**Software de Control de Gastos:**

**Primer Iteración:**

Requerimientos:

Funcionales:

1. El sistema debe permitir registrar ingresos y egresos.
2. El sistema debe asignar los gastos de forma automática por categorías.
3. El sistema debe ordenar las transacciones por fecha de realización.
4. El sistema debe llevar una cuenta de los gastos realizados en cada mes.
5. El usuario debe poder filtrar las transacciones por fecha o categoría.
6. El sistema debe crear y mostrar gráficos estadísticos acerca de las transacciones realizadas.
7. El usuario debe poder crearse una cuenta y poder autenticarse.
8. Cada usuario solo debe poder acceder a sus transacciones.
9. El usuario puede crear o editar categorías.

No Funcionales:

*Cumple con*:

1. El sistema de control de gastos debe proporcionar una interfaz de usuario estéticamente agradable que permita una interacción satisfactoria para el usuario.

La aplicación implementa un diseño visual moderno con gradientes, esquemas de colores consistentes, iconos intuitivos y elementos interactivos que facilitan la navegación. La interfaz incluye cards con sombras, bordes redondeados, transiciones suaves y estados hover que mejoran la experiencia visual. Además, proporciona feedback visual mediante notificaciones de éxito y error, garantizando que los usuarios puedan interactuar de manera agradable y eficiente con todas las funcionalidades del sistema.

1. El sistema debe garantizar la autenticidad de los usuarios mediante un mecanismo robusto de verificación de identidad.

La aplicación implementa autenticación basada en tokens JWT que valida la identidad de cada usuario antes de permitir el acceso a funcionalidades protegidas. El sistema incluye un middleware de autenticación obligatorio para todas las rutas sensibles, encriptación de contraseñas usando bcrypt con salt rounds, y separación estricta de datos donde cada usuario solo puede acceder a sus propias transacciones y categorías. Los tokens tienen tiempo de expiración configurado y se validan en cada solicitud, asegurando que solo usuarios autenticados y autorizados puedan realizar operaciones en el sistema.

*Se deberían implementar*:

1. El sistema debería implementar validaciones comprehensivas del lado cliente y servidor

Incluyendo confirmaciones para acciones destructivas, capacidad

de deshacer operaciones, y una función de "papelera" para recuperar transacciones eliminadas accidentalmente.

Se debería agregar validación en tiempo real, confirmaciones modales elegantes, historial de operaciones, y soft-delete en lugar de eliminación permanente.

1. El sistema debería ser completamente responsive y optimizado para dispositivos móviles

El sistema deberá incluir navegación táctil, menús colapsables, y layouts específicos para pantallas pequeñas.

Incluyendo l navegación tipo “hamburger” para optimizar el uso del espacio en pantalla, el uso de cards apilables que permitan presentar la información de manera ordenada y legible en disposición vertical, la habilitación de gestos de desplazamiento (swipe) para realizar acciones rápidas sin necesidad de acceder a menús adicionales, y la reorganización de los elementos bajo un enfoque mobile-first, priorizando la usabilidad en pantallas pequeñas y adaptando posteriormente la interfaz a resoluciones mayores.

UML:

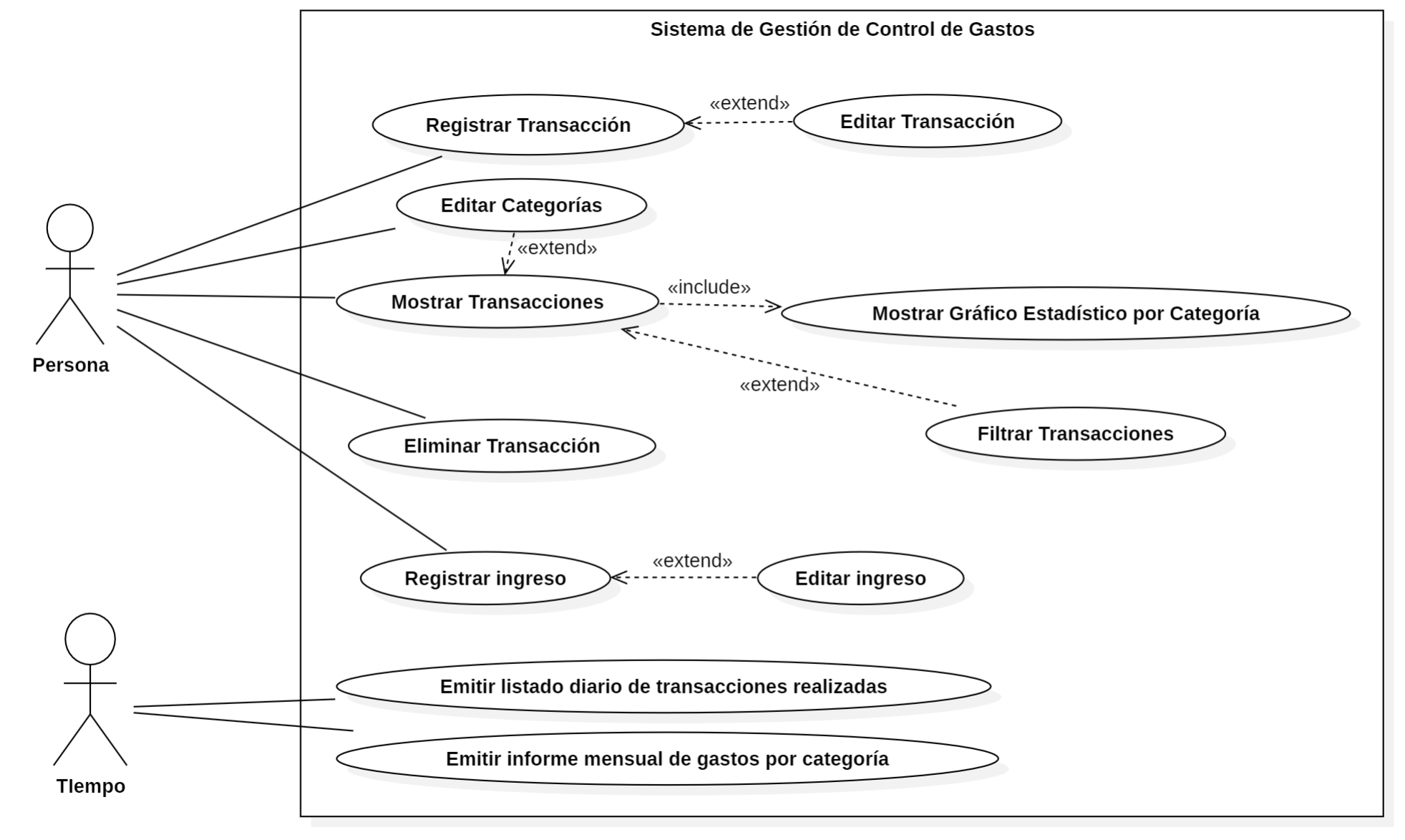
Como parte del análisis funcional del sistema, se utilizó el lenguaje de modelado **UML (Unified Modeling Language)**, en particular el **diagrama de casos de uso**, con el fin de representar de forma clara las funcionalidades ofrecidas por el sistema y las interacciones principales que tendrá el usuario.

Este tipo de diagrama permite describir de manera sencilla y visual **qué puede hacer un usuario dentro del sistema**, sin necesidad de entrar en detalles técnicos sobre cómo se implementan esas funcionalidades. Es especialmente útil en etapas tempranas del diseño, ya que ayuda a delimitar el alcance funcional del software, en este caso, el control de gastos.

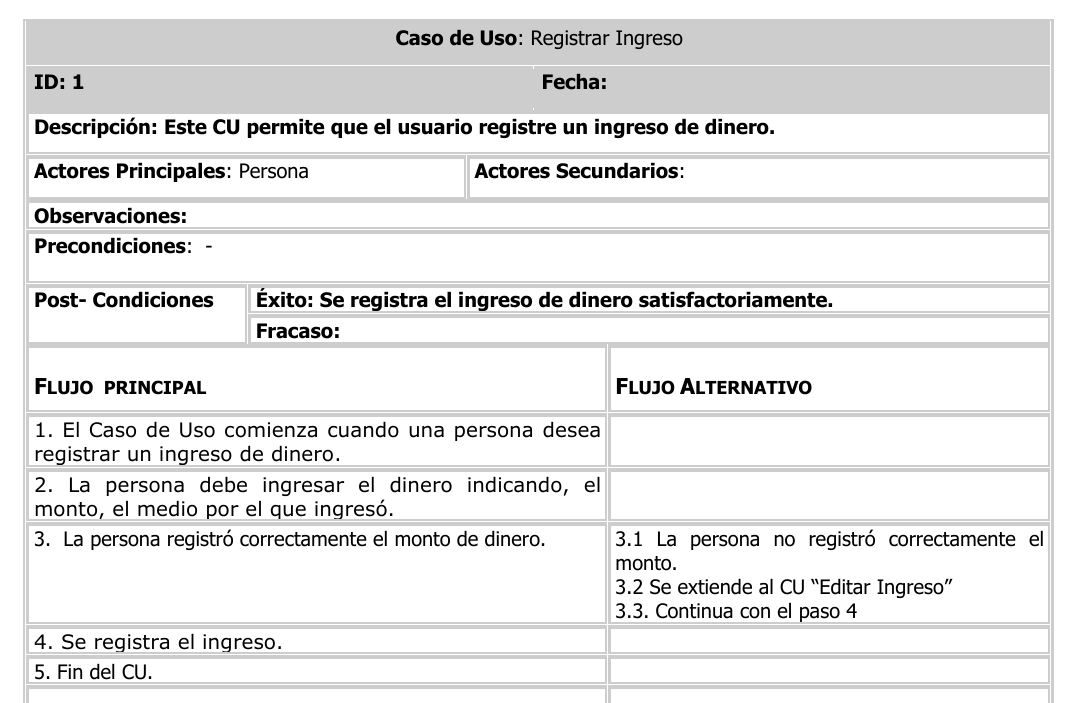
Casos de Uso:

* Mostrar transacciones.
  + Registrar Usuario (extend)
* Registrar transacción.
* Filtrar transacciones.
* Mostrar gráficos estadísticos.
* Crear/Editar categoría.
* Emitir listado de transacciones.
* Eliminar transacción.

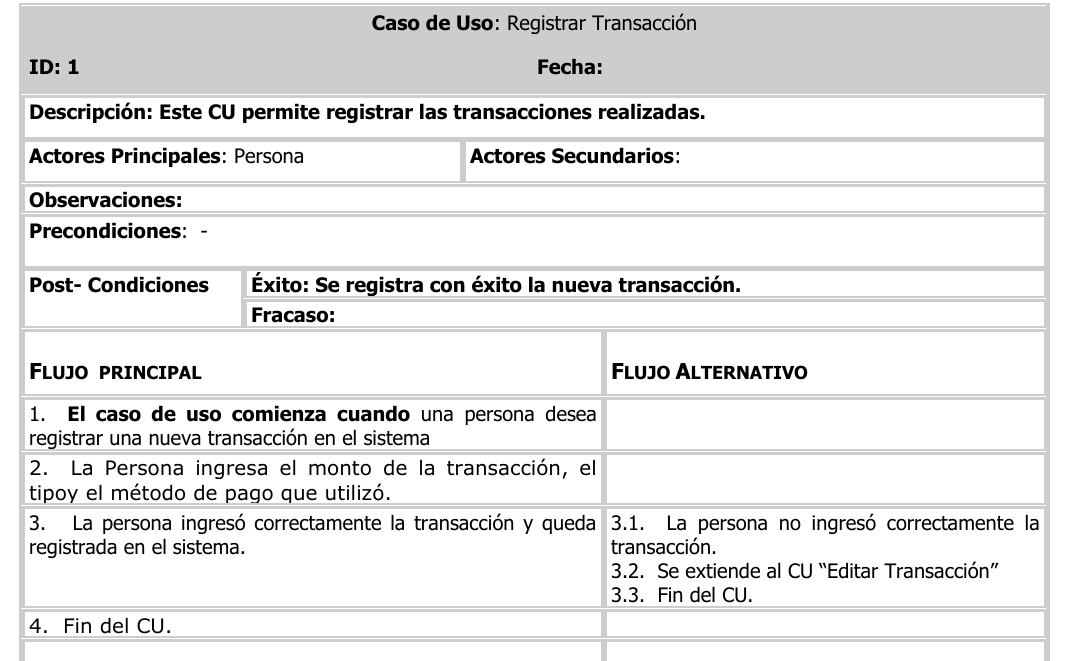
Diagrama de Casos de Uso:



Descripción de Casos de Uso:  
*Registrar Ingreso:*

**

*Registrar Transacción*

**

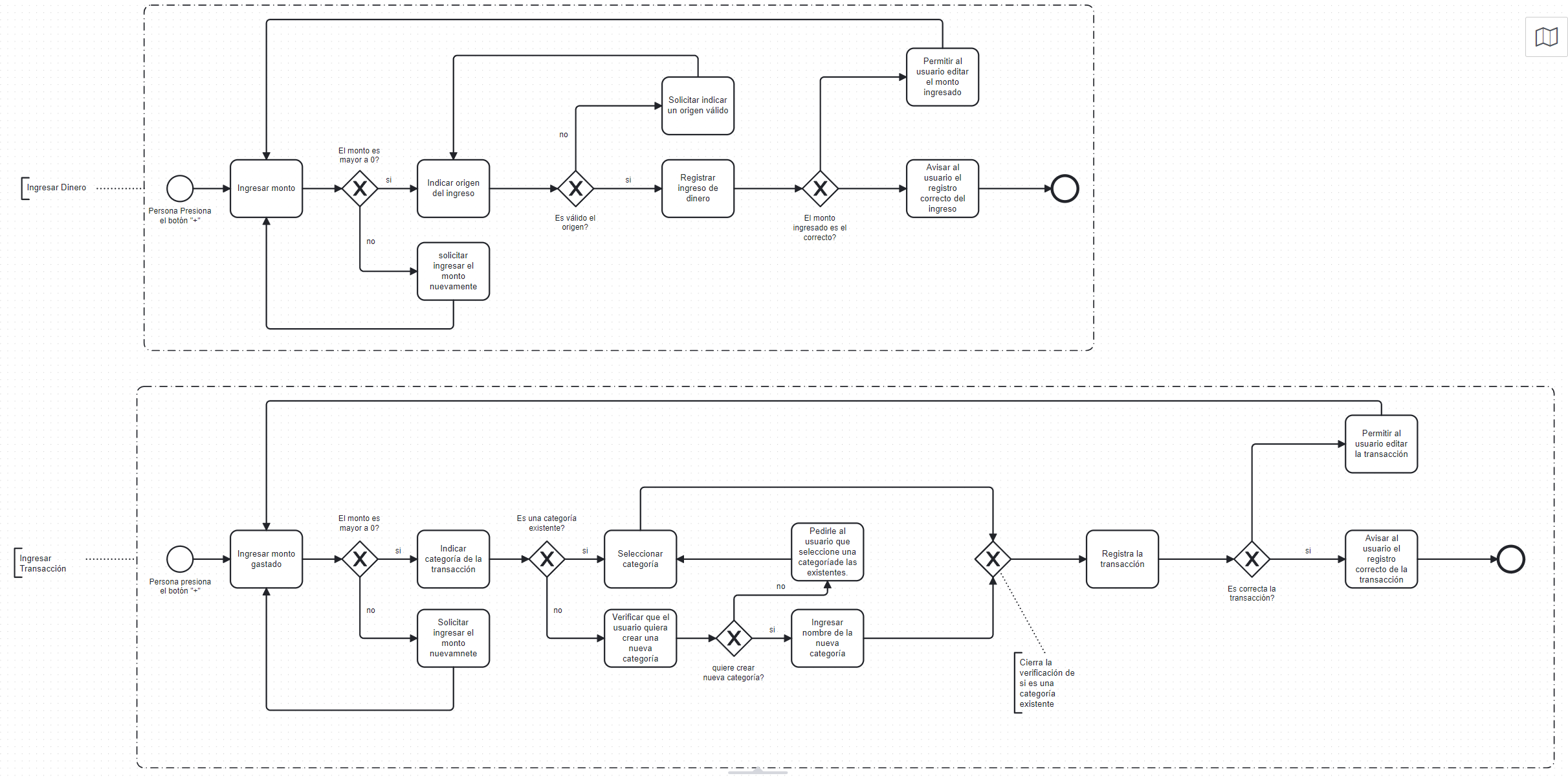
BPMN:

Para modelar los procesos del sistema propuesto, se utilizó el estándar **BPMN (Business Process Model and Notation)**. BPMN es una notación gráfica utilizada para representar de forma clara y estructurada los procesos de negocio. Su objetivo principal es facilitar la comprensión de los flujos de trabajo tanto para perfiles técnicos como no técnicos, lo cual resulta especialmente útil en proyectos de desarrollo de software.

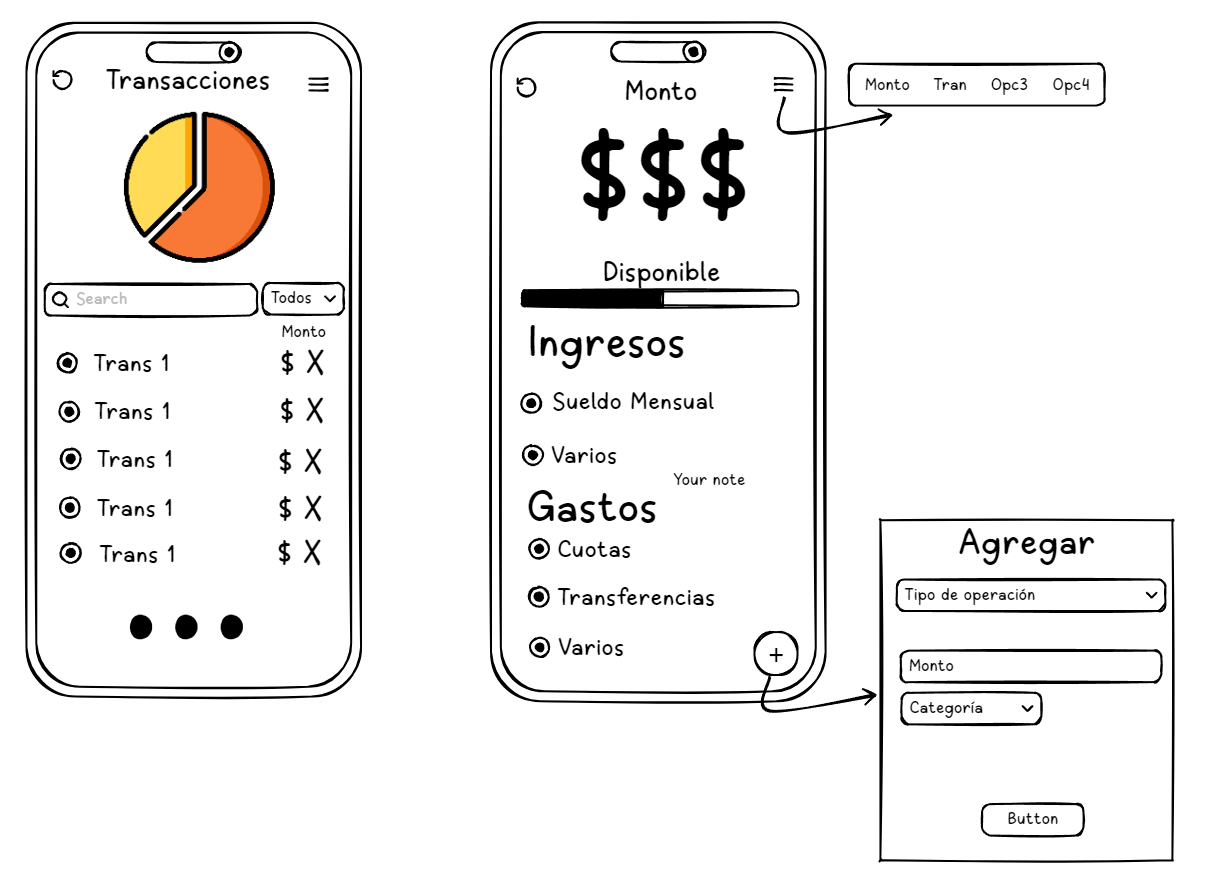
En esta primer iteración de la aplicación de gestión de gastos, se eligió BPMN para modelar dos procesos clave: **Ingreso de Dinero** y **Registro de Transacción**. Estos procesos representan interacciones fundamentales entre el usuario y el sistema, y su correcta definición es esencial para garantizar el correcto funcionamiento de la aplicación.

En este caso, en el proceso de *Ingreso de Dinero*, el modelo BPMN permite representar el flujo que comienza con la acción del usuario al seleccionar la opción de agregar una operación, continúa con la introducción del monto y la categoría, y concluye con la validación y almacenamiento de los datos por parte del sistema. De forma similar, el proceso de *Registro de Transacción* incluye la interacción del usuario con el listado de movimientos, la aplicación de filtros y la visualización de detalles, todo lo cual puede describirse de forma visual mediante BPMN.

En resumen, BPMN se utiliza en este proyecto como una herramienta para representar, analizar y comunicar de manera efectiva los procesos esenciales del sistema, ayudando a alinear el diseño funcional con su posterior implementación técnica.



Prototipado:



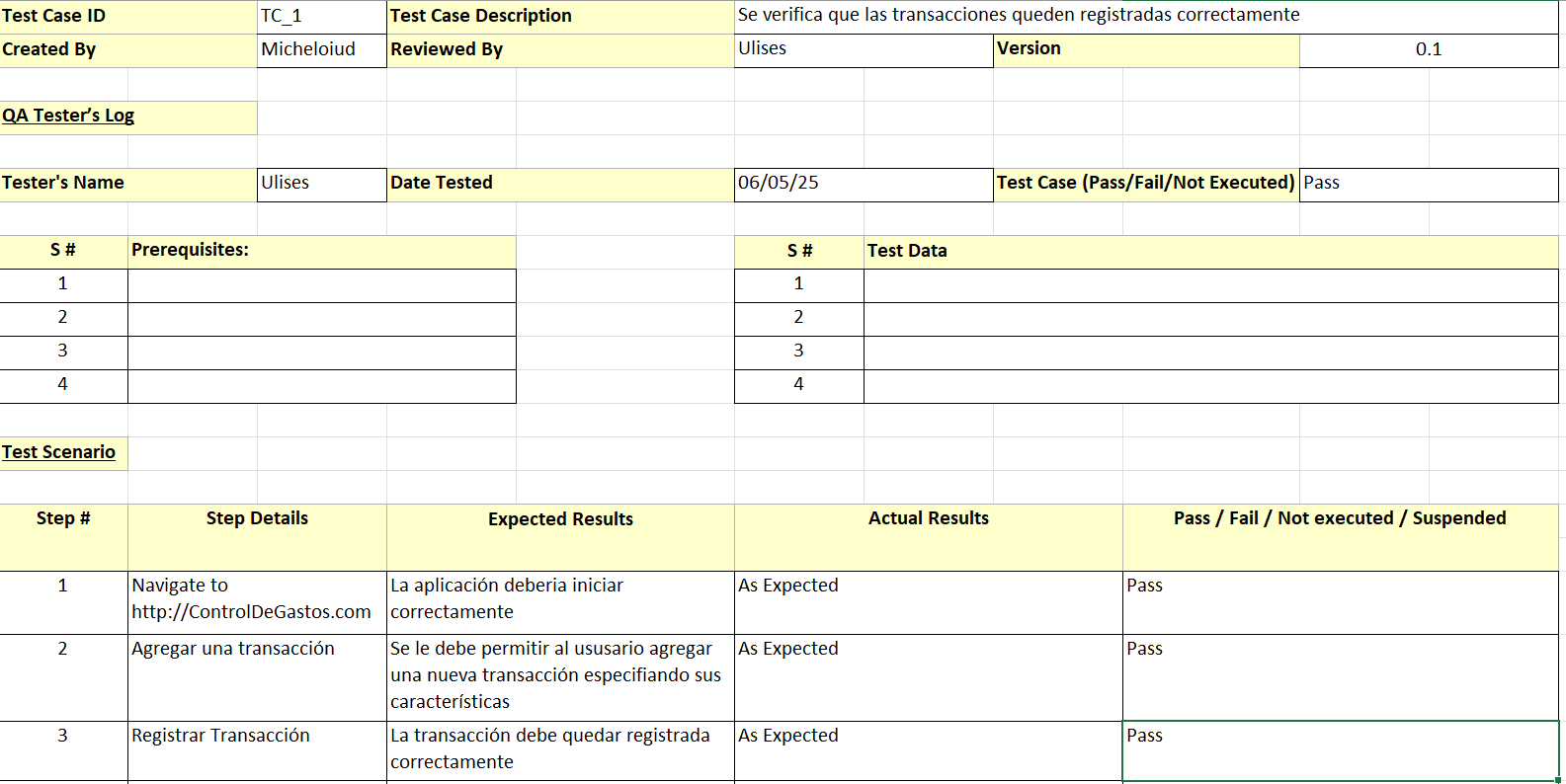
En este primer prototipo del software se pueden observar dos pestañas principales, una pestaña donde se pueden ver las transacciones realizadas por el usuario, indicando de cada una de ellas a quién se realizó, el monto, la categoría y la fecha, además de incluir un gráfico que permite visualizar las de forma clara en qué categorías se realizaron la mayoría de las transacciones.

Por otro lado, está la pestaña donde se muestra el monto que se tiene en total, y una barra por debajo que indica cuánto dinero se utilizó del total que se tiene, además se indican y muestran todos los ingresos y su origen y los gastos, teniendo en cuenta si son gastos permanentes cada cierto periodo de tiempo o si son gastos que se hicieron una sola vez, entre otros gastos varios; esta sección incluye una opción, dada por el botón con el “+”, para poder agregar un nuevo ingreso o una nuevo gasto, mostrando una pequeña pestaña que se superpone al contenido que permite seleccionar el tipo de operación, y con ello el monto y la categoría.

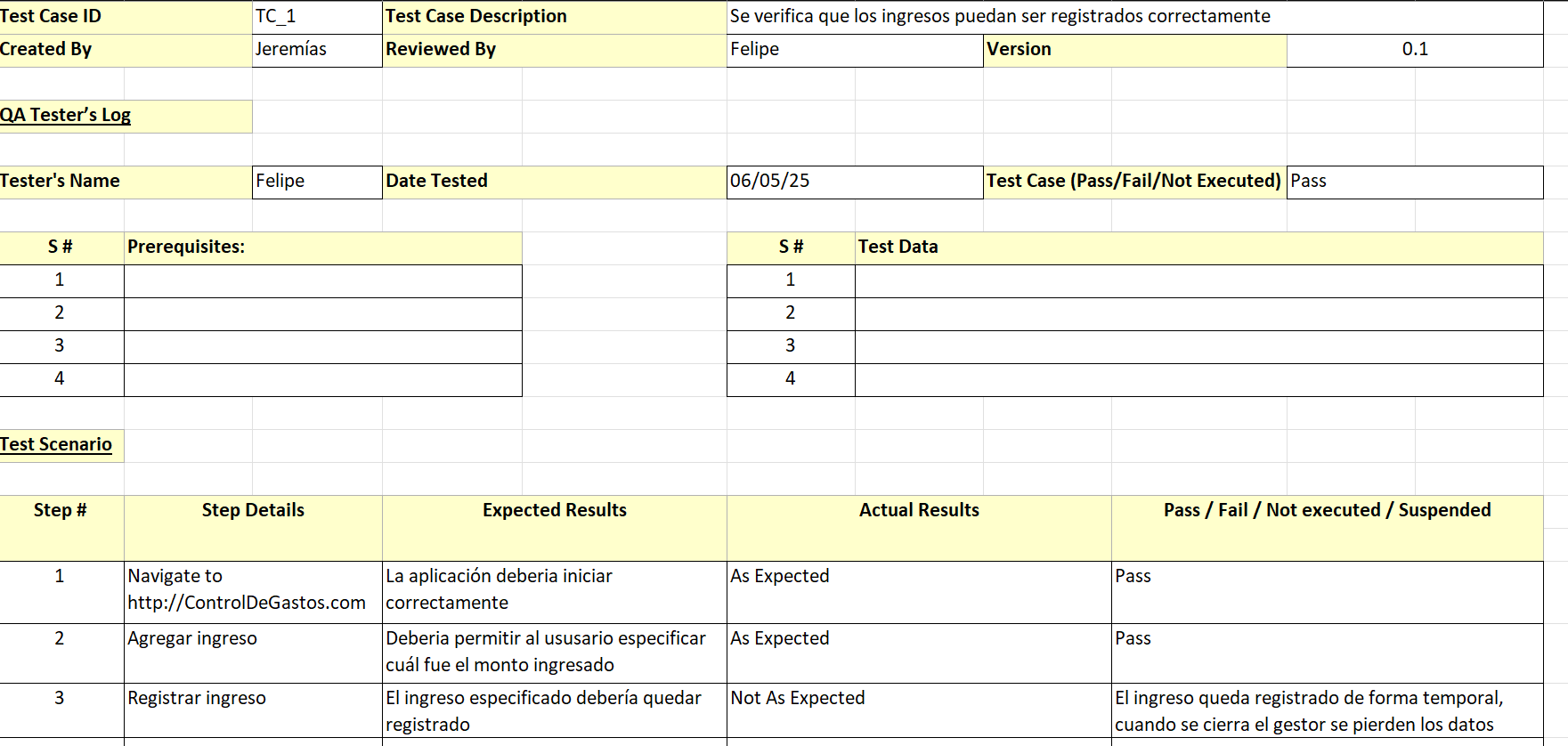
Además, se agregó en la parte superior la opción de recargar la página en caso de que sea necesario, y una barra de navegación que se despliega cuando se hace click en las tres barras de la parte superior, que permite moverse a través de las distintas secciones del gestor.

Testing:

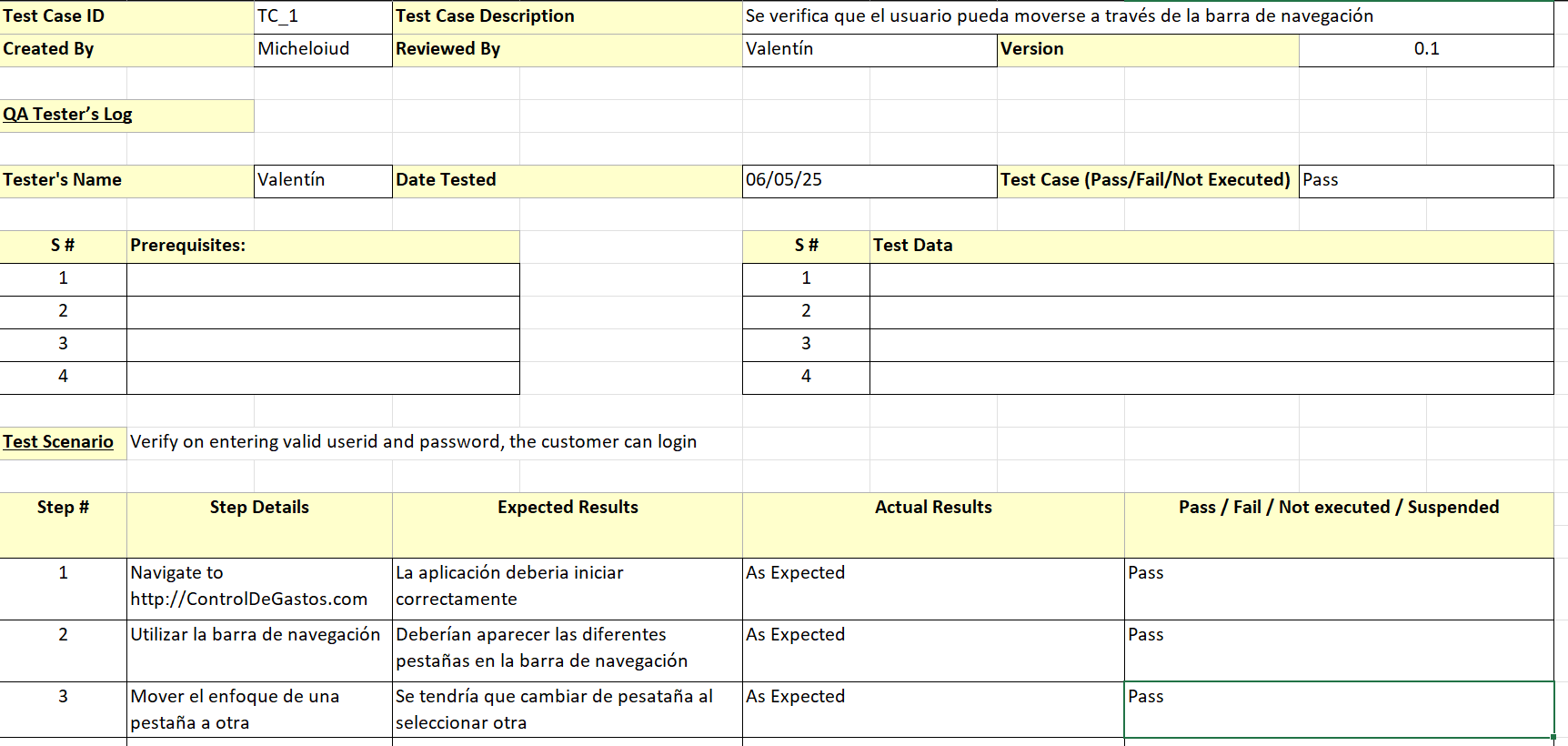
#1:

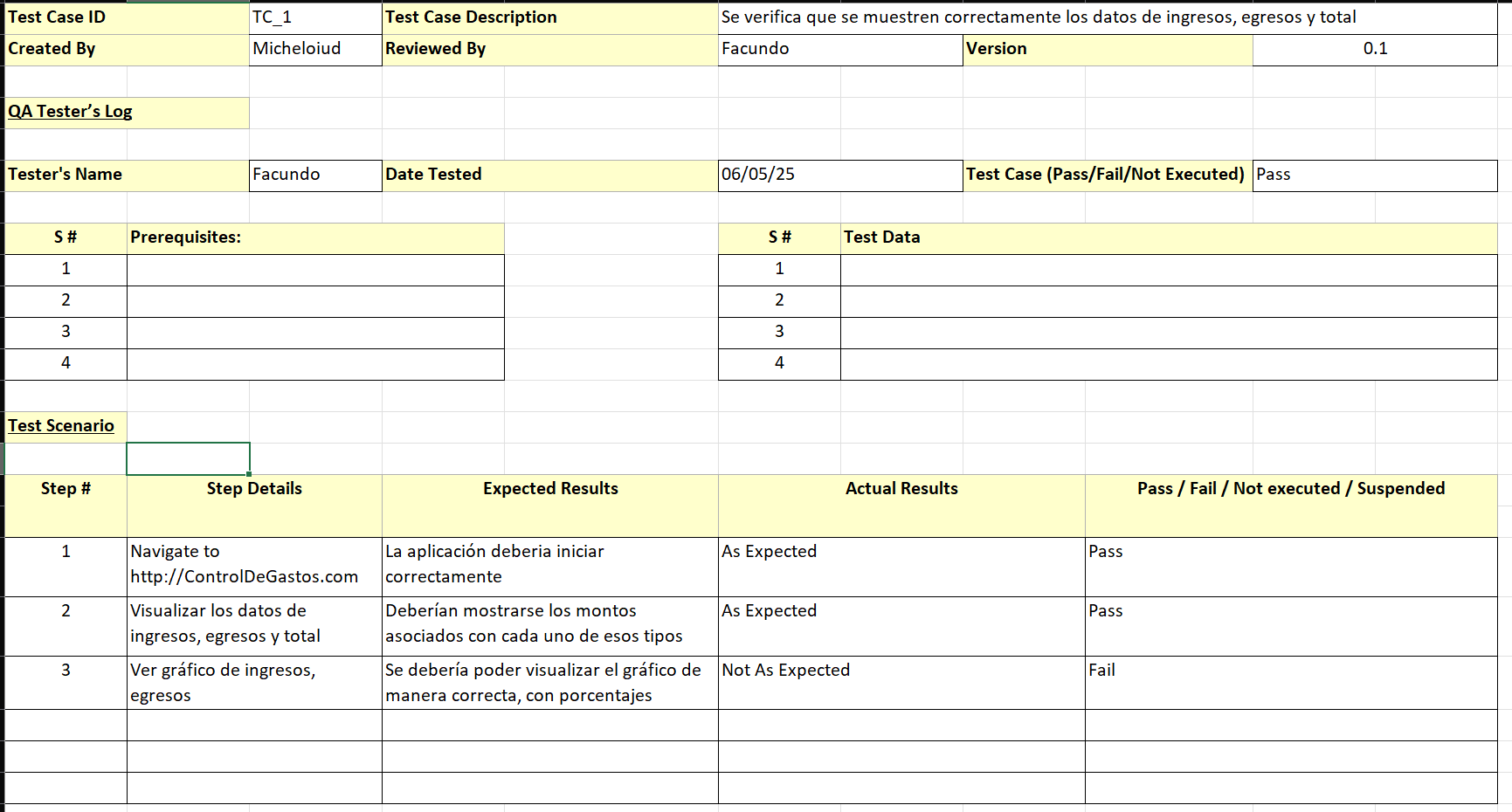


#2:

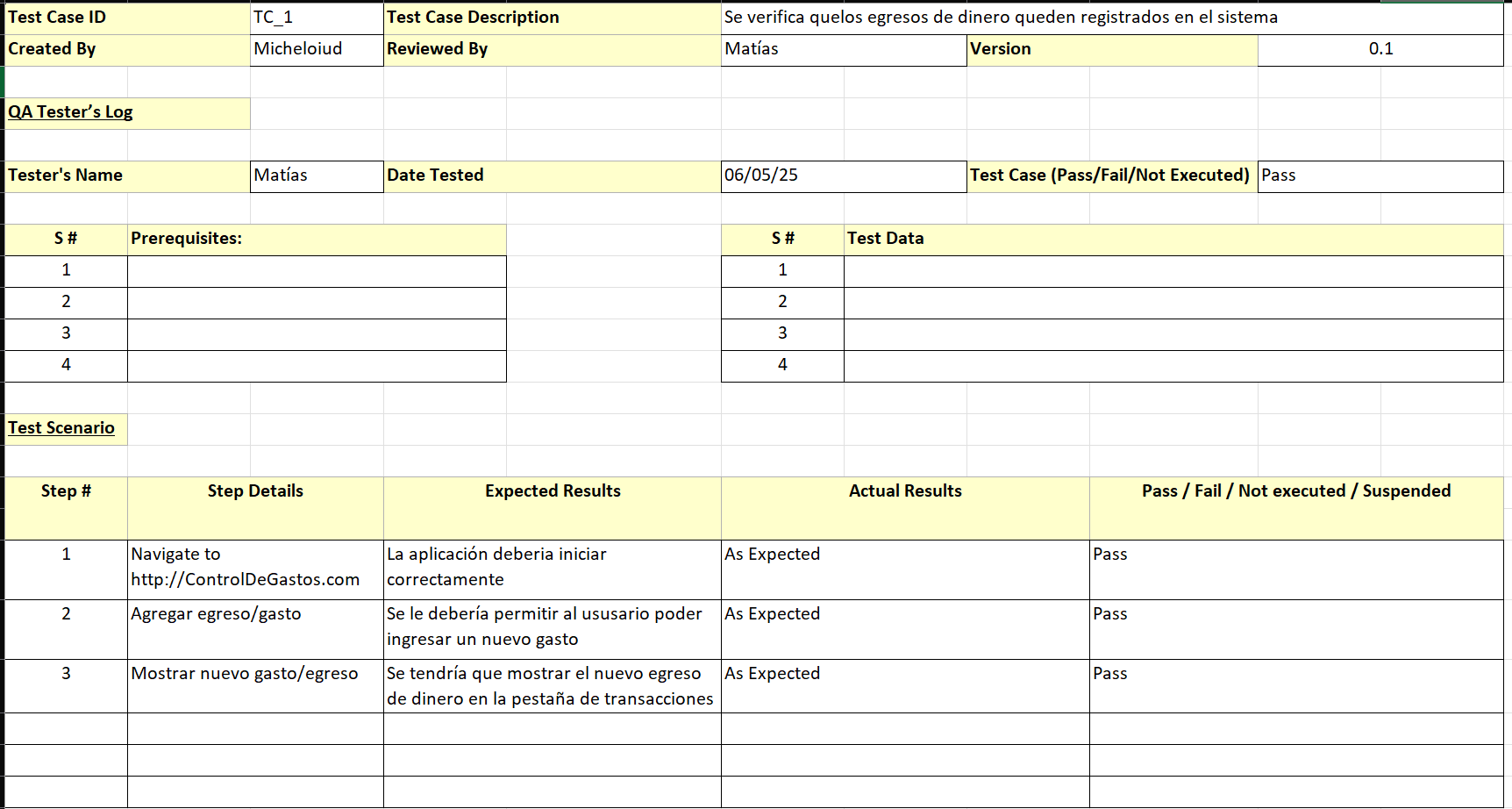


#3



#4

#5



**Segunda Iteración:**

Arquitectura:

La arquitectura de software es la estructura fundamental de un sistema informático. Define cómo se organizan sus componentes, cómo se comunican entre sí y qué principios rigen su diseño. Es decir, representa la forma en la que se construye internamente una aplicación, organizando su lógica, presentación, almacenamiento de datos y comunicación con el entorno.

En este proyecto, se adopta una arquitectura Cliente-Servidor, por ser adecuada para los requerimientos y el alcance del sistema de control de gastos.

*Arquitectura Cliente-Servidor*:

La arquitectura Cliente-Servidor es un modelo donde el sistema se divide en dos partes bien diferenciadas:

Cliente: Es la interfaz con la que interactúa el usuario (por ejemplo, una aplicación web o móvil). Se encarga de enviar solicitudes al servidor y mostrar las respuestas recibidas.

Servidor: Es el componente encargado de procesar las solicitudes del cliente. Contiene la lógica de negocio del sistema y el acceso a la base de datos.

En este modelo, el servidor gestiona todos los datos y operaciones importantes, mientras que el cliente se limita a presentar información y capturar acciones del usuario.

*Justificación de su uso*:

Se utiliza la arquitectura Cliente-Servidor en este sistema por los siguientes motivos:

Separación de responsabilidades: Permite dividir claramente la interfaz de usuario (cliente) y la lógica del sistema (servidor), facilitando el desarrollo y mantenimiento del software.

Seguridad: La base de datos solo es accesible desde el servidor, protegiendo la integridad y privacidad de los datos del usuario.

Escalabilidad: Esta arquitectura permite mejorar cada parte (cliente o servidor) de forma independiente según sea necesario.

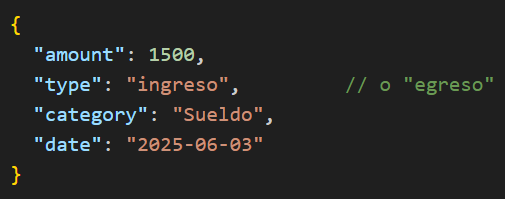
Reutilización: El mismo servidor puede ser utilizado por distintas plataformas cliente, como una aplicación web y una aplicación móvil.

Modelo ampliamente probado: Es una arquitectura estándar, ideal para aplicaciones que requieren autenticación, gestión de usuarios y persistencia de datos.

Flujo de Datos entre Capas:

En la arquitectura Cliente-Servidor adoptada, la comunicación entre el cliente (frontend) y el servidor (backend) se realiza a través de solicitudes HTTP, principalmente utilizando el formato JSON para el intercambio de información.

*Datos enviados del Cliente al Servidor*:  
Cuando el usuario registra una nueva transacción o realiza alguna operación sobre ellas, el cliente envía al servidor un objeto con la siguiente estructura:



Estos datos son enviados mediante una solicitud HTTP POST (para crear), PUT (para actualizar) o DELETE (para eliminar), según la operación.

*Datos enviados del Servidor al Cliente*:  
El servidor responde con objetos que representan las transacciones almacenadas, incluyendo información adicional generada automáticamente, como el identificador único y las fechas de creación y actualización:



Cuando se solicita la lista de transacciones, el servidor devuelve un arreglo de objetos con esta estructura.

Errores clasificados por su gravedad:

Errores críticos

* Fallo en la conexión a la base de datos: Si el backend no puede conectarse a PostgreSQL, ninguna operación de transacciones será posible.
* El backend no inicia: Por ejemplo, si el puerto está ocupado o hay errores graves en la configuración, el sistema queda fuera de servicio.

Errores altos

* Error al crear o actualizar una transacción por datos inválidos: Si el usuario envía un monto vacío, una fecha inválida o un tipo/categoría no permitida, el backend debe rechazar la operación y el frontend debe mostrar un mensaje claro.
* Transacción no encontrada: Si se intenta editar o eliminar una transacción que no existe (por ejemplo, por manipulación de la URL), el backend responde 404 y el frontend debe informar al usuario.
* Fallo en la obtención de datos: Si el frontend no puede cargar la lista de transacciones (por error de red o del servidor), el usuario no puede ver sus movimientos.

Errores medios

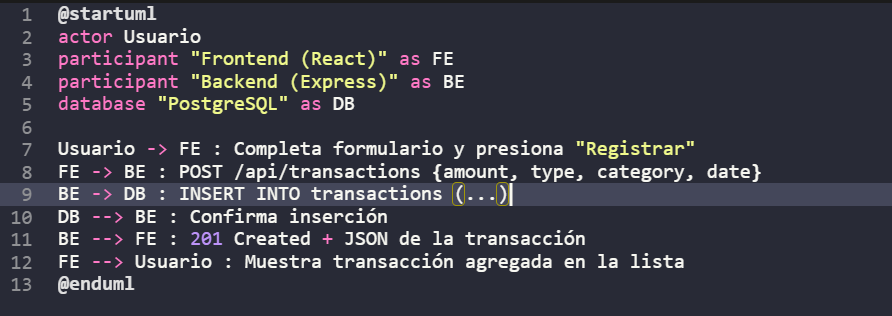
* Errores de validación en el frontend: El usuario intenta enviar el formulario con campos vacíos o mal formateados. El frontend debe evitar el envío y mostrar advertencias.
* Desincronización de datos: El usuario agrega una transacción, pero la lista no se actualiza automáticamente (requiere recarga manual).
* Problemas de formato de fecha o número: El backend recibe datos en formato incorrecto y rechaza la operación.

Errores bajos

* Mensajes de error poco claros o genéricos: El usuario recibe un mensaje como "Error del servidor" sin detalles.
* Errores visuales menores: Por ejemplo, la lista de transacciones no se actualiza instantáneamente tras agregar una nueva, o pequeños problemas de diseño en la interfaz.
* Advertencias en consola: Warnings de React o del navegador que no afectan la funcionalidad.

Diagrama de Secuencia:

El diagrama de secuencia muestra cómo interactúan los objetos del sistema en el tiempo, especificando el orden en que ocurren los mensajes. Es útil para representar un flujo específico de un caso de uso, indicando quién llama a quién y en qué secuencia.



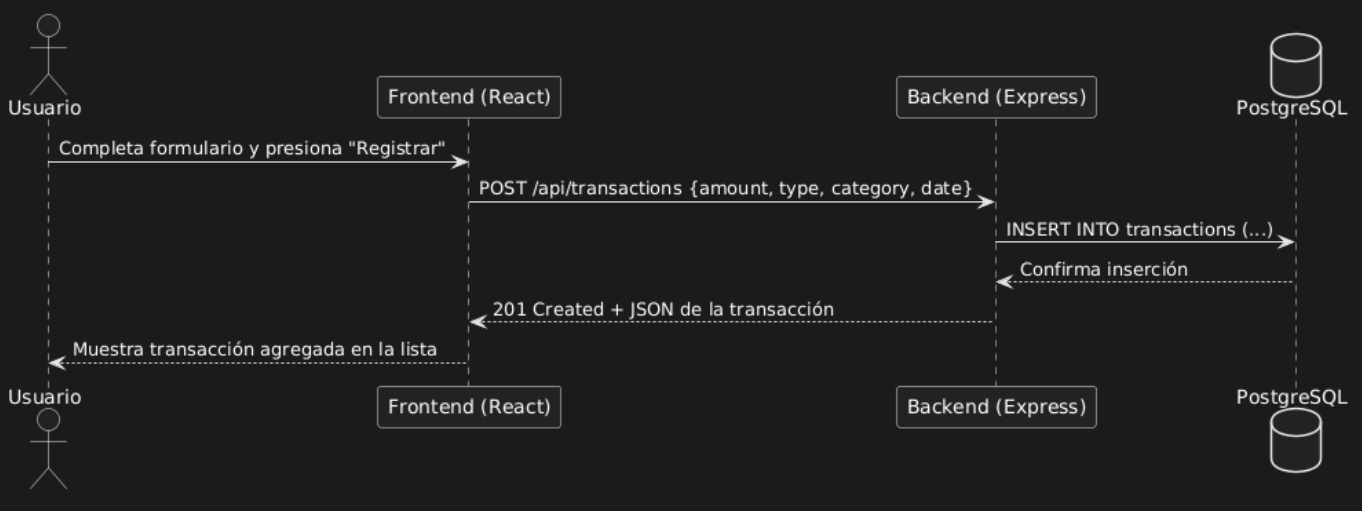
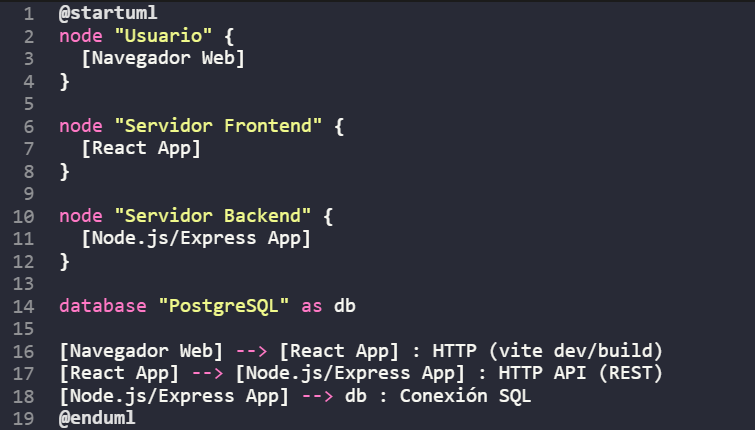


Diagrama de Despliegue:

El diagrama de despliegue muestra la distribución física del sistema: qué componentes de software se ejecutan en qué nodos o dispositivos de hardware. Es útil para visualizar la infraestructura y la arquitectura técnica.



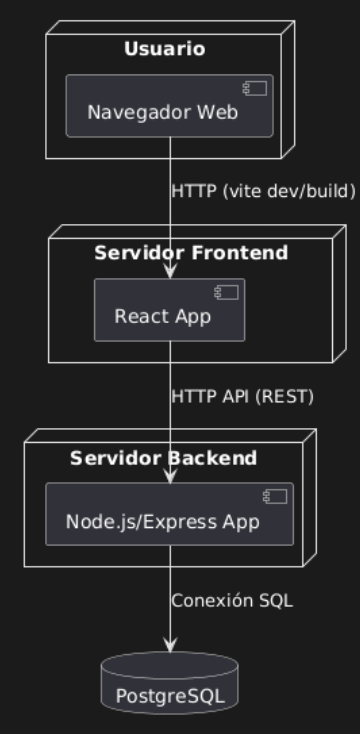


Diagrama de Clases:

Este diagrama representa las clases del sistema, sus atributos, métodos y las relaciones entre ellas (herencia, asociación, composición, etc.). Es clave para entender la estructura estática del software.



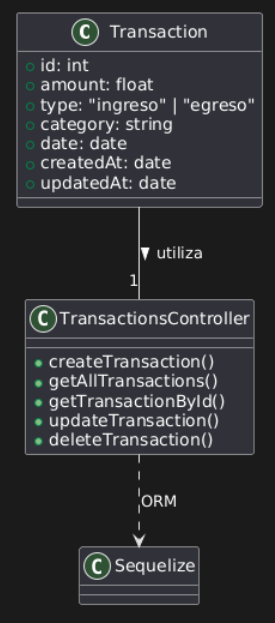
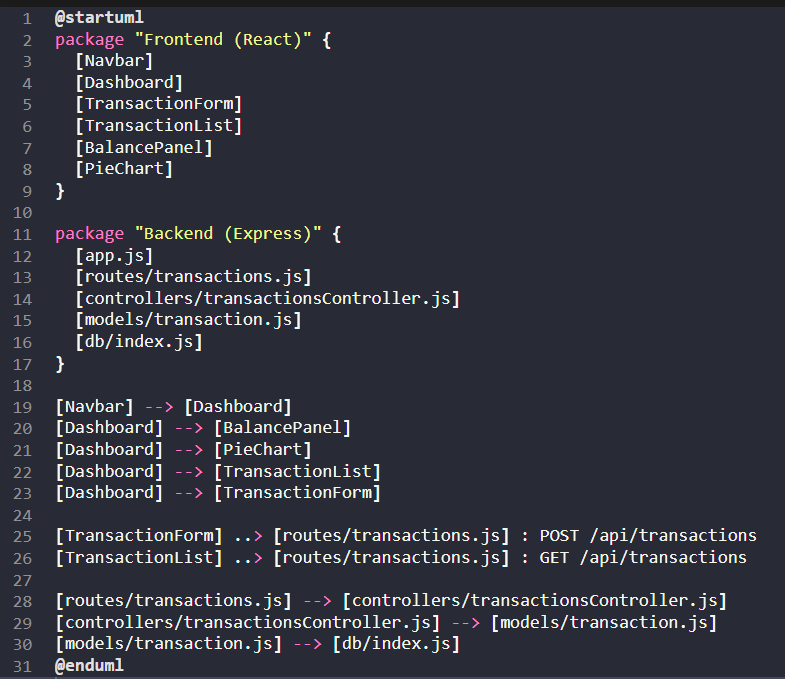
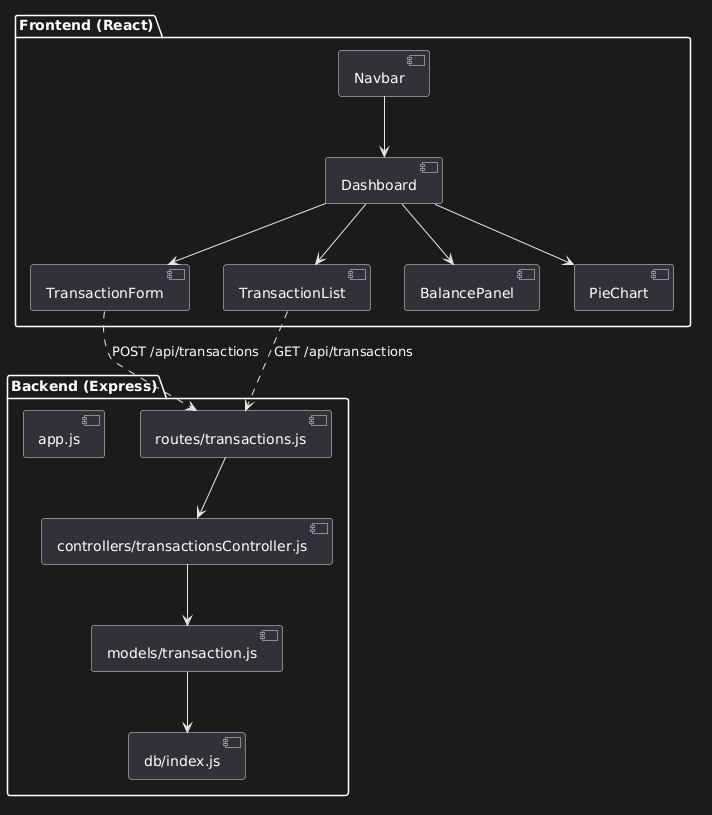


Diagrama de Componentes:

Este diagrama representa los componentes lógicos y reutilizables del sistema, como módulos, bibliotecas o servicios. Muestra cómo se conectan y se comunican entre sí.





**Tercer Iteración:**

Implementación:

Aplicación web full-stack desarrollada para el control de finanzas personales. El sistema está construido con React/Vite en el frontend y Node.js/Express en el backend, utilizando PostgreSQL (Supabase) como base de datos. Implementa un sistema completo de autenticación con JWT que permite a los usuarios registrarse, iniciar sesión y gestionar sus transacciones financieras de manera segura. La aplicación incluye un dashboard principal con visualizaciones de datos (gráficos de barras y torta) que muestran el resumen financiero, gastos por categoría y tendencias mensuales. Los usuarios pueden crear, editar, eliminar y filtrar transacciones (ingresos y egresos) organizadas por categorías. La interfaz es completamente responsiva, desarrollada con Tailwind CSS, e incluye funcionalidades como modales para formularios, validaciones en tiempo real, y un sistema de navegación intuitivo. El backend proporciona una API REST completa con endpoints para autenticación y gestión de transacciones, implementando middleware de seguridad, validaciones robustas y logging. El proyecto utiliza una arquitectura de monorepo con scripts configurados para desarrollo simultáneo de frontend y backend, y está completamente funcional con conexión a base de datos en la nube.

Enpoints:

**Endpoints de Autenticación**

* POST /api/auth/register - Registro de usuarios
* POST /api/auth/login - Inicio de sesión

**Endpoints de Transacciones**

* POST /api/transactions - Crear transacción
* GET /api/transactions - Obtener todas las transacciones
* GET /api/transactions/:id - Obtener transacción por ID
* PUT /api/transactions/:id - Actualizar transacción
* DELETE /api/transactions/:id - Eliminar transacción

**Cuarta Iteración:**

**Definición de Requerimientos NO Funcionales:**

No Funcionales:

*Cumple con*:

1. El sistema de control de gastos debe proporcionar una interfaz de usuario estéticamente agradable que permita una interacción satisfactoria para el usuario.

La aplicación implementa un diseño visual moderno con gradientes, esquemas de colores consistentes, iconos intuitivos y elementos interactivos que facilitan la navegación. La interfaz incluye cards con sombras, bordes redondeados, transiciones suaves y estados hover que mejoran la experiencia visual. Además, proporciona feedback visual mediante notificaciones de éxito y error, garantizando que los usuarios puedan interactuar de manera agradable y eficiente con todas las funcionalidades del sistema.

1. El sistema debe garantizar la autenticidad de los usuarios mediante un mecanismo robusto de verificación de identidad.

La aplicación implementa autenticación basada en tokens JWT que valida la identidad de cada usuario antes de permitir el acceso a funcionalidades protegidas. El sistema incluye un middleware de autenticación obligatorio para todas las rutas sensibles, encriptación de contraseñas usando bcrypt con salt rounds, y separación estricta de datos donde cada usuario solo puede acceder a sus propias transacciones y categorías. Los tokens tienen tiempo de expiración configurado y se validan en cada solicitud, asegurando que solo usuarios autenticados y autorizados puedan realizar operaciones en el sistema.

*Se deberían implementar*:

1. El sistema debería implementar validaciones comprehensivas del lado cliente y servidor

Incluyendo confirmaciones para acciones destructivas, capacidad

de deshacer operaciones, y una función de "papelera" para recuperar transacciones eliminadas accidentalmente.

Se debería agregar validación en tiempo real, confirmaciones modales elegantes, historial de operaciones, y soft-delete en lugar de eliminación permanente.

1. El sistema debería ser completamente responsive y optimizado para dispositivos móviles

El sistema deberá incluir navegación táctil, menús colapsables, y layouts específicos para pantallas pequeñas.

Incluyendo l navegación tipo “hamburger” para optimizar el uso del espacio en pantalla, el uso de cards apilables que permitan presentar la información de manera ordenada y legible en disposición vertical, la habilitación de gestos de desplazamiento (swipe) para realizar acciones rápidas sin necesidad de acceder a menús adicionales, y la reorganización de los elementos bajo un enfoque mobile-first, priorizando la usabilidad en pantallas pequeñas y adaptando posteriormente la interfaz a resoluciones mayores.

**Quinta Iteración**

**Análisis de 5 problemas de seguridad, y explicación técnica de las contramedidas correspondientes**

**1. Falta de control sobre los datos que ingresan los usuarios**

**Problema:**  
Si no revisamos ni limpiamos los datos que los usuarios envían (por ejemplo, al cargar una transacción), pueden ingresar información incorrecta o incluso peligrosa, lo que puede dañar la base de datos o hacer que la aplicación funcione mal.

**Contramedida:**

* Revisar siempre que los datos que llegan sean del tipo y formato correcto.
* Limpiar los datos para evitar que alguien intente engañar al sistema con información maliciosa.
* Esto se logra usando herramientas que ayudan a verificar y limpiar los datos antes de guardarlos o usarlos.

**2. Contraseñas y acceso poco seguros**

**Problema:**  
Si las contraseñas de los usuarios se guardan tal cual las escriben, cualquiera que acceda a la base de datos podría verlas y usarlas. Además, si el sistema de acceso no es seguro, alguien podría entrar sin permiso. Por otro lado, si alguien intenta adivinar la contraseña de un usuario probando muchas veces, podría lograrlo si no hay un límite de intentos.

**Contramedida:**

* Guardar las contraseñas de forma cifrada, para que nadie pueda leerlas, aunque acceda a la base de datos.
* Usar sistemas de acceso modernos que generen “llaves” temporales para identificar a los usuarios.
* Esto requiere cambiar cómo se guardan y revisan las contraseñas, y agregar controles en las partes protegidas de la aplicación.
* Implementar un sistema que bloquee temporalmente la cuenta o retrase los intentos después de varios fallos.

**3. Riesgo de mostrar información secreta**

**Problema:**  
Si archivos con datos importantes (como contraseñas de la base de datos o claves secretas) se suben por error a internet, cualquiera podría verlos y usarlos para atacar la aplicación.

**Contramedida:**

* Asegurarse de que estos archivos nunca se suban al repositorio ni se compartan.
* Usar servicios especiales para guardar información secreta en los servidores.
* Es importante revisar la configuración y asegurarse de que estos archivos estén protegidos.

**4. Riesgo de mostrar contenido peligroso en la web**

**Problema:**  
Si la aplicación muestra en pantalla información que los usuarios escriben, y no la revisa, alguien podría hacer que otros usuarios vean mensajes peligrosos o scripts que se ejecuten en su navegador.

**Contramedida:**

* No mostrar directamente el contenido que los usuarios escriben, sino revisarlo y limpiarlo antes.
* Usar herramientas que eliminan cualquier código peligroso antes de mostrarlo en la web.
* Revisar los lugares donde se muestran datos de usuarios y asegurarse de que sean seguros.

**5. Falta de control sobre lo que puede hacer cada usuario**

**Problema:**  
Si no se controla bien qué puede hacer cada usuario, cualquiera podría ver o modificar información que no le corresponde, como los datos de otros usuarios.

**Contramedida:**

* Definir claramente qué puede hacer cada tipo de usuario (por ejemplo, un usuario normal y un administrador).
* Agregar controles para que la aplicación revise los permisos antes de permitir ciertas acciones.
* Esto implica agregar revisiones en las rutas y funciones que manejan los datos.