Documento de Arquitectura del Sistema Bancario

---

1. Introducción

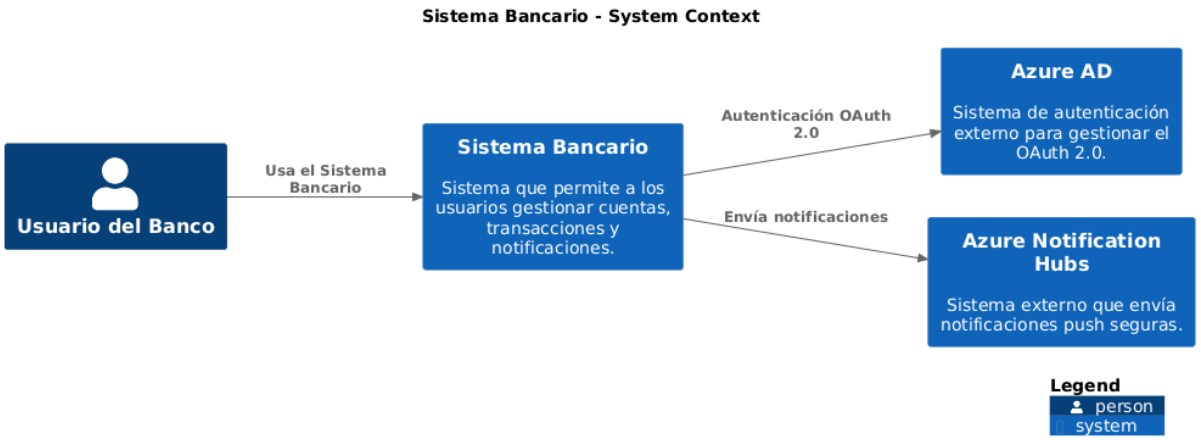
Este documento describe la arquitectura del sistema bancario diseñado para gestionar cuentas, realizar transacciones, y permitir un proceso de autenticación seguro, todo ello manteniendo una alta disponibilidad, tolerancia a fallos, y escalabilidad. La solución está diseñada para cumplir con regulaciones bancarias, seguridad de datos, y estándares de integración.

---

2. Requerimientos Funcionales

- Gestión de cuentas y transacciones de usuarios.
- Proceso de autenticación seguro a través de **OAuth 2.0**.
- Registro de nuevos usuarios utilizando **Onboarding** con reconocimiento facial.
- Envío de notificaciones a través de varios canales, incluyendo notificaciones push.
- Auditoría de las acciones de los usuarios para cumplir con normativas.

Diagrama de Contexto



El **Diagrama de Contexto** proporciona una vista de alto nivel del **Sistema Bancario** y cómo interactúa con los actores y sistemas externos. Es la representación más abstracta del sistema, enfocada en las conexiones externas sin detallar su estructura interna.

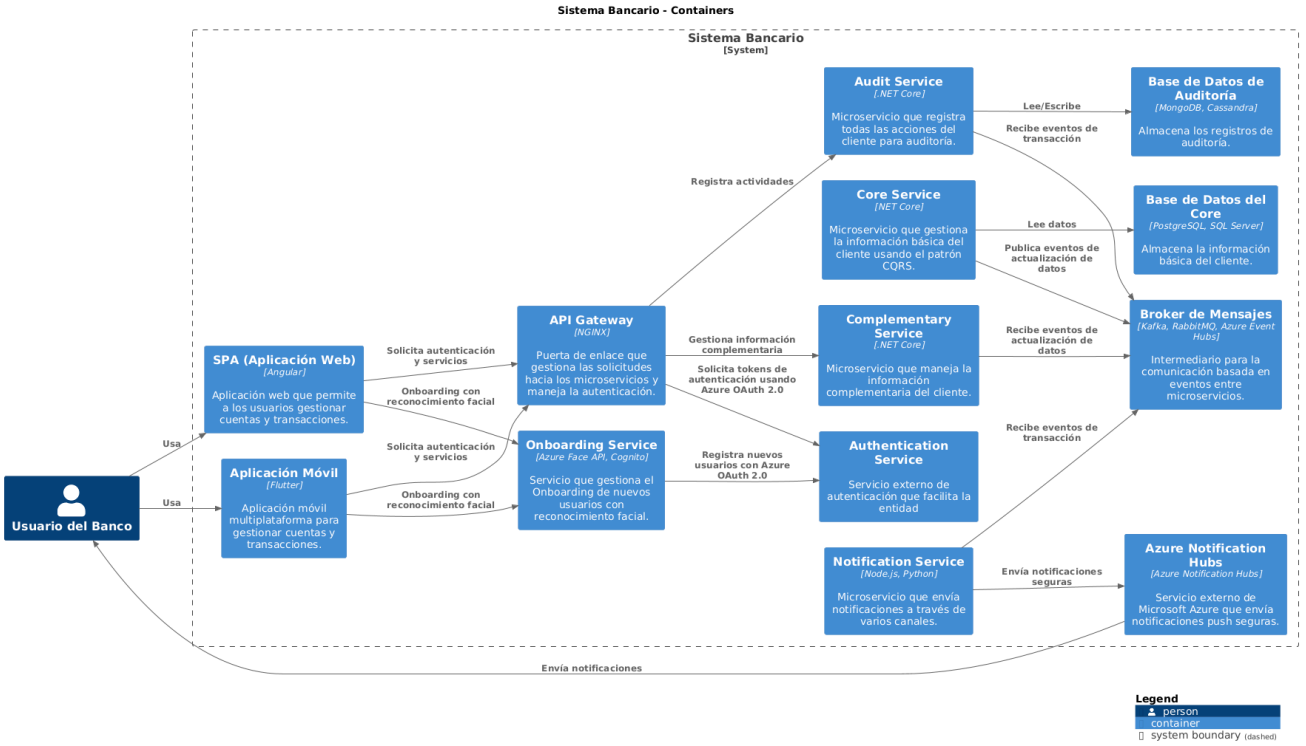
Elementos Clave

- **Sistema Bancario**: El núcleo del diagrama, que encapsula todas las funciones del sistema.
- **Actores Externos (Usuario del Banco)**: El usuario que interactúa con el sistema para realizar transacciones y gestionar cuentas.
- **Sistemas Externos**: Servicios como **Azure AD** (para la autenticación) y **Azure Notification Hubs** (para notificaciones) que interactúan con el sistema.
- **Relaciones**: Las flechas muestran cómo los actores y sistemas externos se comunican con el sistema. Ejemplo: el **Usuario del Banco** usa el sistema, que a su vez solicita autenticación a **Azure AD** y envía notificaciones a través de **Azure Notification Hubs**.

Propósito

El diagrama de contexto simplifica la comprensión del sistema, mostrando de forma clara las interacciones externas clave, ideal para que cualquier persona (técnica o no) entienda cómo se relaciona el sistema con su entorno.

Diagrama de Contenedores



---

3. Componentes de la Arquitectura

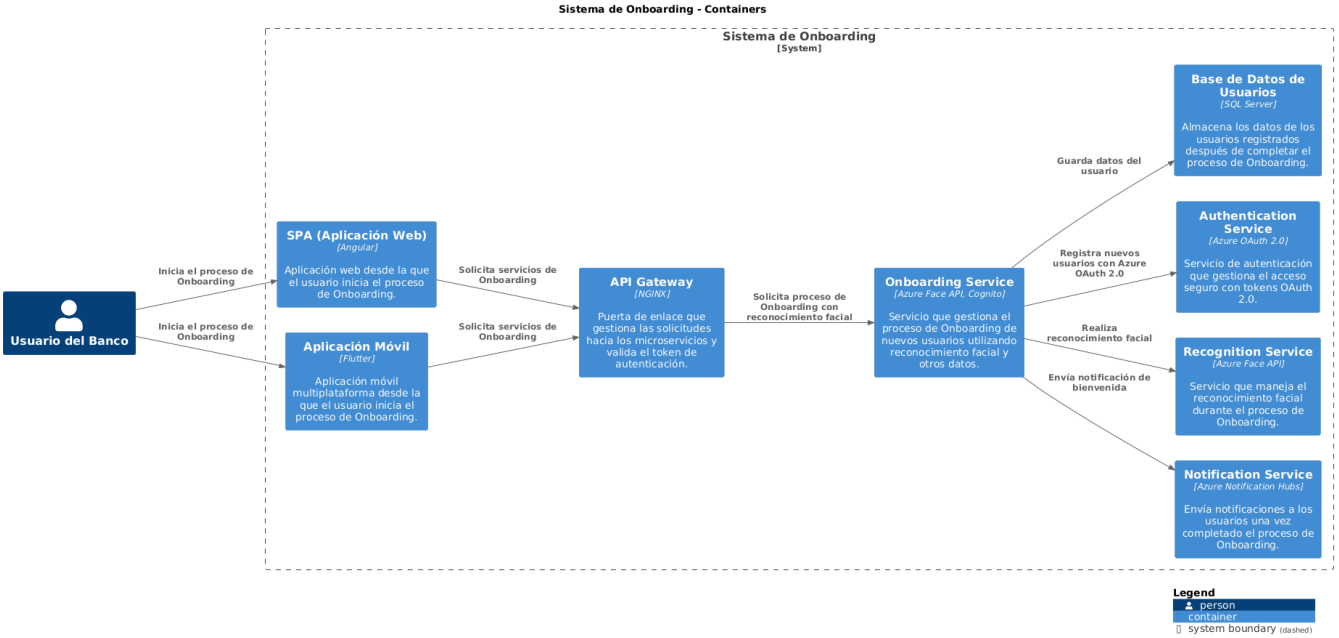
1. **Aplicación Web (SPA)**
    - **Tecnología**: Angular.
    - **Función**: Permite a los usuarios gestionar cuentas y transacciones desde un navegador.
    - **Justificación**: Angular es ideal para aplicaciones bancarias por su estructura modular, escalabilidad, y capacidad para desarrollar aplicaciones de una sola página con una rica experiencia de usuario.
2. **Aplicación Móvil**
    - **Tecnología**: Flutter.
    - **Función**: Proporciona una experiencia nativa en dispositivos móviles.
    - **Justificación**: Flutter permite un desarrollo multiplataforma eficiente, reduciendo costos y tiempo de desarrollo, mientras se mantienen altos estándares de calidad.
3. **API Gateway**
    - **Tecnología**: NGINX.
    - **Función**: Centraliza y gestiona todas las solicitudes hacia los microservicios y maneja la autenticación.
    - **Justificación**: El API Gateway asegura que las solicitudes sean autenticadas de manera centralizada, simplificando la arquitectura y mejorando la seguridad.
4. **Onboarding Service**
    - **Tecnología**: Azure Face API, Cognito.
    - **Función**: Gestiona el proceso de registro de nuevos usuarios mediante reconocimiento facial.
    - **Justificación**: La utilización de **Azure Face API** garantiza un reconocimiento facial preciso, lo que mejora la seguridad y automatiza el proceso de registro de nuevos usuarios.
5. **Authentication Service**
    - **Tecnología**: Azure OAuth 2.0.
    - **Función**: Proporciona autenticación segura a través de tokens OAuth 2.0.
    - **Justificación**: Azure OAuth 2.0 permite seguir estándares de seguridad ampliamente aceptados, lo que garantiza una autenticación robusta.
6. **Core Service (CQRS)**
    - **Tecnología**: .NET Core.
    - **Función**: Gestiona la información básica del cliente.
    - **Justificación**: El patrón **CQRS** (Command Query Responsibility Segregation) permite separar la lógica de escritura y lectura, lo que optimiza el rendimiento y escalabilidad.
7. **Complementary Service**
    - **Tecnología**: .NET Core.
    - **Función**: Gestiona la información complementaria del cliente.

- **Justificación**: La separación de responsabilidades asegura una arquitectura más modular y flexible, permitiendo cambios sin afectar otros servicios.

8. **Audit Service**
   - **Tecnología**: .NET Core.
   - **Función**: Registra todas las acciones del cliente para fines de auditoría.
   - **Justificación**: La auditoría es una función clave en sistemas financieros, necesaria para cumplir con regulaciones y normativas de seguridad.
9. **Notification Service**
   - **Tecnología**: Node.js, Python.
   - **Función**: Envía notificaciones a través de varios canales (correo, SMS, notificaciones push).
   - **Justificación**: Tener un microservicio dedicado a las notificaciones permite agregar nuevos canales sin afectar otros componentes del sistema.
10. **Bases de Datos**
    - **Core Database**: PostgreSQL, SQL Server.
    - **Audit Database**: MongoDB, Cassandra.
    - **Función**: Almacenan la información básica del cliente y los registros de auditoría.
    - **Justificación**: Las bases de datos están optimizadas para su propósito específico, lo que mejora el rendimiento y la resiliencia.
11. **Message Broker**
    - **Tecnología**: Kafka, RabbitMQ, Azure Event Hubs.
    - **Función**: Facilita la comunicación basada en eventos entre microservicios.
    - **Justificación**: La arquitectura orientada a eventos permite la escalabilidad y el desacoplamiento entre servicios, garantizando la resiliencia del sistema.
12. **Azure Notification Hubs**
    - **Función**: Envía notificaciones push de manera segura a los usuarios.
    - **Justificación**: Aprovechar un servicio externo para notificaciones push reduce la complejidad y mejora la escalabilidad del sistema.

Diagrama de Componentes

Arquitectura de Autenticación Con Onbording



Esta arquitectura de onboarding y autenticación para el sistema está diseñada para garantizar un proceso seguro, eficiente y escalable.

- Inicio del Proceso: Los usuarios pueden iniciar el onboarding desde una SPA (Angular) o una Aplicación Móvil (Flutter), que se comunican con un API Gateway (NGINX) para gestionar las solicitudes.
- Reconocimiento Facial: El Onboarding Service utiliza Azure Face API para verificar la identidad del usuario mediante reconocimiento facial, asegurando una autenticación biométrica precisa.
- Autenticación: Una vez verificada la identidad, el usuario es registrado en el sistema mediante OAuth 2.0, generando un token seguro para futuras autenticaciones.
- Notificaciones: Al finalizar el proceso, el Notification Service envía notificaciones en tiempo real a través de Azure Notification Hubs.
- Escalabilidad y Seguridad: La arquitectura es modular y desacoplada, permitiendo que los servicios como el Onboarding Service y el Authentication Service se escalen de manera independiente sin afectar el resto del sistema.

---

4. Justificación de la Arquitectura

La arquitectura fue diseñada bajo principios de **desacoplamiento**, **escalabilidad** y **mantenibilidad**. El uso de microservicios permite que cada componente funcione de manera independiente, facilitando la actualización y escalado de partes individuales sin afectar el sistema completo. A continuación, se describen los puntos clave:

1. **Desacoplamiento**: Los microservicios como el **Core Service**, **Complementary Service**, y **Audit Service** están completamente separados, lo que permite modificaciones o despliegues independientes. Esto reduce los riesgos al implementar nuevas funcionalidades o corregir errores.
2. **Escalabilidad**: Componentes como el **Message Broker** y los microservicios desacoplados permiten escalar solo los servicios que lo necesiten, sin aumentar la carga en todo el sistema. Los servicios en la nube, como **Azure Notification Hubs**, también son escalables automáticamente según la demanda.
3. **Mantenibilidad**: Al usar patrones de diseño como **CQRS** y el **API Gateway**, el sistema es más fácil de mantener y mejorar. Además, los servicios de autenticación y notificación están externalizados, lo que reduce la complejidad de implementación.

---

5. Cumplimiento de Normativas Bancarias y Seguridad

- **Autenticación**: El uso de **OAuth 2.0** asegura que solo usuarios autenticados puedan acceder al sistema, cumpliendo con los estándares de seguridad más rigurosos.
- **Auditoría**: El **Audit Service** registra todas las acciones críticas para cumplir con los requisitos regulatorios de trazabilidad.
- **Cumplimiento de GDPR**: Los datos personales son almacenados y gestionados en bases de datos seguras y están sujetos a procesos de control de acceso y protección de datos.

---

6. Implementación de Alta Disponibilidad y Tolerancia a Fallos

La arquitectura está diseñada para ser altamente disponible y tolerante a fallos. El uso de microservicios y componentes externos como **Azure** permite una recuperación rápida en caso de fallos, mientras que los mecanismos de balanceo de carga y el **Message Broker** garantizan que las operaciones asíncronas no se vean afectadas por fallos individuales.

---

7. Costos y Mantenimiento

El uso de servicios en la nube como **Azure Notification Hubs**, **Azure AD** y **Azure Event Hubs** permite un modelo de precios basado en el uso, lo que asegura costos eficientes a medida que el sistema crece. Además, al estar desacoplado, solo los componentes que requieren mejoras de rendimiento necesitan ser escalados, lo que reduce significativamente los costos de mantenimiento.

Para hacer una estimación realista de costos y tiempos del desarrollo de esta arquitectura para el sistema bancario, te proporcionaré una tabla con los componentes principales del sistema, el tiempo estimado para cada uno y el costo estimado basado en la cantidad de horas de trabajo.

Tabla de Cotización y Tiempos

Componente	Descripción	Tiempo Estimado (horas)	Costo Estimado (USD)
1. Arquitectura y Diseño	Diseño detallado de la arquitectura, diagramas, patrones de desacoplamiento.	40 horas	$2,000
2. Implementación del API Gateway	Desarrollo e integración del **API Gateway** con microservicios.	20 horas	$1,000
3. Desarrollo del SPA	Implementación de la **SPA (Angular)** y flujo de autenticación.	80 horas	$4,000
4. Desarrollo de la App Móvil	Implementación de la **Aplicación Móvil (Flutter)** y proceso de onboarding.	100 horas	$5,000
5. Onboarding Service	Implementación del proceso de **Onboarding** con reconocimiento facial.	60 horas	$3,000
6. Authentication Service	Configuración e integración de **Azure AD** para autenticación.	40 horas	$1,000
7. Core Service (CQRS)	Implementación del **Core Service** con el patrón **CQRS**.	80 horas	$2,000

9. Audit Service	Desarrollo del **Audit Service** para registro de acciones.	50 horas	$3,000
10. Notification Service	Implementación del servicio de notificaciones con **Azure Notification Hubs** y **Pusher**.	60 horas	$2,000
11. Bases de Datos	Diseño y configuración de las bases de datos para **Core** y **Auditoría**.	40 horas	$1,000
12. Integración del Message Broker	Integración de **Kafka/RabbitMQ** para el manejo de eventos.	40 horas	$2,000
13. Pruebas y Validación	Pruebas unitarias, de integración, y validación de seguridad.	80 horas	$2,000
14. Despliegue en la nube (Azure)	Despliegue de servicios y microservicios en **Azure**.	40 horas	$2,000
15. Documentación	Creación de la documentación técnica y manuales para el equipo.	30 horas	$2,000
16. Monitoreo y Tolerancia a Fallos	Implementación de monitoreo y manejo de fallos (Azure Monitor, Logging).	40 horas	$2,000

Resumen de tiempos y costos

Total de Horas	Total Estimado (USD)
840 horas	$32,000

Justificación del Tiempo y Costo

1. **Complejidad del Sistema**: Este es un sistema bancario con requisitos de alta seguridad, alta disponibilidad y cumplimiento regulatorio. La cantidad de

microservicios desacoplados, así como la integración con servicios externos como **Azure**, añade complejidad al desarrollo.

2. **Tiempos de Implementación**:
    - El desarrollo de la **SPA** y la **Aplicación Móvil** son los componentes que tomarán más tiempo debido a la integración con múltiples servicios (API Gateway, Onboarding, Autenticación).
    - La integración con **Azure** y la implementación del **Core Service** y el **Onboarding Service** también requieren tiempo debido a las pruebas de autenticación y reconocimiento facial.
3. **Seguridad y Auditoría**: Como es un sistema bancario, es fundamental implementar un servicio de auditoría sólido, pruebas exhaustivas, y asegurar que los mecanismos de seguridad estén bien configurados, lo que también añade tiempo.
4. **Despliegue en la Nube**: Utilizar **Azure** para el despliegue y la implementación de servicios en la nube requiere planificación y pruebas adicionales, para asegurar una alta disponibilidad y tolerancia a fallos.

Los Costos también podrían reducir utilizando tecnologías que nos ayude en la implementación

---

Plazos

Basado en una dedicación de 40 horas semanales, el proyecto completo tomaría aproximadamente **21 semanas** (casi 5 meses). Los plazos pueden ajustarse dependiendo del tamaño del equipo y la simultaneidad de tareas. Para reducir el tiempo, se puede incrementar el número de desarrolladores.

Tabla de Costos de Tecnologías Externas de Azure

Servicio de Azure	Descripción	Modelo de Precios	Costo Estimado (USD/mes)
Azure Face API	Servicio de reconocimiento facial para el proceso de Onboarding.	Pago por transacción y reconocimiento facial.	$1.50 por 1,000 transacciones
Azure Notification Hubs	Servicio para enviar notificaciones push a los usuarios.	Pago por cantidad de	$200 por 10M notificaciones

		notificaciones enviadas.	
Azure Event Hubs (o Kafka/RabbitMQ)	Servicio de mensajería para eventos basados en microservicios (Event-Driven Architecture).	Pago por uso y volumen de datos procesados.	$0.028/hora (Instancia básica)
Azure Monitor	Servicio para supervisar la disponibilidad y el rendimiento del sistema.	Pago por cantidad de datos monitoreados.	$2.76/GB de datos monitoreados
Azure SQL Database (Base de Datos del Core)	Base de datos relacional para almacenar la información básica del cliente.	Pago por base de datos según DTU (capacidad).	$200 a $500/mes
Azure Cosmos DB (Base de Datos de Auditoría)	Base de datos NoSQL para almacenar los registros de auditoría.	Pago por volumen de datos y unidades de solicitud (RU/s).	$24 por 100 RU/s, $0.25/GB de almacenamiento
Azure Storage	Almacenamiento de datos estáticos o grandes volúmenes de registros de transacciones.	Pago por GB de almacenamiento.	$0.0184/GB (Almacenamiento Hot)
Azure Key Vault	Almacenamiento seguro para claves de API, certificados y secretos.	Pago por operación y cantidad de claves almacenadas.	$0.03 por 10,000 operaciones

---

Estimación de Costos Mensuales Totales (USD)

Servicio	Costo Estimado Mensual (USD)
Azure Face API (50,000 transacciones mensuales)	$75
Azure Notification Hubs (10M notificaciones)	$200
Azure Event Hubs (Mensajería básica)	$40
Azure Monitor (50 GB de datos monitoreados)	$138
Azure SQL Database (Capacidad de 100 DTU)	$300
Azure Cosmos DB (Auditoría - 500 RU/s)	$120
Azure Storage (Almacenamiento de 500 GB)	$9
Azure Key Vault (Almacenamiento de claves)	$5

Total Estimado Mensual: $887 USD/mes

Justificación del Costo por Servicio

1. **Azure Face API**: El costo por transacción es adecuado para un sistema que hace reconocimiento facial, considerando que las solicitudes de onboarding pueden no ser masivas todos los días.
2. **Azure Notification Hubs**: Los costos dependen del número de notificaciones push que envíes al mes. Para un sistema bancario que envía notificaciones después de cada transacción, se considera un volumen alto.
3. **Azure Event Hubs**: Utilizado como **Message Broker**, el costo depende de la cantidad de datos que manejes. Una configuración básica es más que suficiente para manejar eventos de microservicios y mensajería.
4. **Azure Monitor**: Necesitarás monitorear el estado del sistema y su rendimiento, por lo que los costos aquí dependen del volumen de datos monitoreados. Un sistema bancario genera bastantes eventos que deben ser controlados.
5. **Azure SQL Database**: Almacenar la información del cliente en una base de datos escalable y confiable como **Azure SQL** es vital. El costo variará según la cantidad de datos y transacciones.

6. **Azure Cosmos DB**: Para auditoría, una base de datos NoSQL como **Cosmos DB** es ideal. El costo es razonable para almacenar grandes volúmenes de datos de auditoría con alta disponibilidad y replicación global.
7. **Azure Storage**: El costo de almacenamiento es bajo y permite tener un repositorio de datos adicionales (archivos, logs, etc.).
8. **Azure Key Vault**: Para almacenar claves de API, certificados y otros secretos, el costo es mínimo pero necesario para asegurar la gestión de claves.

Los costos también pueden reducir conociendo las estadísticas reales de la concurrencia y en base al porcentaje de uso de la aplicación se pueden reducir los costos de los servicios externos

---

Consideraciones Adicionales

- **Escalabilidad**: Todos estos servicios en **Azure** permiten escalabilidad automática, lo que significa que los costos podrían aumentar si el uso del sistema crece de forma exponencial.
- **Reducción de costos**: En caso de que el volumen de usuarios o transacciones sea bajo inicialmente, algunos servicios pueden ser ajustados para ahorrar costos. Por ejemplo, se pueden bajar los niveles de **DTU** en la base de datos de SQL o reducir las notificaciones push.

Tecnologías usadas en el proyecto

SPA

He escogido **Angular** para SPA debido a que su **marco de trabajo estructurado** nos permite organizar el proyecto de manera clara y eficiente, lo cual es crucial en un entorno bancario donde la consistencia y el mantenimiento son esenciales.

Angular viene con una estructura definida que facilita la separación de responsabilidades, asegurando que cada parte del sistema esté bien organizada. Para este proyecto la estructura de carpetas que vamos a utilizar es la siguiente :

```
src/
├── app/
│   ├── core/
│   │   ├── services/
```

```
│   │   │   ├── auth.service.ts
│   │   │   ├── user.service.ts
│   │   │   └── data.service.ts
│   │   ├── interceptors/
│   │   │   ├── auth.interceptor.ts
│   │   │   └── error.interceptor.ts
│   │   └── guards/
│   │       ├── auth.guard.ts
│   │       └── admin.guard.ts
│   ├── features/
│   │   ├── user/
│   │   │   ├── user.component.ts
│   │   │   ├── user.service.ts
│   │   │   └── user.module.ts
│   │   ├── admin/
│   │   │   ├── admin.component.ts
│   │   │   ├── admin-dashboard/
│   │   │   │   ├── admin-dashboard.component.ts
│   │   │   │   └── admin-dashboard.component.html
│   │   │   └── admin.module.ts
│   ├── shared/
│   │   ├── components/
│   │   │   ├── button/
│   │   │   │   ├── button.component.ts
│   │   │   │   └── button.component.html
│   │   │   ├── modal/
│   │   │   │   ├── modal.component.ts
│   │   │   │   └── modal.component.html
│   │   │   └── form/
```

```
│   │   │       ├── form.component.ts
│   │   │       └── form.component.html
│   │   ├── directives/
│   │   │   ├── highlight.directive.ts
│   │   │   └── autofocus.directive.ts
│   │   └── pipes/
│   │       ├── date-format.pipe.ts
│   │       └── currency-format.pipe.ts
│   ├── assets/
│   ├── environments/
│   └── app.module.ts
```

Esta estructura de carpetas se alinea con la **organización modular** que nos ofrece Angular:

- **Core**: Aquí se ubican los servicios compartidos como la autenticación (`auth.service.ts`) y los interceptores, que nos permiten gestionar los aspectos globales de la aplicación, como la autenticación de solicitudes HTTP.
- **Features**: Contiene los módulos específicos de la aplicación, como `user` y `admin`, permitiendo la escalabilidad y separación de funcionalidades clave.
- **Shared**: Almacena componentes reutilizables, directivas y pipes, lo que facilita la modularización del código y la reutilización de elementos a lo largo de la aplicación.

Esta manera de organizar las carpetas permite una **separación clara de responsabilidades**, haciendo que la aplicación sea **fácil de escalar y mantener**.

App Movil

Para la Aplicacion Movil Vamos Utilizar la tecnologia de Flutter:

He escogido **Flutter** para la aplicación web por su capacidad para ofrecer tanto **escalabilidad** como **mantenibilidad**, lo cual es esencial en una institución bancaria. Flutter permite desarrollar una aplicación multiplataforma con un solo código base, asegurando una experiencia consistente y de fácil de expansión a medida que crecen las necesidades del sistema, además, el uso del **patrón BLoC** separa la lógica de negocio, de la interfaz de usuario, lo que facilita el mantenimiento, reduciendo el riesgo de errores y asegurando que la aplicación se pueda actualizar de manera eficiente y segura a lo largo del tiempo.

Para este proyecto la estructura de carpetas que vamos a utilizar es la siguiente :

```
lib/
├── core/
│   ├── services/
│   │   ├── api_service.dart
│   │   ├── auth_service.dart
│   │   └── user_service.dart
│   └── models/
│       ├── user_model.dart
│       └── transaction_model.dart
├── blocs/
│   ├── auth/
│   │   ├── auth_bloc.dart
│   │   ├── auth_event.dart
│   │   └── auth_state.dart
│   ├── user/
│   │   ├── user_bloc.dart
│   │   ├── user_event.dart
│   │   └── user_state.dart
│   └── transaction/
│       ├── transaction_bloc.dart
```

```
│       ├── transaction_event.dart
│       └── transaction_state.dart
├── features/
│   ├── login/
│   │   ├── login_screen.dart
│   │   ├── login_form.dart
│   │   └── login_bloc.dart
│   ├── dashboard/
│   │   ├── dashboard_screen.dart
│   │   └── widgets/
│   │       ├── balance_widget.dart
│   │       ├── transaction_list_widget.dart
│   │       └── account_summary_widget.dart
│   └── settings/
│       ├── settings_screen.dart
│       └── settings_bloc.dart
├── shared/
│   ├── widgets/
│   │   ├── custom_button.dart
│   │   ├── custom_dialog.dart
│   │   └── loading_spinner.dart
│   ├── utils/
│   │   ├── date_format.dart
│   │   ├── validators.dart
│   │   └── constants.dart
├── resources/
│   ├── strings/
│   ├── images/
│   └── styles/
```

├── main.dart

❖ core/

Contiene los **servicios y modelos** esenciales para la aplicación, los cuales son reutilizables en todo el proyecto.

- **services/**: Aquí se almacenan los servicios que manejan la lógica de negocio y la comunicación con el backend, donde cada servicio se encarga de una función específica:
  - `api_service.dart`: Lógica para realizar llamadas HTTP a la API del sistema.
  - `auth_service.dart`: Gestiona la autenticación de usuarios.
  - `user_service.dart`: Encargado de manejar los datos de los usuarios.
- **models/**: Esta carpeta contiene los **modelos de datos** que representan las entidades del negocio.
  - `user_model.dart`: Modelo que representa al usuario.
  - `transaction_model.dart`: Modelo que gestiona las transacciones bancarias del usuario.

❖ blocs/

Esta carpeta gestiona la **lógica de negocio** de la aplicación utilizando el patrón **BLoC**. Se organiza en subcarpetas por funcionalidad.

- **auth/**: Maneja todo lo relacionado con la autenticación (login/logout).
  - `auth_bloc.dart`: Controla el flujo de estados para la autenticación.
  - `auth_event.dart`: Define los eventos que pueden ocurrir en la autenticación, como el inicio o cierre de sesión.
  - `auth_state.dart`: Define los diferentes estados en los que puede estar la autenticación (por ejemplo, autenticado, no autenticado, cargando).
- **user/**: Controla la lógica relacionada con la gestión del usuario (perfil, datos, etc.).
  - `user_bloc.dart`: Lógica relacionada con la carga, actualización y visualización de los datos del usuario.
  - `user_event.dart`: Eventos que pueden modificar los estados del usuario.
  - `user_state.dart`: Los diferentes estados en los que pueden estar los datos del usuario (por ejemplo, cargando, completado, con error).

- **transaction/**: Gestiona la lógica de negocio relacionada con las transacciones.
    - `transaction_bloc.dart`: Controla las transacciones bancarias del usuario.
    - `transaction_event.dart`: Define eventos relacionados con la creación, edición o eliminación de transacciones.
    - `transaction_state.dart`: Define los estados en los que pueden estar las transacciones (por ejemplo, éxito, error, pendiente).

❖ features/

Esta carpeta organiza las **funcionalidades** o módulos de la aplicación, cada funcionalidad tiene su propia carpeta, que puede incluir pantallas, formularios y BLoC específicos.

- **login/**: Módulo que gestiona el inicio de sesión.
    - `login_screen.dart`: Pantalla principal de login.
    - `login_form.dart`: Formulario que captura las credenciales de inicio de sesión.
    - `login_bloc.dart`: Lógica BLoC que maneja los eventos y estados del inicio de sesión.
- **dashboard/**: Módulo del tablero de control donde el usuario ve sus datos bancarios.
    - `dashboard_screen.dart`: Pantalla principal del dashboard.
    - **widgets/**: Aquí se almacenan los componentes visuales del tablero.
        - `balance_widget.dart`: Muestra el balance de la cuenta del usuario.
        - `transaction_list_widget.dart`: Lista de transacciones del usuario.
        - `account_summary_widget.dart`: Resumen de la cuenta del usuario.
- **settings/**: Módulo de ajustes donde el usuario puede cambiar configuraciones de la cuenta.
    - `settings_screen.dart`: Pantalla de ajustes.
    - `settings_bloc.dart`: Lógica BLoC que maneja las acciones y eventos en la pantalla de ajustes.

❖ shared/

Aquí se encuentran los **componentes reutilizables** y las **utilidades** que pueden ser usadas en cualquier parte de la aplicación.

- **widgets/**: Componentes visuales reutilizables en distintas partes de la aplicación.
    - `custom_button.dart`: Botón personalizado que puede ser usado en varias pantallas.
    - `custom_dialog.dart`: Diálogo reutilizable para mostrar alertas o confirmaciones.
    - `loading_spinner.dart`: Indicador de carga para mostrar mientras se cargan los datos.
- **utils/**: Funciones utilitarias o constantes que se pueden compartir a lo largo del código.
    - `date_format.dart`: Funciones para formatear fechas.
    - `validators.dart`: Validadores reutilizables para formularios (por ejemplo, para verificar correos electrónicos o contraseñas).
    - `constants.dart`: Variables constantes, como rutas de API o valores predeterminados.

❖ resources/

Aquí se almacenan los **recursos gráficos y de estilo** de la aplicación.

- **strings/**: Texto constante para internacionalización o mensajes repetidos.
- **images/**: Recursos gráficos como íconos o imágenes.
- **styles/**: Archivo de estilos o temas globales, como colores o tipografías.