Diseño de un Sistema de Banca por Internet

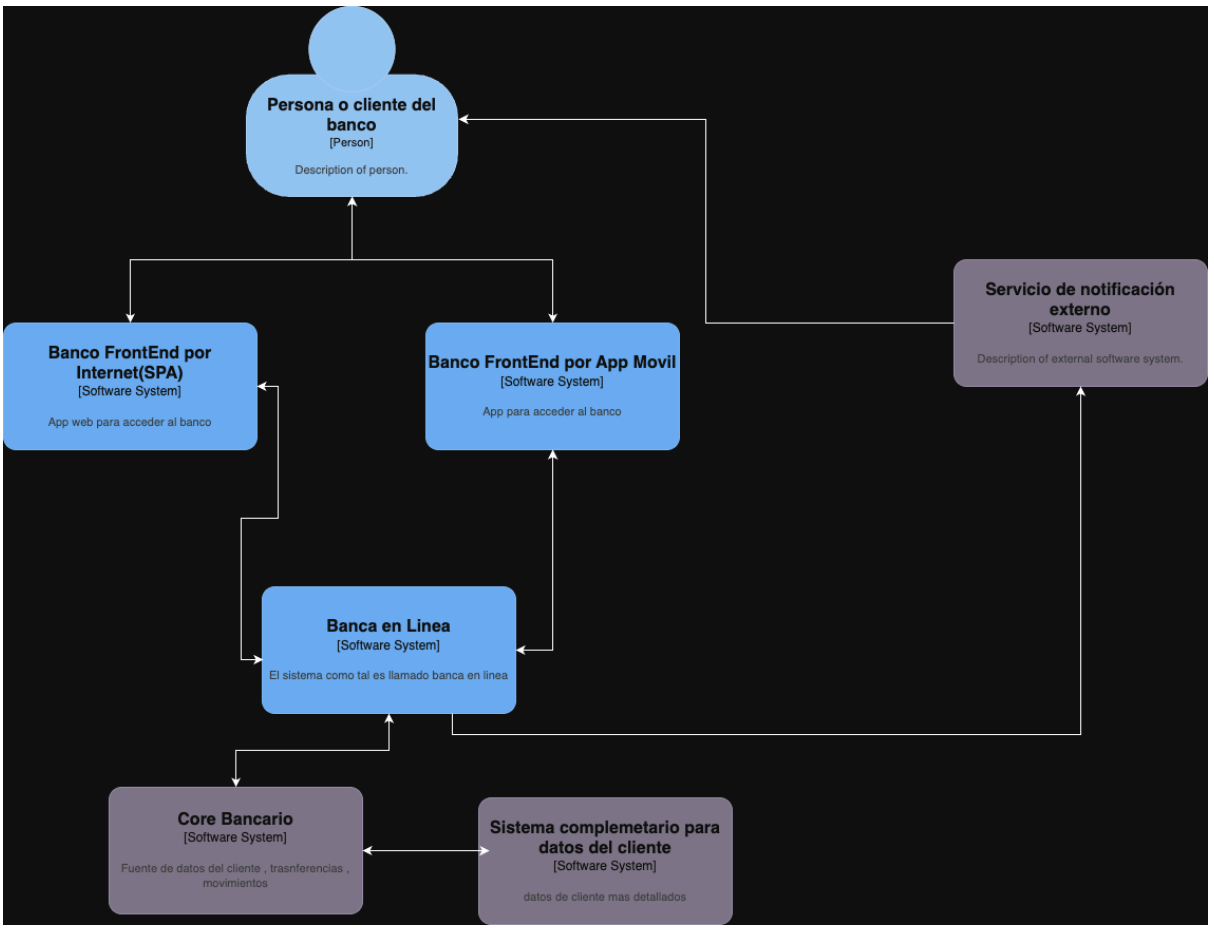
por Fausto Iván Zamora Arias

El siguiente sistema utiliza el framework C4 para describir los componentes que lo conforman, estructurado en diferentes niveles de abstracción.

Nivel 1: Diagrama de Contexto

El sistema interactúa con 7 actores principales:

1) Cliente bancario: Usuario final que accede a los servicios mediante las aplicaciones frontend.
2) Aplicación web (SPA): Single Page Application accesible desde un navegador web.
3) Aplicación móvil: Diseñada para dispositivos inteligentes (smartphones y tablets).
4) Sistema de banca en línea: Actúa como intermediario entre el frontend y los sistemas backend, procesando solicitudes (ej: transferencias) y coordinando su ejecución con el core bancario y las bases de datos.
5) Core bancario: Gestiona datos críticos como información de clientes, transacciones y movimientos históricos.
6) Sistema complementario: Proporciona datos adicionales y específicos de los clientes.
7) Servicio externo de notificaciones: Envía alertas y mensajes al cliente mediante dos canales basados en cloud.



Nivel 2: Diagrama de Contenedores

En este nivel tenemos contenedores que mostrarás la interacción descrita en el nivel 1 pero con mas detalle.

Frontend: Aplicaciones del Usuario
SPA (Single Page Application) y APP Mobil

Aquí tenemos a un usuario que interactúa con un SPA si usa un navegador o una App móvil si usa un teléfono inteligente. En específico el SPA usará ReactJS y se deployar con Amazon Cloudfront. Por otro lado , la app móvil usará un framework multiplataforma de los que yo recomiendo React Native o DotNet MAUI. Recomiendo

React Native(framework para crear aplicaciones nativas) por que usa Javascript igual que la propuesta para el SPA , lo que significa que no habría problemas de saltar de un feature a hacer otro idéntico en la app Web como en la app móvil, además de que tiene tiempo en el mercado y es respaldado por Meta y sigue siendo actualizado. Sin embargo la segunda opción es usar DotNet MAUI que es un framework multiplataforma para crear aplicaciones nativas pero usando como base tecnologia DotNet y su desarrollo debe ser con el lenguaje C#, que aunque discrepe de la tecnología para usar en el SPA tiene la ventaja que C# es un lenguaje muy usado en entornos fintech y banca así que los desarrolladores en este dominio podrían adaptarse fácilmente a DotNet MAUI y tener un inicio más rápido.

Puntos clave del SPA:

- Desarrollada en ReactJS.

- Desplegada en Amazon CloudFront para distribución global y baja latencia.

Puntos clave de la App Móvil

- Tecnología recomendada: React Native (por consistencia con el SPA, soporte de Meta y actualizaciones continuas).

- Ventaja de React Native: Comparte lógica con el SPA (JavaScript/TypeScript), facilitando el desarrollo de features idénticos en ambas plataformas.

- Alternativa: .NET MAUI (basado en C#).

- Ventaja de .Net MAUI: Ideal para entornos fintech, donde C# es ampliamente adoptado, permitiendo una curva de aprendizaje más rápida para equipos bancarios.

Autenticación y Onboarding

Después tenemos los contenedores de Autenticación y Onboarding. Están divididos pues aunque tengan flujos similares, autenticar usuarios, Onboarding necesita algunos pasos tanto en el front end como en el backend para registrar usuarios. Sin embargo ambos usarán Amazon Cognito para guardar y autenticar usuarios. Es así como su lógica se mantendrá en microservicios separados pero relacionados por Cognito. Cabe mencionar que además de esto el sistema contempla una base de datos de usuarios o clientes que será compartida por otros sistemas para el registro de movimientos y actividades que no están relacionados con estos dos sistemas sin embargo el logueo se registrará como una actividad o entradas en dicha base de datos, esto se explica más a fondo en el Nivel 3.

Autenticación:

- Gestionada por Amazon Cognito (almacenamiento y validación de credenciales).

Onboarding:

- Flujo independiente pero integrado con Cognito.

- Registra datos adicionales en una base de datos compartida (para movimientos y auditoría).

Backend for Frontend (BFF)

Después tenemos al BFF(backend for frontend) que ayuda a ejecutar lógica que solo debería ser accedida por medio de la red interna/VPC del banco y se comunica con las aplicaciones de front end. Para poder comunicarse se requiere usar un JWT que debió haber sido creado por medio de la autenticación de los pasos anteriores. En específico el BFF podrá efectuar las operaciones de transacciones interbancarias al llamar servicios del Core Bancario(otro componente), actualizar datos del cliente y mantener un caché para usuarios que se loguean constantemente.

Función:

- Expone APIs internas del banco (protegidas por JWT generado en autenticación).

- Opera dentro de la VPC del banco para seguridad.

Capacidades:

- Ejecuta transacciones interbancarias (mediante Core Bancario).

- Actualiza datos de clientes y mantiene un cache de usuarios frecuentes.

Persistencia de Datos (BFF)

En el caso de las bases de datos de persistencia para usuarios podemos presentar dos soluciones , la primera es usar Redis por clave valor para los usuarios y rápido acceso por ser en memoria; La segunda es usar una base de datos en Dynamo DB propia del microservicio de BFF pues el acceso es rápido por clave primaria y secundaria , sumado al hecho de que sabemos el id del usuario desde el momento de la autenticación así que no se requiere un queri complejo para retornar la información. Además usar un DynamoDB ayudaría al proceso de conciliación de datos.

Redis:

- Almacenamiento clave-valor en memoria (acceso ultrarrápido).

DynamoDB:

- Ideal para búsquedas por ID de usuario (claves primarias/secundarias).
- Facilita la conciliación de datos.

Microservicio de Notificaciones

Para las notificaciones se propone tener un microservicio enfocado totalmente en notificaciones , las cuales se alimentan por medio de la logica del BFF. Por ejemplo si existe una transferencia interbancaria que el usuario la realizó , que el BFF mandó a ejecutar por medio del Core Bancario, pero la operación será comunicada en una hora o más , el microservicio tendrá la opción de escuchar una cola por medio de SQS que puede tener mensajes publicador por medio del Core Bancario , de esta forma una vez obtenido ese mensaje publicarlo en un SNS y en Firebase Cloud Messaging. Como se mencionó anteriormente el sistema tiene dos agentes de notificación de mensajes pues ese es uno de los requerimientos del sistema. El Amazon SNS, Single Notification Service, es un sistema de notificaciones que permite mandar notificaciones push, SMS y/o correo electrónico , lo cual es muy beneficioso para notificar al cliente por varios medios; por ejemplo enviar un SMS notificando al usuario que su transacción ha sido realizada con exito. Además se ha mencionado el uso de Firebase Cloud Messaging, pues es una solución de mensajería multiplataforma por lo cual se puede usar su SDK para que la misma aplicación móvil o el SPA reciba mensajes. Es asi como el componente de notificaciones abarca varios canales y notifica al usuario.

Arquitectura:

- Escucha colas (Amazon SQS) con eventos del Core Bancario (ej: transferencias diferidas).

Publica mensajes en:

- Amazon SNS:
- Notificaciones multicanal (SMS, email, push).

Ejemplo: SMS confirmando una transacción.

Firebase Cloud Messaging (FCM):

- Notificaciones push en tiempo real para el SPA y la app móvil.

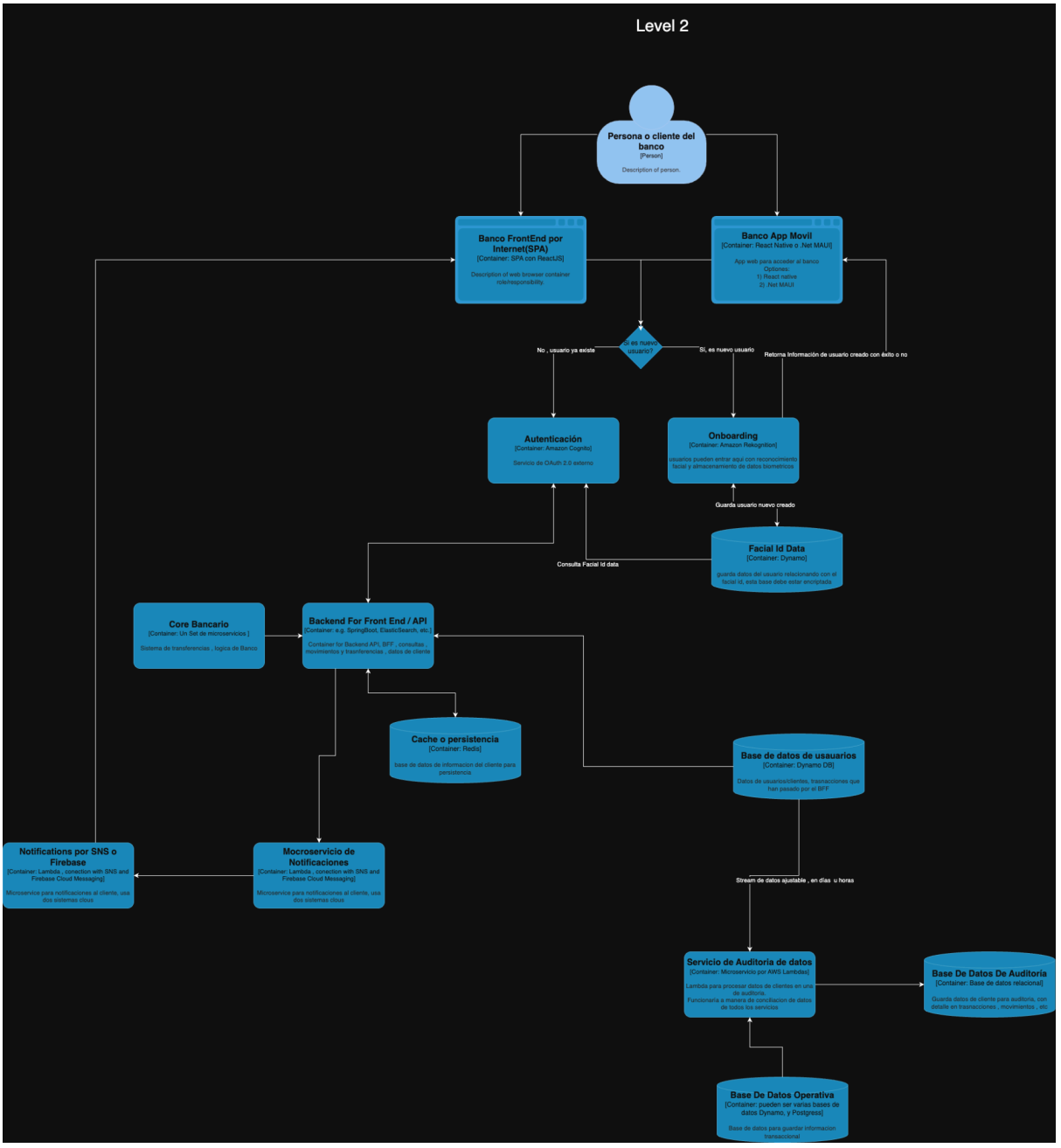
Servicio de Auditoría y Consolidación de Datos

Por último tenemos el servicio de auditoría de datos, o consolidación de datos. Este servicio se encarga de obtener todos los datos que vienen del Core Bancario, de los clientes, los movimientos que captura el BFF y los consolida en una base de datos relacional con tablas. Este tipo de sistemas siempre representan un desafío debido al alto volumen y costos asociados, es por esto que aqui se presentan dos opciones que serán detalladas en el nivel 3, estas son: primero, usar un ETL programado usando AWS Glue(Servicio de integración de datos serverless) cada 6 horas ; o la segunda , usar streams de DynamoDB(asumiendo cada microservicio tiene una tabla en dynamo asociada) transformando los datos a una base de datos relacional como Postgres, que sería inmediato con cada operación de Dynamo.

Objetivo:
Agregar datos del Core Bancario, BFF y clientes en una base de datos relacional (ej: PostgreSQL).

Opciones de Implementación:

- ETL Programado (AWS Glue):
  Procesamiento serverless cada 6 horas (balance entre costo y actualización).
- Streams en Tiempo Real (DynamoDB + PostgreSQL):
  Captura cambios inmediatos en DynamoDB y los replica en PostgreSQL.


Level 2
Persona o cliente del banco
[Person]
Description of person.
Banco FrontEnd por Internet(SPA)
[Container: SPA con ReactJS]
Description of web browser container role/responsibility
Banco App Movil
[Container: React Native o .Net MAUI]
App web para acceder al banco
Opciones:
1) React native
2) .Net MAUI
¿Si es nuevo usuario?
No , usuario ya existe
Si, es nuevo usuario
Retorna Información de usuario creado con éxito o no
Autenticación
[Container: Amazon Cognito]
Servicio de OAuth 2.0 externo
Onboarding
[Container: Amazon Rekognition]
usuarios pueden entrar aqui con reconocimiento facial y almacenamiento de datos biometricos
Guarda usuario nuevo creado
Facial Id Data
[Container: Dynamo]
guarda datos del usuario relacionando con el facial id, esta base debe estar encriptada
Consulta Facial Id data
Core Bancario
[Container: Un Set de microservicios]
Sistema de transferencias , logica de Banco
Backend For Front End / API
[Container: e.g. SpringBoot, ElasticSearch, etc.]
Container for Backend API, BFF , consultas , movimientos y trasnferencias , datos de cliente
Cache o persistencia
[Container: Redis]
base de datos de informacion del cliente para persistencia
Base de datos de usuarios
[Container: Dynamo DB]
Datos de usuarios/clientes, trasnacciones que han pasado por el BFF
Notifications por SNS o Firebase
[Container: Lambda , conection with SNS and Firebase Cloud Messaging]
Microservice para notificaciones al cliente, usa dos sistemas clous
Mocroservicio de Notificaciones
[Container: Lambda , conection with SNS and Firebase Cloud Messaging]
Microservice para notificaciones al cliente, usa dos sistemas clous
Stream de datos ajustable , en dias u horas
Servicio de Auditoria de datos
[Container: Microservicio por AWS Lambdas]
Lambda para procesar datos de clientes en una de auditoria.
Funcionaria a manera de conciliacion de datos de todos los servicios
Base De Datos De Auditoria
[Container: Base de datos relacional]
Guarda datos de cliente para auditoria, con detalle en trasnacciones , movimientos , etc.
Base De Datos Operativa
[Container: pueden ser varias bases de datos Dynamo, y Postgress]
Base de datos para guardar informacion transeaccional

Nivel 3

Este nivel profundiza en la arquitectura interna de cada contenedor, descomponiéndose en componentes específicos y definiendo sus interacciones. Se emplea nomenclatura de AWS para estandarizar la implementación, y dado el nivel de detalle, el contenido se organiza en secciones temáticas.

Frontend: Arquitectura y Controladores

Como se lo había expuesto anteriormente el front end va a estar dividido en dos aplicaciones: SPA y Aplicación Móvil , la primera un SPA que usará ReactJS y se deplora en AWS usando Cloud front, el cual será almacenado en un S3 con Edge Locations. La segunda será desarrollada en React Native aunque también de se la opción de usar DotNet MAUI.

Además este tendrá cuatro controladores, el primero para Logueado a la aplicación con usuario/ contraseña o biométrico , el segundo un controlador que se comunicaría con el BFF para operaciones de update ,transferencias, movimientos y updates del cliente , el tercero un controlador para fotos e imagenes para el uso de Rekognition y el cuarto un controlador para recibir notificaciones en front end desde Firebase Cloud Messaging.

SPA (ReactJS):

- Despliegue en AWS CloudFront con origen en S3 (Edge Locations para baja latencia).

App Móvil:

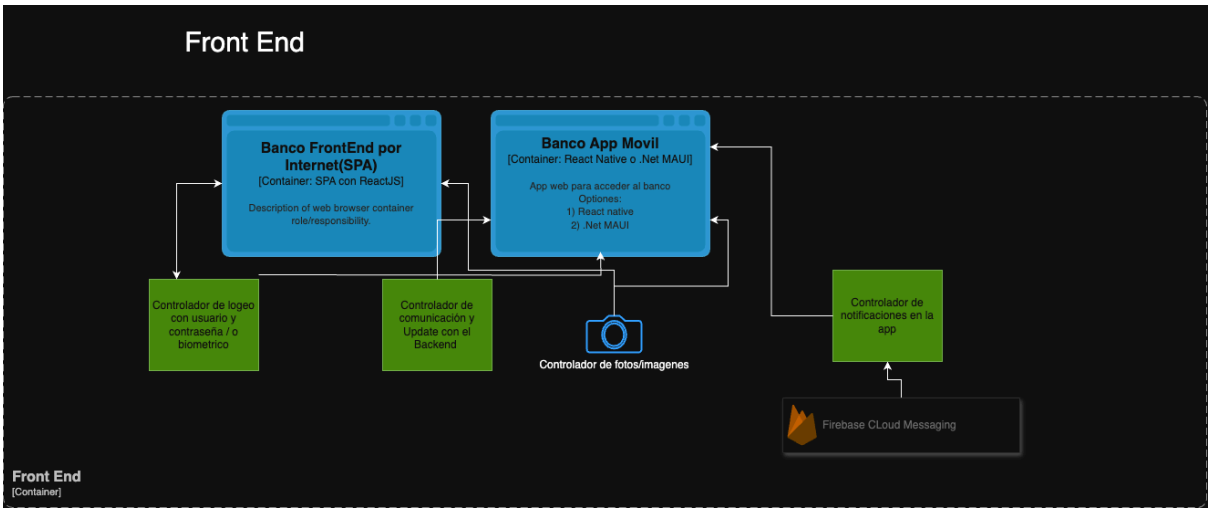
- Tecnología principal: React Native (consistencia con el SPA).
- Alternativa: .NET MAUI (para equipos con expertise en C#).

Controladores Clave:

1) Autenticación:
    a) Gestiona inicio de sesión (usuario/contraseña o biométrico).
2) Operaciones Bancarias:
    a) Comunica con el BFF para transferencias, movimientos y actualizaciones de perfil.

3) Procesamiento de Imágenes:
    a) Integra Amazon Rekognition para verificación biométrica (envío/consulta de fotos).
4) Notificaciones:
    a) Recibe mensajes en tiempo real mediante Firebase Cloud Messaging (FCM).

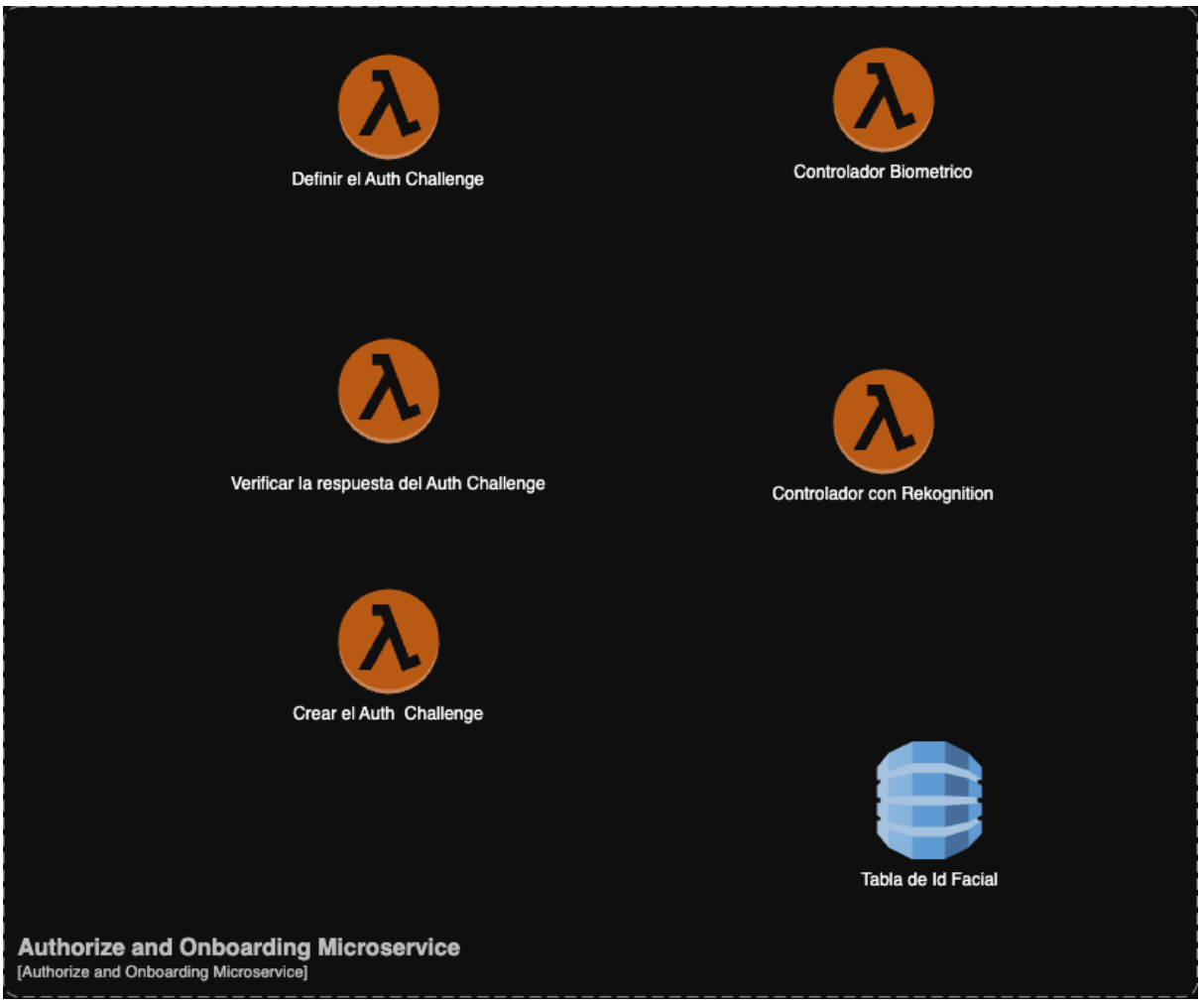
Justificación Técnica:
La separación de controladores sigue el principio de Single Responsibility (SOLID). FCM garantiza notificaciones push multiplataforma, mientras Rekognition asegura identidad con IA.



<u>Microservicio de Autenticación y Onboarding</u>
<u>Arquitectura Basada en Lambdas:</u>

El authorize and onboarding sera el microservicio encargado de lo relacionado a autorización de usuarios y el onboarding pro biométrico pues ambos flujos utilizaran amazon cognito tiene una API específica para obtención de JWT y autenticacion de usuarios , entonces usarán funciones similares. Como es un microservicio lo he dividido en lambdas: Definición de Auth Challenge, Creación de Auth Challenge,

Verificador de respuesta de Auth Challenge , Controlador de lógica Biométrica , Controlador de Amazon Rekognition.



He dividido los flujos en uno para autenticación y el otro para onboarding , sin embargo los dos son complementarios y sobreponen la lógica que usan, pero es mejor tener dos gráficos para explicarlos de manera mas ordenada:

Autenticación normal con cognito:
1) en el front end el usuario introduce sus credenciales
2) El controlador de logueo del front end envia una petición al servicio de autenticación para Amazon Cognito
3) Amazon Cognito retorna un token de acceso
4) El token de acceso se puede usar en otras aplicaciones, en este caso el BFF mas adelante

Onboarding y Bio autenticación:

1) El usuario se registra con sus datos en el front
2) el controlador de onboarding envia sus datos al microservicio de autorización y branding para registrar los datos del usuario en un user pool de Cognito
3) El usuario adjunta una foto con ayuda del controlador de fotos e imágenes del front end que se carga en un S3 Bucket.
4) La lambda de controlador biométrico de rekognition se dispara con el cambio del S3 Bucket con un buffer y la compara con la colección de Rekognition.
5) La lambda al terminar la operación de Rekognition indexa al nombre del usuario con los datos de la imagen en una tabla en Dynamo que la llamaremos Tabla de Id Facial, se debería guardar el usuario, email y los meta objetos de Rekognition.
6) Una vez que ya esta registrada la imagen y el usuario , el front end puede pedir de nuevo un correo electrónico y tomar una imagen para la solicitud de autenticación con Cognito User Pools.
7) El grupo de usuarios en cognito invocará a la lambda de definición de auth challenge, que determina el desafío personalizado a ser creado.
8) Luego el grupo de usuarios invocará la función de Crear Auth Challenge que consultará en la tabla de Dynamo llamada Tabla de Id Facial para recuperar los datos del id facial.
9) Después el grupo de usuarios invocará la función de Verificar Auth Challenge , que comprueba que la imagen que se envió en signin(paso 6) son de la misma persona, básicamente comparando dos imágenes.
10) La petición de definición del grupo de usuarios en cognito retorna con exito si la verificación fue un exito , y si es asi retorna un JWT token

Luego de estos pasos el JWT debería estar dentro front end para ser usado en las operaciones con el BFF, cabe mencionar que como es un movimiento de logueo en la aplicación el BFF lo registrará como tal en la base de datos de transacciones de cliente.

Consideraciones:

- Amazon Cognito guarda información de logueo y signin de usuarios así que datos relacionados a autenticación van a estar guardados aquí.
- Como punto de mejora importante el front end debería tener eventos para monitorear los pasos que dan los usuarios en el onboarding o autenticación por

medio de google analytics.

Auth Challenge Flow (Cognito):

DefineAuthChallenge: Determina el tipo de desafío (ej: biométrico).

CreateAuthChallenge: Genera el reto (consulta datos en DynamoDB).

VerifyAuthChallenge: Valida respuestas (ej: comparación facial vía Rekognition).

Flujo de Onboarding Biométrico:

1 Usuario sube foto a S3 (dispara Lambda de Rekognition).

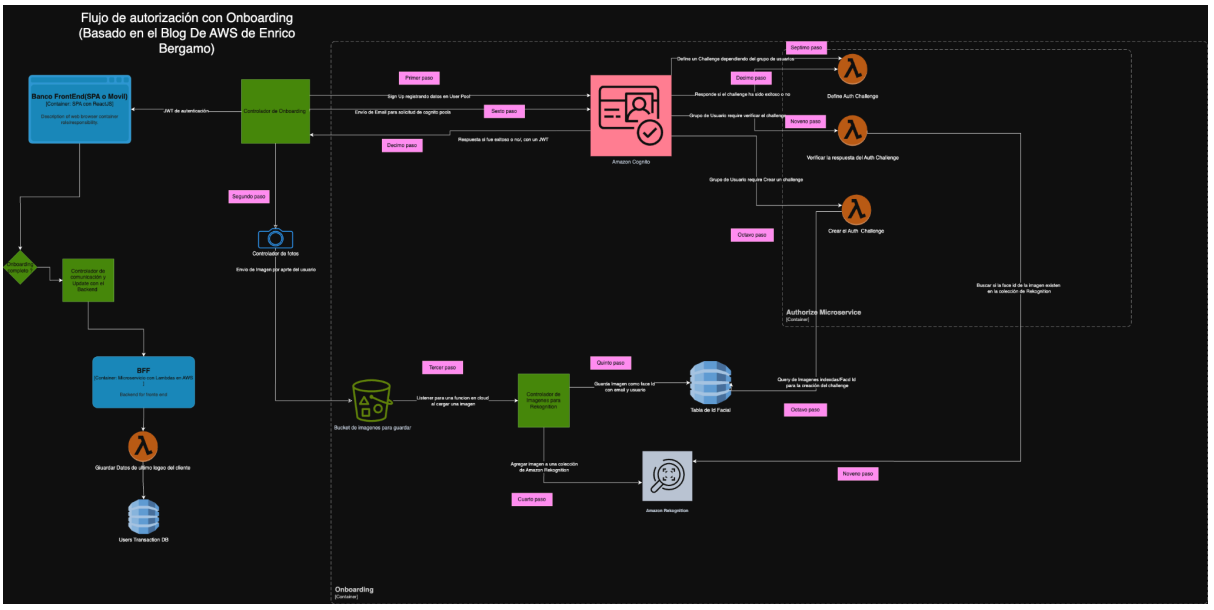
2 Rekognition indexa rostro en DynamoDB ("Tabla de ID Facial").

3 Cognito gestiona el registro y emite JWT tras verificación exitosa.

Consideraciones:

Ventaja: Flujo sin servidores (AWS Lambda + S3 + DynamoDB).

Mejora: Integrar Google Analytics para monitorear abandonos en onboarding.



Backend for Frontend (BFF): Capa de Orquestación

Componentes Principales:

El BFF es Backend for front end y se usa para mantener la lógica pesada y seguridad en la red de la VPC y no exponerlo todo al front end. Para este caso he decidido hacerlo a manera de microservicios con lambdas en AWS.
Antes de explicar que hace cada lambda, hay que entender que en AWS existe el concepto de authorizer que es una lambda que valida el token que esta pasando con el secreto del OAUTH 2.0 de Cognito que se mostró en el paso anterior , entonces para simplificar lo he llamado Controlador Authorizer.
Además en AWS para exponer una lambda al internet se necesita de agregar permisos y un Api Gateway para que se vea en el internet.

En especifico la lógica del microservicio BFF sera de manejar las transacciones interbancarias , la información del cliente, obtención de certificados bancarios y las actualizaciones que se requieran por parte del cliente en el front end. Estas las he divido en lambdas con flujos similares ya que en teoría un banco debe pasar la lógica por medio del Core Bancario, asi que estas lambdas tendrán dentro lógica de fetch y get de APIs de la red interna del banco hacia el Core Bancario. Además cada acción la tomaremos en cuenta como una transacción que el usuario esta generando la cual sera guardada en la tabla Users Transactions DB.

Uno de los requerimientos para este servicio es tener datos de cliente a la mano por medio de un mecanismo de persistencia. Es por esto que he decidido usar Redis en conjunto con dynamo DB. Aunque el uso de Dynamo DB es bastante rápido si se usa como clave secundaria al ID del usuario/cliente pra traer el histórico de movimientos y transferencias. Pero asumiendo el peor de los casos que un usuario necesite de mucho mas volumen de información en cada sesión , una base de datos Redis que se actualice con datos de dynamo db seria óptimo pues se podrian guardar datos calculados complejos como Saldos consolidados, historiales frecuentemente consultados. Además he decidido solo usar redis para operaciones de lectura de información en el endpoint de Client Information.

Por otro lado si seguimos el flujo en el grafico encontraremos el Core bancario , aqui asumo que el core bancario tendrá logica mas compleja con otros subsistemas asi

que solo tendre en cuenta que recibe peticiones y devuelve respuestas de una base de datos del core bancario y tiene la capacidad de suscribirse a una cola de notificaciones en SQS.

Authorizer Lambda:

- Valida JWT con secreto de Cognito (OAuth 2.0).

API Gateway:

- Expone Lambdas al internet con seguridad IAM.

Lambdas de Negocio:

- Transferencias, consulta de saldos, actualización de datos (todas interactúan con el Core Bancario).

Estrategia de Persistencia:

DynamoDB:

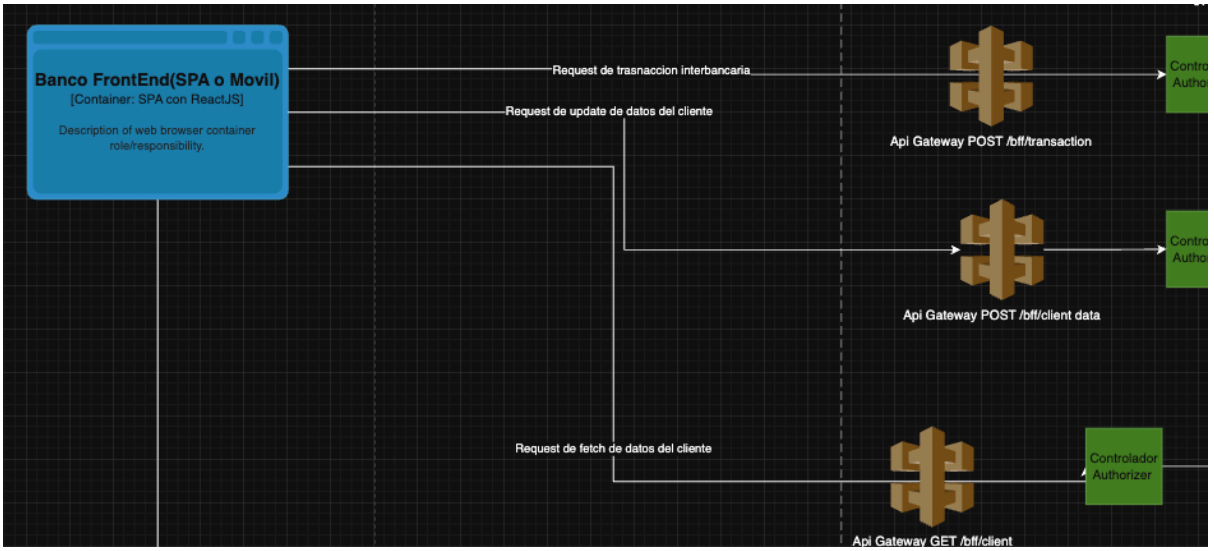
Almacenamiento principal (rápido acceso por ID de cliente).

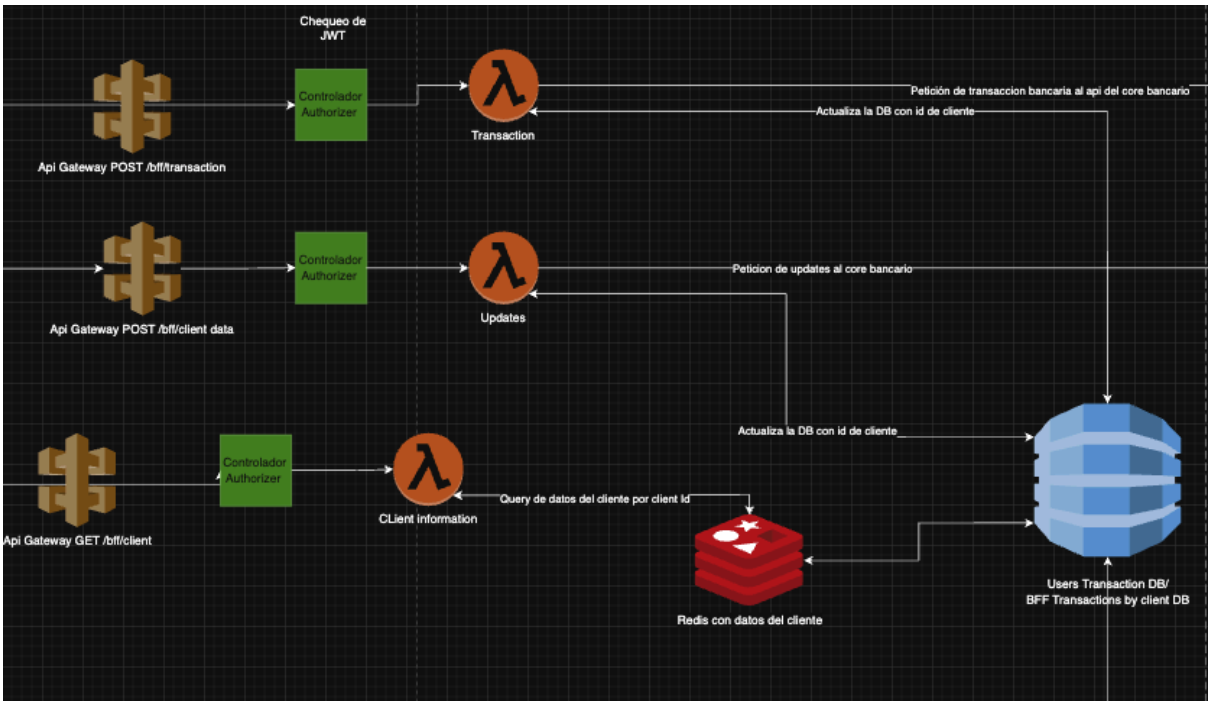
Redis:

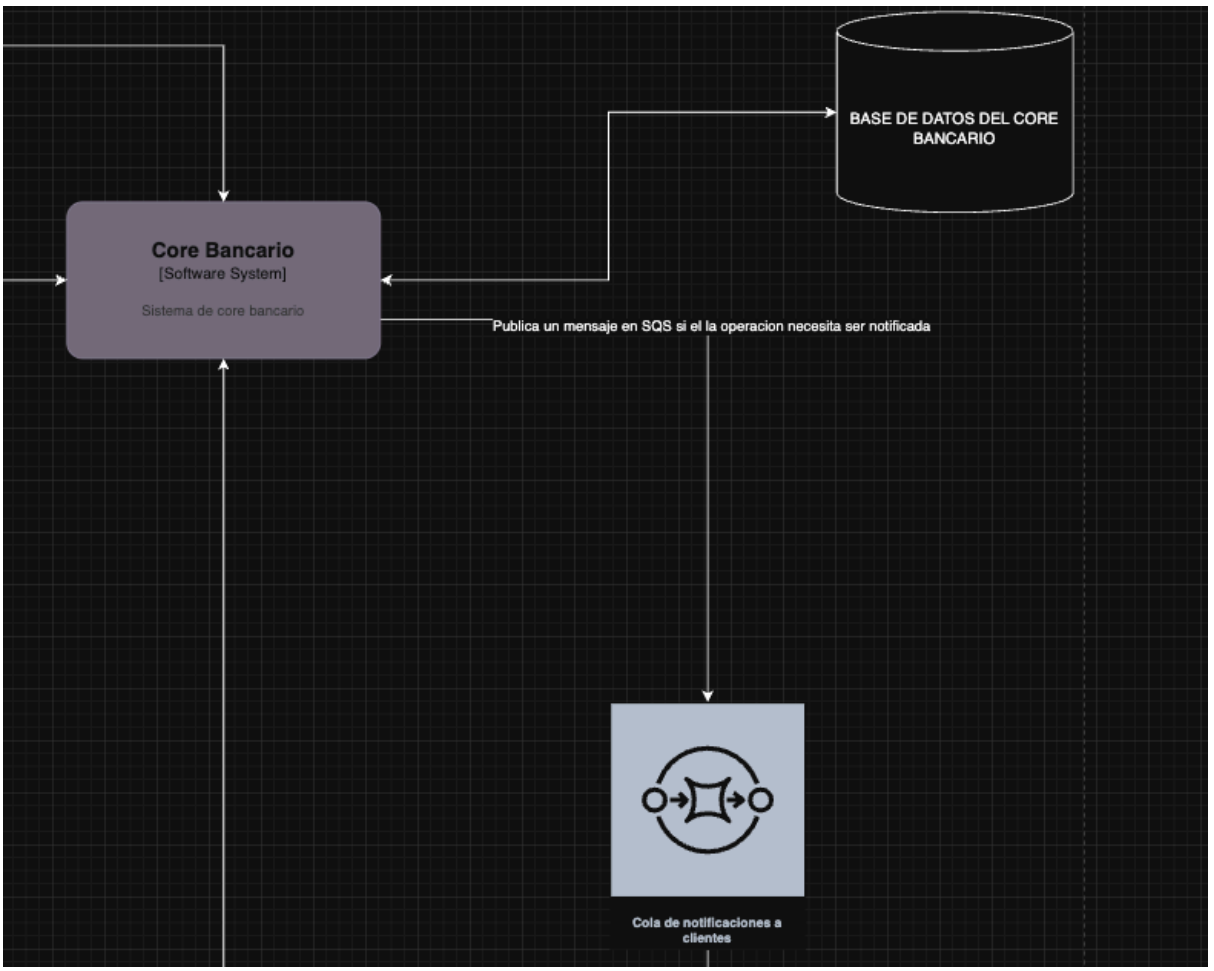
Cache de lecturas frecuentes (ej: saldos consolidados).

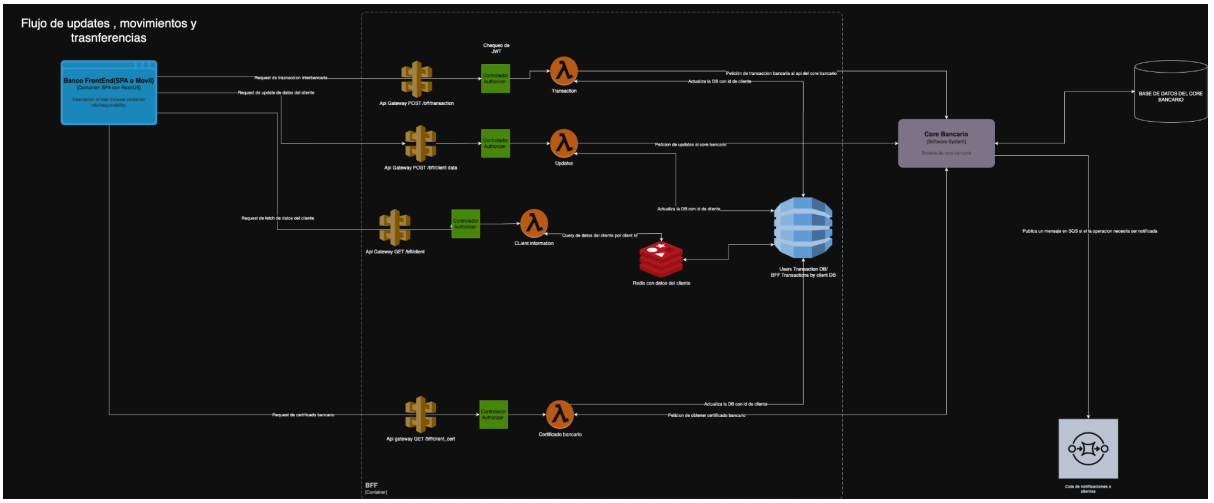
Patrón Clave:

CQRS (Command Query Responsibility Segregation): Separa operaciones de escritura (Dynamo) y lectura (Redis).


Banco FrontEnd(SPA o Movil)
[Container: SPA con ReactJS]
Description of web browser container role/responsibility.
Request de trasnaccion interbancaria
Request de update de datos del cliente
Request de fetch de datos del cliente
Api Gateway POST /bff/transaction
Api Gateway POST /bff/client data
Api Gateway GET /bff/client
Controlador Authorizer

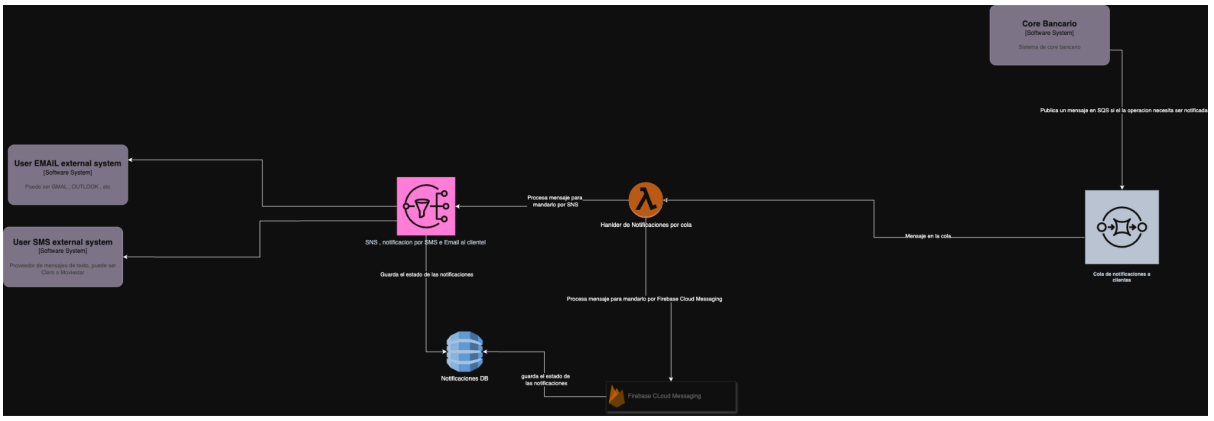

Chequeo de JWT
Controlador Authorizer
Transaction
Petición de transaccion bancaria al api del core bancario
Actualiza la DB con id de cliente
Api Gateway POST /bff/transaction
Api Gateway POST /bff/client data
Updates
Peticion de updates al core bancario
Actualiza la DB con id de cliente
Api Gateway GET /bff/client
CLient information
Query de datos del cliente por client Id
Redis con datos del cliente
Users Transaction DB/
BFF Transactions by client DB


BASE DE DATOS DEL CORE BANCARIO
Core Bancario
[Software System]
Sistema de core bancario
Publica un mensaje en SQS si el la operacion necesita ser notificada
Cola de notificaciones a clientes


Flujo de updates , movimientos y trasnferencias
Banco FrontEnd(SPA o Movil)
Chequeo de JWT
Api Gateway POST /bffctransaction
Transaction
Api Gateway POST /bfficient data
Updater
Api Gateway GET /bfficient
CLient information
Redis con datos del cliente
Core Bancario
BASE DE DATOS DEL CORE BANCARIO
Api gateway GET /bfficient_cert
Certificado bancario
BFF

Sistema de Notificaciones: Eventos Asíncronos

Para las notificaciones se usaran dos SAAS , el primero es SNS de amazon para notificaciones de email y SMS; y el segundo es Firebase Cloud Messaging para notificaciones dentro de la aplicación móvil o SPA.



Aqui como anteriormente se menciona dependeremos de que el core bancario se suscriba a una cola de Amazon SQS para poder escuchar mensajes por medio de una lambda. Esta lambda obtendrá el mensaje de la cola , lo transformara y lo enviará a un tópico de SNS y a el servicio de Firebase Cloud Messaging. Firebase Cloud Messaging necesita tambien estar instalado como dependencia en el frontend para recibir notificaciones. SNS por su aparte puede publicar via SMS al numero del usuario y tambien al Email del usuario. Para mantener datos y trazabilidad se usará una tabla de dynamo llamada Notificaciones DB.

Arquitectura Orientada a Eventos:

- Core Bancario publica mensajes en SQS (ej: transferencia completada).

Lambda de Notificaciones:

- Consume SQS, transforma datos y envía a:
- Amazon SNS (SMS/email).
- FCM (notificaciones push en app).

Trazabilidad:

DynamoDB ("Notificaciones DB") registra todos los envíos.

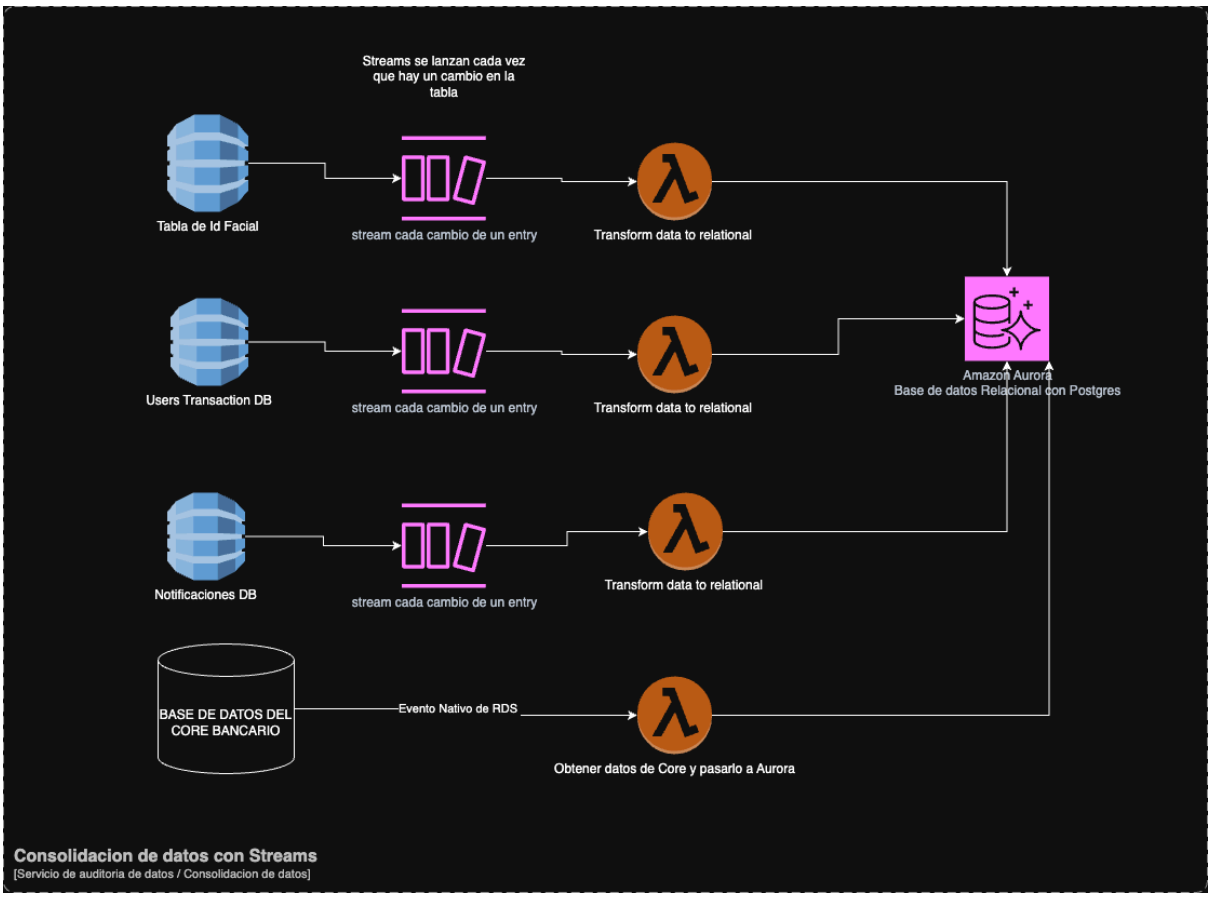
Ventajas:

Multicanalidad (SMS + email + push) cubre diversos casos de uso.

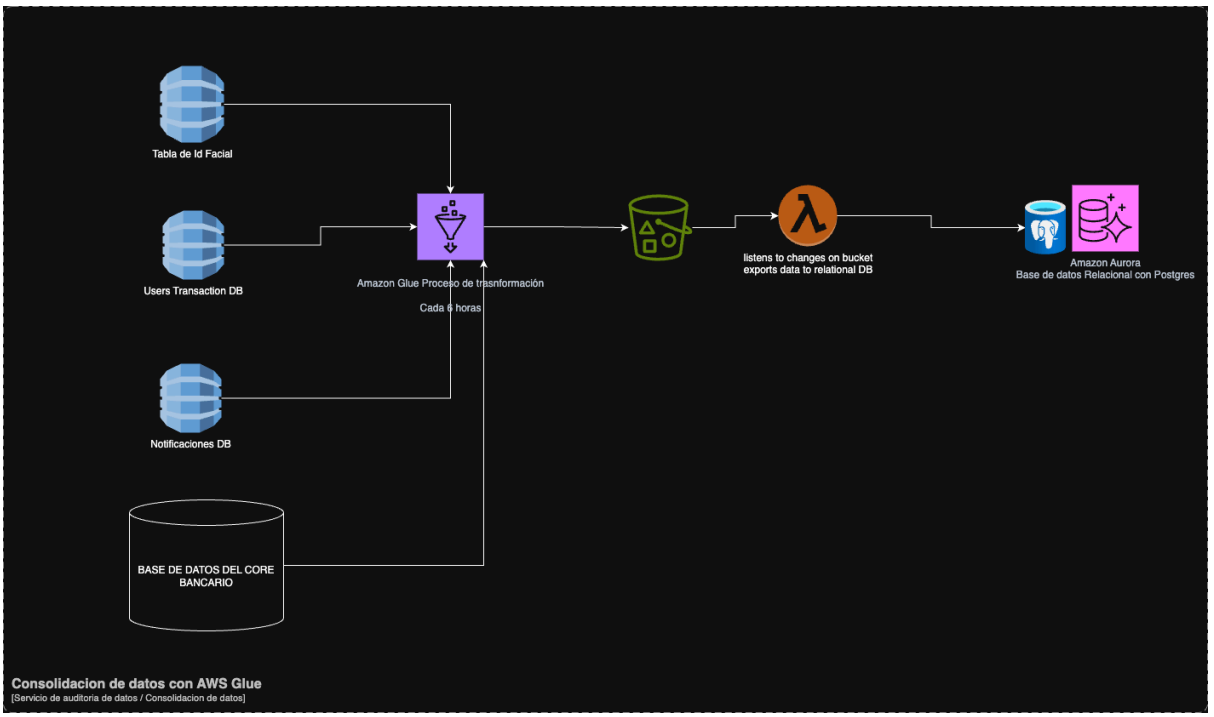
Desacoplamiento: El core bancario no necesita conocer los destinos finales.

Auditoría: Consolidación de Datos

Aquí se explica cuáles serían las estrategias para tener una base de datos de auditoría para entender todo lo que el cliente hizo históricamente y con qué sistemas ha interactuado. Aquí se hacen las suposiciones de que todos los servicios anteriormente mencionados tienen bases de datos propias que funcionan para su lógica interna de microservicios y no tenemos ninguna otra base de datos relacional a excepción del Core Bancario, sin embargo también asumiré que la base de datos del core bancario esta hosteada en AWS en un RDS(relational database).
Como tenemos varias bases de datos en Dynamo para todos nuestros microservicios existen dos posibles soluciones que recomiendo usar para tener una base de datos de auditoría.
La primera está basada en CDC(Change Data Capture) que se refiere a capturar cambios incrementales desde las bases de datos de los microservicios.
Para lograr esto se puede usar los streams de cambios de DynamoDB que tiene cada tabla , y definir lambas que escuchen estos streams y transformen los datos a tablas relacionales en Amazon Aurora con PostgreSQL. Esto también funcionaria con el Core Bancario con eventos nativos de RDS.



La segunda se basa en ETL con Amazon Glue. Este proceso es crear un proceso de transformación en amazon Glue que obtendrá todos los datos de DynamoDB y RDS cada 6 horas , luego los convertirá en un archivo de actualización de datos a un Bucket S3, el cual será escuchado por una lambda que subirá la actualización a una base de datos relacional de Amazon Aurora.



Son dos opciones que son utilizadas pero la primera es mejor para tener datos en tiempo real después de actualizaciones en las tablas de Dynamo , además que el uso de Amazon Glue puede resultar caro por la lectura de varias bases de datos y en los streams la más cara va a ser la tabla con las escrituras lo que significa que se pueden optimizar los métodos de conciliación de datos por tabla.

Opciones de Implementación:

A. CDC con DynamoDB Streams:

Flujo:

Streams de DynamoDB capturan cambios en tablas de microservicios.

Lambdas transforman datos a formato relacional (Aurora PostgreSQL).

Fortalezas:

Tiempo real. Ideal para la conciliación inmediata.

B. ETL con AWS Glue:

Flujo:

Ejecución programada (ej: cada 6 horas) para extraer datos de DynamoDB/RDS.

Transformación a S3 → carga final en Aurora.

Fortalezas:

Costo-eficiente para datos no críticos en tiempo real.

Recomendación:

Usar CDC para transacciones (ej: movimientos) y ETL para reportes históricos.