**Registro LOGIN**

Contenido

[1. Herramientas: 1](#_Toc190413176)

[2. Diccionario de Datos 1](#_Toc190413177)

[1.1. Tabla: Usuarios 1](#_Toc190413178)

[1.2. Tabla: HistorialConsultas 2](#_Toc190413179)

[1.2.1. Tabla: Planetas 2](#_Toc190413180)

[1.2.2. Tabla: Especies 2](#_Toc190413181)

[1.5. Tabla: Personajes 2](#_Toc190413182)

[1.6. Tabla: Peliculas 3](#_Toc190413183)

[1.7. Tabla: Vehiculos 3](#_Toc190413184)

[1.8. Tabla: Personaje\_Pelicula 3](#_Toc190413185)

[1.9. Tabla: Personaje\_Vehiculo 3](#_Toc190413186)

[1.10. Tabla: Personaje Especie 3](#_Toc190413187)

[1.11. Tabla: Pelicula\_Planeta 4](#_Toc190413188)

[Estructura del proyecto 4](#_Toc190413189)

[Ejecución 5](#_Toc190413190)

[1. Clona o descarga el repositorio 5](#_Toc190413191)

[2. Instala las dependencias 5](#_Toc190413192)

[3. Configura la base de datos y Redis 5](#_Toc190413193)

[Opción A: Docker Compose (recomendado) 5](#_Toc190413194)

[Opción B: Instalar PostgreSQL y Redis localmente 5](#_Toc190413195)

[**3.** **Instala node-fetch** 5](#_Toc190413196)

[4. Verifica que el servidor esté corriendo 6](#_Toc190413197)

[Pruebas 6](#_Toc190413198)

[1. POST /api/register → Registro de usuarios. 6](#_Toc190413199)

[Envía la solicitud (Send**)** 8](#_Toc190413200)

[En caso de existir el registro, validar su existencia: 9](#_Toc190413201)

[2. POST /api/login → (TOKEN) Inicio de sesión (devuelve un token de acceso). 10](#_Toc190413202)

[Envía la solicitud (TOKEN) 11](#_Toc190413203)

[Usar el token en otros endpoints 11](#_Toc190413204)

[3. POST /api/logout → Cierre de sesión. 11](#_Toc190413205)

[Envía la solicitud 12](#_Toc190413206)

[¿Qué pasa después del logout? 13](#_Toc190413207)

[4. Obtener y almacenar una película https://swapi.dev/api/films/{id}/: 13](#_Toc190413208)

[ Consulta https://swapi.dev/api/films/{id}/ y almacena la información. 13](#_Toc190413209)

[ Relaciona el personaje con su planeta, películas, vehículos y especies. 14](#_Toc190413210)

[ Qué pasa si Token Incorrecto: 15](#_Toc190413211)

[ Validaciones 15](#_Toc190413212)

[5. Obtener y almacenar una Persona https://swapi.dev/api/people/{id}/: 16](#_Toc190413213)

[ Consulta https://swapi.dev/api/people/{id}/ y almacena la información. 16](#_Toc190413214)

[ Relaciona el personaje con su planeta, películas, vehículos y especies. 17](#_Toc190413215)

[ Qué pasa si Token Incorrecto: 18](#_Toc190413216)

[ Validaciones 18](#_Toc190413217)

[6. Obtener y almacenar una Planetas https://swapi.dev/api/planets/{id}/: 18](#_Toc190413218)

[ Consulta https://swapi.dev/api/planets/{id}/ y almacena la información. 18](#_Toc190413219)

[ Qué pasa si Token Incorrecto: 20](#_Toc190413220)

[ Validaciones 20](#_Toc190413221)

[7. Historial de consultas del usuario 21](#_Toc190413222)

[Base de datos y relaciones en PostgreSQL 21](#_Toc190413223)

[Optimización con Caché: 23](#_Toc190413224)

[Dockerización 23](#_Toc190413225)

[Entrega y puntos a considerar: 23](#_Toc190413226)

# Herramientas:

Para llevar a cabo este proyecto (desarrollar la API, probarla y manejar la base de datos y caché), se necesita las siguientes **herramientas**:

1. **Laravel Framework**: Desarrollo de aplicaciones web robustas y escalables utilizando Laravel, aprovechando características como Eloquent ORM, migraciones, seeders, y sistemas de autenticación integrados.
2. **APIs RESTful y Blade**: Creación de APIs RESTful para integración con frontend y aplicaciones móviles, junto con la construcción de vistas dinámicas utilizando el motor de plantillas Blade.
3. **Integración con servicios externos**: Conexión con APIs de terceros (pagos, envío de correos, etc.) y uso de herramientas como Redis para caché y colas de tareas asíncronas.
4. **Docker y entornos locales**: Configuración de entornos de desarrollo con Docker y Docker Compose para levantar servicios como PostgreSQL y Redis de manera rápida y consistente.
5. **Herramientas de desarrollo**: Uso de Postman para pruebas de APIs, Git para control de versiones, y editores de código como VS Code para una experiencia de desarrollo eficiente.
6. **Optimización y seguridad**: Implementación de buenas prácticas de seguridad (CSRF, XSS, SQL Injection) y optimización de consultas con Eloquent y caché.
7. **Trabajo en equipo ágil**: Colaboración en equipos Scrum, participando en la planificación de sprints, revisiones de código y entregas iterativas.

### Requisitos técnicos mencionados integrados:

* **Node.js**: Para ejecutar herramientas de frontend o scripts adicionales

Node.js estará en <http://localhost:3000>

* **Docker**: Para dockerizar la aplicación y gestionar servicios como PostgreSQL y Redis.
* **PostgreSQL**: Base de datos principal para almacenamiento de datos.
* **Redis**: Para manejo de caché y colas de tareas.
* **Postman**: Para pruebas de endpoints de la API.
* **Git**: Para control de versiones y colaboración en equipo.

### Ejemplo de uso en un proyecto Laravel:

1. **Configuración del entorno**:
   * Usar Docker Compose para levantar servicios de PostgreSQL y Redis.
   * Configurar. env para conectar Laravel con estos servicios.

Laravel estará en <http://localhost:8000>

1. **Desarrollo de APIs**:
   * Crear endpoints RESTful para gestionar recursos (por ejemplo, /api/people/:id).
   * Usar Eloquent para interactuar con la base de datos PostgreSQL.
2. **Optimización**:
   * Implementar caché con Redis para mejorar el rendimiento de consultas frecuentes.
   * Usar colas de tareas para procesamiento asíncrono.
3. **Pruebas**:
   * Probar endpoints con Postman.
   * Escribir pruebas unitarias y de integración con PHPUnit.
4. **Despliegue**:
   * Dockerizar la aplicación para facilitar su despliegue en producción.
   * Configurar CI/CD con herramientas como GitHub Actions o GitLab CI.

# **Diccionario de Datos**

### **Tabla:** Usuarios

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **usuario\_id** | SERIAL | Identificador único del usuario. | PRIMARY KEY |
| **nombre\_usuario** | VARCHAR(100) | Nombre de usuario, no se repite. | NOT NULL, UNIQUE |
| **contraseña\_hash** | VARCHAR(255) | Almacena el hash de la contraseña del usuario. | NOT NULL |
| **email** | VARCHAR(100) | Correo electrónico del usuario, no se repite. | NOT NULL, UNIQUE |

### **Tabla:** HistorialConsultas

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **id** | SERIAL | Identificador único del registro de historial. | PRIMARY KEY |
| **usuario\_id** | INT | Referencia al ID de la tabla Usuarios (usuario que realizó la consulta). | NOT NULL (sin FK explícito, pero semánticamente relacionado) |
| **endpoint** | VARCHAR(255) | Ruta o endpoint consultado (ej. /api/people/1). | NOT NULL |
| **params** | JSON | Parámetros o detalles adicionales de la consulta. | Opcional |
| **fecha** | TIMESTAMP | Fecha y hora en que se registró la consulta. | DEFAULT CURRENT\_TIMESTAMP |

### **Tabla:** Planetas

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **planeta\_id** | SERIAL | Identificador único del planeta. | PRIMARY KEY |
| **nombre** | VARCHAR(100) | Nombre del planeta (ej. "Tatooine"). | NOT NULL |
| **clima** | VARCHAR(100) | Clima principal (ej. "árido", "tropical"). | (Opcional) |
| **terreno** | VARCHAR(100) | Tipo de terreno (ej. "desértico", "bosque"). | (Opcional) |

### **Tabla:** Especies

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **especie\_id** | SERIAL | Identificador único de la especie. | PRIMARY KEY |
| **nombre** | VARCHAR(100) | Nombre de la especie (ej. "Human", "Wookiee"). | NOT NULL |
| **clasificacion** | VARCHAR(100) | Clasificación biológica (ej. "mammal"). | (Opcional) |
| **designacion** | VARCHAR(100) | Designación (ej. "sentient"). | (Opcional) |
| **altura\_promedio** | DECIMAL(5,2) | Altura promedio en cm (ej. 180.00). | (Opcional) |

### 1.5. **Tabla:** Personajes

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **personaje\_id** | SERIAL | Identificador único del personaje. | PRIMARY KEY |
| **nombre** | VARCHAR(100) | Nombre del personaje (ej. "Luke Skywalker"). | NOT NULL |
| **altura** | DECIMAL(5,2) | Altura del personaje en cm (ej. 172.00). | (Opcional) |
| **peso** | DECIMAL(5,2) | Peso del personaje en kg (ej. 77.00). | (Opcional) |
| **planeta\_id** | INT | Referencia al ID del planeta de origen (tabla Planetas). | FOREIGN KEY (Planetas.planeta\_id) |

### 1.6. **Tabla:** Peliculas

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **pelicula\_id** | SERIAL | Identificador único de la película. | PRIMARY KEY |
| **titulo** | VARCHAR(100) | Título de la película (ej. "A New Hope"). | NOT NULL |
| **director** | VARCHAR(100) | Nombre del director (ej. "George Lucas"). | (Opcional) |
| **fecha\_estreno** | DATE | Fecha de estreno (ej. "1977-05-25"). | (Opcional) |

### 1.7. **Tabla:** Vehiculos

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **vehiculo\_id** | SERIAL | Identificador único del vehículo. | PRIMARY KEY |
| **nombre** | VARCHAR(100) | Nombre del vehículo (ej. "Snowspeeder"). | NOT NULL |
| **modelo** | VARCHAR(100) | Modelo o versión del vehículo. | (Opcional) |
| **fabricante** | VARCHAR(100) | Fabricante o marca (ej. "Incom Corporation"). | (Opcional) |
| **costo** | DECIMAL(15,2) | Costo en créditos galácticos (ej. 15000.00). | (Opcional) |

### 1.8. **Tabla:** Personaje\_Pelicula

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **personaje\_id** | INT | Referencia al ID del personaje (tabla Personajes). | FOREIGN KEY (Personajes.personaje\_id), PK compuesta |
| **pelicula\_id** | INT | Referencia al ID de la película (tabla Peliculas). | FOREIGN KEY (Peliculas.pelicula\_id), PK compuesta |

Relación **muchos a muchos** entre Personajes y Peliculas.

### 1.9. **Tabla:** Personaje\_Vehiculo

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **personaje\_id** | INT | Referencia al ID del personaje (tabla Personajes). | FOREIGN KEY (Personajes.personaje\_id), PK compuesta |
| **vehiculo\_id** | INT | Referencia al ID del vehículo (tabla Vehiculos). | FOREIGN KEY (Vehiculos.vehiculo\_id), PK compuesta |

Relación **muchos a muchos** entre Personajes y Vehiculos.

### 1.10. **Tabla:** Personaje Especie

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **personaje\_id** | INT | Referencia al ID del personaje (tabla Personajes). | FOREIGN KEY (Personajes.personaje\_id), PK compuesta |
| **especie\_id** | INT | Referencia al ID de la especie (tabla Especies). | FOREIGN KEY (Especies.especie\_id), PK compuesta |

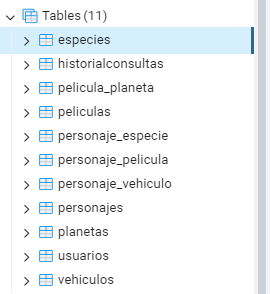
Relación **muchos a muchos** entre Personajes y Especies.

### 1.11. **Tabla:** Pelicula\_Planeta

|  |  |  |  |
| --- | --- | --- | --- |
| **Campo** | **Tipo** | **Descripción** | **Restricciones** |
| **pelicula\_id** | INT | Referencia al ID de la película (tabla Peliculas). | FOREIGN KEY (Peliculas.pelicula\_id), PK compuesta |
| **planeta\_id** | INT | Referencia al ID del planeta (tabla Planetas). | FOREIGN KEY (Planetas.planeta\_id), PK compuesta |

Relación **muchos a muchos** entre Películas y Planetas.

Captura:



# Estructura del proyecto

1. **src/**: Contiene todo el código fuente de la aplicación.
   * **controllers/**: Lógica de negocio.
     + authController.js: Maneja el registro, login y logout de usuarios.
     + swapiController.js: Lógica para consumir la API de Star Wars (SWAPI).
   * **middleware/**: Middlewares personalizados.
     + authMiddleware.js: Valida el token JWT para proteger rutas.
   * **models/**: Modelos de la base de datos.
     + User.js: Define el modelo de usuario y consultas relacionadas.
   * **routes/**: Definición de rutas.
     + authRoutes.js: Rutas para autenticación (registro, login, logout).
     + swapiRoutes.js: Rutas para consumir SWAPI.
   * **utils/**: Utilidades y configuraciones.
     + db.js: Conexión a PostgreSQL.
     + redisClient.js: Conexión a Redis.
2. **Dockerfile**: Archivo para construir la imagen Docker de la aplicación.
3. **docker-compose.yml**: Archivo para orquestar los servicios (app, PostgreSQL, Redis).
4. **index.js**: Punto de entrada de la aplicación.
5. **package.json**: Dependencias y scripts del proyecto.

# Ejecución

## Clona o descarga el repositorio

1. **Clonar con Git:**

git clone <URL\_DEL\_REPOSITORIO>

cd starwars-api

1. **O descargar ZIP:**
   * Descarga el ZIP y descomprímelo.
   * Asegúrate de que la carpeta se llame, por ejemplo, starwars-api/.

## Instala las dependencias

Dentro de la carpeta del proyecto (starwars-api/):

npm install

Esto instalará todas las librerías necesarias (Express, JWT, bcryptjs, pg, redis, etc.)

## Configura la base de datos y Redis

### Opción A: Docker Compose (recomendado)

1. Asegúrate de tener instalado [Docker](https://www.docker.com/) y Docker Compose.
2. En la carpeta raíz del proyecto, ejecuta:

docker-compose up –build

1. Esto levantará:
   * **PostgreSQL** con la base de datos login.
   * **Redis** para la caché.
   * **La aplicación** Node.js en el puerto 3000.

### Opción B: Instalar PostgreSQL y Redis localmente

1. **PostgreSQL**:
   * Instala PostgreSQL en la máquina (o usa un servidor remoto).
   * Crea la base de datos login.
   * Ejecuta los scripts CREATE TABLE para crear las tablas.
2. **Redis**:
   * Instala Redis localmente (o usa un servidor remoto).

## **Instala node-fetch**

En la raíz de tu proyecto (donde está tu package.json), ejecuta:

npm install node-fetch

## **Verificar si Redis está instalado**

En la terminal de PowerShell o CMD, ejecuta:

* redis-server --version

Si no está instalado, descárgalo desde https://redis.io/download o usa choco en Windows:

* choco install redis-64

**Iniciar Redis**  
Si ya está instalado, inicia el servicio:

* redis-server

O, si está configurado como un servicio en Windows:

* net start Redis

## Instalar Sanctum:

composer require laravel/sanctum

1. **Variables de entorno** (.env o configuradas en tu sistema):

DB\_USER=postgres

DB\_PASSWORD=postgres

DB\_HOST=localhost

DB\_PORT=5432

DB\_NAME=login

REDIS\_HOST=127.0.0.1

REDIS\_PORT=6379

JWT\_SECRET=Juan@2025.

JWT\_EXPIRES\_IN=1h

PORT=3000

1. Inicia la aplicación con:

node index.js



## Verifica que el servidor esté corriendo

Si todo salió bien, verás en la consola algo como:

Servidor corriendo en http://localhost:3000

Conectado a Redis

Significa que la aplicación está lista para recibir peticiones en <http://localhost:3000>.



# Pruebas

## POST /api/register → Registro de usuarios.

1) Autenticación segura

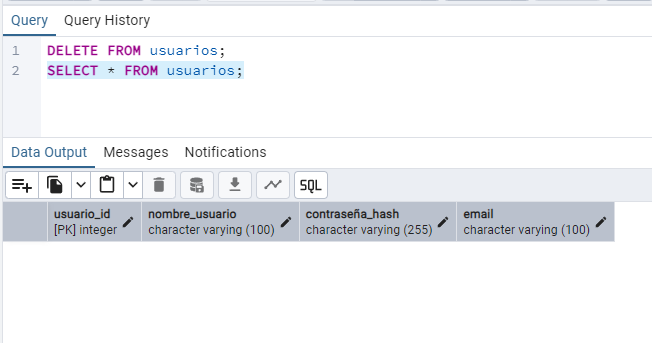
Implementa un sistema de autenticación para que los usuarios puedan registrarse e iniciar

sesión.

● Implementa una solución para autenticación con tokens.

● Protege los endpoints privados con middleware de autenticación.

Tabla Usuarios vacía:



Registrar Usuario:

Para **ejecutar** el endpoint http://localhost:3000/api/register y **registrar un usuario**, seguir los pasos:

1. **Asegúrar de que el servidor esté en marcha**
   * Si se usa Node.js localmente:

node index.js

Verifica que aparezca en la consola:

Servidor corriendo en

* + Si se usa Docker Compose:

docker-compose up –build

Espera a que se inicien los contenedores y verifica que no haya errores.

1. **Abrir Postman (o cualquier cliente HTTP)**
   * Crea una nueva solicitud con:
     + **Método:** POST
     + **URL:** <http://localhost:3000/api/register>
2. **Configurar los Headers**
   * Asegúrate de incluir:

Content-Type: application/json

1. **Body (raw, JSON)**
   * En la pestaña **Body** de Postman, elige **raw** y **JSON** como formato, luego coloca algo como:

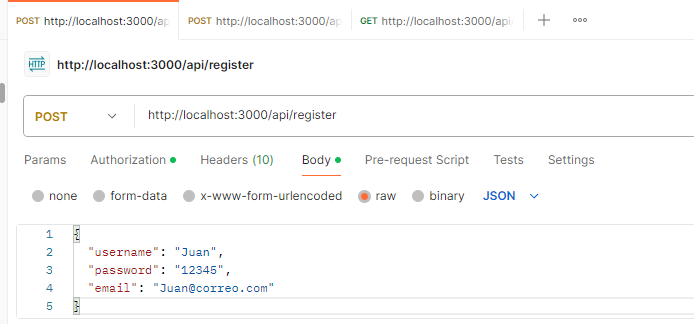
{

  "username": "Juan",

  "password": "12345",

  "email": "Juan@correo.com"

}



* + Ajusta los valores según necesites (username, password, email).

### Envía la solicitud (Send**)**

* + Si todo está correcto, el servidor responderá con un JSON similar a:

{

    "message": "Usuario registrado",

    "user": {

        "usuario\_id": 11,

        "nombre\_usuario": "Juan",

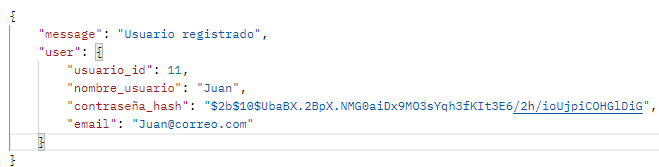
        "contraseña\_hash": "$2b$10$UbaBX.2BpX.NMG0aiDx9MO3sYqh3fKIt3E6/2h/ioUjpiCOHGlDiG",

        "email": "Juan@correo.com"

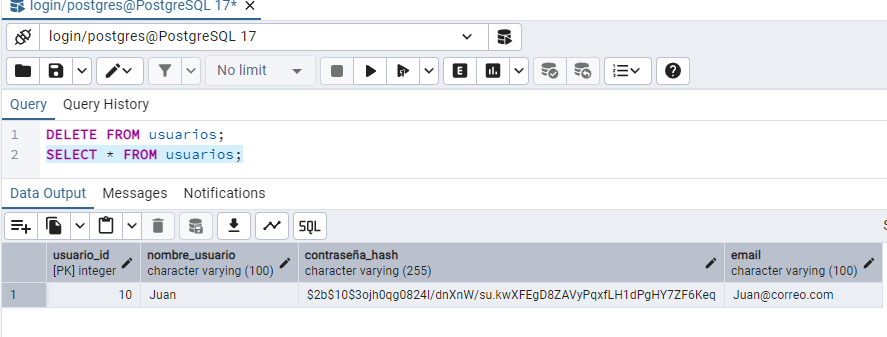
    }

}

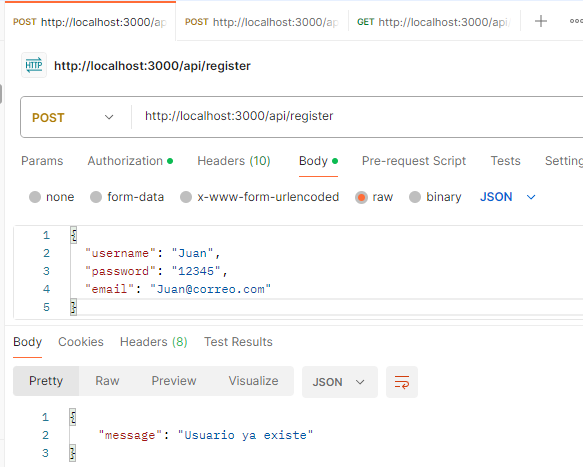
* + Esto confirma que el usuario se creó en la base de datos.



1. **Verifica en la base de datos** (opcional)
   * Si se tiene acceso a PostgreSQL (por ejemplo, con pgAdmin o psql), revisar la tabla Usuarios. Deberías ver una fila con el username, el email y la contraseña hasheada.



### En caso de existir el registro, validar su existencia:



## POST /api/login → (TOKEN) Inicio de sesión (devuelve un token de acceso).

1. **Node.js localmente**

node index.js

Revisa la consola para asegurarte de que se esté ejecutando en

http://localhost:3000.

1. **Docker Compose** (si se usa)

docker-compose up --build

Espera a que se inicien los contenedores (app, PostgreSQL, Redis) y confirma que no haya errores.

1. Configura la solicitud en Postman (o cualquier cliente HTTP)

**Método:** POST

**URL:** http://localhost:3000/api/login

1. Headers

* **Content-Type:** application/json

1. Body (raw, JSON)

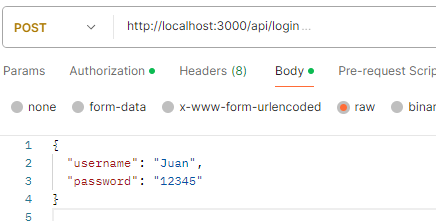
En la pestaña **Body** de Postman, selecciona raw y el tipo JSON. Luego, coloca algo como:

{

  "username": "Juan",

  "password": "12345"

}



Ajusta los valores de username y password de acuerdo al usuario se creó al registrarlo (por ejemplo, con POST /api/register).

### Envía la solicitud (TOKEN)

Haz clic en **Send** (o su equivalente en tu cliente HTTP).

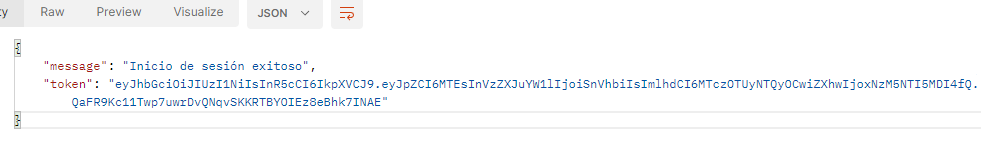
**Respuesta esperada**:

{

"message": "Inicio de sesión exitoso",

"token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTEsInVzZXJuYW1lIjoiSnVhbiIsImlhdCI6MTczOTUyNTQyOCwiZXhwIjoxNzM5NTI5MDI4fQ.QaFR9Kc11Twp7uwrDvQNqvSKKRTBYOIEz8eBhk7INAE"

}



Donde token es el JWT que utilizarás en los endpoints protegidos (por ejemplo, /api/people/:id, /api/history, etc.).

### Usar el token en otros endpoints

Una vez se obtenga el token, copia su valor (sin incluir la palabra "Bearer"). Luego, para endpoints protegidos, configura el header:

Authorization: Bearer <tu\_token\_jwt>

Ejemplo en Postman:

1. **Headers**
   * **Key:** Authorization
   * **Value:** Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
2. **Envía** la solicitud a tu endpoint protegido (GET /api/people/1, etc.).

## POST /api/logout → Cierre de sesión.

1. Asegúrate de tener el servidor en marcha

1. **Node.js localmente**

node index.js

**Docker Compose**

docker-compose up --build

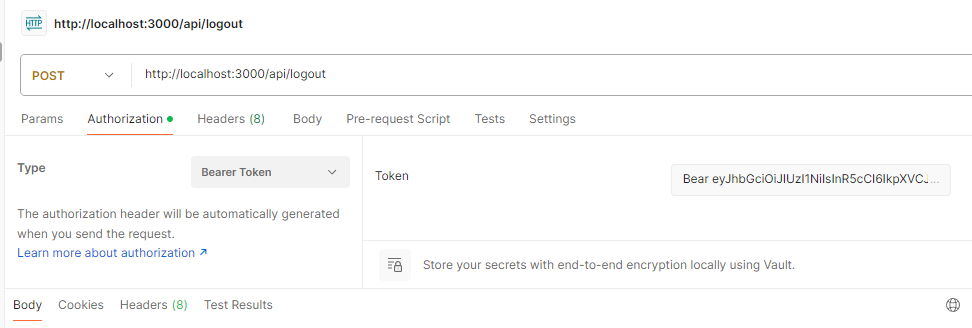
Confirma que el servidor está escuchando en http://localhost:3000.

2. Configura la solicitud en Postman

1. **Método:** POST
2. **URL:** http://localhost:3000/api/logout

Headers

* **Authorization:** Bearer <tu\_token\_jwt>
  + Donde <tu\_token\_jwt> es el token que se obtuvo tras hacer login (POST /api/login).



**Nota:** Aunque muchos sistemas JWT no implementan un “logout” real (porque el token es stateless), tu endpoint /api/logout podría, por ejemplo, eliminar el token de una lista en el servidor o simplemente responder con un mensaje de éxito. En algunos proyectos, el logout solo se maneja en el cliente, invalidando el token localmente.

Body

Normalmente, para un logout con JWT, **no** se envía ningún body (o se envía vacío). Sin embargo, si tu implementación requiere algo adicional, agrégalo según tu lógica. Generalmente, un body no es necesario.

### Envía la solicitud

Haz clic en **Send**. Si tu endpoint está implementado como, por ejemplo:

const logout = (req, res) => {

res.json({ message: 'Cierre de sesión exitoso' });

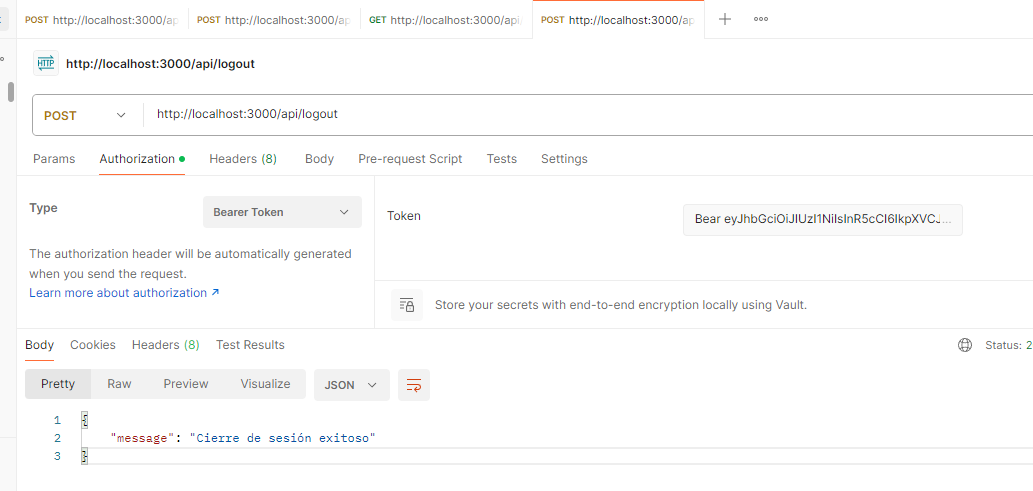
};

Entonces se recibirá la respuesta como:

{

    "message": "Cierre de sesión exitoso"

}



### ¿Qué pasa después del logout?

* **En JWT estándar**, el token sigue siendo técnicamente válido hasta que expire, a menos que tengas un sistema de revocación de tokens en el servidor (por ejemplo, guardando tokens en una blacklist).
* **En el cliente**, normalmente se elimina el token almacenado (por ejemplo, en localStorage) y así el usuario “cierra sesión”.

## Obtener y almacenar una película [https://swapi.dev/api/films/{id}/](https://swapi.dev/api/films/%7bid%7d/):

### Consulta [https://swapi.dev/api/films/{id}/](https://swapi.dev/api/films/%7bid%7d/) y almacena la información.

1. **Levantar el servidor** con:

npm start

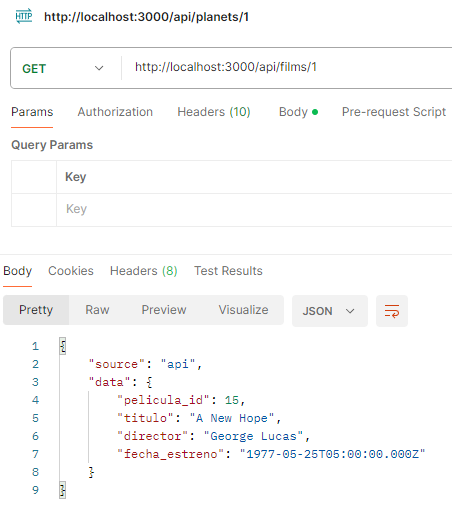
o de usar **Docker**, asegúrate de que está corriendo con:

docker-compose up --build

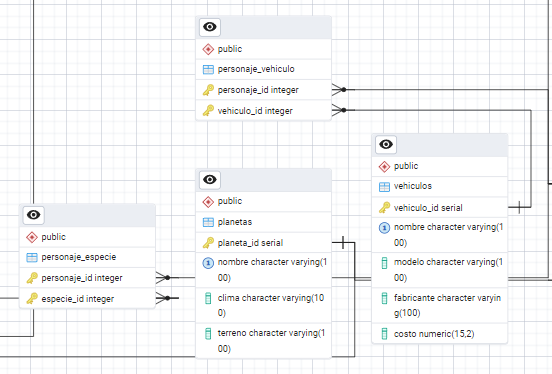
1. **Abrir Postman** y realizar una solicitud **GET** a:

http://localhost:3000/api/films/1

(Cambia 1 por otro ID para probar con diferentes personajes).



### Relaciona el personaje con su planeta, películas, vehículos y especies.



### Qué pasa si Token Incorrecto:

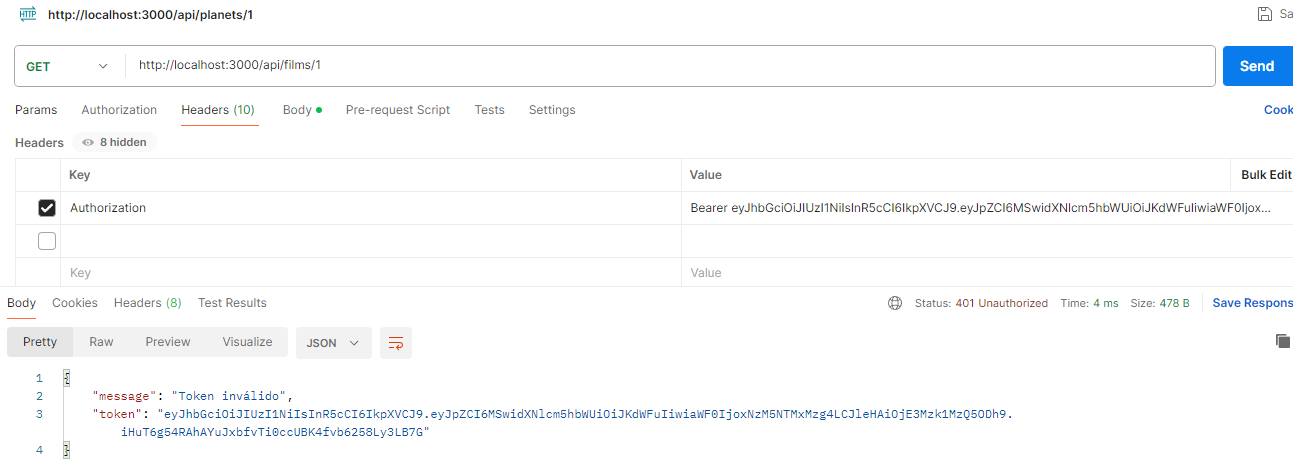
Respuesta:

{

    "message": "Token inválido",

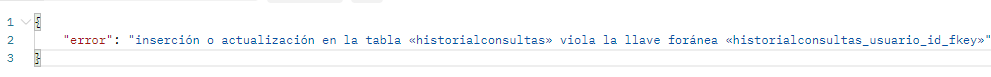
    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwidXNlcm5hbWUiOiJKdWFuIiwiaWF0IjoxNzM5NTMxMzg4LCJleHAiOjE3Mzk1MzQ5ODh9.iHuT6g54RAhAYuJxbfvTi0ccUBK4fvb6258Ly3LB7G"

}

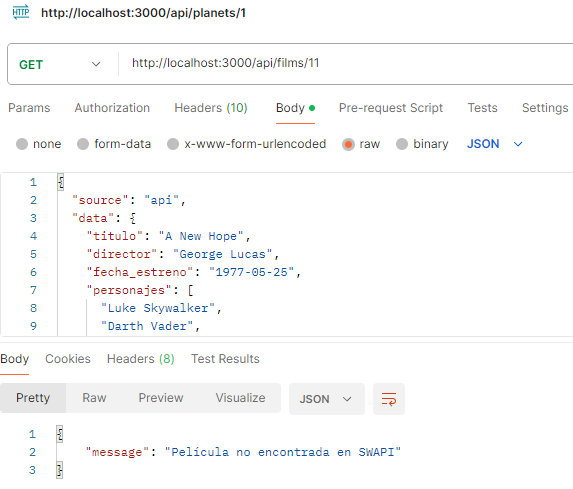


### Validaciones

Error por historial de consultas:



Película no existe en SWAPI: "message": "Película no encontrada en SWAPI"



## Obtener y almacenar una Persona [https://swapi.dev/api/people/{id}/](https://swapi.dev/api/people/%7bid%7d/):

### Consulta [https://swapi.dev/api/people/{id}/](https://swapi.dev/api/people/%7bid%7d/) y almacena la información.

1. **Levantar el servidor** con:

npm start

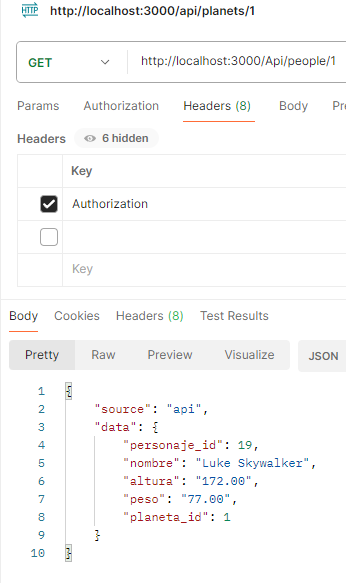
o de usar **Docker**, asegúrate de que está corriendo con:

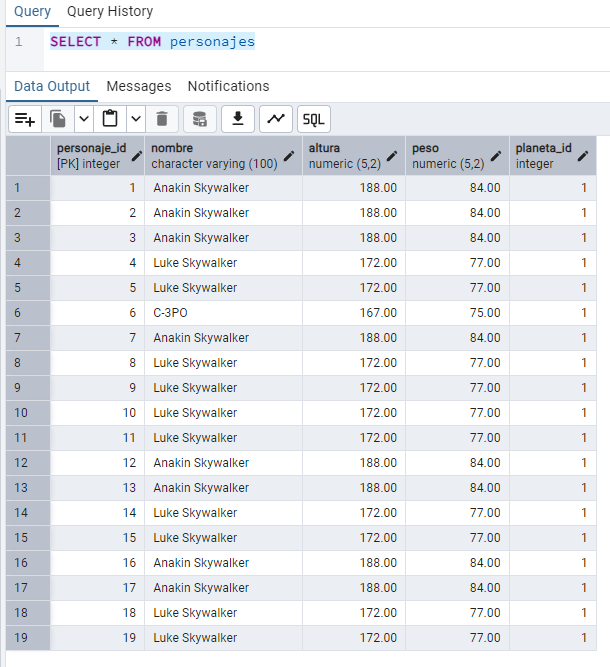
docker-compose up --build

1. **Abrir Postman** y realizar una solicitud **GET** a:

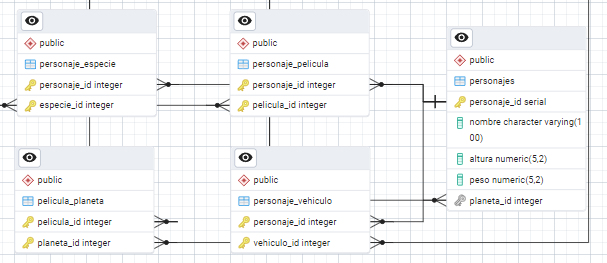
http://localhost:3000/Api/people/1

(Cambia 1 por otro ID para probar con diferentes personajes).





### Relaciona el personaje con su planeta, películas, vehículos y especies.

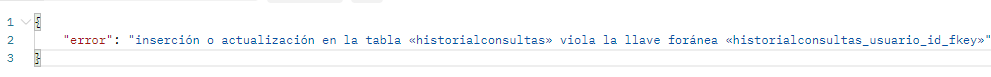


### Qué pasa si Token Incorrecto:

Respuesta:

### Validaciones

Error por historial de consultas:



Película no existe en SWAPI: "message": "Personaje no encontrada en SWAPI"

## Obtener y almacenar una Planetas [https://swapi.dev/api/planets/{id}/](https://swapi.dev/api/planets/%7bid%7d/):

### Consulta [https://swapi.dev/api/planets/{id}/](https://swapi.dev/api/planets/%7bid%7d/) y almacena la información.

1. **Levantar el servidor** con:

npm start

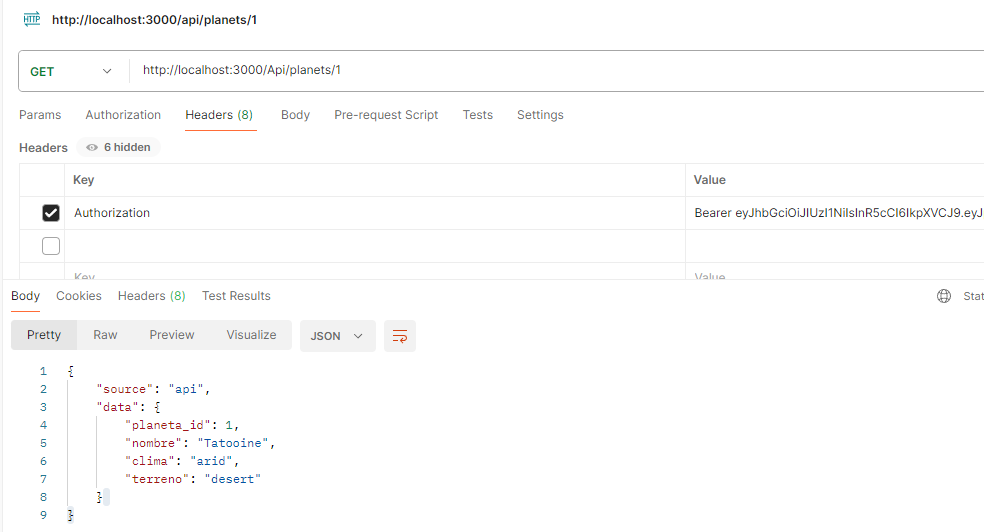
o de usar **Docker**, asegúrate de que está corriendo con:

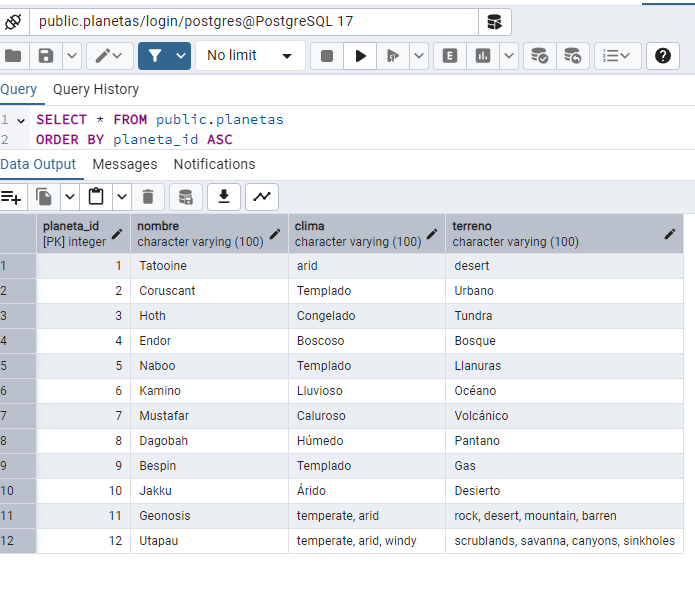
docker-compose up --build

1. **Abrir Postman** y realizar una solicitud **GET** a:

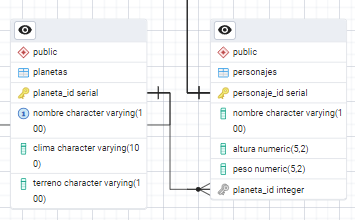
http://localhost:3000/Api/planets/1

(Cambia 1 por otro ID para probar con diferentes personajes).





● Relaciona el planeta con los personajes que lo habitan.



### Qué pasa si Token Incorrecto:

Respuesta:

{

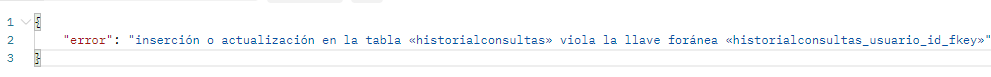
    "message": "Token inválido",

    "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MSwidXNlcm5hbWUiOiJKdWFuIiwiaWF0IjoxNzM5NTMxMzg4LCJleHAiOjE3Mzk1MzQ5ODh9.iHuT6g54RAhAYuJxbfvTi0ccUBK4fvb6258Ly3LB7G"

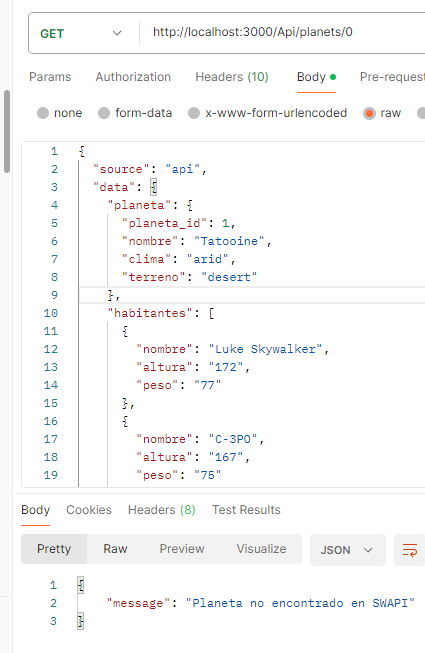
}

### Validaciones

Error por historial de consultas:

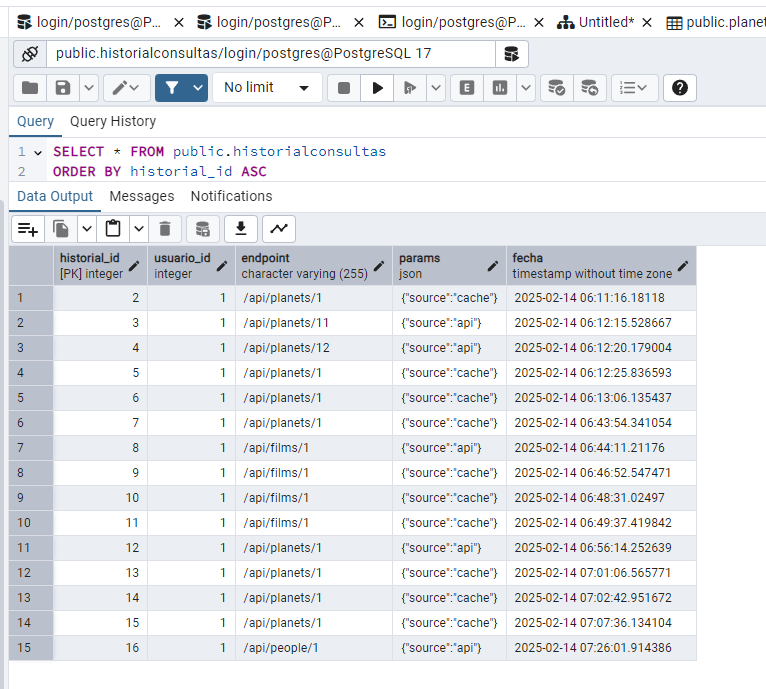


Película no existe en SWAPI: "message": "Planeta no encontrada en SWAPI"



## Historial de consultas del usuario

● Se debe tener el registro de todas las consultas que ha realizado el usuario autenticado.



# Base de datos y relaciones en PostgreSQL

Se deben crear las tablas necesarias con relaciones entre sí, puedes usar la cantidad de

relaciones que creas conveniente para facilitar a la hora de las consultas, como, por ejemplo:

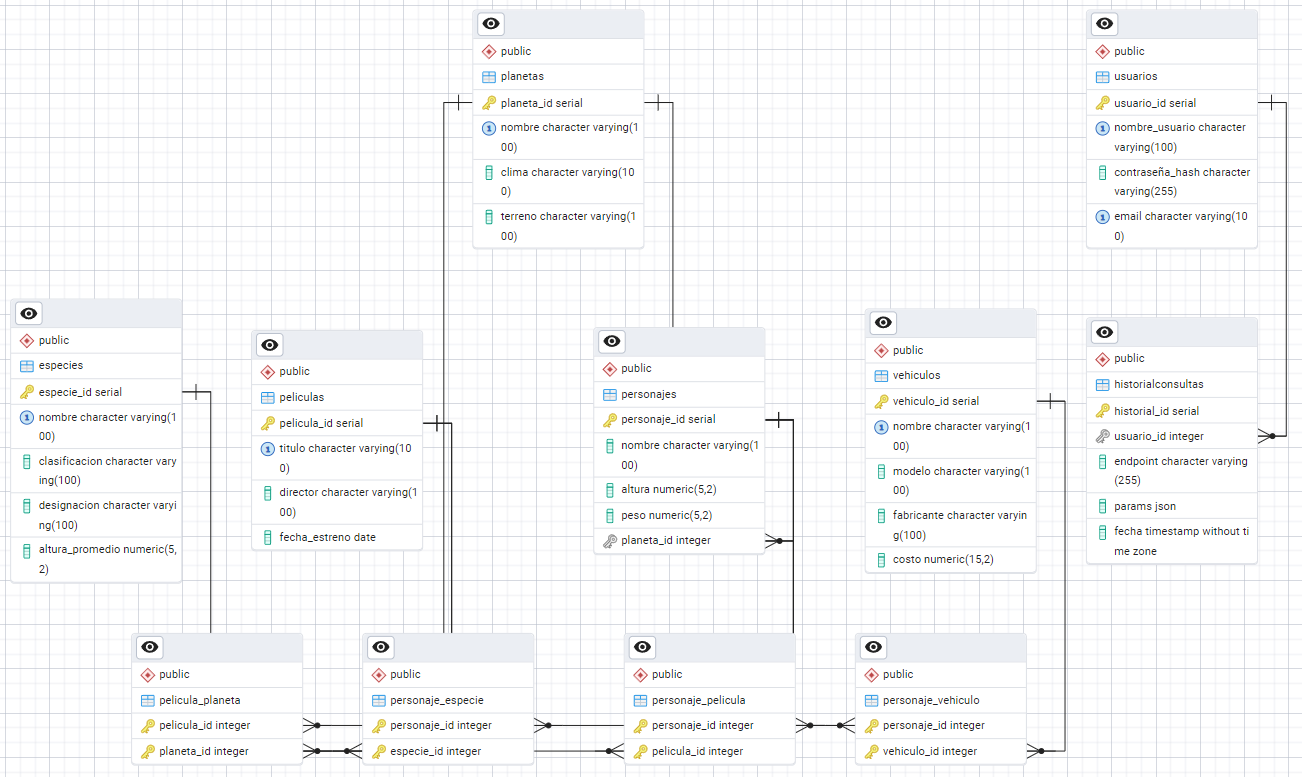
● Un personaje pertenece a un planeta.

● Un personaje puede estar en varias películas.

● Un personaje puede usar varios vehículos.

● Un personaje puede pertenecer a varias especies.

● Una película puede incluir varios personajes y varios planetas.



# Optimización con Caché:

● Si un usuario solicita un personaje/película u otra información ya almacenada, devolver desde caché en lugar de consultar SWAPI.

* **swapiController.js** revisa Redis antes de llamar a SWAPI y almacena la respuesta en caché.

● Puedes usar Redis o alguna herramienta que almacene respuestas para evitar

consultas repetitivas a la API externa.

* **redisClient.js** maneja la conexión a Redis.
* **docker-compose.yml** levanta Redis como servicio para trabajar con la app.

# Dockerización

● Crea un Dockerfile para tu aplicación.

* **Dockerfile** → Define cómo se construye la imagen de la aplicación.

● Utiliza docker-compose para definir y ejecutar los servicios de tu aplicación y la base de datos.

* **docker-compose.yml** → Orquesta los servicios (App, PostgreSQL, Redis).
* **Comandos esenciales** → docker-compose up -d, docker ps, docker-compose down.

# Entrega y puntos a considerar:

Instrucciones de Despliegue:

Proporciona un README.md con instrucciones claras sobre cómo ejecutar y probar tu

aplicación.

* **README.md**→ Guía de instalación.