UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

GimmeBack: Aplicación de gestión de gastos en grupo

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: García Cantero, Joaquín

Tutor/a: Canós Cerdá, José Hilario

CURSO ACADÉMICO: 2022/2023

Resumen

Este trabajo de fin de grado se centra en el diseño y desarrollo de una aplicación móvil para resolver los problemas que surgen al compartir un piso, tales como la organización de las compras, la distribución de tareas y deudas entre los convivientes. Para ello, se ha empleado la metodología de desarrollo ágil y tecnologías como Flutter y Node.js. Los resultados obtenidos incluyen una aplicación móvil funcional e intuitiva para la gestión de la convivencia en un piso, que puede ser utilizada por compañeros de piso y en viajes con amigos o familiares. Las conclusiones del trabajo destacan la importancia de contar con herramientas adecuadas para facilitar la convivencia y la necesidad de seguir explorando nuevas tecnologías para solucionar problemas cotidianos.

Palabras clave: aplicación móvil, gestión, Flutter, Node.js.

Resum

Aquest treball de fi de grau es centra en el disseny i desenvolupament d'una aplicació mòbil per a resoldre els problemes que sorgeixen al compartir un pis, com ara l'organització de les compres, la distribució de tasques i deutes entre els convivents. Per a açò, s'hi ha empleat la metodologia de desenvolupament àgil i tecnologies com Flutter i Node.js. Els resultats obtinguts inclouen una aplicació mòbil funcional i intuïtiva per a la gestió de la convivència en un pis, que pot ser utilitzada per companys de pis i en viatges amb amics o familiars. Les conclusions del treball destaquen la importància de comptar amb eines adequades per a facilitar la convivència i la necessitat de seguir explorant noves tecnologies per a solucionar problemes quotidians.

Paraules clau: aplicació mòbil, gestió, Flutter, Node.js.

Abstract

This undergraduate thesis focuses on the design and development of a mobile application to solve problems that arise when sharing a flat, such as organizing purchases, distributing tasks, and debts among the flatmates. To achieve this, an agile development methodology and technologies such as Flutter and Node.js have been employed. The results include a functional and intuitive mobile application for managing flat-sharing, which can be used by flatmates as well as on trips with friends or family. The conclusions of the work emphasize the importance of having appropriate tools to facilitate cohabitation and the need to continue exploring new technologies to solve everyday problems.

Keywords: mobile application, managing, Flutter, Node.js.

Tabla de contenidos

1. Introducción 9
   1.1 Motivación 9
   1.2 Objetivos 10
   1.3 Metodología 10
      1.3.1 Kanban 10
   1.4 Estructura del trabajo 12
2. Estado del arte 13
   2.1 Tecnologías de desarrollo de aplicaciones móviles 13
      2.1.1 Cordova – Apache 13
      2.1.2 React Native - Facebook 14
      2.1.3 Flutter – Google 14
3. Análisis y Diseño 15
   3.1 Arquitectura del sistema 15
   3.2 Análisis de requisitos 17
      3.2.1 Diagrama de contexto 17
      3.2.2 Diagrama de casos de uso 18
      3.2.3 Casos de uso 21
      3.2.4 Diagrama de clases 24
   3.3 Diseño de la base de datos 26
      3.3.1 Diseño conceptual 27
      3.3.2 Diseño lógico 28
   3.4 Aspecto de la aplicación 30
   3.5 Estructura de la aplicación 31
4. Desarrollo 39
   4.1 Tecnologías utilizadas 39
      4.1.1 Flutter 39
      4.1.2 Android Studio 39
      4.1.3 GetStorage 39
      4.1.4 Node.js y Express.js 40
      4.1.5 Visual Studio Code 40

4.1.6 Passport, Passport-JWT y Jsonwebtoken 40
4.1.7 Bcrypt 41
4.1.8 Postman 41
4.1.9 MySQL y MySQL Workbench 42
4.1.10 Firebase 42
4.1.11 Railway 42
4.2 Base de datos – Diseño físico 43
4.3 Cliente – Flutter 43
4.4 Servidor – Node.js y Express.js 46
4.5 Gestor de imágenes 50
4.6 Gestor de notificaciones 50
4.6.1 Modelo de datos 50
4.6.2 Servidor 51
4.6.3 Cliente 51
5. Pruebas y despliegue 53
5.1 Pruebas de aceptación 53
5.2 Pruebas unitarias - Cliente 53
5.3 API Testing/Pruebas de caja negra – Servidor 54
5.4 Pruebas de componentes 54
5.5 Pruebas de integración y sistema 56
5.6 Pruebas de compatibilidad e interfaz gráfica 56
5.7 Pruebas y cuestionario de usabilidad 57
5.8 Despliegue 58
6. Conclusiones 61
Bibliografía 63
Anexo A 65
Anexo B 67
Anexo C 74
Anexo D 77

Figura 1 Ejemplo de tablero Kanban 11
Figura 2 Arquitectura Cliente-Servidor. 15
Figura 3 Diagrama de contexto 18
Figura 4 Diagrama de casos de uso usuario no autenticado 19
Figura 5 Diagrama de casos de uso usuario autenticado 20
Figura 6 Diagrama de clases 26
Figura 7 Diagrama entidad-relación 28
Figura 8 paleta de colores de la aplicación. Fuente: paletasdecolores.com 30
Figura 9 Prototipos página principal usuario 31
Figura 10 Diagrama de navegación 32
Figura 11 Widget topBar 33
Figura 12 Ventana emergente crear/unir grupo 33
Figura 13 Vista inicio sesión 34
Figura 14 Vista registro 34
Figura 15 Vista grupos 35
Figura 16 Vista deudas 35
Figura 17 Vista ajustes 36
Figura 18 Vista gastos 36
Figura 19 Vista tareas 37
Figura 20 Vista ajustes grupo 37
Figura 21 Vista deudas grupo 37
Figura 22 Ejemplo modelo User 45
Figura 23 Ejemplo estructura pages 45
Figura 24 Estructura providers 46
Figura 25 Ejemplo solicitud devolver todas las deudas de un gasto. Archivo: debtController.js 48
Figura 26 Ejemplo consulta encontrar deudas por id del gasto. Archivo: debt.js 49
Figura 27 Ejemplo endpoint encontrar deudas por id del gasto. Archivo debtRoutes.js 49
Figura 28 Error al registrarse 55
Figura 29 Error al iniciar sesión **Figura 30** Error al unirse a un grupo 55

1. Introducción

La gran subida en los precios de alquiler en estos últimos años está suponiendo un problema para toda una generación de jóvenes y adultos, desde estudiantes que viajan a otros pueblos y ciudades para completar su formación, hasta trabajadores que se encuentran ya totalmente independizados de sus familias. En la Comunitat Valenciana, por ejemplo, encontramos un incremento interanual del 17,5% [1], situándose a la cabeza de la lista como la comunidad que más ha sufrido esta subida. Todas estas personas se han visto en la situación de no ser capaces de afrontar un alquiler individual, o en el caso de estudiantes universitarios el coste de una residencia, encontrando el compartir piso como la solución a este problema.

Compartir piso no es una tarea sencilla, ya sea con amigos o con completos desconocidos siempre hay problemas de convivencia que derivan en discusiones. Por ello, cualquier herramienta que nos pueda facilitar dicha convivencia siempre es bien recibida. Dados los avances tecnológicos, así como la creciente aceptación de estos cada vez más en nuestras vidas, disponemos de estas herramientas al alcance de nuestra mano gracias al teléfono móvil y la gran cantidad de aplicaciones que se crean con el avance de los años.

1.1 Motivación

El desarrollo de este proyecto consistirá en una herramienta que podrá facilitar la gestión del problema expuesto, ya que actualmente, y bajo mi experiencia personal de todos estos años compartiendo piso, no llegué a encontrar ninguna herramienta que cumpliese con todas las necesidades que nos íbamos encontrando mis compañeros y yo. Tras una distendida charla con ellos consideré como una opción desarrollar mi propia aplicación. Otra motivación importante para la realización de este trabajo de fin de grado es que ofrece la posibilidad de aprender dos tecnologías que siempre me han interesado: Flutter, un nuevo kit de desarrollo para aplicaciones móviles desarrollado por Google, y Node.js, un entorno de ejecución de JavaScript orientado a eventos asíncronos.

1.2 Objetivos

Se plantea el diseño y desarrollo de una aplicación móvil que busca solucionar algunos de los problemas asociados al compartir piso, tales como la organización de la compra, la repartición de tareas y deudas entre los convivientes. La aplicación permitirá a cada usuario agregar los gastos que haya costeado y repartirlos, crear tareas indicando su urgencia, comprar elementos necesarios o limpiar la cocina, entre otras funcionalidades. Aunque la aplicación está principalmente orientada para su uso entre compañeros de piso, también puede ser utilizada en viajes con amigos o familiares. Se busca que la aplicación sea intuitiva y fácil de usar, optimizando el tiempo del usuario.

1.3 Metodología

Las metodologías de desarrollo de software son esenciales para estructurar, planificar y controlar todo el proceso de desarrollo de un sistema. Trabajar con una metodología es fundamental para gestionar de manera eficiente los recursos disponibles y optimizar el proceso de desarrollo, ahorrando tiempo y costos a futuro. En este proyecto se seguirá una metodología ágil, debido a la flexibilidad que ofrece y la rápida respuesta para adaptar el proyecto a las dificultades que puedan surgir durante el desarrollo. Entre las diversas metodologías ágiles disponibles, se ha elegido el método Kanban para este proyecto. Este método destaca por su enfoque visual y por permitir la gestión eficiente de tareas y recursos, lo que permitirá avanzar de manera ordenada y eficaz en el desarrollo de la aplicación.

1.3.1 Kanban

La metodología Kanban se implementa por medio de tableros Kanban. Se trata de un método visual de gestión de proyectos que permite a los equipos visualizar sus flujos de trabajo y la carga de trabajo [2]. Como se puede observar en la figura 1, el trabajo se muestra en un tablero organizado por columnas; cada columna corresponde

a un estado del flujo de las tareas y cada tarea está representada por una tarjeta visual que irá desplazándose por las distintas columnas del tablero.

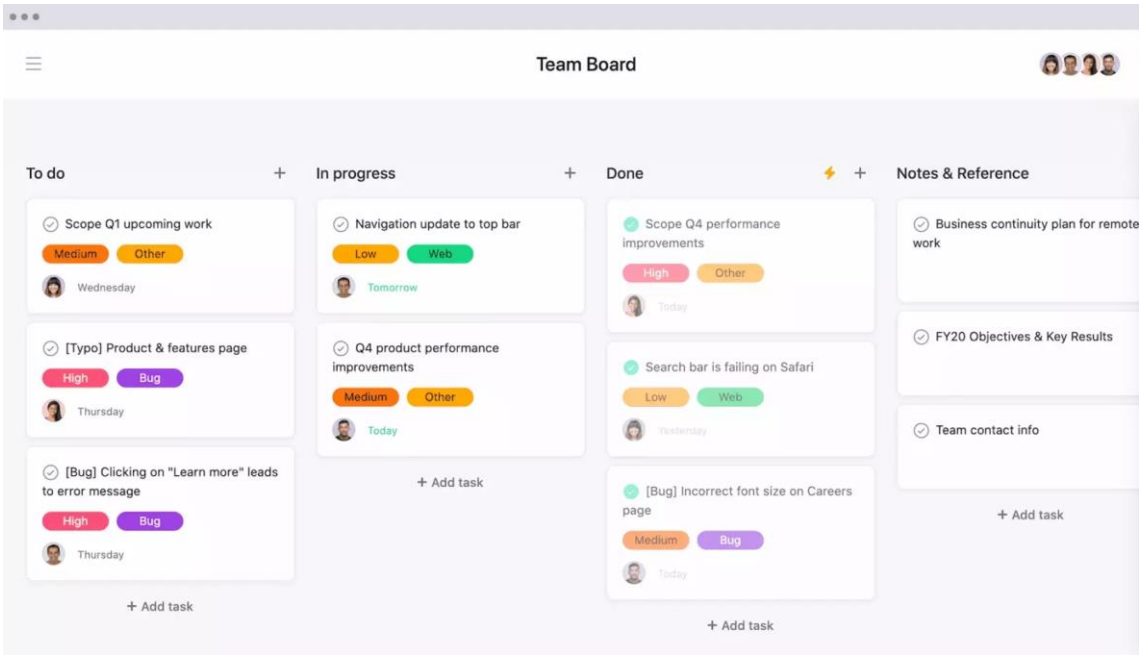

Figura 1 Ejemplo de tablero Kanban

Como herramienta para seguir el método Kanban se utilizará Trello, un entorno visual que permite a los equipos gestionar cualquier tipo de proyecto y flujo de trabajo, así como supervisar tareas [3]. Para empezar en Trello se ha creado un tablero con el nombre del proyecto, dentro del cual se crean las listas, que corresponden a las columnas comentadas anteriormente, donde se indicarán las diferentes fases por las que pasará una tarea. Cada tarea será representada por tarjetas que incluirán toda la información necesaria para realizar el trabajo.

En Trello hemos dividido el flujo de trabajo en 5 listas:

- **BackLog:** Todas las tareas que se espera realizar en este proyecto.
- **En proceso:** Las tareas que se están realizando en este momento.
- **Pasar Pruebas de aceptación:** Tareas que parecen terminadas y están pendientes de pasar las pruebas de aceptación.
- **Revisar/En Espera:** Tareas que han fallado las pruebas de aceptación y hay que arreglar. Una vez corregidas vuelven a la columna anterior.

- **Finalizado:** Tareas que han pasado las pruebas de aceptación correctamente y se pueden dar por finalizadas.

En cuanto a las tarjetas, dentro encontraremos una breve descripción de la tarea una, fecha estimada de vencimiento y las pruebas de aceptación.

1.4 Estructura del trabajo

La memoria consta de los siguientes capítulos:

- **Estado del arte:** En este capítulo se realiza una investigación sobre las tecnologías disponibles para el desarrollo móvil, además de un estudio de mercado sobre posibles aplicaciones móviles que desempeñan una función similar.
- **Análisis y diseño:** En este capítulo se realizan el análisis de requisitos de la aplicación, el diseño de la base de datos, la explicación del diseño escogido para el desarrollo de la aplicación y de los componentes que la componen, las tecnologías utilizadas y por qué fueron escogidas, y la arquitectura que se siguió para crear el sistema.
- **Desarrollo:** En este capítulo se explica todo el proceso que se ha seguido para el desarrollo de la aplicación, además de las decisiones que se han ido tomando.
- **Pruebas:** Se explican las pruebas realizadas sobre la aplicación para comprobar su correcto funcionamiento.
- **Conclusiones y trabajo futuro:** Las conclusiones sobre el trabajo realizado y como se vivió a nivel persona.

2. Estado del arte

2.1 Tecnologías de desarrollo de aplicaciones móviles

La industria del desarrollo de aplicaciones móviles se ha centrado en dos tecnologías principales: Android Studio con Java o Kotlin [4] para desarrollar aplicaciones en Android, y XCode con Swift para desarrollar aplicaciones en iOS [5]. Sin embargo, para desarrollar una aplicación para ambos sistemas operativos, se necesita escribir dos códigos fuente diferentes para lograr el mismo objetivo. Esto implica una gran cantidad de tiempo, lo que se vuelve cada vez más costoso debido a las constantes actualizaciones y la aparición de nuevas tecnologías, así como a los plazos de desarrollo apremiantes que enfrentan los desarrolladores de aplicaciones móviles.

Por suerte, han surgido alternativas para abordar este problema en los últimos años, lo que permite el desarrollo de aplicaciones para ambos sistemas operativos sin tener que escribir dos códigos fuente distintos. Estas alternativas se centran en dos enfoques principales: crear una aplicación web integrada en una aplicación móvil, lo que da la impresión de ser una aplicación nativa, o bien, generar el código nativo para cada sistema operativo a partir de una única fuente de código.

2.1.1 Cordova – Apache

Apache Cordova es un framework de desarrollo móvil de código abierto que utiliza tecnologías web estándar como HTML5, CSS3 y JavaScript para el desarrollo multiplataforma [6]. Con esta tecnología, es posible compilar un único código web como si fuera una aplicación nativa para móviles, lo que significa que se puede tener aplicaciones para diferentes plataformas móviles con un único desarrollo web. Sin embargo, al no generar ningún código nativo, tarda un poco más en procesarse y no suele contar con soporte para efectos visuales específicos de cada sistema operativo.

2.1.2 React Native - Facebook

React Native es un framework de JavaScript que permite desarrollar aplicaciones móviles nativas para iOS y Android. Se basa en React, una biblioteca de JavaScript de Facebook utilizada para crear interfaces de usuario, pero en lugar de dirigirse al navegador, se enfoca en las plataformas móviles. En resumen, los desarrolladores web pueden escribir aplicaciones móviles con una apariencia y sensación "nativa" [7]. Entre sus ventajas se encuentra una experiencia de usuario cercana a la nativa, lo que dificulta diferenciarla de una aplicación nativa real, además de utilizar JavaScript, uno de los lenguajes más comunes, lo que significa que los desarrolladores no tienen que aprender un lenguaje nuevo para desarrollar aplicaciones móviles. No obstante, debido a que es una tecnología relativamente nueva y aún está en desarrollo, puede requerir código en el lenguaje nativo de la plataforma para implementar algunas funciones específicas.

2.1.3 Flutter – Google

Flutter es un framework de código abierto desarrollado por Google que permite la creación de aplicaciones nativas y multiplataforma a partir de un único código base [8]. Su principal ventaja es la rapidez en el desarrollo, gracias a la disponibilidad de widgets preconstruidos y personalizables que reducen la necesidad de crear elementos desde cero. Además, como sucede con React Native, Flutter ofrece una experiencia cercana a la nativa.

No obstante, para utilizar Flutter se requiere aprender un nuevo lenguaje de programación llamado Dart, aunque es un lenguaje sencillo que se puede dominar con facilidad, y cuenta con una amplia comunidad que puede ayudar a resolver problemas.

En última instancia, se optó por Flutter por su capacidad de generar aplicaciones para ambos sistemas operativos a partir de un único código, y por su capacidad de producir una aplicación final con una calidad casi indistinguible de una aplicación nativa. Aunque implica aprender un nuevo lenguaje, Dart no es difícil de aprender y la comunidad de apoyo es amplia.

3. Análisis y Diseño

3.1 Arquitectura del sistema

La estructura de la aplicación se basará en una arquitectura cliente/servidor [9]. Como se puede ver en la Figura 2, esta arquitectura divide el sistema en dos componentes:

- **Cliente:** es el encargado de solicitar los servicios y usar la información provista por el servidor para sus propios fines.

- **Servidor:** es el encargado de ofrecer los servicios y se mantiene a la espera de peticiones para ejecutarlos o proporcionarlos.

De esta manera podemos observar que hasta que el cliente no solicita nada al servidor, este último no puede enviar o empezar procesos. Considerando por ello al cliente como un proceso desencadenante y al servidor como uno reactivo.

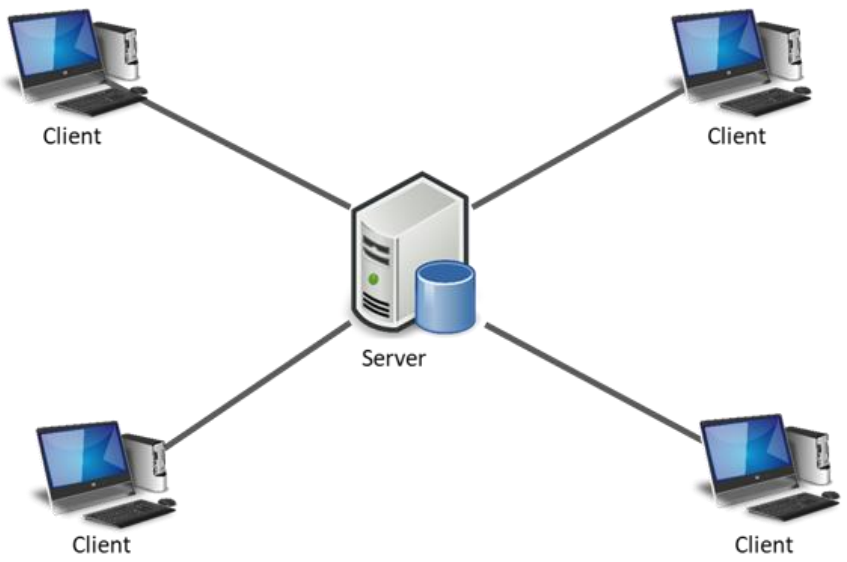

Figura 2 Arquitectura Cliente-Servidor.

Como principal característica de esta arquitectura se debe destacar la separación de responsabilidades clara, ya que estos son desarrollados como dos aplicaciones totalmente diferentes de tal manera que cada una puede ser desarrollada de forma independiente y pueden ser construidas en distintas tecnologías. El servidor será la única entidad encargada de acceder a los datos y ofrecerá sus servicios solo a los clientes que el confía, con esto se consigue proteger la información y la lógica detrás de todo el procesamiento de datos. El cliente en cambio será solo un visor de estos y delega todas las operaciones pesadas en el servidor liberándose así de una gran carga.

Es importante agregar que a pesar de poder construir el cliente y el servidor en distintas tecnologías deben respetar el mismo protocolo de comunicación. En este caso el escogido es el protocolo HTTP (*HyperText Transfer Protocol*), este es el en que se basa la red informática mundial (*WWW*) [10].

Además, toda la información enviada entre cliente y servidor estará en formato JSON (*JavaScript Object Notation*) [11]. Este es un formato ligero de intercambio de datos siendo el más usado porque leerlo y escribirlo es muy simple para las personas, mientras que para los sistemas es simple interpretarlo y generarlo.

En la arquitectura de la aplicación, Flutter se encarga de la parte del cliente, Node.js es el servidor y MySQL es la base de datos. A continuación, describo un esquema de la arquitectura de la aplicación:

- **Cliente:** la parte de la aplicación desarrollada en Flutter que se ejecuta en el dispositivo móvil del usuario. Se encarga de la interfaz de usuario y de la comunicación con el servidor mediante API REST.
- **Servidor:** la parte de la aplicación desarrollada en Node.js que se ejecuta en el servidor y se encarga de recibir solicitudes del cliente, procesarlas y enviar las respuestas correspondientes. También se comunica con la base de datos para almacenar y recuperar información.
- **Base de datos:** la parte de la aplicación que se encarga de almacenar y gestionar la información de la aplicación. En este caso, MySQL será la base de datos utilizada.

En resumen, la aplicación móvil desarrollada en Flutter enviará solicitudes al servidor Node.js a través de API REST, el servidor procesará las solicitudes, se comunicará con la base de datos MySQL para almacenar o recuperar información y enviará la respuesta correspondiente al cliente en Flutter.

3.2 Análisis de requisitos

En este apartado se van a desarrollar los principales modelos del dominio a partir de los cuales extraer los requisitos de nuestra aplicación. Para ello, en primer lugar, se generarán los diagramas de casos de uso para los distintos actores que participan, después se explicará más en detalle cada caso de uso y para terminar se mostrará la estructura del sistema con un diagrama de clases.

La representación de los distintos diagramas se basará en el estándar del Lenguaje de Modelado Unificado o UML (*Unified Modeling Language* en inglés). *"UML es un lenguaje que fue creado para forjar un lenguaje de modelado visual común y semántica y sintácticamente rico para la arquitectura, el diseño y la implementación de sistemas de software complejos, tanto en estructura como en comportamiento"* [12].

3.2.1 Diagrama de contexto

El diagrama de contexto es utilizado para definir el entorno del sistema, es decir, los límites del software que se va a desarrollar, en este caso, la aplicación GimmeBack. Una vez definido el sistema, se procede a identificar los actores externos que van a interactuar con él. Estos actores pueden ser entidades humanas, otros dispositivos o módulos de desarrollo software que quedan fuera del sistema. En la Figura 3 se muestran los cuatro actores externos identificados: usuario no autenticado, usuario autenticado, gestor de notificaciones y gestor de imágenes. Cada uno de estos actores interactúa de una manera específica con el sistema, lo que permite definir las funcionalidades y características que se deben incluir en el desarrollo de la aplicación.

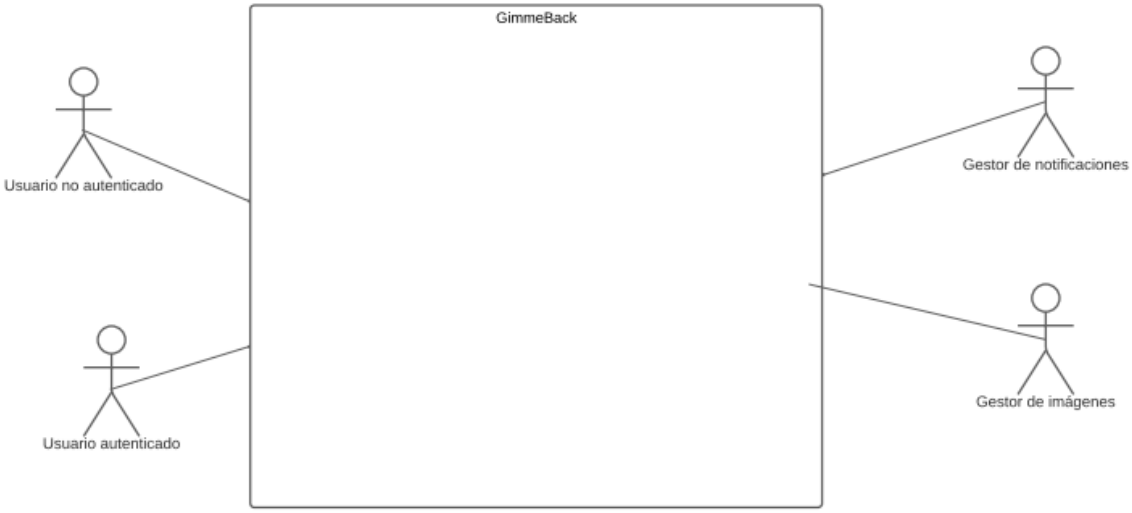

Figura 3 Diagrama de contexto

3.2.2 Diagrama de casos de uso

Una vez identificado el entorno del sistema y sus actores externos se pasan a definir los casos de uso, que son el conjunto de requisitos funcionales o funcionalidad que el sistema proporciona a los actores externos. Para indicar que interacción es la que tienen los actores con cada caso de uso se utilizan las líneas de comunicación. Entre los casos de uso puede haber relaciones de inclusión, cuando se realice el primero siempre se realizará el segundo, o extensión, cuando se realice el primero el segundo solo se realizará si cumple una condición. Más adelante se explicará con algún ejemplo.

Para empezar, se va a representar al usuario que no se encuentra registrado y/o autenticado. En este caso se ha especificado que el comportamiento del usuario se vea limitado a poder registrarse e identificarse en la aplicación, puesto que las principales funcionalidades de la aplicación están basadas en la interacción entre personas, por lo que se descarta el uso del sistema sin una previa autenticación.

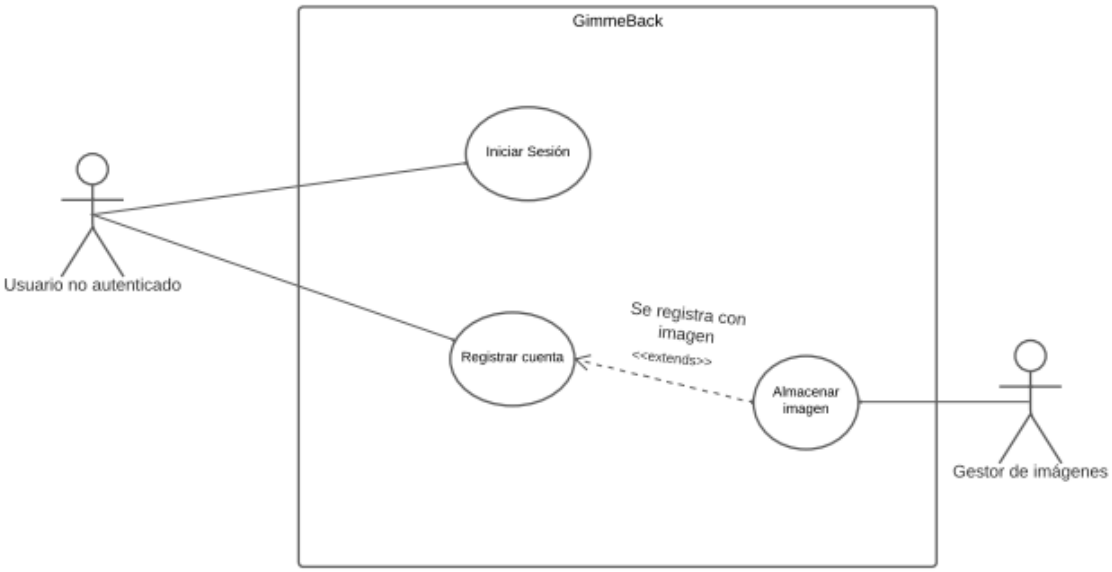

Figura 4 Diagrama de casos de uso usuario no autenticado

Como se puede observar en la Figura 4, aparece una relación de extensión, esto quiere decir que el actor gestor de imágenes solo realizará la funcionalidad *Almacenar imagen* cuando el usuario no autenticado se registre con una.

A continuación, vamos a representar al usuario autenticado con todas las funcionalidades, y al gestor de notificaciones encargado de las notificaciones push que recibirá el usuario.

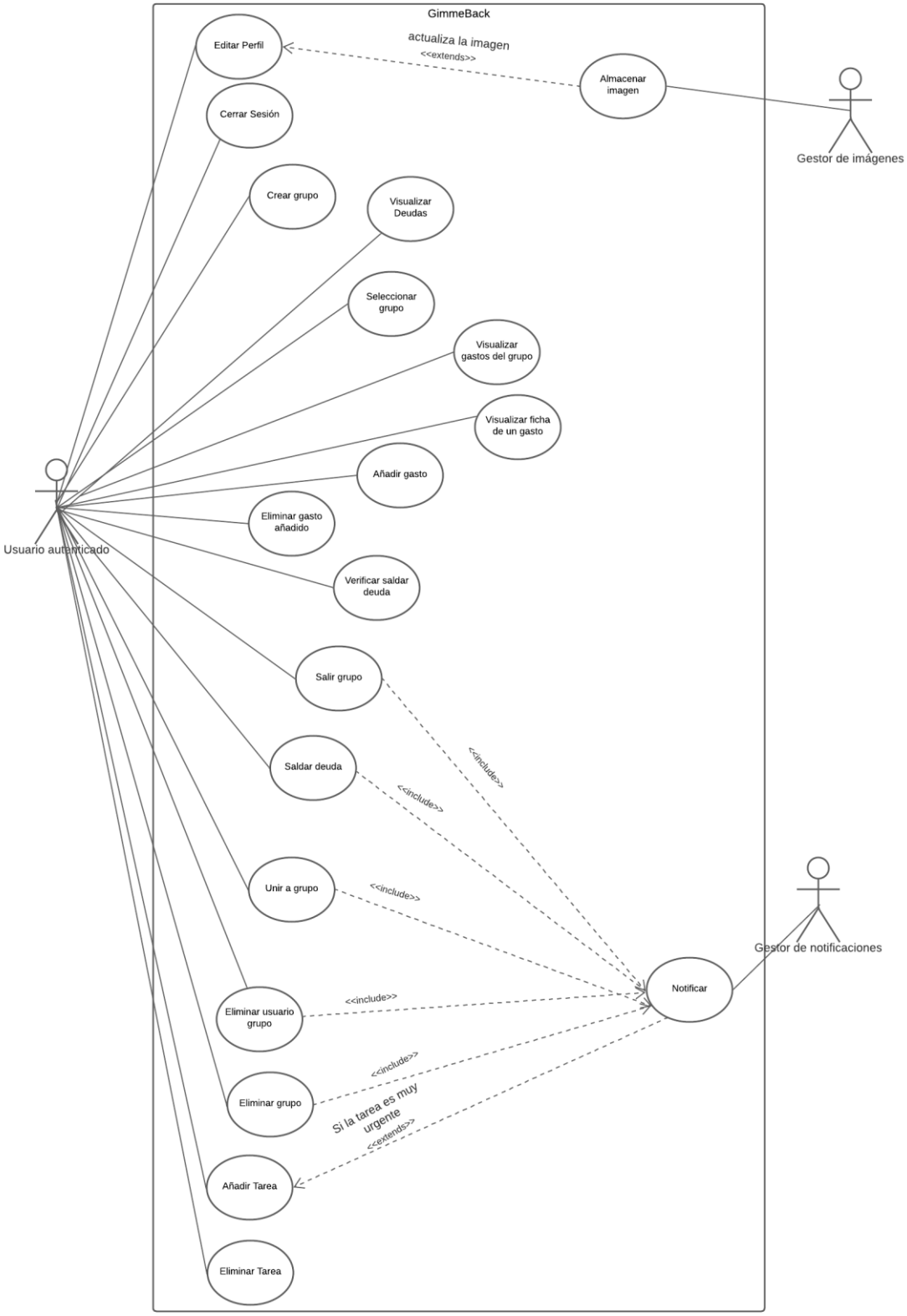

Figura 5 Diagrama de casos de uso usuario autenticado

En este caso como vemos en la Figura 5, notificar tiene varias relaciones de inclusión, por tanto esta se realizará siempre como parte de las funcionalidades con las que tiene dicha relación.

3.2.3 Casos de uso

Realizado el diagrama de casos de uso, se procede a describir los distintos casos de uso que componen la aplicación de una manera más detallada.

Referencia	CU01
Nombre	Registrar cuenta
Descripción	El usuario introduce los datos solicitados. Una vez finalizado el registro, el usuario será redirigido a la pantalla principal de la aplicación.

Referencia	CU02
Nombre	Iniciar Sesión
Descripción	El usuario introduce los datos solicitados. Una vez introducidos, el usuario podrá iniciar sesión a su cuenta y será redirigido a la pantalla principal de la aplicación.

Referencia	CU03
Nombre	Editar perfil
Descripción	El usuario cambia algunos datos que fueron introducidos durante su registro como el nombre y la imagen, por ejemplo.

Referencia	CU04
Nombre	Crear grupo
Descripción	El usuario crea un grupo introduciendo los datos solicitados y será redirigido a la pantalla de inicio del grupo.

Referencia	CU05
Nombre	Cerrar sesión
Descripción	El usuario solicita cerrar su sesión actual y vuelve a la pantalla de inicio de sesión.

Referencia	CU06
Nombre	Visualizar deudas
Descripción	El usuario ve sus distintas deudas a recibir o a dar en cada uno de los grupos.

Referencia	CU07
Nombre	Seleccionar grupo
Descripción	El usuario selecciona cualquiera de los grupos en los que participa para acceder a él.

Referencia	CU08
Nombre	Visualizar gastos grupo
Descripción	El usuario dentro del grupo visualiza los gastos añadidos por los integrantes de este con su nombre y precio.

Referencia	CU09
Nombre	Visualizar ficha de un gasto
Descripción	El usuario dentro del grupo selecciona un gasto y accede a sus detalles.

Referencia	CU10
Nombre	Añadir gasto
Descripción	El usuario introduce los datos solicitados. Una vez introducidos el usuario añadirá el gasto al grupo y será redirigido a la pantalla de inicio del grupo

Referencia	CU11

Nombre	Eliminar gasto añadido
Descripción	El usuario elimina un gasto que ha sido añadido por él mismo.

Referencia	CU12
Nombre	Verificar saldar deuda
Descripción	El usuario verifica que su deuda con otro usuario ha sido saldada

Referencia	CU13
Nombre	Salir grupo
Descripción	El usuario abandona el grupo, si no ha saldado todas sus deudas le saldrá una venta de confirmación, y vuelve a la pantalla principal.

Referencia	CU14
Nombre	Saldar deuda
Descripción	El usuario salda una deuda y esta se elimina de su lista

Referencia	CU15
Nombre	Unir grupo
Descripción	El usuario introduce el código de grupo y el nombre del grupo en la pantalla añadir grupo, se une a este y es redirigido a la pantalla de inicio del grupo.

Referencia	CU16
Nombre	Eliminar usuario grupo
Descripción	El usuario solicita eliminar a un integrante del grupo, tras la confirmación el integrante es eliminado.

Referencia	CU17
Nombre	Eliminar grupo
Descripción	El usuario solicita eliminar el grupo, si quedan deudas por saldar saldrá una ventana de confirmación. Tras la confirmación el grupo

	es eliminado.

Referencia	CU18
Nombre	Añadir tarea
Descripción	El usuario crea una tarea indicando su nombre y la urgencia de esta y se añadirá a la lista de tareas.

Referencia	CU19
Nombre	Eliminar tarea
Descripción	El usuario elimina la tarea.

Referencia	CU20
Nombre	Notificar
Descripción	El gestor de notificaciones envía notificaciones al usuario indicado

Referencia	CU21
Nombre	Almacenar imagen
Descripción	El gestor de imágenes almacena la imagen en la nube.

En el Anexo B se encuentran las tablas completas de los casos de uso con los actores, sus relaciones y las correspondientes precondición y postcondición.

3.2.4 Diagrama de clases

El diagrama de clase captura la estructura estática del sistema, mostrando las clases y las relaciones entre ellas.

Como se puede ver en la Figura 6 se representa mediante unos elementos que siguen la notación UML, en estos elementos se deben destacar:

- **Usuario:** la clase Usuario representa a los usuarios del sistema, y cada instancia de la clase tiene un conjunto de atributos, que incluyen dirección de correo electrónico, contraseña, nombre, apellido y token de sesión. La clase

Usuario se relaciona con la clase Deuda mediante una relación de "0..*" que indica que un usuario puede tener cero o varias deudas asociadas. De manera similar, se relaciona con la clase Gasto mediante una relación de "0..*" que permite a un usuario crear tantos gastos como desee. La clase Usuario también tiene una relación de "0..*" como creador con la clase Grupo, lo que significa que un usuario puede ser creador de tantos grupos como desee. Además, tiene una relación de "0.." como miembro con la clase Grupo, lo que significa que un usuario puede ser miembro de tantos grupos como desee. Por último, se relaciona con la clase Imagen mediante una relación de "0..1", lo que indica que un usuario puede tener una imagen o ninguna.

- **Deuda:** la clase Deuda representa las deudas que los usuarios tienen en el sistema, y cada instancia de la clase Deuda tiene un atributo "valor" que indica el total a deber o que te deben, dependiendo de si es positivo o negativo. La clase Deuda se relaciona con la clase Usuario mediante una relación de "1..1", lo que significa que cada deuda está asociada a un solo usuario.
- **Grupo:** la clase Grupo representa los grupos creados en el sistema por los usuarios y tiene atributos como código, nombre y descripción. La clase Grupo tiene una relación de composición con las clases Gasto y Tarea, lo que significa que está formada por estas dos clases. La relación de composición entre la clase Grupo y la clase Gasto indica que cada instancia de la clase Grupo tiene uno o varios gastos asociados, y estos gastos no existen fuera del contexto del grupo. De manera similar, la relación de composición entre la clase Grupo y la clase Tarea indica que cada instancia de la clase Grupo tiene una o varias tareas asociadas, y estas tareas no existen fuera del contexto del grupo.
- **Gasto:** la clase Gasto representa los gastos creados por los usuarios en los grupos, y cada instancia de la clase Gasto tiene un nombre y una cantidad que indica cuánto costó. La clase Gasto tiene una relación de composición con las clases Deuda e Imagen, lo que significa que está formada por estas dos clases. La relación de composición entre la clase Gasto y la clase Deuda indica que cada instancia de la clase Gasto tiene una o varias deudas asociadas, y estas deudas no existen fuera del contexto del gasto. La relación de composición entre la clase Gasto y la clase Imagen indica que cada instancia de la clase Gasto tiene una o ninguna imagen asociada.

- **Tarea:** la clase Tarea representa las tareas creadas en los grupos, y cada instancia de la clase Tarea tiene un nombre y una urgencia.
  Imagen: la clase Imagen representa las imágenes subidas al sistema, y cada instancia de la clase Imagen tiene un atributo de descripción y fecha. Las imágenes pueden estar asociadas a una instancia de la clase Usuario o a una instancia de la clase Gasto. La relación de "0..1" entre la clase Usuario y la clase Imagen indica que un usuario puede tener una o ninguna imagen asociada a su cuenta, de la misma manera, en la relación de "0..1" entre la clase Gasto y la clase Imagen indica que un gasto puede tener una o ninguna imagen asociada.

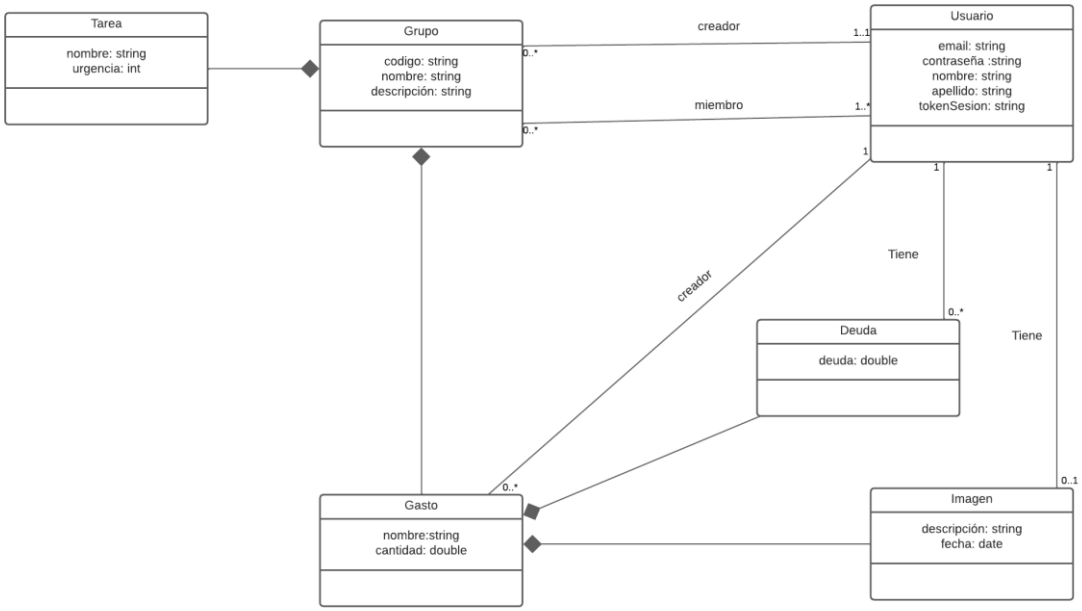

Figura 6 Diagrama de clases

3.3 Diseño de la base de datos

Se ha decidido utilizar una base de datos relacional [13] para trabajar con los datos de la aplicación. Aunque la mayoría de las aplicaciones móviles se basan en bases de datos no relacionales debido a que suelen realizar más lecturas que escrituras, esta aplicación creará, actualizará y eliminará datos constantemente. Por lo tanto, es necesario utilizar la gran versatilidad y potencia que brinda el modelo relacional.

Una de las grandes ventajas del modelo relacional son las restricciones, que son condiciones que los datos de la base de datos deben cumplir. Estas restricciones pueden ser implícitas, definidas en el modelo, o explícitas, impuestas por los usuarios. En cuanto a las restricciones impuestas por los usuarios, se utilizarán las siguientes:

- **Restricción de Clave Primaria:** permite declarar un atributo como la clave primaria.
- **Restricción de Unicidad:** permite que un atributo pueda tomar valores únicos entre las tuplas creadas. Se entiende que la clave primaria siempre tiene esta restricción.
- **Restricción de Obligatoriedad:** permite declarar si el atributo debe tomar siempre un valor.
- **Restricción de Clave Foránea:** se utiliza para vincular relaciones de una base de datos mediante claves foráneas. Cada clave foránea está relacionada con una clave primaria de otra tabla. En caso de que una tupla referenciada intente ser eliminada, se ha decidido aplicar la propagación en cascada para eliminar también todas las tuplas que la referencian y ahorrar trabajo.

Para cumplir con el objetivo de almacenar toda la información necesaria, eliminar la información redundante y mantener el número de tablas al mínimo, se realizará un análisis y diseño conceptual utilizando un diagrama entidad-relación. A partir de este diagrama, se llevará a cabo el diseño lógico para obtener una solución utilizando el modelo relacional.

3.3.1 Diseño conceptual

El diseño conceptual es el primer paso en la creación de una base de datos. En esta etapa, se utiliza el diagrama de clases obtenido durante el análisis para crear una solución de almacenamiento de datos mediante un diagrama entidad-relación. En el diagrama entidad-relación, cada clase del diagrama de clases se convierte en una entidad, y se crea una clave primaria entre sus campos si ninguno de sus atributos puede actuar como tal. La Figura 7 muestra el diagrama entidad-relación resultante.

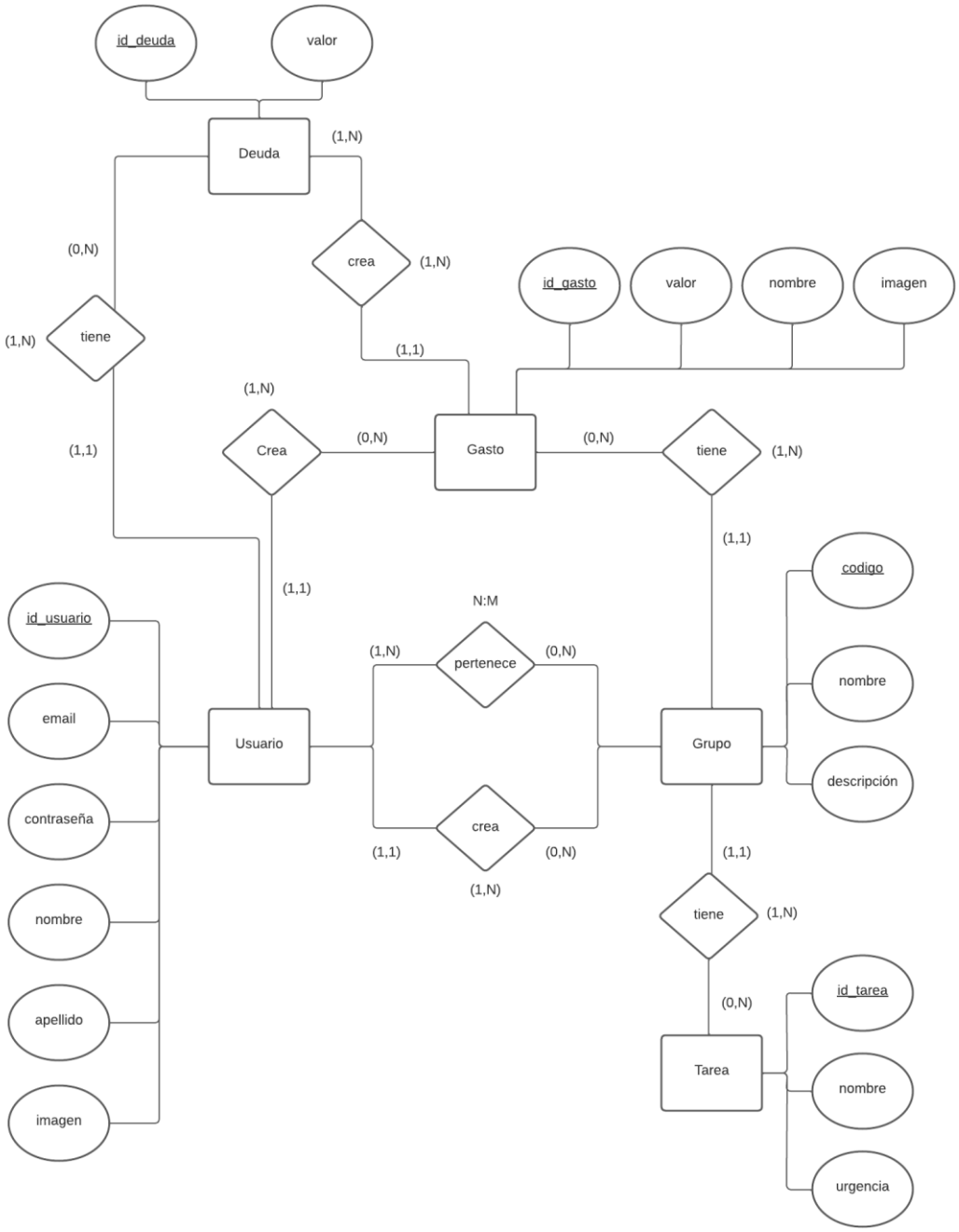

Figura 7 Diagrama entidad-relación

3.3.2 Diseño lógico

Durante esta etapa, se va a obtener un esquema lógico a partir del diagrama entidad-relación, siguiendo el modelo relacional [14]. El procedimiento a seguir será

bastante automático, siempre y cuando el análisis y el diagrama entidad-relación sean correctos. Bastará con seguir unas pocas normas para realizar la conversión:

- Cada entidad del diagrama entidad-relación pasa a ser una tabla del esquema lógico.
- Las relaciones N:M se modelan como una tabla nueva.
- Las relaciones 0:N y 1:N se incluyen en la tabla de cardinalidad N como una Clave Ajena.

El esquema lógico resultante es el siguiente:

Deuda (id_deuda, valor, id_usuario, id_gasto)

CP: id_deuda

CAj: id_usuario → Usuario(id_usuario)

CAj: id_gasto → Gasto(id_gasto)

Gasto (id_gasto, valor, nombre, imagen, id_usuario, id_grupo)

CP: id_gasto

CAj: id_usuario → Usuario(id_usuario)

CAj: id_grupo → Grupo(id_grupo)

Tarea (id_tarea, nombre, urgencia, id_grupo)

CP: id_tarea

CAj: id_grupo → Grupo(id_grupo)

Usuario (id_usuario, email, contraseña, nombre, apellido, imagen)

CP: id_usuario

Grupo (código, nombre, descripción, id_creadorGrupo)

CP: código

CAj: id_creadorGrupo → Usuario(id_usuario)

Pertenece (id_usuario, código)

CP: id_usuario, código

CAj: id_usuario → Usuario(id_usuario)

CAj: código → Grupo(código)

3.4 Aspecto de la aplicación

La paleta de colores de la aplicación se seleccionó considerando la naturaleza delicada del problema que busca solucionar, es decir, las deudas entre compañeros de piso y la resolución de tareas del hogar. Por esta razón, se optó por tonos de azul, que suelen asociarse con la calma, la estabilidad y la confianza [15]. La elección de los colores se puede apreciar en la Figura 8.

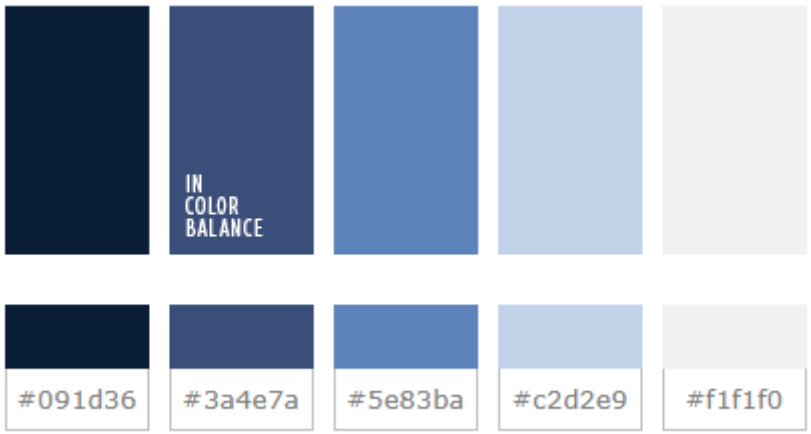

Figura 8 paleta de colores de la aplicación. Fuente: paletasdecolores.com

Se buscó diseñar los prototipos de la aplicación de manera sencilla y simple, con el fin de reducir la cantidad de pasos necesarios para realizar cualquier función y mejorar la experiencia del usuario. Estos prototipos fueron orientativos y se realizaron ajustes a lo largo del trabajo para alcanzar los objetivos de la aplicación. Se pueden observar variantes de las vistas para la página principal del usuario en la Figura 9, considerando diferentes tonalidades de azul y evaluando su eficacia.

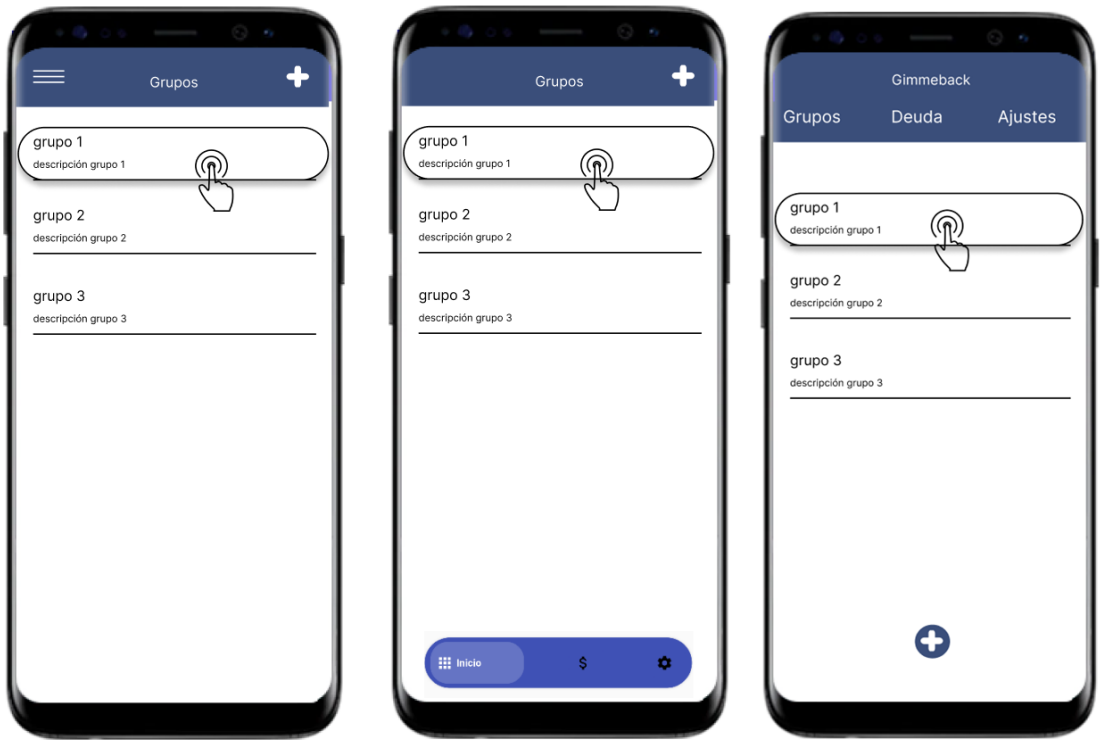

Figura 9 Prototipos página principal usuario

3.5 Estructura de la aplicación

En primer lugar, se va a elaborar un diagrama que describa la navegabilidad entre las vistas dentro de nuestra aplicación, Figura 10.

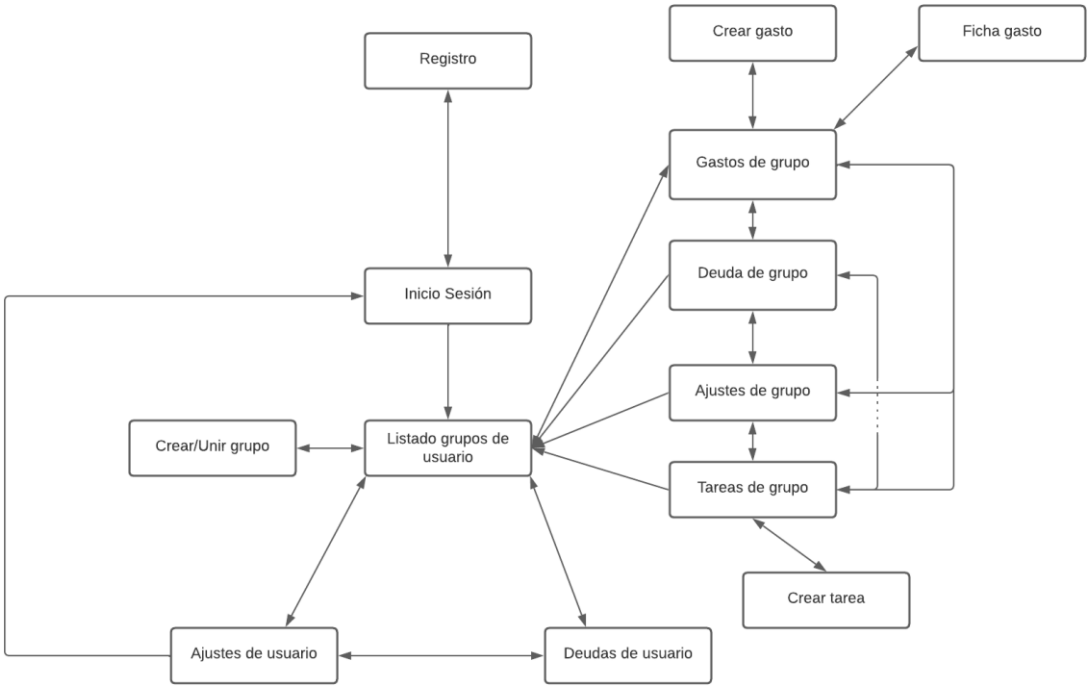

Figura 10 Diagrama de navegación

En la Figura 10 se puede observar una característica destacada de la aplicación: la capacidad de acceder a casi cualquier funcionalidad desde cualquier parte de la aplicación de manera eficiente. Esto se debe a que se tuvo en cuenta desde el inicio del diseño que el usuario ya sabe previamente para qué entrará a la aplicación, por ejemplo, para agregar un gasto al grupo tras haber pagado la cena. Por ello, se buscó que cada tarea se pudiera realizar de manera rápida y sencilla, ocupando poco tiempo.

Para lograr este objetivo, se utilizó en primer lugar el widget topBar, el cual se muestra en la Figura 11. Este widget permite crear vistas con anticipación y mostrarlas en una barra en la parte superior de la pantalla, indicando en qué vista se encuentra el usuario y permitiéndole cambiar de vista con solo pulsar en una de las otras opciones, lo cual permite que el cambio sea instantáneo.

Figura 11 Widget topBar

Después, se tuvo como idea para mejorar la experiencia del usuario al crear o unirse a un grupo, crear un gasto o una tarea. En lugar de generar una vista nueva, se implementó una ventana emergente que ocupa una porción de la pantalla, tal y como se puede apreciar en la Figura 12. Esta solución optimiza el proceso al no requerir cambiar de vista en ningún momento, lo que agiliza la interacción del usuario y, además, se actualiza inmediatamente en la misma vista donde se encuentra la lista correspondiente.

Figura 12 Ventana emergente crear/unir grupo

Cuando el usuario abre la aplicación por primera vez o después de haber cerrado sesión, se encontrará con la vista de inicio de sesión, la cual se muestra en la Figura 13. En esta vista, el usuario podrá ingresar sus credenciales para iniciar sesión, y si los datos son correctos, al pulsar el botón de iniciar sesión, será redirigido a la página principal. En caso contrario, si el usuario no dispone de una cuenta, podrá pulsar el texto destacado en azul en la parte inferior para acceder a la vista de registro, la cual se muestra en la Figura 14. En esta vista, el usuario podrá ingresar sus datos para crear una cuenta y, si los datos son correctos, iniciará sesión en su cuenta.

Figura 13 Vista inicio sesión

Figura 14 Vista registro

Una vez que el usuario ha iniciado sesión, la aplicación recordará sus datos y cada vez que la abra, se mostrará directamente la página principal sin necesidad de volver a iniciar sesión. En la página principal de la aplicación se pueden acceder a tres vistas distintas.

En la primera vista, llamada Grupos y mostrada en la Figura 15, se encuentra una lista con todos los grupos a los que pertenece el usuario, que muestra el nombre y una

breve descripción de cada grupo. Para acceder a un grupo en particular, se debe pulsar sobre él. También se dispone de un botón que al pulsarlo abre una ventana emergente que permite crear un nuevo grupo o unirse a uno ya existente. Para crear un nuevo grupo, simplemente se debe introducir un nombre y una breve descripción, mientras que para unirse a un grupo ya creado se necesita conocer su nombre y código.

En segundo lugar, como se puede apreciar en la Figura 16, se encuentra la vista Deuda, la cual muestra al usuario su deuda total entre todos los grupos a los que pertenece y una lista con el nombre del grupo y la deuda que tiene con él.

Por último, como se muestra en la Figura 17, la tercera vista es la de ajustes, donde se encuentran los datos del usuario y dos botones. El botón situado en la parte superior derecha cierra la sesión del usuario, mientras que el botón llamado "Editar perfil" abre una vista diferente que permite cambiar la imagen y los datos del usuario, como el correo de inicio de sesión, la foto de perfil, el nombre y los apellidos.

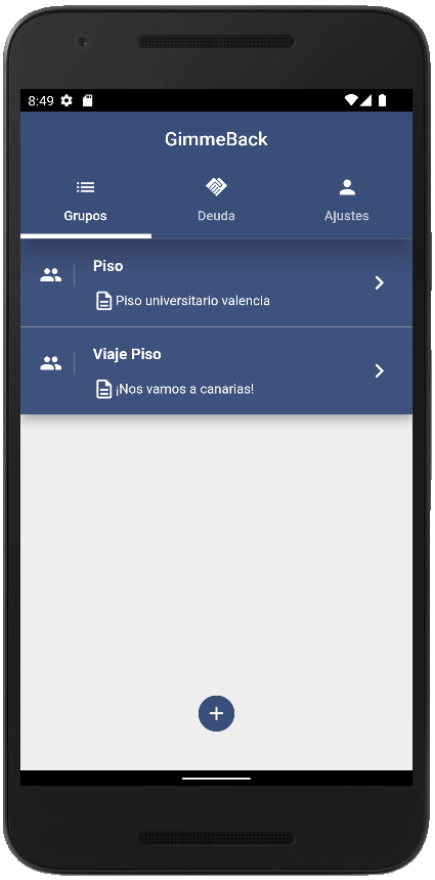

Figura 15 Vista grupos

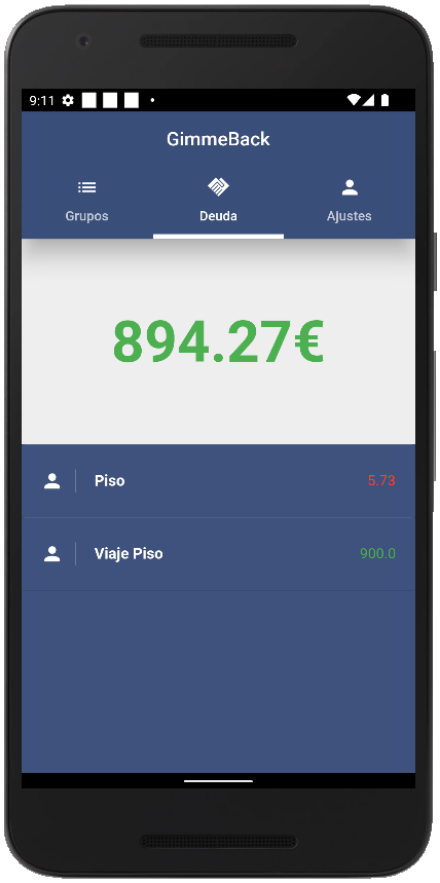

Figura 16 Vista deudas

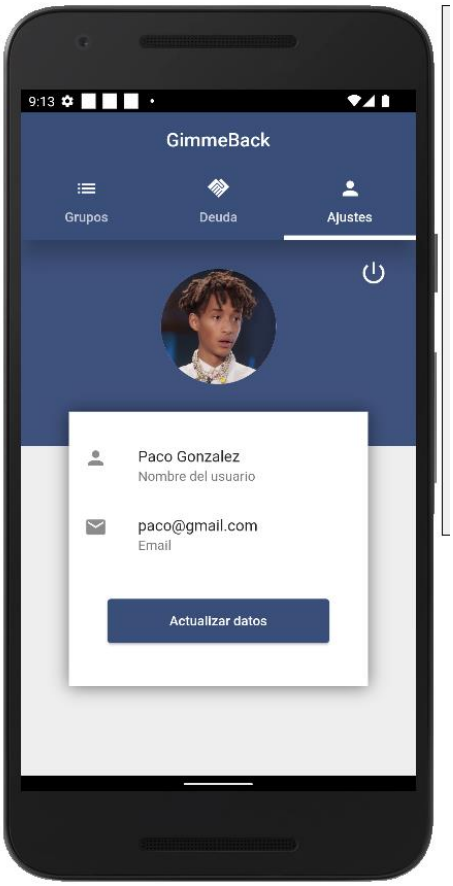

Figura 17 Vista ajustes

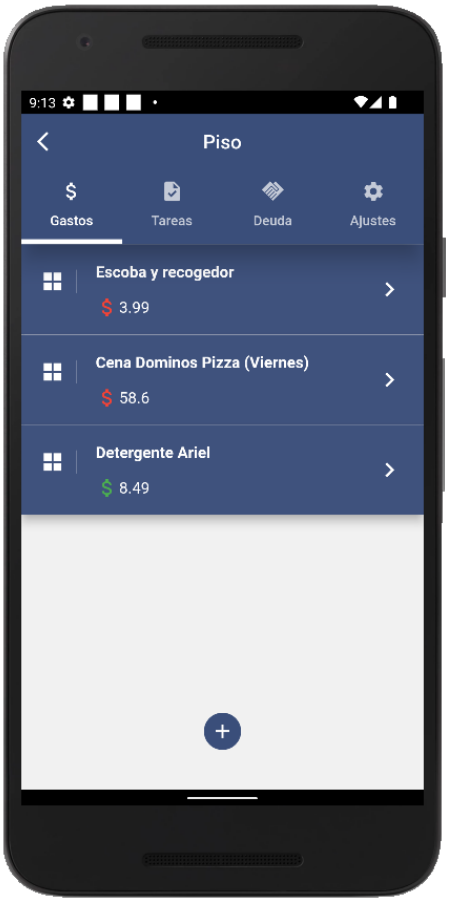

Figura 18 Vista gastos

Si el usuario accede a un grupo pasará a la página de grupo donde podrá acceder a 4 vistas. En la primera, como se observa en la Figura 18, el usuario puede ver una lista con todos los gastos que han añadido los participantes del grupo; el gasto se compone de un nombre, su valor y un icono en forma de dólar que toma el color verde si el gasto fue añadido por el usuario o color rojo si fue añadido por otro integrante del grupo. La segunda vista, Figura 19, es la de tareas, en la que el usuario tendrá todas las tareas que hay pendientes por hacer. Cada tarea está formada por su nombre y dos iconos: una papelera, que al ser pulsado eliminará la tarea, y otro en forma de barra que indicará la urgencia. Si se encuentra verde significará que es poco urgente, amarillo urgente y rojo muy urgente. La tercera vista, Figura 20, llamada deuda, imprime en pantalla en forma de gráfico de barras las deudas de cada uno de los integrantes del grupo. Si la barra es de color verde quiere decir que ese usuario tiene que recibir todo ese dinero, por el contrario, si es de color rojo el dinero que debe. Por último, Figura 21, en la vista de ajustes se encuentra el código del grupo, necesario para que el resto se una, una lista con los integrantes del grupo y una cruz roja al lado,

con la funcionalidad de eliminar del grupo a ese usuario y un botón que puede tomar dos valores. Si el usuario es el creador del grupo, este verá un botón llamado borrar grupo con la funcionalidad de mostrar un aviso de que el grupo será borrado si acepta, en caso de confirmar el grupo será borrado con todas las deudas, gastos y tareas. Por el otro lado, si el usuario no es el creador verá un botón con el nombre de salir del grupo, se mostrará un aviso muy similar al anterior, y si acepta saldrá del grupo, pero sus deudas, gastos y tareas se mantendrán en el grupo.

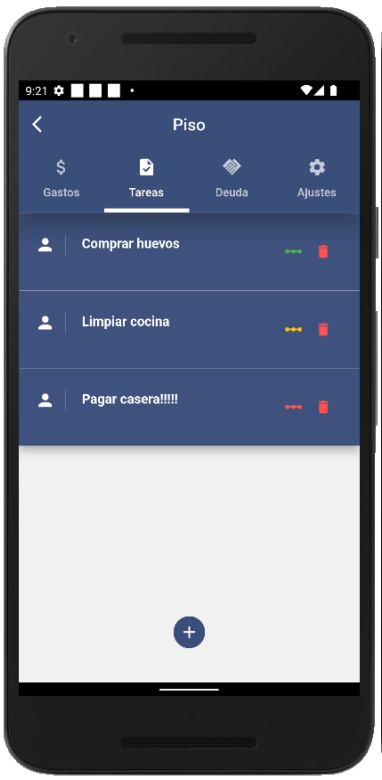

Figura 19 *Vista tareas*

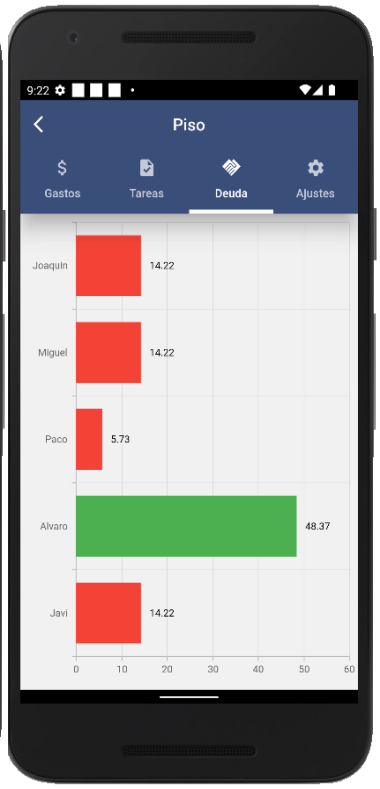

Figura 21 Vista deudas grupo

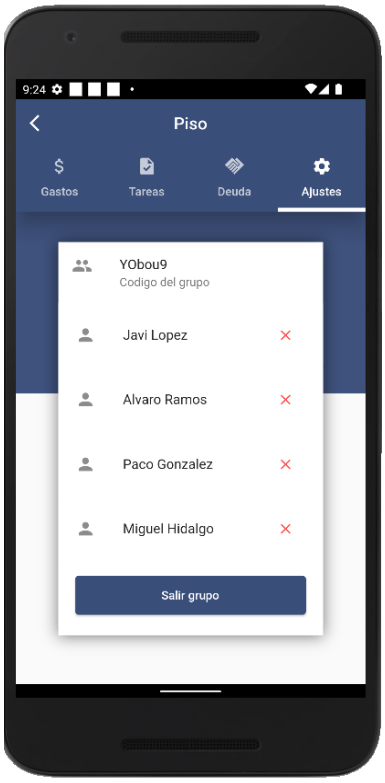

Figura 20 Vista ajustes grupo

4. Desarrollo

En este capítulo se explicará en detalle todo el proceso de desarrollo que se ha llevado a cabo para la implementación de la aplicación.

4.1 Tecnologías utilizadas

A continuación, se realizará una enumeración de las tecnologías seleccionadas con el fin de dar una explicación grosso modo de su funcionamiento y su contexto en el desarrollo actual, así como los motivos que han llevado a su elección.

4.1.1 Flutter

Como se comentó en el estado del arte, Flutter es la tecnología elegida para el desarrollo de la aplicación móvil. Permite generar versiones tanto para Android como para Apple, y ofrece un producto final muy similar a una aplicación nativa.

4.1.2 Android Studio

Android Studio es el IDE (entorno de desarrollo integrado) que se usará para crear el proyecto de Flutter [16]. Se escogió esta opción por su sencillez de uso y su potente depurador, que nos servirá al final del proyecto para realizar las pruebas pertinentes.

4.1.3 GetStorage

Paquete de Flutter que permite guardar datos de la aplicación en el almacenamiento local del móvil, siguiendo una estructura clave-valor [17].

4.1.4 Node.js y Express.js

Node.js es un entorno de tiempo de ejecución de JavaScript que te permite ejecutar código JavaScript en el lado del servidor. Con Node.js, puedes crear aplicaciones web y aplicaciones de servidor que se ejecutan en el backend [18].

Express.js es un framework de Node.js que te permite crear aplicaciones web y APIs (interfaz de programación de aplicaciones) REST de manera más rápida y sencilla. Con Express.js, puedes crear endpoints de API, definir rutas de acceso y manejar las solicitudes HTTP entrantes de manera más fácil y eficiente [18].

Se utilizará Node.js y Express.js para crear una API REST que se comunicará con la aplicación móvil. La API REST proporcionará una interfaz para que la aplicación móvil pueda enviar solicitudes al servidor y recibir respuestas correspondientes. Node.js y Express.js permite definir las rutas y endpoints de la API REST, manejar las solicitudes HTTP entrantes, interactuar con la base de datos y enviar las respuestas correspondientes al cliente.

4.1.5 Visual Studio Code

Entorno de desarrollo elegido para realizar la parte del servidor y crear la API. Se escogió por comodidad y familiaridad de uso.

4.1.6 Passport, Passport-JWT y Jsonwebtoken

Passport [19] es una biblioteca de autenticación para Node.js que simplifica el proceso de autenticación de usuarios en aplicaciones web mediante el uso de estrategias de autenticación. Passport-JWT [20] es una extensión de Passport que utiliza JSON Web Tokens (JWT) para la autenticación y autorización en aplicaciones web.

JSON Web Tokens es un estándar para la creación de tokens de acceso que pueden ser usados para autenticar y autorizar usuarios en aplicaciones web. La

biblioteca jsonwebtoken permite la creación y verificación de tokens JWT en aplicaciones Node.js.

En conjunto, Passport, Passport-JWT y jsonwebtoken [20] permiten la implementación de una autenticación segura y escalable en aplicaciones web de Node.js. Passport maneja las diferentes estrategias de autenticación, Passport-JWT utiliza tokens JWT para la autenticación y autorización, y jsonwebtoken se encarga de la creación y verificación de los tokens. Al utilizar estas tres bibliotecas juntas, se puede implementar una solución robusta de autenticación y autorización en aplicaciones web de Node.js.

4.1.7 Bcrypt

Bcrypt [21] es una biblioteca de Node.js que se utiliza para cifrar y descifrar contraseñas de manera segura. Es una de las bibliotecas más utilizadas para el cifrado de contraseñas, ya que ofrece una protección efectiva contra ataques de fuerza bruta y diccionario.

Bcrypt utiliza un algoritmo de cifrado basado en Blowfish, es un algoritmo de cifrado simétrico utilizado para proteger la privacidad de la información en las comunicaciones digitales, para generar hashes de contraseñas seguros. Además, cuenta con una función de salting automático que añade una cadena aleatoria al hash de la contraseña, lo que dificulta aún más su descifrado.

Al utilizar Bcrypt en una aplicación Node.js, las contraseñas de los usuarios pueden ser cifradas de forma segura antes de ser almacenadas en una base de datos. Cuando un usuario intenta iniciar sesión en la aplicación, la contraseña proporcionada se cifra y se compara con la contraseña cifrada almacenada en la base de datos. De esta manera, se puede verificar si la contraseña es correcta sin necesidad de almacenar la contraseña en texto plano, lo que garantiza una mayor seguridad para los usuarios de la aplicación.

4.1.8 Postman

Postman es una herramienta de desarrollo de API que ayuda a construir, probar y modificar APIs [22]. Va a ser la herramienta utilizada para testear la API.

Se escogió por su sencillez de uso, capacidad de crear colecciones y guardar llamadas ya creadas, ayudando con la organización y posteriores pruebas que se hagan durante y al final del proyecto para comprobar que no se haya roto nada en el proceso.

4.1.9 MySQL y MySQL Workbench

Como sistema de gestión de base de datos relacional usaremos MySQL y su IDE MySQL Workbench como entorno gráfico para ayudarnos a tenerlo todo de una forma más visual [23].

Esta elección se tomó mayoritariamente debido a su popularidad, escalabilidad y facilidad de uso.

4.1.10 Firebase

Firebase es un Backend-as-a-Service (Baas). Proporciona a los desarrolladores una variedad de herramientas y servicios para ayudarles a desarrollar aplicaciones de calidad, hacer crecer su base de usuarios y obtener beneficios. Está construido sobre la infraestructura de Google [24].

Este será el gestor de notificaciones y de imágenes gracias a las herramientas que proporciona Firebase de Firebase Storage y Firebase Messaging.

4.1.11 Railway

Railway es una plataforma de despliegue en la que puedes aprovisionar infraestructura, desarrollar con esa infraestructura localmente y luego desplegarla en la nube [25]. Esta será la herramienta usada que se encargará de alojar nuestra API REST.

Fue escogida principalmente porque tiene un plan gratuito y permite crear una base de datos MySQL en la nube, que será a la que accederá nuestra API. También cuenta con una gran comunidad.

4.2 Base de datos – Diseño físico

Durante el capítulo de diseño y análisis se ha seguido una metodología basada en el modelo relacional, generando una documentación en cada fase que ha servido de base para la siguiente. Gracias a ello, se ha obtenido una base de datos funcional en un sistema gestor de base de datos específico que nos permite representar la información con la que se desea trabajar. A partir del esquema lógico, se han especificado los tipos de datos de cada campo e introducido restricciones convenientes, como ON DELETE CASCADE en las claves externas, que permite eliminar automáticamente un registro relacionado cuando se elimina su clave primaria, evitando la necesidad de realizar consultas adicionales.

El código de creación de las tablas de la base de datos se encuentra en el Anexo C.

4.3 Cliente – Flutter

Por el lado del cliente se ha realizado una aplicación móvil desarrollada en Flutter. Esta será la parte visible para el usuario y es con la que interactuará.

En primer lugar, se decidió realizar un estudio de buenas prácticas en Flutter y nos encontramos con el patrón de arquitectura modelo-vista-controlador. El modelo-vista-controlador [26] es un patrón de arquitectura software que permite separar la lógica de la aplicación, la interfaz de usuario y los datos de una aplicación, facilitando el mantenimiento, la escalabilidad y la realización de las pruebas unitarias. Este patrón se compone de:

- **Modelo (Model):** es la representación de los datos y la lógica de negocio de la aplicación. En Flutter, esto se puede lograr creando clases que

manejen los datos y la lógica de la aplicación. Estos modelos se pueden conectar a bases de datos u otros servicios para obtener o almacenar datos.

- **Vista (View):** es la parte visual de la aplicación, lo que el usuario ve y con lo que interactúa. En Flutter, esto se logra creando widgets que representan la interfaz de usuario de la aplicación. Los widgets pueden contener otros widgets, lo que permite crear interfaces de usuario complejas y bien organizadas.
- **Controlador (Controller):** es el intermediario entre el Modelo y la Vista. Es responsable de recibir las acciones del usuario y actualizar el Modelo en consecuencia. También actualiza la Vista para reflejar los cambios en el Modelo. En Flutter, esto se puede lograr creando clases que manejen las interacciones del usuario y actualicen los modelos y las vistas correspondientes.

A continuación, se indica la estructura interna del cliente y como se aplicó el patrón modelo-vista-controlador:

- **Environment:** en este paquete se encuentra una única clase del mismo nombre. Contiene la URL que apunta a la dirección del servidor.
- **Models:** en este paquete se encuentran las clases que representan los objetos del modelo de base de datos de la aplicación. En estas clases se han implementado los métodos necesarios para transformar los datos recibidos por nuestra API REST, lo que nos permite trabajar de manera más cómoda con nuestros datos. Este paquete corresponde al modelo en el patrón de arquitectura Modelo-Vista-Controlador (MVC). En la Figura 22 se muestra un ejemplo de la clase User del modelo. En esta clase, se han definido las propiedades y los métodos necesarios para convertir la información que se va a enviar o recibir a través de la API REST. El método fromJson() se encarga de recibir un objeto JSON y transformarlo en un objeto User. El método toJson() convierte un objeto User en un objeto JSON, mientras que el método fromJsonList() se encarga de recibir una lista de objetos JSON y transformarla en una lista de objetos User.

```dart
User userFromJson(String str) => User.fromJson(json.decode(str));

String userToJson(User data) => json.encode(data.toJson());

class User {
  User({
    this.id_user,
    this.loginEmail,
    this.password,
    this.name,
    this.lastName,
    this.image,
    this.sessionToken
  });

  int? id_user;
  String? loginEmail;
  String? password;
  String? name;
  String? lastName;
  String? image;
  String? sessionToken;
```

```dart
  factory User.fromJson(Map<String, dynamic> json) => User(
    id_user: json["id_user"],
    loginEmail: json["loginEmail"],
    password: json["password"],
    name: json["name"],
    lastName: json["lastName"],
    image: json["image"],
    sessionToken: json["session_token"],
  );

  Map<String, dynamic> toJson() => {
    "id_user": id_user,
    "loginEmail": loginEmail,
    "password": password,
    "name": name,
    "lastName": lastName,
    "image": image,
    "session_token": sessionToken,
  };

  static List<User> fromJsonList(List<dynamic> jsonList){
    List<User> toList = [];
    jsonList.forEach((jsonUser) {
      User user = User.fromJson(jsonUser);
      toList.add(user);
    });

    return toList;
  }
}
```

Figura 22 Ejemplo modelo User

- **Pages:** Como se puede observar en la Figura 23, aquí se encuentra todo organizado por paquetes con el nombre de la vista. Dentro hay un archivo page, compuesto por los widgets que visualizará el usuario, y un archivo controller encargado de toda la lógica del page y de la comunicación con el servidor. En el patrón de arquitectura el archivo page correspondería a la vista y el controller al controlador.

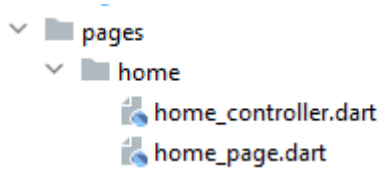

Figura 23 Ejemplo estructura pages

- **Providers:** en este paquete se encuentran las clases con los métodos que realizan la comunicación con el servidor, como se puede ver en la Figura 24 hay una clase por cada modelo.

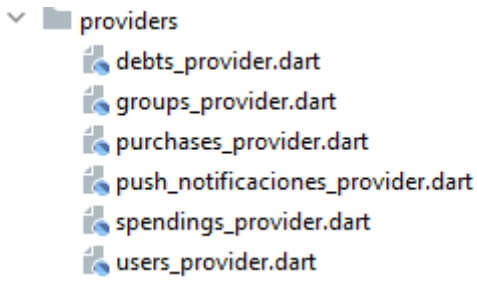

Figura 24 Estructura providers

- **Utils:** en este paquete se encuentran los widgets personalizados o algunos widgets que se repiten con frecuencia en la aplicación para así simplificar el código.

También con la finalidad de mantener la sesión del usuario iniciada, y no tener que volver a iniciar sesión cada vez que cierre la aplicación, se utiliza el paquete GetStorage.

Cuando un usuario inicia sesión en la aplicación móvil, esta crea un objeto de usuario que contiene su información personal, como su nombre de usuario, contraseña, correo electrónico, etc. Para que el usuario no tenga que iniciar sesión cada vez que cierre y abra la aplicación, el objeto de usuario se almacena utilizando GetStorage en el dispositivo móvil del usuario. De esta manera, cuando el usuario cierra la aplicación y la vuelve a abrir, la aplicación recuperará automáticamente el objeto de usuario almacenado en el dispositivo utilizando GetStorage. Como resultado, el usuario no tendrá que volver a iniciar sesión cada vez que abra la aplicación, lo que mejora la experiencia del usuario y reduce la fricción en el uso de la aplicación.

4.4 Servidor – Node.js y Express.js

Por parte del servidor se crea una API REST encargada de obtener y procesar los datos de nuestra base de datos y enviárselos al cliente.

En el servidor, se ha desarrollado una API REST encargada de obtener y procesar los datos de la base de datos y enviarlos al cliente. Para evitar accesos no autorizados a la API, se ha implementado un procedimiento de autenticación. Al iniciar sesión o registrarse, el servidor genera un JSON web token que se envía al cliente junto con los demás datos. Las solicitudes del cliente que no sean las mencionadas anteriormente, deberán pasar una verificación al llegar al endpoint de la API, donde el middleware Passport deshace el hash del JSON web token, extrae el identificador del usuario y realiza una consulta a la base de datos para comprobar su existencia. Si la comprobación es exitosa, la llamada continúa su ejecución normalmente, de lo contrario, se devuelve un mensaje indicando que el acceso no está permitido. Además, para aumentar la seguridad de la aplicación, se utiliza la librería Bcrypt de Node.js para encriptar y desencriptar contraseñas.

Con el objetivo de que el servidor sea lo más fácil de mantener, se ha seguido una estructura separando todo lo posible las responsabilidades de cada archivo, de esta manera si ocurre un error por ejemplo en una consulta sabremos al momento que archivo es y donde. La estructura seguida es la siguiente:

- **Config:** en este paquete encontramos 3 archivos distintos. Dos archivos encargados de la autenticación, uno es la constante secretOrKey y el otro contiene el método encargado de pasar la verificación una vez llegue la llamada a la API. El último archivo es el que realiza la conexión a la base de datos.
- **Controllers:** en este paquete se ha creado un archivo por cada entidad del modelo de base de datos. Cada uno contiene los métodos encargados de gestionar las solicitudes y devolver las respuestas de su entidad relacionada. En la Figura 25 se puede ver un ejemplo del método coger todas las deudas por el identificado del gasto.

```
getAllDebtsBySpendingId (req, res) {

    const id_spending = req.params.id_spending;

    Debt.findBySpendingId(id_spending, (err, data) => {
        if(err){
            return res.status(501).json({
                success: false,
                message: 'Hubo un error encontrando las deudas por el id del gasto',
                error: err
            });
        }


        return res.status(201).json({
            success: true,
            message: 'Las deudas se encontraron correctamente',
            data: data
        });
    });

},
```

Figura 25 Ejemplo solicitud devolver todas las deudas de un gasto. Archivo: debtController.js.

- **Models:** aquí se encuentra un archivo por cada tabla de la base de datos y cada uno contiene los métodos para realizar las consultas necesarias a sus tablas. En la figura 26 se puede observar el método encargado de realizar la consulta a la base de datos de obtener las deudas por el identificador del gasto.

```
Debt.findBySpendingId = (id_spending, result) => {
    const sql = `
    SELECT
        d.id_debt,
        d.value,
        d.id_user,
        d.id_spending
    FROM
        debt d
    WHERE
        d.id_spending = ?
    `;

    db.query
    (
        sql,
        [
            id_spending
        ],
        (err, res) => {
            if(err){
                //console.log('Error: ', err);
                result (err, null);
            }
            else {
                //console.log('Debts encontrados: ', res);
                result(null, res);
            }
        }
    )
}
```

Figura 26 Ejemplo consulta encontrar deudas por id del gasto. Archivo: debt.js

- **Routes:** en este paquete, igual que en el procedimiento seguido en controllers, se ha creado un archivo por cada entidad de nuestro modelo. Aquí encontramos los endpoints de nuestra API. En la Figura 27 se puede puede ver un ejemplo del endpoint al que llega el cliente cuando quiere obtener las deudas por el identificador del gasto.

```
app.get('/api/debts/spending/:id_spending', passport.authenticate('jwt',{session:false}),debtController.getAllDebtsBySpendingId);
```

Figura 27 Ejemplo endpoint encontrar deudas por id del gasto. Archivo debtRoutes.js

- **Server.js:** archivo donde creamos el servidor e importamos todas las rutas creadas en Routes.

En el Anexo D del presente trabajo se encuentra detallada la documentación de la API REST desarrollada para la aplicación móvil. Dicha documentación incluye información acerca de los endpoints disponibles, los métodos HTTP correspondientes, los parámetros necesarios para realizar las solicitudes, así como también las respuestas correspondientes que son devueltas por el servidor. Esta información es

fundamental para comprender el funcionamiento de la API y poder utilizarla de manera adecuada en la aplicación móvil.

4.5 Gestor de imágenes

Para almacenar imágenes en Firebase se utiliza la función Cloud Storage, la cual es un servicio de Google Cloud que permite almacenar objetos en la nube. Estos objetos son archivos inmutables y se almacenan en contenedores llamados buckets, los cuales están asociados con un proyecto específico [27].

Para agregar esta funcionalidad a nuestro proyecto, primero debemos crear un nuevo proyecto en la consola de Firebase y acceder a la pestaña de Storage para crear el bucket que se encargará de almacenar las imágenes y generar el enlace para acceder a ellas. Luego, es necesario generar una clave privada en Firebase, que se descargará como un archivo en formato JSON y contendrá las credenciales necesarias para permitir la comunicación entre Node.js y Firebase.

Además, se debe instalar la librería de "@google-cloud/storage" y crear un objeto Storage para implementar el método que se encargará de la subida de la imagen al bucket. De esta manera, se podrá almacenar de forma segura las imágenes en la nube y acceder a ellas desde cualquier dispositivo conectado a Internet.

4.6 Gestor de notificaciones

Se usará Firebase Cloud Messaging como gestor de notificaciones push. Para integrar las notificaciones en el proyecto se deben realizar cambios tanto en el servidor, cliente y modelo de datos.

4.6.1 Modelo de datos

Se añade un nuevo campo a la tabla User llamado notification_token.

ALTER TABLE `user`

ADD COLUMN `notification_token` VARCHAR(255) NULL AFTER `image `;

Este campo es necesario ya que este token es el encargado de indicarle a Firebase a qué dispositivo irá dirigido la notificación.

4.6.2 Servidor

En el servidor se crea un nuevo archivo en el paquete de controllers llamado pushNotificationController. Este archivo contendrá un método donde se creará un objeto de tipo JSON con la notificación y los notification_token, de los usuarios que van a recibirla, y realizará la petición a Firebase.

4.6.3 Cliente

Por el lado del cliente se deberán instalar 3 paquetes que son los siguientes:

- **Firebase_Messaging:** plugin de Flutter que nos permite usar la API de Firebase Cloud Messaging.
- **FirebaseCore:** dependencia necesaria de Firebase Messaging.
- **Flutter_local_notifications:** plugin que nos permite mostrar la notificación en el móvil.

Para comenzar con la implementación se crea un archivo llamado firebase_config que contiene toda la configuración necesaria para que tanto los dispositivos iOS y Android sean capaces de utilizar Firebase. Este archivo se agrega al paquete de Utils.

Finalmente, se crea un nuevo archivo llamado push_notification_provider que tendrá los métodos necesarios para comunicarse con la API de Firebase Cloud Messaging. Dentro de este archivo se encuentran los métodos encargados de actualizar el notification_token del usuario, de mostrar la notificación en el móvil, de crear una instancia de FirebaseMessaging y un último que permite a la aplicación mantenerse en escucha para esperar las peticiones. Este archivo se agrega al paquete de Providers.

5. Pruebas y despliegue

En este capítulo se explicarán las pruebas realizadas durante y al final del proceso de desarrollo y sus resultados.

5.1 Pruebas de aceptación

Las pruebas de aceptación se han realizado después de cada tarea completada en Trello. Estas pruebas permiten verificar que la funcionalidad implementada en cada tarea se comporta de manera adecuada y cumple con los requisitos especificados. Además, al realizar las pruebas de forma continua, se pueden identificar los errores de forma temprana y corregirlos de manera inmediata, lo que reduce el tiempo y los costos de desarrollo.

5.2 Pruebas unitarias - Cliente

Durante el proceso de desarrollo se fueron realizando pruebas de caja blanca al finalizar una nueva funcionalidad, utilizando el depurador que proporciona Android Studio, para verificar que el nuevo código escrito funcionase según lo esperado.

La lógica del código responsable de las acciones más importantes del cliente, los controladores, fue sometida a estas pruebas. En la mayoría de los casos creación, actualización y borrado de listas.

Como estas pruebas se realizaron durante el desarrollo del cliente, se aseguró que todas las unidades de código de la aplicación funcionen según lo previsto, lo que garantiza la estabilidad para las unidades de código añadidas posteriormente.

5.3 API Testing/Pruebas de caja negra – Servidor

Como las APIs no disponen de una interfaz de usuario, estas pruebas se realizaron mediante la herramienta Postman. Postman nos permite crear llamadas GET, POST, PUT y DELETE a una ruta de nuestro servidor y recibir la respuesta. Cada vez que se creaba una nueva ruta en el servidor se creaba una llamada en Postman, para comprobar que el resultado era el esperado. La llamada se guardaba en Postman y al final del desarrollo se volvieron a ejecutar todas las llamadas creadas durante el proceso, con el fin de confirmar que las respuestas seguían siendo las esperadas.

5.4 Pruebas de componentes

Se realizaron pruebas de caja negra en los diferentes componentes de la aplicación para asegurar que el comportamiento sea el esperado, además de garantizar que funcionen correctamente y de forma conjunta los componentes que estén relacionados de alguna manera. Esto se hizo probando varios casos de error y casos válidos para ver si la aplicación advertía de los errores y respondía apropiadamente.

Al igual que las pruebas unitarias y el API testing, estas pruebas también se llevaron a cabo durante el proceso, con el fin de garantizar una mayor estabilidad y seguridad para los componentes desarrollados posteriormente. A continuación, se muestran algunos ejemplos de pruebas sobre componentes donde pueden aparecer más errores, enseñando así la metodología que se ha seguido en el resto de los errores similares.

Uno de los primeros errores que podrían ocurrir en la aplicación sería cuando el usuario intentase registrarse, por ejemplo, que el usuario no introduzca toda la información solicitada o bien introduzca información errónea o no valida. Como se muestra en la figura 28 la aplicación muestra un mensaje de error entendible y visible para el usuario.

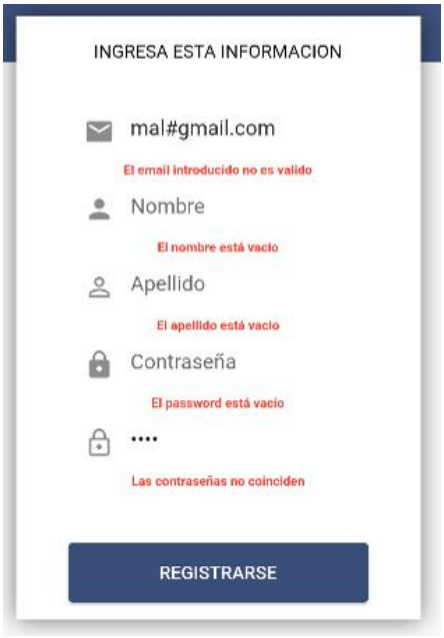

Figura 28 Error al registrarse

En la figura 29, se puede observar el error que se muestra al intentar iniciar sesión con unas credenciales incorrectas. Mismamente, otro error común podría ser equivocarse al poner el nombre y el código de grupo para unirse a un grupo, en cuyo caso el mensaje error mostrado sería el de la figura 30.

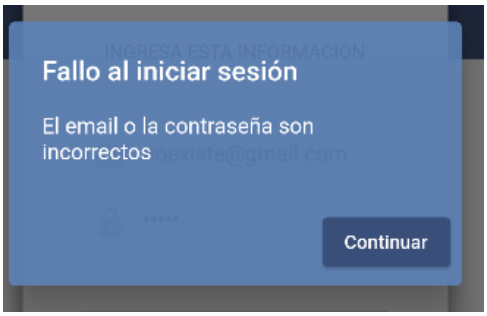

Figura 29 Error al iniciar sesión

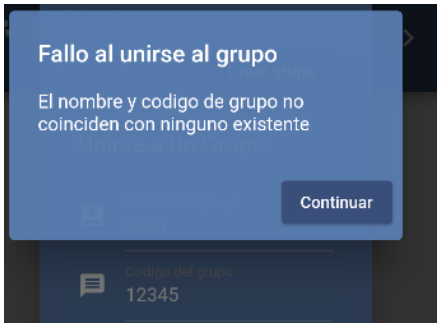

Figura 30 Error al unirse a un grupo

5.5 Pruebas de integración y sistema

Aquí se realizarán las pruebas sobre el funcionamiento completo del sistema, los componentes y sus relaciones con las respectivas llamadas a la API. Estas pruebas se realizan al final de la fase de desarrollo, ya que requiere que la aplicación se encuentre finalizada.

A continuación, se listan algunos ejemplos de pruebas de integración que fueron realizadas:

- Se probó a crear grupos y a unir a usuarios a estos.
- Se probó a añadir gastos a un grupo y comprobar que la deuda se repartía de manera correcta entre los integrantes del grupo.
- Se probó a eliminar un gasto y que se corrigiese la deuda de los integrantes del grupo.
- Se probó a añadir gastos en distintos grupos y que se reflejase en el valor de la deuda total del usuario de la página principal y que la lista de las deudas por grupo fuese correcta.
- Se probó a añadir tareas con los distintos niveles de urgencia y que el color que tomaba la barra dependiendo del nivel fuese el correcto.
- Se probó a editar los datos del usuario en ajustes y que estos se cambiaran y mostraran correctamente.
- Se probó a eliminar el grupo, que se eliminasen todos los gastos y deudas relacionados y se corrigiese la deuda total y la lista de deudas por grupo de la página principal del usuario.
- Se probó el gestor de imágenes al registrar un usuario con imagen y se comprobó que esta se guardara en base de datos y se visualizase en ajustes de usuario.
- Se probó el gestor de notificaciones al unirse a un grupo con un móvil y dos máquinas virtuales distintas comprobando que llegase la notificación.

Las pruebas de integración de la lista junto con las no comentadas obtuvieron resultados positivos.

5.6 Pruebas de compatibilidad e interfaz gráfica

Para garantizar una buena experiencia de usuario en cualquier dispositivo, es fundamental que la interfaz de usuario de la aplicación móvil funcione correctamente y se adapte a diferentes entornos, plataformas y tamaños de dispositivos. Para ello, se

llevaron a cabo pruebas de adaptabilidad de la aplicación en dispositivos tanto reales como virtuales, de diferentes tamaños y resoluciones, en ambas plataformas Android e iOS. En cada caso, se verificó si la interfaz de usuario se mostraba según lo esperado. Los dispositivos utilizados para las pruebas incluyeron el Samsung Galaxy S10 y A6+, el iPhone X, y las máquinas virtuales de Nexus 5X y Pixel 2.

5.7 Pruebas y cuestionario de usabilidad

Como se comentó en los objetivos se busca una aplicación sencilla, intuitiva y rápida de usar, para comprobar si se cumplió con esto se van a realizar unas pruebas y unos cuestionarios sobre algunos usuarios objetivo de la aplicación.

Los usuarios que participaron en las pruebas fueron los que se muestran en la tabla siguiente:

	Edad	Móvil
Usuario 1	19	Samsung Galaxy S10
Usuario 2	23	iPhone X
Usuario 3	28	iPhone X

La prueba a la que fueron sometidos era ver cuanto tardaban en realizar determinadas acciones en la aplicación. En la tabla que se muestra a continuación podemos ver la acción y cuanto tarda el usuario en realizarla.

	Tiempo en realizar la acción		
	Usuario 1	Usuario 2	Usuario 3
Registrarse en la aplicación	1min 10s	1min 3s	48s
Iniciar sesión en la aplicación	19s	15s	15s
Crear un grupo	19s	14s	15s
Crear un gasto	30s	26s	27s
Crear tarea	8s	10s	8s
Eliminar tarea	3s	3s	3s
Comprobar Deuda del usuario total	5s	6s	6s

Eliminar grupo	9s	8s	8s
Cerrar sesión	15s	5s	5s

Los resultados obtenidos parecen muy positivos ya que encontraban como realizar la acción muy rápido y parece ser una buena señal de que la interfaz es muy intuitiva.

Luego, para obtener un valor cuantitativo de su experiencia, se les pidió que respondieran un cuestionario simple en el que los usuarios calificaron las actividades y preguntas sugeridas en una escala de 1 a 5, donde 1 significa totalmente en desacuerdo y 5 significa totalmente de acuerdo.

	Valoración		
	Usuario 1	Usuario 2	Usuario 3
Considero que usaría esta aplicación.	4	4	5
Encontré la aplicación complicada de utilizar	2	1	1
Considero que había inconsistencia en la aplicación	1	1	1
Creo que se aprende rápido a utilizar la aplicación	5	4	5
La aplicación es rápida	4	4	4
La aplicación le avisa cuando hay un error	4	NP	NP
En general opino que la aplicación es buena	4	4	5

5.8 Despliegue

Para empezar con el despliegue se ha creado un repositorio remoto en GitHub [28] y se ha subido el servidor de Node.js. A continuación, se accede a railway.app [29],

encargada del despliegue de nuestro servidor, se procede a crear una cuenta a partir de nuestra cuenta de GitHub.

Una vez dentro se debe crear la base de datos MySQL que usará nuestro servidor, se utiliza el diseño físico ya implementado anteriormente en MySQL Workbench y se crean las tablas necesarias.

Finalmente, se crea un nuevo proyecto, que va a ser un despliegue desde un repositorio GitHub y se selecciona el servidor. Si se observa un Succeeded en verde significa que todo fue según lo esperado y se desplegó correctamente.

Ahora que ya se completó el despliegue satisfactoriamente lo único que se debe cambiar en el cliente es la URL a la que accedía de manera local a la API por el dominio que ofrece Railway.

6. Conclusiones

Para concluir mi memoria, quiero presentar mis breves conclusiones sobre todo el trabajo que realicé y su impacto a nivel personal.

En general, pude cumplir con los objetivos que me propuse al inicio del proyecto. Desarrollé una aplicación móvil funcional, eficiente y atractiva, a pesar de no tener experiencia previa en el desarrollo de aplicaciones móviles ni en la creación de servidores Node.js. Al principio, consideré utilizar una plataforma backend como servicio, como Firebase, pero decidí aprovechar la oportunidad para aprender Node.js, una tecnología que siempre me interesó. Aunque aprender Node.js fue una tarea difícil, considero que todas las horas invertidas en ello fueron una gran inversión de cara al futuro.

El mayor desafío que enfrenté durante el desarrollo de la aplicación fue la ambición desmedida de agregar demasiadas funcionalidades sin considerar el tiempo y esfuerzo que conllevaba. Sin embargo, seguir una metodología ágil me permitió gestionar el proyecto de manera efectiva y adaptarme a las dificultades que surgieron durante el proceso.

Flutter resultó ser un framework muy útil para el desarrollo de la aplicación móvil, no solo por su capacidad para crear aplicaciones nativas a partir de un solo código, sino también por su activa comunidad que proporciona soluciones a los errores encontrados durante el desarrollo y ofrece una gran cantidad de widgets personalizados.

Las asignaturas AER, ISW y BDA jugaron un papel crucial en el análisis y diseño previo de la aplicación, lo que me permitió estructurarla adecuadamente y evitar problemas futuros. Además, las asignaturas PSW, IEI y PIN fueron de gran ayuda en la implementación de la aplicación, proporcionándome conocimientos sobre el funcionamiento de las API y la importancia de seguir una metodología de trabajo adecuada. Estas asignaturas también me brindaron la oportunidad de trabajar en proyectos reales, lo que me permitió comprender mejor que el desarrollo de software es lo que realmente me apasiona.

En cuanto a posibles extensiones de la aplicación en el futuro, revisaré algunas tareas que quedaron pendientes en el Backlog de Trello, siendo la más destacada la funcionalidad que enviaría una notificación al usuario con recomendaciones divertidas para saldar sus deudas. Esta mejora proporcionaría una experiencia de usuario más completa y atractiva, incentivando al usuario a seguir utilizando la aplicación.

En resumen, el trabajo que realicé supuso una excelente experiencia para afianzar los conocimientos adquiridos durante la carrera y madurar como programador. Estoy muy contento con el resultado y el impacto personal que ha tenido en mi desarrollo profesional.

Bibliografía

[1] Joan Batalla, “El alquiler de viviendas se dispara un 17,5% en la C. Valenciana en el último año”. https://www.levante-emv.com/economia/2022/08/30/alquiler-viviendas-dispara-17-5-74648971.html#:~:text=El%20orden%20de%20las%20zonas,(7%2C8%25)%2C%20Catalu%C3%B1a, Accedido en septiembre 2022.

[2] Julia Martins, “¿Qué es Kanban?”. https://asana.com/es/resources/what-is-kanban, Accedido en septiembre 2022.

[3] “Trello”. https://trello.com/es, Accedido en septiembre 2022

[4] “Aplicación Android con Android Studio”. https://developer.android.com/studio/intro?hl=es-419, Accedido en marzo 2023

[5] “Aplicación Apple con Swift”. https://developer.apple.com/swift/, Accedido en marzo 2023

[6] “Introducción Apache-Cordova”. https://cordova.apache.org/docs/en/latest/guide/overview/index.html, Accedido en septiembre 2022

[7] Luis peris, “Desventajas de Apache-Cordova”. https://luisperis.com/apache-cordova/, Accedido en septiembre 2022.

[8] “React”. https://reactnative.dev/, Accedido en septiembre de 2022.

[9] “The Oracle Client/Server architecture”. https://docs.oracle.com/cd/A57673_01/DOC/server/doc/SCN73/ch20.htm. Accedido en octubre 2022

[10] “HTTP”. https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview. Accedido en octubre 2022

[9] “Flutter”. https://flutter.dev/,Accedido en septiembre de 2022

[10] David Bernal González “¿Qué es Flutter?”. https://profile.es/blog/que-es-flutter-sdk/, Accedido en septiembre 2022.

[11] “JSON”. https://www.json.org/json-en.html. Accedido en octubre 2022

[12] Lucidchart, “¿Qué es UML?”. https://www.lucidchart.com/pages/es/que-es-el-lenguaje-unificado-de-modelado-uml, Accedido en septiembre 2022.

[13] UNC. “Modelo relacional” https://oftgu.eco.catedras.unc.edu.ar/unidad-3/sistemas-de-gestion-de-base-de-datos/modelo-relacional-conceptos-basicos-y-fundamentos/ Accedido en septiembre 2022

[14] UOC. “Diseño lógico” http://cv.uoc.edu/annotation/cb826b689abc472d8fb5b2519840058b/699689/PID_00213705/PID_00213705.html#w31aab9c13. Accedido en septiembre 2022

[15] Alheyalle, Mustafa. “User Interface Designing: Colour Therapy Sharing Application.” ESRSA Publication (2014): n. pag. Print.

[16] “Android Studio y Flutter”. https://docs.flutter.dev/development/tools/android-studio. Accedido en octubre 2022.

[17] “get_storage”. https://pub.dev/packages/get_storage. Accedido en octubre 2022

[18] “Introducción a Express/Node”. https://developer.mozilla.org/es/docs/Learn/Server-side/Express_Nodejs/Introduction, Accedido en octubre 2022

[19] “Passport” https://www.passportjs.org/. Accedido en octubre 2022

[20] David Poza Suárez “Autenticación via JWT en express.Js”. https://davidinformatico.com/jwt-express-js-passport. Accedido en octubre 2022

[21] “bcrypt”, https://www.npmjs.com/package/bcrypt. Accedido en octubre 2022

[22] “¿Qué es Postman?”. https://www.postman.com/. Accedido en octubre 2022

[23] “MySQL Workbench”. https://www.mysql.com/products/workbench/. Accedido en octubre 2022

[24] “Firebase”. https://firebase.google.com/?hl=es. Accedido en octubre 2022

[25] “Railway”. https://railway.app/. Accedido en octubre 2022

[26] “MVC”. https://developer.apple.com/library/archive/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html. Accedido en noviembre 2022

[27] “¿Qué es Cloud Storage?”. https://cloud.google.com/storage/docs/introduction?hl=es-419. Accedido en noviembre 2022

[28] “GitHub”. https://github.com/, Accedido en noviembre 2022

Anexo A

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	**Alto**	**Medio**	**Bajo**	**No Procede**
ODS 1. **Fin de la pobreza.**				**X**
ODS 2. **Hambre cero.**				**X**
ODS 3. **Salud y bienestar.**				**X**
ODS 4. **Educación de calidad.**				**X**
ODS 5. **Igualdad de género.**				**X**
ODS 6. **Agua limpia y saneamiento.**				**X**
ODS 7. **Energía asequible y no contaminante.**				**X**
ODS 8. **Trabajo decente y crecimiento económico.**				**X**
ODS 9. **Industria, innovación e infraestructuras.**				**X**
ODS 10. **Reducción de las desigualdades**.				**X**
ODS 11. **Ciudades y comunidades sostenibles**.	**X**			
ODS 12. **Producción y consumo responsables.**	**X**			
ODS 13. **Acción por el clima.**		**X**		
ODS 14. **Vida submarina.**				**X**
ODS 15. **Vida de ecosistemas terrestres.**				**X**
ODS 16. **Paz, justicia e instituciones sólidas.**				**X**
ODS 17. **Alianzas para lograr objetivos.**	**X**			

Al desarrollar este proyecto enfocado en la gestión de gastos y tareas compartidas en viviendas de alquiler, he sido consciente de cómo este trabajo puede contribuir a la consecución de los Objetivos de Desarrollo Sostenible de la UE. En concreto mi proyecto está relacionado con los objetivos de: Ciudades y comunidades sostenibles, Producción y consumo responsables, Acción por el clima y Alianzas para lograr objetivos.

Mi aplicación podría contribuir a fomentar la convivencia y el cuidado de los espacios compartidos en viviendas de alquiler, lo que podría mejorar la calidad de vida y la sostenibilidad de las comunidades urbanas. Además, al promover una mayor transparencia y colaboración en la gestión de los gastos y las tareas compartidas, mi aplicación podría fomentar un consumo más responsable y sostenible.

También he considerado cómo mi aplicación podría contribuir a la lucha contra el cambio climático, por ejemplo, si se utiliza para coordinar el pago de servicios públicos como la luz o el agua, se podría reducir el consumo innecesario y, por tanto, disminuir el impacto ambiental.

Por último, creo que mi aplicación también podría contribuir a la creación de alianzas para la consecución de los ODS al fomentar la colaboración y la cooperación entre personas y grupos que comparten objetivos comunes. La tecnología puede ser una herramienta valiosa para unir a las personas y fomentar la colaboración, lo que podría contribuir significativamente a la consecución de los ODS.

En conclusión, estoy muy contento de que la aplicación desarrollada, además de ser muy útil en su ámbito, pueda contribuir a la consecución de los objetivos de desarrollo sostenible de la UE. Creo que este proyecto demuestra cómo la tecnología puede ser una gran herramienta para fomentar un estilo de vida más sostenible y responsable.

Anexo B

Descripción detallada de los casos de uso

Referencia	CU01
Nombre	Registrar cuenta
Descripción	El usuario introduce los datos solicitados. Una vez finalizado el registro, el usuario será redirigido a la pantalla principal de la aplicación.
Actor	Usuario no autenticado
Relación	Almacenar Imágen
Precondición	El usuario no ha iniciado sesión previamente y accede a la aplicación sin email ni contraseña que introducir.
Postcondicion	Se registra al usuario en el sistema

Referencia	CU02
Nombre	Iniciar Sesión
Descripción	El usuario introduce los datos solicitados. Una vez introducidos, el usuario podrá iniciar sesión a su cuenta y será redirigido a la pantalla principal de la aplicación.
Actor	Usuario no autenticado
Relación	
Precondición	El usuario no ha iniciado sesión previamente y dispone de las credenciales de acceso de su cuenta
Postcondicion	

Referencia	CU03
Nombre	Editar perfil
Descripción	El usuario cambia algunos datos que fueron introducidos durante su registro como el nombre y la imagen, por ejemplo.

Actor	Usuario autenticado
Relación	Almacenar Imagen
Precondición	El usuario se ha identificado previamente
Postcondicion	Los datos son validados y se muestra su perfil actualizado

Referencia	CU04
Nombre	Crear grupo
Descripción	El usuario crea un grupo introduciendo los datos solicitados y será redirigido a la pantalla de inicio del grupo.
Actor	Usuario autenticado
Relación	
Precondición	El usuario se ha identificado previamente
Postcondicion	Se muestra el nuevo grupo creado

Referencia	CU05
Nombre	Cerrar sesión
Descripción	El usuario solicita cerrar su sesión actual y vuelve a la pantalla de inicio de sesión.
Actor	Usuario autenticado
Relación	
Precondición	El usuario se ha identificado previamente
Postcondicion	

Referencia	CU06
Nombre	Visualizar deudas
Descripción	El usuario ve sus distintas deudas a recibir o a dar en cada uno de los grupos.
Actor	Usuario autenticado
Relación	
Precondición	El usuario se ha identificado previamente
Postcondicion	

Referencia	CU07
Nombre	Seleccionar grupo
Descripción	El usuario selecciona cualquiera de los grupos en los que participa para acceder a él.
Actor	Usuario autenticado
Relación	
Precondición	El usuario se ha identificado previamente El usuario ha creado o se ha unido a un grupo previamente
Postcondicion	Se muestra el grupo seleccionado

Referencia	CU08
Nombre	Visualizar gastos grupo
Descripción	El usuario dentro del grupo visualiza los gastos añadidos por los integrantes de este con su nombre y precio.
Actor	Usuario autenticado
Relación	
Precondición	El usuario se ha identificado previamente El usuario ha creado o se ha unido a un grupo previamente
Postcondicion	

Referencia	CU09
Nombre	Visualizar ficha de un gasto
Descripción	El usuario dentro del grupo selecciona un gasto y accede a sus detalles.
Actor	Usuario autenticado
Relación	
Precondición	El usuario se ha identificado previamente El usuario ha creado o se ha unido a un grupo previamente El grupo tiene gastos añadidos
Postcondicion	Se muestran los datos del gasto

Referencia	CU10
Nombre	Añadir gasto

Descripción	El usuario introduce los datos solicitados. Una vez introducidos el usuario añadirá el gasto al grupo y será redirigido a la pantalla de inicio del grupo
Actor	Usuario autenticado
Relación	
Precondición	El usuario se ha identificado previamente El usuario ha creado o se ha unido a un grupo previamente
Postcondicion	Se añade el gasto al grupo y se recalculan las deudas según su valor entre los integrantes

Referencia	CU11
Nombre	Eliminar gasto añadido
Descripción	El usuario elimina un gasto que ha sido añadido por él mismo.
Actor	Usuario autenticado
Relación	
Precondición	El usuario se ha identificado previamente El usuario ha creado o se ha unido a un grupo previamente El usuario es el creador del gasto
Postcondicion	Se elimina el gasto del grupo y se recalculan las deudas según su valor entre los integrantes

Referencia	CU12
Nombre	Verificar saldar deuda
Descripción	El usuario verifica que su deuda con otro usuario ha sido saldada
Actor	Usuario autenticado
Relación	
Precondición	El usuario se ha identificado previamente El usuario ha creado o se ha unido a un grupo previamente Otro usuario ha saldado una deuda que tenía con el usuario
Postcondicion	Se elimina la deuda y se recalculan las deudas según su valor entre los integrantes

Referencia	CU13
Nombre	Salir grupo
Descripción	El usuario abandona el grupo, si no ha saldado todas sus deudas le saldrá una venta de confirmación, y vuelve a la pantalla principal.
Actor	Usuario autenticado
Relación	Notificar
Precondición	El usuario se ha identificado previamente El usuario ha creado o se ha unido a un grupo previamente El usuario ha saldado todas sus deudas
Postcondicion	El usuario ya no visualiza el grupo

Referencia	CU14
Nombre	Saldar deuda
Descripción	El usuario salda una deuda y esta se elimina de su lista
Actor	Usuario autenticado
Relación	Notificar
Precondición	El usuario se ha identificado previamente El usuario ha creado o se ha unido a un grupo previamente El usuario no es el creador de la deuda
Postcondicion	Se elimina el gasto para él y se recalculan las deudas del grupo

Referencia	CU15
Nombre	Unir grupo
Descripción	El usuario introduce el código de grupo y el nombre del grupo en la pantalla añadir grupo, se une a este y es redirigido a la pantalla de inicio del grupo.
Actor	Usuario autenticado
Relación	Notificar
Precondición	El usuario se ha identificado previamente
Postcondicion	Se muestra el grupo

Referencia	CU16

Nombre	Eliminar usuario grupo
Descripción	El usuario solicita eliminar a un integrante del grupo, tras la confirmación el integrante es eliminado.
Actor	Usuario autenticado
Relación	Notificar
Precondición	El usuario se ha identificado previamente El usuario es el creador del grupo
Postcondicion	Se eliminan todas las deudas del usuario y se recalculan las deudas del grupo

Referencia	CU17
Nombre	Eliminar grupo
Descripción	El usuario solicita eliminar el grupo, si quedan deudas por saldar saldrá una ventana de confirmación. Tras la confirmación el grupo es eliminado.
Actor	Usuario autenticado
Relación	Notificar
Precondición	El usuario se ha identificado previamente El usuario es el creador del grupo
Postcondicion	El grupo es eliminado del sistema junto a todas sus deudas

Referencia	CU18
Nombre	Añadir tarea
Descripción	El usuario crea una tarea indicando su nombre y la urgencia de esta y se añadirá a la lista de tareas.
Actor	Usuario autenticado
Relación	Notificar
Precondición	El usuario se ha identificado previamente El usuario ha creado o se ha unido a un grupo previamente
Postcondicion	

Referencia	CU19
Nombre	Eliminar tarea

Descripción	El usuario elimina la tarea.
Actor	Usuario autenticado
Relación	
Precondición	El usuario se ha identificado previamente El usuario ha creado o se ha unido a un grupo previamente
Postcondicion	

Referencia	CU20
Nombre	Notificar
Descripción	El gestor de notificaciones envía notificaciones al usuario indicado
Actor	Usuario autenticado
Relación	Salir grupo, Saldar deuda, Unir grupo, Eliminar usuario grupo, Eliminar grupo, Añadir tarea
Precondición	
Postcondicion	

Referencia	CU21
Nombre	Almacenar imagen
Descripción	El gestor de imágenes almacena la imagen en la nube.
Actor	Usuario autenticado
Relación	Registrar cuenta, Editar perfil
Precondición	
Postcondicion	

Anexo C

Código de creación de las tablas de base de datos

```
CREATE TABLE `user` (
  `id_user` int NOT NULL AUTO_INCREMENT,
  `loginEmail` varchar(255) NOT NULL,
  `password` varchar(255) NOT NULL,
  `name` varchar(255) DEFAULT NULL,
  `lastName` varchar(255) DEFAULT NULL,
  `image` varchar(255) DEFAULT NULL,
  PRIMARY KEY (`id_user`),
  UNIQUE KEY `loginEmail` (`loginEmail`)

CREATE TABLE `group` (
  `code_group ` varchar(255)  NOT NULL,
  `name` varchar(255) NOT NULL,
  `description` varchar(255) DEFAULT NULL,
  `groupCode` varchar(255) NOT NULL,
  `id_creatorGroup` int NOT NULL,
  PRIMARY KEY (`id_group`),
  KEY `Group_fk0` (`id_creatorGroup`),
  CONSTRAINT `Group_fk0` FOREIGN KEY (`id_creatorGroup`) REFERENCES
`user` (`id_user`))

CREATE TABLE `task` (
  `id_task` int NOT NULL AUTO_INCREMENT,
  `name` varchar(255) NOT NULL,
  `urgency` int NOT NULL,
  `code_group` varchar(255)  NOT NULL,
```

```
        PRIMARY KEY (`id_task`),

        KEY `Purchase_fk0` (`code_group`),

        CONSTRAINT `Purchase_fk0` FOREIGN KEY (`code_group`) REFERENCES
`group` (`code_group`) ON DELETE CASCADE)


      CREATE TABLE `debt` (

        `id_debt` int NOT NULL AUTO_INCREMENT,

        `value` float NOT NULL,

        `id_user` int DEFAULT NULL,

        `id_spending` int NOT NULL,

        PRIMARY KEY (`id_debt`),

        KEY `Debt_fk0` (`id_user`),

        KEY `Debt_fk1` (`id_spending`),

        CONSTRAINT `Debt_fk0` FOREIGN KEY (`id_user`) REFERENCES `user`
(`id_user`),

        CONSTRAINT `Debt_fk1` FOREIGN KEY (`id_spending`) REFERENCES
`spending` (`id_spending`) ON DELETE CASCADE)


      CREATE TABLE `spending` (

        `id_spending` int NOT NULL AUTO_INCREMENT,

        `name` varchar(255) NOT NULL,

        `value` float NOT NULL,

        `image` varchar(255) DEFAULT NULL,

        `id_user` int NOT NULL,

        `code_group` varchar(255)  NOT NULL,

        PRIMARY KEY (`id_spending`),

        KEY `Spending_fk0` (`id_user`),

        KEY `Spending_fk1` (`code_group `),

        CONSTRAINT `Spending_fk0` FOREIGN KEY (`id_user`) REFERENCES `user`
(`id_user`),

        CONSTRAINT `Spending_fk1` FOREIGN KEY (`id_group`) REFERENCES
`group` (`code_group`) ON DELETE CASCADE)
```


```
CREATE TABLE `user_group` (
  `id_user_fk` int NOT NULL,
  `code_group_fk` int NOT NULL,
  KEY `user_group_fk0` (`id_user_fk`),
  KEY `user_group_fk1` (`code_group_fk`),
  CONSTRAINT `user_group_fk0` FOREIGN KEY (`id_user_fk`) REFERENCES `user` (`id_user`),
  CONSTRAINT `user_group_fk1` FOREIGN KEY (`code_group_fk`) REFERENCES `group` (`code_group_fk`) ON DELETE CASCADE)
```

Anexo D

Documentación de la API REST

1. Endpoints: /api/users

/create:

Descripción: este endpoint se utiliza para crear un nuevo usuario en la base de datos.

Método POST

Parámetros: los parámetros requeridos son los siguientes:

- **`loginEmail`:** email del usuario.
- **`password`:** contraseña del usuario.
- **`name`:** nombre del usuario.
- **`lastName`:** apellidos del usuario.

Respuestas: devuelve un objeto JSON con el usuario recién creado y un token de sesión.

/createWithImage:

Descripción: este endpoint se utiliza para crear un nuevo usuario en la base de datos con una imagen asociada.

Método POST

Parámetros: los parámetros requeridos son los siguientes:

- **`loginEmail`:** email del usuario.
- **`password`:** contraseña del usuario.

- **`name`:** nombre del usuario.
- **`lastName`:** apellidos del usuario.
- **`image`:** archivo con la imagen del usuario.

Respuestas: devuelve un objeto JSON con el usuario recién creado, la url de la imagen subida a firebase y un token de sesión.

/login:

Descripcion: este endpoint se utiliza para autenticar a un usuario.

Método POST

Parámetros: los parámetros requeridos son los siguientes:

- **`loginEmail`:** email del usuario.
- **`password`:** contraseña del usuario.

Respuestas: devuelve un objeto JSON con la información del usuario y un token de sesión.

/update:

Descripción: este endpoint se utiliza para actualizar la información de un usuario existente en la base de datos.

Método PUT

Parámetros: los parámetros requeridos son los siguientes:

- **`loginEmail`:** email del usuario.
- **`password`:** contraseña del usuario.
- **`name`:** nombre del usuario.
- **`lastName`:** apellidos del usuario.
- **`tokenSession`:** token de sesión necesario para utilizar la API.

Respuestas: devuelve un objeto JSON con la información del usuario actualizado.

/updateWithImage:

Descripción: este endpoint se utiliza para actualizar la información de un usuario existente en la base de datos.

Método PUT

Parámetros: los parámetros requeridos son los siguientes:

- **`loginEmail`:** email del usuario.
- **`password`:** contraseña del usuario.
- **`name`:** nombre del usuario.
- **`lastName`:** apellidos del usuario.
- **`image`:** archivo con la imagen del usuario.
- **`tokenSession`:** token de sesión necesario para utilizar la API.

Respuestas: devuelve un objeto JSON con la información del usuario actualizado.

/updateNotificationToken:

Descripción: este endpoint se utiliza para actualizar el token de notificación de un usuario existente en la base de datos.

Método PUT

Parámetros: los parámetros requeridos son los siguientes:

- **`id_user`:** email del usuario.
- **`notificationToken`:** nuevo token de notificación del usuario.

Respuestas: devuelve un objeto JSON con la información del usuario actualizado con su nuevo token de notificación.

/groups/:idgroup:

Descripción: este endpoint se utiliza para obtener todos los usuarios asociados a un grupo.

Método GET

Parámetros: los parámetros requeridos son los siguientes:

- **`id_group`:** ID del grupo del que se quieren obtener los usuarios asociados.

Respuestas: devuelve un objeto JSON con todos los usuarios asociados al grupo especificado.

Endpoints: /api/groups

/create:

Descripción: este endpoint se utiliza para crear un nuevo grupo en la base de datos.

Método POST

Parámetros: los parámetros requeridos son los siguientes:

- **`name`:** nombre del grupo.
- **`description`:** descripción del grupo.
- **`id_creatorGroup`:** ID del usuario creador del grupo.

Respuestas: devuelve un objeto JSON con el grupo recién creado, el identificador del usuario creador del grupo y un código de grupo generado en el servidor.

/join:

Descripción: este endpoint se utiliza para unir a un usuario a un grupo existente.

Método POST

Parámetros: los parámetros requeridos son los siguientes:

- **`id_user`:** ID del usuario que solicita unirse al grupo.
- **`id_group`:** ID del grupo solicitado.

Respuestas: devuelve un objeto JSON con el grupo al que se ha unido el usuario.

/user/:id_user:

Descripción: este endpoint se utiliza para obtener todos los grupos asociados a un usuario.

Método GET

Parámetros: los parámetros requeridos son los siguientes:

- **`id_user`:** ID del usuario del que se quieren obtener los grupos asociados.

Respuestas: devuelve un objeto JSON con todos los grupos asociados al usuario especificado.

/delete/:id_group:

Descripción: este endpoint se utiliza para eliminar un grupo existente de la base de datos.

Método DELETE

Parámetros: los parámetros requeridos son los siguientes:

- **`id_group `:** ID del grupo que se quiere eliminar.

Respuestas: devuelve un objeto JSON con el grupo que se ha eliminado.

/usergroup/delete/:id_user/:id_group:

Descripción: este endpoint se utiliza para eliminar la relación entre un usuario y un grupo existente en la base de datos.

Método DELETE

Parámetros: los parámetros requeridos son los siguientes:

- **`id_user `:** ID del usuario que se quiere eliminar de la relación. con el grupo
- **`id_group`:** ID del grupo del que se quiere eliminar el usuario.

Respuestas: devuelve un objeto JSON con el grupo del que se ha eliminado al usuario.

Endpoints: /api/tasks

/create:

Descripción: este endpoint se utiliza para crear una nueva compra.

Método POST

Parámetros: los parámetros requeridos son los siguientes:

- **`name`:** nombre de la tarea.
- **`urgency`:** grado de urgencia de la tarea.

- **`id_group`:** ID del grupo del que se quiere añadir la tarea.

Respuestas: devuelve un objeto JSON con la tarea recién creada.

/group/id_group:

Descripción: este endpoint se utiliza para obtener todas las tareas asociadas a un grupo.

Método GET

Parámetros: los parámetros requeridos son los siguientes:

- **`id_group`:** ID del grupo del que se quieren obtener las tareas asociadas.

Respuestas: devuelve un objeto JSON con todas las tareas asociadas al grupo especificado.

/delete:

Descripción: este endpoint se utiliza para eliminar una tarea existente de la base de datos.

Método DELETE

Parámetros: los parámetros requeridos son los siguientes:

- **`id_task`:** ID del task que se quiere eliminar.

Respuestas: devuelve un objeto JSON con la task recién eliminada.

Endpoints: /api/spendings

/create:

Descripción: este endpoint se utiliza para crear un nuevo gasto.

Método POST

Parámetros: los parámetros requeridos son los siguientes:

- **`name`:** nombre del gasto.
- **`value`:** valor del gasto.
- **`id_user`:** ID del usuario creador del gasto.
- **`id_group`:** ID del grupo donde se creó el gasto.

Respuestas: devuelve un objeto JSON con el gasto recién creado.

/createWithImage:

Descripción: este endpoint se utiliza para crear un nuevo gasto.

Método POST

Parámetros: los parámetros requeridos son los siguientes:

- **`name`:** nombre del gasto.
- **`value`:** valor del gasto.
- **`image`:** archivo con la imagen del gasto.
- **`id_user`:** ID del usuario creador del gasto.
- **`id_group`:** ID del grupo donde se creó el gasto.

Respuestas: devuelve un objeto JSON con el gasto recién creado y la url de firebase de la imagen guardada.

/group/:id_group/:id_user:

Descripción: este endpoint se utiliza para obtener todos los gastos de un usuario en un grupo.

Método GET

Parámetros: los parámetros requeridos son los siguientes:

- **`id_group`:** ID del grupo en el que se quiere buscar.
- **`id_user`:** ID del usuario del que se quieren obtener los gastos.

Respuestas: devuelve un objeto JSON con todos los gastos del usuario especificado en el grupo.

/delete/:id_spending:

Descripción: este endpoint se utiliza para eliminar un gasto existente en la base de datos.

Método DELETE

Parámetros: los parámetros requeridos son los siguientes:

- **`id_spending`:** ID del gasto que se quiere eliminar.

Respuestas: devuelve un objeto JSON con el gasto que se ha eliminado.

Endpoints: /api/debts

/group/:id_group:

Descripción: este endpoint se utiliza para obtener todas las deudas asociadas a un grupo.

Método GET

Parámetros: los parámetros requeridos son los siguientes:

- **`id_group`:** ID del grupo del que se quieren obtener las deudas asociadas.

Respuestas: devuelve un objeto JSON con todas las deudas asociadas al grupo especificado.

/spending/:id_spending:

Descripción: este endpoint se utiliza para obtener todas las deudas asociadas a un gasto.

Método GET

Parámetros: los parámetros requeridos son los siguientes:

- **`id_spending`:** ID del gasto del que se quieren obtener las deudas ascoiadas.

Respuestas: devuelve un objeto JSON con todas las deudas asociadas al gasto especificado.

/user/:id_user:

Descripción: este endpoint se utiliza para obtener todas las deudas asociadas a un usuario.

Método GET

Parámetros: los parámetros requeridos son los siguientes:

- **`id_user`:** ID del usuario del que se quieren obtener las deudas ascoiadas.

Respuestas: devuelve un objeto JSON con todas las deudas asociadas al usuario especificado.

/delete/:id_user/:id_spending:

Descripción: este endpoint se utiliza para que un usuario pueda saldar una deuda y eliminarla de la base de datos.

Método DELETE

Parámetros: los parámetros requeridos son los siguientes:

- **`id_user`:** ID del usuario del que se quieren obtener las deudas ascoiadas.

- **`id_spending`:** ID del gasto asociado a la deuda

Respuestas: devuelve un objeto JSON con la deuda que se ha eliminado.

2. Errores de la API REST

La API REST devuelve errores en caso de que se produzca algún problema al procesar una solicitud. Los códigos de error y sus significados son los siguientes:

- **400 Bad Request:** La solicitud enviada por el cliente es incorrecta o no está completa.
- **401 Unauthorized:** El cliente no tiene autorización para acceder al recurso solicitado.
- **404 Not Found:** El recurso solicitado no se ha encontrado en el servidor.
- **500 Internal Server Error:** Se ha producido un error en el servidor.