Protección de APIs REST

Raúl Ruiz Barea

Máster Universitario en Ciberseguridad y Privacidad
Seguridad empresarial

Nombre Tutor/a de TF
Pau del Canto Rodrigo
Profesor/a responsable de la asignatura
Víctor Garcia Font

13/06/2023

Universitat Oberta de Catalunya

FICHA DEL TRABAJO FINAL

Título del trabajo:	*Protección de APIs REST*
Nombre del autor:	*Raúl Ruiz Barea*
Nombre del consultor/a:	*Pau del Canto Rodrigo*
Nombre del PRA:	*Víctor Garcia Font*
Fecha de entrega (06/2023):	*06/2023*
Titulación o programa:	*Máster Universitario en Ciberseguridad y privacidad*
Área del Trabajo Final:	*Seguridad empresarial*
Idioma del trabajo:	*Catalán, castellano o inglés*
Palabras clave	*API, REST, Gateway*

Resumen del Trabajo

El propósito de este proyecto fue implementar y segurizar un servicio API REST. El contexto de la aplicación era un escenario hipotético en el que una universidad necesitaba proporcionar acceso seguro a sus recursos a través de una API REST, al mismo tiempo que se protegía contra los riesgos de seguridad más comunes.

La metodología consistió en dividir el proyecto en diferentes fases, incluyendo la investigación teórica para obtener una compresión profunda de los conceptos de las API, las vulnerabilidades más comunes y las mejores prácticas para protegerlas. Planificación, implementación y evaluación de la solución propuesta garantizando los requisitos y objetivos del proyecto.

Para mitigar los riesgos de seguridad, se identificaron e implementaron las mejores prácticas, incluidas la autenticación y autorización, el cifrado SSL/TLS y el control de acceso. También se implementó un API Gateway para centralizar el control de acceso y el número de llamadas, y el servicio se implementó utilizando Docker.

Los resultados fueron un servicio API REST seguro y confiable que proporcionó acceso controlado a los recursos mientras protegía contra los riesgos de seguridad más comunes. El uso de una metodología ágil permitió flexibilidad y adaptabilidad a lo largo del proceso de desarrollo, lo que resultó en un resultado exitoso del proyecto.

En conclusión, este proyecto demostró la eficacia del uso de una metodología ágil para gestionar el desarrollo de un servicio API REST seguro y la importancia de implementar las mejores prácticas de seguridad para protegerse contra los riesgos de seguridad comunes.

Abstract

The purpose of this project was to implement and secure an API REST service. The context of application was a hypothetical scenario where an organization needed to provide secure access to its resources through an API, while also protecting against common security risks.

The methodology consisted of dividing the project into different phases, including theoretical research to obtain a deep understanding of API concepts, the most common vulnerabilities, and the best practices to protect them. Planning, implementation, and evaluation of the proposed solution guaranteeing the requirements and objectives of the project.

To mitigate security risks, best practices were identified and implemented, including authentication and authorization, SSL/TLS encryption, and access control. An API Gateway was also implemented to centralize access control and rate limiting, and the service was deployed using Docker.

The results were a secure and reliable API REST service that provided controlled access to resources while protecting against common security risks. The use of agile methodology allowed for flexibility and adaptability throughout the development process, resulting in a successful project outcome.

In conclusion, this project demonstrated the effectiveness of using agile methodology to manage the development of a secure API REST service, and the importance of implementing security best practices to protect against common security risks.

Índice

Lista de figuras

1. Introducción

1.1. Contexto y justificación del Trabajo

A medida que las empresas confían cada vez más en la tecnología digital, las API (Application Programming Interfaces) se han convertido en un componente fundamental para permitir la comunicación y el intercambio de datos entre sistemas. Las API proporcionan una interfaz para que diferentes aplicaciones de software se comuniquen entre sí, lo que facilita la integración de sistemas y el intercambio de datos. Sin embargo, las API también pueden presentar importantes riesgos de seguridad si no están debidamente protegidas.

Los piratas informáticos pueden aprovechar las vulnerabilidades de las API para obtener acceso a datos confidenciales, lanzar ataques o interrumpir servicios. Los riesgos de seguridad comunes asociados con las API incluyen acceso no autorizado, ataques de denegación de servicio y ataques de inyección. Estos riesgos pueden resultar en pérdidas financieras significativas, daños a la reputación y responsabilidades legales.

Por lo tanto, existe la necesidad de desarrollar API seguras que puedan proteger contra estos riesgos. Esto requiere una comprensión profunda de los riesgos asociados con las API y la implementación de las mejores prácticas para protegerlas. La seguridad es especialmente crítica para las organizaciones que manejan datos confidenciales, como instituciones financieras, proveedores de atención médica y agencias gubernamentales.

Aplicar protección a las APIs también es importante para el cumplimiento de las normas y estándares de protección de datos, como el Reglamento general de protección de datos (GDPR). Estas regulaciones requieren que las organizaciones protejan la confidencialidad, integridad y disponibilidad de los datos, incluidos los datos intercambiados a través de estas.

La necesidad de APIs REST surgió de la complejidad de construir e integrar sistemas distribuidos a través de Internet. Antes del desarrollo de las API REST, los sistemas distribuidos a menudo se creaban utilizando protocolos como SOAP (Simple Object Access Protocol), que eran complejos, pesados y difíciles de manejar.

Las API REST se diseñaron para ser simples y fáciles de usar. Utilizan métodos HTTP estándar y códigos de estado, lo que los hace fáciles de entender y trabajar con ellas, además de ser diseñadas para ser escalables y flexibles.

Destacan por independientes a la plataforma y pueden ser utilizadas por clientes en cualquier plataforma o dispositivo que pueda enviar y recibir solicitudes HTTP, a nivel de rendimiento son livianas, haciéndolas rápidas y eficientes.

En resumen, el **contexto** de este proyecto es la necesidad de desarrollar una API REST de forma segura para proteger contra los riesgos de seguridad comunes y cumplir con las normas y estándares de protección de datos.

La **justificación** de la realización de este proyecto es mejorar en el conocimiento y habilidades de la segurización de las API REST en relación del proyecto actual de la organización en la cual se colabora. El proyecto ayudará a la organización a proteger sus datos y servicios contra los riesgos de seguridad comunes asociados con las API, además de ayudar a cumplir con las normas y estándares de protección de datos.

Finalmente, la tecnología que actualmente utiliza la organización para la realización de APIs REST y API Gateway es: **NestJS**, debido a las ganas de aprender sobre esta tecnología, y, sobre todo, la necesidad de profundizar en el ámbito de seguridad.

1.2. Objetivos del Trabajo

El **objetivo principal** de este proyecto es profundizar en el conocimiento de las APIs REST e implementar una API REST y un API Gateway de manera segura.

Para la consecución de este objetivo principal abordaremos los siguientes **objetivos parciales**:

- Identificación de los riesgos de seguridad más comunes proporcionadas por OWASP en las API REST y evaluar medidas de seguridad para protegerlas.

- Investigación y evaluación de diferentes medidas de seguridad para las API REST.

- Implementación de una API REST segura con medidas de seguridad adecuadas y aplicar buenas prácticas de diseño y tecnología.

- Implementación de un API Gateway para administrar y proteger la API REST.

- Despliegue de la API REST y API Gateway en un entorno en contenedores para asegurar la implementación y escalabilidad.

A medida que las empresas se vuelven más dependientes de las API, los riesgos de seguridad asociados con ellas también aumentan. Es esencial comprender estos riesgos y desarrollarlas de manera segura para protegerse contra ellos.

1.3. Impacto en sostenibilidad, ético-social y de diversidad

Este trabajo se desarrolla en los siguientes Objetivos de Desarrollo Sostenible (ODS) de las tres dimensiones de la Competencia de compromiso ético y global (CCEG).

Sostenibilidad:

- ODS 9 - Industry, innovation and infrastructure: al promover el desarrollo de servicios API REST seguros y confiables, el proyecto puede contribuir al desarrollo de una infraestructura resiliente que promueva la industrialización sostenible y la innovación.

- ODS 12 - Responsible consumption and production: mediante la implementación de medidas de seguridad que protegen los datos confidenciales y garantizan el cumplimiento de las normas y estándares de protección de datos pertinentes, el proyecto puede contribuir al consumo y la producción responsables mediante la promoción del uso sostenible y la protección de los recursos.

Comportamiento Ético y Responsabilidad Social:

- ODS 16 - Peace, justice and strong institutions: al implementar medidas de seguridad, como autenticación y autorización, encriptación SSL/TLS y control de acceso, el proyecto puede contribuir a la creación de instituciones sólidas que defiendan el estado de derecho y promuevan la paz y la justicia.

Diversidad y derechos humanos:

- ODS 12 - Responsible consumption and production: mediante la implementación de medidas de seguridad que protegen datos sensibles, el proyecto puede contribuir a la protección de datos personales y la promoción de la igualdad de género.

- ODS 10 - Reduced inequalities: al promover el desarrollo y el uso de servicios API REST seguros, el proyecto puede contribuir a la reducción de las desigualdades promoviendo la igualdad de acceso a los datos y a la tecnología.

- ODS 16 - Peace, justice and strong institutions: a través de la implementación de fuertes medidas de seguridad, el proyecto puede contribuir a la promoción y protección de los derechos humanos, como el derecho a la privacidad y el derecho al acceso a la información.

Se han identificado los siguientes impactos positivos:

- Seguridad: Los datos sensibles estén protegidos, promoviendo los principios de confidencialidad, integridad y disponibilidad.

- Compliance: Ayudando a las empresas a cumplir con las normas y estándares de protección de datos relevantes, promoviendo el compromiso ético y el consumo responsable.

- Innovación: Compartiendo los datos de manera segura y eficiente, promoviendo el crecimiento económico y el desarrollo sostenible.

Y los siguientes impactos negativos:

- Coste: La implementación de medidas de seguridad, como el cifrado SSL/TLS, las claves de API o la autenticación OAuth2, pueden aumentar el coste del desarrollo y mantenimiento de las APIs.

- Complejidad: La implementación de medidas de seguridad puede aumentar la complejidad de las APIs, afectando potencialmente la usabilidad y la experiencia del usuario.

- Barrera de entrada: Las empresas que pueden no tener los recursos o la experiencia para desarrollar API seguras.

En general, se puede contribuir a la Competencia de compromiso ético y global (CCEG) y a los Objetivos de Desarrollo Sostenible (ODS) al promover la seguridad, el cumplimiento, la innovación y la colaboración. Si bien puede haber algunos impactos negativos asociados con la implementación de medidas de seguridad, estos pueden gestionarse equilibrando la necesidad de seguridad con la necesidad de usabilidad, accesibilidad y rentabilidad.

1.4. Enfoque y método seguido

La estrategia para llevar a cabo el trabajo será producir un nuevo producto y para poderlo llevar a cabo debemos dividir en distintas fases para ir consiguiendo todos los objetivos.

- Investigación teórica para tener un conocimiento profundo de los conceptos y principios básicos de API, las vulnerabilidades más comunes y las mejores prácticas para protegerlas.

- Planificación de la implementación de la API REST y API Gateway para identificar los mejores patrones de diseño y arquitectura.

- Implementación de la API REST.

- Implementación de medidas de seguridad de la API.

- Implementación del API Gateway.

- Despliegue de la API REST y API Gateway.

- Evaluación de la solución propuesta para asegurar que cumple con los requisitos y objetivos del proyecto.

Para la parte de investigación se hará una lectura de distintas fuentes de información, entre ellas:

- Sobre APIs: “Mastering API Architecture” de James Gough, Daniel Bryant & Matthew Auburn [1].

- Sobre seguridad en APIs: OWASP API Security Project y directrices de NIST sobre desarrollo de API seguro [2].

Y para afrontar la parte de implementación:

- Sobre NestJS: Documentación oficial y curso de Udemy “NestJS The complete developers guide” [7].

- Sobre arquitectura: “Head First Design Patterns” de Eric Freeman & Elisabeth Robson [12].

- Sobre testing: A través de la documentación de las extensiones Thunder Client [13] y REST Client [14] de Visual Studio Code.

Para resolver el problema, el proyecto utilizará las mejores prácticas para proteger una API REST, como autenticación y autorización, cifrado SSL/TLS y control de acceso. También se implementará un API Gateway para centralizar el control de acceso y aplicar rate limiting. A nivel de infraestructura, se utilizará Docker para poder facilitar la implementación y escalabilidad.

Para realizar la evaluación y validar que logramos los resultados esperados, utilizaremos las siguientes estrategias:

- Realización de pruebas de penetración y evaluaciones de vulnerabilidades para identificar posibles vulnerabilidades o debilidades de seguridad en el servicio API REST y API Gateway. Esto ayudará a garantizar que el cifrado SSL/TLS, las claves API o la autenticación OAuth2 y la limitación de velocidad funcionen de manera efectiva para evitar ataques DDoS.

- Verificación del cifrado SSL/TLS esté correctamente configurado e implementado mediante la validación del certificado SSL/TLS. Esto ayudará a garantizar que el servicio API REST y API Gateway utilicen un certificado SSL/TLS de confianza y que el cifrado funcione correctamente.

- Realización de pruebas de carga para evaluar la efectividad de la limitación de velocidad implementada para prevenir ataques DDoS. Esto ayudará a garantizar que el servicio API REST y API Gateway puedan manejar el volumen esperado de tráfico y solicitudes sin experimentar problemas de rendimiento o tiempo de inactividad.

Siguiendo el enfoque propuesto se mantiene el objetivo durante todo el proceso de desarrollo, desde la investigación hasta la implementación. Basándonos en las mejores prácticas, incorporando recursos tanto teóricos como prácticos y enfatizando en la implementación y la evaluación a través de las pruebas indicadas.

1.5. Estudio del arte

Simplificando, una API es una forma en que dos sistemas de software diferentes se comunican entre sí, mientras que un API REST es un tipo de API que sigue un conjunto de estándares para la creación de servicios web.

Figura 1: API styles over time. Fuente: Rob Crowley.

Actualmente las API REST son muy populares por su flexibilidad, escalabilidad y simplicidad.

A través del artículo del estado de APIs realizado por RapidAPI [15] podemos observar que destacan que las API se han convertido en una parte crítica del ecosistema tecnológico, impulsando muchas aplicaciones y servicios. También señala que las API REST se han convertido en el tipo de API más popular, con más del 80% de todas las API utilizando esta arquitectura.

Un informe reciente de RapidAPI, líder en el mercado sobre APIs, descubrió que las API REST son el tipo de API más popular entre los desarrolladores, seguidas de SOAP y GraphQL. El informe también encontró que los marcos de desarrollo más populares para crear API REST son Node.js, Java Spring y Ruby on Rails.

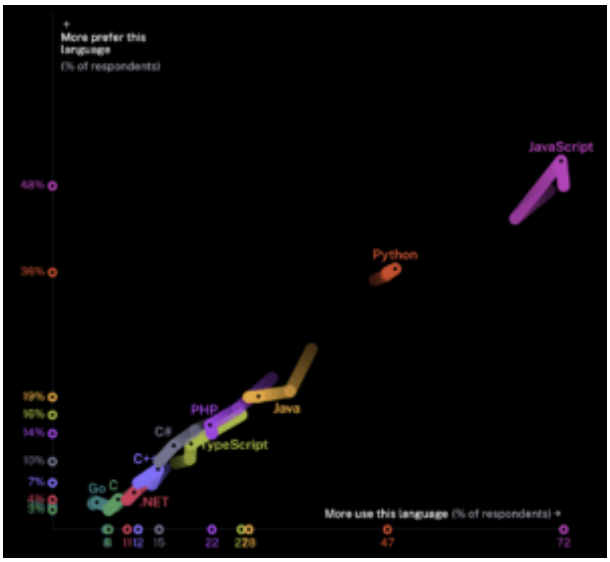


Figura 2: Lenguajes más populares en desarrollo. Fuente: RapidAPI.

Otro estudio realizado por SmartBear, una empresa de pruebas de software, descubrió que las API REST son ahora el tipo de API más común utilizado para el desarrollo de aplicaciones móviles y representan más del 80 % de todas las API utilizadas en este contexto [27].

Cómo observamos JavaScript es uno de los lenguajes más populares en desarrollo, en este caso el framework utilizado es NestJS ya que es un framework basado en Node.js que ofrece varias funciones para facilitar el desarrollo de APIs REST y API Gateways. Estas son algunas de las características principales de NestJS que pueden ayudar con el desarrollo de API:

- TypeScript brindando seguridad en tipos y detección de errores antes de que se conviertan en problemas de tiempo de ejecución.
- La arquitectura modular facilita la estructuración de aplicaciones a gran escala, incluyendo API Gateways.
- El sistema de inyección de dependencias reduce la cantidad de código repetitivo necesario para administrar las dependencias en las API REST y las API Gateway.
- El sistema de middleware nos permite a los desarrolladores agregar funciones personalizadas al ciclo de solicitud-respuesta de una aplicación, lo que facilita el manejo de problemas transversales como la autenticación y la limitación de velocidad.
- Compatibilidad integrada con OpenAPI, lo que simplifica el desarrollo de las API REST al proporcionar una forma estandarizada de documentar las API y generar código de cliente.

Por último, introducimos el concepto de API Gateway, los API Gateway son un componente de las arquitecturas distribuidas modernas que actúa como intermediario entre los clientes y los servicios de back-end, proporcionando un punto central de control para todo el tráfico API entrante y saliente.

Sus características principales incluyen enrutamiento, balanceo de carga, seguridad, almacenamiento en caché y monitoreo y análisis.

Las soluciones más populares para API Gateway incluyen NGINX, Kong, Amazon API Gateway, Apigee y NestJS [28].

1.6. Planificación del Trabajo

Teniendo como objetivo principal la investigación, implementación y evaluación de una solución segura API REST y API Gateway, necesitamos dividir el trabajo en tareas manejables y se ha creado una línea de tiempo para el proyecto siguiendo las fechas propuestas por la universidad.

Por supuesto, las tareas específicas y el cronograma pueden variar según la complejidad de la tarea y la experiencia sobre el área. También es importante el manejo de buffers para resolver problemas o retrasos imprevistos.

	Name	Duration	Start	Finish	Predecessors
1	**TFM: Protección de APIs REST**	**80 days?**	**3/1/23 8:00 AM**	**6/20/23 5:00 PM**	
2	**Plan de trabajo**	**10 days?**	**3/1/23 8:00 AM**	**3/14/23 5:00 PM**	
3	Contexto y justificación del Trabajo	2 days?	3/1/23 8:00 AM	3/2/23 5:00 PM	
4	Objetivos del Trabajo	1 day?	3/3/23 8:00 AM	3/3/23 5:00 PM	3
5	Impacto en sostenibilidad, ético-social y de diversidad	1 day?	3/6/23 8:00 AM	3/6/23 5:00 PM	4
6	Enfoque y método seguido	1 day?	3/7/23 8:00 AM	3/7/23 5:00 PM	5
7	Estudio del arte	1 day?	3/8/23 8:00 AM	3/8/23 5:00 PM	6
8	Planificación del trabajo	1 day?	3/9/23 8:00 AM	3/9/23 5:00 PM	7
9	Breve sumario de productos obtenidos	1 day?	3/10/23 8:00 AM	3/10/23 5:00 PM	8
10	Breve descripción de los otros capítulos de la memoria	1 day?	3/13/23 8:00 AM	3/13/23 5:00 PM	9
11	Revisión	1 day?	3/14/23 8:00 AM	3/14/23 5:00 PM	10
12	Entrega 1	0 days	3/14/23 5:00 PM	3/14/23 5:00 PM	11
13	**Investigación**	**20 days?**	**3/15/23 8:00 AM**	**4/11/23 5:00 PM**	**2**
14	Análisis	16 days?	3/15/23 8:00 AM	4/5/23 5:00 PM	
15	Planificación	3 days?	4/6/23 8:00 AM	4/10/23 5:00 PM	14
16	Revisión	1 day?	4/11/23 8:00 AM	4/11/23 5:00 PM	15
17	Entrega 2	0 days	4/11/23 5:00 PM	4/11/23 5:00 PM	16
18	**Implementación**	**20 days?**	**4/12/23 8:00 AM**	**5/9/23 5:00 PM**	**13**
19	Implementación API REST	10 days?	4/12/23 8:00 AM	4/25/23 5:00 PM	
20	Implementación API Gateway	7 days?	4/26/23 8:00 AM	5/4/23 5:00 PM	19
21	Despliegue	2 days?	5/5/23 8:00 AM	5/8/23 5:00 PM	20
22	Revisión	1 day?	5/9/23 8:00 AM	5/9/23 5:00 PM	21
23	Entrega 3	0 days	5/9/23 5:00 PM	5/9/23 5:00 PM	22
24	**Evaluación y memoria**	**25 days?**	**5/10/23 8:00 AM**	**6/13/23 5:00 PM**	**18**
25	Evaluación	8 days?	5/10/23 8:00 AM	5/19/23 5:00 PM	
26	Redacción	8 days?	5/22/23 8:00 AM	5/31/23 5:00 PM	25
27	Conclusiones y trabajos futuros	3 days?	6/1/23 8:00 AM	6/5/23 5:00 PM	26
28	Glosario	1 day?	6/6/23 8:00 AM	6/6/23 5:00 PM	27
29	Bibliografía	1 day?	6/7/23 8:00 AM	6/7/23 5:00 PM	28
30	Anexos	2 days?	6/8/23 8:00 AM	6/9/23 5:00 PM	29
31	Revisión	2 days?	6/12/23 8:00 AM	6/13/23 5:00 PM	30
32	Entrega 4	0 days	6/13/23 5:00 PM	6/13/23 5:00 PM	31
33	**Presentación en vídeo**	**5 days?**	**6/14/23 8:00 AM**	**6/20/23 5:00 PM**	**24**
34	Preparación PPT	2 days?	6/14/23 8:00 AM	6/15/23 5:00 PM	
35	Preparación vídeo	2 days?	6/16/23 8:00 AM	6/19/23 5:00 PM	34
36	Revisión	1 day?	6/20/23 8:00 AM	6/20/23 5:00 PM	35
37	Entrega 5	0 days	6/20/23 5:00 PM	6/20/23 5:00 PM	36

Figura 3: Planificación del trabajo: Timeline y dependencias.

	Name	Duration	Start	Finish	Predece
1	**TFM: Protección de APIs REST**	**80 days?**	**3/1/23 8:00 AM**	**6/20/23 5:00 PM**	
2	**Plan de trabajo**	**10 days?**	**3/1/23 8:00 AM**	**3/14/23 5:00 PM**	
3	Contexto y justificación del Trabajo	2 days?	3/1/23 8:00 AM	3/2/23 5:00 PM	
4	Objetivos del Trabajo	1 day?	3/3/23 8:00 AM	3/3/23 5:00 PM	3
5	Impacto en sostenibilidad, ético-social y de diversidad	1 day?	3/6/23 8:00 AM	3/6/23 5:00 PM	4
6	Enfoque y método seguido	1 day?	3/7/23 8:00 AM	3/7/23 5:00 PM	5
7	Estudio del arte	1 day?	3/8/23 8:00 AM	3/8/23 5:00 PM	6
8	Planificación del trabajo	1 day?	3/9/23 8:00 AM	3/9/23 5:00 PM	7
9	Breve sumario de productos obtenidos	1 day?	3/10/23 8:00 AM	3/10/23 5:00 PM	8
10	Breve descripción de los otros capítulos de la memoria	1 day?	3/13/23 8:00 AM	3/13/23 5:00 PM	9
11	Revisión	1 day?	3/14/23 8:00 AM	3/14/23 5:00 PM	10
12	Entrega 1	0 days	3/14/23 5:00 PM	3/14/23 5:00 PM	11
13	**Investigación**	**20 days?**	**3/15/23 8:00 AM**	**4/11/23 5:00 PM**	**2**
14	Análisis	16 days?	3/15/23 8:00 AM	4/5/23 5:00 PM	
15	Planificación	3 days?	4/6/23 8:00 AM	4/10/23 5:00 PM	14
16	Revisión	1 day?	4/11/23 8:00 AM	4/11/23 5:00 PM	15
17	Entrega 2	0 days	4/11/23 5:00 PM	4/11/23 5:00 PM	16
18	**Implementación**	**20 days?**	**4/12/23 8:00 AM**	**5/9/23 5:00 PM**	**13**
19	Implementación API REST	10 days?	4/12/23 8:00 AM	4/25/23 5:00 PM	
20	Implementación API Gateway	7 days?	4/26/23 8:00 AM	5/4/23 5:00 PM	19
21	Despliegue	2 days?	5/5/23 8:00 AM	5/8/23 5:00 PM	20
22	Revisión	1 day?	5/9/23 8:00 AM	5/9/23 5:00 PM	21
23	Entrega 3	0 days	5/9/23 5:00 PM	5/9/23 5:00 PM	22
24	**Evaluación y memoria**	**25 days?**	**5/10/23 8:00 AM**	**6/13/23 5:00 PM**	**18**
25	Evaluación	8 days?	5/10/23 8:00 AM	5/19/23 5:00 PM	
26	Redacción	8 days?	5/22/23 8:00 AM	5/31/23 5:00 PM	25
27	Conclusiones y trabajos futuros	3 days?	6/1/23 8:00 AM	6/5/23 5:00 PM	26
28	Glosario	1 day?	6/6/23 8:00 AM	6/6/23 5:00 PM	27
29	Bibliografía	1 day?	6/7/23 8:00 AM	6/7/23 5:00 PM	28
30	Anexos	2 days?	6/8/23 8:00 AM	6/9/23 5:00 PM	29
31	Revisión	2 days?	6/12/23 8:00 AM	6/13/23 5:00 PM	30
32	Entrega 4	0 days	6/13/23 5:00 PM	6/13/23 5:00 PM	31
33	**Presentación en vídeo**	**5 days?**	**6/14/23 8:00 AM**	**6/20/23 5:00 PM**	**24**
34	Preparación PPT	2 days?	6/14/23 8:00 AM	6/15/23 5:00 PM	
35	Preparación vídeo	2 days?	6/16/23 8:00 AM	6/19/23 5:00 PM	34
36	Revisión	1 day?	6/20/23 8:00 AM	6/20/23 5:00 PM	35
37	Entrega 5	0 days	6/20/23 5:00 PM	6/20/23 5:00 PM	36

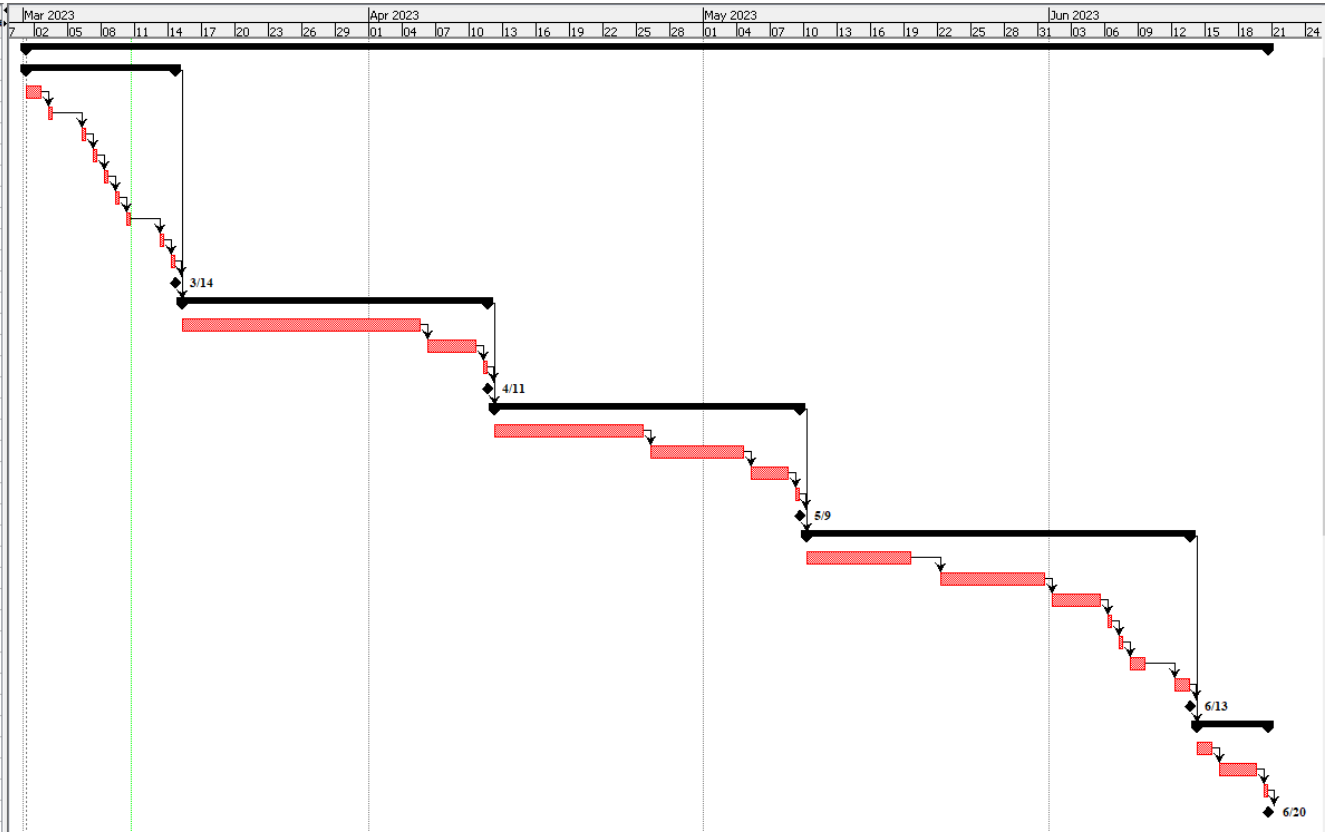


Figura 4: Planificación del trabajo: Gantt.

1.7. Breve sumario de productos obtenidos

Informe de investigación teórica que profundiza en los conceptos y principios básicos de API, las vulnerabilidades más comunes y las mejores prácticas para protegerlas.

Una API REST implementada con medidas de seguridad adecuadas, utilizando el framework de NestJS y las mejores prácticas de diseño y tecnología.

API Gateway implementado para administrar y proteger la API REST.

Solución desplegada en un entorno de contenedores que asegura la implementación y escalabilidad de la API REST y API Gateway.

Documentación completa que describe la arquitectura, las tecnologías utilizadas, los procedimientos de despliegue y las medidas de seguridad implementadas.

Evaluación de la solución propuesta para asegurar que cumple con los requisitos y objetivos del proyecto.

Pruebas de unidad y pruebas de integración para garantizar el correcto funcionamiento de la API REST y el API Gateway.

Código fuente completo de la API REST y el API Gateway, que se puede utilizar como punto de partida para futuros proyectos de API.

2. Investigación

2.1 Conceptos y principios básicos de API

Una API, o Application Programming Interface, es un conjunto de reglas y protocolos que permiten que dos aplicaciones de software se comuniquen entre sí. Mediante el uso de una API, los desarrolladores pueden acceder y utilizar la funcionalidad de otra aplicación sin tener conocimiento del código o la infraestructura detrás de estas.

Las API funcionan al exponer un conjunto de endpoints que los desarrolladores o servicios puedan usar para enviar solicitudes (request) y recibir respuestas (response). Debemos tener en cuenta que estos pueden ser asíncronos y según el tráfico y operaciones que realicen tendremos una respuesta más o menos rápida. Estos endpoints están diseñados para lanzar acciones o recursos específicos, como, por ejemplo: CRUDs (Create, Read, Update y Delete) o la ejecución de funciones específicas.

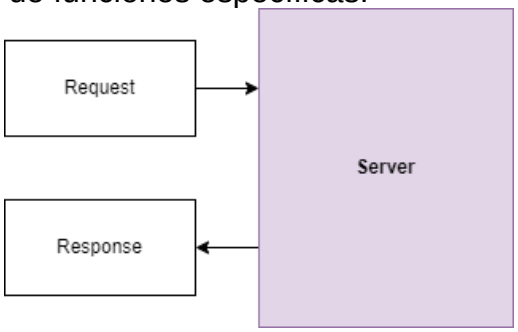


Figura 5: Diagrama: API Petición-Respuesta.

Desde el punto de vista de la seguridad, es importante que el diseño y la arquitectura de la API tenga en cuenta que se utilizan las mejores prácticas, entre estas encontramos la validación de los parámetros de entrada, la encriptación para contraseñas o información sensible, la preparación ante ataques masivos, la identificación y/o el seguimiento de errores o causas raíz de estos, etcétera.

2.2 Conceptos y principios básicos de REST

Una API REST es un tipo de API web que se basa en unos principios que pautan la creación, el mantenimiento y la escalabilidad.

Para no extendernos demasiado sobre este tema, al no ser el propósito de este trabajo la investigación y validación de estos principios, me gustaría indicar algunos ejemplos que podemos encontrar:

- Taxonomía: Cómo gestionar los errores, fallos, …
- Consistencia: Estructuras y longitud de URLs, métodos soportados (HTTP Request/Response), cabeceras, …
- Seguridad: CORS, …
- Versionado: Formatos, cuando versionar, …
- ¡Y mucho más…!

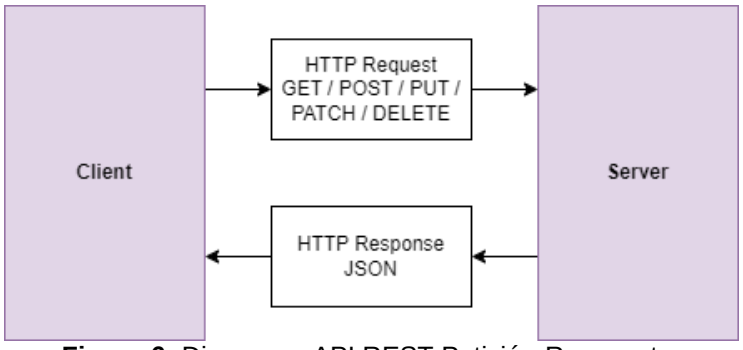


Figura 6: Diagrama: API REST Petición-Respuesta.

Debemos ser agnósticos a todos los niveles, ya que un cliente podría ser móvil, web, 3rd party, además de la implementación, ya que podríamos utilizar un ORM en concreto, y podríamos según las necesidades cambiar de cliente de base de datos cuando fuera necesario.

Estos principios pueden ser adaptados al contexto de las organizaciones mientras cumplan con su cometido, unas guías muy estandarizadas son las que proponen Microsoft REST API Guidelines [16].

Nos remarcan que siguiendo dichas pautas conseguiremos que las API REST sean consistentes, fáciles e intuitivas.

Method	Description	Is Idempotent
GET	Return the current value of an object	True
PUT	Replace an object, or create a named object, when applicable	True
DELETE	Delete an object	True
POST	Create a new object based on the data provided, or submit a command	False
HEAD	Return metadata of an object for a GET response. Resources that support the GET method MAY support the HEAD method as well	True
PATCH	Apply a partial update to an object	False
OPTIONS	Get information about a request; see below for details.	True

Figura 7: Métodos soportados. Fuente: Microsoft REST API Guidelines [16].

Revisando los métodos soportados aparece un concepto muy interesante que también tiene que ver con la seguridad de nuestros sistemas de software, y es la idempotencia.

La idempotencia en este contexto se refiere a cuantas veces se puede repetir una operación debido a un error o fallo sin afectar al resultado. Por ejemplo, en una empresa de venta de productos y su stock, si realizas una compra, si fallara el proceso no queremos que el stock de producto baje más de la cuenta, otro ejemplo sería realizar una compra y descontar el dinero varias veces.

Se podrían convertir en idempotentes identificando las request añadiendo una clave en las cabeceras, cacheándolas y evaluando si la petición ya se realizó.

2.3 Gestión de las APIs

Para poder desarrollar APIs REST de forma estándar y segura, debemos planificar como vamos a realizar la gestión y el mantenimiento de estas. Esto incluye todas las fases del ciclo de desarrollo de aplicaciones seguras, desde la fase de ideación, implementación, despliegue y mantenimiento, hasta el soporte.

Entendiendo bien el modelo de negocio construiremos una arquitectura de software que encaje con las necesidades de los clientes o compañía.

2.4 Diseño y documentación Swagger y OpenAPI

Todo empezó en 2010 con el proyecto de código abierto para implementar y visualizar gracias a la especificación del API, fue adquirido por SmartBear Software en 2015 y se donó a la iniciativa de OpenAPI, ya renombrando la especificación Swagger a especificación OpenAPI.

OpenAPI es una especificación de código abierto para describir y documentar las API REST. Se pueden describir las APIs de forma agnóstica al código, proporcionando endpoints, formatos de solicitud y respuesta y métodos de autenticación, etcétera.

Nos ayuda a mejorar el diseño y desarrollo de las API al promover la coherencia, la claridad y la interoperabilidad entre los proveedores y consumidores.

Nos permite proporcionar una especificación en formato JSON o YAML, el cual se podría trabajar con herramientas de terceros.

Al ser universal podemos obtener muchos beneficios a la hora de trabajar con esta especificación, desde generar código servidor y cliente, testear los endpoints, etcétera.

Gracias al editor online de swagger [17] podemos generar código servidor y cliente:

Figura 8: Swagger Editor, generación de cliente-servidor.

En este proyecto se seguirá la documentación de NestJS [6] sobre OpenAPI, la cual instalando la librería adecuada y utilizando los decoradores adecuados producirán la especificación automáticamente, cuando se genera el documento,

añadiremos todos los datos necesarios, incluida la versión del api la cual formará parte del path, y este, se puede utilizar para generar un archivo en formato YAML.

```
const options = new DocumentBuilder()
  .setTitle('NestJS Security REST API')
  .setDescription('Open University of Catalonia')
  .setVersion('1.0')
  .addBearerAuth(
    {
      type: 'http',
      scheme: 'Bearer',
      bearerFormat: 'Bearer',
      name: 'JWT',
      description: 'Enter JWT token',
      in: 'header',
    },
    'access-token',
  )
  .addServer(`${schema}://${host}:${port}`)
  .build();

const document = SwaggerModule.createDocument(app, options);
fs.writeFileSync('./open-api.yaml', yaml.stringify(document, {}));

SwaggerModule.setup('api', app, document);
```

Figura 9: Configuración y documentación especificación API.

2.5 Conceptos y principios básicos de API Gateway

Un API Gateway es una puerta de entrada a distintas APIs REST, gestiona las peticiones y las enruta al servicio adecuado. Al ser el único punto de entrada, se encarga de manejar todo el tráfico y las políticas de seguridad que establezcamos, aportándonos beneficios en seguridad y escalabilidad.

En este proyecto se implementará un API Gateway personalizado utilizando NestJS, durante la implementación documentaremos como se abordan las distintas medidas de seguridad.

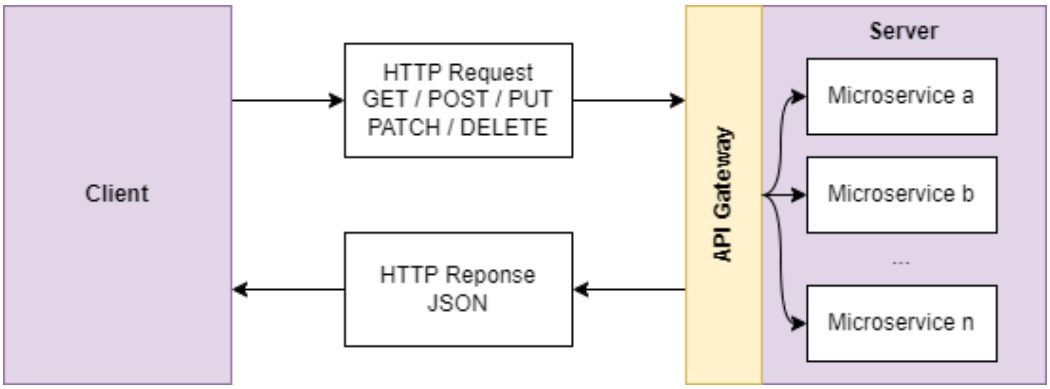


Figura 10: Diagrama: API Gateway cliente-servidor.

2.6 Testeo y monitoreo de las APIs

A medida que se implementan los distintos endpoints se debe probar que la petición y la respuesta son los esperados, además de realizar pruebas de seguridad para identificar brechas (únicamente responder lo que necesita, evitar dar información extra, SQL injection, etcétera.).

Nos ayudará a identificar a través de pruebas de carga y rendimiento, que carga aguanta el sistema, los recursos necesarios para las máquinas y a configurar apropiadamente el escalado de estas.

A nivel de IDE de desarrollo disponemos para hacer pruebas rápidas las extensiones de REST Client y Thunder Client, son versiones minimalistas para probar y depurar las APIs, iremos generando una batería de pruebas sencilla y podremos observar las respuestas a las peticiones.

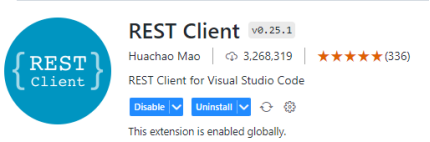


Figura 11: REST Client para Visual Studio.

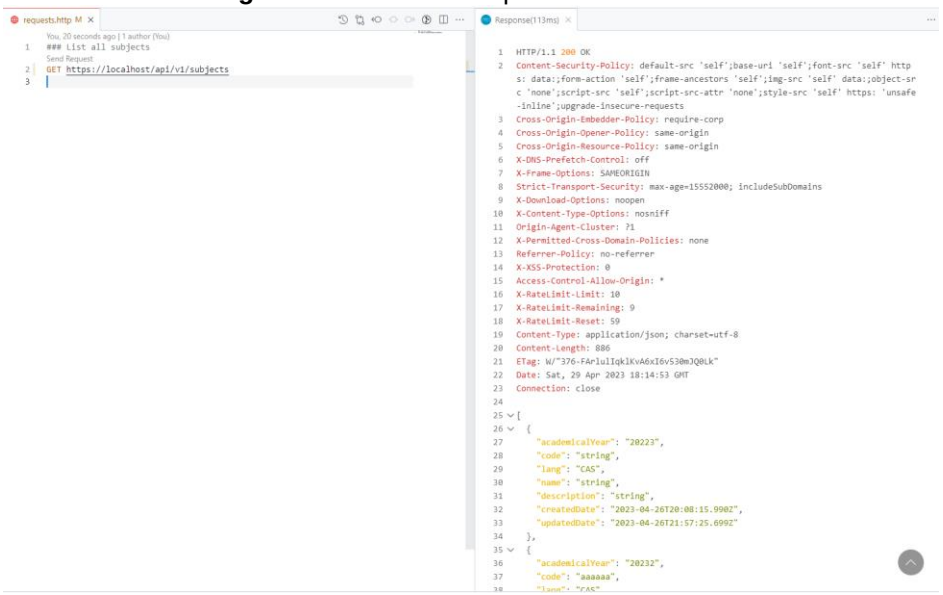


Figura 12: Petición realizada a través de REST Client.

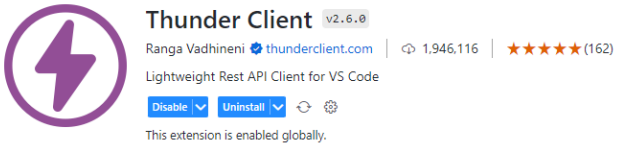


Figura 13: Thunder Client para Visual Studio.

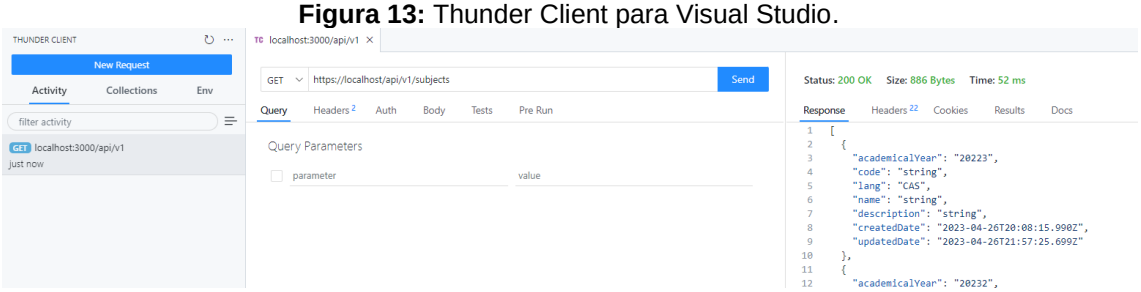


Figura 14: Petición realizada a través de Thunder Client.

Cuando debamos profesionalizar las pruebas, una de las herramientas más utilizadas para testear APIs es Postman [18], nos permite crear y probar APIs de forma rápida y sencilla, se pueden realizar peticiones, visualizar respuestas, preparar pruebas automatizadas y documentar.
Una de las funcionalidades más utilizadas es importar las especificaciones de APIs REST, como ejemplo la de swagger editor, creando un espacio de trabajo preparado para testear la API.

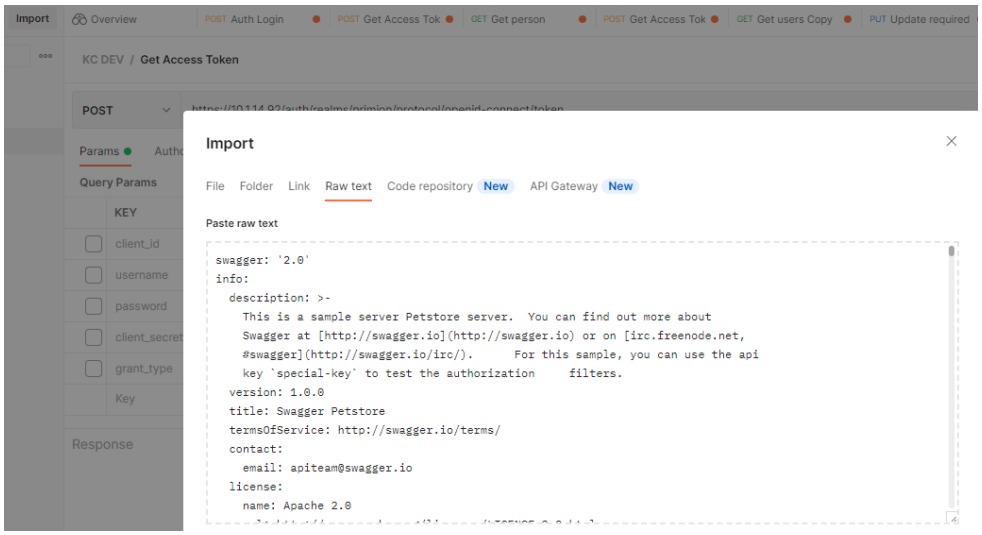


Figura 15: Importación de la definición del API en Postman.

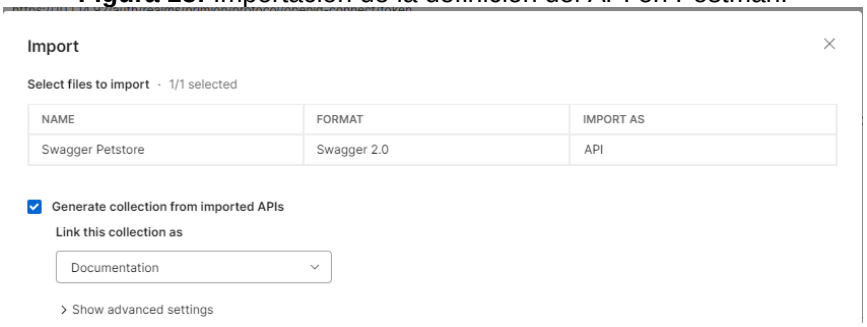


Figura 16: Configuración de la importación del API en Postman.

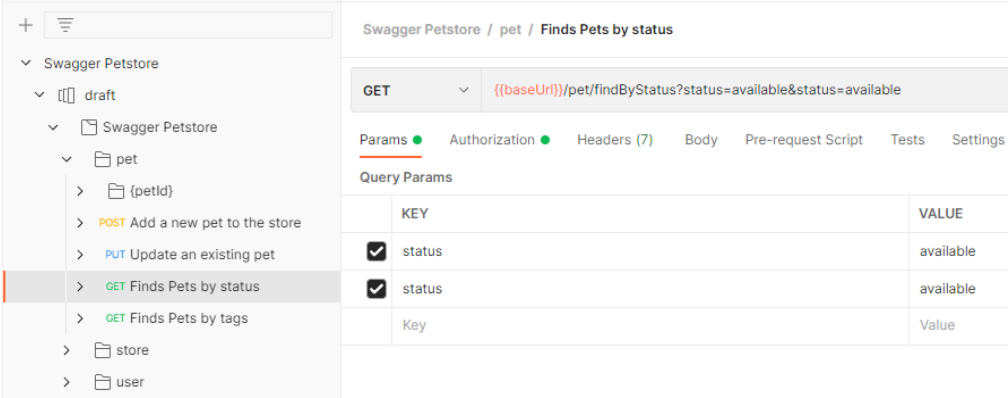


Figura 17: Estructura del API importada en Postman.

A nivel de pruebas de carga, estrés y rendimiento, una de las aplicaciones más utilizadas es Apache JMeter [19], basada en Java y de código abierto.

A nivel de monitoring una herramienta muy interesante es Sentry [20], nos ayuda a monitorear e identificar problemas en tiempo real, también el muy conocido stack de código abierto de Prometheus [21] para monitoreo y alertas, y Grafana [22] para visualización y análisis de datos. En este trabajo utilizaremos el stack de ElastiCSearch [23] y Kibana [24]. Además todas ellas tienen una muy buena integración con NestJS.

Gracias a la herramienta de OpenAPM [25] podemos planear una arquitectura para la gestión del performance y ver que herramientas disponibles hay, y su compatibilidad

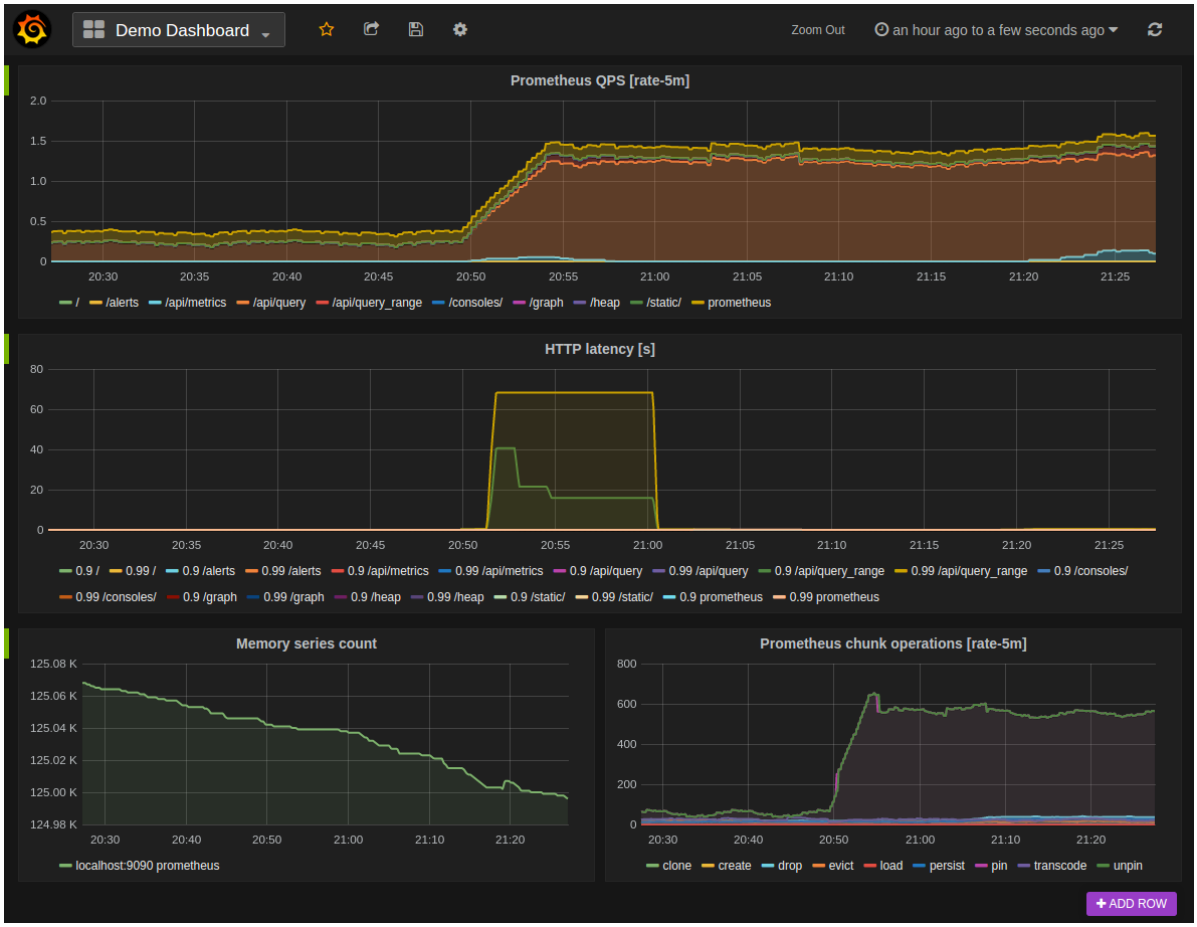


Figura 18: Dashboard de Grafana consultando Prometheus. Fuente: Grafana | Prometheus

2.7 Vulnerabilidades más comunes y mejores prácticas

Antes de identificar de las vulnerabilidades más comunes, debemos comprender que es la seguridad y los principios básicos.

Nos basaremos en OWASP (Open Web Application Security Project), es una organización sin ánimo de lucro que tiene como objetivo mejorar la seguridad del software.

OWASP facilita materiales, recursos, herramientas y documentación de buenas prácticas para afrontar las vulnerabilidades y ataques más comunes.

En ***OWASP*** describen la seguridad como:

> *"Security is simply about controlling who can interact with your information, what they can do with it, and when they can interact with it. These characteristics of control are described through what is called the CIA triad." (OWASP, 2016)*

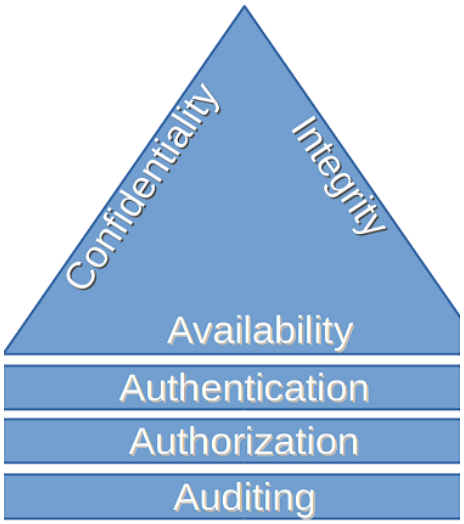


Figura 19: Fundamentos de seguridad. Fuente: OWASP [2]

CIA significa Confidencialidad, Integridad y Disponibilidad, estos son los tres principios clave que sustentan la seguridad.

- **Confidencialidad**: La información solo está disponible para quienes deberían tener acceso.
- **Integridad**: La información debe ser correcta y confiable.
- **Disponibilidad**: La información está disponible cuando se necesita para quienes deberían tener acceso.

Aquí comienza a aparecer el concepto de información, que está ligado con los siguientes conceptos:

- **Amenazas**: Acciones intencionales o no intencionales que pueden reducir el valor de un *activo*.
- **Activos**: Cualquier cosa que consideremos que tiene **valor**.
- **Vulnerabilidades**: Cualquier debilidad en un *activo* que lo hace susceptible de ataque o fallo.

Aplicado a nuestro contexto de APIs REST:

- **Amenazas**: Denegación de servicio, accesos no autorizados, ejecución de código malicioso, …
- **Activos**: La API REST, el API Gateway, la información de base de datos y la infraestructura.
- **Vulnerabilidades**: Arquitectura y/o validación de parámetros pobre, la encriptación o serialización, …

Debido al uso extendido y la criticidad que conllevan las APIs, OWASP comparte el proyecto "API Security Project" [2], el cual nos proporciona un conjunto de pautas, mejores prácticas y herramientas para que podamos utilizar para proteger las APIs.

La versión estable del proyecto es la del 2019 y podemos observar en la página web que ya hay una versión candidata del lanzamiento del 2023.

El top 10 de vulnerabilidades y riesgos de seguridad de las API propuesto es el siguiente:

API1:2019 Broken Object Level Authorization

Es la amenaza más común, las APIs tienden a exponer identificadores de objetos en el path, un atacante podría manipular ese identificador para extraer más datos de otros objetos, pudiendo consumir datos no autorizados.

Debemos comprobar que cuando un usuario va a realizar una acción, tiene permisos para realizarla, además nos debemos asegurar que no ha manipulado las cabeceras.

Soluciones:

- Uso de **UUIDs**, generando un identificador de 128 bits para que no puedan extraer data fácilmente. Ejemplo: Utilizando UUIDs pasaríamos de /asignatura/**1**, /asignatura/**2** a /asignatura/**cd63bb02-21bc-4098-9919-5e81140df616**

Los UUIDs se expresan mediante 32 dígitos hexadecimales divididos en cinco grupos separados por guiones, están estandarizados por la Open Software Foundation (OSF) y la norma ISO/IEC 11578.

Tiene el formato: xxxxxxxx-xxxx-Mxxx-Nxxx-xxxxxxxxxxxx

El carácter hexadecimal M representa la versión del UUID y los primeros 1 a 3 bits más significativos, en formato binario, del carácter N representan la variante del UUID. En nuestro contexto utilizaremos el autogenerado de Postgres.

- Uso de **tokens** para controlar temas de autenticación y autorización, validando la firma de este.

API2:2019 Broken User Authentication

Debido a la complejidad de implementar un método de autenticación seguro puede haber fallos que puedan comprometer los tokens de autenticación, permitiendo hacerte pasar por otros usuarios.

Los endpoints que manejan la creación, la autenticación de usuarios deben tener una capa extra de protección, no debemos permitir que puedan realizar ataques de fuerza bruta para probar usuarios y contraseñas hasta dar con los adecuados.

Soluciones:

- En las respuestas de olvidar o recuperar la contraseña no dar pistas de si el usuario o el email existe o no.
- Establecer unas políticas de contraseñas, tengan cierto tamaño, dígitos, caracteres, no se haya utilizado en las últimas 3 ocasiones, incluso renovarlo cada intervalo de tiempo establecido.
- Establecer rate limiting, un número de peticiones máxima cada intervalo de tiempo.
- Validar los tokens y establecer claves más complejas.
- Establecer autenticación multifactor, ya sea email o teléfono.
- Establecer un bloqueo de cuenta por un número de intentos fallidos.

API3:2019 Excessive Data Exposure

Exposición de toda la información de los objetos en lugar de dar únicamente lo que se espera del endpoint.

Soluciones:

- Responsabilidades de respuestas acotadas a las necesidades del negocio.
- Filtrar la información con el uso de DTOs de respuesta, evitando funcionalidad del tipo toJson o toString.

API4:2019 Lack of Resources & Rate Limiting

Cuando se pueden establecer peticiones que no requieren autenticación y no se establecen restricciones de recursos ni de peticiones permitiendo ataques DoS y de fuerza bruta.

Soluciones:

- Filtrar la información con el uso de DTOs de peticiones, además establecer unos límites en los campos.
- Establecer rate limiting, un número de peticiones máxima cada intervalo de tiempo.
- A través de Docker limitar el consumo de recursos

API5:2019 Broken Function Level Authorization

La complejidad de las organizaciones hace que haya niveles de jerarquía difíciles de manejar, y pueden generar endpoints débiles a estas amenazas, en ocasiones para una entidad el GET será público, y las demás peticiones de PATCH y DELETE deberían ser a través de autorización, para un atacante es sencillo cambiar el método y probar si pueden realizar acciones.

Una jerarquía sencilla de endpoints podría causar que los atacantes pudieran encontrar formas comunes de ataque probando /new /add /créate /all…

Soluciones:

- Establecer guardas validando la autorización a realizar peticiones a los distintos endpoint.

API6:2019 Mass Assignment

Asignar los valores de la petición al modelo de negocio, sin filtrar las propiedades que nos envían. Debemos estar atentos en no exponer comandos de consola en estas propiedades.

Soluciones:

- Filtrar la información con el uso de DTOs de peticiones, hacer un whitelist de las propiedades no esperadas para posteriormente transfórmalos en entidades de negocio.
- Propiedades esenciales deben únicamente ajustarse internamente.
- Aplicar esquemas para aplicar a los payload de entrada.

API7:2019 Security Misconfiguration

Uso de configuración por defecto o desconocimiento de aplicar una configuración de seguridad a niveles de cabeceras o métodos innecesarios, incluido el uso compartido de recursos de origen cruzado (CORS).

También es importante que los logs a nivel de info y verbose solo deben estar aplicando en modo de desarrollo o test, jamás en modo productivo.

Soluciones:

- Aplicar las últimas actualizaciones a todos los niveles.
- Aplicar la comunicación TLS entre los servicios.
- Aplicar la política CORS, limitando los métodos HTTP que podemos recibir.
- Limitar los logs extensivos para desarrollo.

API8:2019 Injection

Interactuar con bases de datos relacionales y no relacionales, explotando la interpretación de los motores de base de datos para poder ejecutar comandos o acceder a datos sin autorización.

Debemos validar los datos, filtrarlos y sanitizarlos, jamás concatenar parámetros de entrada de peticiones.

Soluciones:

- Uso de los parámetros adecuadamente utilizando un ORM.
- A ser posible utilizar query builders.
- Sanitizar los parámetros de entrada de las peticiones, tenemos librerías como class-transform y class-validator, escapar los carácteres especiales.

API9:2019 Improper Assets Management

Al exponer distintos endpoints y estructuras de datos, es importante documentar, para saber que parámetros se esperan, que acciones realizarán y que nos retornará.

Será de vital importancia inventariar para asegurarnos de que estamos exponiendo exclusivamente lo que el negocio requiere, sin estar dando información de más.

Soluciones:

- Documentar las API, indicando toda la información posible incluido el entorno y versión de la API.
- Establecer un rotado de versiones adecuado.
- A través de OpenAPI generar la documentación de manera automática.
- Establecer rate limiting, un número de peticiones máxima cada intervalo de tiempo.

API10:2019 Insufficient Logging & Monitoring

Monitorear y registrar es importante para poder identificar las acciones, peticiones y respuestas que estamos generando.

El seguimiento de lo que sucede, nos ayudará a identificar mucho más rápido comportamientos inadecuados.

Soluciones:

- Establecer un sistema de logging.
- Establecer un sistema de monitoring.
- Establecer un sistema de alertas.

3. Solución propuesta

3.1. Arquitectura de la solución

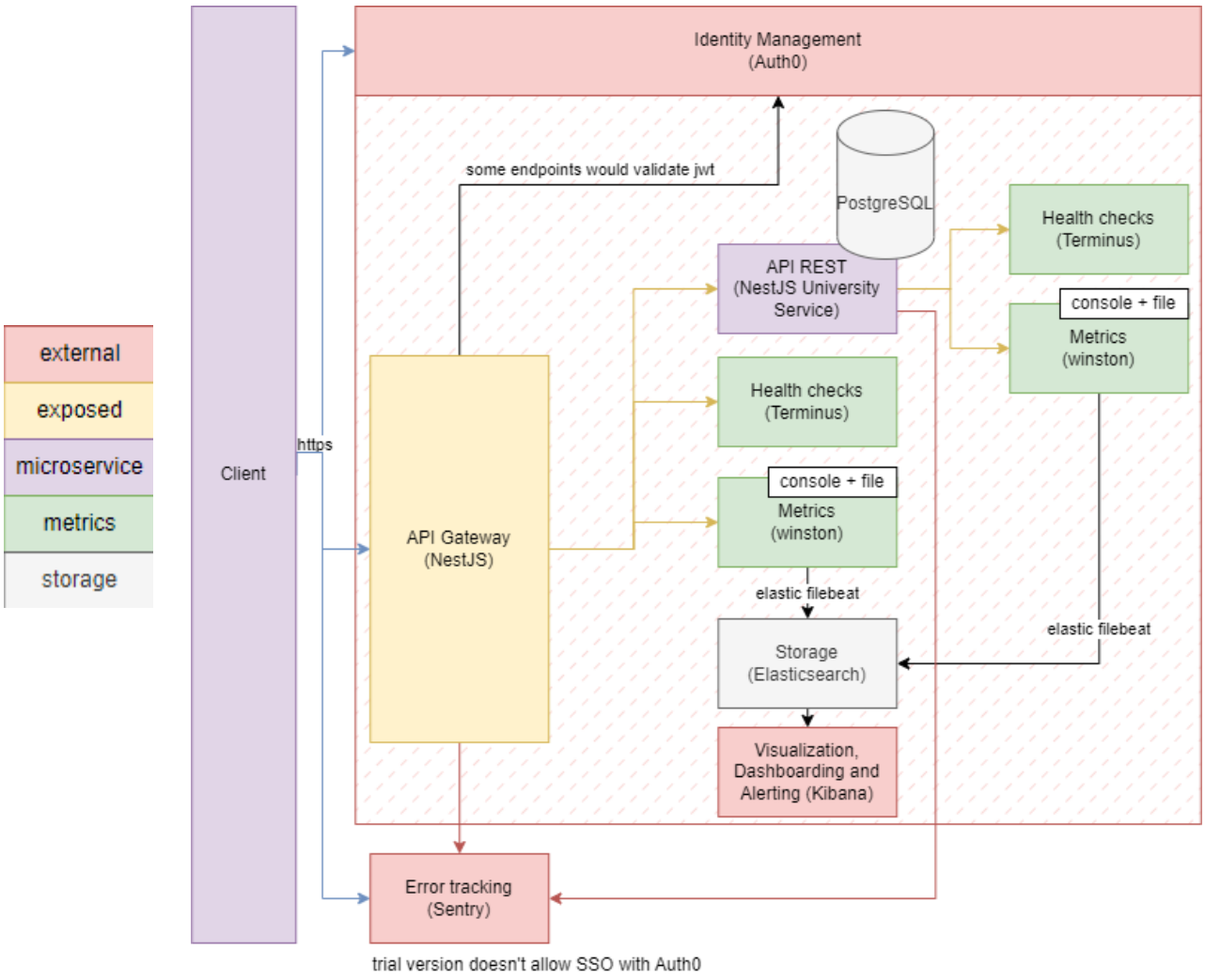


Figura 20: Arquitectura de la solución propuesta.

Para poder afrontar la mayoría de las áreas de seguridad en el proyecto se ha pensado una arquitectura que contiene:

Concepto	Framework o 3rd party	Descripción
API Gateway	NestJS	Punto de entrada de la API REST, health checks, metrics y analytics.
API REST	NestJS	Exposición de servicios de la Universidad.
Identity Management	Auth0	Gestión de usuarios, se utilizará de obtener JWT token y validar las peticiones. Además, nos servirá como SSO para Grafana.
Base de datos	PostgreSQL	Almacenamiento de la información de la Universidad.
Health checks (API Gateway + API REST)	Terminus	Estado de API Gateway y API REST, salud de los servicios.
Metrics (API Gateway + API REST)	Winston	Recolección de métricas de uso y rendimiento (basado en elasticsearch) de manera temporal.

Agente	Elastic Beats	Se encarga de leer de ficheros y a través de un índice enviar los logs a elasticsearch.
Storage	Elasticsearch	Winston escribe en disco y rota los logs, aunque es una buena práctica que se persistan en un formato concreto.
Monitoring	Kibana	Monitorización de los datos en tiempo real y obtención de una visión completa del comportamiento y rendimiento.
Alerting	Kibana	Configuración de alertas para detectar anomalías.
Analytics	Kibana	Visualización de datos y análisis de métricas.
Error tracking	Sentry	Gestión de errores y excepciones. La versión trial no permite hacer SSO con Auth0.

Tabla 21: Tabla de componentes de arquitectura.

El flujo de comunicación empezará desde el lado cliente realizando una petición a un endpoint, según el contexto de cada endpoint se deberá hacer una validación del JWT que proporcionará Auth0 con los usuarios creados.

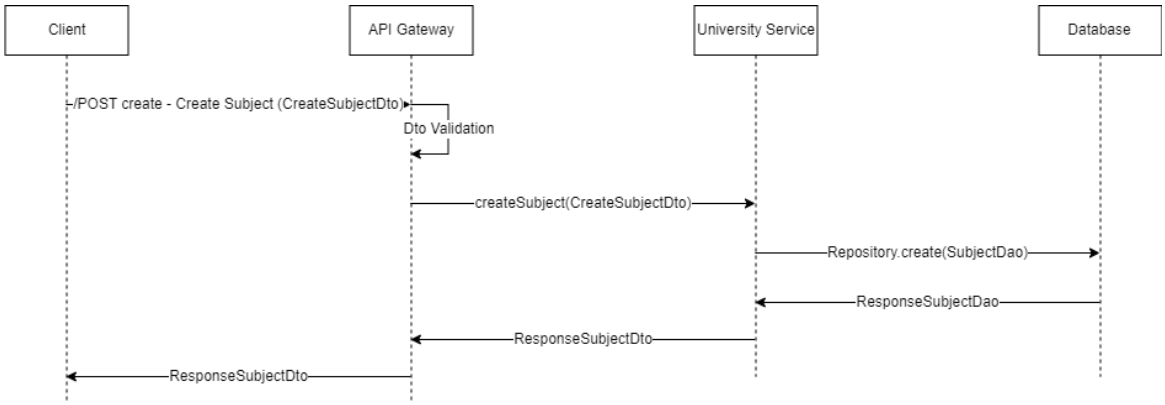


Figura 22: Flujo de comunicación de peticiones a través de API Gateway.

La petición acabará en el API Gateway que gracias a los validation pipes validará los DTO que nos lleguen con las validaciones que indiquemos, entonces redireccionaremos al University Service, el cual hará la transformación del DTO a la entidad para enlazarla con el DAO para guardar en base de datos. Según el resultado de la operación, devolveremos una respuesta o excepción que capturaremos para devolvérsela al cliente.

A nivel de arquitectura de NestJS, el body de las peticiones las validaremos y transformaremos a DTOs para acabar trabajando con entidades las cuales haremos match a los DAO.

Para hacer el mapping entre las entidades y las tablas, haremos uso de un ORM, el ORM escogido es TypeORM, ya que es la recomendada a través de la documentación de NestJS, este nos ayudará a interactuar con la base de datos.

Utilizaremos arquitectura hexagonal, y para ello implementaremos el patrón repositorio para separar la lógica de negocio de base de datos [Figura 23]. La idea sería ser agnósticos del ORM y la base de datos escogida (PostgreSQL) así en el caso de que algún cliente necesitará utilizar una base de datos que no fuera compatible con TypeORM podríamos realizar la migración sin tanto coste.

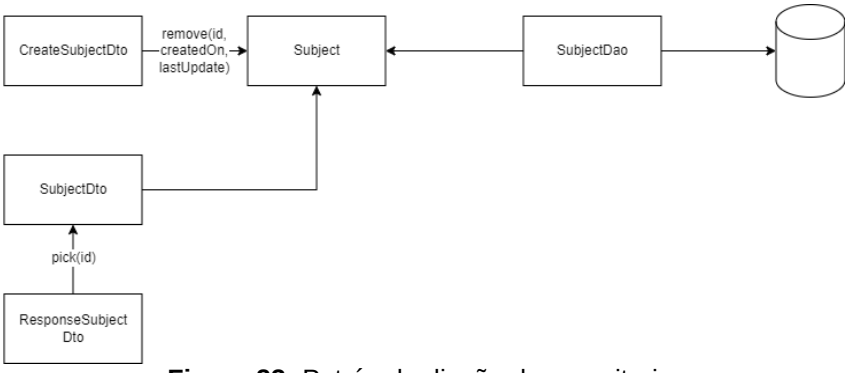


Figura 23: Patrón de diseño de repositorio.

3.2. Detalles de la implementación

A nivel de API Gateway y API REST, estableceremos una conexión al microservicio de universidad utilizando el protocolo de transporte TCP, el cual nos garantizará la integridad de la información controlando la gestión de la red.

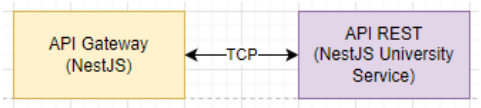


Figura 24: Protocolo de transporte entre API Gateway y API REST.

El registro de la conexión se realizará a nivel de API Gateway y los datos los extraeremos de las variables de entorno.

nestjs-security-rest-api - api-gateway.module.ts

```
47  ClientsModule.registerAsync([
48      {
49        name: ClientServices.UNIVERSITY_SERVICE,
50        inject: [ConfigService],
51        useFactory: async (configService: ConfigService) => ({
52          transport: Transport.TCP,
53          options: {
54            host: configService.get('university.host'),
55            port: configService.get('university.port'),
56          },
57        }),
58      },
59    ]),
```

Figura 25: Registro de conexión en API Gateway con el microservicio de Universidad.

Teniendo la siguiente definición a nivel de microservicio:

nestjs-security-rest-api - main.ts

```
25  const configService = app.get(ConfigService);
26    const port = configService.get('port');
27
28    app.connectMicroservice<MicroserviceOptions>(
29      {
30        transport: Transport.TCP,
31        options: {
32          port: port,
33        },
34      },
35      { inheritAppConfig: true },
36    );
```

Figura 26: Configuración microservicio Universidad.

Para la gestión de variables de entorno se ha definido la configuración y esquema para todas las variables necesarias.

Nos aseguraremos de cargar las variables de entorno adecuadas a través de la propiedad envFilePath, las registraremos en modo global con isGlobal y finalmente aplicaremos las validaciones a nivel de esquema, verificando que están todas las variables necesarias, sin permitir las desconocidas.

nestjs-security-rest-api - api-gateway.module.ts

```
33  ConfigModule.forRoot({
34       load: [configuration],
35       isGlobal: true,
36       //* ignoreEnvFile: true,
37       envFilePath: [
38         `apps/api-gateway/src/environments/.env.${process.env.NODE_ENV}
    `,
39       ],
40       validationSchema: envSchema,
41       validationOptions: {
42         abortEarly: true,
43         allowUnknown: false,
44         stripUnknown: true,
45       },
46     }),
```

Figura 27: Carga y validación de variables de entornos.

Ejemplo de definición de variables de entorno para el API Gateway:

nestjs-security-rest-api - .env.development

```
1  NODE_ENV=development
2
3  APP_NAME=api-gateway
4
5  API_GATEWAY_SCHEMA=https
6  API_GATEWAY_HOST=localhost
7  API_GATEWAY_PORT=443
8
9  UNIVERSITY_SERVICE_HOST=localhost
10 UNIVERSITY_SERVICE_PORT=3001
11
12 SENTRY_DSN=
   https://df525f14858d453dae24b7d52afa5c06@o4504990324555776.ingest.sentr
   y.io/
   4504990363549696
13
14 THROTTLE_TTL=60
15 THROTTLE_LIMIT=10
16
17 AUTH0_ISSUER_URL=https://nestjs-security-rest-api.eu.auth0.com/
18 AUTH0_AUDIENCE=nestjs-security-rest-api
```

Figura 28: Variables de entorno para development.

Dado que ambos servicios utilizarán los mismos Data Transfer Objects (DTOs), se ha decidido establecer el proyecto como un monorepo, el cual nos beneficiará a la hora de compartir código gracias a librerías.

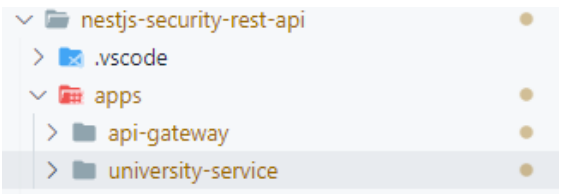


Figura 29: Monorepo en Nestjs, dos proyectos API Gateway y microservicio Universidad.

Se ha creado una librería “shared”, la cual contendrá configuraciones, constantes a nivel de catálogo de excepciones o servicios, filtros y middlewares comunes junto a la definición de tipos de la base de datos (tamaños y enums), además de los DTOs que reciben los endpoints y respuestas que devolvemos.

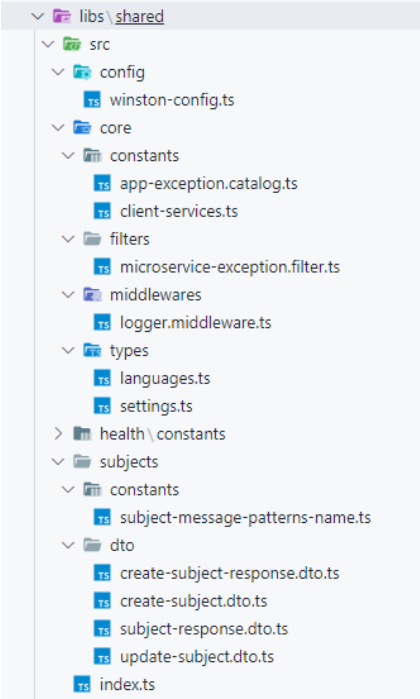


Figura 30: Organización librería shared en NestJS.

Cuando nos llegue una petición al controlador del API Gateway, al llamar al servicio correspondiente, y poder comunicarse con el microservicio adecuadamente, debemos generar un patrón y el payload con la información además de indicar que nos retornará dicho servicio.

Como podemos observar en este ejemplo, se ha definido en la librería “shared” un enumerado con los distintos nombres de patrón para asignaturas, para poderlos hacer coincidir en ambos lados.

En el API Gateway, en el servicio de asignaturas:

nestjs-security-rest-api - subjects.service.ts

```
18  create(
19      createSubjectDto: CreateSubjectDto,
20    ): Observable<CreateSubjectResponseDto> {
21      const pattern = SubjectMessagePatternsName.CREATE;
22      const payload: CreateSubjectDto = createSubjectDto;
23      return this.clientUniversityService.send<CreateSubjectResponseDto>(
24        pattern,
25        payload,
26      );
27    }
```

Figura 31: Servicio API Gateway, conexión, envío y respuesta.

En el microservicio de university-service, en el controlador, esperará el mismo nombre de patrón:

nestjs-security-rest-api - subjects.controller.ts

```
15  @MessagePattern(SubjectMessagePatternsName.CREATE)
16    async create(
17      @Payload() payload: CreateSubjectDto,
18    ): Promise<CreateSubjectResponseDto> {
19      const code: string = await this.subjectsService.create(
20        Subject.fromDto(payload),
21      );
22
23      return new CreateSubjectResponseDto(code);
24    }
```

Figura 32: Microservicio Universidad, vinculación, recepción y tratado de payload.

3.3. Base de datos

Se ha escogido la base de datos PostresQL, la cual nos permitirá hacer migraciones en el caso de realizar cambios en el diseño del dominio.

A nivel del microservicio de universidad, se cargará la configuración del ORM, y además se marcará que se auto carguen las entidades.

nestjs-security-rest-api - university.module.ts

```
42  TypeOrmModule.forRoot({
43        ...ormconfig.default.options,
44        autoLoadEntities: true,
45      }),
```

Figura 33: Carga configuración TypeOrm y auto recarga de entidades.

Los Data Access Objects serán marcados con el decorador @Entity, que indicará a TypeORM las entidades que tenemos, además de indicar los distintos decoradores para construir nuestras tablas.

A destacar que debemos indicar un @PrimaryColumn pudiendo indicar el tipo de valor y la generación de este, las demás tendrán el decorador @Column, y para el caso concreto de fechas de creación y actualización @CreateDateColumn y @UpdateDateColumn para que las autogestione.

nestjs-security-rest-api - subject.dao.ts

```
16  @Entity(DbTableNames.SUBJECT)
17  export class SubjectDao extends Subject {
18    @PrimaryColumn({ type: 'uuid', generated: 'uuid' })
19    id: string;
```

Figura 34: Diseño de la entidad Subject.

A nivel de configuración de TypeORM, indicaremos el tipo de base datos, y los datos de conexión, está preparado para gestionar migraciones (cambios del diseño de baso de datos), y además para poder comprobar las consultas activamos el logging. Finalmente, y solo para modo desarrollo, dejamos el synchronize a true ya que queremos que se sincronice siempre que se relance la aplicación para no tener que andar con migraciones mientras desarrollamos.

nestjs-security-rest-api - ormconfig.ts

```ts
const AppDataSource = new DataSource({
  type: 'postgres',
  host: process.env.DATABASE_HOST,
  port: parseInt(process.env.DATABASE_PORT),
  database: process.env.DATABASE_DB,
  username: process.env.DATABASE_USER,
  password: process.env.DATABASE_PASSWORD,
  entities: [SubjectDao, TeacherDao, SubjectTeacherDao],
  migrations: ['./migrations/*{.ts,.js}'],
  migrationsTableName: 'migrations',
  logging: true,
  synchronize: true,
});
```

Figura 35: Configuración de TypeOrm.

Necesitaremos desplegar y persistir a través volumenes con docker-compose para poder gestionar la base de datos, en este caso se ha utilizado DBeaver para realizar las distintas comprobaciones.

Indicando la configuración de la conexión, podremos navegar y validar los resultados.

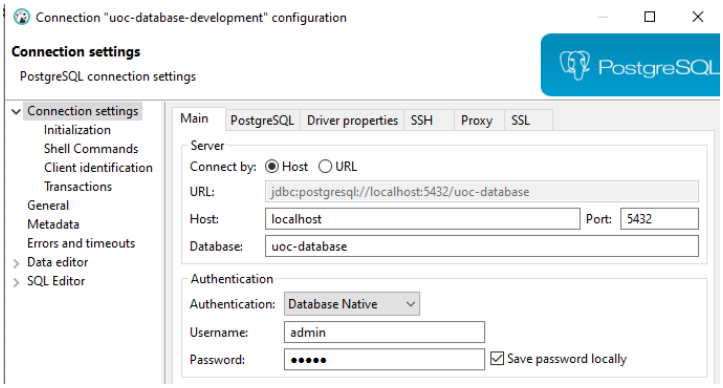


Figura 36: DBeaver configuración de la conexión.

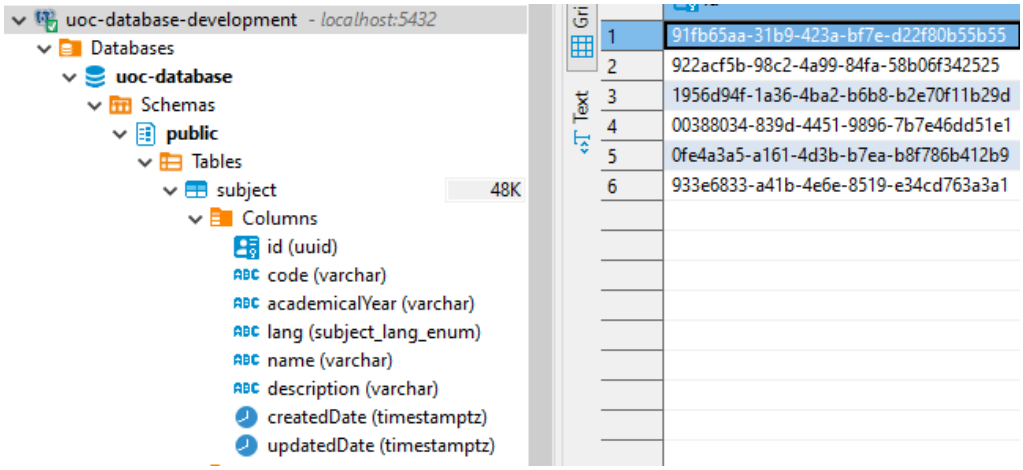


Figura 37: Navegación base de datos, esquemas, tablas y columnas.

3.4. NestJS Security REST API

NestJS a través de la librería de @nestjs/swagger, nos facilita la construcción del documento, el versionado y todos los decoradores posibles para que se realice una documentación adecuada del API.

Se presenta la NestJS Security REST API, la cual está formada por dos controladores:

- Health: Estado de salud del API Gateway y del university-service.
- Subjects: CRUD de las asignaturas.

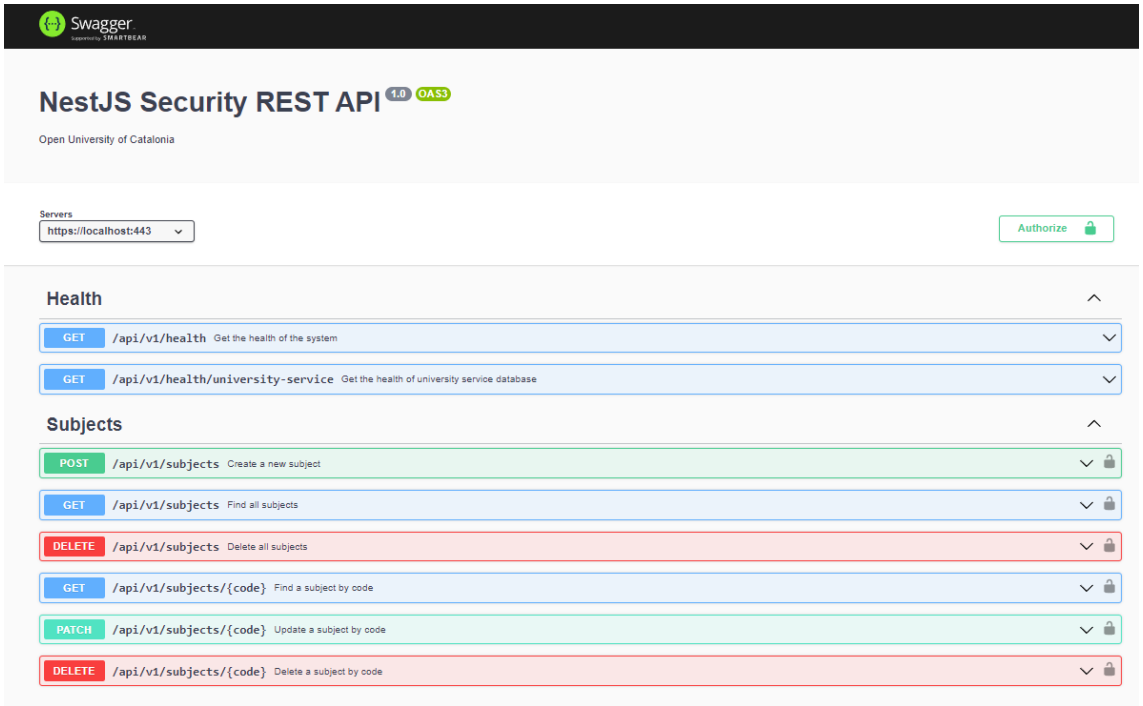


Figura 38: Swagger definición NestJS Security REST API.

Se ha escogido PATCH en lugar de PUT, como decisión de negocio de enviar únicamente lo que se va a cambiar, patch se utiliza para actualizar parcialmente un recurso existente, mientras put se utiliza para reemplazar completamente un recurso existente con una nueva representación. Por lo tanto, el tamaño del body será mucho menor y podemos aplicar permisos a nivel de campos, la principal razón es para evitar reemplazo de información. A nivel de schemas, los DTOs de las peticiones y respuestas de los diferentes endpoints, aquí podremos observar los valores y tipos esperados.

Figura 39: Esquemas de los DTOs esperados.

Cada endpoint dispondrá de su documentación con su descripción, los parametros, body (con ejemplo) y las posibles respuestas.

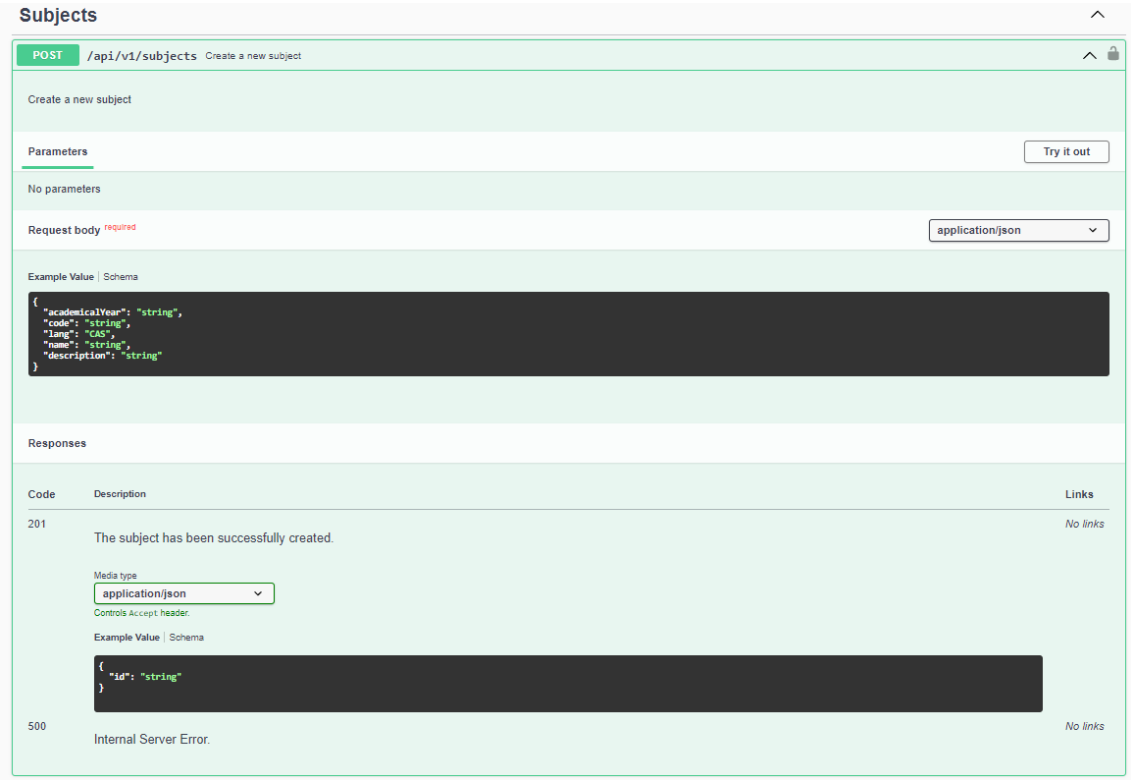


Figura 40: Vista de endpoint en Swagger.

Cada vez que se ejecute la aplicación se genera la definición del api en el archivo: open-api.yaml, el cual utilizaremos más tarde para poder realizar las pruebas a través de Postman.

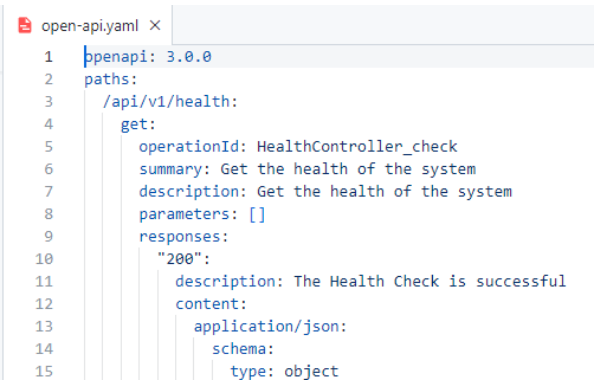


Figura 41: Definición API REST en formato YAML.

3.5. Gestión de la identidad

De cara poder gestionar la autenticación y validación de la identidad de los usuarios que utilizarán el API REST, debemos tener una gestión de la identidad.

Para esta solución se ha escogido Auth0, ya que nos ofrece una solución Cloud, con un plan gratuito suficiente para nuestros casos de uso, además de una configuración muy sencilla de realizar.

Auth0 nos permite generar un API de autenticación, creación de usuarios, roles y permisos, el primer paso será darnos de alta.

Una vez tengamos la cuenta creada deberemos crear un tenant:

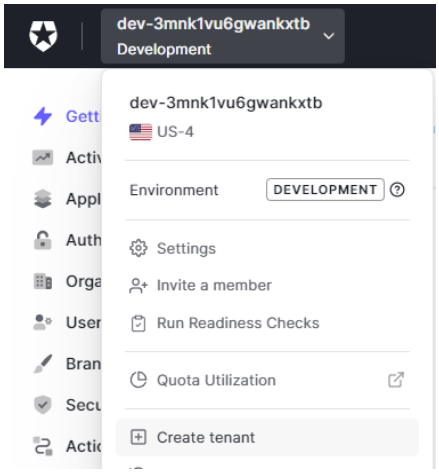


Figura 42: Creación tenant en Auth0.

Se ha generado un tenant: nestjs-security-rest-api, con región EU (Europa) y la etiqueta de “Development”.

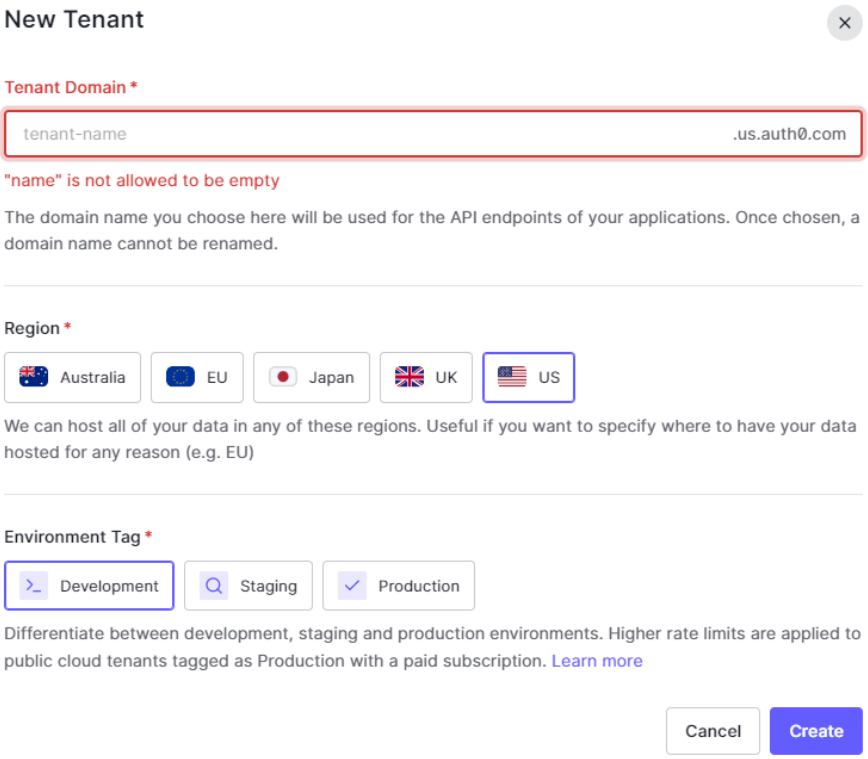


Figura 43: Generación de nuevo tenant en Auth0.

Haremos un Switch al nuevo tenant que acabamos de crear:

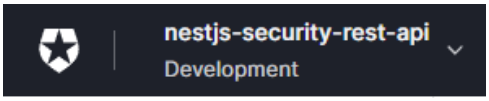


Figura 44: Actual tenant activo en Auth0.

Cada tenant puede tener distintas aplicaciones que serán a las cuales querremos conectarnos, aparecerá la información necesaria para poder obtener el token de acceso gracias al “Client ID” y al “Client Secret”.

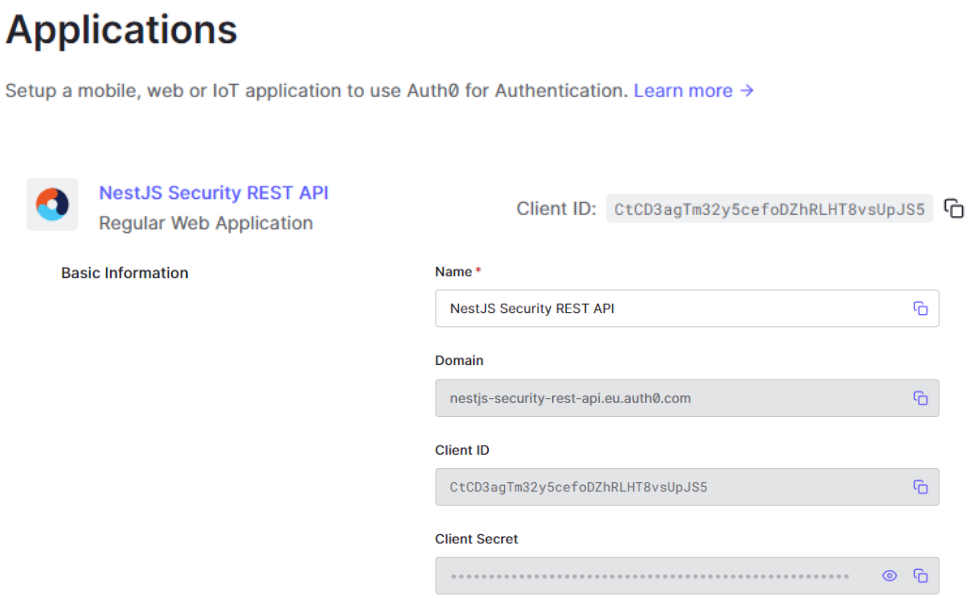


Figura 45: Aplicación NestJS Security REST API en Auth0.

Deberemos asegurarnos de activar el tipo de autenticación de “Password” para poder obtener el token de acceso:

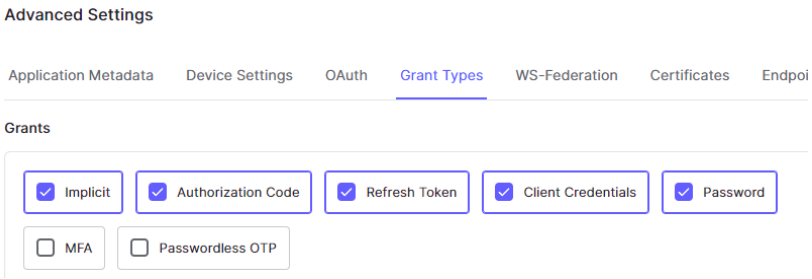


Figura 46: Activación de autenticación con password en Auth0.

El siguiente paso será crear un API para poder realizar la autenticación, indicaremos el nombre del API, en nuestro caso “NestJS Security REST API”, junto a un identificador “nest-security-rest-api” que utilizaremos para validar que el JWT que recibamos sea el correcto.

Figura 47: Creación API autenticación en Auth0.

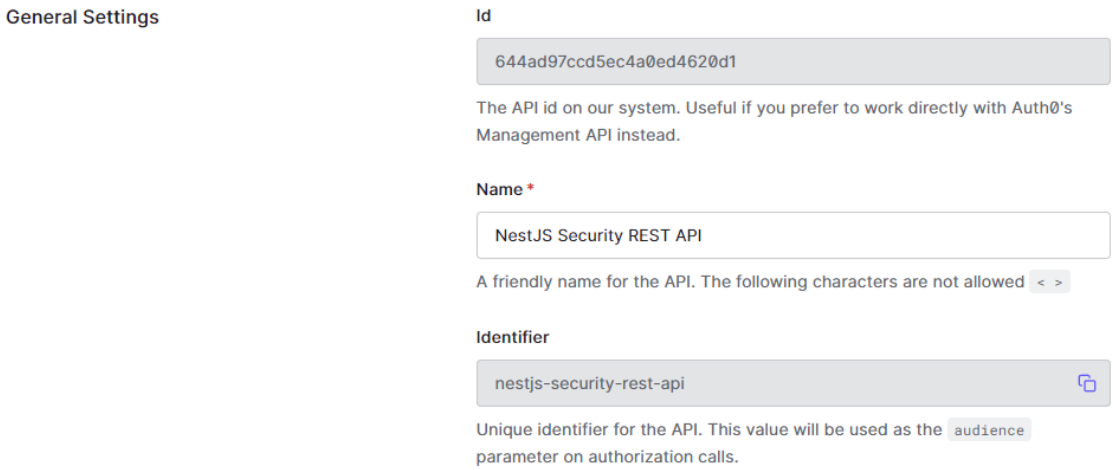


Figura 48: Configuración del API autenticación en Auth0.

Ahora solo nos hace falta crear los usuarios, el caso de uso adecuado sería tener un frontend con el cual pudieramos ingestar usuarios a Auth0, para nuestro ejemplo vamos a crearlo manualmente:

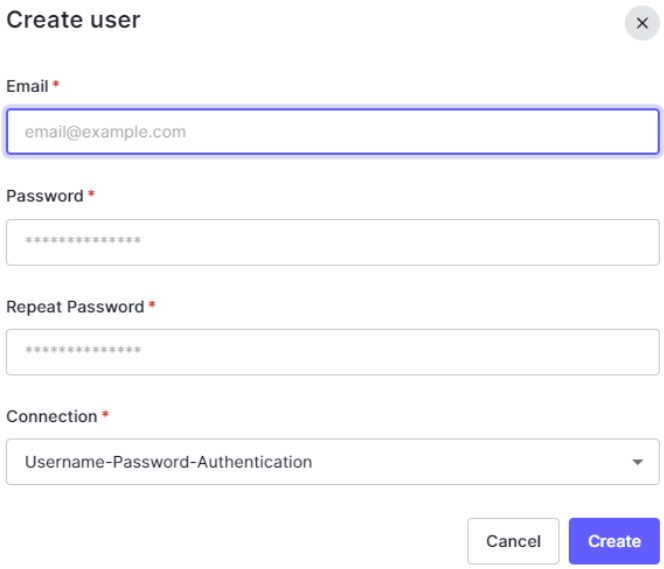


Figura 49: Creación usuario en Auth0.

Finalmente, podremos ver el listado de usuarios del que disponemos:

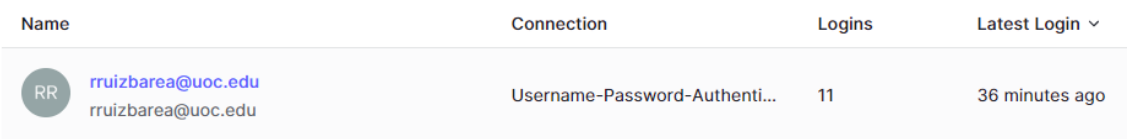


Figura 50: Listado de usuarios en Auth0.

Un gran beneficio de Auth0 es la posibilidad de crear Roles y asignarlos a los usuarios, crearemos dos Roles: Admin y User, para poder gestionar la autorización a nivel del API REST.

Name	Description
Admin	Administrator
User	User

Figura 51: Listado de roles en Auth0.

Una vez realizada la configuración de Auth0, podremos obtener el access_token realizando un CURL o una petición a través de Postman indicando el tipo de acceso, usuario y contraseña, además de la audiencia y el cliente al cual debemos conectar:

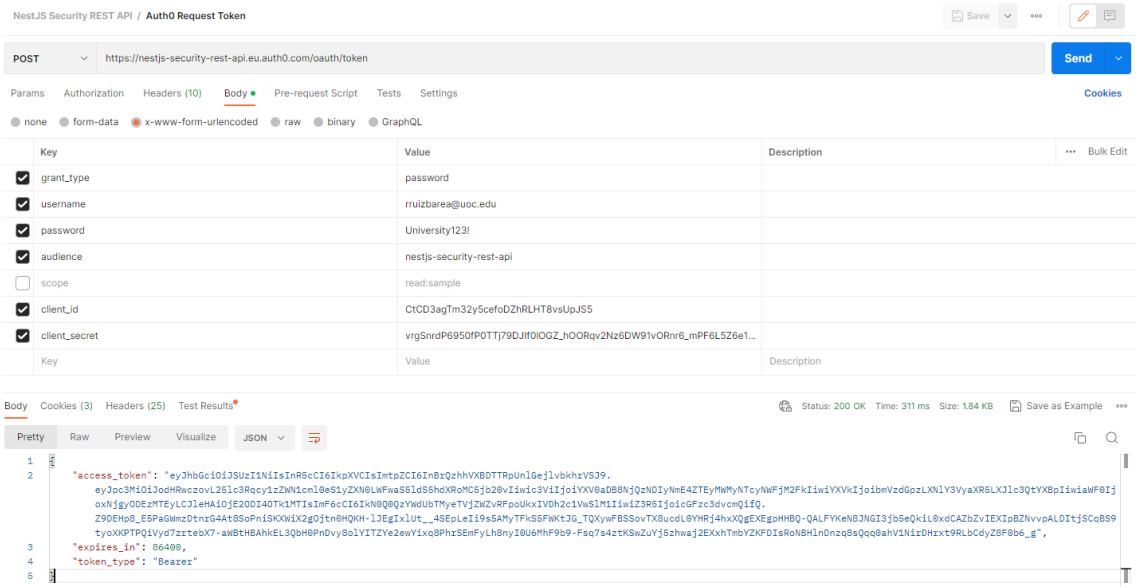


Figura 52: Obtención del token de Auth0 a través de Postman.

Observamos que el token tiene una expiración, que podemos configurar a nivel de API de Auth0 para reducir los tiempos:

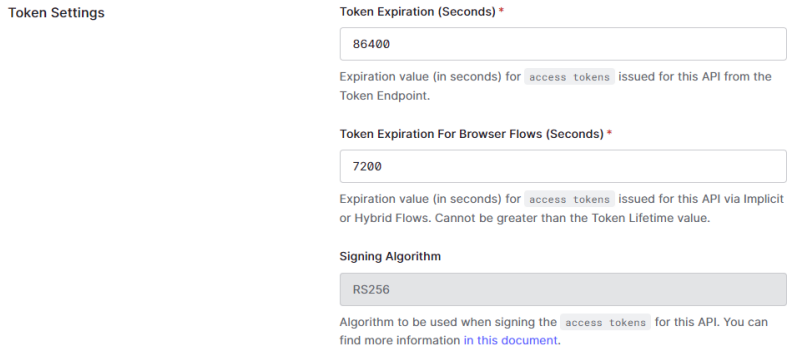


Figura 53: Configuración del token en Auth0.

Los JSON Web Tokens (JWT) son un estándar abierto que define un formato para transmitir información de forma segura.

Están formado por tres partes: cabecera, payload y firma, separadas por puntos (xxxxx.yyyyy.zzzzz). La cabecera contiene el tipo de token y el algoritmo de firma, en el payload contiene la información de la identidad, con información de la entidad que certifica, el tiempo de expiración, la audiencia y otros, finalmente tenemos la firma, que se calcula con la cabecera, el payload codificado y una clave para firmar el token. El resultado son tres strings en formato base64-URL separados por puntos, para validar la firma debemos tener en cuenta el algoritmo de firma, con el secreto se recalcula la firma y se compara con la del JWT, si los valores son iguales se puede confiar en la veracidad de la identidad.

En la estrategia de JWT, deberemos indicar la audiencia y la entidad que certifica junto el tipo de algoritmo, gracias a las funciones provistas de NestJS automáticamente calculará la firma y hará la validación del token.

nestjs-security-rest-api - jwt.strategy.ts

```
15  super({
16    secretOrKeyProvider: passportJwtSecret({
17      cache: true,
18      rateLimit: true,
19      jwksRequestsPerMinute: 5,
20      jwksUri: `${configService.get('auth0.issuerUrl')}
    .well-known/jwks.json`,
21    }),
22
23    jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
24    audience: configService.get('auth0.audience'),
25    issuer: configService.get('auth0.issuerUrl'),
26    algorithms: ['RS256'],
27  });
```

Figura 54: Estrategia JWT para Auth0.

3.6. Gestión de errores y excepciones

A nivel de gestión de errores se ha decidido utilizar Winston como logger para NestJS, se han definido los siguientes transporters:

- Consola: Cualquier log desde ‘info’ que se realice se mostrará por consola, indicando el timestamp y formateándolo de una manera fácil de interpretar.

- Fichero: Cualquier log desde ‘http’, se creará un fichero diario en la carpeta logs con el patrón ‘YYYY-MM-DD’, con un máximo de ficheros de 30 días y en formato de elastic search para interpretarlos en Kibana.

Se capturarán los errores que se generen a nivel de controlador del API Gateway y de los servicios del API REST.

Para los errores se ha generado un catálogo de errores para no dar pistas sobre el error al usuario final, pero poder gestionarlo a nivel de frontend estableciendo unos contratos, asociándolos también con su Http Status Code.

nestjs-security-rest-api - app-exception.catalog.ts

```ts
export enum APP_EXCEPTION {
  BAD_REQUEST = 'APP_4000',
  NOT_FOUND = 'APP_4004',
  CONFLICT = 'APP_4009',
}

```

Figura 55: Catálogo de excepciones.

Para errores desconocidos se indicará un Internal Server Error (500), si permitiéramos devolver el mensaje de error podríamos estar filtrando demasiada información, como, por ejemplo: con TypeORM, si no se controla la inserción de valores nulos, el mensaje de error auto generado indica los nombres de la columna y la tabla, lo cual pone en riesgo la seguridad.

A nivel de respuestas estarán formadas por un código de estado, el momento en que sucedió, el path en el cual se hizo la petición y la información del mensaje de error.

nestjs-security-rest-api - rpc-exception.filter.ts

```ts
response.status(_error.statusCode).json({
      statusCode: _error.statusCode,
      timestamp: timestamp,
      path: request.url,
      info: _error.message,
    });
```

Figura 56: Filtro de respuestas de excepciones.

A nivel interno, todo mensaje de error estará formado por un mensaje, un stack, un código de estado y el momento en que sucedió. No queremos que el stack se devuelva a nivel de respuesta, pero si nos será útil para poder identificar las causas raices de los errores.

nestjs-security-rest-api - microservice-exception.filter.ts

```ts
const timestamp = new Date().toISOString();
    this.logger.error({
      message: exception.message,
      stack: exception.stack,
      extra: {
        statusCode: exception.error.statusCode,
        timestamp,
      },
    });
```

Figura 57: Enriquecimiento de respuestas internas.

Para poder realizar un seguimiento de errores, rastreando y diagnosticando los errores se hará uso de Sentry.

La finalidad sería generar alertas sobre errores no controlados hacía Sentry, para ello tendremos que generar una cuenta al servicio Cloud la cual con la versión gratuita nos será más que suficiente.

El primer paso será crear un proyecto de tipo NodeJS:

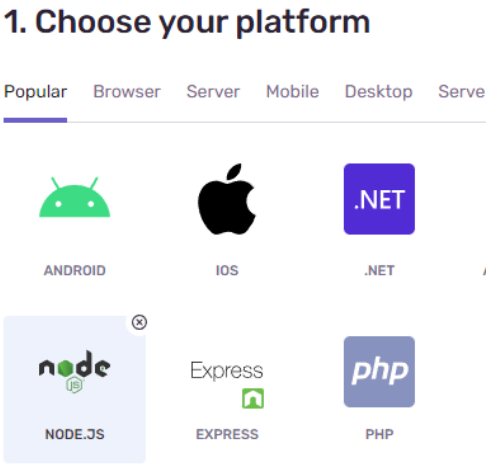


Figura 58: Configuración proyecto Sentry.

Sentry nos facilita la siguiente documentación para conectar nuestra API REST:

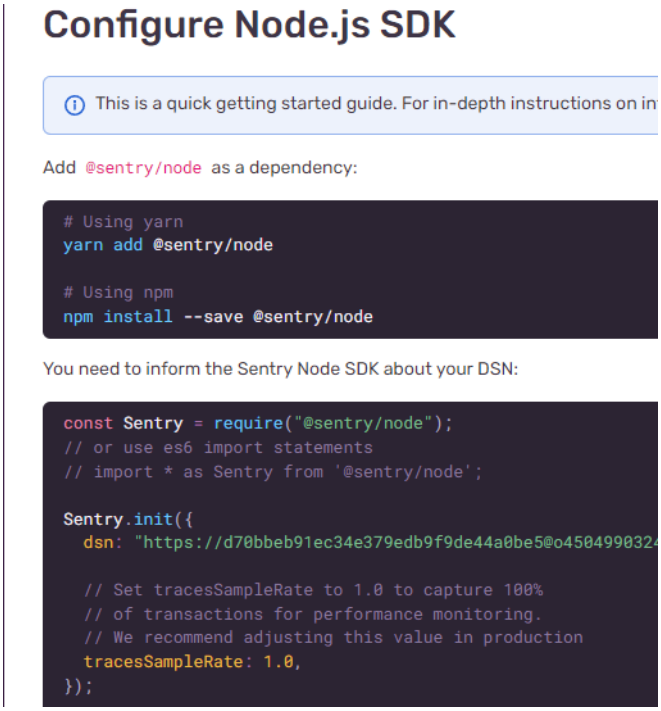


Figura 59: Documentación de Sentry para NodeJS.

.Para ello estableceremos las variables de entorno necesarias para poder comunicarnos:

nestjs-security-rest-api - main.ts

```
49  Sentry.init({
50      dsn: configService.get('sentry.dsn'),
51      tracesSampleRate: 1.0,
52      environment: configService.get('environment'),
53    });
```

Figura 60: Inicialización de Sentry.

Finalmente cuando haya un error a nivel de cualquier microservicio que podamos tener, si el error no es controlado, y por lo tanto, mayor o igual a 500, capturaremos la excepción y la enviaremos a Sentry con toda la información necesaria.

nestjs-security-rest-api - microservice-exception.filter.ts

```
26  if (exception.error.statusCode >= 500) {
27        Sentry.captureException(exception, {
28          extra: {
29            timestamp,
30            statusCode: exception.error.statusCode,
31            message: exception.message,
32            stack: exception.stack,
33          },
34        });
35      }
```

Figura 61: Envío de errores superiores a 500 a Sentry.

Cuando esto suceda nos aparecerá un nuevo issue en Sentry:

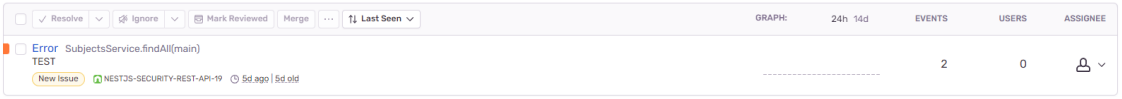


Figura 62: Planificación del trabajo y dependencias.

Si consultamos la información, podremos ver muchos datos sobre el error sucedido, incluido el stack para poder entender que ha podido suceder.

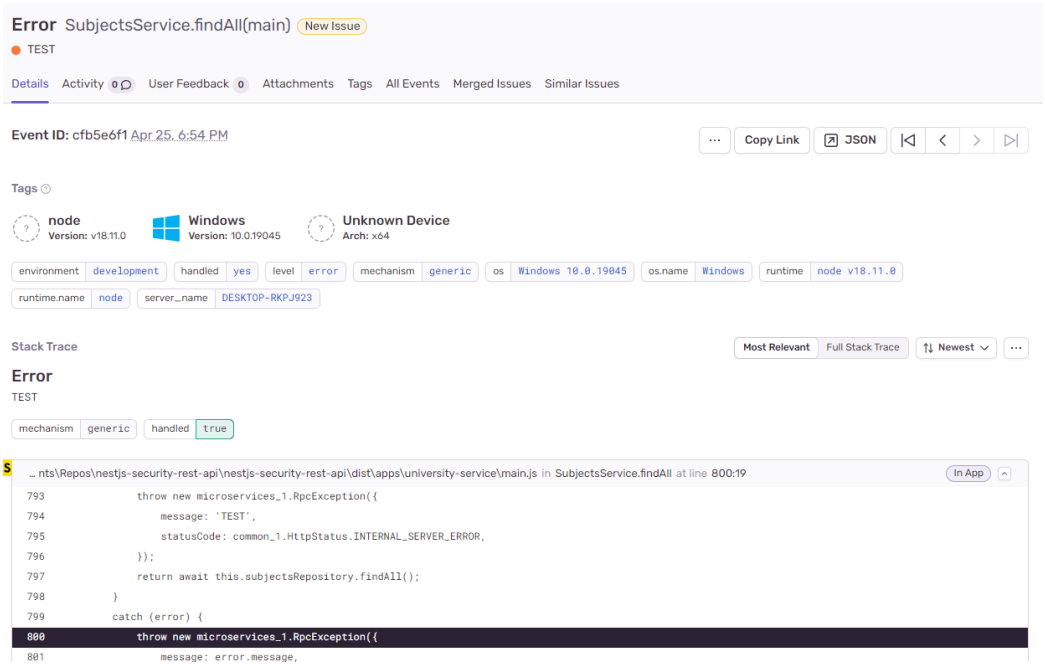


Figura 63: Información recibida en Sentry.

Figura 64: Información adicional en Sentry.

A nivel de proyecto podremos ver estadisticas, total de errores, errores diarios, y hacer un seguimiento de la resolución de dichos errores:

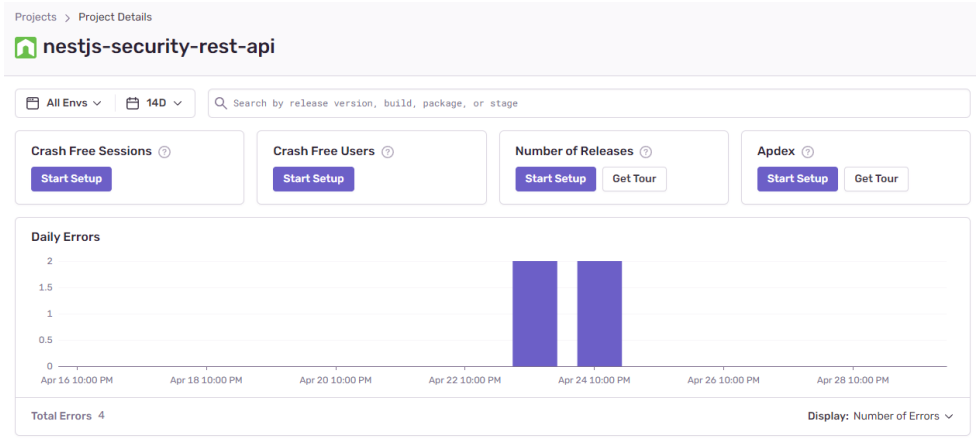


Figura 65: Estadísticas del proyecto de Sentry.

También podremos crear nuevas alertas, para poder actuar lo antes possible para solucionar los errores:

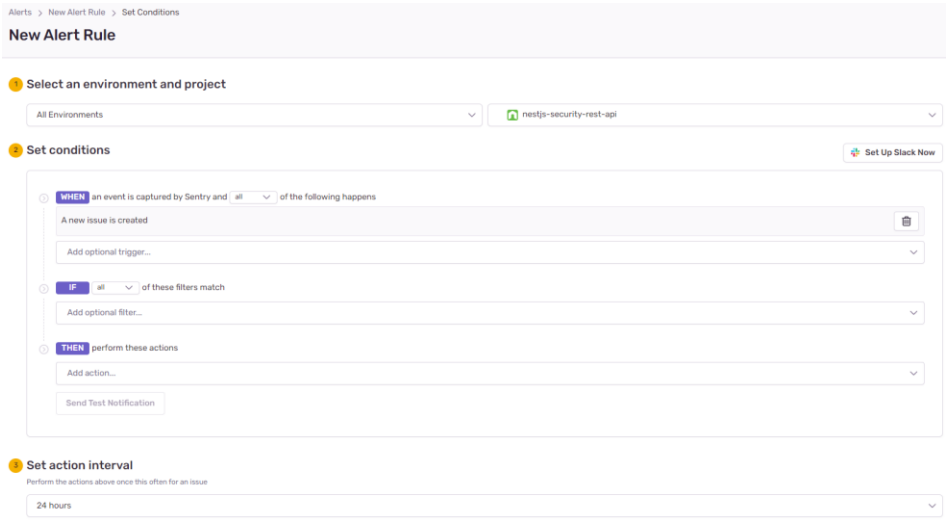


Figura 66: Creación de nuevas alertas en Sentry.

Nos interesará a nivel de API Gateway la gestión de excepciones a nivel de controlador:

nestjs-security-rest-api - subjects.controller.ts

```
55  @Post()
56    create(
57      @Body() createSubjectDto: CreateSubjectDto,
58    ): Observable<CreateSubjectResponseDto> {
59      try {
60        sanitize(createSubjectDto);
61
62        return this.subjectsService.create(createSubjectDto);
63      } catch (error) {
64        throw new HttpException(
65          'Internal Server Error',
66          HttpStatus.INTERNAL_SERVER_ERROR,
67        );
68      }
69    }
```

Figura 67: Gestión de excepciones en el API Gateway.

De cara los microservicios, deberemos gestionar las excepciones a nivel de servicio, y si no controlamos el código de error devolver un Internal Server Error (500, no controlado):

nestjs-security-rest-api - subjects.service.ts

```
13  async create(subject: Subject): Promise<string> {
14      try {
15        return await this.subjectsRepository.create(subject);
16      } catch (error) {
17        throw new RpcException({
18          message: error.message,
19          statusCode: error.error?.statusCode ?? HttpStatus.
    INTERNAL_SERVER_ERROR,
20        });
21      }
22    }
```

Figura 68: Gestión de excepciones en los microservicios.

Y a nivel de repositorio, lanzar excepciones controladas cuando hayan duplicidades, códigos no encontrados, o sean errores no controlados propios de TypeORM y debamos controlar los mensajes de error:

nestjs-security-rest-api - subject-typeorm.repository.ts

```
18  async create(subject: Subject): Promise<string> {
19      const subjectDao: SubjectDao = await this.subjectRepository.
    findOneBy({
20        code: subject.code,
21      });
22
23      if (subjectDao) {
24        throw new RpcException({
25          message: APP_EXCEPTION.CONFLICT,
26          statusCode: HttpStatusCode.Conflict,
27        });
28      }
29
30      const createdSubjectDao: SubjectDao = await this.subjectRepository.
    create(
31        subject,
32      );
33
34      const insertResult: InsertResult = await this.subjectRepository
35        .createQueryBuilder()
36        .insert()
37        .into(SubjectDao)
38        .values(createdSubjectDao)
39        .execute()
40        .catch((exception) => {
41          throw new RpcException({
42            message: APP_EXCEPTION.BAD_REQUEST,
43            statusCode: HttpStatusCode.BadRequest,
44          });
45        });
46
47      return insertResult.identifiers[0].id;
48    }
```

Figura 69: Gestión de excepciones a nivel de base de datos.

Para que las excepciones tengan el formato que esperamos, debemos crear los distintos exception filters, que capturaran las excepciones y podremos darles el formato que queramos.

Aplicación de los filtros de excepciones:

nestjs-security-rest-api - api-gateway.module.ts

```
76  {
77        provide: APP_FILTER,
78        useClass: HttpExceptionFilter,
79      },
80      {
81        provide: APP_FILTER,
82        useClass: RcpExceptionFilter,
83      },
```

Figura 70: Inclusión de proveedores en el API Gateway.

nestjs-security-rest-api - university.module.ts

```
51  providers: [
52      UniversityService,
53      {
54        provide: APP_FILTER,
55        useClass: MicroServiceExceptionFilter,
56      },
57    ],
```

Figura 71: Inclusión de proveedores en los microservicios.

Captura y tratado de las excepciones:

nestjs-security-rest-api - rpc-exception.filter.ts

```
19  catch(exception: any, host: ArgumentsHost) {
20      const timestamp = new Date().toISOString();
21      const ctx = host.switchToHttp();
22      const response = ctx.getResponse<Response>();
23      const request = ctx.getRequest<Request>();
24
25      const _error = this._getError(exception);
26
27      response.status(_error.statusCode).json({
28        statusCode: _error.statusCode,
29        timestamp: timestamp,
30        path: request.url,
31        info: _error.message,
32      });
33    }
```

Figura 72: Captura y tratado de excepciones.

En el caso que quisieramos enriquecer la información indicando tiempos y limites a la hora de hacer el rate limit, podríamos capturar excepciones concretas para añadir headers o mensajes personalizados:

nestjs-security-rest-api - throttler-exception.filter.ts

```
15  catch(exception: ThrottlerException, host: ArgumentsHost) {
16      const ttl = this.configService.get('throttle.ttl');
17      const limit = this.configService.get('throttle.limit');
18      const timestamp = new Date().toISOString();
19      const ctx = host.switchToHttp();
20      const request = ctx.getRequest<Request>();
21      const response = ctx.getResponse();
22
23      response.setHeader('Retry-After', ttl.toString());
24
25      response.status(HttpStatus.TOO_MANY_REQUESTS).json({
26        statusCode: HttpStatus.TOO_MANY_REQUESTS,
27        timestamp: timestamp,
28        path: request.url,
29        info: `Too many requests, limit: ${limit}, ttl: ${ttl}`,
30      });
31    }
```

Figura 73: Enriquecimiento de excepciones sobre rate limits.

3.7. Health checks

Para poder validar el estado de salud del api-gateway, los entornos y los distintos microservicios, se utiliza Terminus, para ello lo importaremos para mostrar los logs más leíbles en cualquier entorno que no sea productivo.

nestjs-security-rest-api - health.module.ts

```
9   TerminusModule.forRoot({
10        errorLogStyle: process.env.NODE_ENV === 'production' ? 'json' : 'pretty',
11      }),
```

Figura 74: Configuración de Terminus.

En en controlador, añadiremos un endpoint el cual haga la validación del disco a nivel de espacio libre, validación de memoria y validación de los microservicios.

nestjs-security-rest-api - health.controller.ts

```ts
@Get()
  @HealthCheck()
  check(): Promise<HealthCheckResult> {
    return this.health.check([
      () =>
        this.disk.checkStorage('storage', {
          path: 'C:\\',
          thresholdPercent: 0.8,
        }),
      () => this.memory.checkHeap('memory_heap', 150 * 1024 * 1024),
      () => this.memory.checkRSS('memory_rss', 150 * 1024 * 1024),
      () =>
        this.microservice.pingCheck('university-service', {
          transport: Transport.TCP,
          options: {
            host: this.configService.get('university.host'),
            port: this.configService.get('university.port'),
          },
        }),
    ]);
  }
```

Figura 75: Health check endpoint.

Al consultar dicho endpoint podremos ver si los distintos controles están levantados o no, gracias a dicha información podremos averiguar si hace falta escalar vertical o horizontalmente, ampliar memoria, espacio de disco o añadir répiclas.

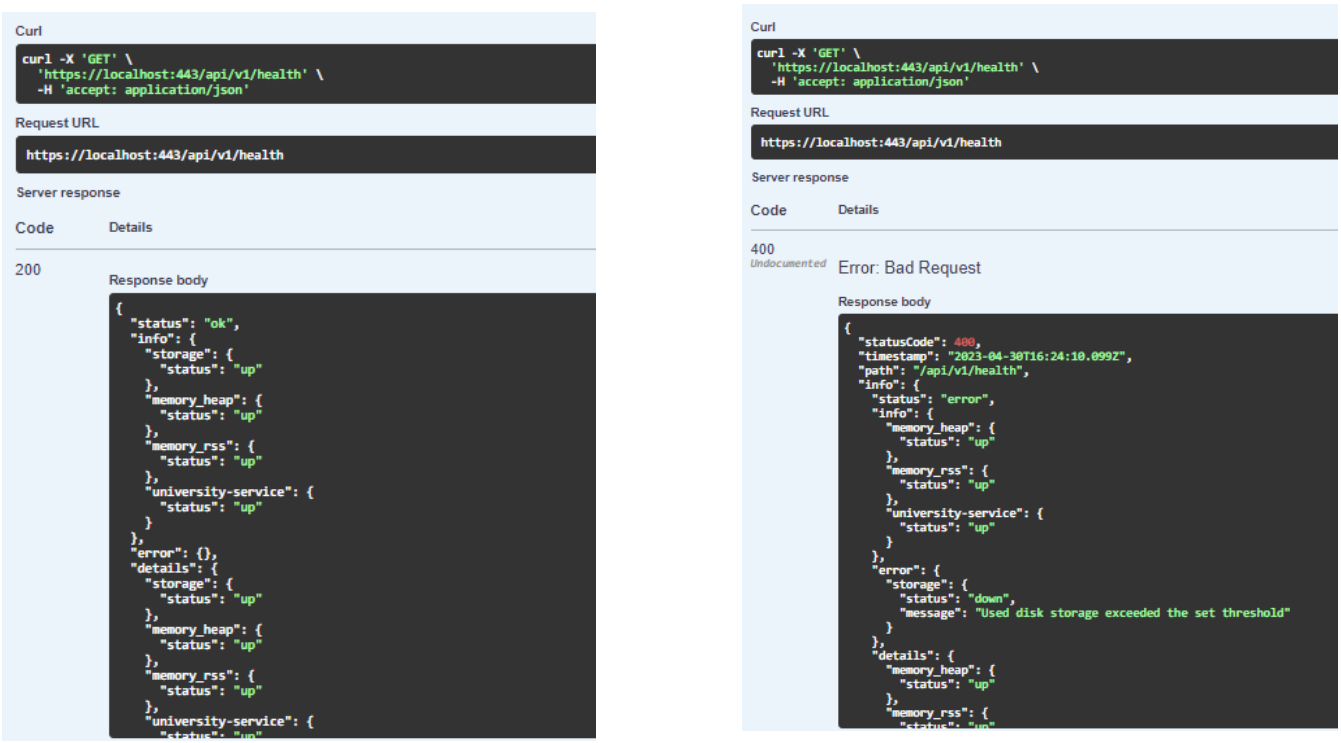


Figura 76: Resultado de los health checks.

3.8. Trazabilidad

Gracias a la herramienta de OpenAPM podemos explorar y comparar las distintas herramientas y compatibilidades de cara poder seleccionar las adecuadas para cada contexto.

En nuestro contexto, utilizaremos Elastic Beats para leer los ficheros de log del API Gateway y del API REST, a través del indexado que realiza, mantener un volumen persistente gracias a Elasticsearch, para después obtener la parte de Dashboarding, Observability y Alerting de Kibana.

Figura 77: Arquitectura del APM.

Para poder generar los ficheros de log utilizaremos Winston junto a la librería 'winston-daily-rotate-file' para poder guardar los ficheros de Log que mantendremos 30 días en formato de Elasticsearch.

nestjs-security-rest-api - winston-config.ts

```
18  const fileTransport = new winston.transports.DailyRotateFile({
19    filename: '%DATE%-application.log',
20    dirname: './logs',
21    datePattern: 'YYYY-MM-DD',
22    zippedArchive: false,
23    maxFiles: '30d',
24    format: ecsFormat({ convertReqRes: true }),
25    level: 'http',
26  });
```

Figura 78: Configuración del rotado de logs.

Podemos observar como se generan los logs a diario con la nomeclatura seleccionada:

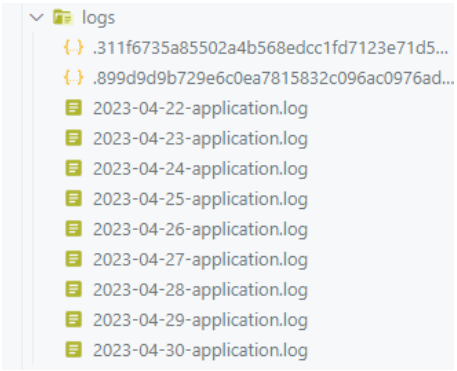


Figura 79: Logs diarios a nivel de aplicación.

El contenido de los logs es en formato JSON de Elasticsearch, comprobamos que tienen todos los datos:

2023-04-30-application.log


```
{"@timestamp":"2023-04-29T23:27:00.008Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-29T23:28:59.018Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T14:17:37.459Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T16:16:06.519Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T16:16:19.815Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T16:16:45.864Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T16:16:48.305Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T16:17:16.975Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T16:17:32.075Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T16:17:40.347Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T16:17:45.563Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T16:17:56.523Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
{"@timestamp":"2023-04-30T16:18:12.913Z","log.level":"http","message":"HTTP Request/Response logged","ecs":{"version":"1.6.0"},"http":{"version":"1.1",
```


Figura 80: Formato elastic sobre los logs.

Para que el fichero se vaya incrementando, tenemos dos puntos criticos: trazar toda petición con su respuesta y almacenar todos los errores sucedidos.

Cualquier uso del Logger de Winston pasará por el transporte a consola y fichero, como existen distintos niveles de error (info, warning, http, error, …) y cada uno tiene una finalidad concreta, se ha tomado la decisión de las trazas http loggearlas a nivel de fichero y no por consola por su extensión, y los errores a nivel de consola y fichero, además de los mayores a 500 también a Sentry.

nestjs-security-rest-api - logger.middleware.ts

```
@Injectable()
export class LoggerMiddleware implements NestMiddleware {
  constructor(
    @Inject(WINSTON_MODULE_PROVIDER) private readonly logger: Logger,
  ) {}

  use(req: Request, res: Response, next: NextFunction) {
    res.on('finish', () => {
      this.logger.http('HTTP Request/Response logged', {
        req,
        res,
      });
    });

    next();
  }
}

```

Figura 81: Logger de las peticiones a través de un middleware.

Como herramienta visual tendremos Kibana, la cual podremos consultar a través de la URL: http://localhost:5601/

Debido a que de serie no tiene ningún Dashboard, lo primero que tendremos que hacer es importar los objetos que hay preparados en el proyecto: nestjs-security-rest-api-infra tendremos el fichero export.ndjson.

export.ndjson

You, 4 days ago | 1 author (You)

```
{"attributes":{"fieldAttrs":"{}","fieldFormatMap":"{}","fields":"[]","name":"NestJS Security REST API","runtimeFi
{"attributes":{"description":"","kibanaSavedObjectMeta":{"searchSourceJSON":"{\"query\":{\"query\":\"\",\"languag
{"attributes":{"columns":[],"description":"","grid":{},"hideChart":false,"isTextBasedQuery":false,"kibanaSavedObj
{"attributes":{"description":"","kibanaSavedObjectMeta":{"searchSourceJSON":"{\"query\":{\"query\":\"\",\"languag
{"attributes":{"description":"","kibanaSavedObjectMeta":{"searchSourceJSON":"{\"query\":{\"query\":\"\",\"languag
{"attributes":{"description":"","kibanaSavedObjectMeta":{"searchSourceJSON":"{\"query\":{\"query\":\"\",\"languag
{"attributes":{"description":"","kibanaSavedObjectMeta":{"searchSourceJSON":"{\"query\":{\"query\":\"\",\"languag
{"attributes":{"buildNum":61109,"isDefaultIndexMigrated":true},"coreMigrationVersion":"8.7.0","created_at":"2023-
{"attributes":{"buildNum":61109,"defaultIndex":"ca34f21d-2143-4129-87ab-47f357ea3aa9","isDefaultIndexMigrated":tr
{"excludedObjects":[],"excludedObjectsCount":0,"exportedCount":9,"missingRefCount":0,"missingReferences":[]}
```

Figura 82: Objetos exportados de Kibana.

Deberemos ir a la sección de gestión de Kibana, para poder importar este fichero:

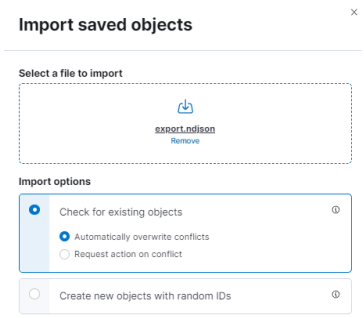


Figura 83: Importación de los objetos guardados en Kibana.

Una vez importado veremos que contienen todos los objetos necesarios, incluido el Dashboard:

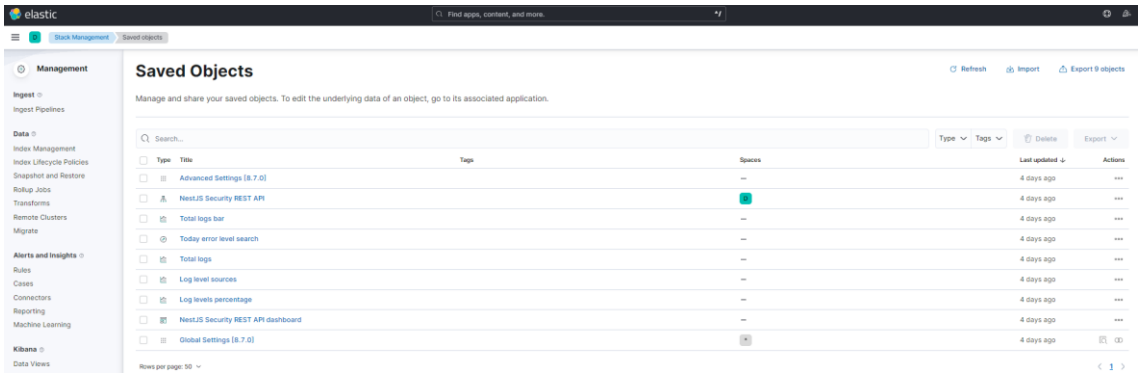


Figura 84: Comprobación de los objetos guardados en Kibana.

Para poder visitar el Dashboard, visitaremos “Analytics” > “Dashboard”:

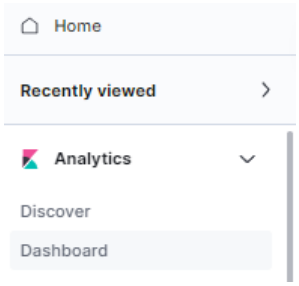


Figura 85: Menú Analytics en Kibana.

En esta Dashboard, podremos ver de entrada los datos del día actual, deberemos filtrar para poder ver otras franjas de tiempo, aquí podemos observar la franja de los últimos 7 días.

Cabe destacar que se pueden ir filtrando datos, pudiendo escoger solo mostrar los errores, el tipo de servicio y cualquier otro dato.

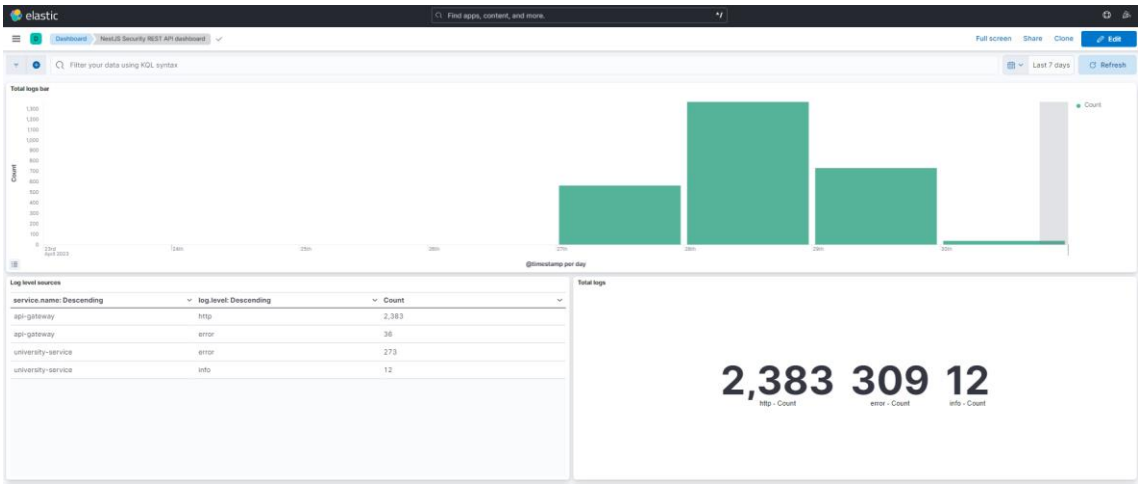


Figura 86: Dashboard de Kibana.

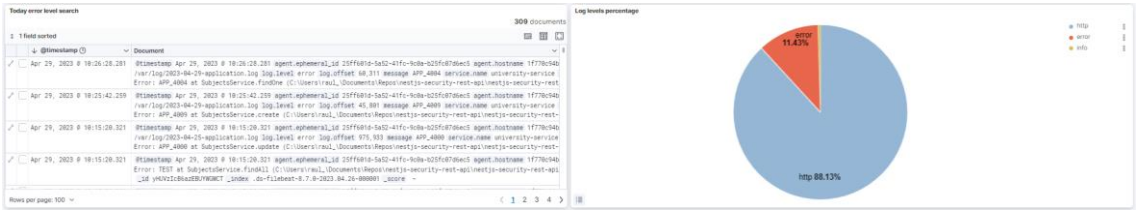


Figura 87: Amplicación del dashboard de Kibana.

Una vez filtrado los datos podemos ver los logs de forma expandida, al tener muchos datos la información viene paginada:

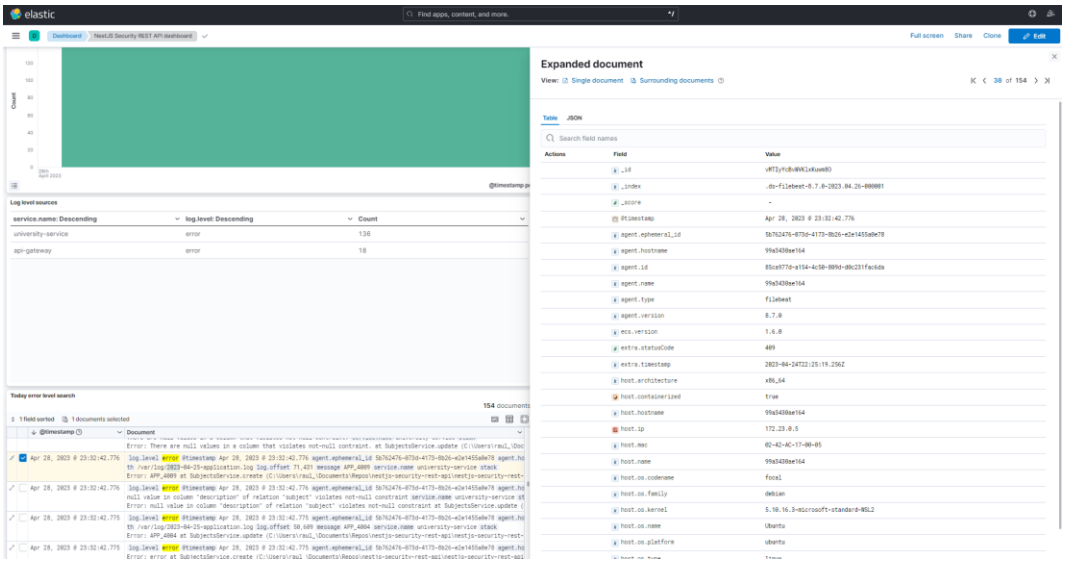


Figura 88: Expansión del dashboard de Kibana.

De cara a los datos más interesantes:

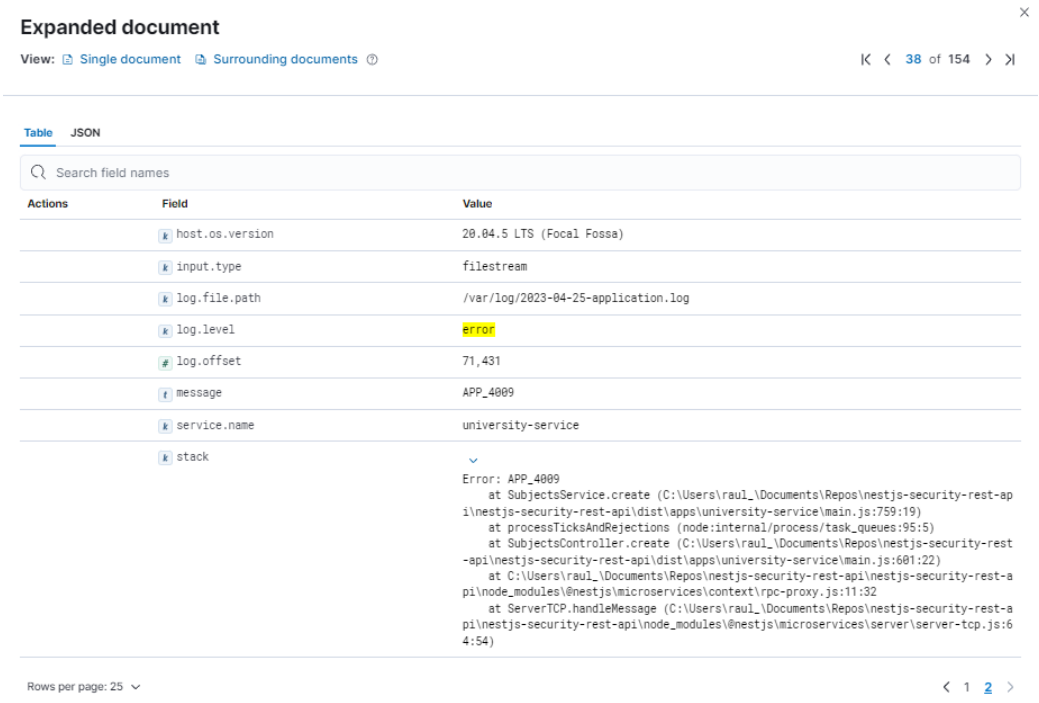


Figura 89: Expansión del documento de Kibana.

También en la parte de observabilidad podemos ver en streaming los logs de errores:

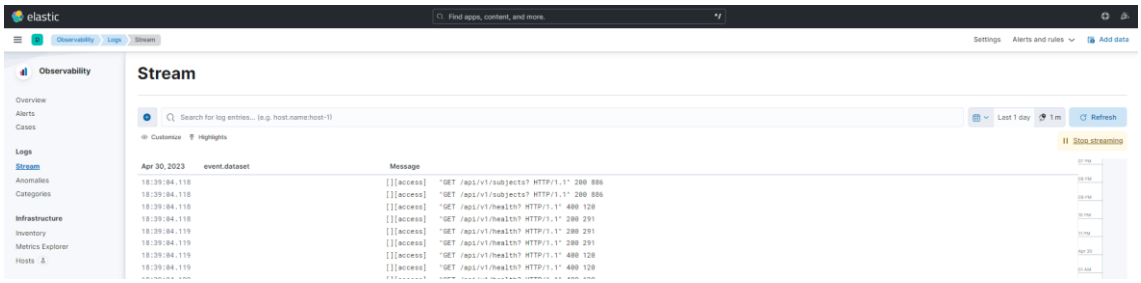


Figura 90: Streaming de logs en Kibana.

3.9. Despliegue

Para poder realizar el despliegue nos hará falta descargarnos el proyecto del repositorio de GitHub: https://github.com/raulruizbarea/nestjs-security-rest-api

Gracias a Docker y los archivos preparados en Docker-compose podemos realizar el despliegue de la API REST y API Gateway en un entorno en contenedores, para asegurar la implementación y escalabilidad.

El entorno debe estar bajo la misma red 'uocnetwork', aunque de cara a exponer los contenedores, solo nos interesa exponer el API Gateway y Kibana, dichos contenedores podrán obtener y/o consultar los datos de los otros al pertenecer a la misma red, añadiendo una capa de protección de acceso.

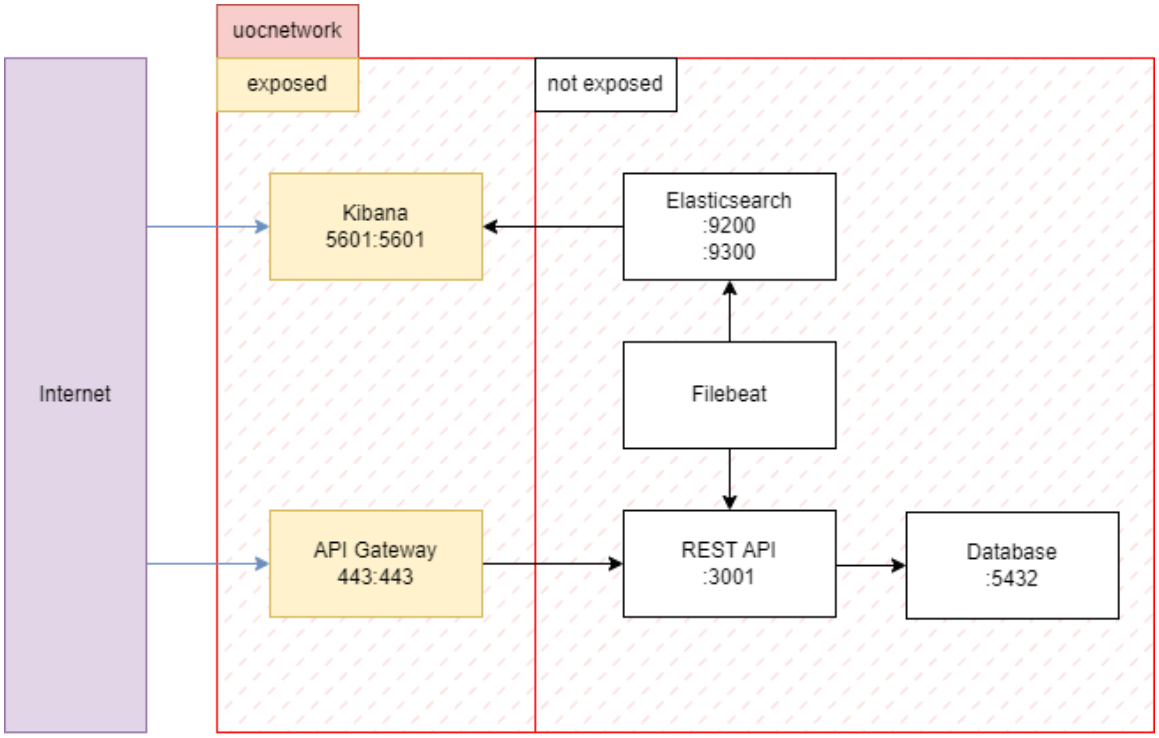
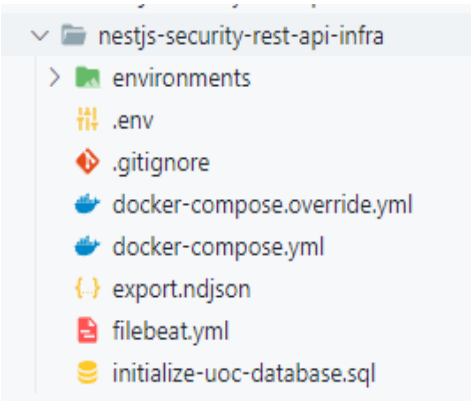


Figura 91: Arquitectura de contenedores de Docker.

En la carpeta ‘nestjs-security-rest-api-infra’ encontraremos todo lo necesario para desplegar y configurar Kibana.



En la carpeta environments encontraremos las variables de entorno de cada contenedor, el archivo .env contiene las versiones de las imágenes y los puertos de los contenedores, se ha preparado un script de inicialización de base de datos, además tenemos la exportación de los objetos de Kibana.

Finalmente tendremos los distintos docker-compose, que contienen toda la definición de los contenedores, configuración y limitaciones de recursos.

Para poder levantar la infrastructura ejecutaremos desde la ubicación del proyecto de infra: docker-compose up -d

```
Path
----
C:\Users\raul_\Documents\Repos\nestjs-security-rest-api\nestjs-security-rest-api-infra


→ C:\Users\raul_\Documents\Repos\nestjs-security-rest-api\nestjs-security-rest-api-infra [main ≡]› docker-compose up -d
[+] Running 6/6
 ✔ Container uoc-elasticsearch       Started
 ✔ Container uoc-kibana              Started
 ✔ Container uoc-filebeat            Started
 ✔ Container uoc-database            Started
 ✔ Container uoc-university-service  Started
 ✔ Container uoc-api-gateway         Started
→ C:\Users\raul_\Documents\Repos\nestjs-security-rest-api\nestjs-security-rest-api-infra [main ≡]›
```

Figura 92: Levantar infraestructura con Docker-compose.

Finalmente desde Docker Destkop podemos verificar que todos los contenedores están funcionando y los puertos que tenemos bindeados, comprobando que solo sea el API Gateway y Kibana.

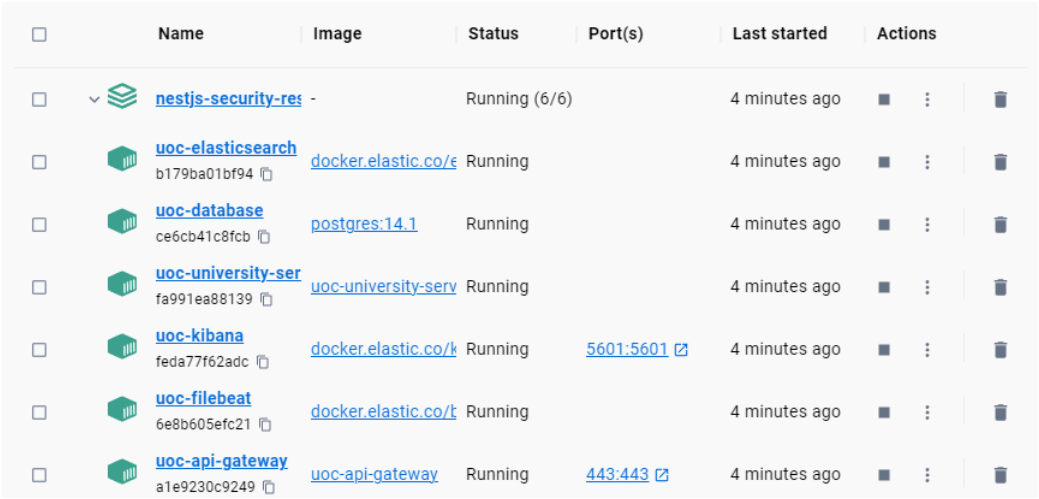


Figura 93: Contenedores Docker bajo proyecto nestjs-security-rest-api.

Para poder persistir la base de datos, los logs y la configuración de elasticsearch, se han generado los siguientes volumenes, los cuales nos servirán para poder hacer backups y validar que todo esté funcionando correctamente.

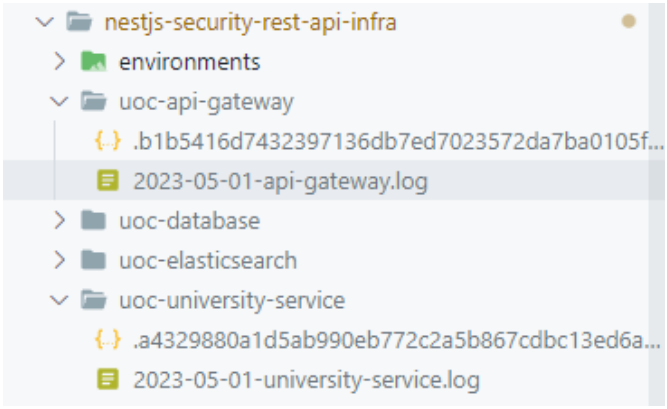


Figura 94: Persistencia de logs.

Desde Docker Desktop podemos validar que los logs de los contenedores del API Gateway y del API REST estén montados correctamente en la ubicación adecuada para que Filebeat los envíe a Elasticsearch.

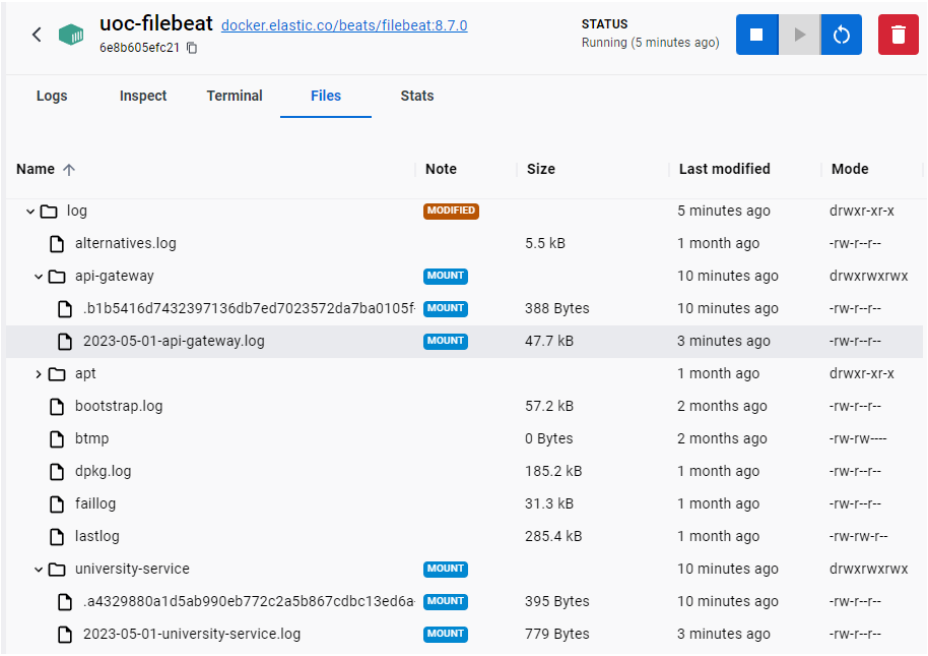


Figura 95: Exploración de logs dentro de los contenedores.

En el primer despliegue Kibana no tendrá ninguna configuración, deberemos ir a Stack Management y en el apartado de Saved Objects importar la configuración de export.ndjson como se indicó en apartados anteriores.

Figura 96: Importación de logs en Kibana.

Finalmente en el menú de Dashboard aparecerá “NestJS Security REST API dashboard”:

Figura 97: Dashboard importado en Kibana.

La configuración permitirá ver los logs en stream, la primera prueba que podemos hacer es desde el API Gateway (https://localhost/api) realizar la creación de una asignatura, y volver a intentar crearla con el mismo código.

Gracias a esta prueba veremos que el primer POST se genera correctamente (estado 201) y el siguiente dará conflicto (estado 409) además de un error APP_4009 que nos servirá para identificar que ha sucedido.

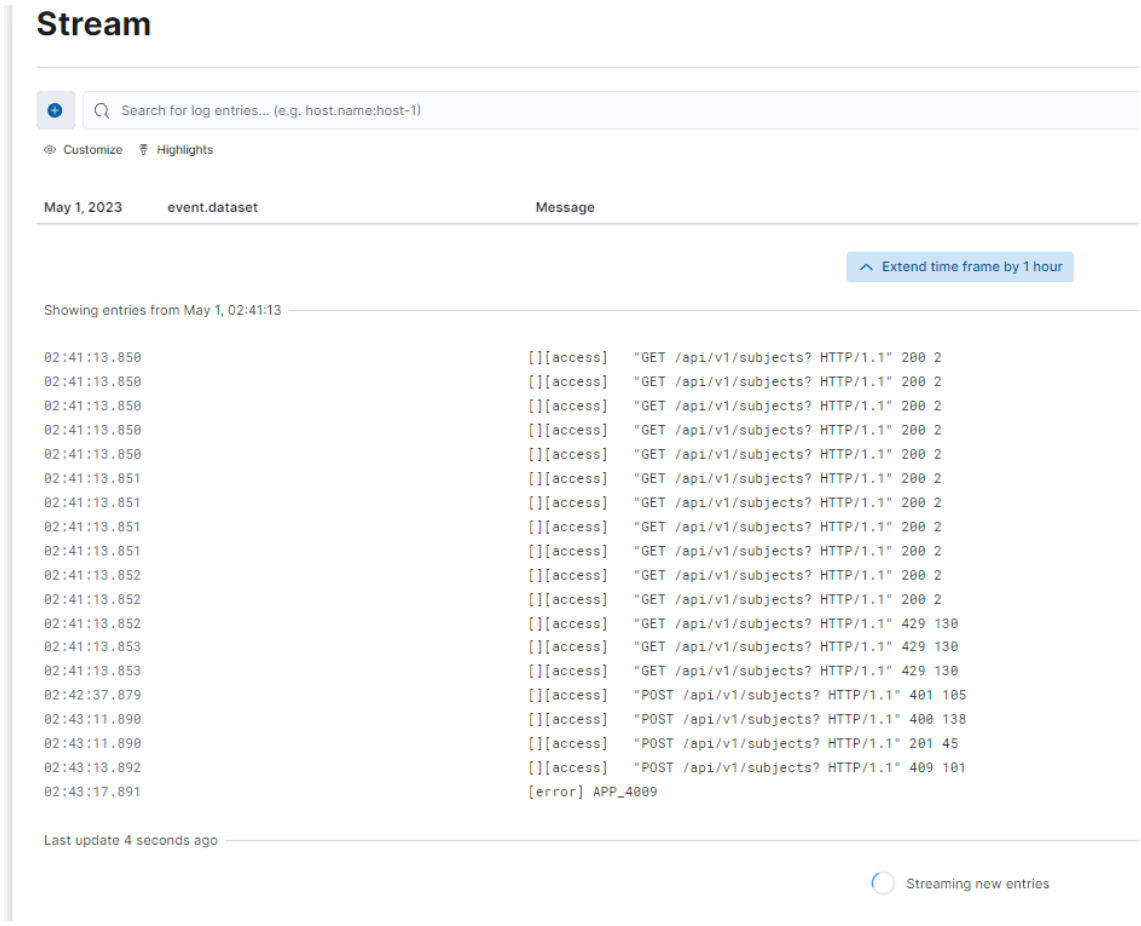


Figura 98: Streaming de logs en Kibana.

4. Implementación de controles de seguridad

Antes de entrar con los 10 principales riesgos de seguridad de las API propuesta por OWASP, debemos tener en cuenta algunos principios de desarrollo.

Deberemos sincronizar las versiones y sus dependencias a nivel de package.json, suele ser una buena práctica quitar los carets (^) para utilizar siempre versiones concretas de las librerías, también debemos prestar atención a las versiones de las imágenes de los contenedores ya que pueden incluir vulnerabilidades.

API1:2019 Broken Object Level Authorization

Debemos establecer el control de acceso a nivel de asignaturas, por lo tanto dos usuarios con el mismo rol, únicamente podrán modificar el objeto si ellos son los creadores de él.

Validamos que el user_id de Auth0 es exactamente el recibido por el token:

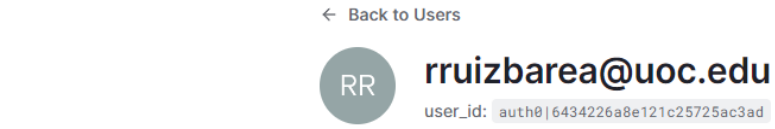


Figura 99: Identificador de usuario en Auth0.

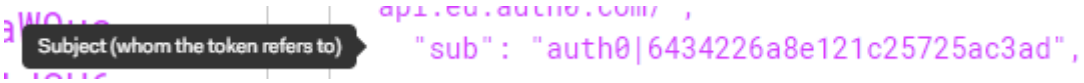


Figura 100: Subject del token obtenido de Auth0.

Para poder validar la información del token, creamos un decorador y obtendremos toda la información necesaria, el identificador de usuario lo obtendremos del Subject.

nestjs-security-rest-api - user.decorator.ts

```
7  export const AuthUser = createParamDecorator((data, ctx: ExecutionContext) => {
8    const req = ctx.switchToHttp().getRequest();
9    const token = req.headers.authorization.split(' ')[1];
10   const tokenInfo: any = jwt.decode(token);
11   const userDto: UserDto = new UserDto();
12   userDto.email = tokenInfo['https://nestjs-security-rest-api/email'];
13   userDto.roles = tokenInfo['https://nestjs-security-rest-api/roles'];
14   userDto.userId = tokenInfo.sub;
15   return userDto;
16 });
```

Figura 101: Obtención de los datos de usuario del token a través de un decorador.

Deberemos aplicar el decorador en los endpoints necesarios para parsear el token y obtener los datos del usuario que necesitemos.

nestjs-security-rest-api - subjects.controller.ts

```
69  @AuthUser(new ValidationPipe({ validateCustomDecorators: true }))
70      userDto: UserDto,
```

Figura 102: Aplicación del decorador custom.

A nivel de Base de Datos si el identificador del usuario que creó la asignatura no es el mismo que se encuentra en el Subject del token no dejaremos continuar con la operación y devolveremos un no autorizado.

nestjs-security-rest-api - subject-typeorm.repository.ts

```
119  if (
120    subjectDao.createdBy !== userDto.userId
121    //&& !userDto?.roles.some((role) => role === 'Admin')
122  ) {
123    throw new RpcException({
124      message: APP_EXCEPTION.UNAUTHORIZED,
125      statusCode: HttpStatusCode.Unauthorized,
126    });
127  }
```

Figura 103: Verificación del usuario creador con el usuario que realiza la petición.

Uso aleatorio e impredecible de UUIDs para los identificadores de las entidades, para no facilitar la exposición de datos evitaremos identificadores sencillos e incrementales, ya que un atacante podría realizar peticiones GET para obtener todos los datos de las asignaturas.

nestjs-security-rest-api - subject.dao.ts

```
15  @PrimaryColumn({ type: 'uuid', generated: 'uuid' })
16    id: string;
```

Figura 104: Columna primaria autogenerada en formato uuid.

```
{
  "id": "3d1dd357-8a34-4713-85e4-ab29cd65dcc5"
}
```

Además trabajaremos a nivel de código (code) de asignaturas y no de id, para no exponer identificadores internos.

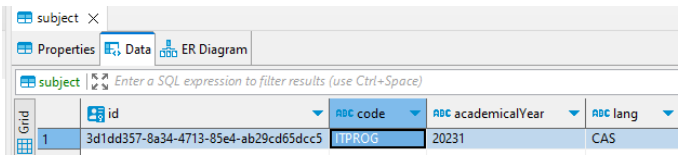


Figura 105: Exploración de columnas de base de datos.

Una parte importante es la realización de pruebas para validar las respuestas y no introducir errores inesperados. Para ello deberemos importar la definición del open api generada.

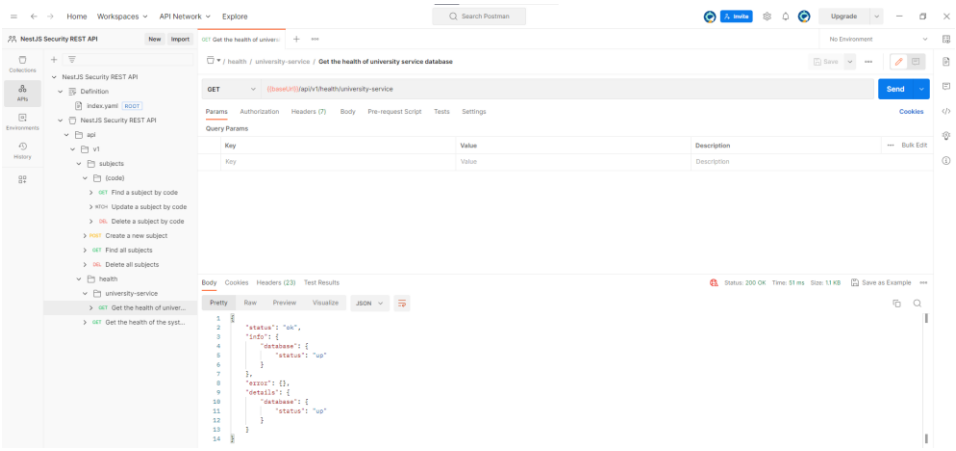


Figura 106: Ejemplo de prueba a través de Postman.

Podemos comprobar que utiliza variables de entorno de Postman como baseUrl, además todos los endpoints heredan la autorización del padre.

En el siguiente ejemplo vemos como el endpoint de salud hereda la autorización y utiliza las variables de entorno.

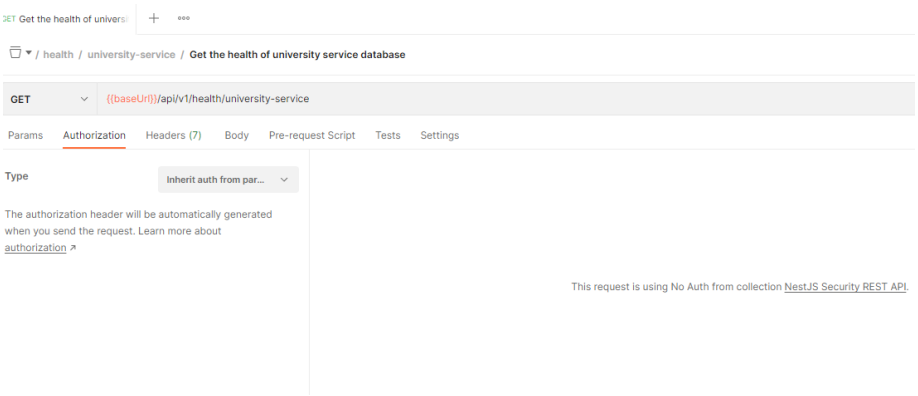


Figura 107: Configuración autorización de Postman.

En la pestaña de variables encontraremos la URL definida.

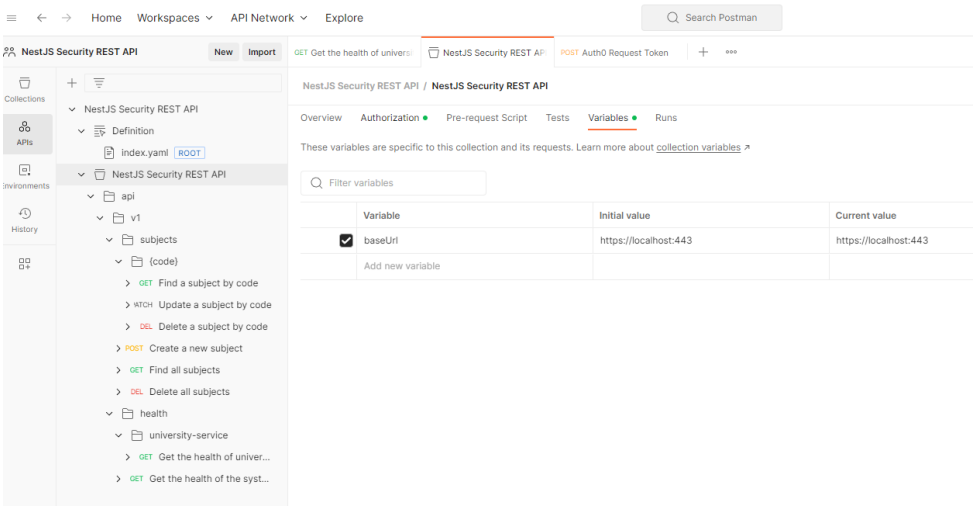


Figura 108: Variables de Postman.

Para que podamos indicar el bearer token, deberemos añadir en la raíz el token, ya que por defecto nos indica que no hay.

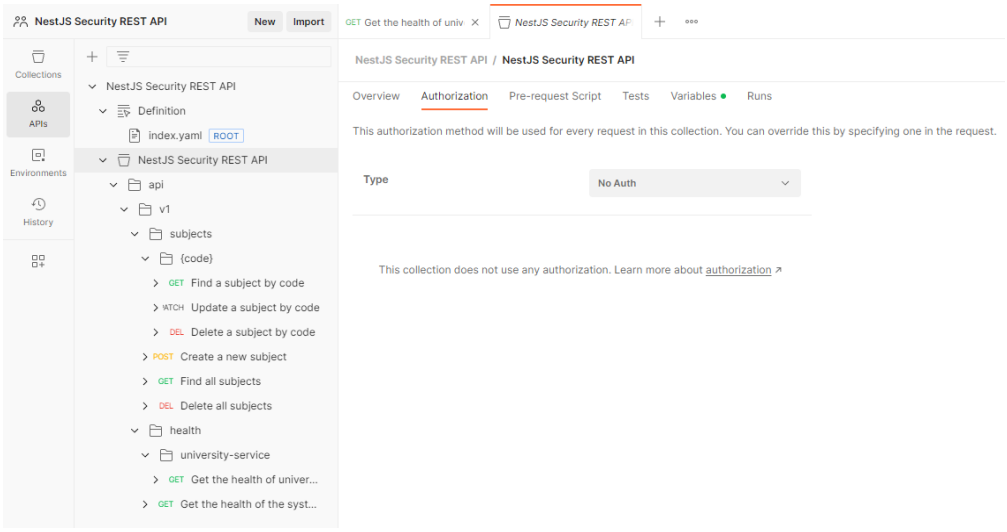


Figura 109: Configuración inicial de la raíz del API en Postman.

Una vez indicamos el Bearer Token que conseguimos de Auth0, al insertarlo en este apartado, será ingestado directamente en todos los endpoints.

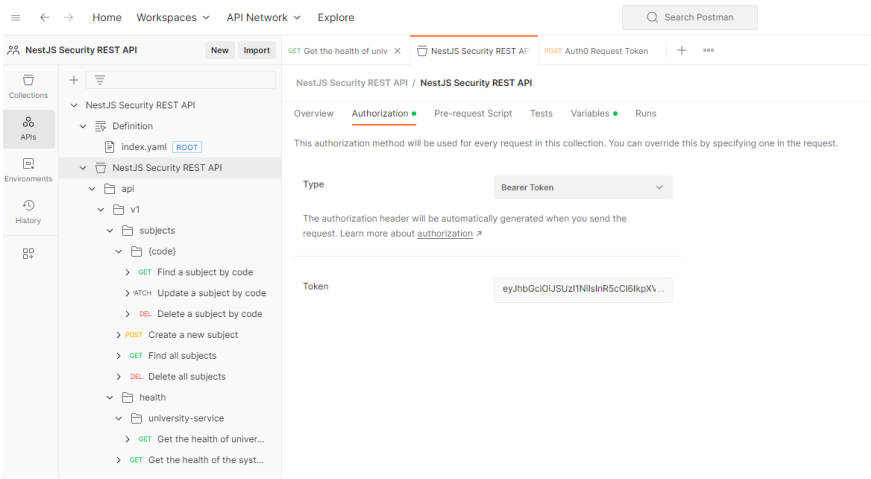


Figura 110: Configuración del token de la raíz del API en Postman.

Ahora ya podremos hacer las llamadas apropiadas, debemos tener en cuenta que si modificamos los roles o queremos probar otros usuarios, deberemos actualizar el token anterior. En este ejemplo, heredamos el token de un usuario con permisos y la baseUrl indicada.

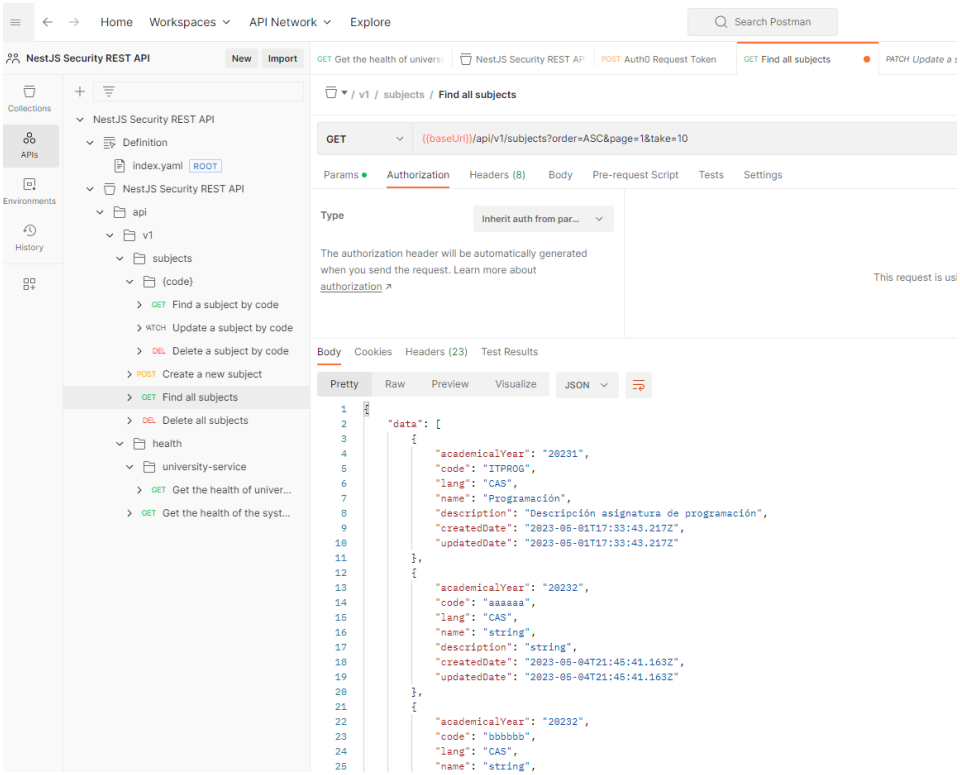


Figura 111: Petición de prueba utilizando el token heredado en Postman.

Para terminar las pruebas, se pueden definir Tests a nivel de endpoints, se definirán diferentes test, los cuales validaremos estados devueltos, tipo de datos, incluso el mensaje de respuesta, estos test pueden automatizarse sobretodo para cuando hay nuevas versiones del API.

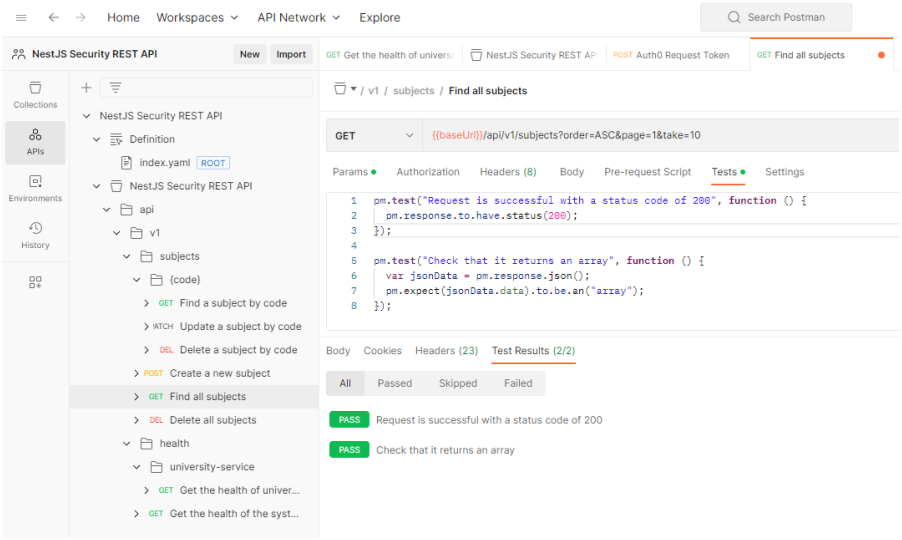


Figura 112: Definición de tests en Postman.

API2:2019 Broken User Authentication

A nivel de autenticación derivamos toda la responsabilidad a Auth0, en nuestro caso de uso la configuración la realizaremos de manera manual a niveles de controles de seguridad y de gestión de usuarios.

Uno de los elementos claves para cualquier aplicación es establecer la autenticación multi-factor, validando a través de otro dispositivo la identidad del usuario.

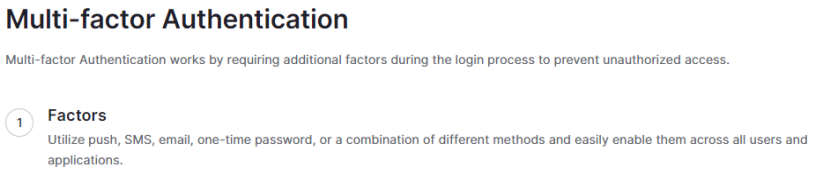


Figura 113: Autenticación multi-factor en Auth0.

Además nos facilita un apartado de seguridad con las vulnerabilidades más comunes, ofreciendo el bloqueo de cuenta (account lockout) si se detectan ataques de fuerza bruta o superación de los límites de peticiones de una misma ubicación.

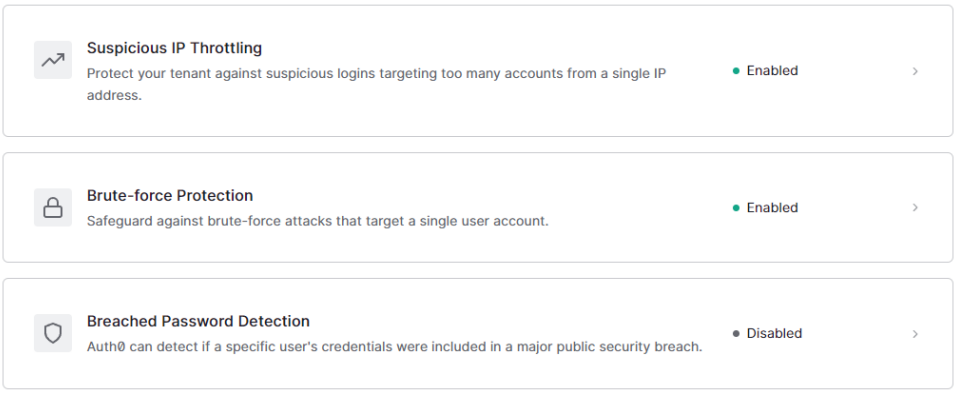


Figura 114: Configuración de seguridad en Auth0.

En Auth0 podemos gestionar la dificultad del password, tener un diccionario y no permitir que contengan valores de otros campos o metadatos. Para no reinventar la rueda OWASP nos recomienda hacer uso de las recomendaciones que nos facilitan.

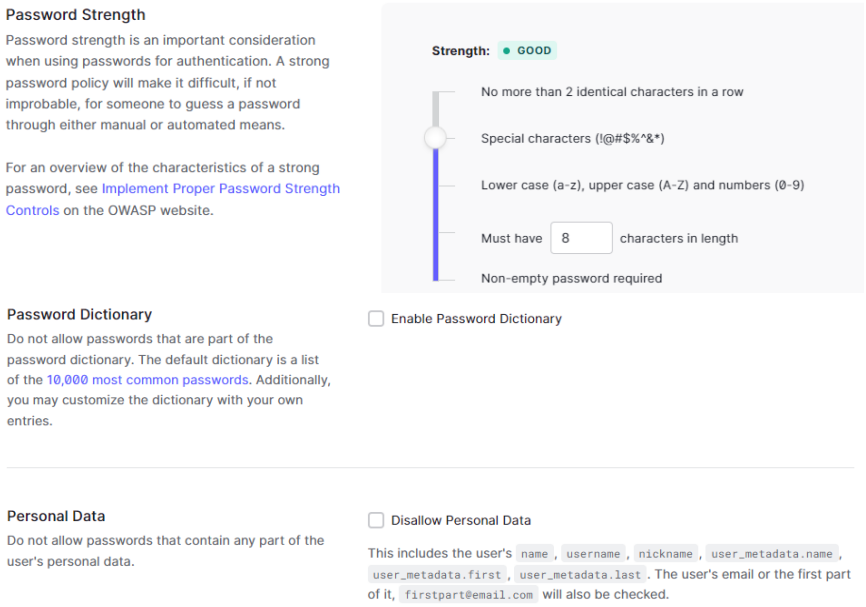


Figura 115: Configuración de contraseñas en Auth0.

A nivel de token de acceso, podemos gestionar cuando van a expirar, a través de que métodos pueden recuperar el token, etc.

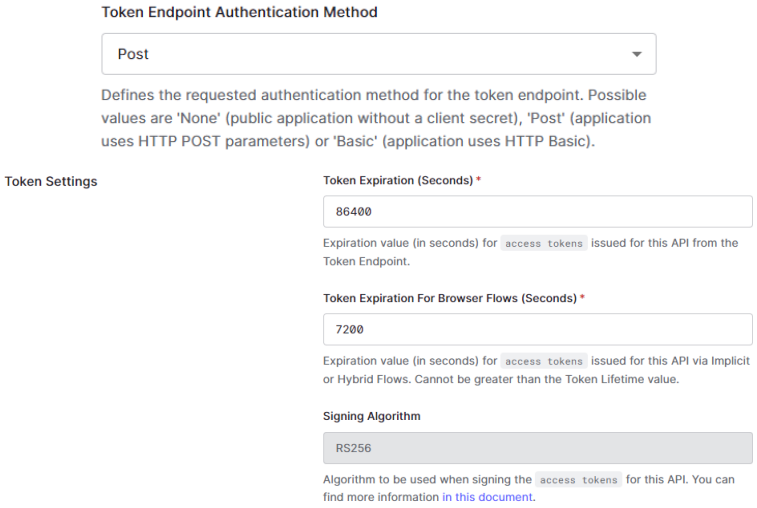


Figura 116: Configuración del token en Auth0.

La creación de usuarios la realizaremos de manera manual desde la gestión de usuarios, añadiendo un usuario con email y password.

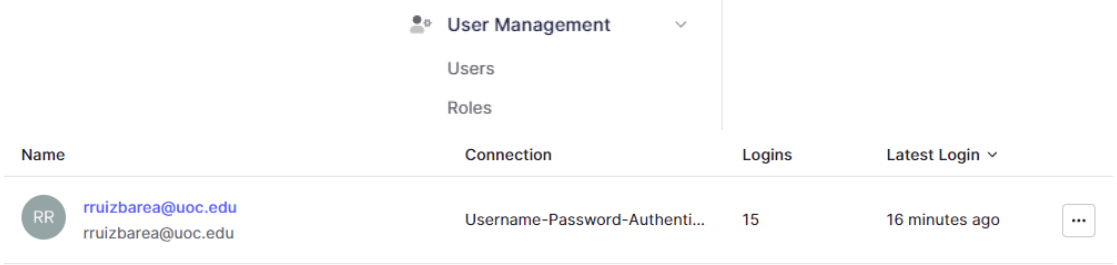


Figura 117: Gestión de usuarios en Auth0.

Para poder realizar las pruebas en el API requerimos del access_token que proporciona Auth0, para ello obtendremos los datos de cliente de la aplicación y la audiencia del API que creamos anteriormente, junto al usuario y password del usuario.

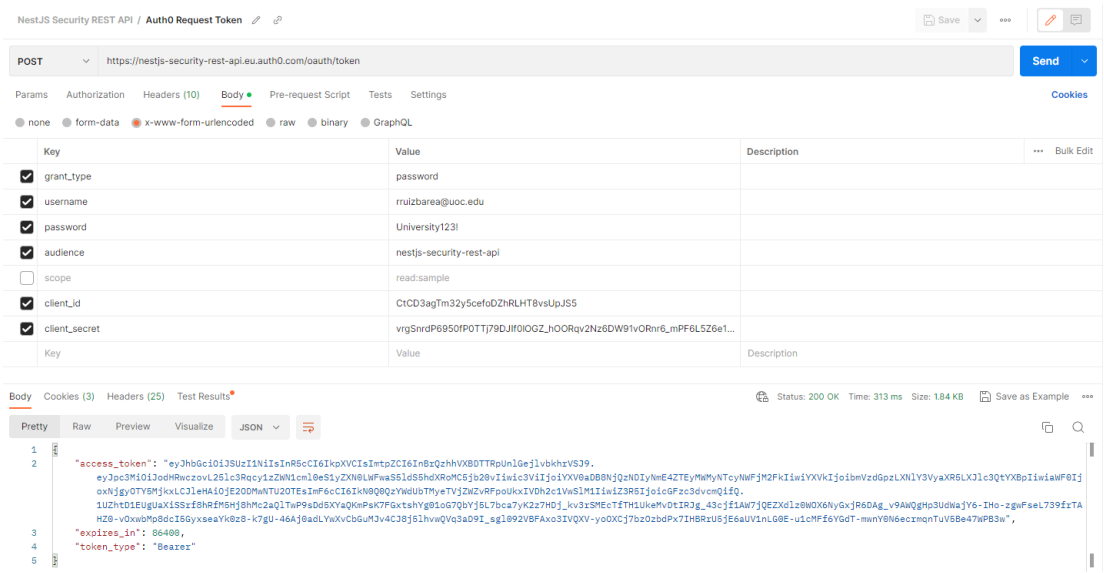


Figura 118: Obtención del token de Auth0 en Postman.

Podemos comprobar el contenido del token obtenido en: https://jwt.io/ vemos que dividen la información en cabeceras, payload y validación de la firma.

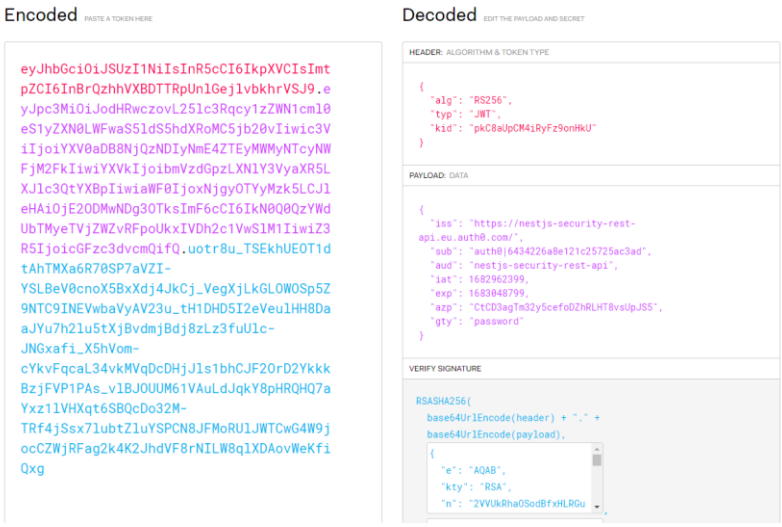


Figura 119: Descodificación del token a través de jwt.io.

Para poder tratar el token, debemos crear un modulo de autenticación, indicando la estrategia que vamos a seguir (en nuestro caso 'jwt').

nestjs-security-rest-api - auth.module.ts

```
@Module({
  imports: [PassportModule.register({ defaultStrategy: 'jwt' })],
  providers: [JwtStrategy],
  exports: [PassportModule],
})
export class AuthModule {}
```

Figura 120: Módulo de autenticación aplicando estrategia jwt.

Dicha estrategia valida el token y en caso contrario devolvemos un 'Unauthorized'.

nestjs-security-rest-api - jwt.strategy.ts

```
@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(private readonly configService: ConfigService) {
    super({
      secretOrKeyProvider: passportJwtSecret({
        cache: true,
        rateLimit: true,
        jwksRequestsPerMinute: 5,
        jwksUri: `${configService.get('auth0.issuerUrl')}.well-known/jwks.json`,
      }),

      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      audience: configService.get('auth0.audience'),
      issuer: configService.get('auth0.issuerUrl'),
      algorithms: ['RS256'],
    });
  }

  async validate(payload: any): Promise<any> {
    if (!payload) {
      throw new HttpException('Invalid token', HttpStatus.UNAUTHORIZED);
    }

    return payload;
  }
}
```

Figura 121: Configuración de desencriptación del token de Auth0.

Para que pueda acerse efectivo debemos añadir las guardas necesarias, en este caso guarda de autenticación de ‘jwt’, se han añadido guardas en todos los endpoints exceptuando los health checks y las búsquedas.

nestjs-security-rest-api - subjects.controller.ts

```
54  @UseGuards(AuthGuard('jwt'))
55    @Post()
56    create(
57      @Body() createSubjectDto: CreateSubjectDto,
58    ): Observable<CreateSubjectResponseDto>
```

Figura 122: Aplicación de las guardas de jwt en los distintos endpoints.

Ejemplo sin introducir el bearer token:

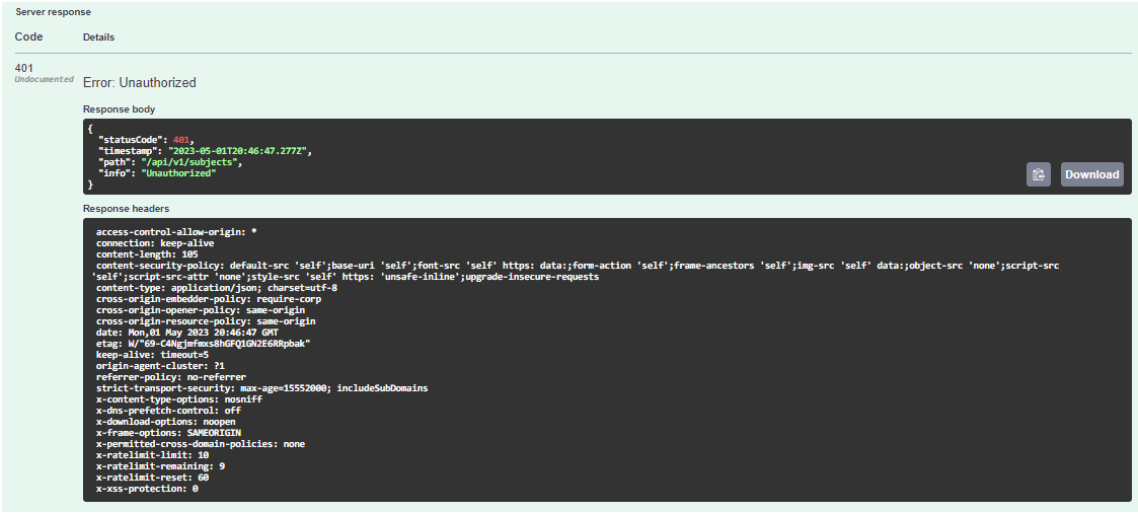


Figura 123: Respuesta petición sin token en Swagger.

Ejemplo con bearer token aleatorio:

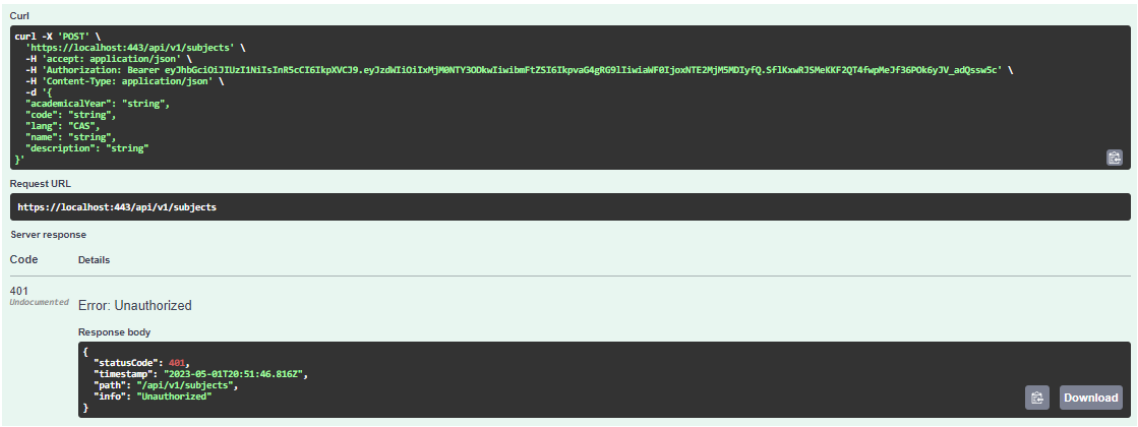


Figura 124: Respuesta petición con token erróneo en Swagger.

API3:2019 Excessive Data Exposure

Para no exponer datos innecesarios en los controladores del API Gateway definiremos en el @Body que tipo de objeto recibimos y cual devolvemos, es

de vital importancia control que tipo de datos retornamos al usuario ya que no queremos filtrar información confidencial.

nestjs-security-rest-api - subjects.controller.ts

```
55  @Post()
56    create(
57      @Body() createSubjectDto: CreateSubjectDto,
58    ): Observable<CreateSubjectResponseDto>
```

Figura 125: Aplicación de DTOs en el Body de una petición.

Cuando aplica el validation pipe, automaticamente controla las validaciones que hayamos indicado a través de la librería 'class-validator'.

nestjs-security-rest-api - create-subject.dto.ts

```
2  import {
3    IsEnum,
4    IsNotEmpty,
5    IsString,
6    Length,
7    MaxLength,
8  } from 'class-validator';
```

Figura 126: Validaciones de la librería class-validator.

Aplica de abajo a arriba, debemos validar el tipo de dato y el contenido de este, según el caso de uso deberemos limpiar espacios en blanco al inicio o fin de cadenas de carácteres.

Para cadenas de caracteres siempre deberemos tener en cuenta el tamaño e indicar un mensaje que sea amigable al usuario final, se establecerán una configuración global para entidades con el tamaño de cada columna si aplica.

nestjs-security-rest-api - create-subject.dto.ts

```
29  @ApiProperty()
30  @Escape()
31  @Length(SubjectSettings.CODE_LENGTH, SubjectSettings.CODE_LENGTH, {
32    message: `code must be exactly ${SubjectSettings.CODE_LENGTH} characters`,
33  })
34  @Trim()
35  @IsString()
36  @IsNotEmpty()
37  code: string;
```

Figura 127: Aplicación de restricciones de longitud de cadenas.

En el caso de la respuesta, queremos que sean readonly y no se pueda tratar.

nestjs-security-rest-api - create-subject-

```
6   @ApiProperty()
7   @IsString()
8   @IsNotEmpty()
9   readonly id: string;
10
11  constructor(id: string) {
12    this.id = id;
13  }
```

Figura 128: Aplicación de solo lectura para evitar la manipulación de los identificadores.

A nivel de microservicio, enlaza la petición a través del patrón de mensaje, teniendo la petición y respuesta del mismo tipo del API Gateway, en este ejemplo comprobamos que retornamos el objeto esperado.

nestjs-security-rest-api - subjects.controller.ts

```
15  @MessagePattern(SubjectMessagePatternsName.CREATE)
16  async create(
17    @Payload() payload: CreateSubjectDto,
18  ): Promise<CreateSubjectResponseDto> {
19    const code: string = await this.subjectsService.create(
20      Subject.fromDto(payload),
21    );
22
23    return new CreateSubjectResponseDto(code);
24  }
```

Figura 129: Retorno de la respuesta esperada en el microservicio.

A nivel de validación nos indicará todos los errores que hayamos introducido a la hora de realizar la petición:

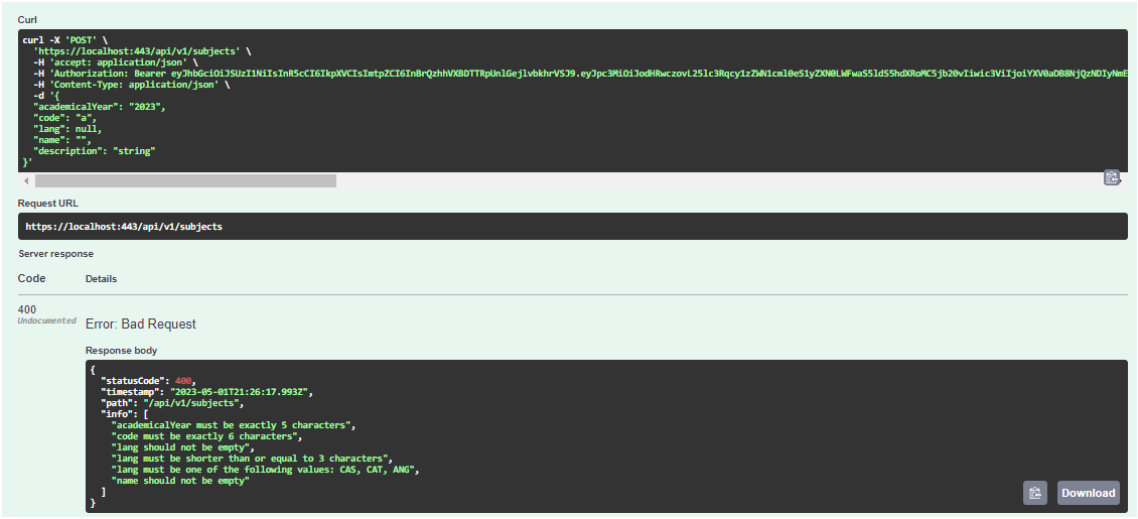


Figura 130: Validación de los errores del body enviado en Swagger.

Estaremos siempre controlando que queremos recibir y devolver, evitando riesgos de filtración de datos, además, podemos extender clases para añadir datos auto generados de base de datos.

A nivel de errores, se ha establecido un estándar de devolver, el código de estado, el momento en el que sucedió, el path de la petición y una información general que ayude a identificar la causa raíz pero no ponga en riesgo la seguridad de la API REST.

nestjs-security-rest-api - rpc-exception.filter.ts

```
27  response.status(_error.statusCode).json({
28        statusCode: _error.statusCode,
29        timestamp: timestamp,
30        path: request.url,
31        info: _error.message,
32      });
```

Figura 131: Estándar de errores.

API4:2019 Lack of Resources & Rate Limiting

Debemos limitar los recursos de las máquinas en las cuales desplegamos la solución y aplicar un reinicio cuando haya un fallo con un máximo de intentos para que no se quede reiniciando eternamente.

A nivel de recursos podemos limitar las CPUs y la memoria, reservando un tamaño y poniendo unos límites para que se paren si exceden (o escalen en el caso de Kubernetes).

También podemos limitar el tamaño de disco reservado, y debemos asegurarnos de que las máquinas pertenezcan a la misma red, dándoles un alias para que puedan resolver la IP que asigne Docker al contenedor.

nestjs-security-rest-api - docker-compose.yml

```
 8  deploy:
 9    restart_policy:
10      condition: on-failure
11      delay: 5s
12      max_attempts: 3
13      window: 120s
14    resources:
15      limits:
16        cpus: '2'
17        memory: '2g'
18      reservations:
19        cpus: '1'
20        memory: '1g'
21  storage_opt:
22    size: '4g'
23  networks:
24    uocnetwork:
25      aliases:
26        - uoc-database
```

Figura 132: Configuración de los límites de los contenedores.

A nivel de contenedores es una buena practica exponer internamente los puertos que son necesarios:

nestjs-security-rest-api - docker-compose.override.yml

```
39  uoc-elasticsearch:
40    container_name: 'uoc-elasticsearch'
41    expose:
42      - '9200'
43      - '9300'
```

Figura 133: Exposición de los puertos internamente en Docker-compose.

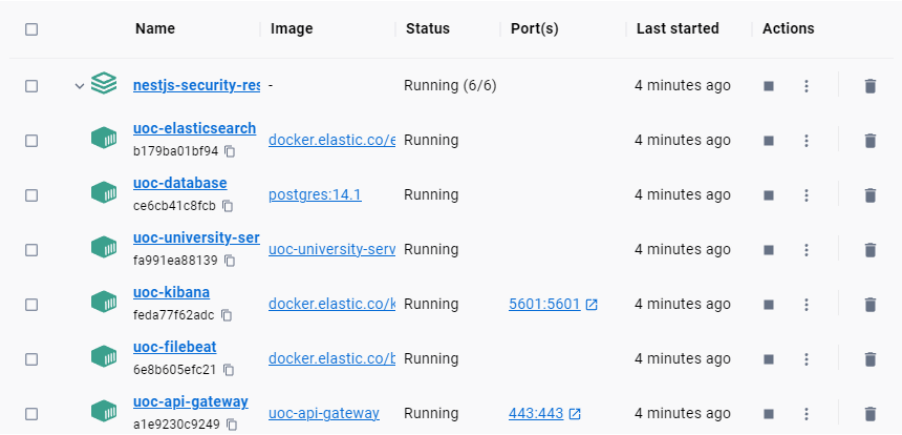
Y únicamente bindear al exterior los puertos del API Gateway y Kibana:

nestjs-security-rest-api - docker-compose.override.yml

```
46  uoc-kibana:
47    container_name: 'uoc-kibana'
48    ports:
49      - '5601:5601'
```

Figura 134: Exposición de los puertos al exterior en Docker-compose.

La fotografía final de como quedarían los distintos contenedores con los puertos expuestos:



Name	Image	Status	Port(s)	Last started	Actions
nestjs-security-res	-	Running (6/6)		4 minutes ago	
uoc-elasticsearch b179ba01bf94	docker.elastic.co/e	Running		4 minutes ago	
uoc-database ce6cb41c8fcb	postgres:14.1	Running		4 minutes ago	
uoc-university-ser fa991ea88139	uoc-university-serv	Running		4 minutes ago	
uoc-kibana feda77f62adc	docker.elastic.co/k	Running	5601:5601	4 minutes ago	
uoc-filebeat 6e8b605efc21	docker.elastic.co/b	Running		4 minutes ago	
uoc-api-gateway a1e9230c9249	uoc-api-gateway	Running	443:443	4 minutes ago	

Figura 135: Configuración contenedores con puertos en Docker.

Para poder bloquear ataques de fuerza bruta, haremos uso de los rate limits que propone NestJS con la librería de 'throttler', estableceremos unas variables de entorno para establecer dichos límites: `THROTTLE_TTL=60 THROTTLE_LIMIT=10`

nestjs-security-rest-api - api-gateway.module.ts

```
82  ThrottlerModule.forRootAsync({
83    imports: [ConfigModule],
84    inject: [ConfigService],
85    useFactory: async (configService: ConfigService) => ({
86      ttl: configService.get('throttle.ttl'),
87      limit: configService.get('throttle.limit'),
88    }),
89  }),
```

Figura 136: Configuración de los rate limits.

Se debe notificar al cliente cuando se excede el límite de peticiones dentro del tiempo establecido, para ello capturaremos las excepciones de Throttler para darles el mismo formato que a todas las excepciones y enriquecer el mensaje al cliente. Además es buena práctica añadir una cabezera con el 'Retry-After' y el tiempo establecido.

nestjs-security-rest-api - throttler-exception.filter.ts

```
11  @Catch(ThrottlerException)
12  export class ThrottlerExceptionFilter implements ExceptionFilter {
13    constructor(private readonly configService: ConfigService) {}
14
15    catch(exception: ThrottlerException, host: ArgumentsHost) {
16      const ttl = this.configService.get('throttle.ttl');
17      const limit = this.configService.get('throttle.limit');
18      const timestamp = new Date().toISOString();
19      const ctx = host.switchToHttp();
20      const request = ctx.getRequest<Request>();
21      const response = ctx.getResponse();
22
23      response.setHeader('Retry-After', ttl.toString());
24
25      response.status(HttpStatus.TOO_MANY_REQUESTS).json({
26        statusCode: HttpStatus.TOO_MANY_REQUESTS,
27        timestamp: timestamp,
28        path: request.url,
29        info: `Too many requests, limit: ${limit}, ttl: ${ttl}`,
30      });
31    }
32  }
33
```

Figura 137: Captura y adaptación de las excepciones de los rate limits.

El cliente recibirá la siguiente información cuando exceda del límite peticiones en el tiempo asignado:

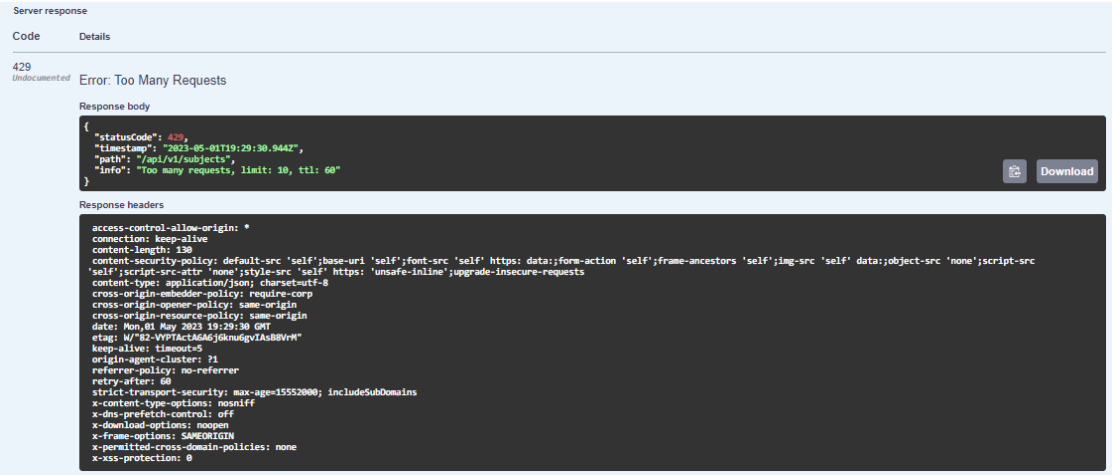


Figura 138: Respuesta excediendo los límites en Swagger.

En NestJS el tamaño límite de cuerpo está acoplado con el motor Express, establecido a 100kb, en el caso que debamos cambiarlo tendremos que configurarlo a nivel de aplicación.

Uno de los principales riesgos, es obtener datos sin limitaciones, estas pueden sobrecargar el sistema, o exponer toda la base de datos debido a cualquier tipo de vulnerabilidad. Para poder afrontar la paginación, tendremos a través de la query, tres parámetros: el orden (ascendiente o descendiente por fecha de creación), la página y la cantidad de elementos que queremos obtener.

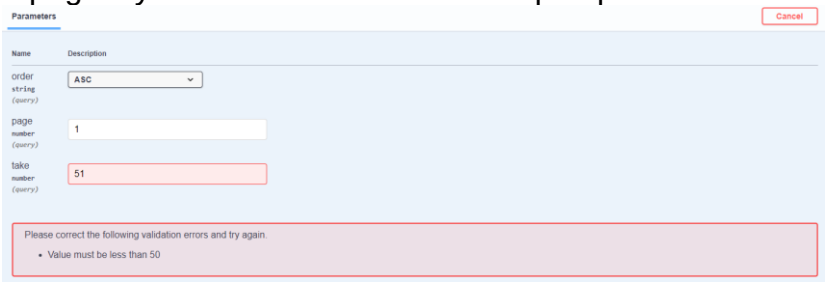


Figura 139: Validación del query string con valores por defecto en Swagger.

Dichos campos deben tener valores por defecto y validaciones, para que no puedan hacerse con toda la información, como resultado devolveremos como metadatos: la página, la cantidad, el total elementos y el total páginas.

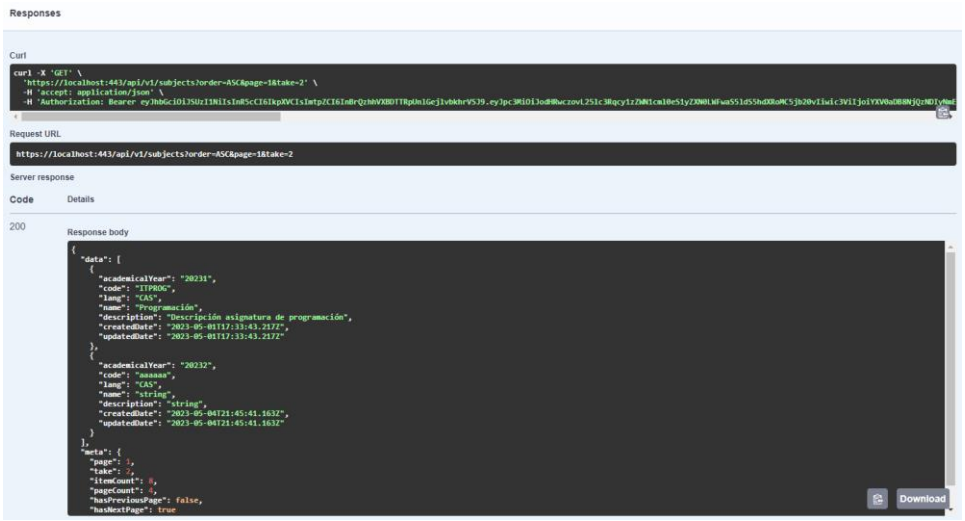


Figura 140: Respuesta con paginación en Swagger.

El resultado que obtenemos por defecto en Postman sin indicar el query string será de la página 1 con 10 elementos.

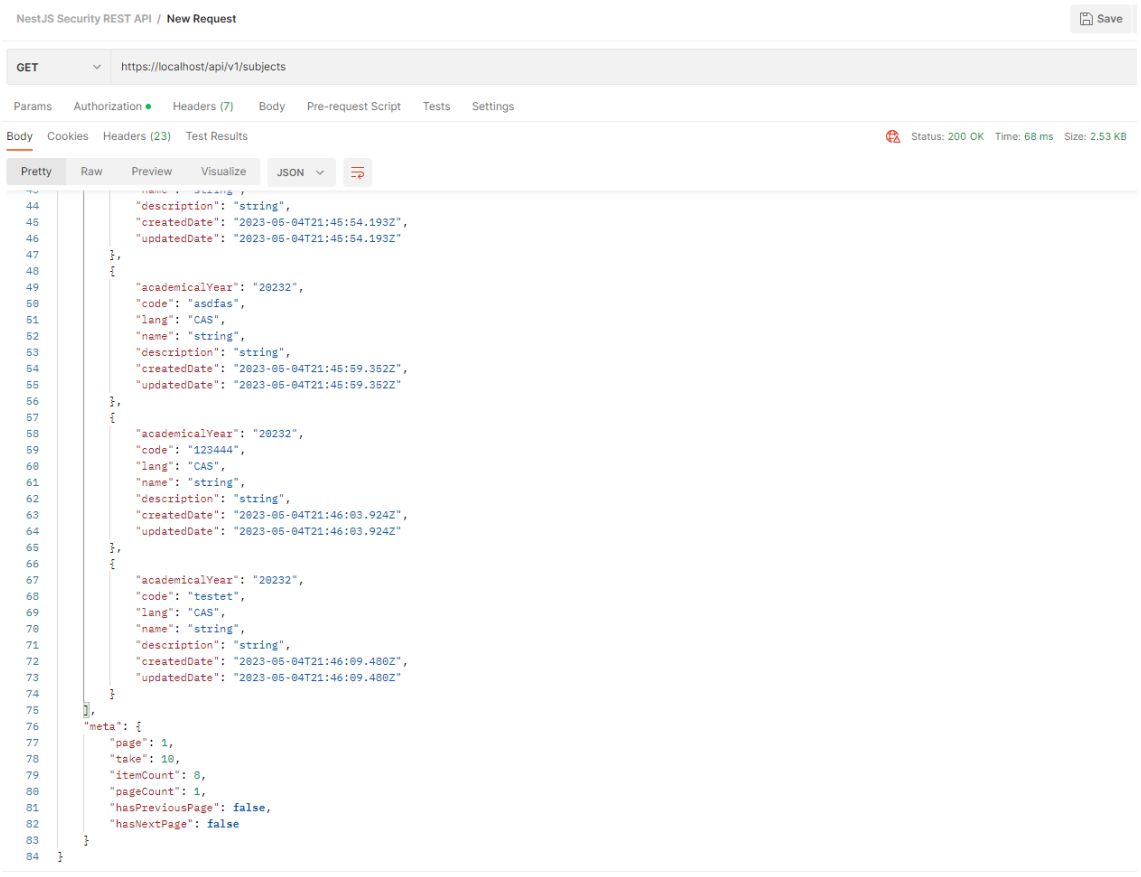


Figura 141: Respuesta con paginación con valores por defecto en Postman.

Comprobamos que únicamente indicando uno de los tres parámetros, el resultado se adapta de forma correcta.

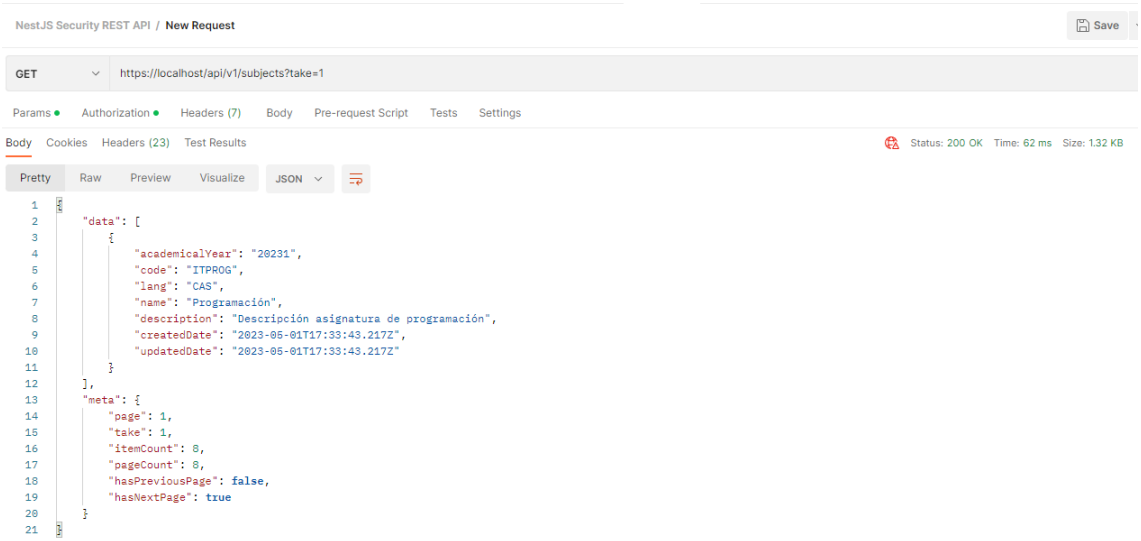


Figura 142: Respuesta con paginación únicamente con cantidad de registros en Postman.

API5:2019 Broken Function Level Authorization

Para poder empezar a desarrollar la parte de roles, debemos configurar Auth0 para admitir en el API de una lista de permisos la cual crearemos un CRUD a nivel de asignaturas (create, read, update y delete).

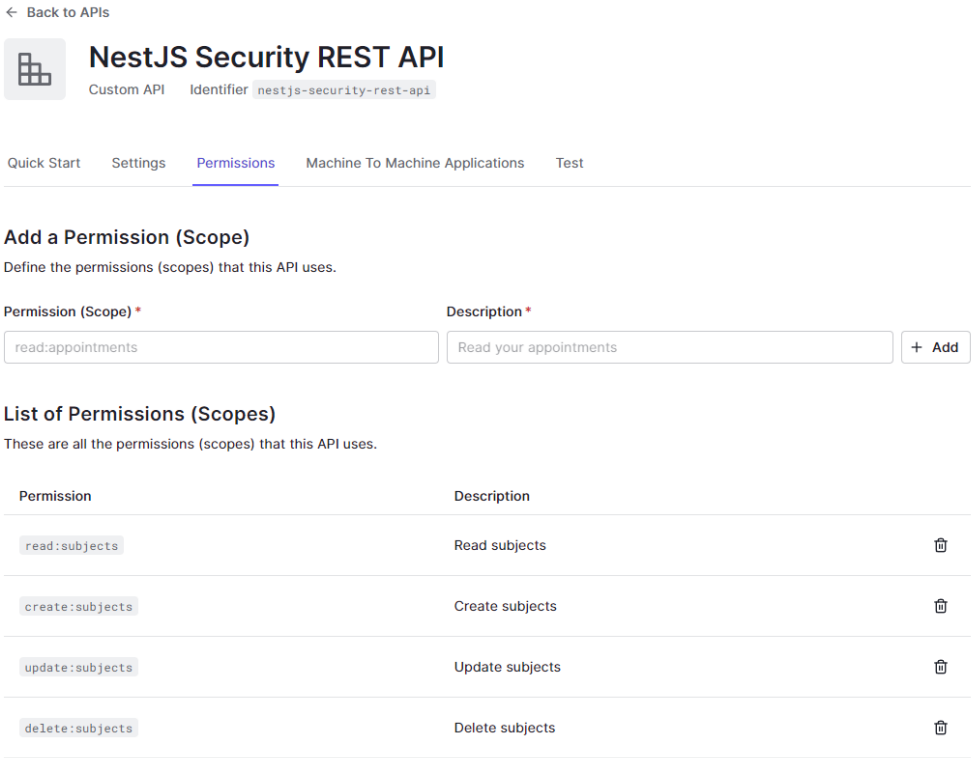


Figura 143: Listado de permisos en Auth0.

Acto seguido deberemos activar la configuración de RBAC (Role-based access control) y añadir los permisos a los token.

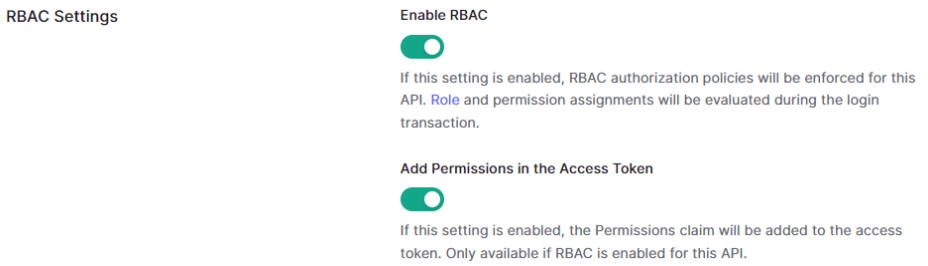


Figura 144: Activación de RBAC y permisos en Auth0.

En el apartado de roles podremos crear dos roles para hacer pruebas, un administrador y un usuario plano.

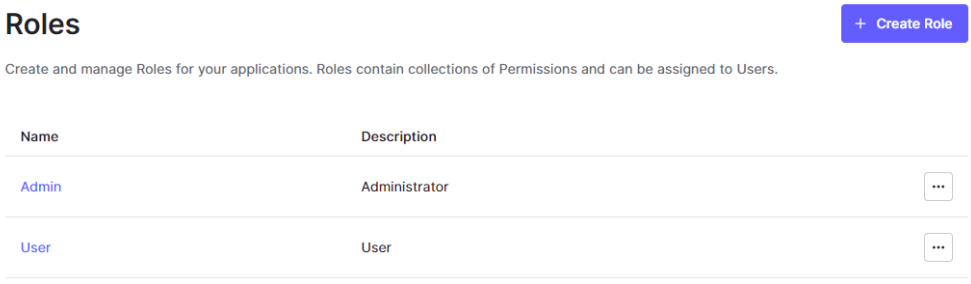


Figura 145: Listado de roles en Auth0.

El siguiente paso será asignar los permisos a los roles, el rol administrador tendrá todas las actividades posibles.

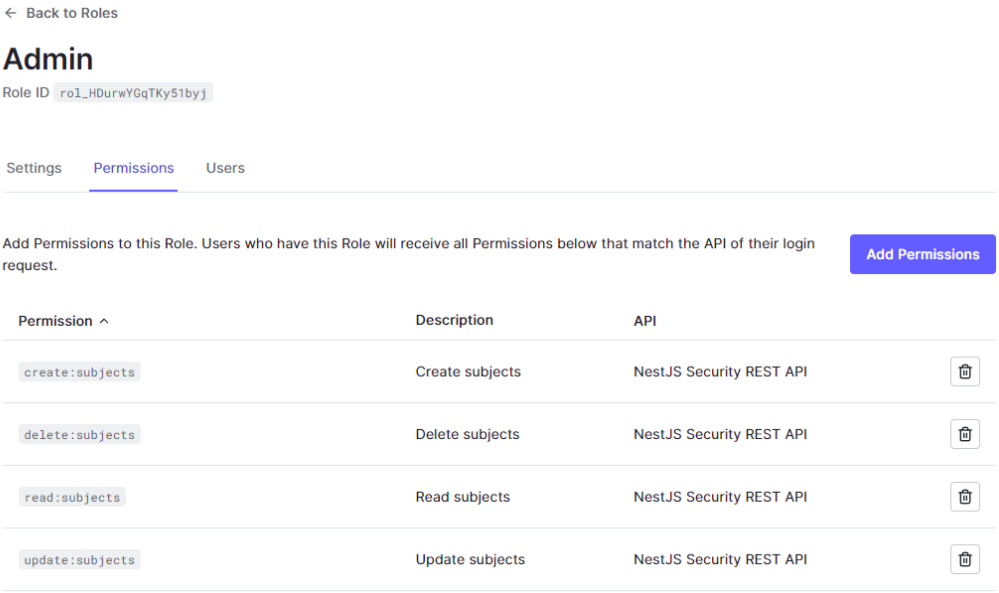


Figura 146: Permisos asignados del Admin en Auth0.

El rol usuario solo tendrá permisos de lectura.

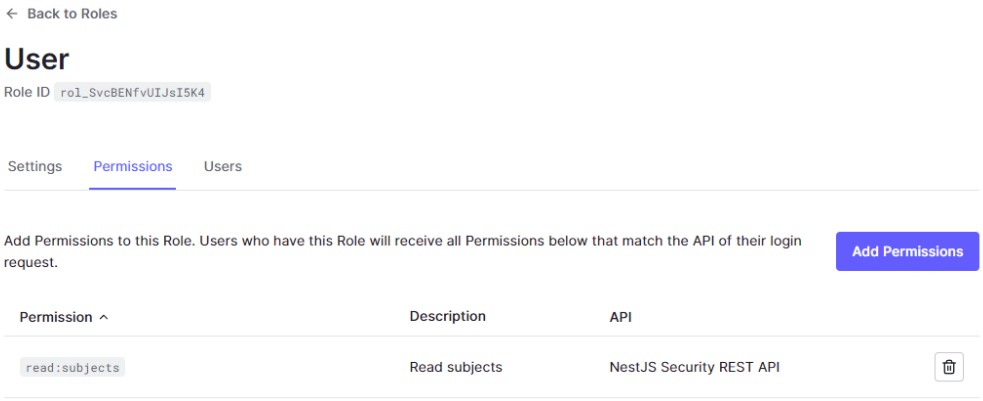


Figura 147: Permisos asignados del User en Auth0.

Para poder gestionar los permisos del token, tendremos que definir un decorador de permisos, el cual tendrá un array de los permisos que contiene dicho rol.

nestjs-security-rest-api - permissions.decorator.ts

```ts
import { SetMetadata } from '@nestjs/common';

export const Permissions = (...args: string[]) =>
  SetMetadata('permissions', args);

```

Figura 148: Decorador de permisos.

Deberemos crear una guarda, la cual activará los endpoints que coincidan con los permisos del usuario, en caso contrario no podremos acceder.

nestjs-security-rest-api - permissions.guard.ts

```ts
@Injectable()
export class PermissionsGuard implements CanActivate {
  constructor(private readonly reflector: Reflector) {}

  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
    const routePermissions = this.reflector.get<string[]>(
      'permissions',
      context.getHandler(),
    );

    const userPermissions = context.getArgs()[0].user.permissions;

    if (!routePermissions) {
      return true;
    }

    const hasPermission = () =>
      routePermissions.every((routePermission) =>
        userPermissions.includes(routePermission),
      );

    return hasPermission();
  }
}

```

Figura 149: Validación de los permisos del usuario coinciden con los del endpoint.

Para aplicar las guardas, utilizaremos la guarda del jwt que teníamos anteriormente añadiendo la guarda de permisos, que contendría los permisos del rol asignado en Auth0.

Entonces ahora vincularemos los distintos endpoints con los permisos que deben tener para poderlos lanzar, en este ejemplo unicamente podrán ejecutarlo los que contengan el permiso de 'read:subjects', entonces cuando validemos el token, si encuentra dicho permiso podrá realizar la petición correctamente.

nestjs-security-rest-api - subjects.controller.ts

```
107  @UseGuards(AuthGuard('jwt'), PermissionsGuard)
108  @Get()
109  @Permissions('read:subjects')
110  findAll(
111    @Query() pageOptionsDto: PageOptionsDto,
112  ): Observable<PageDto<SubjectResponseDto>> {
113    return this.subjectsService.findAll(pageOptionsDto);
114  }
```

Figura 150: Aplicación de las guardas y permisos para un endpoint.

Inicialmente el token sin tener rol asignado, devolverá un array de permisos vacío.

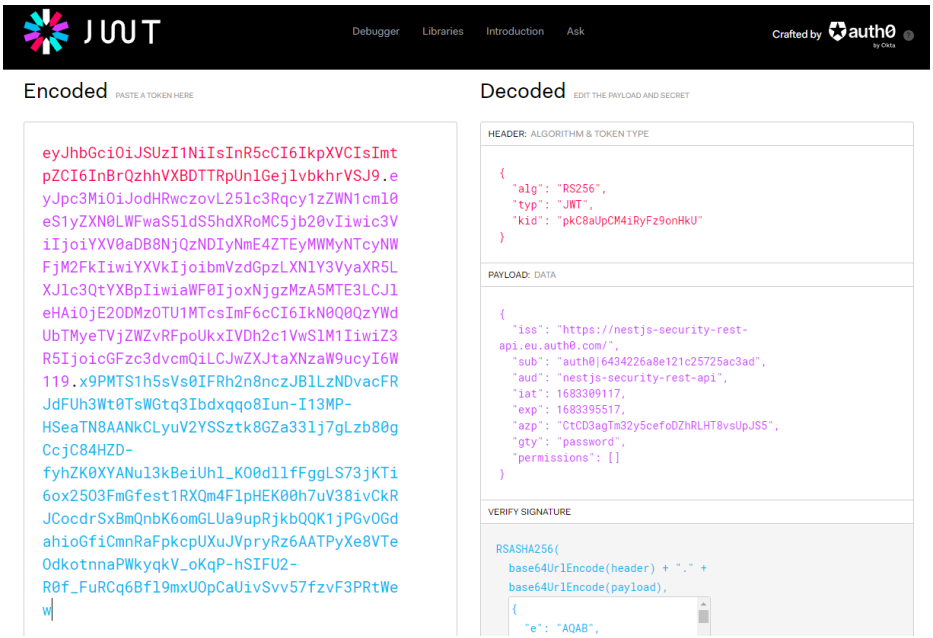


Figura 151: Decodificación de token con usuario sin rol.

Para asignar un rol tendremos que ir a Auth0 y en el usuario en el apartado de roles asignarle el rol adecuado, para este ejemplo añadiremos el rol de Usuario que únicamente tiene permisos de lectura.

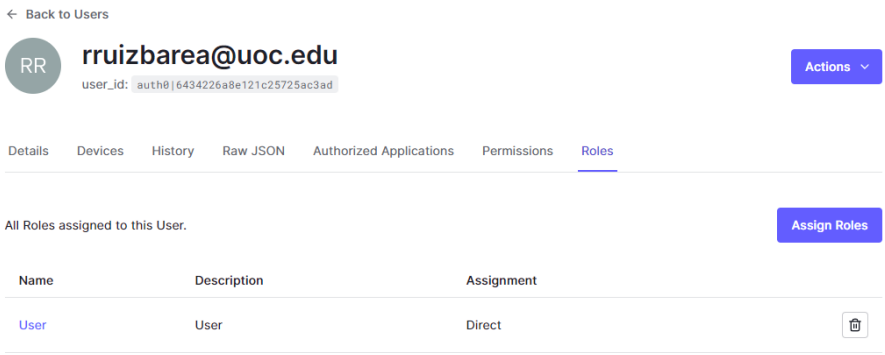


Figura 152: Asignación de roles en Auth0.

Finalmente debemos obtener otra vez el token que ya heredará los permisos adecuados.

Figura 153: Obtención del token con nuevo rol en Postman.

Ahora el token contendrá los permisos del rol de Usuario read-subjects.

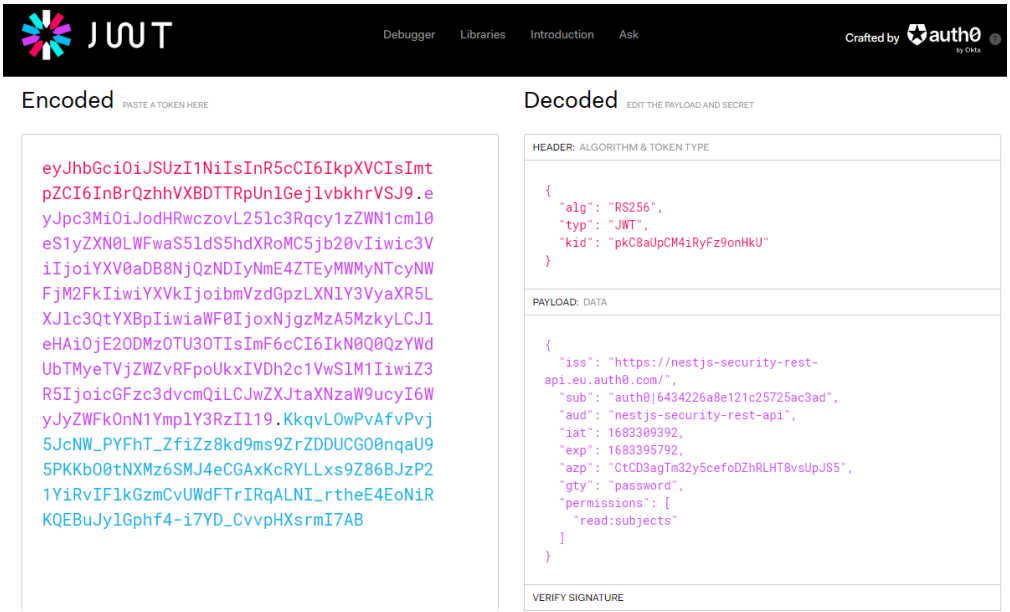


Figura 154: Decodificación de token con rol y permisos.

Validamos que podemos obtener las asignaturas con el rol de Usuario.

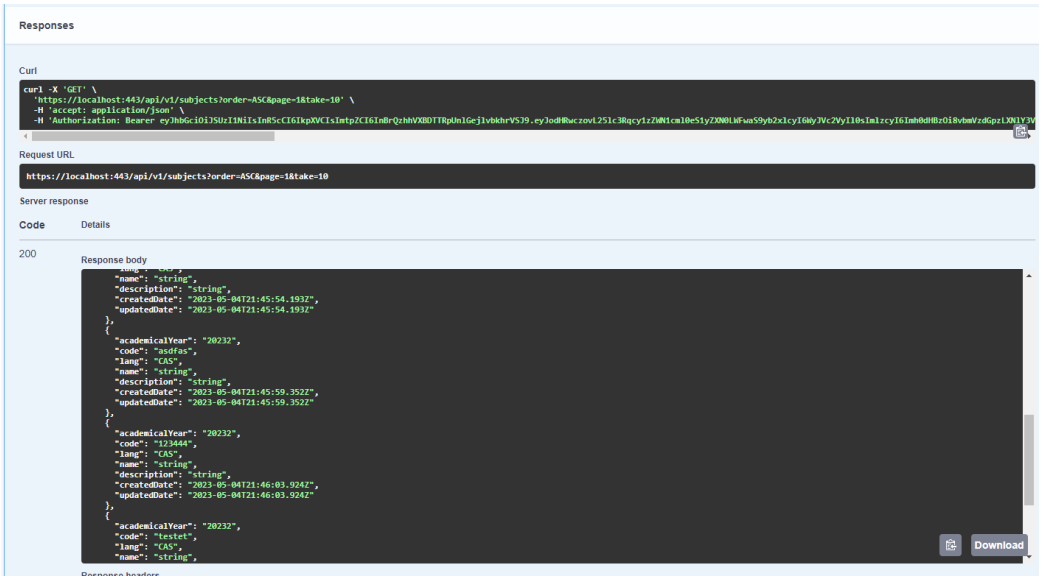


Figura 155: Petición y respuesta validando el token y permisos de usuario.

A diferencia si intentamos crear una asignatura con el rol de Usuario, no nos permitirá realizar la acción y retornará un estado 403 Forbidden resource.

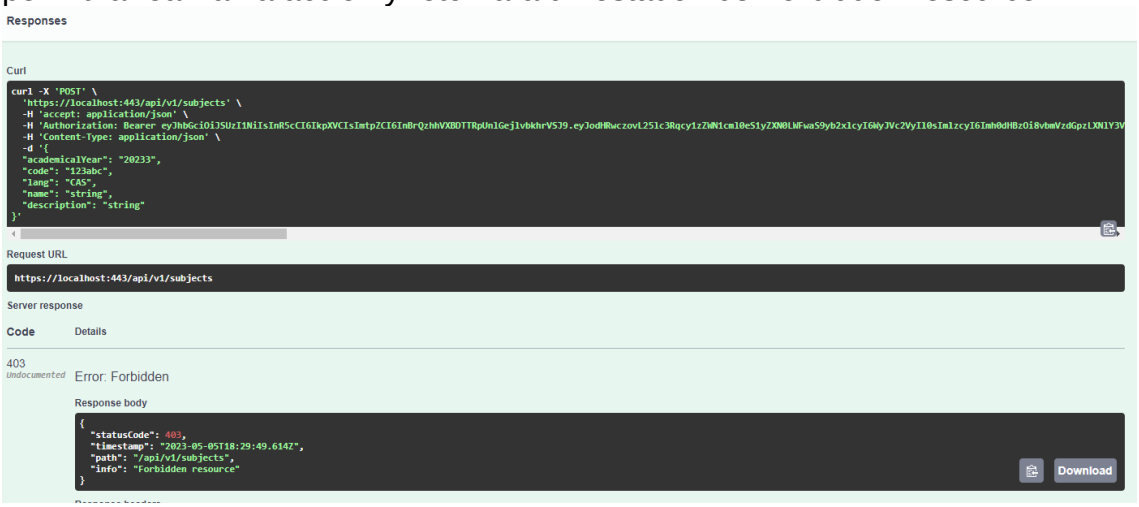


Figura 156: Petición y respuesta con token y sin permisos de usuario.

Validaremos que podemos tener distintos roles a nivel de usuario, añadiremos el rol de Administrador en el usuario creado.

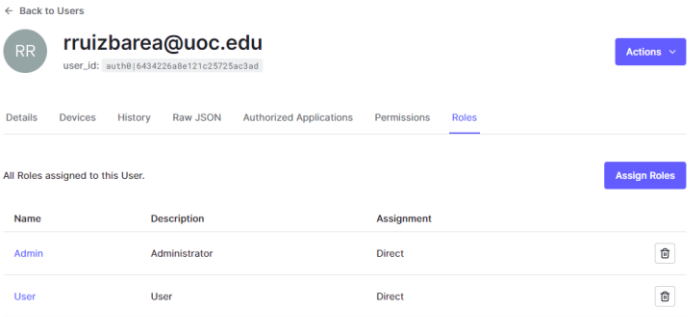


Figura 157: Asignación de varios roles a un usuario en Auth0.

Finalmente vemos que el token, contiene los 2 roles indicados (Admin y User), además del array de permisos coincidentes.

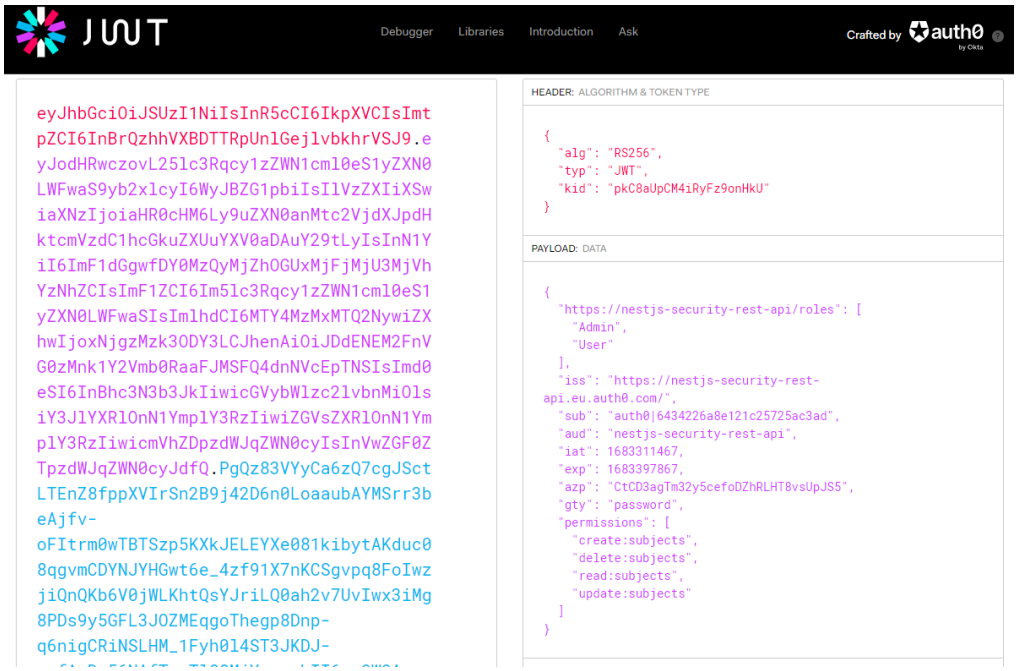


Figura 158: Decodificación del token con varios roles a un usuario.

Renovando el token ya podremos crear asignaturas.

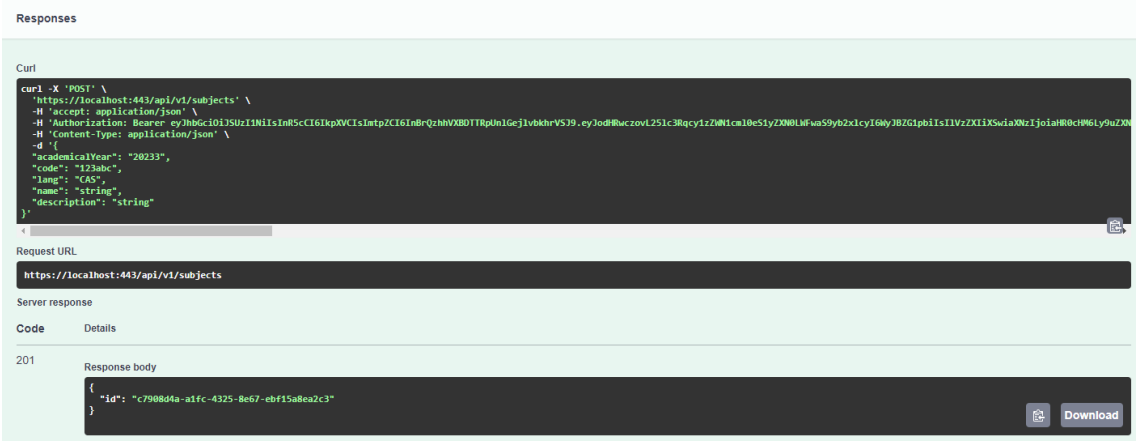


Figura 159: Petición y respuesta con token y permisos válidos.

API6:2019 Mass Assignment

NestJS nos ofrece pipes globales de validación, que podemos configurar como mejor nos convenga, una configuración que cumpla con los estandares de seguridad, sería realizar un whitelist de las propiedades que no esperamos, mostrando exactamente un error indicando cuales son, para facilitar al cliente la identificación del problema. Por otro lado, queremos que transforme automaticamente la entrada a los tipos que hayamos creado.

nestjs-security-rest-api - main.ts

```
27  app.useGlobalPipes(
28      new ValidationPipe({
29        //* properties that don't use any validator decorator automatically removed and throw an exception
30        whitelist: true,
31        forbidNonWhitelisted: true,
32        //* transform payload objects to dto
33        transform: true,
34      }),
35    );
```

Figura 160: Validación de los cuerpos de las peticiones.

En el caso que enviemos una propiedad no esperada, nos indicará que hay una propiedad que no debería existir.

Figura 161: Respuesta de petición con cuerpo erróneo en Swagger.

En el swagger podemos ver todos los schemas de definición con las propiedades asociadas.

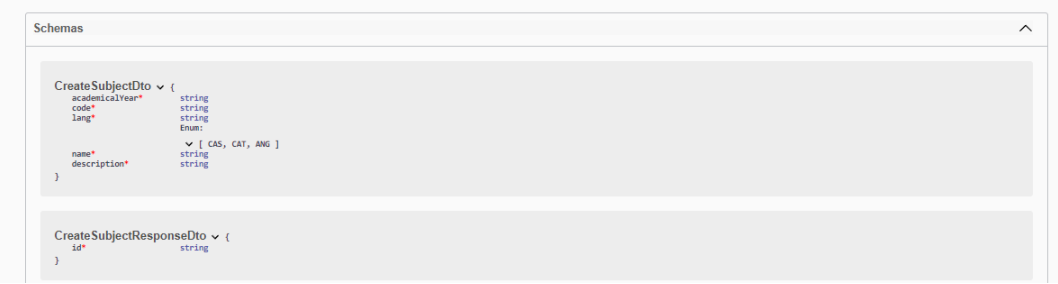


Figura 162: Esquemas de definición en Swagger.

Debemos tener en cuenta que nuestros repositorios solo entienden de entidades, la transformación correrá sobre unos métodos estáticos y a la hora de devolver generaremos las respuestas adecuadas, como por ejemplo:

nestjs-security-rest-api - subject.entity.ts

```
39  static fromDto(
40    createSubjectDto: CreateSubjectDto | UpdateSubjectDto,
41  ): Subject {
42    const subject = new Subject(
43      createSubjectDto.academicalYear,
44      createSubjectDto.code,
45      createSubjectDto.lang,
46      createSubjectDto.name,
47      createSubjectDto.description,
48    );
49
50    return subject;
51  }
52
53  static toDto(subject: Subject): SubjectResponseDto {
54    const subjectResponseDto: SubjectResponseDto = new SubjectResponseDto
    ();
55    subjectResponseDto.academicalYear = subject.academicalYear;
56    subjectResponseDto.code = subject.code;
57    subjectResponseDto.lang = subject.lang;
58    subjectResponseDto.name = subject.name;
59    subjectResponseDto.description = subject.description;
60    subjectResponseDto.createdDate = subject.createdDate;
61    subjectResponseDto.updatedDate = subject.updatedDate;
62
63    return subjectResponseDto;
64  }
```

Figura 163: Transformación de objetos a/desde DTO.

En este contexto no queremos que el usuario final pueda manejar las fechas de creación y actualización, pero si queremos que puedan recibirlas cuando consulten asignaturas, para en el futuro poder ordenar o sacar informes.

nestjs-security-rest-api - subject-response.dto.ts

```
6   export class SubjectResponseDto extends CreateSubjectDto {
7     @ApiProperty()
8     @IsDate()
9     @IsNotEmpty()
10    createdDate: Date;
11    @ApiProperty()
12    @IsDate()
13    @IsNotEmpty()
14    updatedDate: Date;
15  }
```

Figura 164: Extensión respuesta añadiendo campos adicionales.

API7:2019 Security Misconfiguration

Debemos manejar apropiadamente la configuración de las variables de entorno, para que el sistema construido se pueda manejar correctamente además de obtener cierto grado de robustez.

Para ello se importa la configuración de las variables de entorno y se cargan según el modo en el que estemos gracias a la variable NODE_ENV, construiremos un esquema y validaremos todas las variables de entorno, indicando un error si nos falta una variable y quitando las no deseadas.

nestjs-security-rest-api - university.module.ts

```
45  ConfigModule.forRoot({
46    load: [configuration],
47    isGlobal: true,
48    envFilePath: [
49      `apps/university-service/src/environments/.env.${process.env.NODE_ENV}`,
50    ],
51    validationSchema: envSchema,
52    validationOptions: {
53      abortEarly: true,
54      allowUnknown: false,
55      stripUnknown: true,
56    },
57  }),
```

Figura 165: Configuración y validación de las variables de entorno.

A nivel de configuración organizaremos las variables de entorno en formato JSON para poder manejarlas adecuadamente:

nestjs-security-rest-api - configuration.ts

```
1   export default () => ({
2     environment: process.env.NODE_ENV,
3     name: process.env.APP_NAME,
4     port: parseInt(process.env.PORT),
5     university: {
6       host: process.env.UNIVERSITY_SERVICE_HOST,
7       port: parseInt(process.env.UNIVERSITY_SERVICE_PORT),
8     },
9     database: {
10      host: process.env.DATABASE_HOST,
11      port: parseInt(process.env.DATABASE_PORT),
12      db: process.env.DATABASE_DB,
13      user: process.env.DATABASE_USER,
14      password: process.env.DATABASE_PASSWORD,
15    },
16    sentry: {
17      dsn: process.env.SENTRY_DSN,
18    },
19  });
20
```

Figura 166: Configuración de las variables de entorno.

El esquema nos servirá para validar la existencia y los tipos, inclusive si los valores son válidos o no, evitando que haya configuraciones que puedan poner en riesgo la aplicación o sistema.

nestjs-security-rest-api - env.schema.ts

```
import * as Joi from 'joi';

export const envSchema = Joi.object({
  NODE_ENV: Joi.string().valid('development', 'test', 'production').
required(),
  APP_NAME: Joi.string().required(),
  PORT: Joi.number().required(),
  UNIVERSITY_SERVICE_HOST: Joi.string().required(),
  UNIVERSITY_SERVICE_PORT: Joi.number().required(),
  DATABASE_HOST: Joi.string().required(),
  DATABASE_PORT: Joi.number().required(),
  DATABASE_DB: Joi.string().required(),
  DATABASE_USER: Joi.string().required(),
  DATABASE_PASSWORD: Joi.string().required(),
  SENTRY_DSN: Joi.string().required(),
});

```

Figura 167: Esquema de las variables de entorno.

Se ha configurado el protocolo de seguridad HTTPS que encripta la comunicación entre navegador y servidor, protegiendo la privacidad y la integridad de los datos.

Para ello se ha generado un certificado autofirmado para poder hacer pruebas:

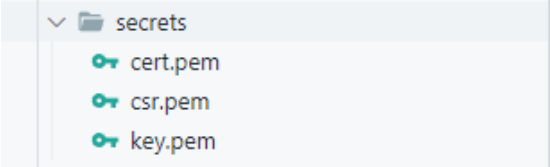


Figura 168: Certificados SSL autofirmados.

Cuando definamos la aplicación en el main del API Gateway deberemos pasar las opciones http indicando la ubicación de estos secretos:

nestjs-security-rest-api - main.ts

```
const app = await NestFactory.create(ApiGatewayModule, {
  bufferLogs: true,
  httpsOptions: {
    key: fs.readFileSync('./secrets/key.pem'),
    cert: fs.readFileSync('./secrets/cert.pem'),
  },
});
```

Figura 169: Configuración de los certificados SSL.

Para poder construir la URL del Swagger necesitaremos el esquema, host y puerto en el que arrancaremos, para el protocolo HTTPS el puerto asignado es el 443.

nestjs-security-rest-api - .env.development

```
API_GATEWAY_SCHEMA=https
API_GATEWAY_HOST=localhost
API_GATEWAY_PORT=443
```

Figura 170: Configuración URL segura.

A nivel de errores, para poder solucionar los problemas que podamos encontrarnos, debemos indicar la pila del error, dicha información nos la quedaremos para resolver la raíz del problema pero nos debemos asegurar de no exponerla.

nestjs-security-rest-api - microservice-exception.filter.ts

```
17  this.logger.error({
18    message: exception.message,
19    stack: exception.stack,
20    extra: {
21      statusCode: exception.error.statusCode,
22      timestamp,
23    },
24  });
```

Figura 171: Inclusión de la pila de stack para uso interno.

Debemos activar la protección CORS, para protegernos contra el acceso no controlado a recursos de distintos dominios.

nestjs-security-rest-api - main.ts

```
25  app.enableCors();
```

Figura 172: Activación de CORS en NestJS.

Tras observar las cabeceras inciales comprobamos que no estamos protegiendonos a ningún nivel, es más indicamos que tipo de motor utilizamos con x-powered-by: Express, lo cual es muy peligroso.

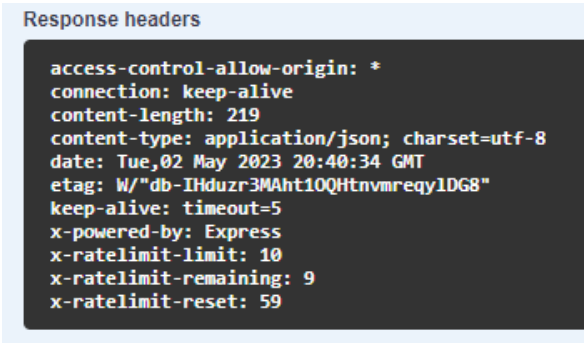


Figura 173: Cabeceras por defecto de Nestjs.

Para ello añadiremos helmet ya que nos ayudará a proteger nuestra API al establecer cabeceras HTTP de seguridad. Protege contra ataques como XSS (cross-site scripting), inyección de encabezado, sniffing de MIME, clicjacking, entre otros.

nestjs-security-rest-api - main.ts

```
23  app.use(helmet());
```

Figura 174: Aplicación de helmet.

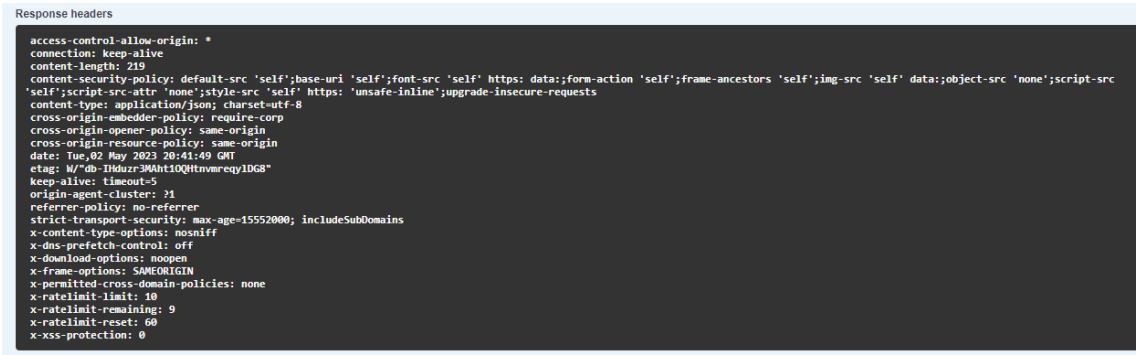


Figura 175: Nuevas cabeceras con protección de ataques XSS, etc.

A nivel de configuración también aplicaría la parte de puertos de Docker indicado anteriormente.
Debemos tener en cuenta que nuestras soluciones correrán en contenedores, dichos contenedores tienen sus imágenes Docker asociadas, las cuales pueden tener vulnerabilidades a nivel de librerías introducidas por la propia imagen o de nuestra solución.

Para validar las vulnerabilidades de las imágenes utilizaremos grype [29], el cual realiza un escaneado, cataloga los paquetes y nos da información clave para resolver las vulnerabilidades más críticas.

Instalamos grype y validamos las imágenes creadas:

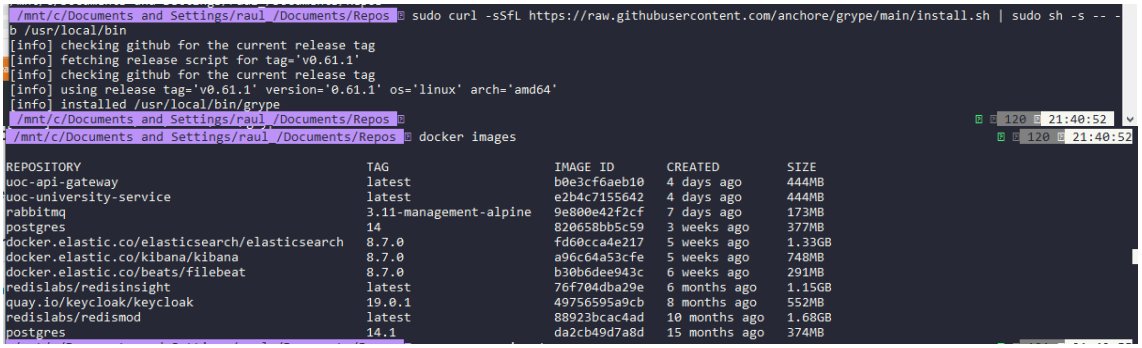


Figura 176: Instalación de grype.

Aplicamos la validación a la imagen generada de nuestro API Gateway creada, en este caso: ‘uoc-api-gateway’. Comprobamos que arrastramos muchas vulnerabilidades, en gran parte es debido a la versión que utilizamos de Node ya que la versión recomendada es la 18.16.0.

```
/mnt/c/Documents and Settings/raul_/Documents/Repos grype uoc-api-gateway                    121 21:42:55

 Vulnerability DB        [updated]
 Loaded image
 Parsed image
 Cataloged packages      [1097 packages]
 Scanning image...       [32 vulnerabilities]
   ├── 3 critical, 15 high, 14 medium, 0 low, 0 negligible
   └── 18 fixed

NAME                INSTALLED  FIXED-IN   TYPE    VULNERABILITY        SEVERITY
http-cache-semantics 4.1.0     4.1.1      npm     GHSA-rc47-6667-2j5j  High
libcrypto1.1        1.1.1n-r0             apk     CVE-2023-0466        Medium
libcrypto1.1        1.1.1n-r0  1.1.1q-r0  apk     CVE-2022-2097        Medium
libcrypto1.1        1.1.1n-r0  1.1.1t-r0  apk     CVE-2022-4304        Medium
libcrypto1.1        1.1.1n-r0  1.1.1t-r0  apk     CVE-2022-4450        High
libcrypto1.1        1.1.1n-r0  1.1.1t-r0  apk     CVE-2023-0215        High
libcrypto1.1        1.1.1n-r0  1.1.1t-r0  apk     CVE-2023-0286        High
libcrypto1.1        1.1.1n-r0  1.1.1t-r2  apk     CVE-2023-0464        High
libcrypto1.1        1.1.1n-r0  1.1.1t-r3  apk     CVE-2023-0465        Medium
libssl1.1           1.1.1n-r0             apk     CVE-2023-0466        Medium
libssl1.1           1.1.1n-r0  1.1.1q-r0  apk     CVE-2022-2097        Medium
libssl1.1           1.1.1n-r0  1.1.1t-r0  apk     CVE-2022-4304        Medium
libssl1.1           1.1.1n-r0  1.1.1t-r0  apk     CVE-2022-4450        High
libssl1.1           1.1.1n-r0  1.1.1t-r0  apk     CVE-2023-0215        High
libssl1.1           1.1.1n-r0  1.1.1t-r0  apk     CVE-2023-0286        High
libssl1.1           1.1.1n-r0  1.1.1t-r2  apk     CVE-2023-0464        High
libssl1.1           1.1.1n-r0  1.1.1t-r3  apk     CVE-2023-0465        Medium
node                16.15.0               binary  CVE-2022-32212       High
node                16.15.0               binary  CVE-2022-32213       Medium
node                16.15.0               binary  CVE-2022-32214       Medium
node                16.15.0               binary  CVE-2022-32215       Medium
node                16.15.0               binary  CVE-2022-32223       High
node                16.15.0               binary  CVE-2022-35255       Critical
node                16.15.0               binary  CVE-2022-35256       Medium
node                16.15.0               binary  CVE-2022-43548       High
node                16.15.0               binary  CVE-2023-23918       High
node                16.15.0               binary  CVE-2023-23919       High
node                16.15.0               binary  CVE-2023-23920       Medium
node                16.15.0               binary  CVE-2023-23936       Medium
npm                 8.5.5      8.11.0     npm     GHSA-hj9c-8jmm-8c52  High
zlib                1.2.12-r0  1.2.12-r2  apk     CVE-2022-37434       Critical
/mnt/c/Documents and Settings/raul_/Documents/Repos                                          122 21:43:27
```

Figura 177: Escaneado de vulnerabilidades de imagen uoc-api-gateway.

Dado a que necesitamos imágenes comprimidas siempre utilizaremos una versión alpine.

```
nestjs-security-rest-api - .env

2  NODE_ALPINE_VERSION=18.16.0-alpine
3
```

Figura 178: Versión imagen Docker de los contenedores.

Regeneraremos la imagen con la versión 18.16.0-alpine, comprobando que se ha generado una nueva imagen con el TAG latest y un tamaño algo superior al anterior.

```
/mnt/c/Documents and Settings/raul_/Documents/Repos docker images                            124 21:55:19
REPOSITORY                                      TAG                    IMAGE ID       CREATED          SIZE
uoc-api-gateway                                 latest                 8009580bd75d   39 seconds ago   507MB
<none>                                          <none>                 325328136f61   4 minutes ago    500MB
<none>                                          <none>                 b0e3cf6aeb10   4 days ago       444MB
uoc-university-service                          latest                 e2b4c7155642   4 days ago       444MB
rabbitmq                                        3.11-management-alpine 9e800e42f2cf   7 days ago       173MB
postgres                                        14                     820658bb5c59   3 weeks ago      377MB
docker.elastic.co/elasticsearch/elasticsearch   8.7.0                  fd60cca4e217   5 weeks ago      1.33GB
docker.elastic.co/kibana/kibana                 8.7.0                  a96c64a53cfe   5 weeks ago      748MB
docker.elastic.co/beats/filebeat                8.7.0                  b30b6dee943c   6 weeks ago      291MB
redislabs/redisinsight                          latest                 76f704dba29e   6 months ago     1.15GB
quay.io/keycloak/keycloak                       19.0.1                 49756595a9cb   8 months ago     552MB
redislabs/redismod                              latest                 88923bcac4ad   10 months ago    1.68GB
postgres                                        14.1                   da2cb49d7a8d   15 months ago    374MB
```

Figura 179: Listado de imágenes Docker.

Volveremos a pasar la validación de grype comprobando que únicamente aparecen dos vulnerabilidades de severidad media.

```
/mnt/c/Documents and Settings/raul_/Documents/Repos grype uoc-api-gateway                    125 21:59:06
 Vulnerability DB        [no update available]
 Loaded image
 Parsed image
 Cataloged packages      [1132 packages]
 Scanning image...       [2 vulnerabilities]
   ├── 0 critical, 0 high, 2 medium, 0 low, 0 negligible
   └── 2 fixed
NAME        INSTALLED  FIXED-IN  TYPE  VULNERABILITY  SEVERITY
libcrypto3  3.0.8-r3   3.0.8-r4  apk   CVE-2023-1255  Medium
libssl3     3.0.8-r3   3.0.8-r4  apk   CVE-2023-1255  Medium
/mnt/c/Documents and Settings/raul_/Documents/Repos                                          126 21:59:34
```

Figura 180: Re escaneado de vulnerabilidades de imagen uoc-api-gateway.

Para poder deshacernos de las vulnerabilidades en el Dockerfile añadiremos la instalación de dichas versiones.

nestjs-security-rest-api - Dockerfile.api-gateway

```
32  RUN apk add --no-cache tzdata libssl3=3.0.8-r4 libcrypto3=3.0.8-r4
33
```

Figura 181: Actualización de las versiones de las librerías con vulnerabilidades.

Volveremos a generar la imagen evitando la cache.

```
C:\Users\raul_\Documents\Repos\nestjs-security-rest-api\nestjs-security-rest-api-infra [main ≡ +0 ~1 -0 !]› docker-compose build uoc-api-gateway --no-cache
[+] Building 114.7s (22/23)
 => [internal] load build definition from Dockerfile.api-gateway                                                                                0.0s
 => => transferring dockerfile: 1.18kB                                                                                                          0.0s
 => [internal] load .dockerignore                                                                                                               0.0s
 => => transferring context: 2B                                                                                                                 0.0s
 => [internal] load metadata for docker.io/library/node:18.16.0-alpine                                                                        114.6s
 => [auth] library/node:pull token for registry-1.docker.io                                                                                     0.0s
 => [internal] load build context                                                                                                               8.8s
 => => transferring context: 4.48MB                                                                                                             8.7s
 => [builder 1/9] FROM docker.io/library/node:18.16.0-alpine@sha256:1ccc70acda680aa4ba47f53e7c40b2d4d6892de74817128e0662d32647dd7f4d             0.0s
 => CACHED [production 2/9] WORKDIR /home/node/api-gateway                                                                                      0.0s
 => CACHED [builder 2/9] WORKDIR /usr/src/api-gateway                                                                                           0.0s
 => [production 3/9] RUN chown node:node /home/node/api-gateway                                                                                 0.5s
 => [production 4/9] RUN mkdir /home/node/api-gateway/logs &&     chown node:node /home/node/api-gateway/logs &&     chmod 775 /home/node/api-gateway/logs  0.7s
 => [production 5/9] RUN apk add --no-cache tzdata libssl3=3.0.8-r4 libcrypto3=3.0.8-r4                                                         2.3s
 => [builder 3/9] COPY package*.json ./                                                                                                         0.2s
 => [builder 4/9] COPY secrets ./                                                                                                               0.1s
 => [builder 5/9] RUN npm install -g @nestjs/cli                                                                                               32.0s
 => [builder 6/9] RUN npm install                                                                                                              34.1s
 => [builder 7/9] COPY . ./                                                                                                                     7.6s
 => [builder 8/9] RUN npm run build shared                                                                                                      6.5s
 => [builder 9/9] RUN npm run build api-gateway                                                                                                 7.1s
 => [production 6/9] COPY --from=builder --chown=node:node /usr/src/api-gateway/package*.json /home/node/api-gateway/                           0.6s
 => [production 7/9] COPY --from=builder --chown=node:node /usr/src/api-gateway/node_modules /home/node/api-gateway/node_modules                6.2s
 => [production 8/9] COPY --from=builder --chown=node:node /usr/src/api-gateway/dist /home/node/api-gateway/dist                                0.1s
 => [production 9/9] COPY --from=builder --chown=node:node /usr/src/api-gateway/secrets /home/node/api-gateway/secrets                          0.1s
 => exporting to image                                                                                                                          5.8s
 => => exporting layers                                                                                                                         5.8s
 => => writing image sha256:947a4ebf76475ad6ab4c48fe80701032a1124cff8c9fd5c7df1d78097c3c1fb7                                                    0.0s
 => => naming to docker.io/library/uoc-api-gateway                                                                                              0.0s
C:\Users\raul_\Documents\Repos\nestjs-security-rest-api\nestjs-security-rest-api-infra [main ≡ +0 ~2 -0 !]›
```

Figura 182: Regeneración de las imágenes a través de Docker-compose.

Finalmente tendremos una imagen libre de vulnerabilidades.

```
/mnt/c/Documents and Settings/raul /Documents/Repos docker images
REPOSITORY                                      TAG                 IMAGE ID       CREATED          SIZE
uoc-api-gateway                                 latest              947a4ebf7647   52 seconds ago   512MB
<none>                                          <none>              8009580bd75d   14 minutes ago   507MB
<none>                                          <none>              325328136f61   17 minutes ago   500MB
<none>                                          <none>              b0e3cf6aeb10   4 days ago       444MB
/mnt/c/Documents and Settings/raul /Documents/Repos grype uoc-api-gateway                                  127 22:12:36
 Vulnerability DB        [no update available]
 Loaded image
 Parsed image
 Cataloged packages      [1132 packages]
 Scanning image...       [0 vulnerabilities]
   ├── 0 critical, 0 high, 0 medium, 0 low, 0 negligible
   └── 0 fixed
No vulnerabilities found
/mnt/c/Documents and Settings/raul /Documents/Repos                                                        128 22:13:07
```

Figura 183: Validación nuevas imágenes y escaneado de imagen con grype.

Deberemos realizar las mismas acciones en el microservicio de universidad.

```
postgres                                14.1                dd2cb49d7a8d   15 months ago    374MB
/mnt/c/Windows/System32 grype uoc-university-service                                       129 23:21:56
 Vulnerability DB        [no update available]
 Loaded image
 Parsed image
 Cataloged packages      [1132 packages]
 Scanning image...       [0 vulnerabilities]
   ├── 0 critical, 0 high, 0 medium, 0 low, 0 negligible
   └── 0 fixed
o vulnerabilities found
```

Figura 184: Validación imagen microservicio con grype.

Finalmente para inventariar las versiones, lo adecuado sería añadirlas a un repositorio de imágenes. Gracias a DockerHub podremos generar los tag adecuados y subirlas a la nube.

```
→ C:\Users\raul_\Documents\Repos\nestjs-security-rest-api\nestjs-security-rest-api-infra [main ≡]› docker tag 947a4eb raulruizbarea/uoc-api-gateway:latest
→ C:\Users\raul_\Documents\Repos\nestjs-security-rest-api\nestjs-security-rest-api-infra [main ≡]› docker images
REPOSITORY                                    TAG                  IMAGE ID       CREATED             SIZE
uoc-university-service                        latest               d6d90b8c264a   19 minutes ago      512MB
raulruizbarea/uoc-university-service          latest               d6d90b8c264a   19 minutes ago      512MB
uoc-api-gateway                               latest               947a4ebf7647   About an hour ago   512MB
raulruizbarea/uoc-api-gateway                 latest               947a4ebf7647   About an hour ago   512MB
<none>                                        <none>               b0e3cf6aeb10   4 days ago          444MB
<none>                                        <none>               e2b4c7155642   4 days ago          444MB
rabbitmq                                      3.11-management-alpine 9e800e42f2cf   7 days ago          173MB
postgres                                      14                   820658bb5c59   3 weeks ago         377MB
docker.elastic.co/elasticsearch/elasticsearch 8.7.0                fd60cca4e217   5 weeks ago         1.33GB
docker.elastic.co/kibana/kibana               8.7.0                a96c64a53cfe   5 weeks ago         748MB
docker.elastic.co/beats/filebeat              8.7.0                b30b6dee943c   6 weeks ago         291MB
redislabs/redisinsight                        latest               76f704dba29e   6 months ago        1.15GB
quay.io/keycloak/keycloak                     19.0.1               49756595a9cb   8 months ago        552MB
redislabs/redismod                            latest               88923bcac4ad   10 months ago       1.68GB
postgres                                      14.1                 da2cb49d7a8d   15 months ago       374MB
→ C:\Users\raul_\Documents\Repos\nestjs-security-rest-api\nestjs-security-rest-api-infra [main ≡]› docker push raulruizbarea/uoc-api-gateway
Using default tag: latest
The push refers to repository [docker.io/raulruizbarea/uoc-api-gateway]
bb178bc275f1: Pushed
36bc88347a13: Pushed
74a08b658805: Pushed
b8feaa354732: Pushed
a01cbbaa25c1: Pushed
f08c0f67d677: Pushed
9870b7b96fac: Pushed
85ee2bc7e260: Pushed
885a5d40fc11: Mounted from raulruizbarea/uoc-university-service
1b6c3782871e: Mounted from raulruizbarea/uoc-university-service
b0e46d71a47b: Mounted from raulruizbarea/uoc-university-service
f1417ff83b31: Mounted from raulruizbarea/uoc-university-service
latest: digest: sha256:82fdab9c030540c9fa8c45a206a90e952ddc926c0fc28027eec8ed5be41324ad size: 2829
→ C:\Users\raul_\Documents\Repos\nestjs-security-rest-api\nestjs-security-rest-api-infra [main ≡]›
```

Figura 185: Generación del tag y publicación a DockerHub.

Comprobación de los repositorios en DockerHub, validando que se han subido las imágenes correctamente.

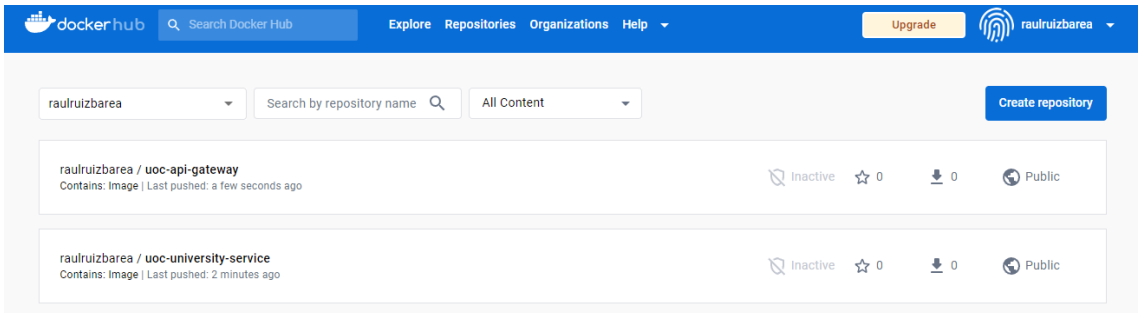


Figura 186: Validación repositorios de DockerHub.

API8:2019 Injection

Deberemos proteger los distintos endpoints validando los datos de entrada introducidos por el cliente, sanitizando código HTML gracias al decorador @Escape() y al validation pipe lo aplicaremos en los campos necesarios.

nestjs-security-rest-api - create-subject.dto.ts

```
16  @Escape()
```

Figura 187: Sanitización de propiedades con librería class-sanitizer.

También podemos ejecutar las validaciones cuando no esperamos un DTO, en este caso cuando actualizamos una asignatura, indicamos el código de la asignatura a actualizar y un objeto que contenga los nuevos valores, aplicando manualmente para el código de asignatura y automáticamente a través de los decoradores para el objeto de actualización de asignatura.

nestjs-security-rest-api - subjects.controller.ts

```
124  code = Sanitizer.escape(code).trim();
125  sanitize(updateSubjectDto);
```

Figura 188: Aplicación de la sanitización de DTOs y propiedades de manera manual y automática.

Otra de las grandes vulnerabilidades es la inyección de SQL, la cual podría obtener información de nuestra base de datos (otros registros, columnas, otras tablas, borrado, etc.). Para afrontar estas debilidades, estamos utilizando TypeORM que ya de entrada nos aporta protección contra la inyección de SQL, una buena práctica es utilizar parámetros y nunca construir en modo de cadena texto las consultas.

nestjs-security-rest-api - subject-typeorm.repository.ts

```
85  const updateResult: UpdateResult = await this.subjectRepository
86    .createQueryBuilder()
87    .update(SubjectDao)
88    .set(subject)
89    .where('code = :code', { code })
90    .execute()
91    .catch((exception) => {
92      throw new RpcException({
93        message: APP_EXCEPTION.BAD_REQUEST,
94        statusCode: HttpStatusCode.BadRequest,
95      });
96    });
```

Figura 189: Aplicación de parámetros en los query builders de TypeOrm.

Es una buena práctica definir los tamaños de las distintas propiedades, para ello se define a nivel de configuración por entidades el tamaño de cada columna.

nestjs-security-rest-api - settings.ts

```
1  export const SubjectSettings = {
2    CODE_LENGTH: 6,
3    ACADEMICALYEAR_LENGTH: 5,
4    LANGUAGES_LENGTH: 3,
5    NAME_LENGTH: 60,
6    DESCRIPTION_LENGTH: 900,
7  };
8
```

Figura 190: Longitud de las columnas de base de datos.

A nivel de campo aplicaremos los decoradores adecuados de ‘class-validator’ para validar el tamaño máximo, mínimo o exacto.

nestjs-security-rest-api - create-subject.dto.ts

```
50  @MaxLength(SubjectSettings.NAME_LENGTH)
```

Figura 191: Aplicación de tamaños máximo a nivel de columnas.

API9:2019 Improper Assets Management

Debemos realizar un buen inventariado del API para poder realizar las pruebas convenientes a todos los niveles, controlar que tenemos y poder establecer un proceso de despliegue para nuevas versiones y/o dar soporte a versiones anteriores.

Para ello activaremos el versionado del API empezando en la versión 1, por lo tanto, nuestras URLs se generarán con /api/v1:

nestjs-security-rest-api - main.ts

```
37  //* API:  /api/v1
38  app.setGlobalPrefix('api');
39  app.enableVersioning({
40    type: VersioningType.URI,
41    //*  if you want to have a specific version set as the default version for every controller/route
42    defaultVersion: '1',
43  });
```

Figura 192: Activación del versionado del API.

Por otro lado NestJS nos ofrece la posibilidad de generar la documentación del API indicando todas las opciones necesarias, y en este momento, aprovecharemos para generar el archivo open-api.yaml que tendrá la definición del API la cual podremos importar en otras herramientas.

nestjs-security-rest-api - main.ts

```
55  const options = new DocumentBuilder()
56    .setTitle('NestJS Security REST API')
57    .setDescription('Open University of Catalonia')
58    .setVersion('1.0')
59    .addBearerAuth(
60      {
61        type: 'http',
62        scheme: 'Bearer',
63        bearerFormat: 'Bearer',
64        name: 'JWT',
65        description: 'Enter JWT token',
66        in: 'header',
67      },
68      'access-token',
69    )
70    .addServer(`${schema}://${host}:${port}`)
71    .build();
72
73  const document = SwaggerModule.createDocument(app, options);
74  fs.writeFileSync('./open-api.yaml', yaml.stringify(document, {}));
75
76  SwaggerModule.setup('api', app, document);
```

Figura 193: Generación de la documentación del API a fichero YAML.

open-api.yaml

```
1   openapi: 3.0.0
2   paths:
3     /api/v1/health:
4       get:
5         operationId: HealthController_check
6         summary: Get the health of the system
7         description: Get the health of the system
8         parameters: []
9         responses:
10          "200":
11            description: The Health Check is successful
12            content:
13              application/json:
14                schema:
15                  type: object
16                  properties:
17                    status:
18                      type: string
19                      example: ok
20                  info:
```

Figura 194: Formato YAML de la documentación del API.

Al importar el open-api.yaml en Postman, nos generará toda la jerarquía con los distintos endpoints preparados con la información necesaria.

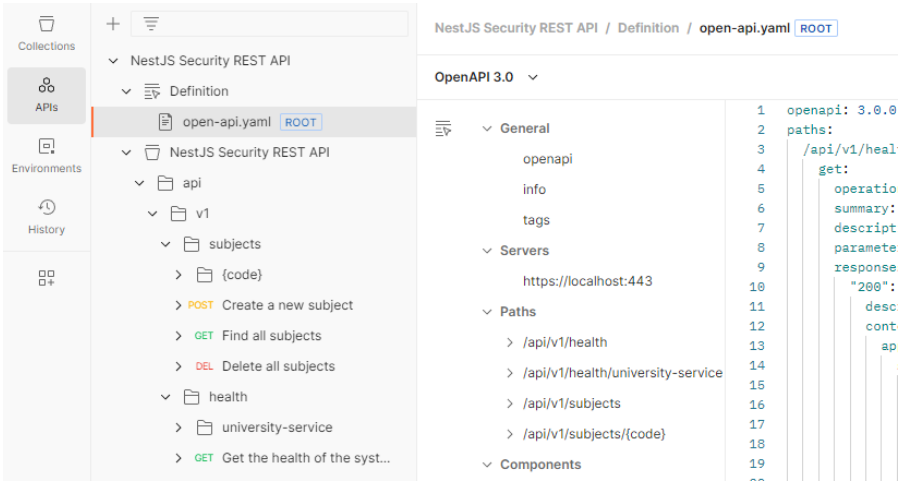


Figura 195: Importación del open-api.yaml en Postman.

Debemos mantener la documentación de la arquitectura actualizada y transparente para la compañía, indicar como se comunican y la sensibilidad de los datos.

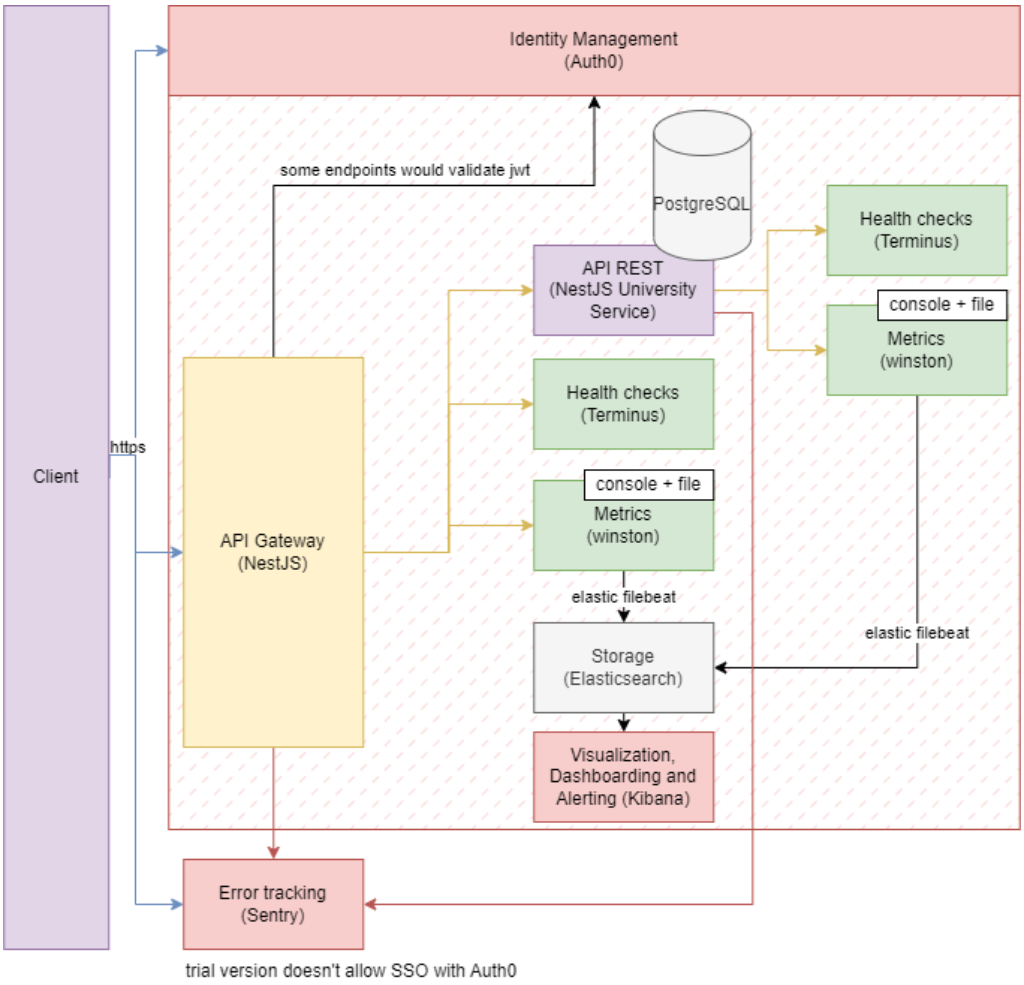


Figura 196: Arquitectura propuesta de la solución.

Junto a la arquitectura debemos documentar la fotografía final de los entornos, que vamos a exponer, también deberíamos incluir versionado de imágenes.

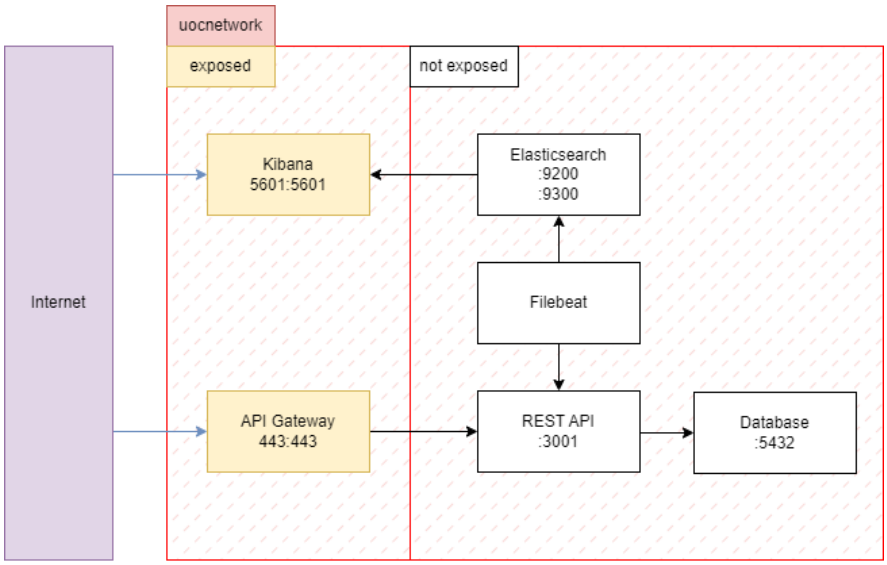


Figura 197: Arquitectura de contenedores Docker.

API10:2019 Insufficient Logging & Monitoring

Para poder afrontar cualquier brecha de seguridad, debemos tener un sistema de logs que nos permitan saber a que nivel está afectando, en nuestro caso Winston ofrece una variedad de niveles según el contexto del log, siendo el nivel error el más grave y silly el menos grave.

```
const levels = {
  error: 0,
  warn: 1,
  info: 2,
  http: 3,
  verbose: 4,
  debug: 5,
  silly: 6
};
```

Figura 198: Niveles de logs con Winston.

Vamos a loggear cualquier petición http a nivel de fichero, y cualquier alerta o error por consola y fichero, todos los errores que puedan surgir a nivel de validación se capturarán a través de los logs de http.

Para tener éxito ya que estamos utilizando el stack de Elasticsearch y Kibana, utilizaremos el formato de Elasticsearch para Winston.

nestjs-security-rest-api - university.module.ts

```
22  const ecsFormat = require('@elastic/ecs-winston-format');
23
```

Figura 199: Formato de elasticsearch para Winston.

Para poder diferenciar el servicio que lanza cada log nos ofrecen campos a nivel de metadatos para capturar dicha información:

nestjs-security-rest-api - university.module.ts

```
41  defaultMeta: { service: { name: configService.get('name') } },
```

Figura 200: Inclusión de metadatos en los logs.

Un ejemplo de error loggeado a nivel de fichero con formato ecs, podemos ver como nos indica el servicio: university-service, y en el mensaje tenemos el stack adecuado con información extra, indicando que no pueden haber nulos en el nombre exacto de la columna y en el nombre exacto de la tabla.

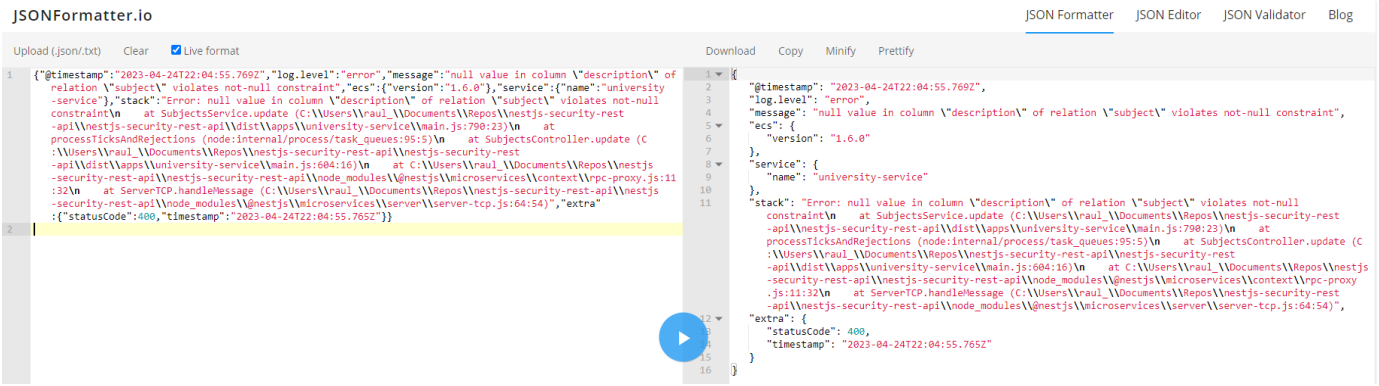


Figura 201: Formateo de los logs para interpretación.

A nivel de seguridad no queremos dar dicha información y generaremos una respuesta que podamos catalogar, gracias al contrato que establezamos daremos pistas sobre a que nivel está surgiendo el error.

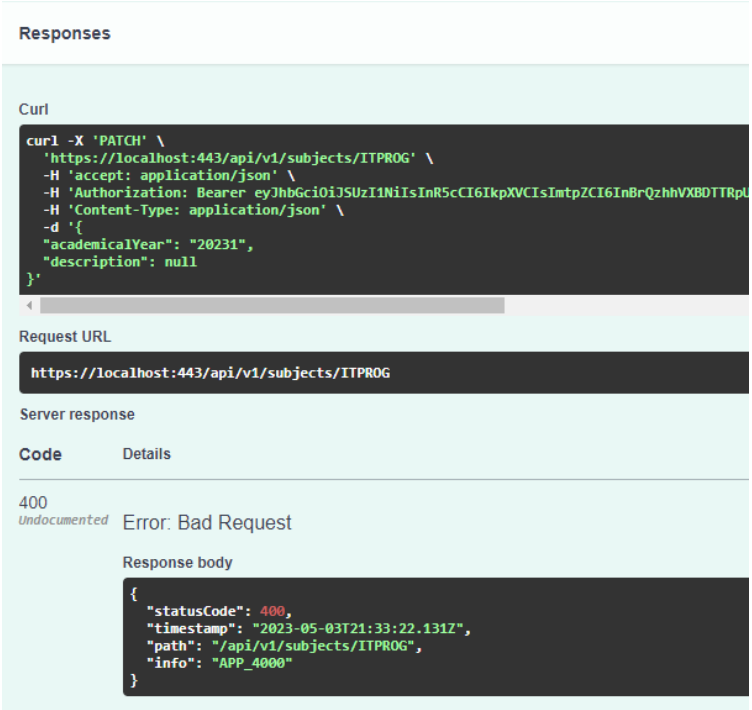


Figura 202: Respuesta personalizada con error a nivel de base de datos.

También al necesitar estar informados de errores no esperados enviaremos dichos errores a Sentry donde podremos hacer un seguimiento, establecer alertas, fechas de resolución, etc.

A nivel de monitoreo en vivo de logs usaremos Kibana, donde podremos generarnos alertas y avisos, extraer reportes para poder identificar cuellos de botella, periodos en los que haya más tráfico inclusive eventos que puedan estar generando retrasos en respuestas.

A nivel de analisis dispondremos del dashboard con el cual podemos gestionar los filtros indicando los campos que nos interesan, incluso jugar con los diferentes gráficos que generamos. Al realizar este tipo de acciones se actualizarán los objetos con dichos filtros permitiendo compartir la información.

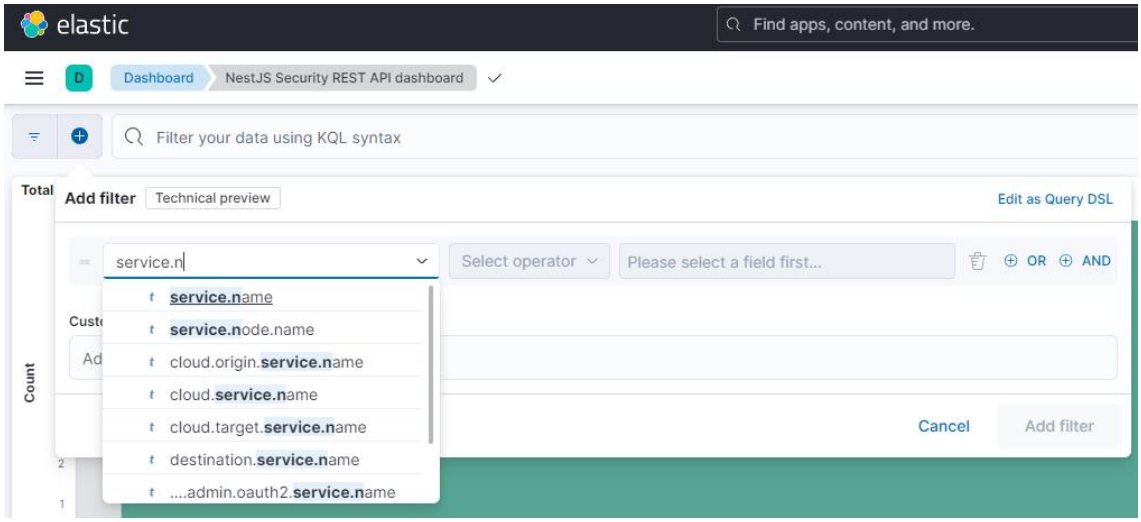


Figura 203: Filtro de datos en Dashboard de Kibana.

A nivel de observabilidad debido a la configuración básica que hemos realizado únicamente disponemos de logs y streaming de logs, pudiendo ver los logs en tiempo real o filtrar por intervalos de fechas.

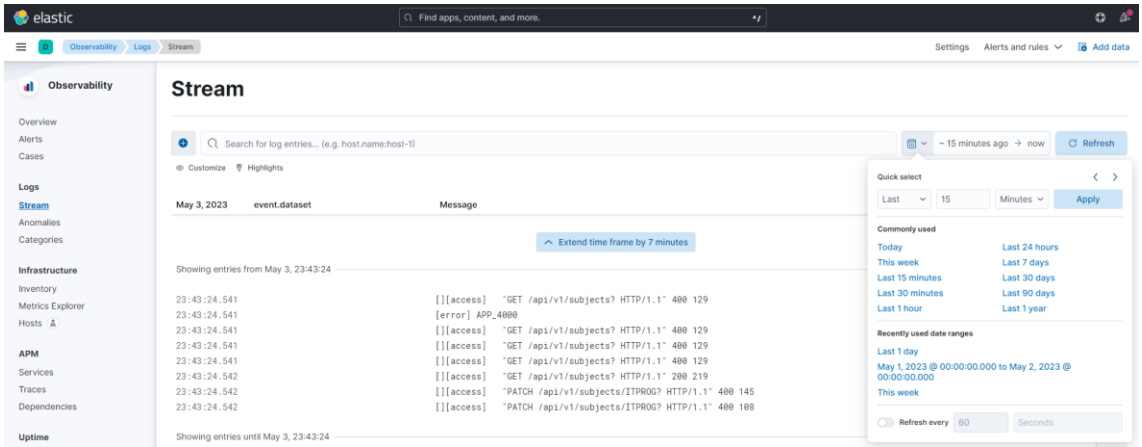


Figura 204: Filtro de datos del streaming de datos de Kibana.

5. Conclusiones

Durante la ejecución de esta Tesis de Máster, se ha conducido una exploración exhaustiva de las API REST y API Gateway, adquiriendo conocimientos relevantes y nuevas prácticas en materia de seguridad.

Se ha profundizado en varios patrones de arquitectura, junto con una detallada exploración de las vulnerabilidades más comunes según la propuesta de OWASP.

Una de las conclusiones significativas es que, en muchos proyectos, no se invierte tiempo suficiente en establecer medidas de seguridad adecuadas, lo cual puede ser resultado de un conocimiento insuficiente en este ámbito o falta de tiempo.

La privacidad y seguridad deben ser elementos primordiales al diseñar cualquier sistema, asegurando el almacenamiento seguro de la información y la integridad de los datos.

En cuanto a la implementación, con una base de conocimiento en Angular, la introducción a NestJS resultó intuitiva, proporcionando la misma estructura modular.

Para el despliegue, se utilizó Docker, permitiendo analizar las vulnerabilidades del entorno y asegurando un despliegue eficaz. Los resultados cumplen con las expectativas: se ha desplegado una solución segura en un entorno de contenedores, y se ha verificado la seguridad de la API REST a través de un API Gateway. Asimismo, la realización de entrevistas en la compañía actual ha validado los conocimientos adquiridos a lo largo de este proceso, lo cual será beneficioso para el avance de mi carrera profesional.

Se han logrado los objetivos principales de profundizar en el concepto de las API REST y desarrollar una API REST y un API Gateway con NestJS.

En cuanto a los objetivos parciales, se han identificado y mitigado los riesgos más comunes de OWASP en las API REST, aplicando diversas medidas de seguridad para proteger tanto a la API REST como al API Gateway. Aunque no se ha logrado la escalabilidad debido a la no inclusión de Kubernetes, se ha desplegado la solución en un entorno de contenedores garantizando límites de recursos.

La gestión de excepciones y errores representó un desafío considerable debido a la inicial falta de conocimiento. La unificación de los mensajes de respuesta en un ambiente de microservicios también fue compleja.

La planificación se llevó a cabo de manera exitosa, requiriéndose días adicionales para la investigación por la inclusión de la arquitectura propuesta.

El proceso de investigación fue continuo durante todo el proyecto, adaptándose la arquitectura a medida que se iba avanzando.

Para trabajos futuros, sería interesante profundizar en los conceptos de Kubernetes relativos a seguridad, control de entrada y balanceo de carga. Además, se debería continuar manteniendo la documentación actualizada y establecer un pipeline de despliegue para pruebas automáticas y aplicación de linting con ESLint o gates de calidad con SonarQube.

Explorar más patrones de diseño, como el circuit breaker o el patrón de CQRS, también sería de gran valor para evitar fallos en cascada en sistemas distribuidos y mejorar el rendimiento y la flexibilidad.

6. Glosario

API: Application Programming Interface, conjunto de reglas y protocolos que nos indican como las aplicaciones de software interactúan entre sí.
CORS: Cross-Origin Resource Sharing, función de seguridad que permite acceder a recursos de diferentes dominios.
CRUD: Create, Read, Update y Delete, cuatro funciones básicas de almacenamiento en base de datos.
DAO: Data Access Object, patrón de diseño que separa la lógica de base de datos.
DTO: Data Transfer Object, estructura de datos para transferir información entre diferentes componentes.
GW: Gateway, sistema que conecta dos o más redes y gestiona el tráfico entre ellas.
HTTP: Hypertext Transfer Protocol, protocol para transferencia de datos a través de la web.
IDE: Integrated Development Environment, aplicación de software que nos proporciona un entorno de desarrollo.
JWT: JSON Web Token, forma compacta y segura de transmitir información entre diferentes partes en formato JSON.
ORM: Object-Relational Mapping, técnica de programación para convertir datos entre sistemas de tipos incompatibles.
OWASP: Open Web Application Security Project, comunidad dedicada a mejorar la seguridad del software.
SOAP: Simple Object Access Protocol, protocolo de mensajería a través de la web.
SQL: Structured Query Language, lenguaje de programación para administrar bases de datos relacionales.
SSL: Secure Sockets Layer, protocolo de seguridad que establece encriptados entre navegador y servidor.
TCP: Transmission Control Protocol, protocolo estándar para transmisión de datos por internet.
TLS: Transport Layer Security, protocolo de seguridad que garantiza la comunicación segura entre dos dispositivos.
URL: Uniform Resource Locator, dirección web de un recurso de internet.
UUID: Universally Unique Identifier, identificador de información en formato de número de 128 bits.
YAML: Yet Another Markup Language, formato de serialización de datos legible comúnmente para configuraciones.

7. Bibliografía

[1] Gough, J., Bryant, D., & Auburn, M. (2022). Mastering API architecture: Defining, connecting, and securing distributed systems and microservices. O'Reilly Media.
[2] OWASP API Security Project. (s/f). Owasp.org. Recuperado el 29 de abril de 2023, de https://owasp.org/www-project-api-security/
[3] ¿Qué es una API y cómo funciona? (s/f). Redhat.com. Recuperado el 29 de abril de 2023, de https://www.redhat.com/es/topics/api/what-are-application-programming-interfaces
[4] What is OpenAPI? (2023, marzo 31). OpenAPI Initiative. https://www.openapis.org/what-is-openapi
[5] Clean code. (S/f-c). Udemy.com. Recuperado el 29 de abril de 2023, de https://www.udemy.com/course/writing-clean-code/
[6] Documentation. (s/f). Documentation | NestJS - A Progressive Node.Js Framework. Recuperado el 29 de abril de 2023, de https://docs.nestjs.com/
[7] NestJS The complete developers guide. (S/f). Udemy.com. Recuperado el 29 de abril de 2023, de https://www.udemy.com/course/nestjs-the-complete-developers-guide/
[8] Microservices with node js and react. (S/f-b). Udemy.com. Recuperado el 29 de abril de 2023, de https://www.udemy.com/course/microservices-with-node-js-and-react/
[9] SAS help center. (s/f). Sas.com. Naming Conventions for PostgreSQL. Recuperado el 29 de abril de 2023, de https://documentation.sas.com/doc/en/pgmsascdc/9.4_3.5/acreldb/p1iw263fz6wvnbn1d6nyw71a9sf2.htm
[10] Arias, D. (2019, octubre 29). Full-stack TypeScript apps: Developing a secure API with NestJS. Auth0 - Blog. https://auth0.com/blog/developing-a-secure-api-with-nestjs-adding-authorization/
[11] Wanago, M. (2021, octubre 11). API with NestJS #51. Health checks with Terminus and Datadog. Marcin Wanago Blog - JavaScript, Both Frontend and Backend. https://wanago.io/2021/10/11/api-nestjs-health-checks-terminus-datadog/
[12] Freeman, E., & Robson, E. (2021). Head first design patterns: Building extensible and maintainable object-oriented software (2a ed.). O'Reilly Media.
[13] Thunder Client - Rest API Client Extension for VS code. (s/f). Thunderclient.com. Recuperado el 29 de abril de 2023, de https://www.thunderclient.com/
[14] Mao, H. (s/f). Vscode-restclient: REST Client Extension for Visual Studio Code.
[15] Khoury, J. (2023, enero 11). State of APIs: growth and more growth on tap for 2023. Rapid Blog; RapidAPI. https://rapidapi.com/blog/state-of-apis-growth-and-more-growth-on-tap-for-2023/
[16] Guidelines.md at vNext · microsoft/api-guidelines. (s/f).
[17] Swagger editor. (s/f). Swagger.Io. Recuperado el 29 de abril de 2023, de https://editor.swagger.io/?_ga=2.201457557.567349853.1680452842-207215649.1677922373&_gac=1.83900523.1677922373.Cj0KCQiA9YugBhCZARIsAACXxeLl_yiCf-OOW8PI-3y32akZH0Q0047eNH5qWRdLJV6FiatkEsR9ZSwaAg79EALw_wcB

[18] Postman. (s/f). Postman.com. Recuperado el 29 de abril de 2023, de https://www.postman.com/
[19] Apache JMeterTM. (s/f). Apache.org. Recuperado el 29 de abril de 2023, de https://jmeter.apache.org/
[20] Application performance monitoring & error tracking software. (s/f). Sentry. Recuperado el 29 de abril de 2023, de https://sentry.io/welcome/
[21] Prometheus. (s/f). Prometheus - Monitoring system & time series database. Prometheus.Io. Recuperado el 29 de abril de 2023, de https://prometheus.io/
[22] Grafana: The open observability platform. (s/f). Grafana Labs. Recuperado el 29 de abril de 2023, de https://grafana.com/
[23] Blokdyk, G. (2018). Elasticsearch: Your Complete Guide. Createspace Independent Publishing Platform.
[24] Kibana. (s/f). Elastic. Recuperado el 29 de abril de 2023, de https://www.elastic.co/kibana/
[25] OpenAPM Landscape. (s/f). OpenAPM.Io. Recuperado el 29 de abril de 2023, de https://openapm.io/landscape
[26] How to create pagination in Nest.js with TypeORM + Swagger | Adrian Pietrzak: Software Engineering Blog. (2022, 1 febrero). How to create pagination in Nest.js with TypeORM + Swagger | Adrian Pietrzak: Software Engineering Blog. https://pietrzakadrian.com/blog/how-to-create-pagination-in-nestjs-with-typeorm-swagger
[27] “State of API 2020 Report | SmartBear.” Smartbear.com, https://smartbear.com/resources/ebooks/the-state-of-api-2020-report/ Accessed 9 May 2023.
[28] “Understanding the API Gateway: When You Need It and How to Implement It.” AltexSoft, https://www.altexsoft.com/blog/api-gateway/
[29] “Anchore/Grype.” GitHub, 9 May 2023, https://github.com/anchore/grype/ Accessed 9 May 2023.