Arquitectura y componentes - Fake Twitter API

1. Resumen General

La aplicación Fake Twitter es una API REST desarrollada en Go que simula funcionalidades básicas de Twitter: publicar tweets, seguir usuarios y consultar el timeline. Utiliza MySQL como base de datos relacional y Docker para la orquestación de servicios y despliegue local.

2. Componentes Principales

2.1. API Backend (Go)

Framework: Go estándar, con gorilla/mux para el ruteo HTTP.

Estructura de Capas:

- Handlers: Reciben y responden a las solicitudes HTTP.
- Usecases: Implementan la lógica de negocio.
- Services: Encapsulan la interacción con los repositorios.
- Repositories: Acceso a la base de datos MySQL.
- Models: Definición de las entidades principales (usuarios, tweets, relaciones).

Variables de entorno: Se gestionan con .env y la librería joho/godotenv.

2.2. Base de Datos (MySQL)

Persistencia: Almacena usuarios, tweets y relaciones de seguimiento.

Inicialización: Se crean tablas e insertan datos de ejemplo mediante migraciones SQL automáticas.

¿Por qué una BD relacional? Tiene mucha ventaja al obtener los tweets de los usuarios seguidos, ya que es ir a buscar las relaciones entre usuarios y tweets, algo que no es tan óptimo de realizar en una BD no relacional como un KVS o DS.

Además, como una de las prioridades era que la aplicación tiene que estar optimizada para lecturas, a través de la implementación de índices en las tablas podemos realizar lecturas de manera más veloz.

2.3. Migraciones

Herramienta: migrate/migrate (contenedor Docker).

Función: Aplica scripts SQL para crear y poblar la base de datos al iniciar el entorno.

2.4. Orquestación y Despliegue (Docker & Docker Compose)

Servicios:

- app: Contenedor de la API Go.
- mysql: Contenedor de la base de datos.
- migrate: Contenedor que ejecuta las migraciones.

Ventajas:

- Aislamiento de dependencias.
- Facilidad para levantar y destruir el entorno completo.
- Persistencia de datos mediante volúmenes Docker.

2.5. Desacoplamiento por interfaces y flexibilidad

Cada capa de la aplicación define interfaces para los métodos que necesita de la capa inferior, en lugar de depender directamente de implementaciones concretas.

Esto permite:

- Desacoplamiento: Las capas solo conocen los métodos que requieren, no cómo están implementados.
- Intercambiabilidad: Se pueden reemplazar implementaciones fácilmente (por ejemplo, usar un mock en tests o cambiar la base de datos en el futuro).
- Facilidad de testing: Permite inyectar dobles de prueba (mocks/fakes) en los tests unitarios.
- Evolución: Si se necesita modificar o ampliar una capa, solo se debe cumplir con la interfaz esperada.

3. Flujo de una Solicitud

El cliente realiza una petición HTTP (por ejemplo, publicar un tweet).

El Handler correspondiente recibe la solicitud, valida los datos y llama al

Usecase.

El Usecase ejecuta la lógica de negocio y utiliza los Services necesarios.

El Service interactúa con el Repositorio para acceder o modificar datos en MySQL.

El Repositorio ejecuta las consultas SQL y retorna los resultados.

La respuesta se propaga de vuelta hasta el Handler, que responde al cliente.

4. Testing

Pruebas Unitarias:

- Handlers, usecases, services y repositories tienen tests table-driven.
- Se utilizan mocks y go-sqlmock para simular la base de datos.

Ejecución:

go test ./...

5. Ventajas de la Arquitectura

- Separación de responsabilidades: Cada capa tiene una función clara.
- Escalabilidad: Fácil de extender con nuevas funcionalidades.
- Portabilidad: El uso de Docker permite correr la aplicación en cualquier entorno compatible.
- Facilidad de pruebas: La estructura modular y el uso de interfaces facilitan el testing.

6. Tecnologías y Librerías Clave

- Go (backend)
- MySQL (base de datos)
- Docker & Docker Compose (orquestación)
- migrate/migrate (migraciones)
- gorilla/mux (ruteo HTTP)
- joho/godotenv (variables de entorno)
- go-sqlmock (tests de repositorios)