UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN DESARROLLO ÁGIL DE SOFTWARE PARA LA WEB

Trabajo Fin de Máster

SEGURIDAD EN SPRING BOOT CON OAUTH

Javier González de Lope

2021

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN

DESARROLLO ÁGIL DE SOFTWARE PARA LA WEB

Trabajo Fin de Máster

"SEGURIDAD EN SPRING BOOT CON OAUTH"

Autor: **Javier González de Lope**

Director: **Salvador Otón Tortosa**

Tribunal:

Presidente: .....

Vocal 1º: ..... Vocal 2º: .....

Calificación: .....

Fecha: ..... de ..... de .....

Quiero dedicar este trabajo

A mis padres, por ayudarme a crecer cómo persona, cuidarme y darme siempre el cariño que he necesitado.

A mi hermana Cris, por creer siempre en mi y demostrarme que el esfuerzo tiene su recompensa.

A Sergio, porque no todo en la vida es estudio y trabajo y tus entrenos me han ayudado a desconectar como pocas cosas durante estos dos años.

A Phoebis, mi fiel compañero de batallas, por haber sido mi mayor punto de apoyo cada día en el trabajo y en cada clase de este curso.

Y, por último, a Desi, mi compañera de vida, por empujarme cada día a ser la mejor versión de mí mismo, aguantarme en los momentos de agobio y ayudarme a construir un futuro a su lado.

ÍNDICE RESUMIDO

ÍNDICE DETALLADO

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

Hace unos años, la mayor parte de las aplicaciones y sistemas software que se podían encontrar en el mercado contaban con una arquitectura monolítica. Estas arquitecturas podían presentar un mayor o menor grado de modularidad interna y calidad en su construcción, pero el modelo de construcción utilizado para su desarrollo distaba mucho de la tendencia actual.

Vivimos en una sociedad que ha hecho de las tecnologías como los ordenadores, smartphones, wearables, dispositivos de IoT, etc. algo cotidiano. Productos que en su día podían parecer artículos de lujo, ahora son una parte indispensable de nuestras vidas. La edad o las diferencias culturales no parecen seguir siendo un obstáculo.

Esto se debe, en gran parte, a cambios en el modelo de desarrollo de los productos y las bases que han sentado gigantes del mundo actual como Amazon, Netflix, AirBnb, Spotify y otras compañías que, con un enfoque de diseño centrado en el usuario en lugar de en el producto, y una adopción de las metodologías ágiles en contraposición a las metodologías tradicionales, han conseguido encontrar su hueco en el mercado y consolidar su posición.

La competencia está a la orden del día y cada vez es mayor, por lo que la flexibilidad y adaptación al cambio son atributos que toman una nueva dimensión mucho más relevante de la que tenían en el pasado con los cambios lentos de las grandes empresas que tenían poder para ello. De esta forma, tanto el usuario como el cliente se han acostumbrado a vivir en un mundo de novedades que hay que satisfacer, y en el que la tecnología ha adquirido el rol del actor protagonista.

Este mercado tan cambiante junto a la irrupción de un modelo de startups tecnológicas ha favorecido que los nuevos productos de software que surgen cada día sigan una arquitectura de microservicios al mismo tiempo que muchas empresas se ven obligadas a refactorizar sus sistemas monolíticos para adoptar este tipo de arquitecturas.

La arquitectura de microservicios ha ganado popularidad hace relativamente poco tiempo y se puede considerar que está en su fase inicial, ya que todavía no existe un consenso sobre la definición exacta de lo que pueden ser los microservicios.

M. Fowler y J. Lewis proporcionan una base de partida al definir las principales características de los microservicios [3]. S. Newman [4] se basa en el artículo de M. Fowler y presenta recetas y mejores prácticas en relación con algunos aspectos de dicha arquitectura. En su trabajo, L. Krause [5] discute los patrones y aplicaciones de los microservicios.

También se han publicado varios artículos que describen los detalles del diseño y la implementación de sistemas que utilizan la arquitectura de microservicios. Así, M. Rahman y J. Gao en [6] describen una aplicación del desarrollo dirigido por el comportamiento (BDD) a la arquitectura de microservicios con el fin de disminuir la carga de mantenimiento de los desarrolladores y fomentar el uso de pruebas de aceptación.

En una aplicación que utiliza microservicios, como en cualquier sistema distribuido, la seguridad se convierte en una preocupación importante. En este sentido, los microservicios sufren las mismas vulnerabilidades de seguridad que la SOA. Como los microservicios utilizan el mecanismo REST y XML con JSON como principales formatos de intercambio de datos, se debe prestar especial atención a la seguridad de los datos que se transfieren. Los microservicios promueven la reutilización de los servicios, por lo que es natural suponer que algunos sistemas

incluirán servicios de terceros. Por lo tanto, un reto adicional es proporcionar mecanismos de autenticación y autorización con servicios de terceros, ámbito en el cual cobran protagonismo protocolos como OAuth2 y OpenID Connect.

1.1. Justificación y motivación del proyecto

La creciente complejidad del software moderno exige nuevos enfoques de diseño arquitectónico y modelado de sistemas. Los sistemas complejos también muestran un alto nivel de concurrencia, es decir, múltiples hilos de ejecución entrelazados que a menudo se ejecutan en hardware diferente, que deben estar sincronizados y coordinados y que comparten información a menudo a través de diferentes paradigmas de comunicación.

Por lo tanto, la mejora de la calidad del software y el despliegue de servicios fiables es la consecuencia de un uso preciso de estilos arquitectónicos óptimos basados en servicios y de técnicas de ingeniería de software bien establecidas para la obtención de requisitos, el diseño, las pruebas y la verificación.

Con el fin de abordar estas cuestiones desde el punto de vista arquitectónico, la arquitectura de microservicios ha aparecido últimamente como un nuevo paradigma para la programación de aplicaciones mediante la composición de pequeños servicios, cada uno de los cuales ejecuta sus propios procesos y se comunica a través de mecanismos ligeros.

Los microservicios son ahora una nueva tendencia en la arquitectura de software, que hace hincapié en el diseño y desarrollo de software altamente mantenible y escalable. Gestionan la creciente complejidad descomponiendo funcionalmente grandes sistemas en un conjunto de servicios independientes y hacen hincapié en el acoplamiento flexible y la alta cohesión, llevando la modularidad al siguiente nivel al hacer que los servicios sean completamente independientes en el desarrollo y la implementación.

Esta modularidad abre nuevas puertas a la hora de enfocar la construcción de un sistema desde el punto de vista tecnológico, ya que existe la posibilidad de usar una tecnología de programación diferente para microservicio del sistema. Los microservicios favorecen un ecosistema de desarrollo en el que unos servicios pueden estar desarrollados en Node.js para aprovechar la eficiencia de este lenguaje en la gestión de peticiones y operaciones de entrada y salida, mientras que otros pueden aprovecharse de la potencia de cálculo de Python para generar cantidades ingentes de información mediante el procesamiento masivo de datos. Esto permite que sea posible incorporar a la arquitectura nuevas tecnologías con un impacto de riesgo mínimo, lo que relaja la importancia de las decisiones tecnológicas que se toman en las fases iniciales del desarrollo.

Por otro lado, las aplicaciones modernas suelen estar diseñadas en torno a las API. Las APIs permiten a las aplicaciones reutilizar la lógica y aprovechar servicios innovadores, proporcionan acceso a datos o servicios valiosos, por lo que normalmente necesitan restringir el acceso a la API a las partes autorizadas. Como consecuencia, las aplicaciones necesitan autorización para llamar a las API. Si una aplicación quiere llamar a una API en nombre de un usuario para acceder a recursos de su propiedad, necesita el consentimiento de éste.

En el pasado, un usuario tenía que compartir a menudo sus credenciales con la aplicación para permitir una llamada a la API en su nombre. Esto le daba a la aplicación una cantidad innecesaria de acceso, sin mencionar la responsabilidad de salvaguardar la credencial que todo ello conllevaba. Con el tiempo, han surgido protocolos como OAuth 2.0 que proporcionan una mejor solución para autorizar a las aplicaciones a realizar estas llamadas a las API.

Utilizando OAuth2 y OpenID Connect, es importante entender cómo se produce el flujo de autorización, quién debe llamar al servidor de autorización, cuáles son y cómo se deben almacenar los tokens utilizados en la comunicación. Además, los microservicios y las aplicaciones cliente, como las aplicaciones móviles y SPA, plantean algunas cuestiones sobre qué flujo se aplica a las arquitecturas modernas de OAuth2.

Como conclusión de este apartado, la motivación por el desarrollo de este trabajo es consolidar los conocimientos base en materia de diseño y arquitectura software, así como en seguridad, profundizando en lograr un mayor entendimiento y comprensión de los conocimientos sobre el protocolo OAuth, así como sus distintos ámbitos de aplicación.

1.2. Objetivos del proyecto

El objetivo principal de este proyecto reside en el desarrollo de una aplicación web que siga una arquitectura de microservicios e implemente el protocolo OAuth para facilitar la implantación de los mecanismos de autorización que proporciona dicho protocolo en cualquier aplicación cuya arquitectura siga este enfoque de diseño.

La realización de este proyecto trae consigo la consecución de una serie de objetivos secundarios con un carácter puramente didáctico y formativo, con el fin de complementar y profundizar en los conocimientos y competencias adquiridos durante el desarrollo del máster, pero que, sin los cuales, no sería posible lograr el desarrollo principal.

Estos objetivos transversales al desarrollo de la aplicación son:

- Adquirir un mayor conocimiento en el diseño de aplicaciones basadas en microservicios, dada la fuerte presencia que tienen las aplicaciones con este diseño en el mercado laboral actual. Realizar un análisis teórico que permita situar este enfoque de diseño en el marco histórico de la arquitectura software con el fin de entender las ventajas que representa frente a otras alternativas más tradicionales, pero también, los peligros y riesgos de su utilización.

- Realizar un estudio que permita conocer y entender el funcionamiento del framework de OAuth2. Analizar en detalle sus diferentes flujos de implementación para conocer las diferentes situaciones en las que se puede utilizar dicho protocolo, sus limitaciones y la confusión que puede generar en cuanto a materia de autenticación y la forma idónea de aplicarlo en la aplicación que se pretende desarrollar, con independencia de la tecnología de programación subyacente que se esté utilizando.

- Profundizar en los conocimientos sobre Spring Framework y estudiar los proyectos y frameworks necesarios para poder llevar a cabo la ejecución del objetivo principal, aprovechando las ventajas de esta tecnología a la hora de crear servicios web independientes de forma rápida.

Como dominio específico, a modo de ejemplo, se plantea desarrollar una web de gestión de cursos académicos. Hay que recalcar que este dominio no es más que un ejemplo y que la consecución del objetivo principal del proyecto pretende lograr una solución general a la hora de aplicar el protocolo OAuth, identificando los actores clave de dicho protocolo y representándolos por medio de microservicios dedicados a un solo cometido.

Por lo tanto, este enfoque de diseño permite hacer frente a requisitos de otro dominio reemplazando los servicios relacionados con la gestión de cursos en pos de aquellos servicios que se deban desarrollar para dar respuesta a los nuevos requisitos.

1.3. Estructura del documento

El contenido de la memoria de este proyecto de fin de máster se divide en cuatro secciones principales.

En la primera sección, que finaliza con este capítulo, se ha intentado realizar un planteamiento general del contexto actual de las aplicaciones basadas en microservicios, de la importancia de utilizar mecanismos de seguridad como el protocolo OAuth para proteger el acceso a los recursos de dichas aplicaciones y de las ventajas que puede presentar este enfoque de diseño a la hora de agilizar el desarrollo de productos para dar respuesta a la creciente y cambiante demanda del mercado actual, para finalizar sentando las bases del proyecto que se pretende conseguir con la realización de este trabajo.

En la segunda sección del documento se presenta el marco teórico de cada uno de los temas principales que se identificaron como objetivos transversales a la realización de la aplicación. En primer lugar, se proporciona el contexto histórico necesario para entender de dónde vienen los microservicios, en qué arquitecturas se basa el enfoque que sigue esta tecnología y qué problemas pretende superar, analizando sus ventajas e inconvenientes. En segundo lugar, se realiza un estudio detallado del protocolo OAuth y su estado del arte, poniendo el foco en los diferentes tipos de concesiones y los actores involucrados en cualquier escenario de comunicación que requiera de un acceso controlado a los recursos de una aplicación. Por último, esta sección concluye con una descripción resumida del proyecto de Spring Framework y los subproyectos empleados en la construcción de la aplicación.

En el tercer bloque, una vez se ha definido el marco teórico, se realiza una descripción del marco práctico del proyecto. Aquí se exponen los requisitos que debe cumplir la plataforma a desarrollar y se lleva a cabo una explicación detallada de la arquitectura del sistema, así como de su implementación y funcionamiento.

En la cuarta sección del documento se encuentra la exposición de las conclusiones derivadas de la realización de este proyecto, se discuten las futuras líneas de desarrollo, así como el presupuesto estimado y se proporciona un listado de todas las referencias bibliográficas utilizadas, tanto para la realización del marco teórico como para el desarrollo del propio sistema.

2. ESTADO DEL ARTE

2.1. Microservicios

2.1.1. Introducción

Los microservicios son pequeños servicios capaces de ser desplegados de forma independiente que se modelan en torno a un dominio empresarial. Un servicio encapsula la funcionalidad relacionada con este dominio y la hace accesible a otros servicios de un sistema o del exterior a través de conexiones en red, permitiendo la construcción de sistemas más complejos a partir de estos bloques de construcción.

De este modo, en una aplicación dedicada a la venta de productos, un microservicio puede representar el inventario, otro puede representar la gestión de pedidos y un tercero el envío, pero juntos pueden constituir el sistema de comercio electrónico.

Los microservicios son una opción de arquitectura que se centra en dar muchas opciones para resolver los problemas a los que un desarrollador se puede enfrentar. Ofrecen un tipo de arquitectura orientada a servicios en el que se pone especial énfasis en cómo deben trazarse los límites de estos servicios, y en la que la capacidad de despliegue independiente es clave. Además, son agnósticos a la tecnología, lo que constituye una de las ventajas que ofrecen.

Desde el exterior, un microservicio es visto como una caja negra. Aloja la funcionalidad empresarial en uno o más *endpoints* de la red a través de los protocolos que sean más apropiados. Los consumidores, ya sean microservicios u otro tipo de programas, acceden a esta funcionalidad a través de estos *endpoints* en red.

Los detalles de la implementación interna (como la tecnología con la que está escrito el servicio o el modo en que se almacenan los datos) están totalmente ocultos al mundo exterior, lo cual significa que las arquitecturas de microservicios intentarán evitar el uso de elementos como bases de datos compartidas en la mayoría de las circunstancias. En su lugar, cada microservicio debe encapsular su propia base de datos si fuera necesario algún tipo de persistencia de la información.

Los microservicios implementan el concepto de ocultación de la información (*information hiding)*, el cual determina que es necesario ocultar la mayor cantidad de información posible dentro de un componente y exponer la menor cantidad posible a través de interfaces externas. Esto permite una clara separación entre lo que es susceptible de cambiar fácilmente y lo que es más complicado que pueda cambiar.

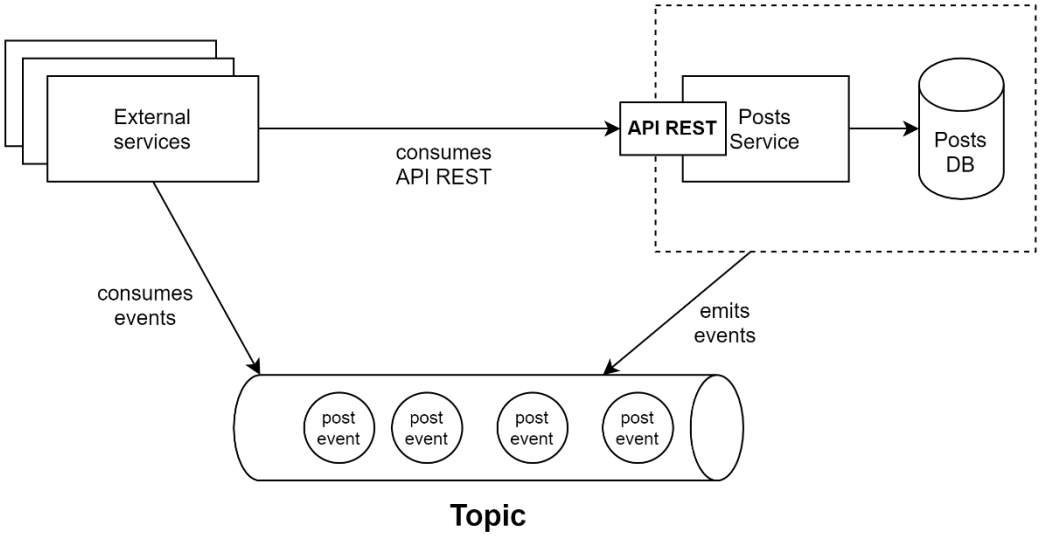

Figura 1. Ejemplo de arquitectura de microservicios simple

La implementación que se oculta a las partes externas puede cambiarse libremente siempre que las interfaces en red que el microservicio expone no incluyan cambios incompatibles con la versión pasada. Los cambios dentro de los límites de un microservicio (como se muestra en la figura 1.1) no deberían afectar a un consumidor anterior, lo que permite que la funcionalidad pueda ser lanzada de forma independiente. Esto es esencial para permitir que los microservicios se desarrollen de forma aislada y se puedan publicar bajo demanda. Tener unos límites de servicio claros y estables, que no cambien cuando la implementación interna cambie, da como resultado sistemas con bajo acoplamiento y una alta cohesión.

El patrón de Arquitectura Hexagonal es un ejemplo muy bueno de cómo ocultar los detalles de la implementación interna. Este patrón describe la importancia de mantener la implementación interna separada de sus interfaces externas, con la idea de que es posible que se quiera interactuar con la misma funcionalidad a través de diferentes tipos de interfaces.

2.1.2. Conceptos clave

A la hora de explorar los microservicios, es necesario pararse a estudiar detenidamente algunas de las ideas básicas en torno a las que giran, resultando vital explorar dichos conceptos para conseguir un mayor entendimiento de cómo funcionan los microservicios. En este apartado, se realiza una exposición de estos conceptos.

2.1.2.1. Despliegue independiente

Con el concepto de capacidad de despliegue independiente (*Independent Deployability)* se hace referencia a que se debe poder realizar un cambio en un microservicio, se despliegue y se publique a los usuarios finales sin que haya que desplegar ningún otro microservicio. Este principio va enfocado a la forma en que se deben gestionar los despliegues de las aplicaciones que siguen esta arquitectura y, si bien es una idea bastante sencilla en la teoría, resulta compleja de llevar a la práctica.

Para garantizar que se cumpla este principio es necesario asegurar que los microservicios estén débilmente acoplados, es decir, se debe poder modificar un servicio sin que esto implique tener que modificar otros servicios. Esto conlleva que se definan contratos de forma estable y explícita entre los servicios; si bien, algunas decisiones de diseño, como el uso de bases de datos compartidas, pueden complicar estos contratos.

Cumplir con este principio aporta muchos beneficios ya que los pasos necesarios para garantizar este cumplimiento ya aportan un valor añadido al sistema por sí mismos, logrando una serie de beneficios auxiliares. Por ejemplo, ayudando a definir los límites de los microservicios.

2.1.2.2. Modelado en torno a un dominio de negocio concreto

Es muy común encontrarse con arquitecturas por capas, como la arquitectura de tres niveles que se presenta en la figura. En este ejemplo, cada capa de la arquitectura representa un límite de servicio diferente, diferenciándose una capa de presentación, una capa de aplicación y una capa de datos. Esta división sería eficiente para gestionar cambios que sólo afecten a una de las capas como, por ejemplo, la capa de presentación. Sin embargo, es bastante frecuente que, en este tipo de arquitecturas, los cambios de una funcionalidad obliguen a realizar modificaciones en varias capas; un problema que se agrava en arquitecturas más complejas que las del ejemplo, donde hay cada capa se estructura en aún más niveles.

Sin embargo, con los microservicios se cambia el enfoque para priorizar la alta cohesión de la funcionalidad empresarial sobre la alta cohesión de la funcionalidad técnica.

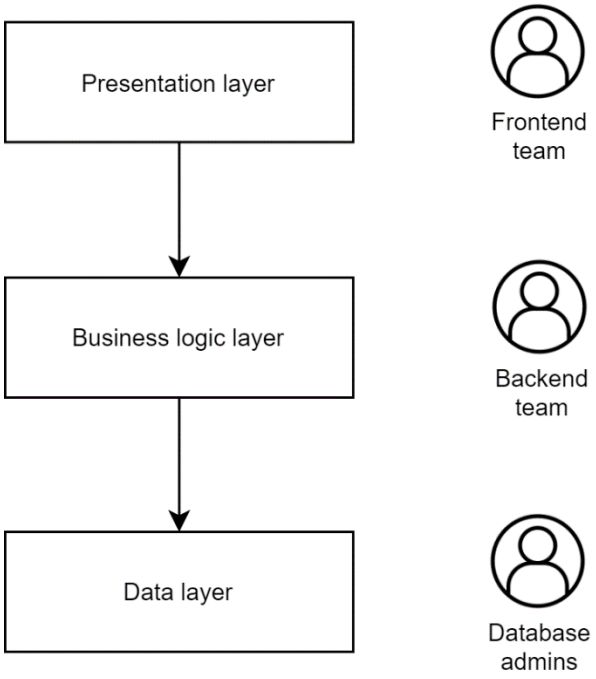

Figura 2. Arquitectura tradicional en tres capas.

Técnicas como el diseño guiado por el dominio (*domain-driven design*) permiten estructurar el código para representar de una forma más fiel el dominio del mundo real en el que opera el software que se construye. Con las arquitecturas basadas en microservicios se utiliza el mismo planteamiento a la hora de definir los límites de un servicio. Modelando los servicios en torno a un dominio del negocio concreto es más sencillo lanzar nuevas funcionalidades y recombinar los microservicios de diferentes maneras para ofrecer nuevas funcionalidades a los usuarios.

Desplegar una función que requiera cambios en más de un microservicio es costoso. Obliga a coordinar el trabajo en cada servicio (y potencialmente en equipos de desarrollo distintos) y a gestionar cuidadosamente el orden de despliegue de las nuevas versiones de estos servicios. Esto lleva mucho más trabajo que hacer el mismo cambio dentro de un solo servicio (o dentro de un monolito). Por lo tanto, hay que encontrar la manera de que los cambios entre servicios sean lo menos frecuentes posible.

2.1.2.3. Autogestión del estado

Uno de los aspectos más complicados de entender y aplicar a la hora de diseñar una arquitectura usando microservicios es la de idea de que los microservicios deben evitar el uso de bases de datos compartidas.

Si un microservicio quiere acceder a los datos que tiene otro microservicio, debe ir a pedir los datos a ese segundo microservicio. Esto da a los microservicios la capacidad de decidir qué se comparte y qué se oculta, lo que permite separar claramente la funcionalidad que puede cambiar libremente (su implementación interna) de la funcionalidad que se debe cambiar con menor frecuencia (el contrato externo que usan los consumidores).

Para hacer realidad el principio de despliegue independiente que se exponía en apartados anteriores es necesario asegurarse de limitar los cambios no retro compatibles en los microservicios. Rompiendo la compatibilidad con los consumidores externos se produce una reacción en cadena donde se obliga a estos consumidores a llevar a cabo más modificaciones para poder seguir consumiendo el microservicio.

Por tanto, tener una delineación limpia entre los detalles de la implementación interna y un contrato externo para un microservicio puede ayudar a reducir la necesidad de cambios incompatibles hacia atrás.

Ocultar el estado interno en un microservicio es análogo a la práctica de la encapsulación de la programación orientada a objetos.

2.1.2.4. Tamaño

Una de las dudas más comunes en los desarrolladores a la hora de diseñar una arquitectura con microservicios es el tamaño que deben tener los microservicios.

Aquí hay que hacer una pequeña reflexión sobre qué significa el tamaño en un servicio y cómo se mide. No parece muy inteligente asociar el tamaño a una cantidad de líneas de código, ya que hay lenguajes de programación más expresivos que otros y que necesitan de menos código para conseguir lo mismo, pero esto no significa que sean mejores o peores.

La conclusión a la que llega varios autores que un microservicio debe mantenerse en un tamaño en el que se pueda entender fácilmente. El reto que aparece aquí es que la capacidad de distintas personas para entender algo no es siempre la misma, y como tal, queda en el desarrollador del microservicio determinar qué tamaño funciona para lo que está construyendo. Un equipo experimentado puede ser capaz de gestionar mejor un código base más grande que otro equipo.

Lo más cercano a que el tamaño tenga algún significado en términos de microservicios es algo que Chris Richardson expone en su libro *Microservice Patterns*: el objetivo de los microservicios es tener una interfaz lo más pequeña posible. Eso se alinea de nuevo con el principio de ocultación de información, pero representa un intento de encontrar un significado al término "microservicios" que no estaba allí inicialmente, ya que cuando el término se utilizó por primera vez para definir estas arquitecturas, el enfoque inicial no era específicamente el tamaño de las interfaces.

Viendo que el tamaño de un microservicio es algo tan circunstancial, resulta más interesante centrarse en decidir cuántos microservicios se pueden manejar simultáneamente (ya que, a mayor cantidad de microservicios en un sistema, mayor será su complejidad) y cuáles van a ser los límites de dominio de los microservicios para sacarles el máximo partido posible.

2.1.2.5. Alineación de la arquitectura con la organización

Supóngase una arquitectura de capas como la mostrada anteriormente en la figura 2. En ella, veíamos una interfaz de usuario web, una capa de lógica empresarial en forma de *backend* y una capa de persistencia de datos en una base de datos tradicional. Estas capas normalmente son gestionadas por diferentes equipos.

Para entender el concepto al que se hace referencia en este apartado se plantea la siguiente temática para la arquitectura del ejemplo: una aplicación de comercio online. ¿Qué ocurriría ante un cambio en la funcionalidad como, por ejemplo, la introducción de direcciones vinculados a los usuarios para agilizar el proceso de compra de un producto? Como ya se adelantaba en el apartado de modelado en torno a un dominio del negocio, ante la aparición de una modificación como esta para introducir una funcionalidad va a ser necesario cambiar la UI, el servicio de *backend* y la base de datos, obligando a cada equipo a gestionar estos cambios y a desplegarlos en el orden correcto. En la Figura 3 se puede ver de forma gráfica el alcance que tendría un cambio en la arquitectura de capas de ejemplo.

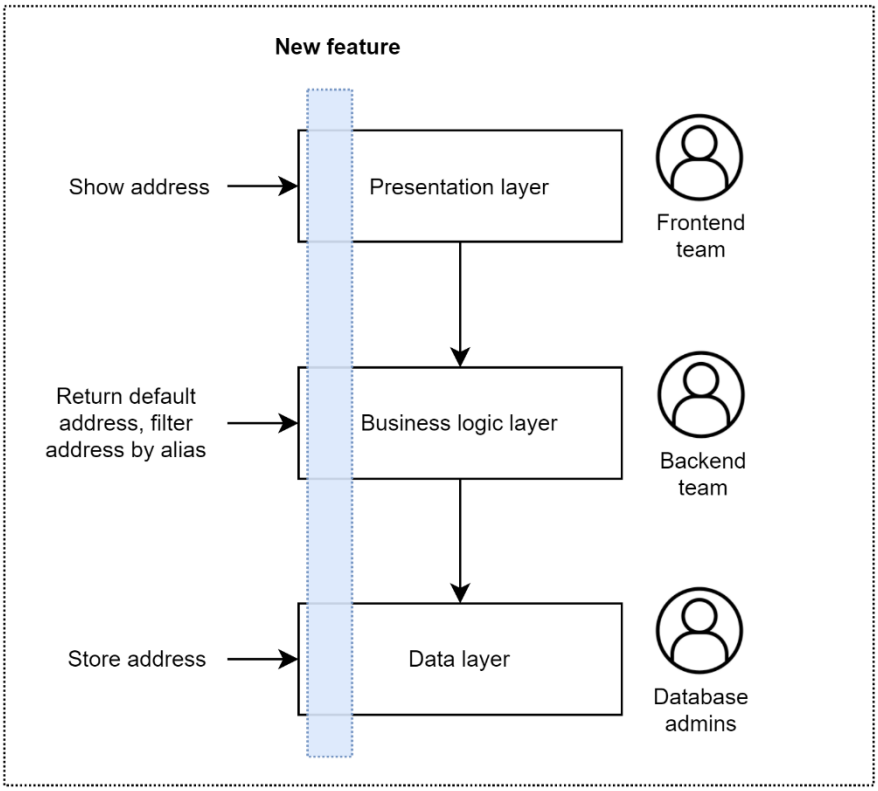

Figura 3. Efecto de la introducción de un cambio en una arquitectura de tres capas.

Esto no quiere decir que la arquitectura de tres niveles sea mala. Es una arquitectura muy extendida universalmente y todo es raro un desarrollador que no haya oído hablar de ella. La tendencia a elegir una arquitectura común que se pueda haber visto en otros lugares es a menudo

una de las razones por las que se sigue viendo este patrón. Sin embargo, en este caso, la mayor razón por la que vemos esta arquitectura una y otra vez es porque toda arquitectura acaba optimizándose en torno a un conjunto de objetivos y, este caso, como dice la ley de Conway: "Las organizaciones que diseñan sistemas [...] están abocadas a producir diseños que son copias de las estructuras de comunicación de dichas organizaciones" [23].

La arquitectura de tres niveles es un buen ejemplo de lo que hace referencia esta ley. En el pasado, la forma principal en que las organizaciones de TI agrupaban a las personas era en función de su competencia principal: los administradores de bases de datos estaban en un equipo con otros administradores de bases de datos; los desarrolladores *backend* estaban en un equipo con otros desarrolladores *backend*; y los desarrolladores de *frontend* estaban en otro equipo. Al final, las personas se agrupan en función de su competencia principal, por lo que se crean activos de IT que pueden alinearse con esos equipos.

Ahora bien, el diseño de arquitecturas basadas en microservicios provoca un cambio de paradigma que hace que las aspiraciones en torno a la construcción del software también cambien. Las organizaciones han comenzado a agrupar a las personas en equipos polivalentes para reducir los traspasos y los silos. El software tiene que ser distribuido mucho más rápido que antes y eso lleva a tomar decisiones diferentes sobre la forma de organizar los equipos.

La mayoría de los cambios que se piden para un sistema están relacionados con cambios en la funcionalidad del negocio. Pero en la Figura 3, dicha funcionalidad empresarial está repartida entre los tres niveles, lo que aumenta la posibilidad de que un cambio en la funcionalidad cruce las capas. Se trata de una arquitectura que tiene una alta cohesión de la tecnología relacionada, pero una baja cohesión de la funcionalidad empresarial. Si se quiere que sea más fácil hacer cambios, hay que cambiar la forma de agrupar el código, eligiendo la cohesión de la funcionalidad del negocio en lugar de la tecnología. Cada servicio puede o no acabar conteniendo una mezcla de estas tres capas, pero eso es una cuestión de implementación local del servicio.

A continuación, se propone una posible arquitectura alternativa ilustrada en la Figura 4. En lugar de una arquitectura y organización en capas horizontales, se desglosa la organización y arquitectura a lo largo de líneas de negocio verticales. Aquí se puede apreciar un equipo dedicado que tiene la responsabilidad completa de realizar cambios en aspectos del perfil del cliente, lo que garantiza que el alcance del cambio en este ejemplo se limita a un equipo. Continuando con el ejemplo del comercio electrónico, en este escenario podría tenerse un equipo encargado del servicio de autorización, otro equipo encargado del servicio del carro de la compra y un tercer equipo encargado de la funcionalidad relacionada con el usuario.

El desarrollo del cambio podría lograrse a través de un único microservicio propiedad del equipo de usuario, que expone una UI para permitir a los clientes actualizar su información, con el estado del cliente también almacenado dentro de este microservicio.

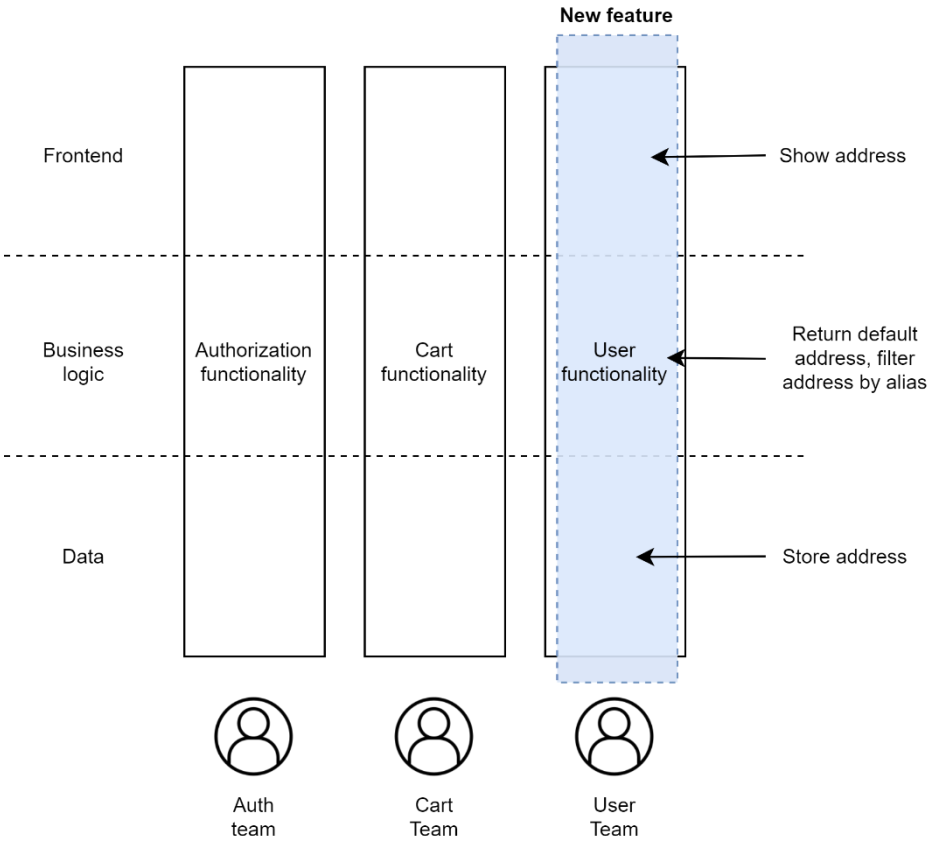

Figura 4. Efecto de la introducción de un cambio en una arquitectura de microservicios.

En esta situación, el microservicio de usuario encapsula una pequeña porción de cada uno de los tres niveles: tiene un poco de interfaz de usuario, un poco de lógica de aplicación y un poco de almacenamiento de datos. El dominio empresarial se convierte en la fuerza principal que impulsa la arquitectura del sistema, lo que, con suerte, facilita la realización de cambios, así como la alineación de los equipos con las líneas de negocio dentro de la organización.

A menudo, la interfaz de usuario no es proporcionada directamente por el microservicio, pero incluso si este es el caso, se esperaría que la parte de la interfaz de usuario relacionada con esta funcionalidad siguiera siendo propiedad del equipo del servicio de usuario, como indica la figura 4.

2.1.3. El papel de las arquitecturas monolíticas en el desarrollo de los microservicios

Los principales lenguajes para el desarrollo de aplicaciones del lado del servidor, como Java, C/C++ y Python, ofrecen abstracciones para descomponer la complejidad de los programas en módulos. Sin embargo, estos lenguajes están diseñados para la creación de artefactos ejecutables únicos, también llamados monolitos, y sus abstracciones de modularización se basan en la compartición de recursos de la misma máquina (memoria, bases de datos, archivos).

A lo largo de este documento se ha hablado de los microservicios y los principios que se deben tener en cuenta para diseñar una arquitectura utilizándolos como base. Hablando de arquitecturas de diseño software, los microservicios tienden a ser presentados como un enfoque alternativo a la arquitectura monolítica. Para comprender este contraste y entender mejor las ventajas de la implementación de microservicios y sus orígenes, se va a realizar una breve exposición sobre el concepto de monolito.

Una aplicación de software monolítica es una aplicación de software compuesta por módulos que no son independientes de la aplicación a la que pertenecen.

Como los módulos de un monolito dependen de dichos recursos compartidos, no son ejecutables de forma independiente. Esto hace que los monolitos sean difíciles de distribuir de forma natural sin el uso de marcos específicos o soluciones ad hoc como, por ejemplo, Network Objects, RMI o CORBA. En el contexto de los sistemas distribuidos basados en la nube, esto representa una limitación importante, en particular porque las soluciones anteriores dejan las responsabilidades de sincronización en manos del desarrollador.

2.1.3.1. Clasificación de arquitecturas monolíticas

A lo largo de este capítulo se utiliza el término monolito principalmente para hacer referencia a una unidad de despliegue. Es decir, cuando toda la funcionalidad de un sistema debe desplegarse conjuntamente, se está ante un monolito. Hay muchas arquitecturas que se ajustan a esta definición, pero, a continuación, se van a detallar aquellas que pueden identificarse con más frecuencia en los sistemas del mundo real: el monolito de proceso único, el monolito modular y el monolito distribuido.

2.1.3.1.1. Arquitectura monolítica de proceso único

El ejemplo más común cuando se habla de monolitos es un sistema en el que todo el código se despliega como un solo proceso (Figura 5). Es posible tener múltiples instancias de este proceso por razones de robustez o escalado, pero todo el código está empaquetado en un único proceso.

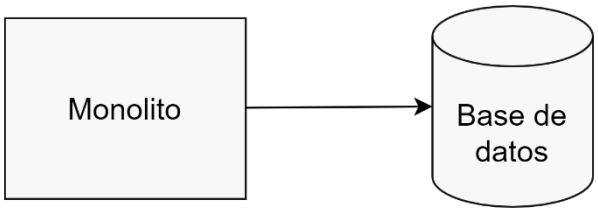

Figura 5. Monolito de proceso único

Aunque este esquema se ajusta a lo que la mayoría de los desarrolladores entienden por un monolito clásico, la mayoría de los sistemas que se encuentran en el mundo real son algo más complejos. Es posible que haya dos o más monolitos estrechamente acoplados entre sí, y en ocasiones existe algún software de proveedor en ellos.

Este tipo de arquitecturas puede tener sentido para muchas organizaciones. David Heinemeier Hansson [24], el creador de Ruby on Rails, ha defendido eficazmente que este tipo de arquitectura tiene sentido para las organizaciones más pequeñas. Sin embargo, cuando la organización crece el monolito puede crecer con ella lo que nos lleva al siguiente tipo de arquitectura monolítica: el monolito modular.

2.1.3.1.2. Arquitectura monolítica modular

Como subconjunto del monolito de proceso único, la arquitectura monolítica modular es una variación en la que el proceso único es descompuesto en módulos separados. Cada módulo puede ser desarrollado de forma independiente, pero todos deben ser combinados juntos para el despliegue, como se muestra en la Figura 6.

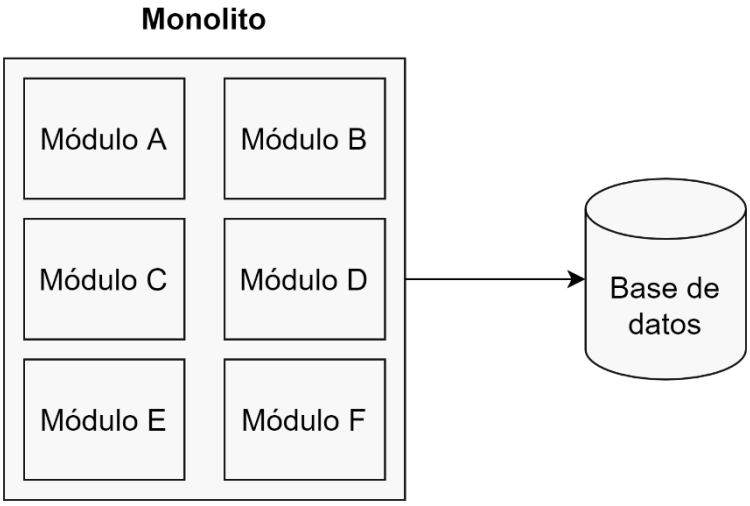

Figura 6. Monolito modular con una única base de datos.

La práctica de dividir el software en módulos tiene sus raíces en el trabajo realizado en torno a la programación estructurada en la década de 1970, e incluso más atrás.

Para muchas organizaciones, el monolito modular puede ser una excelente opción. Si los límites de los módulos están bien definidos, puede permitir un alto grado de trabajo en paralelo, al tiempo que evita los desafíos de la arquitectura de microservicios más distribuida al tener una topología de despliegue mucho más simple. Shopify es un gran ejemplo de una organización que ha utilizado esta técnica como alternativa a la descomposición de microservicios. [25]

Uno de los retos de un monolito modular es que la base de datos no suele presentar la descomposición que se encuentra a nivel de código, lo que conduce a desafíos significativos si se quiere desmontar el monolito en el futuro. Aunque algunos equipos intentan llevar la idea del monolito modular más allá haciendo que la base de datos se descomponga siguiendo las mismas líneas que los módulos, como se muestra en la Figura 7.

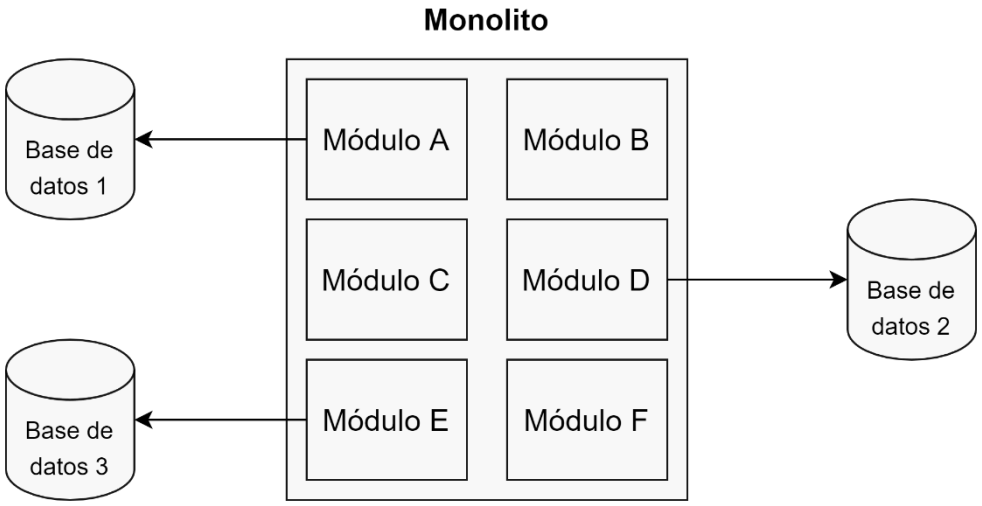

Figura 7. Monolito modular con varias bases de datos.

2.1.3.1.3. Arquitectura monolítica distribuida

Una arquitectura monolítica distribuida es un sistema que consta de múltiples servicios claramente diferenciados que, a la hora de desplegarse, han de hacerlo de manera conjunta.

Un monolito distribuido tiene todas las desventajas de un sistema distribuido, y las desventajas de una arquitectura monolítica de proceso único, sin tener suficientes ventajas de ninguno de los dos.

Este tipo de arquitecturas suelen surgir en un entorno en el que no se ha prestado suficiente atención a conceptos como la ocultación de información y la cohesión de la funcionalidad empresarial. En su lugar, las arquitecturas altamente acopladas hacen que los cambios se propaguen a través de los límites de los servicios, y los cambios aparentemente inocentes que parecen ser de alcance local rompen otras partes del sistema.

2.1.3.2. Ventajas de los monolitos

En este apartado se exponen algunas de las bondades que trae consigo el uso de los monolitos. Algunos tipos de arquitecturas monolíticas, como las de proceso único o las modulares, tienen toda una serie de ventajas. Su topología de despliegue, mucho más sencilla, puede evitar muchos de los escollos asociados a los sistemas distribuidos, lo que puede dar lugar a flujos de trabajo mucho más sencillos para los desarrolladores. Al mismo tiempo, la supervisión, la resolución de problemas y actividades como las pruebas de extremo a extremo también pueden simplificarse en gran medida.

Los monolitos también pueden facilitar la reutilización de código dentro del propio monolito. Si se quiere reutilizar el código dentro de un sistema distribuido, hay que decidir si quiere copiar el código, separar las bibliotecas, o convertir la funcionalidad compartida en un servicio. Con un monolito, las opciones son mucho más sencillas, y son muchos los desarrolladores que prefieren esta simplicidad. Solo hay que definir bien el código y reutilizarlo cuando sea necesario.

Por desgracia, la tendencia actual ha hecho que los desarrolladores tiendan a ver el monolito como algo que debe evitarse, algo inherentemente problemático. Parece que este tipo de arquitecturas son sinónimo de software de legado, lo cual es un problema.

Una arquitectura monolítica es una opción, y una opción válida, muy adecuada como opción por defecto a la hora de empezar a trabajar con los diferentes estilos arquitectónico.

2.1.3.3. Obstáculos de los monolitos

En este apartado, se enumeran los obstáculos más relevantes de los monolitos:

- Los monolitos de gran tamaño son difíciles de mantener y ampliar a largo plazo debido a su creciente complejidad. La búsqueda de errores requiere largos recorridos por su base de código.
- El llamado "infierno de las dependencias", en el que la adición o actualización de librerías da lugar a sistemas inconsistentes que no compilan o ejecutan o, peor aún, presentan un comportamiento inesperado por conflictos entre las librerías.
- Cualquier cambio en un módulo de un monolito implica que haya que reiniciar toda la aplicación. Para proyectos de gran tamaño, este reinicio suele implicar tiempos de inactividad sensibles, lo que dificulta el desarrollo, las pruebas y el mantenimiento del proyecto.
- El despliegue de aplicaciones monolíticas no suele ser óptimo debido a los requisitos conflictivos de los recursos de los módulos que las componen: algunos pueden ser intensivos en memoria, otros en computación y otros requieren componentes ad hoc como bases de datos basadas en SQL en lugar de en gráficos. A la hora de elegir un entorno de despliegue, el desarrollador debe comprometerse con una configuración única que resulte costosa a nivel económica o poco óptima con respecto a los módulos individuales.

- Los monolitos limitan la escalabilidad. La estrategia habitual para gestionar incrementos de peticiones entrantes es crear nuevas instancias de la misma aplicación y repartir la carga entre dichas instancias. Sin embargo, podría darse el caso de que el aumento de tráfico estresara sólo a un subconjunto de los módulos, haciendo que la asignación de los nuevos recursos para los demás componentes sea innecesaria y se acaben por desaprovechar recursos.
- Los monolitos pueden representan un bloqueo tecnológico para los desarrolladores, que están obligados a utilizar el mismo lenguaje de programación y los mismos *frameworks* de la aplicación original.

2.1.4. Beneficios de la adopción de microservicios

Las ventajas que puede ofrecer una arquitectura basada en microservicios a los desarrolladores son muchas y variadas. Algunas de estas ventajas se pueden atribuir a cualquier sistema distribuido, no solo a los microservicios. Sin embargo, los microservicios tienden a lograr estas ventajas en mayor medida, principalmente porque adoptan una postura más decidida en la forma de definir los límites del servicio. Al combinar los principios de ocultación de la información y el diseño orientado al dominio con la potencia de los sistemas distribuidos, los microservicios consiguen ofrecer ventajas significativas sobre otras formas de arquitecturas distribuidas.

2.1.4.1. Heterogeneidad tecnológica

Con un sistema compuesto por múltiples microservicios que colaboran entre sí, los desarrolladores tienen la posibilidad de decidir utilizar diferentes tecnologías dentro de cada uno de ellos. Esto permite elegir la herramienta adecuada para cada desarrollo en lugar de tener que seleccionar un enfoque más estandarizado, una tecnología de propósito general que pueda aplicarse a cada uno de los servicios y que muchas veces no acaba siendo la elección óptima.

Si una parte del sistema necesita una mejora en cuanto a rendimiento, podría decidirse utilizar un *stack* de tecnologías diferente que asegure el cumplimiento de estos requisitos. También podría decidirse que la forma de almacenamiento de datos debe cambiar para las diferentes partes de nuestro sistema. Por ejemplo, en el caso de una red social, las interacciones de los distintos usuarios podrían almacenarse en una base de datos orientada a grafos para reflejar la naturaleza altamente interconectada de un grafo social, pero, por otro lado, las publicaciones que hacen los usuarios podrían almacenarse en una base de datos orientado a documentos, dando lugar a una arquitectura heterogénea como la que se muestra en la Figura 8.

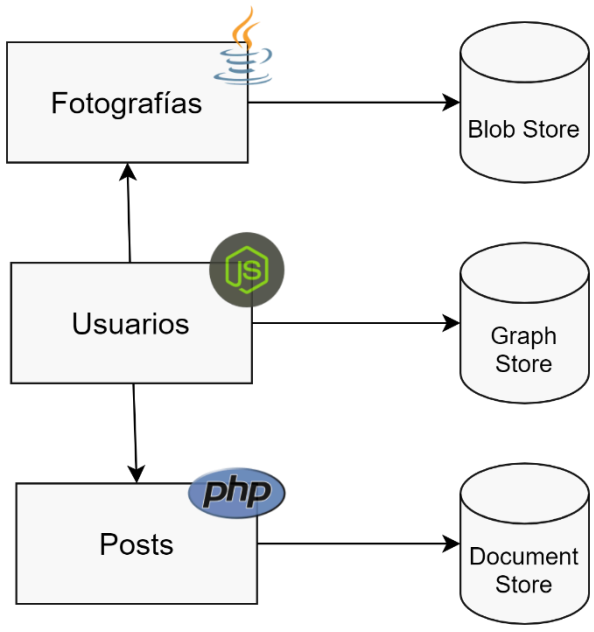

Figura 8. Microservicios usando diferentes tecnologías.

Con los microservicios, también se hace posible adoptar más rápido las tecnologías y comprender cómo pueden ayudar los nuevos avances del sector. Uno de los mayores obstáculos a la hora de probar y adoptar una nueva tecnología son los riesgos asociados a ella. Con una aplicación monolítica, si se desea probar un nuevo lenguaje de programación, base de datos o framework, cualquier cambio afectará a gran parte del sistema. Por otro lado, con un sistema formado por múltiples servicios, aparecen muchas oportunidades donde es viable probar una nueva tecnología. Se puede elegir un microservicio con menor riesgo y utilizar la tecnología allí, sabiendo que cualquier impacto negativo está limitado al ámbito de aplicación de dicho servicio. Muchas organizaciones consideran que esta capacidad de absorber más rápidamente las nuevas tecnologías es una verdadera ventaja.

Adoptar múltiples tecnologías no viene sin gastos, por supuesto. Algunas organizaciones optan por imponer algunas restricciones a la hora de elegir el lenguaje de programación. Netflix y Twitter, por ejemplo, utilizan principalmente la máquina virtual de Java (JVM) como plataforma porque estas empresas conocen muy bien la fiabilidad y el rendimiento de ese sistema. También desarrollan bibliotecas y herramientas para la JVM que facilitan el funcionamiento a escala, pero la dependencia de bibliotecas específicas de la JVM dificulta las cosas para los servicios o clientes no basados en Java.

El hecho de que la implementación de la tecnología interna esté oculta a los consumidores también puede facilitar la actualización de las tecnologías. Toda la arquitectura de microservicios podría estar basada en Spring Boot, por ejemplo, pero podría cambiarse la versión de la JVM o las versiones del framework para un solo microservicio, lo que facilitaría la gestión del riesgo de las actualizaciones.

2.1.4.2. Robustez

Un concepto clave para mejorar la robustez de una aplicación es el patrón *Bulkhead*. Un componente de un sistema puede fallar, pero mientras ese fallo no se propague en cascada, se podrá aislar el problema y el resto del sistema podrá seguir funcionando con normalidad. Los límites del microservicio se convierten en esas barreras que evitan la propagación del error.

Cuando se produce un error en un sistema con una arquitectura monolítica, todo el conjunto del sistema podría dejar de funcionar. Para reducir la posibilidad de fallo, el sistema monolítico podría replicarse y ejecutarse de forma duplicada en varias máquinas. Ahora bien, con los microservicios es posible construir sistemas que gestionen un fallo crítico de algunos de los servicios que constituyen el servicio y degradar la funcionalidad en consecuencia, sin más complicaciones para el resto de la arquitectura.

Sin embargo, es necesario tener cuidado. Para asegurarse de que los sistemas de microservicios puedan adoptar adecuadamente esta robustez mejorada, es necesario comprender los nuevos puntos de fallo con las que tienen que lidiar los sistemas distribuidos. Ahora, además de la implementación interna de un servicio, también pueden fallar las redes de comunicación que se utilizan. Por lo tanto, es responsabilidad de los desarrolladores conocer cómo manejar esos fallos y el impacto que tendrán en los usuarios finales del software.

2.1.4.3. Escalado

En referencia al escalado de una aplicación, con un servicio monolítico se necesita escalar todo el sistema al completo. Es posible que una pequeña parte del sistema global tenga un rendimiento limitado, pero si ese comportamiento está encerrado en una aplicación monolítica, habrá que manejar el escalado de todo como una sola unidad. Con servicios más pequeños, es posible escalar sólo aquellos servicios que necesitan ser escalados, permitiendo ejecutar otras partes del sistema en hardware más pequeño y menos potente, como se ilustra en la Figura 9.

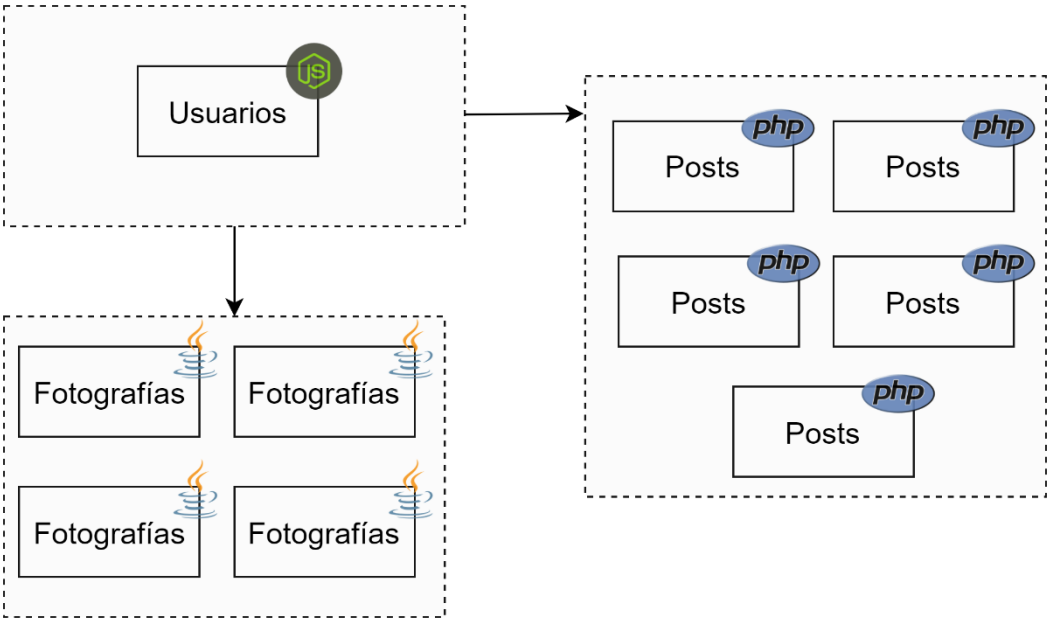

Figura 9. Escalado de servicios.

Al adoptar sistemas de aprovisionamiento bajo demanda como los que ofrecen los proveedores de servicios en la nube como Amazon Web Services, es posible incluso aplicar este escalado bajo demanda a aquellas piezas de software que lo necesiten, lo que facilita controlar los costes económicos de forma mucho más eficaz. No es frecuente que un enfoque arquitectónico pueda estar tan estrechamente relacionado con un ahorro de costes casi inmediato.

En definitiva, las aplicaciones se pueden escalar de múltiples maneras, y los microservicios pueden ser una parte eficaz de ello.

2.1.4.4. Facilidad de despliegue

Un cambio de una sola línea de código en una aplicación monolítica de un millón de líneas requiere el despliegue de toda la aplicación para publicar el cambio y que este llegue al usuario final. Esto convierte el despliegue en un proceso de gran impacto y alto riesgo.

En la práctica, este provoca que este tipo de despliegues acaben produciéndose con poca frecuencia por un miedo comprensible por parte del equipo de desarrollo. Desgraciadamente, esto se acaba traduciendo en que se acumulan una gran cantidad de cambios entre versiones y cuando la nueva versión de la aplicación tiene que entrar en producción, trae consigo una cantidad de cambios muy complicada de manejar en caso de que aparezca algún error en el proceso de despliegue. Cuanto mayor sea el delta entre versiones, mayor será el riesgo de equivocación.

Con los microservicios, se puede realizar un cambio en un solo servicio y desplegarlo independientemente del resto del sistema. Esto permite, a su vez, desplegar el código de una forma mucho más rápida. Si se produce un problema, se puede aislar rápidamente en un servicio individual, lo que facilita regresar a una versión anterior segura en caso de ser necesario. También significa que se pueden hacer llegar las nuevas funcionalidades a los clientes de forma más veloz.

Esta es una de las principales razones por las que organizaciones como Amazon y Netflix utilizan estas arquitecturas, asegurarse de que eliminan el mayor número posible de impedimentos para sacar el software de forma rápida al mercado.

2.1.4.5. Alineación con la organización

Son muy comunes los problemas asociados a los grandes equipos y a las grandes bases de código. Estos problemas pueden agravarse cuando el equipo está distribuido. También se sabe que los equipos más pequeños que trabajan en bases de código más pequeñas tienden a ser más productivos.

Los microservicios permiten alinear mejor la arquitectura con la organización de la empresa, ayudando a minimizar el número de personas que trabajan en cualquier base de código para alcanzar el punto óptimo de tamaño de equipo y productividad.

Los microservicios también posibilitan realizar modificaciones en la propiedad de los servicios a medida que cambia la organización, lo que permite mantener la alineación entre la arquitectura y la organización en el futuro.

2.1.4.6. Componibilidad

Una de las principales promesas que traen consigo el uso de sistemas distribuidos y, en general, las arquitecturas orientadas a servicios es que abren oportunidades para la reutilización de las funcionalidades desarrolladas. Con los microservicios, es posible que una funcionalidad sea consumida de diferentes maneras para diferentes propósitos. Esto puede ser especialmente importante a la hora de analizar la forma que tienen los usuarios de consumir el software.

En el pasado, los equipos de desarrollo únicamente tenían que pensar en cómo construir los sitio web o las aplicaciones móviles. Ahora, en cambio, tienen que pensar en las innumerables formas en las que podrían entrelazarse las capacidades para la web, las aplicaciones nativas, las *webapps*, las aplicaciones para tablets o incluso para dispositivos *wearables*. A medida que las

organizaciones dejan de pensar en términos de canales estrechos y adoptan conceptos más holísticos de compromiso con el cliente, se necesitan arquitecturas que puedan seguir el ritmo de estos requisitos crecientes.

2.1.5. Problemas asociados al uso de microservicios

Las arquitecturas de microservicios traen consigo una gran cantidad de beneficios, como se ha expuesto de forma detallada en la sección anterior. Pero los microservicios también traen consigo una gran complejidad. A la hora de considerar adoptar una arquitectura de microservicios, es imprescindible hacer una balanza entre lo bueno y lo malo asociado a esta tecnología.

En realidad, la mayoría de los aspectos negativos que se derivan del uso de microservicios se pueden achacar a los sistemas distribuidos y, por tanto, sería igual de evidente en un monolito distribuido que en una arquitectura de microservicios.

2.1.5.1. La experiencia de los desarrolladores

A medida que se tienen más y más servicios, la experiencia del desarrollador puede empezar a verse comprometida. Los tiempos de ejecución más intensivos en recursos, como la JVM, pueden limitar el número de microservicios que pueden ejecutarse en una sola máquina de desarrollador. Probablemente podrían ejecutarse cuatro o cinco microservicios basados en JVM como procesos separados en un ordenador portátil, pero lo más probable es que no sea factible ejecutar 10, 15 y, ni mucho menos, 20 servicios.

Incluso con tiempos de ejecución menos exigentes, hay un límite en el número de servicios o procesos que un desarrollador puede ejecutar de forma local en su máquina.

Este escenario se complica aún más si se utilizan servicios en la nube, que no se pueden ejecutar localmente, haciendo que los ciclos de retroalimentación puedan sufrir mucho.

Limitar el alcance de las partes de un sistema en las que debe trabajar un desarrollador puede ser un enfoque mucho más sencillo. Sin embargo, esto podría ser problemático si se quiere adoptar un modelo de propiedad colectiva en el que se espera que cualquier desarrollador pueda trabajar en cualquier parte del sistema.

2.1.5.2. Sobrecarga tecnológica

El peso de la nueva tecnología que ha surgido para permitir la adopción de arquitecturas de microservicios puede ser abrumador. Gran parte de esta tecnología ha sido tildada de "*microservice friendly*" y algunos avances han ayudado legítimamente a lidiar con la complejidad de este tipo de arquitecturas.

Sin embargo, existe el peligro de que esta riqueza de nuevas herramientas pueda conducir a una forma de fetichismo tecnológico en el que muchas empresas adoptan la arquitectura de microservicios e introducen una gran cantidad de tecnología nueva, a menudo extraña, únicamente guiados por la tendencia del mercado, sin pararse a analizar de forma sosegada cuales son las verdaderas necesidades de su negocio.

Como se ha expuesto en el apartado de beneficios de los microservicios, éstos pueden dar la opción de que cada microservicio esté escrito en un lenguaje de programación diferente, que se ejecuten en un tiempo de ejecución diferente o que utilicen una base de datos diferente. Pero todas estas posibilidades no son más que opciones, no requisitos. Hay que equilibrar cuidadosamente la amplitud y la complejidad de la tecnología que se utiliza frente a los costes que puede suponer un conjunto diverso de tecnologías.

Cuando se empieza a adoptar los microservicios, algunos retos fundamentales son ineludibles: habrá que dedicar mucho tiempo a entender los problemas relacionados con la coherencia de los datos, la latencia, el modelado de los servicios, etc.

Intentar entender cómo estas ideas cambian la forma de pensar en el desarrollo de software al mismo tiempo que se empiezan a implementar una gran cantidad de nuevas tecnologías puede dar lugar a un mal diseño de la arquitectura. También vale la pena señalar que el tiempo ocupado en tratar de entender toda esta nueva tecnología reducirá el tiempo del que se dispone en el proceso de desarrollo para hacer que las nuevas funcionalidades lleguen a los usuarios.

Lo ideal es que, a medida que aumente de forma gradual la complejidad de la arquitectura de microservicios, se busque la manera de introducir las nuevas tecnologías en función de las necesidades que surjan. No es necesario un clúster de Kubernetes cuando se tienen únicamente tres o cuatro servicios. Además de asegurar que no se produce una sobrecarga con la complejidad de estas nuevas herramientas, este aumento gradual tiene el beneficio añadido de permitir obtener nuevas y mejores formas de hacer las cosas que, sin duda, surgirán con el tiempo.

2.1.5.3. Coste

Cuando se comienzan a usar microservicios es muy probable que, a corto plazo, se produzca un aumento de los costes por una serie de factores.

En primer lugar, se incurre en una serie de costes económicos estrictamente hablando, derivados del aumento en el número de procesos a ejecutar, equipos informáticos para ejecutar estos procesos, recursos de red, de almacenamiento y software de apoyo, que supondrá el pago de licencias adicionales.

En segundo lugar, cualquier cambio que se introduce en un equipo o una organización produce una disminución del rendimiento de los desarrolladores a corto plazo. Se necesita tiempo para aprender las nuevas ideas y para saber cómo utilizarlas de forma eficiente, y mientras esto ocurre, otras actividades se verán afectadas. Esto se traducirá en una ralentización directa de la entrega de nuevas funcionalidades o en la necesidad de añadir más personal para compensar este coste.

Los microservicios pueden considerarse una mala elección para una organización preocupada principalmente por la reducción de costes, ya que una mentalidad de recorte de costes, en la que las TI se ven como un centro de costes en lugar de un centro de beneficios, será constantemente un lastre para sacar el máximo partido a esta arquitectura. Por otro lado, los microservicios pueden ayudar a ganar más dinero si se es capaz de utilizar estas arquitecturas para llegar a más clientes o desarrollar más funcionalidad en paralelo.

En resumen, los microservicios podrían llegar a ser una forma de obtener más beneficios, pero difícilmente pueden considerarse como un medio para reducir costes.

2.1.5.4. Supervisión y resolución de problemas

Como se vio en la sección de arquitecturas monolíticas, un monolito suele ir acompañado de su propia base de datos. Los *stakeholders* que puedan requerir de análisis estadísticos del sistema, a menudo implicando grandes operaciones de unión a través de los datos, tienen un esquema listo contra el cual obtener sus informes. Pueden ejecutarlos las consultas contra la base de datos monolítica o quizás contra una réplica de lectura.

El problema de la arquitectura basada en microservicios es que este esquema se rompe, pero esto no significa que haya desaparecido la necesidad de elaborar informes sobre todos los datos; simplemente se ha convertido en algo mucho más complejo. Ahora, en lugar de tener los datos centralizados en un punto, están dispersos en múltiples esquemas lógicamente aislados.

Los enfoques más modernos de la elaboración de informes, como el uso de streaming para permitir la elaboración de informes en tiempo real sobre grandes volúmenes de datos, pueden funcionar bien con una arquitectura de microservicios, pero normalmente requieren la adopción de nuevas ideas y tecnologías.

Como alternativa, se podrían publicar los datos de los microservicios en bases de datos de informes centrales y definir APIs que permitan acceder a estos informes según los casos de uso de los informes.

Otra de las ventajas de una aplicación monolítica es que permite tener un enfoque bastante simplista de la monitorización de procesos y recursos. Los desarrolladores solo tienen que preocuparse de un pequeño número de máquinas y el modo de fallo de la aplicación es, en cierto modo, binario: la aplicación funciona o está caída. Con una arquitectura de microservicios, comprender lo que está sucediendo cuando una solo instancia del servicio se cae es mucho más complicado.

2.1.5.5. Seguridad

En un sistema con arquitectura monolítica de proceso único, toda la información se queda dentro de dicho proceso, no es compartida con el exterior. En un sistema basado en microservicios, la información tiene que viajar por la red para ser transmitida de un servicio a otro.

Esto puede hacer que los datos sean más vulnerables a ser observados en tránsito y también a ser potencialmente manipulados como parte de ataques *man-in-the-middle*. Como consecuencia, es necesario prestar más atención a la protección de los datos en tránsito y asegurarse de que los endpoints de los microservicios están protegidos para que sólo las partes autorizadas puedan hacer uso de ellos.

Más adelante se estudiarán los mecanismos más utilizados para garantizar la seguridad de estos accesos en el capítulo sobre el protocolo OAuth 2.0.

2.1.5.6. Testing

Apostar por una arquitectura de microservicios también tiene una repercusión importante a la hora de lidiar con pruebas funcionales automatizadas. Cuanto mayor sea el alcance de la prueba, más confianza tendrá la aplicación. Por otro lado, cuanto mayor sea el alcance de la prueba, más difícil será configurar los datos de la prueba y los accesorios de apoyo, más tiempo puede tardar en ejecutarse la prueba y más difícil puede ser averiguar qué está roto cuando falla.

Las pruebas de extremo a extremo para cualquier tipo de sistema están en el extremo de la escala en términos de la funcionalidad que cubren, y los desarrolladores están acostumbrados a que sean más problemáticas de escribir y mantener que las pruebas unitarias de menor alcance. Sin embargo, a menudo esta complejidad merece la pena porque es preferible la confianza que se deriva de que una prueba de extremo a extremo utilice los sistemas de la misma manera que lo haría un usuario.

Con una arquitectura de microservicios, el alcance de las pruebas de extremo a extremo se vuelve demasiado grande. Ahora es necesario ejecutar pruebas en múltiples procesos, todos los cuales deben desplegarse y configurarse adecuadamente para los escenarios de prueba. También es necesario estar preparado para los falsos negativos que se producen cuando los problemas del entorno, como las caídas de las instancias de servicio o los tiempos de espera de la red de los despliegues fallidos, hacen que las pruebas fallen.

A medida que la arquitectura de microservicios crezca, esta situación empeorará y se obtendrá un retorno de la inversión decreciente cuando se trate de pruebas de extremo a extremo. Las pruebas costarán más, pero no conseguirán darle el mismo nivel de confianza que en el pasado.

Esto impulsa nuevas formas de pruebas, como las pruebas basadas en contratos o las pruebas en producción, así como la exploración de técnicas de entrega progresiva, como las ejecuciones paralelas o los despliegues *canary*.

2.1.5.7. Latencia

Con una arquitectura de microservicios, el procesamiento que antes podía hacerse localmente en un procesador puede dividirse en varios microservicios separados. La información que antes fluía dentro de un solo proceso ahora tiene que ser serializada, transmitida y deserializada a través de la red, lo que puede resultar en un empeoramiento de la latencia del sistema.

Aunque puede ser difícil medir el impacto exacto en la latencia de las operaciones en la fase de diseño o codificación, esta es otra razón por la que es importante emprender cualquier migración de microservicios de forma incremental. Hacer un pequeño cambio y luego medir el impacto.

Esto supone que se dispone de alguna forma de medir la latencia de extremo a extremo de las operaciones que le interesan; las herramientas de seguimiento distribuido como Jaeger pueden ayudar en este caso. Pero también hay que saber qué latencia es aceptable para estas operaciones. A veces, hacer que una operación sea más lenta es perfectamente aceptable, siempre que siga siendo lo suficientemente rápida como para que el sistema en su conjunto funcione correctamente. De ahí la importancia a la hora de definir el grado de tolerancia aceptado.

2.1.5.8. Consistencia de los datos

Pasar de un sistema monolítico, en el que los datos se almacenan y gestionan en una única base de datos, a un sistema mucho más distribuido, en el que múltiples procesos gestionan el estado en diferentes bases de datos, provoca la aparición de nuevos retos en torno a la gestión de consistencia de la información que maneja la aplicación.

Mientras que en el pasado se podría haber confiado en las transacciones de la base de datos para gestionar los cambios de estado, en la mayoría de los casos, el uso de transacciones distribuidas resulta muy problemático para coordinar los cambios de estado.

En su lugar, es posible que haya que empezar a utilizar conceptos como sagas y consistencia eventual para gestionar y razonar sobre el estado del sistema. Estas ideas pueden requerir cambios fundamentales en la forma de pensar sobre los datos en los sistemas. Una vez más, aparece una buena razón para ser cauteloso en la rapidez con la que se descompone la aplicación. Adoptar un enfoque incremental para la descomposición, de modo que sea posible evaluar el impacto de los cambios en la arquitectura en producción, es realmente importante.

2.2. El protocolo OAuth 2.0

2.2.1. Introducción

A la hora de diseñar sistemas basados en microservicios, al igual que ocurre en cualquier sistema distribuido, la seguridad se convierte en una preocupación importante que hay que tener en cuenta desde las primeras fases del proceso de diseño.

Los microservicios sufren las mismas vulnerabilidades de seguridad que las arquitecturas orientadas a servicios (SOA). Como los microservicios utilizan mecanismos REST y XML con JSON como principales formatos de intercambio de datos, se debe prestar especial atención a la seguridad de los datos que se transfieren. Los microservicios promueven la reutilización de los servicios, por lo que es natural suponer que algunos sistemas incluirán servicios de terceros.

Por lo tanto, un reto adicional es proporcionar mecanismos de autenticación y autorización con servicios de terceros, ámbito en el cual cobran protagonismo protocolos como OAuth2 y OpenID Connect.

Antes de pasar al estudio detallado de estos protocolos, es importante definir algunos términos. Hay dos términos en particular que son fundamentales para la comprensión de OAuth 2.0 y su ámbito de aplicación: autenticación y autorización.

Estos términos se confunden a menudo y a veces se intercambian, pero en realidad representan dos conceptos distintos en el mundo de la gestión de identidad y acceso, y es importante entender su distinción antes de profundizar en el protocolo OAuth 2.0.

La **autenticación** es el acto de validar que los usuarios son quienes dicen ser. Es el primer paso de cualquier proceso de seguridad.

Un proceso de autenticación suele contar con mecanismos como:

- Contraseñas. Los nombres de usuario y las contraseñas son los factores de autenticación más comunes. Si un usuario introduce los datos correctos, el sistema asume que la identidad es válida y le concede el acceso.
- Claves de acceso únicas. Conceden acceso para una sola sesión o transacción.
- Aplicaciones de autenticación. Generan códigos de seguridad a través de un tercero que concede el acceso.
- Biometría. Un usuario presenta una huella dactilar o un escáner ocular para acceder al sistema.

En algunos casos, los sistemas requieren la verificación exitosa de más de un factor antes de conceder el acceso. Este requisito de autenticación multifactorial (MFA) se implementa a menudo para aumentar la seguridad más allá de lo que pueden proporcionar los anteriores mecanismos por sí solos.

La **autorización** en la seguridad del sistema es el proceso de dar permiso al usuario para acceder a un recurso o función específica. Este término se utiliza a menudo de forma intercambiable con el de control de acceso o privilegio del cliente.

Dar a alguien permiso para descargar un archivo concreto en un servidor o proporcionar a los usuarios individuales acceso administrativo a una aplicación son buenos ejemplos de autorización.

En los entornos seguros, la autorización debe seguir siempre a la autenticación. Los usuarios deben probar primero que sus identidades son genuinas antes de que los administradores de una organización les concedan acceso a los recursos solicitados.

Ahora que se ha establecido claramente la distinción entre estos dos conceptos, se puede abordar el concepto de OAuth y el propósito para el que fue diseñado. Para empezar, OAuth no es una API o un servicio. Es un estándar abierto para la autorización. En concreto, OAuth es un estándar que las aplicaciones pueden utilizar para proporcionar a otras aplicaciones cliente un acceso delegado seguro.

OAuth cuenta con dos versiones: OAuth 1.0a y OAuth 2.0. Estas especificaciones son completamente diferentes entre sí, y no pueden utilizarse juntas, ya que no hay compatibilidad hacia atrás entre ellas.

Hoy en día, OAuth 2.0 es la forma más utilizada de OAuth y en la literatura relacionada con la autorización es común el empleo del término OAuth para referirse directamente a esta versión.

OAuth se creó como respuesta al patrón de autenticación directa. Este patrón se hizo famoso gracias a la autenticación básica de HTTP que utilizaban muchos sitios web, los cuales pedían al usuario que introdujese su nombre de usuario y contraseña directamente en un formulario y accedían a sus datos en su nombre.

2.2.2. Componentes principales de OAuth

En este apartado se describen algunos de los términos utilizados en la especificación del protocolo Oauth y que resultan claves para entender su funcionamiento.

2.2.2.1. Roles

En primer lugar, OAuth 2.0 define cuatro roles involucrados en una solicitud de autorización:

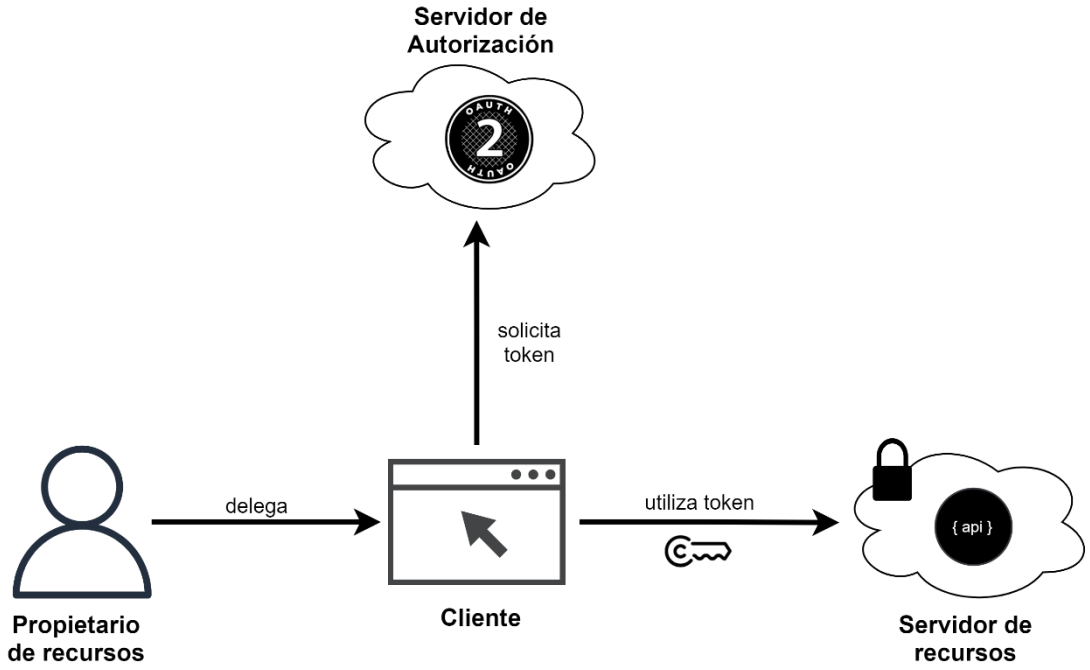

Figura 10. Roles en el proceso de autorización.

- **Servidor de recursos**: Servicio con una API que almacena recursos protegidos, propiedad de los usuarios, a los que puede acceder una aplicación. Los recursos pueden ser datos de los usuarios o sus acciones autorizadas.
- **El usuario o Propietario de recursos**: Usuario u otra entidad que posee los recursos protegidos expuestos en el servidor de recursos.
- **Servidor de Autorización**: Servicio en el que confía el servidor de recursos para autorizar a las aplicaciones a llamar al servidor de recursos y acceder a dichos recursos. Cuando el servidor de autorización se asegura de que un cliente está autorizado a acceder a un recurso en nombre del usuario, emite un token. El cliente utiliza este token para demostrar al servidor de recursos que ha sido autorizado por el servidor de autorización. Es entonces cuando el servidor de recursos permite al cliente acceder al recurso que ha solicitado (si tiene un token válido).
- **Cliente**: Aplicación que accede a los recursos en el servidor de recursos, en nombre del propietario de los recursos. El cliente utiliza un ID de cliente y un secreto de cliente

para identificarse al servidor de autorización. Es importante no confundir estas credenciales con las del usuario.

2.2.2.2. Clientes

Oauth 2.0 define dos tipos de clientes basándose en su capacidad para comunicarse con el servidor de autorización de forma segura. Cada uno de estos tipos de clientes es capaz de enviar peticiones HTTP al servidor de autorización, pero para que esta comunicación sea efectiva, el cliente debe haberse registrado previamente en el servidor de autorización por medio de un identificador de cliente (*client ID*) y un código secreto (*client secret key*). Estas credenciales son necesarias para que el servidor de autorización pueda identificar de forma segura a la aplicación cliente.

El problema se presenta cuando, por la naturaleza del cliente, este no es capaz de almacenar de forma segura su clave secreta, lo que lleva a diferenciar a los clientes en los siguientes dos grupos:

- **Clientes confidenciales**: Aplicaciones que se ejecutan en un servidor protegido y que pueden almacenar de forma segura las claves secretas confidenciales para autenticarse ante un servidor de autorización o utilizar otro mecanismo de autenticación seguro para ese fin. De este modo, estas aplicaciones son capaces de registrarse en el servidor de autorización, obtener su clave secreta y autenticarse de forma segura en él en las peticiones que vengan en el futuro.
- **Clientes públicos**: Aplicaciones que no pueden garantizar la seguridad y confidencialidad de su identificador de cliente ni de claves secretas y, por tanto, no pueden almacenar ningún secreto o utilizar otros medios para autenticarse ante un servidor de autorización. Es el caso de las aplicaciones web ejecutadas en navegadores web como las SPA, las aplicaciones móviles y dispositivos IoT. Siempre que exista la posibilidad de que una aplicación sea descompilada o su código fuente pueda ser visualizado, se estará ante cliente público.

En vista de los diferentes tipos de aplicaciones cliente que se pueden encontrar, OAuth proporciona diferentes flujos de autorización que se adaptan a los diferentes escenarios de comunicación que pueden darse en función del tipo de cliente que esté intentando autenticarse en el servidor de autorización.

2.2.2.3. Tokens

En los apartados anteriores se ha mencionado que el servidor de autorización tiene la responsabilidad de comprobar si un cliente está autorizado por el usuario para acceder a sus recursos y, en tal situación, generar un token que permita el acceso a los recursos protegidos del servidor de recursos.

El **token de acceso** (access token) de OAuth, al que a veces se referencia directamente como token, es un artefacto emitido por el servidor de autorización para una aplicación cliente que

contiene información sobre los derechos delegados por el usuario al cliente. OAuth no define un formato o contenido en sí para el token. Para una aplicación web, el token suele ser una cadena de texto que normalmente es enviada por los clientes por medio de las cabeceras HTTP cuando estos quieren acceder a un *endpoint* del servidor de recursos en particular. Esta cadena de texto puede ser algo tan sencillo como un UUID o puede tener un formato más complejo como el de los JSON Web Token (JWT). En cualquier caso, el token siempre va a representar una combinación de información en la que figurará el acceso requerido por el cliente, el propietario del recurso que autorizó al cliente y los permisos conferidos durante ese proceso de autorización.

Los tokens de acceso son opacos para el cliente, es decir, el cliente no tiene la necesidad de mirar la información que contiene el token. Su único cometido es transportar el token desde que lo recibe del servidor de autorización hasta que lo presenta al servidor de recursos. Únicamente el servidor de autorización y el servidor de recursos tienen la obligación de analizar el contenido del token, ya sea durante su emisión (en el caso del servidor de autorización) o durante su validación (en el caso del servidor de recursos). Este tipo de token está pensado para un uso de corta duración, del orden de horas o minutos.

El **token de actualización** (*refresh token*) de OAuth es un artefacto similar al concepto de *access token* en el sentido de que es emitido por el servidor de autorización y el cliente no tiene ninguna obligación más allá de transportarlo. La diferencia de este token respecto al Access token radica en que el *refresh token* no es enviado en ningún momento al servidor de recursos. En su lugar, el cliente lo utiliza para solicitar al servidor de autorización un nuevo *access token* una vez que la validez del *access token* actual haya sido revocada. Este hecho puede suceder por varios motivos, entre ellos, que el usuario haya revocado el acceso de forma directa desde la aplicación, que el token haya expirado debido a que haya transcurrido una cantidad de tiempo fijada o por que algún mecanismo de seguridad haya provocado esta acción de revocación. Es por eso que el *refresh token* está diseñado para un uso de larga duración, del orden de días, meses o incluso años.

Además de estos dos tipos de tokens, OAuth también define un **código de autorización** (que se verá en mayor detalle más adelante en el apartado de *Authorization Code Grant type*), el cual se utiliza como código intermediario y opaco de un solo uso que es empleado como moneda de cambio por un *access token* y, opcionalmente, un *refresh token*.

Todos los tokens analizados en este apartado tienen en común que se obtienen desde el servidor de autorización. Eso sí, cada uno se consigue a través de diferentes *endpoints* que todo servidor de autorización tiene que implementar. Los dos endpoints principales para este proceso son el ***authorization endpoint*** y el ***token endpoint***. El primero se utiliza para obtener el consentimiento y la autorización del usuario en forma de *authorization code*. El segundo, procesa la concesión obtenida en el *authorization endpoint* para obtener un access token y un refresh token.

Otros *endpoints* interesantes son el **introspection endpoint**, utilizado como mecanismo para obtener información del access token (su validez, su propietario, *scopes* asociados…) y el **revocation endpoint**, utiliza para revocar la validez de los tokens.

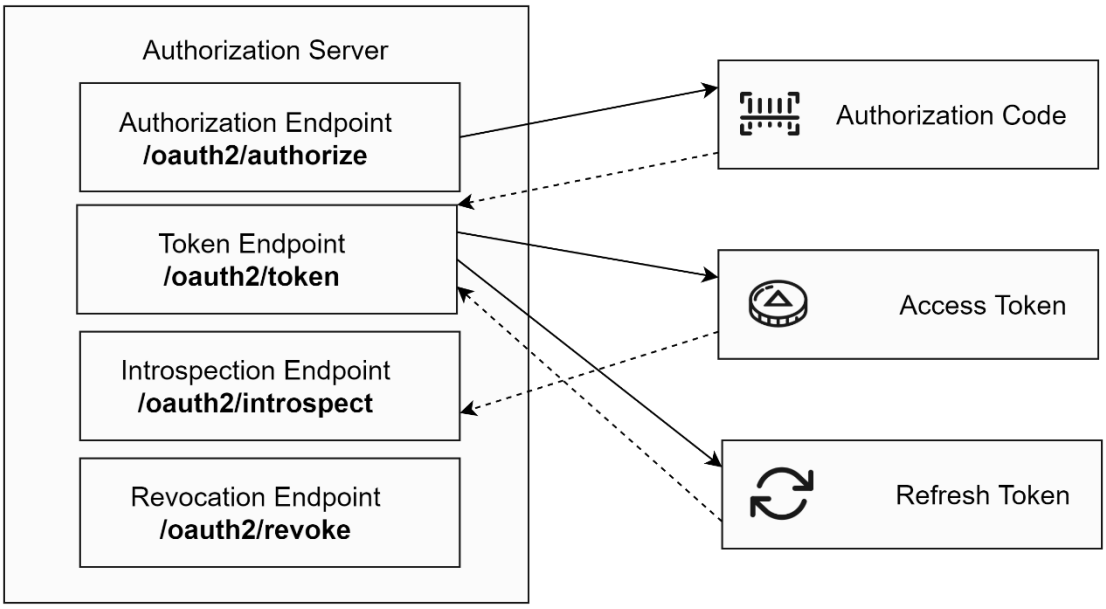

Figura 11. Endpoints del servidor de autorización.

2.2.2.3.1. JSON Web Tokens

En esta sección se analiza una de las formas de implementación de tokens más utilizada en las aplicaciones de hoy en día, los JSON Web Tokens (JWT).

Un JWT es un contenedor que transporta diferentes tipos de aserciones o afirmaciones de un lugar a otro de una forma criptográficamente segura. Una aserción es una declaración sólida sobre alguien o algo emitida por una entidad de confianza. Esta entidad también se conoce como el emisor de la aserción.

Además de aserciones sobre atributos, un JWT puede transportar aserciones de autenticación y autorización. De hecho, un JWT puede interpretarse como un contenedor en el que el usuario puede volcar toda la información que necesite. De este modo una aserción de autenticación podría ser el *username* de un usuario y cómo el emisor autenticó al usuario antes de emitir la aserción, mientras que una aserción de autorización contendrá información sobre los derechos del usuario, o lo que el usuario puede hacer. De este modo, basándose en las afirmaciones que el JWT trae del emisor, el receptor puede decidir cómo actuar.

Al margen de transportar un conjunto de aserciones sobre el usuario, un JWT desempeña otro papel entre bastidores, ya que también contiene información sobre la identidad del emisor.

Se puede extraer mucha información sobre los JWT a partir de su propio nombre:

- JSON: Utiliza JavaScript Object Notation como formato para representar la información que contiene.
- Web: está diseñado para ser utilizado en peticiones web.
- Token: es una implementación de un token.

En la figura se muestra un ejemplo de la forma que presenta un JWT desde el sitio web de jwt.io, que cuenta con una herramienta para depurar y analizar el contenido de este tipo de token.

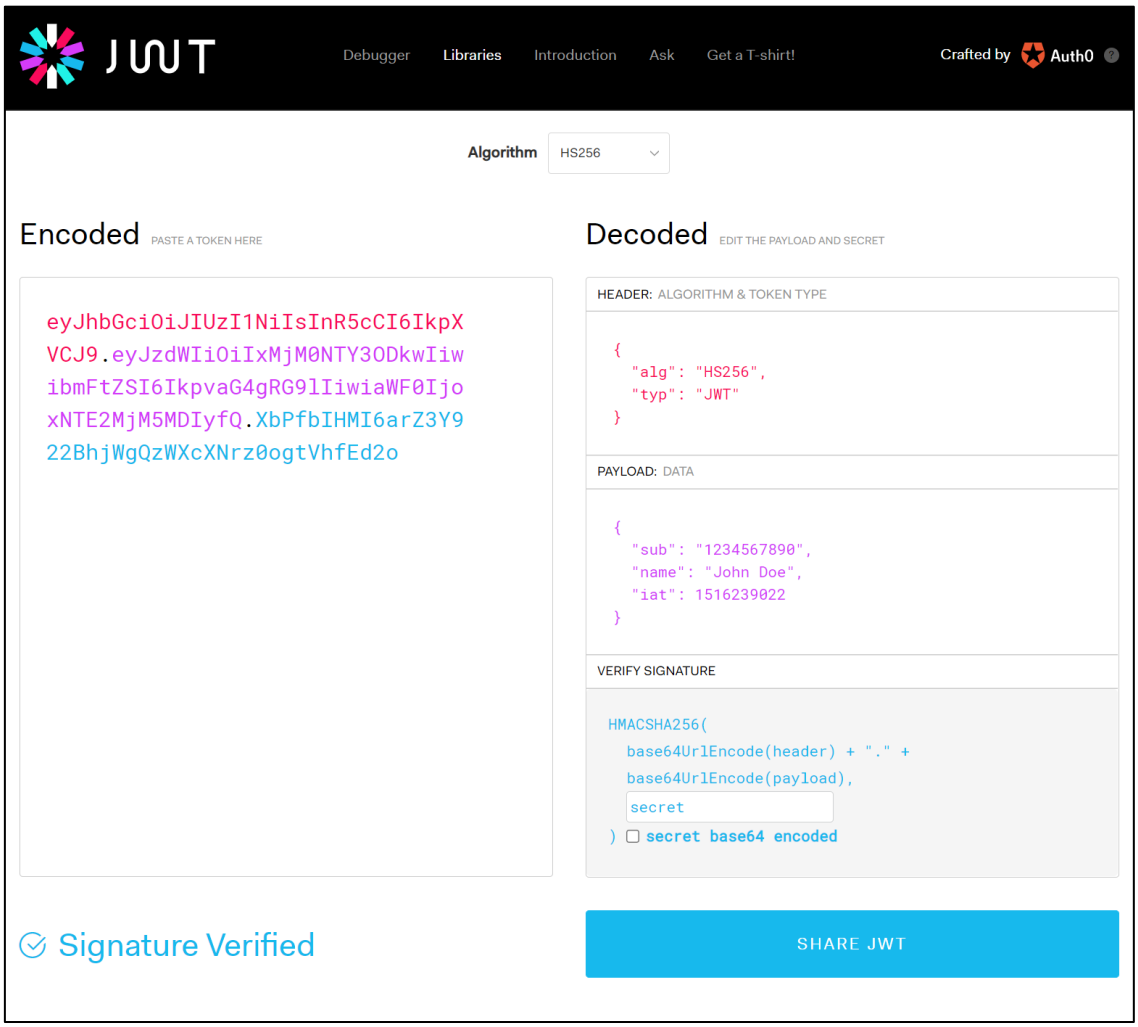

Figura 12. JSON Web Token.

Analizando la figura, se puede ver un JSON Web Signature (JWS), que es el formato más extendido en cuanto a uso de los JWT. En ella, podemos diferenciar tres partes con un punto (.) utilizado como carácter delimitador:

- La primera parte se conoce como cabecera JOSE (JSON Object Signing and Encryption). Esta cabecera es un objeto JSON codificado en Base64Url que contiene metadatos relacionados con el JWT, como el algoritmo de cifrado empleado para firmar el mensaje.
- La segunda parte es el cuerpo del token, carga útil o *claims set*. Esta parte es otro objeto JSON codificado en Base64Url que contiene una serie de aserciones.

La especificación JWT (RFC 7519) define siete atributos opcionales, los *registered claims*. Ahora bien, otras especificaciones que se basan en JWT pueden hacer que alguno de estos atributos sí que sean obligatorios para su cumplimiento. Por ejemplo, la especificación OpenID Connect hace obligatorio el atributo *iss*.

Estos siete atributos que define la especificación JWT están registrados en el registro de reclamaciones de tokens web de la Autoridad de Números Asignados de Internet (IANA). Sin embargo, se deja a voluntad del desarrollador introducir sus propios atributos personalizados en el conjunto de *claims* JWT, siempre que no colisionen en nombre con los *registered claims*.

- La tercera y última parte es la firma. Para crear la parte de la firma hay que tomar la cabecera codificada, la carga útil codificada, un secreto, el algoritmo especificado en la cabecera, y firmarlo. La firma se utiliza para verificar que el mensaje no se ha modificado por el camino y, en el caso de los tokens firmados con una clave privada, también puede verificar que el remitente del JWT es quien dice ser.

Para finalizar este apartado, la tabla de la Figura 14 detalla los *registered claims* definidos en la especificación de JWT en mayor detalle:

Nombre del claim	Descripción del claim
iss	Procede del término *issuer* y representa al emisor del token, es decir, qué autoridad fue la responsable de su creación. En la mayor parte de los casos que se utiliza en un flujo de OAuth será la url del servidor de autorización. Este dato se representa mediante una cadena de texto simple.
sub	Procede del término *subject* y representa al sujeto del token, es decir, sobre quién contiene información el token. En la mayor parte de los casos que se utiliza en un flujo de OAuth será un identificador único sobre el propietario de los recursos. Este dato se representa mediante una cadena de texto simple.
aud	Procede del término *audience* y representa al destinatario del token, es decir, quién se supone que va a aceptar el token. En la mayor parte de los casos que se utiliza en un flujo de OAuth será la url del servidor de recursos. Este dato se representa mediante una cadena de texto simple o mediante un array de cadenas de texto.
exp	Procede del término *expiration* y representa la fecha de caducidad del token. Este dato se representa mediante un número entero del número de segundos desde la época UNIX, la medianoche del 1 de enero de 1970, en la zona horaria GMT.

nbf	Procede de la expresión *not before*. Se trata de un indicador que indica cuándo empezará a ser válido el token, para los despliegues en los que el token podría ser emitido antes de ser válido. Este dato se representa mediante un número entero del número de segundos desde la época UNIX, la medianoche del 1 de enero de 1970, en la zona horaria GMT.
iat	Procede de la expresión *issued at*. Es un indicador que representa la marca de tiempo en el momento de la creación del token. Este dato se representa mediante un número entero del número de segundos desde la época UNIX, la medianoche del 1 de enero de 1970, en la zona horaria GMT.
jti	Procede de la expresión *jwt id* y representa un identificador único para el token. Este valor es único para cada token creado por el *issuer,* y normalmente tiene un valor criptográficamente aleatorio.

Figura 13. Tabla explicativa de los Registered claims.

2.2.2.4. Canales

Para entender mejor el funcionamiento de OAuth es necesario comprender que la comunicación entre el cliente con el servidor de autorización ocurre por medio de dos canales de comunicación diferentes: el *front channel* (canal frontal) y el *back channel* (canal trasero).

El *front channel* es una comunicación indirecta entre el cliente y el authorization endpoint del servidor de autorización a través del agente de usuario del navegador web y utilizando redirecciones HTTP. El hecho de utilizar el protocolo HTTP y de basarse en redirecciones impone una serie de restricciones y limitaciones sobre las propias características de HTTP que se pueden utilizar, tales como el uso obligatorio del método GET y el envío de la información en la propia URI. El mayor inconveniente de este canal es que las solicitudes y respuestas son susceptibles de ser manipuladas por usuarios o software malicioso, así que es común el uso de parámetros especiales tanto en las peticiones como en las respuestas como se define en el RFC 6819 (OAuth 2.0 Threat Model and Security Considerations).

El *back channel* es una comunicación directa entre el cliente y el token endpoint del servidor de autorización que se vio en el apartado anterior. Al establecer una conexión directa con el servidor de autorización, el protocolo no está limitado al uso de redirecciones HTTP, lo que lo hace un medio de comunicación más seguro.

Hay que aclarar que, si bien estos términos son ampliamente utilizados en la descripción de OAuth por su utilidad a la hora de explicar la forma de comunicación entre cliente y servidor de autorización, son términos que no aparecen definidos en el RFC de OAuth 2.0, sino que están tomados del glosario de SAML.

2.2.2.5. Scopes

El **scope** (o ámbito de alcance) define una forma de limitar el acceso de una aplicación a los datos de un usuario. En lugar de conceder acceso completo a la cuenta de un usuario, a menudo es más útil dar a las aplicaciones una forma de solicitar un alcance más limitado de lo que se les permite hacer en nombre del usuario.

Algunas aplicaciones sólo utilizan OAuth para identificar al usuario, por lo que sólo necesitan acceder a un ID de usuario y a la información básica del perfil. Otras aplicaciones pueden necesitar conocer información más sensible, como la fecha de nacimiento del usuario, o pueden necesitar la capacidad de publicar contenido en nombre del usuario, o modificar los datos del perfil.

Los usuarios estarán más dispuestos a autorizar una aplicación si saben exactamente lo que la aplicación puede y no puede hacer con su cuenta. El alcance es una forma de controlar ese acceso y ayudar al usuario a identificar los permisos que está concediendo a la aplicación.

2.2.2.6. Concesiones de autorización

La especificación de OAuth 2.0 define cuatro métodos por los que las aplicaciones cliente obtienen autorización para llamar a una API. Cada uno de estos métodos utiliza un tipo diferente de credencial para representar el proceso de autorización, lo que se conoce comúnmente como *authorization grants* o concesiones de autorización.

Como hay diferentes tipos de aplicaciones cliente y diferentes casos de uso para un mismo tipo de cliente, OAuth define varios flujos con el objetivo de cubrir la mayor cantidad de escenarios posibles, además de ofrecer un framework para crear nuevos tipos de concesiones de autorización.

La figura 14 muestra algunos de los tipos de concesiones de autorización más utilizados y ejemplos de aplicaciones para los que son convenientes.

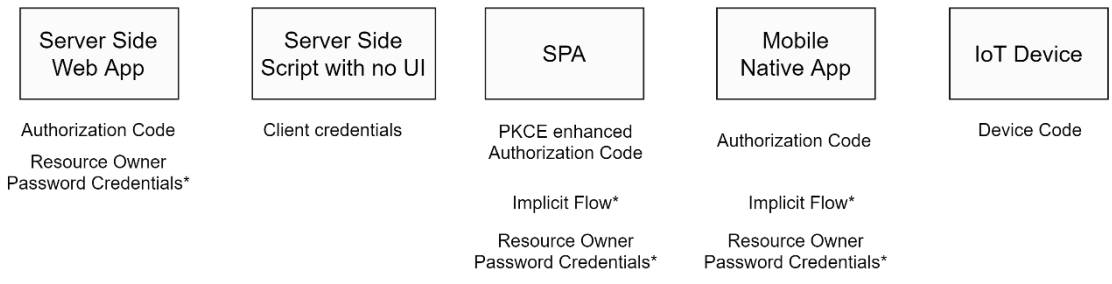

Figura 14. Concesiones de autorización y escenarios de uso.

**Implicit Flow* y *Password Grant* son dos tipos de concesiones de autorización que actualmente se conservan como implementaciones de legado y su uso está desaconsejado.

A continuación, se detalla en que consiste cada uno de los diferentes tipos enumerados.

2.2.2.6.1. Authorization Code

La figura 15 sirve para ilustrar el flujo que sigue este tipo de concesión:

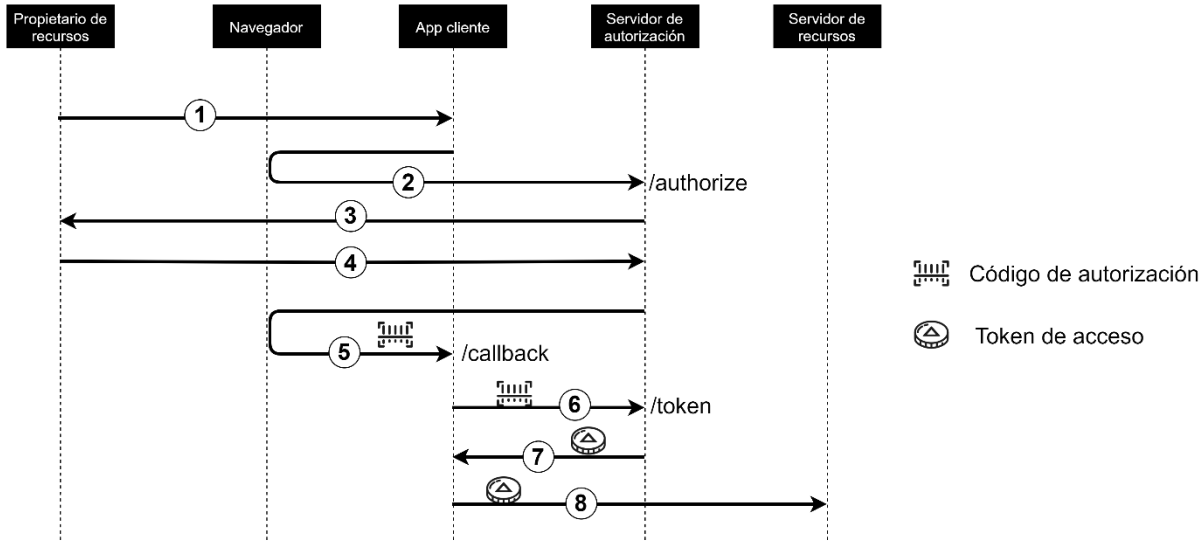

Figura 15. Tipo de concesión Authorization Code.

1. El usuario accede a la aplicación cliente.

2. La aplicación redirige al navegador al authorization endpoint del servidor de autorización con una solicitud de autorización.

3. El servidor de autorización solicita al usuario la autenticación y le muestra una ventana de consentimiento informándole de los permisos que va a otorgar a la aplicación cliente.

4. El usuario completa la autenticación y da su consentimiento.

5. El servidor de autorización redirige el navegador de vuelta a la aplicación con un código de autorización. El lugar de la aplicación al que se hace la redirección viene definido por la url de callback.

6. La aplicación hace una petición al token endpoint del servidor de autorización, adjuntando en ella el código de autorización que acaba de recibir.

7. El servidor de autorización responde con un *access token* (y opcionalmente un *refresh token*).

8. La aplicación se comunica con la API del servidor de recursos portando el *access token*.

El tipo de concesión Authorization Code se diseñó originalmente para los clientes confidenciales. La primera solicitud (de autorización) redirige al usuario al servidor de autorización para que pueda interactuar con el usuario y la segunda solicitud podría ser realizada por el backend de la aplicación directamente al *token endpoint* del servidor de autorización, lo que

permite que el backend de una aplicación, que se supone que es capaz de gestionar de forma segura un secreto de autenticación, se autentique ante el servidor de autorización al intercambiar el código de autorización por el token de acceso. También significa que la respuesta con el token de acceso puede ser entregada al backend de la aplicación, que hará las subsiguientes llamadas a la API.

Una ventaja añadida es que los tokens se devuelven a través de una respuesta segura del backchannel. Sin embargo, aunque originalmente se optimizó para clientes confidenciales, la adición de PKCE permite a los clientes públicos, como las aplicaciones web SPA o las aplicaciones móviles, utilizar también este tipo de concesión.

La figura 16 muestra un ejemplo del formato que tiene la petición de autorización, que se corresponde con el paso 2 del flujo descrito anteriormente.

```
GET /authorize?
response_type=code
& client_id=<client_id>
& state=<state>
& scope=<scope>
& redirect_uri=<callback uri>
& resource=<API identifier>
& code_challenge=<PKCE code_challenge>
& code_challenge_method=S256 HTTP/1.1
```

Figura 16. Petición de autorización en Authorization Code.

Los parámetros de la petición son:

- **response_type**: indica el tipo de flujo de concesión de autorización de Oauth 2.0. En este caso, el valor "code" determina la utilización del tipo de concesión Authorization Code.
- **client_id**: identificador de la aplicación cliente, asignado cuando ésta se registró en el servidor de autorización.
- **state**: cadena difícil de adivinar, única para llamada, opaca para el servidor de autorización y utilizada por el cliente para rastrear el estado entre una solicitud y su correspondiente respuesta para mitigar el riesgo de ataques CSRF. Debe contener un valor que asocie la solicitud con la sesión del usuario.
- **scope**: alcance de los privilegios de acceso para los que se solicita autorización.
- **redirect_uri**: URL a la que el servidor de autorización debe enviar su respuesta con el código de autorización.
- **resource**: también llamado *audience*, es un identificador de una API específica registrada en el servidor de autorización para la que se solicita el token de acceso. Se utiliza principalmente en implementaciones con APIs personalizadas. Es un parámetro opcional que originalmente no figuraba en la especificación de OAuth 2.0.

Los otros dos parámetros restantes, code_challenge y code_challenge_method son exclusivos de la extensión PKCE, que se verá más adelante.

La figura 17 muestra un ejemplo del formato que tiene la petición que se realiza para conseguir el *access token,* que se corresponde con el paso 6 del flujo descrito anteriormente.

```
POST /token HTTP/1.1
Host: authorizationserver.com
Content-Type: application/x-www-form-urlencoded
grant_type=authorization_code
& code=<authorization_code>
& client_id=<client id>
& code_verifier=<code verifier>
& redirect_uri=<callback URI>
```

Figura 17. Petición de token en Authorization Code.

Los parámetros que se encuentran en esta petición son:

- grant_type: debe ser authorization_code para el tipo de concesión Authorization Code.
- code: el código de autorización recibido en la respuesta de la petición de autorización.
- client_id: identificador de la aplicación cliente, asignado cuando ésta se registró en el servidor de autorización.
- redirect_uri: URL a la que el servidor de autorización debe enviar su respuesta. Debe coincidir con el parámetro redirect_uri utilizado en la petición de autorización.

El parámetro *code_verifier* se verá más adelante en la sección de PKCE.

2.2.2.6.2. Authorization Code + PKCE

PKCE (*Proof Key for Code Exchange*) es una extensión del tipo de concesión Authorization Code diseñada con el objetivo de prevenir ataques CSRF y ataques de inyección durante el proceso de autorización. Su especificación se puede encontrar en el RFC 7636.

Originalmente, su objetivo era dotar de un mecanismo de seguridad a este flujo cuando se usaba en aplicaciones móviles, pero su habilidad para prevenir los ataques mencionados anteriormente lo convierten en una herramienta muy útil para cualquier tipo de cliente, incluso aplicaciones web consideradas como clientes confidenciales. Eso sí, hay que tener en cuenta que, pese a esta recomendación, PKCE no es, en ningún caso, un reemplazo de las credenciales secretas utilizadas en la autenticación del cliente. Por lo tanto, no convierte a un cliente público en confidencial por el mero hecho de utilizarlo.

La figura 18 sirve para mostrar el flujo que se sigue en el tipo de concesión Authorization Code mejorado con PKCE:

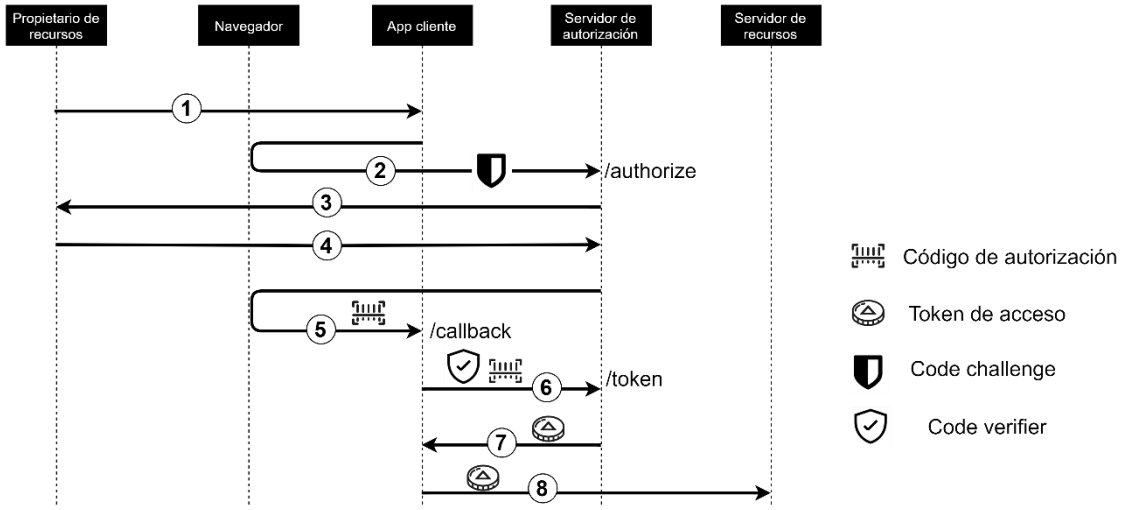

Figura 18. Tipo de concesión Authorization Code con PKCE.

El objetivo principal de PKCE es garantizar la aplicación que solicitó un código de autorización es la misma que luego utiliza este código para obtener el access token. El riesgo que se encuentra en los clientes públicos es que un usuario o software malicioso podría interceptar el código de autorización y utilizarlo para obtener un token de acceso suplantando al usuario.

Para utilizar PKCE, la aplicación crea una cadena criptográfica aleatoria de un tamaño considerable, llamada verificador de código (*code verifier*). A continuación, la aplicación calcula un valor derivado, llamado desafío de código (*code challenge)*, a partir del *code verifier*. Cuando la aplicación envía una solicitud de autorización en el paso 2 de la figura 18, incluye el *code challenge*, junto con el método utilizado para derivarlo, el *code challenge method*.

Cuando la aplicación envía el código de autorización al *token endpoint* del servidor de autorización para obtener el token de acceso en el paso 6, incluye el *code verifier*. El servidor de autorización transforma el valor del *code verifier* utilizando el método de transformación recibido en la solicitud de autorización y comprueba que el resultado coincide con el *code challenge* enviado con la solicitud de autorización. Esto permite al servidor de autorización detectar si una aplicación maliciosa está intentando utilizar un código de autorización robado ya que sólo la aplicación legítima conocerá el *code verifier* que debe pasar en el paso 6 y que coincidirá con el *code challenge* enviado en el paso 2.

La especificación PKCE enumera dos métodos de transformación que pueden utilizarse para derivar el desafío de código del *code verifier*, a saber, "simple" y "S256". Con el método "simple", el *code challenge* y el *code verifier* son idénticos, por lo que no hay ningún mecanismo de seguridad en el caso de que el *code challenge* se vea comprometido. Por lo tanto, es recomendable que las aplicaciones que utilizan la concesión Authorization Code con PKCE utilicen el método de transformación S256, que utiliza un hash SHA256 codificado en base64 del *code verifier* para protegerlo.

2.2.2.6.3. Client Credentials

El tipo de concesión Client Credentials está diseñado para ser utilizado en situaciones en las que una aplicación realiza peticiones a una API para acceder a recursos de los que es dueña, sin que sea necesaria ninguna interacción entre el usuario final y el servidor de autorización.

La figura 19 sirve para ilustrar el flujo que sigue este tipo de concesión:

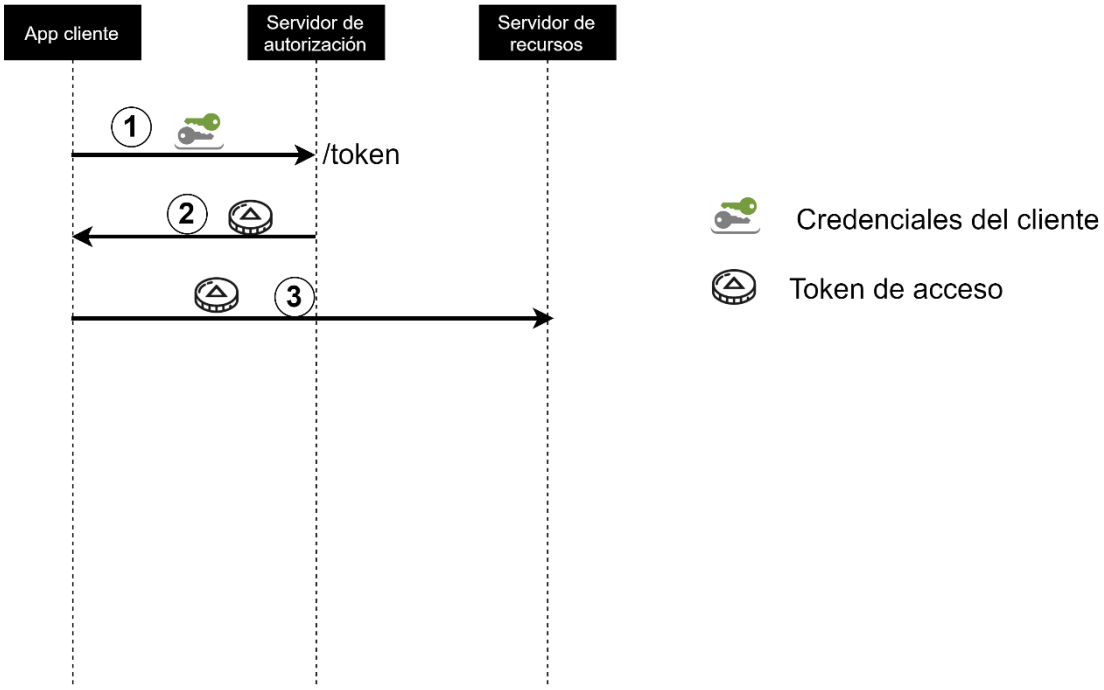

Figura 19. Tipo de concesión Client Credentials.

1. La aplicación envía una petición de autorización al token endpoint del servidor de autorización. Esta petición incluye las credenciales de la aplicación.

2. El servidor de autorización valida las credenciales de la petición y responde con un token de acceso.

3. La aplicación se comunica con la API del servidor de recursos portando el *access token*.

Las credenciales de la aplicación sirven como autorización para la aplicación y se utilizan para solicitar un *access token* desde el *token endpoint* del servidor de autorización. Si el token de acceso caducase, simplemente se volvería a repetir este proceso de comunicación para obtener uno nuevo, sin necesidad de emplear un token de refresco.

La figura 20 muestra un ejemplo del formato que tiene la petición que se realiza para conseguir el *access token,* donde se puede ver como el parámetro grant_type tiene el valor "client_credentials" para establecer el uso de este tipo de concesión.

```
POST /token HTTP/1.1
Host: authorizationserver.com
Authorization: Basic <encoded application credentials>
Content-Type: application/x-www-form-urlencoded
grant_type=client_credentials
& scope=<scope>
& resource=<API identifier>
```

Figura 20. Petición de token en Client Credentials.

2.2.2.6.4. Device Authorization

El tipo de concesión de autorización Device Authorization es una extensión del protocolo OAuth 2.0 que permite a los clientes solicitar la autorización en dispositivos con una capacidad de entrada de texto limitada, o que carecen de un navegador adecuado para llevar a cabo una autorización basada en agente de usuario. Entre estos dispositivos se pueden encontrar Smart TVs, videoconsolas, marcos digitales o impresoras.

En este flujo se indica al usuario que revise la solicitud de autorización en un dispositivo secundario (como un smartphone), en lugar del dispositivo al que se quiere conceder el acceso a los recursos, ya que este dispositivo secundario sí cuenta con las capacidades necesarias para continuar el proceso.

Ahora bien, para que este tipo de concesión de autorización pueda llevarse a cabo deben cumplirse los siguientes requisitos operativos:

- El dispositivo tiene que estar conectado a Internet.
- El dispositivo debe ser capaz de realizar peticiones HTTPS externas.
- El dispositivo debe ser capaz de mostrar o comunicar de otro modo una URI y una secuencia de códigos al usuario.
- El usuario tiene un dispositivo secundario (por ejemplo, un ordenador personal o un smartphone) desde el que puede procesar la petición.

También cabe señalar que en la especificación de este tipo de concesión (RFC 8628) se define un nuevo endpoint que el servidor de autorización debe implementar, el device authorization endpoint. Este nuevo endpoint, tiene la particularidad de que permite a la aplicación cliente interactuar con el servidor de autorización sin la necesidad de usar el agente de usuario.

En este tipo de concesión el flujo principal genera dos escenarios de comunicación simultáneos. El primero de ellos es el siguiente:

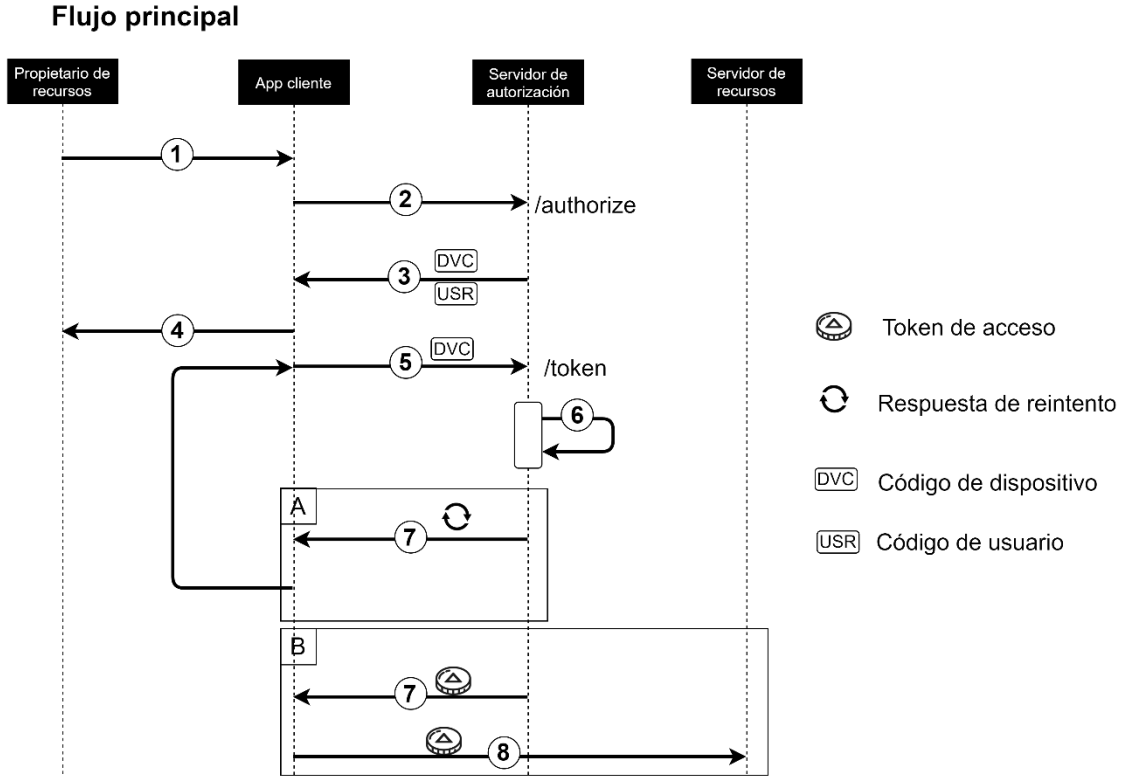

Figura 21. Flujo principal de la concesión Device Authorization Flow.

1. El usuario accede a la aplicación cliente.

2. La aplicación cliente manda una petición al device endpoint del servidor de autorización en la que se incluye su identificador de cliente.

3. El servidor de autorización responde con un código de dispositivo, un código de usuario, una URI de verificación y un tiempo de expiración, que indica la cantidad de segundos antes de que estos códigos dejen de ser válidos.

4. El cliente indica al usuario que utilice un agente de usuario en otro dispositivo para visitar la URI de verificación y le muestra el código de usuario que deberá introducir en ella para continuar con la autorización. Aquí se inicia el flujo secundario.

5. La aplicación cliente comienza a sondear repetidamente al servidor de autorización para saber si el usuario ha completado el proceso de autorización mediante el flujo secundario.

6. El servidor de autorización comprueba si el usuario ha completado el flujo secundario.

7. (A) Si el usuario no ha completado el proceso, el servidor de autorización responde a la aplicación indicando que debe continuar con su sondeo. (B) Si el usuario sí ha completado el proceso, el servidor de autorización devolverá un token de acceso a la aplicación cliente

8. La aplicación se comunica con la API del servidor de recursos portando el *access token*.

El flujo secundario es el siguiente:

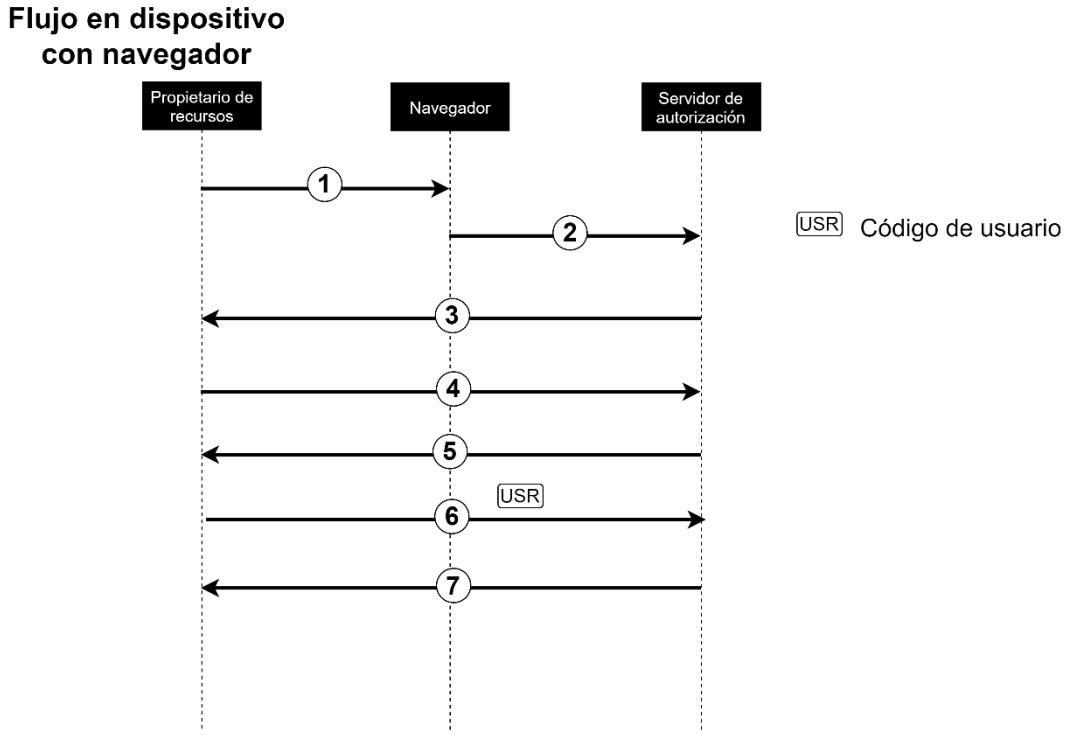

Figura 22. Flujo principal de la concesión Device Authorization Flow.

1. El usuario introduce en el navegador la url que ha recibido en el paso 4 del flujo principal.

2. El agente de usuario del navegador procesa la url y realiza una petición al servidor de autorización.

3. El servidor de autorización solicita al usuario la autenticación y le muestra una ventana de consentimiento informándole de los permisos que va a otorgar a la aplicación cliente.

4. El usuario completa la autenticación y da su consentimiento.

5. El servidor de autorización solicita al usuario su código de usuario.

6. El usuario introduce su código de usuario.

7. El servidor de autorización valida el código de usuario y responde al usuario avisándole de que retorne a su dispositivo principal, donde ya tendrá acceso a los recursos.

La petición de autorización de este tipo de concesión únicamente cambia en cuanto al endpoint de destino, que en lugar de ser el authorization endpoint, es el device authorization endpoint que se ha mencionado anteriormente.

En cambio, en la respuesta sí que se pueden apreciar algunas diferencias, siendo lo más destacable los siguientes parámetros:

- Device_code: parámetro obligatorio que representa el código de verificación del dispositivo.
- User_code: parámetro obligatorio que representa el código de verificación del usuario.
- Verification_uri: parámetro obligatorio que determina la URI de verificación que tendrán que introducir los usuarios en un dispositivo secundario para continuar el proceso de autorización.
- Verification_uri_complete: parámetro opcional que representa una versión alternativa de la uri de verificación en la que se incluye el código de verificación de usuario.
- Expires_in: parámetro obligatorio que establece el tiempo de vida en segundos del device_code y el user_code.
- Interval: parámetro opcional que expresa la mínima cantidad de tiempo expresada en segundos que el cliente debe esperar a la hora de realizar peticiones de sondeo al token endpoint del servidor de autorización. En caso de no proveer ningún valor, los clientes utilizarán 5 segundos como valor por defecto.

La petición de sondeo que realiza el cliente al token endpoint del servidor de autorización se representa con la figura 23:

```
POST /token HTTP/1.1
Host: server.example.com
Content-Type: application/x-www-form-urlencoded

grant_type=urn%3Aietf%3Aparams%3Aoauth%3Agrant-type%3Adevice_code
&device_code=GmRhmhcxhwAzkoEqiMEg_DnyEysNkuNhszIySk9eS
&client_id=1406020730
```

Figura 23. Petición de token en Device Authorization.

En este caso, como principales diferencias respecto a otros tipos de concesión, se puede ver que el grant_type tiene el valor urn:ietf:params:oauth:grant-type:device_code, el cual es necesario para indicar la utilización de este tipo de concesión, y que en la petición se incluye el código de verificación del dispositivo.

En cuanto a la respuesta, es necesario detenerse a estudiar los diferentes escenarios de error ya que son de vital importancia por la particularidad de este flujo en cuanto a las peticiones de sondeo y la forma de obtener el access token, el cual se recibirá como resultado de una respuesta con éxito. Los parámetros más destacados son:

- **Authorization_pending**: indica que la petición de autorización sigue a la espera de que el usuario complete los pasos necesarios, por lo que indica al cliente que debe continuar realizando sus peticiones de sondeo, respetando siempre el número de segundos

indicados en el parámetro "interval" de la respuesta a la petición de autorización de dispositivo junto con cualquier incremento de este valor requerido por el código de error "slow_down".

- **Slow_down**: variación del código de error "authorization_pending" en el que el intervalo de espera debe incrementarse en 5 segundos para las subsiguientes peticiones de sondeo.
- **Access_denied**: indica que la solicitud de autorización fue denegada por el usuario.
- **Expired_token**: indica que el código de dispositivo ha expirado y se ha dado por concluido el proceso de autorización, por lo que el usuario deberá realizar una nueva petición de autorización al device authorization endpoint si quiere volver a iniciar el proceso.

2.2.2.6.5. Implicit Flow

El tipo de concesión Implicit Flow se diseñó para ser utilizado por clientes públicos como las aplicaciones SPA. Es una simplificación del tipo de concesión Authorization Code donde no se emiten credenciales intermedias, sino que el servidor de autorización emite directamente un *access token* como respuesta.

En este flujo, cuando se emite el *access token*, no se produce ningún proceso de autenticación por parte del servidor de autorización al cliente. En algunos casos, la identidad del cliente puede ser verificada a través del URI de redirección utilizada para entregar el token de acceso al cliente.

Fue diseñado en un momento en el que el estándar de CORS aún no estaba adoptado abiertamente por la mayoría de los navegadores, por lo que los clientes sólo podían hacer llamadas al dominio desde el que se cargaba la página. Esto provocaba que las aplicaciones no podían hacer la petición al *token endpoint* del servidor de autorización. Para compensar esto, se reducía el número de mensajes de comunicación necesarios para conseguir el *access token*, que se devolvía directamente en la primera petición, ofreciendo al mismo tiempo una mejora en la capacidad de respuesta y en la eficiencia de la aplicación cliente.

Sin embargo, en la actualidad la mayoría de los navegadores soportan CORS. Como consecuencia, la concesión Implicit Flow ya no es necesaria para su propósito original y, debido a los riesgos que conlleva devolver los access tokens en las redirecciones de HTTP sin que haya ninguna confirmación de que han sido recibidos por el cliente, actualmente su uso está desaconsejado, siendo el tipo de concesión Authorization Code con PKCE la alternativa a utilizar.

La figura 24 sirve para ilustrar el flujo que sigue este tipo de concesión:

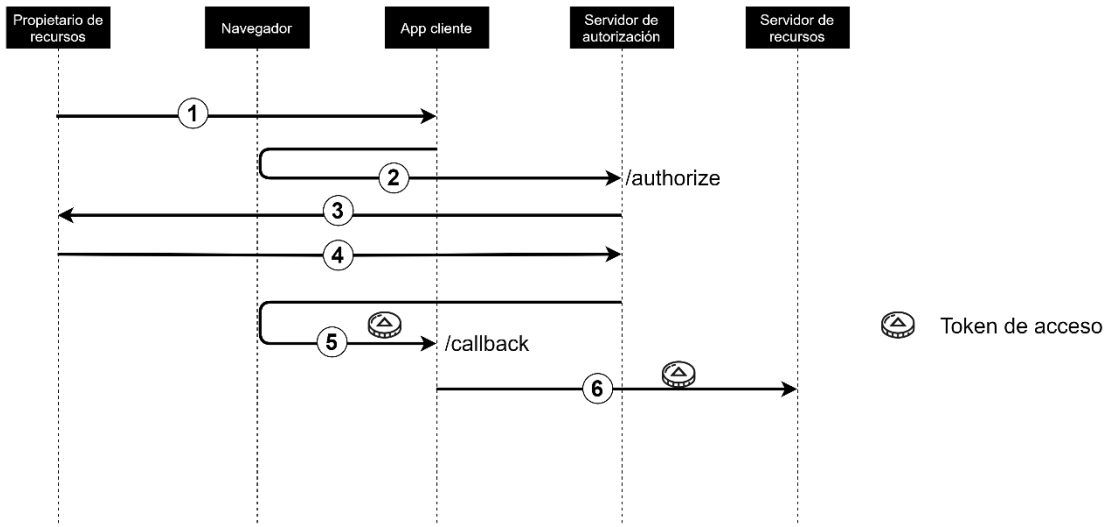

Figura 24. Tipo de concesión Implicit flow.

1. El usuario accede a la aplicación cliente.

2. La aplicación redirige al navegador al authorization endpoint del servidor de autorización con una solicitud de autorización.

3. El servidor de autorización solicita al usuario la autenticación y le muestra una ventana de consentimiento informándole de los permisos que va a otorgar a la aplicación cliente.

4. El usuario completa la autenticación y da su consentimiento.

5. El servidor de autorización redirige el navegador de vuelta a la aplicación con un *access token*. El lugar de la aplicación al que se hace la redirección viene definido por la url de callback.

6. La aplicación se comunica con la API del servidor de recursos portando el *access token*.

La petición de autorización que se emplea en este tipo de concesión tiene el aspecto que muestra la figura 25:

```
GET /authorize?
response_type=token
& response_mode=form_post
& client_id=<client_id>
& scope=<scope>
& redirect_uri=<callback uri>
& resource=<API identifier>
& state=<state> HTTP/1.1
Host: authorizationserver.com
```

Figura 25. Petición de token en Implicit Flow.

Como se puede ver, los parámetros son similares a los utilizados en el tipo de concesión Authorization Code. En el caso del parámetro response_type, ahora se utiliza el valor "token" en lugar de "code" para indicar que se quiere emplear el tipo de concesión Implicit Flow. El otro cambio que se puede percibir es que hay un parámetro llamado response_mode con el valor "form_post". Este valor provoca que la respuesta del servidor de autorización se devuelva codificada en un formulario HTML

2.2.2.6.6. Resource Owner Password Credentials

El tipo de concesión Resource Owner Password Credentials se diseñó para dar soporte a un escenario en el que el servidor de recursos podía confiar en la aplicación cliente para manejar las credenciales del usuario y no se podía utilizar ningún otro tipo de concesión de autorización.

La figura 26 sirve para ilustrar el flujo que sigue este tipo de concesión:

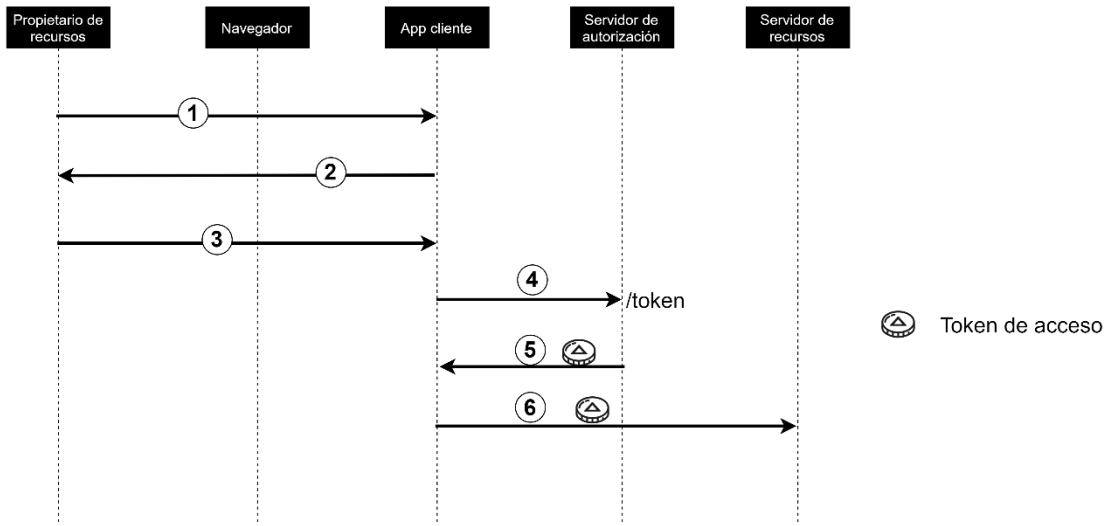

Figura 26. Tipo de concesión Resource Owner Password Credentials.

1. El usuario accede a la aplicación cliente.

2. La aplicación cliente solicita al usuario sus credenciales.

3. El usuario introduce sus credenciales en la aplicación cliente.

4. La aplicación cliente envía una petición al token endpoint del servidor de autorización. Esta petición contiene las credenciales del usuario.

5. El servidor de autorización responde a la aplicación con un *access token* (y opcionalmente un *refresh token*).

6. La aplicación se comunica con la API del servidor de recursos portando el *access token*.

Como se ha podido ver en la explicación del flujo, es la aplicación quien obtiene las credenciales del usuario directamente, en lugar de redirigir al usuario al servidor de autorización y que sea este quien se las solicite.

Precisamente por esta razón, actualmente se desaconseja el uso de este tipo de concesión, ya que expone las credenciales del usuario a la aplicación. Su uso ha quedado relegado a páginas de login embebidas y escenarios en los que es necesario realizar algún tipo de migración de usuarios.

En el pasado también se usó para la comunicación entre aplicaciones móviles y APIs de uso interno, pero actualmente, para este escenario, se recomienda utilizar el tipo de concesión Authorization Code con PKCE.

La petición de autorización que se emplea en este tipo de concesión tiene el aspecto que muestra la figura 26:

```
POST /token HTTP/1.1
Host: authorizationserver.com
Authorization: Basic <encoded application credentials>
Content-Type: application/x-www-form-urlencoded
grant_type=password
& scope=<scope>
& resource=<API identifier>
& username=<username>
& password=<password>
```

Figura 27. Petición de token en Resource Owner Password Credentials.

Como se puede ver, los parámetros son similares a los utilizados en los otros tipos de concesiones. En el caso del parámetro response_type, ahora se utiliza el valor "password" para indicar que se quiere emplear el tipo de concesión Resource Owner Password Credentials. También se envían las credenciales del usuario a través de los parámetros "username" y "password".

La autenticación de la aplicación cliente en el servidor de autorización se hace por medio de la cabecera HTTP de autorización, donde se envían las credenciales de la aplicación cliente obtenidas al registrarse en el servidor de autorización.

2.2.2.6.7. Refresh Token

En secciones anteriores, cuando se describían los diferentes tipos de tokens utilizados por OAuth, se introdujo el concepto de refresh token o token de actualización y se vio que el objetivo de este token era ser utilizado por la aplicación cliente para solicitar al servidor de autorización un nuevo access token, una vez que la validez de este hubiese caducado.

Por muy útil que parezca esta funcionalidad, los refresh tokens no son necesarios en todos los escenarios analizados anteriormente, de ahí que tengan un carácter de emisión opcional. Por ejemplo, en un tipo de concesión Client Credentials, la aplicación puede solicitar un nuevo access token de forma programática en cualquier momento sin necesidad de la intervención del usuario.

Aun así, los refresh tokens proporcionan una buena forma para que las aplicaciones web tradicionales y las aplicaciones nativas obtengan nuevos access tokens, contribuyendo enormemente a que se puedan definir tiempos de expiración cortos para los access tokens (lo que minimiza los riesgos en caso de que sean interceptados por un usuario o software malicioso).

Para muchos desarrolladores, puede ser tentador llevar a cabo esta actualización automática de los access token tan rápido como expiren, pero según las recomendaciones del principio de mínimo privilegio, es mejor actualizar un access token solo cuando sea necesario. Por esta razón, una aplicación debe almacenar los refresh token de forma segura, como si se tratase de una credencial sensible.

La especificación de OAuth 2.0 no incluye ningún mecanismo para las aplicaciones soliciten refresh tokens, dejando su proceso de emisión en manos de los servidores de autorización. De esta forma, hay servidores de autorización que los emiten de forma automática junto con los access tokens, y hay servidores de autorización que esperan a que sea el cliente quien los solicite de forma explícita.

A la hora de solicitar un nuevo access token a un servidor de autorización, se realizaría una llamada al token endpoint del servidor de autorización, pero esta vez utilizando "refresh_token" como valor del parámetro "grant_type", e incluyendo el propio refresh token en un parámetro con el mismo nombre. La figura 28 muestra un ejemplo de esta petición:

```
POST /token HTTP/1.1
Host: authorizationserver.com
Authorization: Basic <encoded application credentials>
Content-Type: application/x-www-form-urlencoded
grant_type=password
& scope=<scope>
& resource=<API identifier>
& username=<username>
& password=<password>
```

Figura 28. Solicitud de Refresh token.

La respuesta que se puede esperar es idéntica a la que se ha detallado en el flujo de concesión Authorization Code.

2.2.3. Diferencias entre OAuth 1 y OAuth 2

OAuth 2.0 es una reescritura completa de OAuth 1.0, compartiendo únicamente los objetivos y la experiencia general del usuario. OAuth 2.0 no es compatible con OAuth 1.0 o 1.1, y debe considerarse como un protocolo completamente nuevo.

Los orígenes de OAuth 1.0 se pueden encontrar en dos protocolos propietarios ya existentes: la API de autorización de Flickr y AuthSub de Google. El proyecto que acabaría convirtiéndose en Oauth 1.0 fue la mejor solución para el momento, pero según pasaban los años y las empresas desarrollaban sus APIs implementando OAuth 1, muchos desarrolladores comenzaron a darse cuenta de que el protocolo suponía un reto para mucha gente. Se identificaron varias áreas específicas que necesitaban mejoras porque limitaban las capacidades de las APIs o eran demasiado difíciles de implementar.

A continuación, se exponen las principales áreas donde puede apreciarse una mayor diferencia entre ambos estándares y las motivaciones que las sustentan.

2.2.3.1. Terminología y roles

Mientras que OAuth 2.0 define cuatro roles (cliente, servidor de autorización, servidor de recursos y propietario de recursos), OAuth 1 utiliza un conjunto diferente de términos para estos roles. Lo equivalente al rol cliente de OAuth 2.0 se conoce como consumidor, el propietario del recurso se conoce simplemente como usuario, y el servidor de recursos es conocido como "proveedor de servicios". OAuth 1 no hace una separación explícita de los roles de servidor de recursos y servidor de autorización.

2.2.3.2. Autenticación y firmas

La mayoría de las implementaciones fallidas de OAuth 1.0 se debieron a los requisitos criptográficos del protocolo. La complejidad de las firmas de Oauth 1.0 era demasiado alta en contraste con la simplicidad de los mecanismos de autenticación mediante usuario y contraseña que se utilizaban en el momento. Ahora, los desarrolladores estaban obligados a buscar, instalar y configurar librerías para hacer peticiones firmadas con claves criptográfica que debían validarse en los endpoints destino.

Con la introducción de los Bearer tokens de OAuth 2.0, todo este proceso se pudo simplificar y volvió a ser posible realizar rápidamente peticiones a APIs mediante simples comandos cURL, utilizando el token de acceso en lugar del nombre de usuario y su contraseña.

Este mecanismo consiguió proporcionar un buen equilibrio entre la facilidad de uso de las API y las buenas prácticas de seguridad.

La desventaja de los *Bearer Token* es que todas las solicitudes deben realizarse a través de una conexión HTTPS, ya que el token contiene texto plano y no hay nada que impida a otras aplicaciones utilizar el mismo token si este es interceptado. Esta es una crítica común a OAuth 2.0.

En circunstancias normales, cuando las aplicaciones protegen adecuadamente los tokens de acceso bajo su control, esto no supone un problema, aunque técnicamente sea menos seguro. Si un servicio requiere un enfoque más seguro, siempre se puede utilizar un tipo de token de acceso diferente que satisfaga los requisitos de seguridad.

2.2.3.3. Experiencia de usuario y opciones de emisión de tokens

El funcionamiento de OAuth 2.0 podría resumirse en dos procesos principales: obtener la autorización para el usuario por medio de un *Access token* y usar dicho token para hacer las peticiones al servidor de recursos en nombre del usuario.

OAuth 1.0 comenzó con tres flujos; uno para aplicaciones basadas en la web, otro para clientes de escritorio y otro para dispositivos móviles o limitados. Sin embargo, la especificación evolucionó y los tres flujos convergieron en uno que, en teoría, permitía los tres tipos de clientes. Sin embargo, en la práctica, el flujo funcionó bien para las aplicaciones web, pero proporcionaba una experiencia inferior en los otros casos.

A medida que más sitios empezaron a utilizar OAuth, especialmente Twitter, los desarrolladores se dieron cuenta de que el flujo único que ofrecía OAuth era muy limitado y a menudo producía malas experiencias de usuario. Por otro lado, Facebook Connect ofrecía un conjunto más rico de flujos adecuados para aplicaciones web, dispositivos móviles e incluso videoconlas.

OAuth 2.0 aborda este problema definiendo de nuevo múltiples flujos denominados tipos de concesión (*Grant types)* con flexibilidad para admitir una amplia gama de tipos de aplicaciones y con mecanismos para desarrollar extensiones que manejen casos de uso no pensados anteriormente.

De este modo, se puede encontrar que:

- Las aplicaciones del lado del servidor utilizan el tipo de concesión **Código de autorización** con un secreto de cliente, donde la seguridad de este flujo se obtiene por el hecho de que la aplicación del lado del servidor utiliza su secreto para intercambiar el código de autorización por un token de acceso.
- Las aplicaciones móviles o SPA utilizan el mismo tipo de concesión, pero no utilizan el secreto del cliente. En su lugar, la seguridad está en la verificación de la URL de redirección, así como en la extensión **PKCE** opcional.
- El tipo de concesión **Contraseña** permite a las aplicaciones recoger el nombre de usuario y la contraseña del usuario e intercambiarlos por un token de acceso, siendo un flujo pensado para ser utilizado únicamente por clientes de confianza, como la propia aplicación de origen de un servicio.
- La concesión **Credenciales de cliente** se utiliza cuando una aplicación accede a sus propios recursos. Este tipo de concesión consiste simplemente en intercambiar el client_id y el client_secret por un token de acceso.

- Existen extensiones de los tipos de concesión que permiten a las organizaciones definir sus propios tipos de concesión personalizados para admitir tipos de clientes adicionales o para proporcionar un puente entre OAuth y los sistemas existentes. Un ejemplo de esto sería el **Flujo de Dispositivos**, que sirve para autorizar aplicaciones en dispositivos que no tienen un navegador web como los dispositivos IoT.

2.2.3.4. Rendimiento

A medida que los proveedores más grandes comenzaron a utilizar OAuth 1.0, la comunidad se dio cuenta de que el protocolo tenía varias limitaciones que dificultaban su escalabilidad a grandes sistemas. OAuth 1.0 requiere la gestión de estados en diferentes pasos y a menudo en diferentes servidores. También requiere generar credenciales temporales que a menudo eran descartados sin llegarse a usar. T normalmente requiere emitir credenciales de larga duración. Una práctica menos segura y más difícil de gestionar.

Además, OAuth 1.0 requiere que los *endpoints* de los recursos protegidos tengan acceso a las credenciales del cliente para validar la solicitud. Esto rompe la arquitectura típica de la mayoría de los grandes proveedores, en la que se utiliza un servidor de autorización centralizado para emitir las credenciales, y un servidor separado para gestionar las llamadas a la API.

OAuth 2.0 resuelve esto utilizando las credenciales del cliente sólo cuando la aplicación obtiene la autorización del usuario. Después de utilizar las credenciales en el paso de autorización, sólo se utiliza el token de acceso resultante al realizar las llamadas a la API. Por lo tanto, los servidores de la API no necesitan conocer las credenciales del cliente, ya que pueden validar los tokens de acceso ellos mismos.

2.2.3.5. Tokens de corta duración con autorizaciones de larga duración

Como ya se ha comentado anteriormente, las APIs que utilizan OAuth 1.0 suelen emitir tokens de acceso de una duración considerablemente larga, estando en el orden de un año o incluso tener una duración indefinida.

Una buena implementación de una API debe permitir a los usuarios ver qué aplicaciones de terceros han autorizado a utilizar su cuenta, y deben permitir revocar estos accesos a las aplicaciones si lo desean. Al revocar una aplicación, la API debería dejar de aceptar los tokens de acceso emitidos para esa aplicación lo antes posible. Dependiendo de cómo se haya implementado la API, esto podría ser un reto o requerir vínculos adicionales entre las partes internas del sistema.

Con OAuth 2.0, el servidor de autorización puede emitir un token de acceso de corta duración y un token de actualización de larga duración. Esto permite a las aplicaciones obtener nuevos tokens de acceso sin que el usuario tenga que volver a intervenir, pero también añade la posibilidad de que los servidores revoquen los tokens con mayor facilidad. Esta característica se adoptó del protocolo BBAuth de Yahoo! y posteriormente de su extensión de sesión OAuth 1.0.

2.2.3.6. Separación de roles

Una de las decisiones de diseño que se tomaron en OAuth 2.0 fue separar explícitamente las funciones del servidor de autorización del servidor de recursos, permitiendo la construcción del servidor de autorización como un componente independiente que únicamente es responsable de obtener la autorización de los usuarios y emitir tokens a los clientes.

Los dos roles pueden estar en servidores físicamente separados, e incluso estar en diferentes nombres de dominio, lo que permite que cada parte del sistema sea escalada independientemente.

El servidor de autorización necesita conocer el client_id y el client_secret de la aplicación, pero el servidor de la API sólo necesitará aceptar tokens de acceso. Al construir el servidor de autorización como un componente independiente, es posible evitar compartir una base de datos con los servidores de la API, lo que facilita el escalado de los servidores de la API independientemente del servidor de autorización.

La ventaja para los proveedores de servicios es que el desarrollo de estos sistemas puede producirse de forma totalmente independiente, por equipos diferentes y en plazos distintos. Al estar completamente separados, pueden ampliarse de forma independiente, o actualizarse o sustituirse sin que ello afecte a las demás partes de los sistemas.

2.2.4. Aclaraciones sobre OAuth 2.0

OAuth se utiliza para diferentes tipos de APIs y aplicaciones y, aunque se está acercando a la ubicuidad, hay muchas cosas que OAuth no es, y es importante entender estos límites cuando se estudia este protocolo para su aplicación en sistemas software del mundo real.

Dado que OAuth se define como un *framework*, históricamente ha habido cierta confusión con respecto a lo que representa OAuth y lo que no. Para los propósitos de este documento, según lo especificado en los RFC 6749 y 6750, se ha considerado OAuth como un protocolo que detalla varias maneras de obtener un token de acceso, y que hace uso de *Bearer* tokens para acceder a los recursos protegidos en las APIs.

Estas dos acciones, cómo obtener un token y cómo utilizarlo, son las partes fundamentales de OAuth. Como veremos en este apartado, hay una serie de tecnologías en el ecosistema de OAuth que trabajan junto con el núcleo de OAuth para proporcionar una mayor funcionalidad. Sin embargo, hay que tener presente que este ecosistema no debe ser confundido con el protocolo en sí mismo.

A continuación, se presentan las aclaraciones más importantes sobre el protocolo OAuth:

- **OAuth no está definido fuera del protocolo HTTP**. Dado que OAuth 2.0 con *Bearer tokens* no proporciona firmas de mensajes, no está pensado para ser utilizado fuera de HTTPS. OAuth requiere un mecanismo de capa de transporte como TLS para proteger la información y las claves secretas sensibles que se transmiten durante el flujo de comunicación. Existe un estándar para presentar tokens OAuth sobre protocolos protegidos por la Capa Simple de Autenticación y Seguridad (*Simple Authentication and Security Layer, SASL*), hay nuevos trabajos para definir OAuth sobre el Protocolo de Aplicación Restringido (*Constrained Application Protocol*, CoAP), y los trabajos futuros podrían hacer que partes del proceso OAuth sean utilizables sobre enlaces no TLS. Pero incluso en estos casos, es necesario que haya un mapeo claro de las transacciones HTTPS en otros protocolos y sistemas.

- **OAuth no es un protocolo de autenticación**, aunque puede ser usado para construir uno. Una transacción OAuth por sí sola no dice nada sobre quién es el usuario, o incluso si está presente en la comunicación. OAuth es, en esencia, un ingrediente que puede utilizarse en una receta más amplia para proporcionar otras capacidades. Además, OAuth utiliza la autenticación en varios lugares, en particular la autenticación del propietario del recurso y del software cliente al servidor de autorización. Esta autenticación incrustada no convierte a OAuth en un protocolo de autenticación.

- **OAuth no define un mecanismo para la delegación de usuario a usuario**, a pesar de que se trata fundamentalmente de la delegación de un usuario a una pieza de software. OAuth asume que el propietario del recurso es el que controla al cliente. Para que el propietario del recurso autorice a un usuario diferente, se necesita algo más que OAuth. Este tipo de delegación no es un caso de uso poco común, y el protocolo de Acceso Administrado por el Usuario (*User Managed Access Protocol*) utiliza OAuth para crear un sistema capaz de delegar de usuario a usuario.

- **OAuth no define mecanismos de procesamiento de autorizaciones**. OAuth proporciona un medio para transmitir el hecho de que se ha producido una delegación de autorización, pero no define el contenido de esa autorización. En su lugar, el uso de los componentes de OAuth depende de la definición de la API del servidor de recursos, así como los ámbitos y los tokens, para definir a qué acciones es aplicable un token determinado.

- **OAuth no define un formato de token**. El protocolo OAuth establece de forma explícita que el contenido del token es completamente opaco para la aplicación cliente. Esto supone un cambio con respecto a protocolos de seguridad anteriores como WS-Security, Security Assertion Markup Language (SAML) o Kerberos, en los que la aplicación cliente debía ser capaz de analizar y procesar el token. Pese a esto, el token aún debe ser entendido por el servidor de autorización que lo emite y el recurso protegido que lo acepta. Esto es lo que ha llevado al desarrollo del formato *JSON Web Token* (JWT) y del protocolo *Token Introspection*. El token en sí sigue siendo opaco para el cliente, pero ahora otras partes pueden entender su formato.

- **OAuth 2.0 no define métodos criptográficos**, a diferencia de OAuth 1.0. En lugar de definir un nuevo conjunto de mecanismos criptográficos específicos para OAuth, el protocolo OAuth 2.0 está construido para permitir la reutilización de mecanismos criptográficos de propósito más general que pueden ser utilizados fuera de OAuth. Esta omisión deliberada ha contribuido al desarrollo del conjunto de especificaciones *JSON Object Signing and Encryption* (JOSE), que proporciona mecanismos criptográficos de propósito general que pueden utilizarse junto a OAuth e incluso fuera de él.

- **OAuth 2.0 no es un protocolo único**. Como se ha comentado anteriormente, la especificación está dividida en múltiples definiciones y flujos, cada uno de los cuales tiene su propio conjunto de casos de uso. El núcleo de la especificación OAuth 2.0 ha sido descrito con cierta precisión como un generador de protocolos de seguridad, porque puede ser utilizado para diseñar la arquitectura de seguridad para muchos casos de uso diferentes; los cuales, no son necesariamente compatibles entre sí.

2.3. OAuth en Java Spring Framework

2.3.1. Introducción

En el presente apartado se describe el lenguaje de programación Java y, más concretamente, su Framework Spring, repasando algunos de los proyectos más interesantes derivados de este framework.

Java es un lenguaje de programación que fue diseñado por James Gosling en 1996. Se caracteriza por ser un lenguaje de programación de alto nivel, es decir, su sintaxis se encuentra más cercana al lenguaje natural que al lenguaje máquina, permitiendo al desarrollador olvidarse por completo del funcionamiento interno de la máquina para la que está diseñando el programa a cambio de introducir la necesidad de un traductor que interprete el código fuente y lo traduzco al lenguaje de la máquina.

Se caracteriza por seguir el paradigma de programación orientado a objetos, que es una forma de programación en la que los problemas se tratan de resolver modelando objetos que representan elementos del mundo real y definiendo una serie de atributos y operaciones que se pueden realizar con ellos.

Otra de las características más importantes de Java es su capacidad para ejecutarse de forma independiente a la plataforma, sin importar el tipo de hardware, como se dice en uno de sus axiomas más famosos: "Write once, run anywhere" [28].

Lo que permite que esto sea posible es el entorno ejecución junto al que se distribuye el lenguaje, la Java Virtual Machine (JVM). La plataforma de Java donde este entorno viene incluida es capaz de generar un código intermedio (conocido como bytecode) a partir del código fuente compilado. Después, este código intermedio es ejecutado en la JVM, que está escrita en código nativo de la plataforma de destino y, por tanto, es capaz de entender su hardware.

El único requisito para desarrollar una máquina virtual de Java es que cumpla con la especificación de JVM, así que, gracias a esto, los programas escritos en Java pueden llegar a ejecutarse en dispositivos muy variados y en aplicaciones distribuidas.

La forma de distribución de Java se realiza en función de diferentes perfiles, según las necesidades de la plataforma o aplicación que se quiera desarrollar. De este modo, para aplicaciones sencillas ejecutadas en ordenadores personales se tiene *Java Standard Edition* (JSE), y para aplicaciones de carácter empresarial se tiene Jakarta EE (antiguamente conocido como Java Enterprise Edition), que incluye todas las funciones de JSE y las amplía con herramientas que buscan reducir la complejidad del código fuente y mejorar el rendimiento de las aplicaciones.

2.3.2. Spring Framework

Spring Framework es un amplio conjunto de bibliotecas y herramientas que simplifican el desarrollo de software, a saber, inyección de dependencias, acceso a datos, validación, internacionalización, programación orientada a aspectos, etc. Es una opción popular para los proyectos Java, y también funciona con otros lenguajes basados en la JVM, como Kotlin y Groovy.

Este framework Open Source surgió en 2003 como respuesta a la creciente complejidad de las primeras especificaciones de JEE. De hecho, el modelo de programación de Spring no adopta la especificación de JEE, sino que integra cuidadosamente algunos de sus especificaciones individuales (como Servlet API, WebSocket API, Bean Validation o JPA).

Una característica clave en Spring es el contenedor de Inversión de Control (IoC), que es soportado por la interfaz ApplicationContext. Spring crea este "espacio" en la aplicación donde el desarrollador y el propio framework pueden enviar algunas instancias de objetos, como pools de conexión a bases de datos, clientes HTTP, etc.

Estos objetos, llamados beans, pueden ser utilizados posteriormente en otras partes de la aplicación, normalmente a través de su interfaz pública para abstraer el código de implementaciones específicas. El mecanismo para referenciar uno de estos beans desde el contexto de la aplicación en otras clases es lo que se conoce como inyección de dependencias, y en Spring esto es posible a través de configuración mediante XML o anotaciones de código Java.

Una de las razones por las que Spring es tan popular es que ahorra mucho tiempo al proporcionar implementaciones integradas para muchos aspectos del desarrollo de software. A lo largo de su evolución, ha dado lugar a otros proyectos de código abierto para diferentes propósitos que constituyen un framework por sí mismos. Algunos de ellos son:

- Spring Boot, que permite la creación rápida de aplicaciones partiendo de plantillas configurables preestablecidas.

- Spring Data, que simplifica el acceso a los datos para las bases de datos relacionales y NoSQL.

- Spring Batch, que proporciona una potente capacidad de procesamiento para grandes volúmenes de registros.

- Spring Security, todo un marco de seguridad que abstrae las características de seguridad a las aplicaciones.

- Spring Cloud, que proporciona herramientas para que los desarrolladores construyan rápidamente algunos de los patrones más comunes en los sistemas distribuidos como los microservicios.

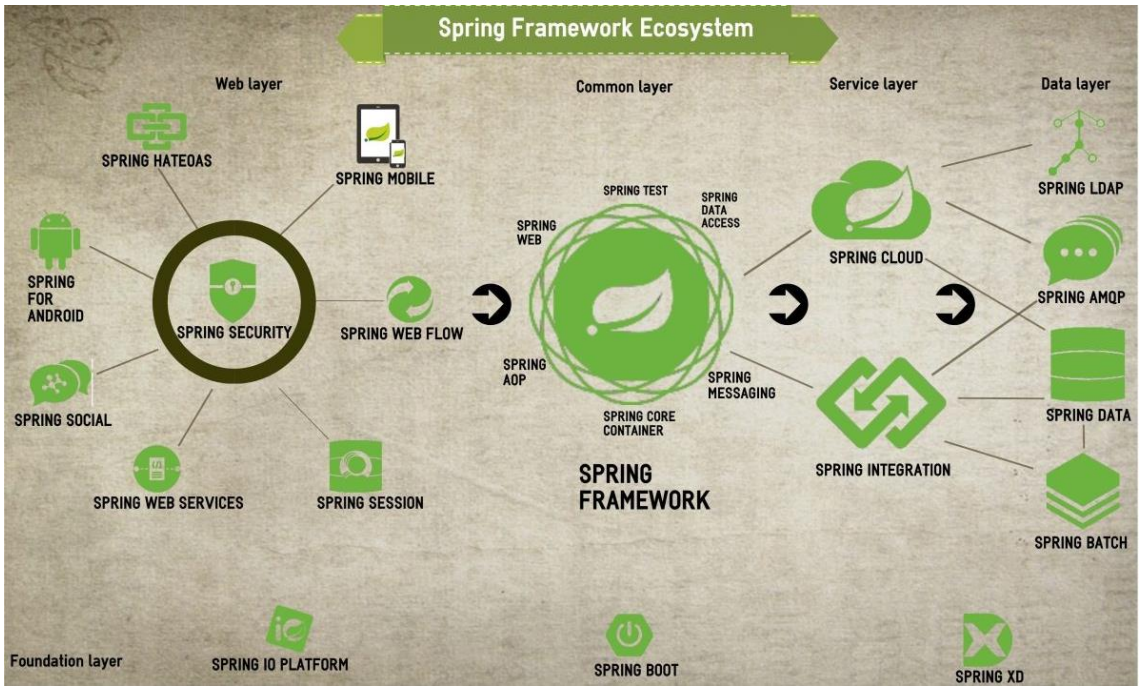

Figura 29. Ecosistema de Spring Framework.

Todos estos módulos están construidos sobre el núcleo de Spring Framework, que establece un modelo de programación y configuración común para las aplicaciones de software. Este modelo en sí es otra razón importante para elegir el framework, ya que facilita buenas técnicas de programación como el uso de interfaces en lugar de clases para desacoplar las capas de la aplicación mediante la inyección de dependencias.

2.3.3. Spring Boot

Spring Framework se caracteriza por tener una gran cantidad de módulos y otras librerías de terceros que pueden combinarse con el framework, lo que lo convierte en una herramienta poderosa a la hora de desarrollar software. Sin embargo, la configuración de este framework es un proceso algo lento pese a los esfuerzos del equipo de Spring por facilitarlo. Además, dicha configuración no es algo que suela ser exclusivo a nivel de aplicación, sino que tiende a repetirse en la mayoría de los casos.

Spring Boot es un framework que aprovecha Spring Framework para crear rápidamente aplicaciones independientes en lenguajes basados en Java, lo que lo convierte en una herramienta muy popular para construir microservicios. Su principal ventaja es que elimina la mayor parte del proceso de configuración de Spring mediante la introducción de una serie de herramientas y configuraciones por defecto que se establecen de forma automática para el desarrollador. Es lo que se conoce como *starter dependencies.*

Al utilizar Spring Boot, un desarrollador puede incluir una serie de *starter dependencies*, que son como colecciones de módulos de Spring distribuidas junto a algunas librerías y herramientas de terceros, mejorando enormemente la experiencia a la hora de iniciar un proyecto o añadir dependencias a uno existente. Además, aunque Spring Boot incluya sus auto-configuraciones para el proyecto, los desarrolladores pueden anular los valores predeterminados y personalizar el proyecto de diferentes maneras. Así, por ejemplo, la dependencia spring-boot-starter-web sirve para ayudar en la construcción de aplicaciones web independientes, agrupando la librería Spring Core Web con Jackson (para el manejo de JSON), validación, herramientas de logging, autoconfiguración e incluso un servidor Tomcat embebido, entre otras herramientas.

Los objetivos principales de este framework, tal y como se enuncian en su web oficial, son:

- Proporcionar una experiencia de inicio radicalmente más rápida y ampliamente accesible para todo el desarrollo de Spring.
- Seguir un diseño "opinionated" desde el principio, pero permitir modificaciones cuando los requisitos empiecen a divergir de los predeterminados.
- Proporcionar una serie de características no funcionales comunes a una gran cantidad de proyectos (como servidores integrados, seguridad, métricas, controles de salud y configuración externalizada).
- No generar absolutamente ninguna clase de código y configuración XML.

*El patrón de diseño Opinionated consiste en establecer unas directrices de configuración para indicar al desarrollador la forma de hacer las cosas. Suele emplearse en situaciones donde es necesario reducir los esfuerzos de los desarrolladores a la hora de establecer la configuración inicial de una aplicación.

Las características clave de Spring Boot incluyen:

- **Spring Boot Starters**: como se ha mencionado anteriormente, son módulos preconfigurados con las dependencias y librerías más utilizadas para un propósito específico con el objetivo de que el desarrollador no tenga que buscar y configurar las versiones compatibles de estas librerías de forma manual y pueda comenzar a desarrollar rápidamente. Algunos ejemplos son spring-boot-starter-web, spring-boot-starter-data-jpa, spring-boot-starter-security, spring-boot-starter-test, spring-boot-starter-data-rest…

- **Spring Boot Autoconfiguration**: Spring Boot aborda el problema de la complejidad de la configuración inicial de las aplicaciones de Spring eliminando la necesidad de establecer manualmente plantillas de configuración. Siguiente un diseño opinionable configura varios componentes automáticamente, registrando *beans* en función de varios criterios: disponibilidad de una clase concreta en el classpath, presencia o ausencia de un *bean* de Spring, presencia de una propiedad del sistema, ausencia de un archivo de configuración. De este modo, si la dependencia spring-webmvc existe en el classpath, Spring Boot asume que se está intentando construir una aplicación web basada en Spring MVC y automáticamente intenta registrar el *DispatcherServlet* si no ha sido registrado previamente.

- **Gestión elegante de la configuración**: Spring soporta la externalización de propiedades configurables mediante la anotación @PropertySource, permite vincular propiedades con tipos seguros a las propiedades de los *beans* y soporta la presencia de múltiples archivos de configuración separados en diferentes perfiles.

- **Spring Boot Actuator**: proporciona una gran variedad de funciones preparadas para ser utilizadas en entornos de producción sin necesidad de que los desarrolladores tengan que escribir una gran cantidad de código. Entre sus características, el Spring Boot Actuator permite visualizar los detalles de la configuración de los *beans* de la aplicación, las asignaciones de las URL de la aplicación, detalles del entorno y valores de los parámetros de configuración, y las métricas de comprobación de estado de los servicios registrados.

- **Soporte de contenedor de servlets incrustado fácil de usar**: Tradicionalmente, al construir aplicaciones web, es necesario crear módulos de tipo WAR y luego desplegarlos en servidores externos como Tomcat, WildFly, etc. Pero al utilizar Spring Boot, es posible crear un módulo de tipo JAR e incrustar el contenedor de servlets en la aplicación muy fácilmente, de modo que la aplicación será una unidad de despliegue autocontenida. Además, durante el desarrollo, se puede ejecutar fácilmente el módulo de tipo JAR de Spring Boot como una aplicación Java desde el IDE o desde la línea de comandos utilizando una herramienta de construcción como Maven o Gradle.

2.3.4. Spring Cloud

Spring Cloud proporciona herramientas para que los desarrolladores implementen rápidamente algunos de los patrones habituales de los sistemas distribuidos como, por ejemplo, gestión de la configuración, descubrimiento de servicios, circuit breakers, enrutamiento inteligente, microproxy, bus de control, tokens de un solo uso, bloqueos globales, elección de liderazgo, sesiones distribuidas o estado del clúster.

La coordinación de los sistemas distribuidos da lugar a patrones de plantilla, y utilizando Spring Cloud los desarrolladores pueden poner en marcha rápidamente servicios y aplicaciones que implementen esos patrones.

Spring Cloud se centra en proporcionar una buena experiencia inmediata para los casos de uso típicos y un mecanismo de extensibilidad para cubrir otros. Sus características principales son:

- Configuración distribuida/versionada
- Registro y descubrimiento de servicios
- Enrutamiento
- Llamadas de servicio a servicio
- Balanceo de carga
- Circuit breakers
- Cerraduras globales
- Elección de líderes y estado del clúster
- Mensajería distribuida

La forma más fácil de empezar a utilizar Spring Cloud es visitar start.spring.io y seleccionar la versión de Spring Boot y los proyectos de Spring Cloud que se desean utilizar. Esto añadirá la versión correspondiente de Spring Cloud BOM al archivo Maven/Gradle cuando se genere el proyecto.

Para una aplicación ya existente de Spring Boot en la que se quiera añadir Spring Cloud, el primer paso es determinar la versión de Spring Cloud que debe utilizar. Dicha versión dependerá de la versión de Spring Boot que se esté utilizando.

La tabla de la figura 30 indica qué versión de Spring Cloud se corresponde con cada versión de Spring Boot.

Spring Cloud	Spring Boot version
2020.0.x (Ilford)	2.4.x, 2.5.x (a partir de 2020.0.3)
Hoxton	2.2.x, 2.3.x (a partir de SR5)
Greenwich	2.1.x
Finchley	2.0.x
Edgware	1.5.x
Dalston	1.5.x

Figura 30. Compatibilidad de Spring Cloud y Spring Boot

Spring Cloud tiene un total de 23 proyectos principales. Para los propósitos de este documento, se va a poner el foco en dos de ellos, Spring Cloud Gateway y Spring Cloud Netflix (más concretamente, en su componente Eureka).

2.3.4.1. Spring Cloud Gateway

Este apartado sirve como breve introducción a la tecnología API Gateway dentro de un esquema de microservicios y a las opciones que tiene Spring para su implementación.

Un gateway es un servidor de enrutamiento dinámico compuesto por filtros, cada uno de los cuales está enfocado en una tarea en particular específica cómo puede ser, por ejemplo, autorización, seguridad, monitorización o análisis estadísticos.

Existe un filtro principal construido en el propio framework que sirve para enrutar los servicios de la aplicación, pero también se permite la posibilidad de que el desarrollador defina sus propios filtros personalizados para, por ejemplo, modificar una respuesta, realizar el manejo de errores, modifica la cabecera o parámetros de una petición, devolver una respuesta estática para una ruta determinada o bien interceptar el acceso a un servicio en particular y modificar su enrutamiento (hacia otro servicio, hacia una versión de prueba o hacia otra versión).

Dentro del ecosistema de Spring, se pueden encontrar dos implementaciones importantes de API Gateway: Spring Zuul y Spring Cloud Gateway.

Zuul es un proyecto desarrollado por Netflix que tenía el objetivo de añadir un conjunto de funciones enrutamiento y seguridad a través del uso de filtros sobre las peticiones entrantes a un sistema, actuando como puerta de entrada al mismo. Aunque actualmente se encuentra en modo de mantenimiento, por lo que no va a recibir nuevas características, su uso está bastante extendido dentro de la industria.

Spring Cloud Gateway es un proyecto propio desarrollado por el equipo de Spring Cloud que se ha constituido como la principal alternativa al uso de Zuul. Utiliza programación reactiva, por lo que sus flujos de comunicaciones son mucho más rápidos, no se bloquean, permiten procesamiento en paralelo independiente, comunicación asíncrona y todas las ventajas asociadas a este paradigma de programación

Spring Cloud Gateway permite crear una puerta de enlace, lo que también se denomina servidor perimetral. En una arquitectura de microservicios es el punto de entrada a este ecosistema y será el encargado de enrutar cada uno de estos servicios a una url base y a partir de esa ruta redirigir el tráfico hacia rutas individuales definidas para cada servicio.

También integra balanceo de carga, a través de Ribbon y del propio Spring Cloud Load Balancer, siendo la segunda opción la más recomendada al estar desarrollada también por el equipo de Spring Cloud. Este proyecto define un balanceador de carga que también maneja filtros propios, construido por debajo, pero también permite extender funcionalidades,

Como sostiene la definición de un gateway, Spring Cloud Gateway maneja filtros propios, pero también permite extender funcionalidades mediante la creación de filtros definidos por el desarrollador. De esta forma se puede definir un filtro que permita rechazar el acceso a un microservicio por motivos de seguridad si se detecta que la petición no contiene ningún token de acceso o dicho token es incorrecto.

Para incluir Spring Cloud Gateway en un proyecto, es necesario usar el starter con group id org.springframework.cloud y artifact id spring-cloud-starter-gateway.

A la hora de trabajar con Spring Cloud Gateway es necesario tener presente los siguientes tres conceptos:

- Ruta: El bloque básico de la puerta de enlace. Se define mediante un ID, una URI de destino, una colección de predicados y una colección de filtros. Un flujo de comunicación va a seguir la ruta siempre que el predicado se evalúe como verdadero.
- Predicado: representa la condición para determinar la coincidencia con una ruta. Spring Cloud Gateway define una serie de factorías de predicados para buscar coincidencias en diferentes atributos de una petición HTTP entrante.
- Filtro: instancias de la clase GatewayFilter, constuidas con una factoría específica, que permiten modificar las peticiones y las respuestas antes o después de enviar la solicitud descendente.

La siguiente figura representa un diagrama de alto nivel para entender el funcionamiento de Spring Cloud Gateway:

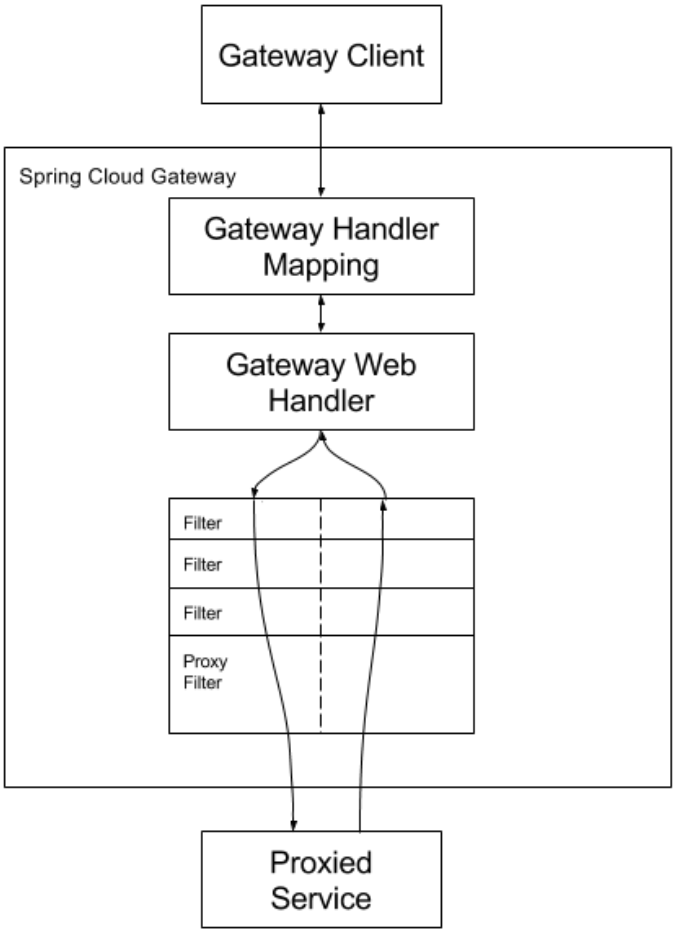

Figura 31. Procesamiento de una petición en Spring Cloud Gateway

Los clientes hacen peticiones a Spring Cloud Gateway. Si el Gateway Handler Mapping determina que una petición coincide con una ruta, esta se envía Gateway Web Handler. Este manejador transporta la solicitud a través de una cadena de filtros que es específica para dicha solicitud. La razón por la que los filtros están divididos por la línea de puntos es que los filtros pueden ejecutar la lógica tanto antes como después de que se envíe la solicitud del proxy. Es decir, primero se podría ejecutar toda la lógica de filtro preprocesamiento. A continuación, se realizaría la solicitud de proxy y, después de realizar la solicitud proxy, se ejecutaría la lógica de filtros postprocesamiento.

Con el objetivo de profundizar un poco más en los mecanismos de Spring Cloud Gateway a la hora de buscar coincidencias en las rutas de una petición, a continuación se exponen los diferentes tipos de factorías de predicados existentes:

- **After Route Predicate Factory / Before Route Predicate Factory**: toma un parámetro de tipo Java ZonedDateTime y comprueba si la petición ha tenido lugar antes (en el caso del Before Route) o después (en el caso del After Route) de dicho parámetro.
- **Between Route Predicate Factory**: toma dos parámetros de tipo Java ZonedDateTime y comprueba si la petición ha tenido lugar después del primer

parámetro y antes del segundo. Para ello, es obligatorio que el segundo parámetro sea una marca de tiempo posterior al primero.

- **Cookie Route Predicate Factory**: recibe un parámetro que representa el nombre de una cookie y otro parámetro en forma de expresión regular. Analiza si la petición contiene una cookie con el nombre recibido en el primer parámetro y los valores de esta cookie coinciden con la expresión regular recibida como segundo parámetro.
- **Header Route Predicate Factory**: recibe un parámetro que representa el nombre de una cabecera y otro parámetro en forma de expresión regular. Comprueba si alguna cabecera tiene el nombre del primer parámetro y su valor coincide con los resultados de la expresión regular recibida como segundo parámetro.
- **Host Route Predicate Factory**: recibe como parámetro un listado de patrones para buscar coincidencias en el campo host de la cabecera de la petición.
- **Method Route Predicate Factory**: recibe como parámetro un listado de cadenas de texto con nombres de métodos HTTP. La función comprobará si la petición contiene alguno de estos métodos.
- **Path Route Predicate Factory**: recibe como parámetros una lista de patrones Spring PathMatcher y un flag llamado matchTrailingSlash (por defecto con valor verdadero). El primer parámetro sirve para buscar coincidencias en la uri de la petición y el segundo, cuando se define como falso, hará que el filtrado de la uri sea estricto en cuanto al carácter "/" final.
- **Query Route Predicate Factory**: recibe un parámetro para comprobar si la petición contiene algún parámetro de consulta con dicha clave y un segundo parámetro (opcional) para comprobar el valor de dicho parámetro de consulta.
- **RemoteAddr Route Predicate Factory**: recibe como parámetros un listado de direcciones en notación CIDR (por ejemplo: 192.168.0.1/16) y comprueba si la dirección de origen de la cabecera de la petición coincide con alguno de los valores.
- **Weight Route Predicate Factory**: recibe como parámetros una cadena de texto que representa un grupo y un número entero que representa un peso. Este tipo de predicados sirve para distribuir las peticiones entrantes a diferentes servicios. En el ejemplo de la figura 32 un 80% de las peticiones serían distribuidas a weighthigh.org y un 20% a weighlow.org.

```yaml
spring:
  cloud:
    gateway:
      routes:
      - id: weight_high
        uri: https://weighthigh.org
        predicates:
        - Weight=group1, 8
      - id: weight_low
        uri: https://weightlow.org
        predicates:
        - Weight=group1, 2
```

Figura 32. Ejemplo de Weight Route Predicate Factory

2.3.4.2. Spring Netflix Eureka

El descubrimiento de servicios es uno de los principios clave de una arquitectura basada en microservicios. Este concepto trata de resolver el problema de encontrar el nombre de host o la dirección IP de una máquina. El descubrimiento de servicios puede ser algo tan simple como mantener un archivo de propiedades con las direcciones de todos los servicios remotos utilizados por una aplicación, o algo tan formalizado como un repositorio de Descripción, Descubrimiento e Integración Universal (UDDI). El descubrimiento de servicios es fundamental para las aplicaciones de microservicios basadas en la nube por dos razones fundamentales:

- Escalado horizontal: este patrón suele requerir ajustes en la arquitectura de la aplicación, como añadir más instancias de un servicio dentro de un servicio en la nube y más contenedores.
- Resiliencia: Este patrón se refiere a la capacidad de absorber el impacto de los problemas dentro de una arquitectura o servicio sin afectar al negocio. Las arquitecturas de microservicios deben ser extremadamente sensibles para evitar que un problema en un solo servicio (o instancia de servicio) se extienda en cascada a los consumidores del servicio.

Para comprender mejor todo lo que conlleva el descubrimiento de servicios, es necesario entender los siguientes cuatro conceptos, compartidos por prácticamente todas las implementaciones de descubrimiento de servicios:

- Registro de servicios: cómo se registra un servicio en el agente de descubrimiento de servicios.
- Búsqueda de la dirección del servicio por parte del cliente: cómo un cliente de servicio busca la información del servicio.
- Intercambio de información: cómo los nodos comparten la información de los servicios

- Monitorización de la salud: cómo los servicios comunican su salud al agente de descubrimiento de servicios

El objetivo principal del descubrimiento de servicios es disponer de una arquitectura en la que los servicios indiquen dónde se encuentran físicamente en lugar de tener que configurar manualmente su ubicación. La siguiente figura muestra cómo se añaden y eliminan instancias de servicio, y cómo actualizan el agente de descubrimiento de servicios y están disponibles para procesar las solicitudes de los usuarios.

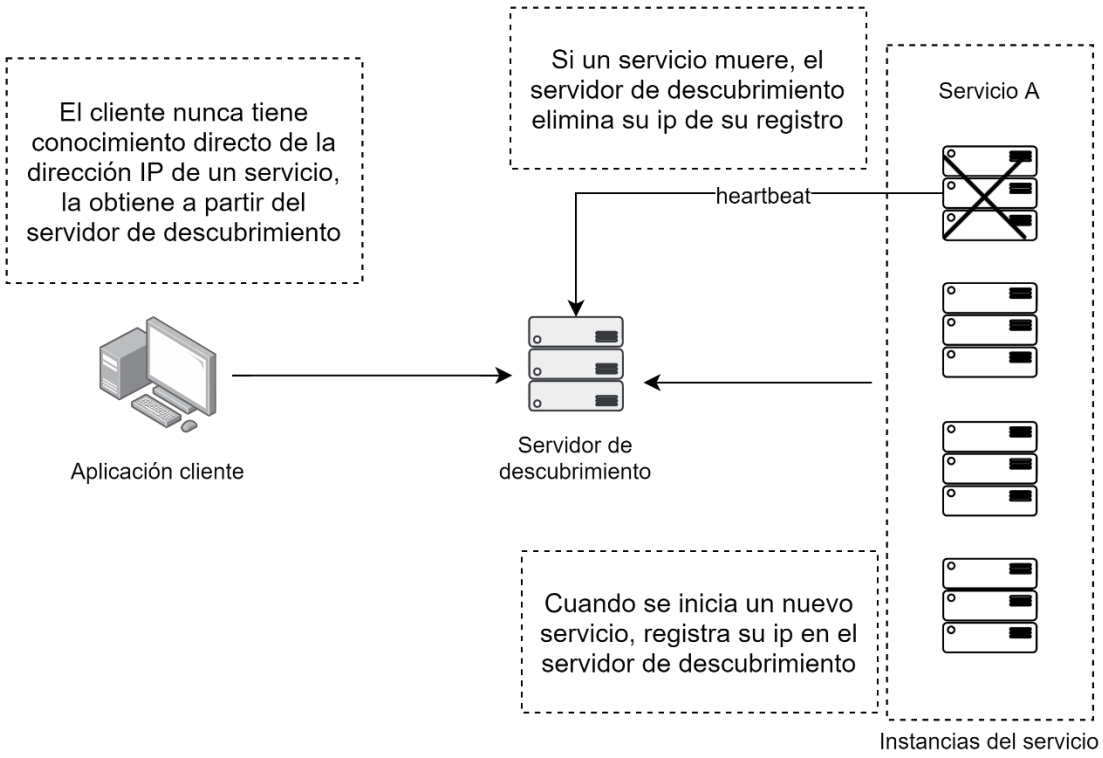

Figura 33. Descubrimiento de servicios

Tratar de configurar a mano cada cliente o alguna forma de convención puede ser un proceso frágil y difícil de realizar. Aquí es donde entra en juego Spring Netflix Eureka. Netflix Eureka implementa el descubrimiento de servicios del lado del cliente, lo que significa que los clientes ejecutan un software que habla con el servicio de descubrimiento, Netflix Eureka, para obtener información sobre las instancias de microservicios disponibles.

El proceso es el siguiente:

1. Cada vez que se pone en marcha una instancia de microservicio este se registra en uno de los servidores de Eureka.

2. De forma periódica, cada instancia de microservicio envía un mensaje *heartbeat* al servidor Eureka, indicándole que la instancia de microservicio está bien y está lista para recibir peticiones.

3. Los clientes utilizan una biblioteca de clientes que solicita regularmente al servicio Eureka información sobre los servicios disponibles.

4. Cuando el cliente necesita enviar una solicitud a otro microservicio, ya tiene una lista de instancias disponibles en su biblioteca de clientes y puede elegir una de ellas sin preguntar al servidor de descubrimiento. Normalmente, las instancias disponibles se eligen de forma rotatoria; es decir, se llaman una tras otra antes de volver a llamar a la primera.

Cuando un cliente se registra en Eureka, proporciona metadatos sobre sí mismo, como el host, el puerto, la URL del indicador de salud, la página de inicio y otros detalles. Eureka recibe mensajes de heartbeat de cada instancia perteneciente a un servicio. Si el heartbeat falla durante un horario configurable, la instancia se elimina normalmente del registro.

Para incluir el cliente Eureka en un proyecto, es necesario utilizar el starter con group id org.springframework.cloud y artifact id spring-cloud-starter-netflix-eureka-client.

Para incluir el servidor Eureka en un proyecto, es necesario utilizar el starter con group id org.springframework.cloud y artifact id spring-cloud-starter-netflix-eureka-server.

2.3.5. Spring Security y OAuth

Spring Security es la principal opción para implementar la seguridad a nivel de aplicación en las aplicaciones Spring. En general, su propósito es ofrecerle una forma altamente personalizable de implementar la autenticación, la autorización y la protección contra los ataques más comunes, incluyendo soporte para estándares de seguridad como LDAP y OAuth2. Spring Security es un software de código abierto publicado bajo la licencia Apache 2.0.

Este framework, además, se integra bien con los otros módulos y proyectos de Spring y puede hacer uso de proxies basados en anotaciones. También funciona muy bien con SpEL (Spring Expression Language).

Spring Security es fácil de ampliar y tiene muchas funciones incorporadas:

- Soporte completo y extensible para la autenticación y la autorización.
- Protección contra ataques como la fijación de la sesión, el clickjacking, la falsificación de peticiones entre sitios, etc.
- Integración de la API de servlets.
- Integración opcional con Spring Web MVC.
- Soporte para OAuth y OAuth2.
- Soporte para SAML.

A modo de resumen, podría decirse que el núcleo de Spring Security gira en torno a dos objetivos: la autenticación, que decide la identidad del usuario (llamado *principal* dentro del framework), y la autorización, que decide qué usuarios tienen acceso a qué recursos.

Respecto al apartado de la autorización, hace muchos años el equipo de Spring comenzó a trabajar en un proyecto open source llamado Spring Security OAuth. Con el paso del tiempo, este proyecto comenzó a desarrollar un grado de madurez muy alto y se convirtió en un proyecto muy popular que soportaba gran parte de la especificación de OAuth. Incluía soporte para las aplicaciones cliente, soporte para desarrollar un servicio de recursos y soporte para el servidor de autorización, todo ello dentro de un único proyecto.

La popularidad de este proyecto fue tan alta que a día de hoy, muchas aplicaciones empresariales cuentan con Spring Security OAuth como una de sus dependencias.

Como consecuencia del crecimiento que estaba experimentando el proyecto, en 2018 el equipo de Spring decidió reescribir por completo el proyecto Spring Security OAuth para hacerlo aún mejor y simplificar su código base, provocando que el proyecto original de Spring Security OAuth se empezase a dejar de lado.

Esto ha generado una confusión importante en los desarrolladores, que se encuentran como alternativas un proyecto nuevo en fase de desarrollo y una versión cada vez con menos soporte del proyecto original, el cual está marcado como "deprecated" en su repositorio de GitHub.

El equipo de Spring ha estado proporcionando parches y actualizaciones de seguridad para el proyecto de Spring Security Oauth hasta mayo de 2021 y según el anuncio de fin de vida más

reciente, el plan de Spring es dar soporte a la versión 2.5.x hasta mayo de 2022, momento en el que el proyecto se abandonará por completo.

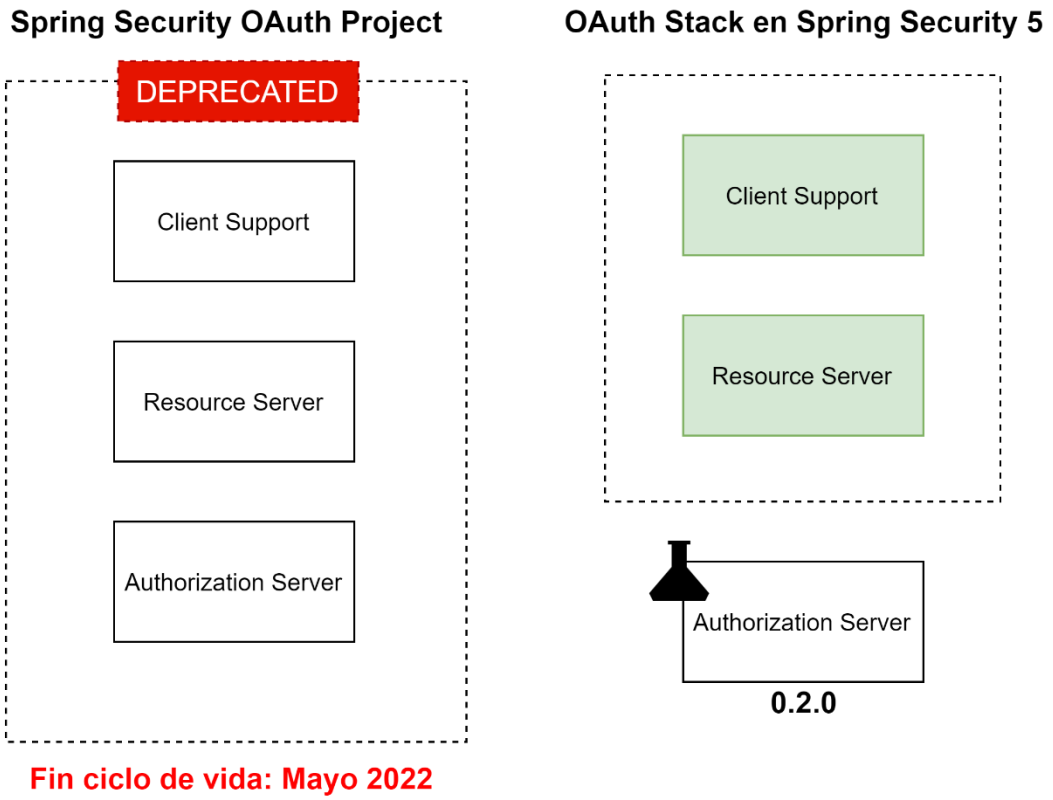

Figura 34. Panorama de Spring Security en relación con OAuth2

Por tanto, la alternativa más factible es utilizar el nuevo proyecto que, si bien no está del todo terminada, está en una fase bastante madura de su ciclo de desarrollo y cuenta con varios componentes terminados, además del soporte oficial del equipo de Spring. El nuevo stack de tecnologías del proyecto, incluidas a partir de la versión 5 de Spring Security, incluyen soporte para aplicaciones cliente y para servidor de recursos, siendo el servidor de autorización el único rol de OAuth que no cuenta con un soporte completo. Este proyecto está intentando llevar a la práctica la nueva versión 2.1 del protocolo OAuth, que aún se encuentra en fase de borrador (https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-03)

Como el servidor de autorización se encuentra en una fase experimental en la que ni siquiera están soportados todos los tipos de concesión de OAuth, su utilización no es recomendada salvo para pruebas de concepto o para colaborar en el desarrollo como parte de la comunidad open source. Por ello, es común encontrarse con aplicaciones que aprovechan los componentes funcionales del nuevo Spring Security (es decir, los proyectos del cliente y servidor de recursos) y utilizan aplicaciones como Keycloak para implementar el rol de servidor de autorización

Los principales módulos relacionados del proyecto OAuth que incluye Spring Security son:

- OAuth 2.0 Core: contiene el núcleo de clases e interfaces necesarias para garantizar el soporte del framework de autorización OAuth 2.0 y para OpenID Connect 1.0. Es una dependencia obligatoria para cualquier aplicación que vaya a representar el rol de cliente, servidor

de recursos o servidor de autorización. Se define en el paquete org.springframework.security.oauth2.core.

- OAuth 2.0 Client: contiene soporte de Spring Security para la implementación de un cliente del framework de autorización OAuth 2.0 y para OpenID Connect 1.0. Se define en el paquete org.springframework.security.oauth2.client.

- OAuth 2.0 JOSE: contiene soporte de Spring Security para el framework JavaScript Object Signing and Encryption (JOSE). Este framework busca proporcionar un método para transferir de forma segura los claims entre las diferentes partes del proceso de comunicación. Está construido sobre una colección de especificaciones: JSON Web Token (JWT), JSON Web Signature (JWS), JSON Web Encryption (JWE) y JSON Web Key (JWK). Contiene los paquetes org.springframework.security.oauth2.jwt y org.springframework.security.oauth2.jose.

- OAuth 2.0 Resource Server: contiene soporte de Spring Security para la implementación de servidores de recursos de OAuth 2.0. Se usa para proteger APIs medio del uso de Bearer Tokens de OAuth 2.0. Se define en el paquete org.springframework.security.oauth2.server.resource

3. DESARROLLO PRÁCTICO

3.1. Análisis y diseño del Sistema

En este apartado se describe la aplicación que se ha desarrollado con el fin de llevar a la práctica los conceptos teóricos que se acaban de exponer en el capítulo anterior.

Como se comentó en el apartado de objetivos del proyecto, el sistema que se desea construir es una aplicación web con una arquitectura basada en microservicios que permita demostrar el funcionamiento del protocolo OAuth 2.0, siendo el dominio específico de la aplicación el de una web de gestión de cursos académicos.

El lenguaje de programación elegido para el desarrollo de este sistema es Java por todas las ventajas que tiene el empleo de Spring Framework y su ecosistema de módulos a la hora de construir aplicaciones distribuidas independientes y aplicar patrones de arquitectura diseñados para solventar los problemas más comunes de estos servicios, además de los mecanismos de seguridad necesarios para aplicar OAuth.

El diagrama de la figura 35, tomando todas estas aclaraciones bajo consideración, representa todos los componentes que integran el sistema, así como la comunicación establecida entre unos y otros.

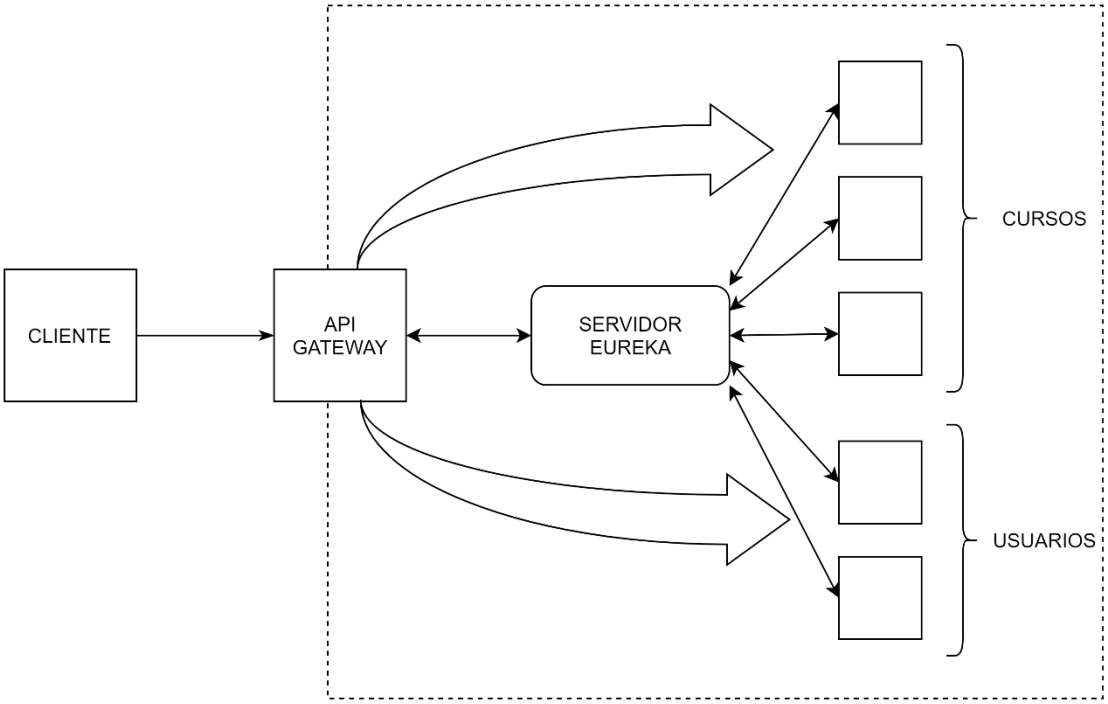

Figura 35. Microservicios principales de la aplicación de gestión de cursos

La aplicación expone dos APIs REST de recursos principales, el servicio de cursos (que contendrá los endpoints necesarios para realizar operaciones CRUD relacionadas con los cursos académicos) y el servicio de usuarios (que contendrá los endpoints necesarios para realizar operaciones CRUD relacionadas con los usuarios de la aplicación).

El acceso a ambas aplicaciones es controlado por medio de un servicio Gateway, que se encarga de procesar las peticiones entrantes al sistema y, tras aplicar su lógica de predicados, redirigirlas al servicio apropiado.

Tanto los servicios de recursos como el servidor api Gateway estarán registrados en un servidor Eureka, que se encargará del descubrimiento de servicios en la aplicación. De este modo, el servidor de Eureka va a ser capaz de conocer las direcciones de cada servicio y si todos ellos están funcionando correctamente.

Vista esta arquitectura, el tipo de concesión de autorización del protocolo OAuth que se va a utilizar es Authorization Code. El flujo que sigue este tipo de concesión implica que, para que una aplicación cliente pueda acceder a las APIs de recursos protegidas, debe iniciar una solicitud al servidor de autorización en la que, tras recibir el consentimiento del dueño de los recursos, este emitirá un código de autorización que la aplicación deberá canjear por un token de acceso. Una vez que se tenga este token de acceso, la aplicación podrá acceder a los recursos protegidos, siempre que la información del token lo permita.

Este flujo se ve reflejado en el diagrama de la figura 36:

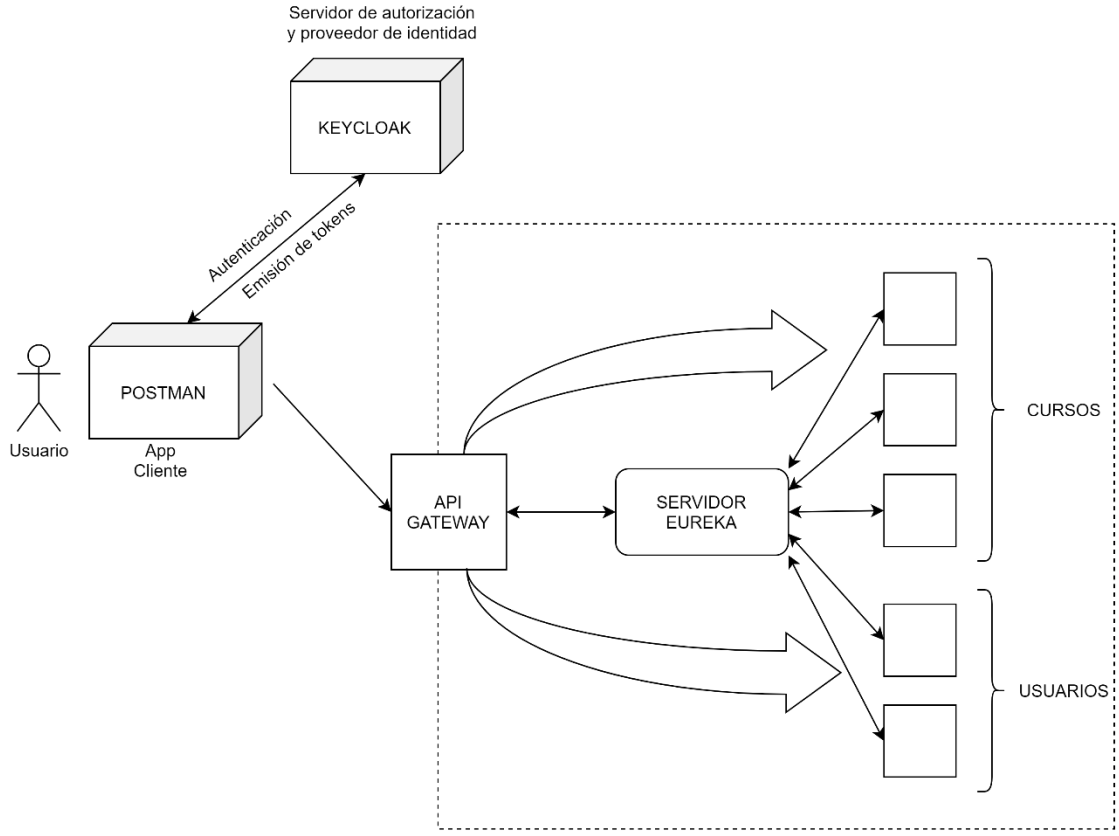

Figura 36. Aplicación de OAuth2.0 sobre los microservicios definidos

Este diagrama introduce dos nuevos componentes que no aparecían en la figura 35. Por un lado, un servidor de Keycloak, que hace las funciones de servidor de autorización y proveedor de identidad; y por otro lado, la aplicación de Postman, que será la encargada de hacer las funciones de cliente. Los recursos protegidos estarían representados por el servicio de cursos y el servicio de usuarios.

3.2. Implementación del sistema

En este apartado se describen los pasos que se han seguido a la hora de implementar cada uno de los microservicios que constituyen el sistema y que se han descrito en el apartado anterior, prestando especial atención a los proyectos de Spring Framework utilizados para conseguir la funcionalidad específica de cada microservicio y a la configuración necesaria para que se cumpla con la especificación del protocolo OAuth.

3.2.1. Servidores de recursos: servicio de cursos y servicio de usuarios

Dentro de la aplicación de gestión de cursos académicos, se pueden diferenciar dos servicios que exponen sus respectivas APIs REST para consumo por parte de clientes.

Ambos servicios se han desarrollado utilizando Spring Boot y los starters spring-boot-starter-web (para incluir las dependencias necesarias para crear una aplicación web, como el contenedor de servlet Tomcat) y spring-boot-starter-data-jpa (que contiene las dependencias necesarias para trabajar con Java Persistence API).

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Figura 37. Dependencias iniciales de los servidores de recursos

Ya que el objetivo principal de este documento se centra en la implementación del protocolo OAuth2.0 no se entrará en gran detalle sobre la implementación del código de estos dos servicios. Únicamente, se proporcionará un esquema de la arquitectura de capas en torno a la que se organiza la aplicación, los principales patrones de acceso a datos utilizados, el modelo de datos del dominio de cada servicio y los endpoints que define cada API.

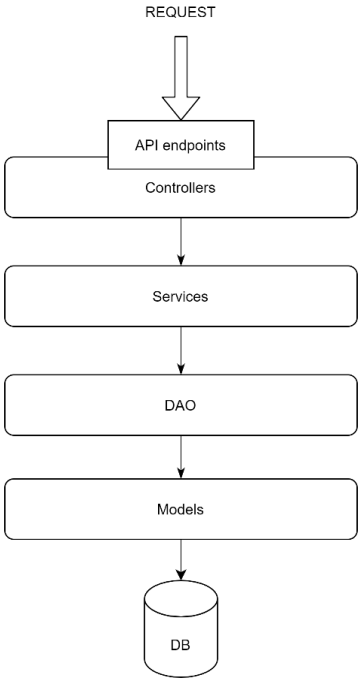

Figura 38. Arquitectura software de los servidores de recursos

La arquitectura software de ambos servicios es la que muestra la figura 38. Las clases del controlador son las encargadas de proporcionar como recursos REST las operaciones definidas en las clases de la capa de Servicio. Las clases del servicio son las encargadas de implementar la lógica de negocio y hacer de enlace con la capa de acceso a datos (DAO). La capa de acceso a datos utiliza el patrón de arquitectura DAO para encapsular toda la lógica de acceso a datos al resto de la aplicación, realizando operaciones sobre las clases definidas como modelos. Finalmente, los modelos son la representación de las entidades de la base de datos mediante objetos POJOs.

En primer lugar, se tiene el servicio de cursos. Este servicio modela la lógica en torno al dominio representado por las entidades curso y alumno. La figura 39 muestra los atributos con los que cuentan estas entidades, así como la relación existente entre ellas.

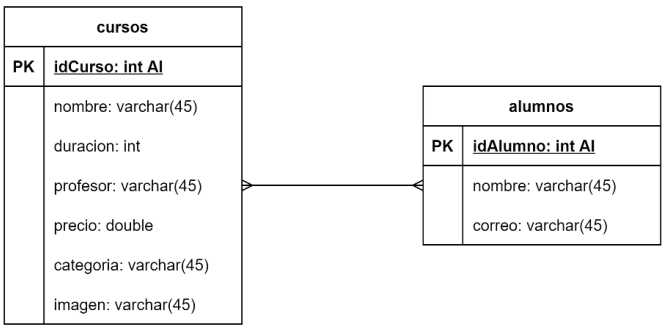

Figura 39. Diagrama ER del microservicio de cursos

A continuación, se enumeran todas las operaciones que se exponen a través de la API del servicio de usuarios junto con la ruta del endpoint para realizar dicha operación y el método HTTP. Al igual que con el servicio de cursos, únicamente se muestra la parte del endpoint que varía en función de la operación, la parte inicial del endpoint siempre será la ubicación del servicio en la red, que se verá en más detalle cuando se describa el servidor Eureka.

Las operaciones expuestas por medio de la API de este servicio son:

Operación	**Endpoint**
GET /cursos	Obtener listado de todos los cursos
GET /cursos/{idCurso}	Obtener un curso a partir de su id
GET /cursos/nombre/{nombre}	Buscar curso por nombre
GET /cursos/categoría/{categoria}	Buscar curso por categoría
GET /cursos/profesor/{profesor}	Buscar curso por profesor
POST /cursos	Crear curso
PUT /cursos	Actualizar curso
DELETE /cursos/{idCurso}	Eliminar curso
GET /alumnos	Obtener listado de todos los alumnos
GET /alumnos/{idAlumno}	Obtener un alumno a partir de su id
GET /alumnos/correo/{correo}	Buscar alumno por correo
POST /alumnos	Crear alumno
PUT /alumnos	Actualizar alumno
DELETE /alumnos/{idAlumno}	Eliminar alumno
GET /alumnos/insc/{idCurso}/{idAlumno}	Inscribir alumno en curso

Figura 40. Relación entre operaciones y endpoints de la API de cursos

En segundo lugar, se tiene el servicio de usuarios. Este servicio modela la lógica en torno al usuario, estando representado por las entidades usuario, matrícula y authority. La figura 41

muestra los atributos con los que cuentan estas entidades, así como la relación existente entre ellas.

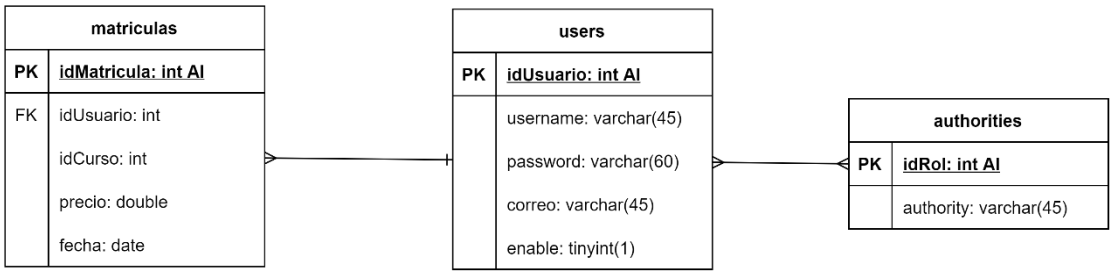

Figura 41. Diagrama ER del microservicio de usuarios

A continuación, se enumeran todas las operaciones que se exponen a través de la API del servicio de usuarios junto con la ruta del endpoint para realizar dicha operación y el método HTTP. Al igual que con el servicio de cursos, únicamente se muestra la parte del endpoint que varía en función de la operación, la parte inicial del endpoint siempre será la ubicación del servicio en la red, que se verá en más detalle cuando se describa el servidor Eureka.

Las operaciones expuestas por medio de la API de este servicio son:

Operación	**Endpoint**
GET /usuarios	Obtener listado de todos los usuarios
GET/ usuarios /{idUsuario}	Obtener un usuario a partir de su id
GET /usuarios /nombre/{nombre}	Buscar usuario por nombre
GET /usuarios /correo/{correo}	Buscar usuario por correo
POST /usuarios	Crear usuario
PUT /usuarios	Actualizar usuario
DELETE /usuarios/{idUsuario}	Eliminar usuario a partir de su id
GET /roles	Obtener listado de todos los roles
GET /roles /{idRol}	Obtener un rol a partir de su id
POST /roles	Crear rol
DELETE /roles /{ idRol }	Eliminar rol a partir de su id
GET /matriculas	Obtener listado de todas las matrículas
GET /matriculas/{ idMatricula }	Obtener una matrículaa partir de su id
GET /matriculas/curso/{idCurso}	Buscar matrícula por id de curso
POST /matriculas	Crear matricula
DELETE /matriculas/{idMatricula}	Eliminar matricula a partir de su id

Figura 42. Relación entre operaciones y endpoints de la API de usuarios

3.2.2. Servicio API Gateway

En una arquitectura de microservicios, es una buena práctica que todos los servicios que expongan una API para ser consumida por clientes externos se coloquen detrás de un API Gateway que actúe como único punto de entrada a la aplicación.

Como se pudo ver en apartados anteriores, Spring cuenta con el proyecto de Spring Cloud Gateway dentro del framework de Spring Cloud para ocuparse de este cometido. Por tanto, será necesario crear un nuevo proyecto de Spring Boot en el que el archivo pom.xml contenga las siguientes dependencias:

```
<properties>
    <java.version>15</java.version>
    <spring-cloud.version>2020.0.0-RC1</spring-cloud.version>
</properties>
```

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-gateway</artifactId>
</dependency>
```

Figura 43. Dependencias iniciales del servicio API Gateway

El servidor API Gateway, ejecutándose en el puerto 8090, se va a encargar de enrutar cualquier petición HTTP entrante hacia el servicio correcto en función de la forma de su url de destino. En el caso de que la petición contenga el path /api/cursos, la petición deberá enrutarse hacia el servicio de cursos y en el caso de que la petición contenga el path /api/usuarios, la petición deberá enrutarse hacia el servicio de usuarios.

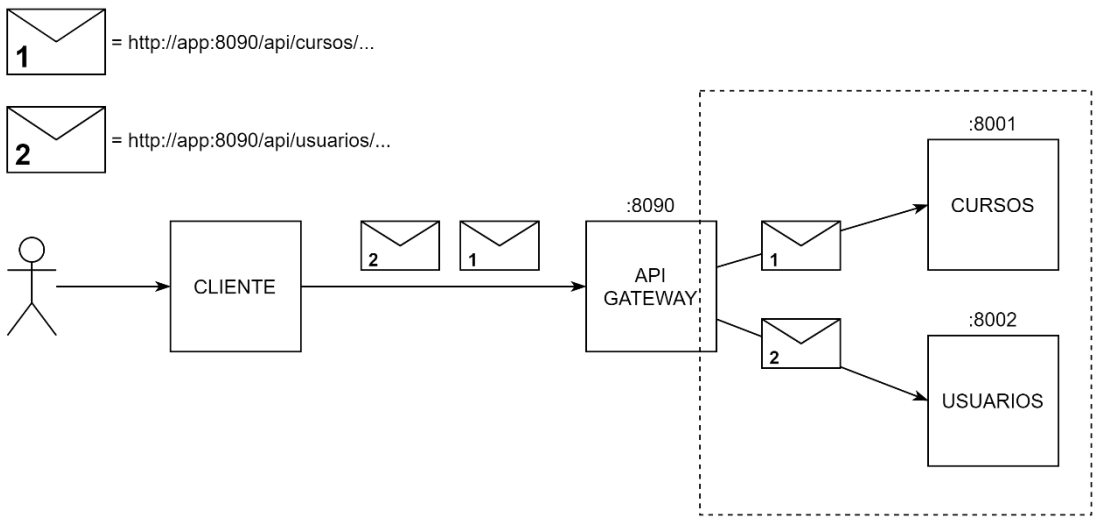

Figura 44. Flujo de comunicación resultante de implementar el servicio API Gateway

Para conseguir esto, la configuración que hay que definir dentro del archivo application.properties es la siguiente:

```
server.port=8090

spring.cloud.gateway.routes[0].id=servicio-cursos-alumnos
spring.cloud.gateway.routes[0].uri=http://localhost:8001
spring.cloud.gateway.routes[0].predicates[0]=Path=/api/cursos/**
spring.cloud.gateway.routes[0].filters[0]=StripPrefix=2

spring.cloud.gateway.routes[1].id=servicio-usuarios-matriculas
spring.cloud.gateway.routes[1].uri= http://localhost:8002
spring.cloud.gateway.routes[1].predicates[0]= Path=/api/usuarios/**
spring.cloud.gateway.routes[1].filters[0]=StripPrefix=2
```

Figura 45. Configuración de application.properties en el servicio API Gateway

Para definir una ruta en Spring Gateway se utiliza la propiedad spring.cloud.gateway.routes. Esta propiedad utiliza un formato de array para permitir la definición de múltiples rutas. A partir de ahí se definen las propiedades de la ruta tal y como se vio en la sección 2.3.4.1 de este documento.

En primer lugar, se especifica el id de la ruta, un valor que permita fácilmente su identificación, por lo que es buena práctica utilizar el valor definido mediante la propiedad application.name de cada servicio. La segunda propiedad que se configura debe ser la uri en la que se está ejecutando el microservicio de destino. A continuación, se configura el predicado o los predicados, es decir, la función que va a analizar la petición entrante en busca de una coincidencia para hacer el direccionamiento hacia la uri especificada. En este caso, se utiliza la factoría Path Route, que usa como parámetro de búsqueda un patrón.

Por último, es necesario definir el filtro StripPrefix para que API Gateway elimine los dos primeros elementos de la ruta, es decir, "api" y "cursos" en el primer caso, y "api" y "usuarios" en el segundo caso. De no aplicar este filtro, la petición se enrutaría correctamente hacia los microservicios, pero estos devolverían un error ya que no exponen ningún endpoint que comience con las palabras /api/cursos ni /api/usuarios.

3.2.3. Service Discovery con servidor Eureka

En esta sección se describe qué es Eureka Server y cómo se configura un servicio de descubrimiento gracias a este proyecto.

El objetivo principal del descubrimiento de servicios es disponer de una arquitectura en la que los servicios indiquen dónde se encuentran físicamente en lugar de tener que configurar manualmente su ubicación.

En el escenario actual, cuando se definieron los servicios de cursos y usuarios hubo que especificar el puerto en el que se están ejecutando. Por lo tanto, si una aplicación cliente (o el propio API Gateway) quiere enviar una petición para consumir su API, necesitará conocer su dirección y su puerto. Esto trae el inconveniente de que, si alguno de los servicios tiene que escalarse a más replicas, será necesario actualizar el cliente para que sea consciente de la ubicación de estas nuevas réplicas y no se limite a comunicarse exclusivamente con la primera de ellas; del mismo modo que si simplemente cambiase el puerto de ejecución, también habría que modificar los datos de la aplicación cliente, lo que no es un comportamiento deseado en una arquitectura de microservicios.

El proceso que se sigue con Eureka Server es el siguiente:

1. Cada vez que se pone en marcha una instancia de microservicio este se registra en uno de los servidores de Eureka. De esta forma, Eureka tiene conocimiento de la dirección y el puerto en el que se está ejecutando el microservicio. Del mismo modo, si un servicio se detuviese, se eliminaría de Eureka de forma automática sin tener que cambiar ningún tipo de configuración.

2. De forma periódica, cada instancia de microservicio envía un mensaje *heartbeat* al servidor Eureka, indicándole que la instancia de microservicio está bien y está lista para recibir peticiones.

3. Los clientes utilizan una biblioteca de clientes que solicita regularmente al servicio Eureka información sobre los servicios disponibles.

4. Cuando el cliente necesita enviar una solicitud a otro microservicio, ya tiene una lista de instancias disponibles en su biblioteca de clientes y puede elegir una de ellas sin preguntar al servidor de descubrimiento. Normalmente, las instancias disponibles se eligen de forma rotatoria; es decir, se llaman una tras otra antes de volver a llamar a la primera.

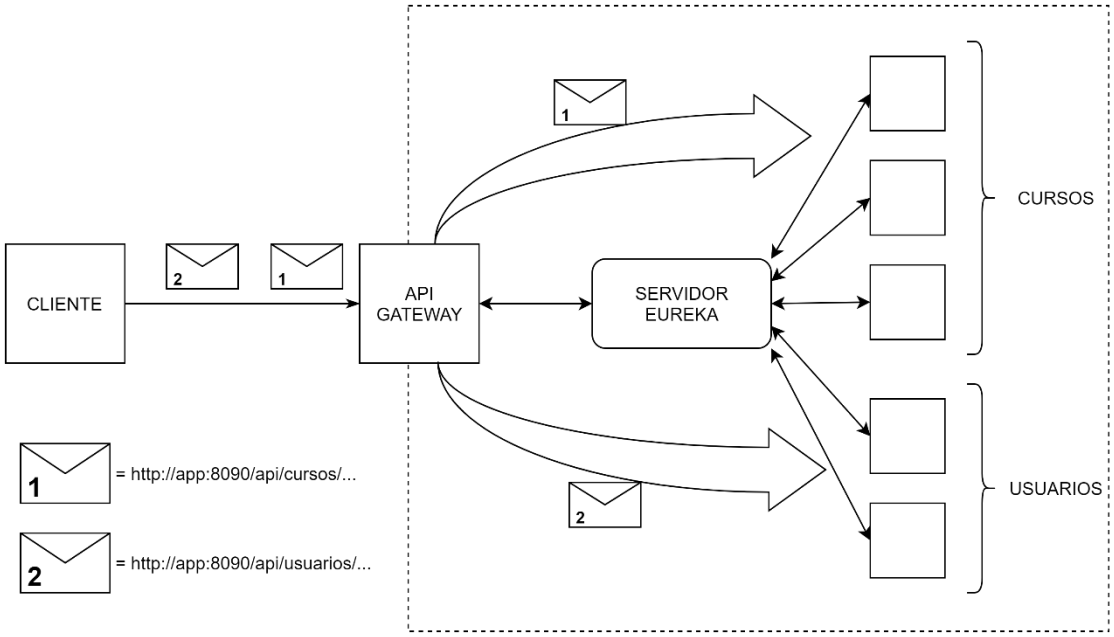

Figura 46. Flujo de comunicación resultante de implementar el servidor Eureka

Para crear el servidor de Eureka es necesario iniciar un nuevo proyecto de Spring Boot con el starter de WebMVC y el de Eureka Server (con group id org.springframework.cloud y artifact id spring-cloud-starter-netflix-eureka-server).

```
<properties>
    <java.version>15</java.version>
    <spring-cloud.version>2020.0.0-RC1</spring-cloud.version>
</properties>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-server</artifactId>
</dependency>
```

Figura 47. Dependencias del servidor Eureka

Después, se especifica la siguiente configuración dentro del archivo application.properties:

```
spring.application.name=eureka-server
server.port=8761
eureka.client.register-with-eureka=false
eureka.client.service-url.defaultZone=http://localhost:8761/eureka
```

Figura 48. Configuración de application.properties en el servidor de Eureka

Por último, en el archivo en el que se define la aplicación del proyecto (en este caso, EurekaServerApplication) se introduce la anotación @EnableEurekaServer

Figura 49. Introducción de la anotación @EnableEurekaServer

El siguiente paso que hay que realizar es configurar el resto de los microservicios como clientes de Eureka, de tal modo que cuando arranquen se registren en el servidor de Eureka que se acaba de definir. Para ello hay que añadir la siguiente dependencia al archivo pom.xml (además de definir la versión de Spring Cloud en el caso de los servidores de recursos):

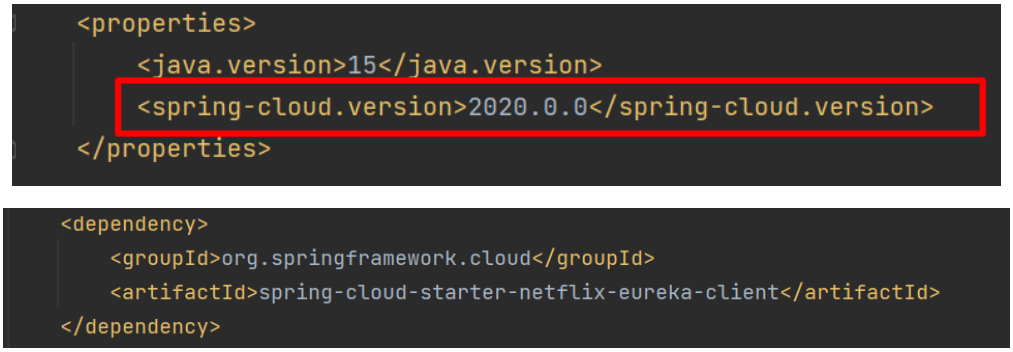

Figura 50. Dependencias para los clientes de Eureka

Dentro del archivo application.properties, será necesario introducir la siguiente línea con la dirección en la que se está ejecutando el servidor de eureka:

eureka.client.service-url.defaultZone=http://localhost:8761/eureka

A diferencia de lo que había que hacer en el archivo principal de la aplicación del servidor de Eureka, en esta ocasión no es necesario añadir ninguna anotación. Al tener spring-cloud-starter-netflix-eureka-client en el classpath, las aplicaciones se registran automáticamente en el servidor Eureka que se especifique en la configuración de application.properties.

El nombre con el que aparecerán en eureka será aquel que se especifique en la propiedad spring.application.name de su correspondiente archivo application.properties.

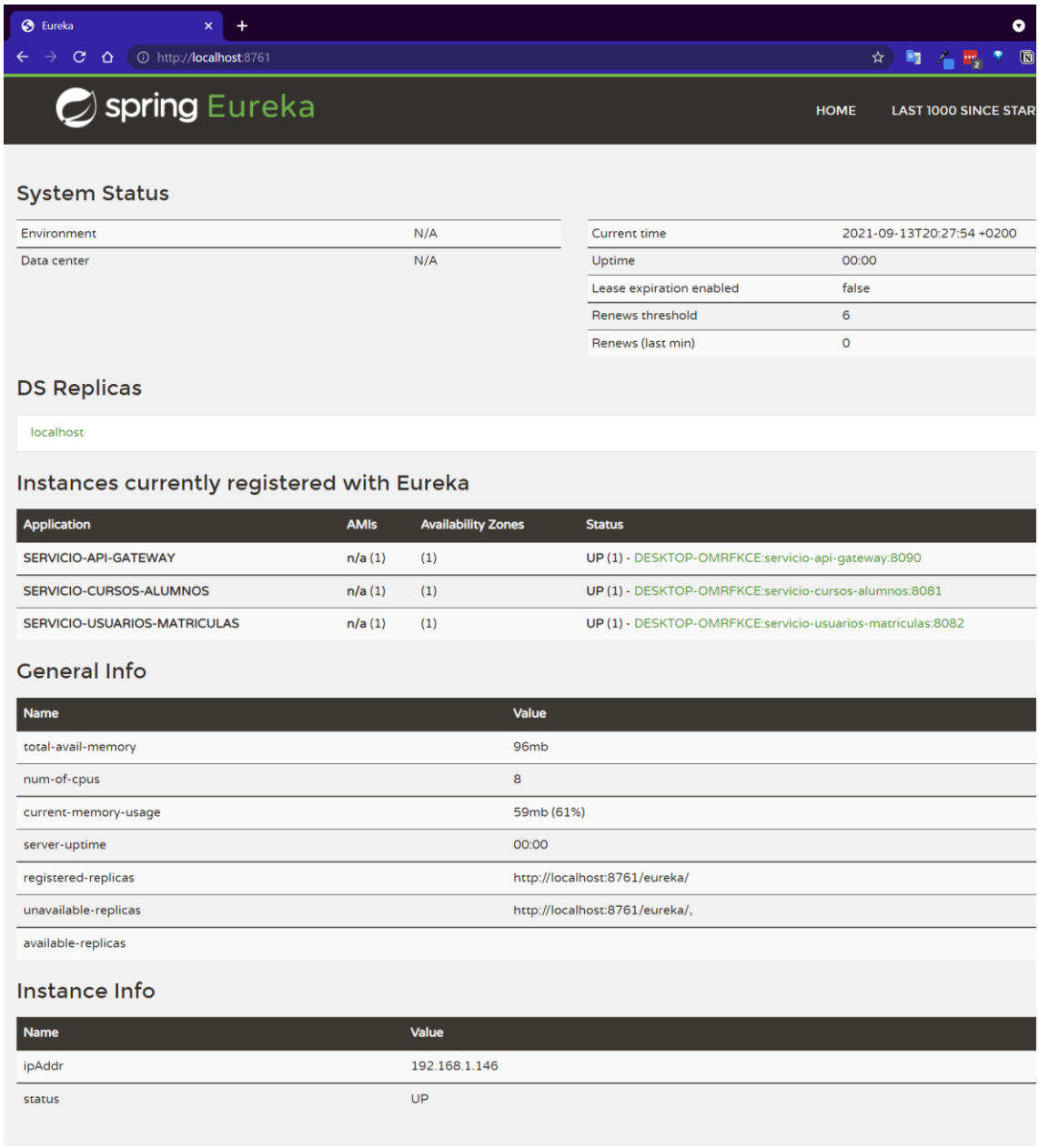

Figura 51. Interfaz de Eureka Server con los microservicios en ejecución

3.2.4. Implementación del protocolo OAuth 2.0

Según se vio en la introducción de este capítulo, por la naturaleza de la aplicación el tipo de concesión más apropiado es el Authorization Code. De cara a aplicar este flujo de concesión de autorización, los diferentes roles que define la especificación de OAuth 2.0 van a estar representados de la siguiente manera:

- Servidor de recursos: microservicio de cursos
- Servidor de autorización: servidor de Keycloak
- Cliente: aplicación Postman.

3.2.4.1. Servidor de autorización con Keycloak

En la sección 2.3.5 se pudo profundizar en detalle sobre el estado de arte de los servidores de autorización de Spring Framework, viendo como actualmente convivían una versión antigua en la fase final de su ciclo de vida y una versión más moderna en fase experimental. Por tanto, a la hora de decir que tecnología utilizar para llevar a la cabo la función de servidor de autorización se ha decidido optar por Keycloak.

Keycloak es una solución Open Source para la gestión de la identidad y el acceso que soporta OpenId Connect y todos los flujos de autorización definidos en OAuth 2.0. Gracias a él, los usuarios van a poder realizar su autenticación en la aplicación cliente y, una vez identificados, podrán recibir un token de acceso y un token de actualización para consumir los recursos de cursos y usuarios.

Keycloak puede utilizarse como un servidor independiente corriendo en una máquina, como un contenedor de Docker. Soporta inicio de sesión único (Single Sign-On) y Social login (lo que permite a los usuarios iniciar sesión mediante sus cuentas de Facebook, Google, Twitter o GitHub). Además, en caso de integrarse en una aplicación donde ya exista una base de datos de usuarios, puede conectarse a dicha base de datos y utilizarla como su fuente principal de información de usuarios.

En cuanto a su configuración, Keycloak ofrece una cómoda interfaz de usuario que permite la configuración de las aplicaciones cliente, credenciales, reglas y cualquier flujo de OAuth que se necesite utilizar, además de roles y permisos de usuario.

En el caso de la aplicación desarrollado, se ha optado por descargar la versión de servidor de Keycloak para ejecutarlo en la máquina local. Para ello, hay que acceder a su página oficial (https://www.keycloak.org/downloads), descargar el ZIP con la distribución del servidor independiente y descomprimir este archivo en la ruta que más le convenga al usuario.

Para iniciar el servidor, será necesario ejecutar el comando standalone.bat que se puede encontrar en el directorio bin de Keycloak. Este comando arrancará el servidor en el puerto 8080 por defecto. Si se quiere cambiar este puerto es necesario llamar al archivo .bat de la siguiente forma:

bin/standalone.bat -Djboss.socket.binding.port-offset={offset}

Donde offset es un numero a sumar al puerto 8080.

Al introducir la url http://localhost/8080 en el navegador, se muestra la interfaz de usuario de Keycloak en su pantalla de bienvenida, donde lo primero que se pedirá al usuario es que configure su cuenta principal de administrador. Hecho esto, aparecerá el siguiente menú principal:

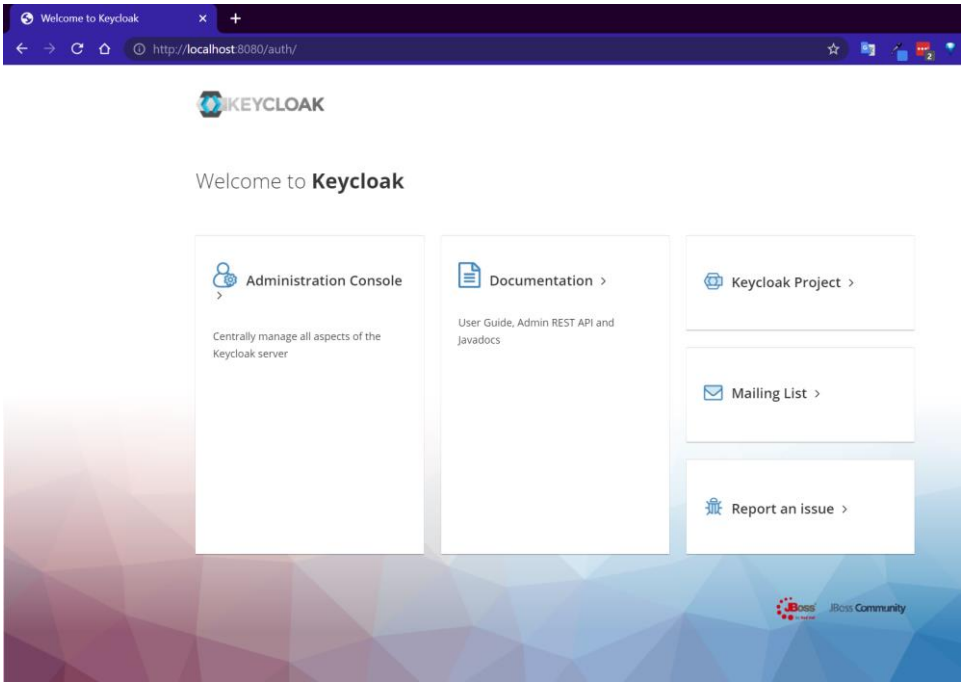

Figura 52. Menú principal de Keycloak

Si se pulsa la opción de Administration Console, se solicitarán las credenciales del usuario administrador y, tras validar el acceso, se accederá a la consola de administración. Está mostrará que el usuario se encuentra dentro dominio master, el cual se creo como opción por defecto al crear el servidor.

Un dominio o realm define un contexto aislado de otros dominios donde se permite la gestión de usuarios, credenciales, roles y grupos. Estos dominios solo pueden gestionar y autenticar a los usuarios que controlan.

El primer paso necesario para configurar la aplicación de cursos es crear su propio dominio, para lo cual hay que situar el botón sobre el nombre "Master" y elegir la opción "Add realm" que aparece en el menú desplegable. A continuación, se introduce el nombre de la aplicación, Appcursos, y se procede con la configuración.

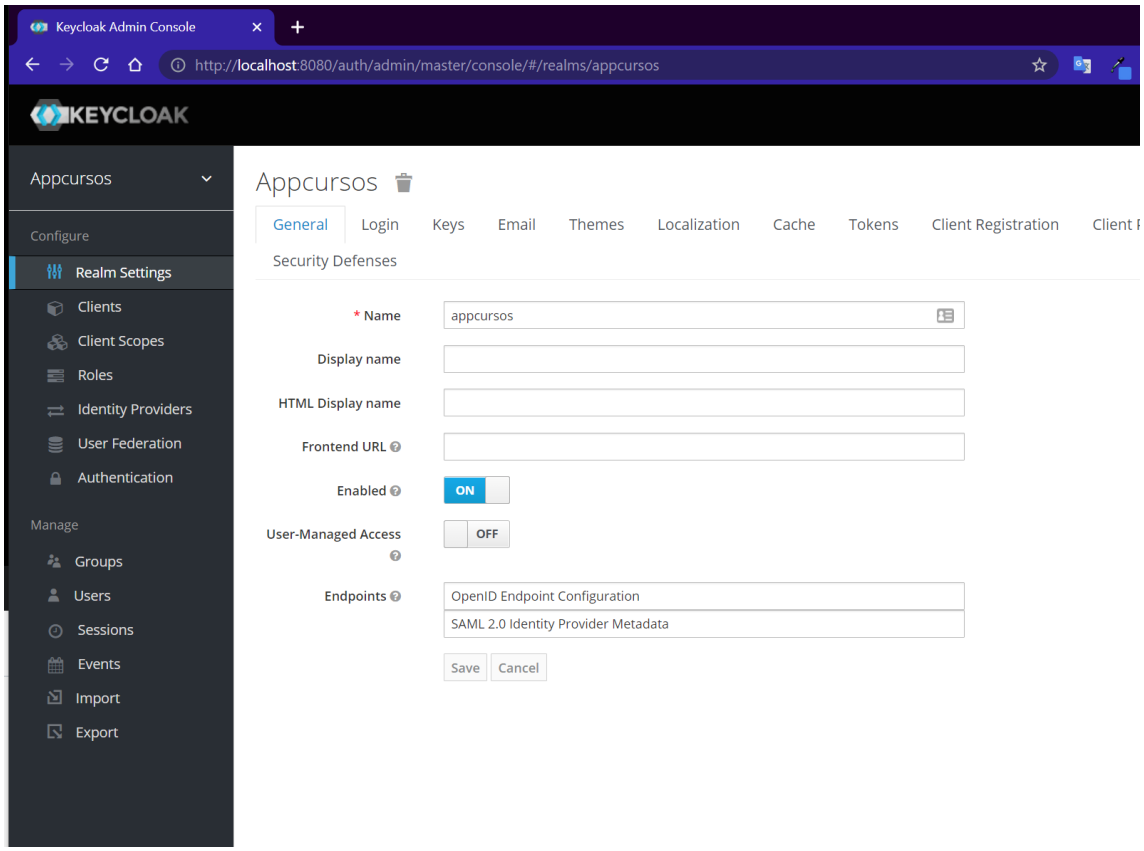

Figura 53. Menú del dominio Appcursos en Keycloak

Para registrar una aplicación cliente en Keycloak, hay que acceder al menú "Clients" del menú lateral de la consola de administración. En este menú se muestra una tabla con todas las aplicaciones registradas y un botón para crear una nueva. Al pulsar este botón, aparecerá un formulario que habrá que completar tal y como se muestra en la imagen de la figura 54:

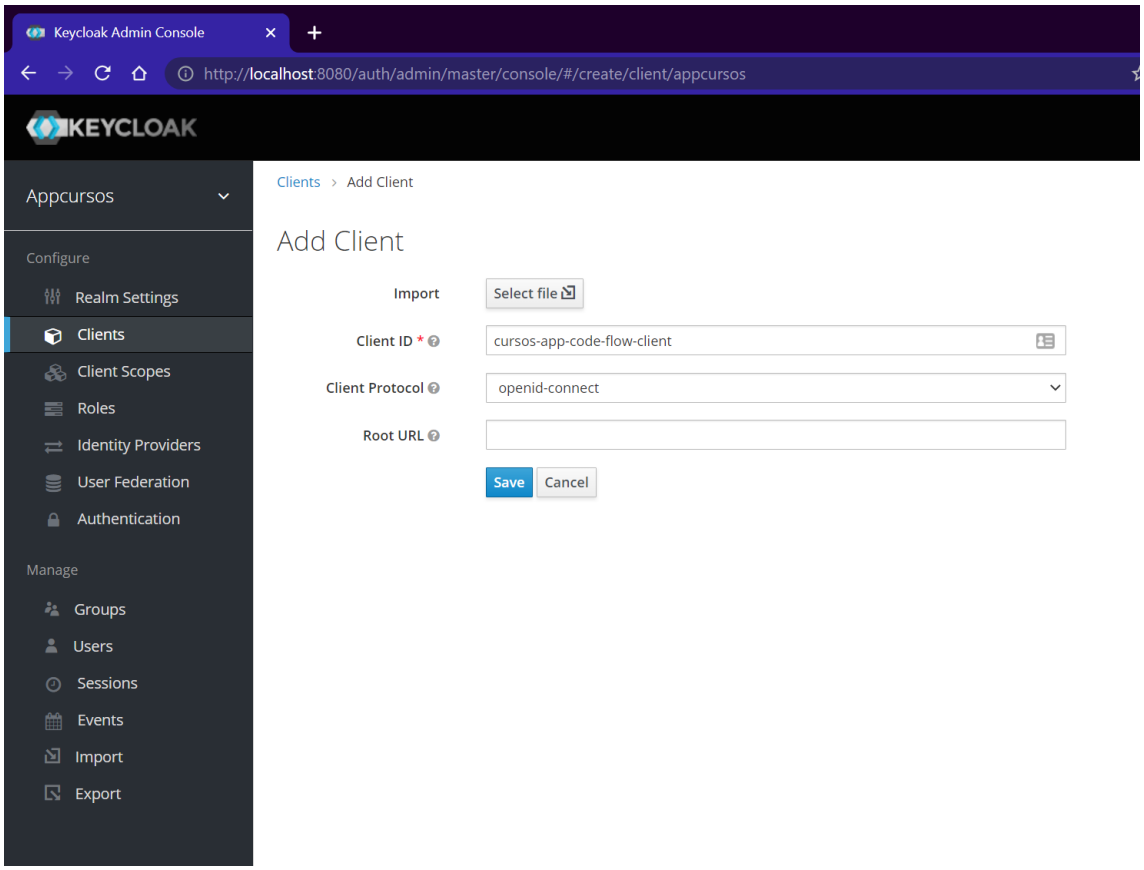

Figura 54. Registro de aplicación cliente en Keycloak

Al guardar esta configuración, se producirá una redirección a la vista de detalles del cliente que se acaba de registrar. Por defecto, esta aplicación cliente ya se ha configurado para utilizar el tipo de concesión Authorization Code. Esto se hace por medio de la opción Standard Flow Enabled, que ya aparece como seleccionada.

Los únicos campos que hay que modificar son:

- "Access Type", donde habrá que elegir el valor "confidential".
- "Valid Redirect URIs", donde habrá que indicar la url de retorno de la aplicación cliente a la que el servidor de Keycloak enviará primero el código de autorización y, después, el token de acceso. Para los propósitos de esta demostración, se utiliza la url http://localhost:8083/callback.

Para terminar la configuración de la app cliente, hay que ir a la pestaña "Credentials", donde se encuentra la clave secreta que Keycloak ha generado para la aplicación. Esta clave secreta junto con el client ID que se definió al registrar la aplicación serán las credenciales que tendrá que enviar la aplicación cliente durante el flujo de concesión de autorización para identificarse en Keycloak.

Ahora que ya se tiene registrada la aplicación en Keycloak, es el momento de definir los roles y los usuarios. Para definir los roles, hay que pulsar en la opción "Roles" del menú lateral de la consola de administración. Al igual que ocurría en la interfaz del proceso de crear un cliente, se muestra una tabla con los roles existentes y un botón para definir un rol nuevo, el cual conduce

a un formulario que solicita el nombre del rol y una descripción. Simplemente hay que rellenar el nombre del formulario (la descripción es opcional) con el valor deseado y después pulsar en el botón de guardar.

La figura 55 representa la tabla donde se definen los roles de usuario que van a existir en la aplicación junto con los permisos que va a tener cada uno de cara al acceso a los recursos.

Rol	Cursos	Alumnos
Admin	ALL	ALL
Profesor	GET	GET
Alumno	GET	-

Figura 55. Relación de roles y permisos

Para crear un nuevo usuario en Keycloak, hay que pulsar en la opción "Users" del menú lateral de la consola de administración y en la pantalla que aparece hay que pulsar en el botón "add user". En este nuevo formulario (figura 55) tan solo es necesario llenar el campo username y marcar la casilla "Email verified" (para que no se solicité al usuario que verifique su correo electrónico).

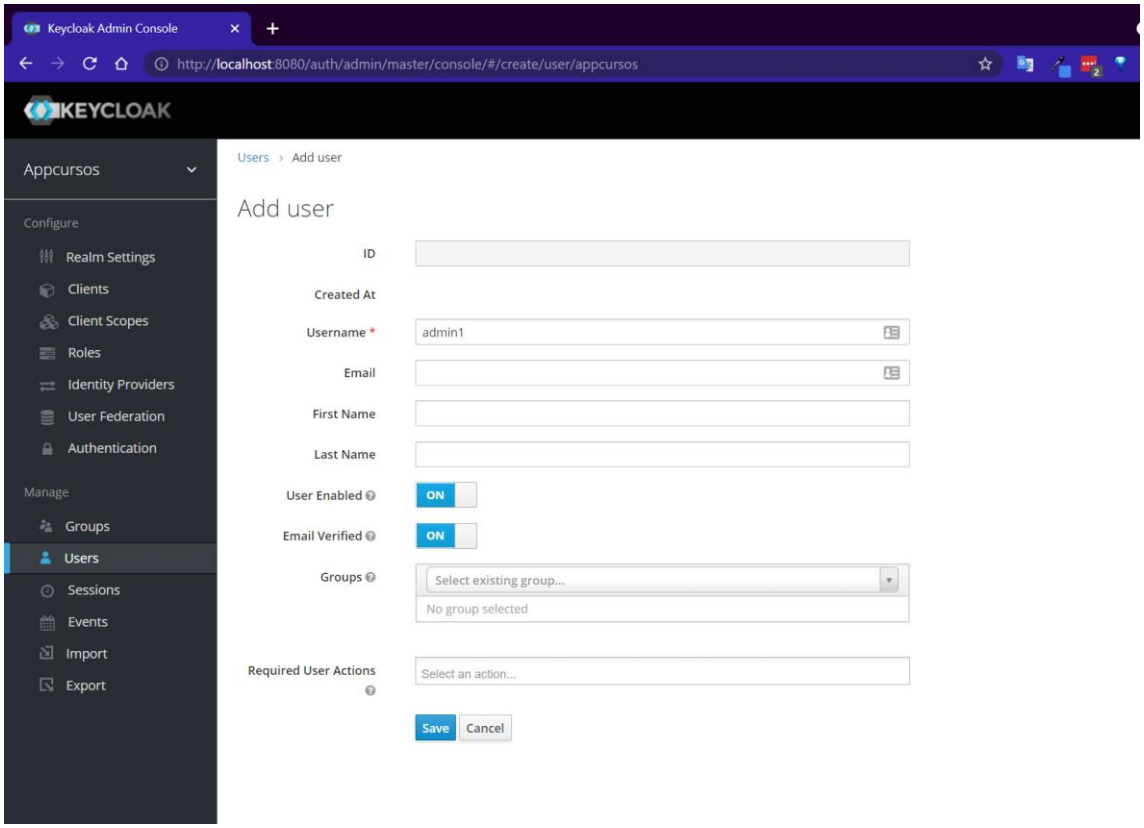

Figura 56. Definición de usuarios en Keycloak

Al pulsar en guardar, se creará el usuario en el sistema y se cargará una vista de detalles. El siguiente paso que hay que realizar es definir la contraseña del usuario. Esto se realiza desde la pestaña de "Credentials", como muestra la figura 57.

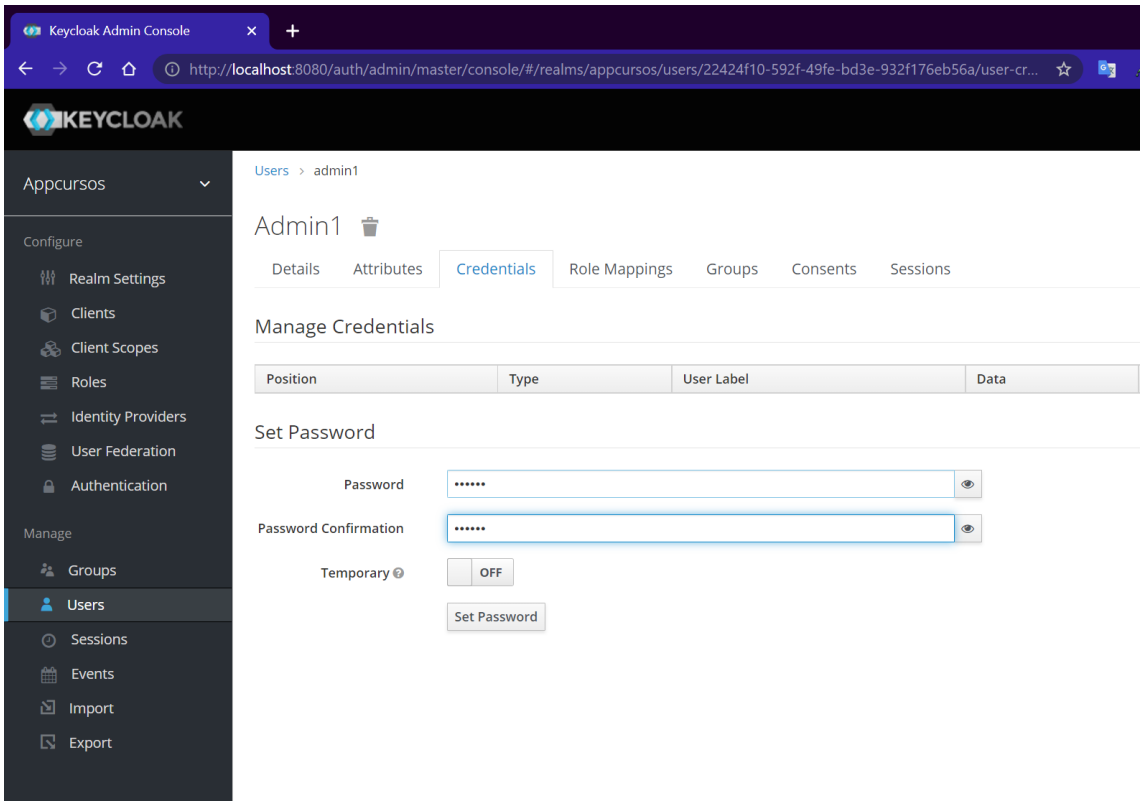

Figura 57. Definición de las credenciales de un usuario en Keycloak

Para finalizar la configuración, solo quedaría asignar al usuario recién creado el rol apropiado. Esto se hace pulsando en la pestaña "Role Mappings", eligiendo el rol de la lista de roles disponible y pulsando en el botón "Add selected".

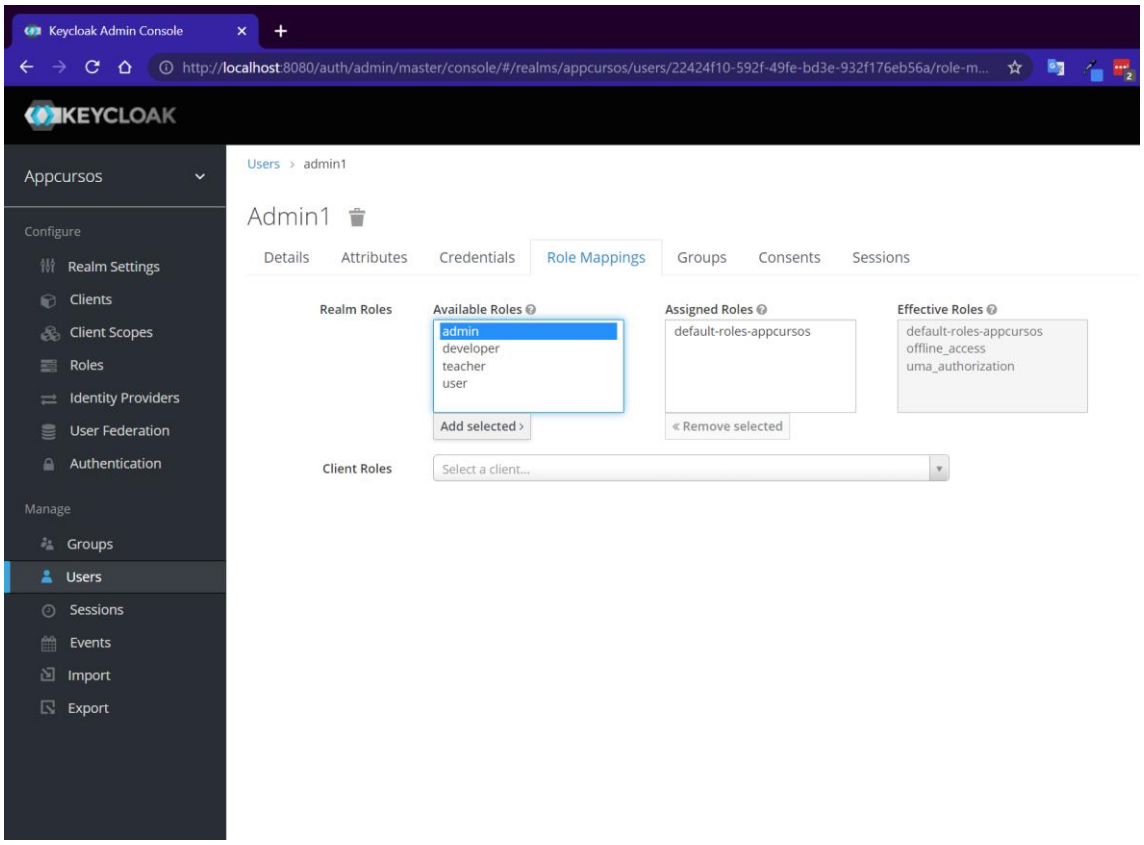

Figura 58. Asignación de roles a un usuario en Keycloak

3.2.4.2. Spring Security OAuth Resource Server

Llegado este punto, solo queda una configuración más por realizar en la aplicación: proteger los servidores de recursos. El primer paso para cumplir con este objetivo es añadir la siguiente dependencia al archivo pom.xml:

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
```

Figura 59. Dependencias para establecer un API como servidor protegido de recursos

Para configurar el servidor de recursos para que valide los access token de las peticiones entrantes contra el servidor de autorización, es necesario añadir la propiedad spring.security.oauth2.resourceserver.jwt.issuer-uri al archivo application.properties y asignarla el valor

http://localhost:8080/auth/realms/{realm}/protocol/openid-connect/certs, donde realm tendrá como valor el nombre del dominio de Keycloak que se desee utilizar. En este caso, appcursos.

Spring Security utilizará esta url para descubrir las claves públicas del servidor de autorización de Keycloak y validar la firma de los tokens de acceso.

En cuanto al código fuente de la aplicación es necesario definir las dos clases siguientes, que se han decidido ubicar dentro de un nuevo paquete llamado security:

- Clase **KeycloakRoleConverter:** extiende de la clase Converter para definir un mecanismo que procese el token portado en la petición para extraer el rol de toda la información que contenga según el formato de Keycloak.

KeycloakRoleConverter.java

```java
package es.uah.cursosAlumnosEureka.security;

import org.springframework.core.convert.converter.Converter;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.oauth2.jwt.Jwt;

import java.util.ArrayList;
import java.util.Collection;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class KeycloakRoleConverter implements Converter<Jwt, Collection<GrantedAuthority>> {

    @Override
    public Collection<GrantedAuthority> convert(final Jwt jwt) {

        // los roles se encuentran dentro de la propiedad realm_access del JWT
        Map<String, Object> realmAccess = (Map<String, Object>) jwt.getClaims().get("realm_access");

        if (realmAccess == null || realmAccess.isEmpty()) {
            return new ArrayList<>();
        }

        Collection<GrantedAuthority> returnValue = ((List<String>) realmAccess.get("roles"))
                .stream().map(roleName -> "ROLE_" + roleName)
                .map(SimpleGrantedAuthority::new)
                .collect(Collectors.toList());

        return returnValue;
    }

}
```

Figura 60. Código fuente de la clase KeycloakRoleConverter

- Clase **WebSecurityConfig:** extiende de la clase WebSecurityConfigurerAdapter para configurar Spring Security a la hora de procesar las peticiones entrantes. Dentro del método configure, primero se instancia un objeto de la clase JwtAuthenticationConverter y se le dice que tiene que utilizar un objeto de nuestra clase KeycloakRoleConverter a la hora de procesar los tokens en formato JWT que traerán consigo las peticiones. Después se especifican los filtros necesarios para controlar el acceso en función del rol encontrado en dicho token. Primero se especifica que todas las rutas que utilicen un método GET y contengan un path del endpoint cursos podrán ser accedidas por los tres roles: admin, teacher y user. Después, se especifica la regla por la que se restringe el acceso a las rutas que utilicen un método GET y contengan un path del endpoint alumnos a los roles admin y teacher, excluyendo al rol user. Finalmente, se especifica que cualquier otra petición con una ruta o método distinto a las anteriores solo podrá ser accedida por usuarios con el rol admin.

WebSecurityConfig.java

```java
package es.uah.cursosAlumnosEureka.security;

import org.springframework.http.HttpMethod;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.oauth2.server.resource.authentication.JwtAuthenticationConverter;

@EnableWebSecurity
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        JwtAuthenticationConverter jwtAuthenticationConverter = new JwtAuthenticationConverter();
        jwtAuthenticationConverter.setJwtGrantedAuthoritiesConverter(new KeycloakRoleConverter()); // delegate

        http.authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
                .antMatchers(HttpMethod.GET, ...antPatterns: "/cursos/**") ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
                .hasAnyRole( ...roles: "admin", "teacher", "user") ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
                .antMatchers(HttpMethod.GET, ...antPatterns: "/alumnos/**") ExpressionUrlAuthorizationConfigurer<...>.AuthorizedUrl
                .hasAnyRole( ...roles: "admin", "teacher") ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
                .anyRequest().hasRole("admin")
                .and() HttpSecurity
                .oauth2ResourceServer() OAuth2ResourceServerConfigurer<HttpSecurity>
                .jwt() OAuth2ResourceServerConfigurer<...>.JwtConfigurer
                .jwtAuthenticationConverter(jwtAuthenticationConverter); // apply jwt converter;

        http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
    }
}
```

Figura 61. Código fuente de la clase WebSecurityConfig

3.2.4.3. Consumiendo los recursos desde Postman

En este apartado se incluyen una serie de imágenes con el objetivo de mostrar el funcionamiento de la aplicación. El rol de cliente va a estar representado por el software de escritorio Postman, el cual ofrece una interfaz para configurar y enviar peticiones, lo que la convierte en una herramienta fundamental a la hora de diseñar y probar APIs.

Las peticiones que se van a realizar son las siguientes:

1. Petición para obtener el código de autorización.

2. Petición para intercambiar dicho código de autorización por un token de acceso.

3. Peticiones de prueba al servicio de cursos empleando el token de acceso.

Para poder comprobar que se han configurado correctamente los controles de autorización, estos 3 pasos se van a repetir para cada uno de los tres roles definidos en el sistema, con los usuarios que se crearon durante la configuración de Keycloak.

El primer paso que habría que realizar sería realizar la petición para obtener el código de autorización. En Keycloak, esto se hace accediendo a /auth/realms/{realm}/protocol/openid-connect/auth y enviando los parámetros que muestra la siguiente imagen:

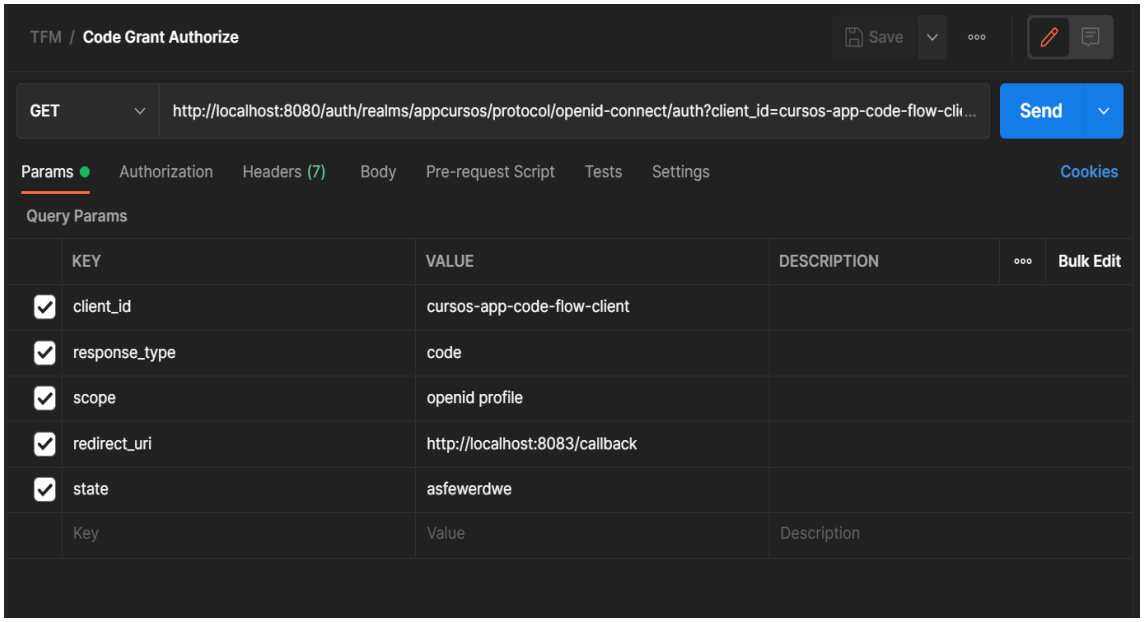

Figura 62. Definición de la petición al authorization endpoint mediante Postman

De esta url, client_id representa el identificador de la aplicación con el que se registró en Keycloak, response_type indica al servidor de autorización que queremos iniciar el tipo de concesión Authorization Code y estamos esperando un código de autorización como respuesta, scope indica los claims de información que debe contener el token (para los propósitos de esta demo, se incluyen los más básicos relacionados con la información del usuario), redirect_uri indica la url a la que el servidor de Keycloak tiene que devolver el código de autorización, y state es un parámetro aleatorio utilizado para prevenir los ataques CSRF en caso de que un usuario o software malicioso intercepte la petición.

Esta petición no hay que enviarla desde Postman, ya que como respuesta va a devolver una página HTML con un formulario para que el usuario se autentique con sus credenciales. Postman se utiliza por la sencillez con la que permite definir los parámetros del método HTTP GET. Si se copia la petición y se introduce en el formulario, aparecerá la siguiente pantalla:

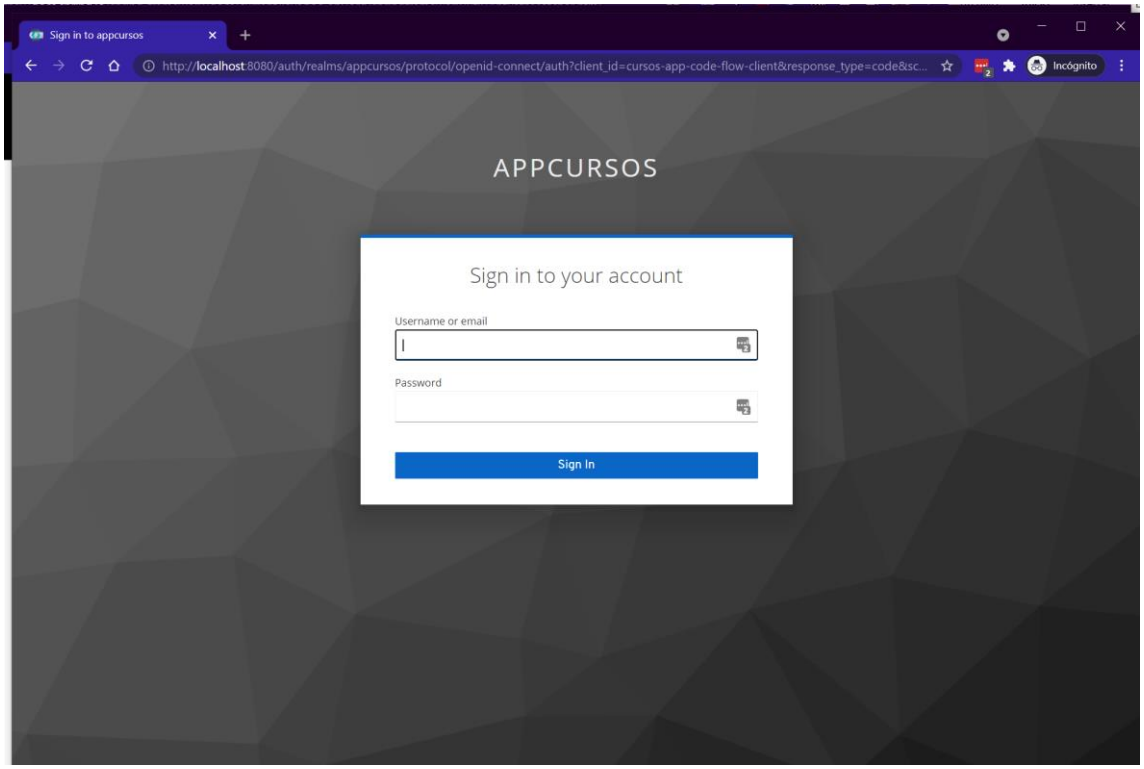

Figura 63. Formulario de autenticación previo a la concesión del código de autorización

Desde aquí habría que introducir las credenciales del usuario del rol que se quiera probar. Por ejemplo, el usuario con rol user. Tras pulsar en el botón para iniciar sesión, se obtendrá una respuesta como esta:

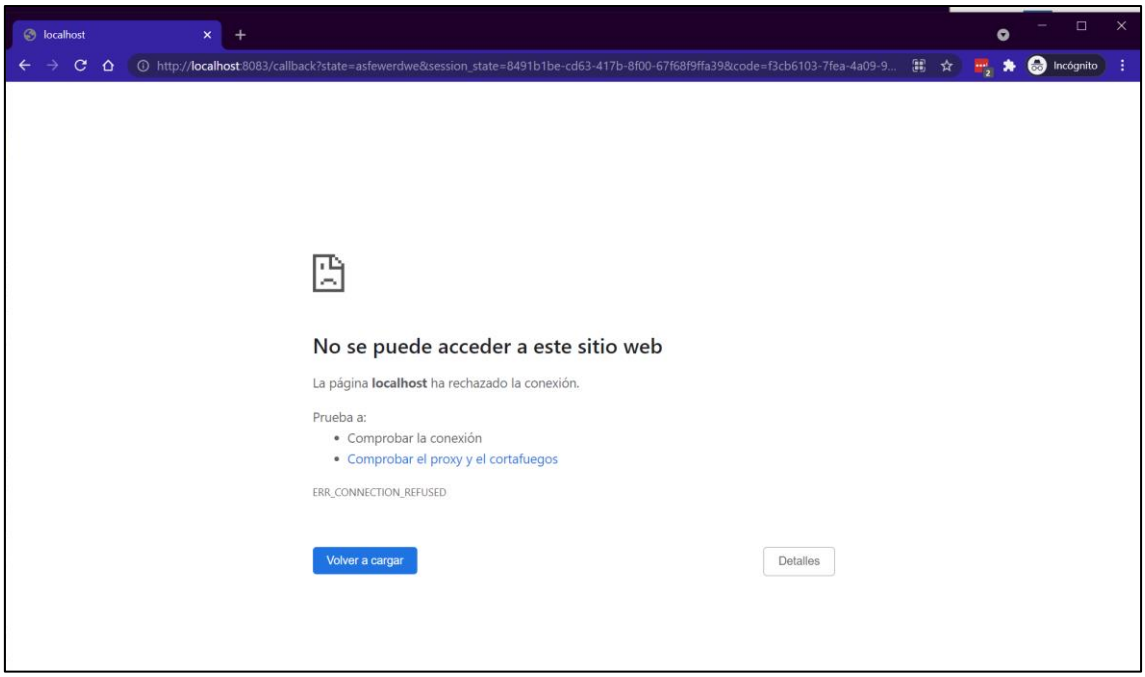

Figura 64. Respuesta de Keycloack con el código de autorización para el usuario autenticado

A pesar de que el navegador muestre un error indicando que no se ha podido acceder al sitio web, sabemos que la petición ha tenido éxito analizando la url de la ventana actual. El error del navegador únicamente se debe a que no hay una aplicación cliente como tal esperando la respuesta, y por tanto no se puede servir ningún contenido al acceder a http://localhost:8083/callback.

El aspecto que tiene la url es algo como esto: http://localhost:8083/callback?state=asfewerdwe&session_state=8491b1be-cd63-417b-8f00-67f68f9ffa39&**code=f3cb6103-7fea-4a09-98f9-d4d12af3fcf9.8491b1be-cd63-417b-8f00-67f68f9ffa39.eecd9121-8c8d-463f-aead-d275b50d7d79**

El siguiente paso que hay que realizar es copiar el parámetro "code" de la url y utilizarlo en la siguiente petición, que es la que se muestra a continuación:

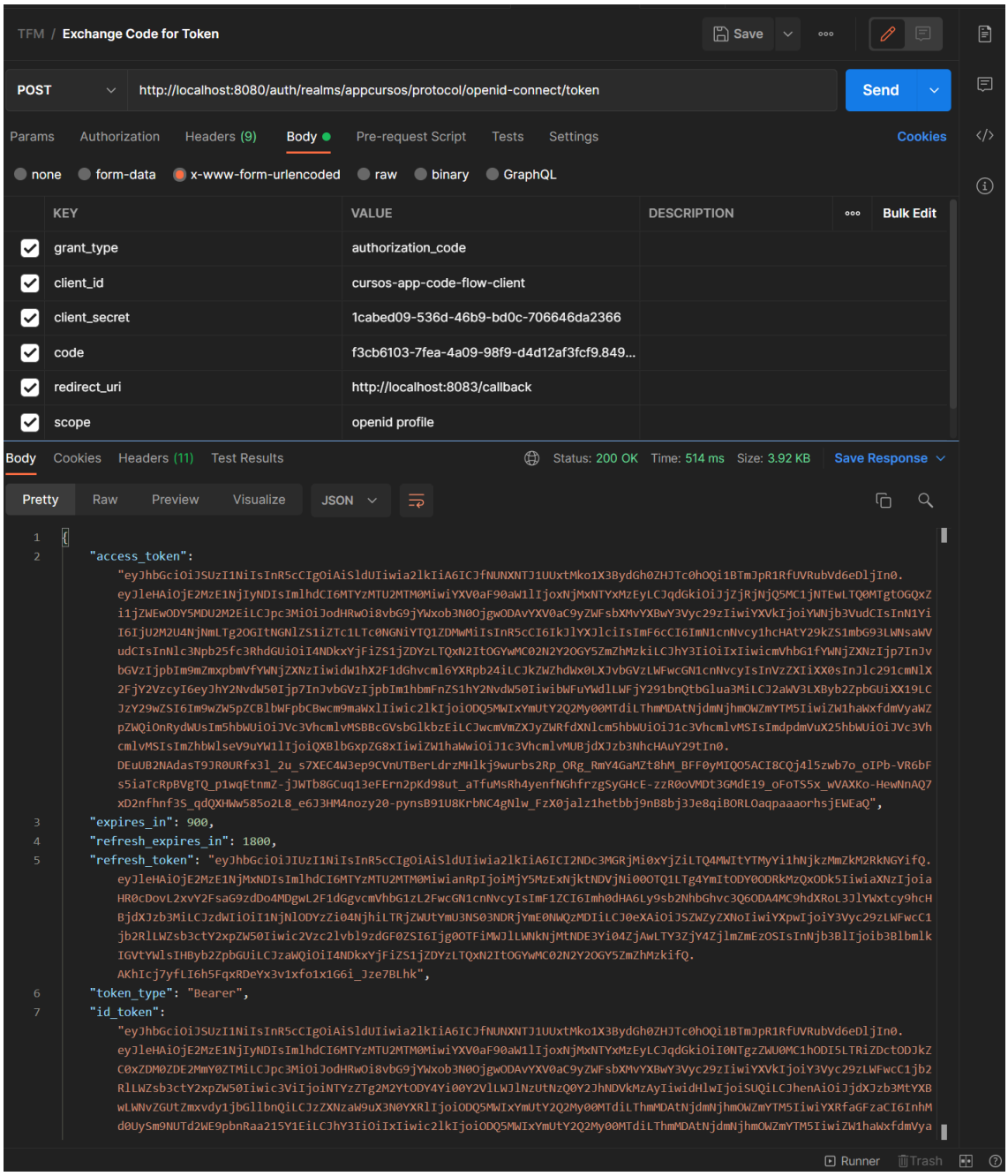

Figura 65. Petición al token endpoint de Keycloak mediante Postman

Enviando una petición POST al endpoint /auth/realms/{realm}/protocol/openid-connect/token, es decir, el token endpoint de Keycloak se intercambia el código de autorización por el token de acceso y una serie de elementos más, como se puede ver en la imagen.

El significado de los elementos que se incluyen en el cuerpo de la petición es el siguiente. En primer lugar, se especifica que grant_type debe tener el valor authorization_code para avisar del tipo de concesión que se va a utilizar para conseguir el token de acceso. Después, es necesario enviar las credenciales con los que la aplicación cliente se registró en el servidor de autorización. Como estamos ante un tipo de cliente confidencial, en este caso se enviaría tanto el id del cliente, como su código secreto. El siguiente parámetro de la petición, "code", es el código de

autorización que se obtuvo en el paso anterior. Por último, hay que volver indicar la url a la que se quiere enviar la respuesta y la información que debe contener el token.

Si copiamos el access token y lo llevamos al depurador que ofrece jwt.io, podemos ver su contenido y en él, comprobar que efectivamente el usuario con el que nos hemos autenticado tiene el rol de user:

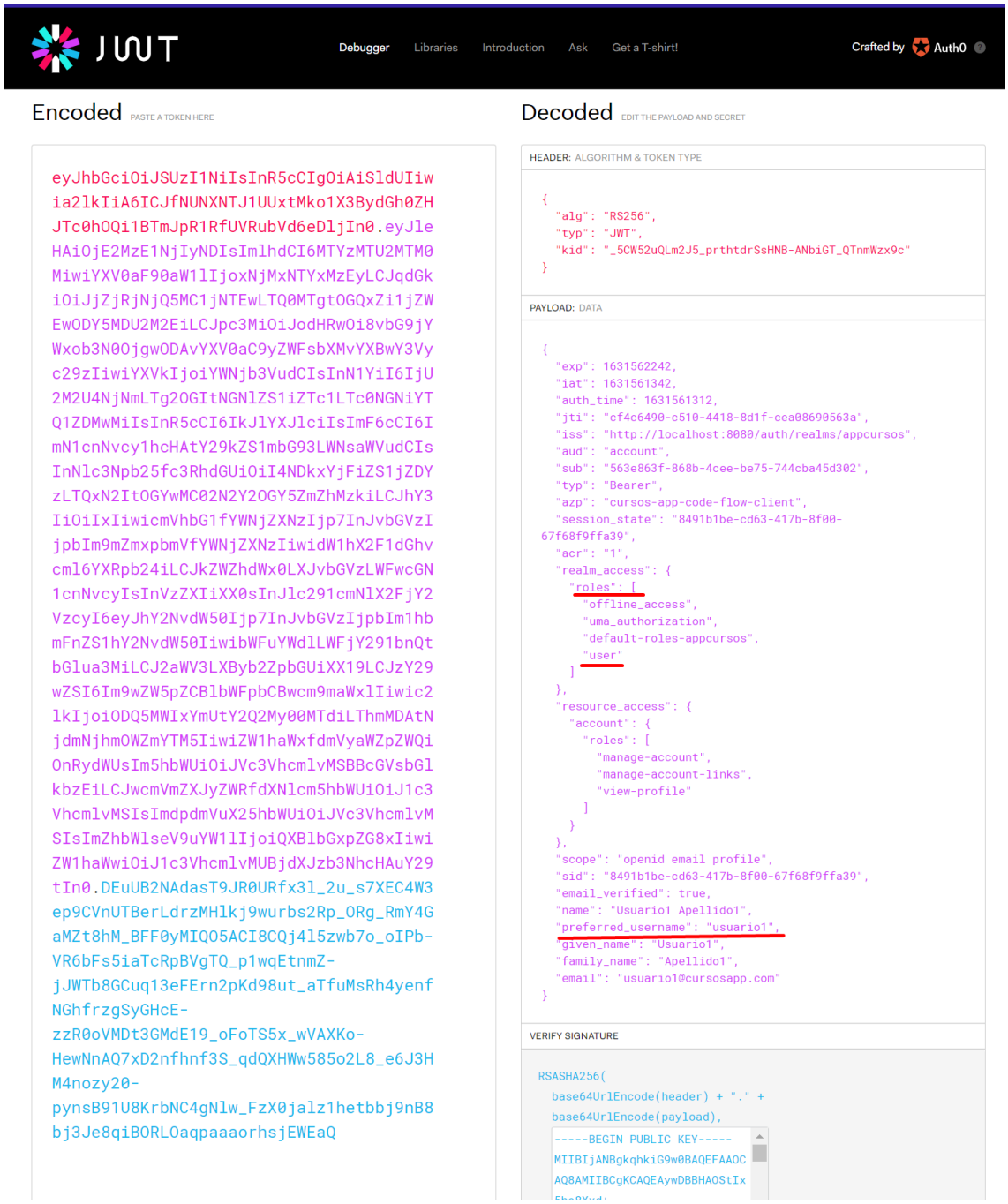

Figura 66. Token de acceso del usuario con rol user

Por último, hay que configurar Postman para que utilice este token en las siguientes peticiones. Esto se hace desde la pestaña Authorization. Allí se selecciona el tipo Bearer Token y se introduce el valor del token en el campo habilitado para ello.

A continuación, se muestran las pruebas realizadas en diferentes endpoints para comprobar los accesos permitidos a las peticiones que porten este token.

En primer lugar, una petición GET al endpoint /cursos del servicio cursos:

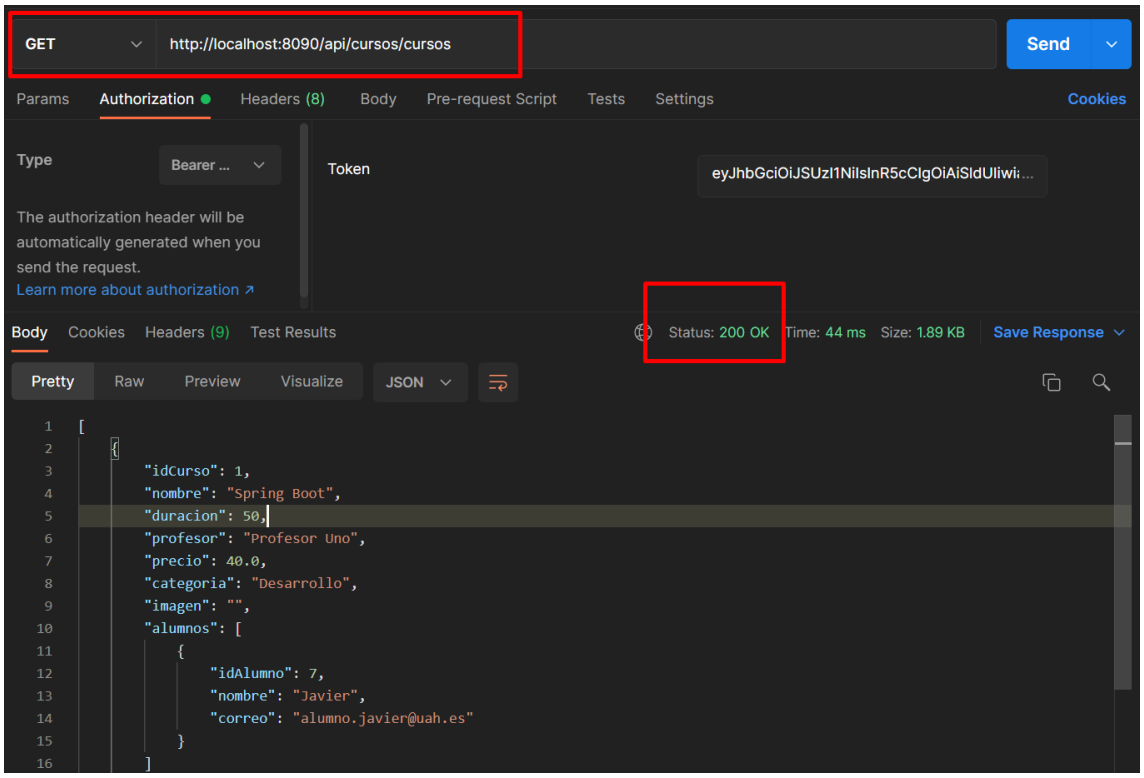

Figura 67. Petición GET al endpoint /cursos con rol user

Seguidamente, una petición con un método distinto a GET a este mismo endpoint:

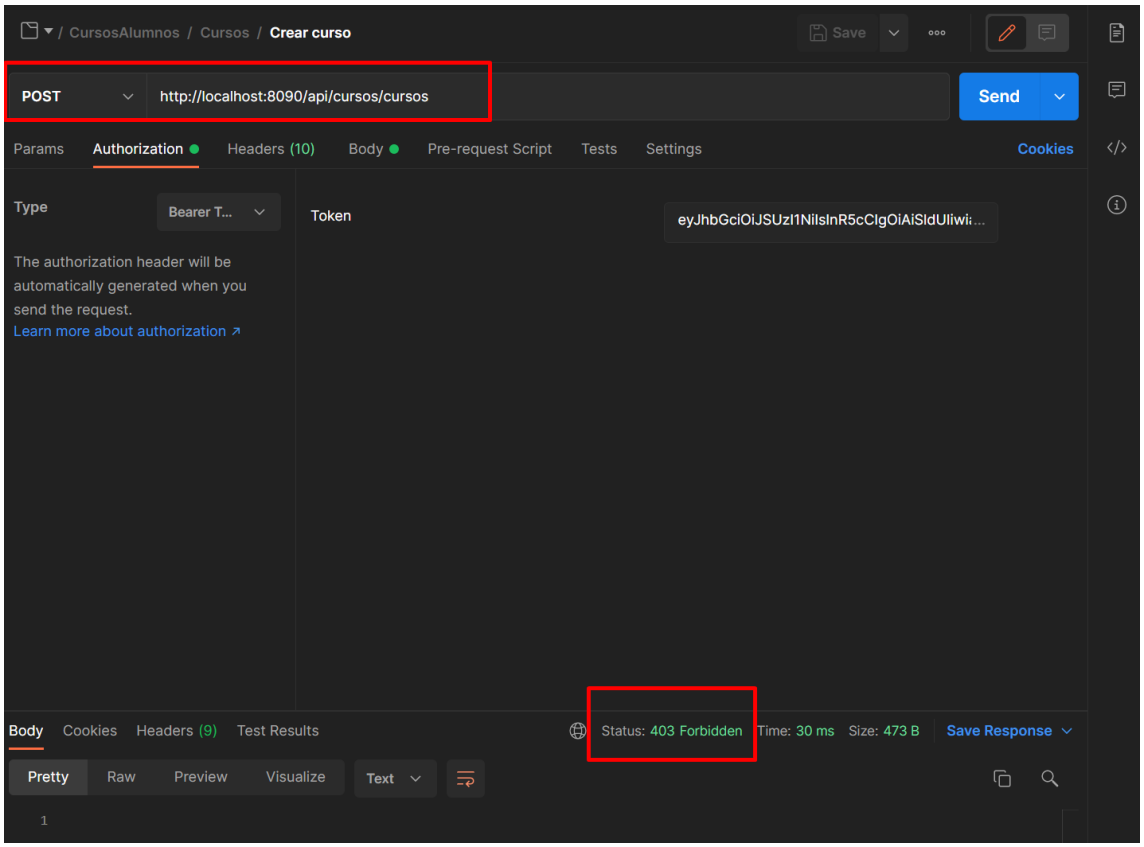

Figura 68. Petición POST al endpoint /cursos con rol user

Finalmente, una petición con el método GET al endpoint /alumnos:

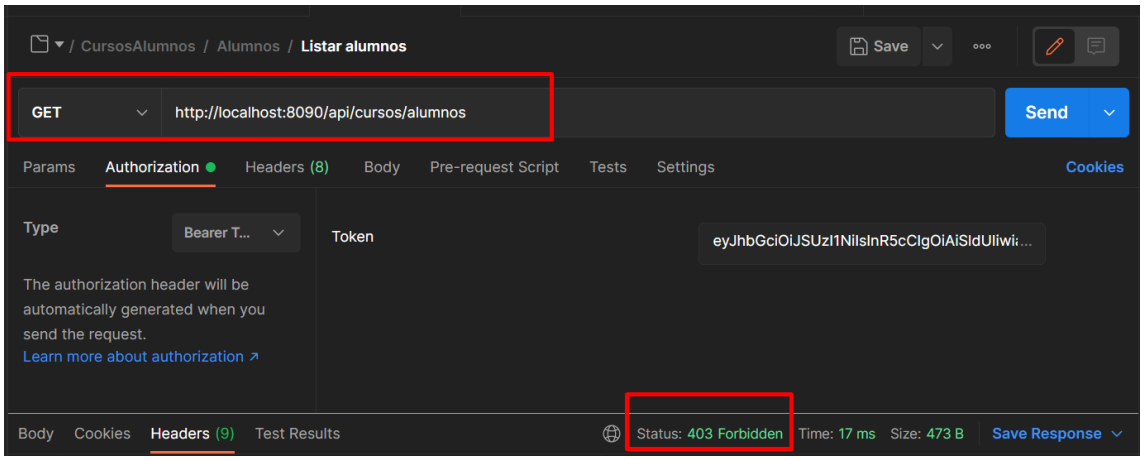

Figura 69. Petición GET al endpoint /alumnos con rol user

Analizando el resultado de estas peticiones, se puede comprobar como el rol user solo tiene el acceso permitido a las peticiones GET que realice sobre el endpoint /cursos, tal y como se definió como requisito.

El siguiente paso en esta demostración, sería repetir este proceso con usuarios que tengan el rol de admin y el rol de teacher. El proceso inicial sería exactamente el mismo hasta llegar al formulario de Keycloak, donde habría que introducir las credenciales apropiadas en función del

tipo de usuario. Tras esto, habría que utilizar el código de autorización obtenido para cambiarlo por el correspondiente token de acceso, configurar Postman para sustituir el token antiguo por el nuevo y probar de nuevo con los enpdoints. En el caso del usuario con rol teacher, va a poder hacer las mismas operaciones que el rol user en el endpoint /cursos, pero, además, va a poder hacer peticiones GET en el endpoint /alumnos. El usuario con rol admin va a tener acceso total a absolutamente todas las combinaciones posibles de métodos HTTP y endpoints.

A continuación, se muestran el aspecto del token para el rol teacher junto con todas las peticiones en Postman, donde se añade un caso más utilizando el método DELETE en el endpoint /alumnos:

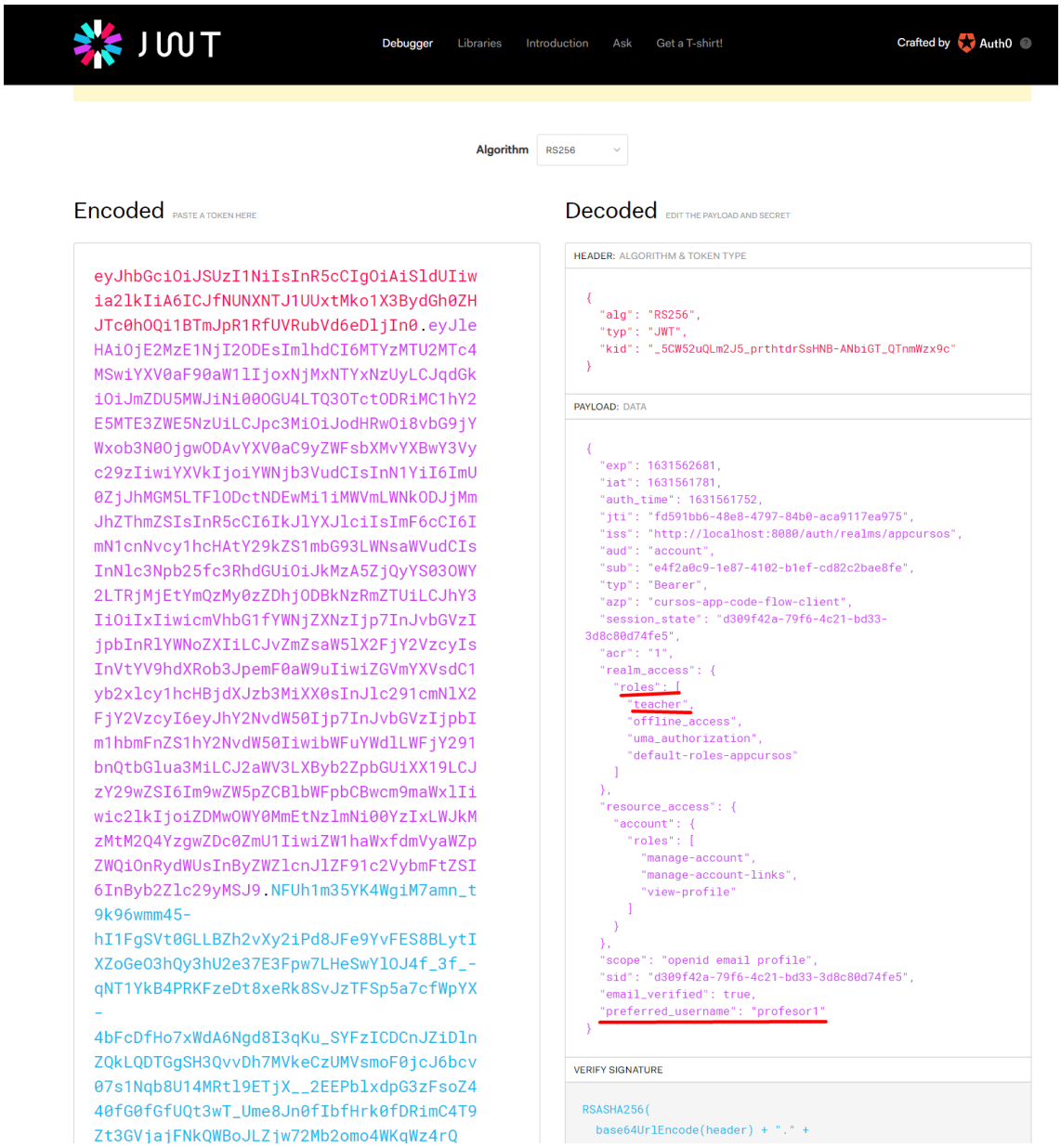

Figura 70. Token de acceso del usuario con rol teacher

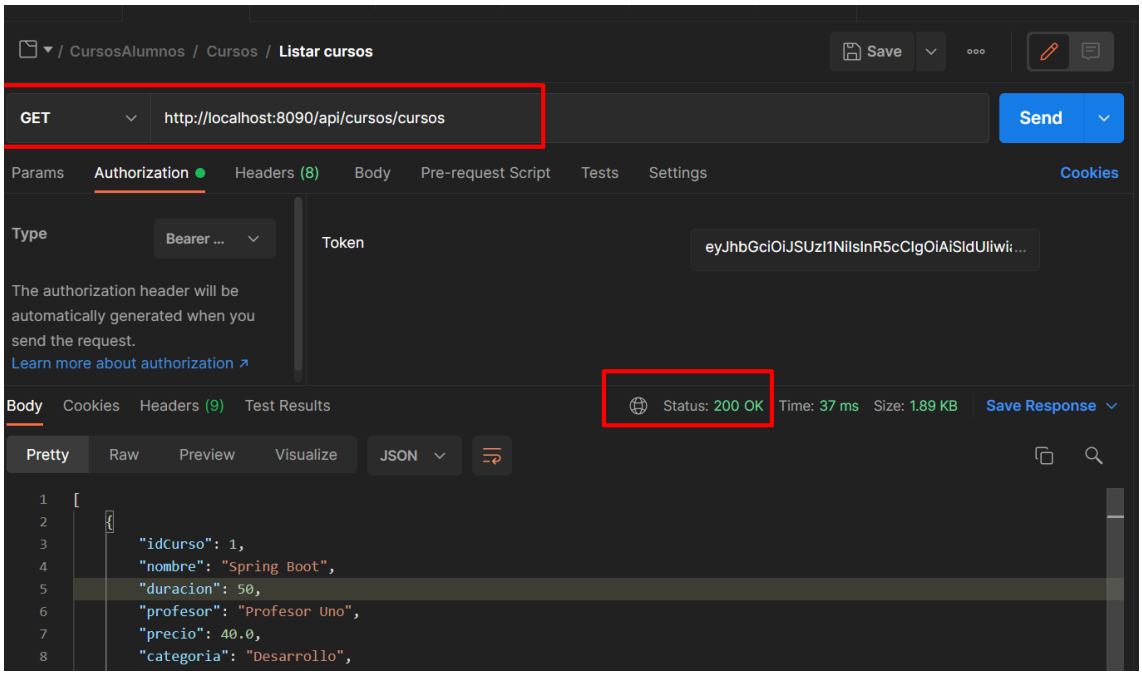

Figura 71. Petición GET al endpoint /cursos con rol teacher

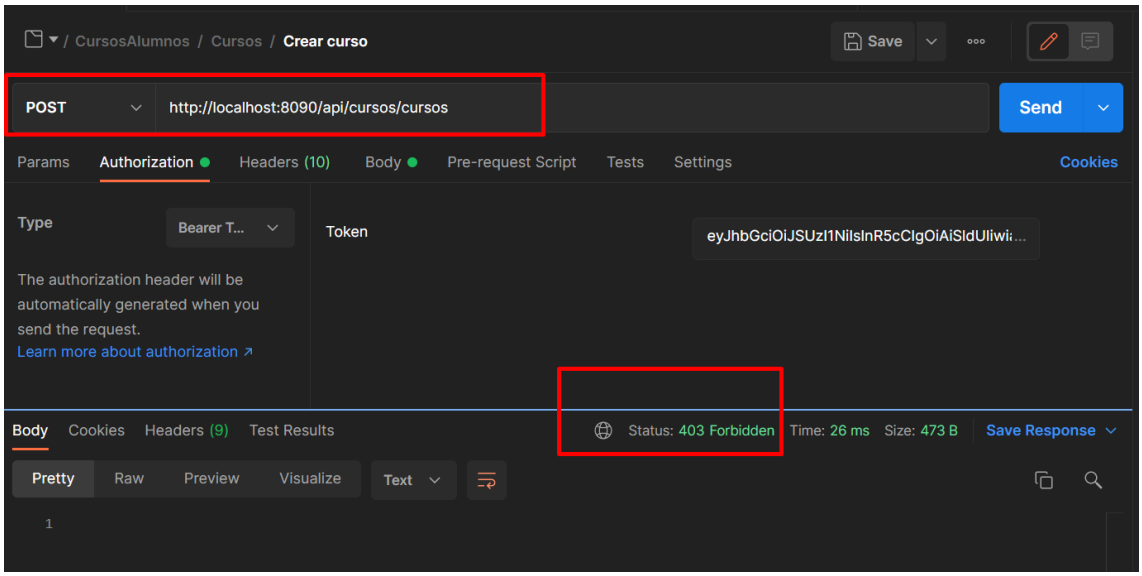

Figura 72. Petición POST al endpoint /cursos con rol teacher

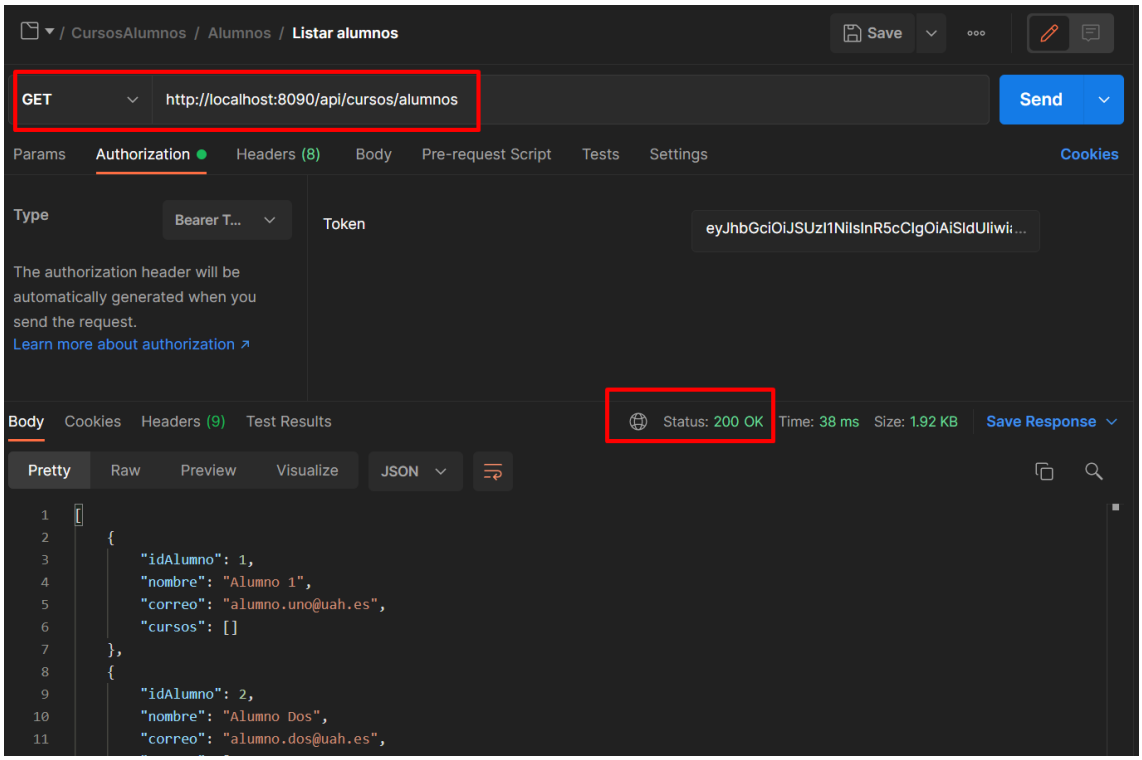

Figura 73. Petición GET al endpoint /alumnos con rol teacher

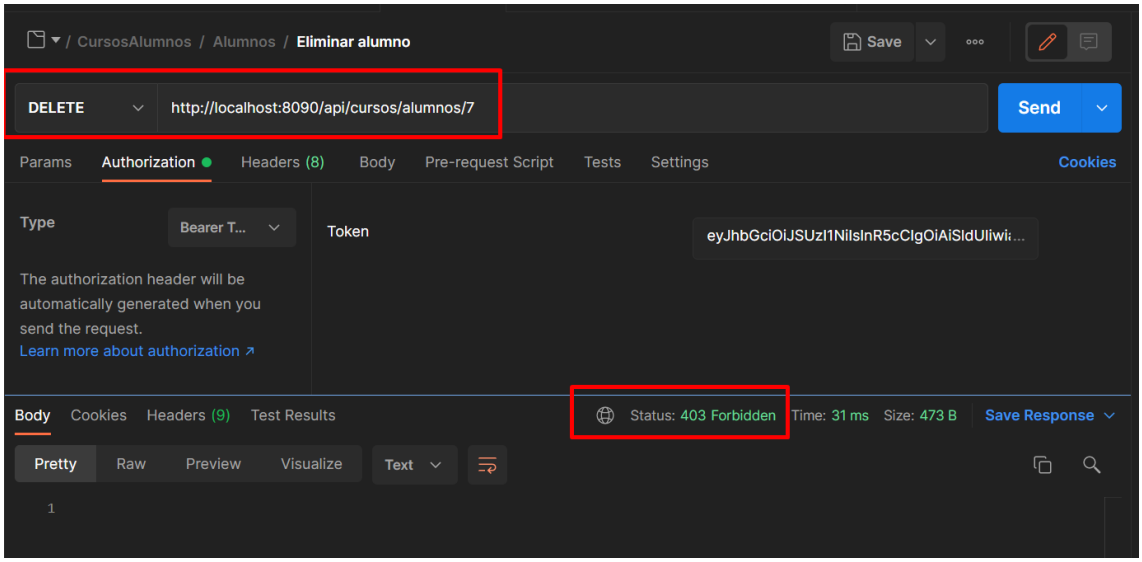

Figura 74. Petición DELETE al endpoint /alumnos con rol teacher

Por último, se muestran el aspecto del token para el rol admin junto con todas las peticiones en Postman de prueba en Postman:

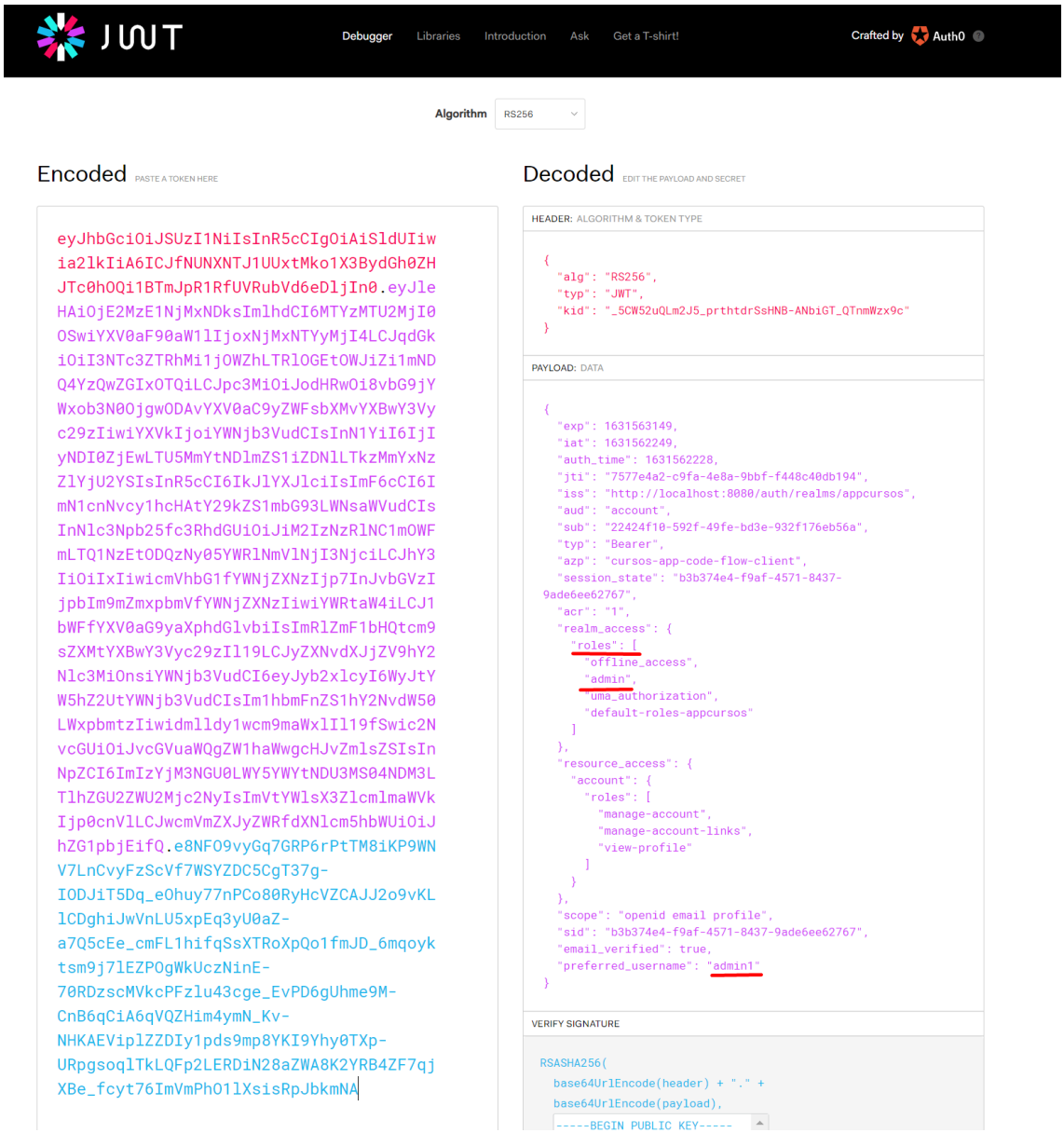

Figura 75. Token de acceso del usuario con rol admin

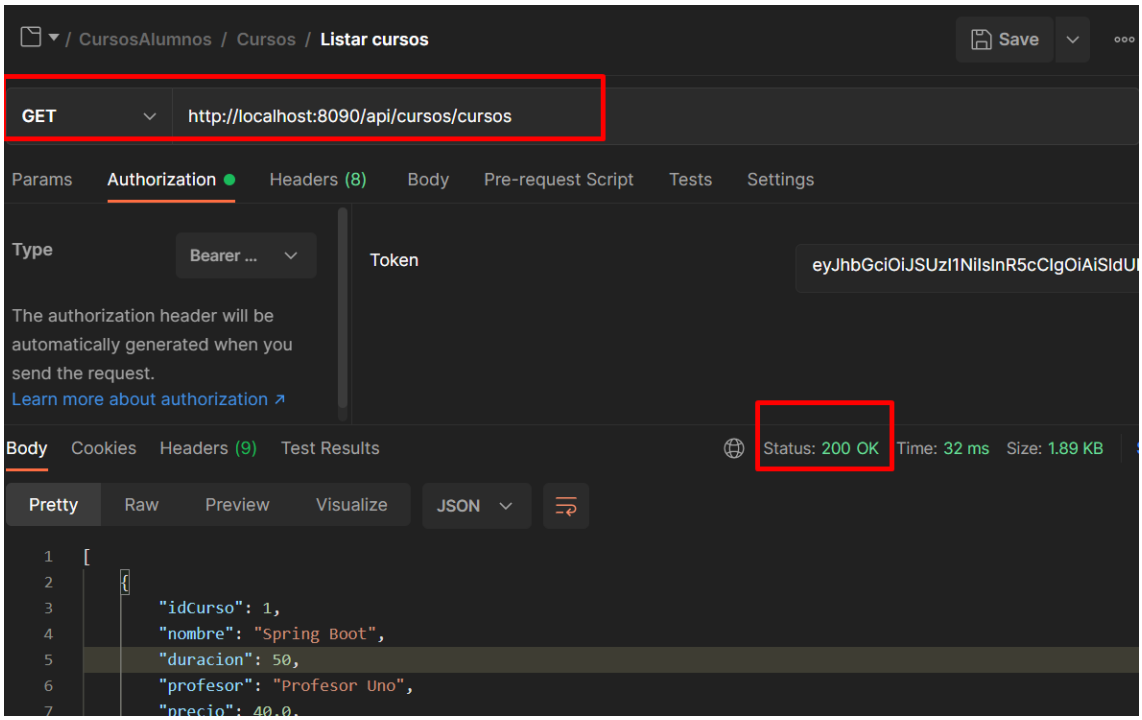

Figura 76. Petición GET al endpoint /cursos con rol admin

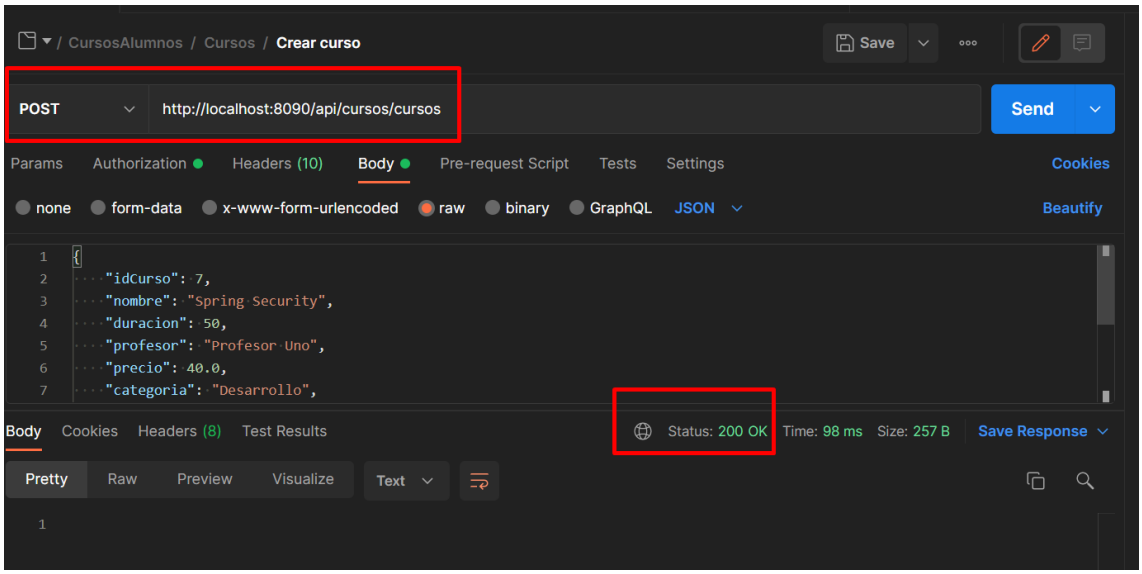

Figura 77. Petición POST al endpoint /cursos con rol admin

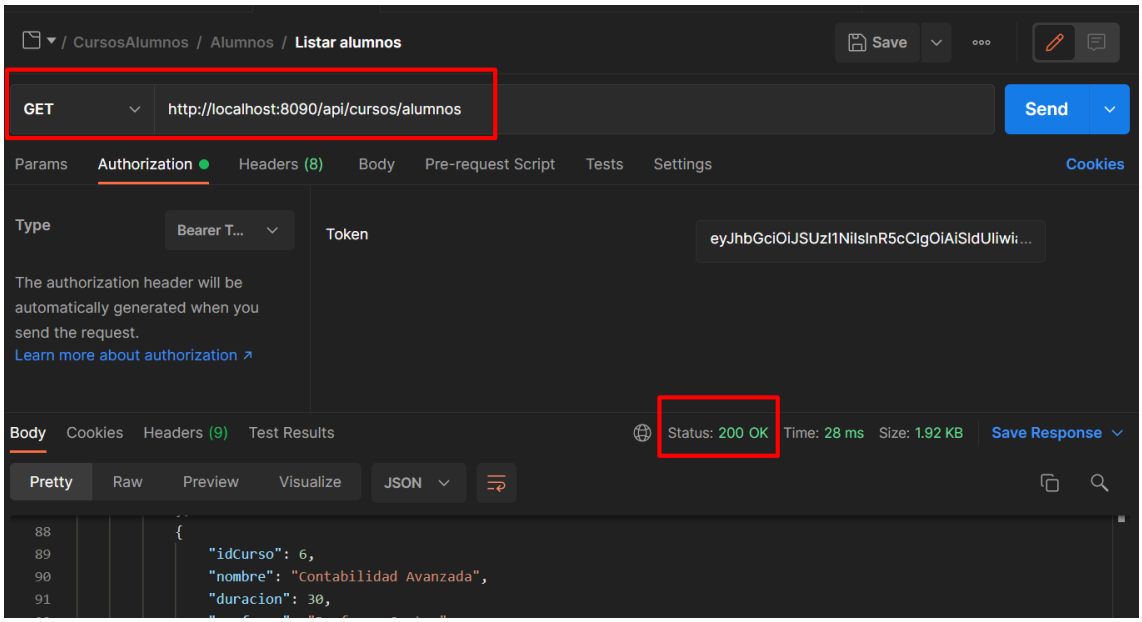

Figura 78. Petición GET al endpoint /alumnos con rol admin

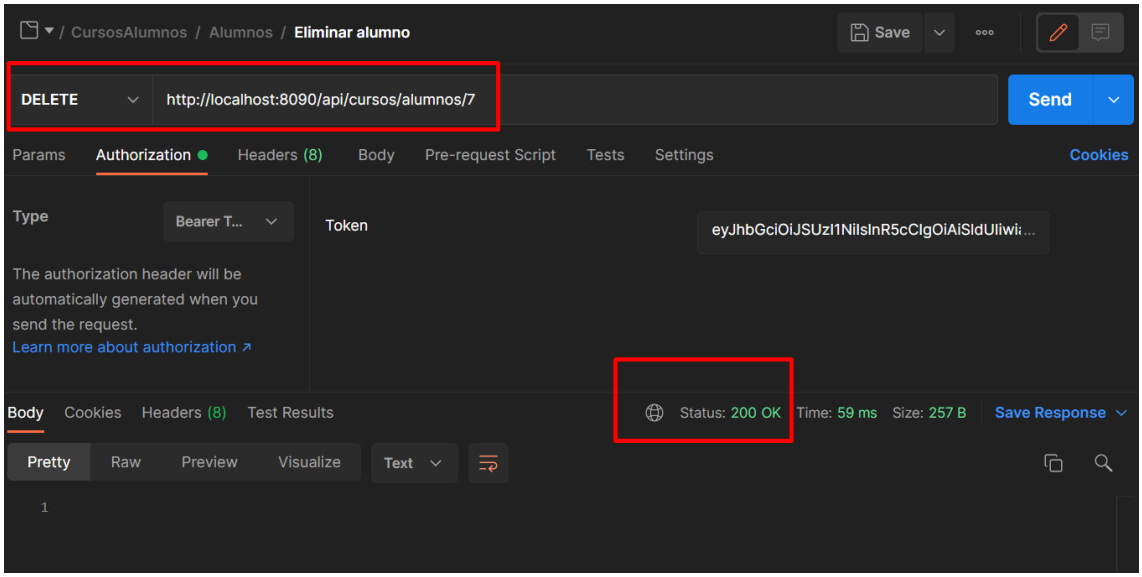

Figura 79. Petición DELETE al endpoint /alumnos con rol admin

4. RESUMEN Y CONCLUSIÓN

4.1. Resumen

La arquitectura de microservicios se ha consolidado como uno de los modelos más populares a la hora de construir aplicaciones software. Esta tendencia apuesta por descomponer el software en bloques funcionales e independientes con el fin de gestionar mejor la creciente complejidad, favoreciendo la aparición de aplicaciones que giran en torno al diseño de APIs. Las APIs proporcionan los datos o servicios de la aplicación, algunos de los cuales deben estar restringidos al personal autorizado. Aquí es donde entran en juego estándares para la autorización de recursos como OAuth 2.0.

En el presente estudio se repasan los principios fundamentales de las arquitecturas basadas en microservicios. Después, se analiza en profundidad la especificación del framework OAuth 2.0 para entender su alcance y ámbito de aplicación. Para terminar, se describe la implementación de un sistema que ponga en práctica todos estos conocimientos y sirva como caso de estudio de Spring Framework.

4.2. Presupuesto del Proyecto

El presupuesto para el presente proyecto se divide en el coste de materiales y el coste de la mano de obra. Las diferentes estimaciones que en el presente documento se recogen tienen en cuenta el salario medio aproximado de cada uno de los puestos de trabajos que se mencionan.

COSTE DE LA MANO DE OBRA

Los puestos de trabajo requeridos para la realización de una aplicación como la descrita en este documento serían los siguientes:

- Arquitecto software: perfil de desarrollador senior que actuará como lead developer de los microservicios, aunque su cometido principal será el diseño de la arquitectura descomponiendo y la configuración de los servicios estructurales de la aplicación (Api Gateway, Eureka y Keycloak). Salario: 40k
- Desarrollador Backend: perfil de desarrollador junior con experiencia en Java para llevar a cabo el desarrollo de los servicios API Rest, tomando en consideración la especificación y la arquitectura software definida por el arquitecto. Salario: 24k

CONCEPTO	HORAS	COSTE/HORA	COSTE TOTAL
INGENIERIA	20	20	400
Arquitecto Software	120	15	1800
Programador backend 1	160	7	1120
Total			**3320**

En el concepto de Ingeniería incluye aspectos como el análisis de la aplicación en función de los requisitos, el modelado de los dominios en tornos a esos requisitos, diseño de tablas, consultas y relaciones, definición de las APIs de los servicios, depuración del código y puesta en marcha y redacción de la memoria.

COSTE DEL MATERIAL

Dentro de este grupo de costes entrarían el apartado del hardware y del software. En el caso del hardware, si bien son necesarios únicamente 2 equipos, estos deben tener unos requisitos mínimos de procesamiento y memoria para no sufrir problemas de rendimiento al desplegar múltiples servicios de forma simultánea

DESCRIPCION	NUMERO	COSTE	TOTAL
Ordenador con 32GB RAM, 500GB de disco duro SSD y procesador Intel Core i7 de 9ª generación como requisitos mínimos.	2	1100	2200
TOTAL			**2200**

A nivel de software no se incurriría en ningún tipo de gasto. Todas las operaciones (escritura del código fuente de los servicios, redacción de memoria, diseño de diagramas) pueden realizarse utilizando software gratuito, y la plataforma de desarrollo de Spring es open source, por lo que el gasto total se puede considerar como nulo.

COSTE GLOBAL

COSTE MANO DE OBRA	3320 €
COSTE MATERIAL	2200 €
TOTAL	**5520 €**

El coste global del desarrollo de la aplicación asciende a la cantidad de cinco mil quinientos veinte euros.

4.3. Conclusiones y Futuras Líneas de Trabajo

El estudio realizado para poder llevar a cabo este documento y la aplicación que en él se describe permite extraer conclusiones muy positivas en cuanto a los objetivos iniciales respecto al desarrollo de una aplicación del protocolo OAuth utilizando el framework de Spring.

Los diferentes frameworks y proyectos que abarca todo el ecosistema de Spring encajan muy bien con la filosofía que hay detrás del diseño de una arquitectura de microservicios, permitiendo establecer los límites de cada servicio de una forma muy clara desde el momento de su conceptualización y ayudando a solventar algunos de los problemas más comunes que presentan los sistemas distribuidos a la hora de comunicarse, como se ha visto con los proyectos de API Gateway y Eureka. La única excepción a esto sería la implementación del servidor de autorización mediante el proyecto de Spring, que finalmente se descartó en busca de una herramienta más estable como lo es Keycloak.

Se ha comprobado que la adopción de una arquitectura basada en microservicios no es algo que se deba decidir a la ligera, ya que también trae consigo una serie de dificultades. De cara al diseño del sistema, hay que analizar bien el escenario de comunicación entre los microservicios, que se vuelve más complejo y lento que en el caso de una aplicación monolítica (donde no existe latencia), la persistencia en bases de datos cambia radicalmente (aparecen transacciones distribuidas y hay que encontrar un mecanismo para gestionar esto correctamente). De cara a los desarrolladores, se complican tareas como la depuración del código o la realización de pruebas de integración.

Aunque la aplicación cumple con los objetivos planteados en un principio, a medida que se avanzaba en la investigación de las tecnologías fueron surgiendo dudas e ideas que dejan muchas líneas de expansión abiertas para futuros desarrollos.

En primer lugar, queda pendiente la implementación de una aplicación del lado cliente que utilice el starter de Spring Oauth2 Client.

En segundo lugar, se podría configurar el servidor de autorización de Keycloak para que, en lugar de crear los usuarios y roles desde su consola de administración, obtuviera esta información del propio servicio de usuarios.

Por último, sería muy interesante realizar una implementación del servidor de autorización utilizando la versión final del proyecto de Spring que actualmente se encuentra en fase experimental. El interés principal de este proyecto podría estar en el desarrollo paralelo de otro servidor de autorización empleando la versión que acaba de quedar obsoleto para comparar ambos proyectos y entender mejor los beneficios y optimizaciones que supuestamente va a traer la nueva versión.

5. BIBLIOGRAFÍA

[1] Dragoni, N. & Giallorenzo, S. & Lluch-Lafuente, A. & Mazzara, M. & Montesi, F. & Mustafin, R. & Safina, L. (2016). *Microservices: yesterday, today, and tomorrow*. Disponible en: https://www.researchgate.net/publication/305881421_Microservices_yesterday_today_and_tomorrow (consultado en agosto de 2021)

[2] O'Brien, L. & Bass, L. & Merson, P., (2005). *Quality Attributes and Service-Oriented Architectures*. Disponible en: https://www.researchgate.net/publication/200086155_Quality_Attributes_and_Service-Oriented_Architectures (consultado en agosto de 2021)

[3] Fowler, M. & Lewis, J. *Microservices*. ThoughtWorks, 2014

[4] Newman, S. (2021). *Building Microservices: Designing Fine-Grained Systems* (2nd ed.). O'Reilly Media.

[5] Krause, L. (2015). *Microservices: Patterns and Applications*. Lucas Krause.

[6] Gao, J & Rahman, M. (2015) *A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development. 2015 IEEE Symposium on Service-Oriented System Engineering*. Published. Disponible en: https://doi.org/10.1109/sose.2015.55 (consultado en agosto de 2021)

[7] Shukla, H. (2020). *What's the difference OAuth 1.0 and OAuth 2.0? - Himanshu Shukla*. Medium. Disponible en: https://medium.com/@greekykhs/whats-the-difference-oauth-1-0-and-oauth-2-0-9f1d22e06963 (consultado en agosto de 2021)

[8] Sydseter, J. (2019). *OAuth2 authorization patterns and microservices* - Norway Community Site. Medium. Disponible en: https://medium.com/capgemini-norway/oauth2-authorization-patterns-and-microservices-45ffc67a8541 (consultado en agosto de 2021)

[9] Worms, D. (2020) *OAuth2 and OpenID Connect for microservices and public applications (Part 2)*. Adaltas. https://www.adaltas.com/en/2020/11/20/oauth-microservices-public-app/ (consultado en agosto de 2021)

[10] Trajkovski, F. (2020). *Securing your microservices with OAuth 2.0. Building Authorization and Resource server*. North-47. Disponible en: https://www.north-47.com/knowledge-base/securing-your-microservices-with-oauth-2-0-building-authorization-and-resource-server/ (consultado en agosto de 2021)

[11] Richer, J., & Sanso, A. (2017). *Oauth 2 in Action*. Manning Publications.

[12] Hardt, D. (Ed.). (2012). *The OAuth 2.0 Authorization Framework*. IETF. Published. https://doi.org/10.17487/rfc6749 (consultado en agosto de 2021)

[13] Jones, M., & Hardt, D. (2012). *The OAuth 2.0 Authorization Framework: Bearer Token Usage*. IETF. Published. Disponible en: https://doi.org/10.17487/rfc6750 (consultado en agosto de 2021)

[14] Mills, W., Showalter, T., & Tschofenig, H. (2015). *A Set of Simple Authentication and Security Layer (SASL) Mechanisms for OAuth*. IETF. Published. Disponible en: https://doi.org/10.17487/rfc7628 (consultado en septiembre de 2021)

[15] Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., Tschofenig, H. (2021). *Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth).* IETF. Draft v45. Disponible en https://datatracker.ietf.org/doc/html/draft-ietf-ace-oauth-authz (consultado en septiembre de 2021)

[16] Bihis, C. (2015). *Mastering Oauth 2.0*. Van Haren Publishing.

[17] Spilca, L. (2020). *Spring Security in Action*. Manning Publications.

[18] Wilson, Y., & Hingnikar, A. (2019). Solving Identity Management in ModernApplications: Demystifying Oauth 2.0, Openid Connect, and Saml 2.0. Apress.

[19] Parecki, A. (2020). *Differences Between OAuth 1 and 2*. OAuth 2.0 Simplified. https://www.oauth.com/oauth2-servers/differences-between-oauth-1-2/ (consultado en agosto de 2021)

[20] Hammer-Lahav, E. (Ed.). (2010). *The OAuth 1.0 Protocol*. IETF. Published. Disponible en: https://doi.org/10.17487/rfc5849 (consultado en agosto de 2021)

[21] https://oauth.net/2/ (consultado en septiembre de 2021)

[22] Raible, M. (2017). *What the Heck is OAuth?*. Okta Developer. Disponible en: https://developer.okta.com/blog/2017/06/21/what-the-heck-is-oauth (consultado en septiembre de 2021)

[23] Conway, M. (1968). *How do committees invent?* Disponible en: https://www.melconway.com/Home/pdf/committees.pdf (consultado en septiembre de 2021)

[24] Hansson, D. (2016). *The Majestic Monolith*. Signal v. Noise. Disponible en: https://m.signalvnoise.com/the-majestic-monolith/ (consultado en septiembre de 2021)

[25] *Deconstructing the Monolith* (Shopify Unite Track 2019). (2019). [Vídeo]. YouTube. https://www.youtube.com/watch?v=ISYKx8sa53g

[26] Okta. (2020). Authentication vs. Authorization. Okta, Inc. Disponible en: https://www.okta.com/identity-101/authentication-vs-authorization/ (consultado en septiembre de 2021)

[27] Paul, R. (2010). *OAuth and OAuth WRAP: defeating the password anti-pattern*. Ars Technica. Disponible en: https://arstechnica.com/information-technology/2010/01/oauth-and-oauth-wrap-defeating-the-password-anti-pattern/ (consultado en septiembre de 2021)

[28] TechTarget. (2002). *Write once, run anywhere?* ComputerWeekly.Com. Disponible en: https://www.computerweekly.com/feature/Write-once-run-anywhere (consultado en septiembre de 2021)

[29] McGloin, M., & Hunt, P. (2013). *OAuth 2.0 Threat Model and Security Considerations*. IETF. Published. Disponible en: https://doi.org/10.17487/rfc6819 (consultado en septiembre de 2021)

[30] Dronia, S., & Scurtescu, M. (2013). *OAuth 2.0 Token Revocation*. IETF. Published. Disponible en: https://doi.org/10.17487/rfc7009 (consultado en septiembre de 2021)

[31] Richer, J. (Ed.). (2015). *OAuth 2.0 Token Introspection*. IETF. Published. Disponible en: https://doi.org/10.17487/rfc7662 (consultado en septiembre de 2021)

[32] Jones, M., Bradley, J., & Sakimura, N. (2015*). JSON Web Token (JWT)*. IETF. Published. Disponible en: https://doi.org/10.17487/rfc7519 (consultado en septiembre de 2021)

[33] Auth0. (2021). *JSON Web Tokens Introduction*. Jwt.Io. Disponible en: https://jwt.io/introduction (consultado en septiembre de 2021)

[34] Bradley, J., & Agarwal, N. (2015). *Proof Key for Code Exchange by OAuth Public Clients*. IETF. Published. Disponible en: https://doi.org/10.17487/rfc7636 (consultado en septiembre de 2021)

[35] Denniss, W., Bradley, J., Jones, M., & Tschofenig, H. (2019). *OAuth 2.0 Device Authorization Grant*. IETF. Published. Disponible en: https://doi.org/10.17487/rfc8628 (consultado en septiembre de 2021)

[36] Hardt, D., Parecki, A., Lodderstedt, T. (2021). *The OAuth 2.1 Authorization Framework*. IETF. Draft v3. Disponible en: https://datatracker.ietf.org/doc/html/draft-ietf-oauth-v2-1-03 (consultado en septiembre de 2021)

[37] Siriwardena, P., & Dias, N. (2020). *Microservices Security in Action*. Manning Publications.

[38] Carnell, J., & Sánchez, H. I. (2021). *Spring Microservices in Action*, Second Edition. Manning Publications.

[39] Larsson, M. (2019). *Hands-On Microservices with Spring Boot and Spring Cloud: Learn how to develop and deploy microservices that are scalable, resilient and manageable using Spring Cloud and Kubernetes*. Packt Publishing.

[40] García, M. M. (2020). *Learn Microservices with Spring Boot: A Practical Approach to Restful Services Using an Event-Driven Architecture, Cloud-Native Patterns, and Containerization* (2nd ed.). Apress.

[41] Spring. (2021). *Spring Cloud Gateway*. Spring Documentation. Disponible en: https://docs.spring.io/spring-cloud-gateway/docs/current/reference/html/ (consultado en septiembre de 2021)

[42] Spring. (2021). *Spring Security Architecture*. Spring Documentation. Disponible en: https://spring.io/guides/topicals/spring-security-architecture (consultado en septiembre de 2021)

[43] Spring. (2021). *Spring Security Reference*. Spring Documentation. Disponible en: https://docs.spring.io/spring-security/site/docs/current/reference/html5/ (consultado en septiembre de 2021)