Backend para una aplicación de ratings escolares

Esteve Jimenez Vidal

Resumen—El proyecto, como su nombre indica, consiste en la realización, desde cero, de un backend pensado para dar soporte a una aplicación tanto móvil como web que se dedique a la clasificación y a la valoración de tanto las asignaturas como del personal que las imparte, teniendo en cuenta para ello cualquier tipo de estudios de carácter no obligatorio. Este backend, está formado por una API de tipo REST que recibe las peticiones por parte de un cliente externo, las procesa y, mediante la comunicación con una base de datos NoSQL, genera la respuesta. A lo largo de este documento, se explicarán los objetivos del mismo, la metodología seguida, las diferentes herramientas utilizadas para la realización de este trabajo, entre las cuales destacan MongoDB y NestJS; también habrá una explicación del trabajo realizado junto con su desarrollo y, para finalizar, las conclusiones obtenidas después de finalizar el desarrollo.

Palabras clave—API, RESTful, Servicio, NestJS, Node.js, JavaScript, TypeScript, MongoDB, Guards, Backend, App, JSON, JWT, Swagger, NoSQL, IoT

Abstract— The project, as its name indicates, consists of the realization, from scratch, of a backend designed to support both a mobile and web application that is dedicated to the classification and assessment of both the subjects and the staff that teaches them, taking into account for this any type of studies of a non-compulsory nature. This backend is made up of a REST-type API that receives requests from an external client, processes them and, through communication with a NoSQL database, generates the response. Throughout this document, its objectives, the methodology followed, the different tools used to carry out this work, among which MongoDB and NestJS stand out, will be explained; There will also be an explanation of the work carried out along with its development and, finally, the conclusions obtained after finishing the development.

Index Terms—API, RESTful, Service, NestJS, Node.js, JavaScript, TypeScript, MongoDB, Guards, Backend, App, JSON, JWT, Swagger, NoSQL, IoT

1 Introducción

En la última década, sobre todo, la sociedad se ha transformado drásticamente debido al uso de las nuevas tecnologías e internet, mientras que en 2010 solo estaban conectados un 53% de los españoles, en 2022 hay más de un 96% de españoles con conexión a internet. Por tanto, a día de hoy, es casi lo mismo hablar de población como de gente conectada. Debido a este gran cambio, de manera gradual, las personas han integrado el uso de internet en su día a día y con ello, se han popularizado aplicaciones sociales como Facebook, Instagram o, la más reciente, Tik Tok. Esto, ha generado un aumento significativo en el valor de las opiniones de la gente, ya que, dependiendo de estas, un negocio o una persona pueden prosperar o hundirse de la noche a la mañana. Por lo tanto, la influencia de la opinión pública, hoy en día, es tan grande que la mayoría de las personas cuando tienen que tomar decisiones a la hora de cómo organizar un viaje, a que restaurante ir a comer o, incluso, donde ir a cortarse el pelo, previamente, han utilizado una app como TripAdvisor o, las propias reseñas de Google, por ejemplo, para investigar los diferentes locales o agencias y así, decidir dónde ir o que hacer, dependiendo de las opiniones y reseñas de otros clientes previos, generando así competencia y un interés en el desarrollo y mejora propios.

Para el ambiente académico, estas valoraciones, existen de manera oficial como, por ejemplo, el "Academic Ranking of World Universities, ARWU "que entre otros métodos de clasificación se dedican a clasificar las universidades del mundo según diferentes factores como, entre otros, cuantos alumnos han ganado un premio Nobel o cuantos artículos han sido publicados en revistas científicas como Science o Nature. Todos estos métodos de clasificación, no sirven para mejorar en ningún aspecto y, no se tienen en consideración a la hora de tomar ninguna decisión por parte de un nuevo estudiante o por parte del equipo docente.

En EEUU existen 2 aplicaciones web que se dedican a valorar a los profesores de todas sus universidades y, anualmente, publican la lista con los que tienen mayor puntuación; En Europa, en cambio, se hacen encuestas internas por parte de las Universidades y cómo podemos comprobar el índice de participación de las mismas no llega a un 30%.

- *E-mail de contacto: esteve.jimenezv@autonoma.cat*
- *Mención realizada: Tecnologies de la Informació*
- *Trabajo tutorizado por: Julian del Amo Montoya (DEIC)*
- *Curso 2021/22*

2 Objetivos

Este proyecto trata sobre el diseño e implementación de un backend pensado para dar soporte a una aplicación de ratings escolares para estudios de carácter no obligatorio.

Como objetivo principal está la construcción de una API/REST que, mediante la comunicación con una base de datos NoSQL, sea capaz de procesar las valoraciones tanto de las asignaturas como de los profesores que las imparten.

Como objetivo secundario: crear la base para un servicio que beneficie tanto a estudiantes como al equipo docente ya que estará pensado para que los alumnos tengan más información a la hora de escoger y, para que puedan valorar tanto a las asignaturas como a los profesores que las imparten; por otro lado, el equipo docente tendrá una alternativa a los cuestionarios de la cual podrán obtener información en cualquier momento en vez de una vez por curso.

3 State of art

El desarrollo de aplicaciones no ha parado de avanzar y de presentar novedades prácticamente desde su nacimiento y, lo hace, a una velocidad de vértigo. Los desarrolladores están literalmente saturados ante opciones nuevas, lenguajes, arquitecturas, librerías o frameworks. Y esto es parece no tener fin, ya que las alternativas crecen semana tras semana. Ante este panorama de continuo avance, nos encontramos con corrientes que, por sus ventajas y versatilidad, han adquirido un mayor protagonismo, como es el caso del desarrollo basado en API REST; tradicionalmente se basaba en construir sistemas que devolvieran al cliente código HTML, capaz de ser interpretado directamente por el navegador. De esta manera, cuando se programaba en PHP, Python, .NET, etc. lo normal era que se entregase al navegador todo lo necesario para mostrar la página a su usuario.

El punto débil de esta alternativa se basa principalmente en:

1. Hoy no se consume el servicio web solo a través de Internet, sino también por medio de aplicaciones para móviles, etc. Si producimos HTML desde el lado del backend, es muy probable que tengamos que programar varias veces ese backend para cada sistema al que lo queramos portar.

2. Ante la cantidad de alternativas que aparecen con tanta velocidad, la parte que menos cambia es la del backend, por lo que es interesante que podamos desarrollar esa capa de manera que se pueda adaptar a cualquier tipo de librería del lado del frontend.

En cambio, cuando desarrollamos un servicio backend construyendo una API, nos aseguramos que se pueda consumir desde cualquier sistema o cliente. La API no devuelve más que datos, que están desacoplados a cualquier modo de visualización. Si estamos implementando una web, consumiremos la API desde cualquiera de los frameworks o librerías JavaScript populares, como AngularJS, Polymer o ReactJS. También se podrán consumir los servicios de la API desde aplicaciones desarrolladas en Java para Android o Swift/Objective C para iOS, por ejemplo.

Dentro de las alternativas de construcción de una API, REST es un estilo de arquitectura de software que se utiliza para describir cualquier interfaz entre diferentes sistemas que utilice HTTP para comunicarse. Este término significa REpresentational State Transfer (transferencia de estado representacional), lo que quiere decir que, entre dos llamadas cualquiera, el servicio no guarda los datos. Por ejemplo, podemos autenticar a un usuario con su email y contraseña en una llamada, pero la siguiente que hagamos ya se habrá olvidado de la anterior petición de autenticación, esto es algo que se resuelve mediante un token que el cliente debe enviar al servidor en cada solicitud que se realice. Las características que definen una arquitectura REST son las siguientes:

1. El cliente y el servidor están débilmente acoplados, es decir, el cliente no necesita conocer los detalles de implementación del servidor y el servidor no se preocupa de cómo utiliza el cliente los datos.

2. No hay estado, es decir, cada petición que recibe el servidor es independiente.

3. Se utilizan los verbos HTTP GET, POST, PUT y DELETE para el acceso, creación, actualización y borrado de recursos.

4. Las llamadas son cacheables para así evitar pedir varias veces un mismo recurso.

5. La interfaz es uniforme, es decir, cada recurso del servicio REST debe tener una única dirección URI.

4 Metodología y planificación

Para el desarrollo de este proyecto se ha optado por escoger la metodología ágil Kanban ya que por el tipo de proyecto y al estar formado por una solo persona, era el enfoque más cómodo y que generaba un mejor flujo de trabajo debido a la gran flexibilidad que tiene a la hora de organizar y añadir nuevas tareas, permitiendo desglosar grandes procesos en pequeñas tareas para focalizar los puntos más importantes a realizar primero, dejando para más tarde las tareas de menor importancia. Además de contribuir al proceso de mejora continua gracias a poder añadir o modificar tareas existentes para cumplir con las necesidades generadas en cualquier momento del desarrollo.

A partir de aquí se ha realizado una planificación en tres fases dictadas por los periodos de entrega de los informes de progreso:

Una primera fase dedicada al diseño y a la implementación de la base de datos junto con la creación de un esqueleto completo de la API que ya permita interactuar con la

BD de manera local, es decir, que tanto la API como la BD se acceden mediante localhost y no son accesibles desde ningún lugar externo.

Una segunda fase donde, en primer lugar, acabar de implementar todos los métodos y endpoints necesarios teniendo en mente en todo momento que tipo de peticiones nos haría el posible frontend, en segundo lugar, implementar medidas de seguridad y, en tercer lugar, implementar y documentar la API mediante Swagger.

Una tercera fase, dedicada a detectar y corregir posibles fallos, mejorar el código siguiendo las best practices, poblar la base de datos con datos reales y, pasar de local a online para que la API sea accesible desde un cliente externo.

5 BASE DE DATOS

5.1 Elección

La base de datos esta implementada utilizando MongoDB, siendo este, un sistema de base de datos NoSQL que almacena datos en documentos flexibles con una especificación similar a JSON llamados BSON, por lo que los campos pueden variar entre documentos y la estructura de datos puede cambiarse con el tiempo. Además, MongoDB es una base de datos distribuida en su núcleo, por lo que permite una alta disponibilidad, escalabilidad horizontal y la distribución geográfica está integrada. Por estos motivos, incluyendo que es uno de los sistemas más utilizados para integraciones con APIs, siendo gratuito y de código abierto, ha sido la firme elección para este proyecto.

5.2 Diseño

Como hemos mencionado en el subapartado anterior, la base de datos es de tipo NoSQL, orientada a documentos, por lo que utilizamos colecciones para guardar nuestros datos, en vez de tablas como haríamos en una base de datos SQL.

El diseño de la BD ha ido evolucionando de forma paralela con el avance del proyecto, al principio, iba a ser una gran colección en la que cada institución contuviera, de cada titulación, sus asignaturas y, dentro de ellas los profesores que las imparten junto a las valoraciones, tanto de las asignaturas como de los mismos profesores. Esto, provocaba que a cada consulta realizada se tuvieran que transferir grandes cantidades de datos irrelevantes para la petición que tendrían que ser procesados por el frontend. Teniendo en mente responder a las peticiones con la cantidad mínima e indispensable de información, para así tener una mejora en el tiempo de respuesta, una mejora en el procesado de datos y una mejora en las colecciones, haciendo más simple el uso de las mismas, tanto a la hora de obtener los datos contenidos en ellas como a la hora de crearlos o modificarlos. Debido a esta evolución de la lógica del programa, la BD final, está compuesta por diferentes colecciones simples que pueden obtener datos relacionados de otras colecciones mediante el identificador de estos, por tanto, nos queda una BD fragmentada en diversas colecciones que, a cambio de necesitar realizar más peticiones, entrega los datos de manera más precisa y maleable sin grandes cantidades de datos innecesarios en cada entrega.

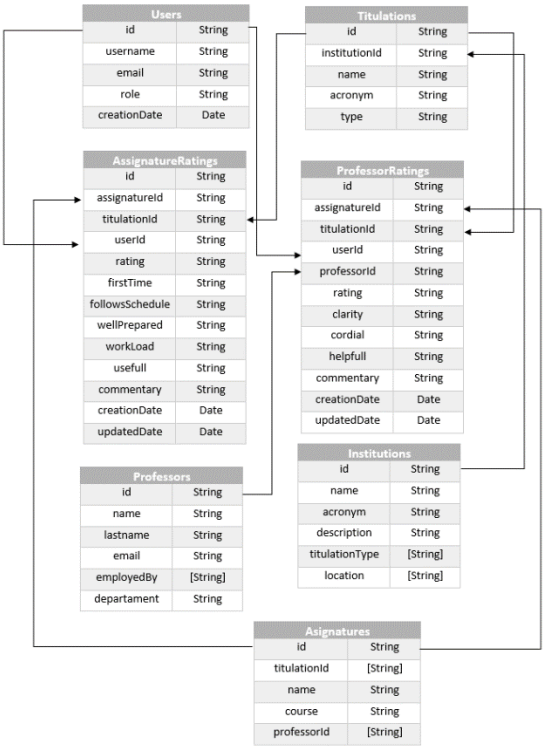

Fig. 1 Schema of the DB collections

Como se puede observar en el esquema anterior además de las relaciones previamente mencionadas, tanto en *"AssignatureRatings"* como en *"ProfessorRatings"*, entre *"rating"* y *"commentary"* hay algunos nombres identificadores del tipo clave valor que pueden generar duda a primera vista y estos se corresponden con las preguntas de las encuestas de evaluación de la UAB que corresponden con:

- *firstTime*: ¿Es la primera vez que se cursa esa asignatura?
- *followsSchedule*: ¿Se ha seguido la programación prevista de la asignatura explicada en la guía docente?
- *wellPrepared*: El material del curso (enunciados de problemas, presentaciones en clase, guiones de prácticas, ¿etc.) están bien preparados y resultan útiles?
- *usefull:* ¿Se considera que con la asignatura se están aprendiendo cosas útiles para la formación propia?
- *clarity:* ¿El profesor se expresa con claridad a la hora de exponer o explicar un tema?
- *cordial:* ¿El profesor mantiene un buen ambiente de relación personal y comunicación con el alumnado?
- *helpfull:* ¿El profesor aprovecha los resultados de las actividades de evaluación para hacer comentarios que ayuden a la mejora de los estudiantes?

6 API

6.1 NestJS

En los últimos años, gracias a Node.js, JavaScript se ha convertido en la "*lingua franca*" de la web tanto para aplicaciones frontend como backend. Esto ha dado lugar a proyectos increíbles como Angular, React y Vue, que mejoran la productividad de los desarrolladores y permiten la creación de aplicaciones frontend rápidas, comprobables y extensibles. Sin embargo, si bien existen muchas bibliotecas y herramientas excelentes para Node (y JavaScript del lado del servidor), ninguno de ellos resuelve de manera efectiva el problema principal de arquitectura. Nest proporciona una arquitectura de aplicaciones novedosa que permite a los desarrolladores y equipos crear aplicaciones altamente comprobables, escalables, débilmente acopladas y fáciles de mantener, estando esta fuertemente inspirada en Angular.

Utiliza JavaScript progresivo, está construido y soporta por completo TypeScript, aunque permite a los desarrolladores escribir en puro JavaScript, combina elementos de OOP (Programación Orientada a Objetos), FP (Programación Funcional) y FRP (Programación Reactiva Funcional).

A nivel interno, hace uso de frameworks como Express por defecto para la creación de servidores HTTP robustos.

Nest mayormente se organiza en tres tipos de módulos distintos y, aunque puedas crear funciones y clases independientes, estas funcionaran para complementar uno de estos módulos. Estos módulos son:

1. *Controller:* Es el que se encarga de recibir y responder a las peticiones externas a la aplicación, el mecanismo de enrutamiento es el que decidirá que controller se encargará de recibir que peticiones, es decir, funcionara como endpoint para las peticiones externas.

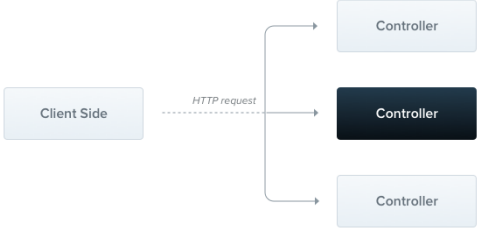

Fig. 2 Controller example

2. *Module*: Es el que se encarga de proveer los metadatos necesarios para organizar la estructura de las aplicaciones. Como mínimo, tiene que haber un module inicial, este se encargara de ser el punto de arranque de toda la estructura interna de datos que utiliza nest para resolver las dependencias y relaciones que generan sus módulos y proveedores.
3. Service: Es el que se encarga de proveer la funcionalidad y normalmente es utilizado por algún controller para acceder o manipular los datos.

Es decir, el controller se encarga de recibir las peticiones externas a la aplicación mediante el sistema de enrutamiento que le configuremos, este pasa la petición al service que se encarga de ofrecer los métodos necesarios para realizar la petición, una vez realizada, el controller responde con esta, mientras tanto, el module organiza la interacción entre estos dos y el resto de la aplicación, normalmente aparecen los 3 formando un conjunto con el mismo nombre para facilitar la identificación y la organización de los mismos.

6.2 Estructuras de datos

Durante la realización de este proyecto se han utilizado tres tipos de objetos para tratar los datos y aunque son muy parecidos e incluso dos de ellos son casi idénticos, tienen funcionalidades diferentes. Estos son:

1. *DTO (Data Transfer Object):*
   Se trata de un patrón que consiste en la creación de objetos planos (POJO) con una serie de atributos que puedan ser enviados o recuperados del servidor en una sola invocación, de tal forma que un DTO puede contener información de múltiples fuentes o tablas y concentrarlas en una única clase simple. Se utiliza para crear diferentes estructuras de datos que sirven tanto para el envío como para la recepción de datos desde el servidor a un cliente externo o viceversa.
2. *Interfaz o interface (en inglés)*
   Es un tipo de estructura que describe para cualquier clase que derive de ella la estructura del objeto a seguir, lo que significa que describe que propiedades y métodos debería tener una clase, pero no los implementa. Se utiliza para el tratamiento de los datos dentro de la propia aplicación.
3. *Schema:*
   Muy similar a la interface, incluso tienen relación, pero estos se utilizan para definir modelos, estos modelos realmente son una manera bonita de llamar a los constructores compilados a partir de los Schemas, estos son los que se encargan de crear y leer documentos de la BD de MongoDB. Es decir, definen como serán las colecciones y se encargan de tratar los datos entre la BD de MongoDB y la API

6.3 Mongoose

Es la herramienta de modelado de objetos de MongoDB más popular para Node.js, Nest, además facilita una librería derivada de ella con el mismo nombre y, esta, mediante llamadas asíncronas, gracias a las promesas, nos permite conectarnos y utilizar las funciones de lectura y escritura en la BD de MongoDB utilizando los Schemas mencionados en la subsección anterior.

- Para realizar la conexión con la BD se debe importar y utiliza, en el module que inicia la aplicación, *MongooseModule* y, con el método *forRoot("mongodb://localhost/nombre de tu BD")* si la utilizas de manera local o *forRoot("mongodb+srv://usuario:contraseña@direccion/nombre*

de tu BD") en caso de utilizar un servidor.

- Para poder hacer la conexión con las diferentes colecciones y, después de tener una conexión con la BD, necesitaremos importar el mismo *MongooseModule* en el module que queremos que tenga acceso y esta vez con el método *forFeature()* definiremos el nombre de la colección y el Schema que va a seguir, si esta colección no existe, al interactuar con ella se creara automáticamente.
- Por último, para interactuar con la colección a la que tengamos acceso, desde el service de ese mismo module, debemos importar las librerías *Model* e *InjectModel* de *mongoose* para, en el constructor, poder declarar el modelo con el que vamos a utilizar los métodos de *find()* y derivados que tenemos para interactuar con el documento (tanto de lectura como escritura).
  Un ejemplo de cómo instanciar el modelo dentro del constructor de mi clase *Assignatures* seria: *@InjectModel("Assignatures") private assignaturesModel: Model<Assignatures>,* entonces ya podríamos utilizar *this.assignaturesModel.findById(id)*, por ejemplo.

6.4 CRUD

La mayoría de los métodos de la API y el enfoque general de esta es el método CRUD, este representa las siglas de Create, Read, Update y Delete, es decir, es un acrónimo de las maneras en las que se puede operar sobre la información almacenada.

Este sistema no es uno que genere dificultades en su comprensión ya que su complejidad depende de las propias restricciones que el desarrollador quiera implementar. En MongoDB los métodos para esto serian:

1. *find():* Retorna la lista de todos los objetos de la colección
2. *findById():* Retorna el que coincida con el id de mongo pasado por valor
3. *findOne():* Retorna el primero que coincida
4. *save():* Crea un nuevo objeto en la colección
5. *findByIdAndUpdate():* Modifica el objeto encontrado por el id de Mongo con los nuevos datos
6. *findByIdAndDelete():* Elimina el objeto con la id de Mongo enviada.

Estos son los más importantes o más comúnmente utilizados, pero hay variaciones. Si se siguen las best practices y, el diseño propio de Nest, estos métodos deberían ser parte del service y ser llamados por el endpoint pertinente localizado en el controller.

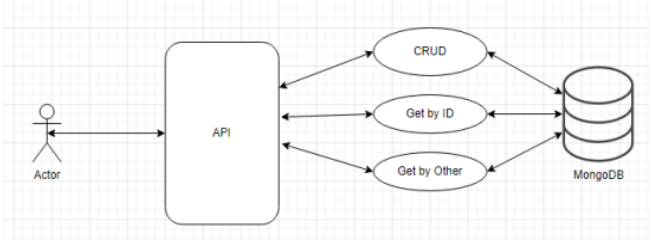

Fig. 3 Interacciones de la API con la BD

6.5 Filtrado de datos

Las estructuras de datos como los DTOs o los Schemas mencionados en la subsección 6.2 definen los parámetros de los objetos, pero se les puede dar una mayor utilidad ya que se pueden filtrar los datos para que lleguen con un formato especifico y solo se almacenen los datos que queramos almacenar y no otros. Para ello hay 3 maneras:

1. Mediante la utilización de validation pipes en los DTOs para comprobar que la información recibida contiene todos los campos no opcionales y eliminar cualquier campo que no esté incluido en el propio DTO.
2. Mediante la utilización de decoradores de la clase de *class-validator*, estos funcionan como condiciones que determinan si el valor recibido es válido o no, algunos de estos pueden ser *@IsNotEmpty* que si encuentra un valor vacío retorna error o *@Max("number")* que si el valor recibido supera el numero especificado retorna error.
3. En la parte del service, automatizar la inicialización de las variables opcionales del DTO que no hayan sido recibidas o estén con valores nulos, inicializar el valor de variables que no provengan del exterior como podría ser la fecha o, validar si el dato introducido tiene el formato valido como podría pasar con una id de MongoDB que tienen una especificación de longitud ya definida y, cualquier id que se le pase que no cumpla con esa especificación, devuelve un error.

6.6 Autentificación

La autentificación por parte de la API utiliza tanto los *Guards* como *Passport* para identificar si el usuario existe y validar que dice ser quien es, en caso de no ser así se le deniega el acceso.

Para empezar, *Passport* es un middleware de autentificación para Node.js y puede ser utilizado por cualquier aplicación web basada en *Express*, por consecuente, *Passport,* permite una gran cantidad de estrategias para comprobar que, al identificarte con los parámetros requeridos, realmente eres quien dices ser.

La estrategia escogida para este proyecto es la de *passport-local*, una estrategia básica que requiere la introducción de los parámetros *usuario y contraseña,* teniendo esto en cuenta, queda la implementación de la estrategia que, en mi caso, consiste en comprobar que exista el nombre de usuario y, si es así, comprobar que la contraseña introducida coincide con la contraseña almacenada.

No se ha mencionado antes, pero, para mantener la seguridad de los datos, al crear o modificar un usuario con una nueva contraseña, de esta, se almacena un hash generado utilizando los métodos de encriptación de la librería *Bcrypt* y, al hacer la comprobación, se compara la contraseña introducida con el hash almacenado en la BD.

Si la identificación resulta negativa, el programa lanza una UnauthorizedException, en cambio, si resulta positiva, devuelve un done como verificación.

Por otra parte, los *Guards*, son clases con un decorador

del tipo @Injectable() que deberían implementar la interfaz *canActivate*, y como su nombre implica, su función es proteger una ruta añadiendo lógica entre la petición y el enrutador. Tiene la responsabilidad de decidir si una petición será entregada al enrutador o descartada inmediatamente dependiendo de ciertas condiciones.

Para esta primera instancia en la que todavía no estamos autenticados, creamos un custom guard que, utilizando *AuthGuard* de *Passport* y, con la estrategia creada derive la petición para realizar la autentificación antes de entrar al endpoint, en mi caso se llama *LocalAuthGuard*. Por tanto, si la comprobación resulta negativa no llegara a entrar al endpoint y, si resulta positiva necesitaremos una manera de seguir autentificados ya que, si recordamos las propiedades de REST, no hay estado y cada petición es independiente.

La solución al problema de estado la podemos encontrar mediante la generación de un token al finalizar el método de autentificación, más concretamente un JSON Web Token (JWT), este, es el código referente a un objeto encriptado que contiene un secreto, en este primer caso contendrá la combinación con el usuario y la contraseña que hayamos introducido y haya sido validada.

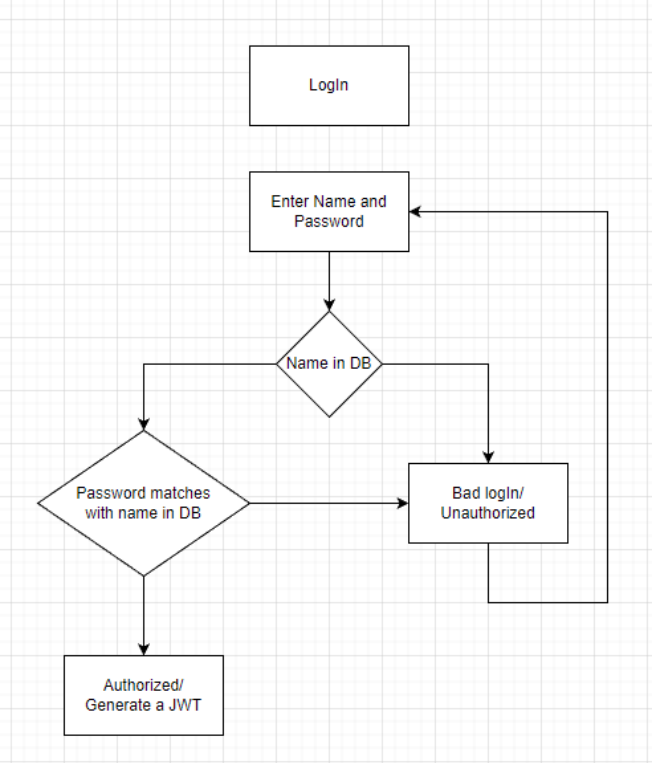

Fig. 4 Simplified LogIn Flow chart

Para le generación de JWT hay que generar un método nuevo mediante el *JwtModule* que genere un payload con los datos que queremos y lo firme con el código secreto que hemos introducido, en mi caso, además, el token tiene una fecha de expiración de 1h.

Una vez tenemos el JWT necesitamos un nuevo método que lo desencripte y valide que los datos coinciden con los de la base de datos de la misma manera que lo hacia el método login. Para ello, debemos crear un nuevo custom guard, al igual que *LocalAuthGuard,* este procesara el JWT antes de llegar al endpoint y verificara si el JWT es válido y si el usuario y la contraseña coinciden con el usuario y la contraseña almacenados en la BD. El proceso para utilizar los guards se realiza mediante el decorador *@UseGuards(),* donde se introducen los guards que quieres que se ejecuten antes de cada endpoint, por ejemplo, el *JwtAuthGuard* que correspondería a este último explicado, lo utilizo en todos los métodos excepto el de login y el de crear un usuario nuevo, impidiendo el acceso para cualquiera que no esté registrado.

6.7 Autorización

La autorización es el proceso que determina lo que un usuario tiene permitido hacer y, aunque la autorización es independiente de la autentificación, requiere de ella.

En este proyecto hay 3 tipos de usuarios, por lo tanto, un RBCA (Role-Based Acces Control) es un buen mecanismo de control para determinar el acceso definido por los roles y sus privilegios. Los tres tipos de usuario son:

1. Los usuarios base llamados users, son los que deberían tener acceso a las funciones de creación, modificación y, eliminación tanto de los ratings de asignatura como de profesores.
2. Los usuario administradores, son los que se dedican a mantener los datos con respecto a las instituciones, asignaturas, titulaciones y profesores, por tanto, deberían poder acceder a la creación, modificación y eliminación de estas.
3. Un super usuario llamado developer, este tiene acceso a todos los métodos posibles ya que se realizarán los test necesarios utilizando este rol.

Para realizar el control de acceso necesitamos: primero, generar una clase Role que sirva de enumerador, ya que podría estar en la base de datos, pero tampoco hace falta. Después, y para ahorrarnos repetir código innecesario creamos un decorador @Roles() que sirva para especificar que roles tendrán permitido el acceso.

Consecuentemente, ya podemos utilizar en cada método el decorador junto a los roles a los que les queramos dar acceso, pero esto no haría nada ya que al igual que en el apartado de autentificación, el que se encarga de denegar el paso es un *Guard* que habrá que generar.

Para la creación de este nuevo *Guard,* tendremos que obtener los metadatos de la petición y para ello importamos la clase *Reflector*, esta se encargara de revisar y obtener el rol asociado a la petición. Para ello, hay que modificar el payload que se genera en el login e incluir el campo *Role,* para que, al hacer el proceso de autentificación, se desencripte el JWT y, al hacer el proceso de autorización el reflector lo encuentre y lo puedas comparar con los roles autorizados, si coinciden se permite el acceso, si no, se manda una *ForbiddenException.*

6.8 Documentación con Swagger

Cuando hablamos de Swagger nos referimos a una serie de reglas, especificaciones y herramientas que nos ayudan a documentar nuestras APIs. De esta manera, podemos realizar documentación que sea realmente útil y entendible ya

que, esta, se basa en la OpenAPI Specification (OAS); La OAS, define una interfaz estándar independiente del idioma para las API RESTful que permite que tanto los humanos como las computadoras descubran y comprendan las capacidades del servicio sin acceso al código fuente, la documentación o a través de la inspección del tráfico de red. Cuando se define correctamente, un consumidor puede comprender e interactuar con el servicio remoto con una cantidad mínima de la lógica de la implementación.

Las herramientas de generación de documentación pueden usar una definición de OpenAPI para mostrar la API, herramientas de generación de código para generar servidores y clientes en varios lenguajes de programación, herramientas de prueba y muchos otros casos de uso.

Sus ventajas predominan de tal manera que Swagger puede definirse como la aplicación estándar por excelencia para la descripción de interfaces en las API de RESTful. Como otras tantas aplicaciones de código abierto, Swagger disfruta de una gran divulgación y, con ello, de compatibilidad con muchas herramientas. El gremio de Swagger está formado por grandes de la tecnología como Microsoft, IBM y Google, con lo que la especificación OpenAPI cuenta con respaldo incondicional, aunque haya alternativas como RESTful API Modelling Language (RAML). Este también se basa en YAML y desarrolla definiciones todavía más complejas que Swagger, pero incluso su creador (Mulesoft) ha entrado a formar parte de la OpenAPI Initiative.

Por suerte, Nest provee un módulo dedicado que permite generar la especificación Swagger sin la necesidad de crear un documento YAML externo, en cambio, el documento es autogenerado y tiene que ser iniciado en el main de la aplicación pasándole los parámetros de configuración iniciales.

Una vez realizada la configuración inicial, ya se autogenera el documento Swagger con todos los endpoints y, es accesible mediante *http://url/api* o, en mi caso, *http://url/api/docs*, esta no genera ningún tipo de información como podrían ser las posibles respuestas o los parámetros que necesita recibir mediante JSON. Para ello, ya que Nest nos provee con este módulo dedicado, hay que ir implementando, mediante decoradores, los métodos para generar estas especificaciones que faltan, hay varios tipos y los más utilizados en este proyecto son:

1. Se pasan como parámetro dentro del método del controller para poder ser usados más adelante, estos son *@Body*, *@Param* y *@Query*, que indican respectivamente si, se va a recibir un JSON y de que tipo mediante los DTOs que hayas especificado previamente, si se va a recibir un parámetro como podría ser una id por la url o, si se va a recibir una query mediante la url.
2. Estrechamente relacionados con el anterior están *@ApiBody*, *@ApiParam* y *@ApiQuery* que habilitan la introducción de estos mediante los docs de Swagger.
3. Para los parámetros dentro de los DTOs tenemos *@ApiProperty* y *@ApiPropertyOptional,* que sirven para definir el tipo de dato y asignarle un nombre.
4. *@ApiBearerAuth* sirve para indicar los métodos que requieren estar autorizado
5. *@Api…Response,* junto a una descripción, sirve para indicar las posibles respuestas ya sean de ok o excepciones que lanzara el programa.
6. *@ApiTags,* especifican el nombre de un conjunto para mejorar luego su búsqueda e identificación.

7. De local a online

El proceso realizado para hacer esta transición ha sido, más o menos, simple de realizar, ya que, por una parte, ya estaba el código del programa en GitHub debido a que se ha estado utilizando la plataforma para guardar el progreso y tener un control de versiones.

Para la parte de la base de datos, MongoDB tiene una plataforma llamada Atlas que permite de manera gratuita crear y mantener un espacio limitado en el que crear y mantener BDs alojadas en la nube, por lo que la transición ahí, ha sido cambiar la dirección a la que conectar la BD en la API y exportar las colecciones de una BD a la otra utilizando MongoDB Compass que te permite monitorizar tus BDs de MongoDB. Al construir la BD con Atlas se consigue un nivel de seguridad extra porque te permite definir desde que direcciones se puede acceder a la BD y, por tanto, se puede definir de tal manera para que solo sea accesible desde la dirección en la que este alojada la API.

Para la parte de la API se han utilizado los servicios de Amazon para generar una máquina virtual donde crear el entorno y poner en producción el programa para que sea accesible desde una IPv4 publica, es decir, que se pueda acceder mediante internet desde cualquier dispositivo externo. Mas concretamente, se ha utilizado el EC2 de AWS que te permite, mediante una imagen que elijas, arrancar una máquina virtual, configuras el grupo de seguridad para permitir el acceso a la IP publica y ya estaría la maquina lista para su funcionamiento. Para acceder a ella y poder montar la API, se generan un par de claves y mediante putty, ssh u otros que lo permitan, se puede realizar el acceso a la MV.

Una vez dentro, hay que descargar e instalar lo necesario como, en mi caso, node y git, una vez los tenemos, mediante git clonamos el repositorio donde tengamos el programa, instalamos todas las dependencias, construimos la imagen del programa y la ejecutamos.

Hay que tener en cuenta que cada vez que la MV se detenga la IP publica cambia, por lo que hay que actualizar la IP tanto en Atlas como en el navegador si queremos acceder a la API.

8. Resultados

El resultado final del proyecto es una API con acceso a través de internet, documentada con Swagger, por lo que permite identificar e interactuar de manera simple directa con cada uno de los endpoints sin necesidad de usar programas externos como Postman. Esta API, se conecta a una BD NoSQL en la nube solo accesible desde la propia API, que almacena todas las colecciones necesarias para poder dar soporte a un servicio de ratings tanto de asignaturas como

de los profesores que las imparten. La API tiene varios sistemas de seguridad, tanto para la protección de los usuarios como para la protección de los datos almacenados en la BD, estos son, para la parte de usuarios, una función hash a la hora de almacenar las contraseñas, la lista de los usuarios registrados solo es posible de obtener con el rol developer, por lo que la privacidad de los mismos es absoluta y, solo es posible obtener el usuario propio mediante id; Para la parte de la protección de los datos, solo son accesibles mediante una autentificación y teniendo los privilegios de acceso a datos vinculados a un rol asignado.

Por tanto, tenemos un backend robusto y seguro, fácil de utilizar que permitirá tantas interacciones como el servidor de Amazon y la nube de mongo permitan de manera simultánea.

9. Conclusión

Para mí, este trabajo, creo que ha sido muy beneficioso porque, aunque sí que es verdad que ya había utilizado NestJS con anterioridad, nunca lo había hecho con un proyecto nuevo en el que haya tenido que generar y desarrollar todo desde 0. Esto me ha forzado a pensar constantemente en como quiero que sea este servicio y como poder entregarlo de la mejor manera posible, cosa que ha influenciado por completo en el diseño final de la base de datos, en cómo hacer para asegurar los datos y que no todo el mundo pudiera acceder de manera libre y modificarlos y, al diseñarlo de esta manera, intentar hacer que, el código, quede lo más organizado posible ya que, con tanta cantidad de clases y métodos, es muy fácil desubicar los métodos y, encontrar un error en el código o, querer modificar un método para añadirle funcionalidad puede llegar a complicarse más de lo necesario.

La base de datos puede parecer la parte más liviana del proyecto o, por lo menos a mí me lo parecía, pero cada modificación en su estructura genera una ola de cambios en el programa, por ello, trabajar con MongoDB y un sistema de colecciones tan seccionado ha sido un gran acierto ya que, me ha permitido modificar cada una de las colecciones sin necesidad de modificar absolutamente nada de las otras.

A la parte de la API en sí, le veo muchísimo potencial ya que, aunque funciona y estoy contento con el resultado final, creo que podría llegar a ser una aplicación que cumpla con los objetivos del proyecto de manera real. Aún tiene margen para la mejora como, por ejemplo, en la autentificación, incorporar medidas más avanzadas como una autentificación en dos pasos, ya sea vía SMS o mail.

El mayor problema que encuentro que puede llegar a tener esta aplicación es el mantenimiento de los datos, ya que sí que es verdad que hay un rol administrador que en teoría se encarga de introducir y actualizar los datos con respecto a las instituciones y cada una de las titulaciones que tienen, pero si las instituciones no dedican un responsable a esta tarea, la carga de trabajo que genera la introducción de estos por parte de alguien externo es demasiado grande como para seguir con este modelo y, quizás, habría que cambiar a uno donde los propios usuarios introdujesen las asignaturas y profesores, aunque esto haría la información mucho menos veraz.

Me hubiera gustado, adicionalmente, haber podido programar un frontend funcional bien hecho con el framework de angular, por ejemplo, para explotar mucho mejor el potencial del backend, ya que esto permitiría visualizar mucho mejor las consultas y presentar los datos de una manera más dinámica y entendible para el usuario final ya que, aunque la API este documentada con Swagger, queda mucho más vistoso y agradable de interactuar con una interfaz preparada para mostrarte los datos, que teniendo que hacer, de manera manual, diversas consultas en las que, en vez de rellenar un recuadro, haya que crear un JSON, por ejemplo.

Bibliografía

[1] Amazon. (s.f.). *Amazon AWS*. Obtenido de https://aws.amazon.com/es/

[2] Barcelona, U. A. (s.f.). *Enquestes d'avaluació de l'actuació docent*. Obtenido de https://www.uab.cat/web/enquesta-avaluacio-docent-1345665543124.html

[3] Chakray. (2022). *Chakray*. Obtenido de https://www.chakray.com/es/swagger-y-swagger-ui-por-que-es-imprescindible-para-tus-apis/

[4] Esteso, M. P. (s.f.). *geekytheory*. Obtenido de https://geekytheory.com/que-es-una-api-rest-y-para-que-se-utiliza/

[5] Instituto Nacional de Estadistica. (2021). *Instituto Nacional de Estadistica*. Obtenido de https://www.ine.es/ss/Satellite?L=es_ES&c=INESeccion_C&cid=1259925529799&p=1254735110672&pagename=ProductosYServicios/PYSLayout#:~:text=El%2095%25%20de%20los%20hogares,de%20los%20hogares%20ten%C3%ADan%20Internet.

[6] Kanbanize. (s.f.). *Qué es Kanban: Definición, Características y Ventajas*. Obtenido de https://kanbanize.com/es/recursos-de-kanban/primeros-pasos/que-es-kanban

[7] Media, S. (s.f.). *Diagrama web*. Obtenido de https://app.diagrams.net/

[8] MongoDB. (s.f.). *MongoDB Atlas*. Obtenido de https://www.mongodb.com/es/atlas/database

[9] MongoDB. (s.f.). *MongoDB Manual*. Obtenido de https://www.mongodb.com/docs/manual/introduction/

[10] MongoDB. (s.f.). *Mongoosejs docs*. Obtenido de https://mongoosejs.com/docs/index.html

[11] Mysliwiec, K. (s.f.). *Documentation NestJS*. Obtenido de https://docs.nestjs.com/

[12] Nicholson, M. (2015). *passport-jwt*. Obtenido de https://www.passportjs.org/packages/passport-jwt/

[13] redhat. (s.f.). *what is a rest api*. Obtenido de https://www.redhat.com/es/topics/api/what-is-a-rest-api

[14] ShanghaiRanking. (s.f.). *ShanghaiRanking's Academic Ranking of World Universities Methodology 2021*. Obtenido de https://www.shanghairanking.com/methodology/arwu/2021

[15] Universitat de Barcelona. (s.f.). *Enquesta d'avaluació de l'actuació docent del professorat de la UAB*. Obtenido de https://www.uab.cat/doc/EnquestaAvaluacioActuacioD

ocent2018

[16] Universitat de Barcelona. (s.f.). *Enquesta de satisfacció d'assignatura de la UAB*. Obtenido de https://www.uab.cat/doc/QuestionariEnquestaAssignatures

[17] Varaksina, S. (18 de Febrero de 2022). *Mindstudios*. Obtenido de https://themindstudios.com/blog/web-app-development-trends/

APENDICE

A1. Ejemplo de documentación en Swagger

Como se puede observar en las siguientes figuras, la documentación realizada con Swagger es bastante visual y ordenada ya que organiza según las etiquetas todos los endpoints con su enrutamiento y, dependiendo de los métodos de petición HTTP les asigna un color, además, muestra todos los DTOs y, si accedemos a alguno de los métodos, podemos ver las posibles respuestas, los parámetros que hay que introducir ya sean mediante un JSON y/o un parámetro.

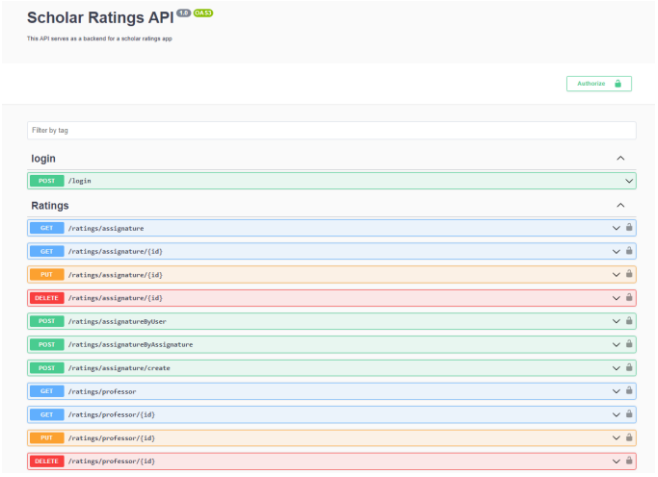

Fig. 5 Algunos endpoints Swagger

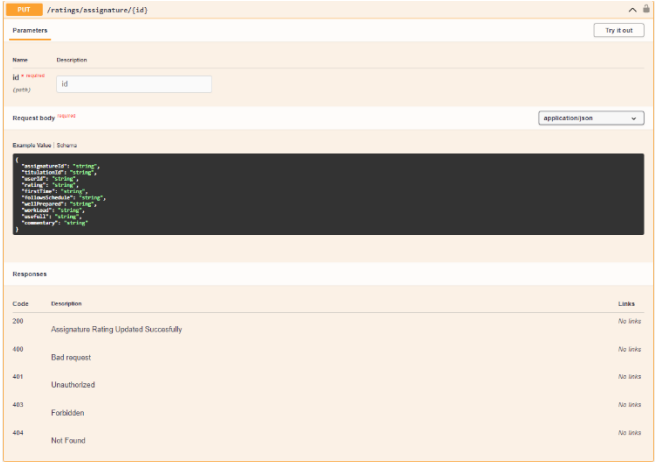

Fig. 6 Visualización de un endpoint y sus parámetros

A2. Autentificación y autorización

En las siguientes tres figuras podemos ver los 3 casos relacionados con la autentificación y la autorización, dos de ellas son respuestas de error y una de ellas es válida, la valida además de serlo, muestra el login que permite recibir el JWT y evitar las otras dos respuestas de error o, por lo menos la de autentificación, la de autorización dependerá de los permisos asignados a tu rol.

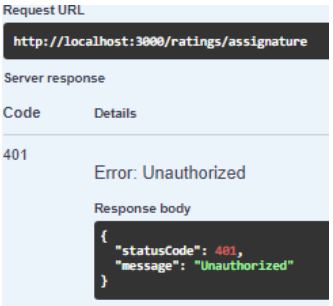

Fig. 7 Respuesta sin estar autentificado

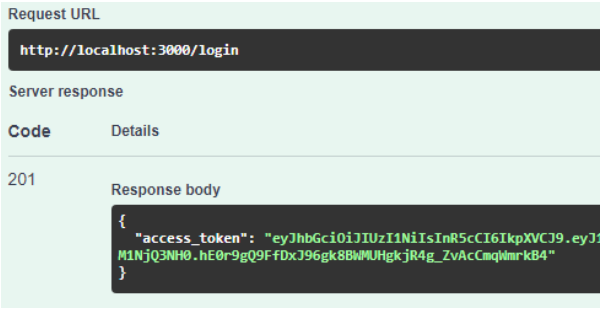

Fig. 8 Respuesta al autentificarte de manera correcta

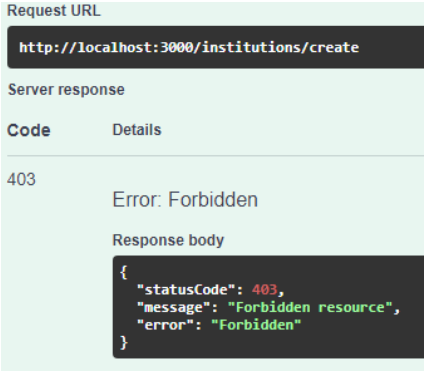

Fig. 9 Respuesta al estar autentificado, pero no autorizado

A3. Ejemplo de CRUD

Para este ejemplo, estando identificados con el rol de administrador, obtendremos la lista de instituciones actuales en la BD, crearemos una nueva institución, la modificaremos y la eliminaremos para mostrar los 4 tipos básicos de interacción con la BD.

1. Para empezar, al enviar la petición a la url …/institutions mediante el método *GET*, podemos ver que hay solo 2 instituciones en la BD.
2. Al enviar la petición a la url …/institutions/create mediante el método *POST* y, el JSON requerido, se almacena la institución en la BD.
3. Al enviar la petición a la url …/institutions mediante el método *PUT* junto con el JSON requerido y la id de Mongo del objeto ejemplo, se modifica el objeto que había en la BD con ese id.
4. Al enviar la petición a la url …/institutions mediante el método *DELETE* y el id del objeto ejemplo, se borra el objeto de la BD con ese id.

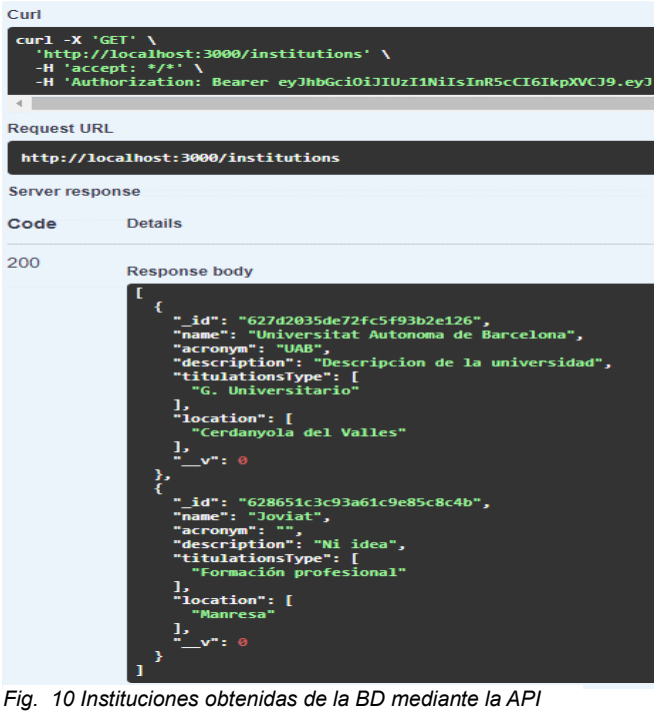

Fig. 10 Instituciones obtenidas de la BD mediante la API

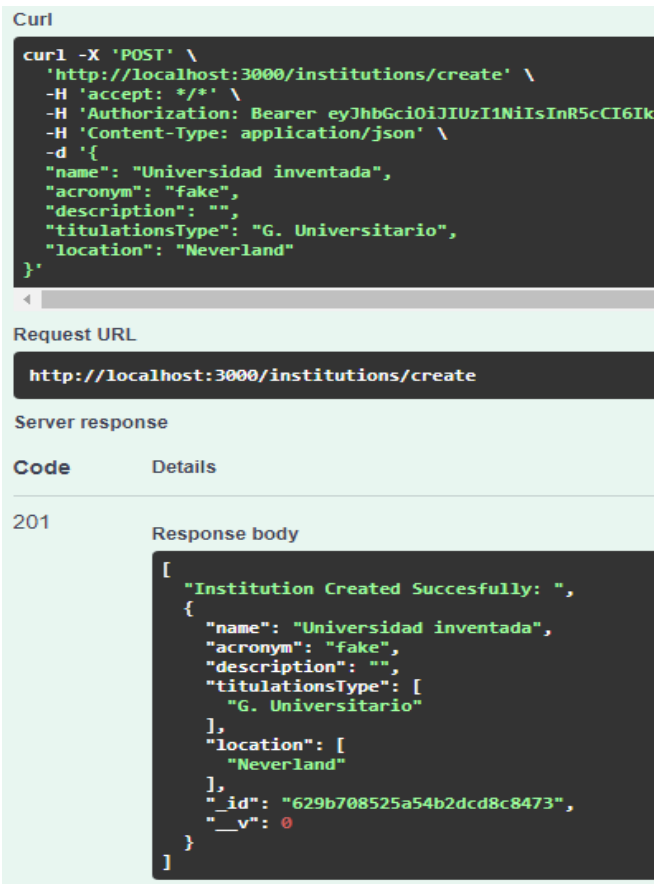

Fig. 11 Creación de una institución mediante la API

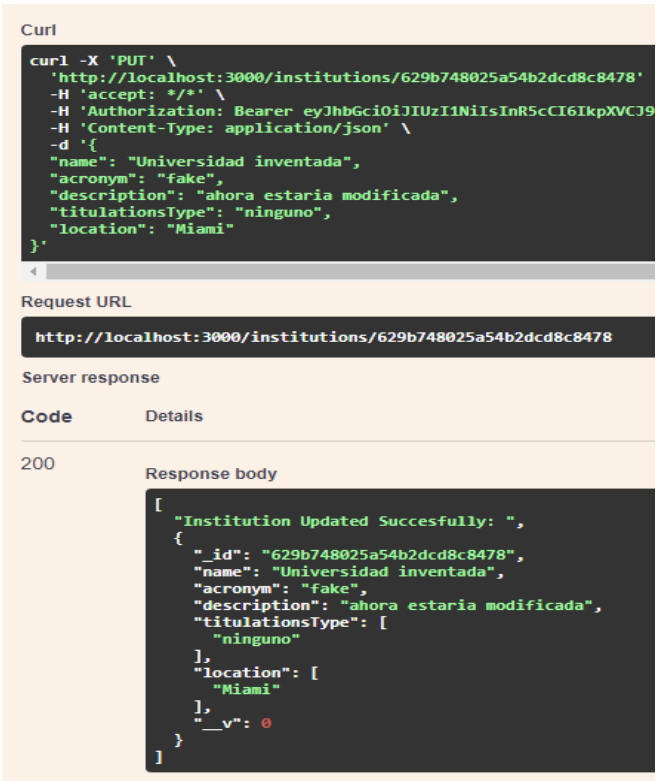

Fig. 12 Modificación de la institución falsa mediante la API

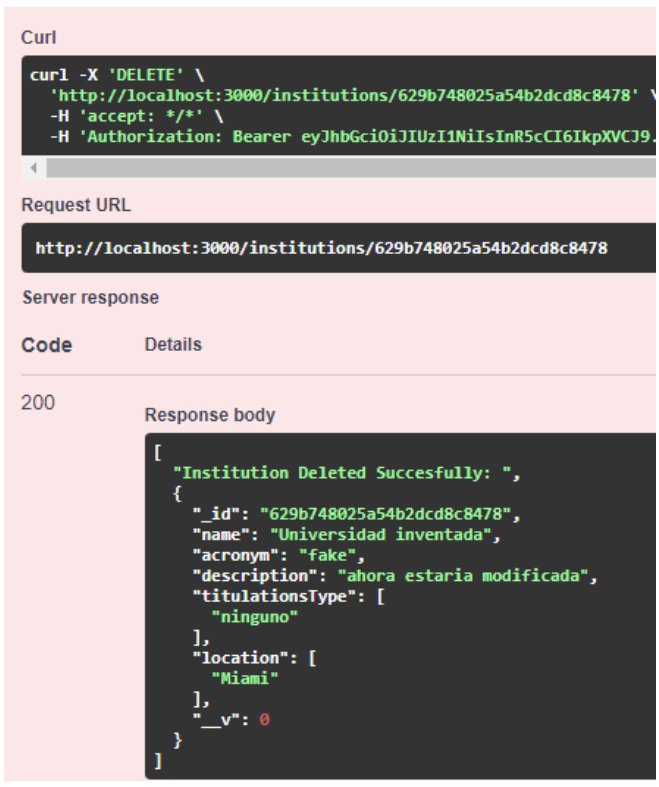

Fig. 13 Eliminación de la institución falsa mediante la API