SISTEMA DE BANCA POR INTERNET

Desarrollo de ejercicio práctico

Descripción breve

Resolución del caso propuesta en Azure

Richard Leonardo Berrocal Navarro
Richard.berrocal@gmail.com / asesor@richardberrocal.com

Tabla de contenido

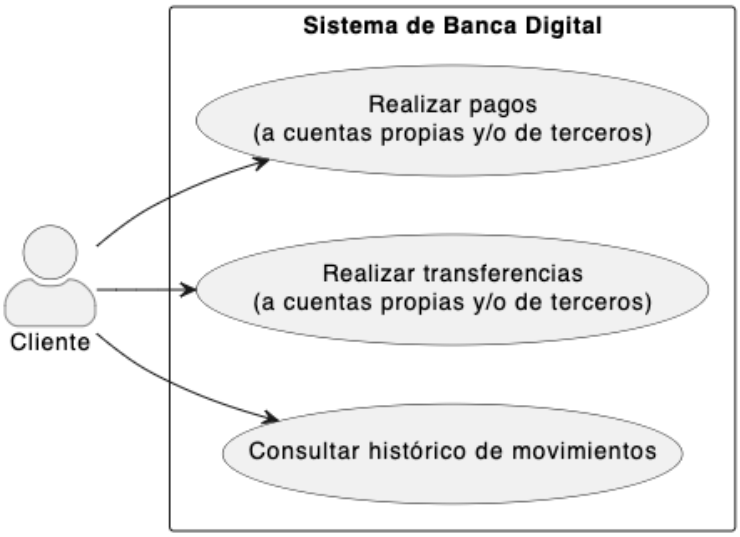
Sistema de banca por Internet

Funcionalidades relevadas con el usuario

1.Consultar histórico de movimientos

Tanto a cuentas propias y/o de terceros podrá:
2.Realizar transferencias
3.Realizar pagos

Se muestra el siguiente diagrama de casos de uso:



Mapeo de ecosistema de aplicaciones

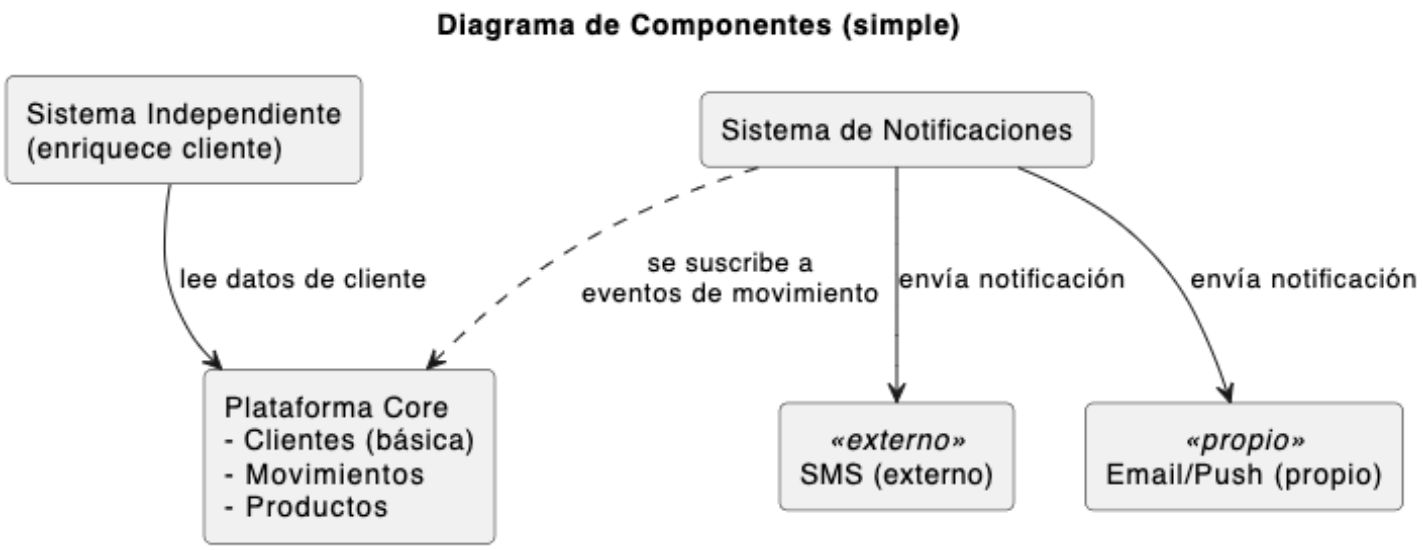
Plataforma Core:

- Información basica de clientes
- Movimientos
- Productos

Sistema Independiente;

- Completa informacion de cliente

Sistema de notificaciones
Por normativa, los usuarios deben ser notificados sobre los movimientos realizados. El sistema usará al menos dos mecanismos (externos o propios) de envío de notificaciones.



Detalle de solucion

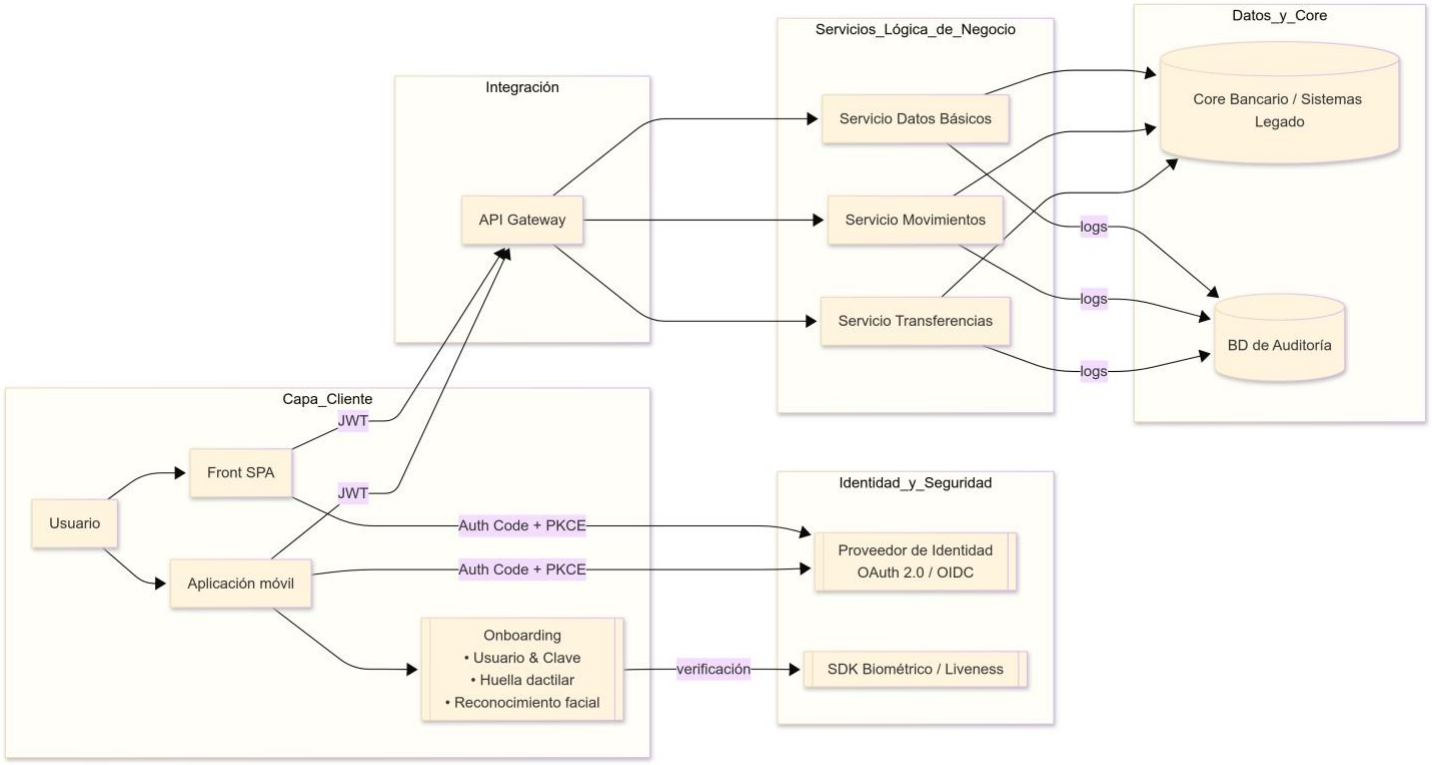
Front

SPA: Single page application
Oauth 2 flujo de autenticación

Aplicación móvil
- On Boarding
  - Usuario y clave
  - Reconocimiento de huella dactilar
- Reconocimiento facial

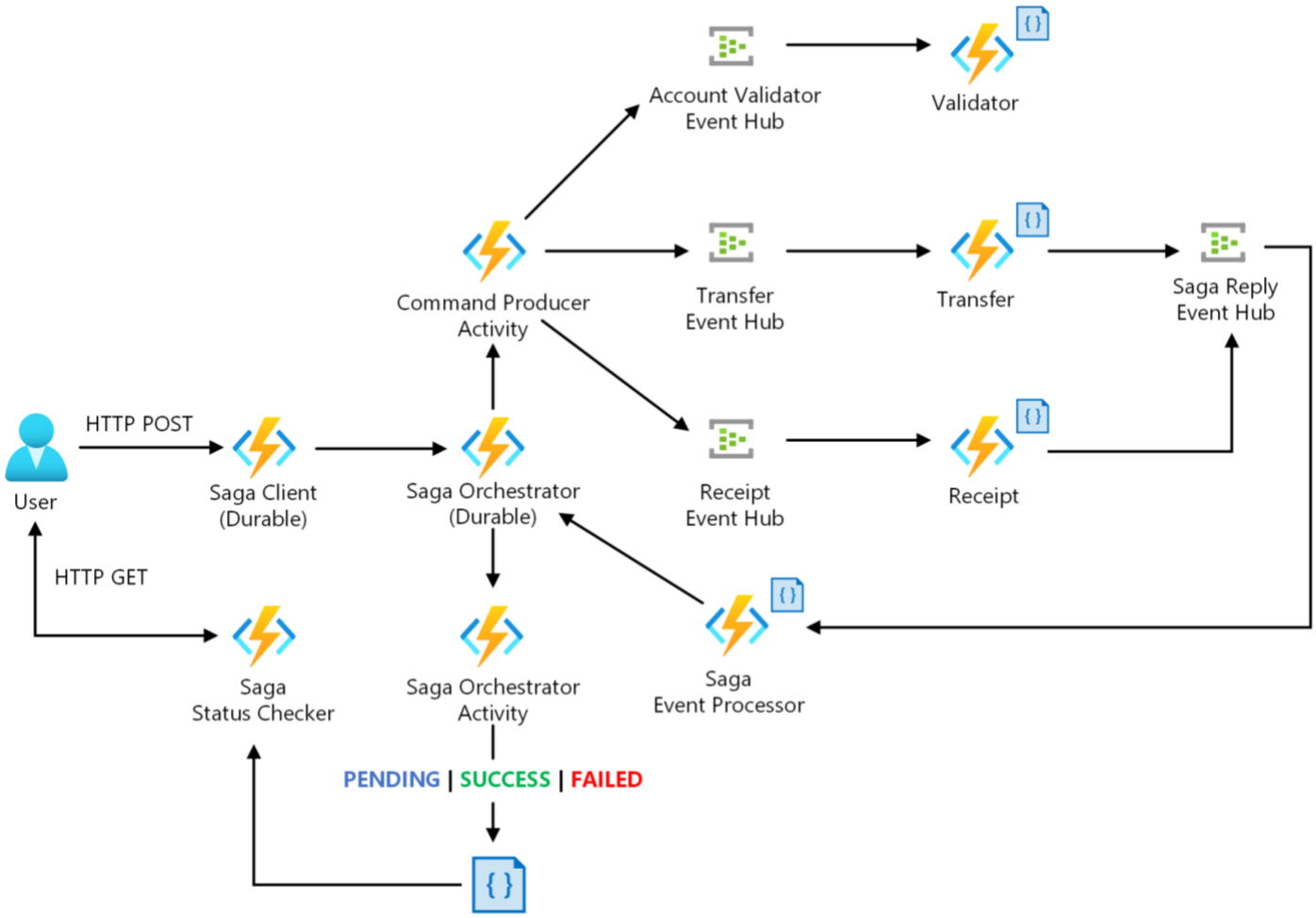
Auditoria
Basado en patrones de diseño

Api Gateway
- Consulta de datos basicos
- Consulta de movimientos
- Transferencias



Uso de Patron Saga:

Se muestra un diagrama de arquitectura que muestra el flujo del patron Saga en Azure.



Consideraciones:

LPDP
HA Tolerancia a fallos
DR
Seguridad y monitoreo
Auto-healing
Excelencia operativa
Garantizar baja latencia
Arquitectura desacoplada
Elementos reusables
Elementos cohesionados

Puntos que resuelve el diagrama:

Consistencia entre múltiples servicios. La Orchestrator (Durable) coordina pasos y compensaciones si algo falla.

Tolerancia a fallos y reintentos: cada Activity y cada consumidor de Event Hub puede reintentarse con backoff; el Orchestrator guarda historial y reanuda.

Experiencia de canal predecible: el **Saga Client** recibe un sagaId y el **Status Checker** permite polling hasta SUCCESS|FAILED, evitando timeouts largos en el front.

Desacoplo fuerte: "Command Producer Activity" publica a *Event Hubs* (topics por dominio: Validation, Transfer, Receipt) y cada microservicio evoluciona sin acoplar contratos síncronos.

Componentes:

Saga Client / Orchestrator (Durable Functions): inicia y coordina Saga; mantiene el estado durable por sagaId.

Event Hubs: buses de dominio (Account Validator EH, Transfer EH, Receipt EH, Saga Reply EH); clave para pub/sub, particionado por clave de negocio (p. ej., cuenta).

Validators / Transfer / Receipt (Functions/Containers): consumidores idempotentes de comandos; publican eventos de éxito/fracaso al Saga Reply.

Saga Event Processor: consume respuestas y notifica al Orchestrator para cerrar la saga o disparar compensaciones.

Sustento de la propuesta en banca:

Resiliencia y continuidad
Durable Functions replay + checkpointing: la orquestación sobrevive reinicios/escala.
Reintentos, timeouts y compensaciones modelados explícitamente.

Escalabilidad elástica y costo

Serverless: Functions + Event Hubs escalan por carga (picos de TEF/transferencias) y se paga por consumo.

Idempotencia y orden por clave
Con Event Hubs, particiona por cuenta/cliente para conservar orden local; combínado con idempotency keys (p. ej., reqId) para "al-menos-una-vez".

Observabilidad y auditoría
Durable history (+ Application Insights, OpenTelemetry) ofrece línea de tiempo auditable por sagaId (cumplimiento & forensics).
Métricas por etapa (validación, débito, crédito, recibo) → SLO's transaccionales.

Evolución y bajo acoplamiento
Nuevos pasos (p. ej., AML/Fraude**,** Límites o PSP alterno) se añaden como Activities/consumidores sin romper al resto.

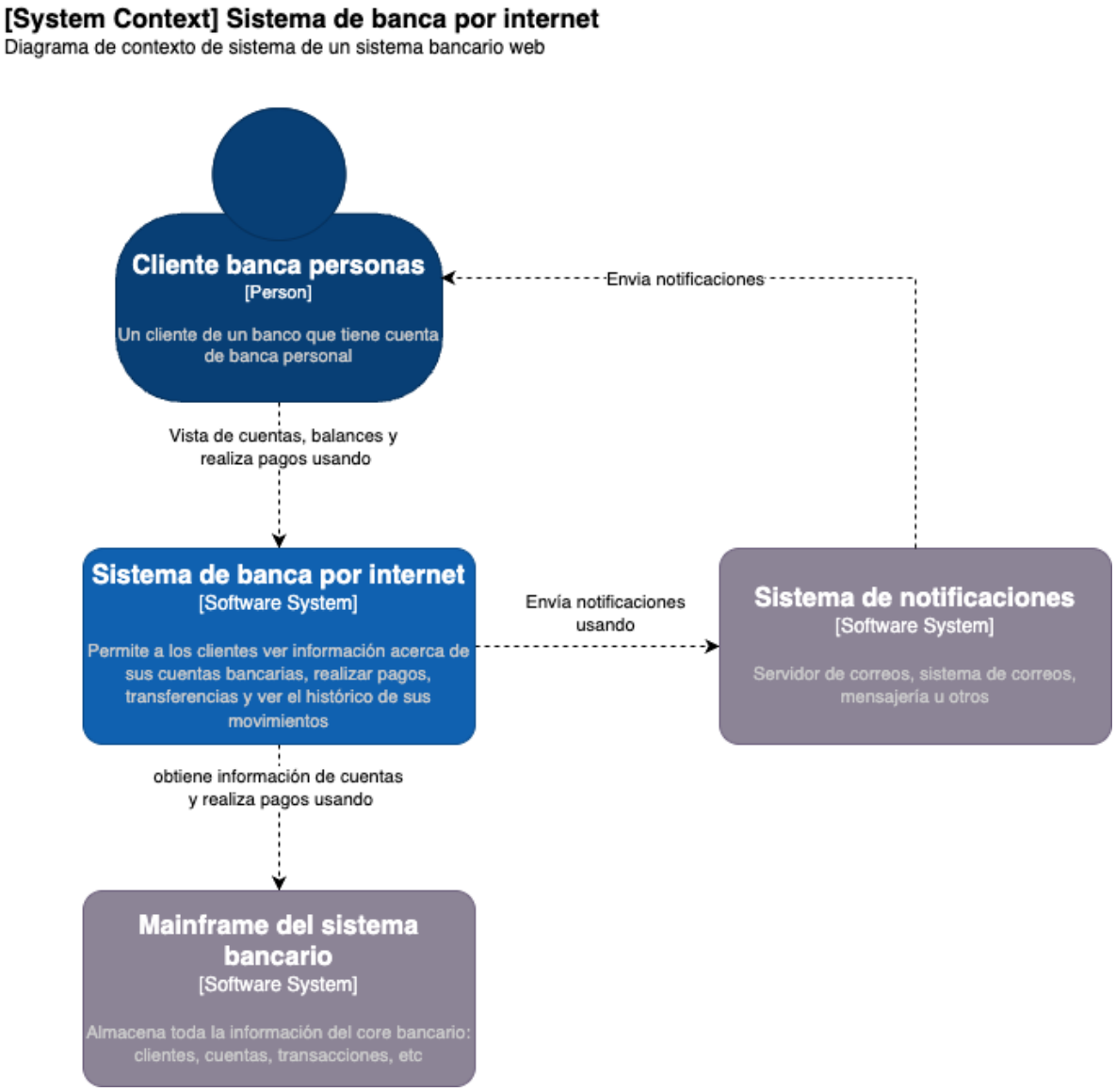
Seguridad de grado empresarial
Managed Identity entre Functions y Event Hubs; Private Endpoints, VNET Integration, RBAC, cifrado en reposo y en tránsito.

Recomendaciones adicionales:

Arquitectura KEDA con soporte a java dado que determina cómo la solución debe escalar cualquier contenedor dentro de Kubernetes. La decisión se basa en la cantidad de eventos que necesita procesar. KEDA, que cuenta con diferentes tipos de escaladores, admite múltiples tipos de cargas de trabajo, es compatible con Azure Functions y no depende del proveedor. (Referencias oficiales de azure)

Diagramas de arquitectura de software C4

Modelo de contexto



A continuación una breve explicación:

Sistema de banca por internet (software). Orquesta consultas, pagos, transferencias del cliente y dispara notificaciones según corresponda, a su vez permite ver el historico de movimientos

Elementos y relaciones

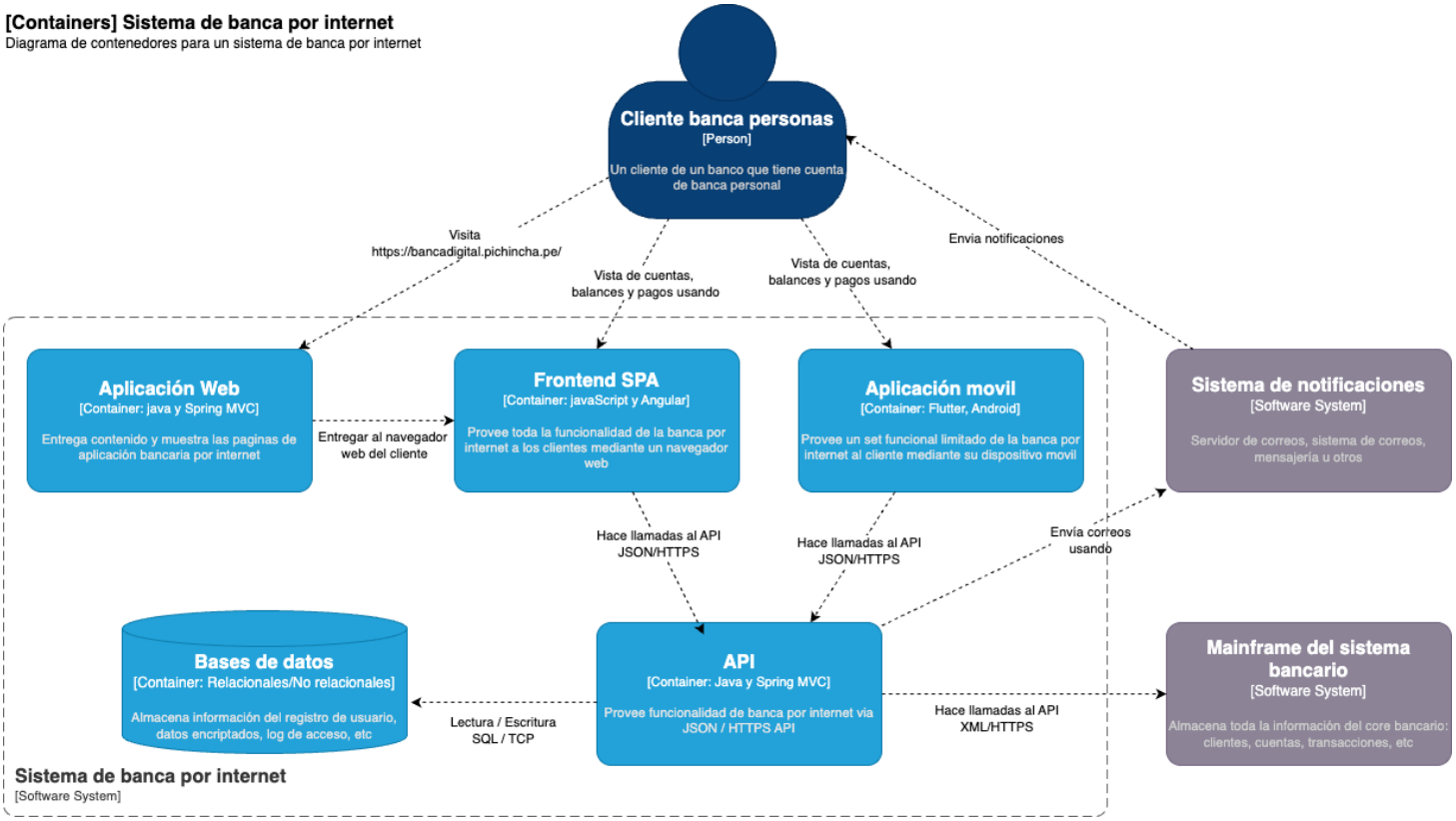
Cliente banca personas (Person)
Interactúa con el Sistema de banca por internet para visualizar cuentas, realizar pagos y transferencias ya sea a cuentas propias o de terceros.

Mainframe del sistema bancario (Software System)
El Sistema de banca por internet obtiene información de cuentas y realiza pagos “usando” este Mainframe**.**

Sistema de notificaciones (Software System)
El Sistema de banca por internet envía notificaciones usando este sistema.
El Sistema de notificaciones envía notificaciones al Cliente banca personas**.**

El Cliente solo conversa con el Sistema de banca por internet; este consulta y opera contra el Mainframe y, cuando corresponde, pide al Sistema de notificaciones que envíe avisos al Cliente. No hay otras dependencias ni actores en este nivel.

Modelos de aplicación / contenedor



Contenedores y responsabilidades

Aplicación Web *(Java y Spring MVC)*: entrega contenido y muestra páginas al navegador del cliente.

Frontend SPA *(JavaScript y Angular)*: provee la funcionalidad de banca por internet en el navegador; **hace llamadas al API vía JSON/HTTPS**.

Aplicación móvil *(Flutter, Android)*: ofrece un set funcional limitado en el dispositivo móvil; **hace llamadas al API vía JSON/HTTPS**.

API *(Java y Spring MVC)*: expone la funcionalidad de banca por internet como **API JSON/HTTPS**; **lee/escribe** en la base de datos; **llama al Mainframe vía XML/HTTPS**; **envía correos usando** el sistema de notificaciones.

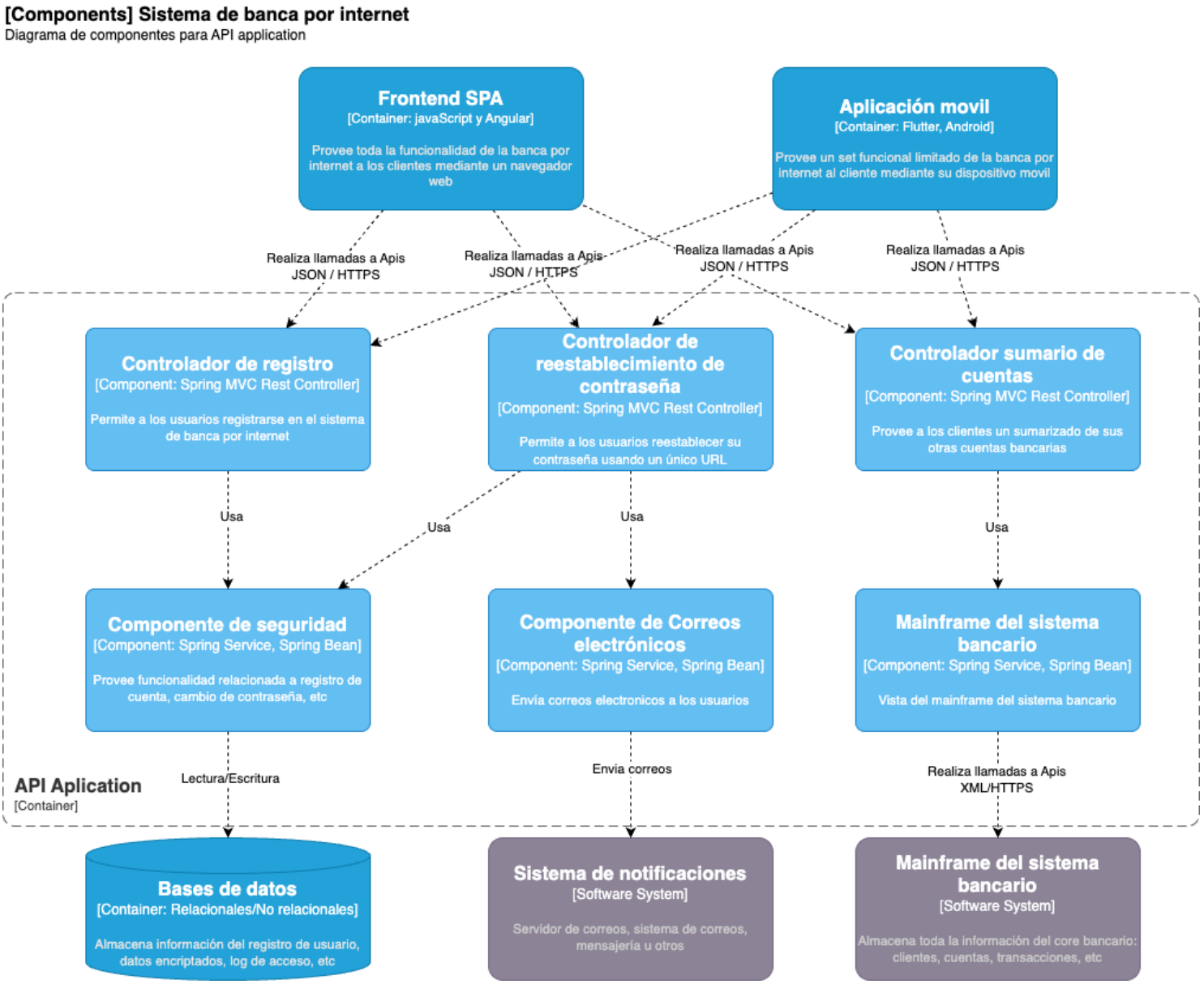
Bases de datos *(Relacionales/No relacionales)*: almacena registro de usuarios, datos encriptados, logs de acceso, etc.; acceso por **SQL/TCP**.

Sistemas externos

Mainframe del sistema bancario: almacena la información del core (clientes, cuentas, transacciones). La **API** le **hace llamadas XML/HTTPS**.

Sistema de notificaciones: servidor/sistema de correos o mensajería. La **API envía correos usando** este sistema, que **envía notificaciones** al **cliente**.

Modelo de componentes



Actor y canales

Cliente banca personas usa:

- **Frontend SPA** *(JavaScript/Angular)* → hace llamadas **JSON/HTTPS** a la API.
- **Aplicación móvil** *(Flutter/Android)* → hace llamadas **JSON/HTTPS** a la API.

API Application (contenedor) y componentes

Controladores (expuestos como REST)

- **Controlador de registro** *(Spring MVC Rest Controller)*
  Permite registrarse. **Usa** el **Componente de seguridad**.
- **Controlador de restablecimiento de contraseña** *(Spring MVC Rest Controller)*
  Permite restablecer contraseña mediante un único URL. **Usa** el **Componente de Correos electrónicos**.
- **Controlador sumario de cuentas** *(Spring MVC Rest Controller)*
  Entrega al cliente un **sumario de sus cuentas**. **Usa** el **Mainframe del sistema bancario (componente)**.

Servicios/Beans internos

- **Componente de seguridad** *(Spring Service/Bean)*
  Funcionalidad de **registro de cuenta** y **cambio de contraseña**.
- **Componente de Correos electrónicos** *(Spring Service/Bean)*
  Envía correos electrónicos a los usuarios.
- **Mainframe del sistema bancario (componente)** *(Spring Service/Bean)*
  Provee **vista**/acceso al **mainframe**.

Persistencia

- **Bases de datos** *(Relacionales/No relacionales)* - **Lectura/Escritura SQL/TCP**
  Guarda registro de usuario, **datos encriptados**, logs de acceso, etc.

Sistemas externos

- **Sistema de notificaciones** - la API **envía correos** usando este sistema.
- **Mainframe del sistema bancario** - el componente de mainframe **realiza llamadas a APIs vía XML/HTTPS**.

Decisiones Arquitectonicas

1) Framework móvil (multiplataforma)
Evaluadas: Flutter vs React Native (RN).
Selección: Flutter por consistencia visual, performance en listas/animaciones, UX estable.
Alternativa: RN por talento JS/TS; desventaja: dependencia de bridges en vistas muy dinámicas.
Con esto se cumple con la necesidad de una app móvil con framework multiplataforma

2) Flujo OAuth 2.0 / OIDC
Opciones: Implicit, ROPC, Authorization Code, Authorization Code + PKCE.
Selección: Authorization Code + PKCE para SPA y móvil (clientes públicos), mitigando robo de tokens.
Proveedor: Microsoft Entra External ID (Azure AD B2C) por integración nativa, MFA, políticas.
Con esto se cumple con recomendar "el mejor flujo" con un producto configurable

3) Onboarding con reconocimiento facial
Selección: FaceTec (liveness robusto) + verificación documental (Onfido/Keesing) si KYC reforzado.
Razón: Minimiza PII en tránsito, detección de fraude y UX rápida.

4) Patrones de back-end
• BFF (Web/Mobile), CQRS + Read Models para consultas, Saga para orquestación de transferencias/pagos, Outbox + Event Bus para confiabilidad, Circuit Breaker/Retry/Timeouts para resiliencia.

5) Capa de integración
Selección: Azure API Management (APIM) por políticas, validación JWT, throttling, mTLS y analítica nativa.

6) Persistencia, performance y caché
Cache-Aside con Azure Cache for Redis. Read model de movimientos con Elasticsearch. OLTP en Azure PostgreSQL.

7) Auditoría inmutable
Event Sourcing mínimo para auditoría: Azure Event Hubs → Bus → Storage/Data Lake con política WORM + SQL Ledger opcional.

8) Notificaciones
Push (FCM/APNs) + SMS (Twilio) + Email (SendGrid) opcional.

9) Nube y latencia
Azure (Front Door/CDN, AKS, APIM, Service Bus, Key Vault, Monitor/App Insights), despliegues multi-zona/region por ejm regiones de Sudamerica cercanas.

Normativa, seguridad excelencia operativa

Normativa y seguridad:
Ley de Protección de Datos Personales (Perú): consentimiento, minimización, cifrado en tránsito (TLS 1.2+) y en reposo, retención y derechos ARCO.

Seguridad Financiera
(SBS/OWASP/PCI): OWASP ASVS, PCI DSS si aplica (tokenización), WAF (Front Door), DDoS, Key Vault, mTLS intra-servicios.

Registro/Auditoría:
inmutabilidad (WORM), trilateración de logs (App Insights + Log Analytics + Data Lake).

Alta disponibilidad, tolerancia a fallos, DR, excelencia operativa:
AKS multi-zona/región; stateless activo/activo, stateful activo/pasivo con réplicas.
RPO <= 5 min / RTO <= 30–60 min.
Auto-healing (liveness/readiness, HPA, KEDA).
Observabilidad OpenTelemetry → Azure Monitor/App Insights; tracing distribuido, tableros de latencia/errores.
Cost-aware: autoscaling, Functions para tareas esporádicas, ILM/retención en ES.

Persistencia para Clientes Frecuentes, Auditoría y Notificaciones

Mecanismo de persistencia para clientes frecuentes:
Cache-Aside con Redis (perfiles, catálogos, cuentas destino frecuentes), TTL e invalidación por evento; objetivo: hit-ratio > 80%.

Diseño de auditoría:
Eventos de dominio (CommandHandled, TransferInitiated, MovementQuery, LoginSucceeded, etc.) publicados por Outbox.
Canal: Event Hubs (particionado por customerId)
Data Lake con política WORM + metadatos indexados.
SQL Ledger opcional para búsquedas rápidas, retención acorde normativa.

Notificaciones:
Push (FCM/APNs), SMS (Twilio) y Email (SendGrid) con workers idempotentes, reintentos y plantillas i18n.