Universidad de la Cuenca del Plata
Autorización Definitiva. Decreto Poder Ejecutivo Nacional N°091/2006

Trabajo Integrador

CARRERA: Ingeniería en sistemas de información

MATERIA: Paradigmas y Lenguajes de Programación III

COMISIÓN: “U” (única) “A”

PROFESOR: Mgter. Ing. Agustín Encina

ESTUDIANTES: Benitez Lucia.

FECHA: 07-10-2025

REPOSITORIO: https://github.com/luciabz/AHORRA-MAS

Sitio Web: https://ahorra-mas.vercel.app/

Introducción

El proyecto Ahorra+ surge como una propuesta de aplicación web destinada a la gestión y organización de finanzas personales, con el objetivo de brindar a los usuarios herramientas simples y visuales para controlar ingresos, egresos y metas de ahorro.

Asimismo, el propósito inicial de este trabajo fue aplicar buenas prácticas en el uso de HTML y CSS, además de incorporar una metodología de desarrollo ordenada, que facilite la escalabilidad y mantenibilidad del sistema a futuro.

En esta primera etapa se trabajó en el diseño de la arquitectura inicial, la estructura del proyecto y la integración de tecnologías base, priorizando la modularidad y la claridad en la organización del código.

La segunda etapa se centró en el despliegue tanto del backend como del frontend, logrando la integración completa y la subida de la base de datos para almacenar la información de los usuarios. Además, se realizó la adaptación responsive del diseño para asegurar la correcta visualización y funcionalidad en diferentes dispositivos.

Finalmente, se implementaron diversas mejoras y correcciones clave. Se corrigio un error de recarga en la sección de categorías, se modificó el diseño de la funcionalidad de ahorro para mejorar su usabilidad y se incorporaron calculadoras de tiempo para facilitar la planificación financiera. Con estas adiciones y ajustes, el proyecto Ahorra+ quedó listo para su uso.

Desarrollo

Parte 1

Actualmente, el proyecto se encuentra en una fase inicial de desarrollo, donde se implementaron principalmente la estructura y el diseño de los componentes. Se construyó una arquitectura organizada en capas, con carpetas para components, pages, shared, utils y constants.

Se desarrollaron páginas base

- Dashboard
- FixedExpenses
- SavingsGoal
- VariableIncome

Incorporaciones

Se incorporaron componentes de interfaz reutilizables, como tablas, formularios de gastos y selectores de mes.

Los datos se simulan mediante archivos JSON, lo cual permite probar comportamientos sin necesidad de contar aún con una base de datos real.

Se implementó una funcionalidad condicional básica: cuando el JSON define que el ingreso es de tipo *variable*, en el Dashboard aparece un botón que redirige a una página exclusiva para usuarios con ingresos variables.

En este punto, las funcionalidades son mínimas y se limitan a algunos cálculos simples en JavaScript y a la visualización de la estructura de la interfaz.

Tecnologías Utilizadas y Justificación

El stack tecnológico fue seleccionado considerando la eficiencia en el desarrollo, la escalabilidad futura y la facilidad de mantenimiento.

- React

  Se utilizó para el desarrollo de la interfaz basada en componentes reutilizables, lo cual permite una organización clara y un mantenimiento más sencillo.

- Tailwind CSS

  Se incorporó para el diseño visual, ya que facilita el desarrollo de interfaces responsive y modernas con menor cantidad de código CSS personalizado.

- JavaScript, HTML y CSS

  Constituyen la base del desarrollo web, utilizados para la lógica principal, estructura y estilos complementarios.

- Vite

  Elegido como compilador y empaquetador por su rendimiento superior en comparación con herramientas tradicionales como Webpack. Esto agiliza los tiempos de desarrollo y mejora la experiencia del programador.

- Recharts

  Librería especializada en visualización de datos que se usará para representar

gráficamente ingresos, gastos y metas de ahorro, permitiendo un análisis más intuitivo de la información.

- Lucide Icons

  Implementados para mejorar la estética del proyecto mediante íconos simples y modernos.

- Archivos JSON

  Utilizados para la simulación de datos en ausencia de una base de datos real, lo cual permite verificar la estructura y las reglas de negocio de forma preliminar.

- Recursos PNG

  Imágenes utilizadas para reforzar la identidad visual del sistema.

Avances Técnicos

- Se implementó un sistema de navegación básica entre páginas.
- Se estructuraron componentes de dashboard como tablas de gastos, formularios de ingreso fijo y variable, y resúmenes de totales.
- Se realizaron cálculos básicos en JavaScript para pruebas iniciales de sumatorias y diferencias.

Se desarrolló un primer control de acceso simulado, basado en los datos del JSON, que diferencia entre usuarios con ingresos fijos y variables.

Futuras Implementaciones

Se plantean como próximos pasos los siguientes objetivos:

- Gestión de estado global con Redux: implementar Redux Toolkit para un manejo más eficiente de los estados de la aplicación.

- Arquitectura Limpia (Clean Architecture): reorganizar el proyecto en capas de dominio, aplicación, infraestructura y presentación, con el fin de aumentar la mantenibilidad y escalabilidad.

- Persistencia de datos: reemplazar los JSON por una base de datos y una API que permita registrar información real de los usuarios.

Parte 2

Evolución a Clean Architecture e Integración de API

Para transformar el prototipo en una aplicación funcional, se reemplazaron los datos simulados por una integración completa con una API, reestructurando el proyecto bajo los principios de **Clean Architecture**. Esta arquitectura separa el código en capas independientes:

1. Capa de Infraestructura (`src/infrastructure/api/`) Aquí reside la configuración de la comunicación con el exterior.

- **`axiosInstance.js`**: Se creó una instancia de Axios configurada que actúa como cliente HTTP. Incluye un interceptor automático que añade el token de autorización (JWT) a todas las peticiones, centralizando la lógica de autenticación.

2. Capa de Datos (`src/data/`) Esta capa implementa los repositorios que se comunican con la API. Su función es separar la lógica de negocio del acceso a los datos, manejando de forma consistente las respuestas y los errores del servidor. Se crearon repositorios para autenticación, transacciones, categorías, metas y transacciones programadas.

3. Capa de Presentación (`src/presentation/`) Contiene la lógica de la interfaz de usuario y cómo esta interactúa con las otras capas.

- **Hooks Personalizados (`src/presentation/hooks/`)**: Se desarrolló un conjunto de hooks que encapsulan toda la lógica de interacción con la API. Esto hace que los componentes sean más limpios y declarativos.
    - `useAuth.js`: Gestiona la autenticación de usuarios (login, register, logout), el token JWT y los datos del usuario en `localStorage`.
    - `useTransactions.js`: Proporciona el CRUD completo para transacciones, filtrado por tipo, categoría y mes, y el cálculo automático de totales.
    - `useCategories.js`: Maneja el CRUD de categorías, con filtros y búsqueda por nombre.
    - `useGoals.js`: Administra el CRUD de metas de ahorro, calcula el progreso y gestiona su estado (activas, completadas, vencidas).

- `useScheduledTransactions.js`: Permite el CRUD de transacciones programadas y el filtrado por frecuencia.

- **Contexto de Autenticación (`src/presentation/contexts/`)**:
  - `AuthContext.jsx`: Proporciona un contexto global para el estado de autenticación. Esto permite que cualquier componente en la aplicación pueda acceder a la información del usuario y a las funciones de autenticación de manera sencilla y eficiente.

3. Despliegue y Configuración de Proxy (HTTPS ↔ HTTP)

Uno de los desafíos técnicos fue desplegar el frontend en una plataforma segura (HTTPS, ej. Vercel) mientras el backend operaba en un servidor estándar (HTTP). Esto genera errores de **"Mixed Content"**, ya que los navegadores modernos bloquean peticiones inseguras desde un contexto seguro.

Solución Implementada: Estrategia de Doble Proxy

Se implementó una solución transparente que no requiere cambios en el backend:

1. **Proxy Principal (Vercel Rewrites)**: Se configuró el archivo `vercel.json` para que todas las peticiones del frontend a la ruta `/api/:path*` sean redirigidas internamente por Vercel al backend `http://3.85.57.147:8080/api/:path*`. Para el navegador, la petición siempre se realiza al mismo dominio (HTTPS), eliminando el error de "Mixed Content".

2. **Proxy Fallback (Función Serverless)**: Como respaldo y para manejar casos más complejos, se utiliza una función serverless que toma el control si el rewrite falla, asegurando la robustez de la comunicación.

Configuración por Entorno

El sistema detecta automáticamente si está en un entorno de desarrollo o producción:

- `.env.development`: `VITE_URL_API` apunta directamente a la IP del backend (`http://3.85.57.147:8080`).
- `.env.production`: `VITE_URL_API` se deja vacía. Esto hace que las peticiones usen URLs relativas (`/api/...`), activando así el proxy de Vercel.

4. Estado Funcional y Características Implementadas

Gracias a la integración de la API, la aplicación ahora cuenta con un conjunto completo de funcionalidades:

- **Autenticación**: Login y registro de usuarios, con manejo automático de tokens JWT y rutas protegidas.
- **Transacciones**: CRUD completo de transacciones, integrado en el Dashboard y un gestor de transacciones.
- **Categorías**: Gestión de categorías de ingresos y gastos, disponibles en los formularios de transacciones.
- **Metas de Ahorro**: Creación, actualización, eliminación y seguimiento del progreso de las metas.

- **Transacciones Programadas**: Gestión de transacciones recurrentes (ej. sueldos, alquileres).
- **Manejo de Errores y Carga**: Todos los hooks gestionan estados de carga y manejan errores de la API, proveyendo feedback visual al usuario. Un interceptor global gestiona los errores 401 (token inválido), cerrando la sesión automáticamente.

Componentes Actualizados Los siguientes componentes fueron refactorizados para utilizar la nueva arquitectura de hooks y contexto: `Dashboard.jsx`, `TransactionManager.jsx`, `GoalsManager.jsx`, `LoginForm.jsx` y `PrivateRoute.jsx`.

Funcionalidades realizadas

- Registro y autenticación de usuarios.
- Configuración y registro de categorías, transacciones variables y programadas.
- Calculo de totales para visualizacion del usuario.
- Interfaz responsive apta para todo dispositivo

5. Futuras Implementaciones

Con la base actual sólidamente establecida, los próximos pasos se centran en optimizar y expandir las funcionalidades:

- **Optimización de Rendimiento**:
    - Implementar paginación en listas con grandes volúmenes de datos (ej. historial de transacciones).
    - Agregar un sistema de caché local para reducir las peticiones a la API y mejorar la velocidad de carga.
- **Mejoras de Funcionalidad**:
    - Implementar sincronización offline para que la aplicación pueda ser utilizada con conectividad limitada.
    - Añadir notificaciones push para recordar sobre transacciones programadas o metas por vencer.
- **Expansión de Visualización de Datos**: Explotar en mayor medida la librería **Recharts** para ofrecer una visión más detallada y personalizable de los movimientos financieros.

Implementación y despliegue del backend

El backend de Ahorra+ fue desarrollado en **Node.js** con **TypeScript**, usando **Express** como framework y **TypeORM** como capa de persistencia sobre **PostgreSQL**. Se implementó una API REST organizada siguiendo principios de arquitectura limpia (separación en capas: infraestructura, datos y presentación). El servicio se desplegó en una **instancia EC2 de AWS** (puerto 8080). Para pruebas funcionales se creó una colección de **Postman** que muestra el

correcto funcionamiento de los endpoints principales (ej.: `POST /api/v1/auth/register` retorna `201 Created` con el mensaje “Usuario creado exitosamente”).

Tecnologías y librerías (backend)

Productivo

- `pg` — driver de PostgreSQL para Node.js.
- `reflect-metadata` — metadatos en TypeScript (necesario para TypeORM).
- `typeorm` — ORM para entidades y consultas.
- `express` — framework web.
- `express-validator` — middleware para validación de entradas (integrado, quedan reglas por completar).
- `jsonwebtoken` — emisión/validación de JWT.
- `argon2` — hashing seguro de contraseñas.

- `winston` — logging estructurado y persistente.
- `cors` — habilitación de CORS.
- `dotenv` — carga de variables de entorno.
- `morgan` — logging de peticiones HTTP.
- `helmet` — cabeceras de seguridad HTTP.

Endpoints principales

La colección de Postman contiene las rutas más importantes, organizadas por recurso. Ejemplos representativos:

- **Auth**
  - `POST /api/v1/auth/register` — crear usuario (body: `{ "name": "...", "email": "...", "password": "..." }`). Respuesta: `201 Created` → `"Usuario creado exitosamente"`.
  - `POST /api/v1/auth/login` — login (retorna JWT).

  - `GET /api/v1/auth/me` — obtiene datos del usuario autenticado.

- **Transaction**

  - `GET /api/v1/transaction` — listar transacciones (filtros por mes, tipo, categoría).

  - `POST /api/v1/transaction` — crear transacción.

  - `GET /api/v1/transaction/:id` — detalle.

  - `PATCH /api/v1/transaction/:id` — actualizar.

  - `DELETE /api/v1/transaction/:id` — eliminar.

- **Schedule-transaction** (transacciones programadas): CRUD completo.

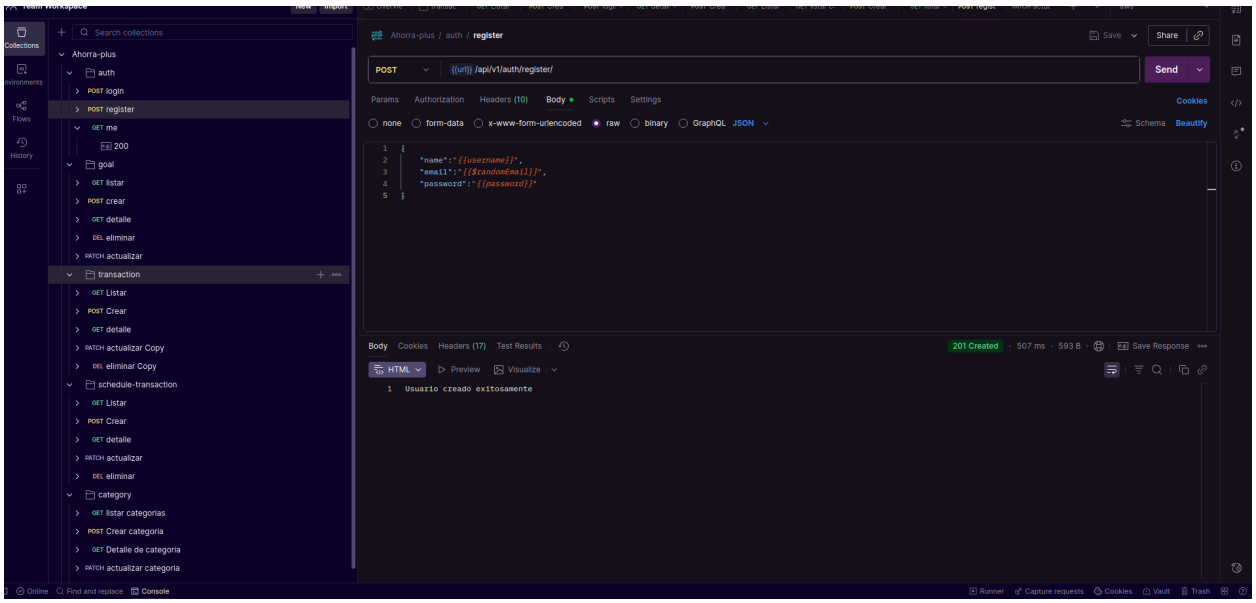
- **Goal**

  - `GET /api/v1/goal` / `POST` / `GET /:id` / `PATCH` / `DELETE`.

- **Category**

    - Endpoints para listar, crear, actualizar y eliminar categorías.

En la colección Postman se usan variables de entorno (por ejemplo `{{url}}`) para apuntar al servidor desplegado. En la captura incluida en el informe se puede ver la request de registro y la respuesta `201 Created`.



Autenticación y seguridad

- **JWT**: La autenticación se realiza mediante tokens JWT. El token se envía en la cabecera `Authorization: Bearer <token>`.

- **Hashing de contraseñas**: `argon2` se utiliza para almacenar contraseñas de forma segura.

- **Cabeceras de seguridad**: `helmet` para reducir la superficie de ataques a través de cabeceras HTTP.

- **CORS**: configurado mediante `cors` para permitir que el frontend (desplegado en Vercel) consuma la API.

- **Buenas prácticas aplicadas**: variables sensibles (secrets, credenciales de DB) gestionadas mediante `.env` (no subidas al repo).

- **Nota**: hay un interceptor que detecta `401` y provoca el cierre de sesión del cliente para evitar estados inválidos.

Despliegue en AWS

- El backend fue desplegado en una **instancia EC2** de AWS y expuesto en el **puerto 8080**.

- **Accesibilidad**: el Security Group asociado permite el tráfico entrante HTTP hacia el puerto 8080 desde el frontend/proxy (restringir a orígenes necesarios es recomendable).

- El frontend está desplegado en **Vercel**, y en producción se usan **rewrites** para proxy inverso (`/api/:path*` → backend) para evitar errores de “Mixed Content” al consumir un backend HTTP desde un frontend HTTPS. En desarrollo `VITE_URL_API` apunta directamente a la IP pública de la EC2 (`http://3.85.57.147:8080`), mientras que en producción las llamadas usan URL relativas para beneficiarse del rewrite de Vercel.

- **Recomendación para producción**: mover el backend a un balanceador con certificado TLS (ELB + ACM) o usar un reverse proxy (NGINX) con HTTPS para asegurar la comunicación directa hacia la API.

Logging, monitoreo y manejo de errores

- **Logging**: `winston` y `morgan` se usan para registrar eventos y peticiones HTTP. Los logs permiten depurar errores y auditar el comportamiento en producción.

- **Manejo de errores**: existe un handler global que unifica la estructura de respuesta ante errores (código HTTP, mensaje legible y, opcionalmente, detalles en entorno de desarrollo).

- **Mejoras pendientes en errores**: se deben estandarizar los mensajes de error, retornar códigos y payloads consistentes y asegurar que no se filtren detalles sensibles al cliente.

- Configuración cargada con dotenv en el arranque de la aplicación.

Pendientes y mejoras recomendadas

1. **Validaciones completas**: completar reglas con express-validator (mensajes por campo, manejo de errores de validación).

2. **Mensajes de error personalizados**: retornar errores con formato uniforme y mensajes amigables.

3. **Pruebas automatizadas**: agregar tests unitarios e integrados (Jest / Supertest) para endpoints críticos.

4. **HTTPS en backend**: habilitar TLS (load balancer o reverse proxy) para evitar depender únicamente del proxy de Vercel.

5. **Rate limiting y protección**: agregar express-rate-limit y protección contra fuerza bruta en endpoints de auth.

6. **Migrations y backups**: asegurar migraciones (TypeORM migrations) y estrategia de backup de la DB.

7. **Monitoreo y alertas**: integrar un sistema de métricas/logs centralizado (CloudWatch, Sentry, ELK).

8. **Políticas de contraseña y verificación de email**: implementar validaciones de fortaleza de contraseña y verificación por email.

9. **CI/CD:** pipeline para despliegues automatizados a AWS (por ejemplo GitHub Actions → despliegue a EC2/Elastic Beanstalk/ ECS).

Parte 3: Consolidación y Optimización

Esta fase final del proyecto se centró en la depuración, optimización de la usabilidad y aplicación de los últimos ajustes de estilo para la presentación y el uso operativo del sistema.

1. Mejoras de Estabilidad y Usabilidad

Corrección de Errores de Recarga

Se identificó y corrigió un error crítico que provocaba un comportamiento inconsistente o una recarga innecesaria en la sección de gestión de categorías al interactuar con el *hook* useCategories.js.

- **Problema:** Tras agregar, actualizar o eliminar una categoría, el estado no se refrescaba correctamente, y provocaba una pagina en blanco, afectando la experiencia de usuario.
- **Solución:** Se revisó la lógica de gestión de estado dentro del *hook* de categorías, asegurando que las funciones de mutación (addCategory, updateCategory, deleteCategory) invaliden y refetchteen los datos de la lista de categorías de forma optimizada, utilizando las capacidades de gestión de estado asíncrono y caché provistas.

Optimización del Módulo de Metas de Ahorro (SavingsGoal)

Se refactorizó el módulo de metas de ahorro para hacerlo más intuitivo y funcional, proporcionando a los usuarios herramientas de planificación financiera.

- **Ajuste de Diseño:** Se mejoró la interfaz de usuario para la creación y edición de metas, haciendo el flujo más claro y la visualización del progreso más prominente.
- **Incorporación de Calculadoras Financieras:** Se integraron utilidades en el formulario de metas para asistir en la planificación:
    - **Calculadora de Tiempo:** Permite al usuario, ingresando el monto objetivo y un monto de ahorro mensual deseado, estimar el tiempo (meses/años) que tardará en alcanzar la meta.
    - **Calculadora de Ingreso:** Permite al usuario, ingresando el monto objetivo y la fecha límite, calcular la cantidad de dinero que debe ahorrar mensualmente para cumplir la meta a tiempo.

2. Refinamiento de Estilos y Diseño Responsive

Se realizó un pase final de pulido visual y de estilos en todo el frontend, garantizando una experiencia de usuario coherente y profesional, utilizando la potencia de **Tailwind CSS**.

Componente	Ajuste Realizado	Objetivo
Dashboard	Ajustes de espaciado y tipografía en resúmenes de totales.	Mayor legibilidad y jerarquía visual.
Formularios	Diseño unificado de botones de acción (Submit, Cancel).	Consistencia y claridad en las interacciones.
Adaptación Responsive	Revisión de *breakpoints* en componentes críticos (tablas y formularios grandes).	Asegurar la correcta visualización y funcionalidad en dispositivos móviles.
Navegación	Pequeños ajustes en la barra lateral (Sidebar) para optimizar el espacio vertical.	Mejora de la usabilidad en pantallas pequeñas.

3. Checklist de Funcionalidades Finales

Los siguientes puntos confirman que la aplicación está lista para su uso y presentación:

Funcionalidad	Estado	Descripción
Registro/Login	Completo	Autenticación basada en JWT, rutas protegidas.

Funcionalidad	Estado	Descripción
CRUD de Transacciones	Completo	Gestión total de ingresos y gastos.
CRUD de Metas de Ahorro	Completo	Incluye las nuevas calculadoras de tiempo e ingreso.
CRUD de Categorías	Completo	Sin errores de recarga.
Despliegue Frontend	Estable	En Vercel, con proxy configurado para HTTPS.
Despliegue Backend	Estable	En AWS EC2, API REST funcional.
Diseño Responsive	Completo	Adaptación para todo tipo de dispositivo.

Conclusión

El proyecto Ahorra+ ha superado con éxito su fase inicial, evolucionando de un prototipo con datos simulados a una aplicación web funcional con una arquitectura robusta y escalable. La implementación de **Clean Architecture**, la integración de una API real mediante hooks personalizados y la resolución de desafíos de despliegue como la comunicación HTTPS-HTTP, han sentado una base técnica sólida.

El desafío actual ya no es construir la estructura fundamental, sino refinar la experiencia del usuario, optimizar el rendimiento y expandir el conjunto de características para transformar Ahorra+ en una herramienta de gestión financiera completa y competitiva.

El backend de Ahorra+ se encuentra funcional y expuesto públicamente en AWS, con endpoints de autenticación y CRUD funcionando y probados mediante Postman. La arquitectura aplicada (TypeORM + Express + capas separadas) facilita la mantenibilidad y la evolución del sistema. Quedan tareas de robustecimiento (validaciones, manejo uniforme de errores, pruebas automatizadas y asegurar TLS en el backend) antes de considerar el sistema listo para un entorno de producción con tráfico real.