Prueba Técnica

Cristian Alejandro Muñoz Reyes.

Empresa: BTG Pactual

Parte 1 – Fondos (80%)

Necesidad de negocio:

BTG Pactual desea crear una plataforma que permita a los clientes gestionar sus fondos de inversión sin necesidad de contactar a un asesor. Las funcionalidades requeridas son:

Funcionalidades del sistema:

1. Suscribirse a un nuevo fondo (apertura).
2. Cancelar la suscripción a un fondo actual.
3. Ver historial de transacciones (aperturas y cancelaciones).
4. Enviar notificación por email o SMS según preferencia del usuario al suscribirse a un fondo.

Reglas de negocio:

Monto inicial del cliente: COP $500.000.

Cada transacción debe tener un identificador único.

Cada fondo tiene un monto mínimo de vinculación.

Al cancelar una suscripción, el valor de vinculación se retorna al cliente.

Si no hay saldo suficiente, mostrar:

“No tiene saldo disponible para vincularse al fondo <Nombre del fondo>”

Información de los fondos:

ID Nombre Monto mínimo Categoría

1 FPV_BTG_PACTUAL_RECAUDADORA COP $75.000 FPV

2 FPV_BTG_PACTUAL_ECOPETROL COP $125.000 FPV

3 DEUDAPRIVADA COP $50.000 FIC

4 FDO-ACCIONES COP $250.000 FIC

5 FPV_BTG_PACTUAL_DINAMICA COP $100.000 FPV

Actividades solicitadas:

1. Tecnologías sugeridas: Python con FastAPI, .NET Core, Java Springboot o Express - Node.js

Respuesta del punto:

Para la presente prueba usare como tecnología base Java Springboot, ya que es la tecnología en la que tengo más experiencia.

2. Diseñar un modelo de datos NoSQL que soporte las operaciones.

Respuesta del punto:

Para el presente punto se debe definir las clases POJOs que represente las tablas cliente, fondo y Transacción, dicha clase también contendrá los set y get de cada campo, a continuación la descripciones de las tablas:

- Cliente

Campo	Tipo de Dato	Rol en DynamoDB	Descripción
clientId	String (UUID)	Partition Key (PK)	Identificador único para cada cliente.
balance	Number	Attribute	Saldo disponible del cliente para invertir.
notificationPreference	String	Attribute	Preferencia de notificación del usuario (EMAIL o SMS).
email	String	Attribute	Correo electrónico del cliente.
phoneNumber	String	Attribute	Número de teléfono del cliente.

- fondo

Campo	Tipo de Dato	Rol en DynamoDB	Descripción
fundId	String	Partition Key (PK)	Identificador único para cada fondo de inversión.
name	String	Attribute	Nombre completo del fondo.
minAmount	Number	Attribute	Monto mínimo requerido para la suscripción.
category	String	Attribute	Categoría del fondo (ej. FPV, FIC).

- Transacción

Campo	Tipo de Dato	Rol en DynamoDB	Descripción
clientId	String (UUID)	Partition Key (PK)	ID del cliente que realiza la transacción.
transactionId	String	Sort Key (SK)	Identificador único de la transacción.
fundId	String	Attribute	ID del fondo involucrado en la transacción.
fundName	String	Attribute	Nombre del fondo (desnormalizado para eficiencia).
type	String	Attribute	Tipo de operación (SUBSCRIPTION o CANCELLATION).
amount	Number	Attribute	Monto de la transacción.
timestamp	String	Attribute	Fecha y hora exactas de la transacción.

3. Construir una API REST que implemente las funcionalidades descritas.

Respuesta del punto:

- Definiciones de endpoints

Suscribirse a fondo (POST) /clients/{clientId}/subscriptions

Cancelar suscripcion (DELETE) /clients/{clientId}/subscriptions/{fundId}

Historial (GET) /clients/{clientId}/transactions

- Implementar controlador

Este elemento es el encargado de recibir todas las peticiones HTTP y enrutar la petición al servicio apropiado para ejecutar allí la lógica de negocio y devolver una respuesta.

- Uso de DTOs dentro del controlador

Es una buena practica y parte de una arquitectura limpia el uso de los DTOs para este caso:

SubscriptionRequestDTO: es el DTO que viaja desde el cliente hacia el API, se usa cuando el cliente se quiere suscribir a un fondo. Aquí se trabaja con la información mínima y necesaria para realizar la suscripción.

TransactionDTO: es la respuesta exitosa que devuelve el API al cliente (suscripción, cancelación o historial). Este presenta un resumen de la transacción.

4. Incluir:

- Manejo de excepciones.

En este punto se deben manejar la excepciones con el fin de evitar caídas de la aplicación y dar información clara al usuario en caso de que algo salga mal. Para tal fin se debe crear y manejar excepciones personalizadas, que espongan los errores de lógica de negocio. Además de controlar las excepciones en un manejador global, el cual controla excepciones lanzadas desde cualquier controlador.

- Código limpio (Clean Code).

Para aplicar esta arquitectura se debe:

1. Usar nobres significativos.
2. Usar el principio de usabilidad única, cada clase y método debe tener una sola razón para cambiar.
3. Métodos cortos y enfocados, cada método debe hacer una sola acción.

- Pruebas unitarias.

Con esto se hace un testeo sobre cada unidad lógica de negocio, estro con el fin de comprobar su correcta ejecución de forma aislada. El foco debe ser la capa de servicio y esto se puede probar haciendo uso de herramientas como Junit y/o Mockito.

- Buenas prácticas de seguridad y mantenibilidad.

Para garantizar estos puntos se debe:

Seguridad

- Validación de Entradas: se debe usar anotaciones de validación (@Valid, @NotBlank, @Min) en los DTOs para rechazar datos malformados antes de que lleguen a la lógica de negocio.
- Manejo de Secretos: se deben usar variables de entorno o un servicio de gestión de secretos como AWS Secrets Manager.
- Principio de Mínimo Privilegio: El rol de IAM que use la función Lambda en AWS solo debe tener los permisos estrictamente necesarios para operar (ej. leer/escribir en las tablas de DynamoDB específicas del proyecto y nada más).
- Seguridad en Dependencias: se debe mantener las dependencias actualizadas para evitar vulnerabilidades conocidas.

Mantenibilidad

- Logging: Usar un framework de logging como SLF4J para registrar eventos importantes, advertencias y errores. Esto es vital para diagnosticar problemas en producción.
- Configuración Externalizada: Todas las configuraciones que puedan cambiar entre entornos (desarrollo, producción) deben estar en application.properties o application.yml, no en el código.
- Documentación de Código: el uso de Javadoc en los métodos públicos de tus servicios y controladores para explicar qué hacen, qué parámetros reciben y qué devuelven.
- Consistencia: Se debe mantener un estilo de código consistente en todo el proyecto (nomenclatura, formato, etc.).

5. Definir los procesos de autenticación, autorización, perfilamiento por roles y

encriptación.

Respuesta del punto:

Entendido. A continuación, un resumen en tercera persona de los cuatro procesos de seguridad definidos en el punto 5.

- Autenticación
  El sistema debe hacer uso de un token JWT, el cual es como una credencial usado para identificar las peticiones al API.
- Autorización
  El sistema debe verificar la identidad contenida en el JWT con el finde impedir el acceso a datos de otro usuario
- Perfilamiento por Roles
  El API debe implementar un sistema de perfiles basado en roles a través de Grupos de Cognito (ej. "Clientes" y "Administradores"). Los roles del usuario se incluyen dentro del JWT, lo que permite a la aplicación proteger endpoints específicos y restringir el acceso a funcionalidades sensibles únicamente a los usuarios con los permisos adecuados.
- Encriptación
  La protección de los datos se garantiza mediante una estrategia de encriptación completa. La encriptación en tránsito se logra con el uso obligatorio de HTTPS en API Gateway.

6. Despliegue: El backend debe poder desplegarse mediante AWS CloudFormation, con

documentación incluida.

Respuesta del punto:

Para el despliegue de la aplicación:

- **Amazon API Gateway** recibe las peticiones HTTP de os clientes.
- API Gateway invoca una función de **AWS Lambda** que contiene tu aplicación Spring Boot.
- La aplicación en Lambda ejecuta la lógica de negocio, interactuando con las tablas de **Amazon DynamoDB**.

Parte 2 – SQL (20%)

Base de datos: BTG

Escriba las consultas SQL correspondientes, para ello, tenga en cuenta la base de datos llamada “BTG” la cual tiene las siguientes tablas (tenga en cuenta que se puede presentar el caso de que no todas las sucursales ofrecen los mismos productos).

Tablas disponibles:

Cliente			
id	nombre	apellidos	ciudad
number	varchar	varchar	varchar
PK	NN	NN	NN

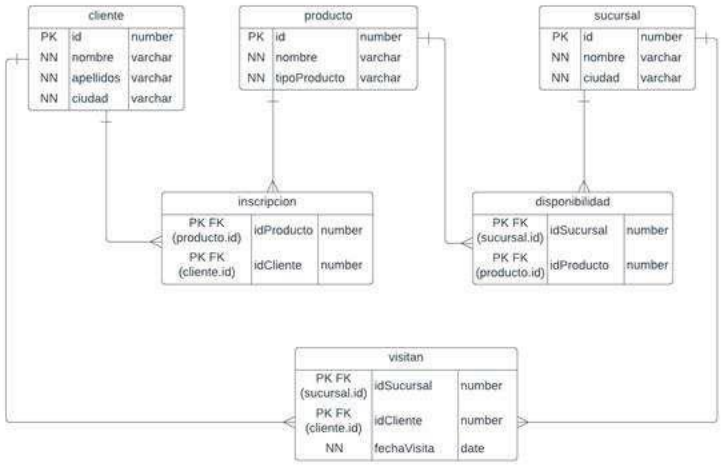
Sucursal		
id	nombre	ciudad
number	varchar	varchar
PK	NN	NN

Producto		
id	nombre	tipoProducto
number	varchar	varchar
PK	NN	NN

Inscripción	
idProducto	idCliente
number	number
PK FK(Producto.id)	PK FK(Cliente.id)

Disponibilidad	
idSucursal	idProducto
number	number
PK FK(Sucursal.id)	PK FK(Producto.id)

Visitan		
idSucursal	idCliente	fechaVisita
number	number	Date
PK FK(Sucursal.id)	PK FK(Cliente.id)	NN



Consulta solicitada:

Obtener los nombres de los clientes que tienen inscrito algún producto disponible solo en las sucursales que visitan.

Respuesta del punto:

```
SELECT DISTINCT
   c.nombre,
   c.apellidos
FROM
```

```
    Cliente c
JOIN
    Inscripcion i ON c.id = i.idCliente
WHERE
    NOT EXISTS (
        SELECT 1
        FROM
            Disponibilidad d
        WHERE
            d.idProducto = i.idProducto
            AND NOT EXISTS (
                SELECT 1
                FROM
                    Visitan v
                WHERE
                    v.idCliente = c.id
                    AND v.idSucursal = d.idSucursal
            )
    );
```