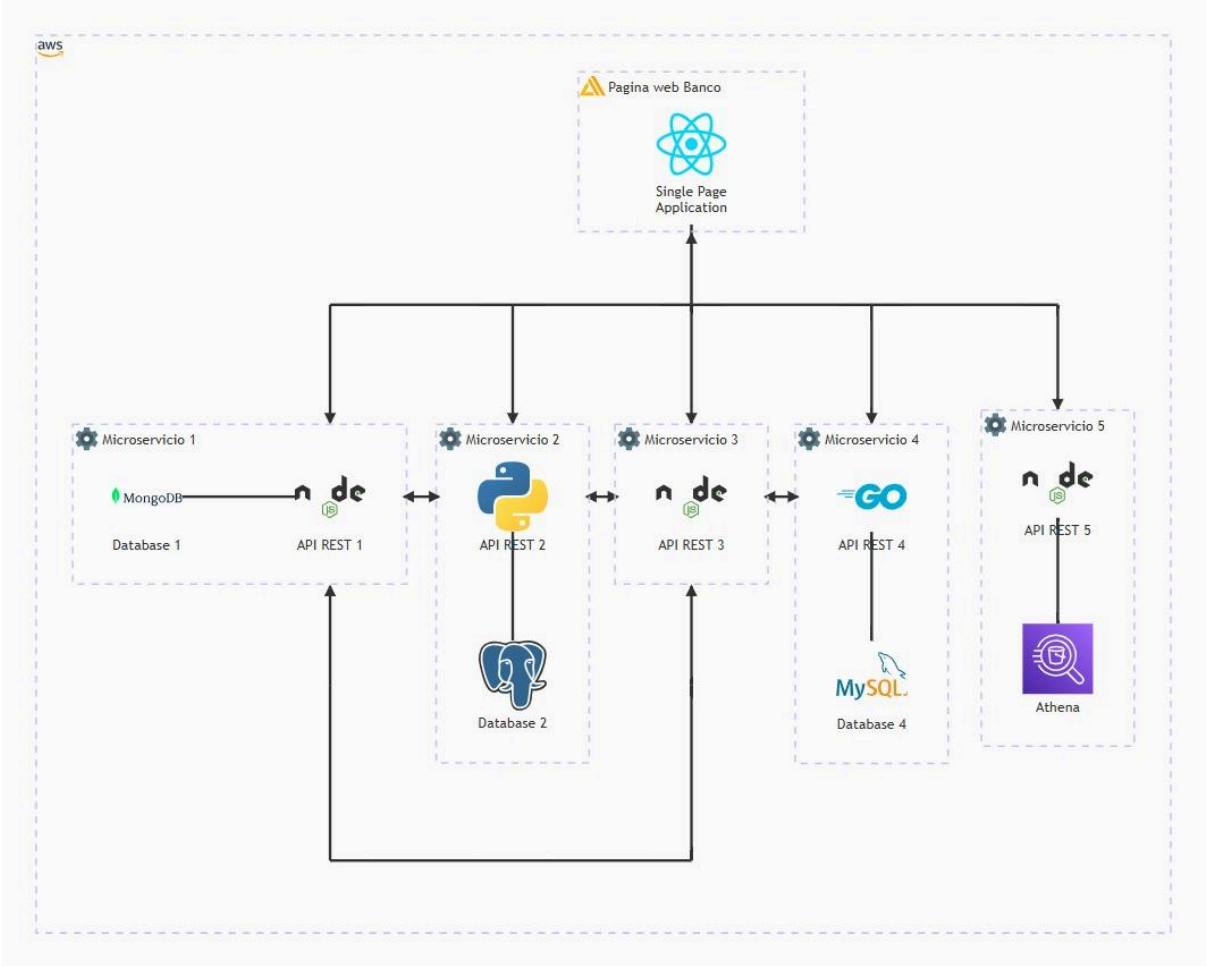
Integrantes:

- Anthony Sleiter Aguilar Sanchez
- Efrén Paolo Centeno Rosas
- Franco Stefano Panizo Muñoz
- Jhonatan Eder Ortega Huaman
- @

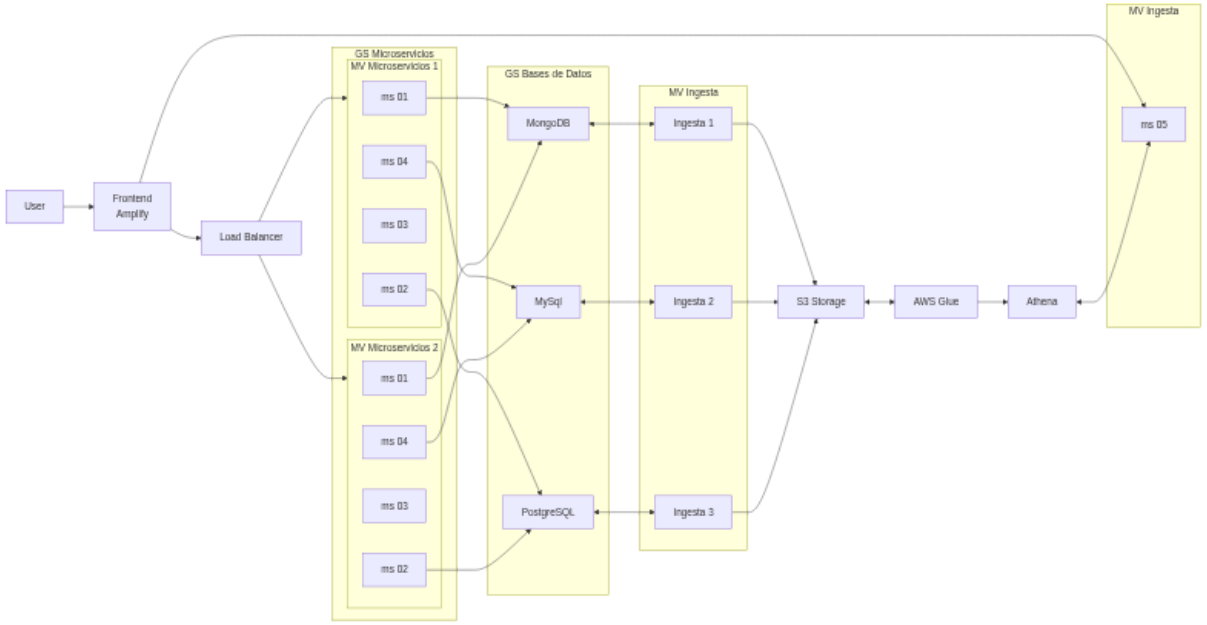
Repositorio:

warleon/cloud-computing-project

Arquitectura de los microservicios:



Arquitectura de la nube:



Microservicio 1

Propósito del Microservicio

El Microservicio 1 (MS1), denominado "Customer Service", es responsable de la gestión de clientes dentro de la arquitectura de la aplicación. Su objetivo principal es proporcionar una API RESTful para crear, leer, actualizar y eliminar información de clientes, así como validar y manejar la lógica de negocio relacionada con los mismos.

Tecnologías Utilizadas

- Node.js y TypeScript: Lenguaje principal y tipado estático para mayor robustez.
- Express.js: Framework para la creación de la API REST.
- MongoDB: Base de datos NoSQL para almacenamiento de datos de clientes.
- Mongoose: ODM para interactuar con MongoDB.
- Docker: Contenedorización del microservicio y sus dependencias.
- Swagger/OpenAPI: Documentación interactiva de la API.
- Postman: Colección para pruebas de endpoints.

Estructura del Proyecto

- `src/`: Código fuente principal.
  - `controllers/`: Lógica de controladores (CustomerController).
  - `models/`: Definición de esquemas de datos (Customer).
  - `routes/`: Definición de rutas de la API.
  - `services/`: Integración con servicios externos.
  - `middleware/`: Manejo de errores y seguridad.
  - `validators/`: Validación de datos de entrada.
  - `config/`: Configuración de la base de datos.
- `public/`: Documentación y recursos estáticos.
- `Dockerfile` y `docker-compose.yml`: Archivos para despliegue y orquestación.

Funcionamiento General

1. Recepción de Peticiones: El microservicio expone endpoints REST para operaciones CRUD sobre clientes.
2. Validación: Los datos recibidos son validados antes de ser procesados.
3. Lógica de Negocio: Los controladores gestionan la lógica y delegan en los servicios y modelos.
4. Persistencia: Los datos se almacenan y consultan en MongoDB.
5. Manejo de Errores: Middleware especializado captura y responde ante errores.
6. Documentación: La API está documentada y disponible mediante Swagger.

Endpoints Principales

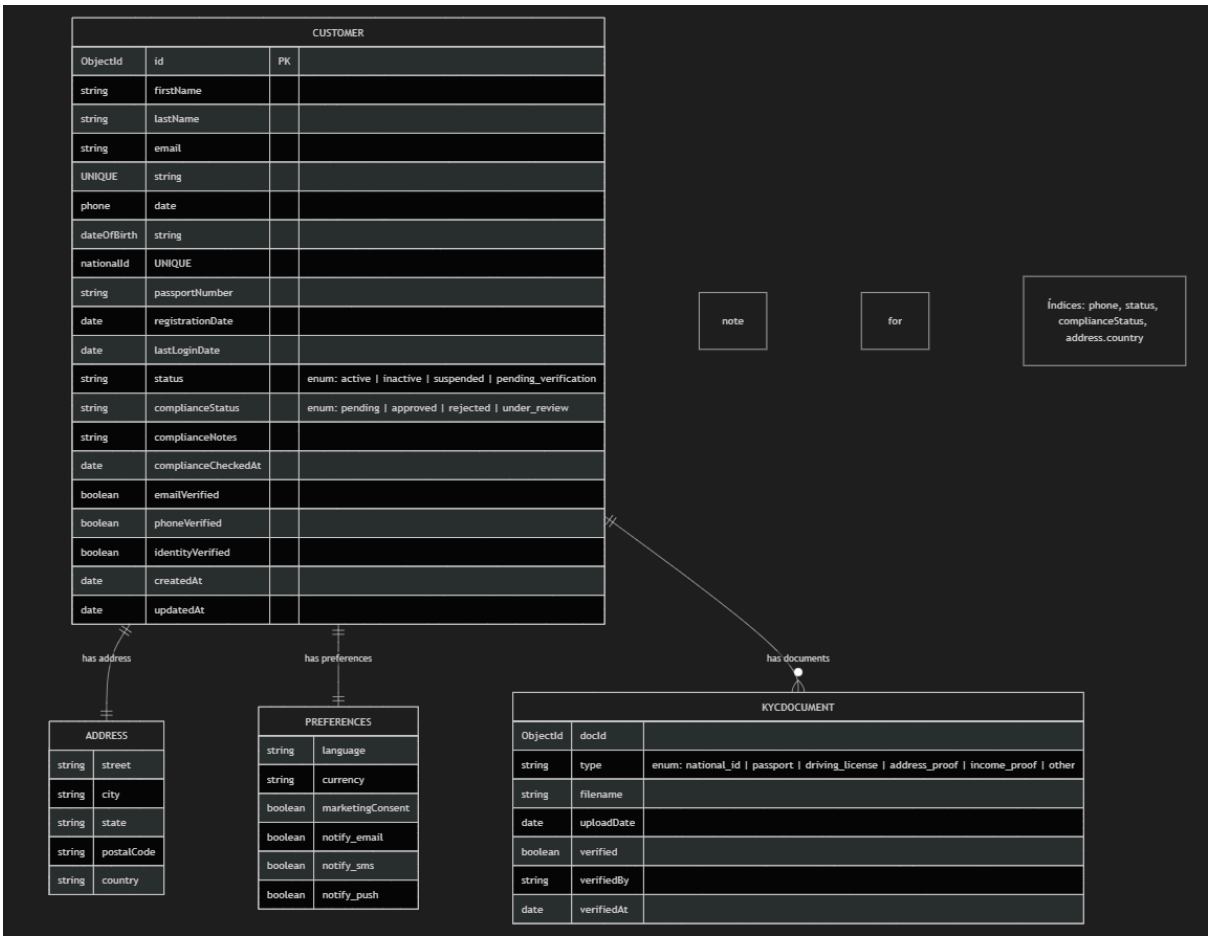
- `GET /customers`: Listar clientes
- `POST /customers`: Crear cliente
- `GET /customers/:id`: Obtener cliente por ID
- `PUT /customers/:id`: Actualizar cliente
- `DELETE /customers/:id`: Eliminar cliente

Seguridad y Validación

- Middleware para validación de datos y manejo de errores.
- Seguridad básica implementada en middleware.

Despliegue y Orquestación

- Uso de Docker y Docker Compose para facilitar el despliegue y la integración con MongoDB.



Conclusión

El Microservicio 1 es un componente fundamental para la gestión de clientes, implementado con tecnologías modernas y buenas prácticas de desarrollo, asegurando escalabilidad, mantenibilidad y facilidad de integración en arquitecturas basadas en microservicios.

Microservicio 2

- Objetivo general

Diseñar, implementar y desplegar un **microservicio de gestión de cuentas bancarias (Account Service)**, completamente desacoplado, seguro, escalable y capaz de interactuar con el microservicio de clientes (MS1) para validar identidades y operaciones.

- Objetivos específicos

1. Modelar y mantener la base de datos de cuentas, monedas, transacciones y tarifas.
2. Exponer una API RESTful documentada y accesible para otros servicios (MS3, MS4).
3. Validar la existencia de clientes mediante comunicación directa con MS1.
4. Implementar políticas de tolerancia a fallos y manejo de excepciones HTTP.
5. Contenerizar el servicio usando Docker y publicarlo en Docker Hub.
6. Garantizar un despliegue reproducible a través de Docker Compose y variables de entorno.

- Propósito y alcance del MS2

El **Account Service (MS2)** es responsable de gestionar toda la información y operaciones relacionadas con las **cuentas bancarias** de los usuarios.
Provee los endpoints necesarios para:

- Crear, consultar y cerrar cuentas.
- Asociar cuentas a clientes verificados en MS1.
- Registrar operaciones monetarias y estados de cuenta.
- Integrar transacciones y tarifas a través de entidades relacionadas.
- Servir como fuente de datos para MS3 (Transacciones) y MS4 (Compliance).

El microservicio se diseña para ser **autónomo**, con su propio modelo de datos, lógica de negocio y comunicación inter-servicio asíncrona.

- ## API REST DEL MICROSERVICIO

Descripción general

La API está construida con **FastAPI**, aprovechando la validación automática de datos, documentación OpenAPI/Swagger y asincronismo nativo.

Cada recurso se implementa mediante rutas (`routes/account_routes.py`) y modelos Pydantic (`models/account_model.py`).

Endpoints principales

Método	Endpoint	Descripción	Ejemplo de respuesta
`POST`	`/accounts/`	Crea una nueva cuenta asociada a un cliente validado por MS1	`{ "account_id": "ACC-001", "customer_id": "CUST-123", "currency": "PEN", "balance": 0 }`
`GET`	`/accounts/{account_id}`	Consulta información detallada de una cuenta específica	`{ "account_id": "ACC-001", "status": "ACTIVE" }`
`GET`	`/accounts/customer/{customer_id}`	Lista todas las cuentas de un cliente (consulta cruzada con MS1)	`[{"account_id": "ACC-001"}, {"account_id": "ACC-002"}]`
`PATCH`	`/accounts/{account_id}/close`	Cierra una cuenta activa (cambia estado a CLOSED)	`{ "message": "Account closed successfully" }`
`DELETE`	`/accounts/{account_id}`	Elimina una cuenta del sistema (solo si no hay transacciones asociadas)	`{ "status": "deleted" }`
`GET`	`/health`	Endpoint de verificación del servicio	`{ "status": "MS2 running", "timestamp": "2025-10-06T18:22Z" }`

- ## Documentación con Swagger

MS2 - Accounts Service

1.0.0 OAS 3.1

/openapi.json

Microservicio **MS2 - Accounts**
Parte del proyecto Cloud Computing (Banco).

Funcionalidades principales:

- **Gestión de cuentas bancarias**:
  - Crear cuentas nuevas asociadas a clientes.
  - Consultar información detallada de cuentas.
  - Actualizar el estado de las cuentas (ACTIVA, BLOQUEADA, CERRADA).
  - Obtener saldo disponible en la cuenta.
- **Ledger (movimientos contables)**:
  - Consultar el historial de movimientos (débitos y créditos).
- **Integración interna**:
  - Endpoint de transferencia, consumido por el MS3 (Transacciones), ejecutando débitos y créditos de manera atómica.

Swagger/OpenAPI disponible en `/docs`
Healthcheck disponible en `/healthz`

Cloud Computing Team - MS2 - Website

Accounts

POST `/accounts` Create Account

GET `/accounts` List Accounts

GET `/accounts/{account_id}` Get Account

GET `/accounts/{account_id}` Get Account

PUT `/accounts/{account_id}/status` Update Account Status

GET `/accounts/{account_id}/balance` Get Account Balance

Ledger

GET `/ledger/{account_id}` Get Ledger Entries

Internal

POST `/internal/transfer` Transfiere fondos entre dos cuentas (idempotente por requestId)

System

GET `/healthz` Health Check

GET `/` Root

Schemas

AccountBalanceOut › Expand all `object`

Schemas

AccountBalanceOut › Expand all `object`

AccountCreate › Expand all `object`

AccountOut › Expand all `object`

AccountUpdateStatus › Expand all `object`

HTTPValidationError › Expand all `object`

LedgerEntryOut › Expand all `object`

TransferRequest › Expand all `object`

TransferResponse › Expand all `object`

ValidationError › Expand all `object`

Ejemplo de flujo de validación con MS1

1. El cliente envía un `POST /accounts/` con `customer_id`.
2. MS2 llama al MS1 mediante `httpx.AsyncClient`:

```
response = await
_fetch_with_retries(f"{MS1_BASE_URL}/customers/{customer_id}")

if response.status_code != 200:

    raise HTTPException(status_code=404, detail="Customer not found
in MS1")
```

3. Si MS1 confirma la existencia del cliente, MS2 registra la cuenta localmente.
4. Devuelve respuesta JSON con los datos de la cuenta creada.

- INTEGRACIÓN CON MS1 (CUSTOMER SERVICE)

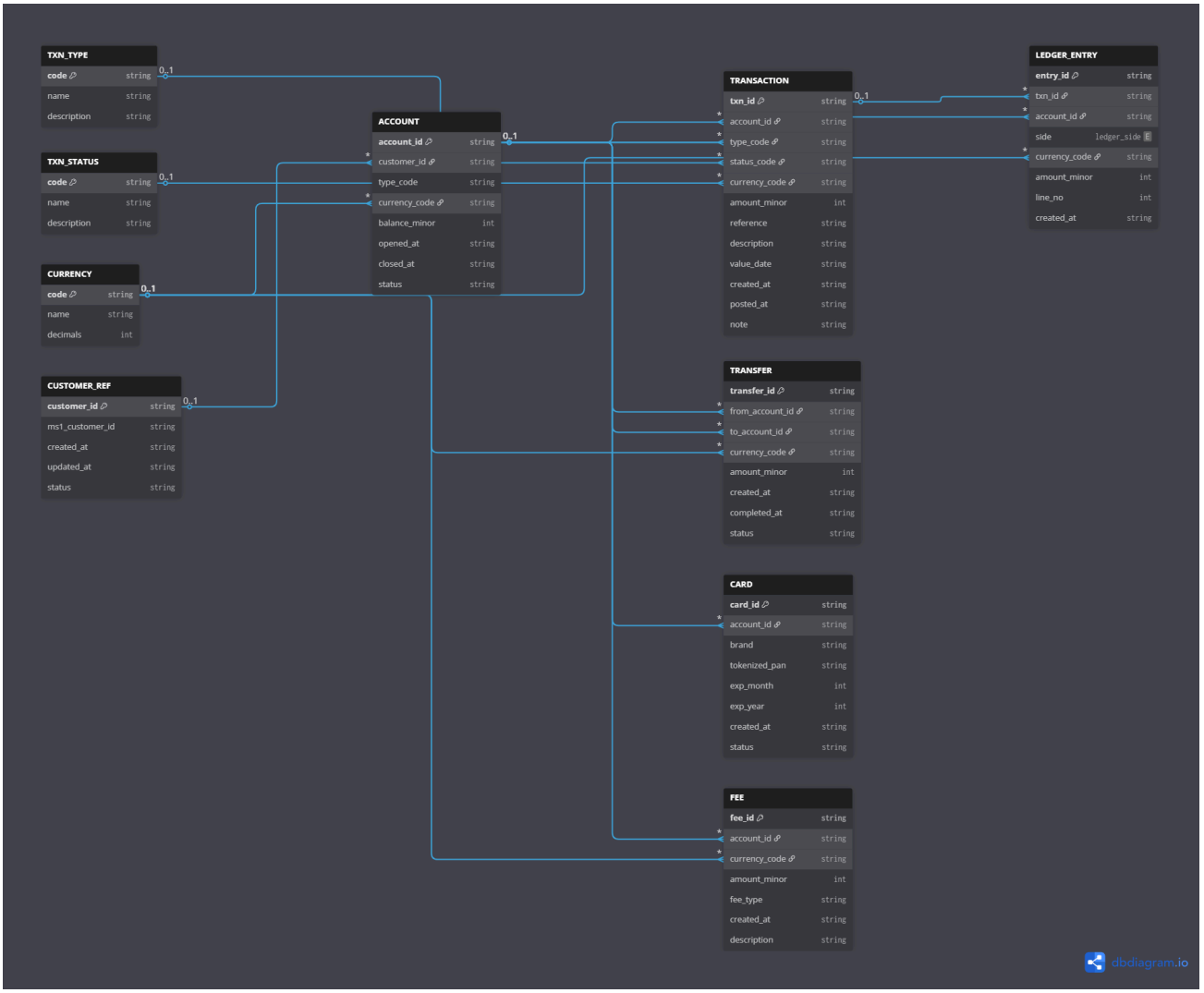
MS2 depende de **MS1** para validar que los `customer_id` sean válidos.
Esto se realiza mediante un **cliente HTTP reutilizable (`ms1_client.py`)** que implementa:

- Retries automáticos ante fallos.
- Control de tiempo de espera (`MS1_TIMEOUT`).
- Manejo de errores con `HTTPException`.
- Logs configurables (`LOG_LEVEL`).

Ejemplo:

```
MS1_BASE_URL = os.getenv("MS1_BASE_URL",
"http://ms1-customer-service:3000/api")

MS1_TIMEOUT = float(os.getenv("MS1_TIMEOUT", "3"))
```

- **MODELO ENTIDAD–RELACIÓN DEL MS2**



Entidad	Propósito / Descripción	PK	FK (→ tabla.campo)	Relaciones y cardinalidad
ACCOUNT	Núcleo del dominio de cuentas bancarias: tipo, moneda, balance y estado.	`account_id`	`customer_id` → CUSTOMER_REF.`customer_ref_id` • `type_code` → TXN_TYPE.`code` • `currency_code` → CURRENCY.`code`	CUSTOMER_REF **1:N** ACCOUNT • ACCOUNT **1:N** TRANSACTION • ACCOUNT **1:N** LEDGER_ENTRY • ACCOUNT **1:N** CARD • ACCOUNT **1:N** FEE • ACCOUNT **1:N** TRANSFER (como origen y como destino).
CUSTOMER_REF	Referencia local al cliente de MS1 (vínculo inter-microservicio).	`customer_ref_id`	—	CUSTOMER_REF **1:N** ACCOUNT.
CURRENCY	Catálogo de monedas soportadas y cantidad de decimales.	`code`	—	CURRENCY **1:N** ACCOUNT/TRANSACTION/TRANSFER/FEE/LEDGER_ENTRY.

TXN_TYPE	Catálogo de tipos de transacción (depósito, retiro, transferencia, comisión, etc.).	`code`	—	TXN_TYPE **1:N** TRANSACTION.
TXN_STATUS	Catálogo de estados de transacción (pending, posted, failed, etc.).	`code`	—	TXN_STATUS **1:N** TRANSACTION.
TRANSACTION	Movimiento monetario de una cuenta (monto, tipo, estado, fechas).	`txn_id`	`account_id` → ACCOUNT.`account_id` • `type_code` → TXN_TYPE.`code` • `status_code` → TXN_STATUS.`code` • `currency_code` → CURRENCY.`code`	ACCOUNT **1:N** TRANSACTION • TRANSACTION **1:N** LEDGER_ENTRY.
TRANSFER	Transferencia entre cuentas (origen/destino) con su moneda y estado.	`transfer_id`	`from_account_id` → ACCOUNT.`account_id` • `to_account_id` → ACCOUNT.`account_id` • `currency_code` → CURRENCY.`code`	ACCOUNT **1:N** TRANSFER (como origen) y ACCOUNT **1:N** TRANSFER (como destino).
CARD	Tarjetas asociadas a una cuenta (marca, token PAN, vencimiento).	`card_id`	`account_id` → ACCOUNT.`account_id`	ACCOUNT **1:N** CARD.
FEE	Comisiones/tarifas aplicadas a una cuenta (monto menor, tipo, descripción).	`fee_id`	`account_id` → ACCOUNT.`account_id` • `currency_code` → CURRENCY.`code`	ACCOUNT **1:N** FEE.
LEDGER_ENTRY	Asientos del libro mayor por transacción (doble partida por línea).	`entry_id`	`txn_id` → TRANSACTION.`txn_id` • `account_id` → ACCOUNT.`account_id` • `currency_code` → CURRENCY.`code`	TRANSACTION **1:N** LEDGER_ENTRY • ACCOUNT **1:N** LEDGER_ENTRY.

- TECNOLOGÍAS Y PROTOCOLOS

Lenguaje y Framework

- **Python 3.12**
- **FastAPI:** framework asíncrono que facilita la creación de APIs REST seguras, documentadas y de alto rendimiento.
- **HTTPX:** cliente asíncrono para comunicación con MS1.
- **Uvicorn:** servidor ASGI eficiente y liviano.

Protocolos empleados

Capa	Protocolo	Uso
Aplicación	HTTP/HTTPS	Comunicación inter-servicios
Serialización	JSON	Intercambio de datos estructurados
Autenticación	JWT / API Key	Validación de peticiones entre servicios
Transporte	TCP/IP	Conexión entre contenedores
Despliegue	Docker	Aislamiento y replicación de servicios

- ## Arquitectura de despliegue

El despliegue del MS2 sigue una arquitectura de tres niveles dentro del ecosistema del sistema bancario distribuido:

Componente	Función principal	Tecnología
MS2 (Account Service)	Expone la API REST para operaciones sobre cuentas.	Python 3.12 + FastAPI
PostgreSQL (ms2_postgres)	Base de datos relacional donde se almacena la información de cuentas, movimientos y clientes referenciados.	Postgres 16-alpine
MS1 (Customer Service)	Microservicio remoto utilizado por MS2 para validar la existencia del cliente antes de registrar una cuenta.	Node.js / Fastify (según arquitectura general)

Los servicios se comunican entre sí mediante la red virtual **ms_net**, definida dentro de Docker Compose, permitiendo la resolución de nombres de servicio (por ejemplo, `http://ms1:3000/api`).

Dockerfile del MS2

El siguiente **Dockerfile** define la construcción del contenedor del MS2:

```
FROM python:3.12-slim

ENV PYTHONUNBUFFERED=1 \
    PIP_NO_CACHE_DIR=1

WORKDIR /app
```

```
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY src/ ./src/

ENV APP_PORT=8080
EXPOSE 8080

CMD ["uvicorn", "src.main:app", "--host", "0.0.0.0", "--port",
"8080"]
```

Explicación técnica

1. **Base Image:**
   Se utiliza una imagen oficial y ligera (`python:3.12-slim`) para minimizar el tamaño final del contenedor y optimizar la carga.

2. **Variables de entorno:**

   - `PYTHONUNBUFFERED=1`: evita el almacenamiento en buffer del log, útil para el monitoreo.

   - `PIP_NO_CACHE_DIR=1`: impide que pip almacene cachés para reducir tamaño.

   - `APP_PORT=8080`: define el puerto interno de ejecución.

3. **Dependencias:**
   Se instalan desde `requirements.txt`, garantizando consistencia en entornos de desarrollo, pruebas y producción.

4. **Ejecución:**
   Se lanza el servidor con `uvicorn` para exponer la aplicación FastAPI en `0.0.0.0:8080`.

Docker-compose.yml

El archivo `docker-compose.yml` orquesta la ejecución de los servicios de MS2, PostgreSQL y la conexión con MS1:

```
services:
  ms2:
    build:
```

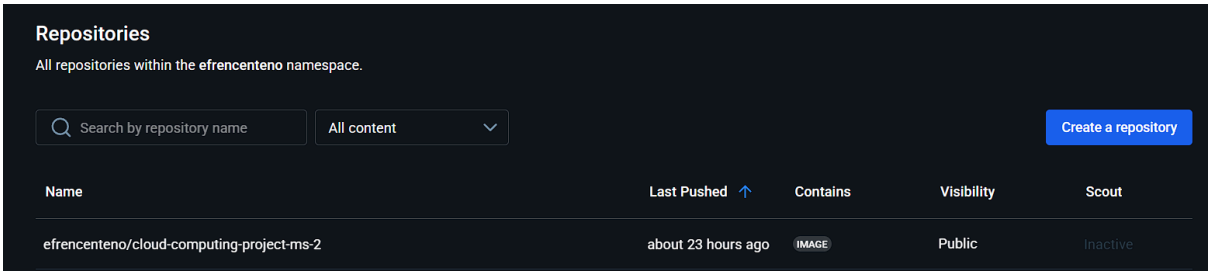
```
    context: ./cloud-computing-project-ms-2
    dockerfile: Dockerfile
  image: efrencenteno/cloud-computing-project-ms-2:latest
  container_name: ms2
  ports:
    - "5002:8080"
  environment:
    APP_PORT: "8080"
    PYTHONUNBUFFERED: "1"
    DB_HOST: "postgres"
    DB_PORT: "5432"
    DB_NAME: "ms2_accounts"
    DB_USER: "ms2"
    DB_PASSWORD: "secret"
    MS1_BASE_URL: "http://ms1:3000/api"
    MS1_VALIDATE: "true"
    MS1_TIMEOUT: "3"
    SERVICE_KEY_FOR_MS3: "changeme"
    ALLOWED_ORIGINS: "http://localhost:3000,http://34.192.101.95"
    LOG_LEVEL: "info"
  depends_on:
    - postgres
    - ms1
  networks:
    - ms_net
  restart: unless-stopped
  profiles: ["loadBalanced", "ms2"]

postgres:
  image: postgres:16-alpine
  container_name: ms2_postgres
  restart: unless-stopped
  environment:
    POSTGRES_DB: ms2_accounts
    POSTGRES_USER: ms2
    POSTGRES_PASSWORD: secret
  ports:
    - "5432:5432"
  volumes:
    - postgres_data:/var/lib/postgresql/data
  networks:
```

```
      - ms_net

networks:
  ms_net:

volumes:
  postgres_data:
```

Puntos técnicos importantes

Aspecto	Descripción y justificación
Puerto	El contenedor expone el puerto 8080 y se publica como 5002 en el host (5002:8080).
DB_HOST	Debe usar el nombre del servicio (ms2_`postgres`) en lugar del container_name.
MS1_BASE_URL	La URL apunta al nombre del servicio `ms1` para la resolución DNS interna en la red Docker.
depends_on	Asegura el orden de inicio de PostgreSQL y MS1 antes de que MS2 intente conectarse.
Red `ms_net`	Facilita la comunicación entre todos los microservicios del sistema bancario.
Volumen `postgres_data`	Garantiza persistencia de datos entre reinicios del contenedor.

La imagen de dicho microservicio publicada exitosamente en docker hub :



- Flujo de funcionamiento principal

1. **Recepción de solicitudes:**
   Los clientes o microservicios externos (como MS3) envían peticiones HTTP a la API

de MS2 en `http://localhost:5002`.

2. **Validación de cliente (MS1):**
   Cuando se intenta crear una nueva cuenta, MS2 consulta a MS1 (`/api/customers/{id}`) para confirmar la existencia del cliente.
   Si MS1 devuelve un error, la creación de cuenta se rechaza.

3. **Registro en base de datos (PostgreSQL):**
   Una vez validado, MS2 persiste la cuenta y su información asociada en la base de datos `ms2_accounts`.

4. **Gestión de transacciones:**
   Las entidades `TRANSACTION`, `TRANSFER`, `FEE` y `LEDGER_ENTRY` permiten registrar depósitos, transferencias, comisiones y asientos contables.

5. **Respuestas JSON:**
   Todas las operaciones responden en formato JSON estandarizado con códigos HTTP adecuados (`201 Created`, `404 Not Found`, `500 Internal Server Error`).

6. **Logging y observabilidad:**
   La variable `LOG_LEVEL` controla el nivel de detalle de los logs.
   Además, `PYTHONUNBUFFERED=1` garantiza la salida inmediata a consola, útil para monitoreo con Docker logs o herramientas externas.

- Conclusiones

1. **Integración efectiva:**
   MS2 se integra exitosamente con MS1, gestionando la validación de clientes de forma remota y segura.

2. **Despliegue estandarizado:**
   Gracias a Docker y Docker Compose, el microservicio puede levantarse en segundos con su entorno completo (API + Base de datos).

3. **Diseño limpio y extensible:**
   La estructura modular (`src/main.py`, `services/`, `models/`, `routes/`) y la documentación OpenAPI facilitan futuras ampliaciones.

4. **Cumplimiento de buenas prácticas:**
   El uso de entornos virtuales, variables de entorno, logs parametrizables y una base relacional sólida aseguran mantenibilidad y robustez.

5. **Publicación profesional:**
   La imagen pública en Docker Hub (`efrencenteno/cloud-computing-project-ms-2:latest`) garantiza accesibilidad, versionado y colaboración dentro del equipo.

Microservicio 3

Propósito y rol en la arquitectura

El microservicio de Transacciones (ms-3) actúa como el **orquestador central** para todas las operaciones de transferencia de dinero dentro del sistema. Su responsabilidad principal no es almacenar datos de cuentas ni saldos, sino garantizar que una transferencia monetaria se complete de manera segura y consistente, coordinando las acciones entre otros microservicios especializados.

Para lograr esto, implementamos el **patrón de diseño SAGA**, que permite gestionar transacciones que abarcan múltiples servicios. Este enfoque asegura que, si un paso del proceso falla, se puedan tomar acciones compensatorias para mantener la integridad del sistema.

Tecnologías utilizadas

El microservicio está construido sobre un stack moderno de JavaScript/TypeScript:

- Entorno de Ejecución: Node.js (v20)
- Framework Web: Express.js
- Lenguaje: TypeScript
- Documentación de API: Swagger (con swagger-jsdoc y swagger-ui-express)
- Pruebas (testing): Jest y Supertest
- Contenerización: Docker
- Variables de entorno: dotenv

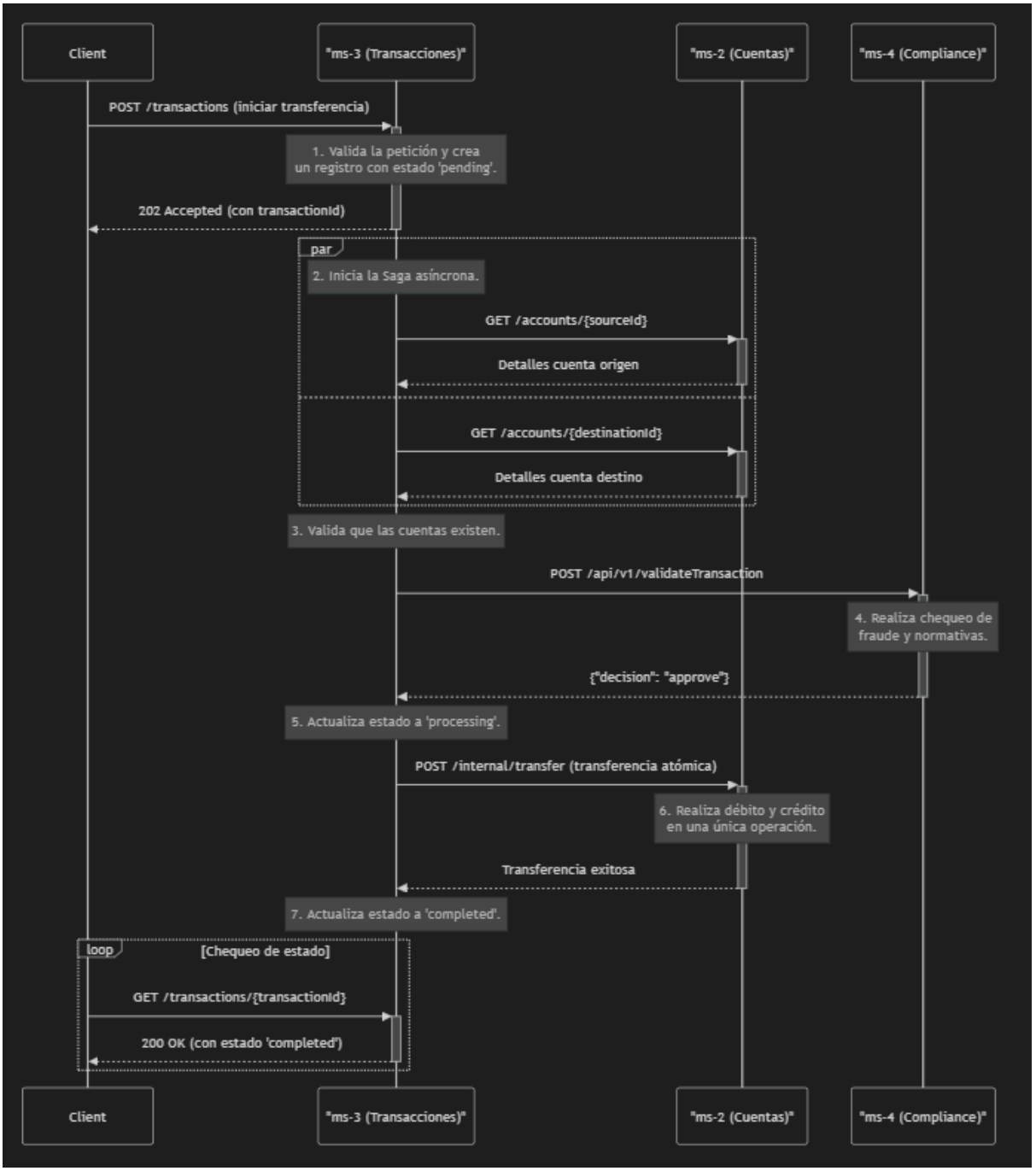
Funcionalidades y endpoints de la API

La API expone los siguientes endpoints para gestionar las transacciones:

- POST / transactions
    - Descripción: Inicia una nueva transferencia de dinero de forma asíncrona. El servicio válida la petición, crea un registro de la transferencia con estado ‘pending’ y responde inmediatamente con un ‘202 accepted’. El procesamiento real (validaciones, compliance, débito/crédito) ocurre en segundo plano.
    - Cuerpo: TranferRequestBody
    - Respuesta exitosa: 202 Accepted con el ID de la transacción.
- GET /transactions/:transactionId
    - Descripción: Permite consultar el estado y los detalles de una transacción específica en cualquier momento. Es el mecanismo

principal para que un cliente sepa si una operación asíncrona se completó, falló o sigue en proceso.
    - Parámetros: TransactionID (en la URL).
    - Respuesta exitosa: 200 ok con el registro completo de la transacción.
- GET /accounts/:accountId/transactions
    - Descripción: Devuelve una lista paginada de todas las transacciones (tanto enviadas como recibidas) asociadas a una cuenta.
    - Parámetros: accountId (en la URL), limit y offset (opcionales, en la query).
    - Respuesta exitosa: 200 ok con una lista de transacciones.
- GET /accounts/:accountId/balance (helper)
    - Descripción: un endpoint de ayuda para depuración y prueba. Consulta directamente al microservicio de Cuentas (ms-2) para obtener el saldo actual de una cuenta.
    - Parámetros: accountID (en la URL).
    - Respuesta exitosa: 200 ok con el objeto Money que representa el saldo.

Diagrama de flujo de la API



Seguridad implementada

La seguridad es un aspecto clave, especialmente al tratarse de operaciones financieras.

- Autenticación de Servicio a Servicio: La comunicación con endpoints internos críticos, como `/internal/transfer` en ms-2, está protegida. ms-3 debe presentar una clave secreta ('x-service-key') en la cabecera de la petición, la cual es validada por ms-2. Esta clave se gestiona de forma segura a través de variables de entorno.

- Validación de Entradas (Input Validation): Todos los endpoints validan rigurosamente los datos de entrada. Se rechazan peticiones con campos faltantes, tipos de datos incorrectos o lógica de negocio inválida (ej: transferir a la misma cuenta).
- Manejo de Errores: El sistema está diseñado para fallar de forma segura. Si una llamada a un microservicio dependiente falla (ej: por un error de red o porque una cuenta no existe), la saga se detiene y el estado de la transacción se marca como 'failed', evitando dejar el sistema en un estado inconsistente.

Estructura del microservicio

La organización de la carpeta `ms-3` sigue un patrón claro y escalable:

```
ms-3/
├── dist/               # Código JavaScript compilado (para producción)
├── src/                # Código fuente en TypeScript
│   ├── mocks/          # (No presente, pero usado en tests) Mocks para simular otros servicios.
│   ├── index.ts        # Archivo principal: servidor Express, rutas y lógica de la Saga.
│   ├── swagger.ts      # Configuración centralizada de Swagger para la documentación.
│   └── types.ts        # Definiciones de interfaces y tipos de TypeScript (ej: Money).
├── tests/              # Pruebas automatizadas
│   └── ms3.smoke.test.ts # Pruebas de humo que validan los flujos principales.
├── .env                # (No versionado) Variables de entorno (claves, URLs de servicios).
├── Dockerfile          # Define cómo construir la imagen Docker del microservicio.
├── package.json        # Dependencias y scripts del proyecto.
└── tsconfig.json       # Configuración del compilador de TypeScript.
```

propósito de los archivos clave:

- src/index.ts: Es el corazón de la aplicación. Define el servidor Express, registra los endpoints de la API y contiene la lógica de orquestación del patrón Saga.
- src/swagger.ts: Abstrae toda la configuración de la documentación de la API, manteniendo `index.ts` enfocado en la lógica de negocio.
- tests/ms3.smoke.test.ts: Contiene pruebas de integración cruciales que simulan el comportamiento completo de una transacción, incluyendo la verificación de saldos antes y después, asegurando que la lógica asíncrona funciona como se espera.
- Dockerfile: Utiliza una construcción multi-etapa (multi-stage build) para crear una imagen de Docker optimizada, ligera y segura para producción.

Consideraciones adicionales

Persistencia de datos:
Actualmente, el estado de las transacciones se almacena en una estructura de

datos en memoria (`Map`). Esto es adecuado para desarrollo y pruebas, pero NO ES PERSISTENTE y se perderá si el servicio se reinicia.

Para un entorno de producción, este `Map` debería ser reemplazado por una solución de base de datos persistente, como:

- Redis: Para un almacenamiento rápido de clave-valor.
- MongoDB o PostgreSQL: Si se requiere una mayor capacidad de consulta o durabilidad.

Manejo de Errores y compensación:
El patrón Saga se implementa con bloques `try...catch` en el flujo asíncrono.

- Fallo en Validación: Si una cuenta no existe o el chequeo de compliance falla, la saga se detiene y el estado de la transacción se marca como 'failed'. No se requiere compensación porque aún no se ha realizado ninguna modificación monetaria.
- -Fallo en Transferencia: El endpoint `/internal/transfer` de ms-2 se asume que es ATÓMICO. Si falla, no deja saldos inconsistentes. Por lo tanto, ms-3 simplemente marca la transacción como 'failed'. En un escenario más complejo donde la transferencia no fuera atómica, ms-3 sería responsable de invocar operaciones de compensación (ej: revertir un débito si el crédito falla).

Estrategia de Pruebas:
El archivo `tests/ms3.smoke.test.ts` implementa pruebas de humo (smoke tests) que validan los flujos más críticos:

- Transacción exitosa: Verifica que el dinero se transfiera correctamente de una cuenta a otra.
- Rechazo por Compliance: Asegura que una transacción que no cumple las reglas es marcada como 'failed' y los saldos no cambian.
- Fallo y Reversión: Simula un fallo durante la transferencia y confirma que los saldos permanecen intactos, demostrando la robustez del sistema.

Estas pruebas son fundamentales para garantizar la fiabilidad del orquestador.

Microservicio 4, Compliance

Propósito

MS4 es el servicio de cumplimiento (Compliance & Risk) encargado de aplicar reglas (reglas de negocio / políticas de riesgo) sobre transacciones y mantener reglas configurables en la base de datos. Sus responsabilidades principales son:

- Validar transacciones entrantes y devolver una decisión (aprobada / rechazada) con motivo.
- CRUD (crear, leer, listar, actualizar, eliminar) de reglas de cumplimiento almacenadas en MySQL.
- Exponer documentación Swagger para explorar la API.

El servicio se utiliza como un punto central de decisión antes de procesar o autorizar transacciones en otros componentes del sistema.

Tecnologías y dependencias

- Lenguaje: Go.
- Framework HTTP: Gin (router y middleware).
- ORM: GORM para acceso a MySQL.
- Documentación: Swag / Swagger (swaggo/gin-swagger).
- Configuración: variables de entorno.
- Logging: Logrus.

Estructura relevante (archivos clave)

- internal/main.go — punto de entrada: carga configuración, inicializa DB y rutas, registra handlers y Swagger.
- internal/handlers/compliance.go — handlers HTTP (endpoints): ValidateTransaction y CRUD de rules.
- internal/service/compliance.go — lógica de negocio: evaluación de reglas y orquestación de consultas al repositorio.
- internal/repository/mysql_repository.go — implementación del repositorio contra MySQL (CRUD de reglas, consultas por tipo de regla, comprobación de listas de sanciones, audit logs).
- internal/repository/rules/ — modelos y tipos de reglas (p. ej. AmountThresholdRule, Sanction/Blacklist y base de reglas).

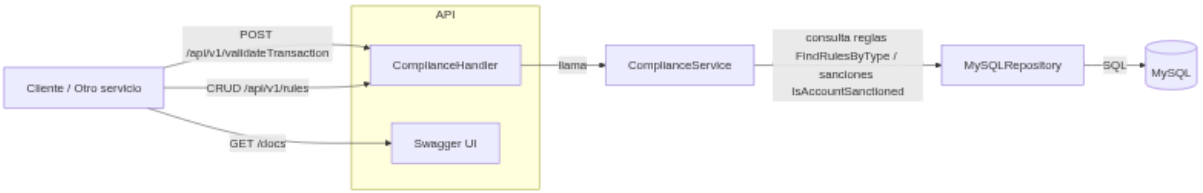
Endpoints expuestos

- POST /api/v1/validateTransaction
    - Entrada: payload JSON con datos de la transacción (dto.Transaction).
    - Salida: Decision (aprobada / no aprobada y razón).
- POST /api/v1/rules
- GET /api/v1/rules
- GET /api/v1/rules/:id
- PUT /api/v1/rules/:id
- DELETE /api/v1/rules/:id
- GET /docs/*any — Swagger UI

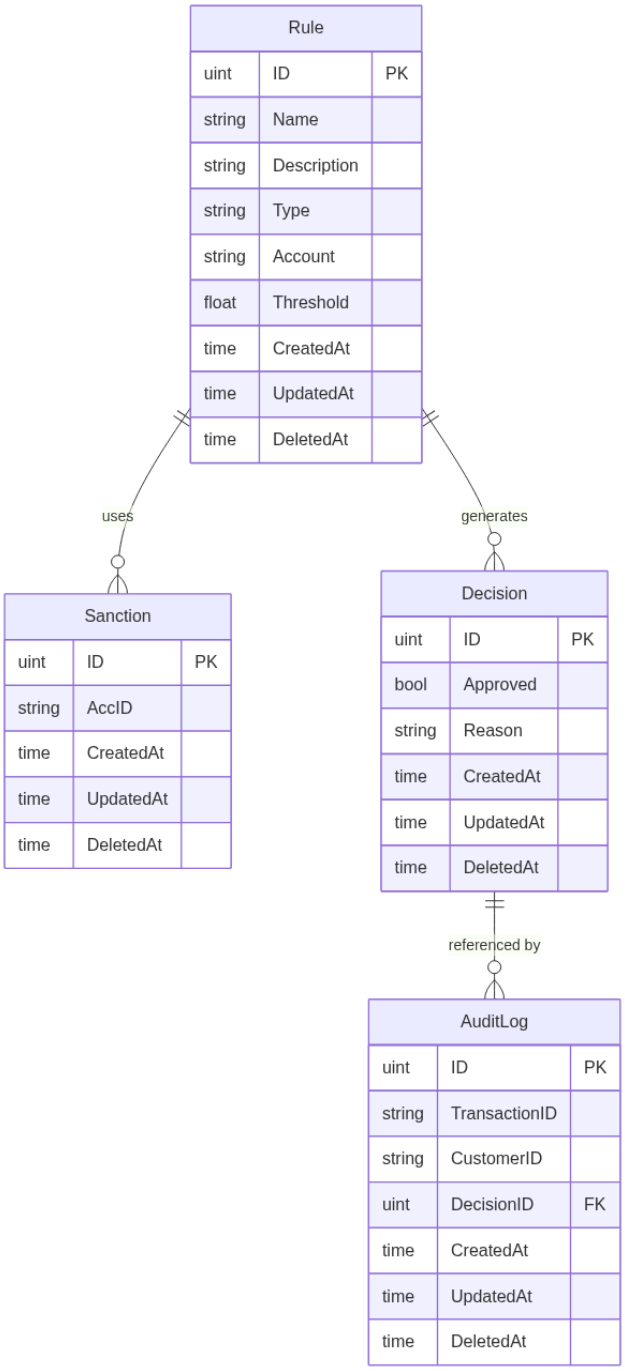
Nota: la API está montada bajo el prefijo /api/v1 (ver internal/main.go).

Diagrama

El siguiente diagrama muestra el flujo desde el cliente hasta la base de datos y dónde intervienen los distintos componentes:



El siguiente diagrama muestra el flujo el diseño de la base de datos, entidades y relaciones:



Cómo funciona (flujo de ValidateTransaction)

1. El handler ValidateTransaction hace binding del JSON entrante a dto.Transaction y delega a service.ValidateTransaction.
2. El servicio consulta al repositorio las reglas del tipo AmountThreshold (FindRulesByType).
3. Cada regla de tipo umbral se convierte en una implementación concreta (AmountThresholdRule) y se valida contra la transacción. Si alguna regla niega la transacción, se devuelve una Decision negativa inmediatamente.
4. Si pasa los umbrales, el servicio consulta si la cuenta origen o destino está sancionada (IsAccountSanctioned). Si alguna lo está, devuelve Decision{Approved:false} con motivo.
5. Si todas las comprobaciones pasan, devuelve Decision{Approved:true, Reason:"OK"}.

Modelos y DTOs

- dto.Transaction — (campos inferidos por uso en el código):
  - FromAcc (string) — cuenta origen
  - ToAcc (string) — cuenta destino
  - Amount (número) — importe de la transacción
  - (puede contener más campos como Currency, TxnID, Timestamp)
- rules.Rule — modelo persistente con campos comunes (ID, Type, metadata, opcional Threshold para reglas de cantidad, etc.).
- rules.Decision — resultado de la evaluación: {Approved bool, Reason string}.

Persistencia y migraciones

- El servicio usa GORM y ejecuta db.AutoMigrate(...) sobre las tablas listadas en repository.RepositoryTables y rules.RuleTables al inicio (internal/main.go).
- Tablas principales: reglas (rules.Rule), sanciones (rules.Sanction), y AuditLog.

Consideraciones de diseño y casos borde

- Las reglas se evalúan en un orden lógico: primero umbrales, luego sanciones. Esto permite cortar temprano cuando una regla niega.
- Las reglas mal formadas (p. ej. sin Threshold) se ignoran en tiempo de evaluación.
- La comprobación de sanciones está delegada al repositorio para aprovechar índices y consultas directas en la tabla de sanciones.
- Errores del repositorio devuelven 500 al cliente; se podría enriquecer con códigos y trazabilidad.

Casos borde a considerar para producción:

- Transacciones muy grandes / concurrentes → revisar locks y consultas optimizadas en IsAccountSanctioned.
- Latencia en DB → aplicar cache para listas de sanciones o usar un servicio de cache/fast lookup.
- Gestión de versiones de reglas → añadir versionado o fecha de vigencia en rules.Rule.

Microservicio 5