Universidad Politécnica
de Madrid

Escuela Técnica Superior de Ingenieros Informáticos

Grado en Ingeniería informática

Trabajo Fin de Grado

Diseño e Implementación de una Plataforma de Gestión Financiera Personal

Autor: Xinghong Wu

Tutor(a): Vicente Martínez Orga

Madrid, mayo de 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

Trabajo Fin de Grado
Grado en Ingeniería informática
Título: Diseño e Implementación de una Plataforma de Gestión Financiera Personal
Mayo de 2024

Autor: Xinghong Wu

Tutor:
Vicente Martínez Orga
Departamento de Inteligencia Artificial
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

Este Trabajo Fin de Grado tiene como objetivo crear una plataforma web que permita a los usuarios gestionar sus finanzas personales. El desarrollo de la plataforma se dividió en dos partes: backend y frontend. El backend, implementado con Spring Boot, utiliza diversas herramientas para la gestión de usuarios, autenticación, cuentas, categorías, facturas, notas y planes de ahorro. El frontend, desarrollado con Vue.js, utiliza componentes de ElementUI para construir las interfaces. Además, la plataforma cuenta con dos interfaces distintas: una destinada a los usuarios y otra diseñada para administradores.

En el documento se describen las distintas fases de desarrollo, abarcando desde el estudio previo del mercado y las aplicaciones similares, hasta el análisis de resultados, centrándose sobre todo en el diseño y desarrollo de la plataforma.

Abstract

This Bachelor's Thesis aims to create a web platform that allows users to manage their personal finances. The development of the platform was divided into two parts: backend and frontend. The backend, implemented with Spring Boot, handles user management, authentication, accounts, categories, invoices, notes, and savings plans. The frontend, developed with Vue.js, uses ElementUI components to build the interfaces. Additionally, the platform features two distinct interfaces: one for users and another for administrators.

The document describes the various development phases, covering the preliminary market study and similar applications, to the analysis of results, with a particular focus on the design and development of the platform.

Tabla de contenidos

Tabla de ilustraciones

Índice de tablas

1 Introducción

En la actualidad, la gestión financiera personal se ha convertido en una habilidad esencial para mantener la estabilidad económica y alcanzar objetivos financieros a corto y largo plazo. La capacidad de gestionar ingresos, gastos, ahorros e inversiones de manera eficaz es fundamental para asegurar una buena salud financiera y evitar situaciones de endeudamiento o crisis económicas personales. [1]

Con el aumento de la complejidad de las finanzas personales, impulsado por factores como el incremento en el costo de vida, la diversificación de las fuentes de ingresos, y la proliferación de productos financieros, se hace cada vez más necesario contar con herramientas que faciliten esta tarea. Las decisiones financieras informadas pueden tener un impacto significativo en la calidad de vida, permitiendo a los individuos planificar para el futuro, enfrentar imprevistos y alcanzar metas personales y familiares.

A pesar de la existencia de diversas herramientas digitales destinadas a facilitar la gestión financiera, muchas de ellas presentan importantes limitaciones. En primer lugar, la falta de personalización es un problema común. Muchas aplicaciones financieras ofrecen soluciones genéricas que no se adaptan a las necesidades específicas de cada usuario, lo que puede resultar en una experiencia insatisfactoria y en una menor efectividad en la gestión de las finanzas.

Estas deficiencias en las herramientas actuales dificultan su adopción por parte de un amplio espectro de usuarios, quienes a menudo buscan soluciones más accesibles, seguras y adaptables a sus necesidades particulares.

Ante este panorama, surge la necesidad de desarrollar una plataforma que proporcione una solución eficaz, permitiendo a los usuarios llevar un control detallado de sus finanzas, establecer y seguir presupuestos, analizar sus hábitos de gasto y ahorro, y planificar sus objetivos financieros a corto y largo plazo.

1.1 Motivación

La gestión financiera personal es un aspecto crucial para la estabilidad económica y el bienestar individual. Con el avance de las tecnologías y el acceso cada vez mayor a dispositivos digitales, surge la necesidad de herramientas eficientes que permitan a los usuarios llevar un control detallado y organizado de sus ingresos, gastos y objetivos financieros.

1.2 Objetivos del Proyecto

El objetivo principal de este trabajo es diseñar e implementar una plataforma web para la gestión financiera personal que registra los gastos e ingresos. Para la implementación de la plataforma se utilizará Spring Boot para el backend y Vue.js para el frontend. Los objetivos establecidos son:

1. Desarrollo de arquitectura:

- Utilizar Spring Boot para construir un backend.
- Implementar Vue.js para desarrollar un frontend.

2. Implementación de mecanismos de seguridad:
    - Integrar la autenticación mediante JWT (JSON Web Tokens).
    - Configurar CORS.

3. Proveer funcionalidades completas de gestión financiera:
    - Facilitar la gestión de cuentas, transacciones y planes financieros.
    - Ofrecer herramientas para el análisis y la visualización de datos financieros.

4. Diseño de una interfaz de usuario intuitiva:
    - Crear una experiencia de usuario amigable que simplifique la navegación y el uso de la plataforma.

2 Estado del Arte

El estado del arte en la gestión financiera personal abarca una variedad de herramientas y aplicaciones que han sido desarrolladas para ayudar a los usuarios a administrar sus finanzas de manera eficiente. Estas herramientas van desde simples hojas de cálculo hasta aplicaciones móviles que integran inteligencia artificial y aprendizaje automático para proporcionar recomendaciones personalizadas. A continuación, se describen algunas de las categorías más relevantes y las principales características de las soluciones actuales en el mercado.

1. Aplicaciones Bancarias: Tenemos como ejemplos, Mint, YNAB (You Need a Budget). Estas aplicaciones permiten a los usuarios vincular sus cuentas bancarias, tarjetas de crédito y otros activos financieros para ofrecer una visión consolidada de sus finanzas. Proporcionan herramientas como el seguimiento de ingresos y gastos, categorización automática de transacciones y generación de informes financieros. Además, estas aplicaciones también ofrecen funcionalidades de presupuesto y alertas de gastos. [2]

Ilustración 1: Mint, YNAB [2]

2. Software de Presupuestación tenemos ejemplos com EveryDollar y Goodbudget, Las características de estas aplicaciones son que están diseñadas específicamente para la creación y seguimiento de presupuestos, estas herramientas permiten a los usuarios asignar fondos a diferentes categorías de gasto y monitorear su cumplimiento. Ofrecen funcionalidades para ajustar presupuestos, planificar gastos futuros y analizar patrones de gasto a lo largo del tiempo. [3]

Ilustración 2: EveryDollar, [4] Goodbudget [5]

3. Herramientas de Inversión y Planificación Financiera tenemos como ejemplos Quicken [6] y Wealthfront. [7] Además de las funcionalidades básicas de gestión financiera, estas aplicaciones proporcionan herramientas avanzadas para la planificación de inversiones y la gestión de patrimonios. Ofrecen asesoramiento financiero, análisis de cartera y simulaciones de planes de retiro.

Ilustración 3: Quicken, Wealthfront

4. Aplicaciones de Contabilidad Personal tenemos como ejemplos GnuCash y HomeBank. Estas aplicaciones son más adecuadas para usuarios con conocimientos en contabilidad. Ofrecen funcionalidades completas de contabilidad, incluyendo la gestión de activos y pasivos, conciliación bancaria y generación de informes financieros detallados. [8]

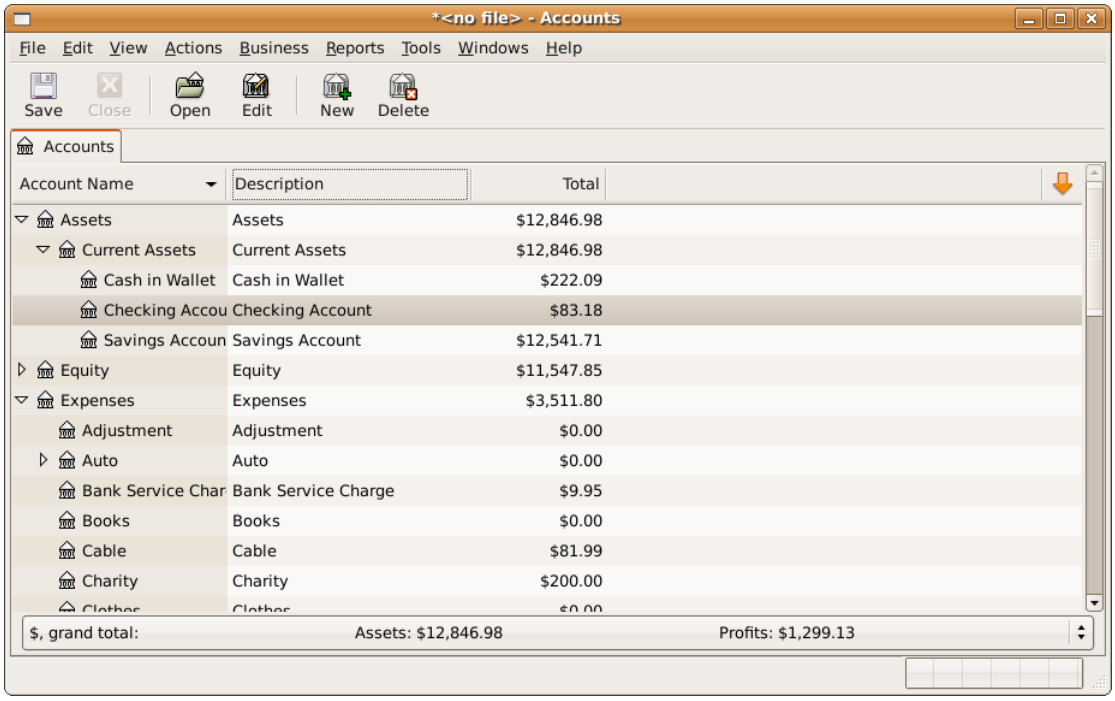


Ilustración 4: GnuCash [9]

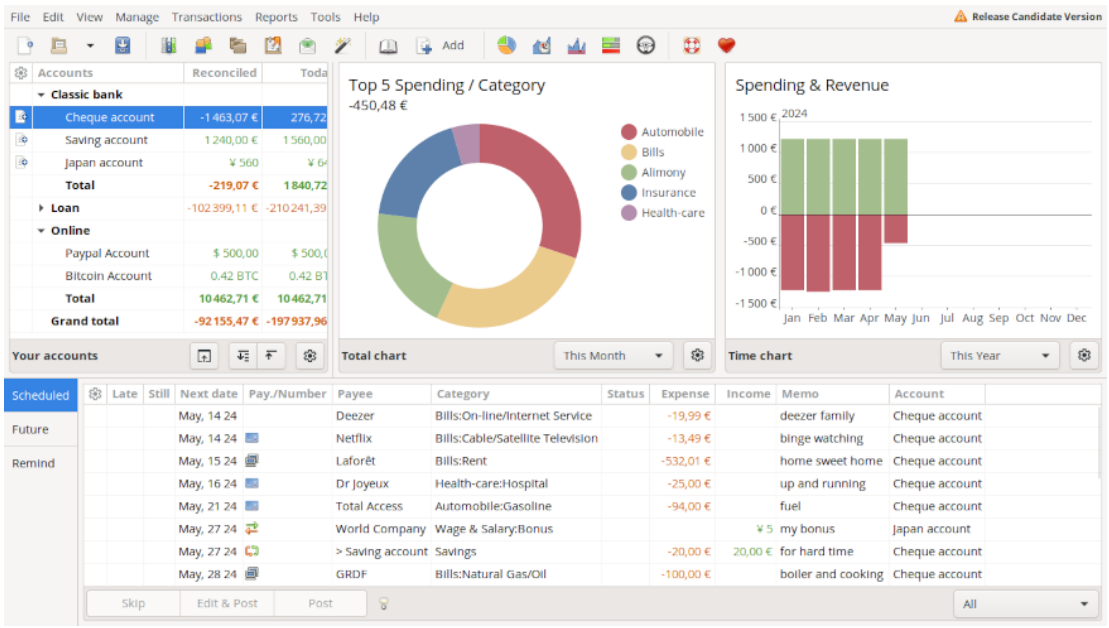


Ilustración 5: HomeBank [8]

3 Entorno de desarrollo

Para el desarrollo de la plataforma, se han utilizado diversas herramientas y tecnologías que se describen a continuación.

3.1 IntelliJ IDEA

Para desarrollar el backend en Spring Boot y frontend en Vue.js se utilizó IntelliJ IDEA, es un entorno de desarrollo integrado (IDE) desarrollado por JetBrains. Es ampliamente utilizado para el desarrollo de aplicaciones en Java y proporciona una serie de características que facilitan el desarrollo de software. Una de las ventajas que proporciona es que ofrece soporte integrado para el desarrollo de aplicaciones Spring Boot, incluyendo asistente de creación de proyectos, soporte para configuración de aplicaciones y herramientas de depuración. También proporciona potentes herramientas de refactorización que ayudan a mejorar el código de manera eficiente. Además, IntelliJ se integra fácilmente con sistemas de control de versiones como Git, permitiendo una gestión de código fuente eficiente. [10]

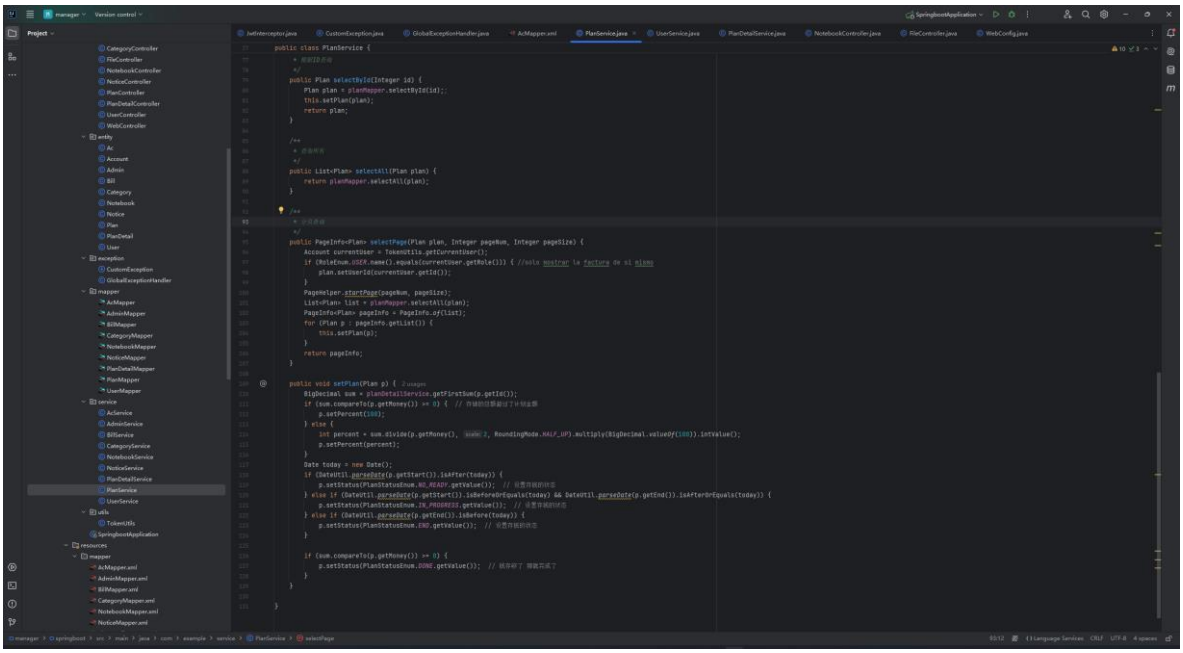


Ilustración 6: IntelliJ IDEA

3.2 Navicat for MySQL

Para gestionar la base de datos del proyecto se utilizó Navicat, es una herramienta de administración de bases de datos que permite a los desarrolladores gestionar y administrar bases de datos. Navicat soporta una amplia variedad de sistemas de gestión de bases de datos, incluyendo MySQL, PostgreSQL, SQLite, y más. En mi caso utilice MySQL. Además, proporciona una interfaz gráfica que facilita la administración de bases de datos, incluyendo

la creación, modificación y eliminación de tablas, vistas, procedimientos almacenados, y más. [11]

Ilustración 7: Navicat for MySQL [11]

4 Preproducción y Diseño

En esta sección se describe el diseño del sistema del proyecto, donde incluire los requisitos funcionales y no funcionales, y los diagramas que ilustran la arquitectura del sistema, el flujo de datos y las interacciones entre los componentes.

4.1 Especificación de requisitos

Los requisitos del sistema son fundamentales para el desarrollo de este proyecto, ya que en ellos definiremos lo que el sistema debe hacer (requisitos funcionales) y cómo debe comportarse (requisitos no funcionales).

4.1.1 Requisitos Funcionales

Tabla 1: Requisito Registro de Usuarios

ID	RF_01
Título	Registro de Usuarios
Descripción	Permitir a los nuevos usuarios registrarse en la plataforma.

Tabla 2: Requisito Autenticación de Usuarios

ID	RF_02
Título	Autenticación de Usuarios
Descripción	Autenticar a los usuarios mediante JWT (JSON Web Tokens).

Tabla 3: Requisito Gestión de Perfiles de Usuario

ID	RF_03
Título	Gestión de Perfiles de Usuario
Descripción	Permitir a los usuarios visualizar y editar su perfil.

Tabla 4: Requisito Gestión de Cuentas

ID	RF_04
Título	Gestión de Cuentas
Descripción	Crear, editar y eliminar cuentas.

Tabla 5: Requisito Visualización de Saldos y Transacciones

ID	RF_05
Título	Visualización de Saldos y Transacciones
Descripción	Mostrar el saldo y las transacciones de cada cuenta.

Tabla 6: Requisito Registro de Ingresos y Gastos

ID	RF_07
Título	Registro de Ingresos y Gastos
Descripción	Registrar ingresos y gastos en diferentes cuentas.

Tabla 7: Requisito Categorización de Transacciones

ID	RF_08
Título	Categorización de Transacciones
Descripción	Categorizar automáticamente las transacciones según su tipo.

Tabla 8: Requisito Planificación y Seguimiento de Objetivos Financieros

ID	RF_10
Título	Planificación y Seguimiento de Objetivos Financieros
Descripción	Establecer y seguir objetivos financieros personales.

Tabla 9: Requisito Generación de Informes Transacciones

ID	RF_11
Título	Generación de Informes de Transacciones
Descripción	Generar informes de transacciones detallados.

Tabla 10: Requisito Visualización de Datos Financieros

ID	RF_13
Título	Visualización de Datos Financieros
Descripción	Proporcionar gráficos y tablas para datos financieros.

4.1.2 Requisitos no funcionales

Los requisitos no funcionales describen los atributos de calidad del sistema, como la seguridad, el rendimiento y la usabilidad. Son esenciales para asegurar que el sistema no solo funcione correctamente, sino que también sea seguro, eficiente y fácil de usar.

Tabla 11: Requisito Seguridad

ID	RNF_01
Título	Seguridad
Descripción	Implementar autenticación y autorización seguras, y cifrado de datos

Tabla 12: Requisito Rendimiento

ID	RNF_02
Título	Rendimiento
Descripción	Asegurar una respuesta rápida del sistema y escalabilidad para soportar múltiples usuarios concurrentes.

Tabla 13: Requisito Usabilidad

ID	RNF_03
Título	Usabilidad
Descripción	Diseñar una interfaz intuitiva y fácil de usar para usuarios de todos los niveles.

Tabla 14: Requisito Disponibilidad

ID	RNF_04
Título	Disponibilidad
Descripción	Garantizar alta disponibilidad del sistema y tolerancia a fallos.

Tabla 15: Requisito Compatibilidad

ID	RNF_05
Título	Compatibilidad
Descripción	Soportar diferentes navegadores.

4.2 Mapa de navegación

Un mapa de navegación es una representación gráfica de la estructura y flujo de navegación de un sitio web o aplicación. Organiza y jerarquiza el contenido, mostrando cómo los usuarios pueden moverse a través de las diferentes secciones y funcionalidades. Incluye todas las pantallas necesarias, interacciones y conexiones entre las partes del sistema, facilitando la visualización del recorrido del usuario.

Además, proporciona claridad y estructura, estableciendo una jerarquía que asegura que los elementos críticos estén accesibles y organizados de manera lógica. También ayuda a detectar errores en la fase de diseño, permitiendo corregir problemas antes de la implementación.

El proyecto tiene dos pantallas principales: una para el administrador y otra para los usuarios. Los usuarios acceden a sus respectivas pantallas mediante el inicio de sesión, filtrando por su rol.

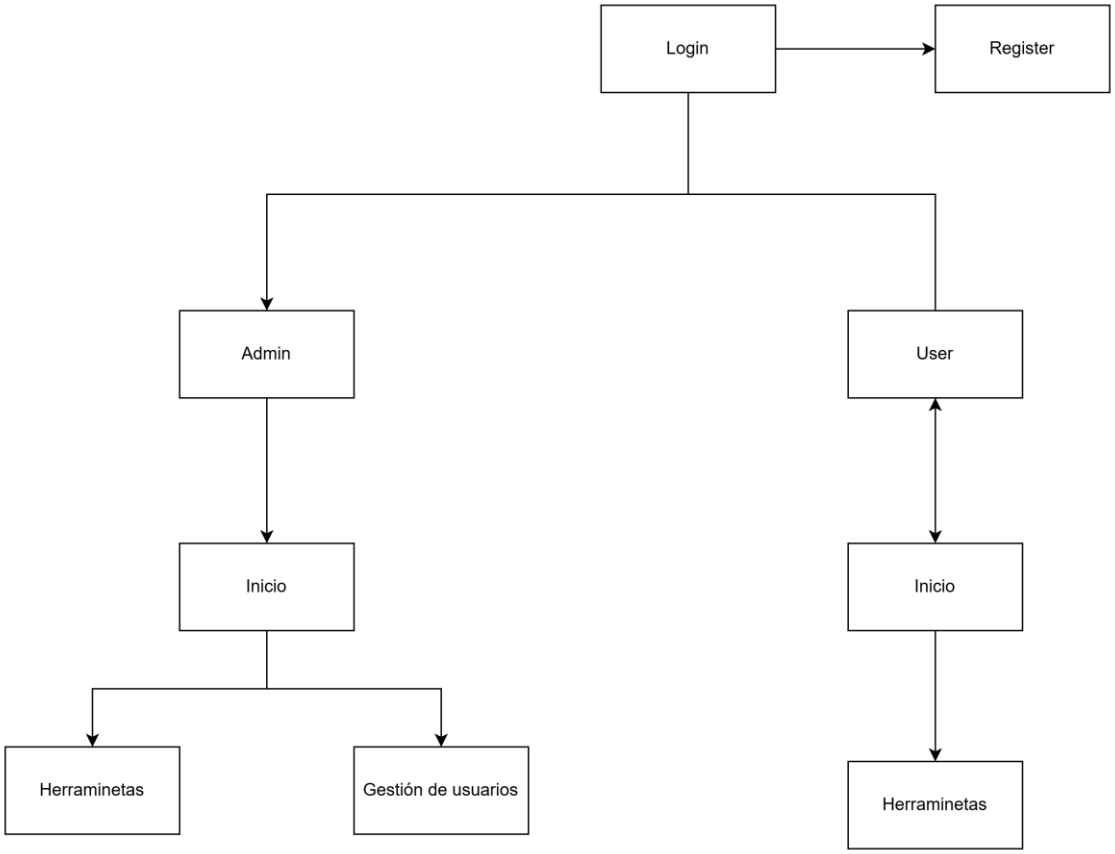


Ilustración 8: Mapa de navegación

4.3 Diagrama de flujo

Un diagrama de flujo es una representación gráfica que muestra los pasos secuenciales y las decisiones en un proceso o sistema, utilizando símbolos estándar como rectángulos para acciones, diamantes para decisiones y flechas para indicar el flujo de control. En el desarrollo de software, estos diagramas ayudan a visualizar cómo se moverán los datos y cómo interactuarán los diferentes componentes del sistema.

La plataforma cuenta con 3 funcionalidades independientes peincipales.

1. Login: es el sistema de inicio de sesión de la aplicación, que distingue entre usuarios y administradores

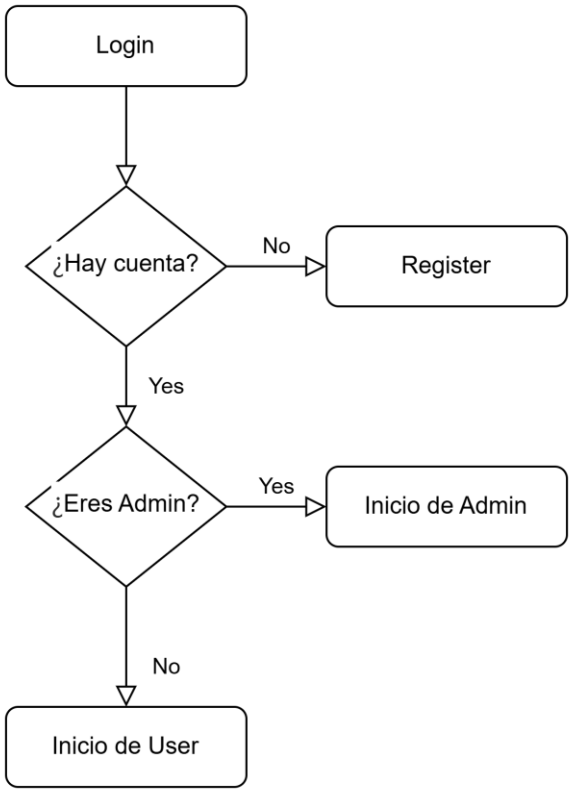


Ilustración 9: Diagrama de flujo de inicio de sesión

2. Búsqueda: el sistema de búsqueda es una funcionalidad esencial que permite a los usuarios encontrar rápidamente cuentas, usuarios, facturas, planes, notas y categorías dentro de sus respectivas ventanas.

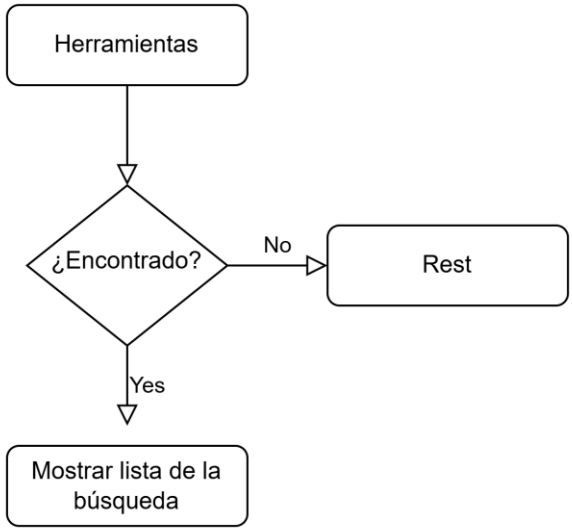


Ilustración 10: Diagrama de flujo de Búsqueda

3. Creación y modificación: el sistema de creación y modificación es la funcionalidad más importante, permite a los administradores crear o modificar cuentas, usuarios, facturas, planes, notas y categorías de manera eficiente. Los usuarios solo tendrán permiso en facturas, planes y notas.

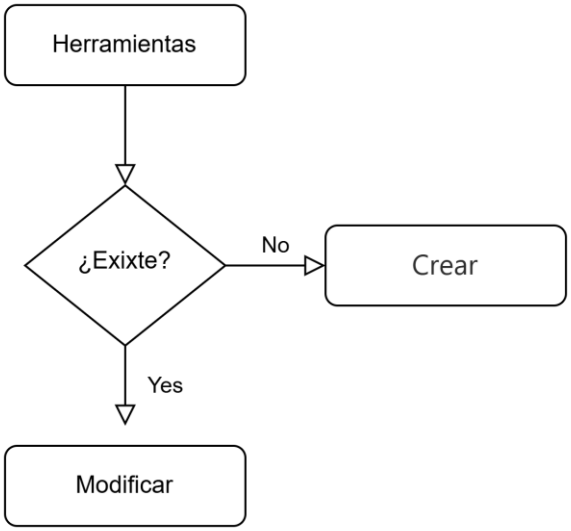


Ilustración 11: Diagrama de flujo de Creación y modificación

4.4 Diseño de la base de datos

Para detallar el diseño del sistema de base de datos en MySQL mostrado en la ilustración12, a continuación, se presenta una descripción de la estructura de las tablas con sus relaciones.

Para el almacenamiento de información clave, se han creado nueve tablas, cada una correspondiente a una de las funcionalidades principales del sistema. Las tablas de usuarios y administradores se registran por separado para proporcionar una mayor seguridad. Los registros de los planes se guardan en una tabla separada, permitiendo un seguimiento detallado. La tabla ac se vincula a un único usuario, y la tabla bill se vincula con category para clasificar los registros.

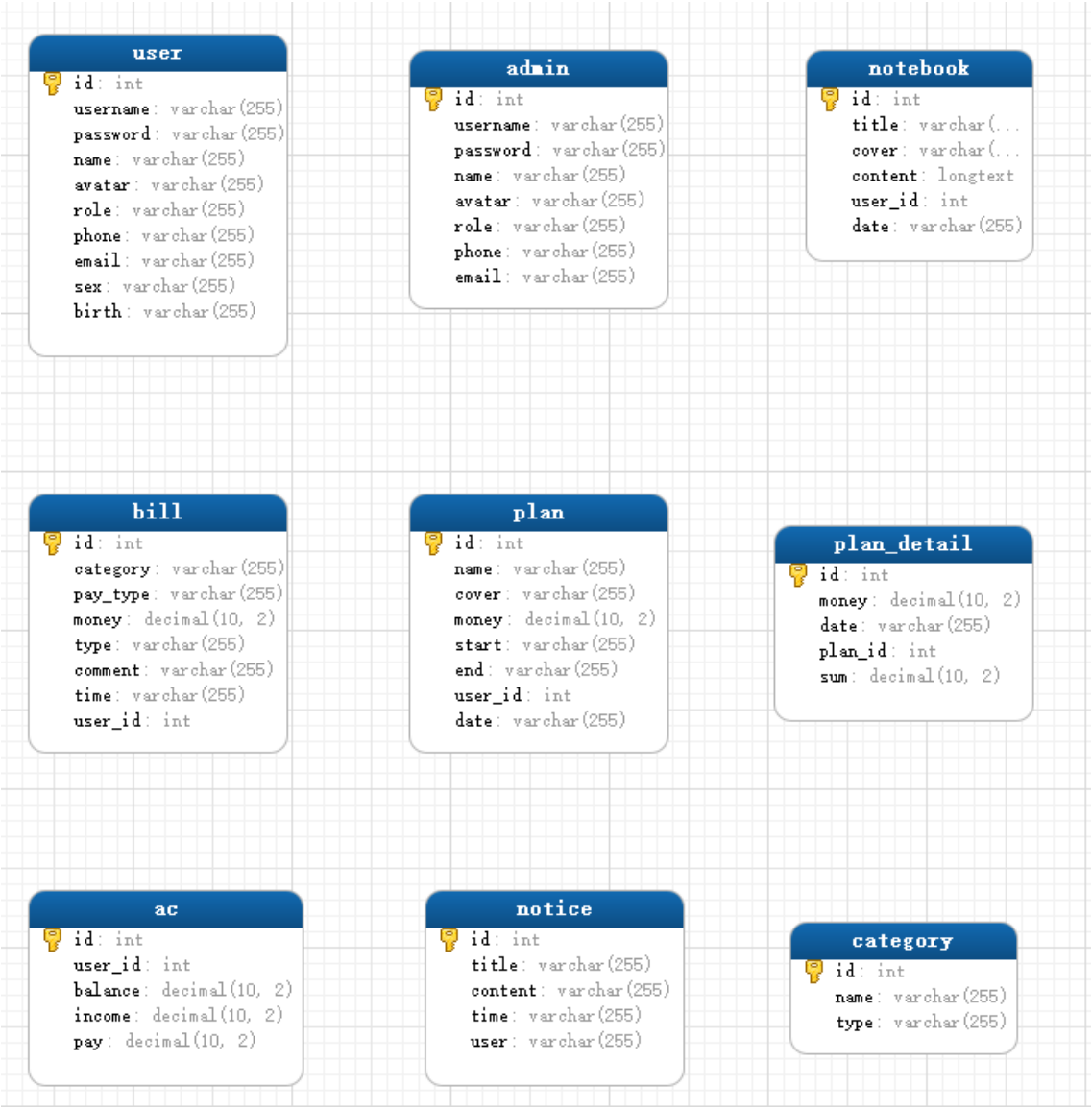


Ilustración 12: Base de datos

4.5 Diseño de las pantallas

En esta sección se describen las principales ventanas de la plataforma, proporcionando una guía detallada para su implementación posterior.

4.5.1 Pantalla de inicio de sesión.

Esta es la pantalla de inicio de sesión. Está diseñada como la primera interfaz con la que los usuarios interactúan al acceder a la plataforma. La pantalla cuenta con dos cuadros de entrada: uno para el nombre de usuario y otro para la contraseña. Además, incluye dos selectores que permiten elegir el tipo de usuario para la sesión. También contiene un botón para acceder a la pantalla de creación de una nueva cuenta. Por último, el botón más importante es el de "Iniciar sesión", que permite acceder a la pantalla principal si los campos de

nombre de usuario y contraseña coinciden con la información guardada en la base de datos.

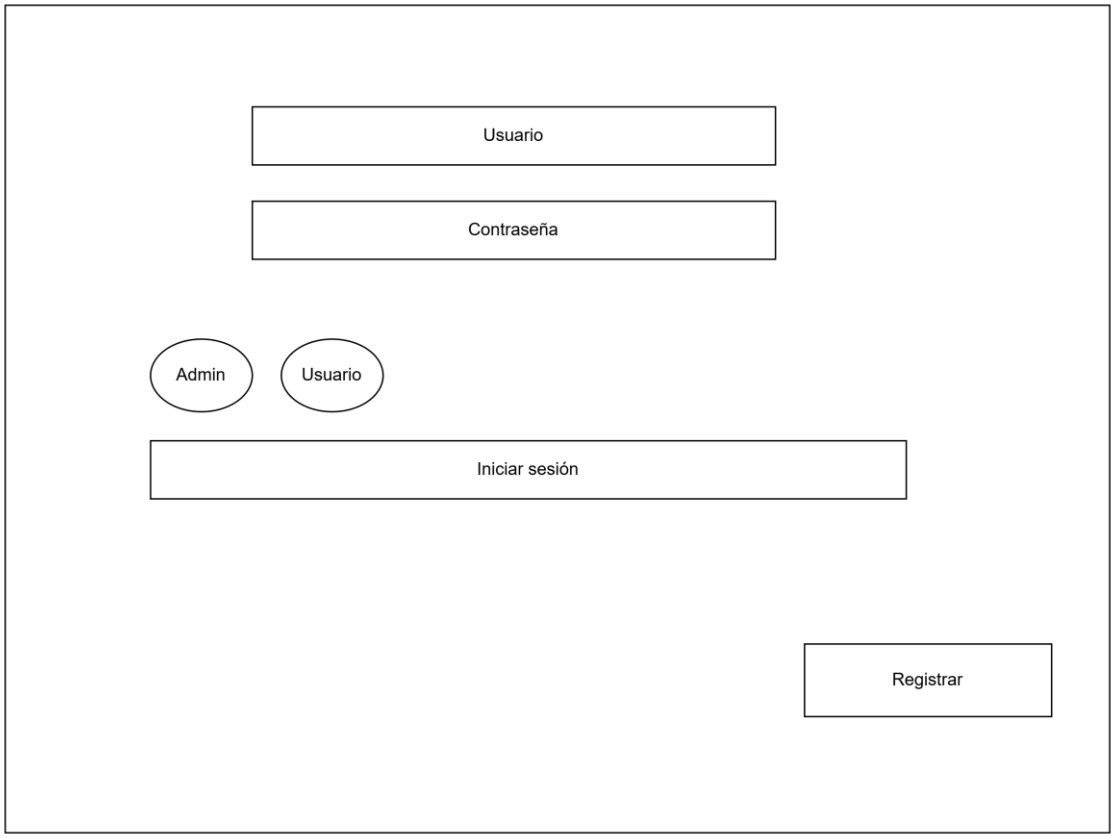


Ilustración 13: Pantalla de inicio de sesión

4.5.2 Pantalla de registro

Esta pantalla contiene tres cuadros de entrada: usuario, contraseña y repetir contraseña, respectivamente. Además, incluye un botón "Registrar" para guardar el registro de usuario. También cuenta con un botón para volver a la pantalla de inicio en caso de que el usuario ya tenga una cuenta registrada.

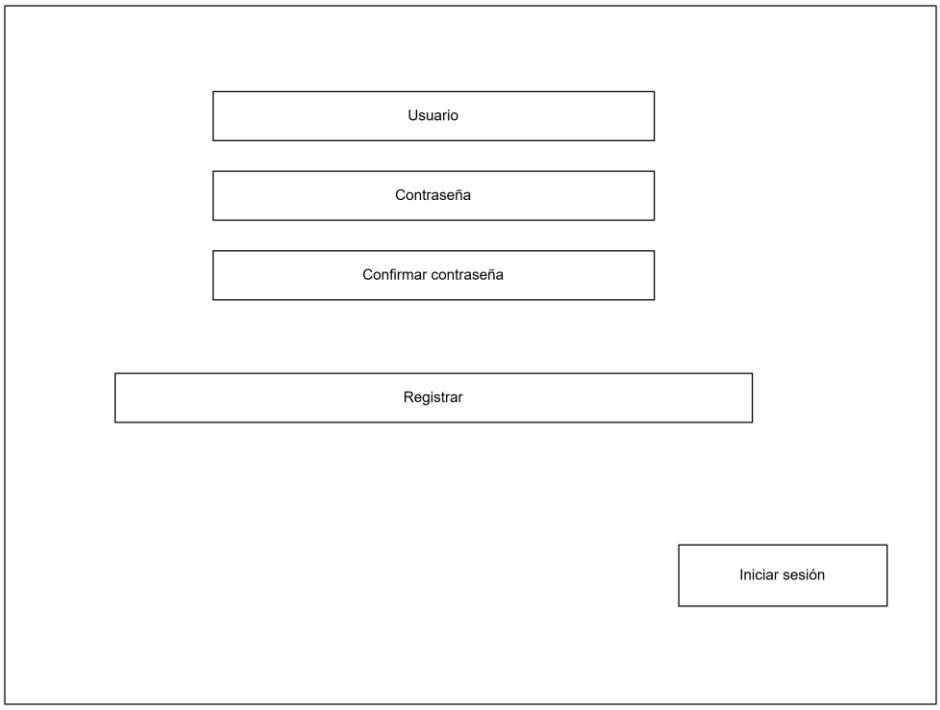


Ilustración 14: Pantalla de registro

4.5.3 Pantalla principal de usuario

Esta pantalla incluye secciones de bienvenida, resumen financiero, gastos e ingresos con gráficos circulares, y una lista de novedades en formato cronológico. También contiene un menú lateral para acceder a las tres herramientas de la aplicación.

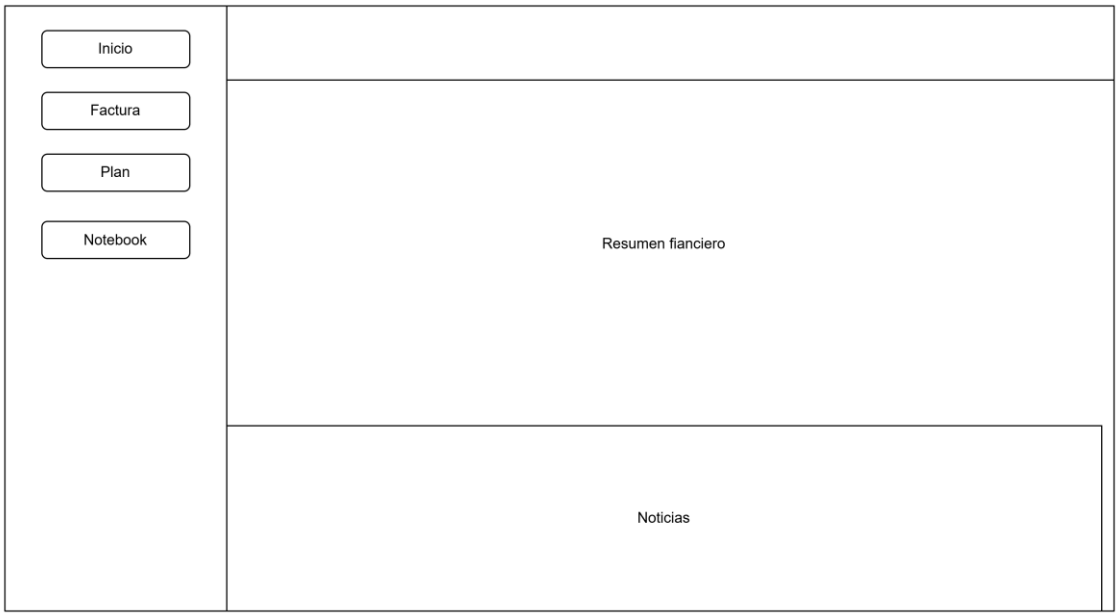


Ilustración 15: Pantalla principal de usuario

4.5.4 Pantalla principal de administrador

En la pantalla principal del administrador se presenta una barra de navegación lateral a la izquierda con botones para acceder a las secciones principales, tales como Inicio, Gestión de Cuentas, Gestión de Facturas, Gestión de Planes, Gestión de Categorías, Gestión de Noticias y Gestión de Usuarios.

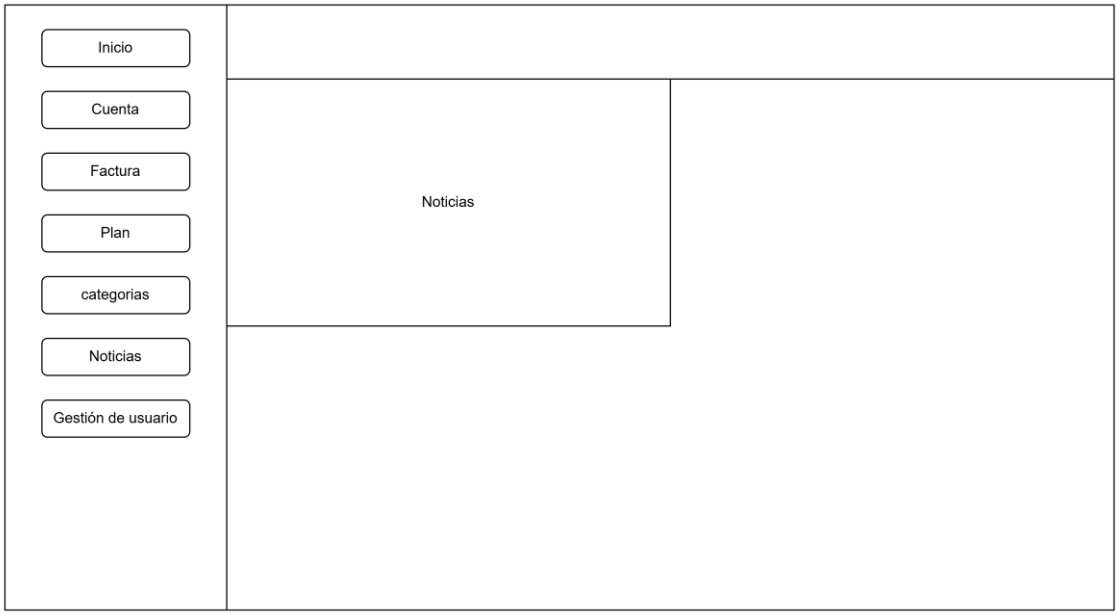


Ilustración 16: Pantalla principal de usuario

4.5.5 Pantalla de facturas

En esta pantalla se mostrará una lista de los registros de gastos e ingresos. También contará con un botón para añadir nuevas facturas, así como con un botón para eliminar múltiples registros simultáneamente. Además, cada factura tendrá un botón para modificarla y otro para eliminarla.

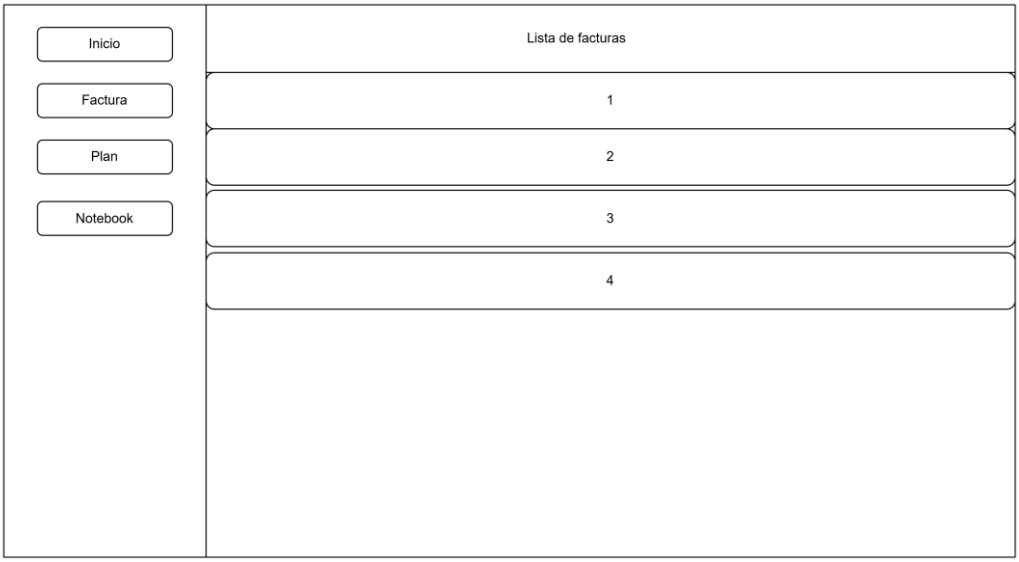


Ilustración 17: Pantalla de facturas

4.5.6 Pantalla de planes

En esta pantalla se mostrará una lista de los planes financieros. Es similar a la pantalla de facturas, pero con una visualización diferente, como se muestra en la ilustración 18. Además, cada plan tendrá una barra de progreso y un botón para registrar nuevos ingresos.

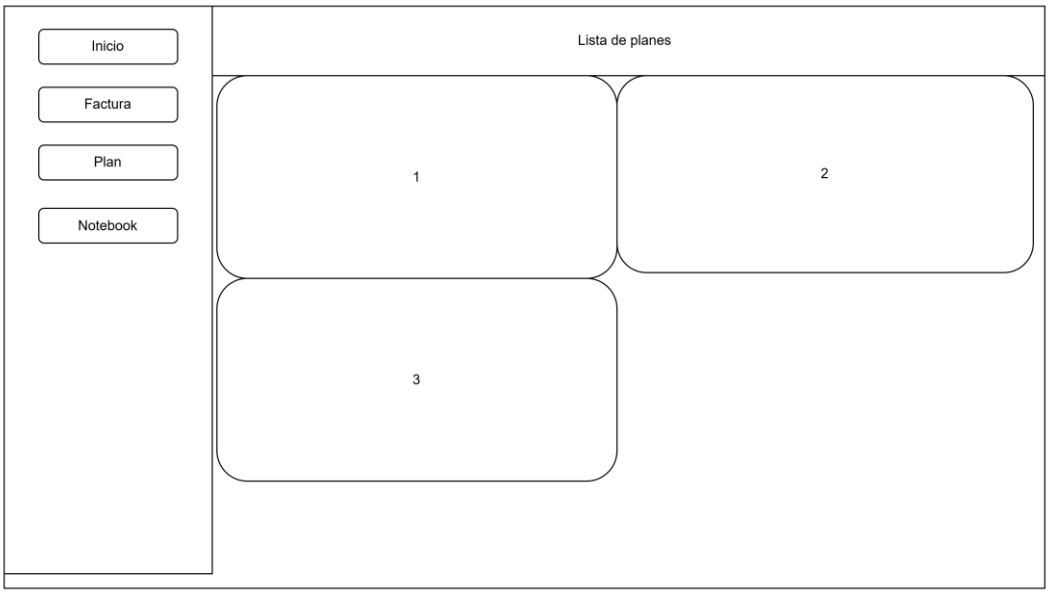


Ilustración 18: Pantalla de planes

4.5.7 Pantalla de diarios

En esta pantalla se mostrará una lista de los diarios escritos por el usuario, funcionando como un bloc de notas. Como se muestra en la ilustración 19, esta

pantalla es bastante similar a las dos pantallas anteriores. Sin embargo, en esta pantalla se mostrarán el título y la portada de cada nota.

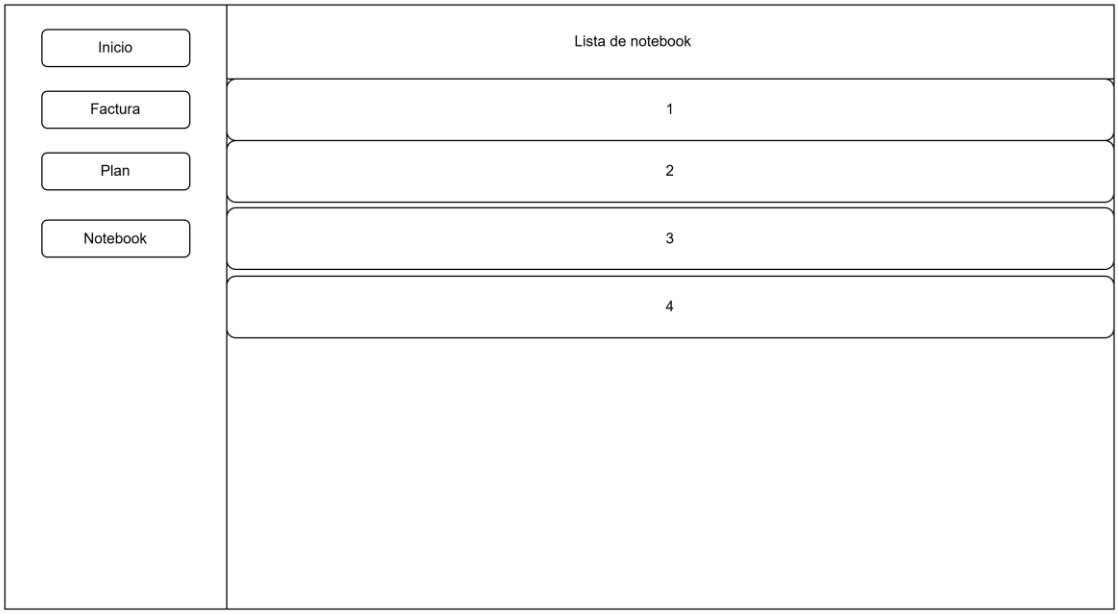


Ilustración 19: Pantalla de diarios

5 Desarrollo

En este capítulo se detallará el proceso de desarrollo de la plataforma, abarcando desde la estructura y organización del proyecto hasta la implementación de las funcionalidades clave. El objetivo es proporcionar una visión general de cómo se ha llevado a cabo el desarrollo, destacando las decisiones de diseño que han guiado la creación del sistema.

El proyecto se divide en dos componentes principales: el backend y el frontend. El backend, desarrollado con Spring Boot. Por otro lado, el frontend, construido con Vue.js.

5.1 Estructura del proyecto

En esta sección explicaremos la estructura del proyecto, proporcionando una visión detallada de cómo se ha organizado el código y los componentes del sistema.

5.1.1 Spring Boot

La estructura del proyecto se muestra en la ilustración 20, y se detalla a continuación, explicando el propósito y la funcionalidad de cada paquete y archivo.

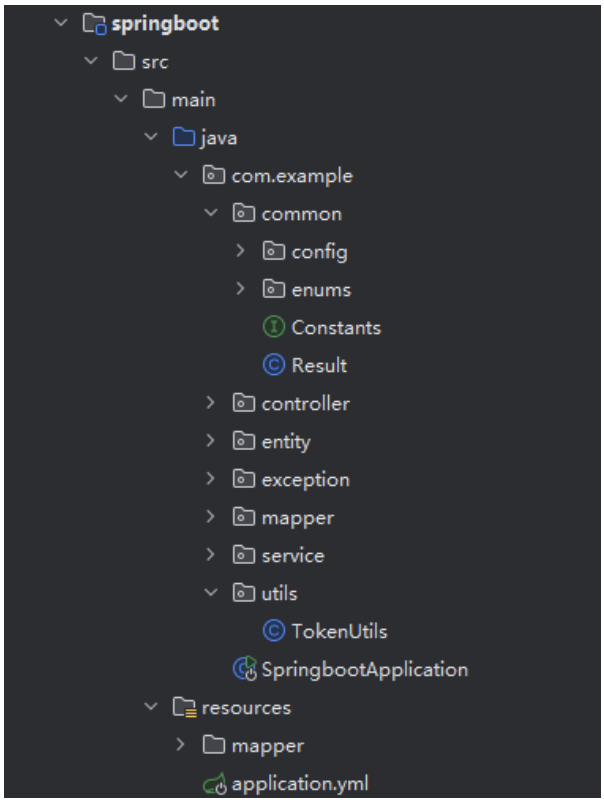


Ilustración 20: Estructura de Spring Boot

El paquete common incluye subpaquetes como config, que contiene las clases de configuración del proyecto, incluyendo configuraciones de seguridad, CORS, JWT y WebConfig; enums, que define las enumeraciones utilizadas en el proyecto, proporcionando un conjunto de constantes que representan valores fijos; y clases como Constants y Result, que contienen constantes globales y definen el formato de las respuestas estándar de la API, respectivamente.

El paquete controller maneja las solicitudes HTTP y define los endpoints de la API. Este paquete actúa como la capa de presentación que dirige las solicitudes a los servicios y devuelve las respuestas correspondientes. En el paquete entity, se definen las entidades del dominio que representan las tablas de la base de datos, utilizando anotaciones de JPA para mapear las entidades a las tablas correspondientes.

El paquete exception maneja las excepciones personalizadas y globales para el manejo de errores. Por otro lado, el paquete com.example.mapper contiene las interfaces CrudRepository para realizar operaciones CRUD en la base de datos.

El paquete service implementa la lógica de negocio y coordina las operaciones entre los controladores y los repositorios, asegurando que la lógica de negocio esté separada de los detalles de la implementación del controlador y del acceso a los datos. El paquete com.example.utils incluye TokenUtils, que gestiona la creación y validación de tokens JWT para la autenticación y autorización.

5.1.2 Vue.js

La estructura del proyecto se muestra en la imagen adjunta, y se detalla a continuación.

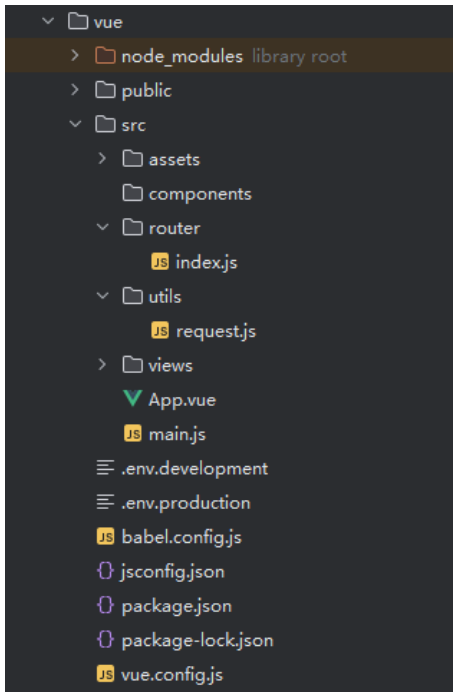


Ilustración 21: Estructura de Vue

Dentro de src, el directorio assets alberga recursos estáticos como imágenes y archivos CSS globales utilizados a lo largo de la aplicación.

La configuración de las rutas de navegación de la aplicación se encuentra en el directorio router, específicamente en el archivo index.js, que mapea las URL a los componentes correspondientes. El directorio utils contiene request.js, que maneja las solicitudes HTTP y la interacción con la API backend. Por su parte, el directorio views alberga las vistas principales de la aplicación, con cada vista representando una pantalla o página completa de la aplicación.

Adicionalmente, el proyecto incluye varios archivos de configuración importantes. Los archivos env.development y .env.production definen variables específicas para los entornos de desarrollo y producción. babel.config.js contiene la configuración para Babel, el transpilador de JavaScript. jsconfig.json habilita características como la importación de módulos absolutos. package.json y package-lock.json enumeran las dependencias del proyecto y las versiones bloqueadas de esas dependencias, respectivamente. Finalmente, vue.config.js permite la personalización del comportamiento de Webpack para el proyecto Vue.js.

5.2 Desarrollo backend

Se describirá el desarrollo del backend implementado con Spring Boot.

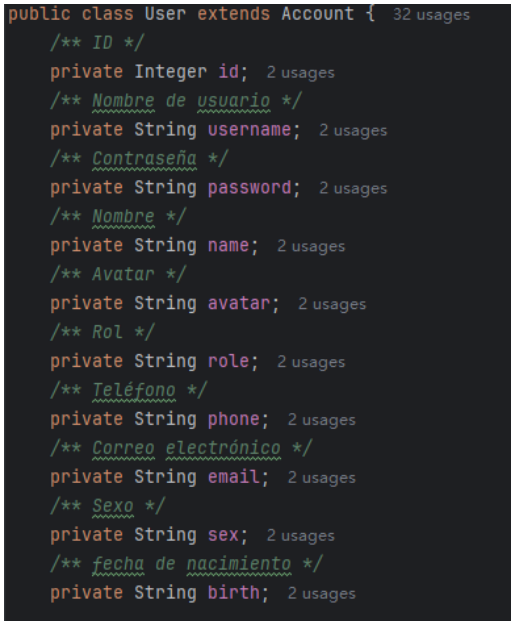
5.2.1 Gestión de usuarios

Esta sección detalla cómo se implementan las operaciones relacionadas con las funcionalidades de los usuarios.

5.2.1.1 Registrar

Para gestionar la creación de nuevos usuarios en el sistema, he seguido el siguiente proceso. Primero, definí la entidad User en el archivo User.java, que incluye atributos como id, username, password, name, avatar, role, phone, email, sex, y birth. Estos son los atributos básicos necesarios para identificar a los usuarios, junto con los métodos getter y setter necesarios para estos atributos. Esta entidad sirve como base para representar a los usuarios dentro de la aplicación.

El atributo role es clave para identificar si el usuario pertenece al grupo de usuarios regulares o al de administradores. Además, es importante destacar los atributos password y username, ya que estos dos se utilizan para autenticar el inicio de sesión de los usuarios.



```
public class User extends Account {  32 usages
    /** ID */
    private Integer id;  2 usages
    /** Nombre de usuario */
    private String username;  2 usages
    /** Contraseña */
    private String password;  2 usages
    /** Nombre */
    private String name;  2 usages
    /** Avatar */
    private String avatar;  2 usages
    /** Rol */
    private String role;  2 usages
    /** Teléfono */
    private String phone;  2 usages
    /** Correo electrónico */
    private String email;  2 usages
    /** Sexo */
    private String sex;  2 usages
    /** fecha de nacimiento */
    private String birth;  2 usages
```

Ilustración 22: Usuario

Luego, implementé los servicios de usuarios en UserService.java, desarrollando, entre otros, el método add(User user) para añadir nuevos usuarios. Este método verifica si el nombre de usuario ya existe en la base de datos para evitar duplicados. Al principio del desarrollo, cuando el usuario está en el proceso de registrarse, si no se proporciona una contraseña, el método asigna una predeterminada y también asigna el rol de usuario por defecto antes de insertar el nuevo usuario en la base de datos. Posteriormente, en el desarrollo del frontend, se obligó a los usuarios a introducir la contraseña.

```
/**
 * ADD
 */
public void add(User user) {
    User dbUser = userMapper.selectByUsername(user.getUsername());
    if (ObjectUtil.isNotNull(dbUser)) {
        throw new CustomException(ResultCodeEnum.USER_EXIST_ERROR);
    }
    if (ObjectUtil.isEmpty(user.getPassword())) {
        user.setPassword(Constants.USER_DEFAULT_PASSWORD);
    }
    if (ObjectUtil.isEmpty(user.getName())) {
        user.setName(user.getUsername());
    }
    user.setRole(RoleEnum.USER.name());
    userMapper.insert(user);
}
```

Ilustración 23: Creación de nuevo usuario

Posteriormente, en el archivo UserController.java, definí varios endpoints HTTP para gestionar las operaciones relacionadas con los usuarios. Entre estos endpoints se incluye el endpoint para añadir nuevos usuarios, que utiliza el método add del UserService.

```java
@PostMapping("/add")
public Result add(@RequestBody User user) {
    userService.add(user);
    return Result.success();
}
```

Ilustración 24: add del UserService

Para asegurar la autenticación y autorización dentro del sistema, implementé un interceptor JWT en JwtInterceptor.java, mostrado en la ilustración 28. Y configuré este interceptor en WebConfig.java para que se aplique a todas las rutas, excepto a aquellas específicas como login y registro.

```java
@Override  no usages
public void addInterceptors(InterceptorRegistry registry) {
    registry.addInterceptor(jwtInterceptor).addPathPatterns("/**")
            .excludePathPatterns("/")
            .excludePathPatterns("/login")
            .excludePathPatterns("/register")
            .excludePathPatterns("/files/**")
            .excludePathPatterns("/bill/export");
}
```

Ilustración 25: WebConfig

Finalmente, en WebController.java, manejé la autenticación y el registro de usuarios. Implementé el endpoint /register para registrar nuevos usuarios llamando al método register del UserService.

```java
@PostMapping("/register")
public Result register(@RequestBody Account account) {
    if (StrUtil.isBlank(account.getUsername()) || StrUtil.isBlank(account.getPassword())) {
        return Result.error(ResultCodeEnum.PARAM_LOST_ERROR);
    }
    if (RoleEnum.USER.name().equals(account.getRole())) {
        userService.register(account);
    } else {
        return Result.error(ResultCodeEnum.PARAM_ERROR);
    }
    return Result.success();
}
```

Ilustración 26: Register

5.2.1.2 Inicio de sesión

En la implementación del inicio de sesión tantos de usuarios como administrador. Primero, implementé en UserService.java el método login(Account account). Este método verifica si el nombre de usuario existe en la base de datos. Si el usuario no existe, lanza una excepción personalizada CustomException con un código de error específico. Luego, compara la contraseña proporcionada con la almacenada en la base de datos. Si las credenciales son correctas, se genera un token JWT usando la contraseña del usuario como clave de firma. Este token contiene información codificada del usuario, como su ID y rol. Finalmente, el token se asigna al objeto Account que se devuelve al usuario como parte de la respuesta de inicio de sesión.

```
public Account login(Account account) {  1 usage
    Account dbUser = userMapper.selectByUsername(account.getUsername());
    if (ObjectUtil.isNull(dbUser)) {
        throw new CustomException(ResultCodeEnum.USER_NOT_EXIST_ERROR);
    }
    if (!account.getPassword().equals(dbUser.getPassword())) {
        throw new CustomException(ResultCodeEnum.USER_ACCOUNT_ERROR);
    }
    // generar token
    String tokenData = dbUser.getId() + "-" + RoleEnum.USER.name();
    String token = TokenUtils.createToken(tokenData, dbUser.getPassword());
    dbUser.setToken(token);
    return dbUser;
}
```

Ilustración 27: Login

En JwtInterceptor.java, configuré un interceptor JWT para manejar la autenticación de todas las solicitudes HTTP. Este interceptor extrae el token del encabezado de la solicitud y lo verifica usando la biblioteca JWT. Decodifica el token para obtener el ID del usuario y su rol, y luego consulta la base de datos para obtener los detalles del usuario. Si el token es válido, la solicitud procede, de lo contrario, se lanza una excepción que indica un error de autenticación.

```java
@Override  no usages
public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) {
    // 1. Obtener el token del header de la solicitud HTTP
    String token = request.getHeader(Constants.TOKEN);
    if (ObjectUtil.isEmpty(token)) {
        // Si no se obtiene, intentar obtenerlo de los parámetros
        token = request.getParameter(Constants.TOKEN);
    }
    // 2. Ejecutar la autenticación
    if (ObjectUtil.isEmpty(token)) {
        throw new CustomException(ResultCodeEnum.TOKEN_INVALID_ERROR);
    }
    Account account = null;
    try {
        // Analizar el token para obtener los datos almacenados
        String userRole = JWT.decode(token).getAudience().get(0);
        String userId = userRole.split( regex: "-")[0];
        String role = userRole.split( regex: "-")[1];
        // Consultar la base de datos según el userId
        if (RoleEnum.ADMIN.name().equals(role)) {
            account = adminService.selectById(Integer.valueOf(userId));
        } else if (RoleEnum.USER.name().equals(role)) {
            account = userService.selectById(Integer.valueOf(userId));
        }
    } catch (Exception e) {
        throw new CustomException(ResultCodeEnum.TOKEN_CHECK_ERROR);
    }
    if (ObjectUtil.isNull(account)) {
        throw new CustomException(ResultCodeEnum.USER_NOT_EXIST_ERROR);
    }
    try {
        // Verificar el token usando la contraseña del usuario
        JWTVerifier jwtVerifier = JWT.require(Algorithm.HMAC256(account.getPassword())).build();
        jwtVerifier.verify(token); // Verificar el token
    } catch (JWTVerificationException e) {
        throw new CustomException(ResultCodeEnum.TOKEN_CHECK_ERROR);
    }
    return true;
}
```

Ilustración 28: JwtInterceptor

Para integrar el servicio de autenticación en la aplicación, seguí un proceso similar al de la configuración del registro de usuarios. Configuré el archivo WebConfig.java para gestionar los aspectos de seguridad y autenticación.

Finalmente, en WebController.java, implementé el endpoint /login para manejar las solicitudes de inicio de sesión. Este endpoint recibe las credenciales de usuario a través de un objeto Account. El método login verifica que el nombre de usuario, la contraseña y el rol no estén vacíos antes de delegar la autenticación al UserService o AdminService, según el rol especificado. Si las credenciales son válidas, el método devuelve un objeto Result que contiene el token JWT y otros detalles del usuario, indicando un inicio de sesión exitoso.

```java
@PostMapping("/login")
public Result login(@RequestBody Account account) {
    if (ObjectUtil.isEmpty(account.getUsername()) || ObjectUtil.isEmpty(account.getPassword())
            || ObjectUtil.isEmpty(account.getRole())) {
        return Result.error(ResultCodeEnum.PARAM_LOST_ERROR);
    }
    if (RoleEnum.ADMIN.name().equals(account.getRole())) {
        account = adminService.login(account);
    } else if (RoleEnum.USER.name().equals(account.getRole())) {
        account = userService.login(account);
    } else {
        return Result.error(ResultCodeEnum.PARAM_ERROR);
    }
    return Result.success(account);
}
```

Ilustración 29: Login

5.2.1.3 Cambio de contraseña

Para gestionar el cambio de contraseña en el sistema, he implementado un proceso que abarca desde la verificación de las credenciales actuales del usuario hasta la actualización segura de su nueva contraseña.

Primero, en UserService.java, desarrollé el método updatePassword(Account account). Este método comienza verificando si el nombre de usuario existe en la base de datos. Si el usuario no existe, se lanza una excepción personalizada CustomException con un código de error específico. Luego, compara la contraseña actual proporcionada con la almacenada en la base de datos. Si las credenciales coinciden, se actualiza la contraseña del usuario con la nueva proporcionada.

```java
//cambio de password
public void updatePassword(Account account) {  1 usage
    User dbUser = userMapper.selectByUsername(account.getUsername());
    if (ObjectUtil.isNull(dbUser)) {
        throw new CustomException(ResultCodeEnum.USER_NOT_EXIST_ERROR);
    }
    if (!account.getPassword().equals(dbUser.getPassword())) {
        throw new CustomException(ResultCodeEnum.PARAM_PASSWORD_ERROR);
    }
    dbUser.setPassword(account.getNewPassword());
    this.updateById(dbUser);
}
```

Ilustración 30: updatePassword de UserService

Para integrar esta funcionalidad, también he definido un endpoint específico en WebController.java como login y register. El endpoint /updatePassword recibe las credenciales del usuario a través de un objeto Account. Este objeto contiene

el nombre de usuario, la contraseña actual y la nueva contraseña. El método verifica que todos los campos necesarios no estén vacíos. Dependiendo del rol del usuario (ADMIN o USER), delega la operación de cambio de contraseña al AdminService o UserService respectivamente.

```
@PutMapping("/updatePassword")
public Result updatePassword(@RequestBody Account account) {
    if (StrUtil.isBlank(account.getUsername()) || StrUtil.isBlank(account.getPassword())
            || ObjectUtil.isEmpty(account.getNewPassword())) {
        return Result.error(ResultCodeEnum.PARAM_LOST_ERROR);
    }
    if (RoleEnum.ADMIN.name().equals(account.getRole())) {
        adminService.updatePassword(account);
    } else if (RoleEnum.USER.name().equals(account.getRole())) {
        userService.updatePassword(account);
    } else {
        return Result.error(ResultCodeEnum.PARAM_ERROR);
    }
    return Result.success();
}
```

Ilustración 31: updatePassword de WebController

5.2.2 Cuentas

Para gestionar el registro del saldo, y de los ingresos y gastos totales de los usuarios, se ha desarrollado la clase "Ac". Esta clase incluye un identificador único ("id") que se asocia a un usuario específico mediante el identificador del usuario ("user_id").

```
public class Ac implements Serializable {  32 usages
    private static final long serialVersionUID = 1L;  no usages

    /** ID */
    private Integer id;  2 usages
    /** USER ID */
    private Integer userId;  2 usages
    /** SALDO */
    private BigDecimal balance;  2 usages
    /** INGRESO */
    private BigDecimal income;  2 usages
    /** GASTO */
    private BigDecimal pay;  2 usages
    /** USER NAME */
    private String userName;  2 usages
```

Ilustración 32: Cuentas

5.2.3 Categorías

Además, se ha creado la clase "Category". Esta clase se encarga de categorizar los registros financieros y está diseñada con un identificador único y un nombre para cada categoría. Adicionalmente, clasifica las categorías como de gasto o ingreso.

```
public class Category implements Serializable {  27 usages
    private static final long serialVersionUID = 1L;  no usages

    /** ID */
    private Integer id;  2 usages
    /** Nombre de la categoria */
    private String name;  2 usages
    /** Tipo */
    private String type;  2 usages

    public Integer getId() {
        return id;
    }
```

Ilustración 33: Categorías

5.2.4 Facturas

La clase Bill se ha creado para registrar los detalles de cada gasto e ingreso. Esta clase está diseñada para almacenar información detallada sobre las transacciones financieras de los usuarios. Entre los atributos de esta clase se encuentra category, que se encarga de diferenciar la categoría de la transacción, permitiendo al usuario saber en qué ha gastado su dinero.

Además del atributo category, la clase Bill incluye otros atributos esenciales. Estos atributos permiten una gestión detallada de los registros de transacciones de los usuarios, facilitando el seguimiento.

```java
public class Bill implements Serializable {  40 usages
    private static final long serialVersionUID = 1L;  no usages


    private Integer id;  2 usages

    private String category;  2 usages

    private String payType;  2 usages

    private BigDecimal money;  2 usages

    private String type;  2 usages

    private String comment;  2 usages

    private String time;  2 usages

    private Integer userId;  2 usages

    private String userName;  2 usages

    private String start;  2 usages

    private String end;  2 usages

    private Integer percent;  2 usages
```

Ilustración 34: Facturas

5.2.4.1 Añadir nueva transacción

En el archivo BillService.java, implementé el método add para manejar la creación de nuevas facturas. Primero, obtengo la hora actual y el usuario autenticado, asignando el userId si el rol es USER. Luego, inserto la factura en la base de datos.

Después, verifico si el usuario tiene una cuenta (Ac). Si no la tiene, creo una nueva con saldos iniciales en cero. Finalmente, actualizo el saldo y los totales de ingresos o gastos de la cuenta según el tipo de factura (gasto o ingreso).

```java
@Transactional
public void add(Bill bill) {
    bill.setTime(DateUtil.now());
    Account currentUser = TokenUtils.getCurrentUser();
    if (RoleEnum.USER.name().equals(currentUser.getRole())) {  // solo user
        bill.setUserId(currentUser.getId());
    }
    billMapper.insert(bill);

    // cerar nueva cuenta
    Ac ac = acService.selectByUserId(currentUser.getId());
    if (ac == null) {
        ac = new Ac();
        ac.setUserId(currentUser.getId());
        ac.setBalance(BigDecimal.ZERO);
        ac.setPay(BigDecimal.ZERO);
        ac.setIncome(BigDecimal.ZERO);
        acService.add(ac);
    }

    // actualizo cuenta
    if (AcTypeEnum.PAY.getValue().equals(bill.getType())) {  // gasto
        ac.setPay(ac.getPay().add(bill.getMoney()));
        ac.setBalance(ac.getBalance().subtract(bill.getMoney()));  // saldo
    } else {
        ac.setIncome(ac.getIncome().add(bill.getMoney()));  // ingreso
        ac.setBalance(ac.getBalance().add(bill.getMoney()));  // saldo
    }
    acService.updateById(ac);
}
```

Ilustración 35: Añadir nueva transacción

5.2.4.2 Exportación de registros

Para gestionar la exportación de transacciones en el sistema, he implementado un proceso que permite a los usuarios exportar sus transacciones financieras a un archivo Excel.

```java
@GetMapping("export")
public void export(@RequestParam(required = false) Integer userID, HttpServletResponse response) throws UnsupportedOperationException, IOException {
    List<Bill> billList;
    if(userID != null){
        Bill bill = new Bill();
        bill.setUserId(userID);
        billList = billService.selectAll(bill);
    } else {
        billList = billService.selectAll( bill: null);
    }

    ExcelWriter excelWriter = ExcelUtil.getWriter();
    excelWriter.write(billList);
    response.setContentType("application/vnd.openxmlformats-officedocument.spreadsheetml.sheet;charset=utf-8");
    response.setHeader( s: "Content-Disposition", s1: "attachment;filename=" + URLEncoder.encode( s: "Registro", enc: "UTF-8") + ".xlsx");
    ServletOutputStream os = response.getOutputStream();
    excelWriter.flush(os, isCloseOut: true);
    excelWriter.close();
    os.close();

}
```

Ilustración 36: Export

5.2.5 Notas

Con el fin de proporcionar a los usuarios una interfaz donde puedan almacenar información importante, se ha creado la clase "Notebook", que funciona como un bloc de notas.

```java
/**
 * Accounting Diary
 */
public class Notebook implements Serializable {  27 usages
    private static final long serialVersionUID = 1L;  no usages

    /** ID */
    private Integer id;  2 usages
    /** Title */
    private String title;  2 usages
    /** Cover */
    private String cover;  2 usages
    /** Content */
    private String content;  2 usages
    /** User ID */
    private Integer userId;  2 usages
    /** Date of Publication */
    private String date;  2 usages

    private  String userName;  2 usages
```

Ilustración 37: Notas

5.2.6 Ahorro

Proporcionaremos una herramienta que permita a los usuarios establecer metas de ahorro y realizar un seguimiento de su progreso.

Para ello, hemos creado las clases "Plan" y "PlanDetail". La clase "Plan" se utiliza para definir y gestionar los objetivos de ahorro, mientras que la clase "PlanDetail" sirve para registrar los detalles específicos de cada transacción dentro de un plan de ahorro. "PlanDetail" se relaciona con "Plan" por el identificador del plan de ahorro(planId).

```
public class Plan implements Serializable {  31 usages
    private static final long serialVersionUID = 1L;  no usages

    /** ID */
    private Integer id;  2 usages
    /** nombre */
    private String name;  2 usages
    /** portada */
    private String cover;  2 usages
    /** meta */
    private BigDecimal money;  2 usages
    /** inicio */
    private String start;  2 usages
    /** final */
    private String end;  2 usages
    /** user ID */
    private Integer userId;  2 usages
    /** comienzo */
    private String date;  2 usages

    private Integer percent;  2 usages

    private String status;  2 usages

    public String userName;  2 usages
```

Ilustración 38: Ahorro

```
public class PlanDetail implements Serializable {  31 usages
    private static final long serialVersionUID = 1L;  no usages

    /** ID */
    private Integer id;  2 usages
    /** cantidad */
    private BigDecimal money;  2 usages
    private BigDecimal sum;  2 usages
    /** fecha */
    private String date;  2 usages
    /** plan ID */
    private Integer planId;  2 usages
```

Ilustración 39: Detalles de ahorro

5.2.6.1 Crear plan de ahorro

Para agregar un nuevo plan al sistema en el add de PlanService se asigna el ID del usuario actual al plan y valida las fechas de inicio y fin, lanzando excepciones si son incorrectas. Luego, establece la fecha actual en el plan y lo inserta en la base de datos.

```
public void add(Plan plan) {
    Account currentUser = TokenUtils.getCurrentUser();
    if (RoleEnum.USER.name().equals(currentUser.getRole())) {
        plan.setUserId(currentUser.getId());
    }
    if (DateUtil.parseDate(plan.getStart()).isBefore(new Date())) {
        throw new CustomException(ResultCodeEnum.DATE_START_ERROR);
    }
    if (DateUtil.parseDate(plan.getStart()).isAfterOrEquals(DateUtil.parseDate(plan.getEnd()))) {
        throw new CustomException(ResultCodeEnum.DATE_ERROR);
    }
    plan.setDate(DateUtil.today());
    planMapper.insert(plan);
}
```

Ilustración 40: add de PlanService

5.2.6.2 Actualizar plan de ahorro

Cuando el usuario añade dinero a un plan, este se actualiza de la siguiente manera: primero, se obtiene la suma total almacenada más reciente para este plan. Luego, se actualiza esta suma añadiendo el monto de la nueva aportación y se establece esta nueva suma. Finalmente, se inserta el detalle en la base de datos. Este proceso asegura que cada nuevo detalle del plan esté actualizado con la suma total correcta y la fecha actual.

```
 */
public void add(PlanDetail planDetail) {
    BigDecimal sum = this.getFirstSum(planDetail.getPlanId());
    // Insertar nuevos datos totales
    planDetail.setSum(sum.add(planDetail.getMoney()));
    planDetail.setDate(DateUtil.today());
    planDetailMapper.insert(planDetail);
}

/**
 *
 * Obtenga la última cantidad total almacenada
 */
public BigDecimal getFirstSum(Integer planId) {  2 usages
    // Primero verifique el monto total depositado
    PlanDetail param = new PlanDetail();
    param.setPlanId(planId);
    List<PlanDetail> planDetailList = planDetailMapper.selectAll(param);// Consulta la lista detallada
                                                                         // de todos los planes del plan actual.
    BigDecimal sum = BigDecimal.ZERO;
    if (CollUtil.isNotEmpty(planDetailList)) {
        PlanDetail detail = planDetailList.get(0);// El último plan de ahorro de dinero.
        sum = detail.getSum();  // Obtenga el último total de almacenamiento
    }
    return sum;
}
```

Ilustración 41: Actualizar plan de ahorro

5.2.6.3 Mostrar plan de ahorro

La consulta paginada de planes es diferente a las otras consultas. Ya que es necesario actualizar los detalles de cada plan. Para ello se ha implementado una función auxiliar setPlan, donde se establece los detalles.

```
public PageInfo<Plan> selectPage(Plan plan, Integer pageNum, Integer pageSize) {
    Account currentUser = TokenUtils.getCurrentUser();
    if (RoleEnum.USER.name().equals(currentUser.getRole())) { //solo mostrar la plan de si mismo
        plan.setUserId(currentUser.getId());
    }
    PageHelper.startPage(pageNum, pageSize);
    List<Plan> list = planMapper.selectAll(plan);
    PageInfo<Plan> pageInfo = PageInfo.of(list);
    for (Plan p : pageInfo.getList()) {
        this.setPlan(p);
    }
    return pageInfo;
}

public void setPlan(Plan p) {  2 usages
    BigDecimal sum = planDetailService.getFirstSum(p.getId());
    if (sum.compareTo(p.getMoney()) >= 0) {  // La cantidad total almacenada excede la cantidad planificada
        p.setPercent(100);
    } else {
        int percent = sum.divide(p.getMoney(),  scale: 2, RoundingMode.HALF_UP).multiply(BigDecimal.valueOf(100)).intValue();
        p.setPercent(percent);
    }
    Date today = new Date();
    if (DateUtil.parseDate(p.getStart()).isAfter(today)) {
        p.setStatus(PlanStatusEnum.NO_READY.getValue());  // Establecer estado de depósito
    } else if (DateUtil.parseDate(p.getStart()).isBeforeOrEquals(today) && DateUtil.parseDate(p.getEnd()).isAfterOrEquals(today)) {
        p.setStatus(PlanStatusEnum.IN_PROGRESS.getValue());  // Establecer estado de depósito
    } else if (DateUtil.parseDate(p.getEnd()).isBefore(today)) {
        p.setStatus(PlanStatusEnum.END.getValue());  // Establecer estado de depósito
    }

    if (sum.compareTo(p.getMoney()) >= 0) {
        p.setStatus(PlanStatusEnum.DONE.getValue());  //Una vez que haya ahorrado suficiente dinero, estará listo.
    }
}
```

Ilustración 42: Mostrar plan de ahorro

5.2.7 Función de estadística

Para una mejor visualización, hemos decidido añadir gráficos estadísticos que muestran el porcentaje de gastos e ingresos de cada categoría. Ha sido necesario añadir una nueva función en la clase Bill. Su función es calcular el resumen de las facturas agrupadas por categoría para un usuario específico y devolver una lista de objetos Bill con los montos totales y los porcentajes correspondientes para cada categoría.

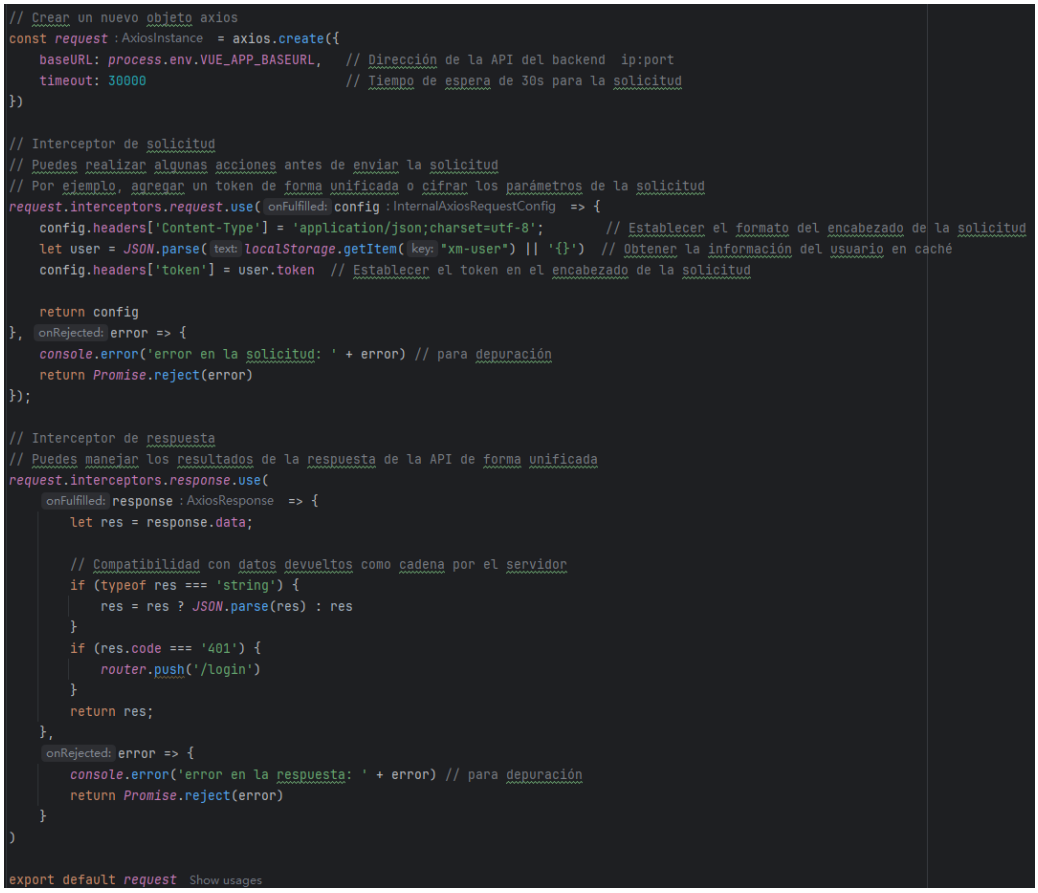
```
public List<Bill> count(String type) {  1 usage
    Bill bill = new Bill();
    Account currentUser = TokenUtils.getCurrentUser();
    bill.setUserId(currentUser.getId());
    bill.setType(type);
    List<Bill> billList = billMapper.selectAll(bill);  //Consultar toda la información de facturación según el ID de usuario actualmente conectado
    BigDecimal sum = billList.stream().map(Bill::getMoney).reduce(BigDecimal::add).orElse(BigDecimal.ZERO);// Obtenga todos los montos de factura del tipo actual
    List<String> categoryList = billMapper.selectCategoryByType(type);
    List<Bill> list = new ArrayList<>();
    for (String category : categoryList) {
        Bill b = new Bill();
        b.setCategory(category);
        //Calcular el resumen de todos los importes de esta categoría.
        BigDecimal categorySum = billList.stream().filter(bi -> bi.getCategory().equals(category)).map(Bill::getMoney).reduce(BigDecimal::add).orElse(BigDecimal.ZERO);
        //Devuelve el importe total de la categoría actual.
        b.setMoney(categorySum);
        // obtener porcentaje de la factura
        b.setPercent(categorySum.divide(sum,  scale: 2, RoundingMode.HALF_UP).multiply(BigDecimal.valueOf(100)).intValue());
        list.add(b);
    }
    return list;
}
```

Ilustración 43: Función de estadística

5.3 Desarrollo frontend

En esta sección se describira los procesos más importantes en el desarrollo del frontend, utilizando Vue.js como framework principal.

Antes de diseñar las interfaces, lo primero de todo es configurar Axios para manejar peticiones HTTP con interceptores para solicitudes y respuestas, añadiendo interceptores para gestionar los encabezados de las solicitudes y las respuestas.



```
// Crear un nuevo objeto axios
const request : AxiosInstance  = axios.create({
    baseURL: process.env.VUE_APP_BASEURL,   // Dirección de la API del backend  ip:port
    timeout: 30000                          // Tiempo de espera de 30s para la solicitud
})

// Interceptor de solicitud
// Puedes realizar algunas acciones antes de enviar la solicitud
// Por ejemplo, agregar un token de forma unificada o cifrar los parámetros de la solicitud
request.interceptors.request.use( onFulfilled: config : InternalAxiosRequestConfig  => {
    config.headers['Content-Type'] = 'application/json;charset=utf-8';        // Establecer el formato del encabezado de la solicitud
    let user = JSON.parse( text: localStorage.getItem( key: "xm-user") || '{}')  // Obtener la información del usuario en caché
    config.headers['token'] = user.token  // Establecer el token en el encabezado de la solicitud

    return config
}, onRejected: error => {
    console.error('error en la solicitud: ' + error) // para depuración
    return Promise.reject(error)
});

// Interceptor de respuesta
// Puedes manejar los resultados de la respuesta de la API de forma unificada
request.interceptors.response.use(
    onFulfilled: response : AxiosResponse  => {
        let res = response.data;

        // Compatibilidad con datos devueltos como cadena por el servidor
        if (typeof res === 'string') {
            res = res ? JSON.parse(res) : res
        }
        if (res.code === '401') {
            router.push('/login')
        }
        return res;
    },
    onRejected: error => {
        console.error('error en la respuesta: ' + error) // para depuración
        return Promise.reject(error)
    }
)

export default request  Show usages
```

Ilustración 44: request.js

Después, hay que configurar las rutas de la aplicación usando Vue Router. Detallando las rutas de cada interfaz.

```
const routes : [{redirect: string, path: stri...  = [
  {
    path: '/',
    name: 'Manager',
    component: () : Promise<{readonly default?: {...}}>  => import('../views/Manager.vue'),
    redirect: '/home',
    children: [
      { path: '403', name: 'NoAuth', meta: { name: 'NoAuth' }, component: () : Promise<{...}>  => import('../views/manager/403') },
      { path: 'home', name: 'Home', meta: { name: 'System' }, component: () : Promise<{...}>  => import('../views/manager/Home') },
      { path: 'admin', name: 'Admin', meta: { name: 'Admin' }, component: () : Promise<{...}>  => import('../views/manager/Admin') },
      { path: 'adminPerson', name: 'AdminPerson', meta: { name: 'Personal information' }, component: () : Promise<{...}>  => import('../views/manager/AdminPerson') },
      { path: 'password', name: 'Password', meta: { name: 'Change password' }, component: () : Promise<{...}>  => import('../views/manager/Password') },
      { path: 'notice', name: 'Notice', meta: { name: 'Notice' }, component: () : Promise<{...}>  => import('../views/manager/Notice') },
      { path: 'user', name: 'user', meta: { name: 'user' }, component: () : Promise<...>  => import('../views/manager/User') },
      { path: 'UserPerson', name: 'UserPerson', meta: { name: 'UserPerson' }, component: () : Promise<{...}>  => import('../views/manager/UserPerson') },
      { path: 'ac', name: 'Ac', meta: { name: 'Cuenta' }, component: () : Promise<{...}>  => import('../views/manager/Ac') },
      { path: 'category', name: 'category', meta: { name: 'Categorias' }, component: () : Promise<{...}>  => import('../views/manager/Category') },
      { path: 'bill', name: 'bill', meta: { name: 'Factura' }, component: () : Promise<{...}>  => import('../views/manager/Bill') },
      { path: 'notebook', name: 'notebook', meta: { name: 'Notebook' }, component: () : Promise<{...}>  => import('../views/manager/Notebook') },
      { path: 'plan', name: 'plan', meta: { name: 'Plan' }, component: () : Promise<{...}>  => import('../views/manager/Plan') },
      { path: 'planDetail', name: 'planDetail', meta: { name: 'PlanDetail' }, component: () : Promise<{...}>  => import('../views/manager/PlanDetail') },
    ]
  },

  { path: '/login', name: 'Login', meta: { name: 'Login' }, component: () : Promise<{...}>  => import('../views/Login.vue') },
  { path: '/register', name: 'Register', meta: { name: 'Register' }, component: () : Promise<{...}>  => import('../views/Register.vue') },
  { path: '*', name: 'NotFound', meta: { name: '404' }, component: () : Promise<{...}>  => import('../views/404.vue') },
]

const router : VueRouter  = new VueRouter( options: {
  mode: 'history',
  base: process.env.BASE_URL,
  routes
})


export default router  Show usages
```

Ilustración 45: index.js

Finalmente, utilizamos componentes de ElementUI para construir el formulario de las diferentes pantallas.

5.3.1 Inicio de sesión

El componente de inicio de sesión (Login.vue) se encarga de autenticar al usuario. Incluye validación de los datos de entrada y comunicación con el servidor.

```vue
<script>
export default {  Show usages
  name: "Login",
  data() {
    return {
      form: { role: 'ADMIN' },
      rules: {
        username: [
          { required: true, message: 'User', trigger: 'blur' },
        ],
        password: [
          { required: true, message: 'password', trigger: 'blur' },
        ]
      }
    }
  },
  created() {

  },
  methods: {
    login() {
      this.$refs['formRef'].validate((valid) => {
        if (valid) {
          // Verificación aprobada
          this.$request.post( url: '/login', this.form).then(res => {
            if (res.code === '200') {
              localStorage.setItem("xm-user", JSON.stringify(res.data))  //Almacenar datos de usuario
              this.$router.push('/')  //Saltar a la página de inicio
              this.$message.success( text: 'Login Successful')
            } else {
              this.$message.error(res.msg)
            }
          })
        }
      })
    }
  }
}
</script>
```

Ilustración 46: Inicio de sesión

5.3.1.1 Inicio

En la pantalla de inicio se encuentran los componentes más importantes de la aplicación, como el menú lateral y las estadísticas de las transacciones.

```
mounted() {
  if (this.user.role === 'USER') {

    let pieDom = document.getElementById( elementId: 'pie');
    let pieChart = echarts.init(pieDom);


    let pieDom1 = document.getElementById( elementId: 'pie1');
    let pieChart1 = echarts.init(pieDom1);


    this.$request.get( url: '/bill/count', config: { params: {type: 'Gastos'} }).then(res => {
      this.payList = res.data || []
      let arr = []
      this.payList.forEach(item => {
        arr.push({ name: item.category, value: item.money })
      })
      pieOption.series[0].data = arr
      pieChart.setOption(pieOption)
    })

    this.$request.get( url: '/bill/count', config: { params: {type: 'Ingresos'} }).then(res => {
      this.incomeList = res.data || []

      let arr = []
      this.incomeList.forEach(item => {
        arr.push({ name: item.category, value: item.money })
      })
      pieOption1.series[0].data = arr
      pieChart1.setOption(pieOption1)
    })
  }
```

Ilustración 47: Pantalla de inicio

5.3.2 Perfil

La pantalla de perfil es igual tanto para los usuarios como para los administradores. Cuenta con información básica del usuario.

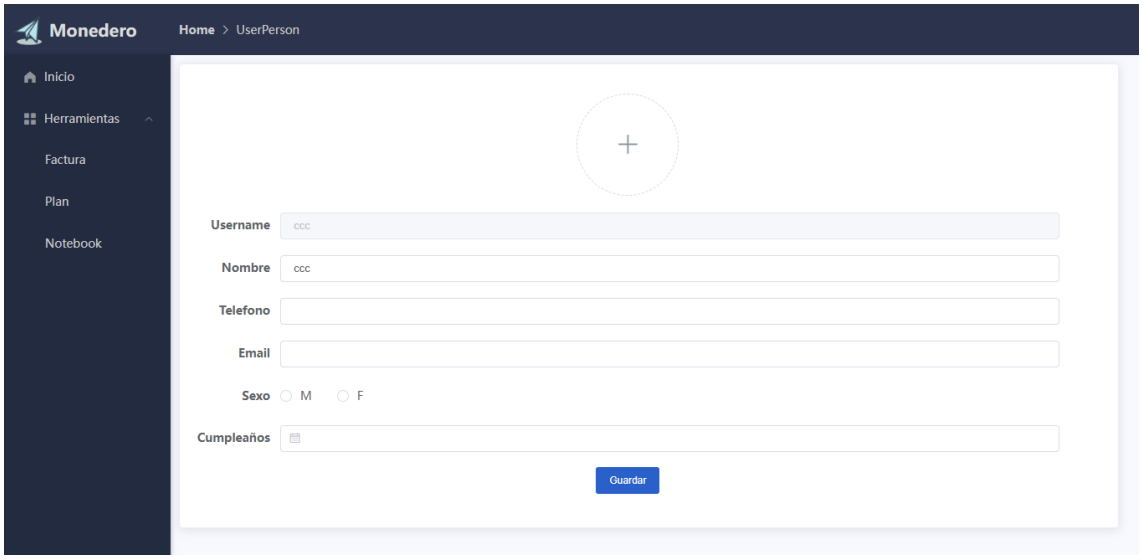


Ilustración 48: Pantalla de perfil

5.3.3 Ocultar página

Para ocultar las ventanas de gestión de administrador, se hace un filtro para los usuarios.

```
        <i class="el-icon-menu"></i><span>Herramientas</span>
      </template>
      <el-menu-item index="/ac" v-if=" user.role === 'ADMIN'">Cuenta</el-menu-item>
      <el-menu-item index="/bill">Factura</el-menu-item>
      <el-menu-item index="/plan">Plan</el-menu-item>
      <el-menu-item index="/notebook">Notebook</el-menu-item>
      <el-menu-item index="/category" v-if=" user.role === 'ADMIN'">Categorias</el-menu-item>
      <el-menu-item index="/notice" v-if=" user.role === 'ADMIN'" >Notice</el-menu-item>
    </el-submenu>

    <el-submenu index="user" v-if=" user.role == 'ADMIN'">
      <template slot="title">
        <i class="el-icon-menu" ></i><span>Gestión de usuarios</span>
      </template>
      <el-menu-item index="/admin">Admin</el-menu-item>
      <el-menu-item index="/user">User</el-menu-item>
    </el-submenu>
  </el-menu>
</div>
```

Ilustración 49: Filtro de usuario

5.3.4 Casos de error

Para tratar las páginas no encontradas en la aplicación se incluye el componente 404.vue. Este componente avisa a los usuarios cuando una URL no corresponde a ninguna ruta definida.

```
<template>  Show component usages
  <div>
    <div style="...">
      <div style="...">404 Page not found <router-link to="/">Return</router-link></div>
    </div>

  </div>
</template>

<script>
export default {  Show usages
  name: "404",
  data() {
    return {}
  },
  created() {

  },
  methods: {}
}
</script>

<style scoped>

</style>
```

Ilustración 50: Error 404

6 Resultados y conclusiones

Este proyecto ha sido más que un simple proyecto académico sino un viaje de aprendizaje profundo y transformador. Este proyecto no solo ha cumplido con los objetivos establecidos, sino que también ha superado mis expectativas en muchos aspectos

Desde el inicio del proyecto, me he enfrentado a numerosos desafíos que han requerido un esfuerzo considerable y una dedicación constante. Encontré desafíos técnicos, como la implementación de la autenticación segura y la gestión eficiente de bases de datos. Sin embargo, estos obstáculos se convirtieron en oportunidades de aprendizaje. Cada problema resuelto incrementó mi comprensión de las tecnologías utilizadas y fortaleció mis habilidades en desarrollo de software. La elección de estas dos tecnologías Spring Boot y Vue.js fue un reto emocionante. Aprender y dominar estas herramientas permitió construir esta plataforma.

Personalmente, el impacto del proyecto ha sido profundo. He aprendido a gestionar un proyecto complejo de principio a fin, desarrollando habilidades técnicas y de gestión de proyectos que serán invaluables en mi carrera futura. También he experimentado de primera mano la importancia de diseñar con el usuario en mente, asegurando que las soluciones tecnológicas sean accesibles y útiles para todos.

En resumen, la creación de esta plataforma ha sido un viaje de aprendizaje lleno de desafíos y satisfacciones. Cada obstáculo superado me ha permitido crecer tanto profesional como personalmente, reforzando mis habilidades técnicas y mi capacidad para resolver problemas complejos.

6.1 Resultado de la plataforma

En este apartado se incluirá capturas del producto final. Con el fin de presentar las funcionalidades más importantes de la plataforma.

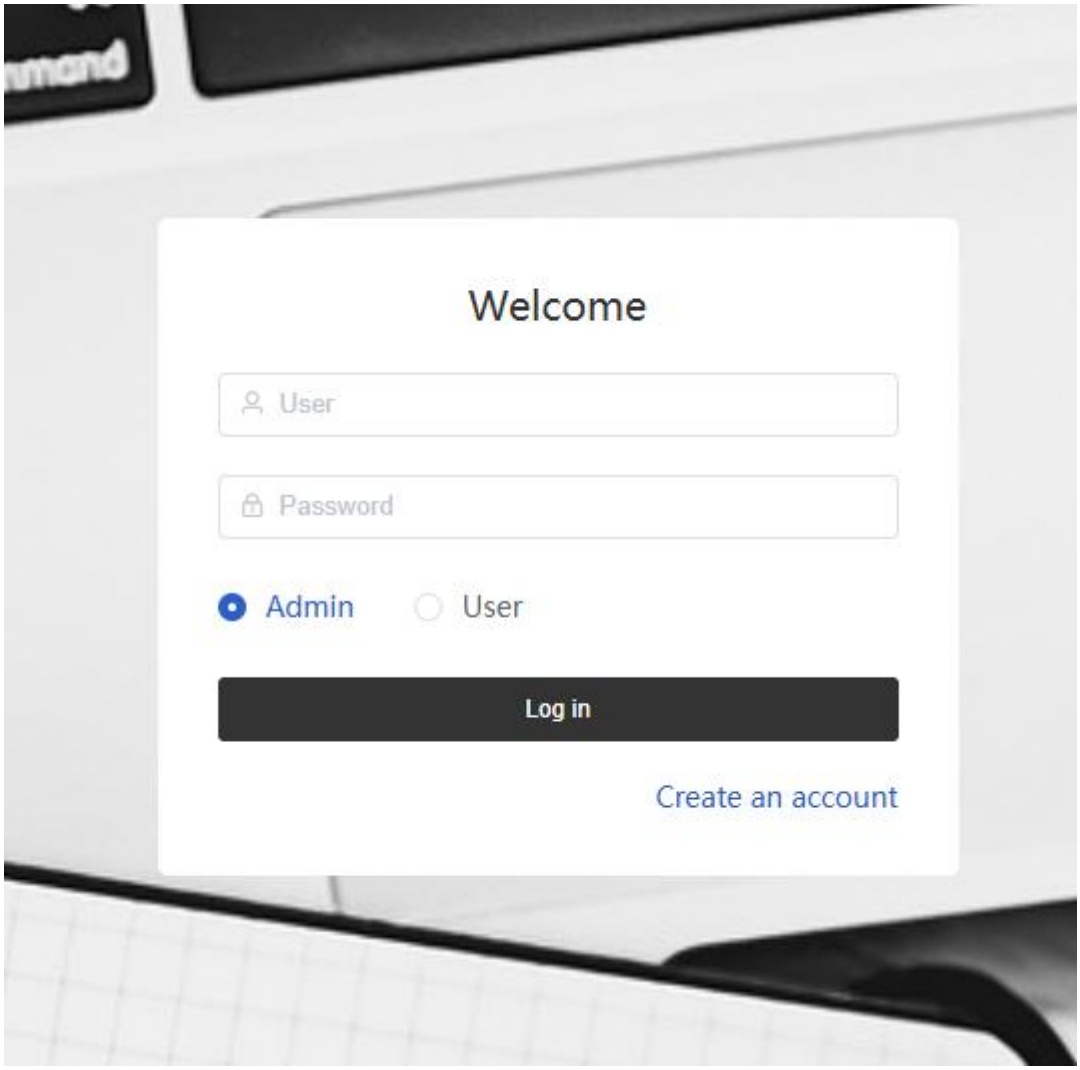


Ilustración 51: Inicio de sesión

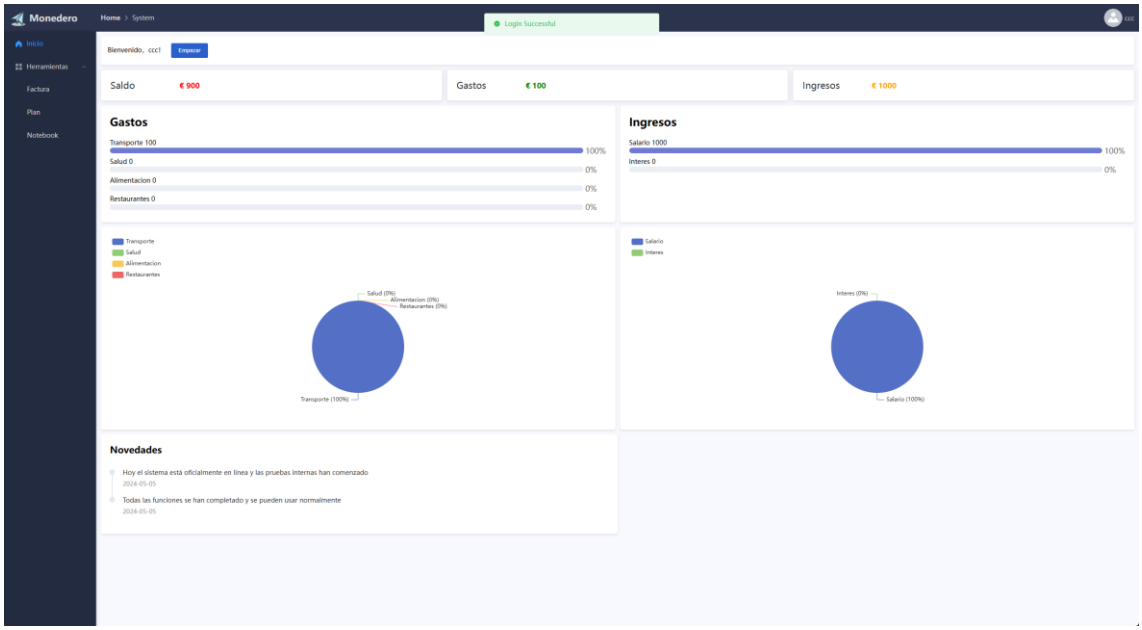


Ilustración 52: Pagina principal

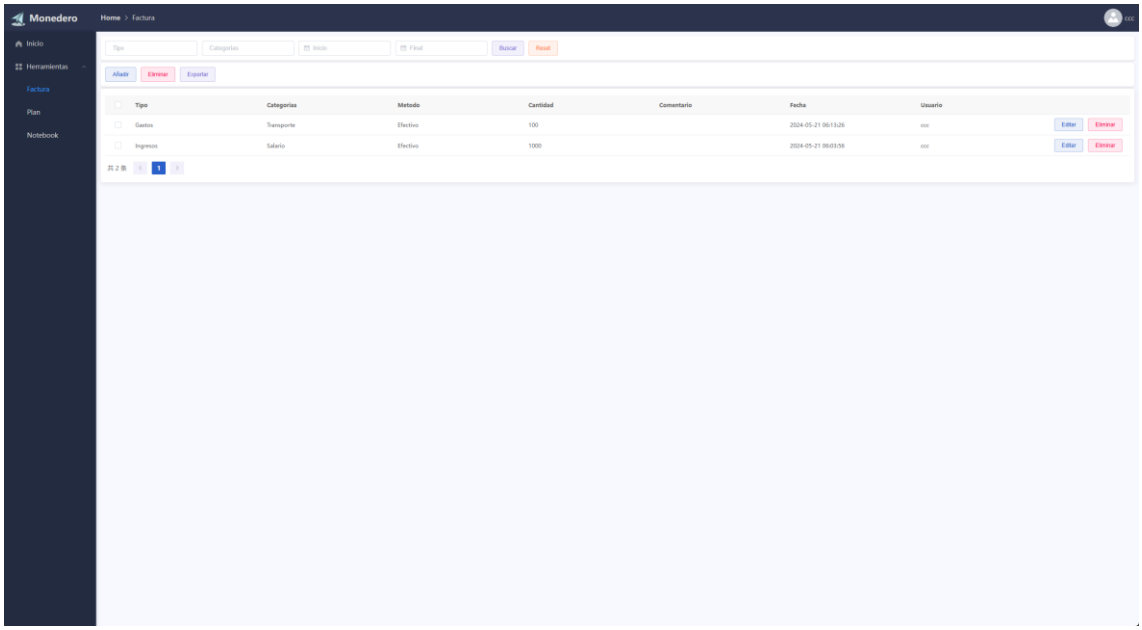


Ilustración 53: Facturas

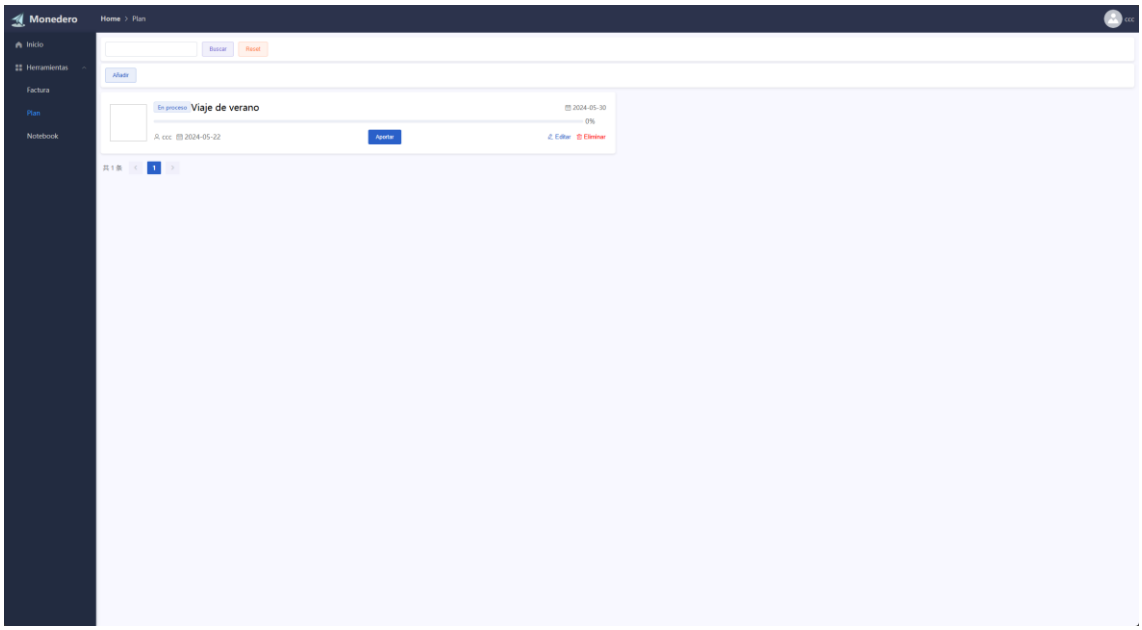


Ilustración 54: Planes

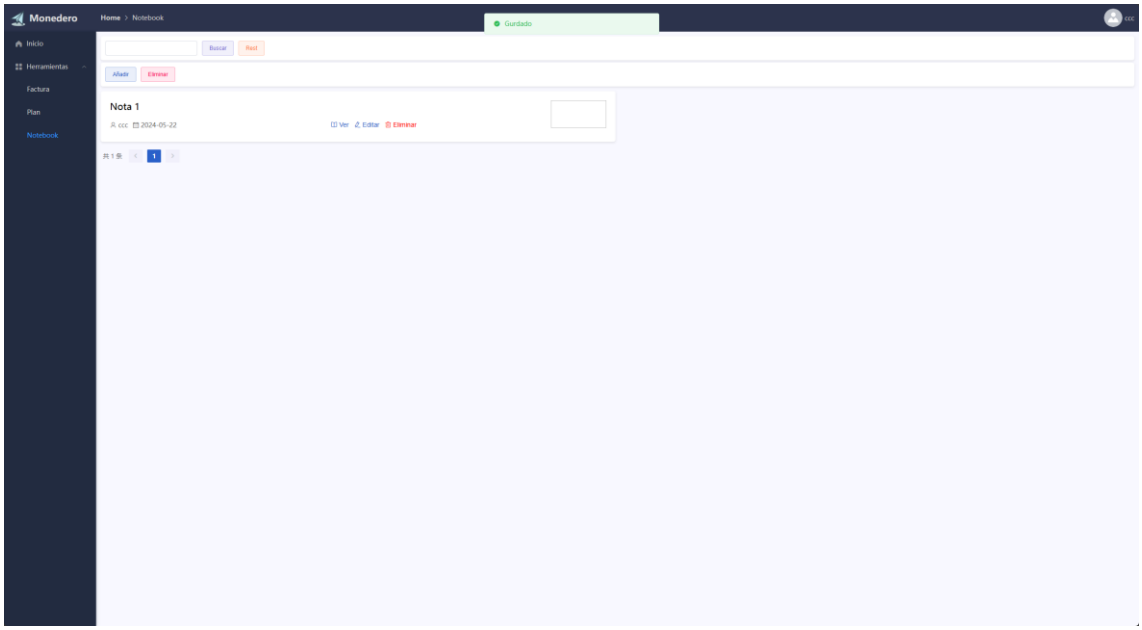


Ilustración 55: Bloc de nota

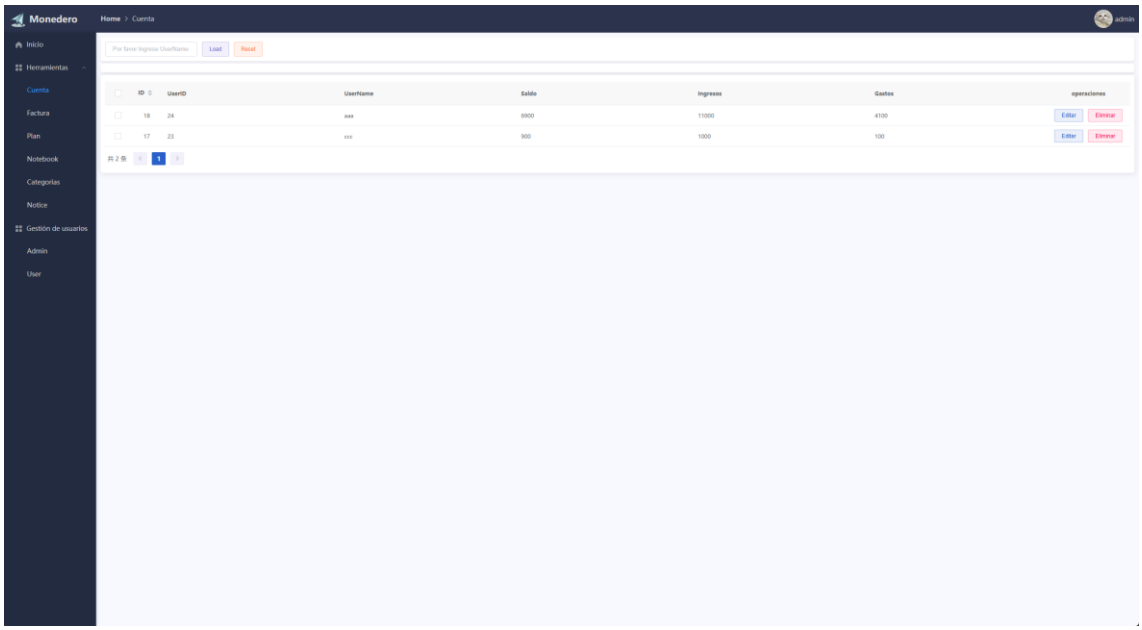


Ilustración 56: Gestión de cuentas

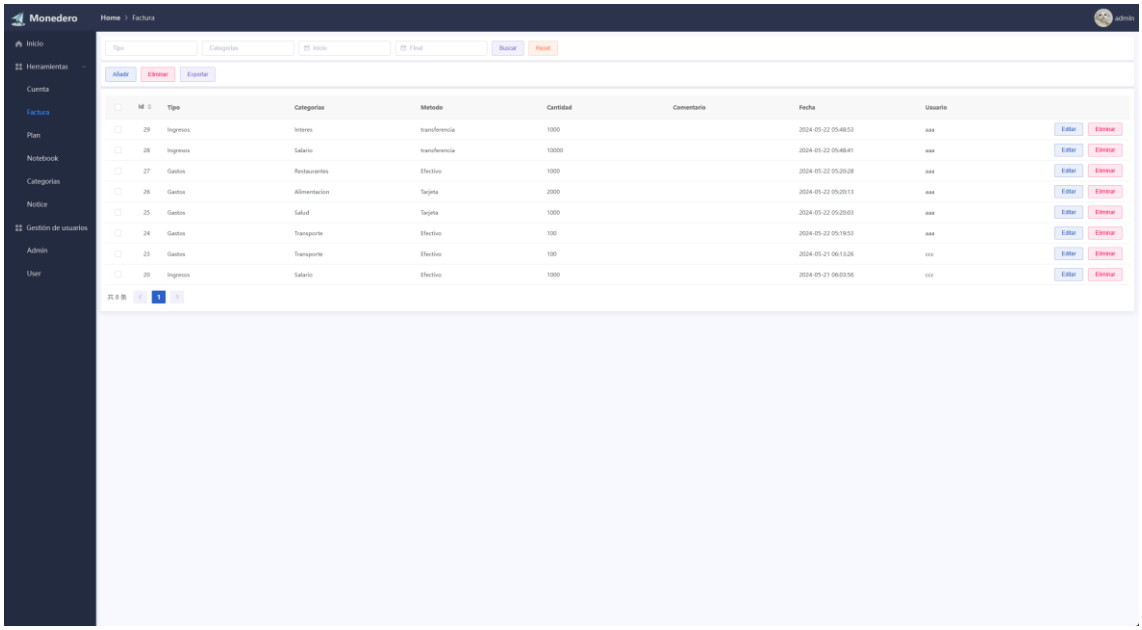


Ilustración 57: Gestión de transacciones

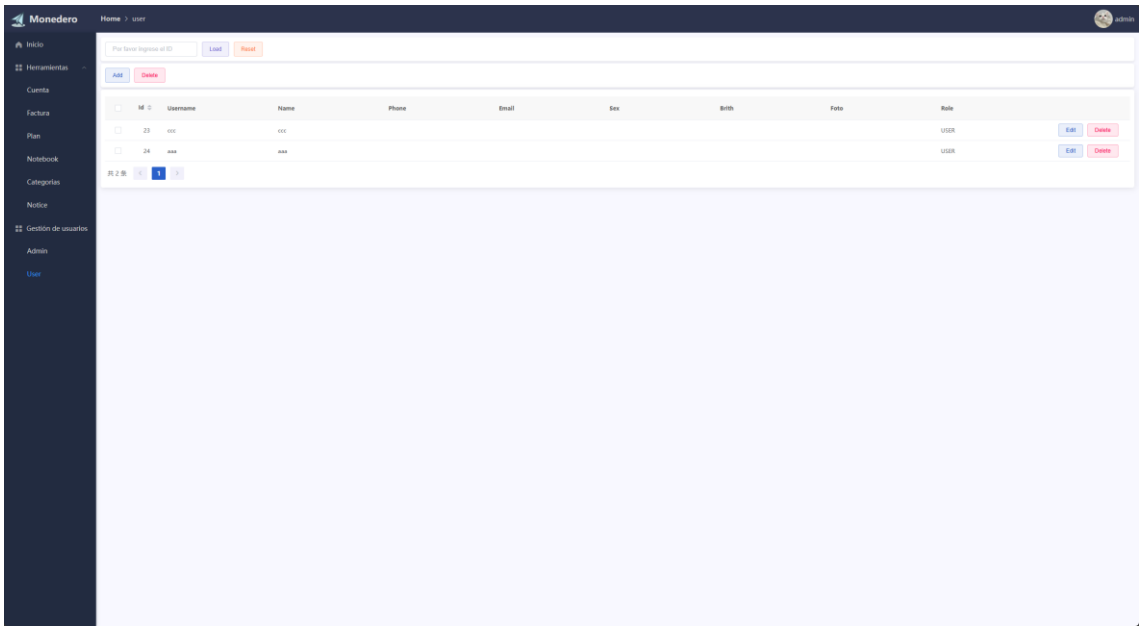


Ilustración 58: Gestión de usuarios

7 Análisis de Impacto

El desarrollo e implementación de la plataforma propuesta en este Trabajo Fin de Grado genera impactos en diversos contextos. A continuación, se detallan los impactos esperados en los ámbitos personal, empresarial, social, económico, medioambiental y cultural, así como su alineación con los Objetivos de Desarrollo Sostenible (ODS) de la Agenda 2030.

Impacto Personal: A nivel personal, la plataforma permitirá a los usuarios gestionar sus finanzas de manera más eficiente y efectiva. La capacidad de controlar ingresos, gastos, ahorros e inversiones contribuirá a una mayor estabilidad económica individual. Esta herramienta fomentará una mayor disciplina financiera y una mejor planificación a corto y largo plazo.

Impacto Empresarial: En el ámbito empresarial, especialmente para pequeñas empresas y emprendedores, la plataforma ofrece una herramienta valiosa para la gestión financiera. La posibilidad de registrar y monitorizar transacciones, gestionar presupuestos y planificar financieramente puede optimizar la administración de recursos y mejorar la toma de decisiones estratégicas.

Impacto Social: Socialmente, la plataforma tiene el potencial de aumentar la educación financiera y la alfabetización digital entre la población. Se espera que más personas puedan aprender y adoptar mejores prácticas financieras.

Impacto Económico: Económicamente, la adopción generalizada de la plataforma puede contribuir a una mayor estabilidad financiera a nivel macroeconómico.

Impacto Medioambiental: Aunque el impacto medioambiental directo de la plataforma es limitado, la digitalización de la gestión financiera puede contribuir a la reducción del uso de papel y otros recursos físicos. La plataforma también puede integrar y promover prácticas sostenibles entre sus usuarios, incentivando comportamientos responsables con el medio ambiente.

Impacto Cultural: Culturalmente, la adopción de tecnologías avanzadas para la gestión financiera puede fomentar una cultura de innovación y adaptación tecnológica.

Conforme con los Objetivos de Desarrollo Sostenible (ODS) establecidos. La plataforma se alinea con varios ODS de la Agenda 2030 de la ONU, incluyendo:

ODS 1: Fin de la Pobreza, proporciona herramientas que ayudan a las personas a gestionar mejor sus recursos financieros y evitar situaciones de endeudamiento.

ODS 8: Trabajo Decente y Crecimiento Económico, Apoya a pequeñas empresas y emprendedores en la gestión eficiente de sus finanzas, promoviendo un crecimiento económico sostenido y sostenible.

ODS 9: Industria, Innovación e Infraestructura, fomentar la adopción de tecnologías avanzadas y la innovación en la gestión financiera.

ODS 12: Producción y Consumo Responsables, contribuir a la reducción del uso de papel y otros recursos físicos mediante la digitalización de procesos financieros.

8 Bibliografía

[1] «obs business school,» [En línea]. Available: https://www.obsbusiness.school/blog/educacion-financiera-que-es-y-como-gestionar-tus-finanzas.

[2] «Mindful Budgets,» [En línea]. Available: https://mindfulbudgets.com/mint-vs-ynab/.

[3] «Goodbudget,» [En línea]. Available: https://goodbudget.com/blog/2022/12/goodbudget-vs-everydollar-which-budget-app-is-for-you/.

[4] «Softonic,» [En línea]. Available: https://everydollar.softonic.com/android.

[5] «Google play,» [En línea]. Available: https://play.google.com/store/apps/details?id=com.dayspringtech.envelopes&hl=es_419.

[6] «Quicken,» [En línea]. Available: https://www.quicken.com/.

[7] «Wealthfront,» [En línea]. Available: https://www.wealthfront.com/.

[8] «HomeBank,» [En línea]. Available: https://www.gethomebank.org/en/index.php.

[9] wikipedia, «wikipedia,» [En línea]. Available: https://es.wikipedia.org/wiki/GnuCash.

[10] «ventajas y desventajas,» [En línea]. Available: https://ventajasydesventajastop.com/ventajas-y-desventajas-de-intellij-idea/.

[11] «Navicat,» [En línea]. Available: https://www.navicat.com/en/products/navicat-for-mysql.

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Fri May 31 17:16:40 CEST 2024
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)