**INFORME PROYECTO COMPUTACION EN INTERNET III**

**Miembros:**

* Santiago Jose Belalcazar
* Manuel Herrera
* Victor Manuel Garzon

**1. Descripción General**

La API de la aplicación ofrece funcionalidades de autenticación, autorización y CRUD (Crear, Leer, Actualizar, Eliminar) para gestionar entidades como **usuarios**, **contenido** (películas, series, animes), **comentarios**, **listas** y **reacciones**. Además, la aplicación utiliza una base de datos **PostgreSQL** para la persistencia y **TypeORM** como el ORM para interactuar con la base de datos.

**2. Endpoints y Funcionalidades**

Cada entidad de la aplicación (Usuarios, Contenido, Comentarios, Reacciones y Listas) tiene sus propios endpoints para realizar operaciones CRUD. A continuación, se describe cada uno de los endpoints implementados, incluyendo sus parámetros y posibles respuestas.

**Usuarios**

* GET /users: Obtiene una lista de todos los usuarios.
  + Respuesta: Array de usuarios.

* GET /users/:id: Obtiene la información de un usuario por su ID.
  + Parámetro: id (UUID del usuario)
  + Respuesta: Objeto usuario o error 404 si no se encuentra.

* PUT /users/:id: Actualiza la información de un usuario.
  + Parámetros: `id` (UUID del usuario), JSON con los datos a actualizar.
  + Respuesta: Objeto usuario actualizado o error 404.
* DELETE /users/:id: Elimina un usuario.
  + Parámetro: `id` (UUID del usuario)
  + Respuesta: Confirmación de eliminación o error 404.

**Autenticación y Autorización**

* POST /auth/register: Registra un nuevo usuario.
  + Parámetros: JSON con `fullName`, `email`, y `password`.
  + Respuesta: Objeto usuario creado.
* POST /auth/login: Autentica un usuario y devuelve un token JWT.
  + Parámetros: JSON con `email` y `password`.
  + Respuesta: JWT token.
* Autorización: Se implementa utilizando **JWT**, protegiendo rutas con decoradores como @Auth(ValidRoles.user) para garantizar que solo usuarios autenticados puedan acceder a rutas específicas.

**Contenido**

* POST /api/v1/content: Crea un nuevo contenido (accesible solo para admins).
  + Parámetros: JSON con los datos del contenido.
  + Respuesta: Objeto contenido creado.
* GET /api/v1/content: Lista todo el contenido disponible.
  + Respuesta: Array de contenidos.
* GET /api/v1/content/:id: Obtiene un contenido específico por ID.
  + Parámetro: `id` (UUID del contenido)
  + Respuesta: Objeto contenido o error 404.
* PATCH /api/v1/content/:id: Actualiza un contenido existente (solo admins).
  + Parámetros: `id` (UUID del contenido), JSON con los datos a actualizar.
  + Respuesta: Contenido actualizado o error 404.
* DELETE /api/v1/content/:id: Elimina un contenido (solo admins).
  + Parámetro: `id` (UUID del contenido)
  + Respuesta: Confirmación de eliminación o error 404.

**Comentarios**

* POST /api/v1/comments: Crea un nuevo comentario.
  + Parámetros: JSON con `userId`, `contentId`, y el contenido del comentario.
  + Respuesta: Comentario creado.
* GET /api/v1/comments: Obtiene todos los comentarios.
  + Respuesta: Array de comentarios.
* GET /api/v1/comments/:id: Obtiene un comentario específico por ID.
  + Parámetro: `id` (UUID del comentario)
  + Respuesta: Comentario o error 404.
* POST /api/v1/comments/reply/:id: Responde a un comentario existente.
  + Parámetros: `id` (UUID del comentario padre), 'userId' (id del usuario que comenta) , 'contentId' (id del contenido comentado), JSON con los datos del comentario de respuesta.
  + Respuesta: Comentario de respuesta creado.
* GET /api/v1/comments/parent/:id: Obtiene todas las respuestas a un comentario.
  + Parámetros: `id` (UUID del comentario padre).
  + Respuesta: Array de comentarios.
* PATCH /api/v1/comments/:id: Actualiza un comentario existente.
  + Parámetros: `id` (UUID del comentario), JSON con los datos a actualizar.
  + Respuesta: Contenido actualizado o error 404.
* DELETE /api/v1/comments/:id: Elimina un comentario (solo admins).
  + Parámetro: `id` (UUID del comentario)
  + Respuesta: Confirmación de eliminación o error 404.

**Reacciones**

* POST /api/v1/reactions: Crea una reacción en un comentario.
  + Parámetros: JSON con `userId`, `commentId`, y el tipo de reacción (`like` o `dislike`).
  + Respuesta: Reacción creada.
* GET /api/v1/reactions: Obtiene todas las reacciones.
  + Respuesta: Array de reacciones.
* GET /api/v1/reactions/comment/:id: Obtiene las reacciones de un comentario.
  + Parámetro: `id` (UUID del comentario)
  + Respuesta: Array de reacciones.
* GET /api/v1/reactions/user/:userId: Obtiene las reacciones de un usuario.
  + Parámetro: `userId` (UUID del usuario)
  + Respuesta: Array de reacciones.
* DELETE /api/v1/reactions/:id: Elimina una reacción.
  + Parámetro: `id` (UUID de la reacción)
  + Respuesta: Confirmación de eliminación o error 404.

**Listas**

* POST /api/v1/lists: Crea una nueva lista de contenido para un usuario.
  + Parámetros: JSON con `userId`, `contentIds` y el estado de la lista.
  + Respuesta: Lista creada.
* GET /api/v1/lists/:userId: Obtiene las listas de un usuario.
  + Parámetro: `userId` (UUID del usuario)
  + Respuesta: Array de listas.
* DELETE /api/v1/lists/:id: Elimina una lista específica.
  + Parámetro: `id` (UUID de la lista)
  + Respuesta: Confirmación de eliminación o error 404.

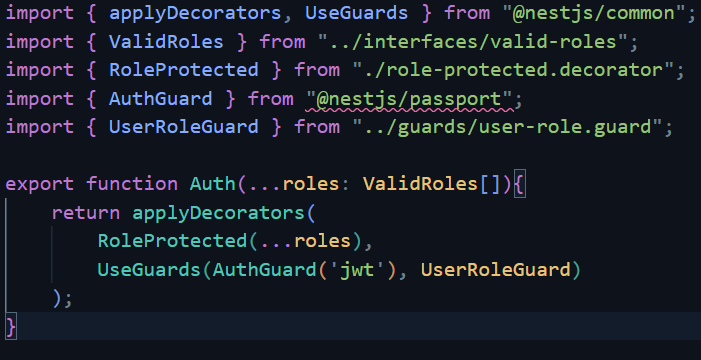
**3. Autenticación y Autorización**

La autenticación de la API se implementa utilizando **JWT (JSON Web Tokens)**. Los usuarios deben autenticarse enviando un **POST** al endpoint /auth/login, lo que devuelve un token que debe ser incluido en las cabeceras de futuras solicitudes como **Bearer Token**.

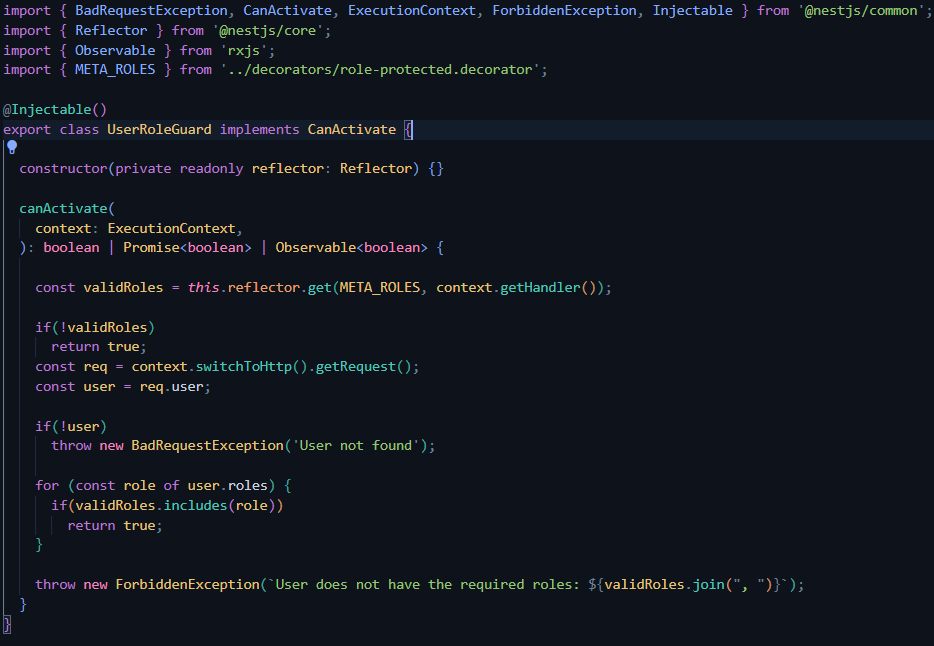
**Guards**

Los **guards** en NestJS sirven para aplicar lógica de autorización antes de que un controlador o una ruta sean ejecutados. En esta aplicación, se utilizan varios guards:

- **AuthGuard:** Este guard se utiliza para validar si un usuario está autenticado verificando su JWT. Si el token es válido, el guard permite el acceso a la ruta; de lo contrario, devuelve un error de autenticación.

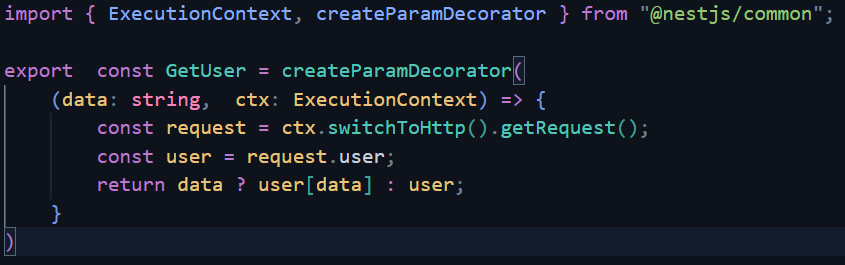


- **UserRoleGuard:** Este guard verifica que el usuario autenticado tenga los roles necesarios para acceder a una ruta. Si no tiene los permisos adecuados, se lanza una excepción de autorización.

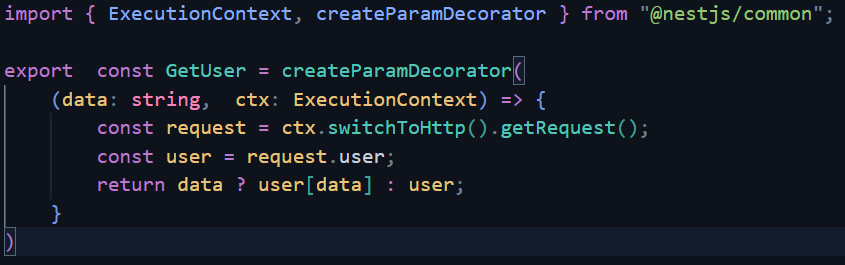


**Decoradores**

- **@Auth():** Decorador que combina la verificación de autenticación (JWT) y de roles. Permite especificar qué roles tienen acceso a una ruta.



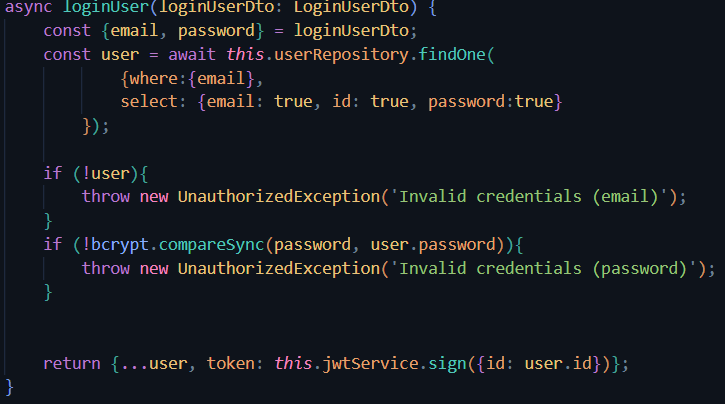
- **@GetUser():** Decorador que obtiene la información del usuario autenticado desde el JWT. Facilita el acceso a los datos del usuario en los métodos de los controladores.



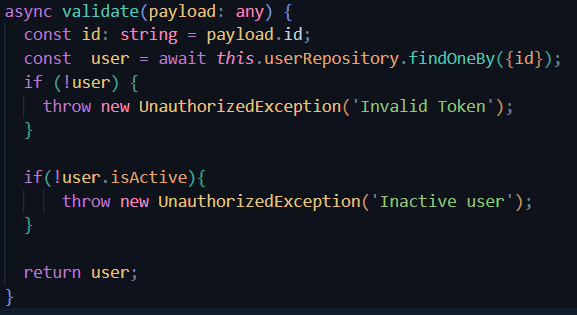
**JWT Strategy**

La estrategia JWT es la base del sistema de autenticación. En la implementación de esta aplicación:

1. **Generación del JWT:** Cuando un usuario inicia sesión, se genera un JWT que incluye la información del usuario, como su `id`. Este token se devuelve al cliente y se usa en las siguientes solicitudes.



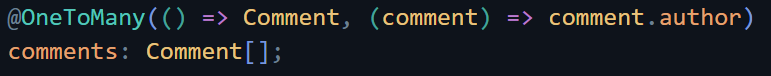
2. **Validación del JWT:** Cada vez que se hace una solicitud a una ruta protegida, el AuthGuard utiliza el **JwtStrategy** para validar el token. Si el token es válido, se permite el acceso.



**4. Persistencia en la Base de Datos**

La persistencia de datos en la API se maneja a través de **TypeORM** y una base de datos **PostgreSQL**. Todas las entidades (usuarios, contenido, comentarios, listas y reacciones) están mapeadas a tablas en la base de datos utilizando las clases de TypeORM.

- **Relaciones entre entidades:** La aplicación utiliza relaciones **OneToMany** y **ManyToOne** para gestionar las dependencias entre entidades como usuarios y comentarios, o comentarios y reacciones. TypeORM permite manipular estas relaciones de forma transparente.



- **Sincronización automática:** Durante el desarrollo, se utilizó la opción (synchronize: true) para sincronizar los cambios en las entidades con la base de datos, permitiendo que las tablas se actualicen automáticamente al cambiar las entidades.

- **DTOs y Validaciones:** Las validaciones se realizan utilizando **class-validator**, que asegura que los datos de entrada en los DTOs (Data Transfer Objects) cumplan con los requisitos. Por ejemplo, se valida que los correos electrónicos tengan un formato correcto o que los campos obligatorios no estén vacíos.

**5. Validaciones**

Las validaciones de los datos de entrada se realizan utilizando **class-validator**. Cada DTO tiene reglas de validación que aseguran que los datos enviados en las solicitudes cumplan con los requisitos específicos, como la longitud mínima de cadenas, formatos de correos electrónicos, y tipos de datos específicos como enums.