Introducción

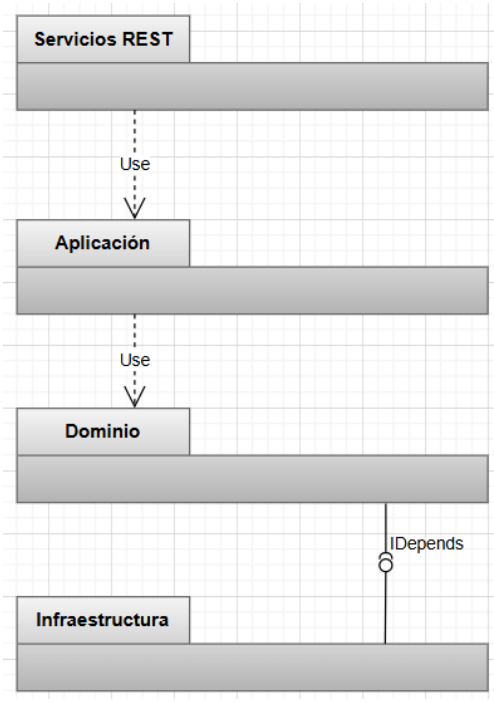
La estructura propuesta para el proyecto de gestión de transacciones HSA en Python, basada en los principios de Domain-Driven Design (DDD) y una arquitectura en capas, servirá como base para que cada grupo implemente la mayor cantidad de requisitos y funcionalidades de sus proyectos asignados. Esta organización modular facilita la asignación de responsabilidades claras y la colaboración efectiva entre los equipos, permitiendo que cada grupo se enfoque en desarrollar componentes específicos dentro de la arquitectura general.

Al seguir esta estructura, los equipos podrán trabajar de manera independiente en diferentes módulos, como la capa de dominio, la capa de aplicación, la infraestructura y los servicios REST, asegurando que las funcionalidades desarrolladas se integren de forma coherente en el sistema global. Además, esta organización promueve la mantenibilidad y escalabilidad del proyecto, ya que cada componente puede evolucionar de manera autónoma sin afectar negativamente a otros módulos.

Es fundamental que cada grupo comprenda las responsabilidades y relaciones de los componentes dentro de la arquitectura propuesta, garantizando que las implementaciones individuales se alineen con los principios de DDD y contribuyan al objetivo común de desarrollar un sistema robusto y eficiente para la gestión de transacciones HSA.

Arquitectura del Sistema

El diagrama representa una arquitectura de **Domain-Driven Design (DDD)** dividida en capas, donde cada capa tiene responsabilidades específicas y depende de la capa inmediatamente inferior, excepto en el caso de la **inversión de dependencias**.



Capas del Diagrama

1. Servicios REST (Interfaces del Sistema)

- **Descripción**:
    - Es la capa más externa y sirve como punto de interacción entre los usuarios (otros sistemas) y el sistema.
    - Esta capa no contiene lógica de negocio, solo delega las solicitudes a la capa de Aplicación.

2. Aplicación (Application Layer)

- **Descripción**:
    - Orquesta los casos de uso del sistema (Requisitos).
    - Coordina la interacción entre las interfaces de usuario (Servicios REST) y la lógica de negocio (Dominio).
    - No contiene lógica de negocio, pero asegura que las operaciones de las entidades en el Dominio se ejecuten de manera correcta.

3. Dominio (Domain Layer)

- **Descripción**:

- Es el núcleo del sistema y contiene la lógica de negocio más importante.
- Define las entidades, objetos de valor, interfaces (como IDepends) y cualquier regla que sea clave para el modelo del dominio.

4. Infraestructura (Infrastructure Layer)

- **Descripción**:
  - Gestiona las preocupaciones técnicas del sistema, como la persistencia de datos, redes o acceso a sistemas externos.
  - Implementa las interfaces definidas en el Dominio, asegurando que este último permanezca desacoplado de los detalles técnicos.

Relaciones Clave del Diagrama

1. **Flujo de Dependencia**:
   - El flujo comienza desde la capa de **Servicios REST** hacia abajo, pasando por Aplicación, Dominio e Infraestructura.
   - Cada capa depende solo de la siguiente capa, excepto en el caso de la **inversión de dependencias**.
2. **Inversión de Dependencias**:
   - El Dominio define interfaces (IDepends) que la Infraestructura implementa.
   - Esto desacopla la lógica de negocio (Dominio) de los detalles técnicos (Infraestructura).

Caso de Estudio: Gestión de Transacciones HSA

Introducción

Este caso de estudio está diseñado para proporcionar una **descripción parcial** de un sistema de gestión de **Health Savings Accounts (HSA)**, enfocándose en las transacciones y la generación de informes financieros. El diseño del sistema sigue una arquitectura en capas basada en los principios de **Domain-Driven Design (DDD)**, permitiendo la separación de responsabilidades y facilitando la extensibilidad.

¿Qué es una Health Savings Account (HSA)?

Una **HSA** es una cuenta de ahorros especializada diseñada para cubrir gastos médicos calificados. Estas cuentas son comunes en sistemas de salud como el de los Estados Unidos. Los usuarios pueden depositar fondos libres de impuestos y utilizarlos para pagar medicamentos, tratamientos y otros servicios de salud. Las características principales de las HSA incluyen:

- **Flexibilidad**: Los fondos no utilizados pueden acumularse para años futuros.
- **Beneficios fiscales**: Los depósitos, retiros (para fines médicos) y los intereses generados están exentos de impuestos.
- **Propiedad individual**: Cada cuenta está asociada a un usuario único, quien tiene control total sobre los fondos.

El objetivo del sistema en este caso de estudio es gestionar las transacciones de manera eficiente y facilitar la generación de informes financieros asociados a estas cuentas.

Objetivos del Proyecto

1. Diseñar una arquitectura en capas para el sistema, garantizando la separación de responsabilidades.
2. Implementar funcionalidades clave utilizando principios de diseño orientado al dominio (DDD).
3. Proveer una representación detallada de los componentes principales mediante diagramas UML.
4. Establecer un flujo claro de datos entre las capas de Servicios REST, Aplicación, Dominio e Infraestructura.

Requisitos del Sistema

Gestión de Transacciones

- Registrar transacciones de tipo:
    - Depósito.

    - Retiro.
- Consultar una transacción específica por su ID.
- Listar todas las transacciones asociadas a una cuenta.
- Generar informes financieros detallados para cada cuenta.

Gestión de Cuentas y Usuarios

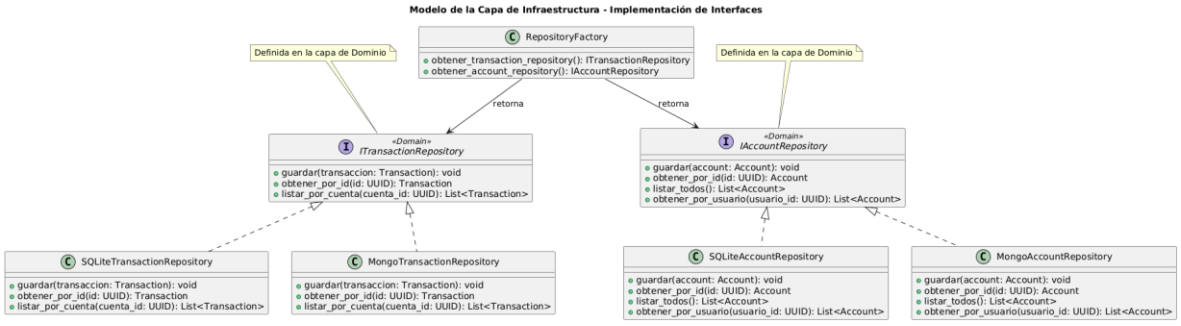
- Consultar el saldo actual de una cuenta.
- Asociar cuentas a usuarios.

Informes Financieros

- Generar informes detallados por cuenta:
    - Total de depósitos.
    - Total de retiros.
    - Saldo promedio mensual.

Diseño del sistema

Capa de Infraestructura (Parcial)



La **capa de infraestructura** es responsable de implementar las interfaces definidas en el dominio, proporcionando los mecanismos necesarios para la persistencia de datos y garantizando la inversión de dependencias.

Descripción General del Modelo

1. **Fábrica de Repositorios**:
    - La **RepositoryFactory** abstrae la creación de repositorios concretos. Esta fábrica utiliza las interfaces del dominio (ITransactionRepository e IAccountRepository) y retorna las implementaciones específicas (SQLite o Mongo) según la configuración del sistema.
2. **Repositorios Concretos**:

- Existen implementaciones específicas para transacciones y cuentas:
    - **Transacciones**: SQLiteTransactionRepository y MongoTransactionRepository.
    - **Cuentas**: SQLiteAccountRepository y MongoAccountRepository.
- Estas clases implementan las interfaces del dominio y contienen los métodos para interactuar con las bases de datos subyacentes.

3. **Relación con el Dominio**:
    - Las interfaces (ITransactionRepository e IAccountRepository) están definidas en el dominio para garantizar que esta capa dependa de abstracciones y no de implementaciones concretas.

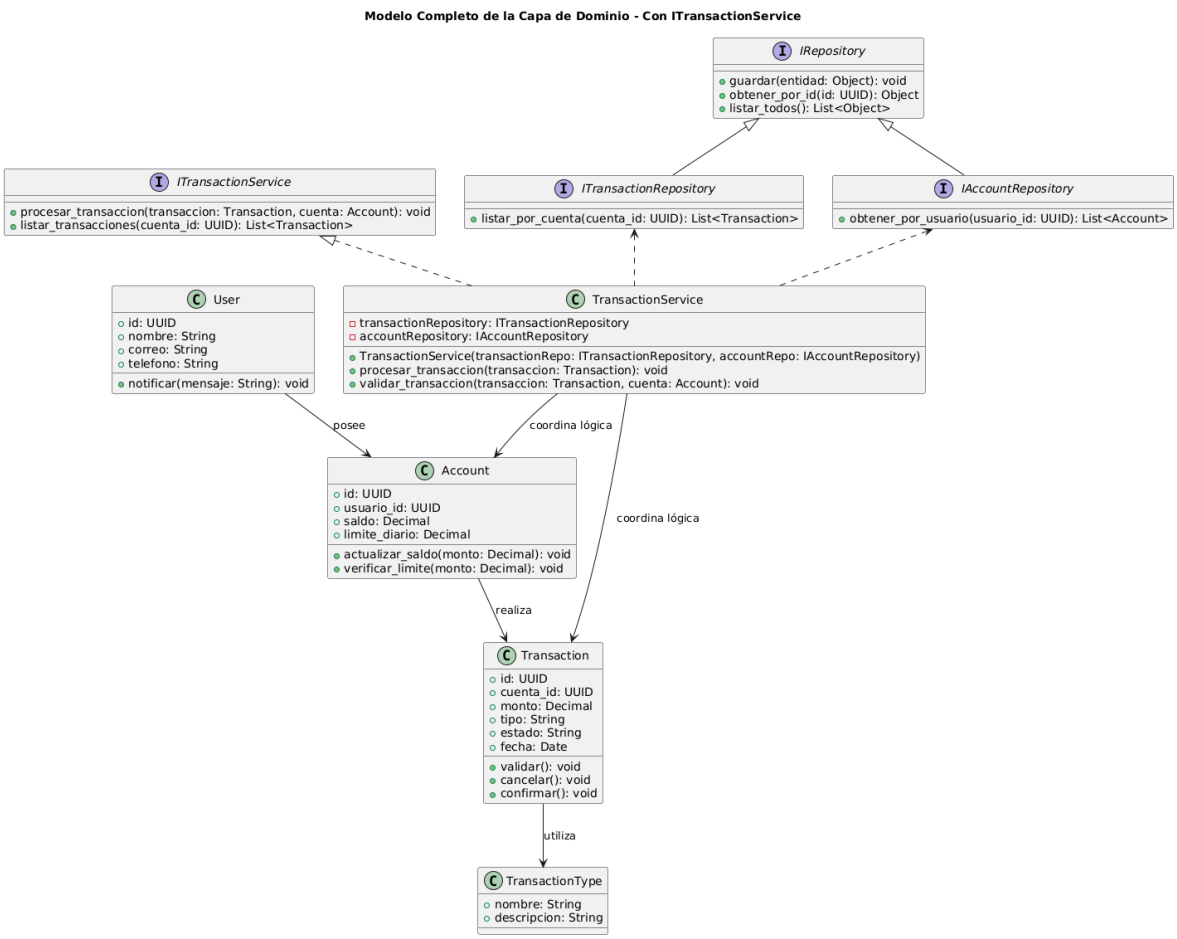
Flujo de Interacción

- Los servicios de aplicación (en la capa de aplicación) solicitan a la **RepositoryFactory** las implementaciones necesarias.
- La fábrica retorna la implementación específica (por ejemplo, SQLiteTransactionRepository) que se encarga de ejecutar operaciones de persistencia en la base de datos seleccionada.

Coherencia con DDD

- Esta capa cumple con el **principio de inversión de dependencias**, asegurando que el dominio no dependa de implementaciones concretas.
- La abstracción mediante la fábrica de repositorios facilita **la extensibilidad del sistema** y el cambio de tecnologías de persistencia sin afectar otras capas.

Capa de Dominio (Parcial)



La capa de dominio es el núcleo del sistema y encapsula la lógica de negocio principal. Está diseñada para ser independiente de las capas externas y garantizar que las reglas de negocio estén centralizadas y bien definidas.

- **Componentes Principales**:
    1. **Entidades**:
        - **User (Usuario)**: Representa a un usuario del sistema que puede poseer una o más cuentas.
        - **Account (Cuenta)**: Representa las cuentas HSA asociadas a un usuario, con lógica para actualizar saldos y verificar límites de retiros.
        - **Transaction (Transacción)**: Modela las operaciones de depósito o retiro, con estados que pueden ser validados, confirmados o cancelados.
        - **TransactionType**: Define el tipo de transacción (depósito/retiro), aportando claridad al dominio.
    2. **Servicios de Dominio**:

        - **TransactionService**: Coordina la lógica de negocio para las transacciones, como validarlas, procesarlas o listar las asociadas a una cuenta. Este servicio utiliza las entidades y delega operaciones de persistencia a las interfaces de repositorio.

    3. **Interfaces**:
        - **ITransactionService**: Define las operaciones relacionadas con las transacciones, permitiendo que la lógica de negocio sea accesible de manera abstracta.
        - **ITransactionRepository**: Abstracción para las operaciones de persistencia relacionadas con las transacciones.
        - **IAccountRepository**: Abstracción para las operaciones de persistencia relacionadas con las cuentas.
        - **IRepository**: Una interfaz genérica base para repositorios.

- **Coherencia con DDD**:
    - La lógica de negocio se mantiene en las entidades y los servicios de dominio.
    - Las interfaces garantizan que el dominio no dependa de detalles de implementación específicos, respetando la inversión de dependencias.
    - El diseño está centrado en el dominio, asegurando que las reglas de negocio sean consistentes y claras.

2. Capa de Aplicación

- **Servicios de Aplicación**:
    - TransactionApplicationService: Gestiona casos de uso como realizar transacciones y generar informes financieros.
- **DTOs (Data Transfer Objects)**:
    - TransactionDTO: Representa datos de transacciones para el transporte entre capas.
    - InformeDTO: Contiene información financiera agregada (total de depósitos, retiros, saldo promedio).

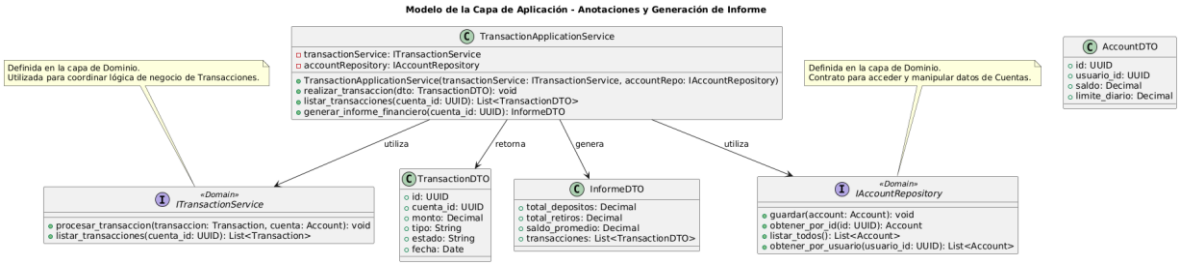
3. Capa de Infraestructura

- **Repositorios**:
    - ITransactionRepository e IAccountRepository (interfaces definidas en el dominio).

- Implementaciones concretas:
    - SQLiteTransactionRepository y MongoTransactionRepository para manejar persistencia de transacciones.
    - SQLiteAccountRepository y MongoAccountRepository para cuentas.

- **Fábrica de Repositorios**:
    - RepositoryFactory: Abstrae la creación de implementaciones específicas según la configuración

Capa de Aplicación



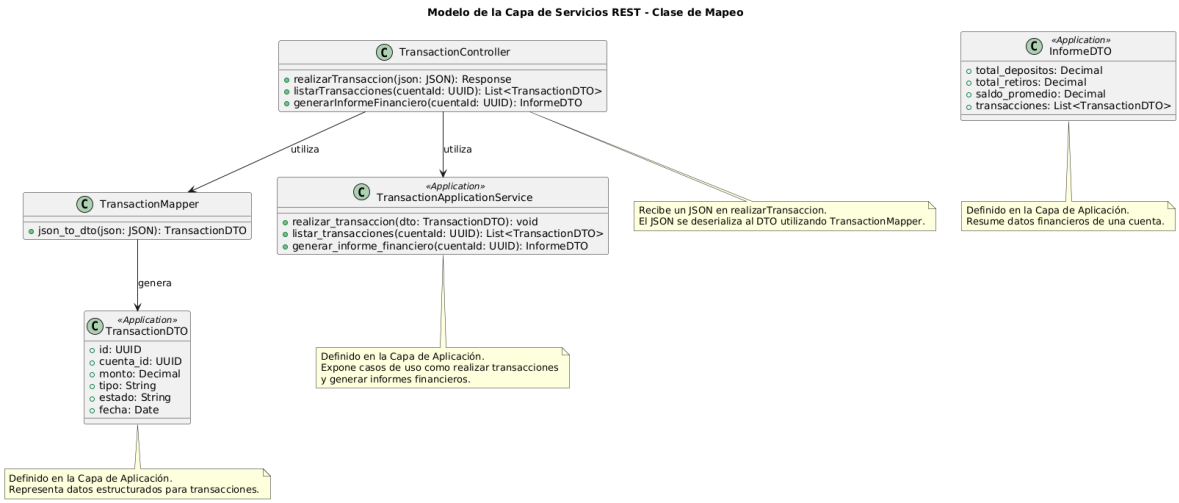
La **Capa de Aplicación** en una arquitectura basada en Domain-Driven Design (DDD) actúa como un intermediario entre la capa de presentación (o servicios REST) y la capa de dominio. Su principal responsabilidad es coordinar y orquestar las operaciones del sistema, implementando los casos de uso definidos por el negocio sin contener lógica de negocio propia.

Componentes Principales:

1. **Servicios de Aplicación**:
    - **TransactionApplicationService**: Gestiona casos de uso como realizar transacciones y generar informes financieros. Este servicio interactúa con los servicios de dominio y otros componentes necesarios para cumplir con las operaciones solicitadas.
2. **Data Transfer Objects (DTOs)**:
    - **TransactionDTO**: Representa los datos de una transacción para su transporte entre capas, facilitando la comunicación y evitando exponer directamente las entidades del dominio.
    - **InformeDTO**: Contiene información financiera agregada, como el total de depósitos, retiros y saldo promedio, estructurada para su presentación o procesamiento en otras capas.

Coherencia con DDD:

- La capa de aplicación se encarga de coordinar las operaciones del sistema según los casos de uso, delegando la lógica de negocio específica a la capa de dominio.
- Al utilizar **DTOs**, se facilita la transferencia de datos entre capas sin comprometer la integridad del modelo de dominio, manteniendo una separación clara de responsabilidades.
- Esta estructura permite que la lógica de negocio permanezca centralizada en la capa de dominio, mientras que la capa de aplicación se enfoca en la orquestación y coordinación de procesos, alineándose con los principios de DDD.



La **Capa de Servicios REST** en una arquitectura basada en Domain-Driven Design (DDD) actúa como la interfaz entre el sistema y los clientes externos, gestionando las solicitudes HTTP y facilitando la comunicación con las capas internas. Su función principal es recibir, procesar y responder a las peticiones de los usuarios, asegurando que la lógica de negocio se ejecute correctamente y que los datos se transfieran de manera adecuada.

Componentes Principales:

1. **Controladores REST**:
    - **TransactionController**: Maneja las operaciones relacionadas con las transacciones, como realizar una transacción, listar transacciones de una cuenta y generar informes financieros.
        - *Métodos*:
            - realizarTransaccion(json: JSON): Response

        - listarTransacciones(cuentaId: UUID): List<TransactionDTO>
        - generarInformeFinanciero(cuentaId: UUID): InformeDTO

2. **Mapeadores (Mappers)**:
    - **TransactionMapper**: Convierte datos entre diferentes representaciones, como transformar JSON en objetos DTO (json_to_dto(json: JSON): TransactionDTO).
3. **DTOs (Data Transfer Objects)**:
    - **TransactionDTO**: Representa los datos de una transacción que se transfieren entre las capas.
    - **InformeDTO**: Contiene información financiera agregada, como el total de depósitos, retiros y saldo promedio.

Coherencia con DDD:

- La Capa de Servicios REST se centra en la comunicación y orquestación, delegando la lógica de negocio a la Capa de Aplicación y la Capa de Dominio.
- Los DTOs facilitan la transferencia de datos entre capas sin exponer directamente las entidades del dominio, manteniendo la integridad y encapsulación del modelo de negocio.
- El uso de mapeadores asegura que los datos se transformen adecuadamente entre las representaciones externas (JSON) y las internas (DTOs), promoviendo la separación de responsabilidades y la cohesión del sistema.

Repositorio

Para implementar un repositorio en Python siguiendo los principios de Domain-Driven Design (DDD), es fundamental estructurar el proyecto de manera que refleje las capas y responsabilidades definidas en el modelo. A continuación, se presenta una estructura básica del repositorio, alineada con las capas de Dominio, Aplicación, Infraestructura y Servicios REST:

```
gestion_transacciones_hsa/
├── app/
│   ├── __init__.py
│   ├── controllers/
│   │   ├── __init__.py
│   │   └── transaction_controller.py
│   ├── mappers/
│   │   ├── __init__.py
│   │   └── transaction_mapper.py
│   └── main.py
├── application/
```

```
│   ├── __init__.py
│   ├── services/
│   │   ├── __init__.py
│   │   └── transaction_application_service.py
│   ├── dtos/
│   │   ├── __init__.py
│   │   ├── transaction_dto.py
│   │   └── informe_dto.py
├── domain/
│   ├── __init__.py
│   ├── entities/
│   │   ├── __init__.py
│   │   ├── user.py
│   │   ├── account.py
│   │   ├── transaction.py
│   │   └── transaction_type.py
│   ├── services/
│   │   ├── __init__.py
│   │   └── transaction_service.py
│   └── repositories/
│       ├── __init__.py
│       ├── i_transaction_repository.py
│       └── i_account_repository.py
├── infrastructure/
│   ├── __init__.py
│   ├── repositories/
│   │   ├── __init__.py
│   │   ├── sqlite_transaction_repository.py
│   │   ├── sqlite_account_repository.py
│   │   ├── mongo_transaction_repository.py
│   │   └── mongo_account_repository.py
│   └── factory/
│       ├── __init__.py
│       └── repository_factory.py
├── tests/
│   ├── __init__.py
│   ├── test_app/
│   │   ├── __init__.py
│   │   └── test_transaction_controller.py
│   ├── test_application/
│   │   ├── __init__.py
│   │   └── test_transaction_application_service.py
│   ├── test_domain/
│   │   ├── __init__.py
│   │   └── test_transaction_service.py
│   └── test_infrastructure/
│       ├── __init__.py
│       └── test_sqlite_transaction_repository.py
├── requirements.txt
└── README.md
```

Descripción de la Estructura:

- app/: Contiene la lógica de la capa de presentación y servicios REST.
    - controllers/: Define los controladores que manejan las solicitudes HTTP y delegan operaciones a los servicios de aplicación.
        - transaction_controller.py: Controlador responsable de las operaciones relacionadas con transacciones.
    - mappers/: Contiene los mapeadores que transforman datos entre diferentes representaciones.
        - transaction_mapper.py: Mapeador que convierte JSON en objetos DTO y viceversa.

    - main.py: Punto de entrada de la aplicación.
- application/: Gestiona los casos de uso y coordina la interacción entre la capa de presentación y el dominio.
    - services/: Implementa la lógica de los casos de uso específicos de la aplicación.
        - transaction_application_service.py: Servicio que maneja operaciones como realizar transacciones y generar informes financieros.
    - dtos/: Define los Data Transfer Objects utilizados para la comunicación entre capas.
        - transaction_dto.py: DTO que representa los datos de una transacción.
        - informe_dto.py: DTO que contiene información financiera agregada.
- domain/: Encapsula la lógica de negocio principal y es independiente de las capas externas.
    - entities/: Define las entidades del dominio con sus atributos y comportamientos.
        - user.py: Entidad que representa a un usuario del sistema.
        - account.py: Entidad que representa una cuenta HSA.
        - transaction.py: Entidad que modela una transacción.
        - transaction_type.py: Entidad que define el tipo de transacción (depósito/retiro).
    - services/: Contiene la lógica de negocio que no encaja directamente en una entidad específica.
        - transaction_service.py: Servicio que coordina la lógica de negocio para las transacciones.
    - repositories/: Define las interfaces para las operaciones de persistencia.
        - i_transaction_repository.py: Interfaz para las operaciones de persistencia relacionadas con transacciones.
        - i_account_repository.py: Interfaz para las operaciones de persistencia relacionadas con cuentas.
- infrastructure/: Implementa detalles técnicos y de persistencia.
    - repositories/: Contiene las implementaciones concretas de las interfaces de repositorio.
        - sqlite_transaction_repository.py: Implementación de ITransactionRepository utilizando SQLite.
        - sqlite_account_repository.py: Implementación de IAccountRepository utilizando SQLite.
        - mongo_transaction_repository.py: Implementación de ITransactionRepository utilizando MongoDB.
        - mongo_account_repository.py: Implementación de IAccountRepository utilizando MongoDB.

    - factory/: Proporciona una fábrica para crear instancias de los repositorios según la configuración.
        - repository_factory.py: Fábrica que abstrae la creación de implementaciones específicas de repositorios.
- tests/: Contiene las pruebas unitarias y de integración para las distintas capas.
    - test_app/: Pruebas para la capa de presentación y servicios REST.
        - test_transaction_controller.py: Pruebas para `Transaction