WALLET-TEST

Documentación Técnica

Estructura del Proyecto

Se implementó una arquitectura basada en Clean Architecture, con separación por capas. El proyecto está compuesto por los siguientes proyectos en la solución:

- `API/`: Capa de presentación (ASP.NET Core Web API).
- `Application/`: Casos de uso, CQRS con MediatR, lógica de aplicación.
- `Domain/`: Entidades del dominio y lógica de negocio.
- `Infrastructure/`: Persistencia de datos, configuración de servicios externos.
- `Test/`: Pruebas unitarias e integración.
- `docker-compose`: Orquestación lista para contenerización.

Configuración Inicial y Dependencias

Se crearon los proyectos en GitHub y se estructuró la solución conforme a los principios de código limpio. Se instalaron las siguientes dependencias clave:

- Entity Framework Core: para la persistencia de datos.
- Autenticación (Identity/Auth): incluida y lista para ser utilizada (aunque no implementada completamente).
- MediatR: utilizado como patrón de mediador para desacoplar la API de la capa de aplicación.
- Inyección de dependencias: se configuraron los servicios de las capas Application e Infrastructure mediante el contenedor de inyección de .NET.
- Docker: se añadió soporte para contenedor Docker, lo que permite orquestar la solución fácilmente.
- xUnit + Moq: Framework de testing y mocks.

Patrón de Resultados y Manejo Global de Errores

Se implementó un patrón de resultados para estandarizar las respuestas de la API, acompañado de un middleware de manejo global de errores que permite controlar y devolver respuestas claras ante errores comunes o excepciones de validación.

Arquitectura con MediatR

Aunque no era requisito, se integró MediatR como mediador entre la API y la capa de aplicación. Esta implementación:

- Permite una arquitectura más escalable y desacoplada.
- Transforma los servicios generales por entidad en servicios específicos por caso de uso (por ejemplo, CreateWalletHandler, CreateMovementHandler).
- Mejora la mantenibilidad del código al aislar la lógica de cada funcionalidad.

Inyección de Servicios

Se configuró la inyección de los siguientes servicios:

- Servicios de la capa de aplicación (Application), incluyendo MediatR.
- Servicios de la capa de infraestructura (Infrastructure), incluyendo:
  - DbContext de Entity Framework para modelar entidades de dominio y generar el modelo relacional.
  - Redis (u otro servicio de cache/logs si aplica).
  - PaginateService Se implementó un servicio de paginación genérico que permite paginar tanto listas (List) como IQueryable desde base de datos.

Patron de Resultados

Se implementó un patrón Result para estandarizar las respuestas entre capas. Este patrón permite:

- Respuestas claras (Success, Failure, etc.).
- Errores ricos en contexto (code, message, type).
- Mejor integración con validaciones y manejo global de errores.

Construcción de Entidades y Casos de Uso

Se definieron las entidades del dominio:

- Wallet: representa la billetera digital.
- Movement: representa el historial de movimientos asociados.

Se utilizó un patrón de factoría para construir las entidades, evitando constructores públicos y favoreciendo métodos estáticos para una inicialización controlada.

Entre los casos de uso implementados se encuentran:

- CreateWallet (Crea una nueva billeter)
- CreateMovement (Registra un movimiento asociado a una billetera)
- UpdateWallet (actualización de campos descriptivos)

- UpdateFunds (feature independiente para modificar fondos)
- GetWallets (Consulta de wallets paginada)
- GetMovements (Consulta de movimientos paginados)
- DeleteWallet (Elimina una wallet)

Cada uno de estos casos de uso cuenta con su correspondiente Command, Handler y tests.

Pruebas

Se realizaron pruebas unitarias y de integración para validar los siguientes escenarios:

- Creación de billeteras (CreateWalletCommandHandlerTest)
    - Se valida la creación exitosa con datos válidos.
    - Se previene la creación si ya existe una billetera con el mismo documentId.
- Actualización de billeteras (UpdateWalletCommandHandlerTest)
    - Se actualizan correctamente los datos descriptivos.
    - Se devuelve un error si la billetera no existe.
- Modificación de fondos (UpdateFundsCommandHandlerTest)
    - Se permite agregar o retirar fondos según el ActionType.
    - Se valida que no se puedan retirar más fondos de los disponibles.
    - Se controla el envío de una acción no válida.
- Eliminación de billeteras (DeleteWalletCommandHandlerTests)
    - Se elimina exitosamente una billetera existente.
    - Se devuelve error si la billetera no existe.
- Creación de movimientos (CreateMovementCommandHandlerTest)
    - Se registra correctamente un movimiento válido.
    - Se controla que existan las billeteras de origen y destino.
- Paginación de listas (PagedListTest)
    - Se valida el comportamiento de metadatos de paginación (página actual, total de elementos, páginas, etc.).
    - Se asegura la correcta división de datos en páginas.

Estado Actual

Actualmente se encuentran funcionales los endpoints y casos de uso relacionados con:

- Creación de billetera (CreateWallet)

- Creación de movimiento (CreateMovement)
- Transferencia de fondos (Transfer)
- Actualización de billetera (UpdateWallet)
- Actualización de fondos (UpdateFunds)
- Consulta de billeteras
- Consulta de movimientos
- Eliminacion de billeteras

Autenticación y validaciones con FluentValidation no fueron implementadas por cuestiones de tiempo, pero el sistema está preparado para integrarlos, en especial las validaciones en command y querys.

El proyecto está preparado para ser ejecutado en entornos Docker, con:

- Dockerfile y docker-compose.override.yml
- Base de datos PostgreSQL incluida.
- Soporte para aplicar migraciones automáticas.

Gracias a su arquitectura limpia y desacoplada, la solución es:

- Altamente escalable y mantenible
- Lista para extenderse con nuevas entidades (como Client, Loan, User, etc.)
- Capaz de incorporar validaciones, filtros, autenticación, colas de eventos, y más.