Facultad
de
Ciencias

DESARROLLO DE UNA APLICACIÓN MÓVIL PARA LA ADMINISTRACIÓN DE GASTOS

(Development of a mobile application for managing expenses)

Trabajo de Fin de Grado
para acceder al

GRADO EN INGENIERÍA INFORMÁTICA

Autor: Adrián Becerril Crespo

Director: Patricia López Martínez

Septiembre - 2018

Resumen

En los últimos tiempos la tecnología ha evolucionado considerablemente hasta tal punto que ha conseguido instaurarse en todos los ámbitos de la sociedad provocando una revolución en la forma de vida de las personas.

Uno de los máximos exponentes de este fenómeno es el sector de los teléfonos móviles ya que, con la aparición de los Smartphones, ahora nos es posible hacer uso desde cualquier parte del mundo de una serie de servicios prácticamente idénticos a los que un ordenador tradicional es capaz de ofrecer. Debido a todos los beneficios que esto conlleva, el uso de los Smartphones se ha extendido rápidamente en la sociedad y actualmente la mayoría de las personas lleva encima al menos uno en todo momento.

Este desarrollo tecnológico ha permitido también iniciar un proceso de sustitución de los medios de pago tradicionales como el dinero en efectivo por otros nuevos como los cargos a la factura del móvil, PayPal, las tarjetas virtuales, los pagos móviles... Si a este amplio abanico de posibilidades para pagar le añadimos que en nuestra vida diaria estamos realizando continuamente actividades económicas, obtenemos que administrar nuestro dinero resulta una tarea compleja de llevar a cabo.

Por tanto, este proyecto tiene como principal objetivo desarrollar una aplicación móvil que permita a un usuario llevar un seguimiento de todas sus transacciones económicas de forma precisa y que además permita gestionar los gastos compartidos de grupos de personas.

La aplicación móvil ha sido desarrollada para la plataforma Android. Así pues, se ha realizado una aplicación completamente nativa capaz de ser ejecutada óptimamente en el sistema operativo Android. Paralelamente, se ha diseñado e implementado un servicio REST que permita dar soporte a la gestión de los gastos compartidos por un grupo de personas.

Palabras clave: aplicación móvil, Android, transacciones económicas, servicios REST.

Abstract

In recent times, technology has evolved considerably to such an extent that it has managed to establish itself in all areas of society causing a revolution in the way of life of people.

One of the greatest examples of this phenomenon is the mobile phones sector, since with the appearance of smartphones, it is now possible for us to use a series of services virtually identical to those of a traditional computer from anywhere in the world. able to offer. Due to all the benefits that this entails, the use of smartphones has spread rapidly in society and currently most people carry over at least one at all times.

This technological development has also allowed us to start a process of replacing traditional means of payment such as cash with new ones such as charges to the mobile bill, PayPal, virtual cards, mobile payments ... If this broad range of possibilities to pay we add that in our daily life we are continuously doing economic activities, we get that managing our money is a complex task to carry out.

Therefore, this project has as its main objective to develop a mobile application that allows a user to track all their economic transactions in a precise manner and to manage the shared expenses of groups of people.

The mobile application has been developed for the Android platform. Thus, a completely native application has been made capable of being optimally executed in the Android operating system. In parallel, a REST service has been designed and implemented to support the management of shared expenses by a group of people.

Keywords: mobile app, Android, economic transactions, REST services.

Índice de contenidos

Índice de figuras

Índice de tablas

1. Introducción

Este capítulo se compone de tres secciones. En la primera sección se analizarán las principales motivaciones para llevar a cabo este proyecto, así como el contexto tecnológico en el que se desarrolla. En la segunda sección se tratarán los principales objetivos que se persiguen con la realización de este trabajo. Finalmente, en la última sección, se mostrará la estructura que seguirá el presente documento.

1.1 Motivación y contexto tecnológico

A lo largo de la historia, se han producido gran cantidad de avances tecnológicos que han contribuido a mejorar la calidad de vida de las personas. Con el paso del tiempo, la tecnología ha comenzado a evolucionar cada vez más y con mayor rapidez. En los últimos años, el progreso tecnológico ha sido de tal calibre que la tecnología ha conseguido consolidarse en todos los ámbitos de la sociedad provocando una revolución en la vida cotidiana de las personas. Actualmente podemos afirmar que la tecnología es una pieza fundamental en nuestras vidas.

Uno de los máximos exponentes de este fenómeno y que ha logrado situarse como uno de los iconos tecnológicos del siglo XXI es el teléfono móvil. El teléfono móvil comenzó siendo un dispositivo cuya única funcionalidad era permitir la comunicación a distancia entre usuarios y al que solamente podían acceder las personas con alto poder adquisitivo. Esta situación es muy diferente en nuestros días ya que, con la aparición de los teléfonos inteligentes o smartphones, podemos realizar una serie de operaciones que se equiparan a los de un ordenador tradicional. Así pues, con un smartphone, nos es posible realizar no solo las funciones básicas que ofrece un teléfono móvil tradicional sino también otras funciones más avanzadas como fotografiar, filmar, ver películas, escuchar música, la navegación web… En consecuencia, estos dispositivos se han extendido rápidamente de manera que se podría decir que hoy en día todo el mundo lleva al menos uno en su bolsillo.

Considerando los factores anteriormente expuestos se podría decir que los teléfonos móviles han actuado en los últimos años como depredadores digitales ya que han conseguido sustituir en mayor o menor medida a libros, periódicos, reproductores de música, radios, televisiones, cámaras, relojes, juegos de mesa, ordenadores o calculadoras. Teniendo en cuenta todo esto, podemos deducir que seguramente acabarán sustituyendo también a una cartera ya que, probablemente, los pagos móviles acaben siendo utilizados de forma habitual para efectuar todo tipo de compras, tanto online como en tiendas físicas. Sin embargo, la realidad es que hasta ahora los pagos móviles no han conseguidos relevar a los medios de pago más tradicionales y han tenido que conformarse con ser una alternativa más.

En la actualidad, las alternativas de pago se han diversificado gracias a los avances tecnológicos. En un mercado todavía marcado por el uso mayoritario de métodos de pago más tradicionales [1] como el dinero en efectivo, las tarjetas de débito y crédito y las

transferencias, han comenzado a tener presencia otras opciones de pago nuevas como cargos a la factura del móvil, PayPal, las tarjetas virtuales, los pagos móviles… Generalmente, una persona no utiliza el mismo método de pago para todas sus compras, sino que se decanta por un método u otro en función del gasto que quiera efectuar. Este aspecto junto con que en nuestra vida diaria estamos constantemente realizando transacciones económicas, conlleva a una situación en la que llevar un seguimiento de todos nuestros ingresos/gastos resulta una tarea compleja de llevar a cabo. Tener un control de todas nuestras transacciones económicas resulta fundamental para conseguir administrar nuestro dinero de la forma más eficiente.

1.2 Objetivos

Dentro del contexto expuesto anteriormente, la aplicación que se presenta en este documento, denominada FinanWallet, tiene como propósito general facilitar al usuario la planificación de su presupuesto a través de una interfaz de usuario sencilla e intuitiva que permita consultar y modificar su estado financiero rápidamente y en cualquier lugar. Más concretamente, la aplicación pretende dar soporte a la gestión personal de los gastos y a la gestión de deudas entre un grupo de personas. A continuación, en la Figura 1.1, se muestra el logotipo utilizado por la aplicación móvil.

Figura 1.1 Logotipo de la aplicación.

En cuanto a la gestión personal de los gastos, se pretende dar soporte a la administración de las transacciones económicas que un individuo realiza en el transcurso de su vida cotidiana. Estas transacciones pueden ser el pago de un billete de bus, el pago de una entrada de cine, el ingreso del sueldo mensual… Las principales funcionalidades que se requieren para esta parte de la aplicación son las siguientes:

- Administrar todo tipo de transacciones económicas (ingresos/gastos).
- Administrar las categorías para clasificar una transacción económica.
- Consultar las transacciones económicas con la posibilidad de aplicar filtros de búsqueda: todas, por fecha o por categoría.
- Consultar el balance de ingresos y gastos resultante entre dos fechas concretas.
- Exportar el historial de transacciones económicas a un archivo en formato CSV.

Respecto a la gestión de grupo, el objetivo principal es facilitar la administración de los gastos compartidos entre grupos de personas. Un viaje en grupo, una cena, comprar un regalo entre todos para uno de los amigos o comprar productos para la casa entre un grupo de

compañeros de piso son situaciones en las que poner orden en las cuentas resulta una tarea compleja de llevar a cabo. Puede ocurrir que alguien no recuerde exactamente cuánto costó una comida, que un deudor se olvide de su pago, que se pierda el control de lo que alguien debe o no debe… Con los teléfonos móviles es más fácil gestionar este tipo de situaciones ya que, prácticamente, todo el mundo lleva al menos uno en su bolsillo y el hecho de registrar un gasto común, una transferencia entre integrantes del grupo o directamente acceder al balance económico de cada integrante se convierte en una labor sencilla y rápida. Así pues, las principales funcionalidades que se solicitan en esta parte son las siguientes:

- Administrar grupos y sus integrantes.
- Administrar los gastos que se produzcan en el grupo de manera que se autocalculen las deudas para cada uno de los integrantes del grupo.
- Administrar las transferencias que se produzcan entre integrantes del grupo.
- Consultar el estado de cada integrante del grupo y los movimientos que se hayan producido con la posibilidad de aplicar filtros de búsqueda: todas o por fecha.

1.3 Organización de la memoria

Este documento consta de diversas secciones que, en su conjunto, tendrán como principal objetivo describir el proceso de desarrollo de una aplicación móvil nativa para el sistema operativo Android. A lo largo del documento se mostrarán los requisitos demandados por futuros posibles usuarios, las decisiones tomadas en cuanto al diseño de la aplicación y los aspectos básicos relativos a su implementación.

En definitiva, los contenidos de este documento se han distribuido de la siguiente forma:

- En primer lugar, en el capítulo 2 se tratará el enfoque metodológico adoptado para el desarrollo de la aplicación, así como las tecnologías y herramientas que se han utilizado.
- A continuación, en el capítulo 3 se recogen en un principio los requisitos funcionales y no funcionales definidos para la aplicación. Después se mostrarán los diferentes casos de uso y se detallarán aquellos que por un motivo u otro se consideren de mayor complejidad. Para acabar, se especificarán los requisitos de la interfaz de usuario.
- Después, el capítulo 4 se centrará en describir el diseño de la aplicación desarrollada. Más concretamente, se expondrá el diseño arquitectónico y el diseño detallado del sistema.
- Seguidamente, en el capítulo 5 se explicará el proceso de implementación de la aplicación y se entrará en mayor detalle en aquellas partes que resulten más complejas.
- Posteriormente, en el capítulo 6 se describirá el proceso de pruebas realizado sobre la aplicación tras su implementación.
- Por último, en el capítulo 7 se tratarán las conclusiones que se han obtenido tras la realización de la aplicación y además se comentarán las posibles mejoras recomendables a realizar en un futuro.

2. Tecnologías y herramientas

Este capítulo consta de tres secciones. En la primera sección se detallará la metodología de desarrollo de software que se ha adoptado para realizar la aplicación móvil. Posteriormente, en la segunda sección y en la última se describirán respectivamente las tecnologías y herramientas que se han utilizado durante el desarrollo de este proyecto.

2.1 Metodología de desarrollo de software

Para desarrollar un producto software, una de las principales decisiones que debemos tomar es escoger una metodología de desarrollo que sea óptima para conseguir los resultados deseados. Una metodología de desarrollo de software describe las diferentes etapas que debe atravesar el proceso de desarrollo de un producto software, desde su fase inicial hasta su fase final, para conseguir realizar un producto de calidad a un coste razonable.

En este caso, se ha optado por una metodología iterativa e incremental, derivada del ciclo de vida en cascada. Esta metodología se basa en dividir el proceso de desarrollo del producto software en la iteración de varios ciclos de vida en cascada, en los que, incrementalmente, se desarrollará la funcionalidad de la aplicación. De esta forma, al final de cada iteración, se consigue una versión nueva del producto más estable y con nuevas funcionalidades respecto a versiones anteriores. Las iteraciones se repetirán hasta obtener el producto software deseado. A continuación, se muestra un diagrama donde se representan las diferentes fases de la metodología iterativa e incremental.

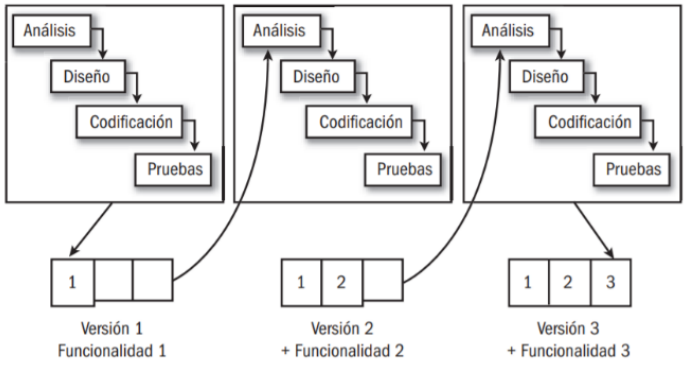

Figura 2.1. Diagrama del ciclo de vida iterativo e incremental [2].

Esta metodología resulta adecuada para la realización de pequeños proyectos como es este caso, donde la aplicación en cuestión deberá ser realizada por una sola persona con lo que el proceso de desarrollo tiene que ser asequible. Con este modelo de desarrollo, nos es posible aplicar mejoras en las fases ya desarrolladas en función de las necesidades que surjan durante el desarrollo de la aplicación y a la experiencia adquirida en cada iteración. Al mismo tiempo, al separar la complejidad del producto en diversas iteraciones, se obtiene como resultado un producto software con mayor calidad, siempre y cuando se aplique en cada fase un proceso de pruebas y gestión de la calidad adecuado.

2.2 Tecnologías

2.2.1 Android

Android [3] es un sistema operativo, basado en el núcleo de Linux, diseñado principalmente para dispositivos móviles. Es un sistema libre, gratuito y multiplataforma que con el paso del tiempo ha conseguido situarse como el sistema operativo más utilizado en el mercado de los dispositivos móviles. Android posee una arquitectura de 4 capas o niveles, en la que cada nivel proporciona un conjunto de servicios a las capas superiores. Los componentes que conforman el sistema operativo Android, desde el nivel más alto hasta el más bajo, son las siguientes:

- Applications (Aplicaciones): en esta capa se encuentran todas las aplicaciones del dispositivo, tanto las que vienen integradas en el dispositivo Android como las aplicaciones de terceros que instala el usuario. Todas estas aplicaciones hacen uso de la capa inferior (Application Framework) para acceder a los servicios del dispositivo.
- Application Framework (Marco de trabajo de aplicaciones): esta capa proporciona los servicios necesarios para que las aplicaciones puedan realizar sus funciones correctamente. Estos servicios son proporcionados a las aplicaciones en forma de clases Java.
- Android Runtime (Entorno de ejecución Android): se trata de un conjunto de librerías que proporcionan la mayor parte de las funcionalidades disponibles en las librerías habituales del lenguaje Java. No se considera una capa como tal a pesar de apoyarse en las bibliotecas pertenecientes a la capa que se expondrá a continuación. Esto se debe a que está formado también por bibliotecas.
- Libraries (Librerías): esta capa contiene el conjunto de librerías que suministran las principales funcionalidades del sistema operativo Android. Todas estas bibliotecas están desarrolladas en C y C++ y compiladas en el hardware específico del dispositivo. Su objetivo es proporcionar a las capas superiores una serie de funcionalidades para tareas que se repiten con frecuencia.
- Linux Kernel (Núcleo de Linux): esta capa es la más básica y, por tanto, la que se encuentra en el primer nivel de la arquitectura. Se corresponde con el núcleo del sistema operativo Android que actúa como interfaz entre el hardware del dispositivo y el resto de la pila software. Adicionalmente, se encarga de realizar funciones principales del sistema tales como la gestión de procesos, la gestión de la memoria, la gestión de la seguridad…

2.2.2 REST

REST (Representational State Transfer) [4] es un estilo de arquitectura software que define un conjunto de restricciones para el desarrollo de servicios web respetando el estándar HTTP. Un servicio web basado en REST se caracteriza por publicar recursos en vez de un conjunto arbitrario de métodos u operaciones. REST introduce el concepto de recurso para hacer referencia a una entidad que representa un concepto de negocio que puede ser accedido públicamente. Cada recurso posee un identificador único y global que lo distingue de cualquier

otro, así como un estado interno al que no se puede acceder directamente desde el exterior. Así pues, para acceder al estado interno de cada recurso, el cliente se comunica con el servidor a través del protocolo de comunicación estándar HTTP e intercambian representaciones del estado de los recursos. Se entiende por representación del estado de un recurso al formato de datos concreto utilizado, normalmente XML o JSON, para transferir una copia del estado público de un recurso entre el cliente y el servidor. Como se ha mencionado anteriormente, un servicio REST carece de operaciones arbitrarias sobre los recursos, por lo que, para operar con los recursos del servicio, todos y cada uno de estos poseen una interfaz única y constante con las mismas operaciones. Un servicio REST puede definir cuatro operaciones para que el cliente pueda operar con los recursos:

- POST/PUT: operación para crear un nuevo recurso en el servidor. La diferencia entre usar POST o PUT es que el primero se utiliza cuando el servidor está a cargo de decidir la URI del nuevo recurso mientras que el último se usa cuando es el cliente quien tiene dicha responsabilidad. Es conveniente destacar que para crear un nuevo recurso es más común utilizar POST que PUT.
- PUT: operación que permite sobrescribir el estado del recurso del servidor en base a la copia o representación del estado del recurso que tiene el cliente.
- DELETE: operación para eliminar un recurso del servidor.
- GET: operación para leer una representación del estado de un recurso del servidor.

No obstante, cabe destacar que no es fundamental que el conjunto completo de los recursos del servicio tenga que responder a todos y cada uno de los métodos expuestos, es decir, la implementación del servicio es libre para decidir prohibir alguna de estas operaciones para un recurso concreto. Otro de los aspectos por los que se caracteriza REST es que las peticiones entre cliente y servidor resultan independientes entre sí, cada mensaje HTTP debe contener toda la información necesaria para poder comprender la petición de manera que ni el cliente ni el servidor tengan que guardar el estado de las comunicaciones entre mensajes. En la actualidad, REST se está convirtiendo en la opción más utilizada para llevar a cabo la implementación de servicios web.

2.2.3 SQLite

SQLite [5] es una librería software que proporciona un motor de base de datos SQL transaccional de código abierto. Es una herramienta muy popular en la actualidad gracias a su capacidad para almacenar información de una manera sencilla, rápida, eficaz y potente en dispositivos con limitadas capacidades hardware, como puede ser un teléfono móvil. No requiere de una configuración compleja, el tamaño de la biblioteca es pequeño y permite una fácil integración con cualquier aplicación. Además, a diferencia de otros sistemas gestores de base de datos, SQLite no precisa de un proceso servidor para funcionar, por lo que lee y escribe los datos directamente en archivos en memoria, permitiendo que una base de datos SQL completa con múltiples tablas, índices, disparadores y vistas esté contenida en un solo archivo. SQLite forma parte de las librerías presentes en el núcleo de Android, por lo que se puede hacer uso de esta herramienta mediante el uso de interfaces dentro del paquete

`android.database.sqlite` de Android facilitando así el proceso de persistencia en las aplicaciones.

2.2.4 MySQL

MySQL [6] es un sistema gestor de base de datos relacional, multihilo, multiusuario y de código abierto. Se caracteriza por su gran adaptación a los diferentes entornos de desarrollo y su integración con gran variedad de sistemas operativos. Así mismo, MySQL es capaz de manejar bases de datos de gran tamaño con un alto rendimiento. Destaca por su alta velocidad en la búsqueda de datos e información y por el bajo costo en requerimientos para la elaboración de bases de datos, esto último es debido a su bajo consumo lo que hace que pueda ser ejecutado en una máquina con recursos limitados. En la actualidad, MySQL es muy utilizado en aplicaciones web, en plataformas de desarrollo web y en herramientas de seguimiento de errores.

2.3 Herramientas

2.3.1 Android Studio

Android Studio [7] es el entorno de desarrollo integrado oficial para el desarrollo de aplicaciones Android. Esta herramienta, basada en el software IntelliJ IDEA, permite desarrollar código para la plataforma Android de una manera rápida y eficiente ya que ofrece renderizado en tiempo real, refactorización específica de Android, una estructura simple y organizada para los proyectos que se pretenden desarrollar, plantillas para elaborar diseños comunes de Android y otros componentes, una consola de desarrollador que muestra consejos de optimización y estadísticas de uso, la posibilidad de simular la ejecución del código a través de un emulador de un dispositivo Android o directamente desde el móvil, un editor de diseño intuitivo que facilita el desarrollo de la interfaz de usuario y herramientas Lint para detectar problemas relacionados con el rendimiento, usabilidad y compatibilidad de versiones. Android Studio se encuentra disponible para los principales sistemas operativos: Windows, GNU/Linux y MacOS.

2.3.2 Eclipse

Eclipse [8] es un entorno de desarrollo integrado, multiplataforma y de código abierto basado en Java. Por diseño, la plataforma no proporciona una gran variedad de funcionalidades por sí misma. Sin embargo, la potencia de este IDE reside en la posibilidad de utilizar plugins que permiten añadir funcionalidades a la plataforma haciendo que Eclipse tenga un campo de acción considerablemente amplio. En consecuencia, Eclipse sirve de IDE para casi cualquier lenguaje, como puede ser Java, C++, PHP, Python o Perl, entre otros. Actualmente, Eclipse es uno de los entornos más conocidos y utilizados, especialmente para el desarrollo software en lenguaje Java.

2.3.3 StarUML

StarUML [9] es un software para el modelado de sistemas software basado en los estándares UML y DMA. Esta herramienta da soporte a las actividades de análisis y diseño de software para sistemas basados en la programación orientada a objetos. StarUML está orientada al ámbito profesional contribuyendo a maximizar la productividad e incrementar la calidad de los proyectos software.

2.3.4 MySQL Workbench

MySQL Workbench [10] es una herramienta visual para el diseño y documentación de bases de datos relacionales para el motor de base de datos MySQL. Esta herramienta se caracteriza por elaborar una representación visual de las tablas, vistas, procedimientos almacenados y claves foráneas de la base de datos. MySQL Workbench está disponible para los principales sistemas operativos existentes: Windows, Linux y MacOS X.

2.3.5 Postman

Postman [11] es una herramienta que permite interactuar con una API y testear su funcionamiento. Esta herramienta permite realizar peticiones HTTP a una API sin necesidad de escribir código y, en consecuencia, un gran ahorro de tiempo en todas las etapas del desarrollo de un servicio. Postman es una aplicación simple pero considerablemente potente que se ha convertido en una herramienta habitual en el desarrollo web.

2.3.6 GlassFish

GlassFish [12] es un servidor de aplicaciones de software libre que implementa las tecnologías definidas en la plataforma Java EE. En otras palabras, este servidor de aplicaciones es capaz de soportar tecnologías como Java JavaServer Pages (JSP), JavaServer Faces (JSF), Java Servlet, Enterprise JavaBeans (EJB), Java API for XML Web Services (JAX-WS), Java Architecture for XML Binding (JAXB) o Java Persistence API (JPA). Glassfish se encuentra entre los servidores más utilizados en la actualidad y es considerado la implementación de referencia de Java EE.

2.3.7 NinjaMock

NinjaMock [13] es una herramienta online gratuita para la creación de mockups y wireframes con un acabado de dibujo a mano alzada. Esta herramienta resulta muy eficiente cuando se busca generar unas ideas básicas, un diseño rápido y un primer boceto de una aplicación.

3. Análisis y especificación de requisitos

En este apartado se describe el conjunto completo de requisitos que se tendrán en cuenta para el desarrollo de la aplicación.

Para llevar a cabo la captura de requisitos, se ha tenido en cuenta, por un lado, la experiencia propia, especialmente en el ámbito de los gastos compartidos, para la definición de los requisitos básicos de la aplicación y, por otro lado, las múltiples reuniones mantenidas con futuros posibles usuarios interesados en la aplicación para complementar así a los requisitos anteriormente mencionados. De esta forma, se pretende poder satisfacer el mayor número de necesidades y diseñar así un número de escenarios de ejecución óptimo.

3.1 Especificación de requisitos funcionales

Los requisitos funcionales representan las características que describen la interacción entre la aplicación y su entorno (el usuario o cualquier otro sistema externo que interactúe con ésta) independientemente de la implementación realizada. Es decir, permiten describir todas las actividades que la aplicación debe ser capaz de llevar a cabo.

A continuación, se muestra una tabla con todos los requisitos funcionales capturados, especificando para cada uno de ellos un identificador único y una descripción:

Tabla 3.1. Requisitos funcionales.

Identificador	Descripción
RF1	El usuario debe poder gestionar sus propias transacciones económicas.
RF1.1	El usuario debe poder añadir/eliminar una transacción económica (ingreso/gasto) para la cual deberá especificar un título, la cantidad, la categoría a la que pertenece, una descripción, una fecha y su periodicidad.
RF1.2	El usuario debe poder añadir/eliminar una categoría de manera que permita clasificar una transacción económica para así tener la información estructurada. Para ello, se deberá especificar un título identificador y una descripción.
RF1.3	El usuario debe poder consultar sus transacciones económicas filtradas de diversas formas: todas, por título, por fecha o por categoría.
RF1.4	El usuario debe poder exportar su historial de transacciones económicas a un archivo con formato CSV.
RF1.5	El usuario debe poder añadir/eliminar recordatorios especificando un título y una fecha.
RF2	El usuario debe poder gestionar los gastos compartidos de un grupo de personas.
RF2.1	El usuario debe poder crear/eliminar un grupo de gestión de gastos especificando el nombre del grupo en cuestión.
RF2.2	Cualquier usuario puede unirse a un grupo siempre y cuando conozca el

	identificador y el nombre del grupo.
RF2.3	Cualquier usuario dentro de un grupo debe poder añadir un nuevo gasto especificando un título, la cantidad gastada, la persona que ha realizado el desembolso, las personas involucradas y una fecha.
RF2.4	Cuando un usuario añade un nuevo gasto, las deudas de las personas integradas se autocalcularán a partes iguales por defecto, a menos que se especifique otra división del gasto.
RF2.5	Cualquier usuario dentro de un grupo debe poder añadir una nueva transferencia especificando un título, la cantidad traspasada, la persona que realiza la transferencia, la persona que recibe la transferencia y una fecha.
RF2.6	Cualquier usuario dentro de un grupo debe poder consultar los balances de los integrantes del grupo y las deudas pendientes filtrando de diversas formas: todas o por nombres de usuario.
RF2.7	Cualquier usuario dentro de un grupo debe poder consultar las transacciones económicas grupales filtradas por: todos, por tipo de transacción, por fecha o por título del gasto.

3.2 Especificación de requisitos no funcionales

Los requisitos no funcionales representan aquellas características no referentes de manera estricta a la funcionalidad de la aplicación, pero también necesarias a tener en cuenta en el momento de diseñar e implementar la aplicación. Es decir, son todas aquellas restricciones en el diseño o en la implementación que permiten hacer que el producto adquiera cierto grado de calidad de manera que resulte atractivo, rápido, usable…

A continuación, se muestra una tabla con todos los requisitos no funcionales capturados, especificando para cada uno de ellos un identificador único, una descripción y la categoría del requisito:

Tabla 3.2. Requisitos no funcionales.

Identificación	Descripción	Categoría
RNF1	La aplicación debe ser soportada por el sistema operativo Android con la versión 4.2 o superior.	Portabilidad
RNF2	La aplicación debe soportar inicialmente un sólo idioma (Español - España) y utilizar como formato de moneda el Euro (€). En posteriores versiones, soportará más idiomas y más formatos de monedas.	Usabilidad
RNF3	La aplicación debe ser en todo momento intuitiva de manera que pueda ser utilizada cómoda y sencillamente.	Usabilidad
RNF4	La aplicación debe responder a cualquier petición del usuario en un periodo de tiempo razonable.	Rendimiento
RNF5	La aplicación debe ser desarrollada adoptando un diseño y una arquitectura que permita una fácil escalabilidad y mantenibilidad.	Soporte

3.3 Especificación de casos de uso

El diseño de casos de uso es una técnica que permite describir de manera clara y concisa la comunicación y el comportamiento que tendrá una aplicación cuándo ésta interactúe con los diferentes usuarios o sistemas externos que demanden una funcionalidad. Normalmente, debido a que se proporciona información muy a alto nivel, los diagramas de casos de uso son utilizados para obtener una idea general sobre qué puede hacer la aplicación y a quién se le da acceso a unas u otras funcionalidades.

A continuación, se muestran los diferentes diagramas de casos de uso que se han elaborado para especificar la aplicación móvil que se desea desarrollar. En la Figura 3.1 podemos observar un primer diagrama de casos de uso global, donde podemos observar las funcionalidades de alto nivel que ofrecerá la aplicación.

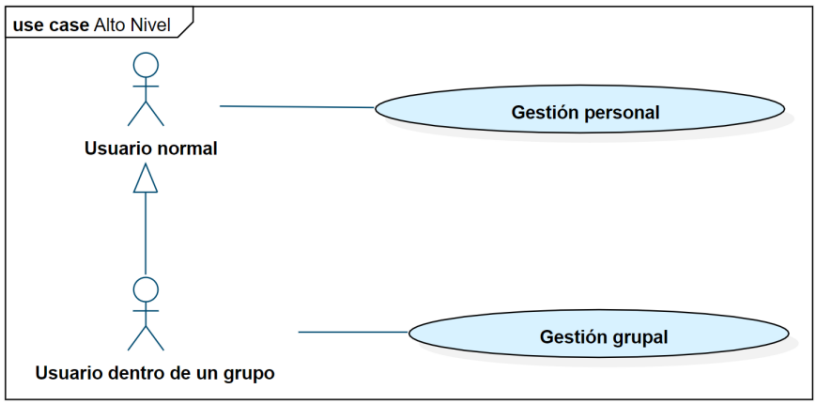

Figura 3.1. Diagrama de casos de uso de alto nivel

Una vez realizado el diagrama global, debemos entrar a detallar cada uno de los casos de uso de alto nivel. Para llevar a cabo esta actividad, tendremos que diseñar nuevos diagramas de casos de uso que permitan describir las funcionalidades de la aplicación a un nivel de detalle más bajo. Como resultado, obtenemos los diagramas de casos de uso que se muestran en las figuras 3.2 y 3.3.

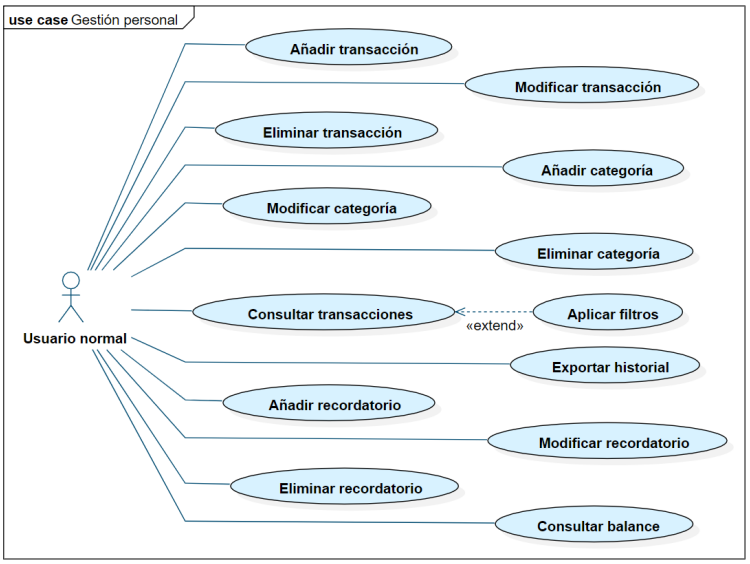

Figura 3.2. Diagrama de casos de uso correspondiente a la gestión personal

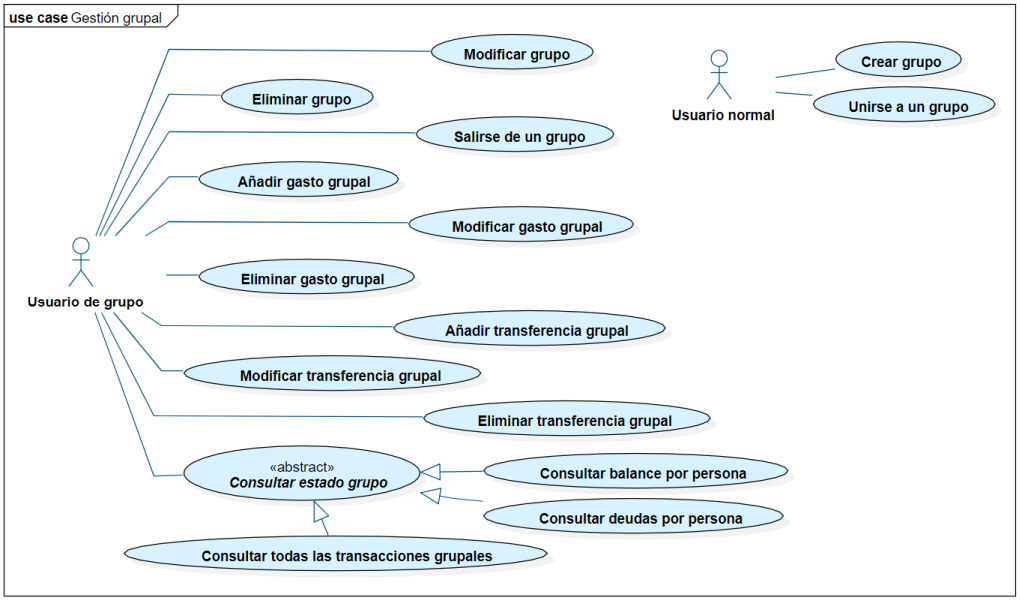

Figura 3.3. Diagrama de casos de uso correspondiente a la gestión grupal

3.4 Especificación detallada de los casos de uso

Una vez realizados los diagramas de casos de uso, es preciso aumentar el nivel de detalle y llevar a cabo, para cada caso de uso, su correspondiente especificación detallada. Una especificación detallada de un caso de uso es una descripción del modo en el que los actores interactúan con el sistema, es decir, permite definir iterativamente el comportamiento esperado en cada momento. Esto hace que se consideren no solo las situaciones habituales de ejecución del caso de uso, sino también las situaciones alternativas y/o de error. La especificación puede ser realizada de manera textual o bien usando diagramas UML como pueden ser los diagramas de secuencia, los diagramas de estado o los diagramas de actividades. En este apartado nos centraremos en la especificación detallada de casos de uso de manera textual.

Para realizar la especificación detallada de los diferentes casos de uso de manera textual, utilizaremos una plantilla que nos permita estructurar y detallar el caso de uso en cuestión siguiendo unos patrones y obteniendo por tanto el mayor nivel de detalle posible. Al no existir ninguna propuesta definida en UML, la elección de la plantilla mencionada anteriormente es arbitraria. En este caso, se ha escogido una plantilla basada en las propuestas de Arlow & Neustad en [14] y de Alistair Cockburnen en [15]. Así pues, el modelo de plantilla utilizado será el siguiente:

Tabla 3.3. Plantilla para la especificación de casos de uso.

	Significado
Identificador	Identificador del caso de uso.
Nombre	Nombre del caso de uso.
Descripción	Párrafo con el objetivo del caso de uso.
Actores	Actores involucrados en el caso de uso.

Precondiciones	Restricciones que ha de cumplir el estado del sistema para que el caso de uso pueda ejecutarse.
Flujo principal	Pasos del escenario de ejecución normal del caso de uso, el escenario de éxito.
Postcondiciones	Estado del sistema al finalizar el caso de uso.
Flujos alternativos	Pasos de los escenarios alternativos de ejecución, que suelen corresponder a escenarios de error o eventuales.

A continuación, se realizará la especificación detallada de los casos de uso que se consideran de mayor complejidad y que por tanto necesitan de una mayor documentación.

Tabla 3.4. Especificación de caso de uso: Consultar transacciones

Identificador	CU1
Nombre	Consultar transacciones
Descripción	El sistema muestra todas las transacciones que ha realizado el usuario.
Actores	Usuario normal
Precondiciones	
Flujo principal	1. El usuario selecciona la opción “Transacciones” de la aplicación. 2. El sistema muestra todas las transacciones que ha realizado el usuario, tanto los ingresos como los gastos. 3. Extension point: Aplicar filtros.
Postcondiciones	El usuario visualizará todas las transacciones.
Flujos alternativos	En cualquier momento el usuario puede abandonar la aplicación.

Tabla 3.5. Especificación de caso de uso: Aplicar filtros.

Identificador	CU2
Nombre	Aplicar filtros
Descripción	El usuario filtra el historial de transacciones a partir de un título, una categoría o una fecha.
Actores	Usuario normal
Precondiciones	El usuario debe encontrarse en la sección de “Transacciones”
Flujo principal	1. El usuario selecciona el icono de búsqueda e introduce un título, una categoría o una fecha. 2. El sistema buscará en el historial las transacciones que tengan una correlación parcial o completa a los parámetros introducidos. 3. El sistema muestra el listado de todas las transacciones encontradas.
Postcondiciones	El usuario visualizará el historial de transacciones filtrado en función de los parámetros introducidos.
Flujos alternativos	2.a. No existe ninguna transacción en el historial para los parámetros introducidos. a1. El sistema notifica al usuario del suceso. En cualquier momento el usuario puede abandonar la aplicación.

Tabla 3.6. Especificación de caso de uso: Añadir transacción.

Identificador	CU3
Nombre	Añadir transacción
Descripción	El usuario añade una nueva transacción económica (ingreso o gasto) al historial.
Actores	Usuario normal
Precondiciones	El usuario debe encontrarse en la sección de "Transacciones".
Flujo principal	1. El usuario selecciona la opción de añadir una nueva transacción económica. 2. El sistema muestra un formulario y solicita al usuario que lo rellene con los datos de la nueva transacción que se pretende añadir. 3. El usuario rellena el formulario con los datos requeridos y pulsa el botón de "Guardar". 4. El sistema registra la transacción económica.
Postcondiciones	La transacción económica ha sido registrada en el sistema.
Flujos alternativos	3.a. El usuario no ha completado todos los campos obligatorios del formulario. a1. El sistema notifica al usuario del suceso. En cualquier momento el usuario puede abandonar la aplicación.

Tabla 3.7. Especificación de caso de uso: Crear grupo.

Identificador	CU4
Nombre	Crear grupo
Descripción	El usuario crea un grupo para poder gestionar las deudas que surjan entre sus integrantes.
Actores	Usuario normal
Precondiciones	
Flujo principal	1. El usuario selecciona la opción "Crear grupo" del menú lateral de la aplicación. 2. El sistema solicitará al usuario un nombre de grupo. 3. El usuario introducirá el nombre del grupo y pulsará el botón de "Crear grupo". 4. El sistema registrará el nuevo grupo y generará un código que se asociará al grupo de manera que pueda ser identificado.
Postcondiciones	El grupo ha sido registrado en el sistema al igual que el usuario como integrante del grupo.
Flujos alternativos	En cualquier momento el usuario puede abandonar la aplicación.

Tabla 3.8. Especificación de caso de uso: Unirse a un grupo.

Identificador	CU5
Nombre	Unirse a un grupo
Descripción	El usuario se une a un grupo.
Actores	Usuario normal

Precondiciones	El grupo debe haber sido creado previamente.
Flujo principal	1. El usuario selecciona la opción “Unirse a un grupo”. 2. El sistema solicitará al usuario que introduzca los datos relativos al nombre e identificador del grupo al que pretende unirse. 3. El usuario introduce los datos y pulsa el botón de “Unirse”. 4. El sistema registra el nuevo integrante para el grupo en cuestión.
Postcondiciones	El usuario ha sido registrado en el sistema como integrante del grupo.
Flujos alternativos	3.a. No existe ningún grupo con el par de valores introducidos. a1. El sistema notifica al usuario del suceso. En cualquier momento el usuario puede abandonar la aplicación.

Tabla 3.9. Especificación de caso de uso: Añadir gasto grupal

Identificador	CU6
Nombre	Añadir gasto grupal
Descripción	El usuario añade un nuevo gasto al grupo.
Actores	Usuario de grupo
Precondiciones	El usuario ha seleccionado un grupo.
Flujo principal	1. El usuario selecciona la opción “Añadir gasto”. 2. El sistema muestra un formulario y solicita al usuario que lo rellene con los datos del nuevo gasto de grupo que se pretende añadir. 3. El usuario introduce los datos solicitados. 4. El sistema calculará automáticamente la división del gasto entre los participantes indicados por el usuario a menos que éste indique un reparto concreto. 5. El usuario pulsará el botón “Guardar”. 6. El sistema registra el nuevo gasto de grupo y recalcula las deudas entre los participantes del gasto.
Postcondiciones	El nuevo gasto grupal ha sido registrado. Las deudas entre los participantes del gasto se han recalculado.
Flujos alternativos	3.a. El usuario no ha completado todos los campos obligatorios del formulario. a1. El sistema notifica al usuario del suceso. En cualquier momento el usuario puede abandonar la aplicación.

Tabla 3.10. Especificación de caso de uso: Añadir transferencia grupal.

Identificador	CU7
Nombre	Añadir transferencia grupal
Descripción	El usuario registra una transferencia entre participantes del grupo.
Actores	Usuario de grupo
Precondiciones	El usuario ha seleccionado un grupo.
Flujo principal	1. El usuario selecciona la opción “Añadir transferencia”. 2. El sistema muestra un formulario y solicita al usuario que lo rellene

	con los datos de la transferencia que se pretende realizar. 3. El usuario introduce los datos solicitados y pulsará el botón "Guardar". 4. El sistema registra la nueva transferencia realizada y recalcula las deudas de las partes involucradas en la transferencia.
Postcondiciones	La nueva transferencia realizada ha sido registrada. Las deudas entre los participantes de la transferencia se han recalculado.
Flujos alternativos	3.a. El usuario no ha completado todos los campos obligatorios del formulario. a1. El sistema notifica al usuario del suceso. En cualquier momento el usuario puede abandonar la aplicación.

3.5 Requisitos de la interfaz de la aplicación

Adicionalmente a los requisitos descritos anteriormente, se han establecido una serie de requisitos acerca de la interfaz de la aplicación que se deberán tener en cuenta durante el desarrollo de la aplicación. Este tipo de requisitos permiten describir como se producirá la interacción entre la aplicación y el usuario. En otras palabras, estos requisitos permiten establecer una primera aproximación sobre qué aspecto debe tener la aplicación y sobre cómo deben ser las transiciones entre las diferentes pantallas.

Es conveniente resaltar que diseñar una buena interfaz de usuario es fundamental para que el producto software desarrollado tenga éxito. Si el usuario final encuentra una interfaz difícil de entender y de manejar, entonces el producto software desarrollado estará condenado al fracaso, independientemente de la calidad interna del sistema.

Con el fin de obtener una visión cercana a la versión final de la aplicación, se han desarrollado diferentes Mockups o maquetas digitales de las diferentes pantallas y transiciones que tendrá la aplicación.

El contenido de la aplicación se encuentra distribuido en diversas secciones de varias pantallas. Para navegar por cada una de las secciones se hará uso de un menú lateral, de manera que se pueda disponer del menú completo de secciones desde cualquier lugar y así optimizar la usabilidad de la aplicación. A continuación, se muestra, en la Figura 3.4, el Mockup del menú lateral con las principales secciones de la aplicación y, en la Figura 3.5, los Mockups de la sección de Transacciones de manera esquematizada. De esta forma, se puede apreciar también la navegación entre las diferentes vistas que tendrá la aplicación. No obstante, en el Anexo del presente documento se adjuntan los Mockups esquematizados de todas las secciones.

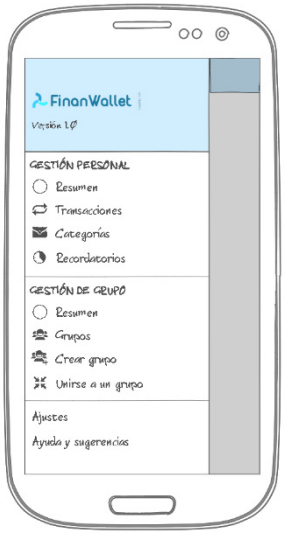

Figura 3.4. Menú lateral de navegación de la aplicación.

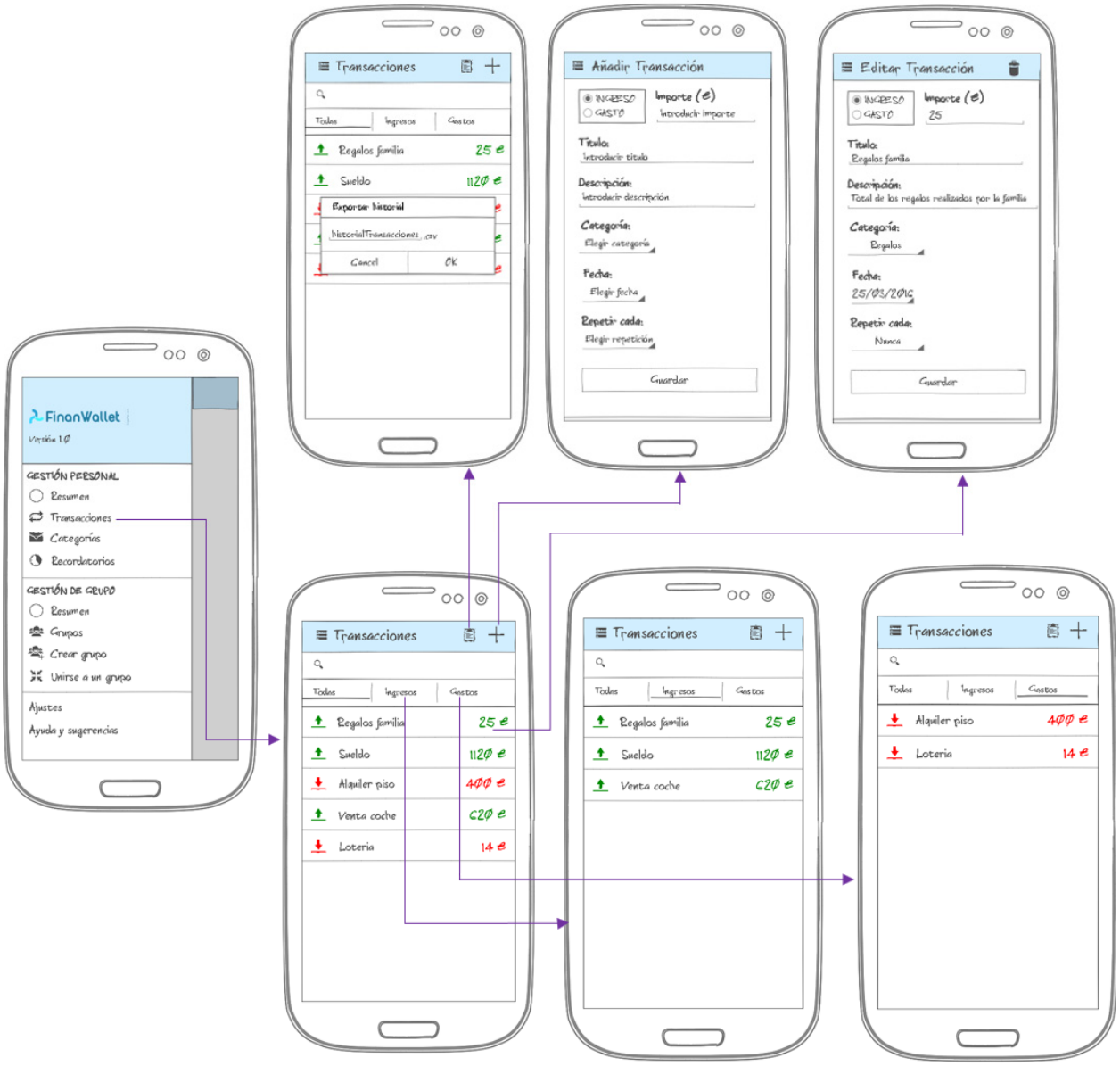

Figura 3.5. Sección Transacciones para la gestión personal.

4. Diseño software

En este capítulo se describe el conjunto de decisiones de diseño que se han adoptado para desarrollar un sistema software capaz de satisfacer los requisitos especificados durante la fase de análisis. El diseño software se divide en dos fases: diseño arquitectónico y diseño detallado. Así pues, este capítulo se centrará concretamente en el desarrollo de cada una de estas fases.

4.1 Diseño arquitectónico

El diseño arquitectónico, también conocido como diseño de alto nivel, es el proceso que define la colección de componentes hardware y software que conforman el sistema software, establece diferentes responsabilidades a cada componente, determina la manera en la que interactuarán y específica dónde se desplegarán.

4.1.1 Diseño del sistema

El diseño arquitectónico de un sistema software es una de las etapas fundamentales y, en muchas ocasiones, la más importante en el desarrollo software. Se debe encontrar una propuesta de diseño que permita al sistema cumplir con los requisitos funcionales y no funcionales establecidos. Es necesario resaltar que los requisitos no funcionales constituyen un elemento crítico en la elección de una arquitectura ya que dependiendo de la decisión que se tome se potenciarán ciertos requisitos en detrimento de otros. Es fundamental considerar lo anteriormente expuesto ya que ningún sistema es capaz de satisfacer todos los requisitos no funcionales completamente, por lo que se deberá realizar un proceso de priorización.

Para escoger una arquitectura software, no se parte de cero, sino que existen diferentes modelos o patrones de diseño que ya han sido probados y que ofrecen buenos resultados. Así pues, para el desarrollo de este proyecto, se ha escogido una arquitectura en tres capas. La arquitectura en tres capas es un modelo de desarrollo software que separa el sistema software en tres partes independientes o capas asignando un rol específico a cada una.

Con el objetivo de lograr una mayor comprensión sobre el diseño arquitectónico elegido, se detallará a continuación cada una de las capas que conforman la arquitectura, así como la función que desempeñan. Además, se apoyará la explicación con la Figura 4.1 que muestra un esquema visual de la interacción que se produce entre las diferentes capas.

- Capa de presentación: se encarga de la interacción con el usuario final. Más concretamente, se responsabiliza de comunicar las solicitudes del usuario a la capa de negocio y de presentar al usuario los resultados recibidos respectivamente. En este caso, la capa de presentación está formada por la aplicación móvil.
- Capa de negocio: se encarga de dotar al sistema de las operaciones necesarias para el correcto funcionamiento de la aplicación. Esta capa se comunica con la capa de presentación, para recibir las peticiones del usuario y presentar los resultados demandados, y con la capa de persistencia, para solicitar al gestor de base de datos

recuperar o almacenar datos. En este proyecto, la capa de negocio está formada por el servicio REST.

- Capa de persistencia o de datos: se encarga gestionar el almacenamiento y manipulación de los datos del sistema. La capa de persistencia está formada por uno o más gestores de bases de datos. En este caso, se emplea el gestor de bases de datos MySQL.

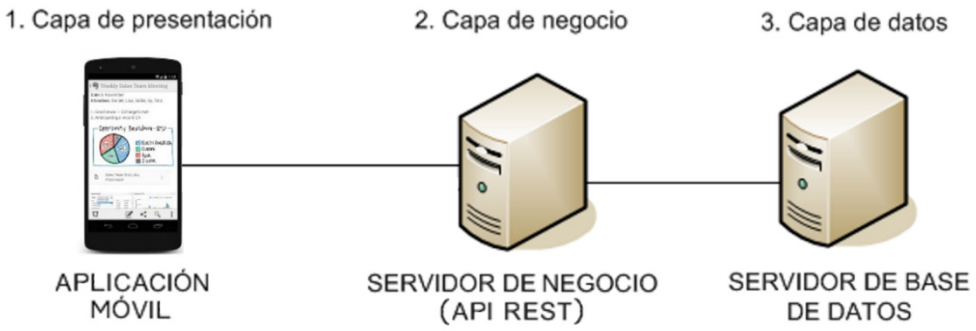

Figura 4.1 Arquitectura del sistema.

4.1.2 Diseño de la aplicación móvil

En cuanto a la aplicación móvil, el diseño arquitectónico escogido se basa en el patrón Modelo-Vista-Presentador (MVP). Este patrón de diseño, derivado del conocido patrón Modelo-Vista-Controlador (MVC), se fundamenta en la separación del código en tres partes claramente diferenciadas: Modelo, Vista y Presentador. Las diferentes partes están acotadas por su responsabilidad y, como consecuencia, el patrón favorece la escalabilidad, mantenibilidad y reusabilidad de la aplicación. Es necesario resaltar que, una de las principales razones de ser de este patrón es conseguir separar las vistas de la aplicación de la lógica de presentación.

A continuación, se desarrollará con mayor nivel de detalle cada una de las partes que conforman este diseño, indicando el papel que desempeñan. La exposición se complementará con la Figura 4.2 que muestra un esquema visual de cómo interactúan las partes anteriormente mencionadas.

- Modelo: incluye tanto la representación de los datos como la lógica de negocio de la aplicación. En otras palabras, es la parte responsable de la gestión de los accesos a los datos con los que opera la aplicación y de las clases que contienen las reglas de negocio.
- Vista: es la presentación del modelo a través de una interfaz de usuario. Es decir, es la parte responsable de mostrar los datos que se requieran en cada instante y, de esta forma, permitir a la aplicación interaccionar con el usuario. Delega al presentador todo aquello que no sea relacionado con la vista propiamente dicha.
- Presentador: es el intermediario entre el modelo y la vista. Por un lado, dirige hacia el modelo los eventos que se producen en la vista y, por otro lado, actualiza la vista con la información procedente del modelo. Es decir, es el encargado de gestionar toda la lógica de presentación.

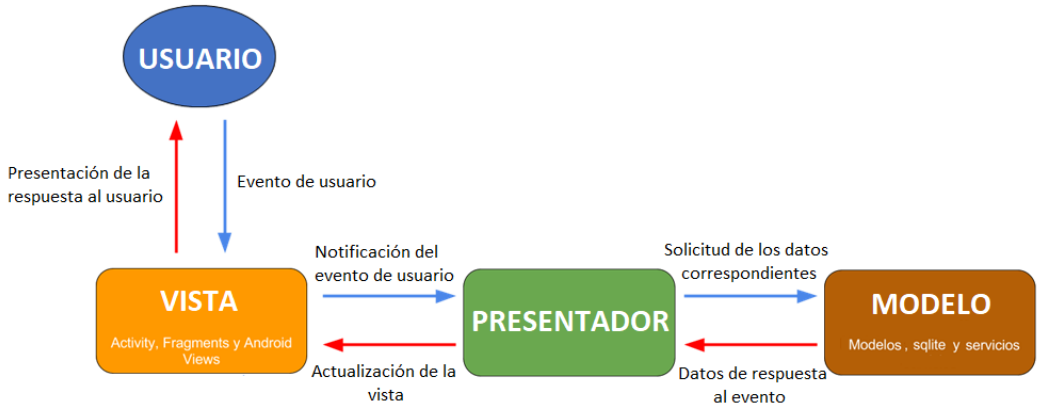

Figura 4.2 Patrón MVP.

Una vez elegido el patrón arquitectónico para llevar a cabo el desarrollo de la aplicación, es conveniente modelar la arquitectura de la aplicación a través de diagramas de componentes. Un diagrama de componentes permite representar cada uno de los componentes software de alto nivel que conforman la arquitectura de la aplicación, así como las interacciones que existen entre ellos a través de interfaces.

Así pues, por un lado, en la Figura 4.3 se muestra el diagrama de componentes correspondiente a la aplicación en cuestión. Se ha definido, en cada parte de la arquitectura, los componentes necesarios para implementar las funcionalidades presentes en la aplicación. En otras palabras, se han mapeado los casos de uso tanto de la gestión personal como de la gestión de grupo realizando agrupaciones funcionales en base a los conceptos lógicos del negocio. Por otro lado, en la Figura 4.4 se muestran las interfaces a nivel operacional de los componentes que corresponden a la parte del modelo de la aplicación.

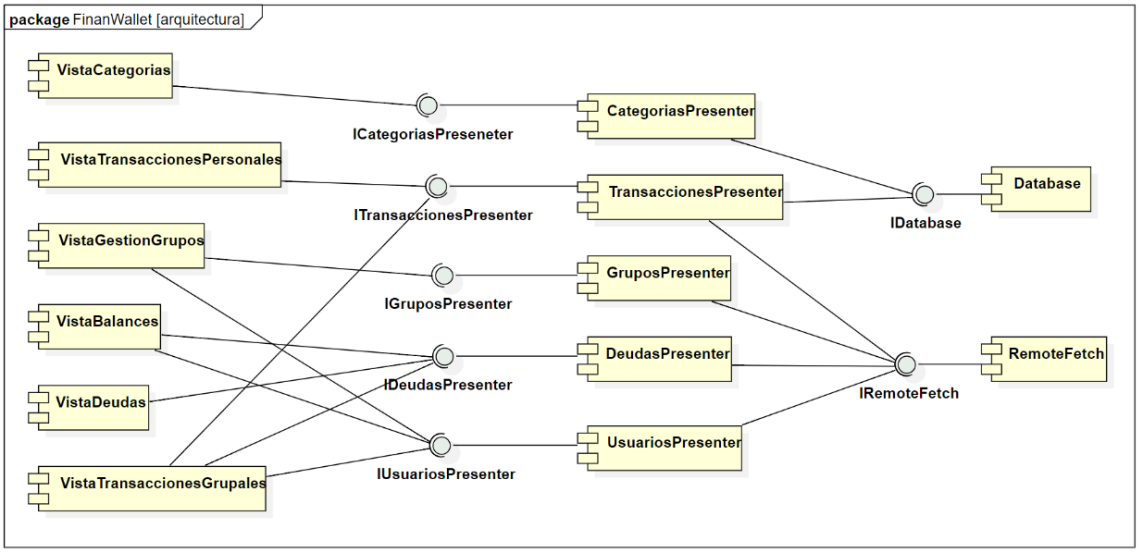

Figura 4.3 Diagrama de componentes de la aplicación móvil.

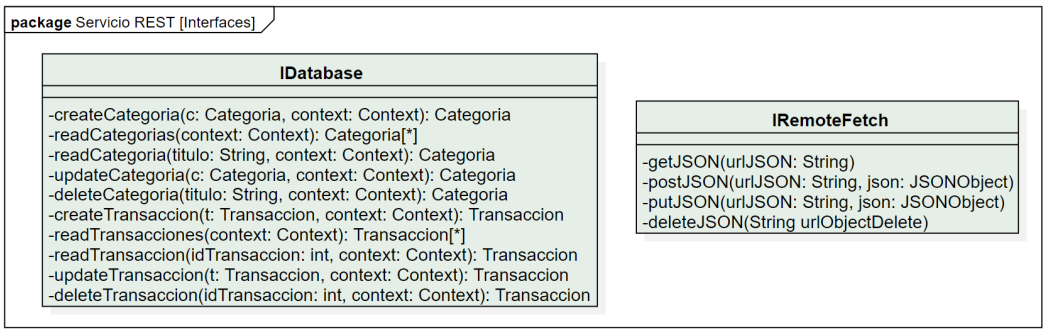

Figura 4.4 Interfaces del modelo de la aplicación móvil.

4.1.3 Diseño de la API REST

Para llevar a cabo la gestión de grupo, se ha implementado una API REST que permita la manipulación de los datos de un grupo desde diversos dispositivos a través de la aplicación. Así pues, el servicio REST se conectará con la base de datos MySQL donde se almacenan los datos correspondientes a los grupos y se encargará de gestionar dicha base de datos en función de las peticiones que realicen los usuarios de grupo de la aplicación.

Para realizar el diseño de la API REST, se deben seguir una serie de pautas que permitan detallar todos los aspectos que conciernen a la API. De esta forma, una API REST:

- Debe manejar recursos. Los recursos hacen referencia a las entidades que representan los conceptos de negocio que pueden ser accedidos públicamente. Por lo cual, se deben definir todos los recursos que forman el servicio.
- Debe estar basada en URIs. Cada recurso posee una URI que lo identifica unívocamente. Por tanto, se debe decidir que URI se le asignará a cada recurso.
- Debe seguir los métodos HTTP estándar:
  - POST/PUT: para la creación de un nuevo recurso. POST se utiliza cuando el servidor es el responsable de decidir la URI del nuevo recurso mientras que PUT se utiliza cuando el cliente tiene dicha responsabilidad.
  - PUT: para sobrescribir el estado de un recurso en base a la copia o representación del estado del recurso que tiene el cliente.
  - DELETE: para eliminar un recurso.
  - GET: para leer una representación del estado de un recurso.

  Por consiguiente, se debe especificar qué métodos serán soportados por cada recurso (en cada URI) así como los códigos de respuesta que retornará cada método.
- Debe transferir información o representaciones de los recursos utilizando los tipos de formatos válidos en Internet. Normalmente se usan los formatos JSON y XML tanto para el envío como para la recepción de datos. Así pues, para cada uno de los recursos se tiene que decidir cuál va a ser su representación.

A continuación, en la Tabla 4.1 se muestra el diseño de la API REST teniendo en cuenta las pautas anteriormente mencionadas. De esta forma, se detallan los recursos que forman el servicio, las URIs asignadas a cada recurso y los métodos que son capaces de soportar los recursos junto con sus códigos de respuesta.

Tabla 4.1. Diseño de la API REST.

Recurso	URI	Métodos	Códigos de respuesta HTTP
Grupos	/grupos Query Params: email	GET POST	200 201, 415
Grupo	/grupos/{idGrupo}	GET PUT DELETE	200 200, 415 200, 204
UsuariosGrupo	/grupos/{idGrupo}/usuarios	GET	200
UsuarioGrupo	/grupos/{idGrupo}/usuarios/{email}	PUT DELETE	201, 415 200, 204
Deudas	/grupos/{idGrupo}/deudas	GET POST	200 201, 415
Deuda	/grupos/{idGrupo}/deudas/{idDeuda}	PUT	200, 415
Transacciones	/grupos/{idGrupo}/transacciones Query Params: tipo	GET POST	200 201, 415
Transacción	/grupos/{idGrupo}/transacciones/{idTransaccion}	GET PUT DELETE	200 200, 415 200, 204

Además de los códigos de respuesta expuestos en la tabla, existen los siguientes códigos de respuesta que se pueden producir para todos y cada uno de los métodos de cualquiera de los recursos: 400, 404, 500 y 503.

En cuanto a las representaciones de cada uno de los recursos, por cuestiones de limitación de la memoria, solamente se mostrará a modo de ejemplo la representación correspondiente al recurso Transacción debido a que es un concepto básico de la aplicación y que se considera de mayor complejidad que el resto. A continuación, se muestra la representación del recurso en XML y JSON, formatos que serán soportados por la API.

JSON

```
{
    "type": "transferencia",
    "fecha": "22/7/2018",
    "idTransaccion": 64,
    "importe": 7.5,
    "realizador": {
        "email": "pablo_gomez@hotmail.com",
        "nombre": "Pablo Gomez"
    },
    "titulo": "Pago entradas de cine",
    "receptor": {
        "email": "marta_fernandez@gmail.com",
        "nombre": "Marta Fernandez"
    }
}
```

XML

```
<transaccion xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:type="transferencia">
    <fecha>22/7/2018</fecha>
    <idTransaccion>64</idTransaccion>
    <importe>7.5</importe>
    <realizador>
        <email>pablo_gomez@hotmail.com</email>
        <nombre>Pablo Gomez</nombre>
    </realizador>
    <titulo>Pago entradas de cine</titulo>
    <receptor>
        <email>marta_fernandez@gmail.com</email>
        <nombre>Marta Fernandez</nombre>
    </receptor>
</transaccion>
```

Figura 4.5 Representaciones del recurso Transaccion.

Para llevar a cabo la implementación de la API REST, se ha modelado el diagrama de componentes que se muestra en la Figura 4.6. Este diagrama muestra la arquitectura que se utilizará para desarrollar el servicio, así como la interfaz a nivel operacional del componente DAO. Por un lado, un componente `Controller` se encargará de procesar la consulta o consumo de los diferentes servicios de la API y, por otro lado, un componente `DAO` se encargará de interactuar con la base de datos MySQL.

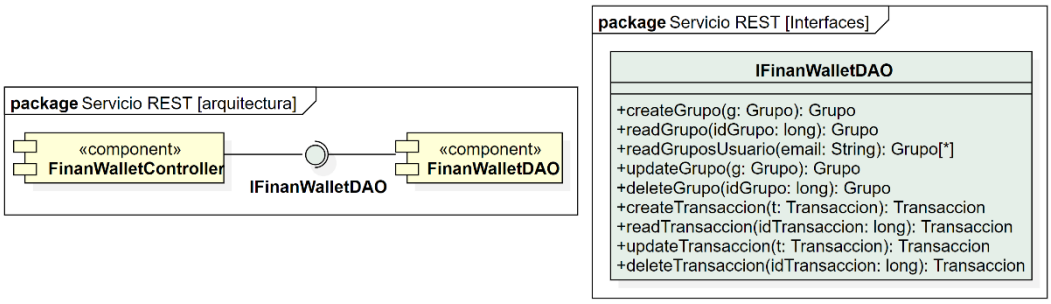

Figura 4.6 Diagrama de componentes del servicio REST y sus interfaces.

4.1.4 Diseño de despliegue

Para proveer una vista completa de la implementación del sistema, es necesario detallar las relaciones físicas existentes entre los componentes software y hardware que lo conforman. El diagrama de despliegue nos permite realizar esta labor proporcionando una representación visual de la distribución, en tiempo de ejecución, de los diferentes componentes software sobre las distintas particiones físicas del sistema. De esta manera, en la Figura 4.8, se muestra el diagrama de despliegue desarrollado para el sistema en cuestión. Como se puede observar, la aplicación FinanWallet se empaqueta en un fichero .apk, formato utilizado en las aplicaciones Android, y se despliega teniendo acceso a la librería SQLite. La aplicación se conecta con el servidor GlassFish donde se aloja el servicio REST y éste a su vez con el servidor de MySQL.

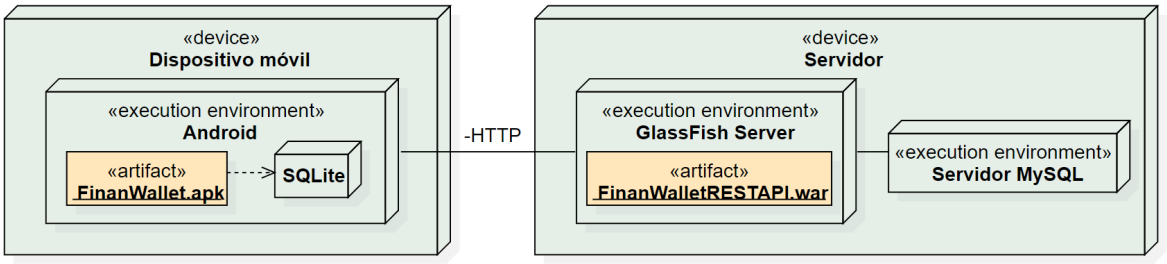

Figura 4.7 Diagrama de despliegue.

4.2 Diseño detallado

El diseño detallado es el proceso que se encarga de profundizar en el diseño preliminar de un sistema o componente hasta llegar a las unidades de programación o clases de implementación que lo conforman. Este proceso tiene como finalidad detallar las funcionalidades y el comportamiento de cada clase de manera que el proyecto quede preparado para su implementación.

4.2.1 Clases

Con el objetivo de aumentar el nivel de detalle, se ha desarrollado, en la Figura 4.9, el diagrama de clases correspondiente a la aplicación. Un diagrama de clases permite representar gráficamente y de manera estática la estructura general de un sistema. Es decir, muestra las

diferentes clases y sus interacciones permitiendo, de esta forma, dar soporte a la implementación del software al constituir elementos cercanos a la programación.

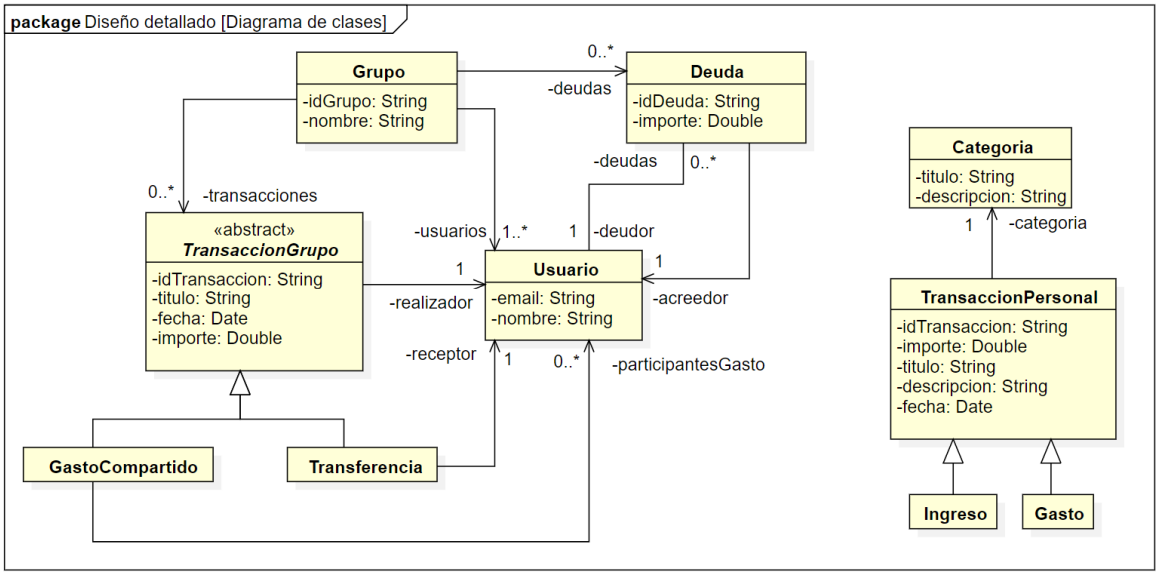

Figura 4.8 Diagrama de clases.

4.2.2 Almacenamiento aplicación móvil

Para llevar a cabo la gestión personal de los gastos, la aplicación móvil posee una base de datos SQLite. En la Figura 4.9, se muestra el diseño desarrollado para la base de datos en cuestión.

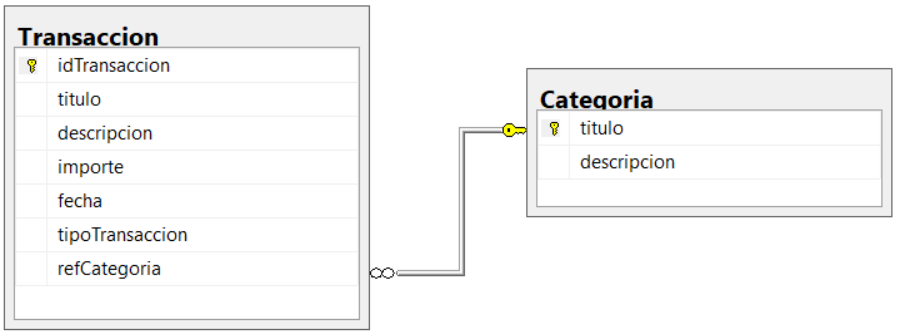

Figura 4.9 Diagrama de la base de datos de la aplicación móvil.

4.2.3 Almacenamiento servicio REST

Para llevar a cabo la gestión grupal de los gastos, se ha diseñado una base de datos externa MySQL capaz de dar soporte a los datos gestionados a través del servicio REST. En la Figura 4.10, se muestra el diseño de la correspondiente base de datos.

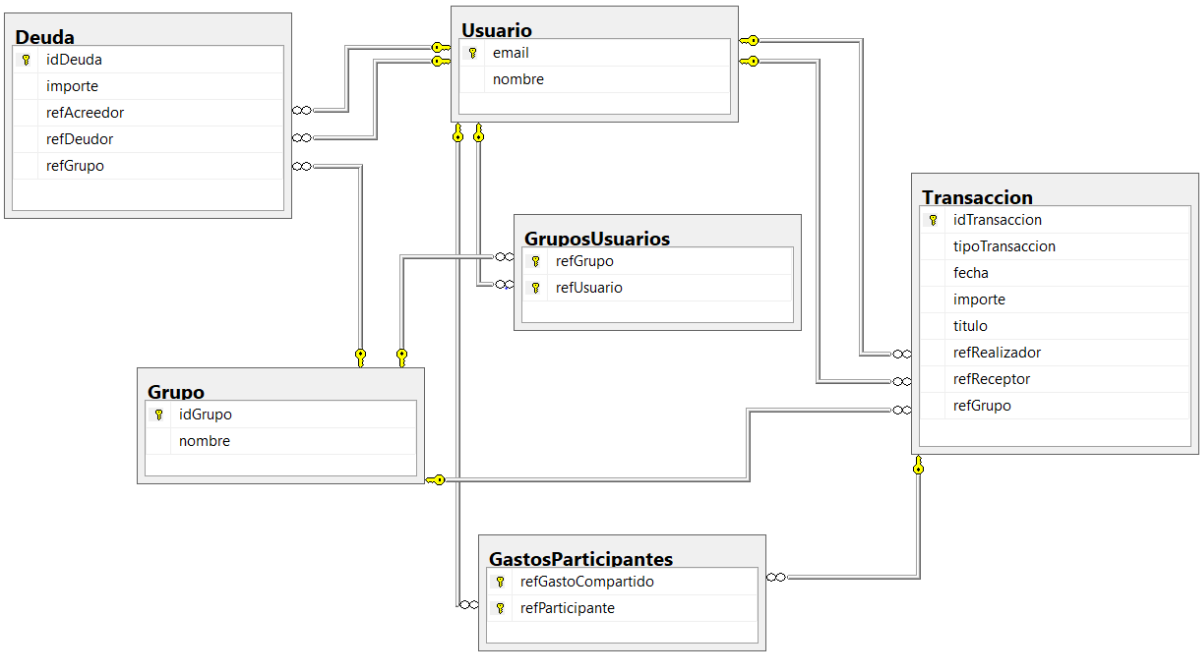

Figura 4.10 Diagrama de la base de datos del servicio REST.

5. Implementación

Una vez completadas las fases de análisis y diseño, se procede al desarrollo de la fase de implementación. Así pues, este capítulo, describe el desarrollo del proyecto en cuestión y el uso de las diferentes tecnologías en base a las decisiones tomadas durante la fase de diseño.

El capítulo consta de dos secciones. En la primera sección, se detallará el desarrollo de la aplicación móvil y, en la segunda sección, se describirá el desarrollo del servicio REST.

5.1 Implementación de la aplicación móvil

5.1.1 Estructura del proyecto

La aplicación móvil ha sido desarrollada utilizando el entorno de desarrollo integrado Android Studio, basado en IntelliJ IDEA y designado por Google como el IDE oficial para el desarrollo de aplicaciones Android.

El código referente al desarrollo de la aplicación Android ha sido estructurado, dentro de Android Studio, en los paquetes que se muestran en la Figura 5.1. A lo largo de este apartado, se describirá la implicación de cada uno de estos paquetes en el desarrollo.

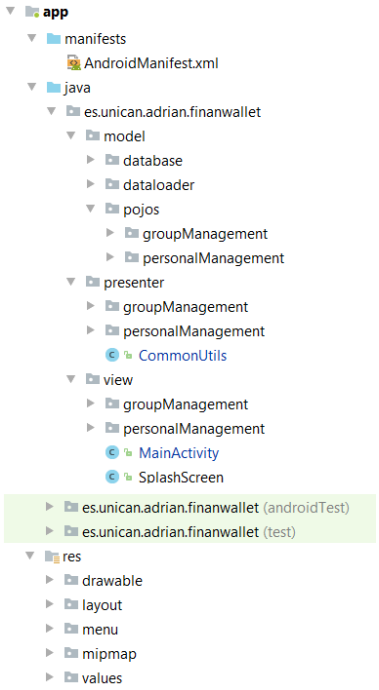

Figura 5.1 Estructura de paquetes de la aplicación.

Es conveniente comentar que un proyecto de Android Studio puede tener uno o varios módulos. Un módulo es un conjunto de archivos de origen y configuraciones de compilación que permiten separar el proyecto en diferentes unidades discretas de funcionalidad. Así pues, en la Figura 5.1, solamente se muestra el módulo app por motivos de simplicidad. Este módulo contiene las jerarquías de archivos más importantes de la aplicación. Más concretamente, el

módulo app proporciona un contenedor para el código fuente de la aplicación, los archivos de recursos que utilizará dicha aplicación y el archivo de configuración a nivel de app.

De esta forma, los archivos quedan agrupados en los siguientes grupos:

- `manifests`: contiene el archivo `AndroidManifest.xml`. Este archivo de configuración proporciona información esencial para que el sistema pueda ejecutar el código de la aplicación. Entre otras cosas, `AndroidManifest.xml` nombra el paquete Java de la aplicación, nombra y describe los componentes de la aplicación (actividades, servicios, receptores de mensajes…), declara los permisos que debe tener la aplicación para acceder a las partes protegidas de una API, declara la compatibilidad con las versiones de Android y determina la actividad principal o punto de entrada de la aplicación.
- `java`: contiene los archivos de código fuente Java, incluido el código de prueba JUnit. Como se puede apreciar, el código referente al desarrollo de la aplicación móvil ha sido dividido en tres paquetes claramente diferenciados: `model`, `view`, `presenter`. Esta separación del código se debe a la decisión tomada durante la fase de diseño de seguir el patrón Modelo-Vista-Presentador (MVP). A continuación, se expone las diferentes funciones de cada paquete:
  - `model`: contiene los siguientes tres subpaquetes: `database`, `dataloader`, `pojos`. Así pues, `database` se encarga de la gestión de los accesos a la información almacenada localmente (para la gestión personal), `dataloader` se encarga de la gestión de los accesos a la información proveniente de la API REST (para la gestión grupal) y `pojos` se encarga de la lógica de negocio de la aplicación. Este último, distribuye la lógica de negocio de la aplicación en dos subpaquetes, uno concerniente a la gestión grupal y otro relativo a la gestión personal.
  - `presenter`: se encarga de intermediar entre `model` y `view`. Es decir, esta parte se encarga de gestionar la lógica de presentación de la aplicación, dirigiendo eventos que se producen en la vista y actualizando la vista con la información procedente del modelo. Igualmente, para una mayor claridad, el código se distribuye en dos subpaquetes en función de si corresponde a la gestión grupal o a la gestión personal de los gastos. La única excepción es la clase `CommonUtils`, que implementa métodos comunes a los dos tipos de gestión.
  - `view`: se encarga de mostrar la información requerida en cada instante y de interaccionar con el usuario. Así pues, delega al paquete `presenter` aquello que no esté estrictamente relacionado con las vistas de la aplicación. Como en las otras partes, el código se ha organizado en función de si está relacionado con las vistas de la gestión personal o de la gestión grupal.
- `res`: contiene todos los archivos de recursos necesarios para el proyecto. Los diferentes tipos de recursos se distribuyen entre las siguientes subcarpetas: `drawable` (contiene las imágenes utilizadas por la aplicación), `layout` (contiene los archivos .xml correspondientes a las vistas de la interfaz gráfica), `menu` (contiene los ficheros .xml concernientes a los menús de la aplicación), `mipmap` (contiene los iconos utilizados por

la aplicación) y `values` (contiene otros archivos .xml como `strings.xml` para definir cadenas de texto o `colors.xml` para definir un conjunto de colores).

5.1.2 Interfaz gráfica

Desde el punto de vista del usuario, una aplicación está conformada por un conjunto de pantallas con las que puede interactuar. Para Android, estas pantallas se denominan actividades y constituyen objetos que extienden de la clase `Activity`. Cuando se inicia una actividad, lo hace con una pantalla temporalmente vacía sobre la que se tendrán que colocar los diferentes elementos gráficos de su interfaz. Así pues, cada actividad se hace responsable de asignar la interfaz de usuario correspondiente y, por tanto, de gestionar los diferentes componentes visuales que contiene. Por componente visual o vista se entiende todo elemento gráfico que extiende de la clase `View`, como puede ser un `Button` o un `CheckBox`. Asimismo, es común utilizar otros elementos más complejos denominados `layouts` que extienden de la clase `ViewGroup` y que permiten contener a su vez a otros componentes, como es el caso de un `LinearLayout` o un `TableLayout`.

La interfaz gráfica de una aplicación Android puede ser definida mediante ficheros XML o bien a través de código dentro de la actividad que gestiona dicha interfaz. Realizar la interfaz mediante código, instanciando cada uno de los objetos y agrupándolos de una manera concreta, resulta una tarea bastante tediosa de llevar a cabo. Mientras que definir la interfaz gráfica en un fichero XML, independiente al código de la actividad, dota al proyecto de una mayor simplicidad y claridad en la descripción de las interfaces gráficas, así como facilidad y rapidez ante un posible cambio de la disposición gráfica. Por tanto, en este proyecto se ha optado por trabajar con ficheros XML para definir las interfaces gráficas.

A continuación, en la Figura 5.2 se muestra, a modo de ejemplo, el código XML que ha sido utilizado para definir la vista correspondiente a la representación de las deudas existentes dentro de un grupo.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical" >
    <ListView
        android:id="@android:id/list"
        android:layout_width="match_parent"
        android:layout_height="wrap_content" >
    </ListView>
</LinearLayout>
```

Figura 5.2 Código XML de la vista de deudas de un grupo.

En la Figura 5.2, podemos apreciar que se ha definido un elemento `ListView`. Este elemento representa una lista vertical en la API de Android y, en este caso, se ha utilizado para mostrar las distintas deudas existentes dentro de un grupo. Esta clase está preparada para recibir los ítems que desplegará en la interfaz a través de una estructura de datos como puede ser un `ArrayList` y, con la ayuda de un `Adapter`, convertirá cada resultado en una vista para permitir así su representación dentro de la lista.

Una vez se ha definido la interfaz, para poder acceder a ella es necesario que desde una clase `Activity` se cargue el fichero XML. En este caso, la interfaz donde se define la lista de deudas del grupo ha sido cargada concretamente dentro de una clase que extiende de `ListFragment`. Esta clase representa una parte de la interfaz de usuario en una `Activity` y tiene como finalidad mostrar una lista de elementos administrados por un adaptador. Así pues, proporciona métodos específicos para administrar las diferentes vistas de una lista, como puede ser `onListItemClick()` que se ejecutará cuando el usuario seleccione una opción de la lista. En la Figura 5.3 se muestran los métodos más relevantes de la clase implementada para la funcionalidad en cuestión.

```
@Override
public View onCreateView(LayoutInflater inflater, ViewGroup container, Bundle savedInstanceState)
{
    this.idGrupo = getArguments().getString( key: "idGrupo");

    searchDeudas = (SearchView) container.getRootView().findViewById(R.id.searchViewGrupo);
    searchDeudas.setQueryHint("Buscar deuda");
    searchDeudas.setOnQueryTextListener(this);

    return inflater.inflate(R.layout.fragment_debts_group_list, container,  attachToRoot: false);
}//onCreateView
@Override
public void onActivityCreated(Bundle savedInstanceState)
{
    super.onActivityCreated(savedInstanceState);
    this.deudasPresenter = new DeudasPresenter( listDeudasView: this);
    this.deudasPresenter.setIdGrupo(idGrupo);
    dialog= new ProgressDialog(getContext());
}//onActivityCreated
@Override
public void onResume()
{
    super.onResume();
    this.deudasPresenter.start();
}//onResume
@Override
public void showList(List<Deuda> deudas)
{
    DeudasListAdapter deudasListAdapter= new DeudasListAdapter(getContext(), deudas);
    getListView().setAdapter(deudasListAdapter);
    getListView().deferNotifyDataSetChanged();
}//showList
@Override
public boolean onQueryTextSubmit(String query)
{
    searchDeudas.clearFocus();
    return false;
}
```

Figura 5.3 Código Java de los métodos de gestión de la vista de deudas de un grupo.

A continuación, se detalla el cometido de cada uno de los métodos que se muestran:

- `onCreateView()`: se encarga principalmente de asignar el fichero XML que define la lista al fragmento de la actividad. Para ello, utiliza el método `inflate()` que se encarga de convertir el XML en la estructura de objetos java equivalente para que se pueda construir y añadir la vista a la interfaz. Además, se encarga de conseguir la referencia del elemento `SearchView` definido en el archivo XML utilizado en la actividad padre para establecer un `Listener` y así dirigir los eventos de dicho elemento.
- `onActivityCreated()`: se ejecuta cuando la actividad que contiene el fragment termina de crearse. En este método se inicializa la clase `DeudasPresenter` en la que se delegará la lógica de presentación de la vista. Además, se inicializa un elemento ProgressDialog o mensaje de espera que será activado por el presentador cada vez que interactúe entre la vista y el modelo. De esta forma, se avisa al usuario de que se

están realizando acciones en segundo plano y que debe esperar a que finalicen para observar el resultado esperado.

- `onResume()`: se ejecuta cuando el fragmento pasa a primer plano, incluso cuando se crea por primera vez. En este método, se delega al presentador la lógica de presentación de la vista.
- `showList()`: se ejecuta cuando el presentador lo solicite y se encarga de mostrar por la interfaz la lista pasada por parámetro que, en este caso, serán las deudas de un grupo. Esto es posible gracias al uso de la clase `DeudasListAdapter` que se encargará de convertir cada deuda o elemento de la lista que se pasa por parámetro en una vista para así conseguir su representación dentro del elemento `ListView` de la interfaz.
- `onQueryTextChange()`: se ejecuta cuando el usuario desea filtrar en la lista en base a un texto introducido. Así pues, la vista capta el evento y delega de nuevo en el presentador las decisiones de presentación ante el evento producido.

Como resultado de todo esto, obtenemos la vista que se muestra en la Figura 5.4 para representar las deudas existentes en un grupo.

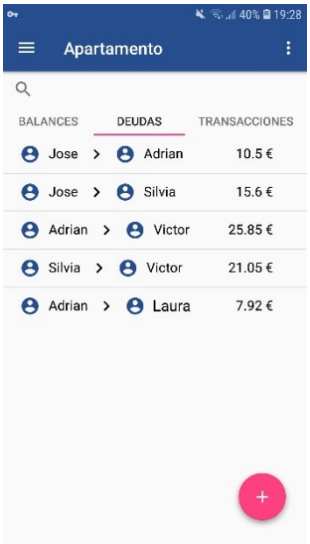

Figura 5.4 Vista de las deudas de un grupo.

5.1.3 Lógica de presentación

En este proyecto, como se ha dicho con anterioridad, se ha seguido un patrón de diseño Modelo-Vista-Presentador (MVP). El objetivo de este patrón es principalmente separar las vistas de la aplicación de la lógica de presentación. Android posee un problema derivado del hecho de que las actividades están fuertemente acopladas tanto a la interfaz como al acceso a los datos, lo cual hace que la aplicación sea difícilmente extensible y mantenible. Así pues, este patrón pone solución a este problema delegando desde las actividades o fragmentos toda la lógica a una entidad conocida como presentador. Es decir, cada vista contiene una referencia al presentador y llamará a uno de sus métodos cada vez que se realice una acción sobre la interfaz. Por tanto, el presentador se encargará de decidir que se pinta en la vista ante los distintos eventos que se produzcan. De esta forma, podríamos cambiar fácilmente la vista y mantener el presentador y el modelo, obteniendo así un código altamente reutilizable.

Continuando con el ejemplo expuesto en el apartado anterior, en la Figura 5.3, se puede observar como la vista, correspondiente a las deudas de un grupo, crea una instancia de la clase `DeudasPresenter`. Esta clase se encargará de interactuar entre todas las vistas relacionadas con las deudas de un grupo y el modelo de datos. Así pues, la vista hace uso de la clase `DeudasPresenter` llamando al método `start()` cada vez que se ejecuta el método `onResume()`, es decir, cada vez que el fragmento pasa a un primer plano se solicita a `DeudasPresenter` la lógica de presentación correspondiente.

A continuación, en la Figura 5.5, se muestran las partes de código más relevantes de la clase `DeudasPresenter` para la lectura de las deudas de un grupo.

```
private class LeerDeudasInternet
        extends AsyncTask<Object, Boolean, Boolean>
{
    @Override
    protected void onPreExecute()
    {
        deudasListView.showProgress( state: true);
    }//onPreExecute

    @Override
    protected Boolean doInBackground(Object... l)
    {
        getDeudas();
        return true;
    }//doInBackground

    @Override
    protected void onPostExecute(Boolean result)
    {
        if (result)
        {
            deudasListView.showList(deudas);
        }
        else
        {
            deudasListView.showErrorMessage();
        }

        deudasListView.showProgress( state: false);
    }//onPostExecute
}//LeerDeudasInternet
```

```
public void start()
{
    LeerDeudasInternet leerDeudasInternet= new LeerDeudasInternet();
    leerDeudasInternet.executeOnExecutor(Executors.newScheduledThreadPool( i: 1));
}// start

public boolean getDeudas()
{
    try
    {
        String url= RemoteFetch.URL_GROUP+"/"+idGrupo+"/deudas";
        RemoteFetch.getJSON(url);
        setDeudas(ParserJSON.readDeudasArray(RemoteFetch.getBufferedInputData()));
        return true;
    }
    catch(Exception e)
    {
        Log.e( tag: "ERROR", msg: "Error en la obtención de las deudas: "+e.getMessage());
        return false;
    }
}//getDeudas

public void filtrarDeudas(String query)
{
    deudasListView.showList(CommonUtils.filtrarDeudas(deudas, query));
}//filtrarDeudas
```

Figura 5.5 Fragmentos de código de la clase DeudasPresenter.

Como se puede apreciar, el método `start()` crea una instancia de una clase auxiliar que extiende de `AsyncTask`. La clase abstracta `AsyncTask`, proporcionada por Android, tiene como objetivo permitir al programador ejecutar operaciones en un segundo plano y mostrar los resultados en el hilo de la interfaz de usuario sin necesidad de tener que manejar manualmente hilos del sistema operativo. Por tanto, la clase `AsyncTask`, proporciona un conjunto de métodos para sobrescribir entre los que se repartirá la funcionalidad de la tarea asíncrona. En este caso, la tarea asíncrona se ha implementado para solicitar al modelo las deudas existentes del grupo y actualizar la vista en base a los resultados. Para ello, se han sobrescrito los siguientes métodos de la tarea asíncrona:

- `onPreExecute()`: se ejecutará antes del código principal de la tarea. En este método se actualiza la vista para mostrar un mensaje de espera al usuario mientras se realizan las operaciones de la tarea en segundo plano.
- `doInBackGround()`: se ejecutará después del método anterior y se encarga de realizar las operaciones principales de la tarea. En este caso, se llama al método `getDeudas()` que, mediante las clases `RemoteFetch` y `ParserJSON` correspondientes al modelo de la aplicación, obtendrá la lista de deudas existentes en el grupo.

- `onPostExecute()`: se ejecutará cuando finalice la tarea o, en otras palabras, cuando finalice el método anteriormente expuesto. En este método, se actualizará la vista de la aplicación para que se muestre la lista de las deudas del grupo obtenida o un mensaje de error en su defecto.

Otra parte del código que conviene resaltar de la Figura 5.6 es el método `filtrarDeudas()`. Este método es invocado por la vista cuando se detecta que el usuario ha introducido un texto en el campo de búsqueda. Así pues, este método invoca a otro de la clase `CommonUtils`, por cuestiones de reusabilidad, que devolverá la lista filtrada en base al texto. De esta forma, `DeudasPresenter` actualiza la vista en base a esa lista.

Por último, es conveniente aclarar que, todas las interacciones Vista-Presentador-Modelo que se producen en la aplicación, se realizan en base al contrato definido en sus interfaces correspondientes.

5.1.4 Modelo de datos

Una vez detallada la implementación de la interfaz gráfica y de la lógica de presentación de la aplicación, es necesario describir la parte responsable de la gestión de los accesos a los datos con los que opera la aplicación. La aplicación móvil ofrece soporte para dos tipos de gestión de gastos: personal y grupal. Así pues, para manejar cada situación, se ha utilizado un tipo de persistencia diferente.

Para llevar a cabo la gestión personal de los gastos, la aplicación móvil utiliza una base de datos SQLite. En otras palabras, la aplicación hace uso de la librería software SQLite para manipular de forma persistente los diferentes datos correspondientes a la gestión personal de gastos. De esta forma, se ha desarrollado la clase `DBHelper` que hereda todas las funcionalidades de `SQLiteOpenHelper`, una clase abstracta que nos ayuda a gestionar una base de datos SQLite en Android. Por consiguiente, `DBHelper` redefine los siguientes métodos:

- `onCreate(SQLiteDatabase db)`: se ejecuta automáticamente cuando sea necesaria la creación de la base de datos. Es decir, se responsabiliza de la creación de todas las tablas necesarias y, en caso de ser necesario, de la inserción de los datos iniciales. Para llevar a cabo esta tarea, este método hace uso de una función de la API de SQLite denominada `execSQL()` que se limita a ejecutar el código SQL que se pasa como parámetro.
- `onUpgrade(SQLiteDatabase db, int oldVersion, int newVersion)`: se ejecuta automáticamente cuando sea necesaria una actualización del esquema de la base de datos.

Posteriormente, se ha desarrollado una clase, denominada `Database`, que define las operaciones CRUD necesarias para permitir el correcto funcionamiento de la gestión personal de gastos. A continuación, se muestra, a modo de ejemplo, la creación de un registro en la tabla `Categoria` de la base de datos.

```java
public static void createCategoria(Categoria c, Context context)
{
    Database.DBHelper dbHelper = new Database.DBHelper(context);
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    ContentValues values = new ContentValues();

    Categoria exists= readCategoria(c.getTitulo(), context);
    if(exists == null)
    {
        try
        {
            values.put(CategoriaEntry.COLUMN_TITULO, c.getTitulo());
            values.put(CategoriaEntry.COLUMN_DESCRIPCION, c.getDescripcion());
            db.insert(Database.CategoriaEntry.TABLE_NAME,  nullColumnHack: null, values);
        }
        catch (SQLException e)
        {
            Log.e( tag: "[DATABASE ERROR]: ",  msg: "No se ha podido insertar la categoria. ¿Valor duplicado?");
        }
    }
}
```

Figura 5.6 Fragmento de código de la creación de un registro en la tabla `Categoria`.

Como se puede apreciar en la Figura 5.6, el patrón que se sigue para realizar operaciones sobre la base de datos es, en primer lugar, conseguir una referencia a un objeto de la clase `DBHelper` y, seguidamente, llamar al método `getReadableDatabase()` o `getWritableDatabase()` para obtener una instancia de la base de datos de sólo lectura o de lectura y escritura respectivamente. Una vez realizados estos pasos, es posible realizar todas las acciones que se deseen sobre la base de datos.

En cuanto a la gestión grupal de gastos, la aplicación móvil utiliza la API REST para enviar peticiones HTTP al servicio desarrollado, que es la parte responsable de manejar la persistencia de los datos relativos a este tipo de gestión de gastos. Así pues, para comunicarse con el servicio REST, se han desarrollado las clases `RemoteFetch` y `ParserJSON`.

La clase `RemoteFetch`, por su parte, implementa el conjunto de métodos necesarios para realizar las diferentes peticiones HTTP especificadas en la API REST del servicio. Es decir, esta clase define de forma genérica, los métodos relativos a la realización de las peticiones GET, POST, PUT y DELETE. Estos métodos reciben como parámetros la URL del recurso al cual se desea lanzar la petición en cuestión y, en el caso de los métodos que realizan las peticiones PUT y POST, la representación del estado del recurso que se desea enviar al servicio. En la Figura 5.7 se muestra el código relativo al método que implementa la realización de una petición GET a un servicio.

```java
public static void getJSON(String urlJSON) throws IOException {
    URL url = new URL(urlJSON);
    HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
    urlConnection.addRequestProperty( s: "Accept",  s1: "application/json");
    bufferedInputData =  new BufferedInputStream(urlConnection.getInputStream());
}//getJSON
```

Figura 5.7 Fragmento de código de una petición GET a un servicio.

Como se aprecia en la figura, se ha escogido JSON como formato de datos para transferir los estados de un recurso entre el cliente y el servidor. Así pues, como se puede presuponer, para permitir que la comunicación entre el cliente y servidor sea posible, es necesario la implementación de una clase auxiliar que ofrezca los métodos necesarios para transformar las cadenas JSON en objetos Java y viceversa. La clase `ParserJSON`, comentada anteriormente, es la encargada de llevar a cabo esta tarea. A continuación, en la Figura 5.8, se muestra el código relativo a la transformación del listado de deudas de un grupo de formato JSON a objetos Java.

```java
public static List<Deuda> readDeudasArray(InputStream in) throws IOException
{
    JsonReader reader = new JsonReader(new InputStreamReader(in, UTF8));
    List<Deuda> debtsList = new ArrayList<>();
    reader.beginObject();
    while (reader.hasNext()) {
        String name = reader.nextName();
        if (name.equals(RECURSOS_DEUDAS_JSON))
        {
            reader.beginArray();
            while (reader.hasNext())
            {
                debtsList.add(readDeuda(reader));
            }

        }
        else {
            reader.skipValue();
        }
    }
    reader.endArray();
    return debtsList;
}//readDeudasArray
```

Figura 5.8 Código Java correspondiente a la conversión de formato JSON a objetos Java de una lista de deudas de un grupo.

5.2 Implementación del servicio REST

El servicio REST ha sido desarrollado haciendo uso del entorno de desarrollo integrado Eclipse. Este servicio, tiene como principal objetivo, proporcionar a la aplicación móvil el soporte necesario para manejar la persistencia de todos los datos concernientes a la gestión grupal de gastos. Este hecho, por tanto, brinda la posibilidad de manipular los datos de un grupo desde diversos dispositivos a través de la aplicación.

Así pues, para lograr este cometido, el servicio REST se conecta con una base de datos MySQL, donde se almacenan los datos correspondientes a los grupos, y gestiona dicha base de datos en base a las peticiones que realizan los usuarios de grupo de la aplicación. De esta forma, el servicio REST ha sido estructurado en dos componentes: un componente `Controller`, que se encarga de procesar las diferentes peticiones que realicen los usuarios de grupo, y un componente DAO, que se encarga de interactuar con la base de datos MySQL. Asimismo, para que todo esto sea posible, se han combinado las siguientes tecnologías:

- Java Persistence API (JPA): es una API de persistencia desarrollada para la plataforma Java EE. Más concretamente, permite realizar la integración entre el sistema orientado a objetos y el sistema relacional de la base de datos abstrayendo casi en su totalidad al programador del proceso de conversión.
- Java Architecture for XML Binding (JAXB): es una API que permite realizar automáticamente el mapeo entre las clases de Java y el esquema XML/JSON. De esta forma, el servicio es capaz de representar las solicitudes y respuestas mediante objetos anotados por JAXB.
- Java API for RESTful Web Services (JAX-RS): es una API de Java que permite la creación de servicios web acorde con el estilo arquitectónico REST. De esta forma, JAX-RS proporciona notaciones para el mapeo entre una clase Java y un recurso web.

En la Figura 5.9 se muestra, a modo de ejemplo, la implementación de la petición GET correspondiente a la consulta de las deudas de un grupo.

```
@GET
@Path("/{idGrupo}/deudas")
@Produces({MediaType.APPLICATION_JSON, MediaType.APPLICATION_XML})
public Response getDeudas(@PathParam("idGrupo") long idGrupo, @Context UriInfo uriInfo)
{
    ResponseBuilder response;
    response= Response.status(Response.Status.NOT_FOUND);

    Grupo grupo= dao.getGrupo(idGrupo);
    if(grupo != null)
    {
        List<Deuda> deudas= grupo.getDeudas();
        if(deudas != null)
        {
            DeudasRepresentation deudasRep= new  DeudasRepresentation(deudas, uriInfo);
            response= Response.ok(deudasRep);
        }
    }

    return response.build();
}
```

Figura 5.9 Código Java correspondiente a la petición GET del recurso deudas.

Como se puede apreciar, el método incluye una serie de anotaciones. Estas anotaciones son proporcionadas por JAX-RS. A continuación, se describe cada una de las anotaciones que aparecen para lograr una mayor comprensión del código:

- @GET: define que el método implementará una operación GET.
- @Path: define la ruta de acceso relativa del método.
- @Produces: define los tipos de medios MIME que el método de recurso es capaz de producir y enviar al cliente. En este caso, el servicio REST es capaz de manejar representaciones de recursos tanto en formato XML como JSON.
- @PathParam: enlaza el parámetro a un segmento de ruta.
- @Context: define el contexto de la solicitud HTTP.

6. Pruebas

En este capítulo se describen las diferentes pruebas que se han llevado a cabo con el fin de evaluar el correcto funcionamiento del software. Las pruebas de software permiten detectar, y en su caso corregir, los fallos y comportamientos incorrectos que se presenten, así como verificar el cumplimiento de los requisitos definidos durante la fase de análisis y especificación. De esta forma, la fase de pruebas constituye una de las partes más importantes en el desarrollo de un proyecto ya que permite dotar a un proyecto software de una mayor calidad.

En este proyecto, se han realizado diversos tipos de prueba con el objetivo de cubrir todos los aspectos posibles y, de esta manera, maximizar la calidad del software desarrollado. A lo largo de este apartado, se describirá con mayor detalle los diferentes niveles de prueba realizados.

6.1. Pruebas unitarias

Las pruebas unitarias, también conocidas bajo el nombre de pruebas modulares, constituyen una forma de comprobar el correcto funcionamiento de cada una de las unidades software que componen el sistema. Cada unidad es probada de manera aislada al resto de los módulos de los que depende. De esta forma, podemos asegurar que cada unidad funciona correctamente por separado.

Es conveniente considerar que, para desarrollar pruebas unitarias con un cierto grado de calidad, éstas deben respetar los siguientes requisitos:

- Automatizables: se deben ejecutar de manera automática, sin requerir de la intervención manual.
- Completas: deben comprobar la mayor cantidad de código posible.
- Reutilizables: deben poder ser ejecutadas tantas veces como sea necesario y en cualquier momento.
- Independientes: deben poder ejecutarse sin depender del resto de pruebas.
- Profesionales: deben ser documentadas y realizadas de la forma más eficiente posible.

En este proyecto, se han realizado las pruebas unitarias necesarias para verificar el correcto funcionamiento tanto de la aplicación móvil como del servicio REST. Estas pruebas han sido desarrolladas haciendo uso del framework JUnit, que permite la realización de pruebas automatizadas en proyectos Java, y de la librería Mockito, utilizada para la creación de objetos Mock (objetos que permiten asegurar el correcto aislamiento de la unidad software que se está testeando mediante la simulación del comportamiento esperado del módulo al que suplantan).

En cuanto al servicio REST, se han realizado pruebas unitarias de los módulos pertenecientes al controlador. Es conveniente resaltar que, como todos los métodos del controlador hacen uso del componente DAO para desempeñar su funcionalidad, la utilización

de los anteriormente citados objetos Mock ha sido fundamental para el desarrollo de estas pruebas. Así pues, en la Figura 6.1, se muestra como ejemplo la prueba unitaria correspondiente al método `getGrupo` de la clase `FinanWalletController`.

```
@Test
public void testDeleteGrupoError() throws URISyntaxException
{
    long idGrupo= 1;
    Grupo grupo= new Grupo("Apartamento");
    grupo.setIdGrupo(idGrupo);

    IFinanWalletGrupoDAORemote dao = Mockito.mock(IFinanWalletGrupoDAORemote.class);
    Mockito.when(dao.getGrupo(idGrupo)).thenReturn(grupo);

    UriInfo uriInfo= Mockito.mock(UriInfo.class);
    Mockito.when(uriInfo.getAbsolutePath()).thenReturn(new URI("http://FinanWalletService/grupos/1"));

    FinanWalletController controller= new FinanWalletController(dao);
    Response response= controller.getGrupo(idGrupo, uriInfo);

    Assert.assertEquals(200, response.getStatus());
    Assert.assertEquals(grupo, response.getEntity());
}
```

Figura 6.1 Código prueba unitaria del método `getGrupo` de la clase `FinanWalletController`.

Como se puede observar, se han utilizado objetos Mock para simular, bajo un comportamiento esperado, los objetos `DAO` y `UriInfo`. De esta forma, conseguimos aislar totalmente el módulo que se está probando del resto para así poder comprobar el correcto funcionamiento del método encargado de responder a las solicitudes GET del recurso Grupo.

En lo que respecta a la aplicación móvil, se han realizado las pruebas unitarias de unidades software pertenecientes al modelo y al presentador. En la Figura 6.2, se muestra, a modo de ejemplo, la prueba unitaria correspondiente a la verificación del correcto funcionamiento del método `readTransaccion` de la clase `ParserJSON`.

```
@Test
public void testReadTransaccionTransfSuccess() throws Exception
{
    try
    {
        InputStream is = InstrumentationRegistry.getTargetContext().getResources().openRawResource(R.raw.transaccion_grupo_test);
        JsonReader reader = new JsonReader(new InputStreamReader(is, charsetName: "UTF-8"));
        Transaccion t = ParserJSON.readTransaccion(reader);

        Assert.assertEquals( expected: "transferencia", t.getTipoTransaccion());
        Assert.assertEquals( expected: "64", t.getIdTransaccion());
        Assert.assertEquals( expected: "Pago entradas de cine", t.getTitulo());
        Assert.assertEquals( expected: "22/7/2018", t.getFecha());
        Assert.assertEquals( expected: 7.5, t.getImporte());
        Assert.assertEquals(new Usuario( email: "pablo_gomez@hotmail.com", nombre: "Pablo Gomez"), t.getRealizador());
        Assert.assertEquals(new Usuario( email: "marta_fernandez@gmail.com", nombre: "Marta Fernandez"), ((Transferencia) t).getReceptor());
    }
    catch(Exception e)
    {
        Assert.fail("Error: lectura de transferencia incorrecta");
        e.printStackTrace();
    }
}
```

Figura 6.2 Código prueba unitaria del método `readTransaccion` de la clase `ParserJSON`.

Como se puede apreciar, la prueba unitaria emplea un recurso local, que simula una representación del estado de un recurso Transaccion del servicio REST, para así poder comprobar, de forma aislada, que el método `readTransaccion` transforma adecuadamente una representación en formato JSON a un objeto `Transaccion` correspondiente a la gestión de grupo.

6.2. Pruebas de integración

Una vez comprobado que todas las unidades software funcionan correctamente de forma aislada, se procede a la realización de las pruebas de integración. Las pruebas de integración son una forma de verificar que las interacciones entre los diferentes módulos software se llevan a cabo de la manera correcta. En otras palabras, las pruebas de integración permiten verificar el correcto funcionamiento de las diversas unidades software existentes, pero, a diferencia de las unitarias, cada unidad es probada con los módulos reales de los que depende y no mediante emulaciones.

Es oportuno mencionar que, las pruebas de integración se han realizado, siempre que haya sido posible, de manera progresiva. Es decir, se ha comenzado realizando pruebas sobre aquellos módulos que poseen un menor número de dependencias y se ha terminado por aquellos que tenían más dependencias. Esta metodología nos permite asegurar que cada componente testeado funciona adecuadamente ya que los anteriores han sido probados previamente.

En este proyecto, se han realizado pruebas de integración del servicio REST y de la aplicación móvil.

En lo referente al servicio REST, se ha utilizado la herramienta Postman para la realización de las pruebas de integración. Esta herramienta permite interactuar con la API del servicio REST desarrollado, así como escribir y ejecutar pruebas para cada solicitud utilizando el lenguaje JavaScript. Así pues, se han desarrollado pruebas automatizadas para cada método proporcionado por el servicio de manera que puedan ser ejecutadas en cualquier momento. En la Figura 6.3, se muestra la prueba de integración correspondiente a una petición HTTP que invoca el método GET para obtener información de una transacción determinada.

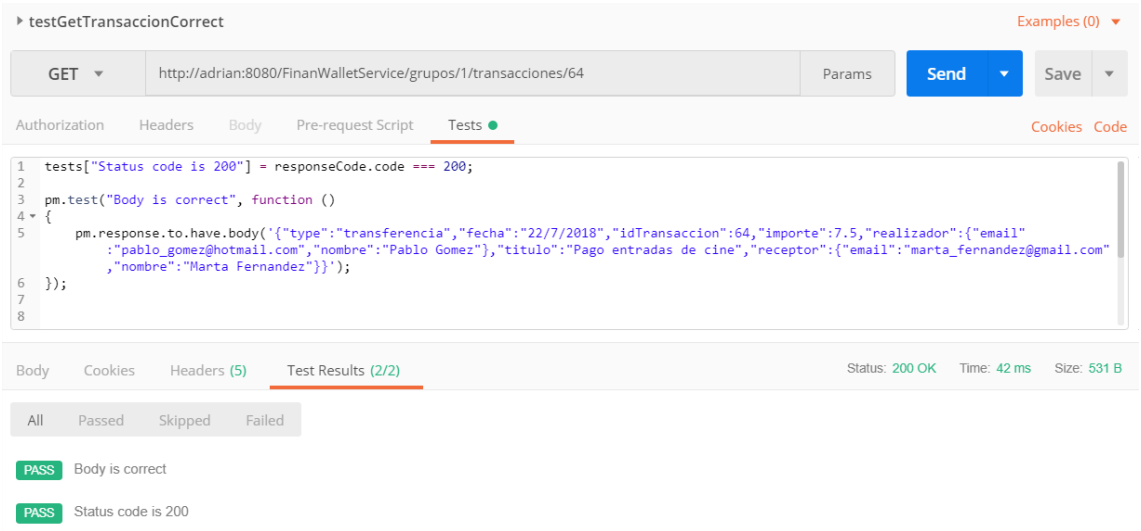

Figura 6.3 Código prueba integración del método GET a un recurso Transacción en la API REST.

En cuanto a la aplicación móvil, se ha hecho uso del framework de testing Espresso. Este framework, lanzado por Google, permite grabar las interacciones realizadas manualmente con un dispositivo de simulación para luego generar de manera automática la prueba

correspondiente y, de esta forma, poder ejecutar las diferentes interacciones efectuadas en cualquier momento y de manera automatizada. Es necesario detallar que, estas pruebas han sido desarrolladas bajo el contexto de que la base de datos de la aplicación móvil tiene que estar vacía y que el dispositivo de simulación utilizado debe ser un Nexus 5 API 25, el cual es proporcionado por Android Studio. Así pues, para ejecutar dichas pruebas correctamente, es necesario tener en cuenta estos requisitos.

6.3. Pruebas de sistema

Las pruebas de sistema son aquellas en las que se verifica el comportamiento del sistema final en todo su conjunto, es decir, se evalúan tanto los aspectos funcionales como los no funcionales en base a los requisitos especificados durante la fase de análisis.

Se han realizado pruebas de sistema para cada requisito establecido, verificando el comportamiento esperado para cada uno de los posibles escenarios de ejecución. A modo de ejemplo, la prueba de sistema que se ha desarrollado para testear la funcionalidad de unirse a un grupo ha sido probada en las siguientes situaciones posibles: “unirse a un grupo cuando no hay conexión a internet”, “unirse a un grupo cuando si hay conexión a internet”, “unirse a un grupo correctamente”, “unirse a un grupo incorrectamente” …

Este tipo de pruebas son ejecutadas en un ambiente de pruebas los más parejo posible al entorno final. Así pues, se han realizado diversas pruebas tanto en dispositivos reales como en emuladores, con diferentes versiones de Android y con distintos tamaños y resoluciones de pantalla. De esta forma, se ha conseguido comprobar el correcto funcionamiento del sistema en un conjunto considerable de dispositivos móviles.

6.4. Pruebas de aceptación

Las pruebas de aceptación son aquellas realizadas por los usuarios finales que desean la aplicación con el objetivo de verificar el correcto funcionamiento de la aplicación móvil y comprobar que el producto software satisface los requisitos especificados durante la fase de análisis. En otras palabras, este tipo de pruebas permiten determinar el grado de satisfacción de los usuarios finales con el producto software final.

Normalmente, las pruebas de aceptación no son realizadas por los desarrolladores, pero, en este caso, el desarrollador constituye uno de los usuarios finales de la aplicación y, por lo tanto, se ha encargado también de llevar a cabo este tipo de pruebas.

7. Conclusiones y trabajos futuros

Una vez finalizado el desarrollo del proyecto, es conveniente analizar el trabajo realizado y, en consecuencia, extraer una serie de conclusiones que permitan mejorar la ejecución de futuros proyectos, así como facilitar la planificación y ampliación del proyecto en cuestión.

De esta forma, el capítulo consta de dos secciones. Por un lado, en la primera sección, se describirán las conclusiones obtenidas tras concluir el desarrollo del proyecto y, por otro lado, en la segunda sección, se expondrán las futuras posibles mejoras o novedades que sufrirá la aplicación móvil en una segunda fase o posteriores de desarrollo.

7.1. Conclusiones

Este proyecto tiene como objetivo la creación de una aplicación móvil que permita facilitar al usuario la planificación de su presupuesto tanto a nivel personal como a nivel de grupo. Más concretamente, la aplicación pretende dar soporte a la gestión personal de las transacciones económicas y a la gestión de los gastos compartidos de un grupo de personas.

En lo que respecta a las especificaciones que fueron definidas en un principio, es necesario comentar que se ha conseguido implementar todas las funcionalidades, a excepción de los recordatorios que, por cuestiones de tiempo, fueron descartados al ser considerados una funcionalidad de menor importancia en comparación al resto. Es conveniente resaltar que, para el desarrollo de estas funcionalidades, se ha puesto especial atención en cumplir de forma cuidadosa con la metodología de desarrollo software especificada originalmente. De esta forma, al utilizar una metodología iterativa e incremental, ha sido posible separar la complejidad del producto en diversas iteraciones y, en consecuencia, al ser solo una persona, el proceso de desarrollo ha resultado ser asequible. Además, esta metodología, ha permitido dotar al proyecto de una mayor versatilidad ya que, ha permitido realizar mejoras necesarias en fases ya desarrolladas de la aplicación para solventar necesidades que han surgido durante el proceso de desarrollo. Un buen ejemplo de ello es la reorganización de ciertos elementos de la interfaz de usuario para conseguir que la aplicación sea más sencilla e intuitiva.

Una de las mayores complicaciones para abordar este proyecto ha sido la de realizar todas las configuraciones necesarias para la correcta integración del servicio REST, con el servidor de aplicaciones GlassFish y con el sistema gestor de base de datos MySQL. En este aspecto, también merece atención, para llevar a cabo el desarrollo del servicio REST, la realización de la configuración necesaria para la combinación de las tecnologías Java Persistence API (JPA), Java Architecture for XML Binding (JAXB) y Java API for RESTful Web Services (JAX-RS), así como el uso simultaneo de estas tecnologías. No obstante, se ha conseguido solventar exitosamente ambos problemas.

Desde un punto de vista más personal, este proyecto me ha servido para especializarme tanto en el desarrollo de aplicaciones Android como en el de servicios REST. En ambos casos, se partía de unos conocimientos muy básicos que, gracias a los conocimientos adquiridos a lo largo de los últimos cuatro años y a la documentación accesible para desarrolladores, han sido

ampliados considerablemente. Esto se debe también al uso de tecnologías, herramientas y librerías, totalmente desconocidas para mí, que han permitido, por mencionar algunos casos, integrar el servicio REST con una base de datos, almacenar de forma persistente pequeñas colecciones de pares clave-valor o mostrar ciertos elementos gráficos por pantalla. Así pues, este proyecto me ha servido para ganar experiencia como desarrollador y motivarme a seguir aprendiendo nuevos aspectos software.

7.2. Trabajos futuros

Este trabajo corresponde al desarrollo de una primera fase en la que se incluyen las funcionalidades básicas de la aplicación móvil.

Así pues, en una segunda fase o posteriores de desarrollo, la aplicación móvil sufrirá las siguientes futuras posibles mejoras o novedades:

- Sistema de mensajería (chat).
- Incorporación del rol de administrador en la gestión de grupos.
- Mejorar la seguridad tanto de la aplicación móvil como del servicio REST.
- Utilización de elementos más gráficos para la representación de los datos.
- Adaptación a las plataformas móviles iOS y Windows Phone.
- Inclusión de nuevos idiomas en la aplicación.
- Inclusión de nuevas divisas.
- Inclusión de diversos formatos de divisa.
- Permitir el almacenamiento de imágenes para cada transacción económica.
- Sincronización con los datos bancarios.
- Permitir el uso de notificaciones para informar sobre datos estadísticos.

Referencias

[1] PwC. https://www.pwc.es/es/sala-prensa/notas-prensa/2015/medios-pago-paisaje-movimiento.html

[2] Roa, B. A. Blogspot. http://isescom.blogspot.com/2013/08/desarrollo-en-cascada-vs-desarrollo.html

[3] Android. https://developer.android.com/guide/

[4] REST. https://www.codecademy.com/articles/what-is-rest

[5] SQLite. https://www.sqlite.org/about.html

[6] MySQL. https://dev.mysql.com/doc/refman/8.0/en/introduction.html

[7] Android Studio. https://developer.android.com/studio/intro/

[8] Eclipse. https://www.eclipse.org/

[9] StarUML. https://docs.staruml.io/

[10] MySQL Workbench. https://www.mysql.com/products/workbench/

[11] Postman. https://www.getpostman.com/

[12] GlassFish. https://javaee.github.io/glassfish/doc/5.0/release-notes.pdf

[13] NinjaMock. https://ninjamock.com/

[14] Jim Arlow & Ila Neustadt. UML 2 and the Unified Process. Addison Wesley. (2002).

[15] Alistair Cockburn. Writing Effective Use Cases. Pearson Professional. (2000)

Anexo: Navegabilidad de la aplicación

A.1 Introducción

Este anexo tiene por objeto mostrar la navegabilidad de la aplicación a través del uso de Mockups o maquetas digitales.

A.2 Mockups

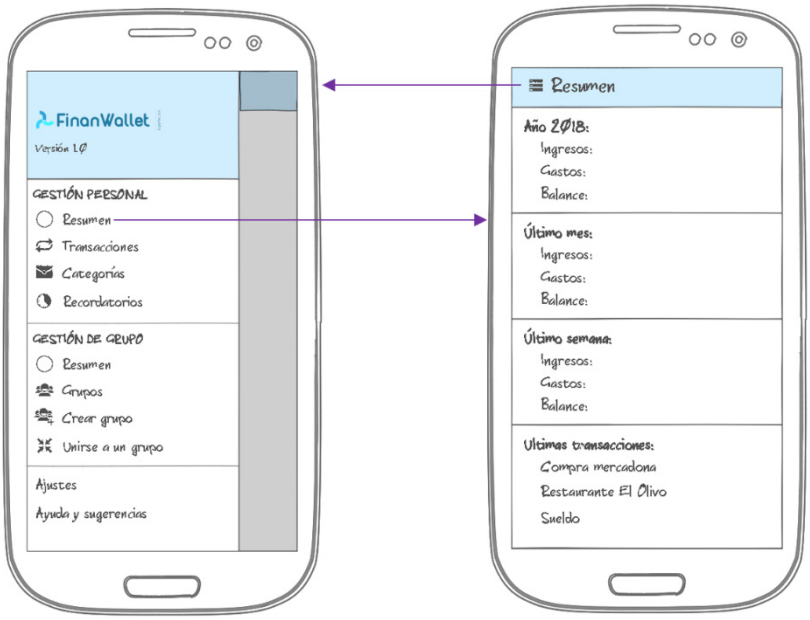

Figura A.1 Sección Resumen para la gestión personal de gastos.

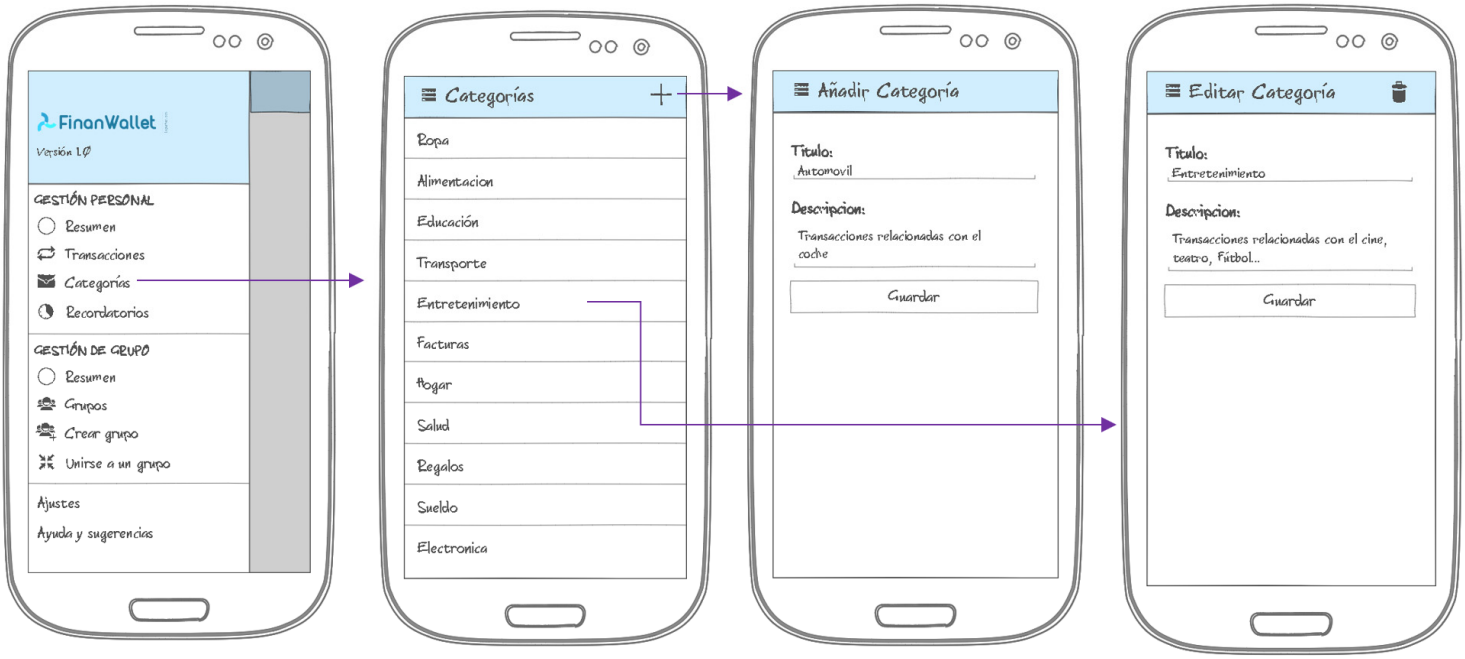

Figura A.2 Sección Categorías para la gestión personal de gastos.

Figura A.3 Sección Transacciones para la gestión personal de gastos.

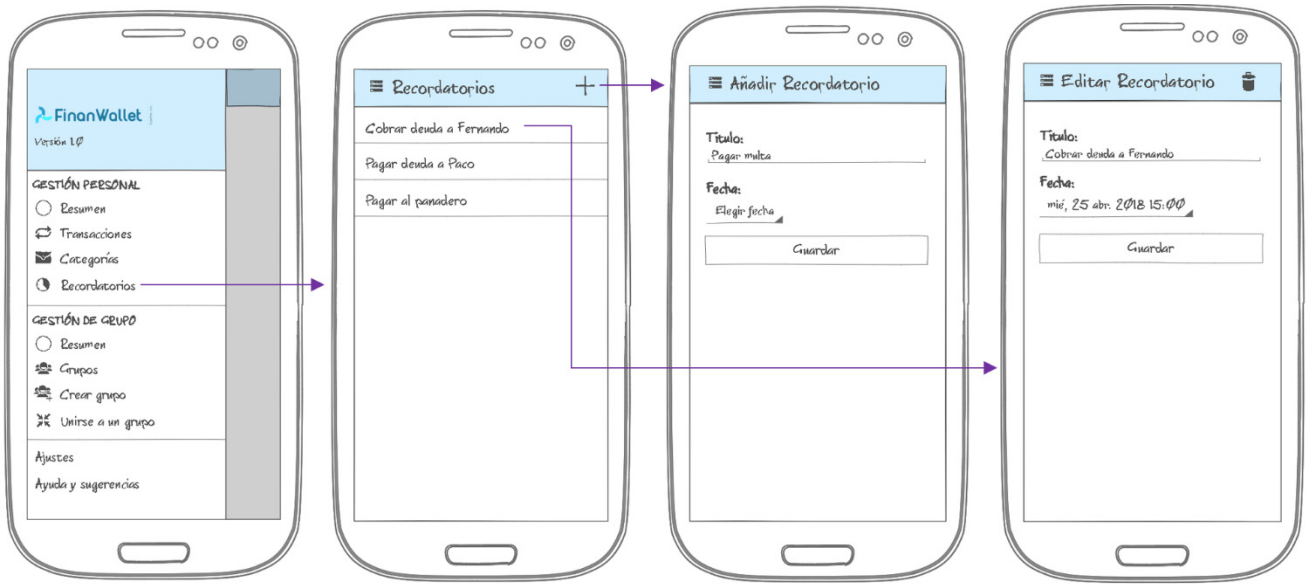

Figura A.4 Sección Recordatorios para la gestión personal de gastos.

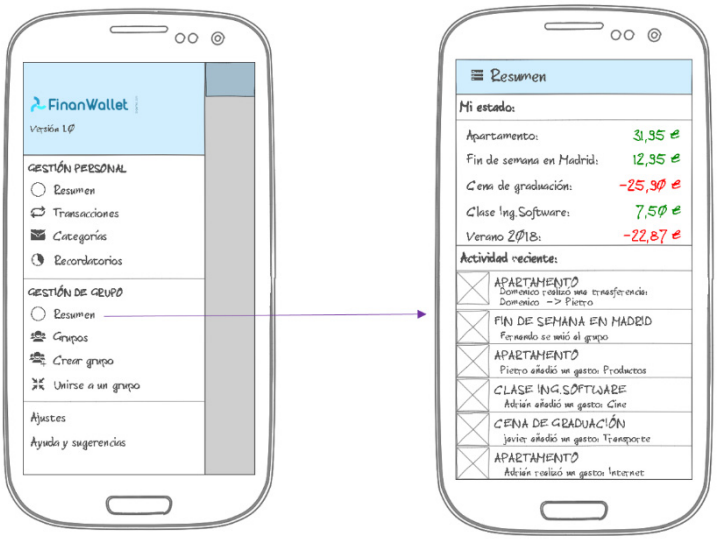

Figura A.5 Sección Resumen para la gestión de gastos de grupo.

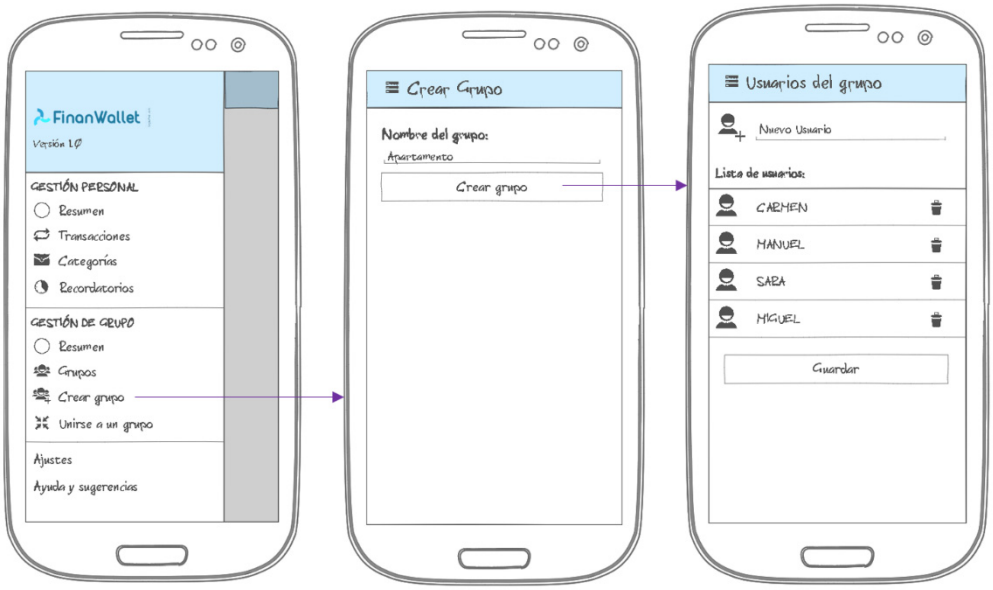

Figura A.6 Sección Crear Grupo para la gestión de gastos de grupo.

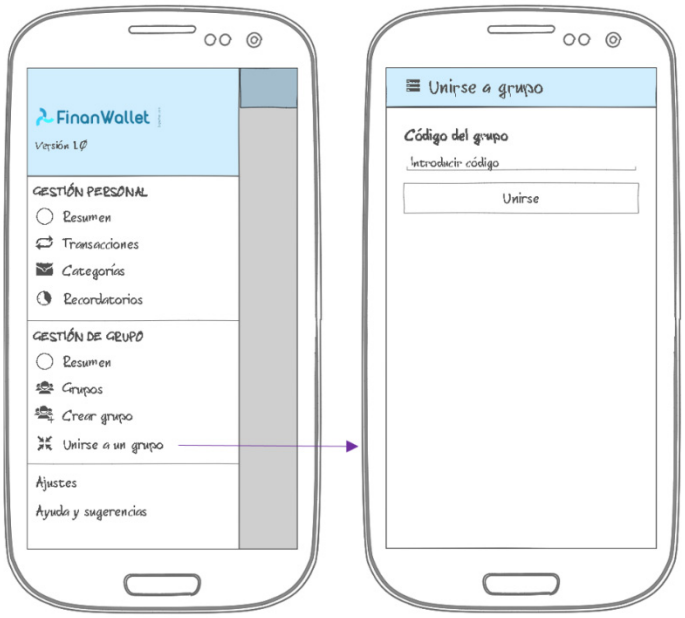

Figura A.7 Sección Unirse a un grupo para la gestión de gastos de grupo.

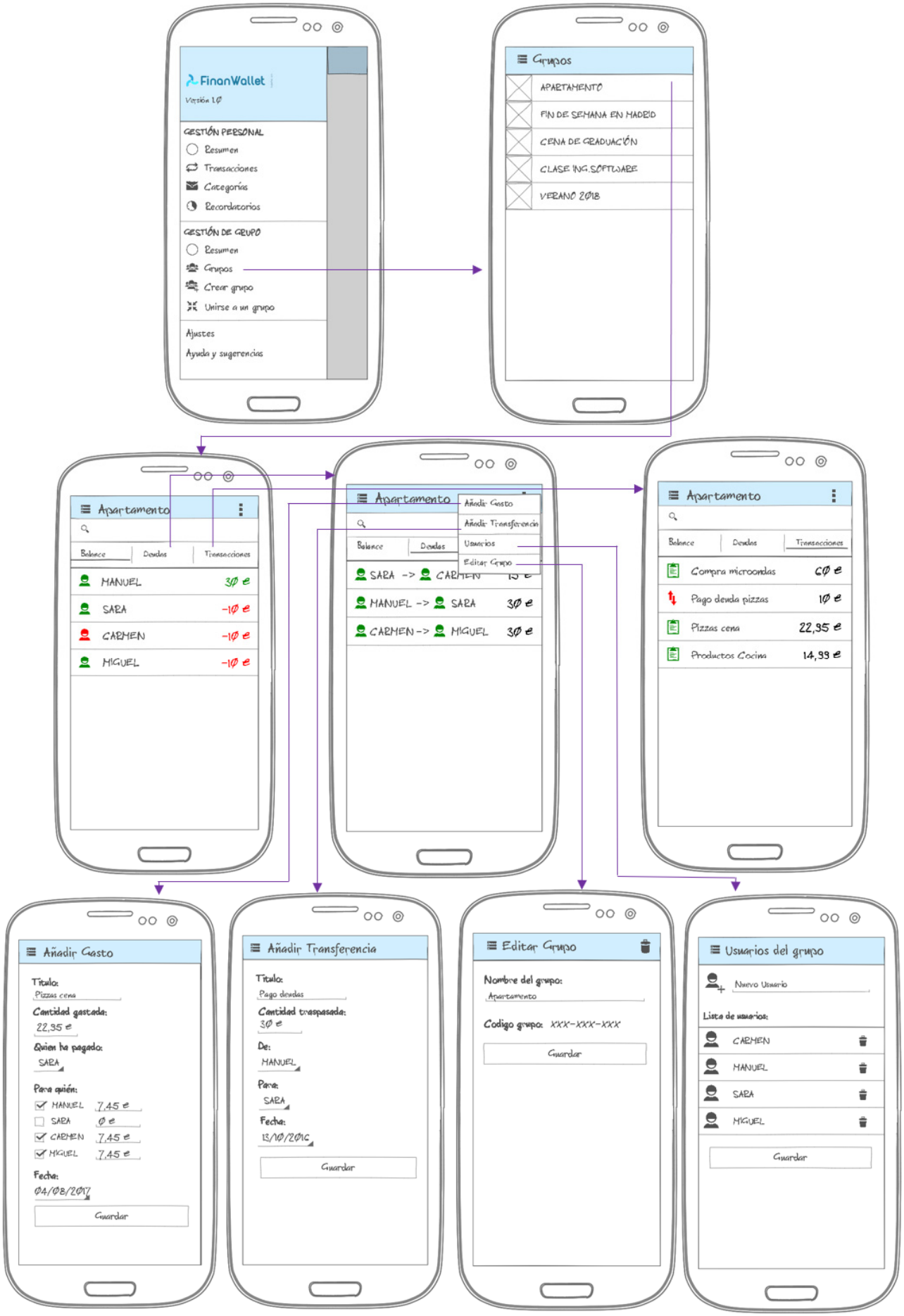

Figura A.8 Sección Grupos para la gestión de gastos de grupo.