Documentación API GraphQL, Apollo Server, PostgresQL

Este proyecto se trata de una **API** diseñada para gestionar **usuarios**, **posts** y **comentarios**, permitiendo realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar). Además, incluye un sistema de **autenticación** de usuarios basado en **JWT (JSON Web Tokens)**, garantizando que solo los usuarios autorizados puedan acceder a ciertas funcionalidades.

La API está desarrollada utilizando **GraphQL**, lo que permite a los clientes hacer consultas personalizadas y obtener solo los datos necesarios. Esto hace que las solicitudes sean más eficientes y precisas

- ▼ Crear, actualizar y eliminar perfiles de usuarios.
- ▼ Crear, actualizar y eliminar **posts** vinculados a cada usuario.
- ▼ Crear, actualizar y eliminar **comentarios** en los posts publicados.
- ▼ Autenticar usuarios de manera segura mediante JWT para controlar el acceso a la API.

Objetivo del proyecto

- Desarrollar las funcionalidades básicas para la gestión de usuarios, posts y comentarios, junto con la implementación del sistema de autenticación mediante **JWT**.

Arquitectura del Sistema y Diagrama de Componentes

La arquitectura del sistema implementa un enfoque modular y escalable. Utiliza **GraphQL** como capa de interacción entre cliente y servidor, y **PostgreSQL** como sistema de gestión de bases de datos. El diseño prioriza la eficiencia, flexibilidad y

facilidad de mantenimiento, con un control centralizado de migraciones mediante **Liquibase**.

Componentes Clave:

- **Backend (Apollo Server con GraphQL)**: Implementa **Apollo Server** para gestionar las solicitudes de **GraphQL**. La API maneja consultas (GET) y mutaciones (POST/PUT) para usuarios, posts y comentarios.
- **Base de datos (PostgreSQL)**: Gestiona el almacenamiento de usuarios, posts y comentarios mediante una estructura relacional que garantiza la integridad y eficiencia de los datos.
- **Autenticación (JWT)**: El sistema de **JWT** controla el acceso a los recursos de la API, permitiendo operaciones solo a usuarios autenticados.
- **Migraciones de Base de Datos (Liquibase)**: **Liquibase** administra las migraciones de forma controlada y reproducible, asegurando cambios ordenados en la estructura de la base de datos.
- **Contenerización (Docker)**: **Docker** facilita la contenerización de la aplicación para un despliegue simplificado en entornos locales y de producción.

Estructura del Proyecto

```
src/
├── bd/
│   └── cnn.js          # Conexión a la base de datos PostgreSQL
├── models/
│   ├── comments.js     # Modelo para los comentarios
│   ├── posts.js        # Modelo para las publicaciones
│   └── users.js        # Modelo para los usuarios
├── resolvers/
│   ├── comments/
│   │   ├── commentsResolvers.js     # Lógica de resoluciones pa
```

```
│   │   └── commentsResolvers.test.js # Pruebas de resoluciones
│   ├── posts/
│   │   ├── postResolvers.js         # Lógica de resoluciones pa
│   │   └── postResolvers.test.js    # Pruebas de resoluciones p
│   ├── users/
│   │   ├── userResolvers.js         # Lógica de resoluciones pa
│   │   └── userResolvers.test.js    # Pruebas de resoluciones p
│   ├── resolvers.js                 # Consolidación de todos lo
│   └── resolvers.test.js            # Pruebas unitarias de todo
├── schemas/
│   └── typeDef.js                   # Definición del esquema Gr
├── errors.js                         # Manejo de errores global
├── app.js                            # Configuración de la apli
└── server.js                         # Inicialización del servi
```

Requisitos del Sistema

- Node.js (versión 16 o superior)
- npm o yarn como gestor de paquetes
- Docker (opcional) para la contenerización de la base de datos y la aplicación
- PostgreSQL (requerido solo si no se utiliza Docker)
- Liquibase para el control de versiones de la base de datos
- Jest para pruebas unitarias

Configuración del Entorno

```
# Clonar el repositorio
git clone https://github.com/vaneessup/PruebaTec

# Instalar dependencias
npm install
```

Configuración de la Base de Datos

Para configurar la base de datos, sigue estos pasos:

1. Configuración de PostgreSQL

⚙️ Usando PostgreSQL localmente

Si prefieres ejecutar PostgreSQL localmente, sigue estos pasos:

1. Instala PostgreSQL y crea una base de datos.
2. Configura las variables de entorno en el archivo `.env` (más abajo).

📝 Configuración del entorno

Crea un archivo `.env.local` en la raíz del proyecto con las siguientes variables:

```
DB_HOST=localhost
DB_PORT=5432
DB_USER=tu_usuario
DB_PASSWORD=tu_clave
DB_NAME=tu_bd
```

QUERY para la creación de la bd y tablas

```
-- Crear la base de datos
CREATE DATABASE nombre_bd;

-- Crear las tablas necesarias
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL UNIQUE,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE posts (
    id SERIAL PRIMARY KEY,
```

```
    title VARCHAR(200) NOT NULL,
    content TEXT NOT NULL,
    user_id INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);

CREATE TABLE comments (
    id SERIAL PRIMARY KEY,
    text TEXT NOT NULL,
    post_id INT NOT NULL,
    user_id INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (post_id) REFERENCES posts(id) ON DELETE CASCADE
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE
);
```

DIAGRAMA ER

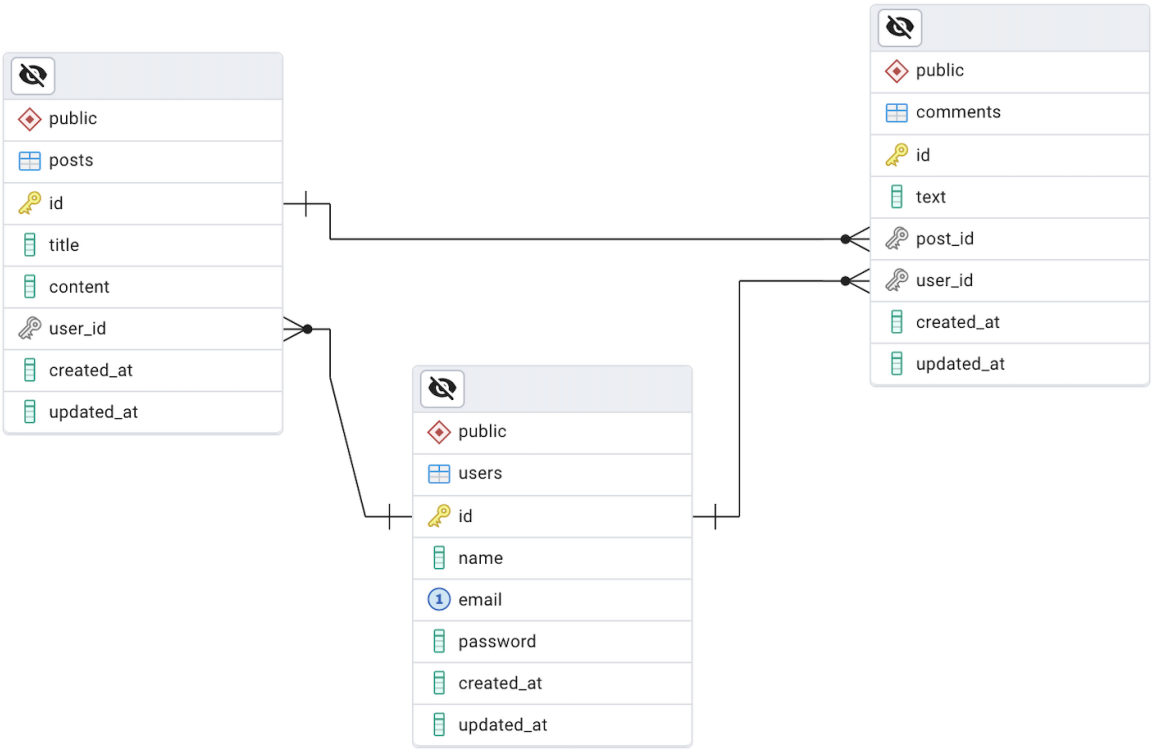


Diagrama ER Resumen:

1. `users` **(1) ↔ (N)** `posts`: Un usuario puede tener muchas publicaciones.
2. `posts` **(1) ↔ (N)** `comments`: Un post puede tener muchos comentarios.
3. `users` **(1) ↔ (N)** `comments`: Un usuario puede realizar muchos comentarios.

2. Archivo de Configuración

Crear un archivo `src/bd/cnn.js`:

```
const { Pool } = require('pg');
require('dotenv').config();

const pool = new Pool({
  user: process.env.DB_USER,
  host: process.env.DB_HOST,
  database: process.env.DB_NAME,
  password: process.env.DB_PASSWORD,
```

```
  port: process.env.DB_PORT,
});

module.exports = pool;
```

Archivos de Configuración Adicionales

1. Configurar el archivo `liquibase.properties`:

```
changeLogFile=db/changelog/db.changelog-master.xml
url=jdbc:postgresql://localhost:5432/nombre_bd
username=tu_usuario
password=tu_password
driver=org.postgresql.Driver
```

2. Configurar el archivo `jest.config.js`:

```
module.exports = {
    testEnvironment: 'node',
    transform: {
      '^.+\\.js$': 'babel-jest',
    },
  };
```

Si estás usando docker

3. Configurar el archivo `docker-compose.yml`

```
services:
  # Servicio para PostgreSQL
  db:
    image: postgres:latest
    container_name: postgres
    environment:
```

```
      POSTGRES_USER: TU_USUARIOS
      POSTGRES_PASSWORD: TU_PASSWORD
      POSTGRES_DB: TU_BD
    ports:
      - "5433:5432"
    volumes:
      - pgdata:/var/lib/postgresql/data
    networks:
      - app-network

  # Servicio para la API de Node.js
  api:
    build: .
    container_name: api
    ports:
    - "4000:4000"
    environment:
      DB_HOST: db
      DB_PORT: 5432
      DB_NAME: TU_BD
      DB_USER: TU_USUARIOS
      DB_PASSWORD: TU_PASSWORD
    depends_on:
      - db
    networks:
      - app-network

  # Servicio para Liquibase
  liquibase:
    image: liquibase/liquibase
    container_name: liquibase
    environment:
      LIQUIBASE_URL: jdbc:TU_USUARIOS://db:5432/TU_BD
      LIQUIBASE_USER: postgres
      LIQUIBASE_PASSWORD: 1234
    volumes:
```

```
      - ./db.changelog-master.xml:/liquibase/changelog/db.change
    depends_on:
      - db
    networks:
      - app-network

volumes:
  pgdata:

networks:
  app-network:
    driver: bridge
```

🐋 Dockerización

Este proyecto está configurado para ser ejecutado dentro de contenedores Docker utilizando `docker-compose`. Los pasos para dockerizarlo son los siguientes:

1. Construir la imagen de la aplicación:

   ```
   docker-compose build
   ```

2. Iniciar los contenedores:

   ```
   docker-compose up
   ```

   Esto levantará tanto la aplicación como PostgreSQL en contenedores.

3. Parar los contenedores:

   `docker-compose down`

4. Si prefieres usar Docker solo para la base de datos, puedes ejecutar:

   `docker-compose up db`

🚀 Uso

Iniciar el servidor:

```
npm run start
```

Este comando iniciará el servidor en modo de desarrollo. Puedes acceder a http://localhost:4000/graphql

🔬 Ejecutar pruebas unitarias

```
npm test
```

Este comando ejecutará las pruebas unitarias de los resolvers de GraphQL para asegurar que todo funcione correctamente.

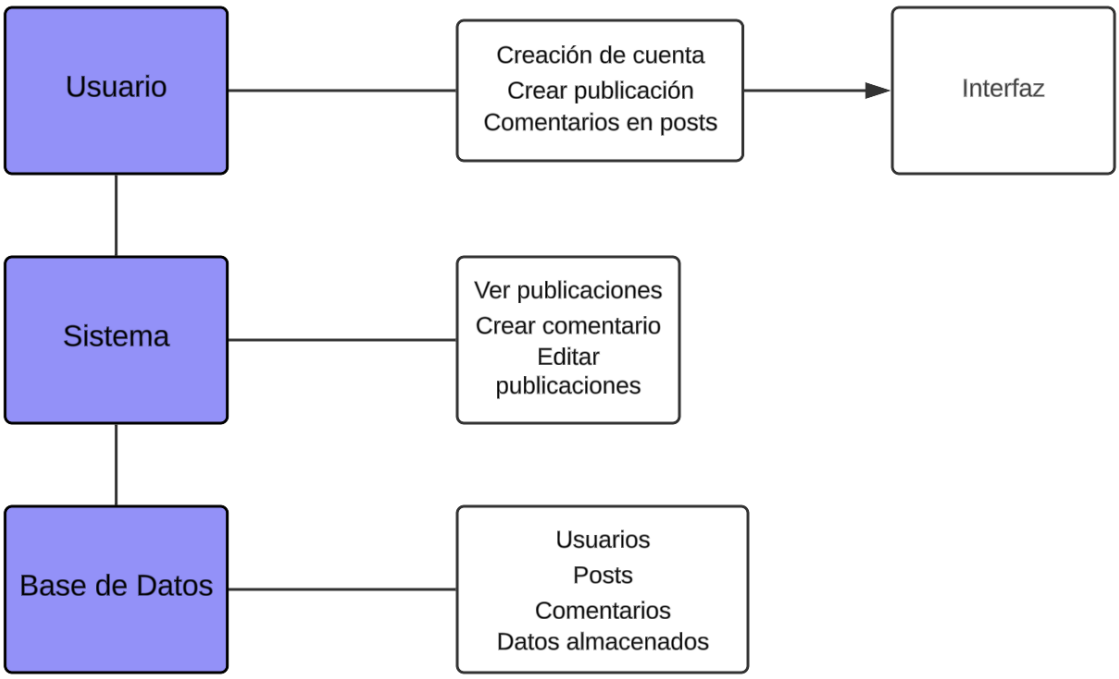
Endpoints GraphQL Principales

La API expone un único endpoint GraphQL en `/graphql` que maneja todas las operaciones:

- **Queries**
  - getAllUsers: Obtener lista de todos usuarios
  - getUserById(id): Obtener usuario por ID
  - getAllPosts: Obtener lista de posts
  - getPostById(id): Obtener post por ID
  - getCommentsByPostId(id): Obtener los comentarios por post
- **Mutations**
  - createUser: Crear nuevo usuario
  - updateUser: Actualizar usuario existente
  - deleteUser: Eliminar usuario existente
  - createPost: Crear nuevo post
  - updatePost: Actualizar post existente
  - deletePost: Eliminar post existente
  - createComment: Crear nuevo comentario

- updateComment: Actualizar comentario existente
- deleteComment: Eliminar comentario existe
- login: Para iniciar sesión y poder usar los recursos

Diagrama de Flujo de Datos (DFD)



En este diagrama:

- **Usuario**: Interactúa con la interfaz de usuario para crear, editar o comentar publicaciones.
- **Sistema**: Procesa las solicitudes del usuario, por ejemplo, verificando credenciales o buscando publicaciones.
- **Base de Datos**: Guarda las entidades como usuarios, publicaciones y comentarios

Ejemplo de consultas

1. **Obtener todos los usuarios**:

```
query {
  users {
    id
    name
    email
  }
}
```

2. **Obtener un usuario por ID**:

```
query {
  user(id: "1") {
    id
    name
    email
  }
}
```

3. **Obtener todas las publicaciones**:

```
query {
  posts {
    id
    title
    content
  }
}
```

4. **Obtener una publicación por ID**:

```
query {
  post(id: "1") {
    id
```

```
    title
    content
    user {
      name
    }
  }
}
```

5. **Obtener comentarios de una publicación**:

```
query {
  comments(postId: "1") {
    id
    text
    user {
      name
    }
  }
}
```

Mutaciones para Crear y Modificar Datos

1. **Crear un nuevo usuario**:

```
mutation {
  createUser(name: "Vanessa Ramirez", email: "vaness77@exa
mple.com", password: "password123") {
    id
    name
    email
  }
}
```

2. **Actualizar un usuario existente**:

```
mutation {
  updateUser(userId: "1", name: "Vanessa Ramirez", email:
"vaness77@example.com") {
    id
    name
    email
  }
}
```

3. **Eliminar un usuario**:

```
mutation {
  deleteUser(userId: "1") {
    id
    name
  }
}
```

4. **Crear una nueva publicación**:

```
mutation {
  createPost(userId: "1", title: "New Post", content: "Thi
s is a new post content") {
    id
    title
    content
  }
}
```

5. **Actualizar una publicación**:

```
mutation {
  updatePost(userId: "1", postId: "1", title: "Updated Tit
le", content: "Updated content") {
    id
```

```
    title
    content
  }
}
```

6. **Eliminar una publicación**:

```
mutation {
  deletePost(userId: "1", postId: "1") {
    id
    title
  }
}
```

7. **Crear un nuevo comentario**:

```
mutation {
  createComment(userId: "1", postId: "1", text: "This is a
comment") {
    id
    text
  }
}
```

8. **Actualizar un comentario**:

```
mutation {
  updateComment(commentId: "1", text: "Updated comment tex
t") {
    id
    text
  }
}
```

9. **Eliminar un comentario**:

```
mutation {
  deleteComment(commentId: "1", userId: "1", postId: "1")
{
    id
    text
  }
}
```

Autenticación

1. **Iniciar sesión**:

```
mutation {
  login(email: "vaness77example.com", password: "password1
23") {
    token
    user {
      id
      name
    }
  }
}
```

Manejo de Errores

La API utiliza códigos de estado HTTP estándar para indicar el resultado de las operaciones. A continuación, se describen los errores más comunes, su significado y cómo resolverlos.

1. Error 401: No Autorizado

Descripción: Este error indica que la solicitud no tiene un token de autenticación válido o que el usuario no está autorizado para acceder al recurso solicitado.

Causas Comunes:

- El token de autenticación no se ha proporcionado en el encabezado de la solicitud.
- El token ha expirado.
- El usuario no tiene los permisos necesarios.

Cómo Resolver:

- Asegúrate de que estás enviando un token válido en el encabezado `Authorization` con el formato `Bearer <token>`.
- Verifica que el token no haya expirado.
- Revisa los permisos del usuario para asegurarte de que tiene acceso al recurso.

Ejemplo de Respuesta:

```
{
  "errors": [
    {
      "message": "No Autorizado",
      "code": "401",
      "status": "Internal Server Error",
      "timestamp": "2025-01-18T17:42:19.136Z"
    }
  ],
  "data": {
    "user": null
  }
}
```

2. Error 404: No Encontrado

Descripción: Este error ocurre cuando el recurso solicitado no existe en la base de datos.

Causas Comunes:

- El ID proporcionado en la solicitud es incorrecto o no existe.
- El recurso ha sido eliminado.

Cómo Resolver:

- Verifica que el ID utilizado en la solicitud sea correcto.
- Asegúrate de que el recurso no ha sido eliminado o modificado.

Ejemplo de Respuesta:

```
{
  "errors": [
    {
      "message": "Usuario no encontrado",
      "code": "404",
      "status": "Not Found",
      "timestamp": "2025-01-18T17:42:57.056Z"
    }
  ],
  "data": {
    "user": null
  }
}
```

3. Error 422: Entidad No Procesable

Descripción: Este error indica que los datos enviados en la solicitud son inválidos o están incompletos.

Causas Comunes:

- Faltan campos obligatorios en la solicitud.
- Los datos no cumplen con las validaciones requeridas.

Cómo Resolver:

- Asegúrate de enviar todos los campos obligatorios.

- Verifica que los datos cumplen con las validaciones necesarias (por ejemplo, formato de email, longitud de la contraseña).

Ejemplo de Respuesta:

```
{
  "error": "Entidad No Procesable",
  "code": 422,
  "message": "Los datos proporcionados son inválidos."
}
```

4. Error 500: Error Interno del Servidor

Descripción: Este error ocurre cuando algo inesperado falla en el servidor.

Causas Comunes:

- Problemas con la base de datos.
- Errores no controlados en el servidor.

Cómo Resolver:

- Verifica los registros del servidor para identificar la causa del error.
- Asegúrate de manejar adecuadamente todas las excepciones en el código.

Ejemplo de Respuesta:

```
{
  "error": "Error Interno del Servidor",
  "code": 500,
  "message": "Ocurrió un error inesperado en el servidor."
}
```

Preguntas Frecuentes (FAQ)

> **¿Cómo autenticarme?**
> Utiliza el endpoint

> `login` para obtener un token JWT.

📄 Licencia

Este proyecto está bajo la licencia MIT.