CLIENTES BANCO

Raúl Fernández, Javier Hernández, Yahya el Hadri, Samuel Cortés, Javier Ruíz y Álvaro Herrero.

INDICE

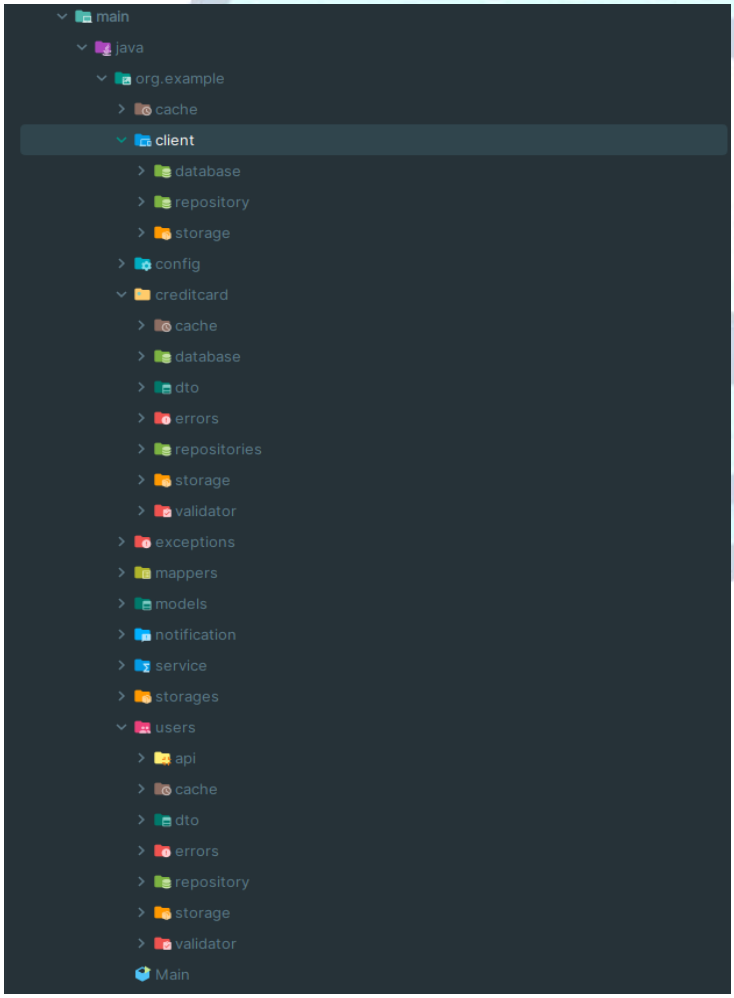
1- Arquitectura del proyecto.

Nuestra aplicación realiza la gestión de clientes de un banco. Para ello hemos usado una arquitectura centrada al dominio.

El modelo de dominio es una representación simplificada y estructurada de la lógica de negocio. En lugar de centrarse en cómo funcionan los detalles técnicos (como bases de datos o sistemas externos), el modelo de dominio refleja directamente los conceptos y reglas del negocio, por ejemplo, clases como Usuario, Tarjeta o Cliente.

Un cliente está constituido por un usuario y este usuario puede tener o no una o varias tarjetas. En cada dominio tenemos sus respectivos repositorios, validadores, caches y gestores de almacenamiento.

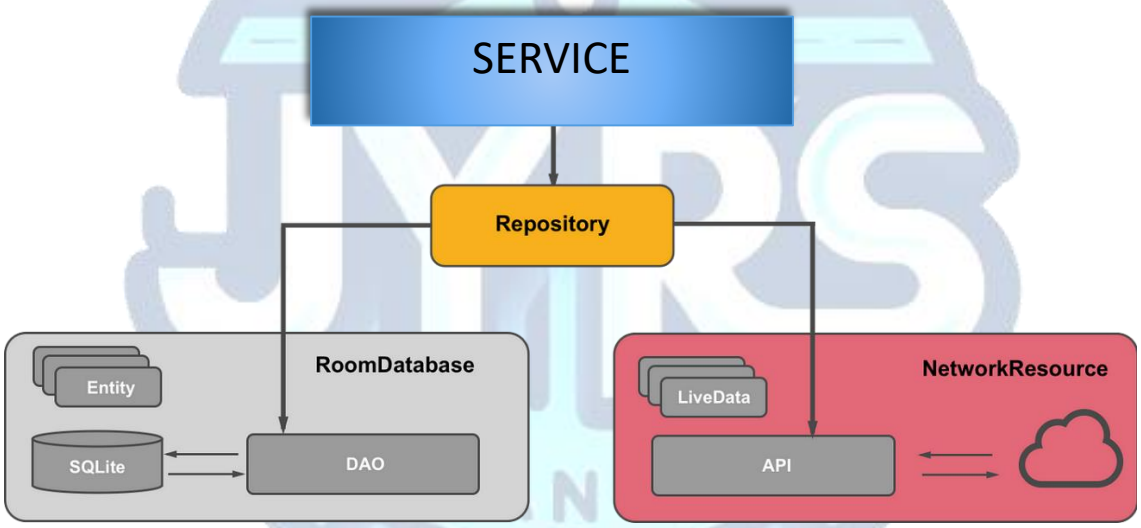
Hemos elegido la arquitectura orientada al domino ya que a la hora de desarrollar el software de la aplicación el lenguaje común ayuda a evitar malentendidos y garantiza que todos tengan una comprensión clara de lo que el software debe hacer. Además, esto permite modificar o escalar partes del sistema sin afectar el todo.



1.1- Patrón de diseño.

Para el desarrollo del proyecto hemos utilizado el patrón repository, de forma que junto al servicio tendremos un sistema que funcionará como una cache multinivel.

Para ello tenemos un servicio que es el principal responsable de manejar toda la funcionalidad de la aplicación. Dicho servicio tendrá una cache LRU que actuará de forma que sea la respuesta más rápida a la hora de realizar consultas de búsqueda. Tanto como para Usuarios como para las Tarjetas de Crédito tendremos un repositorio de almacenamiento local en una base de datos SqlLite. Este almacenamiento local actuará de forma que sea una respuesta media a la hora de realizar operaciones de búsqueda. Finalmente tendremos un almacenamiento remoto tanto en Usuarios como en Tarjetas. En cuanto a los usuarios, tenemos un repositorio que se comunica con una API rest. En cuanto a las Tarjetas, tenemos una base de datos PostgresSql albergada en un contenedor de Docker. Ambos almacenamientos remotos actuarán de forma que sea la respuesta más lenta que podremos obtener a la hora de realizar consultas de búsqueda.



En nuestro caso el viewModel actuaría como el servicio ya que es una aplicación backend local que no tiene conexión con ningún servidor web, por lo tanto no tiene interfaz.

2-Objetivos generales y específicos del proyecto.

El **objetivo general** de este proyecto es desarrollar un sistema en **Java** que permita gestionar los **clientes** de un banco y sus respectivas **tarjetas de crédito**, siguiendo un enfoque modular y eficiente. El sistema debe ofrecer una API REST para la gestión de usuarios, integrarse con bases de datos tanto locales (SQLite) como remotas (PostgreSQL), y garantizar el cumplimiento de una serie de requisitos de rendimiento, validación, exportación de datos y notificaciones. Además, la solución debe implementarse siguiendo buenas prácticas de programación, incluyendo gestión asincrónica, registro de operaciones y pruebas exhaustivas de todos los componentes. Finalmente, el sistema deberá ser desplegado utilizando **Docker** y su infraestructura probada mediante **TestContainers**, asegurando así una implementación confiable y escalable.

Objetivos Específicos

1. **Diseñar e implementar una API REST que permita realizar operaciones CRUD sobre los clientes del banco.**

2. **Integrar una política de caché LRU para la optimización del sistema.**

3. **Configurar la integración de bases de datos relacionales tanto en PostgreSQL como en SQLite.**

4. **Validar los datos de usuarios y tarjetas antes de almacenarlos.**

5. **Implementar la importación y exportación de datos en formato JSON y CSV.**

6. **Desarrollar un sistema de notificaciones que informe sobre los cambios en la información de los clientes.**

7. **Gestionar la configuración del sistema mediante archivos .properties y .env.**

8. **Desarrollar un sistema de logging para registrar todas las operaciones realizadas en el sistema.**

9. **Realizar pruebas unitarias y de integración para asegurar el correcto funcionamiento de los componentes.**

10. **Implementar y desplegar la infraestructura del sistema utilizando Docker.**

11. **Documentar el código y el proyecto mediante un archivo README.md que detalle las instrucciones de uso e instalación.**

3- Tecnologías usadas.

Git

DESCRIPCIÓN
Hemos usado git para el control de versiones del código empleando git Flow para llevar a cabo un desarrollo limpio y organizado.

GitHub

DESCRIPCIÓN
Hemos usado GitHub para albergar el repositorio de forma remota y trabajar mediante Pull Request para cada feature del programa.

PostgreSQL

DESCRIPCIÓN
Hemos usado PostgreSql para almacenar las tarjetas de crédito de forma remota en un contenedor de Docker.

Docker

DESCRIPCIÓN
Hemos usado Docker para almacenar la base de datos Postgre en un contenedor. Además hemos usado Docker para el despliegue continuo de la aplicación de forma que siempre tenga que pasar los tests para generar el archivo .jar de la app.

DockerHub

DESCRIPCIÓN
Hemos usado Docker hub para subir la imagen de la aplicación.

SQLite

DESCRIPCIÓN
Hemos usado la base datos SQLite para almacenar tanto a usuarios y tarjetas de forma local.

4- Algoritmo del proyecto.

El algoritmo principal del programa se basa en el patrón repository y el empleo de diferentes repositorios como una caché multinivel.

Al iniciar el programa el servicio cargará en los datos de los almacenamientos remotos en el almacenamiento local tanto de Usuarios como de Tarjetas y este almacenamiento local se irá actualizando, trayendo los cambios del remoto cada 30 segundos.

En el caso de que queramos buscar a todos los clientes registrados en nuestra aplicación, el servicio se ocupará de realizar una petición al almacenamiento local para que nos muestre todos los clientes almacenados.

En el caso de que queramos buscar a un cliente por su id o nombre, el servicio buscará en la caché un usuario con el id o nombre del cliente. En caso de que no este en la caché se buscara por ese id o nombre en el repositorio local de usuarios. En caso de que no se encuentre en el repositorio local, se hará una petición de búsqueda a la API y en caso de que no este se devolverá un error de que no se ha encontrado el cliente. En caso de encontrarlo, se aplicará el mismo proceso para buscar si existe una tarjeta de crédito asociada al usuario. Se buscará en la cache, luego en repositorio local y luego en el repositorio remoto que se conecta con la BBDD Postgre del contenedor. En caso de no encontrar ninguna tarjeta asociada devolverá el cliente sin ninguna tarjeta.

En caso de que queramos crear un cliente el servicio recibirá el cliente a crear. Primero validará el formato del usuario y luego el de la tarjeta de crédito si es que tiene tarjeta. En caso de que fallen cualquiera de los validadores saltará un error. Si han sido validados con éxito se creará el cliente en el almacenamiento remoto.

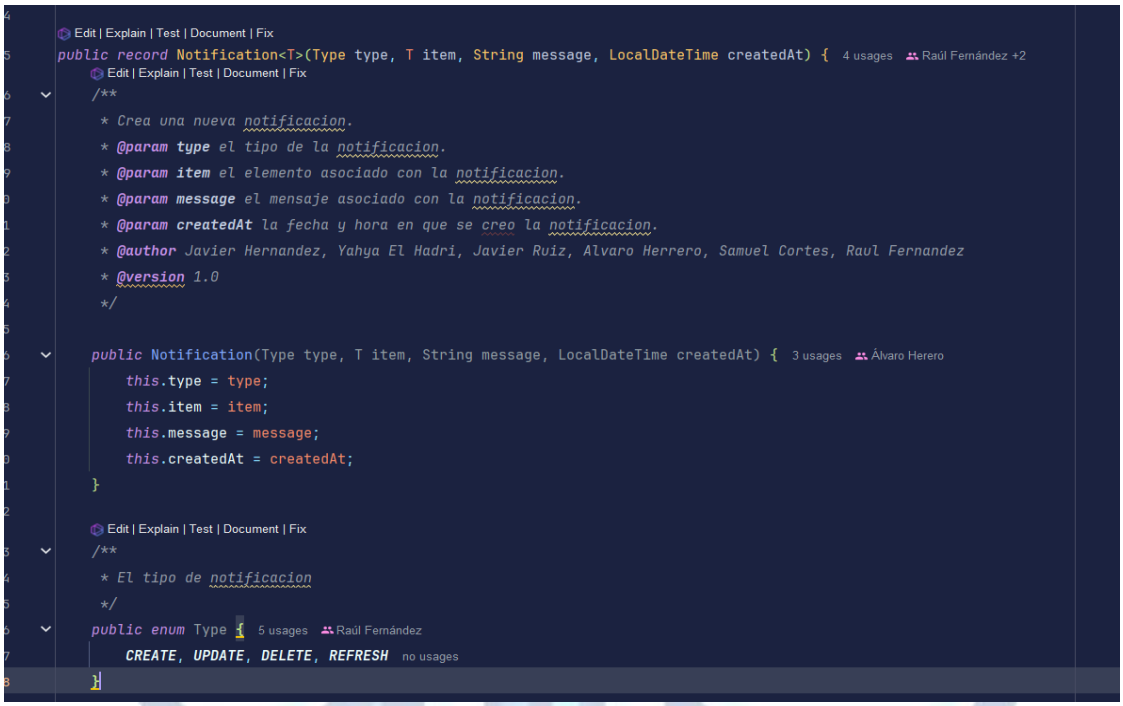
En caso de que queramos actualizar un cliente deberemos de pasarle al servicio el id del cliente que queremos actualizar y el cliente actualizado. Lo primero que haremos es buscar el cliente por el id. Luego validaremos el cliente actualizado. En caso de que sea valido eliminaremos el anterior registro en la cache y añadiremos el usuario y la tarjeta (en caso de que tenga) a los almacenamientos locales.

En caso de que queramos borrar un cliente deberemos introducir el id del cliente que queramos borrar. Lo primero que haremos es buscar si existe, primero en la cache, luego en el almacenamiento local y finalmente en el remoto. En caso de no existir se devolverá un error de que no se ha podido borra el cliente. En caso de que lo encontremos deberemos buscar si tiene alguna tarjeta de crédito asociada para borrarla. Las buscaremos de la misma forma, primero en la cache, si no está en el almacenamiento local y si no está en el remoto. En caso de que no haya ninguna tarjeta solo se borraría el usuario.

5- Principios SOLID.

1.S- Single responsibility Principle (SRP):

Cada clase debe tener **una única responsabilidad** o razón de cambio. Esto significa que una clase debe estar enfocada en hacer solo una cosa, lo que facilita su mantenimiento y evolución.



Este código es buen ejemplo del Single Responsibility Principle porque encapsula la lógica de la notificación de manera clara y eficiente. Cada parte de la clase está diseñada para cumplir con su función específica sin superponerse a otras responsabilidades.

2.O- Open/Closed Principle (OCP):

El código debe estar abierto para la extensión, pero cerrado para la modificación. Esto significa que debes poder agregar nuevas funcionalidades sin cambiar el código existente, normalmente usando la herencia o interfaces.

```
*/
public static Usuario toUserFromCreate(ResponseUserGetAll responseUserGetAll) {  1 usage  Samuel Cortés Sánchez
    return Usuario.builder()
            .id((long) responseUserGetAll.getId())
            .name(responseUserGetAll.getName())
            .username(responseUserGetAll.getUsername())
            .email(responseUserGetAll.getEmail())
            .build();
}
Edit | Explain | Test | Document | Fix
public static Usuario toUserFromCreate(ResponseUserGetByName responseUserGetByName) {  1 usage  Javier
    return Usuario.builder()
            .id((long) responseUserGetByName.getId())
            .name(responseUserGetByName.getName())
            .username(responseUserGetByName.getUsername())
            .email(responseUserGetByName.getEmail())
            .build();
}
```

En este caso, se aplica de manera efectiva porque permite a los desarrolladores agregar nuevos métodos de conversión sin necesidad de modificar los existentes. Esta capacidad de extender el sistema de manera controlada mejora la calidad del código y reduce el riesgo de errores, facilitando el mantenimiento y la evolución del software a lo largo del tiempo.

3.L- Liskov Substitution Principle (LSP):

Los objetos de una clase derivada deben ser reemplazables por objetos de su clase base sin alterar el comportamiento del programa.

Se aplica porque cualquier clase que implemente de CreditCardLocalRepository como CreditCardLocalRepositoryImpl puede ser utilizada de manera intercambiable en el sistema. Si más adelante se implementa otra clase que cumple con la interfaz para un almacenamiento remoto o en la nube, podría sustituirse la implementación actual sin afectar al código cliente que usa la interfaz.

4.L – Interface Segregation Principle (ISP):

Es mejor tener muchas interfaces específicas y pequeñas en lugar de una interfaz grande y general. Las clases no deberían estar obligadas a implementar métodos que no utiliza.

```
public Either<ServiceError, Cliente> getClienteById(Long id) {
    try {
        Optional<Usuario> usuario = Optional.empty();
        if(cacheUsuario.containsKey(id)){
            usuario = Optional.ofNullable(cacheUsuario.get(id));
        }else {
            usuario = usersRepository.findUserById(id);
            if(usuario.isEmpty()) {
                usuario = Optional.ofNullable(userRemoteRepository.getById(id));
            }
            usuario.ifPresent(u → cacheUsuario.put(u.getId(), u));

        }
        if(usuario.isEmpty()){
            return Either.left(new ServiceError.ClienteNotFound("Cliente no encontrado con id: " + id));
        }
        Optional<List<TarjetaCredito>> tarjetas = Optional.ofNullable(cacheTarjeta.buscarPorIdUsuario(id));
        if(tarjetas.isEmpty()){
            tarjetas = Optional.ofNullable(creditCardLocalRepository.findAllCreditCardsByUserId(id));
            if(tarjetas.isEmpty()){
                tarjetas = Optional.ofNullable(creditCardRepository.findAllCreditCardsByUserId(id));
            }
        }
        Cliente clienteDef = new Cliente(usuario.get(), tarjetas.orElse(new ArrayList<>()));
        return Either.right(clienteDef);
    }catch (Exception e){
        logger.error("Error al obtener el cliente con id: {}", id, e);
```

```
@Override 2 usages  Javier
public Either<ServiceError, Usuario> getUserById(Long id){
    try {
        Optional<Usuario> usuario = Optional.empty();
        if(cacheUsuario.containsKey(id)){
            usuario = Optional.ofNullable(cacheUsuario.get(id));
        } else {
            usuario = usersRepository.findUserById(id);
            if(usuario.isEmpty()) {
                usuario = Optional.ofNullable(userRemoteRepository.getById(id));
            }
            usuario.ifPresent(u → cacheUsuario.put(u.getId(), u));
        }
        if(usuario.isEmpty()){
            return Either.left(new ServiceError.UserNotFound("Usuario no encontrado con id: " + id));
        }
        return Either.right(usuario.get());
    }catch (Exception e){
        logger.error("Error al obtener el usuario con id: {}", id, e);
        return Either.left(new ServiceError.UserNotFound("Error al obtener el usuario con id: " + id));
    }
}
```

```
@Override 2 usages  Javier
public Either<ServiceError, TarjetaCredito> getTarjetaById(UUID id){
    try {
        Optional<TarjetaCredito> tarjeta = Optional.empty();
        if(cacheTarjeta.containsKey(id)){
            tarjeta = Optional.ofNullable(cacheTarjeta.get(id));
        } else {
            tarjeta = Optional.ofNullable(creditCardLocalRepository.findCreditCardById(id));
            if(tarjeta.isEmpty()) {
                tarjeta = creditCardRepository.getById(id);
            }
            tarjeta.ifPresent(t → cacheTarjeta.put(t.getId(), t));
        }
        if(tarjeta.isEmpty()){
            return Either.left(new ServiceError.TarjetasLoadError("Tarjeta no encontrada con id: " + id));
        }
        return Either.right(tarjeta.get());
    }catch (Exception e){
        logger.error("Error al obtener la tarjeta con id: {}", id, e);
        return Either.left(new ServiceError.TarjetasLoadError("Error al obtener la tarjeta con id: " + id));
    }
}
```

Estas partes del código están los métodos de ClienteServiceImpl, que obtienen clientes por su id, tarjetas por su id y usuarios por su id, al dividir la interfaz original en varias interfaces más pequeñas y específicas, se mejora la organización y mantenibilidad del código.

5.D – Dependency Inversion Principle (DIP):

Los módulos de alto nivel no deben depender de módulos de bajo nivel, sino que ambos deben depender de abstracciones. Además, las abstracciones no deben depender de detalles; los detalles deben depender de abstracciones.

Se utiliza el DIP en estos casos para garantizar un código más flexible y fácil de mantener, permitiendo modificar o reemplazar implementaciones sin romper dependencias en las clases que las usan.

6- Autores.

- https://github.com/Javierhvicente
- https://github.com/rraul10
- https://github.com/Samuceese
- https://github.com/javi97ruiz
- https://github.com/13elhadri
- https://github.com/alvarito304

7- Bibliografía.

- https://github.com/BaeSuii/FluxNews
- https://jlasoc.medium.com/qué-es-la-arquitectura-centrada-en-el-dominio-10542ea87afe
- https://dev.to/ebarrioscode/patron-repositorio-repository-pattern-y-unidad-de-trabajo-unit-of-work-en-asp-net-core-webapi-3-0-5goj
- https://github.com/joseluisgs

Fin.

YR RS
JYRS
BANCO