - ***charts***: Trabaja con las gráficas de gasto que se muestran.
    - ***group_management***: Incluye todas las funcionalidades relativas a un grupo –crearlo, editarlo, obtención de invitaciones, cálculo de deudas….
    - ***layouts***: Contiene los componentes relativos a la presentación de la aplicación: barra lateral, barra superior, modales y pestañas de la pantalla principal.
    - ***movements***: Incluye funcionalidades de movimientos, como su creación, listado o edición.
    - ***slots***: Los componentes incluidos aquí son los encargados de las operaciones con *slots*.
    - ***utils***: Utilidades de la aplicación. Se encuentra la lógica de almacenamiento de peticiones sin conexión, de recepción de notificaciones y de visualización de mensajes.

Cabe destacar que dentro de cada uno de estos directorios se pueden incluir tanto componentes como servicios. Así, por ejemplo el directorio *charts* contendrá los componentes encargados de representar las distintas gráficas y el servicio que obtiene los datos a representar en ellas.

La estructura de cada componente es bastante sencilla. En un mismo directorio se encuentran cuatro tipos de archivos: un .ts, es el archivo *Typescript* que representa al controlador en el patrón MVC explicado previamente; un .html que, junto con el .css representan la información al usuario, siendo la vista del MVC; y un .spec.ts, que es el archivo de pruebas del componente.

En cuanto a los servicios, constan de un archivo .ts que contiene toda la lógica y un archivo .spec.ts que recoge las pruebas.

5.3 Correspondencias

El objetivo de esta sección es documentar el diseño del servidor mostrando, para cada historia de usuario, la(s) URL(s) que permiten llevarla a cabo, y para cada una de ellas, qué método de qué controlador la implementa. Además, se asociarán a una o varias pantallas que permiten ejecutar la historia al usuario final, complementando la documentación del cliente. Nota: I: IndividualApi, G: GroupApi, N: NotificationApi.

Cuadro 5.1: Cuadro de correspondencias.

Cuadro de correspondencias.			
Historia	Pantalla(s)	REST URL	Controlador
1	4.2	1b	UserApi.authenticateUser
2	4.2	1a	UserApi.registerUser
3	4.3	-	-
4	4.5	2a	I.createIncome
5	4.5	2e	I.createWallet
6	4.5	2i	I.createCategory
7	4.5	2n	I.createDeposit
8	4.9	2p	I.createTransfer
9	4.8	2o	I.createPayment
10	4.6	3a	G.createGroup
11	-	3c	G.generateInviteLink
12	4.5	3h	G.addUnRegisteredUserToGroup
13	4.8	3m	G.createGroupMovement

Cuadro de correspondencias 5.1			
Historia	Pantalla(s)	REST URL	Controlador
14	4.8	3m	G.createGroupMovement
15	4.11	3r	G.getGroupDebts
16	4.11	3s	G.settleGroupDebts
17	4.8	3n	G.createGroupTransfer
18	4.9	2t, 3t	I.getUserMovements, G.getGroupMovements
19	4.10	2q	I.getMovementDetail
20	4.6	3e	G.getAllUserGroups
21	4.6	3g	G.modifyDefaultWallet
22	4.6	3f	G.modifyDefaultCategory
23	4.3	2b, 2f, 2k, 3i	I.getAllIncomes, I.getAllUserWallets, I.getAllUserCategories, G.getAllGroupUnRegisteredUsers
24	4.4	3p	G.getMembers
25	-	-	-
26	4.7	2c, 2g, 2j, 3j	I.getIncomesDetails, I.getWalletDetails, I.getCategoryDetails, G.getUnRegisteredUserDetail
27	4.7	2m	I.deleteSlot
28	4.10	2s, 3u	I.deleteMovement, G.deleteGroupMovement
29	4.10	2r	I.modifyMovement
30	4.11	2v, 2u, 2w, 3w, 3v	I.getUserHistoricalExpense, I.getUserExpenseByCategory, I.getUserExpenseByWallet, GS.getGroupHistoricalExpense, GS.getGroupStatsByUser
31	4.8	2y, 2x	I.createPeriodicMovement, I.getUserPeriodicMovements, G.getGroupPeriodicMovements
32	-	2z, 3y	I.deleteUserPeriodicMovement, G.deleteGroupPeriodicMovement
33	-	-	N.subscribeUserToken
Fin de Cuadro de correspondencias 5.1			

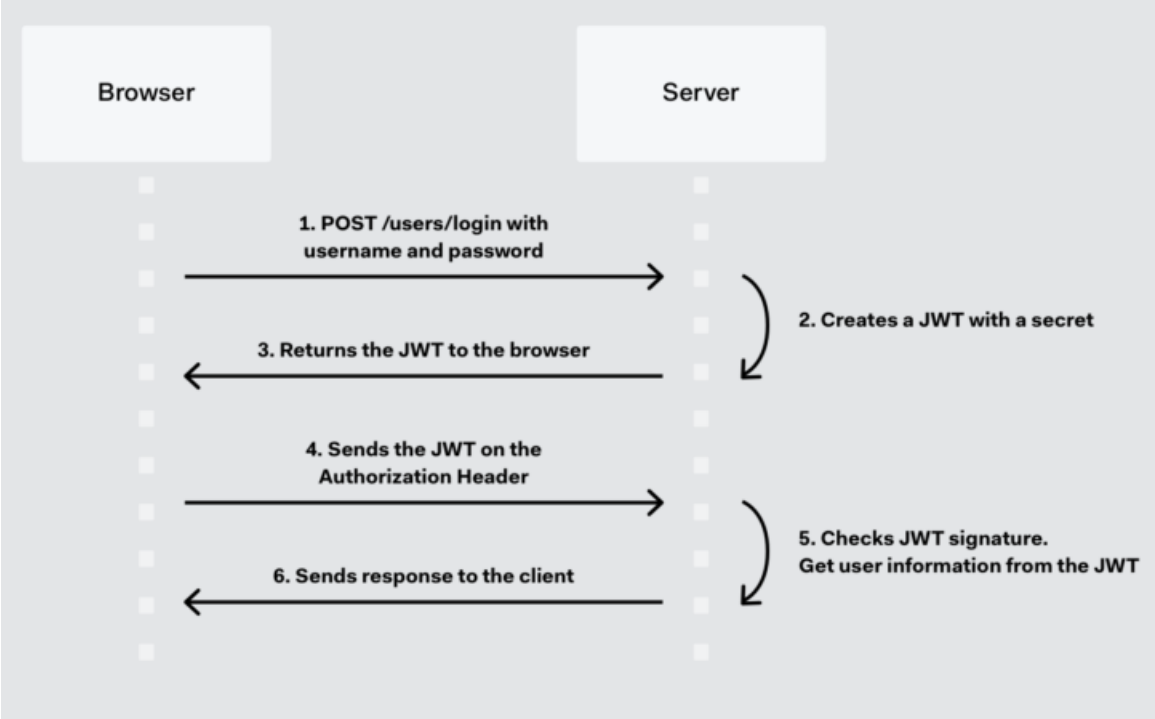

Figura 5.7: Funcionamiento de *JSON Web Token* [4]

5.4 Seguridad

Como se estipuló en uno de los requisitos no funcionales de la Sección 4.1.2, la aplicación debe controlar el acceso a los recursos. Por lo tanto, no se deben permitir solicitudes al servicio REST si no las realiza un usuario autenticado (excepto las solicitudes de inicio de sesión y registro) y, consecuentemente, el cliente deberá proveer un sistema de registro y autenticación al usuario, mediante el cual será identificado de forma inequívoca.

Para ello, se ha optado por el estándar *JSON Web Token (JWT)*, que proporciona una forma segura de transmisión de información basada en JSON y es muy útil para procesos de autorización [49]. Cuando el usuario inicia sesión, se le asigna un *token*, que será incluido en las cabeceras de las subsecuentes peticiones como método de identificación. Este funcionamiento se puede ver de forma gráfica en la Figura 5.7.

De esta forma, el servidor no almacena ningún tipo de sesión del usuario (es un sistema *stateless*), reduciendo el número de consultas a base de datos. Además, el tamaño de los JWTs es bastante reducido comparado con otras aproximaciones, por lo que el tamaño de las peticiones no se resiente al usarlos [4]. La implementación concreta de este estándar se explica en detalle en la Sección 6.1.6.

Para proteger los datos de los usuarios de la aplicación frente a intentos malintencionados

de obtener su información, se ha tenido en cuenta los siguientes tipos de ataques:

- ***SQL Injection***: Trata de insertar SQL en las peticiones al servidor para ejecutar consultas que lean información sensible de la base de datos o la modifiquen [50]. Para prevenirlo, todas las consultas que se construyen con datos introducidos por el usuario utilizan consultas parametrizadas de HQL, lo que impide cualquier intento de este ataque.

- ***Cross-site Scripting (XSS)***: Inyecta código malicioso en páginas web, dándole acceso a datos almacenados en el navegador [51]. En el caso de esta aplicación, teniendo acceso al almacenamiento de un usuario con la sesión iniciada se podrían hacer peticiones en su nombre y modificar información sensible del mismo. Angular es el encargado de prevenir este tipo de ataques, inspeccionando los valores que se insertan en el DOM y modificando aquellos que son sospechosos [52].

- ***Cross-site Request Forgery (CSRF)***: En este tipo de ataques, se provoca que el usuario visite una página, mientras tiene una sesión iniciada en una aplicación web, para enviar una petición maliciosa al servidor haciéndose pasar por el usuario [53]. *Spring security* previene este problema por defecto enviando una *cookie* al cliente que tiene un token generado de manera aleatoria; desde ese momento, las peticiones recibidas tendrán que traer ese token para ser aceptada [54]. Angular también tiene soporte por defecto para prevenir este tipo de ataques, añadiendo automáticamente la *cookie* en las peticiones salientes [55].

Capítulo 6

Implementación y pruebas

6.1 Implementación

En esta sección se tratarán aquellos algoritmos que se han considerado más importantes en el correcto funcionamiento de la aplicación y otros detalles de la implementación que complementan lo tratado en capítulos previos.

6.1.1 Algoritmo de cálculo de deudas

Uno de los objetivos de este proyecto es permitir a un grupo de usuarios anotar pagos que afecten a varios miembros para posteriormente establecer las deudas individuales entre ellos. Este problema está considerado un problema de complejidad NP [56].

Para resolverlo, se ha desarrollado un algoritmo basado en el que usa *Settle Up* [56] que asegura, para un grupo de N miembros, encontrar una solución de N-1 transacciones en el peor de los casos. Además, el algoritmo elegido minimiza la cantidad total de dinero transferida, mejorando a otras alternativas. El único dato que se tiene en cuenta para llevar a cabo los cálculos es el balance de cada usuario, que será positivo cuando se le deba dinero, y negativo si, por el contrario, debe dinero. Para acelerar el cálculo, en esta implementación los balances se almacenan en base de datos, actualizándose en cada movimiento grupal y por lo tanto evitando realizar cálculos a mayores a la hora de saldar deudas.

Para calcular las deudas, se ordenan los miembros en función de su balance, de más positivo a más negativo, y se determina una transacción del último al primero. El valor de la misma será el menor balance de los dos en valor absoluto, por lo que como mínimo uno de los dos siempre quedará con balance cero y se dejará fuera de los siguientes cálculos. El proceso se repite hasta que todos los miembros queden con balance cero.

Para evitar transferencias muy pequeñas que no tuviesen sentido en un escenario real, se estableció un umbral a partir del cual el balance se considera 0, reduciendo el número de cálculos innecesarios y el número de transacciones finales. A continuación se muestra el

código de la implementación del algoritmo:

```
List<Member> members = memberRepository.findAllByAssociatedGroupId(groupId);
List<DebtDTO> transfers = new ArrayList<>();
int settledMembers = 0;
Map<Member, Double> originalBalances = new HashMap<>();

for (Member member : members) {
  originalBalances.put(member, member.getBalance());
  if (Math.abs(member.getBalance()) <= alpha) {
    member.setBalance(Double.NEGATIVE_INFINITY);
    settledMembers++;
  }
}

while (settledMembers < members.size()) {
  // Members sorted from lowest balance to highest
  sortByBalance(members);
  // Member with the lowest balance (the one who owes the most money)
  Member source = members.get(0 + settledMembers);
  // Member with the highest balance
  Member dest = members.get(members.size() - 1);

  double transferAmount = 0;
  double sourceBalance = source.getBalance();
  double destBalance = dest.getBalance();

  // The amount to transfer is the lowest between the source member's balance
      and the dest member's
  transferAmount = (Math.abs(sourceBalance) > Math.abs(destBalance)) ?
      Math.abs(destBalance) : Math.abs(sourceBalance);
  sourceBalance += transferAmount;
  destBalance -= transferAmount;

  // DebtDTO is a DTO for transferring the debt calculations
  DebtDTO transfer = new DebtDTO(source, transferAmount, dest);
  transfers.add(transfer);

  source.setBalance(sourceBalance);
  dest.setBalance(destBalance);

  // alpha = 0.009 -> valor umbral
  if (Math.abs(sourceBalance) <= alpha) {
    settledMembers++;
    source.setBalance(Double.NEGATIVE_INFINITY);
  }
  if (Math.abs(destBalance) <= alpha) {
    settledMembers++;
    dest.setBalance(Double.NEGATIVE_INFINITY);
  }
}
```

6.1.2 Funcionalidad *offline*

Otro de los grandes objetivos de este proyecto ha sido ofrecer a los usuarios la posibilidad de anotar sus operaciones en ausencia de conexión a Internet. Para ello, se encontraron dos grandes problemas: permitir interactuar con la aplicación sin tener que cargar los recursos de *back-end* (páginas, iconos, estilos, datos...) y poder almacenar cambios (movimientos, modificaciones, etc.) para que se materialicen una vez que vuelva la conexión.

Para lo primero se ha hecho uso del *service worker* que proporciona Angular. Tras añadirlo al proyecto a través de su CLI, se proporciona el archivo *ngsw-config.json* para poder configurar diferentes parámetros del mismo de acuerdo con la especificación de la Sección 5.1.3 y que está disponible en el apéndice A.

Este *service worker* se encarga por una parte de cachear el contenido estático de la aplicación. Los archivos .html, .css y .js son almacenados en modo *prefetch*, es decir, se obtienen todos al cachear la aplicación por primera vez [45]. Aunque esta opción conlleve mucho uso de Internet la primera vez que se carga la aplicación, permite su uso por completo sin conexión aun a aquellas partes de la misma que no se hayan visitado previamente. En cambio, recursos como iconos, imágenes o fuentes son cacheados en modo *lazy*, siendo almacenados únicamente cuando son requeridos por primera vez al ser elementos de mucho mayor tamaño [45].

Por otra parte, este *service worker* también es el encargado de cachear las peticiones GET al servidor REST para complementar el cacheado de los archivos estáticos. De esta forma, se permite navegar por la aplicación en ausencia de conexión, mostrando los datos cacheados de la última vez que se tuvo acceso al *back-end*. En el caso de las peticiones HTTP, sólo son cacheadas cuando son enviadas (al contrario que los recursos estáticos), por lo que para poder ver estos datos sin conexión han tenido que ser requeridos previamente.

Para permitir a los usuarios almacenar cambios sin conexión se tuvo que llevar a cabo una implementación ajena a los *service workers*, lo que conllevó la detección de conexión a Internet y el almacenamiento de los cambios cuando no se tenga.

Algoritmo de detección de conexión

Para desarrollar una aplicación que permita al usuario trabajar *offline*, uno de los principales retos ha sido detectar si existe conexión con el servidor y las fluctuaciones de la misma.

La primera aproximación desarrollada se basó en los eventos de *JavaScript ononline* y *onoffline* [57, 58]. Estos eventos detectan bien las modificaciones manuales de la conexión (encender/apagar el *WiFi*, conectar/desconectar la conexión *Ethernet*) pero dan falsos positivos cuando, por ejemplo, existe señal *WiFi* pero no se tiene conexión real a Internet.

Para una mejora de la detección de la conexión, se llevaron a cabo varios cambios. A la

hora de detectar caídas en la red, además de usar el evento citado, se hacen comprobaciones en cada petición saliente que no sea GET, para lo cual se usa el método *intercept* del *CacheInterceptor*, implementado de la interfaz *HttpInterceptor* de Angular que permite capturar todas las peticiones salientes. Para cada petición, se comprueba si devuelve un código de error con valor 0, lo que indicaría que no se puede conectar con el *back-end* y por lo tanto, que el usuario está *offline*. No se puede llevar a cabo esta detección sobre peticiones GET porque éstas son cacheadas por el *service worker* y cuando no hay conexión a Internet, éste devuelve la respuesta almacenada en caché y se detectaría como que se tiene conexión con el servidor, cuando no es así.

Por otro lado, para detectar la reanudación de la conexión, se dejó de utilizar el evento *ononline*, sustituyéndolo por un algoritmo de detección de conexión propio que envía peticiones al servidor y comprueba si es alcanzable, es decir, si no devuelve un código de error 0.

Las peticiones son enviadas con un tiempo de espera entre ellas que crece de forma exponencial, hasta un máximo de 15 minutos entre una petición y la siguiente, partiendo de una espera de 1 segundo. Esto se ha hecho para detectar de forma rápida si la conexión se ha caído sólo durante unos segundos, al entrar en una zona de mala cobertura o en una breve caída del servicio; pero evitando enviar muchas peticiones cuando se desactiva intencionadamente la conexión a Internet (al estar en un país extranjero, por ejemplo) que, de hacerse de forma muy constante, supondrían un impacto en la batería del dispositivo.

También se utilizan las peticiones del usuario a la hora de detectar que se vuelve a estar *online*. Cuando se recibe una respuesta que no contiene el error 0, el estado de la aplicación es *offline* y el método de la solicitud no es GET (por el motivo explicado previamente) significa que el servidor vuelve a ser alcanzable y por lo tanto se dejan de enviar peticiones de detección de conexión al servidor.

Por último, al entrar en la aplicación también se hace una comprobación del estado de la conexión para enviar peticiones pendientes, en el caso de que las hubiese y se haya restaurado la conexión.

Almacenamiento de cambios sin conexión

Para almacenar los cambios realizados de forma *offline* en el navegador del usuario se ha utilizado IndexedDb, un sistema de base de datos en el navegador que puede funcionar sin conexión y que los datos sean almacenados durante largos periodos de tiempo, manteniéndose independientemente de si el usuario cierra el navegador o apaga el dispositivo [15]. A la hora de trabajar con IndexedDb, se ha utilizado Dexie, una librería que trabaja sobre IndexedDb y ofrece una API más sencilla [15].

El encargado de controlar el almacenamiento de peticiones es *CacheInterceptor* que, cuan-

do le llegue una petición, comprobará si la aplicación tiene conexión, y en caso contrario, la almacenará para su posterior envío. De las operaciones de almacenamiento se encarga el *OfflineService* para separar las responsabilidades entre componentes. Para que una petición sea almacenada tiene que cumplir los requisitos especificados en la Sección 5.1.3.

Por cada petición se almacenará:

- Un **identificador** numérico, generado automáticamente al añadir una nueva petición.
- El **usuario** que realizó la petición, para distinguir posteriormente a qué usuario pertenece cada una (en aquellos casos que varios usuarios utilicen el mismo navegador para usar la aplicación).
- La propia **petición**.
- El **número de intentos** de ejecutar la petición. Será 0 al almacenarla, y se incrementará cada vez que se envíe y se obtenga un error como respuesta. El valor máximo será 4, momento en el que la petición no se reintentará y será descartada.

Cuando se detecte que existe conexión otra vez, se enviarán todas las peticiones almacenadas por el usuario. Una vez hayan retornado todas, se comprobará si alguna devolvió un error, en cuyo caso se incrementará su contador de intentos. En caso contrario, la petición se eliminará del almacenamiento.

6.1.3 Concurrencia

Una parte crítica en una aplicación de este tipo es el manejo de la concurrencia. Por cómo están diseñados los niveles de aislamiento en Hibernate, acciones como crear dos movimientos desde el mismo monedero en el mismo instante pueden tener problemas de *lost update*. Una *lost update* se da cuando dos transacciones intentan actualizar la misma columna de la misma fila en el mismo momento; ambas leen el mismo valor inicial y lo sobreescriben, por lo que una de las dos operaciones se pierde [59].

Este problema se podría dar si se enviase un lote de peticiones realizadas *offline* por un usuario que modificasen datos comunes (varios pagos desde el mismo monedero, sobre la misma categoría…) o si varios usuarios realizasen un pago en un grupo en el mismo momento. Por lo tanto, su prevención es clave para el correcto funcionamiento de la aplicación.

Para ello, se añadió la anotación *@Transactional(isolation = Isolation.SERIALIZABLE)* a los métodos que realizan operaciones que son susceptibles del problema de las *lost updates*. Esta anotación previene este problema y otros relacionados con la concurrencia (*dirty reads*, *non-repeatable reads* y *phantom reads*) [60].

Utilizando esta anotación, cuando se intenta actualizar la misma fila de forma concurrente por dos operaciones distintas, una de las dos lanzará una excepción y no se llevará a cabo.

Cuando esto sucede, se reintenta la operación fallida hasta que se completa, o hasta que produce un error no relacionado con la concurrencia.

Aunque establecer un nivel de aislamiento tan alto restrinja en cierto modo la concurrencia de la aplicación, se ha tenido en cuenta que sólo algunas operaciones se han especificado con este nivel de aislamiento, y que aquellas que en las que sí se ha especificado no siempre serán ejecutadas simultáneamente, por lo que se mantendrán unos niveles de concurrencia bastante altos.

6.1.4 Envío y recepción de notificaciones

Para enviar notificaciones desde servidor, y recibirlas y mostrarlas en el cliente, se ha utilizado *FCM (Firebase Cloud Messaging)*, una solución que ofrece una API para enviar notificaciones desde Java y recibirlas en un cliente web, para lo cual se basa en dos protocolos:

- ***Notifications API***: permite mostrar notificaciones del sistema desde una página web, incluso cuando ésta se encuentra en segundo plano o cerrada [61].

- ***Push API***: permite recibir mensajes enviados desde un servidor a una página web, independientemente de que esté o no en primer plano, para lo cual se utiliza un *service worker* [17].

El proceso de envío de notificaciones comienza solicitando permiso al usuario, tras lo cual se obtiene una *PushSubscription*, que identifica al dispositivo del usuario [62]. Este identificador es almacenado en el servidor para posteriormente poder enviar notificaciones al usuario que corresponda. La lógica de estas operaciones reside en el servicio *MessagingService* del cliente, que además se usa para recibir mensajes en primer plano y eliminar *tokens* cuando el usuario cancela la suscripción.

El siguiente paso es el envío de una notificación desde servidor. Para ello, es necesario tener un proyecto en *Firebase* que, en nuestro caso, ya estaba creado porque es donde se aloja la parte cliente del proyecto. Después, se tiene que generar un archivo de configuración único para el proyecto a través del cliente de *Firebase*, que será utilizado para enviar notificaciones.

En el proceso de envío de notificaciones intervienen cuatro elementos:

- El **servidor**, que se encarga de enviar las notificaciones. Para ello, obtiene los *tokens* de los usuarios que se desea notificar, genera un mensaje con los datos a enviar y parámetros de configuración, y lo envía utilizando la API proporcionada por *Firebase*. De los parámetros de configuración cabe destacar el *TTL (Time-to-Live)* que indica cuánto puede esperar un mensaje a ser recibido antes de descartarlo.

- El **Push Service**. Actúa de intermediario entre servidor y cliente, y se encarga de recibir mensajes del servidor, validarlos y reenviarlos al navegador del cliente cuando tenga conexión. Si no la tuviese, *Push Service* almacena el mensaje hasta que el navegador recupera la conexión, o hasta que el *TTL* es superado [63].
- El **navegador** del cliente, que recibe la notificación del *Push Service*, desencripta los datos y se los pasa al *service worker* asociado a la suscripción correspondiente al mensaje, o a la propia aplicación web si el mensaje se recibe con ella en primer plano [63].
- El ***service worker*** se encarga de llevar a cabo las operaciones oportunas al recibir una notificación, que en nuestro caso consisten en mostrarla al usuario. A la hora de configurar un *service worker* con este fin, hubo que crear dos adicionales al que proporciona *Angular*, (*firebase-messaging-sw.js* y *sw-master.js*), ya que no permite su configuración más allá de los parámetros ya comentados.

  El primero de los dos creados usará la librería proporcionada por *Firebase* para detectar notificaciones entrantes y mostrarlas al usuario cuando la aplicación está en segundo plano. La lógica de estas operaciones viene implementada en la librería de *Firebase*, por lo que en el *service worker* sólo se llevan a cabo las operaciones de importación e inicialización. El segundo *service worker* se encarga de importar los otros dos, el generado por Angular y el de *Firebase*, de forma que actúen conjuntamente.

En cuanto a cuándo se envían las notificaciones, se han determinado dos casos. Primero, cuando un movimiento periódico es efectuado, se notifica al usuario dueño del mismo; y segundo, cuando un usuario realiza una operación en un grupo, se notifica a todos los miembros afectados (exceptuando el propio usuario que realiza el movimiento).

6.1.5 Tareas periódicas

Permitir a un usuario programar movimientos a través de operaciones periódicas supuso un reto, pues hubo que lanzar una tarea en segundo plano que se encargase de comprobar cuándo había que materializar esos movimientos.

Con este fin, *Spring* proporciona la anotación *@Scheduled*, con la que se puede marcar un método para que se ejecute cuando nosotros deseemos. Para ello, el método no puede devolver nada ni aceptar ningún parámetro, y se tiene que permitir la ejecución de tareas de este tipo en el proyecto, anotando con *@EnableScheduling* la clase que actúa como *entry point* de la aplicación (*SmartMoneyRestApplication*) [64].

Además, se debe indicar en el método información adicional para saber cuándo tiene que ser ejecutado. En este caso, se usó una *cron expression*, una cadena de texto en la que se especifican varios campos que permiten interpretar con qué frecuencia se tiene que llevar a cabo una tarea [65].

Se decidió que la tarea periódica debía ser ejecutada todos los días a una hora fija, y comprobaría, entre los movimientos periódicos, cuáles cumplían los criterios para ser ejecutados. El código es bastante sencillo y se muestra debajo, sólo se encarga de definir la periodicidad con la que debe ser ejecutado, y delega en un método del *PeriodicMovementService* para ejecutar la lógica correspondiente. Está dentro de la clase *ScheduledTasks* pensada para dar cabida a este tipo de métodos si en un futuro se necesitasen más.

```
@Scheduled(cron = "0 0 2 * * ?") // at 2AM every day
public void generatePeriodicMovements() {
  log.info("Periodic movements check. {}", Instant.now());
  this.periodicMovementService.runPendingMovements();
}
```

6.1.6 Seguridad

A la hora de securizar el **servicio REST** con JWT se hizo uso de *Spring security*, otro *framework* proporcionado por Spring, que facilita los procesos de autenticación y control de acceso [66]. Provee por defecto de un sistema de autenticación y autorización, que se puede modificar para adaptarlo a la aplicación que se está desarrollando. En este caso, se incluyeron los siguientes componentes para trabajar con JWT:

1. *WebSecurityConfig*: clase en la que se especifican varios parámetros de configuración de seguridad a *Spring security*:

   - *BCryptPasswordEncoder*, una clase proporcionada por el propio *framework*, será la encargada de cifrar y las contraseñas de los usuarios para su almacenamiento, y descifrarlas posteriormente cuando sean requeridas.
   - Las peticiones HTTP han de estar autenticadas, excepto aquellas URLs expuestas para iniciar sesión y registrar un usuario (como se vio, las que comiencen por /api/auth).
   - Uso de sesiones *stateless*, que no vienen seleccionadas por defecto, para trabajar con JWT.
   - Inclusión de un filtro previo en la cadena de procesado de peticiones detallado en el punto 3.

2. *JwtProvider*: clase que se encarga de generar y validar JWTs, y extraer el usuario que realiza la petición de la información incluida en el JWT.

3. *JwtAuthTokenFilter*: se ejecuta cada vez que se recibe una petición y se encarga de obtener el *token* de la cabecera y utilizar el método ofrecido por *JwtProvider* para validarlo.

También se ha marcado cada método de cada controlador con @PreAuthorize(User), una anotación de Spring que indica el rol que debe tener el usuario para tener acceso a esa URL del servicio REST. Aunque en la implementación actual, todos los usuarios tendrán rol *User*, ya que no se ha considerado la Figura de un administrador, ni ningún rol específico de usuario, sí se ha dejado preparada la aplicación por si en un futuro se deciden añadir nuevos roles.

Por último, el servicio REST expone dos *endpoints*, 1a y 1b, para el inicio de sesión y registro, respectivamente. Para iniciar sesión, al recibir unas credenciales, se comprueba que sean correctas (para lo cual se delega en el *AuthenticationManager* de *Spring security*) y se genera un JWT para el usuario, que se devuelve en el cuerpo de la respuesta. Por otro lado, al registrarse, se comprueba que el nombre de usuario y correo obtenidos no existan en la base de datos y se crea un nuevo usuario. Una vez hecho esto, se transfiere la ejecución al método de inicio de sesión para generar un JWT y enviarlo.

En cuanto al lado **cliente**, el servicio *auth-service* contiene las peticiones de inicio de sesión y registro. Cuando se envía la petición de inicio de sesión, si las credenciales son correctas, se utiliza el servicio *token-storage* para el almacenamiento de las mismas. Si el usuario selecciona la opción de recordar su sesión, éstas serán almacenadas en el *Local Storage*, un almacenamiento en el navegador sin tiempo de expiración y cuyos datos se mantienen aún cerrando el navegador [67]; sin embargo, si el usuario no marca la opción de recordar sus credenciales, serán almacenadas en el *Session Storage*, que se limpia cada vez que la pestaña se cierra.

Por otra parte, el *AuthInterceptor* (cuya función se explica en el punto 1 de la Sección 5.1.3) es el encargado de interceptar todas las peticiones salientes e incluirles la cabecera con el JWT, que obtendrá del *Local Storage* o *Session Storage* según corresponda.

6.2 Pruebas

En esta sección se detallará el tipo de pruebas que han sido llevadas a cabo a lo largo del proyecto para asegurar que la calidad del producto supera unos umbrales que aseguran el correcto funcionamiento del mismo. Se han dividido principalmente en dos categorías:

6.2.1 Pruebas funcionales

Comprueban el correcto funcionamiento del sistema contra una serie de requisitos funcionales. Para ello, las funcionalidades son probadas pasándoles entradas y comparando el resultado obtenido con el esperado [68]. Se suele diferenciar entre varios niveles de pruebas funcionales: de unidad, de integración y de aceptación. Las primeras ponen a prueba la unidad más pequeña de funcionalidad (un método), reemplazando las dependencias del objeto probado con *mocks*, objetos que simulan el funcionamiento de las dependencias de una clase

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
es.udc.fic.tfg.pablo.service		94%		92%	35	342	23	819	16	207	0	9
es.udc.fic.tfg.pablo.controller		86%		47%	27	116	74	368	6	81	0	5
es.udc.fic.tfg.pablo.security.jwt		29%		0%	9	16	35	45	5	12	0	3
es.udc.fic.tfg.pablo.model		94%		77%	5	45	8	167	0	34	0	16
es.udc.fic.tfg.pablo.utils		0%		0%	7	7	6	6	2	2	1	1
es.udc.fic.tfg.pablo.service.dto		64%		n/a	1	3	4	11	1	3	0	1
es.udc.fic.tfg.pablo		43%		n/a	2	5	5	8	2	5	0	2
es.udc.fic.tfg.pablo.controller.dto		99%		93%	1	16	0	57	0	8	0	8
es.udc.fic.tfg.pablo.security		100%		n/a	0	8	0	15	0	8	0	2
es.udc.fic.tfg.pablo.security.service		100%		n/a	0	11	0	15	0	11	0	2
es.udc.fic.tfg.pablo.controller.form		100%		n/a	0	2	0	9	0	2	0	2
es.udc.fic.tfg.pablo.exception		100%		n/a	0	6	0	12	0	6	0	3
es.udc.fic.tfg.pablo.message.response		100%		n/a	0	1	0	6	0	1	0	1
Total	826 of 9.866	91%	80 of 396	79%	87	578	155	1.538	32	380	1	55

Figura 6.1: Ejemplo del informe generado por JaCoCo de las pruebas de todo el proyecto Java

de forma que en la prueba sólo interviene el funcionamiento de la unidad probada [69].

Por el contrario, en las pruebas de integración intervienen todas aquellas dependencias del método probado, de forma que se pone a prueba la interacción de los componentes del sistema junto con la funcionalidad específica que se está probando [70]. Por último, las pruebas de aceptación se hacen a nivel de historia de usuario, poniendo a prueba todos los componentes que forman parte del sistema y que intervienen en la ejecución de las historias de usuario.

En este proyecto se han llevado a cabo, principalmente, pruebas de integración y de aceptación, ya que se ha considerado que las de unidad requerían demasiado tiempo para ser implementadas debido a la necesidad de utilizar *mocks*, por lo que su función ha sido sustituida por las de integración. Estas pruebas han sido incluidas en el ciclo de desarrollo del proyecto, siendo necesaria su realización antes de marcar una historia de usuario como «Terminada».

Para su implementación se ha utilizado la librería JUnit, un *framework* para escribir y ejecutar *tests* repetibles en Java, proporcionando herramientas para agilizar el proceso [26]. Se ha complementado con JaCoCo, una librería para medir cobertura de código en Java [27]. Ambos se integran con Maven de forma que al ejecutar *mvn test* JUnit comprueba la correcta ejecución de todas las pruebas del proyecto, y si posteriormente se ejecuta *mvn jacoco:report*, JaCoCo genera un informe de la cobertura de las mismas, localizado en target/site/jacoco/index.html y cuyo resultado se puede observar en la Figura 6.1.

A la hora de valorar las pruebas de integración realizadas, se ha buscado que la cobertura fuese la mayor posible porque, aunque no sea un seguro de que no existan *bugs* en el código, reduce las posibilidades de su existencia. En total, se han implementado 354 *tests*, que principalmente se han dividido en dos grupos:

- *Tests* de los **servicios**, donde reside casi toda la lógica de la aplicación. En ellos se ha buscado forzar el mayor número de casos posibles para intentar asegurar que todas las combinaciones de operaciones del usuario hayan sido comprobadas previamente.

- *Tests* de los **controladores**, donde prácticamente no hay lógica, por lo que las pruebas comprueban que los parámetros recibidos son correctamente aceptados, y las respuestas

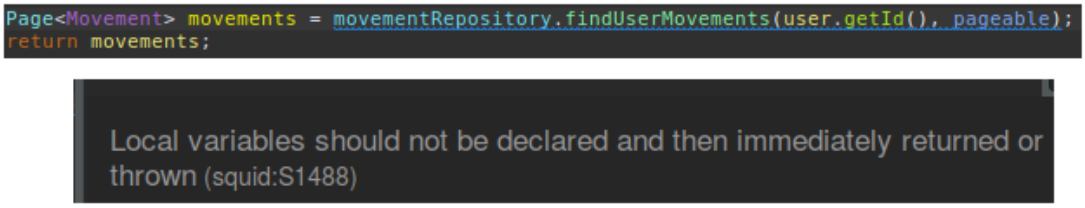

Figura 6.2: Ejemplo de un trozo de código mal implementado notificado por *SonarLint*

contienen toda la información esperada.

Como se ve en la Figura 6.1, la cobertura de instrucciones es superior al 90%, indicando que las pruebas han pasado por casi todas las sentencias de código. Por otra parte, la cobertura de ramas ronda el 80%. Este tipo de cobertura indica, para cada sentencia de decisión en el código (*ifs*, bucles, etc.) si se han cubierto todas las posibles ramas. Se puede ver que es sobre todo inferior en los controladores, y se debe a que en ellos es donde se lleva a cabo la captura de errores de concurrencia explicados en la Sección 6.1.3 y que no se ha podido probar en este tipo de *tests*. El resto de código mantiene una cobertura de ramas más elevada.

En cuanto a las pruebas de aceptación, se han llevado a cabo también durante todo el proyecto, primero por el equipo de desarrollo mientras se realizaban las iteraciones, comprobando que las funcionalidades incluidas no fallasen, y después en cada reunión de todo el equipo al término de un *sprint*, en las que se ejecutaron todas las historias de usuario de ese incremento, comprobando que la funcionalidad fuese la correcta y careciese de fallos.

6.2.2 Pruebas estructurales

Son pruebas que se ocupan de la calidad interna del *software* desarrollado. Para ello, se ha incluido la extensión *SonarLint* [71] para *Eclipse*, la cual se ha encargado de analizar el código desarrollado según se ha ido escribiendo, resaltando los problemas que ha ido detectando y ofreciendo una explicación sobre ellos.

Por ejemplo, en la Figura 6.2 se ve un trozo de código en el que se obtiene el resultado de un método, se asigna a una variable, y justo a continuación se devuelve la variable como resultado. Como se puede ver, la extensión lo resalta porque detecta que no es correcto hacer eso, y en su lugar se debería devolver directamente el resultado del método llamado.

Gracias a este tipo de ayudas, se ha conseguido desarrollar un código más limpio y por lo tanto, más mantenible, mientras que se han reducido el número de posibles *bugs* debidos a un mal uso del lenguaje de programación.

Capítulo 7

Solución desarrollada

El objetivo de este capítulo es ilustrar el resultado de la aplicación desarrollada. Para ello, se comentará cómo se puede llevar a cabo un despliegue de la misma, se describirán sus características principales y se terminará con una pequeña visita guiada por sus principales funcionalidades.

7.1 Despliegue

Para desplegar la aplicación se distinguirá entre el despliegue en una máquina local y el despliegue a un servidor remoto:

7.1.1 Local

Los requisitos para ejecutar el proyecto en local son los siguientes:

- **Maven** versión 2.0 mínimo.
- **Java JDK** versión 1.8 mínimo.
- **Angular CLI**.
- **Node.js y npm**.
- **Http-server**

Para levantar el servidor REST se ejecuta desde la raíz del proyecto Java

```
mvn spring-boot:run
```

que por defecto estará levantado en el puerto 8080. Para hacer lo propio con el cliente Angular, se instalan las dependencias, se compila el proyecto y se levanta un servidor *http-server* con los archivos generados, alojados en *dist* a través de los siguientes comandos:

```
npm install
ng build --prod
http-server -c -1 -p 4200 dist/smart-money-client
```

7.1.2 Remoto

Para el despliegue de la aplicación en los servidores de *Firebase* y *Heroku* se necesitarán ambas *CLIs* instaladas y configuradas (sesión iniciada y un proyecto disponible), además de Node.js, npm y git. El despliegue del servidor REST se hace con el siguiente comando

```
git push heroku master
```

que a través de *git* almacena los archivos en el servidor. *Heroku* se encarga internamente de la detección del tipo de proyecto y su levantamiento. El despliegue del cliente, una vez éste se ha compilado como se explica en el despliegue en local, se hace con el siguiente comando

```
firebase deploy
```

que se encarga de detectar los archivos generados y desplegarlos.

7.2 *Single Page Application*

Una de las características de la aplicación desarrollada con valor para el usuario final es el hecho de que ésta se adscriba a las directrices de las *Single Page Applications (SPAs)*. Este tipo de aplicaciones web actualizan los elementos mostrados de forma individual, evitando recargar la página por completo a la hora de hacer cualquier cambio a la interfaz [72]. Para ello, el navegador sólo descarga una página HTML, que es actualizada dinámicamente a medida que el usuario interactúa con ella. Esto produce una experiencia de uso más cercana a la ofrecida por las aplicaciones nativas, al no haber recargas de la página para moverse de una vista de la aplicación a otra.

Aún así, se mantienen las opciones de navegación propias de una página web tradicional:

- Los botones para moverse a la vista anterior y posterior en el historial de páginas visitadas siguen siendo funcionales.
- Se puede pegar una URL a una página de la aplicación en concreto en el navegador, y ésta será capaz de interpretarla.

Para ofrecer una mejor experiencia al usuario, las transiciones entre páginas se han complementado con un *throbber*, un elemento gráfico animado que se utiliza para indicar al usua-

rio que se está realizando una tarea en segundo plano y que le proporciona un *feedback* de que se están cargando datos [73].

7.3 *Progressive Web App*

A la hora de acercar al usuario aún más a una experiencia nativa, se ha configurado la aplicación para que sea una *Progressive Web App (PWA)*, un tipo de aplicaciones web que son instalables en cualquier sistema operativo, tanto móvil como de escritorio, y con las que se puede trabajar sin conexión, entre otras ventajas [74]. Combina las facilidades de una aplicación web a la hora de acceder a ella, pudiendo ser probada sin necesidad de instalarla y compartida con sólo un enlace, con la experiencia nativa de poder utilizar la aplicación sin la necesidad de abrir un navegador [74].

Así, la aplicación desarrollada cuenta con las siguientes características de una *PWA*:

- **Instalable** en cualquier dispositivo con un navegador actualizado, permitiendo al usuario acceder a la aplicación sin la necesidad de hacerlo a través de un navegador. Para ello, el usuario tiene que cumplir con una heurística de uso de la aplicación, que en el momento de escribir esta memoria requiere que el usuario haya interactuado con el dominio al menos durante 30 segundos. Al utilizar la versión instalada, se mejora la experiencia de uso, ya que se ocultan elementos del navegador que reducen el tamaño de pantalla disponible para la aplicación, como se puede apreciar en la Figura 7.1. También cuenta con una pantalla de inicio que se muestra cada vez que se inicia la aplicación y que también se puede observar en la Figura 7.1.

- Envío de **notificaciones** al usuario cuando se lleva a cabo una operación en uno de sus grupos o cuando se materializa un movimiento periódico. Para que el usuario pueda recibir estas notificaciones tiene que habilitarlas previamente cuando se le pide permiso, y están disponibles con la aplicación instalada o no, y en dispositivos móviles y de escritorio —un ejemplo de notificación está disponible en la FIGURA NOTIFICACIÓN. Cada dispositivo en el que inicie sesión el usuario se trata independientemente, pudiendo activar notificaciones en todos o sólo en algunos. A la hora de su recepción, se distinguen dos casos:

  - En **escritorio**, se requiere que el navegador (o la aplicación instalada) estén ejecutándose (en primer o segundo plano) para recibir notificaciones. Si están cerrados, o el dispositivo está apagado, las notificaciones llegarán la próxima vez que se abra el navegador/aplicación.
  - En dispositivos **móviles** las notificaciones llegarán con el navegador/aplicación en segundo plano, cerrado o en una pestaña ajena a la aplicación. Si el dispositivo

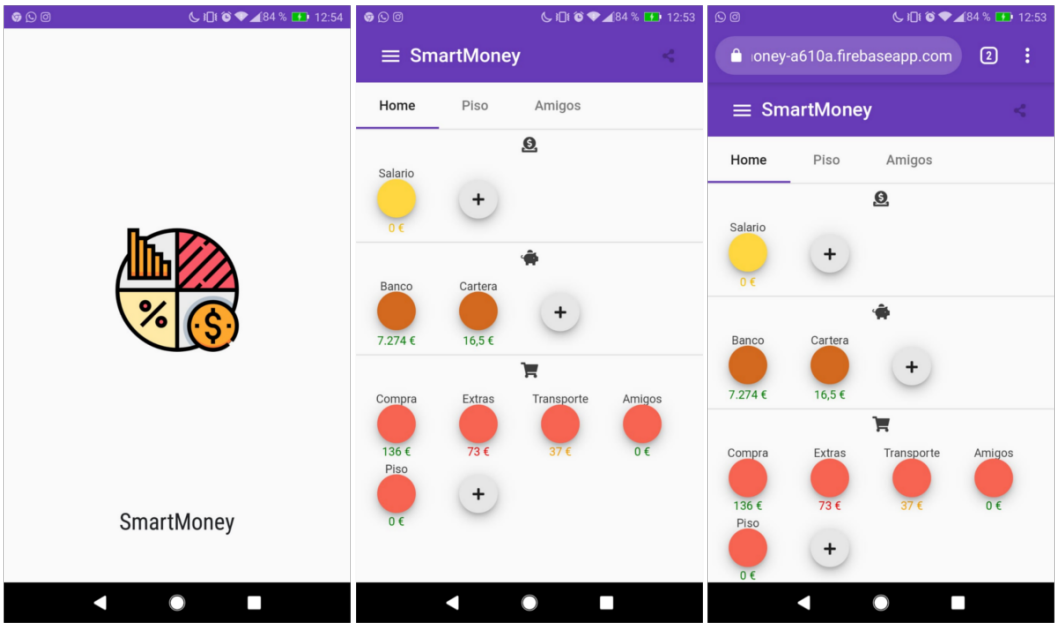

Figura 7.1: Pantalla de inicio, pantalla principal en aplicación descargada, y pantalla principal en navegador

está apagado, se recibirán al encender de nuevo el dispositivo. En primer plano se mostrará un aviso dentro de la propia aplicación.

- Funcionalidad **sin conexión**, que se detalla en la Sección 7.4.

- ***Responsive***. La aplicación se adaptará bien a los diferentes tamaños de pantalla, aunque esté mayormente centrada en los dispositivos móviles. En la Figura 7.2 se pueden observar las diferencias en la pantalla principal de la aplicación de escritorio y de la aplicación móvil. En la versión para pantallas más grandes, el menú lateral se vuelve fijo, mientras que en la versión móvil está accesible a través de un botón en la esquina superior izquierda. También se hace uso del espacio extra para permitir visualizar un mayor número de *slots*.

Todos estos añadidos hacen que, pese a ser una aplicación web, se sienta casi como una aplicación nativa en las diferentes plataformas, tanto de escritorio como móvil, pero que además puede ser utilizada desde cualquier navegador.

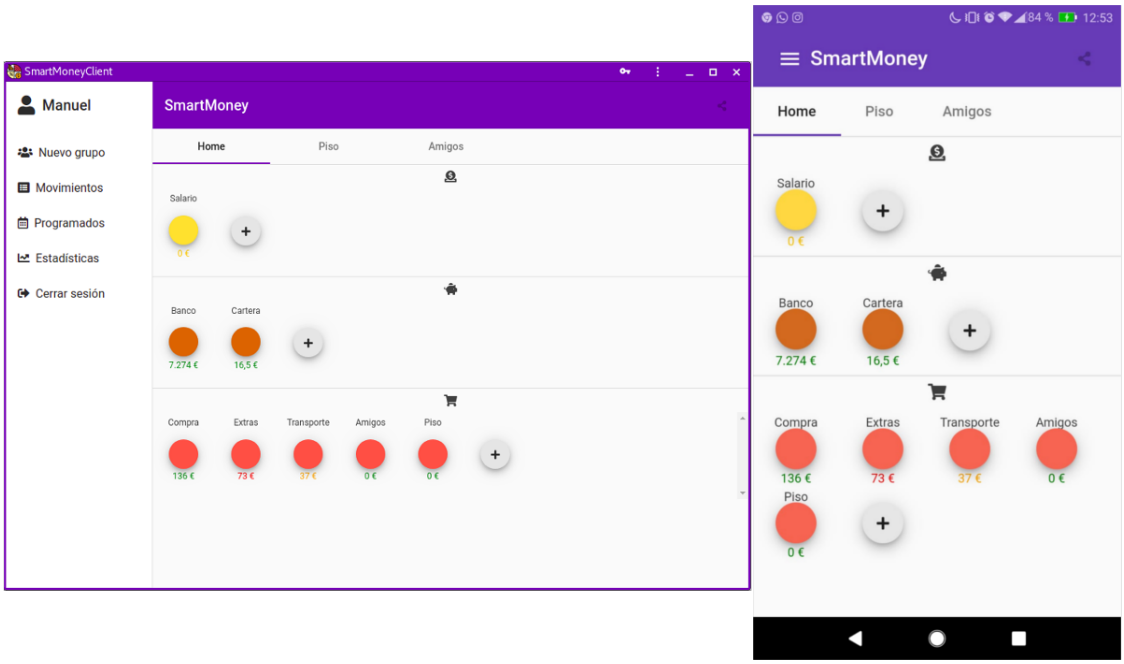

Figura 7.2: Vista principal en las versiones de escritorio y móvil, respectivamente

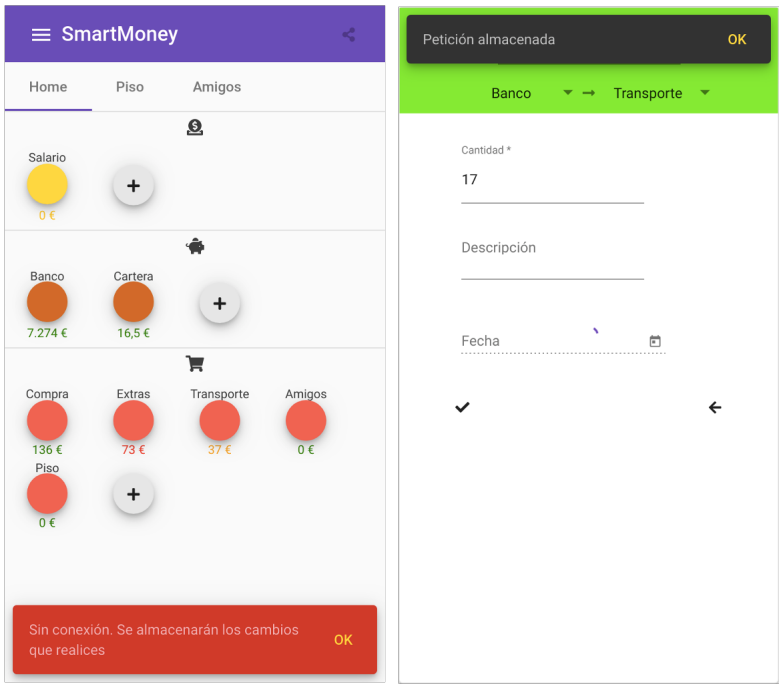

Figura 7.3: Avisos de conexión perdida y operación almacenada

7.4 Funcionamiento *offline*

La aplicación desarrollada puede ser usada sin conexión a Internet. En el momento en que se detecta que no hay conexión, se muestra al usuario un aviso de que desde ese momento, las operaciones realizadas serán almacenadas hasta tener una conexión con el servidor. Este mensaje se muestra en la Figura 7.3. Así, cada vez que un usuario realice un pago, cree un *slot* o modifique algún dato, se le mostrará otro aviso de que la operación ha sido almacenada, el cual se puede observar en la Figura 7.3.

En el momento en que se recupera la conexión con el servidor, se muestra un aviso similar, indicando que la aplicación vuelve a estar *online* y se intenta el envío de las peticiones almacenadas. Cuando todas hayan finalizado, se le notifica al usuario de que se han llevado a cabo dichas operaciones con éxito. En caso de fallar alguna, se le indica en el aviso cuántas peticiones no han podido ser completadas. Tras varios intentos fallidos, se descartan y se avisa al usuario de que han sido eliminadas.

Los datos que se muestren mientras la aplicación esté *offline* no representarán las operaciones realizadas sin conexión hasta que ésta se recupere, por lo que las estadísticas, el listado de movimientos, el detalle de las deudas, etc., se corresponderán con los datos obtenidos por última vez con conexión. Aún así, el usuario podrá interactuar con ellos de la misma manera que estando *online*.

Por motivos de seguridad, las operaciones de inicio de sesión y registro de usuarios no estarán disponibles sin conexión con el servidor. Por lo tanto, para largos periodos de actividad sin conexión a Internet, es recomendable marcar la opción de recordar los datos al iniciar sesión para poder entrar en la aplicación en modo desconectado. Además, la operación de generar un enlace de invitación para un grupo tampoco estará disponible sin conexión, ya que los enlaces tienen una fecha de expiración y en muchos casos no pueden ser reutilizados.

7.5 Tour de la aplicación

En este apartado se ilustrarán las principales características de la aplicación a través de imágenes de ella en funcionamiento, y se compararán con los prototipos comentados en la Sección 4.3.

Para comenzar, la pantalla principal es la que aparece en las ilustraciones de la Figura 7.2. En ella se muestran los diferentes *slots* junto con información acerca de cada uno de ellos: para las fuentes de ingreso se muestra el ingreso realizado por esa fuente en los últimos 30 días; en los monederos, el gasto en ese periodo de tiempo con ese monedero; y en las categorías, el gasto en cada categoría. Además, la información se complementa con colores que indican lo próximo que están las categorías o los monederos a su tope de gasto.

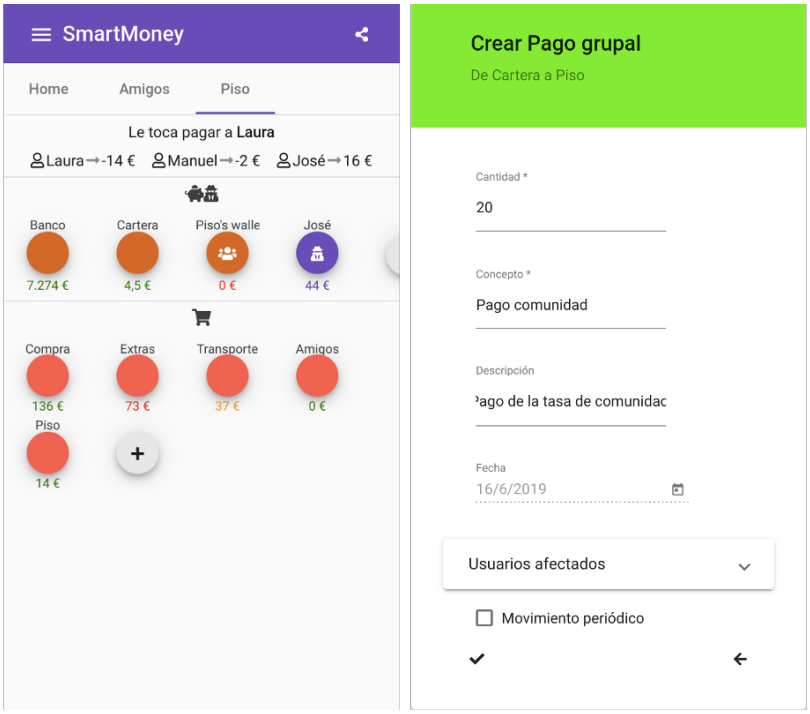

Figura 7.4: Pantalla grupal y de pago grupal

A través de las pestañas de la parte superior se navega entre la vista individual y las de los diferentes grupos a los que pertenece un usuario. La vista grupal se muestra en la Figura 7.4 y como se puede ver, es muy similar a la individual, sustituyéndose las fuentes de ingreso por el estado de cuentas y añadiendo los usuarios no registrados a la zona de los monederos.

Para realizar un movimiento, se arrastrará un *slot* sobre otro (no pudiendo hacer cosas como arrastrar un ingreso sobre una categoría) y se accederá a la segunda pantalla que se muestra en la Figura 7.4, donde se indicarán los datos del movimiento. Además, si se realiza en un grupo, se podrá indicar qué miembros participaron y la cantidad que le afectó a cada uno en el desplegable «Usuarios afectados». Para el caso especial de las transferencias en grupo, se arrastrará un monedero sobre el monedero de grupo, y en la pantalla de creación de movimiento se podrá seleccionar a qué usuario se le hace la transferencia.

Para obtener un enlace de invitación a un grupo se dispone del botón situado en la esquina superior derecha, que sólo estará disponible desde la vista de grupo. Para crear un grupo se hará desde el desplegable de la zona izquierda, donde además se encuentra el acceso a diversas funcionalidades:

- Listado de **movimientos**, individuales o de grupo (dependiendo desde qué pantalla se acceda), mostrado en la Figura 7.5. Contrariamente a lo que se había diseñado en los prototipos, el detalle de un movimiento no va en una vista ajena, sino que se puede des-

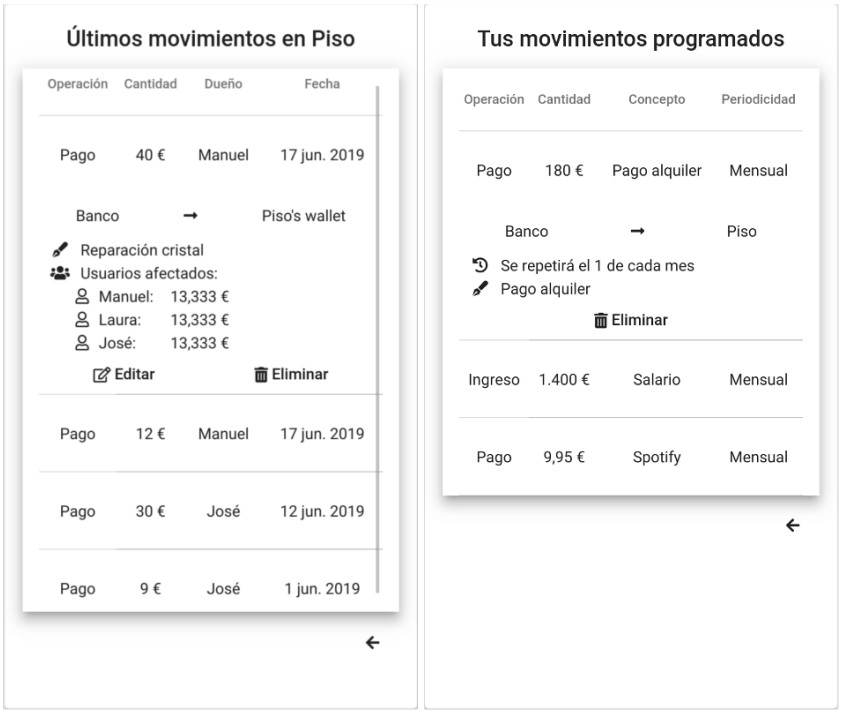

Figura 7.5: Listado y detalle de movimientos grupales y movimientos periódicos individuales, respectivamente

plegar al tocar sobre un movimiento, donde además se permitirá acceder a la edición del mismo o eliminarlo. Los movimientos están paginados pero de una forma muy cómoda para el usuario, que le permite cargar más movimientos al ir haciendo *scroll* sobre el listado.

- Listado de **movimientos periódicos**, individuales o grupales, disponible en la Figura 7.5 y con un funcionamiento muy similar al listado de movimientos.
- Consulta de **estadísticas**, individuales o grupales, a lo largo de un rango de fechas. En la 7.6 se puede ver el histórico de datos grupal y el desglose de gasto por categoría, pudiendo acceder además al histórico de datos individual, el desglose de gasto por monedero (individual) y por miembro de cada grupo.
- Cálculo de **deudas** en un grupo que proporciona información sobre cómo llevar a cabo el saldo de las deudas y dando la posibilidad de saldar todas de golpe, como se ve en la Figura 7.7.
- **Ajustes de grupo**, en los que se puede modificar el nombre del mismo, la categoría o monedero por defecto, y se pueden eliminar miembros del grupo. Todo esto se puede ver en la Figura 7.7.

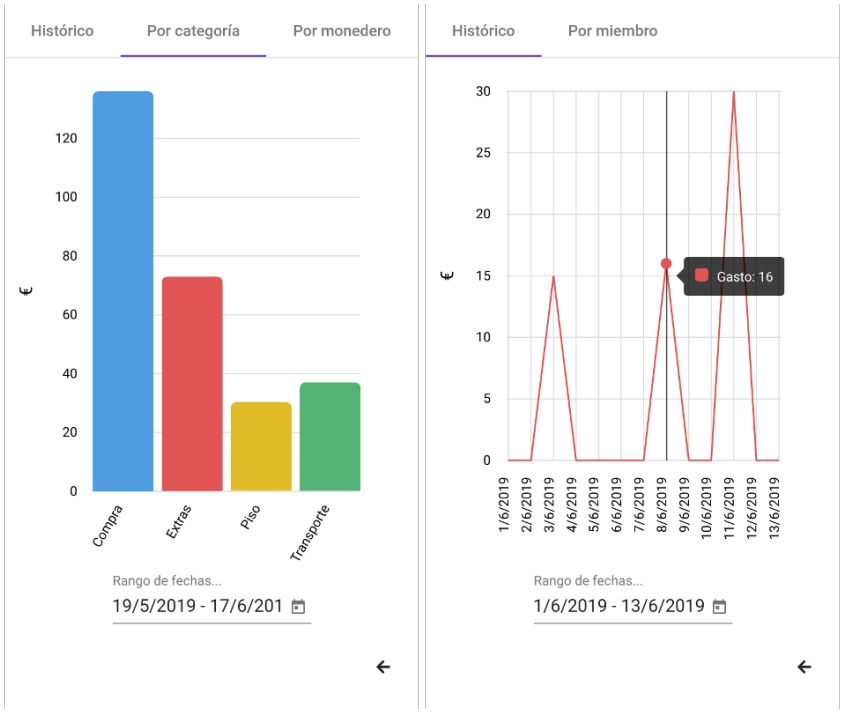

Figura 7.6: Estadísticas individuales de gasto agrupadas por categoría e histórico de gasto en un grupo

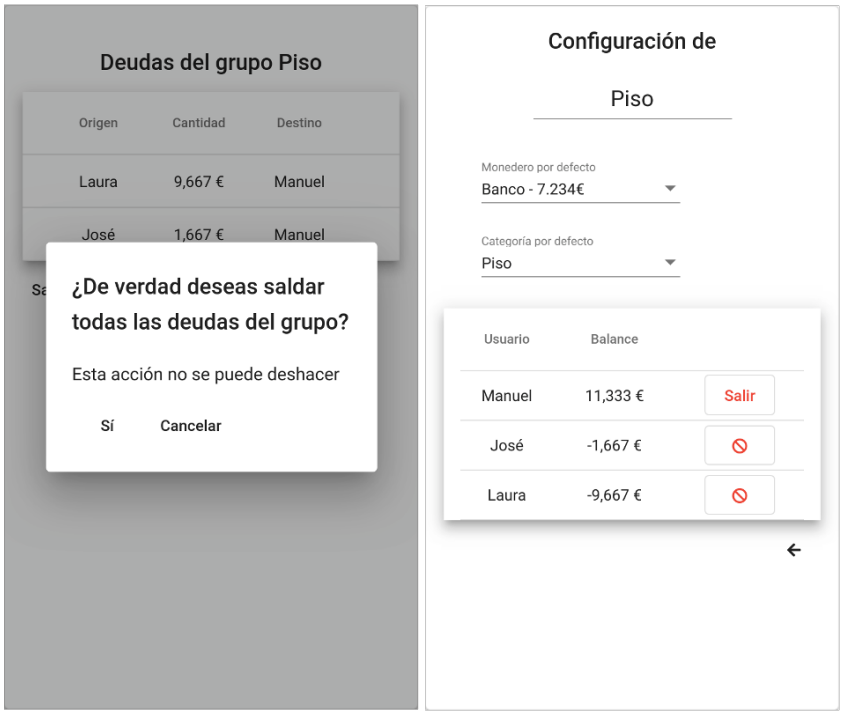

Figura 7.7: Pantalla de saldo de deudas y de ajustes de grupo

Capítulo 8

Conclusiones y trabajo futuro

El resultado producido al término de este proyecto, en líneas generales, cumple con holgura los objetivos que se planteaban al principio de este proyecto:

- Se ha desarrollado una aplicación web que por sí sola permite la gestión de las finanzas personales de un usuario y sus cuentas conjuntas de forma interrelacionada, obteniendo datos interesantes a medida que hace uso de ella.
- La aplicación hace uso de los beneficios de ser una *Progressive Web App* y los lleva un paso más allá, permitiendo al usuario disfrutar de una experiencia *offline* sin límites.
- Se ha producido un software robusto que permite realizar cálculos complejos, como determinar las deudas dentro de un grupo, o la obtención de diversas estadísticas relacionadas con los movimientos del usuario.
- La solución desarrollada ofrece una experiencia agradable, intuitiva y fácil de usar que aporta gran valor para el usuario final.

Para llevar a cabo todo este trabajo ha sido muy útil poder contar con una serie de conocimientos proporcionados a lo largo de toda la titulación sin los que habría sido imposible afrontar un proyecto de este calibre, entre los que se pueden destacar:

- Conocimientos **técnicos** que han permitido montar un proyecto desde cero y han servido de base para aprender nuevas tecnologías con el fin de desarrollar funcionalidades más avanzadas.
- Conocimientos sobre **metodologías de desarrollo** que se han podido aplicar para planificar las distintas etapas del proyecto y organizarlo, posibilitando que saliese adelante.
- Conocimientos de **técnicas de diseño de software** que han ayudado a generar un producto mantenible y de calidad.

- Conocimientos de **herramientas** que han facilitado llevar a cabo todas las tareas de un proyecto, desde su planificación hasta las pruebas.
- Probablemente lo más importante, la **capacidad de enfrentarse a problemas** nuevos y resolverlos utilizando los conocimientos obtenidos y la experiencia adquirida durante toda la carrera.

Además, se han aprendido una serie de lecciones de gran valor:

- Se ha profundizado mucho en las **tecnologías del lado cliente**, trabajando con un *framework* (Angular) muy actual y usado por un gran número de empresas punteras, y utilizando conceptos avanzados del área como *interceptors* o *service workers*, lo que ha supuesto el descubrimiento de nuevos límites en el área del desarrollo web.
- Se ha tenido más en cuenta el **cuidado de la estética**, aplicando nociones del *material design* para producir unas interfaces agradables e intuitivas de usar, y se ha mejorado en las **técnicas de desarrollo *frontend*** al afrontar un proyecto que buscaba tener una interfaz muy pulida y con la que se interactuase a través de gestos.
- Se han **mejorado los conocimientos ya existentes** de tecnologías *backend* a través de un uso más avanzado de las mismas: tareas periódicas, un mayor cuidado de la seguridad al trabajar con datos delicados, etc.
- Se ha comprobado de primera mano la importancia de las **pruebas** y de que sean hechas a tiempo. En un proyecto con tanto código, es muy importante poder detectar de forma rápida y fácil dónde reside un error, y las pruebas son de gran ayuda en estas situaciones.
- Se ha aprendido a **desplegar** un proyecto en un entorno real, haciéndolo accesible a través de Internet, y no sólo en local.

A pesar de todo esto, el proyecto desarrollado se podría mejorar en varios aspectos:

- **Importación y exportación de datos**, a través de un estándar (por ejemplo, CSV) para permitir a un usuario introducir en la aplicación de forma fácil datos que ha registrado previamente en otra aplicación similar, o importar automáticamente movimientos realizados en una de sus cuentas bancarias.
- **Internacionalización** de los textos para que la aplicación pueda ser usada en cualquier región del mundo.
- Relacionado con ello, trabajo con múltiples **divisas**, ofreciendo la posibilidad de realizar pagos en países extranjeros y que el usuario no se tenga que preocupar de conversiones manuales o esperar para introducir los pagos.

Apéndices

Apéndice A

Configuración del *service worker* proporcionado por Angular

```
{
  "index": "/index.html",
  "dataGroups": [
  {
    "name": "api-freshness",
    "urls": [
    "/**"
    ],
    "cacheConfig": {
      "strategy": "freshness",
      "maxSize": 100,
      "maxAge": "3d",
      "timeout": "5s"
    }
  }
  ],
  "assetGroups": [
  {
    "name": "app",
    "installMode": "prefetch",
    "resources": {
      "files": [
      "/favicon.ico",
      "/index.html",
      "/*.css",
      "/*.js"
      ]
    }
  }, {
    "name": "assets",
    "installMode": "lazy",
    "updateMode": "prefetch",
    "resources": {
```

```
      "files": [
      "/assets/**",
      "/*.(eot|svg|cur|jpg|png|webp|gif|otf|ttf|woff|woff2|ani)"
      ]
    }
  }
  ]
}
```

Apéndice B

Escenario planteado para la comprobación del modelo

A continuación se describirá el escenario planteado como base para comprobar que el modelo diseñado sería capaz de dar solución a los problemas relacionados con cuentas conjuntas:

1. Un usuario se crea una cuenta para compartir gastos con 3 amigos en una noche de sábado - usuario, amigo1, amigo2, amigo3.
2. Crea un monedero llamado ”cartera” para representar el dinero efectivo en ese momento.
3. Crea una fuente de ingreso llamada ”papa”.
4. Ingresa 40€ desde la fuente de ingreso ”papa” al monedero ”cartera”. Concepto ”sábado”.
5. Crea un nuevo grupo llamado ”noche”.
6. Invita a amigo1, amigo3.
7. Amigo2 no tiene cuenta, por lo que crea un usuario no registrado para él.
8. Amigo1 paga una ronda de cañas, igual precio para todos, 6€, para la categoría del grupo.
9. Amigo1 paga otra ronda y dos raciones a compartir entre todos. 22€.
10. Amigo2 paga una ración y otra ronda. 15€.
11. Amigo3 se tiene que ir, por lo que se calculan las deudas en el momento.
12. Se efectúan los pagos correspondientes. Las cuentas quedan a 0.
13. Usuario paga 14€, que sólo le afectan a él y a amigo1.

14. Amigo2 paga 20€, que afectan a todos por igual.
15. Amigo2 paga 15€ que sólo afectan a usuario y a él mismo.
16. Amigo2 paga 10€ que afectan a todos por igual.
17. Se acaba la noche y al día siguiente usuario decide calcular las deudas que quedan, y se paga lo que le corresponde a cada uno.

Y a continuación se detallan las operaciones llevadas a cabo por debajo en cada momento, materializadas en las Figuras B.1, B.2 y B.3. Cabe destacar que los estados de las cuentas se han separado de la tabla Miembro para poder representar cómo han ido avanzando los balances individuales en momentos importantes.

1. Nuevo fila en usuario con nombre, contraseña, e-mail y gasto máximo mensual.
2. Nuevo monedero, nombre = cartera, FK usuario 1, saldo 0.
3. Nueva fuente de ingreso, nombre papá, FK usuario 1.
4. Nuevo movimiento, importe 40, concepto sábado, FK origen 2, FK destino 1. Actualización del saldo de monedero ”cartera” para reflejar este cambio.
5. Nuevo grupo, nombre noche.
    - Implica nueva fila en MIEMBRO entre el grupo ”noche” y el usuario ”usuario”.
    - Implica monedero asociado al grupo -> se establece el monedero Cartera del usuario como monedero por defecto para el grupo.
    - Se crea categoría de grupo para el usuario añadido al grupo, y se establece por defecto para movimientos ajenos en el grupo.
6. Se unen amigo1, amigo3 al grupo.
    - Dos nuevas filas en miembro con los datos de estos usuarios (que suponemos que ya estaban registrados anteriormente).
    - Se crea categoría de grupo para cada usuario añadido al grupo.
7. Se crea usuario no registrado en grupo. Nueva fila en MIEMBRO entre el usuario no registrado y el grupo.

    // Movimientos grupales. Para cada movimiento grupal, se llevan a cabo dos tipos de movimientos.

(a) Movimiento del monedero del usuario que realiza el pago al monedero grupal. Importe = total del pago. - Se comprueba el dueño del monedero origen, se añade a su balance en el grupo el total pagado y se le resta del saldo del monedero utilizado.

(b) Un movimiento, por cada afectado, del monedero grupal a la categoría individual especificada para el grupo (por defecto, la categoría asociada al grupo). - Importe por defecto = total del pago / afectados. Se puede especificar la cantidad que afecta a cada miembro. - Para cada uno, se le resta de su balance en el grupo el importe que le afecta, y se añadirá al gasto de la categoría seleccionada.

Nota: Los movimientos con destino monedero grupal suman el importe en el balance del usuario, y los movimientos con origen monedero grupal restan el importe en el balance del usuario. // End Movimientos grupales

8. Se crean los movimientos adecuados, 6€ a dividir por igual entre todos.

9. Se crean los movimientos adecuados, 22€ a dividir por igual entre todos.

10. Se crean los movimientos adecuados, 15€ a dividir por igual entre todos.

11. Cálculo de deudas. Se obtiene el balance en este momento, y se calculan los siguientes movimientos:

    - 10,75 de usuario a amigo1
    - 6,5 de amigo2 a amigo1
    - 4,25 de amigo2 a amigo no registrado

    // Algoritmo de cálculo de deudas básico.

    - Ordenar balances de forma descendente.
    - El usuario con balance más positivo recibe dinero del usuario con balance más negativo. - Si deuda(menor) > acree(mayor) -> mayor balance a 0 y se le ignora en siguientes iteraciones. - Si acree(mayor) > deuda(menor) -> menor balance a 0 y se le ignora en siguientes iteraciones. - Si acree(mayor) = deuda(menor) -> ambos a 0 y se les ignora en siguientes iteraciones.
    - Repetir paso 2 hasta que el balance de todos sea 0.

    // End Algoritmo de cálculo de deudas básico.

    // Transferencias grupales. Por cada transferencia, se realizan dos movimientos.

- Movimiento de monedero seleccionado por el usuario de origen (o monedero por defecto para grupos) a monedero grupal. Importe = total transferencia. - Actualización del saldo del monedero origen y destino. Se suma el importe al balance en el grupo del usuario origen.
- Movimiento de monedero de grupo a monedero por defecto de usuario destino. Importe = total transferencia. - Actualización de los saldos de los monederos. Se resta el importe al balance en el grupo del usuario destino.

// End Transferencias grupales.

12. Se efectúan los movimientos calculados arriba. Todos los balances del grupo quedan a 0.
13. Se crean los movimientos adecuados. 14€ a dividir entre usuario y amigo1.
14. Se crean los movimientos adecuados. 20 € a dividir entre usuario, amigo1 y amigo2.
15. Se crean los movimientos adecuados. 15€ a dividir entre usuario y amigo2.
16. Se crean los movimientos adecuados. 10€ a dividir entre usuario, amigo1 y amigo2.
17. Cálculo de deudas. Se obtiene el balance en este momento, y se calculan los siguientes movimientos:

    - 17€ de amigo1 a amigo2.
    - 10,5 de usuario a amigo2.

    Se efectúan estos movimientos, y el balance en el grupo de todos queda a 0.

MOMENTO

USUARIO

	ID	NOMBRE	PASSWORD	E-MAIL	MAX_GASTO
	1	amigo1	aflj4afl$		
	2	amigo3	aflj4afl$		
1	3	usuario	aflj4afl$		300,00 €

MONEDERO

	ID	NOMBRE	DESCRIPCIÓ	SALDO	USUARIO_ID	GRUPO_ID
2	1	Cartera		4,75	3	
5	4	Monedero_noche		0		1
	8	Mndro_amg1		-27,75	1	
	9	Mndro_amg3		-10,75	2	

F_INGRESO

	ID	NOMBRE	DESCRIPCIÓ	USUARIO_ID
3	2	Papá		3

CATEGORÍA

	ID	NOMBRE	DESCR	MAX_GASTO	BALANCE	USUARIO_ID
6	5	Cat_noche			27,75	1
6	6	Cat_noche			10,75	2
5	7	Cat_noche			35,25	3

AMIGO_N_R

	ID	NOMBRE	DESCRP	USUARIO_ID	GRUPO_ID
7	3	amigo2			1

GRUPO

	ID	NOMBRE
5	1	Noche

Figura B.1: Tablas de usuario, *slots* y grupo

	MOVIMIENTO						
MOMENTO	ID	IMPORTE	CONCEPTO	MVN_PADRE	ORIGEN_ID	DEST_ID	GROUP_ID
4	1	40	Sábado		2	1	
8.1	2	6	Cañas		8	4	1
8.2	3	1,5	Cañas	2	4	5	1
8.2	4	1,5	Cañas	2	4	6	1
8.2	5	1,5	Cañas	2	4	7	1
8.2	6	1,5	Cañas	2	4	3	1
9.1	7	22	Raciones		8	4	1
9.2	8	5,5	Raciones	6	4	5	1
9.2	9	5,5	Raciones	6	4	6	1
9.2	10	5,5	Raciones	6	4	7	1
9.2	11	5,5	Raciones	6	4	3	1
10.1	12	15	Tercera ronda		3	4	1
10.2	13	3,75	Tercera ronda	10	4	5	1
10.2	14	3,75	Tercera ronda	10	4	6	1
10.2	15	3,75	Tercera ronda	10	4	7	1
10.2	16	3,75	Tercera ronda	10	4	3	1
12	17	10,75	Saldo deudas		1	4	1
12	18	10,75	Saldo deudas	17	4	8	1
12	19	6,5	Saldo deudas		9	4	1
12	20	6,5	Saldo deudas	19	4	8	1
12	21	4,25	Saldo deudas		9	4	1
12	22	4,25	Saldo deudas	21	4	3	1
13.1	23	14			1	4	1
13.2	24	7		23	4	7	1
13.2	25	7		23	4	5	1
14.1	26	20			3	4	1
14.2	27	6,6666666667		26	4	5	1
14.2	28	6,6666666667		26	4	7	1
14.2	29	6,6666666667		26	4	3	1
15.1	30	15			3	4	1
15.2	31	7,5		30	4	7	1
15.2	32	7,5		30	4	3	1
16.1	33	10			3	4	1
16.2	34	3,3333333333		33	4	5	1
16.2	35	3,3333333333		33	4	7	1
16.2	36	3,3333333333		33	4	3	1
17	37	17			8	4	1
17	38	17		37	4	3	1
17	39	10,5			1	4	1
17	40	10,5		39	4	3	1

Figura B.2: Tabla de movimientos

BALANCE(momento 11)

GRUPO_ID	USUARIO_ID	ANR_ID	BALANCE
1	1		17,25
1		3	4,25
1	2		-10,75
1	3		-10,75

↓

BALANCE(momento 12)

GRUPO_ID	USUARIO_ID	ANR_ID	BALANCE
1	1		0
1		3	0
1	2		0
1	3		0

↓

BALANCE(momento 16)

GRUPO_ID	USUARIO_ID	ANR_ID	BALANCE
1	1		-17
1		3	27,5
1	2		0
1	3		-10,5

↓

BALANCE(momento 17)

GRUPO_ID	USUARIO_ID	ANR_ID	BALANCE
1	1		0
1		3	0
1	2		0
1	3		0

MIEMBRO

MOMENTO	USUARIO_ID	GRUPO_ID	ANR_ID	CATEG_DEFECTO_ID	
5	3	1		7	
6	1	1		5	
6	2	1		6	
7		1	3		

Figura B.3: Estados de cuentas en diversos momentos interesantes

Apéndice C

Contenido del CD

El CD incluido contiene los siguientes ficheros:

- **Memoria del trabajo en formato PDF**: PerezRodriguez_Pablo_TFG_2019.pdf.
- **Resumen del trabajo en formato PDF**: PerezRodriguez_Pablo_TFG_2019_resumo.pdf.
- **Código fuente de la aplicación**: PerezRodriguez_Pablo_TFG_2019_anexo.zip. En él se incluyen de forma separada el proyecto cliente (*smart-money-client*) y el proyecto servidor (*smart-money-rest*).

Apéndice D

Glosario de acrónimos

API Application Programming Interface

CRUD Create-Read-Update-Delete

CSRF Cross-site Request Forgery

CSS Cascading Style Sheets

CLI Command Line Interface

DAO Data Access Object

HQL Hibernate Query Language

HTML HyperText Markup Language

HTTP Hypertext Transfer Protocol

HTTPS HyperText Transport Protocol Secure

IDE Integrated Development Environment

Java EE Java Enterprise Edition

JSON JavaScript Object Notation

JWT JSON Web Token

MVC Model-View-Controller

NPM Node Package Manager

ORM Object-Relational Mapping

PWA Progressive Web App

REST Representational State Transfer

SPA Single Page Application

URI Uniform Resource Identifier

URL Uniform Resource Locator

XSS Cross-site Scripting

Apéndice E

Glosario de términos

Backend Parte de un sistema informático que no es accedida directamente por el usuario y típicamente se encarga de almacenar y manipular los datos del sistema.

Bug error o problema de un programa informático.

Feedback Información u opiniones acerca de algo, que son usadas como base para mejorar.

Framework Entorno software reusable que proporciona una funcionalidad particular para facilitar el desarrollo de una aplicación informática.

Frontend Parte de un sistema informático con la que el usuario interactúa de forma directa.

Issue En informática, unidad de trabajo para completar una mejora en un sistema.

Offline Dicho de un dispositivo, que no está conectado a una red, normalmente Internet.

Online Dicho de un dispositivo, que está conectado a una red, normalmente Internet.

Responsive Que reacciona bien y positivamente.

Throbber Animación utilizada para indicar al usuario que un programa está trabajando en una tarea.

Bibliografía

[1] "Coinkeeper alternatives and similar apps and websites," Jun 2019. [Online]. Available: https://alternativeto.net/software/coinkeeper/

[2] "Sharing expenses with friends?" [Online]. Available: https://settleup.io/

[3] "Angular architecture." [Online]. Available: https://angular.io/guide/architecture

[4] "Get started with json web tokens." [Online]. Available: https://auth0.com/learn/json-web-tokens/

[5] "Coinkeeper: Control de gastos - apps en google play." [Online]. Available: https://play.google.com/store/apps/details?id=com.disrapp.coinkeeper.material&hl=es_419

[6] "Split expenses with friends." [Online]. Available: https://www.splitwise.com/

[7] "Spring." [Online]. Available: https://spring.io/

[8] "Hibernate orm." [Online]. Available: https://hibernate.org/orm/

[9] "Project lombok." [Online]. Available: https://projectlombok.org/

[10] "Angular," 2019. [Online]. Available: https://angular.io/guide/architecture

[11] "Design- material design." [Online]. Available: https://material.io/design/

[12] "Angular material." [Online]. Available: https://material.angular.io/

[13] "Rxjs - introduction." [Online]. Available: https://rxjs-dev.firebaseapp.com/guide/overview

[14] Microsoft, "Typescript," May 2019. [Online]. Available: https://github.com/microsoft/TypeScript

[15] "Indexeddb api." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

[16] "Dexie.js." [Online]. Available: https://dexie.org/

[17] "Push api." [Online]. Available: https://www.w3.org/TR/push-api/

[18] "Set up a javascript firebase cloud messaging client app | firebase." [Online]. Available: https://firebase.google.com/docs/cloud-messaging/js/client

[19] C. Larman, *Agile and iterative development: a managers guide.* Addison-Wesley, 2004.

[20] J. A. Highsmith, *Agile software development ecosystems.* Addison-Wesley, 2006.

[21] K. Schwaber and J. Sutherland, "The scrum guide™," 2017. [Online]. Available: https://www.scrumguides.org/scrum-guide.html

[22] "Gitlab agile delivery." [Online]. Available: https://about.gitlab.com/solutions/agile-delivery/

[23] Atlassian, "What is a kanban board?" [Online]. Available: https://www.atlassian.com/agile/kanban/boards

[24] "Git flow workflow." [Online]. Available: https://leanpub.com/git-flow/read

[25] B. Porter, J. v. Zyl, and O. Lamy, "Welcome to apache maven." [Online]. Available: https://maven.apache.org/

[26] Junit-Team, "Junit." [Online]. Available: https://github.com/junit-team/junit4/wiki/Getting-started

[27] "Jacoco java code coverage library," May 2019. [Online]. Available: https://www.eclemma.org/jacoco/

[28] "npm." [Online]. Available: https://www.npmjs.com/

[29] "Pencil project." [Online]. Available: https://pencil.evolus.vn/

[30] "Draw.io - free diagrams online." [Online]. Available: https://www.draw.io/

[31] M. Inc, "Magicdraw." [Online]. Available: https://www.nomagic.com/products/magicdraw

[32] "The heroku product suite." [Online]. Available: https://www.heroku.com/products

[33] "Firebase hosting." [Online]. Available: https://firebase.google.com/products/hosting/

[34] M. Richards, "Software architecture patterns." [Online]. Available: https://www.oreilly.com/library/view/software-architecture-patterns/9781491971437/ch01.html

[35] D. Alur, J. Crupi, and D. Malks, *Core J2EE patterns best practices and design strategies.* Prentice Hall PTR, 2003.

[36] T. Erl, *SOA with REST: principles, patterns and constraints for building enterprise solutions with REST.* Prentice Hall, 2013.

[37] V. Xia, "What is mobile first design? why it's important and how to make it?" Dec 2017. [Online]. Available: https://medium.com/@Vincentxia77/what-is-mobile-first-design-why-its-important-how-to-make-it-7d3cf2e29d00

[38] "Global mobile phone internet user penetration 2019 | statistic." [Online]. Available: https://www.statista.com/statistics/284202/mobile-phone-internet-user-penetration-worldwide/

[39] Oracle, "Dao pattern." [Online]. Available: https://www.oracle.com/technetwork/java/dataaccessobject-138824.html

[40] "Building a restful web service." [Online]. Available: https://spring.io/guides/gs/rest-service/

[41] D. Anikin and D. Anikin, "What an in-memory database is and how it persists data efficiently," Oct 2016. [Online]. Available: https://medium.com/@denisanikin/what-an-in-memory-database-is-and-how-it-persists-data-efficiently-f43868cff4c1

[42] "Asp.net mvc pattern | .net." [Online]. Available: https://dotnet.microsoft.com/apps/aspnet/mvc

[43] M. Karén and M. Karén, "Interceptors in angular," Mar 2019. [Online]. Available: https://blog.angularindepth.com/top-10-ways-to-use-interceptors-in-angular-db450f8a62d6

[44] "Service worker api." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

[45] "Service worker config." [Online]. Available: https://angular.io/guide/service-worker-config

[46] Baeldung, "Hibernate inheritance mapping," Oct 2018. [Online]. Available: https://www.baeldung.com/hibernate-inheritance

[47] T. Janssen, "Complete guide: Inheritance strategies with jpa and hibernate," Apr 2019. [Online]. Available: https://thoughts-on-java.org/complete-guide-inheritance-strategies-jpa-hibernate/

[48] J. Frank and E. Gostischa-Franta, "Interface pattern." [Online]. Available: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/interface.html

[49] auth0.com, "Json web tokens introduction." [Online]. Available: https://jwt.io/introduction/

[50] OWASP, "Sql injection." [Online]. Available: https://www.owasp.org/index.php/SQL_Injection

[51] ——, "Cross-site scripting (xss)." [Online]. Available: https://www.owasp.org/index.php/Cross-site_Scripting_(XSS)

[52] Angular, "Angular security." [Online]. Available: https://angular.io/guide/security

[53] OWASP, "Cross-site request forgery (csrf)." [Online]. Available: https://www.owasp.org/index.php/Cross-Site_Request_Forgery_(CSRF)

[54] Spring, "Spring csrf." [Online]. Available: https://docs.spring.io/spring-security/site/docs/5.0.x/reference/html/csrf.html

[55] Angular, "Angular csrf." [Online]. Available: https://angular.io/guide/http#security-xsrf-protection

[56] D. Vávra, "Mobile application for group expenses and its deployment," Ph.D. dissertation, Settle Up, 2012. [Online]. Available: http://www.settleup.info/files/master-thesis-david-vavra.pdf

[57] "Window: offline event." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/offline_event

[58] "Window: online event." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/online_event

[59] M. Data, "Lost update databases," Feb 2015. [Online]. Available: https://www.morpheusdata.com/blog/2015-02-21-lost-update-db

[60] "Transaction definition," Dec 2011. [Online]. Available: https://docs.spring.io/spring/docs/3.0.x/javadoc-api/org/springframework/transaction/TransactionDefinition.html#ISOLATION_SERIALIZABLE

[61] "Notifications api." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Notifications_API

[62] "How push works | web fundamentals | google developers." [Online]. Available: https://developers.google.com/web/fundamentals/push-notifications/how-push-works

[63] "How push works | web fundamentals | google developers." [Online]. Available: https://developers.google.com/web/fundamentals/push-notifications/how-push-works

[64] E. Paraschiv, "The @scheduled annotation in spring," Dec 2018. [Online]. Available: https://www.baeldung.com/spring-scheduled-tasks

[65] "Cron expressions." [Online]. Available: https://docs.oracle.com/cd/E12058_01/doc/doc.1014/e12030/cron_expressions.htm

[66] "Spring security." [Online]. Available: https://spring.io/projects/spring-security

[67] "Window.localstorage." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Window/localStorage

[68] "Functional testing," Jul 2018. [Online]. Available: http://softwaretestingfundamentals.com/functional-testing/

[69] Archiveddocs, "Stubs and mocks." [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff798400(v=pandp.10)

[70] "Integration testing," Mar 2018. [Online]. Available: http://softwaretestingfundamentals.com/integration-testing/

[71] "Sonarlintin eclipse." [Online]. Available: https://www.sonarlint.org/eclipse/

[72] M. Wasson, Nov 2013. [Online]. Available: https://msdn.microsoft.com/en-us/magazine/dn463786.aspx

[73] "What is a throbber? - definition from techopedia." [Online]. Available: https://www.techopedia.com/definition/23814/throbber

[74] "Progressive web apps." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/Progressive_web_apps