Práctica Banco

PROGRAMACIÓN – 2º DAW IES LUIS VIVES

Jaime León Mulero

Germán Fernández Carracedo

Alba García Orduña

Natalia González Alvarez

Mario de Domingo Álvarez

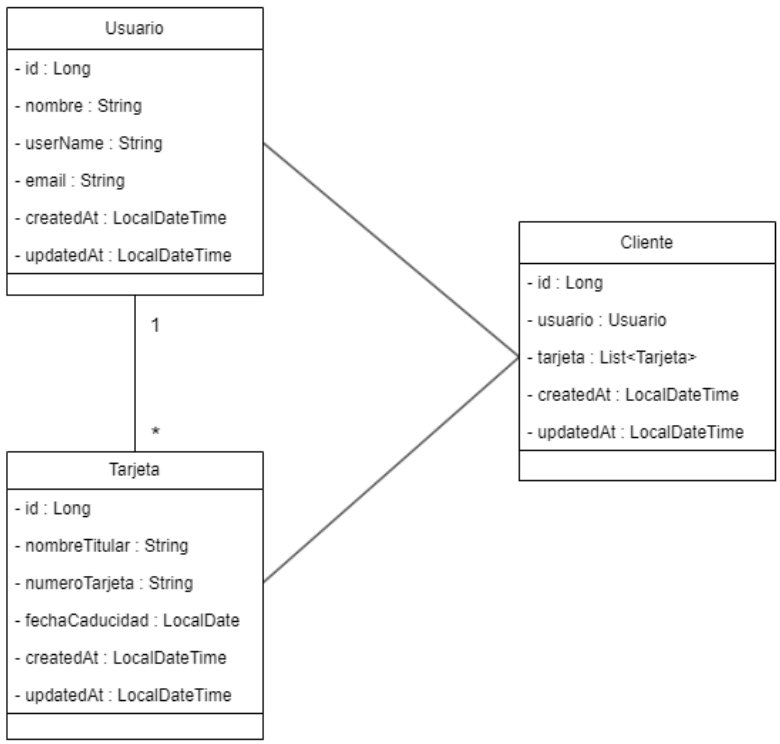
ÍNDICE

Introducción

En esta práctica vamos a realizar el servicio de un banco. Vamos a registrar los usuarios y sus tarjetas, teniendo en cuenta que un usuario puede tener 0 o más tarjetas, pero una tarjeta no puede no tener usuario.

Vamos a seguir una estructura orientada al dominio teniendo en cuenta los principios SOLID. A su vez, tendremos un patrón repositorio con caché multinivel que consta de un servicio compuesto por un sistema de notificaciones, una caché LRU y tres repositorios (uno local y dos remotos).

Diagrama de Clases



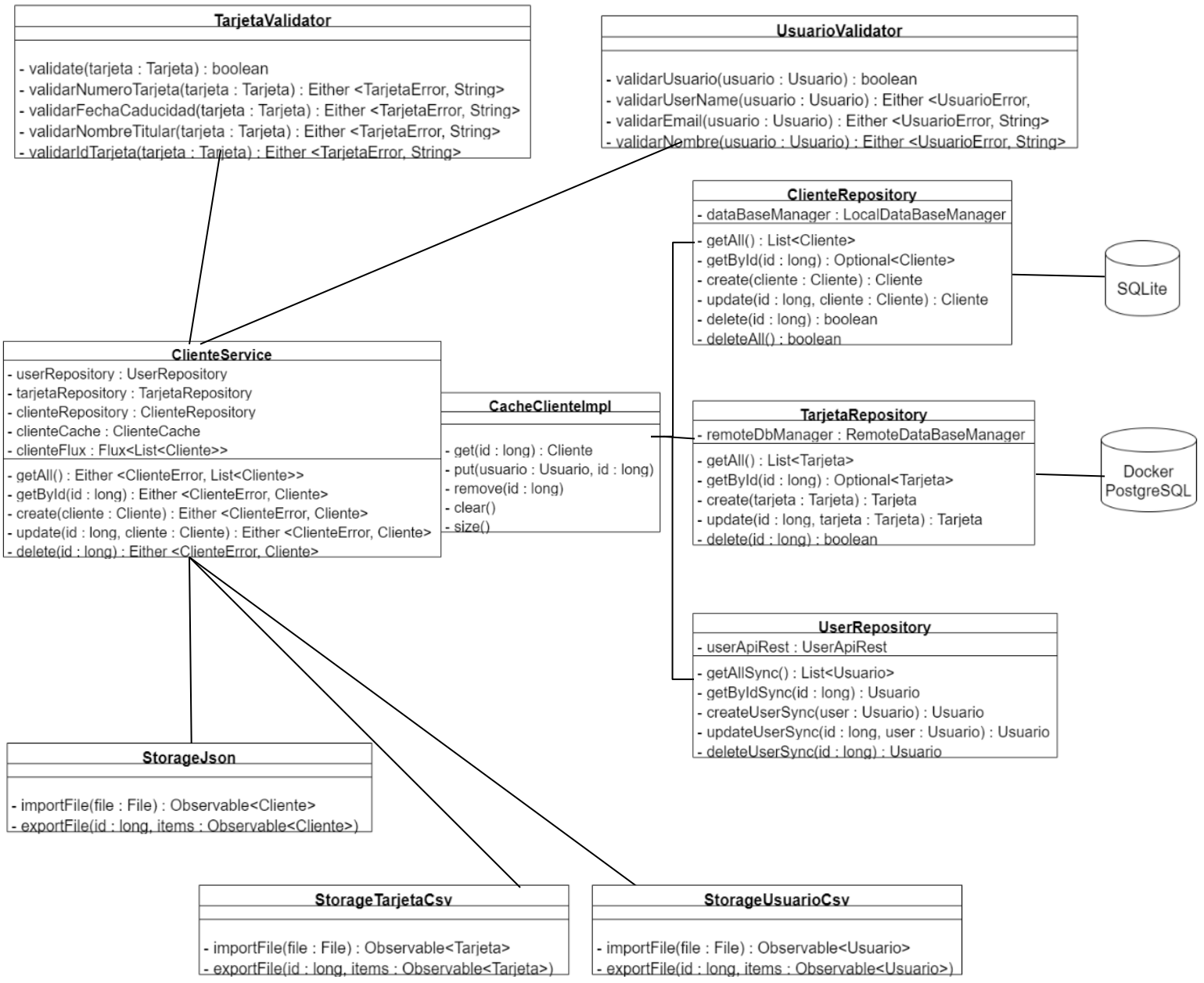
Un diagrama de clases es un tipo de diagrama de estructura estática que describe la estructura de un sistema mostrando las clases del sistema, sus atributos, operaciones (o métodos) y las relaciones entre los objetos.

En este caso contamos con una clase Usuario que contiene los datos: id, nombre, username, email y las fechas de creación y actualización del usuario del cliente. También tenemos una clase Tarjeta, la cual acumula: id, titular, número de tarjeta, fecha de caducidad, fecha de creación y actualización de la tarjeta de crédito del cliente.

Ambas clases se recogen en la clase Cliente, la cual alberga el id del cliente, su usuario (con todos sus datos), una lista de tarjetas, la fecha de creación y la fecha de actualización del cliente.

Es fundamental establecer una relación clara entre los usuarios y sus tarjetas. Cada tarjeta estará vinculada a un único usuario. Esto permitirá que un usuario tenga múltiples tarjetas bajo su cuenta, pero asegurará que cada tarjeta pertenezca a un solo usuario.

Esquema del funcionamiento del programa



Este diagrama representa la estructura que sigue nuestro programa. A continuación, explicaremos cada uno de los componentes del diagrama y su relación en el sistema.

ClienteService:

Este servicio actúa como el núcleo de la lógica de negocio del sistema. Su principal responsabilidad es la gestión de los clientes y la interacción con varios repositorios y servicios de validación.

Se compone de los métodos:

- getAll(): Recupera todos los clientes, interactuando con los diferentes repositorios.
- getById(): Obtiene un cliente por su ID.
- create(): Crea un nuevo cliente, almacenando la información tanto en el repositorio local como en la caché.
- update(): Actualiza los datos de un cliente.
- delete(): Elimina un cliente del sistema.

ClienteRepository:

Se encarga de las operaciones CRUD sobre los clientes en la base de datos local (usando SQLite).

Se compone de los métodos:

- getAll(): Devuelve una lista de todos los clientes.
- getById(): Busca un cliente por su identificador.
- create(): Inserta un nuevo cliente en la base de datos.
- update(): Actualiza la información de un cliente.
- delete(): Elimina un cliente de la base de datos.
- deleteAll(): Borra todos los registros de clientes.

TarjetaRepository:

Responsable de la gestión de las tarjetas asociadas a los usuarios, interactúa con una base de datos remota (Docker PostgreSQL) para operaciones como la creación, actualización y eliminación de tarjetas.

Se compone de los métodos:

- getAll(): Obtiene una lista de todas las tarjetas.
- getById(): Busca una tarjeta por su identificador.
- create(): Inserta una nueva tarjeta en la base de datos.
- update(): Actualiza los datos de una tarjeta existente.
- delete(): Elimina una tarjeta de la base de datos.

UserRepository:

Administra las operaciones sobre los usuarios mediante una API externa (userApiRest).

Se compone de los métodos:

- getAllSync(): Obtiene todos los usuarios de manera síncrona.
- getByIdSync(): Recupera un usuario por su identificador de manera síncrona.
- createUserSync(): Crea un nuevo usuario de manera síncrona.
- updateUserSync(): Actualiza un usuario de manera síncrona.
- deleteUserSync(): Elimina un usuario de manera síncrona.

TarjetaValidator:

Contiene las reglas de validación para las tarjetas, asegurándose de que los datos de la tarjeta sean correctos antes de que se almacenen o utilicen en el sistema.

Se compone de los métodos:

- validate(): Valida la tarjeta en su totalidad.
- validarNumeroTarjeta(): Valida el número de tarjeta.
- validarFechaCaducidad(): Verifica que la fecha de caducidad de la tarjeta sea válida.
- validarNombreTitular(): Comprueba si el nombre del titular de la tarjeta es correcto.
- validarIdTarjeta(): Valida el identificador único de la tarjeta.

UsuarioValidator:

Encargado de validar la información de los usuarios antes de proceder con cualquier operación.

Se compone de los métodos:

- validarUsuario(): Valida el objeto usuario en su conjunto.
- validarUserName(): Verifica el nombre de usuario.
- validarEmail(): Comprueba que el correo electrónico del usuario sea válido.
- validarNombre(): Valida el nombre del usuario.

CacheClienteImpl:

Implementa un sistema de caché para almacenar temporalmente información de los clientes. Esto permite mejorar el rendimiento, reduciendo las consultas a la base de datos cuando los datos ya se encuentran en la memoria.

Se compone de los métodos:

- get(): Recupera un cliente por su ID desde la caché.
- put(): Almacena un cliente en la caché.
- remove(): Elimina un cliente de la caché.
- clear(): Limpia toda la caché.
- size(): Devuelve el tamaño actual de la caché.

StorageJson:

Permite la importación y exportación de datos de clientes en formato JSON.

Se compone de los métodos:

- importFile(): importa los clientes desde el archivo JSON de manera reactiva.
- exportFile(): exporta los clientes a un archivo JSON de manera reactiva.

StorageTarjetaCsv:

Se encarga de la importación y exportación de datos de tarjetas en formato CSV. Se compone de los métodos:

- importFile(): importa las tarjetas desde el archivo CSV.
- exportFile(): exporta las tarjetas a un archivo CSV.

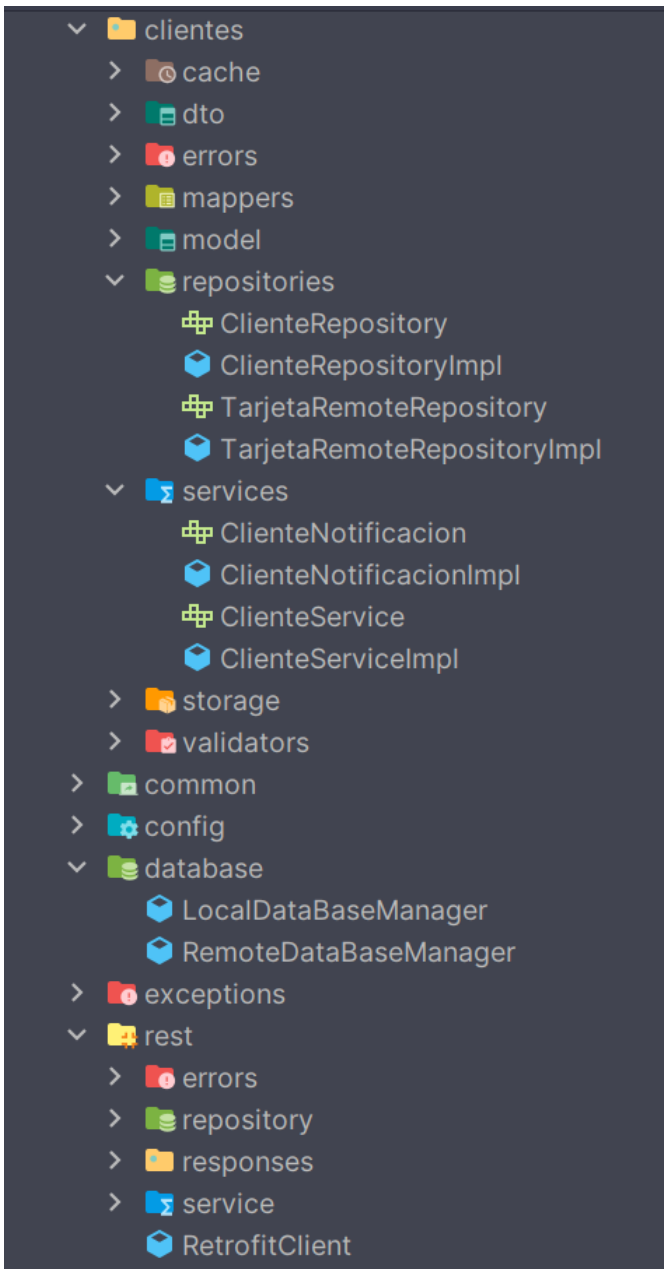
StorageUsuarioCsv:

Similar a StorageTarjetaCsv, pero orientado a los usuarios. Permite importar y exportar datos de usuarios en archivos CSV.

Se compone de los métodos:

- importFile(): importa los usuarios desde el archivo CSV.
- exportFile(): exporta los usuarios a un archivo CSV.

Al final nuestro proyecto quedará organizado de la siguiente manera

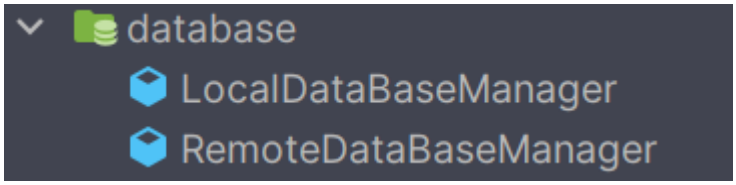


Bases de Datos

El sistema utiliza dos bases de datos.

SQLite es la base de datos que usamos para almacenar la información de los clientes de manera local. Esta base de datos es bastante ligera y sencilla de utilizar, lo que la hace ideal para aplicaciones que no requieren conectarse a una red o manejar grandes cantidades de datos. Es útil cuando queremos que los datos se queden en el dispositivo o en un entorno más pequeño y controlado.

Por otro lado, para el almacenamiento de las tarjetas de los usuarios, usamos PostgreSQL en un entorno Docker. PostgreSQL es una base de datos más avanzada y robusta, lo que la convierte en la mejor opción cuando necesitamos manejar una gran cantidad de datos o trabajar en un sistema distribuido, como en la nube.



DockerFile

Un Dockerfile es un archivo de texto que define los pasos para crear una imagen de Docker. En este caso, se usa una estrategia de construcción en dos fases.

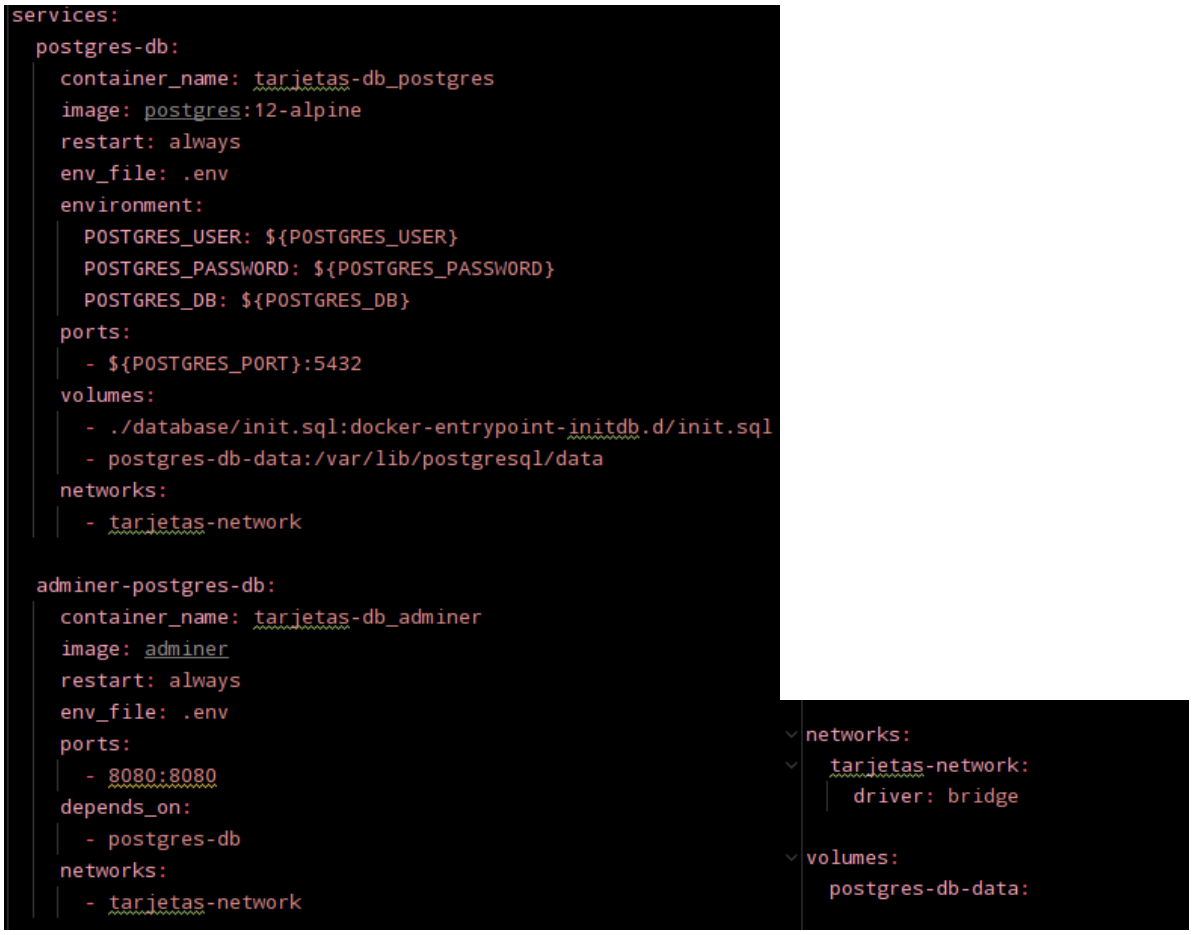
```
FROM gradle:jdk21-alpine AS build
WORKDIR /app

COPY build.gradle.kts .
COPY gradlew .
COPY gradle gradle
COPY src src

ARG DOCKER_HOST_ARG=tcp://host.docker.internal:2375
ENV DOCKER_HOST=$DOCKER_HOST_ARG

RUN ./gradlew build

FROM eclipse-temurin:21-jre-alpine AS run

WORKDIR /app

COPY --from=build /app/build/libs/*SNAPSHOT.jar /app/my-app.jar

ENTRYPOINT ["java","-jar","/app/my-app.jar"]
```

Primero, en la etapa de "build", se utiliza la imagen *gradle:jdk21-alpine*, que incluye Gradle y Java. Aquí se copian los archivos del proyecto, como el código fuente y las configuraciones de Gradle, y se ejecuta el comando *./gradlew build* para compilar el proyecto, generando un archivo .jar que contiene la aplicación lista para ejecutarse.

En la segunda fase, llamada "run", se usa la imagen *eclipse-temurin:21-jre-alpine*, que es más ligera porque solo contiene el entorno de ejecución de Java (JRE). Luego, se copia el archivo .jar generado en la primera fase y se define el comando de arranque del contenedor para que ejecute la aplicación usando *java -jar*. Esto permite que la imagen final sea más pequeña y eficiente, ya que solo incluye lo necesario para correr la aplicación. Esta estrategia optimiza el tamaño final de la imagen, garantizando que solo lo necesario para ejecutar la aplicación esté presente.

Docker-Compose

Un archivo Docker Compose es una herramienta que permite definir y ejecutar aplicaciones que constan de múltiples contenedores Docker. Utiliza un archivo de configuración en formato YAML para especificar los servicios, redes y volúmenes necesarios para que la aplicación funcione correctamente.



```yaml
services:
  postgres-db:
    container_name: tarjetas-db_postgres
    image: postgres:12-alpine
    restart: always
    env_file: .env
    environment:
      POSTGRES_USER: ${POSTGRES_USER}
      POSTGRES_PASSWORD: ${POSTGRES_PASSWORD}
      POSTGRES_DB: ${POSTGRES_DB}
    ports:
      - ${POSTGRES_PORT}:5432
    volumes:
      - ./database/init.sql:docker-entrypoint-initdb.d/init.sql
      - postgres-db-data:/var/lib/postgresql/data
    networks:
      - tarjetas-network

  adminer-postgres-db:
    container_name: tarjetas-db_adminer
    image: adminer
    restart: always
    env_file: .env
    ports:
      - 8080:8080
    depends_on:
      - postgres-db
    networks:
      - tarjetas-network

networks:
  tarjetas-network:
    driver: bridge

volumes:
  postgres-db-data:
```

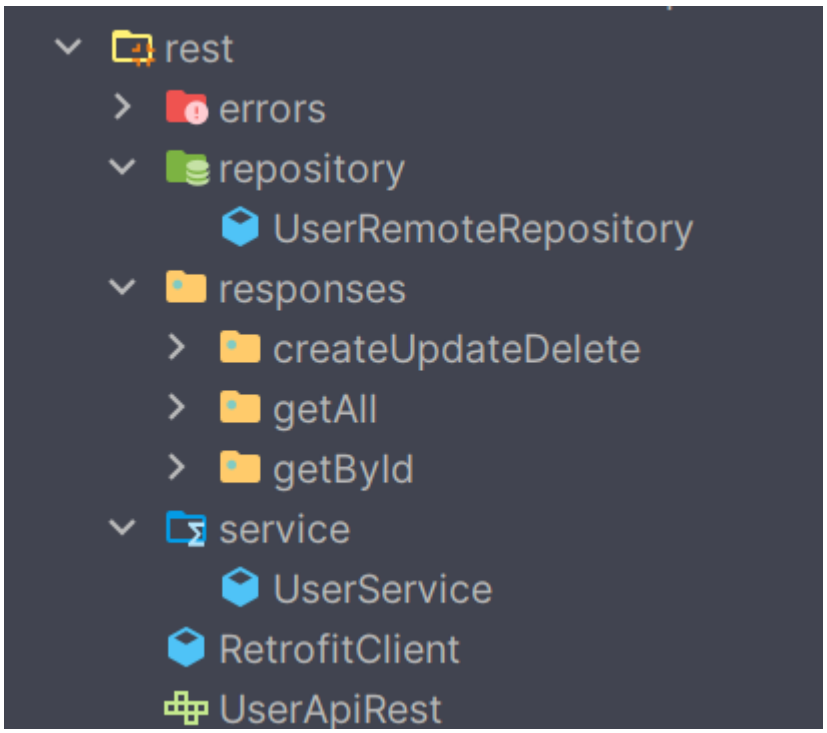
En el archivo Docker Compose configuramos dos servicios: postgres-db y adminer-postgres-db.

El servicio postgres-db utiliza la imagen *postgres:12-alpine*. Dentro de esta sección, se definen variables de entorno que se cargan desde un archivo *.env*, lo que permite establecer fácilmente el usuario, la contraseña y el nombre de la base de datos que se crearán de forma automática. Además, el puerto definido en la variable *${POSTGRES_PORT}* del archivo .env se asigna al puerto 5432 del contenedor. También se especifican dos volúmenes: uno que inicializa la base de datos utilizando el archivo *init.sql* y otro que asegura la persistencia de los datos, de modo que no se pierdan al reiniciar el contenedor.

Por otro lado, el segundo servicio, hace uso de la imagen *adminer*, que es una herramienta útil para gestionar bases de datos de forma visual. Este servicio está diseñado para depender del contenedor de *postgres-db*, lo que significa que se levantará automáticamente una vez que el servicio de PostgreSQL esté en funcionamiento. Adminer está configurado para estar disponible en el puerto 8080, lo que permite acceder a su interfaz a través de un navegador web, facilitando la gestión de las bases de datos.

Además, el archivo establece una red llamada *tarjetas-network*, que utiliza el controlador de puente (bridge). Esto permite una comunicación segura y eficiente entre los dos servicios, asegurando que puedan interactuar entre sí sin complicaciones.

API Rest



En nuestro proyecto grupal se ha decidido separar las distintas responsabilidades entre repositorios, servicios y manejo de respuestas para que se pudiera manejar en paralelo en distintas partes del sistema sin conflictos. Además, hemos usado Retrofit como cliente HTTP para poder simplificar la comunicación con el servidor, mejorando la mantenibilidad a largo plazo del proyecto.

De esta forma se nos ha permitido un uso de la gestión eficiente de los datos y la comunicación con un servidor remoto, asegurando que los usuarios finales obtengan una experiencia fluida y sin interrupciones.

A continuación, vamos a detallar el funcionamiento de la API REST, basándonos en la estructura de carpetas y archivos. La estructura se presenta separando las responsabilidades en diferentes módulos, lo que facilita la comprensión, el mantenimiento y la escalabilidad del proyecto.

1. Directorio rest

Este directorio es el núcleo del proyecto, donde se centralizan los principales componentes de la API. Aquí se encuentran las carpetas que agrupan las funcionalidades esenciales como el manejo de errores, los repositorios de datos, las respuestas a las solicitudes HTTP y los servicios que se comunican con la API.

2. Subdirectorio errors

En este apartado se ubican los archivos destinados al manejo de errores.Incluye clases que gestionan excepciones personalizadas y controlan las respuestas de error que la API devuelve ante solicitudes fallidas. El objetivo de este módulo es asegurar que los errores sean manejados de manera adecuada y que se devuelvan respuestas claras al cliente, lo que contribuye a una mejor experiencia de usuario y facilita la depuración en caso de fallos.

3. Subdirectorio repository

- UserRemoteRepository: Este archivo contiene la implementación del repositorio remoto, que es responsable de realizar operaciones sobre los datos de los usuarios desde una fuente remota, como un servidor REST.El uso del patron Repository nos permite desacoplar la lógica de acceso a los datos de la lógica de negocio. Esto facilita cambios en la fuente de datos, como cambiar de una base de datos local a una API remota sin afectar el resto de la aplicación.

Este componente gestiona todas las operaciones CRUD (Crear, Leer, Actualizar, Eliminar) sobre los datos de los usuarios, interactuando con los servicios de red y asegurando que la aplicación reciba o envíe la información correcta al servidor.

4. Subdirectorio responses

Este directorio agrupa las clases y métodos encargados de manejar las respuestas que la API devuelve al cliente. Se subdivide en tres carpetas, cada una especializada en diferentes tipos de operaciones:

- createUpdateDelete: Esta carpeta se encarga de procesar las respuestas que se generan al realizar operaciones de creación, actualización o eliminación de usuarios en el servidor. Los métodos que se encuentren aquí mapearán estas respuestas en objetos comprensibles para el sistema.
- getAll: En esta carpeta se manejan las respuestas que devuelven todos los usuarios solicitados a través de la API. Estas respuestas generalmente incluyen listas o colecciones de objetos de usuarios.
- getById: Aquí se procesan las respuestas que involucran la obtención de información de un usuario específico a través de su ID. Este tipo de solicitud es común cuando se necesita mostrar o actualizar información detallada de un usuario en particular.

5. Subdirectorio service

El directorio *service* contiene la lógica de negocio y de comunicación con la API REST. Los servicios son los encargados de orquestar las interacciones entre el repositorio remoto y otras capas de la aplicación. Este directorio contiene tres componentes principales:

- UserService: Esta clase es responsable de gestionar las operaciones relacionadas con los usuarios, como la creación, lectura, actualización y eliminación de registros. Es la capa intermedia que comunica las solicitudes del cliente con el repositorio remoto, y puede definir métodos para realizar llamadas a la API de manera eficiente.
- RetrofitClient: Esta clase configura el cliente Retrofit,que se encarga de realizar llamadas HTTP a servicios REST. Aquí se gestionan aspectos importantes como la configuración de la base URL del servidor, los interceptores para manejar autenticación o errores, y los adaptadores de llamada que facilitan la conversión de las respuestas JSON a objetos Java.
- UserApiRest: Este archivo contiene la interfaz que define las llamadas HTTP que se realizarán a la API. Utiliza anotaciones como *@GET*, *@POST*, *@PUT* y *@DELETE* para indicar el tipo de operación que se va a realizar. Cada método en esta interfaz representa una llamada directa a la API REST, permitiendo obtener, crear, actualizar o eliminar usuarios en el servidor.

Flujo General de la API

El flujo de trabajo de nuestra API REST se describe de la siguiente manera:

1. Solicitud del cliente: Un cliente realiza una solicitud para interactuar con los datos de los usuarios. Esta solicitud es manejada por la clase *UserService*, que comunica la acción al repositorio remoto *UserRemoteRepository*.
2. Interacción con el servidor: A través del cliente Retrofit, las solicitudes HTTP se dirigen al servidor utilizando la clase *UserApiRest*. Esta clase contiene los métodos que se encargan de realizar las operaciones CRUD con el servidor, gestionando la creación, lectura, actualización y eliminación de datos de usuario.
3. Procesamiento de respuestas: Las respuestas recibidas del servidor son manejadas en el directorio *responses*. Dependiendo de la operación solicitada (crear, obtener todos los usuarios, obtener por ID, actualizar o eliminar), se procesan las respuestas de manera adecuada, ya sea en formato JSON o en otro formato estructurado, y se convierten en objetos que el resto de la aplicación puede utilizar.
4. Gestión de errores: Si alguna solicitud falla, la lógica de manejo de errores presente en el subdirectorio *errors* se encarga de devolver una respuesta adecuada al cliente, permitiendo identificar el problema y gestionar la respuesta de manera adecuada.

Test

En el desarrollo de software, los tests son una parte fundamental para asegurar la calidad y fiabilidad del proyecto. Los tests permiten verificar que cada componente de la aplicación funciona correctamente, tanto de forma aislada como en conjunto. Implementar tests ayuda a detectar errores tempranamente en el ciclo de desarrollo, lo que reduce el tiempo y costo de corregir fallos en etapas más avanzadas. Además, garantizan que el código siga funcionando como se espera después de cualquier cambio, ya sea una nueva funcionalidad o una refactorización.

Existen varios tipos de tests, como los unitarios, que verifican el comportamiento de una función o método específico; los tests de integración, que prueban cómo interactúan diferentes módulos del sistema; y los tests funcionales, que aseguran que el sistema cumpla con los requisitos del usuario.

Uso de JUnit y Mockito en el Proyecto

En nuestro proyecto, que originalmente estaba desarrollado en Java, hemos utilizado dos herramientas clave para llevar a cabo nuestras pruebas:

1. JUnit: Es una de las bibliotecas más utilizadas para realizar tests unitarios en aplicaciones Java. JUnit nos ha permitido verificar de manera automática el comportamiento de métodos individuales y asegurar que cada unidad de código cumpla con lo que se espera de ella. Al usar JUnit, hemos podido definir casos de prueba claros y concisos, con la capacidad de generar resultados rápidos y fáciles de interpretar. Esto nos ayuda a asegurar que el código funcione como debería, incluso cuando se realicen cambios o mejoras en el futuro.
2. Mockito: Para simular y verificar el comportamiento de dependencias externas en nuestras pruebas, hemos usado Mockito, lo que nos ha permitido simular dependencias como bases de datos, servicios remotos y otros componentes complejos. Gracias a esto, hemos podido aislar las funciones que estamos probando, sin necesidad de depender de la implementación real de otras partes del sistema. Esto facilita la creación de tests unitarios más eficientes y garantiza que los resultados de las pruebas sean confiables.

Beneficios de los Tests en Nuestro Proyecto

Al implementar JUnit y Mockito para testear nuestro proyecto, hemos conseguido:

- Mayor confianza en el código: Sabemos que, después de realizar cualquier cambio o refactorización, los tests garantizarán que no se introduzcan errores nuevos.
- Detección temprana de errores: Gracias a los tests unitarios, es más sencillo identificar fallos en el código a medida que avanzamos en el desarrollo, lo que nos permite solucionarlos de inmediato.
- Facilidad para mantener el código: Los tests nos han permitido refactorizar y mejorar el código sin miedo a romper funcionalidades existentes, ya que los tests nos avisan inmediatamente si algo deja de funcionar.
- Aislamiento de dependencias: Con Mockito, hemos sido capaces de probar componentes aislados, lo que significa que no necesitamos configurar bases de datos reales o servicios externos para validar el comportamiento de nuestras clases.

Uso de Testcontainers en Nuestro Proyecto

En nuestro proyecto, que utiliza **Java** y **JUnit** para las pruebas, hemos empleado **Testcontainers** para probar las interacciones de nuestro repositorio remoto con una base de datos **PostgreSQL**. Hemos definido una clase de prueba llamada *TarjetaRemoteRepositoryImplTest*, en la cual utilizamos Testcontainers para levantar una instancia de PostgreSQL en un contenedor durante las pruebas. Esto nos ha permitido realizar pruebas de integración realistas sin necesidad de depender de una base de datos externa.

A continuación, se detallan los principales componentes de nuestras pruebas:

1. **Configuración de PostgreSQL con Testcontainers**:
    - En la clase de pruebas, definimos un contenedor de PostgreSQL que se levanta automáticamente utilizando la anotación *@Container*. Esto asegura que la base de datos esté configurada con un script de inicialización (*init.sq*l) para que contenga los datos necesarios para nuestras pruebas.
    - Este contenedor se levanta antes de la ejecución de los tests y se detiene al final de las pruebas, lo que garantiza que las pruebas se realicen en un entorno limpio y consistente.

Bibliografía

Repositorios en GitHub

González Sánchez, J.L. (2024). *DesarrolloWebEntornoServidor- 01 -2024-2025* [Página web].

https://github.com/joseluisgs/DesarrolloWebEntornosServidor-01-2024-2025

González Sánchez, J.L. (2024). *JavaLocalAndRemote* [Página web].

https://github.com/joseluisgs/JavaLocalAndRemote

González Sánchez, J.L. (2024). *DespliegueAplicacionesWeb-03-2024-2025* [Página web].

https://github.com/joseluisgs/DespliegueAplicacionesWeb-03-2024-2025

González Sánchez, J.L. (2024). *DespliegueAplicacionesWeb-02-2024-2025* [Página web].

https://github.com/joseluisgs/DespliegueAplicacionesWeb-02-2024-2025

González Sánchez, J.L. (2024). *DespliegueAplicacionesWeb-01-2024-2025* [Página web].

https://github.com/joseluisgs/DespliegueAplicacionesWeb-01-2024-2025

González Sánchez, J.L. (2024). *Docker-tutorial* [Página web].

https://github.com/joseluisgs/docker-tutorial

Gitignore [Página web].

https://www.toptal.com/developers/gitignore

Valentin, A. (n.d.). *JSONPlaceholder* [API].

https://jsonplaceholder.typicode.com/