Sistema de Banca por Internet

Autor: Juan Galarza N.

juan_galarza@yahoo.com

Gestión de Cambios

Fecha	Autor	Descripción	# cambios	Comentarios
09/16/2024	Juan Galarza N.	Documento inicial		Prototipo

Tabla de contenido

Diseño del aplicativo de banca personal.

El aplicativo de sistema de banca por internet, conocido como system context in C4, está formado por los siguientes contenedores:

+ FrontEnd
+ API Gateway
+ Movimientos
+ Pagos
+ Perfil de usuario

1. Aplicación móvil

Framework

Esta aplicación será desarrollada con el **framework flutter**.

La selección de flutter es porque es un framework multiplataforma, ligero, moderno y fuertemente soportado por Google.

E incluye, nativamente, características tales como reconocimiento facial, requerido por el sistema.

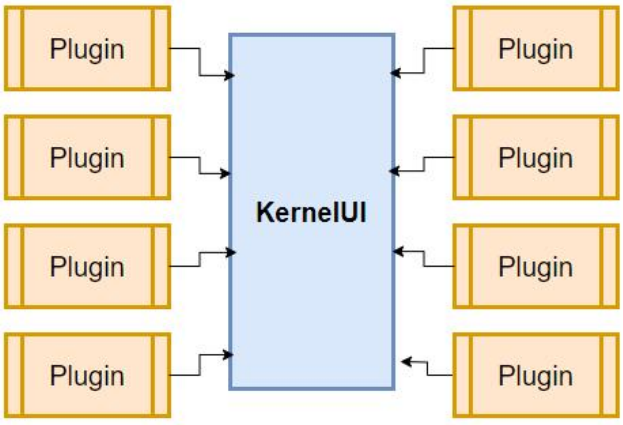
La arquitectura de la aplicación móvil incluirá una capa de abstracción, el **KernelUI**, el cual va a encapsular todas las llamadas.

En caso de no utilizar flutter, **la siguiente alternativa sería React Native**.

La selección de React Native es porque es un framework maduro, estable y ampliamente soportado.

Además al utilizar JavaScript como lenguaje de desarrollo reduce significativamente la curva de aprendizaje, en comparación con flutter.

Arquitectura



La arquitectura a utilizar será modular, de **microkernel** o basada en plugins.

Existirá un módulo Core, el microkernel o **KernelUI**, el cual encapsulará todas las llamadas de gestión del resto de módulos.

Las ventajas de utilizar plugins son seguridad, estabilidad, normalización, encapsulamiento, desarrollo paralelo y actualización selectiva.

El desarrollo con arquitectura de microkernel puede reducir los tiempos de desarrollo en un **30%**, en equipos de desarrollo medianos.

Módulos

Toda la funcionalidad será proporcionada a manera de módulos, o plugins, respetando los principios **SOLID**.

NOTA
No todos los módulos tienen un UI propio, ni son accesibles directamente por el usuario.
Algunos módulos funcionan como bibliotecas de recursos, exponiendo funcionalidad a otros módulos, y algunos módulos son específicos de una plataforma, iOS, Android, SPA Web.

Además del KerneUI, los siguientes módulos son sugeridos:

Inicio
Es responsable del Onboarding y página inicial, de resumen del usuario. Invoca, internamente, la funcionalidad del módulo de **Login**.Almacena la lógica para reconocimiento facial.

Login (Biblioteca)
Encapsula el flujo OAuth2.0 de autorización del usuario. El flujo recomendado, según la norma, es: *Authorization Code Flow with Proof Key for Code Exchange (PKCE)*.

Cuentas
Presenta las cuentas del usuario y permite realizar consulta de movimientos en las cuentas.

Perfil
Permite al usuario ajustar sus preferencias e información personal. Nombre, fotografía, email, cambio de contraseña, etc.

Notificaciones
Consulta periódicamente, y presenta, las notificaciones del sistema.

Transferencias
Provee la funcionalidad necesaria para realizar pagos y transferencias.

Branding (Biblioteca)
Provee los recursos gráficos, colores y hojas de estilo para reforzar el branding del banco.

Common (Biblioteca)
Provee código de propósito general, reutilizable por el resto de módulos.

2. Aplicación Web SPA

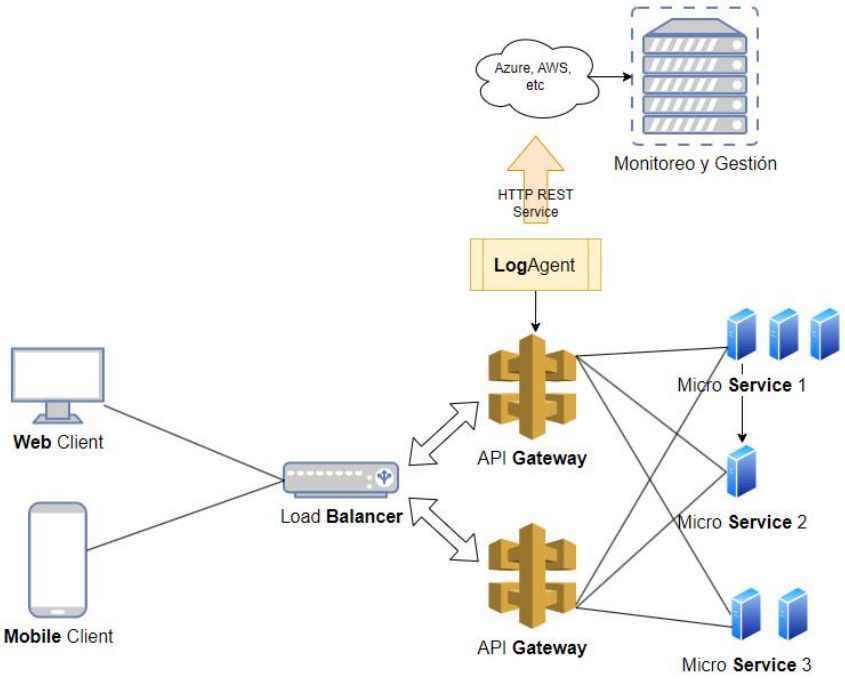
Esta aplicación será desarrollada con el framework flutter. La selección de flutter es porque, al ser utilizado ya en la aplicación móvil, nos permite reutilizar la mayor parte de la funcionalidad desarrollada, 90% aproximadamente, y tener una librería común de fácil mantenimiento.

En caso de que se utilice React Native en la aplicación móvil, se utilizará **React Native for Web**, en la aplicación SPA.

La arquitectura y conjunto de módulos **es la misma** que para la aplicación móvil.

3. Backend API

Arquitectura



El Backend será implementado utilizando la **arquitectura de microservicios**.

La adopción de la arquitectura de microservicios permiten escalar la aplicación, selectivamente, en los servicios que tengan un mayor impacto en rendimiento.

Además de beneficios tales como la modularidad y encapsulamiento.

Comunicación

La comunicación desde el Frontend al Backend será a través del protocolo **HTTP2/REST**, implementando el patrón de diseño **Command**.

El API Gateway funcionara como un **Command Receiver** y el Frontend será un **Command Invoker**.

La comunicación basada en comandos simplifica la implementación de un **Unified Logging System**, con funcionalidades tales como:

+ Application Usage
+ Logs
+ Auditing
+ Recovery
+ Session Tracing
+ Automated Testing

Microservicios

Por cuestiones de seguridad el API Gateway **no** se comunicará directamente con servidores internos del banco, AS/400, RDBMS, etc, sino a través de los microservicios creados para este fin.

La comunicación entre el API Gateway con los microservicios se realizará utilizando **HTTP3/REST**, para un rendimiento mejorado.

Para efectos de normalización, los microservicios son responsables de exponer la **misma interfaz** de procesamiento de comandos, **Command Receiver.**

Internamente realizarán las invocaciones correspondientes, de acuerdo al comando recibido.

NOTA
Todos los microservicios deben reportar al **LogAgent**, independientemente, errores, alarmas, y las métricas **relevantes** a su operación específica.

Inicialmente, los siguientes microservicios han sido considerados:

Perfil
Gestiona la información básica del usuario, incluyendo foto de perfil, nombre, direccción, **notificaciones del sistema**. En caso de no utilizar un **SSO** externo, también gestionará cambios de contraseña.

Esta información puede precargarse en caché durante el proceso de login para una mejor **experiencia de usuario**.

Movimientos
Realiza consultas, de solo lectura, a la plataforma correspondiente, con información detallada de los movimientos. Puede manejar criterios de filtrado, ordenamiento y paginación de datos.

Esta información **no** debería almacenarse en caché, ya que su contenido es **volátil**.

Transferencias
Ejecuta transacciones de acreditación, pagos y transferencias. Puede tener que comunicarse con plataformas de pagos de terceros, tales como servicios básicos, operadores de tarjetas o pasarelas de pago.

Considerando una cantidad variable, y creciente, de proveedores de pagos, se recomienda que funcione como Frontend para un **PaymentGateway**.
Cuando su operación es asíncrona, puede encolar un mensaje o ejecutar un hilo independiente (no muy recomendado) y devuelve un mensaje de confirmación inmediatamente al usuario.

Éso mantiene los tiempos de espera cortos y la buena experiencia del usuario

Una vez que el proceso termine, exitosamente o fallido, enviará una notificación al usuario, a través de microservicio de Perfil.

NOTA
El **PaymentGateway**, en caso de existir, debería ser modular, extensible, resilente, y de **muy alto rendimiento**. Su definición está fuera del alcance de este análisis.

Crash Recovery

A pesar de ser el único punto de contacto con los clientes, la carga de procesamiento del API Gateway se espera que sea mínima. Sin embargo, se puede empezar con un grupo de instancias del API Gateway sirviendo bajo un esquema de balanceo de tipo **Round-Robin**.

Si bien hay múltiples algoritmos de balanceo, Round-Robin es la mejor alternativa para manejar **tolerancia a fallos** en un entorno de instancias homogéneas, dentro de la **misma ubicación geográfica**.

Dependiendo de las capacidades del Load Balancer, se puede habilitar AutoSpawn, en caso de falla en una de las instancias.

Si el **Load Balancer** no soporta AutoSpawn, pero las instancias están corriendo bajo un super servidor, tal como **inetd** en Linux o **IIS** en Windows, éste será responsable de iniciar nuevas instancias en base a las necesidades.

Si ninguna de estas opciones está disponible, es necesario habilitar un proceso de monitoreo independiente, responsable de monitorear, reportar y relanzar nuevas instancias, en caso de falla.

El sistema proveerá un EndPoint de tipo **Heart-Beat**, el cuál podrá ser consumido por los agentes de monitoreo interesados en verificar el estado del servicio.

API Gateway

El API Gateway recibe todas las peticiones de los clientes remotos a manera de comandos, y las reenvia a los servicios correspondientes.

Algunas de sus funciones incluyen:

Validación de entrada
La utilización de comandos obliga a la normalización de llamadas, ofreciendo una primera línea de defensa contra ataques por llamadas desnormalizadas tales como: SQL Injection, DoS, buffer overrun, etc.

Canalización de llamadas
El API Gateway es responsable de recibir y canalizar las llamadas entrantes hacia el servicio correspondiente. En casos de alta demanda, la canalización puede ser hacia un grupo de instancias de servicio.

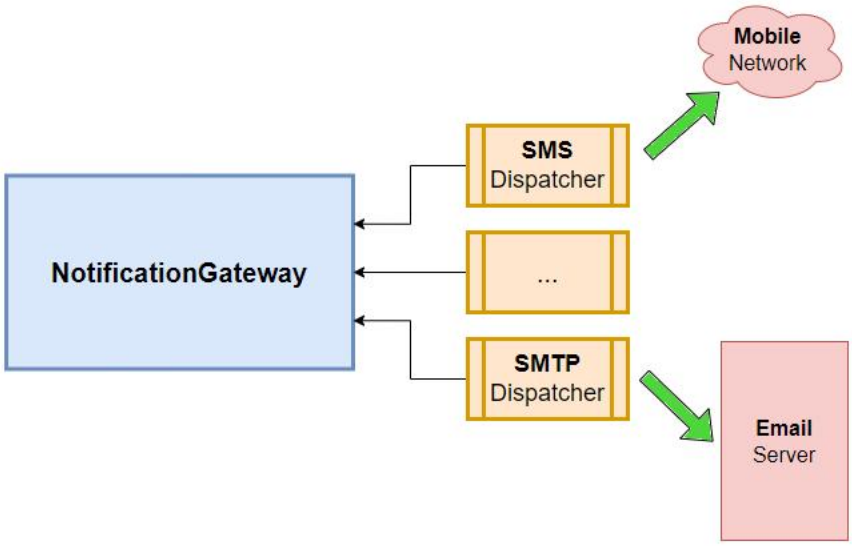
Hot Reload
Para un manejo simplificado y alta disponibilidad, el API Gateway atenderá activamente a modificaciones en su archivo de rutas. Ésto permitiría hacer actualizaciones de servicios y rutas "en caliente" sin tener que dar de baja el servicio.

Monitoreo

Todos los comandos emitidos durante la sesión de un usuario son enviados al servidor de monitoreo a través del **LogAgent**, para efectos de diagnóstico, seguimiento, auditoría y recuperación.

Para proteger la identidad del usuario, los mensajes viajarán con el id de la sesión, la cuál no puede mapearse directamente a un usuario específico, y ocultando información sensible.

4. NotificationGateway



En un sistema que involucra múltiples plataformas la solución ideal sería tener un bus empresarial que comunique todos los nodos.

Si ya hay un bus empresarial instalado y conectando todos los centros de notificaciones, Email Server, SMS Gateway, el proceso de notificación a usuarios sería enviar un mensaje a los nodos correspondientes.

La instalación de un bus empresarial y su correspondiente período de estabilización es un esfuerzo costoso, en tiempo y dinero, y está fuera de este alcance.

Si no hay un bus instalado la siguiente alternativa es crear un NotificationGateway utilizando arquitectura de plugins.

El NotificationGateway trabajará con los canales de comunicación vigentes hoy, habilitados a manera de plugins, y será extensible hacia nuevos canales que puedan surgir en el futuro, implementando nuevos plugins.

NOTA
Las aplicaciones móviles, y SPA, soportan notificaciones del sistema. Es decir que las notificaciones serían un plugin más dentro de los canales del disponibles para el NotificationGateway.

5. Monitoreo y Gestión

El sistema de Monitoreo y Gestión es una aplicación de Monitoreo **independiente**, que puede estar alojada en otro servidor, quizás en la nube, en otra ubicación geográfica.

El sistema de gestión involucra los siguientes componentes clave:
+ LogAgent
+ LogService
+ Dashboard Unificado

LogAgent

El LogAgent es un singleton que se ejecuta de forma independiente y va recopilando información de uso de aplicación, profiling y errores.

Esta información es encolada y, periódicamente, cada 5 minutos, envía toda la información al servidor de monitoreo.

En el caso de información crítica, errores de ejecución, por ejemplo, el intervalo de envío es menor. Puede ser cada minuto.

No se recomienda el envío inmediato de mensajes al servidor, porque éste puede saturarse en caso de un error de recursividad infinita o un DDoS.

LogService

LogService es un servicio **HTTP/REST** expuesto en el servidor de gestión para recibir los datos de **profiling**, uso de aplicación y errores.

Este servicio será consumido remotamente por instancias del LogAgent, corriendo en distintos contextos y servidores, y por invocaciones directas, desde otros ambientes fuera del ámbito del LogAgent, tales como Linux, DOS, Solaris, PL/SQL, etc.

El LogService es la interfaz de entrada para el Dashboard unificado.

Dashboard Unificado

La mayoría de plataformas modernas proveen su propia consola de gestión y monitoreo. Incluso las más antiguas proveen alguna forma de monitoreo basada en archivos de log, o triggers.

Cuando queremos monitorear la salud de nuestra aplicación tenemos dos opciones:

Accedemos a las diferentes consolas de administración de las distintas plataformas intervinientes o creamos una **consola unificada** que las abarque a todas.

El Dashboard Unificado representa la segunda opción. Nos permite monitorear el estado de nuestro sistema (y todas las plataformas intervinientes) y, de ser el caso, tomar medidas correctivas, de forma **proactiva**.

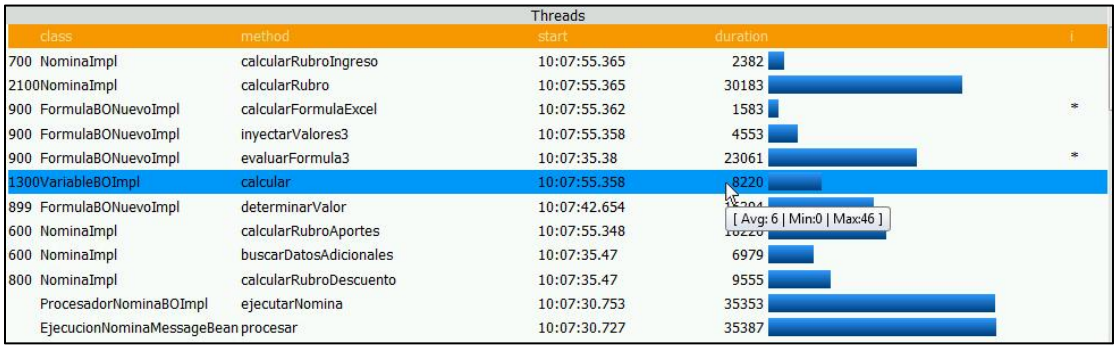
Identificando y resolviendo un problema, antes de que se convierta en un problema.

NOTA
El Dashboard Unificado no crea la lógica completa de consola para cada plataforma, sino conectores con la lógica mínima necesaria para ingerir datos desde las consolas actuales de cada plataforma, a fin de presentarlos de forma normalizada.

PerformanceCounters

El sistema presentará gráficos de carga y rendimiento de todas las plataformas y contextos intervinientes, a fin de poder **identificar rápidamente** cualquier evento de caída o saturación.

ProfileViewer



Permite revisar los tiempos de ejecución de los diferentes procesos intervinientes en una transacción a fin de poder realizar el diagnóstico y optimización correspondientes, en las áreas más sensibles, que demuestren mayor lentitud.

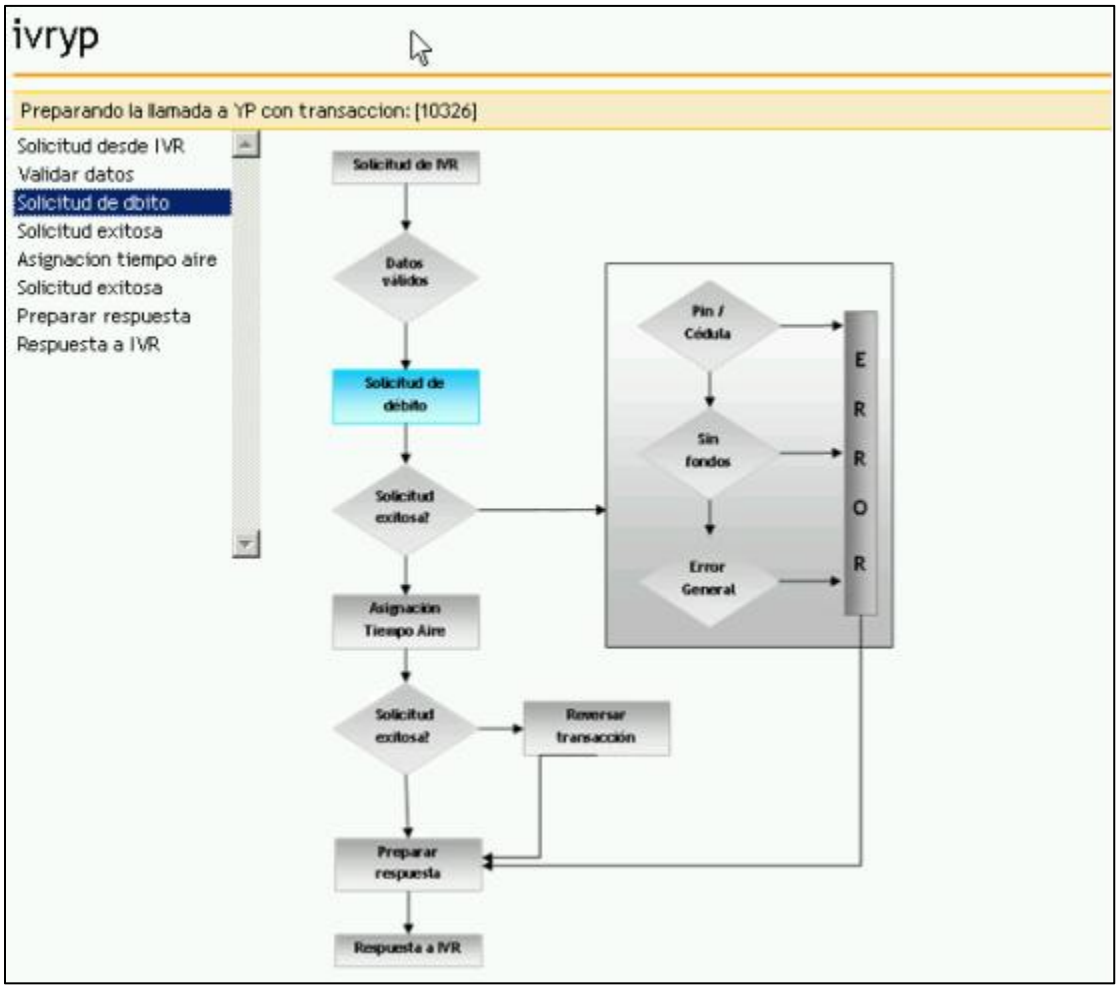
Esta característica corre en Producción, permitiendo **Live Profiling**, por lo cual debe habilitarse sólo cuando sea necesario, para evitar tráfico innecesario.

LogViewer

Permitirá visualizar los últimos mensajes de log reportados por los distintos contextos y plataformas, de forma categorizada. Log de errores, información y depuración.

Es útil para identificar errores de código, **excepciones no controladas**, en ambientes de producción.

SessionTracing



El Dashboard Unificado tendrá una opción para consultar las últimas n sesiones visualmente, a manera de flujograma, para identificar posibles errores.

Protegiendo la identidad del usuario, será posible dar seguimiento a todos los pasos realizados durante su sesión, consultando el LogStore.

LogStore

Es una base de datos que mantiene toda la información de logs y auditoría generados por el sistema.

Permite el registro, consulta y seguimiento de las sesiones de los usuarios así cómo todos los contadores de rendimiento, notificaciones y errores reportados por la aplicación.

Al llevar un registro de las transacciones puede utilizarse como herramienta de recuperación de errores catastróficos.

NOTA
La comunicación hacia el Dashboard debería estar restringida por IP de origen e, idealmente, accesible solo a través de una **VPN**.

Proveedores de servicios en la nube

Azure, AWS o proveedores de Cloud

Desde el punto de vista de rendimiento, no hay beneficio en mantener la aplicación corriendo en Azure, AWS o cualquier otro proveedor de nube externo.

Asumiendo que la mayoría de los clientes de el banco residan en Ecuador, mantener los servicios **localmente** y con conexiones simétricas, redundantes a distintos ISP locales, conectados al **NAP.EC**, ofrecería los mejores tiempos de latencia para los usuarios.

Incluso la normativa vigente, sugiere mantener información sensible de los usuarios, dentro del país.

Considerando el presupuesto disponible para el Cloud, se podría manejar un esquema de **red híbrida**, con el servicio de **Monitoreo y Gestión** alojado en el Cloud, ya que cumple con las características de ser tráfico no crítico, puede ser de alta latencia, y que provee **Disaster Recovery**.

Se puede mantener cuentas Blob storage para enviar copias nocturnas, encriptadas, de las base de datos, para efectos de protección geográficamente distribuida.

No se especifica la RDBMS a usar, pero la mayoría de ellas ofrecen facilidades de replicación, **off-site**, con **actualizaciones diferenciales**.

Es recomendable crear un proceso de recuperación automática, que permita regenerar la base de datos a partir de la copia nocturna, de la base de datos, y el log generado periódicamente, durante el día.

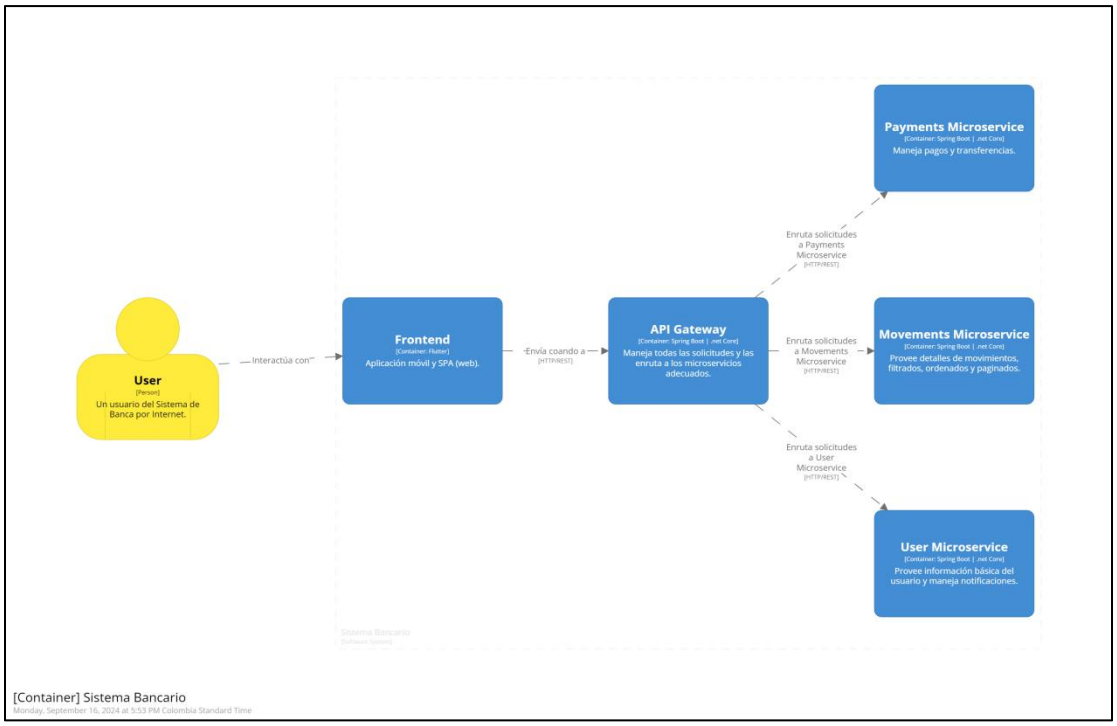
El proceso de restauración automático, incluiría un mecanismo de **Heart-Beat**, el cual detectaría fallos y, proactivamente, empezaría la configuración de un nuevo servidor de base datos con la información actualizada.

CloudFront, Akamai o proveedores de CDNs

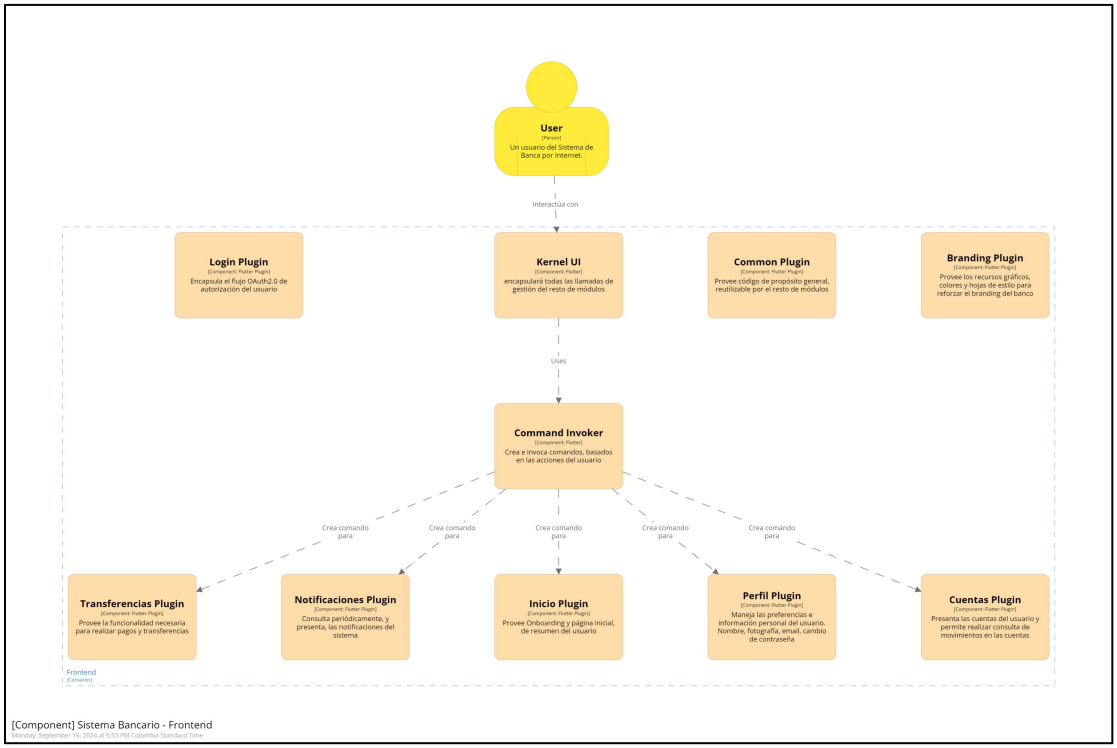
Considerando que la mayor parte del contenido será generado dinámicamente, de forma transaccional, no hay mayor beneficio en mantener un sistema de caché con Akamai, o CloudFront, los cuales están conectados directamente al NAP.EC.

C4 MODEL DIAGRAMS

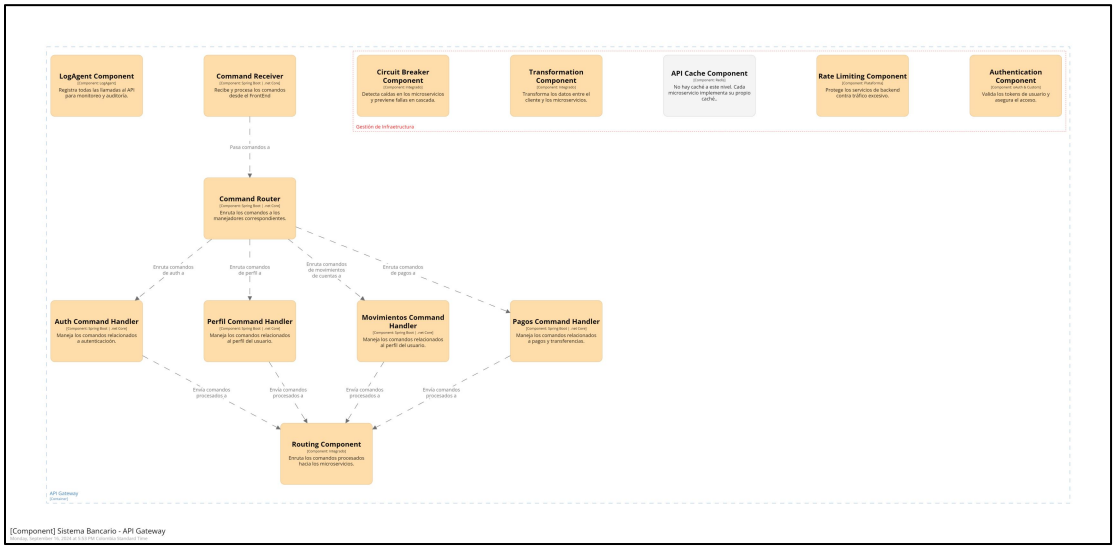
SYSTEM CONTEXT: Banca Móvil



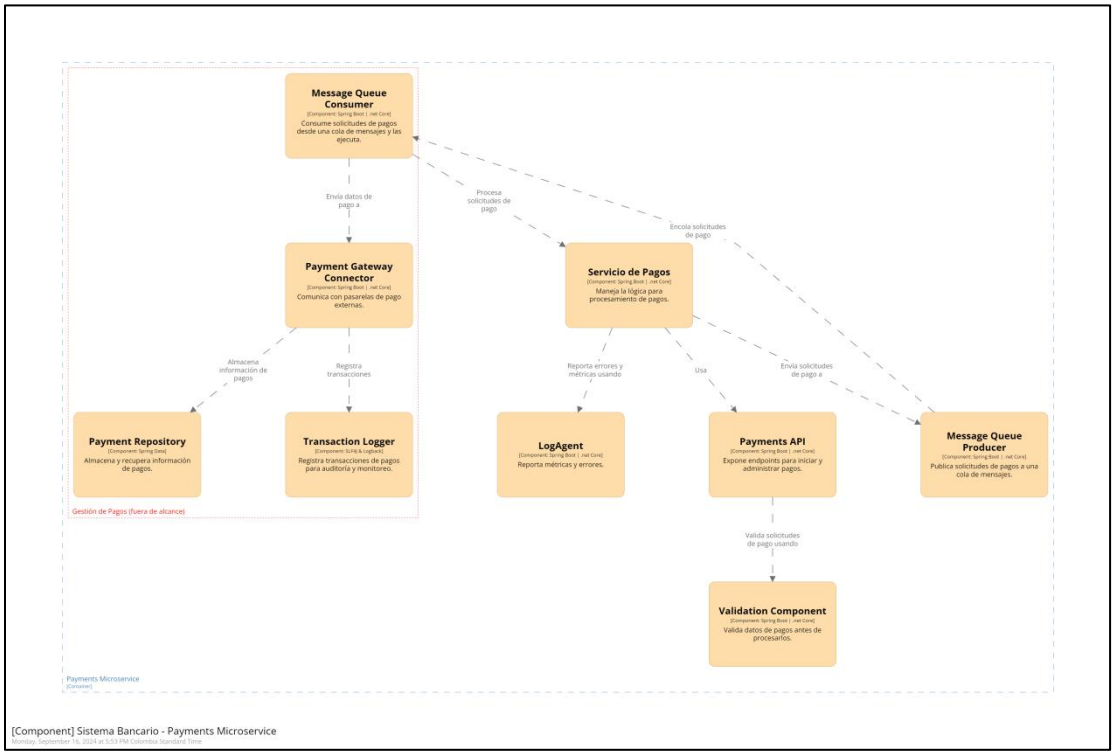
COMPONENT: FrontEnd



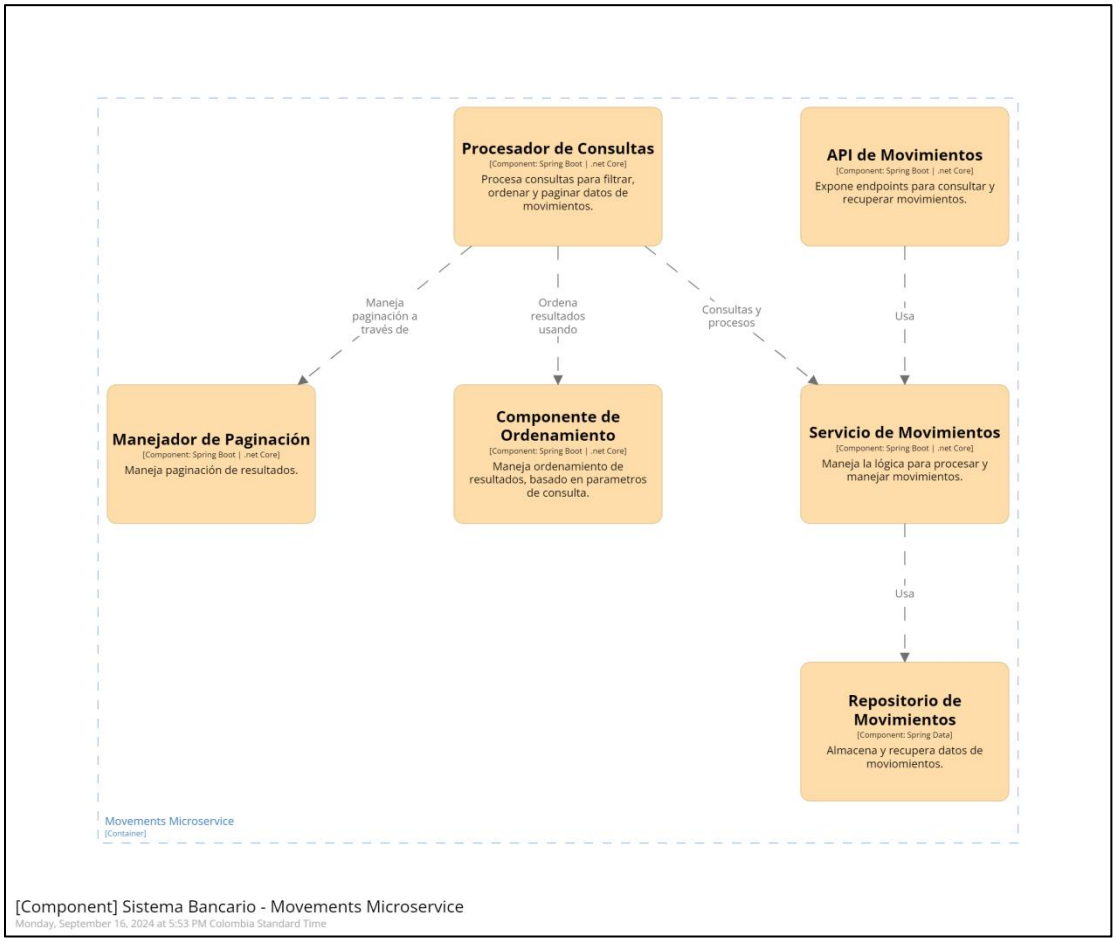
COMPONENT: API Gateway



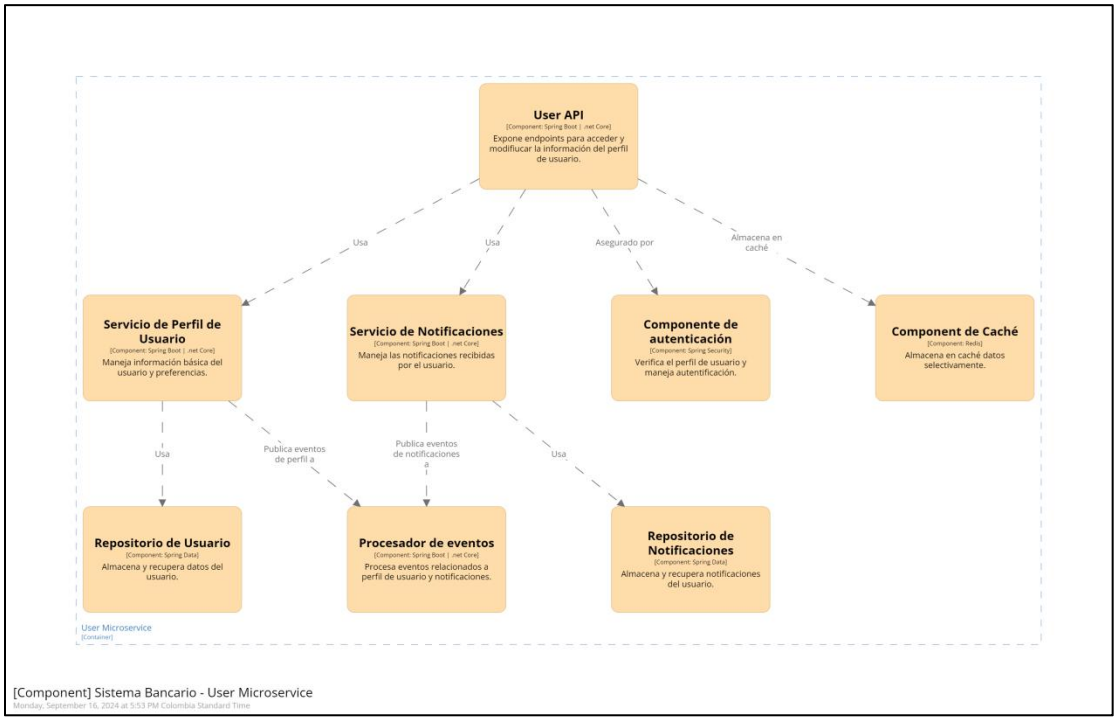
COMPONENT: Pagos



COMPONENT: Movimientos



COMPONENT: User Profile



References

https://www.jetbrains.com/help/kotlin-multiplatform-dev/cross-platform-frameworks.html

https://trends.stackoverflow.co/?tags=flutter,react-native,ionic-framework,nativescript,kotlin

