Neobank – Guía de Onboarding y Arquitectura

Onboarding: Introducción al Proyecto

Neobank es una plataforma de banca digital implementada en Go, diseñada para gestionar cuentas financieras de clientes y sus transacciones de forma segura y escalable. El sistema está dividido en múltiples microservicios especializados, lo que facilita su mantenimiento y escalabilidad. A grandes rasgos, Neobank permite registrar clientes, verificar su identidad (KYC), abrir cuentas contables para ellos y procesar transferencias de dinero peer-to-peer (P2P) entre usuarios.

Propósito del proyecto: brindar una solución básica de banca digital que sea modular y extensible. A continuación, se resumen los objetivos generales del sistema:

- **Experiencia de onboarding simple:** Facilitar el registro de nuevos clientes, incluyendo la recopilación de datos y documentos para cumplir con políticas KYC, de forma ágil y comprensible.

- **Cumplimiento y seguridad:** Asegurar la verificación de identidad (Know Your Customer) y validaciones reglamentarias antes de habilitar funcionalidades financieras, protegiendo al mismo tiempo la confidencialidad de los datos.

- **Gestión confiable de cuentas y balances:** Crear y administrar cuentas contables para cada cliente, llevando un libro mayor (ledger) confiable que garantice la integridad de los saldos y transacciones.

- **Transferencias P2P eficientes:** Permitir a los usuarios enviar y recibir fondos al instante entre sí, con garantías de consistencia (cada transferencia se registra con doble partida contable) e idempotencia (evitar duplicados).

- **Arquitectura escalable y modular:** Emplear microservicios desacoplados que puedan evolucionar independientemente, comunicándose mediante APIs eficientes (gRPC) y eventos, lo cual sienta las bases para escalar el sistema y añadir nuevas funcionalidades fácilmente en el futuro.

En resumen, Neobank busca que cualquier desarrollador o equipo pueda entender rápidamente el dominio bancario que cubre, a la vez que provee una base técnica sólida para construir aplicaciones financieras confiables.

Flujo Funcional General del Sistema

El sistema abarca varios flujos funcionales clave que involucran a los diferentes servicios. A continuación, se describe de forma simplificada el recorrido típico desde que un cliente se registra hasta que realiza una transferencia P2P:

1. **Onboarding de clientes:** Un nuevo usuario se registra proporcionando sus datos personales básicos y, de ser necesario, documentación de identidad. El `account-service` almacena estos datos y marca al usuario como pendiente de verificación. Este paso inicial crea una entidad de cliente en la base de datos de usuarios (con estado `registrado, pendiente` KYC).

2. **Validaciones KYC:** Luego del registro, se activa el proceso KYC (Know Your Customer, "Conoce a tu Cliente"), donde se verifica la identidad y la información del usuario. El `account-service` coordina esta validación —revisión de documentos, comprobación contra listas de bloqueo y otros controles reglamentarios—. Hasta no completar el KYC, el cliente no puede usar funcionalidades financieras. Una vez aprobado, su estado cambia a `verificado` y queda habilitado para operar.

3. **Creación de cuentas contables:** Al confirmar que el cliente superó KYC, el `account-service` llama vía gRPC a `CreateAccount` en el `ledger-service`, enviando el identificador del cliente y la moneda base.

    - El `ledger-service` crea un registro de cuenta contable en su base de datos con saldo inicial cero y devuelve confirmación al `account-service`.

    - Inmediatamente después, el `ledger-service` emite un evento Kafka `AccountCreated`, que podrán consumir servicios de notificaciones, analítica u otros interesados.

    - Cada cuenta contable tiene un identificador único; un cliente puede tener múltiples cuentas (por ejemplo, para varias monedas o productos) creadas de forma independiente.

4. **Flujo de transferencias P2P:** Con el cliente verificado y su cuenta contable activa, puede enviar y recibir transferencias internas. Cuando un usuario inicia una transferencia P2P:

    a. El `payments-service` valida que remitente y destinatario existan y estén verificados.

    b. Consulta vía gRPC el saldo disponible con `GetBalance` y comprueba fondos suficientes.

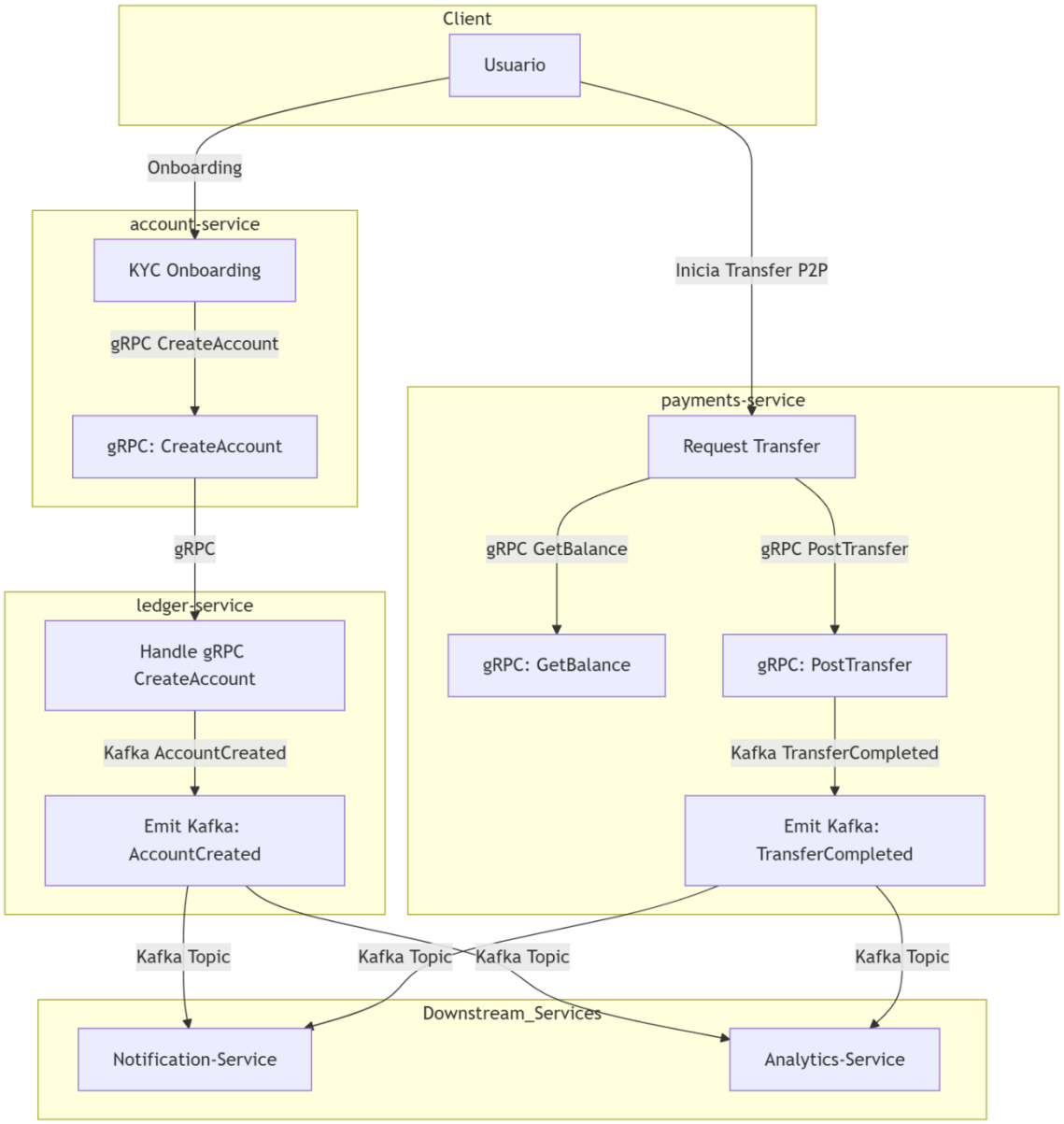
    c. Genera un `externalID` para garantizar idempotencia y llama vía gRPC `PostTransfer` en el `ledger-service`.

    d. El `ledger-service` ejecuta la operación de doble entrada (débito al remitente y crédito al beneficiario) dentro de una misma transacción atómica, verifica balance neto cero y devuelve el resultado.

    e. Tras el éxito, el `ledger-service` emite un evento Kafka `TransferCompleted`.

    f. El `payments-service` informa al usuario origen y destino del estado de la

transferencia; en caso de error (fondos insuficientes, cuentas inexistentes, etc.), la operación se aborta sin afectar balances y se notifica el fallo.



Arquitectura Basada en Microservicios

El sistema está construido bajo una arquitectura de microservicios, donde cada componente se enfoca en un dominio específico de la lógica de negocio. Esto implica que diferentes aspectos (clientes, cuentas contables, pagos, etc.) son manejados por servicios independientes que se comunican entre sí cuando es necesario. A continuación, se presentan los principales componentes y sus responsabilidades, así como la forma en que interactúan:

- **Account-Service (Servicio de Cuentas y Clientes):** El `account-service` es el microservicio encargado de la gestión de clientes y sus cuentas a nivel de negocio. Sus responsabilidades incluyen: el registro de nuevos usuarios, el manejo del flujo de onboarding, y la coordinación del proceso de KYC. Este servicio mantiene la información del cliente (perfil, documentos, estado de verificación) en su propia base de datos. Una vez que un cliente es verificado, el `account-service` se encarga de solicitar la creación de la cuenta contable correspondiente para el cliente – para ello invoca al `ledger-service` mediante una llamada gRPC `CreateAccount`. Posteriormente, el `ledger-service` crea el registro contable, devuelve la confirmación y emite el evento Kafka `AccountCreated` para notificaciones y procesos downstream. Además, es probable que el `account-service` exponga APIs hacia el exterior (por ejemplo, endpoints HTTP/REST) que permitan a aplicaciones cliente (móvil o web) interactuar con el sistema: crear usuarios, consultar su estado, iniciar transferencias, etc. De esta manera, actúa en parte como fachada o puerta de entrada al ecosistema de microservicios, orquestando las llamadas internas necesarias. (En una implementación robusta, a veces se podría utilizar un API Gateway dedicado, pero para simplificar asumimos que el `account-service` cumple ese rol de exposición de servicios al cliente final).

- **Ledger-Service (Servicio de Libro Mayor):** El `ledger-service` es el corazón contable de Neobank: expone una API gRPC (normalmente en el puerto `50051`) para que el `account-service`, el `payments-service` u otros consumidores soliciten operaciones sobre el libro mayor. Sus tareas clave incluyen crear cuentas contables a petición del `account-service` vía gRPC, registrar transferencias mediante el principio de doble partida (débito y crédito en la misma transacción atómica) y ofrecer consultas de saldo por cuenta. Internamente gestiona reglas críticas como idempotencia (usando un `externalID` para evitar duplicados) y la garantía de balance cero en cada movimiento, todo ello respaldado por su propia base de datos PostgreSQL (esquema `ledger`) con tablas de cuentas, transferencias y asientos. Además de atender las llamadas sincrónicas, el `ledger-service` juega un rol en el flujo asíncrono de notificaciones y procesos downstream: tras confirmar internamente la creación de una cuenta o el éxito de una transferencia, publica en Kafka los eventos `AccountCreated` o `TransferCompleted`. De este modo, servicios de notificaciones, analítica y otros módulos pueden reaccionar sin acoplarse directamente, mientras que ningún otro componente accede de forma directa a la base de datos contable, preservando la integridad y la trazabilidad del sistema.

- **Payments-Service (Servicio de Pagos P2P):** El `payments-service` orquesta las transferencias P2P dentro de Neobank, actuando como intermediario lógico entre el cliente y el `ledger-service`. Cuando recibe una solicitud de pago valida que remitente y destinatario existan y estén verificados, luego consulta **vía gRPC** `GetBalanceCents` en el `ledger-service` para asegurar fondos suficientes y genera un `externalID` de idempotencia. A continuación, invoca **vía gRPC** `PostTransfer`, donde el `ledger-service` aplica la doble partida (débito y crédito) en una única transacción atómica y devuelve el resultado. Aunque el `payments-service` aplica reglas de negocio adicionales (límites de monto, reintentos, reversos y notificaciones inmediatas al usuario), **no** emite él mismo el evento de finalización. Tras completar la transferencia, el `ledger-service` publica en Kafka el evento `TransferCompleted`, que es consumido por los sistemas de notificaciones, analítica y otros procesos asíncronos. De este modo mantenemos nuestro patrón híbrido: gRPC para la ejecución crítica y Kafka para la difusión de eventos.

Comunicación e Integración entre Servicios

Los microservicios de Neobank se comunican entre sí de dos formas principales:

- **Llamadas gRPC (sincrónicas):** Para operaciones directas y consultas inmediatas, se utilizan Remote Procedure Calls sobre gRPC. Por ejemplo, el `account-service` llama al método remoto `CreateAccount` del `ledger-service` para crear una nueva cuenta contable al instante, o el `payments-service` invoca `GetBalanceCents` y `PostTransfer` en el `ledger-service` para ejecutar una transferencia. Estas llamadas son sincrónicas: el emisor espera la respuesta del receptor para continuar, lo cual es adecuado cuando se necesita confirmar una acción en tiempo real (p. ej., informar al usuario que su transferencia fue realizada con éxito). Los mensajes y datos se serializan usando Protocol Buffers, garantizando interfaces bien definidas y tipado estricto entre los servicios.

- **Eventos asíncronos (Kafka):** Para desacoplar procesos y lograr mayor flexibilidad, el `ledger-service` publica eventos en Kafka una vez que completa internamente una operación crítica. Por ejemplo:

    - Tras crear la cuenta contable, emite `AccountCreated`.

    - Tras procesar una transferencia, emite `TransferCompleted`.

    Otros servicios (notificaciones, analítica, etc.) se suscriben a esos tópicos para enviar recibos, generar métricas, etc., todo ello sin impactar el flujo principal de la operación. Gracias a Kafka, el envío y recepción de mensajes es tolerante a fallos temporales y facilita escalar cada componente según la carga de eventos.

En combinación, gRPC y Kafka aseguran una arquitectura híbrida: consistente y rápida para las operaciones en línea, y a la vez flexible y extensible mediante notificaciones asíncronas.

Detalles Técnicos y Principios de Diseño

A continuación, se presentan los aspectos técnicos clave de la implementación y los principios de diseño que guían Neobank:

- **Lenguaje Go**
    - Todos los microservicios están escritos en Go por su rendimiento, modelo de concurrencia ligero (goroutines) y compilación estática.
    - Go ofrece soporte nativo para construir servidores gRPC de alta velocidad y bajo consumo de recursos.
- **gRPC y Protocol Buffers**
    - La comunicación interna se basa en Protocol Buffers (`.proto`) y gRPC, garantizando:
        - Interfaces tipadas y bien definidas por servicio.
        - Generación automática de stubs (cliente/servidor) en Go.
        - Latencia baja y serialización compacta.
    - Métodos habituales:
        - `CreateAccount` (account → ledger)
        - `GetBalanceCents` (payments → ledger)
        - `PostTransfer` (payments → ledger)
- **PostgreSQL por Servicio**
    - Cada microservicio dispone de su propia base de datos PostgreSQL, siguiendo el patrón *database-per-service*.
    - Ventajas: consistencia transaccional aislada, integridad referencial y escalado independiente.
    - Conexión configurable vía variables de entorno; los esquemas (p. ej. `ledger`) versionados con migraciones SQL.
- **Kafka**
    - Apache Kafka actúa como bus de eventos para flujos asíncronos y notificaciones:

        - Tópicos como `AccountCreated` y `TransferCompleted`.
        - Múltiples consumidores (notificaciones, analítica, auditoría) sin acoplamiento directo.
        - Alta tolerancia a fallos, persistencia de mensajes y escalabilidad horizontal.

- **Docker y Contenedores**
    - Cada servicio se empaqueta en una imagen Docker ligera, facilitando despliegues locales y en producción.
    - Orquestación posible vía Docker Compose (desarrollo) o Kubernetes (producción).
- **Observabilidad y Logging**
    - Se integra Prometheus para métricas de desempeño y Grafana para dashboards.
    - Logging estructurado con Zap (uber-go/zap), incluyendo trazas de request-id para correlación de llamadas gRPC y eventos.
- **Librerías y utilidades**
    - Uso de paquetes Go estándar y librerías especializadas para mantener el código ligero:
        - Zap para logging de alta velocidad.
        - pgx o lib/pq como driver de PostgreSQL.
        - Migrator (por ejemplo, golang-migrate) para gestionar versiones de esquema.
    - Evitamos frameworks pesados, apostando por un conjunto controlado de dependencias que garantizan simplicidad y mantenibilidad.

Persistencia de Datos y Modelo de Información

Cada microservicio de Neobank gestiona su propia capa de persistencia (base de datos SQL), reforzando el encapsulamiento y la seguridad de los datos. Hemos elegido **PostgreSQL** por la naturaleza transaccional y regulada del negocio financiero.

1. **Ledger-Service**

- **Esquema `ledger`:**
  - **LedgerAccounts:** tablas de cuentas contables.
  - **Transfers:** registro de transacciones.
  - **Entries:** asientos de doble partida (débito/crédito).
- **Transacciones SQL:** inserción de Transfers + Entries en un único bloque con commit/rollback automático.
- **Saldo "on-the-fly":** no se materializa; se calcula sumando las Entries de la cuenta. Esto garantiza que la fuente de verdad sean siempre los asientos, evitando inconsistencias. Se pueden añadir índices o una tabla de balances materializada si se requiere optimización.

2. **Account-Service**
   - **Esquema `account`** (por ejemplo):
     - **Customers:** datos de usuario (perfil, estado KYC).
     - **KycRequests:** historial y resultados de validaciones.
     - **Accounts:** mapeo entre `customer_id` y `ledger_account_id`.
   - Cuando el cliente aprueba KYC, el `account-service` almacena la relación con la cuenta contable creada en el `ledger-service`.
3. **Payments-Service**
   - **Persistencia ligera:**
     - Registro opcional de solicitudes de pago, estados temporales o cron jobs (pagos programados, reintentos).
     - Cualquier tabla que se cree vive en su propia base de datos independiente.
   - La fuente de verdad de la transferencia sigue estando en el `ledger-service`.
4. **Aislamiento y Acceso a Datos**
   - No hay lecturas directas entre bases de datos de diferentes servicios.
   - Para conocer un saldo o estado, se invocan siempre las APIs (p. ej. `GetBalanceCents` en `ledger-service`), preservando la segregación de responsabilidad.

5. **Modelo Compartido Mínimo**
    - Los servicios intercambian solo identificadores y campos estrictamente necesarios.
    - El `ledger-service` maneja únicamente `ledger_account_id` y montos; el `account-service` traduce las acciones del usuario a esos IDs.
    - Esta aproximación, alineada con Domain-Driven Design, evita el acoplamiento innecesario y mantiene cada dominio bien encapsulado.

Observabilidad y Métricas

Para que Neobank sea mantenible y los equipos puedan diagnosticar problemas con rapidez, cada microservicio integra logging estructurado, métricas de monitoreo y una futura trazabilidad distribuida. Usando la librería Zap (uber-go/zap), los servicios emiten logs en formato JSON que incluyen contexto relevante —request-ID, `customer_id`, `ledger_account_id`, montos, estados de operación—, facilitando búsquedas y filtrado en herramientas como Elastic Stack o Grafana Loki. Por ejemplo, el `ledger-service` registra eventos críticos (inicialización, errores de conexión a la base de datos) y transaccionales (creación de cuentas, resultados de transferencias) con niveles de severidad adecuados (info, warning, error, debug).

En paralelo, cada servicio expone un endpoint de métricas Prometheus, recopilando tanto indicadores de infraestructura (CPU, memoria, latencia gRPC, tasa de errores) como métricas de negocio (transacciones por segundo, volumen de transferencias diarias, porcentaje de aprobaciones KYC, número de cuentas activas). Estas métricas se visualizan en dashboards de Grafana para ofrecer un panorama en tiempo real del estado del sistema y alertar sobre anomalías —por ejemplo, un pico en la latencia de `PostTransfer` o una caída en las aprobaciones KYC— antes de que impacten a los usuarios.

Finalmente, aunque la trazabilidad distribuida está en roadmap, el diseño ya contempla propagar un trace-ID y aprovechar el `externalID` de cada transferencia como identificador común. Con OpenTelemetry (o un equivalente), se podrá seguir el recorrido completo de una operación —desde la llamada inicial en `account-service`, pasando por `payments-service` y `ledger-service`, hasta los eventos Kafka—, lo que permitirá diagnosticar cuellos de botella o errores en cualquier eslabón del flujo. Juntas, estas prácticas garantizan que Neobank no sea una “caja negra”, sino una plataforma transparente y operable en producción.

Conclusión

Neobank ofrece un ecosistema de microservicios coherente y escalable que cubre todo el ciclo de vida del cliente: desde el registro y la verificación KYC hasta la gestión contable y las

transferencias P2P. Gracias a una comunicación híbrida —RPCs gRPC para los procesos críticos y Kafka para la difusión de eventos— cada servicio mantiene responsabilidades bien definidas y contratos claros, lo que facilita el desarrollo paralelo, las pruebas aisladas y la evolución independiente. El uso de Go, PostgreSQL y una arquitectura event-driven garantiza alto rendimiento, consistencia transaccional y flexibilidad para integrar nuevos consumidores de eventos sin afectar al path crítico.

Además, la adopción de buenas prácticas de observabilidad (logging estructurado, métricas Prometheus/Grafana y trazabilidad distribuida) proporciona visibilidad end-to-end, acelerando la detección y resolución de incidencias en producción. Con este diseño, cualquier desarrollador que se incorpore puede concentrarse en un único dominio, confiando en interfaces estables y mecanismos de monitoreo robustos. De este modo, Neobank establece una base sólida para crecer hacia funcionalidades financieras avanzadas, manteniendo siempre la fiabilidad y la calidad del servicio.