Arquitectura de Solución

Sistema de Banca por Internet BP

Ing. Walter Cun Bustamante
20/11/2025
Version: 0.1.0

Tabla de contenido

Ing. Walter Cun Bustamante
20/11/2025
Version: 0.1.0

1. Introduccion y contexto

1.1. Alcance del sistema:

El sistema permitirá a los clientes de BP realizar operaciones clave a través de una aplicación web (SPA) y una aplicación móvil, incluyendo consultas de saldo y movimientos, transferencias propias e interbancarias, y un proceso de onboarding digital con reconocimiento facial.

1.2. Objetivos

El sistema debe permitir

- ✓ Consultar el historial de movimientos de sus cuentas.
- ✓ Realizar transferencias y pagos entre cuentas propias.
- ✓ Realizar transferencias y pagos interbancarios.
- ✓ Recibir notificaciones de las transacciones realizadas vía email y SMS.
- ✓ Incorporar nuevos clientes a través de un proceso de onboarding digital con reconocimiento facial en la app móvil.

1.3. Requisitos

1.3.1. Funcionales

- Un cliente debe poder autenticarse en el sistema.
- Un cliente debe poder consultar el saldo y el historial de movimientos de sus cuentas.
- Un cliente debe poder realizar transferencias entre cuentas propias.
- Un cliente debe poder realizar transferencias y pagos a cuentas de otros bancos (interbancarios).
- El sistema debe notificar al cliente por email y SMS sobre las transacciones realizadas.
- Un nuevo cliente debe poder completar el proceso de onboarding a través de la aplicación móvil utilizando reconocimiento facial.
- Tras el onboarding, un cliente debe poder ingresar usando usuario/clave o métodos biométricos.

1.3.2. No Funcionales

- **Seguridad :** Todas las comunicaciones deben estar encriptadas tanto en tránsito **(TLS 1.2+)** como en reposo **(AES-256)**.
  - El sistema debe ser resistente a vulnerabilidades comunes **(OWASP)**.
  - Si se procesan tarjetas, el entorno debe cumplir con los estándares de seguridad de datos para la industria de tarjetas de pago **(Normativa PCI-DSS)**.
  - PSD2 / Open Banking: Preparar la arquitectura para exponer APIs seguras si la regulación futura lo exige los bancos.
- **Rendimiento :** El tiempo de respuesta de las APIs críticas debe ser inferior a 500ms (percentil 95).
- **Disponibilidad:** El sistema debe tener una disponibilidad del 99.9%.
- **Escalabilidad:** La arquitectura debe poder escalar horizontalmente para manejar picos de demanda.
- **Mantenibilidad:** El código debe ser modular, bien documentado y fácil de probar.
- **Cumplimiento:** El sistema debe cumplir con la ley de protección de datos (GDPR / Ley Local) y las normativas financieras vigentes.

PD: KYC (Know Your Customer) & AML (Anti-Money Laundering): Los procesos de onboarding y monitoreo de transacciones deben cumplir con las normas de prevención de lavado de activos.

2. Analisis Arquitectonico

2.1. Restricciones

2.1.1. Tecnicas

Ing. Walter Cun Bustamante
20/11/2025
Version: 0.1.0

- ✓ Se debe utilizar el producto de autenticación OAuth 2.0 ya existente en la compañía.
- ✓ La integración con el **Sistema CORE Bancario** debe realizarse a través de sus APIs existentes, basadas en estándares SOAP/XML **(Estandar de Bancos)**.
- ✓ El presupuesto para infraestructura es la nube es ampo pero no ilimitado.

2.1.2. Regulatorias

- ✓ Cumplimiento obligatorio con las leyes de protección de datos personales.
- ✓ Adherencia a las normativas de ciberseguridad para entidades financieras.
- ✓ Implementación de protocolos de "Conoce a tu Cliente" (**KYC**) y Prevención de Lavado de Dinero (**PLD**)

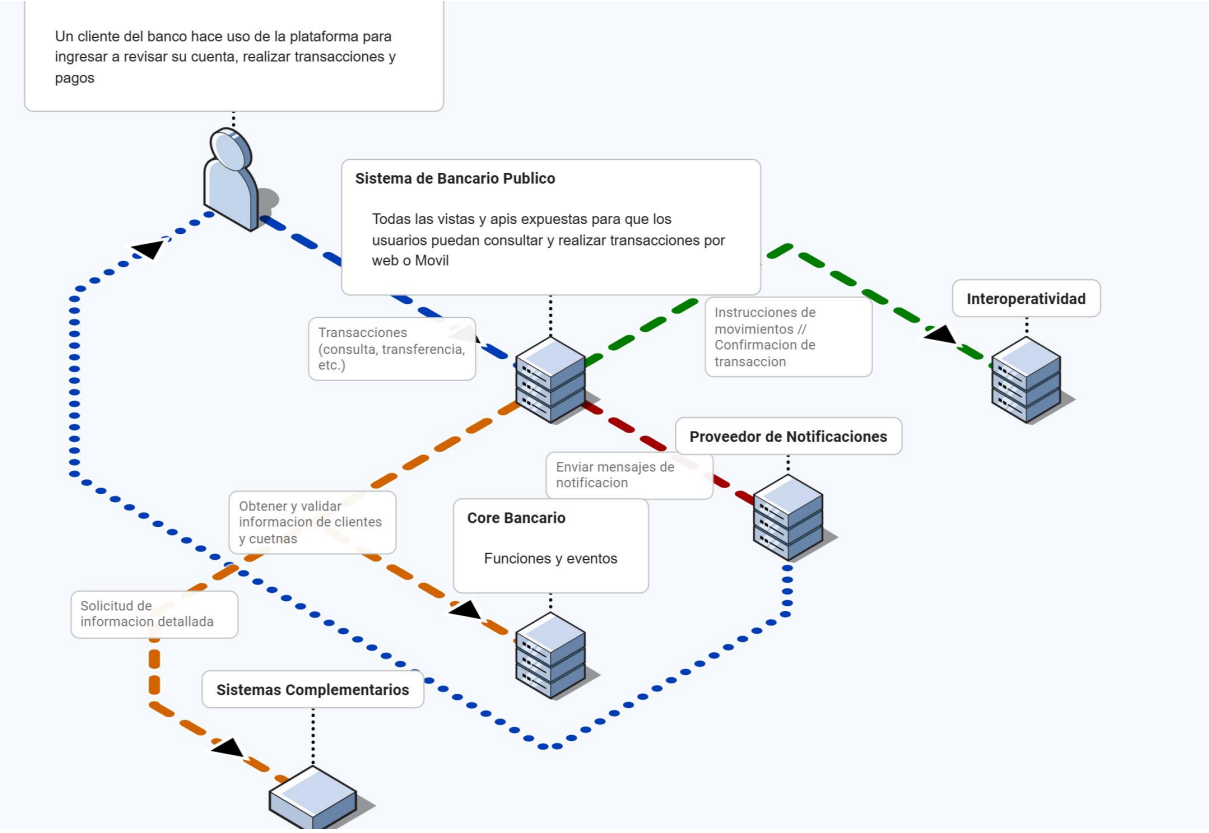
3. Diagramas

3.1. Contexto

El diagrama de contexto esta destinado para los usuarios no tecnicos, muestra sistemas internos y externos como actores clave, con breces descripciones y conexiones explicativas.

3.1.1. Actores y sistemas.

- ✓ **Cliente de BP**: El usuario final que interactúa con el sistema a través de la web o la app móvil.
- ✓ **Sistema Bancario Publico:** Linea de acceso donde estan expuestas las apis que consumira los clientes web y movil
- ✓ **Sistema CORE Bancario:** Sistema legado que contiene la información principal de clientes, productos y cuentas.
- ✓ **Sistema Complementario**: Sistema externo que provee información detallada del cliente.
- ✓ **Proveedor de Notificaciones:** Servicio externo (Mailgun, SendGrid, Amazon SES) para el envío de notificaciones por email o SMS (Twilio, Vonage)
- ✓ **Red de Pagos interoperatividad:** Entidad externa que procesa las transferencias a otros bancos.

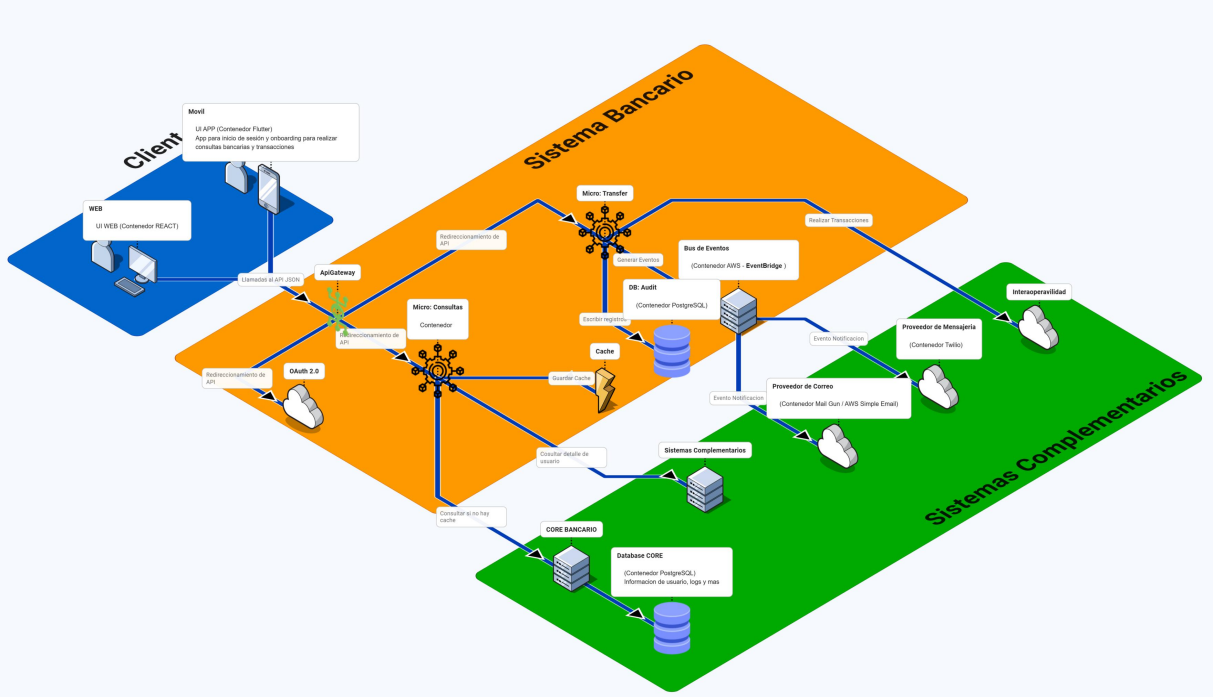


3.2. Diagrama de Contenedores

El diagrama de esta enfocado para tenicos, representa aplicaciones, servicios, base de datos y mensajeria, incluyendo componentes de cloud sin mucho detall, conexiones con descripciones breves.

3.2.1. Contenedores y Aplicaciones

- ✓ **Aplicación Web (SPA):** Interfaz de usuario basada en navegador para clientes de escritorio.
- ✓ **Aplicación Móvil (Multiplataforma):** Aplicación nativa para iOS y Android.
- ✓ **API Gateway:** Punto de entrada único para todas las peticiones de los clientes. Gestiona enrutamiento, autenticación, rate limiting y logging.
- ✓ **Servicio de Autenticación (OAuth 2.0):** Servicio existente en BP, configurable para gestionar el ciclo de vida de la autenticación de usuarios.
- ✓ **Microservicio de Consultas:** Expone datos de cliente y movimientos, integrándose con el CORE y el sistema complementario.
- ✓ **Microservicio de Transferencias:** Orquesta la lógica de negocio para transferencias y pagos.
- ✓ **Base de Datos de Auditoría:** Almacena un registro inmutable de todas las acciones realizadas por los clientes para fines de cumplimiento y seguridad.
- ✓ **Cache de Datos (Redis):** Almacena temporalmente datos de clientes frecuentes para mejorar el rendimiento y reducir la carga en los sistemas backend.
- ✓ **Bus de Mensajería:** Desacopla la generación de eventos (ejemplo: login, transferencia o pago) de su consumo (ejemplo: el envío de notificaciones).



3.2.2. Justificacion

Arquitectura Analizada: Microservicios vs Monolito **(Sin limitaciones de presupuesto)**

- ✓ **Conclusion:** Se opta por una arquitectura de microservicios.
- ✓ **Justificación Teórica:** Permite el despliegue independiente, escalabilidad granular (el servicio de transferencias puede necesitar más recursos que el de consultas), y fomenta la **cohesión** y el **desacoplamiento**. Cada equipo puede trabajar en un servicio de forma autónoma.

Aplicacion de API Gateway:

- ✓ **Justificación Teórica:** Implementa el patrón API Gateway. Centraliza preocupaciones transversales como seguridad, enrutamiento y monitoreo, evitando la duplicación de código en cada microservicio. Simplifica la comunicación para los clientes frontend.

Aplicacion de Bus de Mensajería:

- ✓ **Justificación Teórica:** Implementa el patrón **Arquitectura Orientada a Eventos (EDA)**. Desacopla al microservicio de transferencias de los sistemas de notificación. Si se añade una nueva forma de notificación por ejemplo **"Push Notification"** o **"Whatsapp Notification"**, solo se necesita añadir un nuevo consumidor al bus, sin modificar el servicio de transferencias. Esto mejora la resiliencia y la escalabilidad.

3.3. Diagrama de Componentes

El diagrama de componentes contiene un mayor detalle tecnico, incluye los microservicios, patrones arquitectonicos y protocolos de comunicacion con seguridad, destaca el uso de los componentes cloud.

3.3.1. Componentes

A. Cliente

a) Aplicación Web

i. Componentes UI (React)

Renderizar la interfaz de usuario que el cliente ve e interactúa en su navegador.

ii. Gestor de Estado

Almacenar y gestionar el estado global de la aplicación de forma predecible por ejemplo:

1. datos del usuario autenticado
2. saldo de cuentas
3. historial de transacciones

iii. Cliente API

Realiza todas las peticiones HTTP desde la aplicación web hacia el backend (API Gateway).

iv. Manejador de Auth (OAuth2 PKCE "clave de prueba para intercambio de códigos")

Gestionar el flujo de autenticación OAuth 2.0 con PKCE.

1. **Justificación:** Este componente no maneja la lógica de autenticación, sino que orquesta la interacción con el Servicio de Autenticación. Se encarga de redirigir al usuario, generar el code_challenge y code_verifier, intercambiar el código de autorización por un token y gestionar el ciclo de vida de dicho token (acceso y refresco).

b) Aplicación Móvil (Flutter)

i. Capa de Presentación (Widgets)

Construir la interfaz de usuario nativa para iOS y Android.

1. **Justificación:** En Flutter, todo es un Widget. Este enfoque permite una composición de UI extremadamente rápida y flexible. El motor de renderizado propio de Flutter (Skia) garantiza que la UI se vea y se sienta idéntica en ambas plataformas, con un rendimiento de 60/120 fps.

ii. **Gestor de Estado (BLoC / Provider / Riverpod)**

Separar la lógica de negocio de la presentación y gestionar el estado de la aplicación de forma reactiva.

1. **Justificación:** El patrón BLoC (Business Logic Component) es ideal para Flutter. Utiliza Streams para emitir estados. La UI se "suscribe" a estos streams y se reconstruye automáticamente cuando un nuevo estado es emitido. Esto crea una clara separación de responsabilidades y una UI muy responsiva a cambios de datos asíncronos como por ejemplo el estado de una transferencia.

iii. **Cliente API (Dio / http)**

Realizar las peticiones HTTP desde la aplicación móvil.

1. **Justificación:** Dio es el equivalente a Axios en el ecosistema Flutter. Ofrece un potente conjunto de características, incluyendo interceptores para la gestión de tokens, cancelación de peticiones y soporte para timeouts, lo que lo hace ideal para un entorno móvil donde la conectividad puede ser inestable.

iv. **Almacenamiento Seguro (Keystore/Keychain)**

Almacenar de forma segura información sensible en el dispositivo, como los tokens de refresco de OAuth.

1. **Justificación:** Utiliza los mecanismos de seguridad a nivel de hardware y sistema operativo. En Android, interactúa con el Keystore, y en iOS, con el Keychain. Estos almacenes están encriptados y diseñados para resistir el acceso no autorizado, incluso si el dispositivo está comprometido (rooteado).

v. **Manejador Biométrico**

Interactuar con las APIs biométricas del dispositivo (Face ID, Touch ID, sensor de huellas).

1. **Justificación:** Utiliza el paquete local_auth de Flutter, que es un wrapper sobre las APIs nativas (LocalAuthentication en iOS, BiometricPrompt en Android). Este componente nunca accede a los datos biométricos; simplemente solicita al sistema operativo que confirme la identidad del usuario y recibe una respuesta de "éxito" o "fracaso", lo cual es un diseño seguro y que respeta la privacidad.

vi. **SDK de Onboarding (Onfido/Veriff)**

Gestionar el proceso de onboarding digital, incluyendo la captura de documentos de identidad y la verificación facial con prueba de vida (liveness detection).

1. **Justificación:** Integrar un SDK de un tercero especializado es una decisión de seguridad y cumplimiento normativo. Estos servicios utilizan algoritmos de IA/ML avanzados para detectar fraudes, validar documentos y cumplir con los estándares KYC (Know Your Customer). Delegar esta funcionalidad reduce drásticamente el riesgo y la complejidad del desarrollo.

B. **Backend (Lado del Servidor)**

a) API Gateway (Componente de Enrutamiento)

Recibir todas las peticiones externas y dirigirlas al microservicio correcto basándose en la URL y el método HTTP

1. /api/transfers -> Microservicio de Transferencias.
2. /api/consults -> Microservicio de Consultas

ii. Validación de Token

Verificar que cada petición entrante incluya un token JWT válido. Si no es válido o ha expirado, rechaza la petición antes de que llegue a los microservicios.

iii. Limitación de Tasa (Rate Limiting)

Proteger los backend de abusos y ataques de denegación de servicio (DDoS) limitando el número de peticiones que un cliente puede realizar en un período de tiempo.

b) Microservicio de Consultas

i. API Controller (REST)

Exponer los endpoints HTTP

1. GET /accounts/{id} -> que los clientes pueden invocar para obtener datos basicos de cuenta.
2. GET /hostory/{id} -> que los clientes pueden invocar para consultar el historico de movimientos

ii. Middleware de Autenticación

Añadir una capa de seguridad adicional. Aunque el API Gateway ya validó el token, este middleware puede realizar comprobaciones más granulares, por ejemplo si el usuario tiene permisos para ver la información solicitada.

iii. Lógica de Caché (Patrón Cache-Aside)

Gestionar la interacción con la caché para mejorar el rendimiento.

1. **Justificación:** Implementa el patrón Cache-Aside. Cuando se solicitan datos, primero consulta a Redis. Si los encuentra (cache hit), los devuelve. Si no (cache miss), los pide al Adaptador de Integración CORE, los almacena en Redis por tiempo limitado para futuras consultas y luego los devuelve. Esto reduce drásticamente la latencia y la carga en el **sistema CORE**.

iv. Adaptador de Integración CORE

Actuar como un traductor. Recibe peticiones del microservicio en un formato moderno (gRPC/REST) y las traduce al protocolo que el Sistema CORE entiende (probablemente SOAP/XML). Aísla al microservicio de la complejidad del legado.

c) **Microservicio de transacciones**

i. API Controller (REST)

Exponer los endpoints para iniciar y procesar transferencias

1. POST /transfers/{object} -> que los clientes pueden invocar para crear transacciones internas o externas, el objeto “object” contiene la informacion y metodos para realizar la transferencia.

ii. Lógica de Negocio
El cerebro del servicio. Contiene todas las reglas de negocio: validar saldos, verificar límites de transacción, aplicar comisiones, etc.

iii. Publicador de Eventos
Una vez que una transferencia se completa con éxito, este componente publica un evento por ejemplo transferencia realizada o ingreso al sistema al bus de mensajeria. Esto desacopla la lógica de transferencias de las acciones que deben ocurrir después.

iv. Componente de Auditoría
Escribir un registro inmutable y detallado de cada intento de transferencia en la Base de Datos de Auditoría. Este registro es crucial para la seguridad, la resolución de disputas y el cumplimiento normativo.

d) Data & Messaging

i. Cache de Datos (Redis)

Almacenar temporalmente datos de acceso frecuente para acelerar las lecturas.

1. **Justificación:** Redis es una base de datos en memoria, lo que la hace extremadamente rápida para operaciones de lectura y escritura. Es el estándar de la industria para aumentar el rendimiendo de respuesta de servicios.

e) Base de Datos de Auditoría (DynamoDB/CosmosDB)

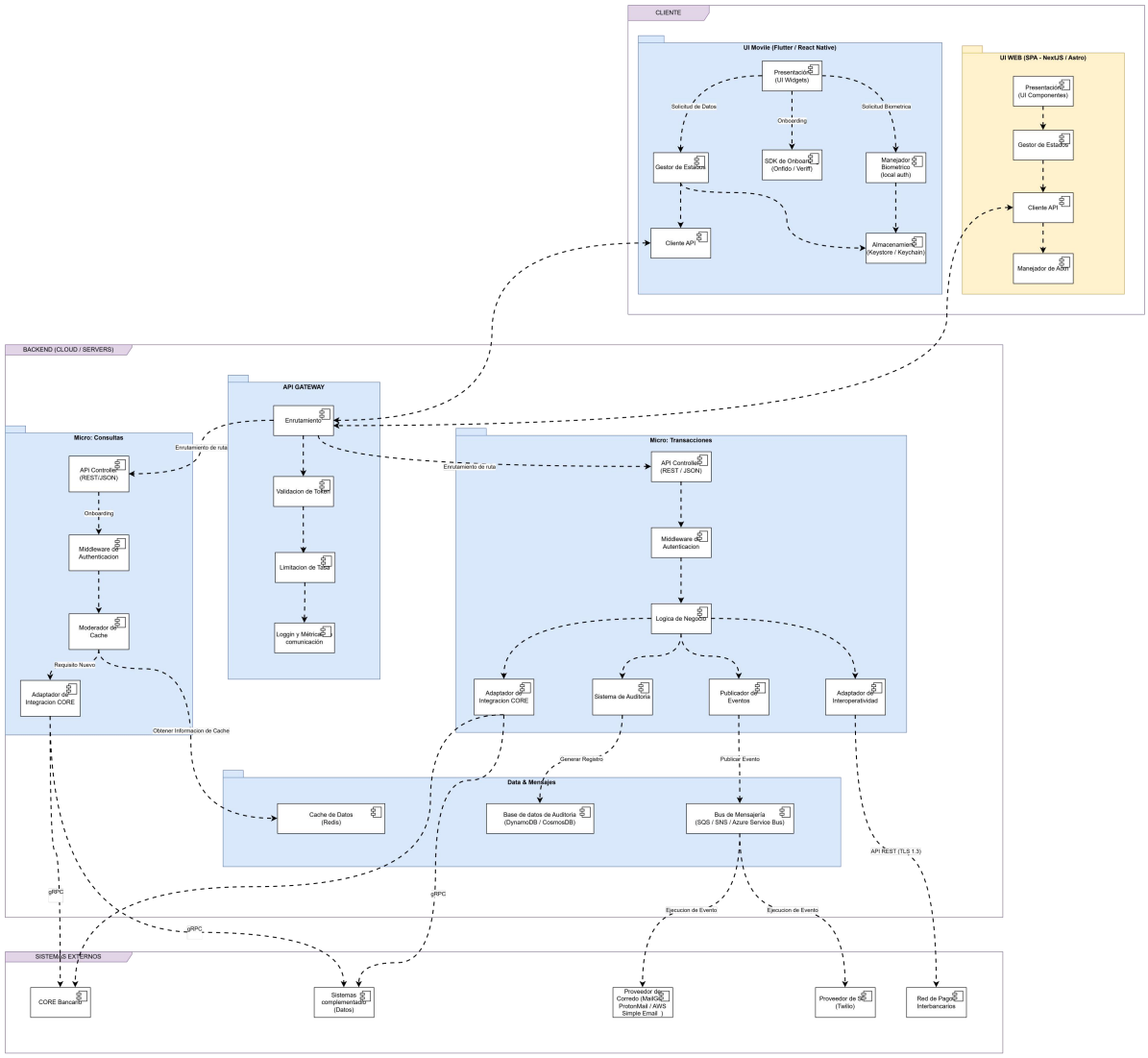
Almacenar un registro cronológico e inmutable de todas las acciones relevantes del sistema

i. **Justificación:** Se elige una base de datos NoSQL porque está optimizada para altas tasas de escritura y puede manejar esquemas flexibles. Los registros de auditoría a menudo tienen diferentes campos, y una NoSQL como DynamoDB permite esto sin necesidad de migraciones de esquema complejas. Su escalabilidad horizontal es ideal para un volumen de auditoría que crece constantemente.

f) Bus de Mensajería (SQS/SNS)

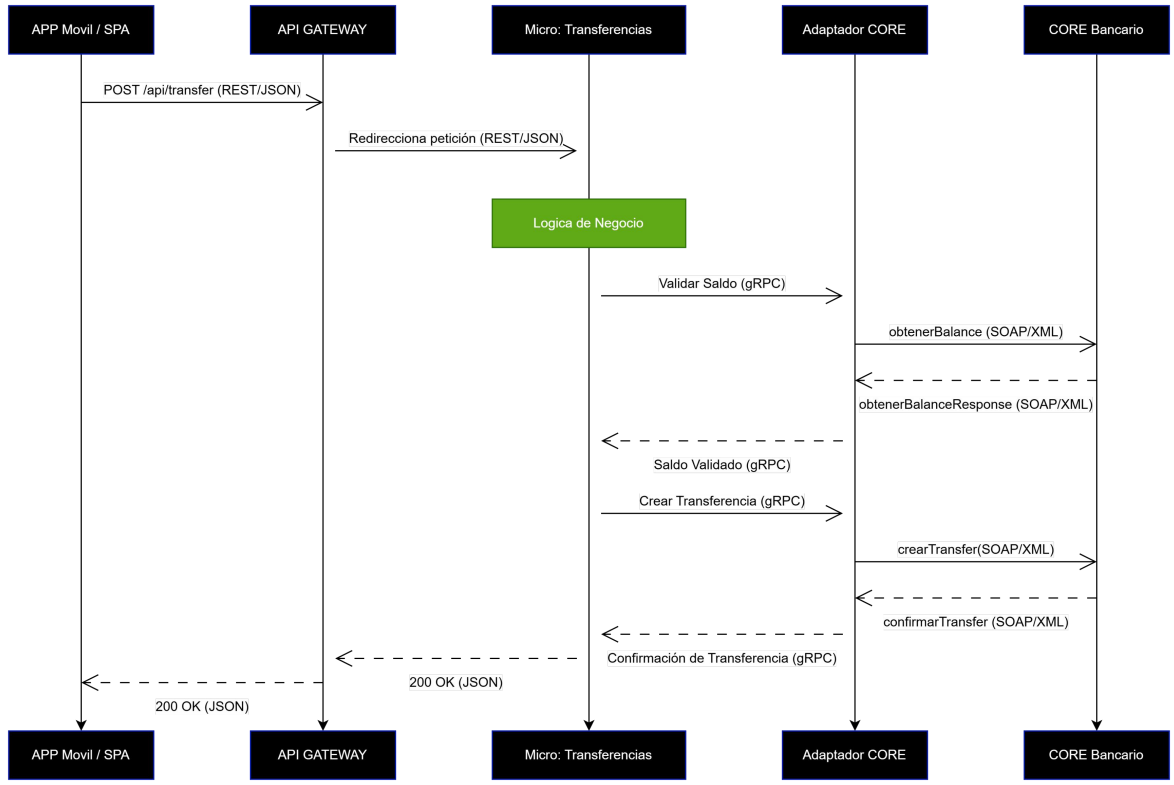
Actuar como un intermediario que desacopla a los productores de eventos (Microservicio de Transferencias) de los consumidores (Proveedores de Notificación).

i. **Justificación**: Este componente es la base de la **Arquitectura Orientada a Eventos**. Garantiza que si un servicio de notificación cae, las notificaciones no se pierden, sino que se quedan en la cola para ser procesadas cuando el servicio se recupere. Esto aumenta enormemente la resiliencia del sistema.



- **Comunicación entre Microservicios (Analisis adicional):**
  - **Analisis:** Usar gRPC para comunicación síncrona entre microservicios internos. Mantener REST/JSON para la comunicación externa (desde el API Gateway).
  - **Justificación Teórica:** gRPC es más eficiente y rápido que REST/JSON, ya que utiliza HTTP/2 y Protocol Buffers para la serialización. Es ideal para comunicación de máquina a máquina en un entorno de red controlado como un clúster de Kubernetes o red interna. Se mantiene REST para el exterior por su simplicidad y amplia adopción en comunicaciones entre cliente y servidor.

Comunicación con Sistemas Legados (Adaptador CORE y CORE BANK): Protocolo Nativo del Legado por defecto SOAP/XML. El Adaptador será el único componente que maneje esta complejidad.



- **Base de Datos de Auditoría:**
  - **Analisis:** Usar una base de datos de documentos como Amazon DynamoDB o Azure Cosmos DB.
  - **Justificación Teórica:** El patrón de auditoría a menudo implica escribir grandes volúmenes de datos inmutables con un esquema flexible. Las bases de datos NoSQL ofrecen un alto rendimiento de escritura, escalabilidad horizontal y un esquema flexible, lo que las hace ideales para este caso de uso. Se puede configurar un TTL (Time-To-Live) para archivar datos antiguos a un almacenamiento más económico como S3 (recomendable).
- **Persistencia de Datos para Clientes Frecuentes (Cache):**
  - **Analisis:** Implementar el patrón Cache-Aside.
  - **Justificación Teórica:** Este patrón es simple y efectivo. La lógica de la aplicación es responsable de mantener la caché sincronizada con la base de datos.
    1. La aplicación recibe una solicitud de datos.
    2. Comprueba primero en la caché (Redis).
    3. Si encuentra los datos (cache hit), los devuelve.
    4. Si no los encuentra (cache miss), los busca en la base de datos (CORE), los almacena en la caché de forma temporal, segun el tipo de consulta para futuras peticiones y los devuelve a la aplicación.

4. Recomendaciones Arquitectónicas

4.1. Seguridad y Autenticación

Flujo de Autenticación OAuth 2.0:

- **Decisión:** Se recomienda el flujo **Authorization Code Flow** with **Proof Key for Code Exchange (PKCE)**.
- **Justificación Teórica:** Tanto la SPA como la aplicación móvil son consideradas "clientes públicos" (no pueden guardar un secreto de forma segura).
  - **Authorization Code Flow** es el flujo más seguro para clientes públicos porque el código de autorización se intercambia directamente por un token de acceso en el backend, sin exponerlo al navegador del usuario.
  - **PKCE** añade una capa adicional de seguridad al prevenir ataques de "intercepción de código de autorización". La aplicación móvil genera un "code verifier" aleatorio y su hash "code challenge". El servidor de autenticación solo canjeará el código si recibe el **"code verifier"** correcto, impidiendo que un atacante que intercepte el código pueda canjearlo.

Autenticación Biométrica:

- **Decisión:** Para el onboarding, integrar un servicio de terceros especializado como Onfido o Veriff. Para el login posterior, usar los APIs biométricos nativos del dispositivo.
- **Justificación Teórica:**
  - **Onboarding:** El reconocimiento facial con prueba de vida (liveness detection) es un problema complejo y regulado. Usar un proveedor especializado garantiza un alto grado de precisión, cumplimiento normativo (KYC - Conoce a tu Cliente) y reduce drásticamente el riesgo de fraude. Estos servicios manejan la captura de documentos, verificación de identidad y biometría.
  - **Login Post-Onboarding:** Los sistemas operativos modernos (iOS/Android) proveen marcos seguros (Keychain/Keystore) y APIs (Face ID/Touch ID/BiometricPrompt) para la autenticación local. Usarlos es más seguro y proporciona una mejor experiencia de usuario que implementar una solución propia. La aplicación solo recibe una respuesta de éxito/fracaso, nunca los datos biométricos en sí. Esto se alinea con el estándar FIDO2 / WebAuthn.

4.2. Frontend: SPA y Aplicación Móvil

Aplicación Web (SPA):

- **Tecnologia Analizada:** React vs NextJS para aplicacion SPA.
- **Justificación Teórica**
  - **React:** Es una libreria que permite la creación de componentes reutilizables, lo que puede acelerar el desarrollo y mantener el código más limpio. React tiene un ecosistema extremadamente maduro, una gran comunidad y una amplia oferta de librerías para cualquier necesidad (gestión de estado, UI components, etc.). Su modelo de componentes reutilizables acelera el desarrollo y facilita el mantenimiento.
  - **Next.js:** Es un framework construído sobre React que ofrece características como el renderizado del lado del servidor y la generación de sitios estáticos, las cuales pueden no ser necesarias para este proyecto específico.
  - **Conclusion:** Si la aplicación se centra en la interacción del cliente y no requiere SEO avanzado o tiempos de carga rápidos a través del renderizado del servidor, entonces React por sí solo sería suficiente. En resumen, dado que no se necesitan las capacidades adicionales de Next.js, iniciar con React facilitaría una implementación más sencilla y enfocada. React en este caso puede ser beneficioso debido a su facilidad para construir interfaces de usuario interactivas y su gran comunidad de soporte.

Aplicación Móvil:

- **Tecnologia Analizada :** Flutter vs React Native.
- **Justificación Teórica**

Se evalúa Flutter frente a React Native por lo siguente.

- **Flutter:** Compila a código ARM nativo, lo que generalmente resulta en un mejor rendimiento y una UI más fluida y consistente en ambas plataformas, ya que no depende de un "puente" de JavaScript para comunicarse con los componentes nativos. Su motor de renderizado propio (Skia) garantiza que la UI se vea idéntica en iOS y Android. Esto es crucial para una aplicación bancaria donde la consistencia y el rendimiento son clave.
- **React Native:** Aunque tiene un ecosistema robusto, su rendimiento puede verse afectado por la comunicación a través del puente, lo que puede causar latencia en UI complejas.
- **Conclusión:** Para una aplicación bancaria que prioriza el rendimiento, una UI fluida y la coherencia visual, Flutter presenta una ventaja técnica superior.

4.3. Alta Disponibilidad - Tolerancia a Fallos con Recuperación ante Desastres

Alta disponibilidad y Tolerancia a Fallos:

- **Decisión:** Para la alta disponibilidad del servicio la infraestructura se desplegará en múltiples Zonas de Disponibilidad **(AZs)**. Se usarán **Auto Scaling Groups de AWS** para escalar servicios automáticamente y Read Replicas o bases de datos multi-AZ para la persistencia de datos (si se usan relacionales) o aprovechar la replicación **multi-AZ** nativa de las bases de datos NoSQL.
- **Justificación Teórica:** Las AZs son centros de datos aislados dentro de la misma región. Si una AZ falla, el tráfico se redirige automáticamente a las AZs sanas, garantizando que la aplicación siga funcionando. Los **Auto Scaling Groups** reemplazan automáticamente las instancias no saludables para mantener la disponibilidad del servicio.

Recuperacion ante desastres:

- **Decisión:** Para estas preparado ante los desastres (caida de servidores) se implementará una estrategia de **Pilot Light o Warm Standby** en una región geográfica separada (varias si el presupuesto lo permite sino unicamente un respaldo). Las copias de seguridad de datos críticos se replicarán en múltiples regiones se puede utilizar **"Amazon S3 Cross-Region Replication"**.
- **Justificación Teórica:** Esto protege contra desastres a gran escala (terremotos, cortes de energía regional). Una estrategia de Pilot Light mantiene una versión mínima del entorno corriendo en la región de Diferente, lo que permite una recuperación rápida en caso de fallo total de la región primaria.

4.4. Monitoreo y Excelencia Operativa

- **Decisión:** Utilizar un stack de monitoreo unificado como **AWS CloudWatch o Azure Monitor** (métricas, logs, alarmas) y **AWS X-Ray (tracing distribuido)** o sus equivalentes en **Azure (Azure Monitor, Application Insights)**. Implementar dashboards para visualizar métricas clave (KPIs) como latencia de transacciones, tasa de error, y uso de CPU/memoria.
- **Justificación Teórica:** El tracing distribuido es crucial en una arquitectura de microservicios para seguir una petición a través de todos los servicios que toca, identificando cuellos de botella y errores rápidamente. Las alarmas automatizadas permiten la auto-curación (self-healing), por ejemplo, reiniciando un servicio que no responde o escalando recursos automáticamente bajo carga.

5. Consideraciones Regulatorias y de Cumplimiento

El diseño de la arquitectura debe tener en cuenta las siguientes normativas para entidades financieras:

- ✓ **Ley de Protección de Datos(GDPR “Internacional” y Ley local):** Se implementará la encriptación de datos en tránsito (TLS 1.2+) y en reposo. Se garantizará el consentimiento del usuario y el derecho al olvido.
- ✓ **Normativas de Ciberseguridad Financiera:** Se aplicarán controles de acceso estrictos, segmentación de redes y monitoreo de seguridad continuo. Requieren controles de acceso estrictos, segmentación de redes, monitoreo de seguridad constante, planes de respuesta a incidentes y auditorías de seguridad periódicas (pentesting)
- ✓ **Prevención de Lavado de Dinero:** El onboarding digital cumplirá con los protocolos de **"Conoce a tu Cliente"** (KYC) y el sistema debe ser capaz de monitorear transacciones inusuales y generar reportes a las autoridades competentes. La Base de Datos de Auditoría es fundamental para este propósito.