Fineancer, una aplicación web para la gestión integral financiera de gastos grupales y personales

Fineancer, a web application for comprehensive financial management of group and personal expenses

Trabajo de Fin de Grado
Curso 2024–2025

Autor

Jorge Calvo Fernández
Jorge Lumbreras Camps

Director

Ramón González del Campo Rodríguez Barbero

Grado en Ingeniería de Software e Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

Fineancer, una aplicación web para la gestión integral financiera de gastos grupales y personales
Fineancer, a web application for comprehensive financial management of group and personal expenses

Trabajo de Fin de Grado en Ingeniería de Software e Ingeniería Informática

Autor
Jorge Calvo Fernández
Jorge Lumbreras Camps

Director
Ramón González del Campo Rodríguez Barbero

Convocatoria
Junio 2025

Calificación
Jorge Calvo Fernández – *10*
Jorge Lumbreras Camps – *10*

Grado en Ingeniería de Software e Ingeniería Informática
Facultad de Informática
Universidad Complutense de Madrid

26 de Mayo de 2025

Dedicatoria

A nuestras familias, por su apoyo incondicional. A nuestros profesores, por compartir su conocimiento. Y a nuestros amigos, por las risas, llantos y las noches sin dormir. Gracias por acompañarnos en este viaje.

Agradecimientos

Este trabajo no es solo el resultado de meses de esfuerzo, sino de años de aprendizajes, aciertos, errores y evolución personal. Por eso, queremos dedicar estas líneas a quienes nos acompañaron en el camino, más allá del código y los exámenes.

Agradecemos sinceramente al profesorado por su dedicación y por transmitirnos no solo conocimientos, sino también el valor del pensamiento crítico y la perseverancia. A nuestro tutor, gracias por su orientación y disponibilidad en cada etapa de este proyecto.

A nuestras familias, por su apoyo constante y por ser un pilar fundamental incluso en los momentos de mayor exigencia. Y a nuestros compañeros y compañeras, por el compañerismo, la colaboración y todos los momentos compartidos que hicieron este recorrido más llevadero.

Este trabajo es también reflejo de todo lo aprendido dentro y fuera del aula, y a todos los que formaron parte de ello, les estamos profundamente agradecidos.

Resumen

Fineancer, una aplicación web para la gestión integral financiera de gastos grupales y personales

Fineancer es una aplicación web desarrollada para facilitar la gestión de gastos compartidos en diversos contextos sociales y personales. La plataforma se centra en ofrecer una experiencia al usuario intuitiva y transparente, permitiendo la creación de grupos, registro de gastos categorizados, cálculo automático de deudas y establecimiento de presupuestos compartidos. Se caracteriza por una arquitectura cliente-servidor moderna (*Python/FastAPI* y *React/Next.js*) y con un enfoque mobile-first. La aplicación implementa funcionalidades innovadoras como: lectura automática de gastos por imagen, una categorización jerárquica de gastos, sugerencias para presupuestos y visualización gráfica de datos financieros. El proyecto destaca por la implementación de modelos de Inteligencia Artificial LLM (Large Language Model) para resolver diversos problemas, ofreciendo una herramienta completa que fomenta la organización financiera colectiva y personal.

Palabras clave

Finanzas, Gastos, Presupuestos, Aplicación, Next.js, React, Python, FastAPI, IA, LLM.

Abstract

Fineancer, a web application for comprehensive financial management of group and personal expenses

Fineancer is a web application developed to facilitate the management of shared expenses in various social and personal contexts. The platform focuses on offering an intuitive and transparent user experience, allowing the creation of groups, registration of categorized expenses, automatic debt calculation, and establishment of shared budgets. It is characterized by a modern client-server architecture (*Python/FastAPI* and *React/Next.js*) with a mobile-first approach. The application implements innovative features such as: automatic expense reading through image processing, hierarchical expense categorization, budget suggestions, and graphical visualization of financial data. The project stands out for its implementation of LLM (Large Language Model) Artificial Intelligence models to solve various problems, offering a comprehensive tool that promotes collective and personal financial organization.

Keywords

Finance, Expenses, Budgets, Application, Next.js, React, Python, FastAPI, AI, LLM.

Índice

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Fineancer surge como una propuesta para optimizar la gestión de gastos compartidos a través de una aplicación web que prioriza la intuitividad y la transparencia. El proyecto nace a partir de la identificación de limitaciones en las herramientas existentes y plantea una solución que responde a necesidades reales mediante objetivos técnicos y funcionales bien definidos. El plan de trabajo contempla el desarrollo de una plataforma robusta, centrada en la usabilidad, la seguridad y la escalabilidad.

1.1. Motivación

En la vida cotidiana, compartir gastos es una situación habitual en numerosos contextos: viajes en grupo, celebraciones, pisos compartidos, actividades académicas o laborales, entre otros. En muchos de estos casos, mantener un control claro y justo de los gastos puede convertirse en una tarea compleja, especialmente cuando intervienen múltiples personas y transacciones.

Aunque existen aplicaciones destinadas a la gestión de gastos compartidos, muchas de ellas no cubren todas las necesidades reales de los usuarios. La falta de flexibilidad en la división de gastos, la escasa personalización en la gestión de grupos o una experiencia de uso poco clara son algunas de las carencias detectadas en soluciones ya existentes.

En este contexto, surge la motivación de desarrollar una aplicación que facilite la organización y el control de los gastos comunes de una manera más sencilla, accesible y adaptada a diferentes tipos de usuarios y situaciones. Este proyecto responde al deseo de crear una herramienta útil para la vida diaria, que fomente la transparencia y reduzca los conflictos derivados de la gestión económica compartida.

1.2. Objetivos

El objetivo principal de este proyecto es diseñar y desarrollar una aplicación web llamada *Fineancer*, orientada a facilitar la gestión y seguimiento de gastos tanto a nivel personal como grupal. La aplicación se construirá utilizando tecnologías modernas y priorizando la experiencia de usuario, la escalabilidad y la mantenibilidad del sistema.

En concreto, los objetivos específicos del proyecto son los siguientes:

- Analizar el problema real de la gestión de gastos en grupo, identificando los casos de uso más comunes y las limitaciones presentes en herramientas similares existentes.
- Diseñar una estructura funcional de la aplicación, estableciendo las entidades principales (usuarios, grupos, gastos, categorías, etc.) y cómo se relacionan entre sí.
- Permitir la creación, edición y eliminación de grupos, así como la incorporación de nuevos miembros mediante un sistema de invitaciones gestionado por los propios usuarios.
- Implementar un sistema de autenticación de usuarios, que permita el acceso individualizado y seguro a la plataforma, controlando las sesiones de forma persistente y almacenando los datos en servidores.
- Diseñar e implementar un sistema de control de permisos que garantice la integridad y veracidad de los datos registrados, donde cada usuario pueda modificar o eliminar únicamente los datos que él mismo ha registrado.
- Ofrecer un sistema intuitivo para registrar y consultar gastos, incluyendo información relevante para todos los miembros del grupo, como la descripción, importe, participantes y categoría.
- Permitir la categorización de los gastos registrados, de manera que los usuarios puedan clasificar sus transacciones según distintos tipos (como alimentación, transporte o alojamiento), lo que facilita el análisis posterior del uso del dinero y promueve una mejor organización financiera.
- Incorporar un sistema de reparto de gastos, que calcule las deudas finales entre participantes en función de su contribución a cada gasto.
- Incorporar la posibilidad de que los usuarios marquen sus deudas como saldadas, una vez hayan efectuado los pagos correspondientes fuera de la aplicación, con el fin de mantener un registro claro del estado de las obligaciones financieras de cada miembro.
- Desarrollar funcionalidades para definir presupuestos dentro de cada grupo, con el objetivo de establecer un límite de gasto común que sirva como referencia para todos los miembros, con el fin de fomentar la transparencia y la

planificación compartida, ayudando al grupo a mantener un control colectivo sobre el nivel de gasto.

- Diseñar una interfaz de usuario sencilla, clara y adaptable, con especial atención a su funcionamiento en dispositivos móviles, siguiendo un enfoque *mobile-first*.
- Organizar y almacenar la información de manera estructurada y persistente, garantizando que los datos del usuario, grupos y gastos estén disponibles de forma consistente y desde cualquier dispositivo.
- Preparar y realizar el despliegue de la aplicación en un entorno funcional de producción, con una configuración reproducible y estable, y asegurando que el sistema sea accesible desde cualquier red.

Con estos objetivos se busca no solo resolver un problema común de forma eficaz, sino también construir una base tecnológica sólida que pueda ser ampliada en el futuro con funcionalidades como más métodos de invitación, divisiones personalizadas, analíticas de gasto o exportación de informes.

1.3. Plan de trabajo

El desarrollo del proyecto *Fineancer* se estructuró en varias fases metodológicas, diseñadas para asegurar una progresión lógica desde la concepción de la idea hasta la implementación y el despliegue de la aplicación. Este plan de trabajo permitió abordar los objetivos de manera organizada y sistemática:

1. **Fase 1: Investigación y Definición de Requisitos**
   - Análisis exhaustivo de aplicaciones similares existentes (Tricount, Splitwise, etc.) para identificar funcionalidades comunes, fortalezas y debilidades.
   - Definición detallada de los requisitos funcionales y no funcionales de *Fineancer*, basándose en las carencias detectadas y los objetivos del proyecto (como se detalla en el Capítulo 3).
   - Esbozo inicial de la arquitectura del sistema y selección preliminar del stack tecnológico (Python/FastAPI para el backend, React/Next.js para el frontend, PostgreSQL como base de datos).
2. **Fase 2: Diseño del Sistema**
   - Diseño detallado de la arquitectura general: cliente-servidor, API REST.
   - Diseño del modelo de datos y esquema de la base de datos (entidades, relaciones, tipos de datos), como se presenta en la Figura 3.2.
   - Diseño del modelo de seguridad y autenticación (gestión de sesiones, hashing de contraseñas con Argon2, modelo de permisos viewer-owner).

- Diseño de los Data Transfer Objects (DTOs) para la API, tanto para las entidades de la base de datos (Figura 3.3) como para operaciones específicas de la API (Figura 3.4).
- Diseño de la estructura de los endpoints de la API REST (Figura 3.5).
- Diseño de la interfaz de usuario (UI) y experiencia de usuario (UX) con un enfoque *mobile-first*, incluyendo wireframes y mockups de las pantallas principales (Dashboard, Grupos, Gastos, Presupuestos, Perfil).

3. **Fase 3: Desarrollo del Backend**

   - Configuración del entorno de desarrollo del backend (Python, FastAPI, SQLModel, PostgreSQL).
   - Implementación de los modelos de la base de datos y migraciones iniciales.
   - Desarrollo de los servicios y la lógica de negocio para la gestión de usuarios y autenticación (registro, login, logout, gestión de sesiones).
   - Implementación de los controladores y endpoints para la gestión de grupos, miembros, e invitaciones.
   - Desarrollo de la funcionalidad para registrar, modificar, y eliminar gastos, incluyendo la lógica de división de gastos (SplitService).
   - Implementación de la gestión de categorías (jerárquicas) y presupuestos.
   - Desarrollo del sistema de almacenamiento y gestión de imágenes (temporales y permanentes) asociadas a los gastos.
   - Implementación del servicio de sugerencias de gastos a partir de imágenes y sugerencias de presupuestos basadas en el historial, integrando LangChain.
   - Creación de pruebas unitarias y de integración para asegurar la robustez del backend.

4. **Fase 4: Desarrollo del Frontend**

   - Configuración del entorno de desarrollo del frontend (Next.js, React, Tailwind CSS).
   - Desarrollo de los componentes React reutilizables de la interfaz (botones, formularios, listas, modales) basados en Radix UI y estilizados con Tailwind CSS.
   - Implementación de las páginas principales de la aplicación: inicio de sesión/registro, dashboard, gestión de grupos, listado y detalle de gastos, gestión de presupuestos y perfil de usuario.
   - Integración con la API del backend para todas las funcionalidades (CRUD de usuarios, grupos, gastos, categorías, presupuestos, imágenes).
   - Implementación de la lógica de gestión de estado en el cliente y manejo de la autenticación (cookies de sesión).

- Desarrollo de las visualizaciones de datos de gastos y presupuestos con el uso de gráficos.
- Aseguramiento de la responsividad y la experiencia *mobile-first*.
- Implementación de la validación de formularios.

5. **Fase 5: Integración, Pruebas y Refinamiento**
   - Pruebas exhaustivas de la integración entre el frontend y el backend.
   - Pruebas de usabilidad con usuarios potenciales para recoger feedback y realizar ajustes en la interfaz y flujos de navegación.
   - Corrección de errores (bugs) identificados durante las pruebas.
   - Optimización del rendimiento tanto del frontend como del backend.
   - Revisión de la seguridad general de la aplicación.
6. **Fase 6: Despliegue y Documentación**
   - Preparación del entorno de producción (servidor, base de datos, configuración de dominio).
   - Configuración de herramientas para la reproducibilidad del despliegue (Nix, Justfile) e integración continua (GitHub Actions).
   - Despliegue de la aplicación en el servidor de hosting (Hetzner).
   - Pruebas finales en el entorno de producción.
   - Redacción de la memoria final del proyecto, incluyendo la documentación técnica y de usuario.
   - Preparación de la presentación del proyecto.

Este plan de trabajo se concibió como una guía flexible, permitiendo ajustes y solapamientos entre fases según las necesidades y la progresión del desarrollo. El objetivo principal fue mantener un desarrollo iterativo e incremental, enfocado en entregar valor de forma continua y asegurar la calidad del producto final.

Capítulo 2

Estado de la Cuestión

2.1. Aplicaciones similares

Antes de desarrollar *Fineancer*, una de las partes más importantes del proyecto ha sido analizar qué soluciones existen ya en el mercado y cómo las utilizan las personas en su día a día. Hay una variedad bastante amplia de aplicaciones que ayudan a organizar gastos compartidos, pero cada una lo hace con su propio enfoque y diferentes herramientas. Algunas son muy simples, mientras que otras ofrecen más opciones para quienes necesitan llevar un control más detallado. En este apartado se comentan algunos ejemplos de las más conocidas, poniendo énfasis en sus ventajas, sus limitaciones y en cómo nos han servido como punto de partida e inspiración para mejorar la experiencia de usuario y crear nuevas funcionalidades.

2.1.1. *Tricount*

Tricount (Tricount, 2025b) es una de las aplicaciones más descargadas y recomendadas en este campo. Según su sitio web oficial, la aplicación se posiciona como una solución para dividir gastos con otras personas, especialmente en contextos como viajes o actividades puntuales donde hay que ajustar gastos (Tricount, 2025c). La mecánica es bastante directa: cada persona añade lo que ha pagado y a partir de ahí, la app hace los cálculos necesarios para repartir los gastos y mostrar quién debe qué. Una característica que destaca de *Tricount*, documentada en su material promocional (Tricount, 2025a), es que busca minimizar la cantidad de pagos entre miembros, buscando la forma más sencilla de saldar las deudas con el menor número de transferencias. Esta es la principal funcionalidad de la aplicación y lo que la ha hecho tan popular entre usuarios que buscan simplicidad. Sin embargo, no ofrece más funcionalidades aparte de la creación de gastos y la división de pagos entre usuarios de un grupo, lo que la limita cuando se quieren buscar funcionalidades más complejas.

2.1.2. *Splitwise*

Splitwise (Splitwise, 2025a) es otra herramienta muy utilizada, especialmente entre personas que suelen tener gastos comunes de forma regular y prefieren hacer una gestión más a largo plazo. Según la documentación oficial de la aplicación (Splitwise, 2025b), a diferencia de *Tricount*, está más pensada para un uso continuo en el tiempo. En ella se pueden crear grupos que se mantienen activos, donde cada integrante va registrando los gastos a medida que surgen, y la aplicación se encarga de ir actualizando los saldos de cada uno.

Una de sus ventajas es lo visual que resulta todo. Es fácil ver cuánto debe cada persona, a quién y por qué concepto. Esto ayuda a que todos tengan claro su estado de cuenta y evita malos entendidos. Además, permite sincronizar con plataformas externas de pago como *PayPal* o cuentas bancarias (Splitwise, 2025c), lo que facilita el pago directo desde la app.

A nivel de herramientas, según el análisis de características premium (Splitwise, 2025d), esta ofrece una mayor cantidad brindando más posibilidades a los usuarios, aunque la mayoría de herramientas están detrás de una suscripción mensual lo que queda fuera del alcance de muchos usuarios. Entre estas funciones están: búsqueda de gastos, añadir fotos a gastos y visualizar datos con gráficos.

2.1.3. Otras herramientas y enfoques

Más allá de las aplicaciones especializadas, no es infrecuente que los grupos recurran a soluciones más genéricas o incluso manuales para la gestión de gastos compartidos. Las hojas de cálculo (como *Google Sheets* o *Microsoft Excel*) son una opción común, ofreciendo máxima flexibilidad pero requiriendo una configuración manual y disciplina por parte de todos los miembros para mantener los datos actualizados y realizar los cálculos de deudas correctamente. Para situaciones más sencillas o grupos pequeños, a menudo se utilizan simplemente grupos de mensajería instantánea (como *WhatsApp* o *Telegram*) donde se anotan los gastos, aunque este método carece de estructuración, dificulta el seguimiento a largo plazo y puede llevar a errores u olvidos. Estos enfoques, si bien accesibles, evidencian la necesidad de herramientas dedicadas que automaticen y simplifiquen el proceso, especialmente a medida que aumenta el número de transacciones o participantes.

2.2. Comparativa de las soluciones existentes y propuesta de *Fineancer*

Si bien las soluciones existentes como *Tricount* o *Splitwise* cubren necesidades básicas en la gestión de gastos compartidos, analizando el uso de estas aplicaciones se observan algunas limitaciones en la experiencia de uso cotidiana. Existen algunas áreas de mejora que *Fineancer* busca abordar. La propuesta de *Fineancer* se centra en ofrecer un conjunto de funcionalidades que no solo faciliten el registro y la división

Tabla 2.1: Tabla comparativa de funcionalidades en aplicaciones de gestión de gastos.

Funcionalidad	***Tricount***	***Splitwise***	***Fineancer***
División básica de gastos	Sí	Sí	Sí
Minimización de transacciones	Sí	Sí	Sí
Integración con plataformas de pago	No	Sí	No
Categorización de gastos	Básica	Básica/Media	Avanzada
Soporte para subcategorías	No	No	Sí
Presupuestos grupales	No	No	Sí
Sugerencias de gastos (IA por imágenes)	No	No	Sí
Sugerencias de presupuesto (IA por historial)	No	No	Sí
Sistema de invitaciones	Variable/Email	Variable/Email	Códigos únicos por grupo
Adjuntar imágenes a gastos	No	Sí (Premium)	Sí
Acceso Web	Limitado/No	Sí	Sí
Exportación de datos	Limitada	Sí (Premium)	Planificada

de gastos, sino que también aporten valor añadido en la organización de gastos y la planificación.

A continuación, se presenta una tabla comparativa que resume algunas funcionalidades clave y cómo se abordan en las aplicaciones mencionadas y en *Fineancer*:

Las limitaciones identificadas en otras plataformas que *Fineancer* busca superar incluyen:

- **Planificación financiera con presupuestos:** Una carencia notable es la falta de herramientas integradas para la planificación. *Fineancer* aborda esto con la funcionalidad de **presupuestos grupales**, permitiendo a los usuarios establecer límites de gasto y realizar un seguimiento. Además, incorpora un sistema de **sugerencias de presupuesto basadas en IA** que analiza el historial de gastos del grupo para proponer cifras realistas, fomentando una mejor planificación.
- **Entrada de datos inteligente y asistida:** El proceso de registrar gastos puede ser tedioso. *Fineancer* introduce **sugerencias de gastos mediante el análisis de imágenes** (como tickets o facturas) utilizando IA. El usuario puede subir una imagen y el sistema propondrá rellenar campos como el importe, la fecha o incluso la categoría, agilizando considerablemente esta tarea.

Estas características, entre otras detalladas en los objetivos del proyecto (Capítulo 1), buscan posicionar a *Fineancer* como una solución robusta, intuitiva y adaptada a las necesidades reales de los usuarios que gestionan finanzas en grupo, poniendo especial énfasis en la usabilidad, la personalización y el uso inteligente de la tecnología para facilitar la vida financiera compartida.

Capítulo 3

Descripción del Trabajo

3.1. Requisitos de sistema

Los requisitos de sistema definen los criterios que la aplicación debe cumplir para funcionar correctamente y responder a las necesidades de los usuarios. Se dividen en dos grandes categorías: los requisitos funcionales, que especifican las capacidades concretas de la aplicación, como la gestión de usuarios, grupos, gastos y presupuestos; y los requisitos no funcionales, que abarcan aspectos de calidad como la usabilidad, seguridad, rendimiento y escalabilidad. En conjunto, estos requisitos sirven como base para guiar el diseño, desarrollo y evaluación del sistema, asegurando que se alcancen los objetivos propuestos y se ofrezca una experiencia sólida y accesible.

3.1.1. Requisitos funcionales

Los requisitos funcionales definen las capacidades y comportamientos esperados del sistema en función de las necesidades de los usuarios. La aplicación desarrollada debe ser capaz de ofrecer las siguientes funcionalidades:

- **Registro y autenticación de usuarios:** El sistema debe permitir a los usuarios registrarse, iniciar sesión y cerrar sesión de forma segura mediante autenticación basada en sesiones. Cada usuario tendrá acceso únicamente a la información vinculada a su cuenta.
- **Creación y administración de grupos:** Los usuarios podrán crear nuevos grupos. Solo el propietario de un grupo, es decir, el usuario que lo creó, podrá modificar aspectos del mismo, como su nombre, o eliminarlo por completo. Únicamente los miembros de un grupo serán capaces de ver los datos contenidos en él y modificarlos.
- **Gestión de participantes:** El sistema debe permitir que el propietario de un grupo genere un código de invitación, que podrá compartir a las personas que quiere que se unan al grupo. Dichas personas, una vez registradas en la

aplicación, podrán introducir dicho código para unirse automáticamente al grupo. A su vez, los miembros de cualquier grupo podrán abandonarlo en cualquier momento excepto si son el propietario del mismo, en cuyo caso solo podrán eliminarlo directamente.

- **Registro y visualización de gastos:** Los usuarios podrán añadir, editar y eliminar gastos asociados a un grupo. Cada gasto incluirá información asociada como su título, descripción, importe, participantes involucrados en la división, fecha y hora en la que se realizó el gasto, y una categoría de forma opcional.

- **Definición de presupuestos grupales:** Los usuarios podrán establecer presupuestos comunes para cada grupo, de forma que todos los participantes puedan conocer en todo momento cuánto se ha gastado en el mes actual respecto al límite fijado. A diferencia de los gastos, como los presupuestos son información *global* del grupo y no solo registros puntuales referentes a un solo miembro, todos los miembros podrán modificar y eliminar cualquier presupuesto, además de crear nuevos.

- **Clasificación de gastos por categorías:** La aplicación debe permitir a los usuarios definir categorías para cada grupo para organizar los gastos, facilitando la organización y análisis posterior por tipo de gasto. Se incluirá también soporte para subcategorías para permitir la mayor granularidad posible. Al igual que los presupuestos, al tratarse de recursos *globales* y no referentes a ningún usuario en específico, cualquier miembro será capaz de modificar categorías y eliminarlas.

- **División automática de gastos:** El sistema calculará de manera automática la distribución de cada gasto entre los participantes implicados, actualizando en tiempo real las deudas individuales de cada miembro del grupo, y simplificando las transacciones al máximo, de manera que entre cada par de miembros, solo se realice *una transacción como máximo*.

- **Registro de deudas como saldadas:** Los usuarios podrán indicar que han realizado un pago fuera de la aplicación y marcar su deuda como saldada. Esto permite reflejar acuerdos externos sin forzar a los usuarios a utilizar un sistema de pago integrado.

- **Generación de sugerencias para gastos:** El sistema ofrecerá sugerencias automáticas para completar campos al registrar nuevos gastos, basándose en imágenes proporcionadas por el usuario, ya sea de un ticket, una factura, u otros documentos referentes al gasto.

- **Generación de sugerencias para presupuestos:** En el momento de crear un nuevo presupuesto, el sistema tendrá en cuenta gastos pasados, analizando posibles tendencias temporales y otros patrones, para sugerir un presupuesto de ahorro fiel a las necesidades del grupo en cada momento.

3.1.2. Requisitos no funcionales

Los requisitos no funcionales, que definen las características de calidad que debe cumplir la aplicación y sus restricciones en cuanto a su comportamiento general, usabilidad y mantenimiento, son las siguientes:

- **Accesibilidad multiplataforma:** La aplicación debe funcionar correctamente en los principales navegadores web modernos (*Chrome* y *Firefox*) y adaptarse automáticamente a distintos tamaños de pantalla. El diseño debe seguir una estrategia *mobile-first*, priorizando la experiencia en dispositivos móviles.
- **Interfaz centrada en el usuario:** La interfaz gráfica debe ser clara, coherente y fácil de usar. Las acciones principales deben estar accesibles en menos de tres clics, y los flujos deben estar diseñados para minimizar errores del usuario. Deben incluirse mensajes de ayuda, validación de formularios y notificaciones claras.
- **Despliegue reproducible y estable:** El sistema debe permitir un despliegue consistente en distintos entornos mediante herramientas de gestión declarativa. Todas las configuraciones y dependencias deben estar versionadas para garantizar una instalación sin ambigüedades, evitando diferencias entre entornos de desarrollo y producción.
- **Tiempo de respuesta aceptable:** Las operaciones habituales (como registrar un gasto o consultar un saldo) deben ejecutarse en menos de 500 ms en condiciones normales de red. Las operaciones intensivas, como la generación de sugerencias basadas en el historial, podrán tardar hasta 30 segundos como máximo, siempre mostrando retroalimentación de carga al usuario.
- **Persistencia y fiabilidad de los datos:** Los datos introducidos por los usuarios deben guardarse de forma persistente y sin pérdidas ante errores comunes o interrupciones. Cualquier operación que modifique información crítica deberá confirmar su éxito antes de cerrar el flujo.
- **Consistencia visual:** La aplicación debe mantener un estilo visual unificado (tipografía, colores, espaciado y componentes de interfaz), evitando discrepancias entre vistas. El uso de una librería de componentes o un sistema de diseño debe facilitar esta uniformidad.
- **Escalabilidad razonable:** La arquitectura debe permitir incorporar nuevas funcionalidades o adaptarse a más usuarios sin necesidad de reestructurar completamente el sistema. Las decisiones de diseño deben facilitar el crecimiento de la aplicación.

3.2. Diseño del sistema

El diseño del sistema constituye uno de los pilares fundamentales de cualquier aplicación informática. Un diseño bien estructurado no solo facilita la implementa-

ción inicial, sino que también sienta las bases para la evolución del sistema a lo largo del tiempo, permitiendo su adaptación a nuevos requisitos funcionales y su despliegue eficiente en distintos entornos. La arquitectura general, el modelo de datos, el modelo de seguridad y la separación de responsabilidades entre cliente, servidor y base de datos deben diseñarse e integrarse de forma coherente para garantizar un comportamiento predecible, seguro y sostenible del sistema. En las siguientes secciones se detallan los componentes principales del diseño propuesto para la aplicación, incluyendo la arquitectura general, el modelo de autenticación y seguridad, así como la estructura de datos subyacente que da soporte a todas las funcionalidades de la plataforma.

3.2.1. Arquitectura general

La arquitectura general de la aplicación *Fineancer* se ha diseñado siguiendo un modelo cliente-servidor, como se ilustra en la figura 3.1. Esta estructura modular separa las responsabilidades entre el *frontend*, con el que interactúa el usuario, y el *backend*, que gestiona la lógica de negocio y la persistencia de datos.

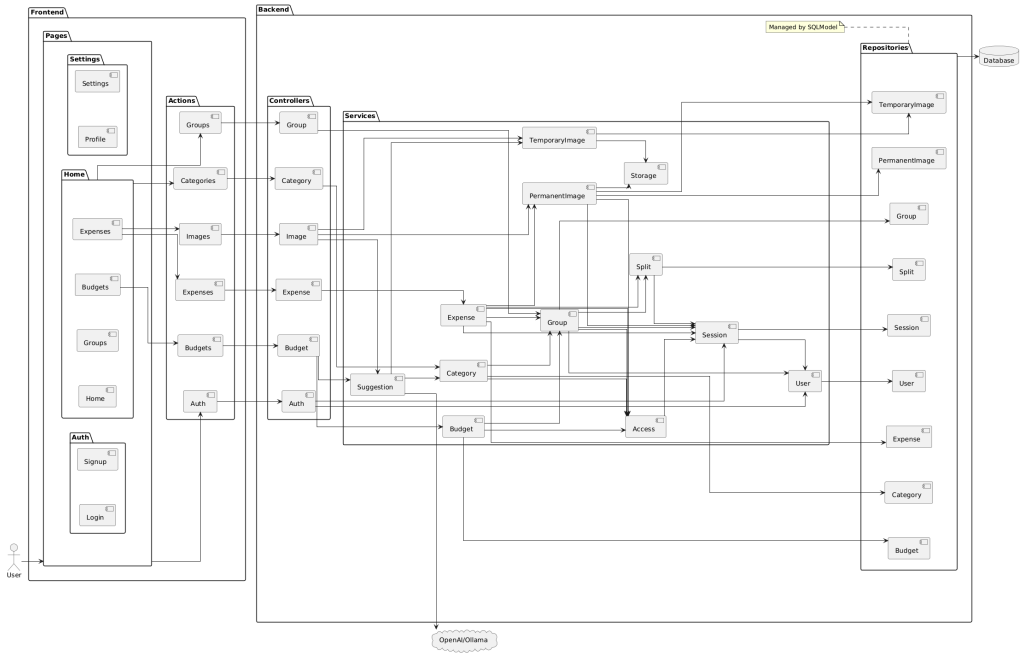

Figura 3.1: Diagrama de la arquitectura general del sistema *Fineancer*.

Como se observa en el diagrama, el **Usuario** interactúa directamente con el **Frontend**, desarrollado con tecnologías web modernas como *Next.js* y *React*.

El *frontend* está compuesto principalmente por las páginas y componentes públicos de acceso al usuario. Estas páginas se comunican con el **Backend** a través de una *API REST*, integrada en las **Actions** existentes para cada una de las entidades de la aplicación.

El *backend*, construido con *FastAPI* y *Python*, está organizado en varias capas:

- **Controladores (Controllers)**: Actúan como punto de entrada para las peticiones *HTTP* del cliente. Validan las entradas y delegan las operaciones a la capa de servicios.
- **Servicios (Services)**: Contienen la lógica de negocio principal de la aplicación. Manejan las interacciones entre los diferentes componentes, como los repositorios y servicios externos. Por ejemplo, el `ExpenseService` gestiona la creación y modificación de gastos, mientras que el `SuggestionService` interactúa con proveedores de IA.
- **Repositorios (Repositories)**: Encapsulan el acceso a la base de datos *PostgreSQL*, utilizando *SQLModel* para la definición de los modelos y la interacción con la base de datos. Esta capa abstrae los detalles de la persistencia de datos de la lógica de negocio.

La **Base de Datos** (*PostgreSQL*) es el sistema de almacenamiento persistente para toda la información de la aplicación, como usuarios, grupos, gastos y presupuestos.

Finalmente, la arquitectura incluye la integración con un **Proveedor de IA externo** (*OpenAI / Ollama*), que es consumido por el `SuggestionService` en el *backend* para ofrecer funcionalidades avanzadas como la generación de sugerencias para gastos y presupuestos.

3.2.2. Modelo de seguridad y autenticación

El diseño del sistema incorpora un modelo de seguridad basado en el principio de mínimos privilegios, cuyo objetivo es restringir las acciones de cada usuario en función de su rol dentro del sistema. Para ello, se emplea un esquema de permisos **viewer-owner**. En este modelo, el *owner* (propietario) es el único con capacidad de modificación sobre un recurso (por ejemplo, un gasto o un grupo), mientras que el resto de usuarios con permisos de *viewer* tienen acceso únicamente de lectura. Esta separación garantiza que los datos registrados por un usuario no puedan ser modificados unilateralmente por otros miembros del grupo, preservando así la integridad de la información. Además, aquellos usuarios que no tengan permisos de *viewer* de un recurso, no podrán consultar la información referente a él, permitiendo que los usuarios puedan compartir datos financieros sensibles, siempre controlando quién puede acceder a ellos.

En cuanto a la autenticación, el sistema se apoya en un mecanismo de *sesiones persistentes*, donde tras una autenticación exitosa, se genera un identificador de sesión asociado al usuario. Este identificador será almacenado como *cookie* en el navegador y, por tanto, será incluido en cada petición posterior al servidor, con la finalidad de validar la identidad del usuario en cada operación que realice. Las sesiones se almacenan en la base de datos y contienen campos como fecha de creación y expiración, lo que permite implementar políticas de caducidad y revocación para incrementar aún más la seguridad y reducir al máximo la superficie de ataque.

Para proteger las credenciales, las contraseñas de los usuarios nunca se almacenan en texto plano. En su lugar, se aplica el algoritmo de hash criptográfico

seguro *Argon2* (Biryukov et al., 2015) en el *backend* durante el proceso de registro y autenticación. La operación de hashing se realiza exclusivamente en el servidor, garantizando que en caso de filtración de la base de datos, un atacante no sería capaz de completar el proceso de autenticación únicamente con los hashes.

Argon2 es una función de hash moderna diseñada específicamente para el almacenamiento seguro de contraseñas. A diferencia de algoritmos más antiguos como *bcrypt* o *MD5*, incorpora mecanismos de resistencia frente a ataques de fuerza bruta realizados con hardware especializado, como GPU o ASIC. Esto lo logra al ser una función *memory-hard*, es decir, que requiere una cantidad significativa de memoria para ejecutarse correctamente, lo cual dificulta la paralelización masiva típica de estos ataques. Gracias a esta característica, *Argon2* ofrece una protección más robusta en contextos donde la seguridad y la resistencia a ataques automatizados son fundamentales, como ocurre en las aplicaciones financieras.

Adicionalmente, desde una perspectiva de seguridad estructural, el sistema ha sido diseñado para **no gestionar pagos reales ni datos bancarios** de los usuarios. Una gran parte de la población es reticente a insertar datos bancarios en plataformas digitales, debido a los problemas que ello pueda ocasionar en caso de una brecha de seguridad. Por ello, las funcionalidades del sistema se limitan estrictamente a la organización y visualización de gastos compartidos, evitando así riesgos innecesarios asociados al tratamiento de información financiera sensible. Esta decisión, no obstante, supone un leve empeoramiento de la experiencia de usuario, ya que el sistema no permite saldar deudas directamente a través de la propia plataforma. En su lugar, los usuarios deben recurrir a una aplicación o medio de pago externo (como *Bizum*, transferencia bancaria, etc.) para realizar la transacción, y posteriormente registrar manualmente el pago en la aplicación. Si bien este proceso introduce pasos adicionales y cierta carga operativa, a cambio reduce considerablemente la superficie de ataque y facilita el cumplimiento de buenas prácticas en materia de seguridad y privacidad de datos.

En conjunto, el modelo de seguridad implementado proporciona un equilibrio adecuado entre funcionalidad, control de acceso y protección de la información, resultando adecuado para una aplicación colaborativa orientada al registro de gastos entre usuarios.

3.2.3. Diseño de la base de datos

El diseño de la base de datos constituye uno de los componentes más fundamentales de la arquitectura de cualquier aplicación, especialmente en aquellas centradas en la colaboración y el intercambio de información, como es el caso de *Fineancer*. Sin una buena estructura de datos, cualquier sistema se vuelve rápidamente complicado y difícilmente mantenible y escalable, por lo que es crucial prestar especial atención a esta parte del diseño de la aplicación. En la figura 3.2 se presenta el *diagrama entidad-relación* (*ERD*) propuesto para la base de datos del sistema.

El modelo ha sido diseñado con el objetivo de lograr una representación fiel de los distintos escenarios de uso dentro de un sistema de gastos colaborativos, así como de

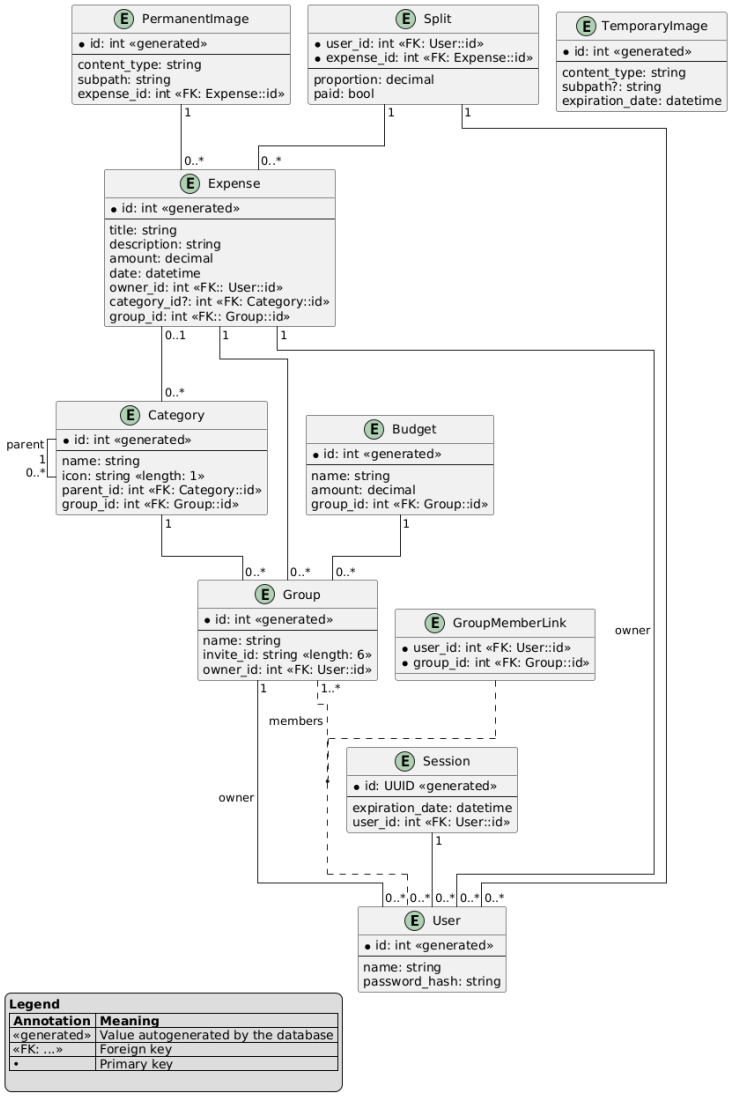

Figura 3.2: Diagrama entidad-relación de la base de datos

facilitar futuras extensiones funcionales. A continuación, se explicarán los detalles más importantes del diseño de la base de datos.

3.2.3.1. Autenticación: usuarios y sesiones

La autenticación de usuarios es un componente esencial en cualquier sistema que maneje datos personales o financieros. En el caso de esta aplicación, se ha optado por un sistema de autenticación basado en **sesiones**, que permite identificar de forma segura a los usuarios en cada interacción con el *backend*, manteniendo el equilibrio entre usabilidad y protección frente a accesos no autorizados.

La entidad `User` representa a cada usuario registrado en el sistema. Su diseño ha sido intencionadamente minimalista, incluyendo únicamente la información esencial para la autenticación y la identificación dentro del sistema. Los campos principales son:

- `id`: Identificador único del usuario.
- `name`: Nombre del usuario, utilizado en la interfaz para identificar a cada participante.
- `password_hash`: *Hash* de la contraseña del usuario. En ningún momento se almacena la contraseña en texto plano.

El sistema no almacena información sensible adicional, como direcciones de correo electrónico o datos personales extensos, ya que estos no son necesarios. Esto reduce la superficie de exposición en caso de incidente de seguridad, respondiendo a un diseño centrado en la privacidad y en el principio de minimización de datos.

Para gestionar el acceso de los usuarios autenticados, se ha implementado un mecanismo de *sesiones persistentes*, representadas mediante la entidad `Session`. Este enfoque consiste en:

- Tras iniciar sesión con credenciales válidas, el servidor crea un objeto `Session` con una `UUID` determinada, que se devuelve al cliente para que pueda identificar al usuario.
- Por seguridad, las sesiones tienen una fecha de expiración de treinta días desde el último acceso a la aplicación. Esto garantiza que, en caso de que algún usuario olvide su sesión iniciada en una máquina ajena, esta se cierre al detectar inactividad, reduciendo la superficie de ataque de la aplicación.

Además, la entidad permite que un mismo usuario disponga de varias sesiones iniciadas al mismo tiempo mejorando las capacidades multiplataforma de la aplicación y la experneica de usuario.

3.2.3.2. Grupos: comunidades privadas de usuarios

La entidad `User` representa a cada persona registrada en la plataforma, constituyendo la base de la comunicación en la aplicación. No obstante, si la información (gastos, presupuestos, etc.) de un usuario fueran visibles por el resto de los usuarios en su totalidad, la aplicación tornaría caótica muy rápidamente. Para organizazr la colaboración entre usuarios, permitiendo que ellos mismos elijan con qué grupo de usuarios se quieren relacionar dentro de la aplicación, se introduce la entidad `Group`, que funciona a modo de contenedor de usuarios que comparten información financiera, restringiendo así el acceso indiscriminado a la información personal de cada comunidad.

La relación *many-to-many* entre usuarios (miembros) y grupos se modela mediante la entidad auxiliar `GroupMemberLink`, lo que permite que:

- Un usuario pueda participar en múltiples grupos simultáneamente, como por ejemplo, referentes a un viaje, un piso compartido, o un evento determinado.

- Un grupo tenga múltiples miembros, que pueden compartir información entre sí.

Para garantizar cierto orden y la integridad en el grupo, se designa un *propietario*, quien creó el grupo, que será el responsable de su administración y el único usuario con los permisos necesarios para modificar la información del grupo, invitar nuevos miembros, o eliminarlo por completo. Este enfoque facilita la organización y administración de grupos, permitiendo modelar desde grupos informales entre amigos, hasta estructuras más formales.

3.2.3.3. Gastos compartidos: trazabilidad y división proporcional

Los gastos (`Expense`) son una de las entidades centrales del sistema, ya que representan las transacciones económicas registradas dentro de un grupo. Cada gasto, además de contener datos clave como un título, la fecha en la que se realizó el gasto, o el importe total, está relacionado con los siguientes datos referentes a la aplicación:

- Un grupo al que pertenece.
- Un usuario que lo crea: el propietario.
- Una categoría opcional que lo clasifica.

El importe de los gastos se modela utilizando el tipo de dato `Decimal` en lugar de `float`, con el objetivo de evitar los errores de precisión inherentes a la representación de números en coma flotante. Este enfoque garantiza una mayor exactitud en las operaciones aritméticas, especialmente en contextos financieros donde los redondeos incorrectos pueden generar inconsistencias acumulativas. Al utilizar `Decimal`, se asegura que los cálculos económicos, como sumas, divisiones o distribuciones de gastos, se realicen de forma precisa y predecible, sin pérdidas de por medio.

Siguiendo una lógica similar a la aplicada en el diseño de los grupos, cada gasto cuenta también con un usuario designado como propietario. Este propietario es el único autorizado para modificar los detalles del gasto, mientras que el resto de los miembros del grupo únicamente pueden visualizarlo. Esta restricción busca preservar la integridad y veracidad de los datos, evitando posibles manipulaciones. Aunque se parte de la premisa de que los miembros de un grupo comparten cierto grado de confianza, de esta forma se previenen situaciones en las que un usuario pueda alterar un gasto registrado por otro para eludir su responsabilidad de pago.

El sistema de gastos compartidos se fundamenta en la idea de que cualquier miembro de un grupo debe poder registrar un gasto común, informando así al resto de participantes de que ha asumido un coste en nombre del grupo. Este enfoque promueve la transparencia y la colaboración, permitiendo que todos los usuarios estén al tanto de los compromisos económicos que se generan dentro del grupo. Además, el sistema garantiza que cualquier miembro pueda consultar en todo momento tanto el estado de sus deudas como su causa. Esta trazabilidad continua favorece una gestión clara, justa y comprensible de los saldos individuales y colectivos.

Además, se introduce la entidad `Split`, que permite modelar la división de un gasto entre varios miembros, indicando qué proporción del total corresponde a cada uno y si ya ha sido pagado o no. Al crear un gasto, el propietario debe especificar cómo se va a realizar la división y qué miembros están involucrados en ella, que no necesariamente tienen que ser todos los miembros del grupo. Esta división estructurada permite, como se explicará en secciones posteriores, calcular deudas y balances entre miembros, para asegurar un reparto justo de las responsabilidades financieras del grupo.

3.2.3.4. Datos auxiliares de grupos: categorización y control presupuestario

La entidad `Category` permite clasificar los gastos para mejorar la organización de los grupos, especialmente en aquellos grandes. Su diseño admite jerarquías en forma de árbol, es decir, categorías con subcategorías, que a su vez pueden dividirse en más subcategorías, como por ejemplo $Transporte \rightarrow Tren$ o $Hogar \rightarrow Comida \rightarrow Supermercado$. Este sistema de categorización permite una organización muy granular de los registros de cada grupo y mejora la claridad y la transparencia de los gastos creados por cada miembro.

Por otro lado, la entidad `Budget` permite asignar límites de gasto por grupo, facilitando el control financiero colectivo. Así, un grupo puede establecer un presupuesto para una actividad y controlar en tiempo real cuánto ha sido gastado, permitiendo que los miembros comuniquen qué expectativas se tienen sobre los gastos del grupo.

3.2.3.5. Imágenes asociadas a gastos

El sistema contempla la posibilidad de adjuntar imágenes a los gastos, útil para almacenar tickets o facturas asociadas al mismo, para aportar mayor trazabilidad y justificación de las transacciones registradas. Para ello, se introducen dos entidades diferenciadas: `PermanentImage` y `TemporaryImage`.

La entidad `PermanentImage` representa imágenes que ya han sido asociadas de forma definitiva a un gasto concreto. Estas imágenes se conservan en el sistema como parte de los datos estructurados del gasto, y pueden ser consultadas por los usuarios con permisos de visualización sobre dicho recurso. Se consideran parte integral de la información registrada.

Por otro lado, la entidad `TemporaryImage` modela aquellas imágenes que han sido subidas por el usuario, pero que aún no han sido vinculadas a ningún gasto. Esta separación es necesaria debido a una funcionalidad avanzada del sistema: el servicio de sugerencias. Este servicio analiza las imágenes asociadas a un gasto y genera automáticamente sugerencias de gasto, incluyendo posibles valores como el importe total, la categoría, o la fecha. No obstante, para que esto sea posible, es necesario permitir la subida de imágenes de forma previa al registro definitivo del gasto, por lo que se almacenan de forma temporal como entidades auxiliares `TemporaryImage` mientras se procesan, y al confirmar la creación del gasto, la imagen se transfiere de

la colección temporal a la permanente.

Este diseño modular permite una experiencia de usuario fluida y facilita la implementación de funcionalidades inteligentes sin incrementar la complejidad de la estructura lógica del modelo de datos principal.

En las dos entidades de imagen, el campo `content_type` almacena el tipo *MIME* (Freed y Borenstein, 1996) de cada imagen, lo que permite identificar su formato de forma estandarizada. Algunos ejemplos serían `image/jpeg`, `image/png` o `image/webp`. Este campo es fundamental para que tanto el *frontend* como el servicio de sugerencias puedan interpretar correctamente el contenido recibido y procesarlo de acuerdo con su tipo.

3.3. Desarrollo del *backend*

El desarrollo del *backend* constituye uno de los pilares fundamentales del sistema, ya que se encarga de gestionar la lógica de negocio, el acceso a los datos y la exposición de servicios a través de una *API*. Además de ser responsable del correcto funcionamiento de las funcionalidades centrales, el *backend* desempeña un papel crucial en la seguridad de la aplicación, asegurando la separación de privilegios entre usuarios y controlando el acceso a los distintos recursos de forma estricta. Esta sección describe las decisiones tecnológicas adoptadas, la estructura y los principales componentes implementados para dar soporte a funcionalidades como la autenticación de usuarios, la gestión de grupos y gastos, y el resto de funcionalidades del sistema. Se ha priorizado el uso de herramientas modernas y buenas prácticas de desarrollo con el objetivo de garantizar un *backend* seguro, escalable y mantenible.

3.3.1. Tecnologías utilizadas

Para el desarrollo del *backend* se ha optado por un conjunto de tecnologías actuales ampliamente utilizadas en la industria, seleccionadas en función de su robustez, comunidad activa, facilidad de integración y adecuación a los objetivos del proyecto.

La base de datos empleada es **PostgreSQL 16** (PostgreSQL Global Development Group, 2023), un sistema de gestión relacional de código abierto que ofrece un excelente rendimiento, soporte para tipos de datos avanzados y extensiones útiles para aplicaciones con requisitos complejos. Su popularidad en el sector de desarrollo de aplicaciones web, su fiabilidad y robustez respecto a la integridad de datos y su política de código abierto la convierten en un candidato idóneo para el sistema.

El lenguaje de programación principal del *backend* es **Python 3.12** (Python Software Foundation, 2024), cuya sintaxis clara y ecosistema rico lo hacen especialmente adecuado para el desarrollo rápido de aplicaciones. Además, su integración nativa con bibliotecas de inteligencia artificial y procesamiento de lenguaje natural fue determinante para incorporar el sistema de sugerencias basado en modelos de lenguaje.

Para la construcción de la *API REST* se ha utilizado el framework **FastAPI**

(Ramirez, 2022a), que destaca por su rendimiento, su compatibilidad con especificaciones OpenAPI y su enfoque en la validación automática a partir de anotaciones de tipo. En conjunto con **SQLModel** (Ramirez, 2022b), una librería que combina las ventajas de `pydantic` (Colvin et al., 2025) y `SQLAlchemy` (Bayer, 2012), se ha logrado un diseño estrictamente tipado, legible y mantenible, lo que mejora la fiabilidad del sistema y facilita el desarrollo orientado a tipos, lo cual ha sido de gran ayuda a la hora de desarrollar el *backend* y de conectarlo con el *frontend*.

El servicio de sugerencias basado en IA se ha implementado con **LangChain** (Chase, 2022), una biblioteca orientada a la creación de aplicaciones que integran modelos de lenguaje. Para facilitar el uso de tanto modelos locales, principalmente para desarrollo y testing, como remotos, para producción, se han utilizado las integraciones **langchain-ollama** y **langchain-openai**.

En cuanto a la seguridad, se ha incorporado la librería **argon2-ffi** (Schlawack, 2015) para el hash seguro de contraseñas. Esta implementación del algoritmo *Argon2* es una de las más famosas en el ecosistema de *Python*.

Por último, se ha empleado **Mypy** (Lehtosalo, 2012) como herramienta de comprobación estática de tipos en *Python*. *Mypy* ha demostrado ser increíblemente útil durante todas las etapas del desarrollo de la aplicación, permitiéndonos detectar errores en las etapas tempranas de desarrollo, asegurando que no se introducían *breaking changes* y manteniendo la coherencia de los tipos a lo largo del código, reforzando la calidad y la mantenibilidad del sistema.

3.3.2. Diseño de las entidades

Exponer directamente todos los datos almacenados en la base de datos a través de la *API* no es una buena práctica por varias razones. En primer lugar, desde el punto de vista de la seguridad, incluir campos sensibles como `password_hash` en las respuestas podría suponer un riesgo grave si no se controla adecuadamente el acceso, ya que facilitaría ataques si estos datos llegaran a manos no autorizadas. Además, enviar todos los campos, incluidos los que no son relevantes para el cliente, puede aumentar innecesariamente el tamaño de los *payloads* de respuesta, afectando al rendimiento de la aplicación, especialmente en entornos móviles o de baja latencia. Por otra parte, mantener una separación clara entre los modelos internos y los datos expuestos permite una mayor flexibilidad para modificar la estructura de la base de datos sin romper la interfaz pública de la *API*, y también evita que se expongan accidentalmente detalles de implementación que deberían permanecer encapsulados. En conjunto, estas consideraciones justifican el uso de modelos específicos para respuesta, llamados *Data Transfer Object* (*DTO*), donde se controla explícitamente qué información se devuelve al cliente.

La figura 3.3 muestra los diferentes *DTOs* definidos a partir del *ERD* presentado en secciones anteriores. En ella se representan, con visibilidad pública, todos los campos que forman parte del contrato de datos expuesto por la *API*. Además, se destacan aquellos campos que, aunque existen en el modelo de base de datos, se consideran privados y están destinados exclusivamente al uso interno del *backend*.

Estos últimos se indican con visibilidad privada, reflejando que no se incluirán en las respuestas del sistema por motivos de seguridad, eficiencia o encapsulamiento.

El diagrama también incluye, para cada entidad, los campos derivados de las relaciones entre tablas de la base de datos, los cuales se representan separados por una línea discontinua. Dichos campos son poblados y gestionados automáticamente por *SQLModel* e internamente, se obtienen a partir de un `JOIN` tradicional. Si una relación tiene visibilidad pública, su correspondiente *DTO* se incorporará como un atributo anidado dentro del *DTO* principal. Por el contrario, si la relación se clasifica como privada, su acceso queda restringido exclusivamente al *backend* y no se incluye en la interfaz pública expuesta por la *API*.

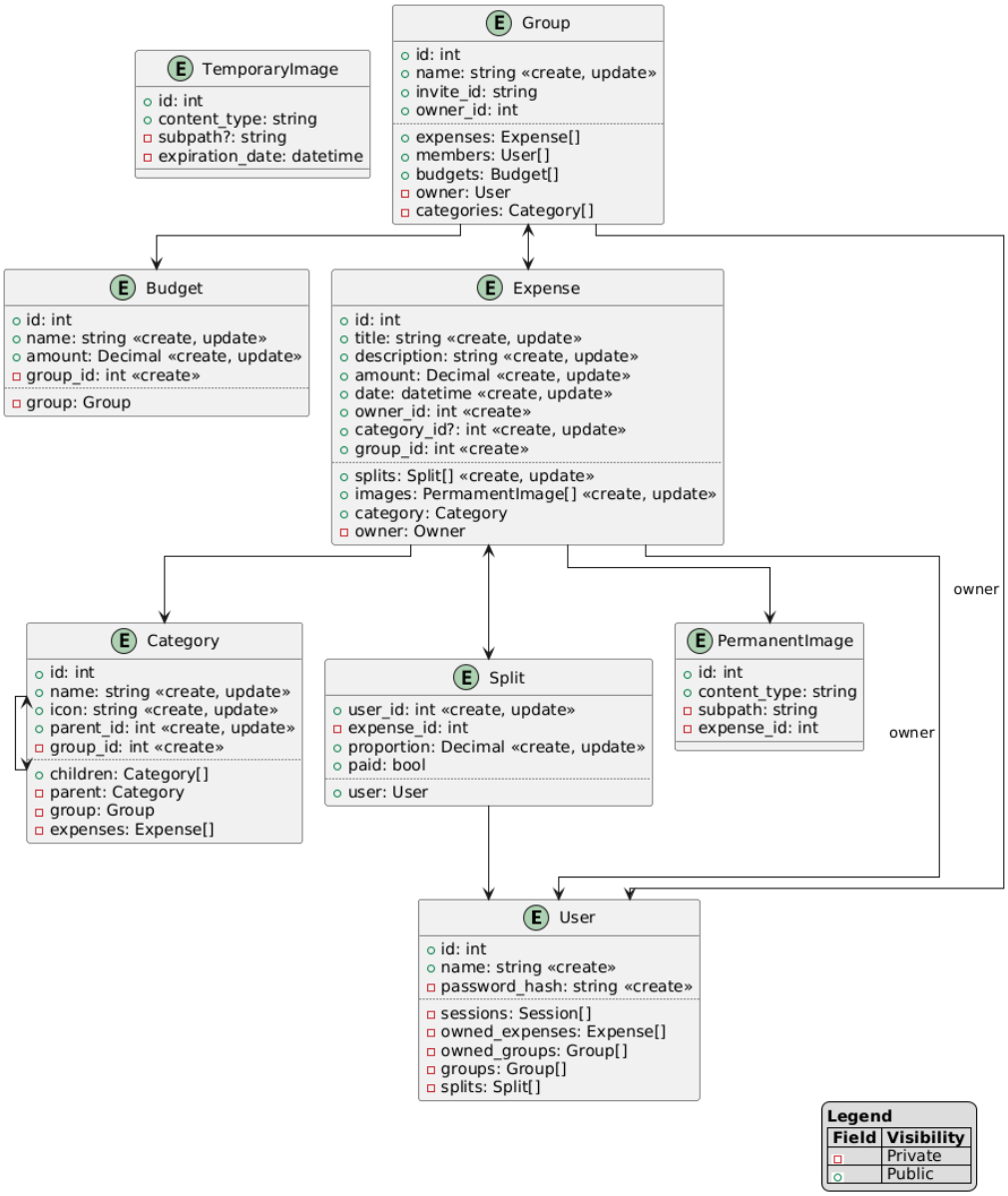

Figura 3.3: *DTOs* relacionados con la base de datos

Además de aquellos *DTOs* directamente relacionados con el modelo de la base

de datos, el sistema también dispone de algunos otros *DTOs* diseñados para facilitar la interacción con la *API*, especialmente en operaciones que no se corresponden de forma directa con las entidades de la base de datos, sino con cálculos no persistentes sobre ellas. Estas estructuras de datos se utilizan principalmente para encapsular la información intercambiada durante determinadas operaciones de negocio, como todas aquellas relacionadas con las sugerencias y las deudas, y para facilitar la validación y transformación de los datos en la capa de servicio.

En la figura 3.4, se describe un diagrama con los *DTOs* utilizados para estas operaciones adicionales, clasificados en tres grupos principales: liquidaciones de deudas (*Settlements*), sugerencias de gastos (*Expense Suggestions*) y sugerencias de presupuestos (*Budget Suggestions*).

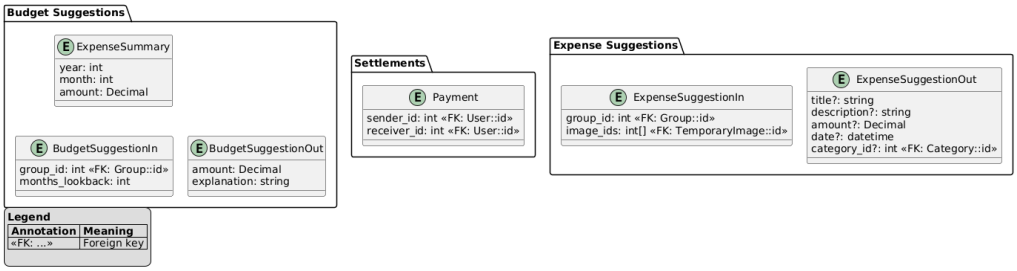

Figura 3.4: *DTOs* relacionados con la *API*

3.3.2.1. Liquidaciones (*Settlements*)

El *DTO* `Payment` representa una transacción económica entre dos usuarios. Aunque en la versión actual de la aplicación se utiliza únicamente para modelar pagos destinados a saldar deudas entre miembros de un grupo, ha sido diseñado pensando en la escalabilidad, de modo que pueda adaptarse a cualquier tipo de pago entre usuarios en futuras ampliaciones del sistema. Este objeto contiene los siguientes campos:

- `sender_id`: Identificador del usuario que realiza el pago.
- `receiver_id`: Identificador del usuario que recibe el pago.
- `amount`: Importe total del pago.

Este *DTO* se utiliza para describir los pagos que debe realizar cada usuario en dos escenarios distintos. En el primero, relacionado con los gastos, se emplea para indicar qué parte del importe total del gasto debe ser asumida por cada participante. En el segundo, vinculado a los grupos, se utiliza en la funcionalidad de liquidación de deudas para enumerar todos los pagos necesarios que permitirían saldar las deudas entre los miembros del grupo.

3.3.2.2. Sugerencias de gastos (*Expense Suggestions*)

Los DTOs relacionados con las sugerencias de gastos permiten al sistema proponer automáticamente nuevos gastos a partir de imágenes, por ejemplo, de un recibo o una factura.

El *DTO* `ExpenseSuggestionIn` se utiliza para recibir datos de entrada en el sistema, con los siguientes campos:

- `group_id`: Identificador del grupo al que se dirige la sugerencia de gasto. Se utiliza para recuperar información contextual del grupo, como las categorías disponibles, relevantes para generar una propuesta coherente con las preferencias y configuraciones del grupo.
- `image_ids`: Lista de identificadores de imágenes temporales, que pueden ser analizadas por el sistema. Las imágenes deben ser temporales ya que a la hora de elaborar una sugerencia de gasto, el gasto aún no ha sido confirmado, por lo que la relación entre este y la imagen aún no ha sido fijada.

El *DTO* `ExpenseSuggestionOut` representa una propuesta de gasto generada como resultado de una petición de análisis. Sus campos son equivalentes a los que se requieren al crear un gasto manualmente, con la excepción de aquellos que el usuario no introduce de forma explícita, como `owner_id` o `group_id`, así como la división del gasto entre participantes, ya que esta información no puede inferirse automáticamente a partir de una imagen de un recibo o factura.

3.3.2.3. Sugerencias de presupuesto (*Budget Suggestions*)

Los DTOs relacionados con las sugerencias de presupuesto permiten al sistema generar recomendaciones automáticas sobre cuánto debería presupuestarse mensualmente para un grupo determinado, basándose en el historial reciente de gastos para generar un objetivo realista de ahorro.

El *DTO* `BudgetSuggestionIn` se utiliza para enviar al sistema los parámetros necesarios para llevar a cabo el análisis y la generación de la sugerencia. Sus campos son:

- `group_id`: Identificador del grupo para el cual se desea obtener una sugerencia de presupuesto. Este valor permite acceder al historial mensual de gastos específico del grupo, dato crucial para la generación del presupuesto.
- `months_lookback`: Número de meses anteriores al momento actual que deben considerarse en el análisis. Este parámetro define la ventana temporal sobre la cual se calcularán los patrones de gasto y, por tanto, influye directamente en la precisión y adecuación de la sugerencia generada. Por ejemplo, un valor de `months_lookback = 0` solo consideraría los gastos del mes actual hasta el día de hoy (incluido) y un valor de `months_lookback = 12` consideraría los gastos del último año, además de aquellos del mes actual.

El *DTO* `BudgetSuggestionOut` encapsula la respuesta generada por el sistema, que consiste en una sugerencia de importe mensual a presupuestar, acompañada de una explicación textual que justifica la recomendación, para dar al usuario más contexto sobre la decisión.

Finalmente, el *DTO* `ExpenseSummary` actúa como una estructura auxiliar utilizada para representar el gasto mensual total de un grupo. Se emplea tanto desde los controladores para proporcionar al usuario un resumen de los gastos por mes, como internamente en el proceso de análisis económico que sustenta las sugerencias de presupuesto. Su objetivo principal en la generación de sugerencias de presupuestos es simplificar la información que se entrega al modelo, facilitando así estimaciones más precisas y eficientes. Este objeto contiene los siguientes campos:

- `year`: Año correspondiente al resumen de gastos.
- `month`: Mes correspondiente al resumen de gastos.
- `amount`: Importe total de gastos del grupo en el período especificado.

3.3.3. Diseño general de controladores y *endpoints*

La primera capa de abstracción del *backend* se estructura mediante controladores, que encapsulan la lógica y la exponen de manera estructurada a través de una *API REST*. Cada conjunto de funcionalidades relacionadas se agrupa bajo rutas comunes, siguiendo una estructura de árbol, lo que facilita su organización, reutilización y comprensión. Esta organización modular permite separar las responsabilidades del sistema, al mismo tiempo que asegura una clara correspondencia entre las operaciones del cliente y los servicios del servidor.

Los controladores están definidos como objetos `router` de *FastAPI*, y exponen diversos *endpoints*, siguiendo el patrón *CRUD* (crear, leer, actualizar y eliminar). El código de los controladores constituye una capa fina que se limita a recibir las peticiones HTTP y delegarlas directamente a los servicios correspondientes. Esto es posible gracias al uso de *Pydantic* junto a *SQLModel*, que automatizan la validación de los datos de entrada a partir de las definiciones de las propias clases de los DTOs, lo que permite evitar lógica adicional de validación dentro del propio controlador. Esto reduce la complejidad del código, mejora su legibilidad y facilita su mantenimiento.

En la figura 3.5 se muestra un diagrama de árbol describiendo la organización jerárquica de los distintos *endpoints* definidos en la aplicación. En las siguientes secciones, se explicarán detenidamente cada uno de los controladores.

3.3.4. Gestión de autenticación, usuarios y sesiones (`/auth`)

Este controlador gestiona la autenticación y las sesiones de usuarios. Está formado por los siguientes *subendpoints*.

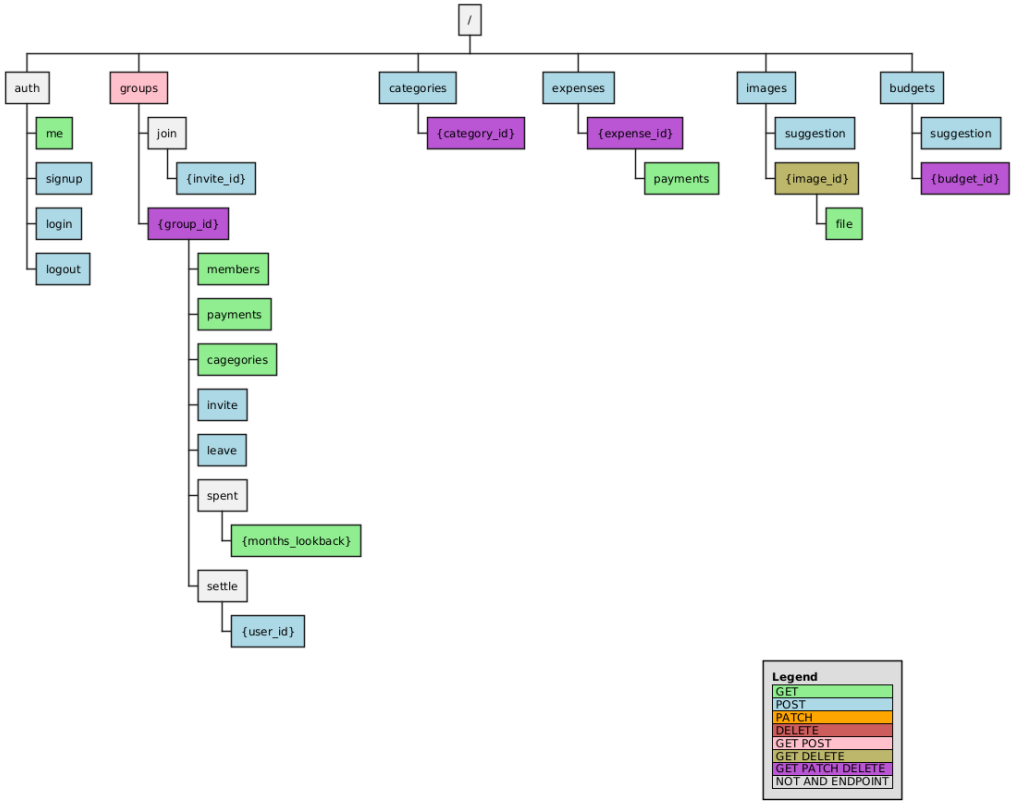

Figura 3.5: Estructura de los *endpoints* de la *API*

- `POST /signup`
- `POST /login`
- `GET /me`
- `POST /logout`

El *endpoint* `POST /signup` permite registrar un nuevo usuario en la aplicación. El *endpoint* recibe la información sobre el nuevo usuario: su nombre de usuario y contraseña, y la transfiere al servicio `UserService`. Este servicio valida que no exista otro usuario con el mismo nombre. Si el nombre ya está en uso, se retorna un error `HTTP 409 (Conflict)`. En caso contrario, se genera un *hash* seguro de la contraseña usando el algoritmo *Argon2* y se añade el nuevo usuario en la base de datos. Finalmente, se devuelve una respuesta con los datos finales del nuevo usuario junto al código `HTTP 201 (Created)`, lo que garantiza un registro seguro y controlado.

El *endpoint* `POST /login` permite iniciar sesión en la aplicación a partir de las credenciales proporcionadas por el usuario, de la misma forma que el endpoint anterior, nombre y contraseña. El proceso es gestionado una vez más por el servicio `SessionService`, el cual primero verifica que no exista ya una sesión activa; en caso contrario, devuelve un error `HTTP 400 (Bad request)`. Si no hay sesión previa, se

delega la autenticación al `UserService`, que valida la existencia del usuario y, en caso de existir, comprueba la contraseña utilizando el algoritmo *Argon2*. Si las credenciales son incorrectas, se devuelve de nuevo un error `HTTP 401 (Unauthorized)`. Una vez autenticado el usuario, se crea una nueva sesión con una expiración de treinta días, se almacena en la base de datos, y se envía una cookie identificadora al cliente, bajo el nombre de (`session_id`). Por último, el *endpoint* retorna los datos del usuario autenticado, permitiendo que el *frontend* los gestione directamente.

El endpoint `GET /me` permite a la aplicación obtener la información del usuario actualmente autenticado. Para ello, utiliza el servicio `SessionService`, el cual gestiona la sesión del usuario a partir de la cookie de sesión previamente almacenada por `GET /login`. Esta cookie se utiliza para recuperar la sesión activa desde la base de datos, verificar si sigue siendo válida (es decir, que exista en la base de datos y que no haya expirado), y en caso de serlo, actualizar su fecha de expiración, para posteriormente devolver los datos del usuario asociado. En caso de que no haya sesión activa o esta sea inválida (ya sea porque haya expirado o porque la cookie `session_id` se haya manipulado), se lanza un error con código `HTTP 401 (Unauthorized)`.

Finalmente, el *endpoint* `POST /logout` permite cerrar la sesión del usuario actualmente autenticado. La lógica está delegada al servicio `SessionService`, obtiene la sesión activa, la elimina de la base de datos y borra la cookie de sesión del cliente (`session_id`). Este proceso asegura que la sesión quede completamente invalidada tanto en el servidor como en el navegador del usuario con el fin de evitar posibles errores. La operación no devuelve ningín dato más que un código `HTTP 204 (No Content)` en caso de ejecutarse correctamente, indicando así que el cierre de sesión se ha realizado satisfactoriamente.

3.3.5. Gestión de presupuestos (`/budgets`)

Endpoints para la gestión de presupuestos:

- `POST /`
- `GET /{budget_id}`
- `PATCH /{budget_id}`
- `DELETE /{budget_id}`
- `POST /suggestion`

El *endpoint* `POST /` permite crear un nuevo presupuesto asociado a un grupo determinado. El controlador delega esta operación en el servicio `BudgetService`, el cual valida que el grupo especificado en el presupuesto exista. En caso afirmativo, se inserta el nuevo presupuesto en la base de datos y se devuelve su información junto a un código de estado `201 Created`. Si el grupo indicado no existe, se lanza un error con código `404 Not Found`.

El *endpoint* `GET /{budget_id}` permite obtener un presupuesto concreto a partir de su identificador. Si el presupuesto existe y el usuario tiene permisos de visibilidad, se devuelve con código `200 OK` en formato `BudgetPublic`. Si el presupuesto no existe, se devuelve un `404 Not Found`. Asimismo, si el usuario no tiene acceso al presupuesto solicitado, se lanza también un error con código `403 Forbidden`, utilizando el mecanismo de control de accesos encapsulado en el `AccessService`.

El *endpoint* `PATCH /{budget_id}` permite actualizar un presupuesto ya existente. El servicio valida la existencia del presupuesto y si existe y el usuario tiene permisos de visibilidad, posteriormente actualiza los datos especificados del presupuesto,. Si la operación es exitosa, se devuelve el presupuesto actualizado con código `200 OK`. En caso de que el presupuesto no exista, se devuelve un `404 Not Found`, y si el usuario no tiene permisos de modificación, se retorna un `403 Forbidden`.

El *endpoint* `DELETE /{budget_id}` elimina un presupuesto de forma permanente. Si el presupuesto existe y el usuario tiene permisos de visibilidad, se ejecuta la operación y se devuelve un código `204 No Content`, indicando éxito sin cuerpo de respuesta. Si el presupuesto no existe, se lanza un `404 Not Found`. Si existe pero el usuario no tiene los permisos adecuados, se devuelve un `403 Forbidden`.

Por último, el endpoint `POST /suggestion` genera una sugerencia de presupuesto personalizada utilizando modelos de lenguaje (*LLMs*), a partir del historial de gastos del grupo. Si la operación es exitosa, se devuelve la sugerencia con código `200 OK`. En caso de error interno durante la generación de la sugerencia (por ejemplo, si falla el modelo de *IA* o no se puede obtener el historial), se lanza una excepción personalizada con código `500 Internal Server Error`.

3.3.6. Gestión de categorías (`/categories`)

Operaciones de gestión de categorías:

- `POST /`
- `GET /{category_id}`
- `PATCH /{category_id}`
- `DELETE /{category_id}`

El *endpoint* `POST /` permite crear una nueva categoría, que debe estar asociada a un grupo existente. Si se especifica una categoría padre (`parent_id`), dicha catagoría también debe existir. El servicio `CategoryService` valida la existencia del grupo y de la categoría padre (si procede), antes de insertar el nuevo registro en la base de datos. Si todo es correcto, se devuelve la categoría creada con código `201 Created`. Si el grupo o la categoría padre no existen, se lanza un error con código `404 Not Found`.

El *endpoint* `GET /{category_id}` recupera una categoría a partir de su identificador. Si la categoría existe y es visible para el usuario autenticado, se devuelve

con código `200 OK` en formato `CategoryPublic`. Si no existe, se lanza un error con código `404 Not Found`. Si el usuario no tiene permisos para visualizar la categoría, se lanza un error con código `403 Forbidden`, gestionado por el servicio de control de acceso `AccessService`.

El *endpoint* `PATCH /{category_id}` permite actualizar una categoría. Antes de aplicar los cambios, se valida que la categoría objetivo y, si aplica, su nueva categoría padre existen. Si la operación se realiza correctamente, se devuelve la categoría actualizada con código `200 OK`. Si la categoría o su nuevo padre no existen, se lanza un error con código `404 Not Found`. Si el usuario no tiene permisos de visibilidad, se lanza un `403 Forbidden`.

El endpoint `DELETE /{category_id}` elimina una categoría de un grupo determinado. Si la categoría existe y el usuario tiene permisos de eliminación, se borra de la base de datos y se devuelve un código `204 No Content`, sin cuerpo en la respuesta. Si la categoría no existe, se devuelve un `404 Not Found`, y si el usuario no tiene los permisos de visibilidad necesarios, un `403 Forbidden`.

3.3.7. Gestión de gastos (/expenses)

Endpoints de gestión de gastos y pagos asociados:

- `POST /`
- `GET /{expense_id}`
- `PATCH /{expense_id}`
- `DELETE /{expense_id}`
- `GET /{expense_id}/payments`

El *endpoint* `POST /` permite crear un nuevo gasto asociado a un grupo y, opcionalmente, a una categoría existente. El servicio `ExpenseService` valida la existencia y visibilidad del grupo y, si procede, también las de la categoría. El gasto se crea con el usuario autenticado como propietario y puede incluir imágenes temporales, que se convertirían en permanentes al pasar a estar asociadas con un gasto. También se crean las divisiones de gasto (`Splits`) asociadas según las proporciones especificadas desde el *frontend*. Si todo es correcto, se devuelve el gasto creado con código `201 Created`. Si el grupo o la categoría no existen, se lanza un error con código `404 Not Found`. Si el usuario no tiene visibilidad sobre el grupo o la categoría, se retorna un `403 Forbidden`.

El *endpoint* `GET /{expense_id}` recupera un gasto existente por su identificador. Si el gasto existe y es visible para el usuario, se devuelve con código `200 OK` en formato `ExpensePublic`. Si el gasto no existe, se lanza un `404 Not Found`. Si existe pero el usuario no tiene permisos de visibilidad, se lanza un error con código `403 Forbidden`.

El *endpoint* `PATCH /{expense_id}` permite modificar parcialmente un gasto. El servicio valida la existencia del gasto, la autoría (el usuario debe ser el creador) y la visibilidad de la nueva categoría (si se ha especificado). También puede actualizar las divisiones del gasto si se proporcionan. Si la operación tiene éxito, se devuelve el gasto actualizado con código `200 OK`. Si el gasto no existe, se lanza un `404 Not Found`. Si el usuario no es el propietario del gasto o no tiene acceso a la categoría, se lanza un error con código `403 Forbidden`.

El *endpoint* `DELETE /{expense_id}` elimina un gasto existente. Si el gasto existe y el usuario autenticado es su propietario (es decir, quien lo creó originalmente), se borra de la base de datos y se devuelve un código `204 No Content`. En caso contrario, se lanza un error `404 Not Found` si el gasto no existe, o un `403 Forbidden` si el usuario no es el propietario.

El *endpoint* `GET /{expense_id}/payments` devuelve los pagos calculados para un gasto, a partir de la lógica de reparto gestionada por el servicio `SplitService`. Si el gasto existe y el usuario tiene visibilidad, se devuelven los pagos con código `200 OK`. Si el gasto no existe, se retorna un `404 Not Found`. Si existe pero no es accesible por el usuario, se lanza un `403 Forbidden`.

3.3.8. Gestión de grupos (`/groups`)

Funcionalidades de gestión de grupos, invitaciones y miembros. Es el controlador más grande de todo el sistema, dada su complejidad de administración de miembros, invitaciones y permisos.

- `POST /`
- `GET /`
- `GET /{group_id}`
- `PATCH /{group_id}`
- `DELETE /{group_id}`
- `GET /{group_id}/members`
- `GET /{group_id}/payments`
- `GET /{group_id}/categories`
- `GET /{group_id}/spent/{months_lookback}`
- `POST /{group_id}/settle/{user_id}`
- `POST /{group_id}/invite`
- `POST /join/{invite_id}`
- `POST /{group_id}/leave`

El *endpoint* `POST /groups/` permite a un usuario crear un nuevo grupo. Para ello, se debe enviar un objeto con los datos necesarios, como el nombre del grupo. El usuario autenticado se convierte automáticamente en el propietario del grupo y se le añade como miembro inicial.

Mediante el *endpoint* `GET /groups/`, se obtienen todos los grupos a los que pertenece el usuario autenticado. Internamente, se realiza una búsqueda de los grupos asociados al identificador del usuario mediante la tabla intermedia de `GroupMemberLink`, devolviendo únicamente aquellos a los que el usuario tiene acceso.

El *endpoint* `GET /groups/{group_id}` permite obtener los detalles de un grupo concreto a partir de su identificador. Si el grupo no existe, se devuelve un error `404 Not Found`. Además, si el usuario no es miembro del grupo, se devolverá un error `403 Forbidden` indicando que no tiene permiso para acceder a dicha información.

A través del `endpoint PATCH /groups/{group_id}`, el propietario de un grupo puede actualizar sus atributos, como el nombre. Si el grupo no existe, se devuelve un error `404 Not Found`. Si el usuario no es el propietario, se retorna un error `403 Forbidden`.

El *endpoint* `DELETE /groups/{group_id}` elimina de forma permanente un grupo. Solo el propietario puede llevar a cabo esta operación. Si el grupo no existe o el usuario no tiene los permisos necesarios, se devolverán errores `404 Not Found` o `403 Forbidden`, respectivamente.

`GET /groups/{group_id}/members` devuelve únicamente la lista de miembros del grupo indicado. Si el grupo no existe, se devuelve un error `404 Not Found`. Si el usuario no pertenece al grupo, se lanza un error `403`. Este endpoint es útil cuando se desea obtener únicamente los usuarios, sin los demás datos específicos del grupo.

Con el *endpoint* `GET /groups/{group_id}/payments`, se calculan los pagos mínimos necesarios para saldar todas las deudas entre los miembros del grupo, mediante un algoritmo de grafos que se explicará más detenidamente en las secciones siguientes. Si el grupo no existe, se devuelve un error `404 Not Found`. Si el usuario no tiene acceso al grupo, se retorna un error `403 Forbidden`. Este endpoint es esencial para facilitar el equilibrio de cuentas entre los participantes, y es uno de los puntos clave de nuestra aplicación.

El *endpoint* `GET /groups/{group_id}/categories` devuelve las categorías asociadas al grupo. Estas pueden estar organizadas jerárquicamente mediante relaciones padre-hijo. Devuelve una coleción de árboles de categorías para modelar correctamente las relaciones de anidación y facilitar el recorrido y representación de las categorías en el frontend. Si el grupo no existe o no es visible para el usuario, se devuelven errores `404 Not Found` o `403 Forbidden`, respectivamente.

`GET /groups/{group_id}/spent/{months_lookback}` proporciona un resumen de los gastos del grupo en los últimos meses. El parámetro `months_lookback` indica cuántos meses en el pasado se desea considerar, además del mes actual. Si el grupo no existe o no es accesible por el usuario, se devuelven errores `404 Not Found` o `403 Forbidden`. Este *endpoint* es útil para que el usuario tenga una representación clara de la evolución temporal de los gastos tanto en el mes actual como durante los

meses anteriores, y es crucial para la generación de sugerencias de presupuestos.

El endpoint `POST /groups/{group_id}/settle/{user_id}` permite marcar como saldadas todas las deudas entre el usuario autenticado y otro miembro del grupo. Si no existe ninguna deuda entre ellos, se lanza un error `400 Bad Request`. Si el grupo no existe o el usuario no tiene acceso, se devuelven errores `404 Not Found` o `403 Forbidden`. Además, si el usuario especificado no existe, también se lanza un error `404 Not Found`.

Mediante `POST /groups/{group_id}/invite`, se genera o regenera un código de invitación único asociado al grupo. Solo el propietario del grupo puede realizar esta operación. Si el grupo no existe o el usuario no es el propietario, se devolverán errores `404 Not Found` o `403 Forbidden`, respectivamente. El código consiste en seis carácteres alfanuméricos permitiendo únicamente mayúsculas, con el fin de facilitar su lectura.

Finalmente, el *endpoint* `POST /groups/join/{invite_id}` permite a un usuario unirse a un grupo utilizando un código de invitación válido. Si el código no corresponde a ningún grupo, se lanza un error `404 Not Found`. Si el usuario ya es miembro del grupo, se devuelve un error `400 Bad Request`. En caso contrario, se añade al usuario al grupo y se le da acceso.

El *endpoint* `POST /groups/{group_id}/leave` permite a un miembro abandonar un grupo. Sin embargo, si el usuario autenticado es el propietario del grupo, se lanza un error `400 Bad Request` que indica que el propietario no puede abandonar el grupo, ya que debría eliminarlo en su lugar. También se retornan errores `403 Forbidden` y `404 Not Found` si el grupo no existe o no es accesible.

3.3.9. Acceso a recursos protegidos

El acceso a recursos compartidos de la aplicación es gestionado por el servicio `AccessService`, que actúa como un sistema de control de acceso. Su función principal es garantizar que los usuarios solo puedan interactuar con los recursos (grupos, gastos, categorías, etc.) para los cuales tienen los permisos adecuados, implementando así un modelo de seguridad basado en roles y relaciones.

La diferencia fundamental entre **owner** (propietario) y **viewer** (visualizador) radica en el nivel de control sobre el recurso:

- El *owner* es el creador o responsable principal del recurso y tiene derechos completos sobre él (incluyendo modificación y eliminación)
- El *viewer* tiene permisos limitados, generalmente solo de lectura o uso controlado del recurso. En aquellos casos en los que el recurso no tiene un propietario definido, todos los visualizadores tienen permisos plenos sobre el recurso.

En este sistema, los diferentes recursos manejan estos conceptos de forma particular:

- **Grupos, gastos e imágenes**: Tienen un *owner* claro, el creador, y *viewers* (miembros del grupo).
- **Categorías y presupuestos**: Al considerarse recursos no cruciales y comunitarios en vez de personales (como podría ser un gasto o un grupo), todos los miembros de un mismo grupo tienen permisos plenos sobre ellos. Solo se implementa el concepto de visibilidad (acceso para miembros del grupo) y todos los miembros son capaces de modificar y eliminar estos recursos.

La finalidad general del servicio es implementar una **división clara de permisos** que:

- Protege la integridad de los datos previniendo accesos no autorizados.
- Permite y facilita el trabajo colaborativo dentro de grupos controlados, sin restricciones innecesarias que lo dificulten.
- Proporciona retroalimentación clara cuando se violan las políticas de acceso.
- Centraliza la lógica de permisos para mantener consistencia en toda la aplicación

Este enfoque sigue el principio de mínimo privilegio, donde cada usuario recibe solo los accesos estrictamente necesarios para su funcionamiento, mejorando tanto la seguridad como la experiencia de usuario al evitar accesos a información irrelevante.

3.3.10. Gestión de imágenes (`/images`)

Endpoints para el manejo de imágenes temporales y permanentes, junto a la gestión y guardado de archivos.

- `POST /`
- `POST /suggestion`
- `GET /{image_id}`
- `GET /{image_id}/file`
- `DELETE /{image_id}`

El *endpoint* `POST /` permite a los usuarios subir imágenes temporales que serán utilizadas como base para sugerencias de gastos. Este servicio está diseñado específicamente para el procesamiento inicial de imágenes, donde los archivos tienen una vida limitada de un día antes de ser eliminados automáticamente por una tarea en segundo plano. El *endpoint* valida estrictamente los archivos recibidos, verificando que sean imágenes reales (mediante el *MIME Type*), que tengan nombre y que

cumplan con los requisitos básicos. Cada imagen subida se almacena por medio del `StorageService` con un nombre aleatorio para evitar conflictos y se registra en la base de datos con su fecha de expiración, proporcionando un mecanismo temporal para el procesamiento inicial sin comprometer el almacenamiento permanente. Este mecanismo de imágenes temporales es necesario para poder generar sugerencias de gastos a partir de ellas, sin necesariamente haber creado ningún gasto; supone una división clara entre los conceptos de aquellas imágenes que están asociadas a un gasto, y aquellas que aún no han sido procesadas totalmente.

El *endpoint* `POST /suggestion` se encarga de generar sugerencias de gastos a partir de las imágenes previamente cargadas. Este realiza un procesamiento inteligente para extraer información relevante como importes, posibles categorías u otros detalles del gasto representado en la imagen. Esta funcionalidad la lleva a cabo el servicio de `SuggestionService` via *LLMs*.

Para consultar información sobre imágenes ya procesadas y guardadas permanentemente, el *endpoint* `GET /{image_id}` proporciona los metadatos de una imagen específica. Este servicio verifica primero que la imagen exista y luego aplica controles de acceso mediante el `AccessService`, asegurando que solo usuarios con permisos adecuados (miembros del grupo relacionado al gasto) puedan ver la información. El *endpoint* devuelve datos estructurados sobre la imagen pero no el archivo en sí, siendo útil para interfaces que necesitan mostrar información sobre la imagen antes de decidir si descargarla, con el fin de ahorrar recursos en dispositivos móviles.

En aquellos casos en los que se requiere el archivo de imagen completo, el *endpoint* `GET /{image_id}/file` permite descargar el contenido binario de la imagen permanente. Al igual que el *endpoint* anterior, realiza verificaciones de existencia y permisos, pero además accede al sistema de almacenamiento `StorageService` para recuperar y transmitir los bytes de la imagen con el *MIME Type* adecuado. Este servicio es esencial para visualizar las imágenes asociadas a gastos en la interfaz de usuario, manteniendo al mismo tiempo todas las restricciones de seguridad.

Finalmente, el endpoint `DELETE /{image_id}` ofrece la capacidad de eliminar imágenes permanentes, pero con importantes restricciones de seguridad. Solo el propietario original de la imagen (quien la subió inicialmente) puede eliminarla, validación que realiza el `AccessService` antes de cualquier operación. El proceso elimina tanto el registro en la base de datos como el archivo físico en el sistema de almacenamiento, manteniendo la consistencia de los datos. Este control estricto evita que usuarios no autorizados puedan borrar evidencia de gastos compartidos en el sistema.

3.3.11. División de gastos y cálculo de deudas

Fineancer implementa un sistema de división de deudas en un grupo, diseñado para calcular y simplificar las transacciones necesarias para equilibrar los gastos compartidos. El proceso consta de dos etapas principales: cálculo de deudas netas y simplificación de pagos, siguiendo un enfoque eficiente que minimiza el número total de transacciones.

3.3.11.1. Análisis y complejidad del problema

El problema que este algoritmo resuelve es *NP-Complete* (Mohan K, 2019). Ello conlleva que su solución óptima no sea obtenible a partir de un algoritmo de complejidad polinomial, por lo que el tiempo de procesamiento necesario podría ascender considerablemente, especialmente en grupos grandes.

No obstante, al tratarse de una operación **no crítica** (es decir, que la diferencia entre la solución óptima y la solución obtenida no tiene un impacto significativo), es preferible optar por una mejor eficiencia y un menor tiempo de ejecución, en vez de por una operación óptima garantizada. Por ello, se ha utilizado un algoritmo voraz, que no garantiza que la solución sea siempre la idónea (es decir, podría existir otro conjunto de menos transacciones que también resolviera el sistema de deudas) pero optimiza considerablemente el cálculo.

3.3.11.2. Cálculo de deudas netas

El algoritmo comienza calculando las deudas netas entre cada par de usuarios en el grupo. Para cada gasto (`Expense`), se consideran las divisiones (`Split`) no pagadas, donde cada división representa la proporción del gasto que un usuario debe al propietario (pagador) del gasto. La clave del proceso está en normalizar las relaciones entre usuarios:

- **Normalización de pares**: Los pares de usuarios se ordenan de forma que el propietario siempre tenga el *ID* menor (ejemplo: si el propietario tiene ID 5 y el deudor ID 3, se registra como $3 \rightarrow 5$ y se adecúa el signo del importe). Esto evita duplicados, ya que para la calcular la deuda no es necesario calcular tanto la dirección $3 \rightarrow 5$ como la $5 \rightarrow 3$.
- **Cálculo del importe neto**: Para cada par, se suma el total adeudado entre todas las divisiones de gastos, considerando la dirección de la deuda (el signo se ajusta según quién es el propietario del gasto y quién el deudor). La fórmula aplicada es:

  $$net = \sum direction \cdot amount \cdot proportion$$

  Donde *direction* es positivo si el *ID* del propietario es menor que el del deudor, y negativo en caso contrario.

3.3.11.3. Simplificación de Pagos

Una vez calculadas las deudas brutas entre pares, el algoritmo optimiza las transacciones mediante un enfoque voraz (*greedy*):

- **Balances netos**: Primero, se calcula el balance neto para cada usuario (diferencia entre lo que debe y lo que le deben). Los usuarios se clasifican en deudores, con balance total negativo y acreedores, con balance positivo.

- **Asignación óptima**: Se emparejan deudores con acreedores, liquidando la deuda con el importe máximo posible en cada transacción. Para ello, se toma el deudor con la mayor deuda y el acreedor con el mayor crédito, se transfiere la cantidad mínima entre la deuda y el crédito y se actualizan los balances. El proceso se repite hasta que todas las deudas se liquidan.

3.3.11.4. Ejemplo Práctico

Supongamos tres usuarios y el siguiente escenario, en el cual:

- A debe \$50 a B y \$30 a C.
- B debe \$20 a C.

Representado de forma gráfica, la distribución de las deudas en el grupo sería el mostrado en la figura 3.6.

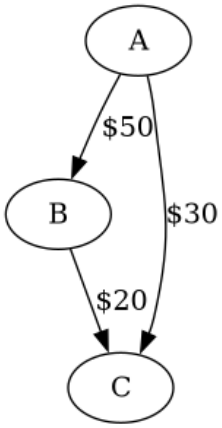

Figura 3.6: Deudas entre usuarios del ejemplo práctico

El algoritmo calcula balances netos: A ($-$\$80), B (\$30), C (\$50), y posteriormente simplifica pagos: A paga \$50 directamente a C y \$30 a \$$B$, eliminando la necesidad de múltiples transacciones intermedias, tal y como se muestra en la figura 3.7.

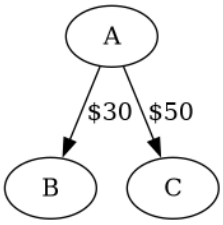

Figura 3.7: Pagos a realizar para saldar deudas en el ejemplo práctico

3.3.12. Generación de sugerencias: autocompletado de gastos y presupuestos futuros

Una característica destacada de *Fineancer* es la capacidad de ofrecer sugerencias inteligentes para facilitar la introducción de datos, mejorando la experiencia de usuario. Esta funcionalidad se implementa a través del `SuggestionService`, desarrollado con la biblioteca *LangChain* (Chase, 2022), que facilita la integración con modelos de lenguaje.

El servicio está diseñado con una arquitectura modular que se apoya en servicios auxiliares como `GroupService`, `CategoryService` y `TemporaryImageService`. Esta estructura permite el funcionamiento con diferentes proveedores de modelos, ya sea `Ollama` para desarrollo local o `OpenAI` para entornos de producción, adaptándose a distintos escenarios de despliegue sin modificar la interfaz.

Para el proveedor de *OpenAI* se evaluaron modelos como `gpt-4o` (OpenAI, 2024), `gpt-4.1-nano` (OpenAI, 2025a) y `gpt-4o-mini` (OpenAI, 2025b), eligiendo finalmente este último por su bajo coste y buen rendimiento, con tiempos de respuesta rápidos y resultados sólidos en tareas de visión. En el caso de *Ollama*, se consideraron modelos como `gemma3` (Google, 2025) y `mistral-small3.1` (MistralAI, 2025), pero fue `llama3.2-vision` (MetaAI, 2024) en su versión de **once mil millones de parámetros** el que ofreció el mejor desempeño en las pruebas, destacando por su precisión en tareas de visión.

Para las sugerencias de gastos basadas en imágenes, el sistema sigue un proceso estructurado: el usuario sube imágenes de tickets o facturas, que se almacenan temporalmente; luego, al solicitar una sugerencia, el sistema recopila tanto las imágenes como el árbol de categorías del grupo y utiliza una cadena de procesamiento (*chain*) para analizar la información visual. Esta cadena genera una estructura de datos `ExpenseSuggestionOut` con campos como título, descripción, importe, fecha y categoría, validando finalmente la existencia de la categoría sugerida.

En el caso de los presupuestos, el sistema analiza el historial de gastos del grupo durante el periodo especificado mediante `months_lookback`. Empleando una cadena similar, el modelo identifica patrones, tendencias estacionales y recomienda un presupuesto óptimo, incluyendo una explicación textual del razonamiento seguido. La implementación utiliza el objeto `ExpenseSummary` para representar de forma simplificada la información mensual.

Las sugerencias generadas se integran en la interfaz de usuario: para gastos, autocompletando formularios que el usuario puede ajustar; para presupuestos, presentando recomendaciones justificadas que facilitan la toma de decisiones informadas. Este enfoque híbrido combina automatización y control humano, agilizando la introducción de datos sin comprometer la precisión.

3.4. Desarrollo del *frontend*

Una de las partes más importantes que hemos tenido en cuenta para el desarrollo de la aplicación, ha sido el desarrollo del *frontend*. Nuestra prioridad ha sido la de facilitar la interacción directa del usuario sobre las funciones y herramientas que ofrecemos en el *backend*, ya que se busca crear una aplicación accesible y de uso rápido en el día a día de los usuarios, amenizando el trabajo a realizar para cumplir todas las acciones que realicen los usuarios. Por ello, el desarrollo se ha centrado en crear una experiencia de usuario fluida, intuitiva y eficiente.

3.4.1. Tecnologías utilizadas

El *frontend* de la aplicación ha sido desarrollado con diferentes tecnologías web modernas que permiten una construcción más eficiente y accesible. Las principales tecnologías utilizadas el el proyecto han sido:

- **Next.js 15**: *Framework* basado en *React* que permite la carga de datos desde el servidor (SSR) y generación de paginas estáticas (SSG), ideal para aplicaciones web con buen rendimiento y con diseño web adaptable (Inc., 2024b).
- **React 19**: Biblioteca de *JavaScript* para la construcción interfaces de usuario basadas en componentes (Inc., 2024a).
- **Tailwind CSS**: *Framework* de CSS que facilita la creación de diseños modernos y adaptables a diferentes pantallas con clases prediseñadas (Wathan y Reinink, 2024).
- **Radix UI**: Biblioteca de componentes *React* accesibles y predefinidos usada como base de componentes para la interfaz (WorkOS, 2023).
- **Lucide React**: Biblioteca de iconos SVG integrables en *React*, utilizada para mejorar la interfaz (Contributors, 2023).
- **Zod**: Librería de validación de esquemas en *TypeScript* utilizada para la validación de formularios y datos provenientes de la *API* (Garfield, 2023).
- **Recharts**: Librería de *React* para construir gráficos interactivos y visualizaciones de los gastos del usuario (Group, 2023).

3.4.2. Arquitectura del *frontend*

La arquitectura del *frontend* de *Fineancer* se ha diseñado siguiendo principios de modularidad, reutilización y separación de componentes, con el objetivo de facilitar el desarrollo, las pruebas y el mantenimiento a largo plazo.

- **Enrutamiento Basado en el Sistema de Archivos (*Next.js*)**: *Next.js* gestiona el enrutamiento de la aplicación. Las rutas se definen mediante la estructura de carpetas y archivos dentro del directorio `app` (utilizando el *App Router* de *Next.js* (Inc., 2024b)). Hemos utilizado rutas estáticas para las páginas centrales de la aplicación, y rutas dinámicas para las rutas de acceso a datos únicos de datos sobre entidades, como grupos o gastos.

- **Componentes Reutilizables**: Toda la arquitectura web esta basada en el uso de componentes *React* (Inc., 2024a) pequeños y reutilizables, tanto de librerías de componentes web como *RadixUI* (WorkOS, 2023) para elementos comunes como botones, campos de formularios o tarjetas como de componentes propios usados para mostrar datos de forma dinámica en páginas estáticas de la aplicación. Estos componentes se organizan en directorios específicos para su reutilización y gestión eficiente.

- **Layouts y Plantillas (Layouts en *Next.js*)**: *Next.js* permite definir *layouts* persistentes que envuelven a las páginas. En *Fineancer*, se utilizan *layouts* para estructuras comunes como la barra de navegación principal en la parte inferior, la cabecera de la aplicación, y cualquier otro elemento que deba estar presente en múltiples vistas. Esto evita la duplicación de código y asegura una estructura mas consistente, además de mejorar la eficiencia en carga de la web al solo cargar cambios realizados en las páginas dentro los *layouts*.

- **Servicios y Utilidades**: La lógica que no está directamente ligada a la presentación, como las llamadas a la *API*, el formato de datos, o funciones de validación específicas de los formularios, se encapsulan en módulos de servicio.

3.4.3. Interfaz de Usuario

La interfaz de usuario (UI) de *Fineancer* se ha diseñado con la idea de ser *mobile-first*, priorizando la experiencia en dispositivos móviles, que constituyen el principal punto de acceso para la gestión rápida y cotidiana de gastos compartidos. Este enfoque se debe a la necesidad de hacer una aplicación dirigida a los usuarios de móvil, ya que las funciones están pensadas para realizarse de una forma más amena en el día a día, como la creación de gastos realizados, captura de fotos por los usuarios en el momento y el uso social entre varios usuarios que se busca. Algunos de los elementos que facilitan el uso y el flujo de la navegación han sido el uso de una barra horizontal situada en la parte inferior de la pantalla, presente en gran cantidad de aplicaciones móviles modernas, que ofrece acceso rápido a las secciones principales (*Dashboard*, Gastos, Grupos, Presupuestos) y una barra superior complementaria que gestiona el acceso al perfil del usuario y notificaciones sobre las interacciones del usuario.

3.4.4. Flujo de navegación y pantallas principales

El flujo de la aplicación busca ser intuitivo y que permita al usuario poder acceder a las funcionalidades más importantes de una forma rápida y sencilla. La navegación principal se articula en torno a las siguientes pantallas clave:

1. **Pantalla de Inicio de Sesión / Registro (‘/login‘, ‘/signup‘)**: Es la primera pantalla que visualiza un usuario al entrar a la aplicación, permitiéndole crear una cuenta o entrar con su cuenta creada previamente. Cuenta con dos formularios sencillos que piden nombre de usuario y contraseña. Se incluyen validaciones de entrada y manejo de errores para guiar al usuario.

2. ***Dashboard* Principal (*Home* - ‘/‘)**: Una vez se ha iniciado sesión, la primera pantalla que ve el usuario es la del *dashboard*. Presenta un resumen general de su actividad este último mes en comparación con los meses anteriores, un botón de acceso rápido a la creación de un gasto mediante imagen como recibos, y una lista de los últimos gastos creados en los grupos a los que pertenece.

3. **Pantalla de Gastos (‘/*expenses*‘)**: Muestra un listado detallado de todos los gastos registrados en los grupos del usuario. Permite filtrar los gastos (por grupo y por categoría), ordenarlos y realizar búsquedas. Es posible entrar a cada gasto en la lista para ver sus detalles completos, incluyendo quién participó y cómo se divide.

4. **Pantalla de Grupos (‘/*groups*‘)**: Presenta una lista de todos los grupos a los que pertenece el usuario. Desde aquí, se puede acceder a la vista detallada de cada grupo. También ofrece acciones para crear un nuevo grupo o para unirse a un grupo existente mediante un código de invitación que puede ser compartido por otro usuario miembro de un grupo.

5. **Pantalla de Presupuestos (‘/*budgets*‘)**: Esta pantalla permite a los usuarios visualizar y gestionar los presupuestos definidos para los grupos de los que son miembros. Muestra una lista de presupuestos, su progreso actual (gastado vs. asignado) mediante barras de progreso, y permitir la creación de nuevos presupuestos o la modificación de los existentes, incluyendo la interacción con las sugerencias de presupuesto basadas en IA.

3.4.5. Integración con la *API*

La comunicación entre el *frontend* y el *backend* se realiza a través de una *API RESTful*. En este proyecto se sigue el siguiente enfoque:

1. **Abstracción de peticiones con `fetch`**: Se creó el módulo `frontend/lib/api.ts` que encapsula todas las llamadas HTTP usando `fetch` nativo. Este wrapper:

   - Envía siempre `credentials: ínclude’` para adjuntar la cookie de sesión.

- Añade automáticamente `Content-Type: application/json` y, opcionalmente, un `Authorization` si hubiese un *token*.
- Parsea el *JSON* de respuesta y lanza internamente una `APIError` con código y mensaje extraído de los campos `detail` o `message`.

2. **Server Actions vs. Client Components**:
   - Los accesos a datos del *backend* se implementan en páginas de carga en el servidor (*use server*), aprovechando la carga en el servidor y no en el navegador del usuario, haciendo uso de las *cookies* creadas.
   - La carga inicial de datos o de actualización, se realiza en componentes de cliente o *hooks*, con el uso de funciones como `useEffect` de *React* (Inc., 2024a), `useCurrentUser` o `useGroupData`.
3. **Tipos de datos *TypeScript***: Para cada endpoint se definen interfaces o tipos que reflejan los DTOs del *backend* (por ejemplo, `UserPublic`, `Expense`, `Group`). El módulo de *API* del *frontend* fuerza la consistencia de tipos para evitar problemas de comunicación y uso de datos correctos de las entidades.
4. **Manejo de errores**:
   a) El módulo `api.request` lanza una excepción `APIError` con `status` y `message` basados en la respuesta.
   b) Las *Server Actions* capturan esa excepción y devuelven estados de formulario (errores de validación, mensajes personalizados).
   c) En el cliente, los componentes y *hooks* muestran notificaciones con la librería `sonner` (`toast.error`, `toast.success`, `toast.loading`).
5. **Indicadores de carga**: Se usan estados locales (`loading`, `isLoading`) en *React* (Inc., 2024a) para mostrar mensajes o esqueletos (`Loading...`) mientras se resuelven las peticiones.

3.4.6. Autenticación en el *frontend*

La autenticación del *frontend* sobre usuarios es una de las partes más importantes de la aplicación a nivel de seguridad. Al tratarse de una aplicación web, es necesario gestionar el acceso a las diferentes páginas para evitar un uso indebido y así garantizar la protección de datos y funcionalidades.

Autenticación y Protección de Rutas

- **Flujo de Autenticación:** Al iniciar sesión, el *frontend* envía las credenciales al *backend* mediante una *Server Action*. Si son válidas y el *backend* responde correctamente (estableciendo la *cookie* de sesión a través de la respuesta *HTTP*), el estado de la aplicación se actualiza para reflejar que el usuario está registrado.
- **Autenticación con *cookies***:

  - Al autenticarse mediante `/login`, el *backend* responde con un encabezado `Set-Cookie: session-id` con el *id* unico del usuario *logeado* en la aplicación.
  - Las Server Actions utilizan la API `cookies()` de *Next.js* para extraer y configurar esta *cookie HTTP-only* en el navegador.
  - El módulo de *API* incluye siempre `credentials: ínclude'` para enviar la *cookie* de sesión en todas las peticiones posteriores al *backend*.
  - De este modo, cada petición al *backend* se autentica automáticamente sin exponer *tokens* en el cliente.

- **Persistencia de la Sesión:** La *cookie* de sesión ('session-id'), configurada tras la comunicación con el *backend*, asegura que la sesión del usuario persista entre visitas y recargas de página, hasta que la *cookie* expire o el usuario cierre sesión.

- **Protección de Rutas con *Middleware* de *Next.js*:** Las rutas que requieren autenticación (por ejemplo, '/', '*/groups*', '*/expenses*') se protegen utilizando *middleware* en *Next.js*. El middleware se ejecuta en el servidor antes de que la petición llegue a la página. Verifica que la sesión del usuario tenga una cookie válida para el acceso. Si el usuario no está registrado, el *middleware* lo redirige a la página de inicio de sesión ('*/login*').

- **Cierre de Sesión:** Al cerrar sesión, el *frontend* invoca una *Server Action* específica para el *logout*. Esta acción instruye al navegador para que elimine la *cookie* de sesión y redirige al usuario a la página de inicio de sesión.

3.5. Despliegue y puesta en producción

3.5.1. Entorno de desarrollo y despliegue

La elección de un entorno de desarrollo y despliegue ha sido uno de los elementos más importantes para el desarrollo continuo de *Fineancer*. Se han estudiado diversas opciones con el objetivo de facilitar la ejecución del proyecto en diversas máquinas, para agilizar el proceso tanto en desarrollo como en producción.

Se optó por la implementación de un sistema de gestión de paquetes que permitiera reproducir tanto el *backend* como el *frontend* de manera idéntica en múltiples entornos, sin necesidad de realizar configuraciones manuales o dependientes del sistema operativo.

3.5.1.1. Tecnologías de gestión de paquetes

Las tecnologías de gestión de paquetes y despliegue estudiadas para el desarrollo del proyecto han sido:

- **Nix**: Se trata de un gestor de paquetes y sistema de construcción que permite una definición declarativa y reproducible del entorno de desarrollo. *Nix* almacena paquetes, los cuales tienen su propio subdirectorio único generado durante la construcción y que no se modificarán una vez son creados. Esto ofrece diversas ventajas:
  - Se evitan conflictos entre versiones y dependencias, permitiendo la coexistencia de múltiples versiones de una misma biblioteca.
  - Garantiza que el entorno de desarrollo y ejecución sea exactamente el mismo en todas las máquinas
  - Facilita la incorporación de nuevos desarrolladores al proyecto, o la continuación del mismo, ya que el entorno puede ser replicado con precisión.
- ***Docker***: Se trata de una de las herramientas mas populares entre los desarrolladores, centrada en la creación y ejecución de contenedores, que encapsulan una aplicación junto con sus dependencias, sistema de archivos y configuraciones. También proporciona aislamiento a nivel del sistema operativo.

Aunque inicialmente se consideró *Docker* como alternativa, se descartó como tecnología principal de desarrollo por los siguientes motivos:

- Las imágenes *Docker*, aunque reproducibles, no son inmutables como las de *Nix* y pueden estar sujetas a cambios debido a cambios en las fuentes remotas.
- La curva de mantenimiento de *Dockerfiles* es más elevada para asegurar reproducibilidad completa, y resulta en dificultades para mantener y continuar el proyecto con el paso del tiempo.
- Requiere una infraestructura más pesada, lo cual complica el uso en ciertos entornos como *macOS*, *WSL* o servidores sin soporte completo de contenedores.

Por el contrario, *Nix* proporciona un entorno ligero, portable y totalmente reproducible que ha permitido definir un entorno de desarrollo único para todos los miembros del equipo, garantizando que la instalación de dependencias no varíe entre máquinas ni introduzca errores ocultos. Esta decisión ha resultado especialmente útil en el proyecto, donde se integran múltiples tecnologías (*Next.js*, *FastAPI*, *PostgreSQL*, *Ollama*) y donde la consistencia entre entornos es clave para asegurar un desarrollo fluido.

Nix resuelve varios problemas críticos para el desarrollo y despliegue de *Fineancer*:

- **Reproducibilidad**: Garantiza que todos los miembros del equipo de desarrollo trabajen con exactamente las mismas versiones de *Python*, *Node.js*, bases de datos y todas sus dependencias.

- **Aislamiento**: Permite tener múltiples versiones de herramientas y bibliotecas instaladas sin conflictos, facilitando trabajar en varios proyectos simultáneamente.
- **Gestión automatizada**: Mediante un único comando (`nix develop`), configura todo el entorno de desarrollo con las dependencias exactas que requiere el proyecto, facilitando su ejecución.
- **Despliegue consistente**: El mismo entorno utilizado para desarrollo puede replicarse exactamente en el servidor de producción.

3.5.1.2. Despliegue de la aplicación

El despliegue de *Fineancer* se ha realizado aprovechando las capacidades de *Nix* para generar entornos idénticos entre desarrollo y producción. El proceso consiste en:

1. Definición del entorno mediante un archivo `flake.nix` que especifica todas las dependencias y configuraciones.
2. Generación de un *shell* de desarrollo reproducible mediante `nix develop`, que proporciona acceso a todas las herramientas necesarias.
3. Utilización de `JustFile` como automatizador de tareas, facilitando operaciones comunes como iniciar el servidor, ejecutar pruebas o construir el *frontend*.
4. En producción, el mismo archivo `flake.nix` garantiza que el entorno de ejecución sea idéntico al de desarrollo, minimizando errores relacionados con diferencias entre entornos.

3.5.1.3. Experiencia en el desarrollo del proyecto

La adopción de *Nix* en este proyecto ha tenido un impacto positivo en el desarrollo entre los miembros del equipo. Aunque al inicio se presentaron ciertas dificultades debido a la curva de aprendizaje inicial sobre el desarrollo con *Nix*, una vez establecida la arquitectura de ejecución y despliegue, su implementación ha aportado múltiples ventajas:

- La eliminación completa de problemas relacionados con diferentes versiones de dependencias entre miembros del equipo.
- Capacidad para trabajar en diferentes características del proyecto sin preocuparse por conflictos de dependencias.
- Integración fluida con el sistema de integración continua, donde las pruebas se ejecutan en un entorno exactamente igual al de desarrollo.

- Facilidad para incorporar nuevas dependencias de manera consistente en todos los entornos.
- Instalación y ejecución en el entorno de producción con sencillez, siendo idéntica al del entorno de desarrollo.

3.5.2. Integración continua

El flujo de integración continua está definido mediante *GitHub Actions*. Cada vez que se realiza un *push* directamente a la rama principal `main` o cuando se crea un *pull request* dirigido a la rama `main`, se ejecuta el *workflow* de CI. De esta manera, se asegura que tanto las contribuciones directas como las propuestas de cambios sean validadas antes de integrarse completamente al proyecto. Esto garantiza la calidad del código y la estabilidad del proyecto a lo largo del desarrollo, permitiendo detectar problemas más rápido.

3.5.2.1. Entorno de ejecución

El *workflow* utiliza un entorno de ejecución basado en *Nix*, siguiendo la filosofía de gestión declarativa adoptada en el proyecto. Esta configuración permite que todos los flujos de integración continua se ejecuten en un entorno idéntico al de desarrollo local seguido para la ejecución del proyecto.

La integración de *Nix* en el flujo de CI proporciona varias ventajas fundamentales:

- **Reproducibilidad exacta**: Todas las dependencias son idénticas entre entornos, incluyendo versiones específicas de bibliotecas y herramientas.
- **Aislamiento**: El entorno de ejecución está completamente aislado, evitando interferencias con otras configuraciones.
- **Consistencia**: Se garantiza que todas las pruebas y verificaciones se realizan bajo condiciones idénticas, independientemente de dónde se ejecuten.

3.5.2.2. *Jobs* de validación

El *workflow* define dos *jobs* principales que se ejecutan en paralelo:

1. ***Backend***: Valida el código del *backend* de la aplicación.
2. ***Frontend***: Valida el código del *frontend* de la aplicación con pruebas *lint* de ejecución en un entorno de producción.

Cada *job* sigue una secuencia similar de pasos:

1. ***Checkout* del repositorio**: Obtiene el código fuente desde *GitHub*.

2. **Instalación de *Nix***: Configura el entorno de compilación y ejecución a través de la acción específica para instalar *Nix*.

3. **Caching**: Implementa un sistema de caché para el almacén de *Nix*, acelerando significativamente la ejecución al reutilizar paquetes ya compilados.

4. **Configuración**: Prepara el entorno específico (*backend* o *frontend*) utilizando *JustFile* como automatizador de tareas.

5. **Verificación**: Ejecuta las comprobaciones de calidad de código para detectar posibles errores.

3.5.3. *Hosting* y dominios

El despliegue de *Fineancer* se ha realizado en un entorno de *hosting* remoto preparado para el hospedaje de páginas web. Se trata de un servidor en la plataforma *Hetzner* que permite el acceso remoto a un servidor *Linux* personalizado a través de IP externa.

Hemos utilizado un dominio personalizado para facilitar el acceso de los usuarios, configurando su acceso *DNS* hacia la aplicación de *Fineancer*.

El acceso al sitio se realiza de forma segura mediante certificados SSL/TLS automáticos y conexión HTTPS.

3.5.3.1. Infraestructura del servidor

Para el alojamiento se ha contratado un servidor virtual en la plataforma *Hetzner*, con las siguientes características:

- **Sistema operativo:** *Debian* 12
- **Hardware:** 8 core CPU, 64 GB RAM
- **Servicios:** *Nginx* como *proxy* inverso y gestor de certificados HTTPS.
- **Conectividad:** Conexión de red con ancho de banda garantizado de 1 Gbps.

3.5.3.2. Configuración del servidor web

La arquitectura de servidor implementada utiliza un enfoque moderno de *proxy* inverso y gestión de servicios:

- ***Apache y Nginx*:** Configurado como servidor web principal. *Nginx* se seleccionó para la creación de *proxy* inverso eficiente capaz de redireccionar correctamente los puertos del *frontend* y el *backend* en una IP fija del servidor de producción. Sus principales funciones son:

- Gestionar los certificados SSL/TLS mediante integración con *Certbot* para la renovación automática.
- Redirigir automáticamente todo el tráfico *HTTP* (puerto 80) a *HTTPS* (puerto 443).
- Actuar como *proxy* inverso para las rutas de la API (`/api/*`), redirigiendo las peticiones al puerto interno 8000 donde se ejecuta el *backend FastAPI*.
- Actuar como *proxy* inverso para el *frontend* de *Next.js*, redirigiendo las peticiones al puerto interno 3000.

- ***PM2*:** Gestor de procesos para ejecutar los servicios de las capas de la aplicación, manteniéndolos en segundo plano, esto permite:
  - Iniciar, detener y reiniciar los módulos de *frontend*, *backend* y *Ollama* individualmente.
  - Mantener los servicios en ejecución permanente, reiniciándolos automáticamente en caso de fallo.
  - Registrar *logs* detallados individuales para facilitar la monitorización y el diagnóstico de problemas.
  - Configurar el arranque automático de todos los servicios al iniciar el sistema operativo.

3.5.3.3. Configuración DNS y dominio

Para facilitar el acceso a la aplicación, se implementó un dominio de acceso publico que apunta al servidor:

- **Dominio principal:** Se utilizó el dominio `jorge.cafe` como base, alojado en la plataforma *Cloudflare* para la gestión de registros DNS.
- **Subdominios:** Se configuró el subdominio `fineancer.jorge.cafe` específicamente para la aplicación, permitiendo un acceso directo para los usuarios.
- **Registros DNS:** Registro *A* que apunta la dirección IP pública del servidor *Hetzner*.

Esta configuración de *hosting* garantiza que *Fineancer* esté disponible de forma continua, segura y con un rendimiento óptimo para todos los usuarios, independientemente de su ubicación o dispositivo de acceso.

Capítulo 4

Conclusiones y Trabajo Futuro

4.1. Conclusiones generales del proyecto

Después de completar el desarrollo del proyecto, se ha llegado a las siguientes conclusiones:

- **Cumplimiento de los objetivos principales**: Se ha desarrollado con éxito una aplicación web completa para la gestión de gastos compartidos, cumpliendo con los requisitos planteados inicialmente. La implementación de funcionalidades como grupos, división de gastos, categorización jerárquica y presupuestos grupales ha proporcionado una solución integral al problema abordado.
- **Integración de tecnologías modernas**: La combinación de tecnologías como *FastAPI*, *Next.js* y *PostgreSQL* ha resultado ser una elección acertada, ofreciendo un equilibrio óptimo entre rendimiento, experiencia de usuario y facilidad de desarrollo. La adopción de tecnologías actualmente dominantes ha sido valiosa tanto para el aprendizaje como para la práctica profesional.
- **Valor de la reproducibilidad**: El uso de *Nix* como sistema de gestión de entornos ha demostrado ser fundamental al garantizar la reproducibilidad del proyecto tanto en desarrollo como en producción, eliminando problemas de configuración y facilitando la colaboración entre los miembros del equipo. Aunque requirió una curva de aprendizaje inicial, esta decisión fue clave para el éxito del despliegue.
- **Innovación en asistencia inteligente**: La implementación de funcionalidades basadas en *IA*, como la extracción automática de información de recibos y las sugerencias de presupuestos, ha aportado un valor significativo frente a soluciones existentes, demostrando el potencial de las tecnologías de procesamiento de lenguaje natural en aplicaciones cotidianas.
- **Arquitectura escalable**: El enfoque modular adoptado en la arquitectura del sistema, con una clara separación entre frontend, backend y base de da-

tos, ha permitido desarrollar una aplicación escalable y mantenible, capaz de evolucionar fácilmente con nuevas funcionalidades.

- **Priorización de la experiencia de usuario**: El diseño mobile-first y la atención a la usabilidad han sido clave para crear una aplicación accesible y amigable. La simplicidad de los flujos de usuario y la claridad en la presentación de información financiera fueron prioritarias durante todo el desarrollo.

- **Validación de la propuesta de valor**: Las funcionalidades diferenciadoras implementadas, especialmente en comparación con aplicaciones como *Tricount* o *Splitwise*, confirman que existía margen de mejora en la gestión de gastos compartidos, particularmente en aspectos como la categorización avanzada y la planificación presupuestaria.

El proyecto *Fineancer* ha permitido demostrar las habilidades y conocimientos adquiridos durante los años académicos en la universidad.

Las asignaturas del plan de estudios que más han contribuido al desarrollo del proyecto han sido:

- **Gestión de Proyectos Software y Metodologías de Desarrollo (GPS)**: Esta asignatura fue esencial para la organización del proyecto, permitiendo gestionar el desarrollo entre los miembros del equipo, distribuir tareas y responsabilidades, y establecer hitos y objetivos.

- **Aplicaciones Web (AW)**: Los conocimientos sobre principios web, desarrollo frontend y APIs REST fueron fundamentales para construir la interfaz de usuario de *Fineancer*, así como para estructurar el código siguiendo buenas prácticas.

- **Bases de Datos (BD)**: La modelización de bases de datos SQL y la optimización de consultas fueron críticas para garantizar el rendimiento de la aplicación. Se aplicaron conocimientos de diseño de esquemas relacionales y manejo eficiente de consultas.

- **Programación de Aplicaciones para Dispositivos Móviles (PAD)**: Esta asignatura proporcionó las herramientas necesarias para desarrollar una aplicación responsive con enfoque mobile-first, adaptando la interfaz a diferentes tamaños de pantalla e implementando características específicas para móviles.

- **Sistemas Operativos (SO)**: Los conceptos de gestión de procesos y virtualización ayudaron a implementar el sistema de despliegue y configurar el entorno de desarrollo con *Nix*, permitiendo ejecutar el proyecto en sistemas *Linux* de forma eficiente.

4.2. Dificultades técnicas y aprendizajes personales

Durante el desarrollo de este proyecto, nos hemos enfrentado a varios desafíos:

- **Ingeniería de prompts para LLM**: El diseño de prompts efectivos para la extracción de información de recibos mediante modelos de lenguaje supuso un reto considerable. Fue necesario experimentar con diferentes enfoques y refinar los prompts para obtener resultados precisos y consistentes, libres de alucinaciones. Este proceso mejoró nuestra comprensión de las capacidades y limitaciones de los LLM y desarrolló nuestras habilidades en ingeniería de prompts.
- **Ejecución y despliegue con *Nix***: Aunque *Nix* acabó siendo beneficioso para el proyecto, la curva de aprendizaje inicial fue pronunciada. Configurar el entorno de desarrollo y escribir expresiones *Nix* adecuadas requirió tiempo y estudio, pero finalmente permitió un entorno reproducible y consistente.
- **Comunicación entre frontend y backend**: La implementación de una comunicación eficiente y segura entre el frontend (Next.js) y el backend (FastAPI) presentó desafíos, especialmente en lo relativo a autenticación, gestión de estados y manejo de errores.
- **Ejecución de modelos LLM con *Ollama***: La integración de modelos de lenguaje locales mediante Ollama para el procesamiento de recibos implicó retos de rendimiento y consumo de recursos. Se eligieron modelos eficientes y se optimizó su ejecución para lograr una integración adecuada entre las capas de la aplicación.

4.3. Posibles mejoras futuras

Aunque el sistema actual cumple los objetivos planteados, existen oportunidades claras de evolución tanto a nivel funcional como técnico. Las mejoras identificadas para futuras versiones incluyen:

- **Mejora del sistema de usuarios**: Vincular cuentas con correo electrónico, servicios externos (*OAuth*) como *Google* o número de teléfono. Esto facilitaría la conexión entre usuarios y permitiría funciones como la recuperación de contraseñas. También se podrían añadir atributos personalizables como nombre, edad o foto de perfil.
- **Exportación de datos**: Permitir la exportación de gastos de grupo a formatos como *XLSX* o *CSV*, útil para análisis en hojas de cálculo o copias de seguridad.
- **Lectura de recibos**: Mejorar la extracción automática de información para incluir desglose por artículos, descuentos aplicados, etc. También se podrían dividir gastos de forma precisa, asociando artículos a usuarios específicos, ideal para dividir cuentas de restaurante de forma justa.

- **Extensión del sistema de invitaciones**: Añadir herramientas como invitaciones de un solo uso, por URL o regeneración de códigos de invitación.
- **Presupuestos por categoría**: Permitir asociar presupuestos a categorías concretas mediante sistemas de *whitelist* o *blacklist*, para que solo ciertos gastos se consideren en el cálculo mensual.
- **Sistema de notificaciones**: Implementar notificaciones para alertar a usuarios sobre acciones que les afecten, como la creación de un gasto o presupuesto.

Introduction

Fineancer arises as a proposal to optimise the management of shared expenses through a web application that prioritises intuitiveness and transparency. The project was conceived after identifying limitations in existing tools and proposes a solution that addresses real needs through well-defined technical and functional objectives. The work plan envisages the development of a robust platform focused on usability, security, and scalability.

4.4. Motivation

In everyday life, sharing expenses is commonplace in numerous contexts: group trips, celebrations, shared flats, academic or work activities, among others. In many of these cases, maintaining clear and fair control of expenses can become a complex task, especially when multiple people and transactions are involved.

Although there are applications designed for managing shared expenses, many of them do not cover all the real needs of users. A lack of flexibility in splitting expenses, limited personalisation in group management, or an unclear user experience are some of the shortcomings detected in existing solutions.

In this context, the motivation arises to develop an application that facilitates the organisation and control of common expenses in a simpler, more accessible way and adapted to different types of users and situations. This project meets the desire to create a useful tool for daily life, fostering transparency and reducing conflicts derived from shared financial management.

4.5. Objectives

The main objective of this project is to design and develop a web application called *Fineancer*, aimed at facilitating the management and tracking of expenses both at a personal and group level. The application will be built using modern technologies and will prioritise user experience, scalability, and maintainability.

Specifically, the project's objectives are:

- Analyse the real problem of group expense management, identifying the most common use cases and the limitations present in similar existing tools.
- Design a functional structure of the application, establishing the main entities (users, groups, expenses, categories, etc.) and how they relate to one another.
- Allow the creation, editing, and deletion of groups, as well as the incorporation of new members through a user-managed invitation system.
- Implement a user authentication system that provides secure, individualised access to the platform, managing sessions persistently and storing data on servers.
- Design and implement a permission-control system that ensures data integrity and accuracy, where each user can modify or delete only the data they have registered.
- Offer an intuitive system for recording and consulting expenses, including relevant information for all group members, such as description, amount, participants, and category.
- Enable the categorisation of recorded expenses so that users can classify their transactions according to different types (such as food, transport, or accommodation), facilitating subsequent analysis of spending and promoting better financial organisation.
- Incorporate an expense-splitting system that calculates final debts between participants based on their contribution to each expense.
- Provide the option for users to mark their debts as settled once they have made the corresponding payments outside the application, maintaining a clear record of each member's financial obligations.
- Develop functionality to define budgets within each group, setting a common spending limit that serves as a reference for all members, fostering transparency and shared planning, and helping the group maintain collective control over spending levels.
- Design a simple, clear, and adaptable user interface, with special attention to mobile devices, following a *mobile-first* approach.
- Organise and store information in a structured and persistent manner, ensuring that user, group, and expense data are consistently available from any device.
- Prepare and deploy the application in a functional production environment with a reproducible and stable configuration, ensuring the system is accessible from any network.

These objectives aim not only to solve a common problem effectively but also to build a solid technological base that can be expanded in the future with features such as additional invitation methods, customised splits, spending analytics, or report export.

4.6. Work Plan

The development of the *Fineancer* project was structured into several methodological phases, designed to ensure logical progression from the initial idea to the implementation and deployment of the application. This work plan allowed objectives to be addressed in an organised and systematic manner:

1. **Phase 1: Research and Requirements Definition**
   - Exhaustive analysis of similar existing applications (Tricount, Splitwise, etc.) to identify common features, strengths, and weaknesses.
   - Detailed definition of *Fineancer*'s functional and non-functional requirements, based on identified shortcomings and project objectives (as detailed in Chapter 3).
   - Initial outline of the system architecture and preliminary selection of the technology stack (Python/FastAPI for the backend, React/Next.js for the frontend, PostgreSQL as the database).
2. **Phase 2: System Design**
   - Detailed design of the overall architecture: client–server, REST API.
   - Design of the data model and database schema (entities, relationships, data types), as presented in Figure 3.2.
   - Design of the security and authentication model (session management, password hashing with Argon2, viewer-owner permission model).
   - Design of Data Transfer Objects (DTOs) for the API, both for database entities (Figure 3.3) and specific API operations (Figure 3.4).
   - Design of the REST API endpoint structure (Figure 3.5).
   - User interface (UI) and user experience (UX) design with a *mobile-first* approach, including wireframes and mock-ups of the main screens (Dashboard, Groups, Expenses, Budgets, Profile).
3. **Phase 3: Backend Development**
   - Configuration of the backend development environment (Python, FastAPI, SQLModel, PostgreSQL).
   - Implementation of database models and initial migrations.
   - Development of services and business logic for user management and authentication (registration, login, logout, session management).
   - Implementation of controllers and endpoints for managing groups, members, and invitations.
   - Development of functionality to register, modify, and delete expenses, including expense-splitting logic (*SplitService*).
   - Implementation of hierarchical category and budget management.

- Development of the image storage and management system (temporary and permanent) associated with expenses.
- Implementation of the expense suggestion service based on images and budget suggestions derived from history, integrating LangChain.
- Creation of unit and integration tests to ensure backend robustness.

4. **Phase 4: Frontend Development**

   - Configuration of the frontend development environment (Next.js, React, Tailwind CSS).
   - Development of reusable React components (buttons, forms, lists, modals) based on Radix UI and styled with Tailwind CSS.
   - Implementation of the main application pages: login/registration, dashboard, group management, expense list and detail, budget management, and user profile.
   - Integration with the backend API for all functionalities (CRUD of users, groups, expenses, categories, budgets, images).
   - Implementation of client-side state management and authentication handling (session cookies).
   - Development of data visualisations for expenses and budgets using charts.
   - Ensuring responsiveness and a *mobile-first* experience.
   - Form validation implementation.

5. **Phase 5: Integration, Testing, and Refinement**

   - Comprehensive testing of frontend–backend integration.
   - Usability testing with potential users to gather feedback and adjust the interface and navigation flows.
   - Bug fixes identified during testing.
   - Performance optimisation of both frontend and backend.
   - Review of overall application security.

6. **Phase 6: Deployment and Documentation**

   - Preparation of the production environment (server, database, domain configuration).
   - Configuration of tools for reproducible deployment (Nix, Justfile) and continuous integration (GitHub Actions).
   - Deployment of the application on the hosting server (Hetzner).
   - Final testing in the production environment.
   - Writing the project's final report, including technical and user documentation.

- Preparation of the project presentation.

This work plan was conceived as a flexible guide, allowing adjustments and overlaps between phases according to the needs and progression of development. The main goal was to maintain an iterative and incremental development approach, focused on delivering continuous value and ensuring the quality of the final product. 1.

Conclusions and Future Work

4.7. General Conclusions of the Project

After completing the development of the project, the following conclusions have been reached:

- **Fulfillment of the main objectives**: A complete web application for shared expense management has been successfully developed, meeting the initial requirements. Features such as groups, expense splitting, hierarchical categorization and group budgets provide a comprehensive solution to the addressed problem.
- **Integration of modern technologies**: The use of technologies such as *FastAPI*, *Next.js* and *PostgreSQL* proved to be a solid choice, balancing performance, user experience and development ease. Adopting currently dominant tools was also valuable for learning and practical application.
- **Value of reproducibility**: Using *Nix* for environment management ensured project reproducibility in both development and production. It eliminated configuration issues and made team collaboration easier. Despite an initial learning curve, this decision was essential for successful deployment.
- **Innovation in intelligent assistance**: AI-based features such as automatic receipt parsing and budget suggestions added significant value over existing solutions, showcasing the potential of natural language processing in practical daily applications.
- **Scalable architecture**: The modular system architecture—clearly separating frontend, backend and database—enabled the creation of a scalable, maintainable application that can evolve easily with new features.
- **User experience prioritization**: A mobile-first design and focus on usability were key to building an accessible, user-friendly application. Simple user flows and clear presentation of financial data were priorities throughout development.

- **Validation of the value proposition**: The implemented features, especially compared to apps like *Tricount* or *Splitwise*, confirmed there was room to improve shared expense management, particularly in areas like advanced categorization and budget planning.

The *Fineancer* project has allowed us to demonstrate the skills and knowledge gained throughout our university years.

Courses that most contributed to the project's development include:

- **Software Project Management and Dev. Methodologies (GPS)**: This course was key for organizing the project, distributing tasks and responsibilities and setting goals and milestones.
- **Web Applications (AW)**: Knowledge of web principles, frontend development and REST APIs was essential for building *Fineancer*'s user interface and applying good coding practices.
- **Databases (BD)**: SQL modeling and query optimization were critical to ensuring good performance. We applied relational schema design and efficient query execution.
- **Mobile Application Programming (PAD)**: This course gave us the tools to develop a responsive, mobile-first application, adapt interfaces to various screen sizes and implement mobile-specific features.
- **Operating Systems (SO)**: Concepts like process management and virtualization helped with deploying the system and configuring the development environment using *Nix*, allowing efficient Linux execution.

4.8. Technical Challenges and Personal Learning

During development, we faced several challenges:

- **LLM prompt engineering**: Designing effective prompts for extracting receipt data with language models was a major challenge. We tested multiple approaches to ensure accurate and consistent outputs, free of hallucinations. This deepened our understanding of LLMs and improved our prompt engineering skills.
- **Execution and deployment with *Nix***: While ultimately successful, learning to configure the environment and write proper *Nix* expressions required significant effort. It paid off by creating a reproducible and stable development setup.
- **Frontend-backend communication**: The implementation of secure and efficient communication between the *Next.js* frontend and the *FastAPI* backend was challenging, especially around authentication, state management and error handling.

- **Running LLMs with *Ollama***: Integrating local LLMs for receipt processing via Ollama introduced performance and resource challenges. We selected lightweight models and optimized loading to ensure smooth operation across app layers.

4.9. Potential Future Improvements

While the current system fulfills the objectives, there are many areas for future improvement, both functionally and technically. Planned upgrades include:

- **User system enhancements**: Add support for linking accounts to email, OAuth services like *Google*, or phone numbers. This would improve user connectivity and allow features like password recovery. Other profile attributes like name, age, or photo could be added.
- **Data export**: Allow users to export group expenses in formats like *XLSX* or *CSV* for spreadsheet use or backups.
- **Receipt parsing improvements**: Extract more detailed data from receipts (e.g., item breakdowns, discounts) and support splitting by item assigned to specific users—especially useful for restaurant bills.
- **Invitation system extension**: Add tools like one-time invites, shareable URLs, or invitation code regeneration.
- **Category-based budgets**: Let users specify categories for each budget, using either a blacklist or whitelist system to filter applicable expenses in monthly tracking.
- **Notification system**: Notify users when actions affecting them occur—like a new expense or budget entry.

Contribuciones Personales

Jorge Calvo Fernández

En esta sección se detalla la contribución de Jorge Calvo Fernández al desarrollo del proyecto. El campo en el que se ha centrado es el del frontend y toda la parte visual de la aplicación además del hosting y despliegue en la nube.

Desarrollo de Frontend

Se ha encargado principalmente del desarrollo del frontend de la aplicación, creando la interfaz de usuario y el acceso a los datos ofrecidos por el Backend. Para su implementación se ha utilizado el framework de Next.js con React como biblioteca principal, junto con TypeScript y Tailwind CSS para la creación de las páginas de la aplicación.

Arquitectura y Estructura del Proyecto

Ha diseñado una arquitectura modular basada en componentes reutilizables, siguiendo los principios de separación de responsabilidades. Esta aproximación ha permitido construir una interfaz de usuario coherente, escalable y fácil de mantener. La estructura organizativa incluye carpetas para componentes, páginas, hooks, estilos y recursos estáticos.

Comunicación con el Backend

Ha implementado una capa de comunicación robusta y eficiente con el backend mediante una API centralizada. Esta implementación utiliza un wrapper personalizado sobre la API Fetch nativa que gestiona automáticamente los tokens de autenticación, encabezados HTTP, manejo de errores y serialización/deserialización de JSON. Esta abstracción facilita que todos los componentes del frontend puedan realizar peticiones HTTP de manera consistente y segura.

Implementación de Componentes Clave

Se han desarrollado numerosos componentes para cubrir todas las funcionalidades requeridas. Algunos de los más relevantes incluyen:

Dashboard Principal El dashboard es la vista central de la aplicación, donde el usuario puede visualizar su situación financiera actual, además de un acceso rápido a funciones como crear gastos por imagen y ver los últimos gastos creados por los grupos a los que pertenece.

Paginas de las entidades Se muestran páginas con todas las entidades de la aplicación (Grupos, Gastos y Presupuestos), con la capacidad de realizar todas las funciones básicas de crear, editar y eliminar y además de sistemas de búsqueda y filtros para facilitar el acceso a los datos.

Visualización de Datos Financieros Se han implementado diversos gráficos y visualizaciones de datos, como un gráfico que muestra la evolución de los gastos mensuales del usuario, una muestra de los datos de cada gasto y grupo y también

Captura de imágenes y subida al Backend Se ha desarrollado un sistema completo para la captura y procesamiento de imágenes de recibos que facilita a los usuarios la digitalización de sus gastos. La implementación utiliza la API Web de MediaDevices para acceder a la cámara del dispositivo, permitiendo a los usuarios tomar fotos directamente desde la aplicación sin necesidad de aplicaciones externas.

Experiencia de Usuario (UX/UI)

Se ha prestado especial atención a la experiencia de usuario, implementando:

- Diseño responsive para diferentes dispositivos con un enfoque mobile-first
- Feedback visual inmediato para todas las acciones
- Acceso rápido a todas las funcionalidades importantes de la aplicación
- Carga fluida de la aplicación a través de optimizaciones del proyecto

Hosting y Despliegue

Otra contribución fundamental ha sido la implementación del despliegue de la aplicación, garantizando su disponibilidad, rendimiento y seguridad. Se destacan las siguientes áreas:

Configuración de CI/CD

Se ha implementado un pipeline de integración y despliegue continuo utilizando GitHub, que automatiza el despliegue de la aplicación directamente al entorno de producción cuando se realiza un cambio sustancial en el código.

Acceso por URL

Se ha encargado del acceso a la aplicación de *Fineancer* de forma pública mediante URL. Garantizando el acceso desde cualquier navegador de manera sencilla. Este proceso ha llevado la creación del registro DNS por IP, manejo de dominios en el servidor mediante Apache y nginx y gestión de puertos entre el frontend y el backend para garantizar la conexión mutua.

Jorge Lumbreras Camps

Ha sido el principal responsable del desarrollo del *backend* en el proyecto, implementando la lógica de negocio, los sistemas de control de acceso y los algoritmos centrales de la aplicación. Su trabajo ha sentado las bases para todas las funcionalidades que ofrece el sistema, garantizando su correcto funcionamiento, seguridad y escalabilidad.

Diseño de la base de datos

Diseñó el esquema relacional de la base de datos, implementando una estructura óptima para los requisitos del sistema:

- Modelado de las entidades principales.
- Relaciones many-to-many para usuarios y grupos.
- Jerarquía recursiva de categorías mediante *self-joins*.
- Histórico de gastos con división proporcional.
- Restricciones de integridad referencial.
- Índices optimizados para consultas frecuentes.
- Triggers para mantener consistencia.
- Tablas de asociación para control de acceso.
- Modelado de *ownership*.
- Visibilidad basada en membresía a un grupo.

El diseño logra un balance entre normalización para evitar redundancias y desempeño para operaciones críticas.

Desarrollo de *DTOs*

Diseñó el sistema de *DTOs* usando *Pydantic* para:

- Validación estricta de datos en endpoints.
- Transformación segura entre modelos *DB* y *API*.
- Documentación automática de schemas.
- Validadores custom para reglas de negocio.
- Herencia de esquemas para evitar duplicación.

Desarrollo de la *API REST*

Ha diseñado e implementado la arquitectura completa de la API REST, incluyendo:

- El sistema de rutas y *endpoints* organizado por módulos funcionales.
- Los mecanismos de validación de datos en todas las operaciones.
- La gestión centralizada de errores y respuestas HTTP.

Control de Acceso y Seguridad

Diseñó e implementó la gestión de los permisos de los usuarios, diferenciando claramente entre:

- Propietarios, con derechos completos sobre los recursos.
- Visualizadores, con acceso limitado a ciertas operaciones

Este sistema se aplica consistentemente en todos los módulos del backend, garantizando que los usuarios solo puedan acceder a los recursos para los que tienen permisos.

Gestión de Grupos y Gastos

Desarrolló la lógica completa para:

- Operaciones CRUD de grupos y gastos.
- Invitaciones y gestión de miembros:
- Cálculo de balances y deudas entre usuarios.

- Sistema de categorías anidadas.

Además, implementó *endpoints* complejos como el de liquidación de deudas y el análisis histórico de gastos.

Algoritmo de División de Deudas

Diseñó e implementó el algoritmo voraz desde un enfoque práctico que prioriza la eficiencia sobre la optimalidad absoluta, dada la naturaleza no crítica de esta operación. En concreto, desarrolló las siguientes partes:

- Calcular deudas netas entre usuarios
- Simplificar transacciones minimizando su número
- Optimizar el proceso para grupos grandes

Sistema de Imágenes y Sugerencias

Implementó el módulo completo para:

- Gestión de imágenes temporales y permanentes.
- Procesamiento de recibos mediante modelos de lenguaje.
- Generación automática de sugerencias de gastos.
- Recomendaciones de presupuestos basadas en historial.

Este sistema integra tecnologías de IA mientras mantiene controles estrictos de acceso y validación.

Integración con Frontend

Desarrolló la capa de comunicación entre backend y frontend, incluyendo:

- Serialización/deserialización de datos
- Validación de esquemas de entrada.
- Documentación de la API.

Configuración de entornos con *Nix*

Implementó la configuración *Nix* para garantizar entornos de desarrollo y producción reproducibles, incluyendo:

- Definición de **entornos aislados** para *backend* (*Python/FastAPI*) y *frontend* (Node.js/Next.js).
- Configuración de *flakes* para manejo declarativo de dependencias.
- Scripts de desarrollo que automatizan:
  - La creación de la base de datos local.
  - La generación automática de entornos virtuales para *frontend* y *backend*.
  - El lanzamiento de servicios auxiliares, como el servidor de *Ollama*.

Bibliografía

Bayer, M. Sqlalchemy. En *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks* (editado por A. Brown y G. Wilson). aosabook.org, 2012.

Biryukov, A., Dinu, D. y Khovratovich, D. Argon2: the memory-hard function for password hashing and other applications. *University of Luxembourg*, 2015.

Chase, H. Langchain. https://github.com/langchain-ai/langchain, 2022.

Colvin, S., Jolibois, E., Ramezani, H., Garcia Badaracco, A., Dorsey, T., Montague, D., Matveenko, S., Trylesinski, M., Runkle, S., Hewitt, D., Hall, A. y Plot, V. Pydantic. https://github.com/pydantic/pydantic, 2025.

Contributors, L. Lucide react: Beautiful & consistent icon toolkit made by the community. https://lucide.dev/, 2023.

Freed, N. y Borenstein, N. Multipurpose internet mail extensions (mime) part two: Media types. RFC 2046, RFC Editor, 1996.

Garfield, C. Zod: Typescript-first schema validation with static type inference. https://zod.dev/, 2023.

Google. Introducing gemma 3: The developer guide. https://developers.googleblog.com/en/introducing-gemma3, 2025.

Group, R. Recharts: A composable charting library built on react components. https://recharts.org/, 2023.

Inc., F. React: A javascript library for building user interfaces. https://react.dev/, 2024a.

Inc., V. Next.js 15: The react framework for the web. https://nextjs.org/, 2024b.

Lehtosalo, J. Mypy. https://mypy-lang.org/index.html, 2012.

MetaAI. Llama 3.2: Revolutionizing edge ai and vision with open, customizable models. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices, 2024.

MistralAI. Mistral small 3.1. https://mistral.ai/news/mistral-small-3-1, 2025.

Mohan K, M. Algorithm behind splitwise's debt simplification feature. https://medium.com/@mithunmk93/algorithm-behind-splitwises-debt-simplification-feature-8ac485e97688, 2019.

OpenAI. Hello gpt-4o. https://openai.com/index/hello-gpt-4o/, 2024.

OpenAI. Introducing gpt-4.1 in the api. https://openai.com/index/gpt-4-1/, 2025a.

OpenAI. Introducing openai o3 and o4-mini. https://openai.com/index/introducing-o3-and-o4-mini/, 2025b.

PostgreSQL Global Development Group. Postgresql 16. http://www.postgresql.org/, 2023.

Python Software Foundation. Python 3.18.3 documentation. https://docs.python.org/3/library/functions.html, 2024.

Ramirez, S. Fastapi. https://fastapi.tiangolo.com, 2022a.

Ramirez, S. Sqlmodel. https://sqlmodel.tiangolo.com, 2022b.

Schlawack, H. Argon2-cffi. https://argon2-cffi.readthedocs.io/en/stable, 2015.

Splitwise. Splitwise - split expenses with friends. 2025a.

Splitwise. Splitwise features and pricing. 2025b.

Splitwise. Splitwise payment integrations. 2025c.

Splitwise. Splitwise pro features. 2025d.

Tricount. Tricount - automatic expense tracking. 2025a.

Tricount. Tricount - split group expenses. 2025b.

Tricount. Tricount features. 2025c.

Wathan, A. y Reinink, J. Tailwind css: Rapidly build modern websites without ever leaving your html. https://tailwindcss.com/, 2024.

WorkOS. Radix ui: Unstyled, accessible components for building high-quality design systems. https://www.radix-ui.com/, 2023.