UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

SEGURIDAD EN SPRING BOOT CON OAUTH

Elier Escobar Careaga

2022

UNIVERSIDAD DE ALCALÁ

Escuela Politécnica Superior

MÁSTER UNIVERSITARIO EN

INGENIERÍA DEL SOFTWARE PARA LA WEB

Trabajo Fin de Máster

"SEGURIDAD EN SPRING BOOT CON OAUTH"

Autor: **Elier Escobar Careaga**

Director: **Salvador Otón Tortosa**

Tribunal:

Presidente: .....

Vocal 1º: ..... Vocal 2º: .....

Calificación: .....

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

ÍNDICE RESUMIDO

ÍNDICE DETALLADO

ÍNDICE DE FIGURAS

1. INTRODUCCIÓN

¿Es necesaria la seguridad en las aplicaciones web?. Esta y muchas preguntas son las que nos cuestionamos cuando decidimos comenzar con el desarrollo de una aplicación, independientemente del tipo que sea. Al concluir este trabajo podremos determinar el porqué es necesaria la seguridad en temas de desarrollo de aplicaciones, principalmente en el desarrollo web. Incluso, al concluir se debe tener un amplio conocimiento para poder aplicar seguridad a una aplicaciones web, sean para estudio cotidiano o para el ámbito laboral.

Hace ya muchos años los desarrolladores han estudiado las diferentes formas en las que implementar software. Hay muchas ideas para ello, pero abundan principalmente las implementaciones asociadas al tipo de arquitectura a la hora de llevar un proyecto como un producto final. Se profundizan principalmente en arquitecturas monolíticas, orientadas a servicios y de microservicios. Con estos tipos de arquitecturas salen a relucir aspectos importantes como lo son sus características, ventajas y desventajas, que nos permiten decidir cuándo es conveniente poner en práctica una u otra, además del contexto y área de acción de la aplicación.

Incluso, sabiendo el tipo de arquitectura a desarrollar, no se puede garantizar de cierta manera la seguridad de una aplicación web. Para ello hay que contar con varios mecanismos bien consolidados de seguridad para palear las diferentes vulnerabilidades que se pueden presentar en un futuro. Uno de estos métodos es el uso de la autenticación, donde el usuario se identifica de alguna forma dentro del sistema, utilizando precisamente su nombre de usuario y contraseña u otros factores de autenticación. Pero utilizando esta forma para proveer a nuestras aplicaciones de seguridad, como lo es la autenticación, no estamos evitando que un ciberdelincuente se apodere de dichas credenciales y acceda al sistema sin más. Entonces es ahí cuando surge la seguridad como un paso más allá de simplemente preguntar al usuario ¿Quién eres?. ¿A qué recursos puede acceder el usuario X?, esta pregunta nace como otro de los métodos de garantizar la seguridad en las aplicaciones: la autorización. Por lo que este método de seguridad en breves palabras protege los recursos del sistema permitiendo que los recursos solo sean usados por los usuarios que se les ha concedido la autorización para ello.

Una de las formas de sintetizar este mecanismo de seguridad como lo es la autorización, es el uso de un protocolo que hoy en día implementan grandes empresas como Facebook, Google, Okta, Github denominado OAuth. Este protocolo a través de un RFC define cual es la forma correcta de desarrollar esta implementación de la autorización.

Con este Trabajo de Fin de Máster se pretende hacer uso de diferentes tecnologías como: Spring Boot, Spring Security y OAuth2, para garantizar la seguridad de una aplicación web de gestión de películas con el fin de demostrar cómo se implementan OAuth2 en una arquitectura basada en microservicios.

2. OBJETIVOS DEL PROYECTO

El objetivo de este Trabajo de Fin de Máster se basa principalmente en la **implementación con Spring Boot de una aplicación web utilizando el protocolo OAuth** para garantizar la seguridad. Con un trabajo de este tipo se es capaz de hacer conciencia de la necesidad de aplicar seguridad a las arquitecturas de desarrollo de software de hoy en día, principalmente en las basadas en microservicios. Necesidad inminente ante ataques de cualquier tipo hacia sitios web y de romper con el esquema donde solo la seguridad abarcaría el acceso a sitios web a través de las credenciales del usuario.

Para garantizar el objetivo principal del trabajo se tuvieron en cuenta varios objetivos secundarios. Objetivos pensados con la capacidad de mostrar cuánto ha sido el nivel de aprendizaje adquirido dentro del desarrollo de este trabajo. Estos se pueden definir de la siguiente forma:

- Adquirir conocimientos relacionados con los tipos de arquitectura de desarrollo de software, aprendiendo de cada una de las características, ventajas y desventajas que que te aportan estas arquitecturas hoy en día. Incluso cómo hacer de estas un ecosistema que garantice muchas facilidades en el futuro con la aplicación de buenas prácticas y patrones aprendidos. Sabiendo así que tipo de arquitectura implementar dentro del desarrollo según las necesidades particulares de cada proyecto.

- Conocer en profundidad el Framework Spring para el desarrollo de aplicaciones Java, así como los diferentes proyectos que lo integran. Saber cómo se implementa Spring Security como parte de la seguridad del Framework mencionado anteriormente y para emplear en proyectos futuros de desarrollo de software. Tener conocimiento en temas de seguridad utilizando este proyecto para priorizar la autenticación y autorización a través de la propia implementación con la que cuenta este proyecto (Spring Security).

- Saber que es el protocolo de OAuth, como es que surge y cómo se usaría dentro de aplicaciones tanto empresariales como personales para garantizar la autorización. Conocer la terminología necesaria, sus principales roles de acción, sus tipos de clientes y los diferentes flujos que garantizan su funcionamiento en dependencia de la situación de cada desarrollo. Siempre teniendo en cuenta las diferentes implementación de este protocolo en versiones anteriores y las próximas que se desarrollaran por el proyecto de Spring.

Aplicando todos estos conocimientos adquiridos en base a los objetivos planteados, el producto final de este Trabajo de Fin de Máster sería una aplicación web de gestión de películas donde se implementa la seguridad no solo basada en la autenticación de usuario a través de sus credenciales de acceso, sino también de la autorización de este a determinados recursos protegidos de la aplicación. Destacando dicho desarrollo en arquitecturas basadas en microservicios.

3. ESTADO DEL ARTE

3.1 Introducción

En el presente apartado se abordarán temas relacionados con las principales tecnologías usadas actualmente en el desarrollo de software web. Se señalan varios tipos de diseños arquitectónicos, así como una comparación exhaustiva de ellos principalmente orientada en sus ventajas y desventajas necesario para valor a la hora de hacer una aplicación. Se explica en detalle cómo surgen cada una de estas arquitecturas. En base al estudio que se realizará en este capítulo, al concluir el mismo se podrá hacer una valoración, sobre el tipo de arquitectura a utilizar según el entorno y las necesidades del usuario. Se realizó un estudio en profundidad sobre temas de seguridad basados en el Framework de Spring y las ventajas que aporta la seguridad en el desarrollo web para este tipo de tecnología y principalmente cómo funciona la seguridad en este Framework (Spring Security). Luego del análisis en temas de seguridad en este Framework, se relaciona estos temas con cierta implementación de un protocolo existente llamado OAuth que viene siendo tendencia en estos días. Este protocolo es implementado por este Framework para las aplicaciones web y principalmente para su uso en estas arquitecturas estudiadas independientemente del tipo que sea.

3.2 Arquitecturas de desarrollo de software

En la actualidad, debido a la inminente necesidad de la automatización de procesos ya es una necesidad el desarrollo de software en sectores empresariales tanto privados como públicos. Este Desarrollo de software se han seguido diferentes tendencias impuestas por la plataforma, lenguaje de programación y experiencia del desarrollo en un área determina, entonces nos preguntaremos varias cuestiones como: ¿Qué arquitectura sería la más idónea para el desarrollar un software?, ¿Hay que utilizar una arquitectura determinada a la hora de seleccionar una tecnología específica? A continuación, se responderán estas preguntas y otras, relacionadas con el tipo de arquitectura usada hoy en día y se seleccionará la más idónea para el desarrollo de este proyecto.

3.2.1 Aplicaciones monolíticas

3.2.1.1 ¿Qué es una arquitectura monolítica?

Una arquitectura monolítica es un modelo tradicional de un programa de software que se compila como una unidad unificada y que es autónoma e independiente de otras aplicaciones. La palabra "monolito" suele evocar algo grande y glacial, una imagen que no está alejada de la realidad de las arquitecturas monolíticas para el diseño de software. Esta es una red informática grande y única, con una base de código que aúna todos los intereses empresariales. Para hacer cambios en este tipo de aplicación, y actualizar toda la pila, se requiere acceder a la base de código y compilar e implementar una versión actualizada de la interfaz del lado del servicio. Esto hace que las actualizaciones sean restrictivas y lentas [1].

Existen muchos libros y artículos en internet [2] que definen este tipo de arquitectura de diferente forma:

> *…hace referencia a una aplicación software en la que la capa de interfaz de usuario, lógica de negocios y la capa de acceso a datos están combinadas en un mismo programa y sobre una misma plataforma.*

Incluso para Ian Gorton en [3] plantea lo siguiente haciendo referencia a la arquitectura monolítica:

> *Since the dawn of IT systems, the monolithic architectural style has dominated enterprise application. Essentially, this style decomposes an application into multiple logical modules or services, which are built and deployed as a single application…*

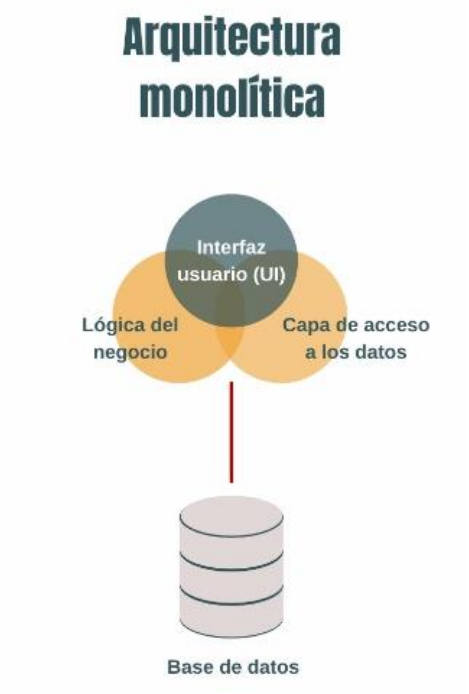


Figura 1. Arquitectura de una aplicación monolítica

Como se puede apreciar en la figura anterior, la arquitectura de aplicaciones monolíticas está basada principalmente en tres componentes:

- **Capa de presentación (Interfaz de usuario (UI)):** Es la encargada de la comunicación con el usuario, tanto para brindarle información como para capturar datos. Esta tiene las características de que debe ser amigable, entendible y fácil de usar. Solamente se comunica con la lógica de negocio.

- **Lógica de negocio:** Es donde se establecen las reglas principales que deben cumplirse dentro del software. Se comunica con la capa de presentación para recibir y enviar los datos del usuario. También es capaz de comunicarse con la capa de acceso a datos para solicitar al gestor de base de datos almacenar o recuperar datos que solicita el usuario.

- **Capa de acceso a datos:** Es la encargada de acceder a los datos para guardar o recuperar la información del usuario que proviene de la capa de negocio.

Los monolitos son prácticos para el comienzo de un proyecto para evitar la exceso de la gestión de código y la implementación. Es posible desplegar todo a la vez cuando se habla de monolito. Esta arquitectura es bastante efectiva si trabajamos en proyectos pequeños y con determinadas características.

Algunas de **las características** de las aplicaciones monolíticas son:

- Realizan todas las operaciones de un extremo a otro para garantizar el cumplimiento de una tarea.
- Cada aplicación administra su propia base de datos.
- Todo funciona sobre una única plataforma

- Se suelen implementar todas las capas por separado, pero dentro de la misma aplicación.
- Permite construir aplicaciones rápidamente.
- Funciona para el 99% de las aplicaciones pequeñas.
- Es perfecta para pruebas de concepto.

Una arquitectura de este tipo es un buen punto para dar comienzo a una solución, para luego ir incorporando microservicios o cualquier otro tipo de arquitectura y patrones según las necesidades. De ahí que se deducen las diferentes **ventajas**:

- **Implementación sencilla:** debido a que solo existe un componente, es muy fácil para un equipo pequeño de desarrollo iniciar un nuevo proyecto y ponerlo en producción rápidamente.
- **Fácil de escalar:** Solo es necesario instalar la aplicación en varios servidores y situarlo detrás de un balanceador de carga.
- **Rendimiento:** Las aplicaciones monolíticas son significativamente más rápidas debido que todo el procesamiento lo realiza localmente y no requieren consumir procesos distribuidos para completar una tarea.
- **Fácil de probar:** debido a que es una sola unidad de código, toda la funcionalidad está disponible desde el inicio de la aplicación, por lo que es posible realizar todas las pruebas necesarias sin depender de nada más.
- **Pocos puntos de fallo:** El hecho de no depender de nadie más, mitiga gran parte de los errores de comunicación, red, integraciones, etc. Prácticamente los errores que pueden salir son por algún bug del programador, pero no por factores ajenos.

Incluso con este tipo de características planteadas anteriormente, se puede deducir un conjunto de desventajas que desfavorecen este tipo de aplicaciones con respecto a otras a la hora de decidir cual implementar. Entre ellas podemos observar:

- **Escalabilidad:** Escalar una aplicación monolítica implica escalar absolutamente toda la aplicación, gastando recursos para funcionalidad que quizás no necesita ser escalada.
- **Implementación:** un pequeño cambio en una aplicación monolítica requiere una nueva implementación de todo el monolito.
- **Falta de flexibilidad:** un monolito está limitado por las tecnologías que se utilizan en él.
- **Barrera para la adopción de tecnologías:** cualquier cambio en el marco o el lenguaje afecta a toda la aplicación, lo que hace que los cambios suelen ser costosos y lentos.
- **Velocidad de desarrollo más lenta:** con una aplicación grande y monolítica, el desarrollo es más complejo y lento.
- **Fiabilidad:** si hay un error en algún módulo, puede afectar a la disponibilidad de toda la aplicación.

A pesar de la mala reputación que se les ha dado a las aplicaciones monolíticas, la realidad es que tiene ventajas que hasta el día de hoy son difíciles de igualar, entre las que destacan su total independencia y el rendimiento que puede llegar a alcanzar en determinadas ocasiones.

Utilizar este tipo de aplicaciones no es para nada malo, lo malo sería quedarnos siempre atascados en este estilo arquitectónico y no ir migrando a otros estilos más sofisticados a medida que nuestra aplicación va

creciendo. Algunos síntomas de que el estilo arquitectónico ya nos comienza a quedar chico sería cuando, cada vez es más difícil dividir el trabajo entre los desarrolladores sin que tengan conflictos con el código, existen una necesidad de escalar ciertas funcionalidades y no todo el proyecto, cuando el proyecto es tan grande que es complicado para los desarrolladores comprender el funcionamiento por la complejidad y la absurda cantidad de clases o archivos, y finalmente, cuando necesitamos empezar a diversificarse con respecto a las tecnologías a utilizar.

3.2.2 Arquitectura orientada a servicios (SOA)

La arquitectura orientada a los servicios (SOA) es un tipo de diseño de software que permite reutilizar sus elementos gracias a las interfaces de servicios que se comunican a través de una red con un lenguaje común [4].

Este diseño (SOA), es para vincular recursos empresariales y computacionales (principalmente organizaciones, aplicaciones y datos) bajo demanda para obtener los resultados esperados para las aplicaciones finales que consumen estos servicios. [5] define SOA como:

> *A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.*

Un servicio es una unidad autónoma de una o más funciones del software diseñada para realizar una tarea específica, como recuperar cierta información o ejecutar una operación. Contiene las integraciones de datos y código que se necesitan para llevar a cabo una función empresarial completa y diferenciada. Se puede acceder a él de forma remota e interactuar con él o actualizarlo de manera independiente [6].

SOA integra principalmente los elementos de software que se implementan y se mantienen por separado, y permite que se comuniquen entre sí y trabajen en conjunto para formar aplicaciones de software en distintos sistemas. Con este tipo de arquitectura logramos separar la lógica de integración de negocio de la implementación, posibilitando que el desarrollador se centre en hacer una aplicación integrada en lugar de enfocarse en detalles de implementación.

SOA es una evolución de la Arquitectura Basada en Componentes, el Diseño Basado en Interfaz (Orientado a Objetos) y los Sistemas Distribuidos de la década de 1990, como DCOM, CORBA, J2EE e Internet en general. No significa necesariamente servicios web como. NET, J2EE, CORBA o ebXML [7]. En cambio, se trata de implementaciones SOA especializadas que incorporan los aspectos centrales de un enfoque de arquitectura orientado a servicios. Cada una de estas implementaciones amplía el modelo de referencia SOA básico. Si bien los servicios web brindan soporte para muchos de los

conceptos de SOA, no los implementan todos. Actualmente no admiten la noción de un contrato de arrendamiento. Además, ninguna especificación oficial proporciona niveles de QoS para un servicio. Una organización no puede implementar una arquitectura completa orientada a servicios dadas estas limitaciones con los servicios web. Además, los consumidores de servicios pueden ejecutar servicios web directamente si conocen la dirección y el contrato del servicio. No tienen que ir al registro para obtener esta información. Hoy, de hecho, la mayoría de las organizaciones implementan servicios Web sin registro. En consecuencia, la medida en que una organización implementa una SOA con servicio web varía mucho.

Para lograr el objetivo de esta arquitectura como se mencionaba anteriormente, de separar la lógica de integración de negocio de la implementación, se crean componentes de servicio individuales necesarios para los procesos de negocio. Por lo tanto, el resultado de ésta es la separación en tres capas: la lógica de integración de negocio, componentes de servicio e implementación [8].

Figura 2. Capas de la arquitectura orientada a servicios (SOA)

El desarrollador de integración puede ensamblar gráficamente los componentes de servicio sin necesidad de conocer los detalles de la implementación. Los componentes de servicio también ofrecen la opción de dejar que el desarrollador de integración o alguien que trabaje con él añada la implementación más tarde. Estos componentes se enlazan entre sí de forma visual. Es decir, el código de los componentes no se expone al usuario. En el nivel de lógica de negocio, los componentes se ensamblan independientemente de su implementación. Por tanto, esta permite centrarse en resolver los problemas de negocio mediante la utilización y reutilización de componentes, en lugar de distraer la atención hacia la tecnología que implementa los servicios que se están utilizando.

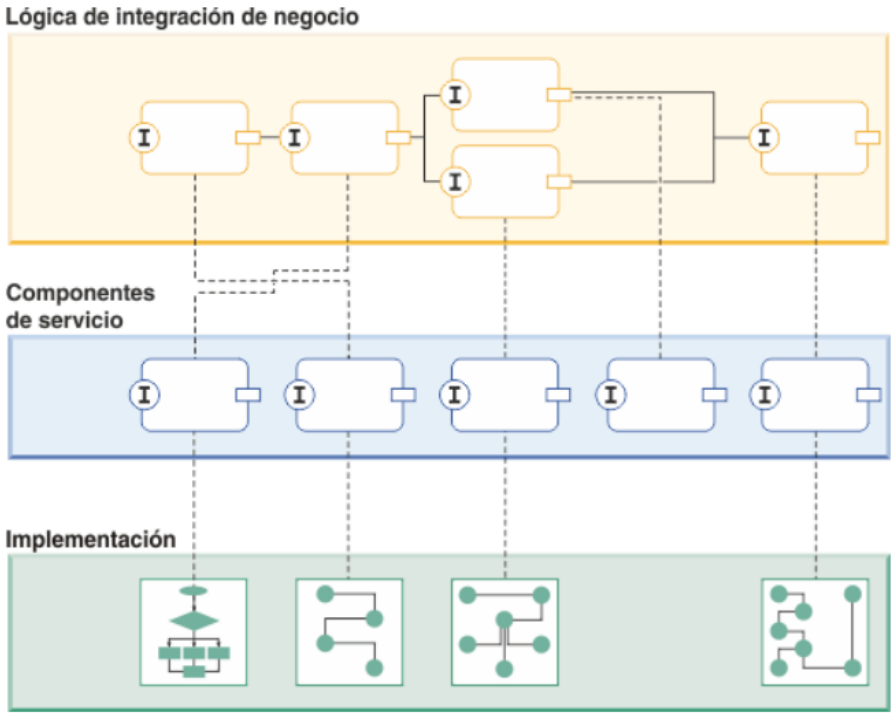


Figura 3. Detalles de la arquitectura orientada a servicios (SOA)

En base a lo plasmado anteriormente podemos concluir que SOA al igual que la arquitectura monolítica cuenta con ventajas y desventajas plasmadas a continuación. Algunas de sus ventajas son:

- **Reducción de costes:** el coste de ampliar o crear nuevos servicios se reduce considerablemente tanto en aplicaciones nuevas como ya existentes, además de su reducido coste de mantenimiento.
- **Riesgos de migración:** al adaptar SOA a partir de una tecnología existente se siguen utilizando los componentes existentes, por lo que se reduce el riesgo de introducir fallos.
- **Reutilización de componentes:** Una vez que se crea un componente como un servicio este puede ser totalmente reutilizable en su posterioridad.
- **Agilidad organizacional:** SOA define los componentes básicos de las capacidades proporcionadas por el software y ofrece algunos servicios que cumplen con algunos requisitos organizacionales; que se pueden recombinar e integrar rápidamente.
- **Aprovechamiento del sistema existente:** este es uno de los principales usos de SOA, que es clasificar elementos o funciones de las aplicaciones existentes y ponerlas a disposición de quienes los van a utilizar
- **Integración neutral del lenguaje:** independientemente del lenguaje de programación utilizado, el sistema ofrece e invoca servicios a través de un mecanismo común. La neutralización del lenguaje de programación es uno de los beneficios clave del enfoque de integración de SOA.
- Mejora en los tiempos de realización de cambios en procesos.

Independientemente del gran número de ventajas con las que cuenta, también disponemos de desventajas muy notables en este tipo de arquitectura, tales como:

- **SOA depende de la implementación de estándares:** Sin estándares, la comunicación entre aplicaciones requiere de mucho tiempo y código.

- SOA **no es para** aplicaciones con **alto nivel de transferencia de datos**, aplicaciones que no requieren de **implementación del tipo request/response** y para aplicaciones que tienen un **corto periodo de vida**.
- Incrementalmente se hace **difícil y costoso** el ser capaz de cumplir con los protocolos y hablar con un servicio.
- Implica **conocer los procesos del negocio, clasificarlos, extraer** las funciones que son comunes a ellos, estandarizarse y formar con ellas capas de servicios que serán requeridas por cualquier proceso de negocio.
- En la medida en que un servicio de negocio, vaya siendo incorporado en la definición de los procesos de negocio, dicho servicio **aumentará su nivel de criticidad**. Por tanto, cada vez que se requiera efectuar una actualización en dicho servicio (por ejemplo, un cambio en el código, una interfaz nueva, etc.), deberá evaluarse previamente el impacto y tener mucho cuidado con su implementación. Sin embargo, parte de la problemática anterior, puede ser solventada en virtud de un buen diseño del servicio.

Con todas estas consideraciones se es capaz de llegar a una arquitectura flexible capaz de adaptarse a condiciones de negocio que cambian con el tiempo. Incluso así, es necesario ir un paso más allá para lograr desarrollar aplicaciones robustas y eficientes, dando paso a nuevas formas de diseñar un software como se evidenciará próximamente.

3.2.3 Microservicios

3.2.3.1 ¿Cómo surgen los microservicios?

La arquitectura de microservicios ha surgido como un patrón común de desarrollo de software a partir de las prácticas de varias organizaciones líderes. Estas prácticas incluyen principios, tecnologías, metodologías y tendencias organizacionales. La arquitectura de microservicios se ha vuelto enormemente popular porque las arquitecturas monolíticas tradicionales ya no satisfacen las necesidades de escalabilidad y ciclo de desarrollo rápido [9], el éxito de algunas grandes empresas en la construcción y despliegue de servicios es una fuerte motivación para que otras consideren hacer el cambio.

En 2009, Netflix tuvo problemas de crecimiento. Su infraestructura no podía seguir el ritmo de la demanda de sus servicios de streaming de video, que crecía a toda velocidad. En esta situación, la empresa decidió migrar su infraestructura de TI de sus centros de datos privados a una nube pública y reemplazar la arquitectura monolítica por otra de microservicios. El único problema era que el término "microservicios" no existía en aquella época, y que la estructura tampoco era muy conocida [1].

Netflix se convirtió en una de las primeras empresas destacadas en migrar de un monolito a una arquitectura de microservicios basada en la nube. Ganó el premio JAX Special Jury de 2015, en parte debido a esta nueva infraestructura que internalizó DevOps. En la actualidad, Netflix tiene más de un millar de microservicios que administran y respaldan partes independientes de la plataforma, mientras que sus ingenieros implementan código con frecuencia, a veces miles de veces al día.

Netflix fue pionero en lo que, desde entonces, es algo cada vez más habitual: la transición de una arquitectura monolítica a otra de microservicios [1].

3.2.3.2 ¿Qué son los microservicios?

La arquitectura de microservicios es una arquitectura orientada a servicios y a la nube que busca descomponer una aplicación en diferentes servicios, con el fin de obtener alta disponibilidad, bajo acoplamiento, descentralización y tolerancia a fallos. Es una arquitectura ideal para escenarios de alto tráfico, alta demanda (usuarios que hacen peticiones) y de alta disponibilidad. Cada microservicio puede estar construido en un lenguaje y tecnología diferente

Los microservicios no disminuyen la complejidad, simplemente hacen que esta sea más visible y gestionable, debido a que dividen las tareas en pequeños procesos que funcionan de forma independiente entre sí y contribuyen al conjunto en su totalidad.

Empresas como Amazon definen la arquitectura de la siguiente forma [10]:

> *Los microservicios son un enfoque arquitectónico y organizativo para el desarrollo de software donde el software está compuesto por pequeños servicios independientes que se comunican a través de API bien definidas. Los propietarios de estos servicios son equipos pequeños independientes.*

Estas arquitecturas hacen que las aplicaciones sean mucho más fáciles de escalar y rápidas de desarrollar. Permitiendo la innovación y aceleración del tiempo de comercialización de las nuevas características.

Los problemas típicos asociados con la arquitectura monolítica son técnicos (p. ej., el sistema se vuelve altamente acoplado, difícil de mantener, presenta efectos secundarios) o relacionados con el negocio (p. ej., mucho tiempo para lanzar nuevas funciones, baja productividad de los desarrolladores). En algunos casos, migrar hacia una arquitectura de microservicios representa la mejor opción para resolver los problemas existentes y, al mismo tiempo, mejorar la mantenibilidad del sistema y la frecuencia de los lanzamientos de productos [11]. En la figura siguiente se muestra gráficamente el concepto de una arquitectura de este tipo

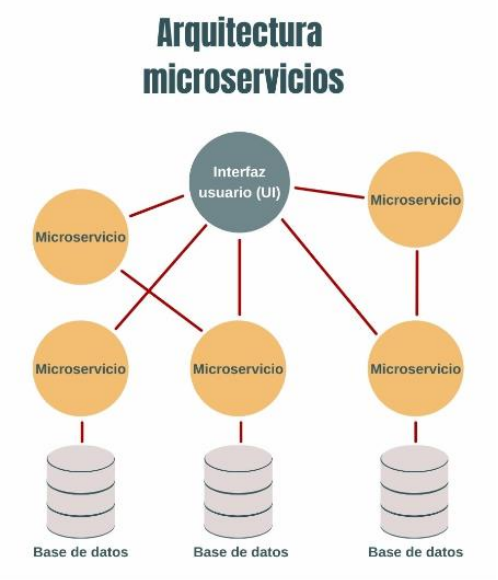


Figura 4. Arquitectura de microservicios

3.2.3.3 Características de los microservicios

- **Puede dividirse en varias partes independientes:** cada uno de los servicios puede implementarse y modificarse sin afectar otros aspectos funcionales de la aplicación.
- La **forma en que se organizan** contrasta con un entorno monolítico, ya que se adaptan a nivel granular a aspectos como las capacidades, necesidades y preferencias del negocio o cliente donde se van a implementar.
- **Cada módulo es independiente**, ya que cada uno de ellos tiene su propia base de datos; en otras palabras, no todos los servicios se conectan con la misma base de datos.
- Dado que varios servicios están operativos, necesitamos tener **sistemas de aviso** y acción en caso de que haya algún fallo en estos servicios. Por ejemplo, en el caso de un problema, recibiremos una advertencia, se enviaría un correo electrónico a soporte, etc.
- Cada servicio está **diseñado** para un **conjunto de capacidades** y se enfoca en **resolver un problema específico**. Si los desarrolladores aportan más código a un servicio a lo largo del tiempo y el servicio se vuelve más complejo, se puede dividir en servicios más pequeños.
- Cada servicio componente en una arquitectura de microservicios **se puede desarrollar, implementar, operar y escalar sin afectar el funcionamiento de otros servicios**. Los servicios no necesitan compartir ninguno de sus códigos o implementaciones con otros servicios. Cualquier comunicación entre componentes individuales ocurre a través de API bien definidas.

3.2.3.4 Comunicación entre microservicios

Existen dos tipos de comunicación entre servicios para la cooperación de estos. Es un reto el tener que resolver funcionalidades que involucren a uno o más servicios, sabiendo que suele tener su propia base de datos y que potencialmente se requiere hacer las operación relacionadas a los datos en múltiples servicios para implementar una funcionalidad. Por tanto, existen dos estrategias para implementar la comunicación entre servicios [12]:

- **Coreografía:** Se encarga de ser responsable de la tomar las decisiones. Se comunican principalmente mediante el intercambio de eventos.
- **Orquestación:** Se centraliza la lógica de coordinación en un orquestador (una especie de maestro de ceremonia). El orquestador envía mensajes a los participantes indicando cuál es la operación que tienen que invocar.

No existe estandarización en cuanto al uso de estos dos enfoques. Sam Newman en su libro [13], destaca que, si utiliza para la comunicación entre microservicios el modelo de orquestación, el sistema tiene un nivel de acoplamiento más bajo y mayor flexibilidad ante cambios. Pero, se requiere trabajo extra para monitorear y controlar todos los procesos. Se destaca también que los sistemas se vuelven más frágiles ya que se introduce un único punto de falla. Precisamente, es por lo que este autor recomienda el uso de coreografía. Chris Richardson en [12], recomienda la utilización de coreografía para interacciones sencillas y orquestación para las comunicaciones que requieren mayor complejidad.

3.2.3.5 Ventajas y desventajas de los microservicios

Al ser un tipo de arquitectura de alguna forma inspirada en SOA, cuenta precisamente con puntos positivos que esta trae, pero además agrega nuevas ventajas como:

- **Independencia:** Cada microservicio se puede implementar de manera independiente del resto. Se pueden implementar en paralelo, se pueden desplegar según sea necesario y el mantenimiento es mucho más simple. Traducido en mejoras rápidas y continuas de cada funcionalidad.
- **Equipos autónomos y ágiles:** Permite que los equipos de desarrollo sean autónomos, desacoplados y pequeños. Cada microservicio es desarrollado por un único equipo, implicando que la velocidad de desarrollo sea considerablemente mayor.
- **Versatilidad de tecnologías:** Los equipos de desarrollo pueden utilizar el lenguaje que estimen conveniente, tecnologías, y Frameworks, incluso la base de datos que mejor se adapte al servicio y funcionalidad. Se pueda experimentar con nuevas tecnologías con mayor agilidad y con un riesgo significativamente menor.
- **Aislamiento y resiliencia: Cuanto mayor cantidad de microservicios sean**, más complejo será y mayor probabilidad de fallas experimentará, por lo que se debe asegurar que cada uno sea tolerante a fallos y pueda recuperarse rápidamente de alguno. Desde el diseño se deben considerar los fallos que estos servicios pueden llegar a tener, además de que ayuda a lograr que el alcance de cada servicio esté bien delimitado.
- **Escalabilidad:** Debido a su independencia es que se puede escalar de manera individual, sea horizontal o verticalmente.

Pero como toda tecnología no está exenta de problemas o complicaciones, esta cuenta con sus propias desventajas [12]:

- **Descomponer en servicios:** No existe un algoritmo concreto y bien definido para descomponer un sistema en servicios. El trabajo de diseño que requiere la etapa de descomposición es extremadamente desafiante ya que un mal diseño podría provocar desaprovechamiento de los beneficios de esta arquitectura.
- **Los sistemas distribuidos son complejos:** Los desarrolladores deben ser capaces de afrontar la complejidad de un sistema distribuido de este tipo, se deben implementar mecanismos de entre cada microservicio, además de diseñarlos sean capaces de manejar fallas parciales y tratar con un servicio remoto ya sea que no esté disponible o que exhiba alta latencia.
- **Decidir cuándo adoptar este enfoque:** Otro problema con el uso de la arquitectura de microservicios es decidir en qué punto durante el ciclo de vida de la aplicación se debe utilizar. Al desarrollar la primera versión de una aplicación, a menudo no se presentan los problemas que esta resuelve, por lo cual el uso de una arquitectura elaborada y distribuida ralentizará el desarrollo. Sin embargo, en etapas más avanzadas, cuando las aplicaciones comienzan a crecer rápidamente y se vuelven más complejas, comienza a tener sentido descomponer funcionalmente la aplicación en un conjunto de microservicios.
- **Consistencia:** Al involucrar una gran cantidad de servicios independientes interactuando entre sí para satisfacer un objetivo en común, sumado a la limitante de no poder aplicar los clásicos protocolos de consistencia como el 2PC hacen de la consistencia de los datos uno de los grandes desafíos a superar.
- **Observación:** El hecho de tener múltiples sistemas desplegados al mismo tiempo complejizan considerablemente la tarea de observación de fallas y seguimiento de la evolución del sistema como un todo.
- **Seguridad:** Cuantos más servicios independientes coexisten mayor es el desafío de la gestión de la seguridad contemplando que se deben aplicar políticas de seguridad que protejan a todos estos servicios

3.2.3.6 Problemas de la Arquitectura de Microservicios

La arquitectura de microservicios provee grandes ventajas al desarrollar una aplicación de gran magnitud incluso para ambientes empresariales de alto estándar. Entre estas ventajas tenemos la escalabilidad, el bajo acoplamiento, el poder crear equipos más pequeños de desarrollo más centrados en un área en concreta del negocio, entre otras ventajas. Pero acarreado a estas ventajas y puntos positivos con los que cuenta, también tenemos grandes problemas pendientes de resolver, todo esto ligado a las desventajas que ya se plantearon anteriormente que hacen de esta una arquitectura con algunos puntos débiles. Entre estos **problemas** podemos mencionar cinco principales:

1. Dificultad técnica superior
    - Hay que lidiar con sistemas distribuidos.
    - La consistencia siempre será eventual.
    - Manejo necesario del canal de comunicación y sus imprevistos.

- Serán necesarias acciones compensatorias con SAGAs.
- Requiere de un equipo maduro y especializado.

2. Compleja gestión de los sistemas y la infraestructura
   - Necesidad de implementar prácticas de integración y despliegue continuo (CI/CD)
   - Alta necesidad de un equipo DevOps

3. Cada microservicio es una barrera de información
   - Múltiples servicios distribuidos
   - Múltiples equipos de desarrollo
   - Dificultad en la gobernanza del sistema como un todo

4. Los costes de operación son superiores
   - Se vuelve dominante casi el uso de proveedores cloud como AWS, Azure y Google Cloud Platform (GCP).
   - Si se usa infraestructura propia debe incrementarse para evitar los problemas de la infraestructura compartida.

5. Alta dificultad para lograr un diseño adecuado
   Es muy difícil definir lo que va en cada servicio, el diseño es muy complejo, en este aspecto es muy recomendado la aplicación de DDD (Enfoque DDD: es un patrón de diseño para la arquitectura de microservicios ya mencionado anteriormente, que se basa en la descomposición por dominio)

3.2 Spring security: autenticación y autorización

Según se ha abordado en las secciones anteriores, está bien tener una arquitectura basada en microservicios que sea capaz de proveer a cualquier aplicación la inmensidad de ventajas que este tipo de arquitectura proporciona. Pero hasta cierto punto estas cumplen con las necesidades de alto estándar que exigen las aplicaciones de hoy en día. Pero, aun así, les falta el punto más importante dentro del mundo tecnológico: "**La Seguridad**".

En el mundo, cada día existen miles de páginas web, servidores web y dispositivos móviles que son el blanco de los llamados **piratas informáticos**. Sitios web que dejan de estar disponibles, información modificada o dañada, filtraciones de direcciones de correo electrónico, tarjetas de crédito, contraseñas, y un sinfín de problemas que tendremos que afrontar si no protegemos correctamente nuestra web. Este **el propósito principal de la seguridad web**: prevenir ataques y proteger sitios web del acceso, uso, modificación, destrucción o interrupción no autorizados.

Entonces, en este apartado tendremos la posibilidad de abarcar varios temas relacionados con la seguridad en las aplicaciones, precisamente en aplicaciones **Java** con **Spring Security.**

3.3.1 ¿Qué es Spring?

Spring, es un **Framework** de código abierto para la creación de aplicaciones **empresariales** Java, con soporte para Groovy y Kotlin; tiene estructura modular y una gran flexibilidad para implementar diferentes tipos de arquitecturas según las necesidades de la aplicación.

En la actualidad (septiembre, 2022), como se comentó anteriormente, Spring cuenta con 22 proyectos activos para diferentes tipos de desarrollo. Incluso, cuentan con otros 12 proyectos que se encuentran como ellos comentan en su web "**Projects in the Attic**", por decirlo de alguna forma, serían proyectos deshechos, a los cuales ya dejaron de darle soporte. En la figura siguiente me muestran los 22 proyectos comentados anteriormente:

Figura 5. Proyectos de Spring [22]

Spring boot: Facilita la creación de aplicaciones independientes basadas en aplicación de producción de Spring que se pueden ejecutar.

Spring Framework: Spring Framework proporciona un modelo integral de programación y configuración para aplicaciones empresariales modernas basadas en Java, en cualquier tipo de plataforma de implementación.

Spring Data: Proporciona un modelo de programación familiar y consistente basado en Spring para el acceso a los datos, al mismo tiempo que conserva los rasgos especiales del almacén de datos subyacentes

Spring Cloud: Proporciona las herramientas necesarias para que los desarrolladores construyan rápidamente algunos de los patrones comunes en sistemas distribuidos, por ejemplo, gestión de configuración, descubrimiento de servicios, interruptores automáticos, enrutamiento inteligente, micro-proxy, bus de control, tokens únicos, bloqueos globales, elección de liderazgo, distribuido sesiones, estado del clúster.

Spring Cloud Data Flow: Proporciona herramientas para crear topologías complejas para transmisión y procesamiento de datos por lotes. El procesamiento y transferencia de datos consisten en aplicaciones Spring Boot, creadas utilizando los marcos de microservicio Spring Cloud Stream o Spring Cloud Task.

Spring Security: Es un Framework de autenticación y autorización potente y altamente personalizable. Se centra en proporcionar autenticación y autorización a las aplicaciones Java. Es el utilizado por defecto para asegurar aplicaciones basadas en Spring y al igual que todos los proyectos de Spring, el verdadero

poder de Spring Security se encuentra en la facilidad con que se puede extender para cumplir con los requisitos personalizados.

Spring for GraphQL: Spring para GraphQL brinda soporte para aplicaciones Spring creadas en GraphQL Java. Es el sucesor del proyecto GraphQL Java Spring del equipo de GraphQL Java. Su objetivo es ser la base para todas las aplicaciones Spring, GraphQL.

Spring Session: Proporciona una API e implementaciones para administrar la información de sesión de un usuario.

Spring Integration: Permite la mensajería ligera dentro de las aplicaciones basadas en Spring y admite la integración con sistemas externos a través de adaptadores declarativos. Esos adaptadores proporcionan un mayor nivel de abstracción sobre el soporte de Spring para la comunicación remota, la mensajería y la programación. El objetivo principal de Spring Integration es proporcionar un modelo simple para construir soluciones de integración empresarial mientras se mantiene la separación de responsabilidades que es esencial para producir código testeable y mantenible.

Spring HATEOAS: Proporciona algunas API para facilitar la creación de servicios REST que siguen el principio HATEOAS cuando se trabaja con Spring y especialmente Spring MVC. HATEOAS es un acrónimo de Hypermedia As The Engine Of Application State (hipermedia como motor del estado de la aplicación), donde de acuerdo con un punto de acceso al API se puede determinar los recursos de acuerdo con la respuesta obtenida por el servidor.

Spring REST Docs: Ayuda a documentar los servicios RESTful. Con este se produce documentación que sea precisa, concisa y bien estructurada.

Spring Batch: Spring Batch proporciona funciones reutilizables que son esenciales en el procesamiento de grandes volúmenes de registros, incluidos el registro de logs, la gestión de transacciones, las estadísticas de procesamiento de tareas, el reinicio de tareas, la omisión y la gestión de recursos. También proporciona servicios y características técnicas más avanzadas que permitirán trabajos por lotes de alto volumen y rendimiento a través de técnicas de optimización y partición.

Spring AMQP: Aplica los conceptos básicos de Spring al desarrollo de soluciones de mensajería basadas en AMQP. Proporciona una "plantilla" como abstracción de alto nivel para enviar y recibir mensajes. Estas librerías facilitan la gestión de los recursos AMQP al tiempo que promueven el uso de inyección de dependencia y configuración declarativa.

Spring CredHub: Proporciona una API para almacenar, generar, recuperar y eliminar credenciales de varios tipos de forma segura.

Spring Flo: Es una biblioteca de JavaScript que ofrece un generador visual HTML5 embebido para automatización de procesos y gráficos simples. Esta biblioteca se usa como la base del generador de flujo en Spring Cloud Data Flow.

Spring for Apache Kafka (spring-kafka): aplica conceptos básicos de Spring al desarrollo de soluciones de mensajería basadas en Kafka. Proporciona una "plantilla" como abstracción de alto nivel para enviar mensajes. También brinda soporte para POJO controlados por mensajes con anotaciones @KafkaListener y un "contenedor de escucha". Estas bibliotecas promueven el uso de inyección de

dependencia y declarativo. En todos estos casos, verá similitudes con la compatibilidad con JMS en Spring Framework y la compatibilidad con RabbitMQ en Spring AMQP.

Spirng LDAP: es una biblioteca para simplificar la programación LDAP en Java, construida sobre los mismos principios que Spring Jdbc.

Spring Shell: Los usuarios del proyecto Spring Shell pueden crear fácilmente una aplicación de shell con todas las funciones (también conocida como línea de comandos) dependiendo de los jars de Spring Shell y agregando sus propios comandos (que vienen como métodos en Spring Beans). Crear una aplicación de línea de comandos puede ser útil, Ej.: para interactuar con la API REST de su proyecto o para trabajar con contenido de archivos locales.

Spring Statemachine: Herramienta para integrar los conceptos de máquinas de estado en las aplicaciones Spring.

Spring Vault: Spring Vault proporciona abstracciones familiares de Spring y soporte del lado del cliente para acceder, almacenar y revocar secretos. Ofrece abstracciones de alto y bajo nivel para interactuar con Vault, lo que libera al usuario de preocupaciones de infraestructura.

Con Vault de HashiCorp, tiene un lugar central para administrar datos secretos externos para aplicaciones en todos los entornos. Vault puede administrar secretos estáticos y dinámicos, como datos de aplicaciones, nombre de usuario/contraseña para aplicaciones/recursos remotos y proporcionar credenciales para servicios externos como MySQL, PostgreSQL, Apache Cassandra, Consul, AWS y más.

Spring Web Flow: Se basa en Spring MVC y permite implementar los "flujos" de una aplicación web. Un flujo encapsula una secuencia de pasos que guían a un usuario a través de la ejecución de alguna tarea comercial. Abarca múltiples solicitudes HTTP, tiene estado, trata con datos transaccionales, es reutilizable y puede ser dinámico y de larga duración.

Spring Web Services (Spring-WS): Es un producto de la comunidad de Spring enfocado en crear servicios web basados en documentos. Tiene como objetivo facilitar el desarrollo del servicio SOAP por contrato, lo que permite la creación de servicios web flexibles utilizando una de las muchas formas de manipular las cargas XML. El producto se basa en Spring, lo que significa que puede utilizar los conceptos de Spring, como la inyección de dependencia, como parte integral de su servicio web.

3.3.2 ¿Qué es Spring Security?

Primeramente, para ponernos en contexto hay que señalar que Spring Security pertenece a una amplia lista de proyectos del Framework **Spring [**22**]** como se pudo apreciar anteriormente. Entonces, **¿Qué es Spring?**

Spring Security es un Framework potente y altamente personalizable enfocado en proporcionar autenticación y autorización a las aplicaciones Java. Es el estándar para proteger las aplicaciones basadas en Spring. Como todos los proyectos de Spring, el verdadero poder de Spring Security se encuentra en la facilidad con la que se puede ampliar para cumplir con los requisitos personalizados.

Características de Spring Security:

- Soporte completo y extensible para autenticación y autorización.
- Protección contra ataques como fijación de sesión, clickjacking, falsificación de solicitudes entre sitios, etc.
- Integración de API servlet.
- Integración opcional con Spring Web MVC.

3.2.3 Conceptos principales

Antes de entrar en detalles sobre Spring Security, es necesario conocer dos conceptos principales dentro de este Framework:

1. Autenticación
2. Autorización
3. Filtros de Servlets

1. Autenticación

En primer lugar, si está ejecutando una aplicación (web) típica, necesita que sus usuarios se autentiquen. Eso significa que su aplicación necesita verificar si el usuario es *quien* dice ser, lo que generalmente se hace con una verificación de nombre de usuario y contraseña.

2. Autorización

En aplicaciones más simples, la autenticación puede ser suficiente: tan pronto como un usuario se autentica, puede acceder a todas las partes de una aplicación.

Pero la mayoría de las aplicaciones tienen el concepto de permisos (o roles). Imagínese: clientes que tienen acceso a la interfaz pública de su tienda web y administradores que tienen acceso a un área de administración separada.

Ambos tipos de usuarios necesitan iniciar sesión, pero el mero hecho de la autenticación no dice nada sobre lo que pueden hacer en su sistema. Por lo tanto, también debe verificar los permisos de un usuario autenticado, es decir, debe autorizar al usuario.

3. Filtros de Servlets

Por último, pero no menos importante, tenemos los filtros de servlet. ¿Qué tienen que ver con la autenticación y la autorización?

¿Por qué usar filtros de servlet?

Cualquier aplicación web de Spring es solo un servlet: el viejo DispatcherServlet de Spring, que redirige las solicitudes HTTP entrantes (por ejemplo, desde un navegador) a sus @Controllers o @RestControllers.

La cuestión es que no hay seguridad codificada en ese DispatcherServlet y es muy probable que tampoco se ande con un encabezado HTTP Basic Auth sin procesar en sus @Controllers. De manera óptima, la autenticación y la autorización deben realizarse antes de que una solicitud llegue a sus @Controllers.

Afortunadamente, hay una manera de hacer exactamente esto en el mundo web de Java: puede poner filtros delante de los servlets, lo que significa que podría pensar en escribir un SecurityFilter y configurarlo en su Tomcat (contenedor de servlet/servidor de aplicaciones) para filtrar cada entrada Solicitud HTTP antes de que llegue a su servlet.

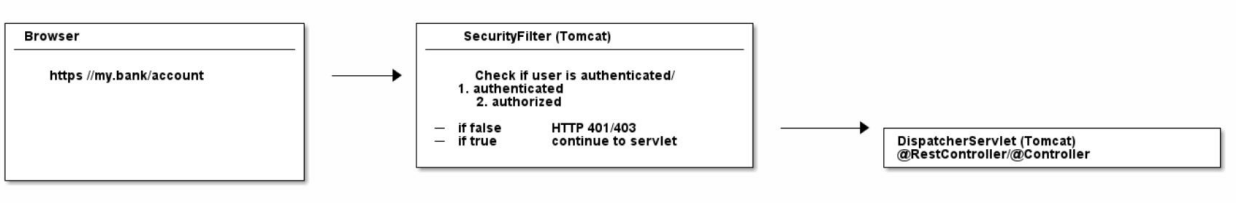


Figura 6. Filtro entre una petición http y un servlet

Un filtro de seguridad (SecurityFilter) trae aproximadamente 4 tareas definidas a continuación. Una implementación sencilla podría verse como en la figura más abajo:

1. Primero, el filtro necesita extraer un nombre de usuario/contraseña de la solicitud. Podría ser a través de un encabezado HTTP de autenticación básica , o campos de formulario, o una cookie, etc.
2. Luego, el filtro necesita validar esa combinación de nombre de usuario/contraseña contra algo, como una base de datos.
3. El filtro debe verificar, después de una autenticación exitosa, que el usuario está autorizado para acceder al URI solicitado.
4. Si la solicitud sobrevive a todas estas comprobaciones, entonces el filtro puede permitir que la solicitud pase a su DispatcherServlet, es decir, a sus @Controllers.

```
import javax.servlet.*;
import javax.servlet.http.HttpFilter;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.IOException;

public class SecurityServletFilter extends HttpFilter {

    @Override
    protected void doFilter(HttpServletRequest request, HttpServletResponse response, FilterChain
chain) throws IOException, ServletException {

        UsernamePasswordToken token = extractUsernameAndPasswordFrom(request);  // (1)

        if (notAuthenticated(token)) {  // (2)
            // username y password incorrecto
            // desafortunadamente, el código de estado HTTP se llama "no autorizado", en lugar de "no
               autenticado"
            response.setStatus(HttpServletResponse.SC_UNAUTHORIZED); // HTTP 401.
            return;
        }

        if (notAuthorized(token, request)) { // (3)
            // ha iniciado sesión, pero no tiene los derechos adecuados
            response.setStatus(HttpServletResponse.SC_FORBIDDEN); // HTTP 403
            return;
        }

        // permite que la solicitud Http vaya a Spring DispatcherServlet
        // y a @RestControllers/@Controllers.
        chain.doFilter(request, response); // (4)
    }

    private UsernamePasswordToken extractUsernameAndPasswordFrom(HttpServletRequest request) {
        // Intente leer un encabezado HTTP de autenticación básica, que viene en forma de usuario:
           contraseña
        // O intenta encontrar parámetros de solicitud de inicio de sesión de formulario o cuerpos
           POST, es decir, "nombre de usuario = yo" y "contraseña = "myPass"
        return checkVariousLoginOptions(request);
    }


    private boolean notAuthenticated(UsernamePasswordToken token) {
        // compara el token con lo que tiene en su base de datos... o en la memoria... o en LDAP...
        return false;
    }

    private boolean notAuthorized(UsernamePasswordToken token, HttpServletRequest request) {
       // comprueba si el usuario actualmente autenticado tiene el permiso para acceder al URI
          de esta solicitud
       // ejemplo: /admin necesita el rol ROLE_ADMIN , /callcenter necesita el rol ROLE_CALLCENTER.
       return false;
    }
}
```

Figura 7. Filtro de seguridad utilizando spring security

FilterChains

El código anterior funciona y compila, pero a largo plazo conduciría a un filtro inmenso con gran cantidad de líneas de código para varios mecanismos de autenticación y autorización. Sin embargo, en el mundo real, dividiría este filtro en varios filtros, que luego se encadenan [23]. Ver la figura siguiente.

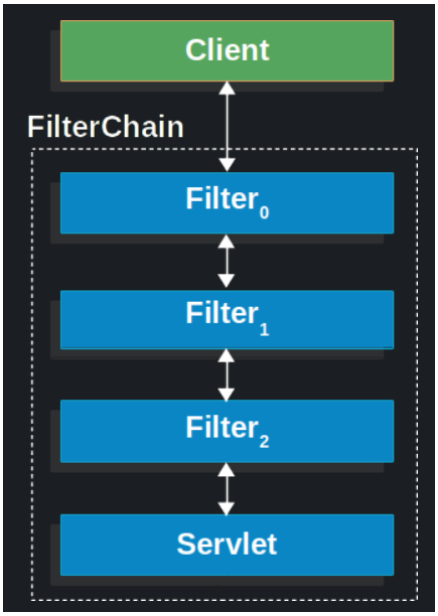


Figura 8. FilterChain [23]

Por ejemplo, una solicitud HTTP entrante sería...

- Primero, va a través de un LoginMethodFilter
- Luego, pasa por un AuthenticationFilter
- Luego, pasa por un AuthorizationFilter
- Finalmente, va a su servlet.

Con un filtro (cadena) de este tipo, básicamente puede manejar todos los problemas de autenticación o autorización que haya en su aplicación, sin necesidad de cambiar la implementación real de su aplicación (piense: sus @RestControllers / @Controllers).

3.3.4 Filterchain en Spring Security

Al iniciar una aplicación que tenga en el archivo pom.xml la dependencia de Spring Security se puede apreciar en la ejecución en consola del proyecto que no solo se instala un filtro, sino varios. Realmente se instala una cadena de filtros completa que consta de 15 filtros diferentes.

Entonces, cuando se hace una solicitud HTTP, pasará por todos estos 15 filtros, antes de que su solicitud finalmente llegue a sus @RestControllers. El orden también es importante, comenzando en la parte superior de esa lista y bajando hasta el final.

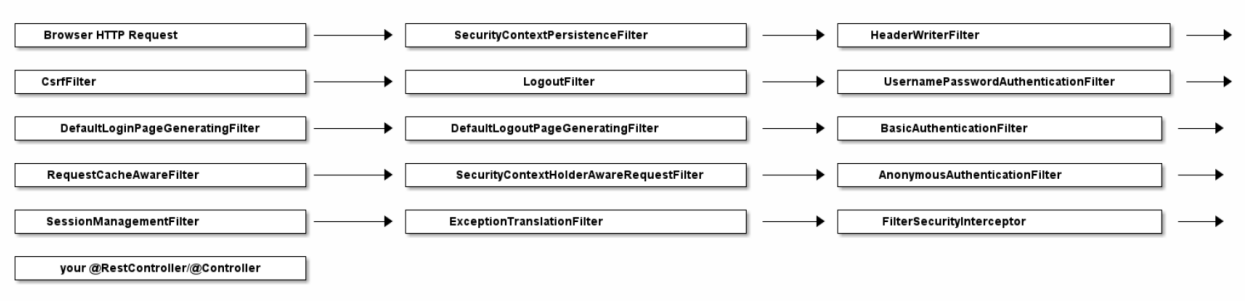


Figura 9. Filtros de Spring Security

A continuación de detallan algunos de estos filtros mencionados anteriormente

- **BasicAuthenticationFilter**: intenta encontrar un encabezado HTTP de autenticación básica en la solicitud y, si lo encuentra, intenta autenticar al usuario con el nombre de usuario y la contraseña del encabezado.
- **UsernamePasswordAuthenticationFilter**: intenta encontrar un parámetro de solicitud de nombre de usuario/contraseña POST y, si lo encuentra, intenta autenticar al usuario con esos valores.
- **DefaultLoginPageGeneratingFilter**: genera una página de inicio de sesión para usted, si no deshabilita explícitamente esa característica. Este filtro es la razón por la que obtiene una página de inicio de sesión predeterminada al habilitar Spring Security.
- **DefaultLogoutPageGeneratingFilter**: genera una página de cierre de sesión para usted, si no deshabilita explícitamente esa función.

Con estos filtros y muchos más, Spring Security le brinda una página de inicio/cierre de sesión, así como la capacidad de iniciar sesión con autenticación básica o inicios de sesión de formulario, así como un par de ventajas adicionales como CsrfFilter.

Spring Security se resume principalmente a estos filtros. Ellos hacen todo el trabajo. Simplemente falta configurar como cada filtro hace su trabajo, es decir, qué URL proteger, cuáles ignorar y qué tablas de base de datos usar para la autenticación.

3.3.5 Como configurar Spring Scurity: WebSecurityConfigurerAdapter

Con las últimas versiones de Spring Security o Spring Boot, la forma de configurar Spring Security es tener una clase que, esté anotada con @EnableWebSecurity y que extienda de WebSecurityConfigurerAdapter, que básicamente le ofrece una configuración DSL/métodos. Con esos métodos, puede especificar qué URI en su aplicación proteger o qué protecciones contra vulnerabilidades habilitar/deshabilitar. Así es como se ve una clase con estas características:

```java
@Configuration
@EnableWebSecurity // (1)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter { // (1)

  @Override
  protected void configure(HttpSecurity http) throws Exception {  // (2)
      http
        .authorizeRequests()
          .antMatchers("/", "/home").permitAll() // (3)
          .anyRequest().authenticated() // (4)
          .and()
       .formLogin() // (5)
          .loginPage("/login") // (5)
          .permitAll()
          .and()
      .logout() // (6)
        .permitAll()
        .and()
      .httpBasic(); // (7)
  }
}
```

Figura 10. Clase configurada a través de WebSecurityConfigurerAdapter

1. Una clase que tiene anotaciones @Configuration y @EnableWebSecurity, que extiende de WebSecurityConfigurerAdapter.
2. Al anular el método configure (HttpSecurity) del adaptador, obtiene un pequeño y agradable DSL con el que puede configurar su FilterChain.
3. Todas las solicitudes dirigidas a "/" y "*/home*" están permitidas: el usuario *no* tiene que autenticarse. Está usando un antMatcher, lo que significa que también podría haber usado comodines (*, **,?) en la cadena.
4. Cualquier otra solicitud necesita que el usuario se autentique *primero*, es decir, el usuario debe iniciar sesión.
5. Está permitiendo el inicio de sesión de formulario (nombre de usuario/contraseña en un formulario), con una página de inicio de sesión personalizada (/*login*, es decir, no la generada automáticamente por Spring Security). Cualquiera debería poder acceder a la página de inicio de sesión, sin tener que iniciar sesión primero (método permitAll).
6. Lo mismo ocurre con la página de cierre de sesión.
7. Además de eso, también está permitiendo la autenticación básica, es decir, enviando un encabezado de autenticación básica HTTP para autenticar.

Teniendo en cuenta que podemos crear una clase que extienda de WebSecurityConfigurerAdapter, ahora toca el turno de analizar cómo se implementa esta clase abstracta.

Figura 11. Clase abstracta WebSecurityConfigurerAdapter

1. Se define qué URL proteger, en este caso todas las solicitudes serán autenticadas.
2. Qué métodos de autenticación están permitidos (formLogin(), httpBasic()) y cómo están configurados.
3. La completa configuración de seguridad de tu aplicación.

La configuración predeterminada de esta clase abstracta es la razón por la cual las aplicaciones inician con restricciones al hacer las solicitudes tan pronto como se agrega Spring Security a un proyecto Java.

3.3.6 Autenticación con Spring Security

Cuando se habla de autenticación en Spring Security, se mencionan tres escenarios importantes:

1. **Predeterminado:** Se puede acceder a la contraseña del usuario (hash), porque tiene sus datos (nombre de usuario y contraseña) almacenados en una tabla de la base de datos.
2. **Menos común:** No se puede acceder a la contraseña del usuario (hash), debido a que los datos del usuario se almacenan en otro lugar, como en un idp (identity management product) de terceros que ofrece servicios REST para la autenticación.
3. **Más popular:** Utiliza OAuth2 o OpenID en combinación con tokens JWT. Se explicará en una sección más adelante.

Según el escenario, se deben especificar diferentes @Beans para que Spring Security funcione como es debido.

UserDetailsService: Tener acceso a la contraseña del usuario

Poniendo como ejemplo que se cuenta con una base de datos donde se almacenan todos los usuarios. Tabla en la que se cuentan con detalles del usuario, pero lo más importante es que la tabla cuenta con

dos columnas (nombre de usuario y contraseña) principales. En este caso, Spring Security necesita que defina dos beans para que la autenticación esté en funcionamiento.

1. Bean de UserSetailsService
2. Bean de PasswordEncoder

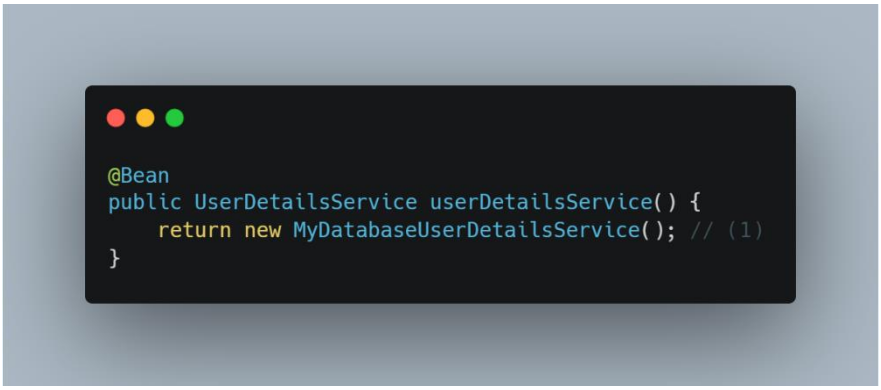


Figura 12. Bean UserDetailsService

Este Bean retorna la clase "MyDatabaseUserDetailsService". Esta clase que se retorna, como se muestra a continuación implementa la interfaz UserDetailsService

Figura 13. Clase MyDatabaseUserDetailsService implementando la interfaz UserDetailsService

Explicando lo mostrado en la figura anterior, decimos lo siguiente:

1. Un UserDetailsService carga UserDetails a través del nombre de usuario del usuario. Tenga en cuenta que el método toma solo un parámetro: nombre de usuario (no la contraseña).
2. La interfaz UserDetails tiene métodos para obtener la contraseña y uno para obtener el nombre de usuario.

3. UserDetails tiene incluso más métodos, como: ver si la cuenta de usuario está activa o bloqueada, si las credenciales han expirado o qué permisos tiene el usuario.

Por lo tanto, puede implementar estas interfaces o usar las existentes que proporciona Spring Security.

3 **JdbcUserDetailsManager**, que es un UserDetailsService basado en JDBC (base de datos). Puede configurarlo para que coincida con la estructura de la tabla/columna del usuario.

4 **InMemoryUserDetailsManager**, que mantiene todos los detalles del usuario en la memoria y es ideal para realizar pruebas.

5 **org.springframework.security.core.userdetail.User** , que es una implementación sensata y predeterminada de UserDetails que podría usar. Eso significaría mapear/copiar entre sus entidades/tablas de base de datos y esta clase de usuario. Una alternativa podría ser que sus entidades implementen la interfaz UserDetails.

Ahora, cómo sería el flujo completo de autenticación básica HTTP, usando Spring Security y Basic Auth. Primero se extrae el usuario y la contraseña del encabezado HTPP en un filtro. Luego se llama a la clase de ejemplo "*MyDatabaseUserDetailsService*" que se mostró anteriormente, para cargar el usuario correspondiente de la base de datos, evaluado como un objeto UserDetails. Luego se toma la contraseña extraída del encabezado HTTP Basic Auth, se le hace un hash automáticamente a esta contraseña y se compara con la contraseña (es un hash la contraseña de UserDetails) que se obtuvo en el objeto UserDetails. Si ambos hashes coinciden, el usuario se autentica correctamente.

Por esa parte todo está correcto. Pero **¿Cómo Spring Security codifica la contraseña que se toma de la cabecera? ¿Con qué algoritmo?**

Codificar la contraseña (PasswordEncoder)

Spring Security no puede adivinar mágicamente su algoritmo de hash de contraseña preferido. Es por lo que necesita especificar otro @Bean, un PasswordEncoder. Si se desea, por ejemplo, utilizar la función de hashing de contraseñas de BCrypt (predeterminada de Spring Security) para todas las contraseñas, se debe especificar este @Bean en su SecurityConfig.

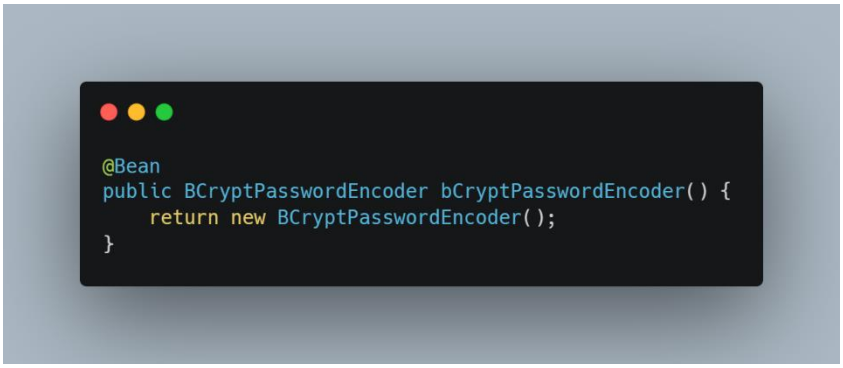


Figura 14. Bean para codificar las contraseñas con BCrypt

¿Qué sucede si tiene varios algoritmos de hashing de contraseñas, porque tiene algunos usuarios heredados cuyas contraseñas se almacenaron con MD5 (recomendable no utilizarlo) y otros más nuevos con Bcrypt o incluso un tercer algoritmo como SHA-256? Entonces, lo idóneo sería utilizar el siguiente bean:

Figura 15. Bean para delegar el algoritmo hash

¿Cómo funciona este DelegatingPasswordEncoder? Verá la contraseña codificada de UserDetails (procedente, por ejemplo, de la tabla de su base de datos), que debe comenzar con un prefijo así: {prefix}. Ese prefijo es tu método hash. Donde en la columna de contraseñas de la base de datos se mostraría cada hash con un prefijo delante. Ejemplo:

username	password
juan@gmail.com	{bcrypt}$2y$12$6t86Rpr3llMANhCUt26oUen2WhvXr/A89Xo9zJion8W7gWgZ/zA0C
pedro@gmail.com	{sha256}5ffa39f5757a0dad5dfada519d02c6b71b61ab1df51b4ed1f3beed6abe0ff5f6

Tabla 1. Usuarios y claves de ejemplo

Spring Security toma el hash y elimina el prefijo ({bcrypt} o {sha256}), según el valor de prefijo utiliza el PasswordEncoder correcto que puede ser BCryptEncoder o SHA256Encoder. Hace un hash a la contraseña entrante con ese PasswordEncoder, que proviene de la cabecera HTTP, y la compara con la almacenada en base de datos.

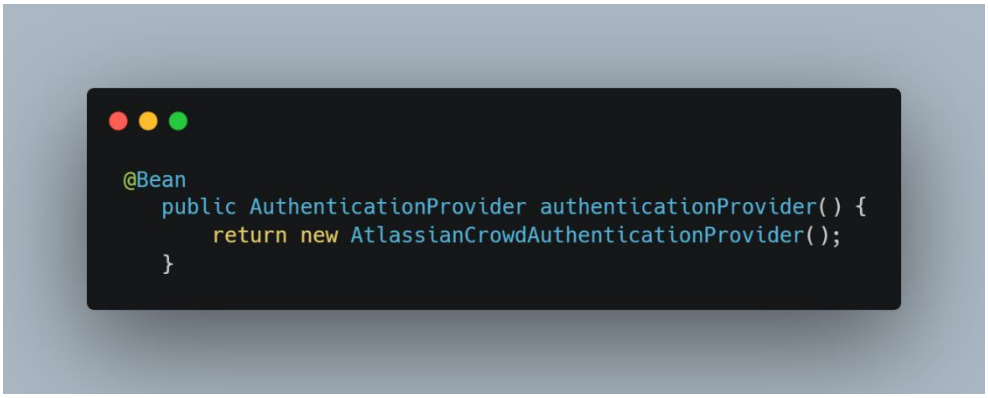
AuthenticationProvider: No tener acceso a la contraseña del usuario

Ahora, utilizando como ejemplo Atlassian Crowd para la gestión centralizada de identidades. Eso significa que todos sus usuarios y contraseñas se almacenan en Atlassian Crowd y ya no en la tabla de su base de datos.

Esto tiene dos implicaciones:

1. Ya no se tienen las contraseñas de usuario en una aplicación, ya que no se puede pedirle a Crowd esas contraseñas.
2. Sin embargo, tiene una API REST con la que puede iniciar sesión, con su nombre de usuario y contraseña. (Una solicitud POST al endpoint "/rest/usermanagement/1/authentication").

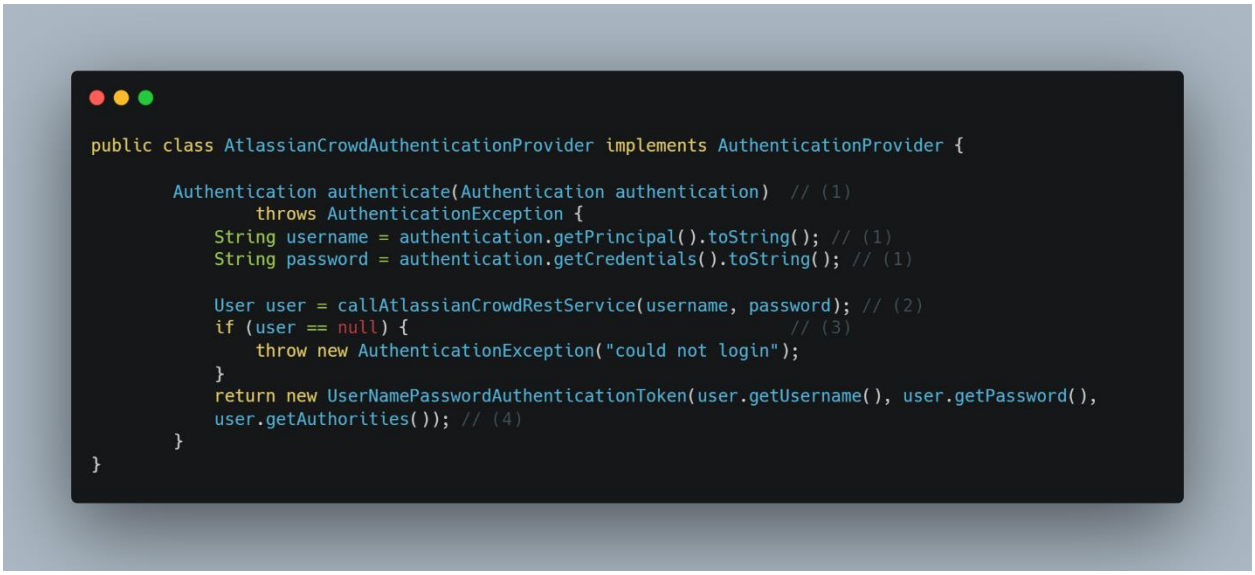
Para este caso ya no puede usar un UserDetailsService, sino que se debe implementar y proporcionar un **AuthenticationProvider** @Bean como el siguiente:



```
@Bean
   public AuthenticationProvider authenticationProvider() {
       return new AtlassianCrowdAuthenticationProvider();
   }
```

Figura 16. Bean de AuthenticationProvider

Una implementación personalizada de la interfaz AuthenticationProvider puede verse de la siguiente forma:



```
public class AtlassianCrowdAuthenticationProvider implements AuthenticationProvider {

        Authentication authenticate(Authentication authentication)  // (1)
                throws AuthenticationException {
            String username = authentication.getPrincipal().toString(); // (1)
            String password = authentication.getCredentials().toString(); // (1)

            User user = callAtlassianCrowdRestService(username, password); // (2)
            if (user == null) {                                          // (3)
                throw new AuthenticationException("could not login");
            }
            return new UsernamePasswordAuthenticationToken(user.getUsername(), user.getPassword(),
            user.getAuthorities()); // (4)
        }
}
```

Figura 17. Implementación personalizada de la interfaz AuthenticationProvider

Detallando un poco que sucede en esta implementación tenemos:

1. En comparación con el método load() de UserDetails, donde solo tenía acceso al nombre de usuario, ahora tiene acceso al intento de autenticación completo, que generalmente contiene un nombre de usuario y una contraseña.
2. Se puede hacer lo que estime conveniente para autenticar al usuario, por ejemplo, llamar a un servicio REST.
3. Si la autenticación falla, debe lanzar una excepción.
4. Si la autenticación tuvo éxito, debe devolver un UsernamePasswordAuthenticationToken completamente inicializado. Es una implementación de la interfaz de autenticación y necesita que el campo autenticado se establezca en verdadero (que el constructor utilizado anteriormente establecerá automáticamente).

Con esto podemos definir un flujo completo de lo que sería la autenticación en Spring Security a través de un proveedor, donde no tiene concretamente su contraseña.

Se extrae el nombre de usuario y la contraseña del encabezado como se hace hasta ahora, a través de HTTP Basic Auth en un filtro. Luego se llama al Authetication Provider con el nombre de usuario de la cabecera para poder realizar la autenticación (Ej.: una llama REST). Con los datos obtenidos de la llamada REST se procede a hacer la autenticación o verificación de las credenciales.

3.3.7 Autorización con Spring Security

Hasta ahora, solo se han abordado temas de autenticación, por ejemplo, verificación de nombre de usuario y contraseña. Pero cuando se habla de que permisos tiene determinado usuario dentro de la aplicación y a que recursos puede acceder en dependencia de esos permisos, pues entonces se habla del término autorización.

Cuando ya contamos con un sitio web de gran magnitud (Ej.: una tienda online de suplementos), ya sería recomendable no solo aplicar el concepto de autenticación, sino también de autorización. Debido a que el sitio puede estar dividido en varias secciones, hay que permitir un acceso diferenciado para los usuarios, según sus autoridades o roles.

¿Qué son las Autoridades? ¿Qué son los roles?

De forma simple, las Autoridades (Authorities) vienen siendo como un "permiso" o "derecho". Esos permisos se expresan normalmente como cadenas (con el método getAuthority()). Esas permiten identificar los permisos y dejar que un determinado usuario acceda o no a determinada parte de la aplicación [24].

Por otra parte, tenemos el término de roles, en Spring Security un rol no es más que un "permiso" con la convención de nomenclatura que dice que es una autoridad que comienza con el prefijo ROLE_, Ejemplo: ROLE_ADMIN [24].

¿Qué son las SimpleGrantedAuthority?

SimpleGrantedAuthority, no es más que una clase que implementa la interfaz GrantedAuthority, en la cual se almacena un String para representar la autoridad. También hay otras clases de autoridad que le permiten almacenar objetos adicionales, no simplemente un String. Esta se visualizará de la siguiente forma:

```
public final class SimpleGrantedAuthority implements GrantedAuthority {

    private final String role;

    @Override
    public String getAuthority() {
        return role;
    }
}
```

Figura 18. Implementación de la clase SimpleGrantedAuthority

Según lo visto en las secciones anteriores tenemos dos escenarios importantes, pero esta vez orientado a temas de autorización:

1. UserDetailsService: ¿Dónde almacenar y obtener autorizaciones?

Suponiendo que se está almacenando los usuarios en la propia aplicación, dentro de una tabla de usuarios. Simplemente se agrega a la tabla de usuarios, una nueva columna llamada authorities. Incluso también se puede tener una tabla Authorities completamente separada. Sucede que estas autoridades comienzan con el prefijo ROLE_, por lo que, en términos de Spring Security, estas autoridades también son roles.

username	password	authorities
juan@gmail.com	{bcrypt}…	ROL_ADMIN
pedro@gmail.com	{sha256} …	ROL_CONSULTANT

Tabla 2. Tabla de usuarios con permisos

Ajustando el UserDetailsService se puede hacer una lectura de la columna authorities. Como se puede apreciar a continuación:

```
public class MyDatabaseUserDetailsService implements UserDetailsService {

  UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
     User user = userDao.findByUsername(username);
     List<SimpleGrantedAuthority> grantedAuthorities = user.getAuthorities().map(authority -> new
         SimpleGrantedAuthority(authority)).collect(Collectors.toList()); // (1)
     return new org.springframework.security.core.userdetails.User(user.getUsername(),
         user.getPassword(), grantedAuthorities); // (2)
  }

}
```

Figura 19. Obteniendo las authorities de la tabla de usuarios

Se obtiene lo que está en la columna authorities de la tabla de usuarios (punto 1 de la imagen anterior) y se almacena en una lista de SimpleGrantedAuthority. Posteriormente se usa la implementación base de Spring Security de UserDetails (punto 2 de la imagen anterior) o se podría crear una clase propia implementando UserDetails, para tener los valores del usuario en cuestión.

2. AuthenticationManager: ¿Dónde almacenar y obtener autorizaciones?

Cuando los usuarios provienen de una aplicación de terceros, como Atlassian Cloud, deberá averiguar qué concepto para respaldar a las autoridades. Atlassian Crowd tenía los conceptos de "roles", pero actualmente está obsoleto (deprecated) y ahora cuentan con "grupos" en vez de roles. Entonces, dependiendo del producto real que esté utilizando, debe asignarlo a una autoridad de Spring Security, en su AuthenticationProvider.

```
public class AtlassianCrowdAuthenticationProvider implements AuthenticationProvider {

    public Authentication authenticate(Authentication authentication)
            throws AuthenticationException {
        String username = authentication.getPrincipal().toString();
        String password = authentication.getCredentials().toString();

        atlassian.crowd.User user = callAtlassianCrowdRestService(username, password); // (1)
        if (user == null) {
            throw new AuthenticationException("could not login");
        }
        return new UserNamePasswordAuthenticationToken(user.getUsername(), user.getPassword(),
            mapToAuthorities(user.getGroups())); // (2)
    }
}
```

Figura 20. Obteniendo las authorities de aplicaciones de terceros

Se autentica en un servicio REST y obtiene un objeto de usuario JSON, que luego se convierte en un objeto atlassian.crowd.User. Ese usuario puede ser miembro de uno o más grupos, que aquí se supone que son solo cadenas de autoridades. Luego, simplemente puede asignar estos grupos a "SimpleGrantedAuthority" de Spring.

WebSecurityConfigurerAdapter para las authorities

Hasta el momento se describe cómo almacenar y recuperar autoridades para usuarios autenticados en Spring Security. Pero ¿Cómo se pueden proteger las diferentes URL con las diferentes authorities en Spring Security?

Figura 21. Proteger las URL con las diferentes authorities (hasAuthority/hasAnyAuthority)

Para acceder a la ruta /admin, el usuario tiene que estar autenticado y tener la autoridad (una cadena simple) ROLE_ADMIN (punto 1). Para acceder a la ruta /consulttant, el usuario tiene que estar autenticado Y tener la autoridad ROLE_ADMIN O ROLE_CONSULTANT (punto 2). Para cualquier otra solicitud, no necesita un rol específico, pero aún debe estar autenticado (punto 3).

Teniendo en cuenta el código que se muestra en la figura anterior (en el punto 1 y 2), es equivalente a lo siguiente:

Figura 22. Equivalencia de hasAnyAuthority con hasRole/hasAnyRole

En lugar de llamar a "hasAuthority", ahora llama a "hasRole" para un solo rol, y en lugar de llamar "hasAnyAuthority", ahora llama a "hasAnyRole".

hasAccess y SpEL (Spring Expression Language)

Como aspecto importante en Spring Security, es que cuenta con una forma poderosa de configurar autorizaciones es con el método de "acceso". Le permite especificar prácticamente cualquier expresión SpEL válida.

Figura 23. SpEL usando el método access de Sprint Security

Donde en el ejemplo anterior se verifica que el usuario tenga ROLE_ADMIN, con una dirección IP específica y una verificación de bean personalizada.

3.3.8 OAuth con el proyecto Spring Security

Spring Security, es un Framework que se enfoca en proporcionar autenticación y autorización a las aplicaciones Java. Su principal punto fuerte es la facilidad con la que se puede ampliar para cumplir con requisitos personalizados.

¿Qué sucede con OAuth y Spring Security?

El equipo de Spring, desde hace años ha estado trabajando en un proyecto llamado: Spring Security Oauth, capaz de implementar el estándar de OAuth. Este proyecto fue cogiendo auge e incluso fue teniendo gran aceptación por la comunidad de desarrolladores y grandes empresas, posibilitando que se pudiera implementar este protocolo de OAuth con dicho proyecto. Incluía soporte para los diferentes roles de OAuth como: servidor de recursos y el servidor de autorización. Hoy en día muchos de los proyectos cuentan con la implementación de esta dependencia dentro de sus archivos.

En enero de 2018, Spring anunció que el proyecto (Spring Security OAuth) estaría oficialmente en modo de mantenimiento. Posteriormente, en noviembre de 2019, proporcionaron una actualización (Spring Security OAuth 2.0 Roadmap), que indicaba que la línea 2.3.x llegará al final de su vida útil en marzo de 2020. Las ramas de la versión admitida para ese entonces eran 2.4.x y 2.5.x, con el lanzamiento de 2.5.0 programado para mayo de 2020, que sería el lanzamiento secundario final.

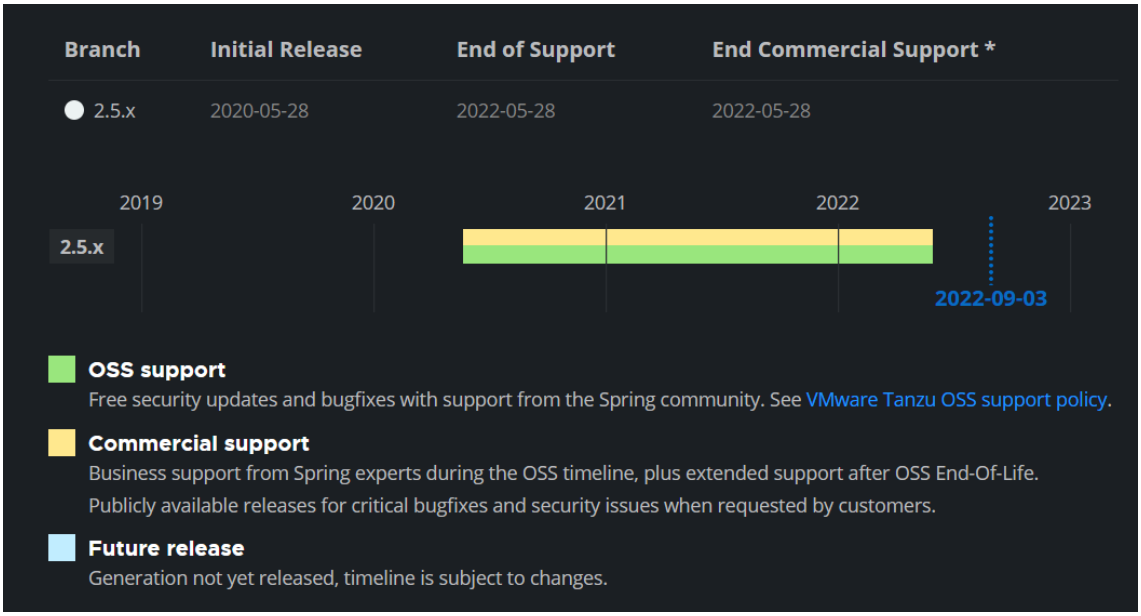


Figura 24. Soporte del proyecto Spring Security OAuth

Con ese fin, el plan fue proporcionar parches y correcciones de seguridad para la línea 2.4.x y 2.5.x hasta mayo de 2021. Además, se admitieron correcciones de seguridad para la línea 2.5.x hasta mayo de 2022, momento en el cual el proyecto llegaría al final de su ciclo de vida (Deprecated). La documentación oficial para este proyecto se eliminó, no se encuentra actualmente en la página de spring.io. No solo se eliminó la documentación, sino que el repositorio del proyecto también se migró al repositorio del proyecto caducado de Spring "spring-attic" y se marcó como de solo lectura. Finalmente, a partir de mayo de 2022, Spring tiene la intención de "abrir" el proyecto, para que pueda ser administrado directamente por los miembros de la comunidad.

A razón de la aceptación de este proyecto, Spring comenzó a desarrollar hace ya tiempo una nueva solución como parte de la versión 5 en adelante de Spring Security. Esta solución se integrará junto con los componentes del protocolo OAuth (Cliente, Servidor de recursos) que ya forman parte de Spring Security. Este proyecto se llama Spring Authorization Server y se encuentra actualmente en la versión 0.3.1. Tendrá soporte oficial por parte del equipo de Spring. Este proyecto es un framework que proporciona implementaciones de las especificaciones OAuth 2.1 y OpenID Connect 1.0 y otras especificaciones relacionadas. Está construido sobre Spring Security para proporcionar una base segura, liviana y personalizable para crear proveedores de identidad OpenID Connect 1.0 y productos de servidor de autorización OAuth2. Spring Authorization Server soporta las siguientes características [25].

Category	Feature	Related specifications

Authorization Grant	• Authorization Code
	  ◦ User Consent
	• Client Credentials
	• Refresh Token
		• The OAuth 2.1 Authorization Framework (draft)
		  ◦ Authorization Code Grant
		  ◦ Client Credentials Grant
		  ◦ Refresh Token Grant
		• OpenID Connect Core 1.0 (spec)
		  ◦ Authorization Code Flow

Token Formats	• Self-contained (JWT) • Reference (Opaque)	• JSON Web Token (JWT) (RFC 7519) • JSON Web Signature (JWS) (RFC 7515)
Client Authentication	• `client_secret_basic` • `client_secret_post` • `client_secret_jwt` • `private_key_jwt` • `none` (public clients)	• The OAuth 2.1 Authorization Framework (Client Authentication) • JSON Web Token (JWT) Profile for OAuth 2.0 Client Authentication (RFC 7523) • Proof Key for Code Exchange by OAuth Public Clients (PKCE) (RFC 7636)

Protocol Endpoints	• OAuth2 Authorization Endpoint
	• OAuth2 Token Endpoint
	• OAuth2 Token Introspection Endpoint
	• OAuth2 Token Revocation Endpoint
	• OAuth2 Authorization Server Metadata Endpoint
	• JWK Set Endpoint
	• OpenID Connect 1.0 Provider Configuration Endpoint
	• OpenID Connect 1.0 UserInfo Endpoint
	• OpenID Connect 1.0 Client Registration Endpoint
		• The OAuth 2.1 Authorization Framework (draft)
		  ◦ Authorization Endpoint
		  ◦ Token Endpoint
		• OAuth 2.0 Token Introspection (RFC 7662)
		• OAuth 2.0 Token Revocation (RFC 7009)
		• OAuth 2.0 Authorization Server Metadata (RFC 8414)
		• JSON Web Key (JWK) (RFC 7517)
		• OpenID Connect Discovery 1.0 (spec)
		  ◦ Provider Configuration Endpoint
		• OpenID Connect Core 1.0 (spec)
		  ◦ UserInfo Endpoint
		• OpenID Connect Dynamic Client Registration 1.0 (spec)
		  ◦ Client Registration Endpoint
		  ◦ Client Configuration Endpoint

Figura 25. Características de Spring Authorization Server

Con este nuevo proyecto (Spring Authorization Server) que ya se ha estado trabajando por parte de Spring hace unos años atrás y los módulos o características de Spring Security, se cuenta con todo lo necesario para implementar el protocolo OAuth, del cual se estará abordando en secciones posteriores.

3.4 Protocolo de autorización OAuth

3.4.1 ¿Qué es OAuth?

OAuth (Open Authorization) es un estándar abierto que permite flujos simples de autorización para sitios web o aplicaciones informáticas. Se trata de un protocolo propuesto por Blaine Cook y Chris Messina, que permite autorización segura de una API de modo estándar y simple para aplicaciones de escritorio, móviles y web [26].

Permite compartir la información de un usuario que pertenece a una aplicación con otra aplicación cliente sin la necesidad de compartir su identidad. Es la forma idónea de interactuar con datos protegidos del usuario. Este protocolo es usado hoy en día por las grandes compañías como Google, Facebook, Microsoft, Twitter y GitHub, permitiendo a los usuarios compartir la información de sus cuentas con otros aplicaciones sin que estos sepan su contraseña.

Utiliza diferentes flujos de autenticación, como el flujo de código de autorización, el flujo de propietario de la contraseña, el flujo implícito, entre otros, así como la extensión de flujos, que también nos permiten definir nuevos flujos.

3.4.2 ¿Por qué surge OAuth?

Este protocolo o estándar abierto surge para solventar la necesidad que se establece del **envío continuo de credenciales entre cliente y servidor**.

Si nos planteamos desarrollar una API REST y una aplicación cliente que pueda consumir nuestros servicios, si queremos tener un mínimo de mecanismos de seguridad, antes de OAuth2 necesitábamos que el usuario nos enviará las credenciales. Si la aplicación cliente es nuestra y la API REST también, posiblemente no existan grandes dificultades. El problema está cuando se quiere hacer una integración con aplicaciones de terceros, ahí es donde aparece la dificultad.

Si queremos que nuestra aplicación sea capaz de generar contenido para Twitter y que aparezca en el timeline de un usuario, tendríamos que pedirle el usuario y contraseña al mismo para poder hacer este proceso, y la verdad es que no es algo razonable ni aceptable. Con OAuth2 el usuario delega la capacidad de realizar ciertas acciones, no todas, a las cuales da su consentimiento para hacerlas en su nombre.

Podríamos delegar en una aplicación como Runtastic, la posibilidad de escribir por nosotros en Twitter cada vez que terminemos una carrera. De esta forma, la aplicación no tiene por qué tener nuestras credenciales de Twitter, pero le permitiremos hacer algo en Twitter por nosotros. De esta forma, cuando se desarrolla una aplicación **no hay por qué almacenar el nombre de usuario y contraseña de este**, sin embargo, vamos a poder hacer algún determinado conjunto de acciones en su nombre.

3.4.3 Casos de uso OAuth

Para comprender cómo funciona OAuth en un ambiente real podemos explicarlo con cierta brevedad en este apartado. Suponiendo que el usuario tiene una cuenta en Google, Facebook o Twitter, y quiere acceder a otra plataforma, pero sin tener que registrarse. En este caso esa aplicación puede pedir al usuario que se autentique por mediación de Google, Facebook o Twitter, y si considera que la información a la que le da acceso es correcta, le podría permitir realizar un conjunto de acciones dentro de su plataforma.

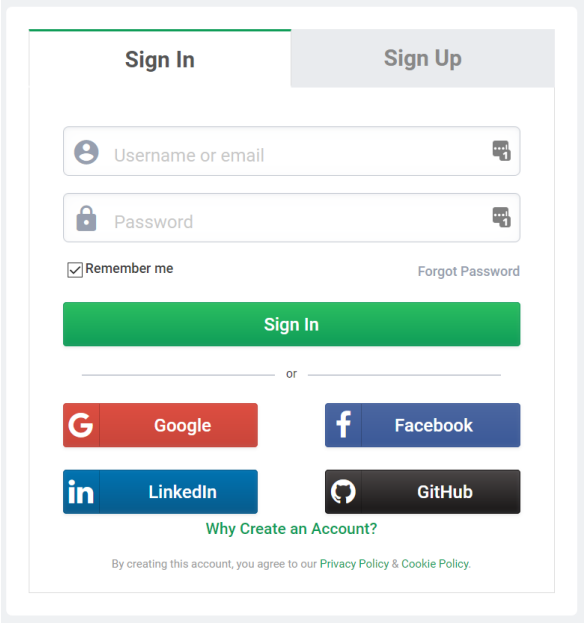


Figura 26. Caso real de implementación OAuth en aplicación cliente

De esa manera, esa plataforma de terceros no tendría por qué tener ese nombre de usuario ni la contraseña almacenados, sobre todo siendo un nombre de usuario y contraseña de otro servicio distinto, como podría ser alguna de estas redes sociales.

3.4.4 Terminología

Resource: un elemento al que se desea dar acceso

Scope: es casi equivalente a resource, se brinda acceso a un scope que representan recursos. Son los nombres de uno o más recursos, por ejemplo "userinfo" o "userAPI", es realmente un nombre que representa un nivel de acceso.

Resource Owner: es el dueño del recurso, en general es el usuario final.

Client: aplicación que quiere acceder a recursos en nombre del **Resource Owner.**

Authorization Server: es quien concede y valida accesos, donde se definen todos los clientes, recursos y scopes.

Resource Server: es donde reside el recurso, puede estar separado del Authorization Server.

Token: es un código (puede ser JWT) que contiene información.

Flujo/Grant: define el modo en que el cliente debe solicitar un token.

Consent: ventana de consentimiento con los permisos para acceder a determinado recurso.

Backchannel: es un canal de comunicación seguro.

Frontchannel: es un canal de comunicación inseguro.

3.4.5 Roles

Según el estándar RFC 6749 [27] Oauth define 4 roles que participan en su proceso:

1. Dueño de recurso (Resource Owner)
2. Client (Client)
3. Servidor de recursos protegidos (Resource Server)
4. Servidor de Autorización (Authorization Server)

3.4.5.1 Resource Owner

El propietario del recurso (Resource Owner) es un usuario capaz de conceder acceso a un cliente (aplicación) a un determinado recurso protegido para que esta haga algunas operaciones en nombre del usuario. El conjunto de cosas que puede hacer la aplicación cliente en nombre del propietario del recurso se denomina alcance o scope, y podría ser, por ejemplo, solamente acceso de lectura y no poder crear ningún tipo de elemento de ningún nuevo recurso en su nombre.

Un ejemplo de esto son los widgets que nos permiten integrar Twitter o Facebook dentro de una web, pero que no nos permiten generar nuevo contenido.

3.4.5.2 Client

El cliente sería la aplicación que desea acceder a los datos protegidos del Resource Owner. Antes de que pueda hacerlo debe ser autorizada por el usuario, y dicha autorización debe ser validada por la API. El cliente puede ser una aplicación web, una aplicación móvil, una aplicación de escritorio, una aplicación para Smart TV, un dispositivo de IoT, otra API que consumiera de esta API, etcétera.

3.4.5.3 Resource Server

El servidor de recursos será la API propiamente, es decir, el servidor que aloja el recurso protegido al cual queremos acceder. Es capaz de aceptar y responder a solicitudes de recursos protegidos utilizando tokens de acceso. Puede que también forme parte de la misma aplicación que el propio servidor de autenticación.

3.4.5.4 Authorization Server

El servidor de autorización es el que autentica al Resource Owner y emite un Access Token temporal para un ámbito o propósito (scope) definido por el propietario del recurso. En la mayoría de las ocasiones el Authorization Server y el Resource Server se utilizan juntos, y se denominan también Oauth Server. La interacción entre el servidor de autorización y el servidor de recursos está más allá del alcance de esta especificación. El servidor de autorización puede ser el mismo servidor que el servidor de recursos o una entidad separada. Un único servidor de autorización puede emitir tokens de acceso aceptados por varios servidores de recursos [27].

3.4.6 Tipos de clientes

Existen varios tipos de clientes para este tipo de protocolo:

- **Clientes confidenciales**: de alguna forma almacenan la contraseña sin que sea expuesta, ej: Una aplicación nativa compilada.
- **Clientes públicos**: no son capaces de mantener esta contraseña protegida, ej: aplicaciones JavaScript y Angular.

Por tanto, en base a determinado cliente es necesario implementar el flujo de OAuth2 de diferentes formas.

3.4.7 Flujos de OAuth

El flujo de autorización de OAuth es el que se muestra en la figura siguiente donde una aplicación cliente (API/Aplicación de escritorio, móvil o web) quiere acceder a un recurso. Este cliente pide autorización al dueño del recurso (Resource Owner). Luego de que el dueño del recurso le concede el permiso a la aplicación cliente, esta se dirige al servidor de autorización para obtener un token de acceso y poder acceder posteriormente a un recurso.

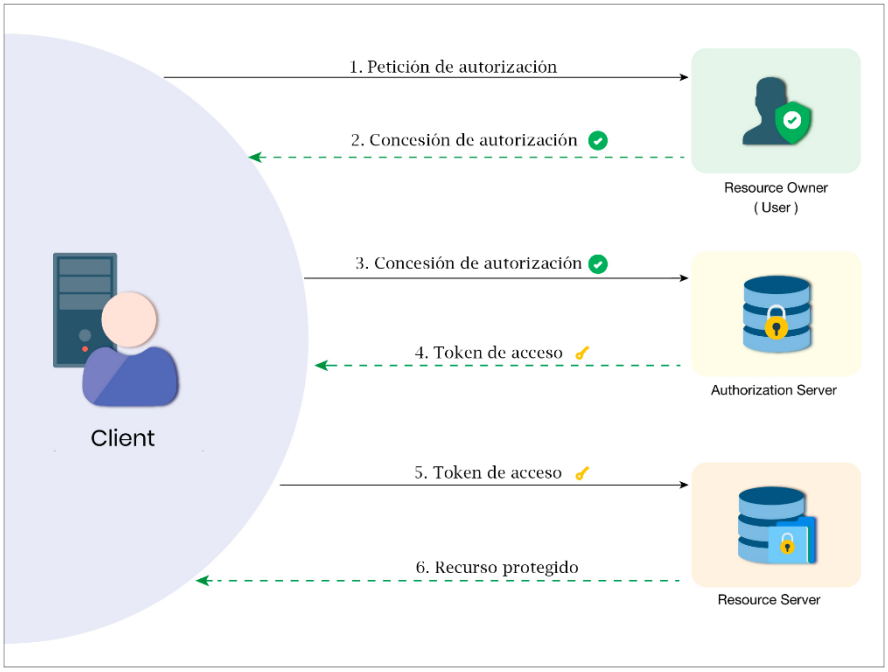


Figura 27. Flujo abstracto de OAuth

Luego de obtenido el token de acceso podemos ir al servidor de recurso y solicitar el recurso protegido con dicho token de acceso. Como se comentó con anterioridad, puede que el servidor de autorización y de recurso no sean dos aplicaciones independientes, sino que estén dentro del mismo servidor.

Este flujo mostrado anteriormente es el general que se desarrolla para el protocolo OAuth. En base al tipo de concesión (Grant Types), podemos poner en marcha uno u otro flujo en particular de OAuth.

Flujos o tipos de concesión (Grant Types) de OAuth

- Authorization code
- Client credentials
- Resource Owner Password Credentials
- Implicit
- Authorization code + PKCE
- Password Grant
- Device code

3.4.7.1 Authorization code

El tipo de concesión **Authorization code** es el más utilizado porque está optimizado para aplicaciones del lado del servidor, donde el código fuente no se expone públicamente y se puede mantener la confidencialidad del secreto del cliente. Este es un flujo basado en la redirección, lo que significa que la aplicación debe ser capaz de interactuar con el agente de usuario (es decir, el navegador web del usuario) y recibir códigos de autorización de API que se enrutan a través del agente de usuario. En la siguiente imagen se describe el flujo de Authorization code.

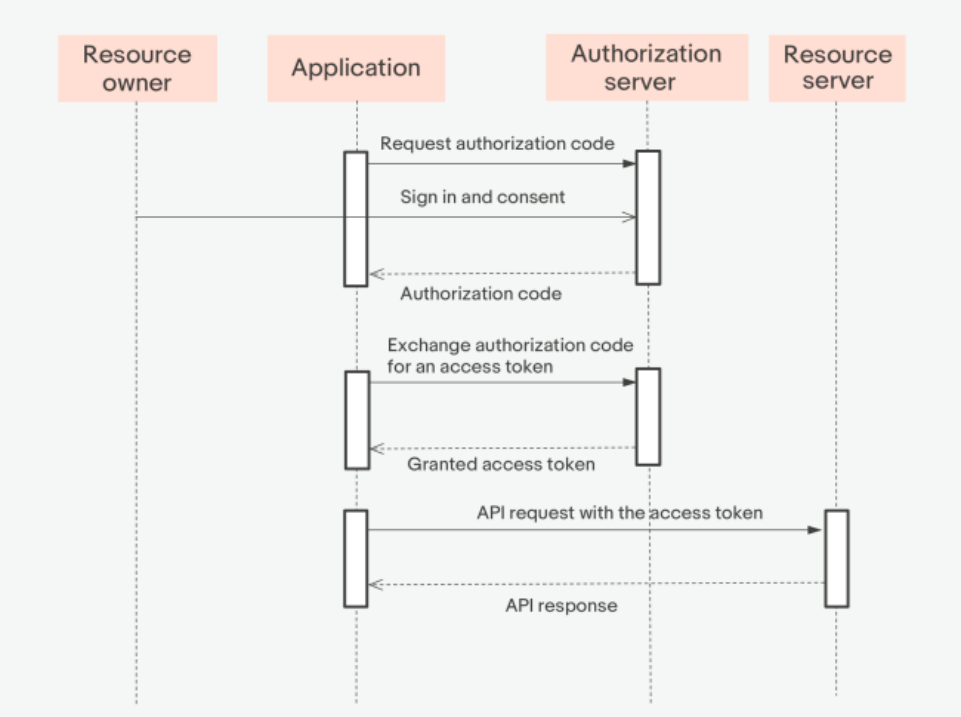


Figura 28. Flujo Authorization code

La aplicación cliente redirige al usuario al servidor de autorización, con una URL que se compone de diferentes parámetros, como el tipo de respuesta, que sería un código (response_type), el client_id o identificador del cliente, una URL de retorno hacia la aplicación (redirect_uri) y un scope como el ámbito de la autorización. La URL quedaría de la siguiente forma:

https://auth-server/authorize
?response_type=codigo
&client_id=CLIENT-ID
&state=abcs
&redirect_uri=https://gestion-peliculas/gp
&scope=ejemplo_write

Que significan cada uno de estos parámetros pasados en dicha url:

response_type=code: Le dice al servidor de autorización que la aplicación está iniciando el flujo Authorization code, en el cual se devuelve como respuesta un código y no un token. En caso de que el parámetro fuese "**response_type=token**", le dice al servidor que se iniciara el flujo Implicit (se detallará posteriormente), del cual se obtendrá directamente en la URL un token de acceso.

client_id: El identificador público de la aplicación, obtenido cuando el desarrollador registró por primera vez la aplicación.

redirect_uri: Le dice al servidor de autorización adónde enviar al usuario después de aprobar la solicitud.

scope: Una o más cadenas separadas por espacios que indican qué permisos solicita la aplicación.
state: La aplicación genera una cadena aleatoria y la incluye en la solicitud. Luego, debe verificar que se devuelva el mismo valor después de que el usuario autorice la aplicación. Esto se utiliza para prevenir ataques CSRF.

Se le pide al usuario que se autentique y luego de haberlo hecho, en la URL de callback se devuelve un código (code) que representa el consentimiento del usuario y su autorización, y un estado (state) que debe ser el mismo que el de la petición. Un ejemplo del código devuelto sería el siguiente:

https://gestion-peliculas/gp
?code=9E1L1E2S2kbeats
&state=abcs

Con el código obtenido podemos hacer una petición de tipo POST con la siguiente estructura:

POST /token HTTP/1.1
Host: servidor.autorizacion.com
Authorization: Basic afds8709afs8790asf
grant_type=authorization_code
&code=9E1L1E2S2kbeats
&redirect_uri= https://gestion-peliculas/gp

Si todo es correcto, se obtendría una respuesta válida de la siguiente forma:

HTTP/1.1 200 OK
Content-Type: application/json;charset=UTF-8
Cache-Control: no-store
Pragma: no-caché

```
{
    "access_token":"2YotnFZFEjr1zCsicMWpAA",
    "token_type":"example",
    "expires_in":3600,
    "refresh_token":"tGzv3JOkF0XG5Qx2TIKWIA", "example_parameter":"example_value"
}
```

Justamente con el token obtenido de la propiedad "access_token", se puede hacer en nombre del usuario todas las cosas para las cuales está definido el token y por los cuales se dio el consentimiento del propietario del recurso (Resource Owner).

3.4.7.2 Client Credentials

Este flujo no admite usuario y contraseña debido a que ya el mismo cliente tiene su client_id y su client_secret. Solo se usa para la comunicación segura (backchannel), precisamente para la comunicación o acceso entre API's. Este tipo de flujo es muy común en la arquitectura de microservicios.

La aplicación solicita un token de acceso enviando sus credenciales, su ID de cliente (client_id) y su secreto de cliente (client_secret) al servidor de autorización. Una solicitud de tipo POST de ejemplo para solicitar un token con este flujo sería similar a lo siguiente:

```
https://auth-server/token
?grant_type=client_credentials
&client_id=CLIENT_ID
&client_secret=CLIENT_SECRET
```

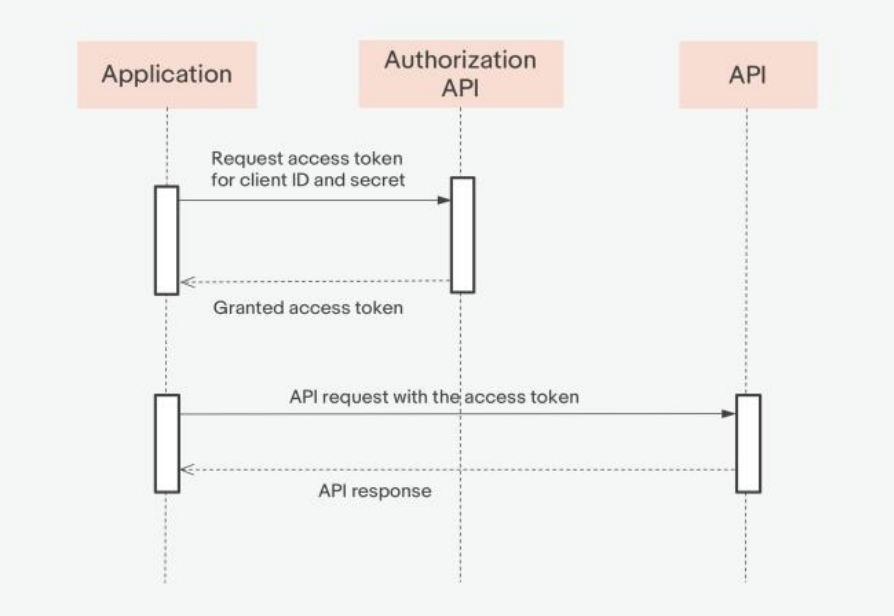


Figura 29. Flujo Client Credentials

Si se verifican las credenciales de la aplicación, el servidor de autorización devuelve un token de acceso a la aplicación. Ahora la aplicación está autorizada para usar los recursos necesarios

3.4.7.3 Resource Owner Password Credential

Este flujo permite intercambiar el nombre de usuario y la contraseña de un usuario por un token de acceso. El usuario proporciona las credenciales (nombre de usuario y contraseña) directamente a la aplicación, por tanto, se considera un flujo interactivo [28].

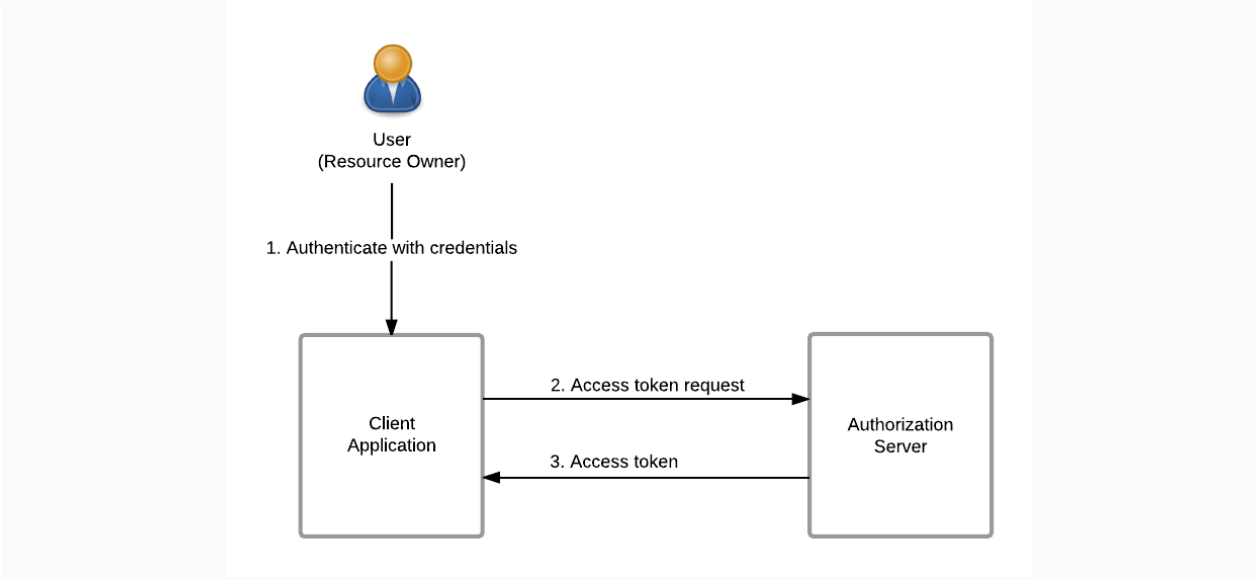


Figura 30. Flujo Resource Owner Password Credential

Luego, la aplicación usa las credenciales para obtener un token de acceso del servicio. Este funciona sin navegador ya que no existe la redirección que convencionalmente se aprecia en los otros flujos, por tanto, es ideal para su uso en aplicaciones de escritorio.

3.4.7.4 Implicit

El flujo Implicit es una forma en la que una aplicación JavaScript (Single Page Application) obtenga un token de acceso sin un paso de intercambio de código intermedio. Originalmente fue creado para su uso por aplicaciones de JavaScript (que no tienen una forma de almacenar secretos de forma segura), pero solo se recomienda en situaciones específicas [29].

Al igual que el flujo de Authorization code, Implicit comienza creando un enlace y dirigiendo el navegador del usuario a esa URL. Esta URL tendría en siguiente forma:

```
https://authorization-server.com/auth
?response_type=token
&client_id=29352910282374239857
&redirect_uri=https%3A%2F%2Fexample-app.com%2Fcallback
&scope=create+delete
&state=xcoiv98y3md22vwsuye3kch
```

La aplicación o cliente lo que hace es enviar al usuario al servidor de autorización a través de esta URL. Cuando el usuario visita la URL, el servidor de autorización le presenta un mensaje preguntándole si desea autorizar la solicitud de esta aplicación. El usuario ve la solicitud de autorización y aprueba la solicitud de la aplicación cliente.

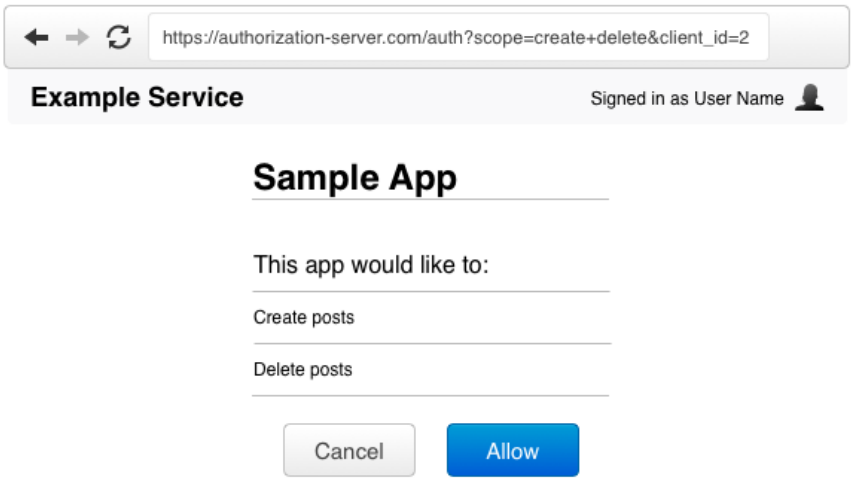


Figura 31. Ventana de consentimiento

Al aprobar la solicitud (ventana de consentimiento), el servidor de autorización redirigirá el navegador de vuelta a la URL del parámetro "redirect_uri", especificado por la aplicación, agregando un **token** y un **state** al fragmento de la URL. Esta quedaría de la siguiente forma:

```
https://example-app.com/redirect
 #access_token=g0ZGZmNj4mOWIjNTk2Pw1Tk4ZTYyZGI3
 &token_type=Bearer
 &expires_in=600
 &state=xcoVv98y2kd44vuqwye3kcq
```

Hay que tener en cuenta las diferencias principales entre este flujo y el código de autorización: se devuelve el token de acceso en lugar del código temporal, y ambos valores se devuelven en el fragmento de URL (después del símbolo #) en lugar de en la cadena de consulta. Al hacer esto, el servidor se asegura de que la aplicación pueda acceder al valor de la URL, pero el navegador no enviará el token de acceso en la solicitud HTTP al servidor.

El valor del estado será el mismo valor que la aplicación estableció inicialmente en la solicitud. Se espera que la aplicación verifique que el estado en la redirección coincida con el estado que estableció originalmente. Esto protege contra CSRF y otros ataques relacionados.

El servidor también indicará la vida útil del token de acceso antes de que caduque. Suele ser una cantidad de tiempo muy breve, del orden de 5 a 10 minutos, debido al riesgo adicional de devolver el token en la propia URL.

Cuando usar el flujo Implicit

En general, existen circunstancias extremadamente limitadas en las que tiene sentido utilizar el flujo Implicit. Este se creó para las aplicaciones de JavaScript mientras se intentaba que también fuera más fácil de usar que la concesión del código de autorización. En la práctica, cualquier beneficio

obtenido de la simplicidad inicial se pierde en los otros factores necesarios para asegurar este flujo. Cuando sea posible, las aplicaciones de JavaScript deben usar la concesión del código de autorización sin el secreto del cliente.

La principal desventaja del tipo de concesión implícita es que el token de acceso se devuelve directamente en la URL, en lugar de devolverse a través de un canal de confianza como en el Código de autorización. El token de acceso en sí se registrará en el historial del navegador, por lo que la mayoría de los servidores emiten tokens de acceso de corta duración para mitigar el riesgo de que se filtre el token de acceso. Debido a que no hay un canal secundario y todo se hace en frontchannel, el flujo implícito tampoco devuelve un token de actualización. Para que la aplicación obtenga un nuevo token de acceso cuando vence el de corta duración, la aplicación debe enviar al usuario de regreso a través del flujo de OAuth nuevamente o usar trucos como iframes ocultos, lo que vuelve a agregar la complejidad al flujo originalmente.

3.4.7.5 Authorization code + PKCE

PKCE, pronunciado "pixy", es un acrónimo de Proof Key for Code Exchange [30]. El flujo de Authorization code + PKCE es un flujo diseñado específicamente para autenticar usuarios de aplicaciones cliente públicas (nativas o móviles). Este flujo es algo parecido al ya mencionado Authorization code, pero este viene a ser precisamente una extensión de él, para prevenir varios ataques y poder realizar el intercambio OAuth desde clientes públicos de forma segura. La diferencia clave entre PKCE y el flujo de Authorization code es que no se requiere el client_secret en el flujo con PKCE. PKCE se desarrolla principalmente para clientes públicos como se comentó anteriormente, ya que no tienen un secreto (client_secret), lo que significa que no tienen una forma real de autenticarse. Con eso, cualquier aplicación puede hacerse pasar por el cliente público.

PKCE usa los siguientes parámetros en su flujo:

- **code_verifier** — Cadena aleatoria entre 43 y 128 caracteres.
- **code_challenge**: creado por SHA256 que codifica el code_verifier y la URL base64 codificando el hash resultante. *Base64UrlEncode(SHA256Hash(code_verifier)).*
- **code_challenge_method** — Simple o S256

El parámetro code_challenge y code_challenge_method son parámetros que se agregan a la solicitud inicial como parte de la URL. Luego de obtenido el código por parte del cliente, este procede a canjearlo por un token, pasando los mismos parámetros que el flujo de Authorization code. El parámetro code_verifier se incluye en la petición al solicitar el token, basado en los parámetros anteriores. Con esto se continua el flujo de Authorization code de forma similar a como ya se conocía. Actualmente este es el método más seguro para escenarios donde existe un frontchannel (navegador web), principalmente porque protege el redirect para estos clientes.

Según el rfc 7636 en [30], definen este flujo abstracto, donde se explica de forma detallada y se visualiza como la figura siguiente.

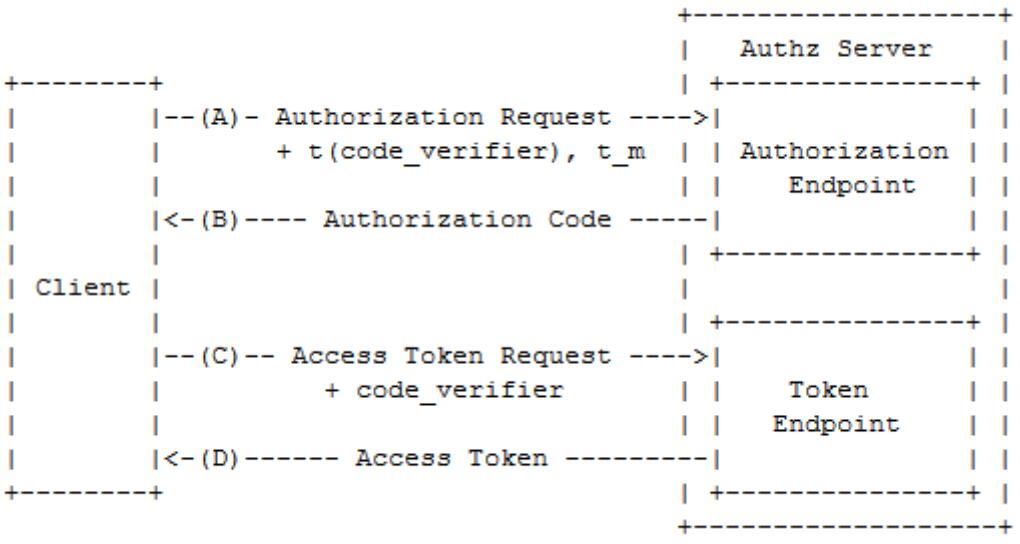


Figura 32. Flujo Authorization code + PKCE

4. CASO DE USO PRÁCTICO

4.1 Introducción

En el siguiente apartado se hará uso de algunas de las tecnologías mencionadas anteriormente para poner en práctica el conocimiento y poder llegar a un mejor entendimiento y aprovechamiento de estas. Debido a las ventajas y según lo concluido en secciones anteriores podemos afirmar que para el siguiente caso a desarrollar no es del todo conveniente usar una arquitectura de microservicios, debido a lo pequeña que sería en un comienzo, pero precisamente es una aplicación que puede llegar a crecer en cantidad de usuarios y habría que pensar con antelación en una forma de escalarla. Aun así, este conocimiento no es suficiente para abarcar todo el proceso o flujo que se llevaría hoy en día para brindar seguridad a una aplicación con la arquitectura orientada a microservicios. Con este tipo de arquitectura y una capa más de seguridad como lo es la autorización frente a lo ya conocido como autenticación, garantizamos hacer de esta una aplicación totalmente robusta y con grandes ventajas.

4.2 Análisis y diseño

El caso práctico se basa en el desarrollo de una aplicación web encargada de gestionar un catálogo de películas con sus correspondientes actores y a su vez los usuarios que acceden a ella, con la posibilidad de hacer críticas por cada película si es de su interés. En este desarrollo se seleccionó el Framework Spring debido al gran ecosistema de módulos que tiene a su disposición y la posibilidad de implementar el protocolo OAuth. Incluso la gran diversidad de patrones que se pueden implementar para lograr el objetivo y el desarrollo de una arquitectura de este tipo. Entre los módulos y tecnologías a usar que se mencionaron en apartados anteriores tenemos: Spring Data, Spring Cloud, Spring Security, microservicios, servicios REST, además de hacer uso del patrón MVC en el diseño de la aplicación y usar para la capa de presentación (cliente web) Thymeleaf, como motor de plantillas. En esta arquitectura y en si la aplicación web.

Para el desarrollo de la aplicación se hicieron dos microservicios principales. El primer microservicio **Películas-Actores**, es el encargado de hacer todas las gestiones relacionadas precisamente con las películas y actores de la aplicación. El segundo microservicio **Usuarios-Críticas**, es el encargado de toda la gestión de usuarios y de las críticas que hacen cada usuario a las diferentes películas del catálogo. También se cuenta con una capa de presentación para el usuario (aplicación cliente o frontend) de donde parten todas las peticiones del usuario y se envían a cada microservicio correspondiente. Estas peticiones pasan por un Gateway (Puerta de enlace) dirigiéndose a los microservicios comentados anteriormente, para procesar, almacenar o leer los datos de la base de datos; como se muestra en la siguiente figura.

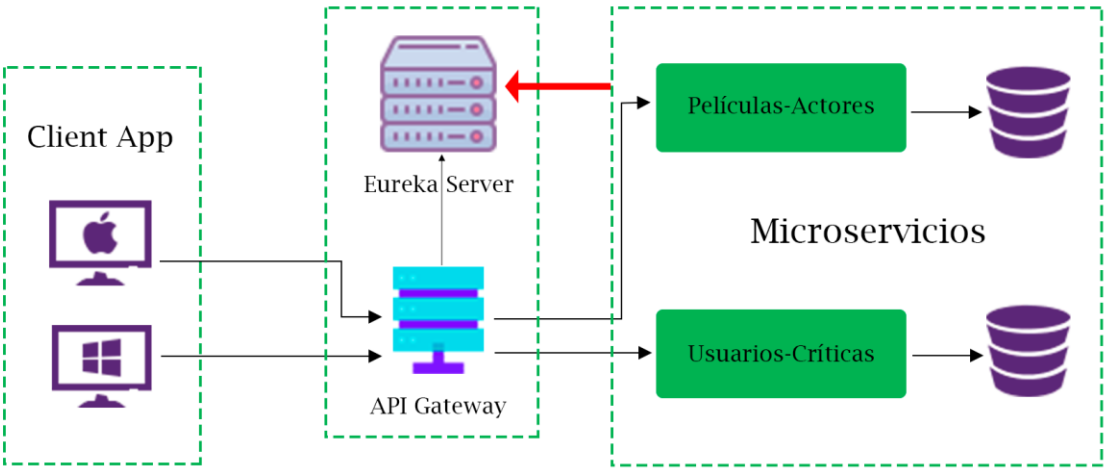


Figura 33. Arquitectura del sitio web de gestión de películas

La arquitectura cuenta con un servidor Eureka que mantiene registrado y localizado a todos los microservicios y los demás componentes, para garantizar un control de estos. También se encontrarán registrados en este el componente API Gateway y otro componente que será mencionado más adelante. La aplicación expone dos API REST sobre las cuales se realizarán las típicas operaciones de Registro, Lectura, Actualizar, Eliminar (CRUD), para cada microservicio contemplado en la arquitectura. El acceso a estas API´s será totalmente controlado por el Gateway de la arquitectura, procesando cada petición entrante y redirigiéndola según la lógica de predicados configurada, a su respectivo servicio.

Esta sería la arquitectura de microservicios pensada e implementada del sitio web de gestión de películas. Con esta arquitectura garantizamos el total funcionamiento del sitio web. Pero aun aquí no cumplimos con el objetivo del proyecto en cuestión. Entonces hasta cierto punto nos preguntaremos: **¿Cómo implementar OAuth2 en esta arquitectura anteriormente plasmada?** Ahora es cuando se debe pensar un punto más allá de cómo garantizar una mayor seguridad a este tipo de arquitectura, que no sea simplemente el uso de usuario y contraseñas y sus respectivos roles de acceso. Exactamente, OAuth2 es el medio para garantizar este punto más allá de la seguridad.

Entonces para aplicar la implementación de OAuth2, este protocolo define cuatro roles que se traducen a la arquitectura anterior.

- El **Resource Owner** (Propietario del recurso), que sería el usuario que usa nuestra aplicación web a través de un navegador.

- El **Client** (Aplicación Cliente), sería nuestra capa de presentación a través de la cual el usuario va a poder interactuar con el sistema, enviando y recibiendo peticiones según la operación a solicitar o recursos a obtener.

- **Resource Server** (Servidor de recursos), en este caso las API´s que se exponen se traducen en nuestros servidores de recursos, que alojan un recurso protegido al cual el usuario va a acceder.

- **Authorization Server** (Servidor de Autorización), este sería el nuevo componente para adicionar en la arquitectura anterior, que complementa la implementación del protocolo OAuth2.

En la siguiente imagen se plasma la arquitectura orientada precisamente a la implementación de OAuth2 en el sitio web de gestión de películas.

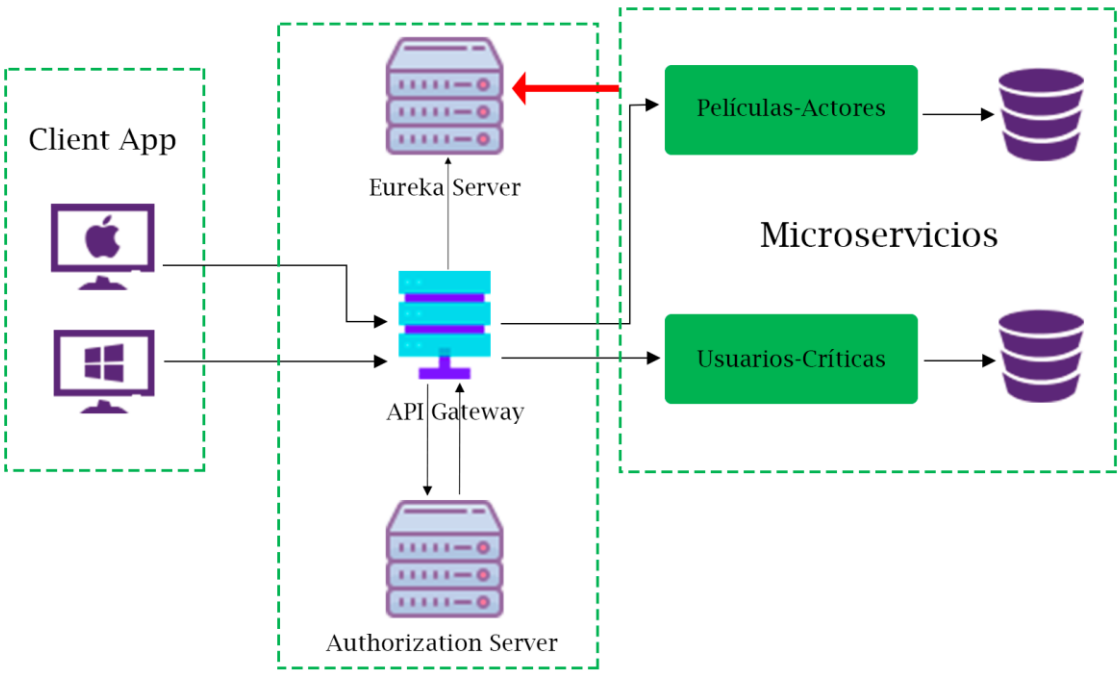


Figura 34. Arquitectura del sitio gestión de películas con OAuth2

La aplicación cliente, que sería nuestra capa de presentación, quiere acceder a un recurso. Este cliente pide autorización al dueño del recurso, que sería el propio usuario que está utilizando la aplicación. Luego de que el usuario le concede el permiso a la aplicación cliente, esta se dirige al servidor de

autorización para obtener un token de acceso y poder acceder posteriormente a una de nuestras API´s. Luego de que se obtiene el token de acceso podemos dirigirnos a nuestras API´s y solicitar el recurso protegido con dicho token de acceso. Así funciona la aplicación de gestión de películas a gran escala.

Con esta vista arquitectónica del sistema de gestión de películas pensado para utilizar la implementación del protocolo OAuth2, podemos aclarar ciertas consideraciones.

El flujo anteriormente mencionado sería en flujo general de cómo funcionaria el sitio web luego de pensada su implementación con OAuth, pero según el tipo de aplicación, es necesario desarrollar uno de los flujos mencionados en la sección 3.4.7. Para este proyecto se desarrolló en flujo de **Authorization code**, debido a que es el más usado hoy en día en el tipo de aplicación web y te garantiza una mayor seguridad con respecto a los otros flujos abordados.

En muchas ocasiones, haciendo énfasis en el **Authorization Server**, este puede estar delegando por un tercero conocido como Facebook, Twitter, Github, Google, Okta, o se puede propiamente implementar, que sería en caso de desarrollo para la elaboración de este proyecto. Con la implementación propia del servidor de autorización tenemos la ventaja de tener controlado todo el ecosistema e incluso podemos llegar a una mayor personalización de este y mayor gestión de los datos que pasan a través de él.

4.3 Implementación de la arquitectura

En este apartado se detalla la implementación de cada uno de los componentes de la arquitectura anteriormente propuesta. Cada componente en sí estará asociado a uno de los roles del protocolo OAuth2.

4.3.1 Aplicación cliente

La aplicación cliente denominada **servicio-cliente-películas** sería la encargada de solicitar el recurso protegido del usuario dentro de toda la aplicación y de la implementación OAuth2. Para su creación contamos con estas principales dependencias dentro del archivo pom.xml que se genera al crear un proyecto nuevo con el Spring Initializr del Framework:

- Spring Security
- Spring Web
- OAuth2 Client
- Spring Data JPA
- Thymeleaf

Spring Security es la dependencia que se encarga de la autenticación y del control de acceso basado en los diferentes filtros. **Spring Web** incluye las dependencias necesarias para la creación del proyecto web en sí y su estructura. **OAuth2 Client** define la aplicación web como un cliente de OAuth para la comunicación con el servidor de Autorización.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<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>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-client</artifactId>
</dependency>
```

Figura 35. Dependencias principales de la aplicación cliente

Spring Data JPA es la dependencia (API) encargada de la persistencia de los datos relacionales (SQL) dentro de la aplicación. **Thymeleaf** es una dependencia de java que implementa el motor de plantillas de la aplicación cliente para ser usado en la web.

También se definió en un archivo application.yml con toda la configuración del cliente y su respectivo comportamiento.

```yaml
server:
  port: 8080
  servlet:
    encoding:
      charset: UTF-8
      enabled: true

spring:
  application:
    name: servicio-cliente-peliculas
  security:
    oauth2:
      client:
        registration:
          api-client-oidc:
            provider: spring
            client-id: Film Affinity
            client-secret: secret
            authorization-grant-type: authorization_code
            redirect-uri: "http://servicio-cliente-peliculas:8080/login/oauth2/code/{registrationId}"
            scope: openid, user.info
            client-name: api-client-oidc
          api-client-authorization-code:
            provider: spring
            client-id: Film Affinity
            client-secret: secret
            authorization-grant-type: authorization_code
            redirect-uri: "http://servicio-cliente-peliculas:8080/authorized"
            scope: user.info
            client-name: api-client-authorization-code
        provider:
          spring:
            issuer-uri: http://auth-server:9000
```

Figura 36. Fichero de configuración de la aplicación cliente

Este es el archivo más importante dentro de la implementación del cliente. Es donde se definen todos los parámetros para su correcto funcionamiento y para la comunicación con el servidor de autorización. Aquí se define la conexión de OpenID Connect y el flujo que se usará (Authorization code) para la implementación de OAuth2. Se indican también los scope necesarios por los cuales se pregunta en los servidores de recurso. Finalmente se define también donde estará ubicado el servidor de autorización implementado.

También se implementó una clase para la configuración de la seguridad, principalmente orientada a los filtros dentro del cliente (SecurityFilterChain ya mencionados en la sección de Spring Security). En la figura que se muestra a continuación se detalla que hacer para cada solicitud. Primeramente, todas las solicitudes que no estén en la lista de solicitudes permitidas (**WHITE_LIST_URLS**) tendrán que ser autenticadas. Luego tenemos dos formas de acceder dentro de este cliente. La primera forma es con un formulario de acceso tipo como se hace normalmente, donde pones tus credenciales de acceso al sitio y se procede en la operación, y todo el acceso se hace a través de la URL "/acceso". La segunda forma de acceder es a través de la página de acceso por defecto del servidor de autorización que se implementa con OAuth2, se redirige al usuario que usa la aplicación cliente a un formulario OAuth2 de acceso que se encuentra dentro de dicho servidor. En este servicio se incluyen los controladores necesarios para gestionar el frontend o páginas de la aplicación.

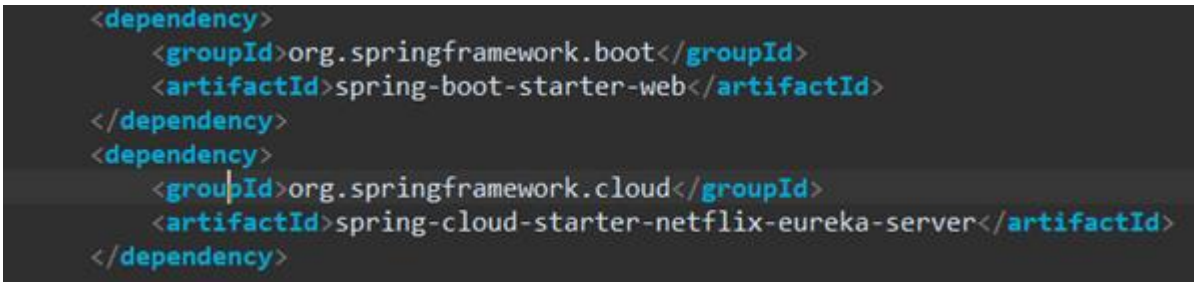
```
private static final String[] WHITE_LIST_URLS = { "/js/**", "/css/**", "/img/**", "/plugins/**",
   "/webfonts/**", "/", "/registro", "/peliculas/**", "/policy", "/oauth2/**", "/login/**",
    "/callback/", "/criterio***", "/acceso"
};

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


 http
    .csrf().disable()
    .antMatcher("/**")
    .authorizeRequests()
    .antMatchers(WHITE_LIST_URLS).permitAll()
    .anyRequest().authenticated()
    .and()
    .formLogin()
        .loginPage("/acceso")
        .defaultSuccessUrl("/", true)
    .and()
    .oauth2Login(oauth2Login -> oauth2Login.loginPage(
              OAuth2AuthorizationRequestRedirectFilter.DEFAULT_AUTHORIZATION_REQUEST_BASE_URI +
              "/api-client-oidc"
    ))
    .oauth2Client(Customizer.withDefaults());

}
```

Figura 37. SecurityFilterChain dentro de la aplicación cliente

En resumen, esta aplicación (Client) hace las peticiones para obtener los recursos protegidos y además redirige correctamente cada una al lugar indicado.

4.3.2 Servidor Eureka

Servidor con la capacidad de interconectar los elementos que intercambien información con los demás elementos del entorno de la arquitectura. Así cada uno sabe la ubicación de los demás componentes dentro de la propia arquitectura. Por lo que para mantener el registro y la localización de los microservicios y componentes se propuso crear este tipo de servidor. Es necesario crear este servidor con el Spring Initializr y las dependencias siguientes.



```
<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 38. Dependencias del Servidor de Eureka

Con estas dependencias logramos la facilidad de crear un servidor que cumpla con las capacidades para las que está diseñado este tipo de componente (registro, estado y localización de microservicios). En uno

de sus ficheros llamado **application.yml**, se configura dicho servidor, donde se definieron propiedades como el puerto por el cual se estará ejecutando el servidor, su nombre y la URL de servicio por la cual está registrando a cada microservicio.

```yaml
eureka:
  client:
    register-with-eureka: false
    fetch-registry: false
    service-url:
      defaultZone: http://localhost:8000/eureka
server:
  port: 8000
spring:
  application:
    name: eureka-server
```

Figura 39. Fichero de configuración del Servidor de Eureka

Cada uno de los microservicios se define como un cliente de Eureka. Al arrancar cada microservicio, este se comunicará con el servidor Eureka para notificarle que está disponible para ser consumido. El servidor Eureka tiene la información de todos sus microservicios registrados además de su estado. Cada microservicio notifica al servidor de su estado cada 30 segundos. Si pasados los 30 segundos el servidor no recibe notificación del microservicio, este es eliminado del registro del servidor.

```java
package com.eurekaserver;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.netflix.eureka.server.EnableEurekaServer;

@EnableEurekaServer
@SpringBootApplication
public class EurekaServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(EurekaServerApplication.class, args);
    }

}
```

Figura 40. Anotación para definir el Servidor de Eureka

Solo falta según las anotaciones de Spring definir que este tipo de aplicación será un Servidor Eureka. Esto se define anotando la clase principal de ejecución con @EnableEurekaSever. Al iniciar el servidor se puede observar cómo se registran cada uno de los microservicios y componentes de la arquitectura ya planteada.

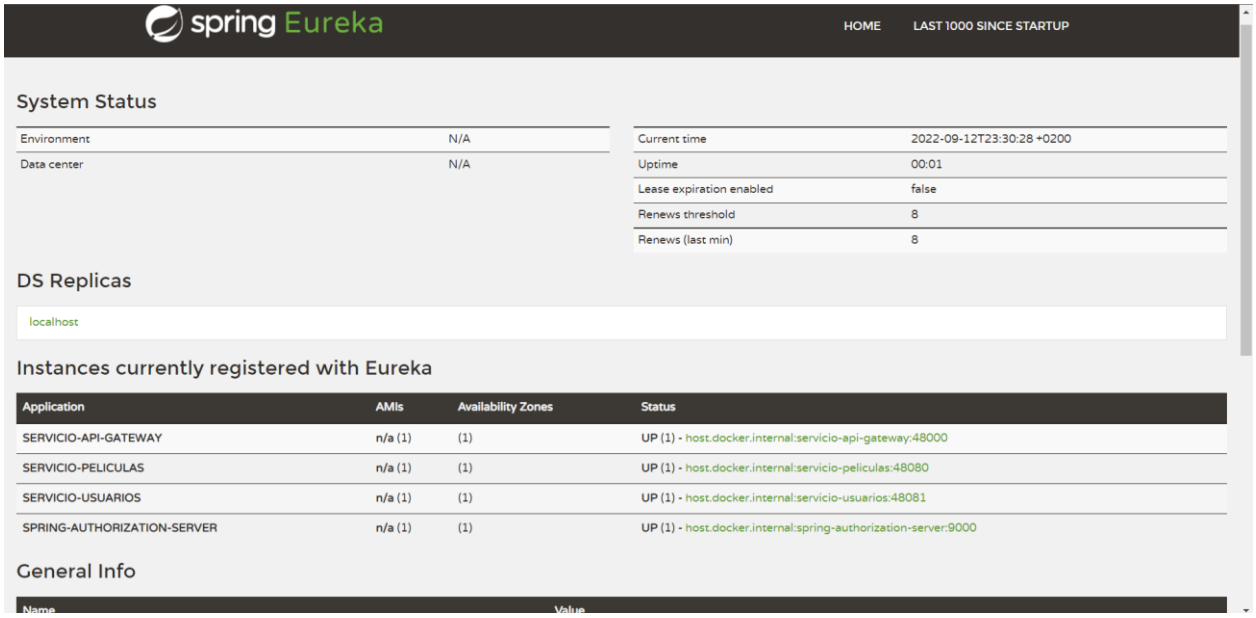


Figura 41. Servidor Eureka con los microservicios y componentes de la arquitectura

Hay que tener en cuenta que cada componente que sea declarado como un cliente de eureka, es registrado en dicho servidor con el propio nombre definido en su archivo de configuración. Anteriormente mostrados como: SERVICIO-API-GATEWAY, SERVICIO-PELICULAS, SERVICIO-USUARIOS, SPRING-AUTHORIZATION-SERVER.

4.3.3 API Gateway

Este componente (**API Gateway**) logra que los microservicios estén totalmente accesibles a través de él, garantizando un acceso seguro y rápido. Cada microservicio expone una API que es accesible por el usuario cuando este usa la aplicación cliente y su comunicación se realiza de forma transparente para él. Esta comunicación pasa por el API Gateway y llega como destino final a la API expuesta en los servicios, que para nuestra implementación de OAuth2 serían los servidores de recursos. Por tanto, el Gateway sería el único punto de entrada para la comunicación con los servidores de recursos.

Para la creación del API Gateway, al igual que los proyectos anteriores se usó el Spring Initializr junto con las dependencias que se visualizan en el fichero pom.xml siguiente:

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

Figura 42. Dependencias del API Gateway

Este Gateway se define como un cliente del servidor Eureka para llevar un control sobre el mismo y además se define el starter del Gateway con toda la lógica necesaria para el funcionamiento. En el

archivo de configuración se definen varias propiedades para garantizar su correcto funcionamiento. Además de definirlo como un cliente de Eureka también se define una lógica de predicados como se puede apreciar en la siguiente imagen.

```yaml
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8000/eureka
server:
  port: 48000
spring:
  application:
    name: servicio-api-gateway
  cloud:
    gateway:
      routes:
      - filters:
        - StripPrefix=2
        id: servicio-peliculas
        predicates:
        - Path=/api/peliculas/**
        uri: http://localhost:48080
      - filters:
        - StripPrefix=2
        id: servicio-usuarios
        predicates:
        - Path=/api/usuarios/**
        uri: http://localhost:48081
      - filters:
        - StripPrefix=0
        id: spring-authorization-server
        predicates:
        - Path=/
        uri: http://localhost:9000
```

Figura 43. Fichero de configuración API Gateway

¿Qué define esta lógica de predicados del fichero application.yml? Con esta lógica se garantiza el enrutamiento de cada petición hecha por el usuario con la aplicación cliente a cada uno de los servidores de recursos y al servidor de autorización.

Primeramente, cada servicio cuenta con unas propiedades específicas para garantizar el correcto funcionamiento como: id, predicates, uri, filters.

La propiedad **id**, identifica el nombre de cada uno de los servicios o servidores de recursos (servicio-películas y servicio-usuarios), o en el caso del servidor de autorización su correspondiente nombre. Las propiedades **predicates** y **uri** indican que todas las peticiones con el Path: /api/peliculas/** y /api/usuarios/** serán enrutadas a los servicios **servicio-peliculas** y **servicio-usuarios** en los puertos 48080 y 48081 respectivamente, definidos para el trabajo en local en cada uri. Las peticiones con Path: / y **uri** localhost a través del puerto 9000, indica que todas las peticiones entrantes a la API Gateway por ese puerto serán enrutadas al servidor de autorización.

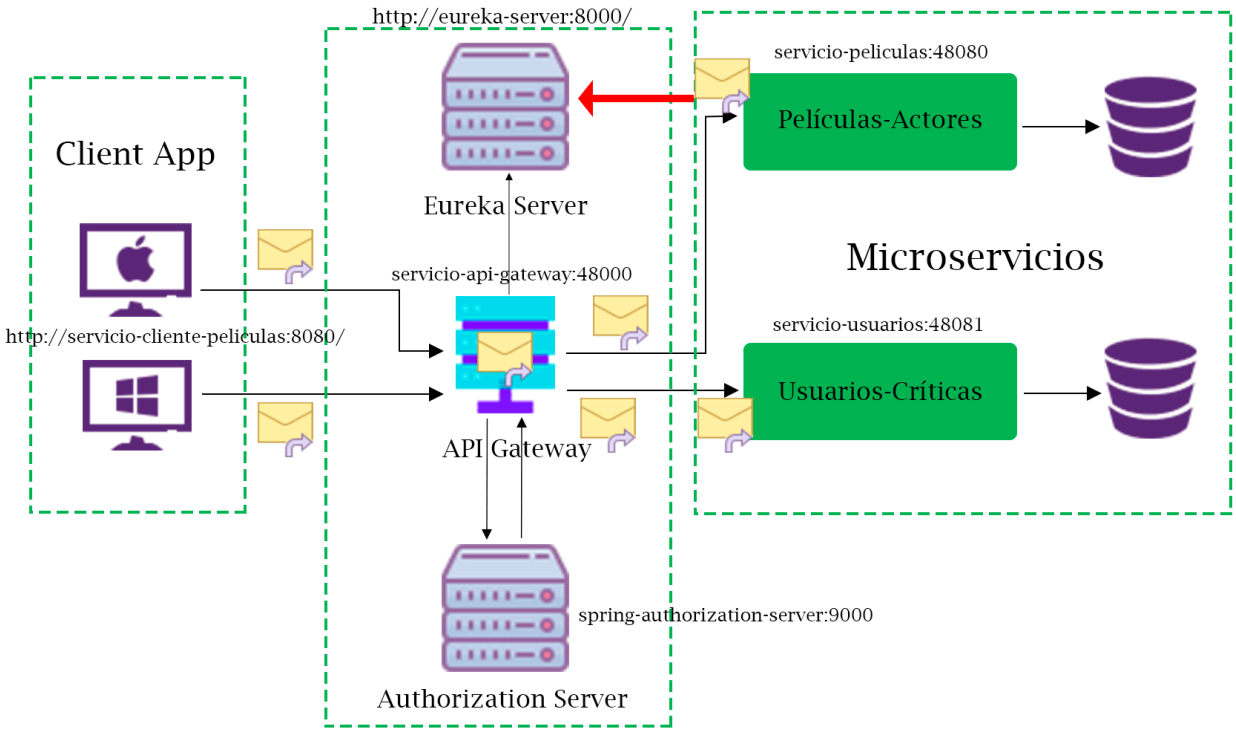


Figura 44. Enrutamiento de API Gateway a las peticiones

La propiedad **Filters** define el filtro **StripPrefix** con el valor 2 para que la API Gateway elimine los dos primeros elementos "/api/peliculas" y "/api/usuarios" de las peticiones que llegan con las rutas definidas. Entonces, en caso de no aplicar este filtro y no eliminar estos dos prefijos las rutas que llegan a los servidores de recursos devolverán un error, ya que en los servidores de recursos o servicios asociados al proyecto no se expone ningún endpoint con los prefijos: "/api/peliculas", "/api/usuarios".

4.3.4 Servidor de Recursos

Los servidores de recursos se definen como parte de la implementación de OAuth2 como los servicios que tienen las diferentes operaciones que exponen las API. Para la implementación de los servidores de recursos, tanto el servicio-películas como para el servicio-usuarios, se definieron un conjunto de dependencias en su creación.

```xml
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-oauth2-resource-server</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-netflix-eureka-client</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Figura 45. Dependencias de los servidores de recursos

En este caso la principal dependencia a destacar respecto a los proyectos anteriores es la de "oauth2-resource-server" de Spring Security. Con esta dependencia dotamos a los servicios que exponen una API de funcionalidad de Servidor de recursos de la implementación de OAuth2.

En cada fichero de configuración (**application.yml**) de ambos servidores de recursos tenemos su definición del servidor de eureka (http://servidor-eureka:8000) al que tiene que registrarse cada uno al iniciar su ejecución. También en la configuración de ambos se definió una configuración para la conexión jdbc con mysql en procesamiento directo de datos. La última y principal configuración de estos servidores es la definición dentro de estos propios servidores de recursos, el valor que identifica al servidor de autorización, en concreto la propiedad **issuer-uri**, donde se descubren las claves públicas del servidor de autorización y posteriormente se validan los JWT entrantes.

```yaml
server:
  port: 48081
eureka:
  client:
    service-url:
      defaultZone: http://localhost:8000/eureka
spring:
  application:
    name: servicio-usuarios
  datasource:
    driver-class-name: com.mysql.cj.jdbc.Driver
    password: root
    username: root
    url: jdbc:mysql://localhost:3306/usuarios_db?useTimezone=true&serverTimezone=UTC
  jpa:
    database-platform: org.hibernate.dialect.MySQL8Dialect
    hibernate:
      ddl-auto: update
    show-sql: true
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://auth-server:9000
```

Figura 46. Fichero de configuración de los Resource Server

El Resource Server **servicio-usuarios** está compuesto por 3 controladores principales que muestran diferentes endpoint que define cada API. Las clases controladoras son las encargadas de proporcionar como recurso las operaciones que se definen en las clases de los paquetes de servicio. La paquetes de servicio (service) implementa la lógica de negocio y se enlaza en paquete de acceso a datos. Los paquetes de acceso a datos se encarga de implementar el patrón DAO y encapsula toda la comunicación y operación precisamente con los datos de la aplicación.

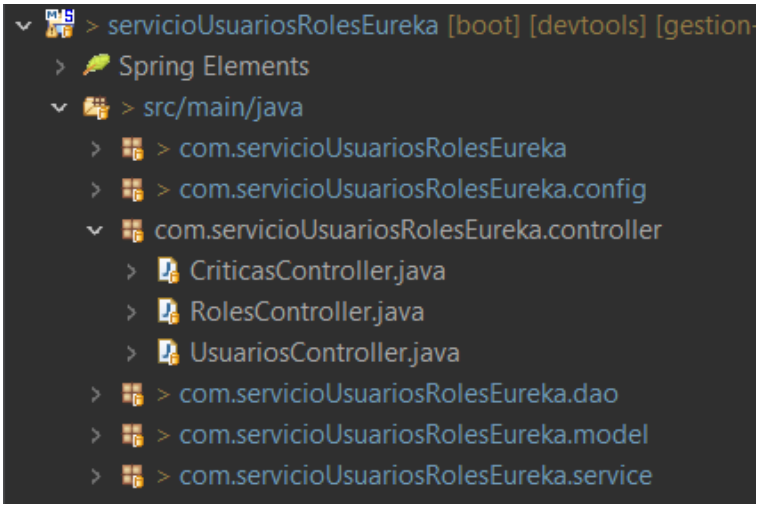


Figura 47. Estructura de paquetes del servicio-usuarios

En los controladores se detallan las diferentes métodos que se exponen a través de la API con las diferentes rutas para realizar las peticiones. Este servicio cuenta con tres controladores principales: **CriticasController**, **RolesController**, **UsuariosController**. Cada uno define los métodos u operaciones necesarios de un **CRUD** para garantizar el funcionamiento, donde las operaciones van a ser accedidas a través de la API por el cliente cuando quiera obtener algún recurso en específico.

También se define una clase de configuración en el paquete **com.servicioUsuariosRolesEureka.config**, que será el encargado de filtrar las solicitudes provenientes del cliente y dar acceso a determinado recurso según lo definido en su interior. En este caso, se usó esta configuración para dar acceso a un recurso protegido en base a un scope definido tanto en el cliente como en el servidor de autorización.

```
@EnableWebSecurity
public class ResourceServerConfig {

    private static final String[] WHITE_LIST_URLS = { "/usuarios"

    };

    @Bean
    SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {


        http
        .csrf().disable()
            .authorizeRequests()
            .mvcMatchers(WHITE_LIST_URLS).permitAll()
            .mvcMatchers("/api/usuarios/usuarios/correo/**")
                .access("hasAuthority('SCOPE_user.info')")
            .and()
            .oauth2ResourceServer()
            .jwt();

        return http.build();
```

Figura 48. Recurso protegido por el scope user.info

En las primeras líneas, se permite todo el tráfico entrante que proviene de la URL "/usuarios" y luego brinda el acceso a una URL en específica del servicio según el scope user.info definido. Esta URL está protegida por el scope y te brinda la información del usuario como el correo electrónico. Más abajo se define este endpoit como parte de la implementación de OAuth2 de ser un servidor de recursos.

Sucede igualmente con el otro servidor de recursos donde se definen dos controladores principales que serían los que muestran el endpoint del servicio a través de la URL que será expuesta por la API. En este caso se implementan los controladores **ActoresController** y **PeliculasController**, donde al igual que el anterior servicio definen las operaciones necesarias de un **CRUD** las cuales serán implementadas por la capa de servicio y la lógica de acceso a datos. Para este caso no es necesario configurar ningún recurso protegido porque son métodos que se pueden acceder con normalidad luego de que el usuario accede al sistema por mediación de OAuth2. En consideraciones posteriores en caso de que sea conveniente proteger algún recurso para luego acceder atrás de un token a estos según la implementación de OAuth2, se debe configurar este servidor de recurso con una clase de configuración al igual que se implementó con el servidor de recurso de los usuarios.

4.3.5 Servidor de Autorización

Debido a que se decidió desarrollar el flujo de **Authorization code** como implementación del protocolo OAuth2, este servidor se encarga de varias funciones a destacar. Estas funciones son: la emisión de un código de autorización que se entregará al cliente con el objetivo de que este sea cambiado posteriormente por un token de acceso, la emisión del token de acceso que es canjeado con el código ya obtenido por el cliente y la validación o revocación de este.

Para desarrollar este servidor se creó un nuevo proyecto con el Spring Initializr con las siguientes dependencias.

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-oauth2-authorization-server</artifactId>
    <version>0.3.1</version>
</dependency>
<dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <scope>runtime</scope>
</dependency>
```

Figura 49. Dependencias del servidor de autorización

La principal dependencia es **spring-security-oauth2-authorization-server** en su versión 0.3.1 del proyecto Spring Security, dependencia necesaria para garantizar el desarrollo de esta implementación como un servidor de autorización. Las otras dependencias no se describirán debido a que se ha detallado anteriormente.

Se definieron un conjunto de paquetes necesarios los cuales están compuestos por clases definidas para el correcto funcionamiento en términos de implementación del protocolo en cuestión.

En el paquete de Service se implementaron dos clases: **CustomAuthenticationProvider** y **CustomUserDetailService**. CustomAuthenticationProvider garantiza la autenticación de una solicitud de este tipo y devuelve un objeto del usuario autenticado.

```
@Autowired
private CustomUserDetailService customUserDetailsService;

@Autowired
private PasswordEncoder passwordEncoder;

@Override
public Authentication authenticate(Authentication authentication) throws AuthenticationException {

    String username = authentication.getName();
    String password = authentication.getCredentials().toString();
    UserDetails user= customUserDetailsService.loadUserByUsername(username);

    return checkPassword(user,password);

}

private Authentication checkPassword(UserDetails user, String rawPassword) {
    if(passwordEncoder.matches(rawPassword, user.getPassword())) {

        UsernamePasswordAuthenticationToken authenticationToken
            = new UsernamePasswordAuthenticationToken(user.getUsername(),
                user.getPassword(),
                user.getAuthorities());

        return authenticationToken;
    }
    else {
        throw new BadCredentialsException("Bad Credentials");
    }
}
```

Figura 50. Clase CustomAuthenticationProvider del Servidor de Autorización

Antes de autenticar al usuario se hace uso de la clase CustomUserDetailService, que se encarga de traer la información del usuario y almacenado en la base de datos. Con los datos del usuario se chequea si coincide la contraseña del usuario traído de base de datos con la contraseña que se introdujo en la ventana de autenticación que muestra el servidor de autorización al pedir las credenciales de acceso. Luego de el chequeo si todo es correcto entonces la clase AuthenticationProvider personalizada (CustomAuthenticationProvider) devuelve en un objeto el usuario autenticado con esas credenciales.

```java
@Service
@Transactional
public class CustomUserDetailService implements UserDetailsService{

    @Autowired
    private UserRepository userRepository;



    @Override
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        Usuario usuario = userRepository.findByUsername(username);


        if(usuario == null) {
            throw new UsernameNotFoundException("Usuario no encontado");
        }

        return new User(
                usuario.getUsername(),
                usuario.getPassword(),
                usuario.getEnable(),
                true,
                true,
                true,
                getAuthorities(usuario.getRoles())
        );
    }


    private Collection<? extends GrantedAuthority> getAuthorities(List<Rol> roles) {

        return roles.stream().map(rol -> new SimpleGrantedAuthority("ROLE_" + rol.getAuthority()))
                .collect(Collectors.toList());
        /*
```

Figura 51. Clase CustomUserDetailService del Servidor de Autorización

Uno de los paquetes creados dentro del servidor es el de configuración, donde se define como su nombre lo indica, toda la configuración necesaria para su funcionamiento y la configuración que va a utilizar el cliente para comunicarse con este servidor. La implementación de la clase **AuthorizationServerConfig** cuenta con cuatro Beans, que se puede identificar más adelante en la siguiente imagen.

```java
@Bean
@Order(Ordered.HIGHEST_PRECEDENCE)
public SecurityFilterChain authorizationServerSecurityFilterChain(HttpSecurity http) throws Exception {

    OAuth2AuthorizationServerConfigurer<HttpSecurity> authorizationServerConfigurer = new OAuth2AuthorizationServerConfigurer<>();
    authorizationServerConfigurer
        .authorizationEndpoint(authorizationEndpoint -> authorizationEndpoint.consentPage(CUSTOM_CONSENT_PAGE_URI));

    RequestMatcher endpointsMatcher = authorizationServerConfigurer.getEndpointsMatcher();

    http.requestMatcher(endpointsMatcher)
        .authorizeRequests(authorizeRequests -> authorizeRequests.anyRequest().authenticated())
        .csrf(csrf -> csrf.ignoringRequestMatchers(endpointsMatcher)).apply(authorizationServerConfigurer);

    return http.formLogin(Customizer.withDefaults()).build();

}

@Bean
public RegisteredClientRepository registeredClientRepository() {
    RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID().toString())

            .clientId("Film Affinity")
            .clientSecret(passwordEncoder.encode("secret"))
            .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)

            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .authorizationGrantType(AuthorizationGrantType.PASSWORD)
            .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)

            .redirectUri("http://servicio-cliente-peliculas:8080/login/oauth2/code/api-client-oidc")
            .redirectUri("http://servicio-cliente-peliculas:8080/authorized")

            .scope(OidcScopes.OPENID)
            .scope("user.info")
            .clientSettings(ClientSettings.builder().requireAuthorizationConsent(true).build())
            .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
}

@Bean
public ProviderSettings providerSettings() {
    return ProviderSettings.builder()
            .issuer("http://auth-server:9000")
            .build();
}


@Bean
public OAuth2AuthorizationConsentService authorizationConsentService() {
    // Will be used by the ConsentController
    return new InMemoryOAuth2AuthorizationConsentService();
}
```

Figura 52. Implementación de los Bean del Servidor de Autorización

En el primer Bean llamado **authorizationServerSecurityFilterChain**, tiene dos funciones principales. La primera función es la de proteger todos los endpoints de este servidor y usa el formulario de acceso por defecto usado por Spring.

```java
RequestMatcher endpointsMatcher = authorizationServerConfigurer.getEndpointsMatcher();

http.requestMatcher(endpointsMatcher)
    .authorizeRequests(authorizeRequests -> authorizeRequests.anyRequest().authenticated())
    .csrf(csrf -> csrf.ignoringRequestMatchers(endpointsMatcher)).apply(authorizationServerConfigurer);
```

Figura 53. Protección de todos los endpoints dentro de Servidor de Autorización.

La segunda función de este primer Bean es la de indicar la llamada a la ventana de consentimiento personalizada que se definió en un controlador del servidor.

```java
OAuth2AuthorizationServerConfigurer<HttpSecurity> authorizationServerConfigurer = new OAuth2AuthorizationServerConfigurer<>();
authorizationServerConfigurer
    .authorizationEndpoint(authorizationEndpoint -> authorizationEndpoint.consentPage(CUSTOM_CONSENT_PAGE_URI));
```

Figura 54. Llamada a la ventana de consentimiento personalizada

El segundo Bean llamado **registeredClientRepository**, se usa para el registro de los diferentes clientes en el servidor, concretamente definiendo los parámetros para que la aplicación cliente se conecte al servidor de autorización, como el **clientId, clientSecret,** el método de autenticación básica entre el cliente y el servidor de autorización (**clientAuthenticationMethod**). También se definen los tipos de accesos o flujos implementados por OAuth2, principalmente el de Authorization code usado para implementar toda la arquitectura, además de los scope a usar.

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
    RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID().toString())

            .clientId("Film Affinity")
            .clientSecret(passwordEncoder.encode("secret"))

            .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)

            .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
            .authorizationGrantType(AuthorizationGrantType.PASSWORD)
            .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)

            .redirectUri("http://servicio-cliente-peliculas:8080/login/oauth2/code/api-client-oidc")
            .redirectUri("http://servicio-cliente-peliculas:8080/authorized")

            .scope(OidcScopes.OPENID)
            .scope("user.info")
            .clientSettings(ClientSettings.builder().requireAuthorizationConsent(true).build())
            .build();

    return new InMemoryRegisteredClientRepository(registeredClient);
}
```

Figura 55. Bean de registro de clientes en el Servidor de Autorización

El tercer y cuarto Bean solamente se define cual es el la URL del servidor de autorización y el servicio para utilizar la ventana de consentimiento personalizada implementada en el controlador del servidor.

```
/**
 * Acceptable URL of the authorization server
 */
@Bean
public ProviderSettings providerSettings() {
    return ProviderSettings.builder()
            .issuer("http://auth-server:9000")
            .build();
}


@Bean
public OAuth2AuthorizationConsentService authorizationConsentService() {
    // Will be used by the ConsentController
    return new InMemoryOAuth2AuthorizationConsentService();
}
```

Figura 56. Tercer y cuarto Bean de configuración del Servidor de Autorización

5. CONCLUSIONES Y FUTURAS LÍNEAS DE TRABAJO

Podemos concluir que se ha dado cumplimiento al objetivo principal de este trabajo. Se ha dotado de **seguridad a un proyecto web** que está basado en la arquitectura de microservicios con la **implementación del protocolo OAuth**. Se abordaron los conocimientos necesarios para llegar a este objetivo, desde conocer elementos esenciales de la arquitectura a desarrollar, temas de seguridad con el Framework Spring, su implementación y conceptos principales del protocolo. Se implementaron los diferentes roles de OAuth además de hacer coincidir esta implementación a la los diferentes elementos de arquitectura orientada a microservicios. Se hizo una implementación personalizada del elemento principal como lo es el servidor de autorización, donde según en el ámbito de la seguridad es de gran importancia tener totalmente controlado qué elementos y librerías se usan de terceros que a veces no se tiene el control de lo que realmente estas implementan y puede terminar introduciendo vulnerabilidades dentro de nuestro sistema. Esta implementación propia del servidor de autorización sería una gran ventaja con respecto a la utilización de servidores de autorización externos como lo son Okta, Keycloack, Google, Facebook, debido a que en gran medida se controla toda la información que va dirigida a este servidor y no se delega a un servidor sobre el cual se tiene un control parcial de la misma. Aun así, se pudo llegar en gran medida con esta implementación del servidor de autorización y los demás componentes de la arquitectura de microservicios a un nivel tan detallado de personalización y explicación de la implementación de sus componentes.

Luego de haber finalizado el trabajo completamente y cumplido el objetivo de este, nos podríamos estar preguntando cuáles serían los aspectos para tener en cuenta y las posibles líneas de trabajo futuro.

Primeramente y como algo pendiente a desarrollar para un futuro tanto de aprendizaje como de posible implementación en entornos o aplicaciones reales, sería idóneo **crear una página web personalizada** para la **autenticación de usuarios dentro del servidor de autorización**. El servidor de autorización a la hora de solicitar las credenciales del usuario muestra el típico formulario por defecto de Spring Security donde solicita usuario y contraseña y sin ningún tipo de personalización. Sería conveniente dotar a esta página con poco un más de estilo y detalles que sean más agradables al usuario final.

Otro de los aspectos a señalar para el trabajo en un futuro sería la **independencia en la gestión de los usuarios**. Debido a que nuestro servidor de autorización en un futuro puede pasar a estar ubicado de forma independiente de nuestros servicios, por tanto, es conveniente que este implemente su propia base de datos. ¿Cómo se encuentra actualmente?. Nuestro servidor de autorización hace uso de la base de datos que gestiona nuestro servicio de usuario de la arquitectura. Simplemente, a la hora de aislar nuestro servidor de autorización en algunas ocasiones, no necesariamente, sería más recomendable separar en dos bases de datos diferentes, la base de datos a la cual accede el servicio de los usuarios de la aplicación y la base de datos de dicho servidor. Actualmente hay servidores que no están aislados del servidor de recursos pueden llegan a trabajar quizás con una base de datos compartida como lo hace Google o Facebook al hacer la función de servidor de autorización y de recursos a la vez. Ya la decisión va asociada al conocimiento del desarrollador de la aplicación y de las necesidades y tamaño de esta.

6. BIBLIOGRAFÍA

[1] C. Harris. *Comparación entre la arquitectura monolítica y la arquitectura de microservicios*. Available: https://www.atlassian.com/es/microservices/microservices-architecture/microservices-vs-monolith

[2] Wikipedia. (2022). *Aplicación monolítica*. Available: https://es.wikipedia.org/wiki/Aplicaci%C3%B3n_monol%C3%ADtica

[3] I. Gorton, *Foundations of Scalable Systems*. O'Reilly, 2022.

[4] (2020). *¿Qué es la arquitectura orientada a los servicios (SOA)?* Available: https://www.redhat.com/es/topics/cloud-native-apps/what-is-service-oriented-architecture

[5] O. Open. *Organization for the advancement of structured information standards*. Available: http://www.oasis-open.org

[6] N. Bieberstein, S. Bose, M. Fiammante, K. Jones, and R. Shah, *Service-oriented architecture compass: business value, planning, and enterprise roadmap*. FT Press, 2006.

[7] M. H. Valipour, B. AmirZafari, K. N. Maleki, and N. Daneshpour, "A brief survey of software architecture concepts and service oriented architecture," in *2009 2nd IEEE International Conference on Computer Science and Information Technology*, 2009, pp. 34-38: IEEE.

[8] IBM. (2022). *Arquitectura orientada a servicios*. Available: https://www.ibm.com/docs/es/baw/20.x?topic=designer-service-oriented-architecture

[9] S. J. O. Newman and A. Inc, "Building microservices: Designing fine-grained systems," 2015.

[10] AWS. *Microservicios*. Available: https://aws.amazon.com/es/microservices/

[11] P. Di Francesco, P. Lago, and I. Malavolta, "Migrating towards microservice architectures: an industrial survey," in *2018 IEEE International Conference on Software Architecture (ICSA)*, 2018, pp. 29-2909: IEEE.

[12] C. Richardson, *Microservices patterns: with examples in Java*. Simon and Schuster, 2018.

[13] S. Newman, *Building microservices*. O'Reilly Media, Inc, 2021.

[14] C. Richardson. *Pattern: Decompose by business capability*. Available: https://microservices.io/patterns/decomposition/decompose-by-business-capability.html

[15] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 2003.

[16] S. Petunin. (2021). *A Guide to Transactions Across Microservices*. Available: https://www.baeldung.com/transactions-across-microservices

[17] K. Brown. (2020). *Apply the Strangler Fig Application pattern to microservices applications*. Available: https://developer.ibm.com/articles/cl-strangler-application-pattern-microservices-apps-trs/

[18] R. Bhojwani. (2022). *Design Patterns for Microservices*. Available: https://dzone.com/articles/design-patterns-for-microservices

[19] AWS, "Patrón de base de datos compartida por servicio."

[20] M. Fowler. (2005). *Event Sourcing*. Available: https://martinfowler.com/eaaDev/EventSourcing.html

[21] G. Marigi. (2019). *Saga: The new era of transactions in a microservices architecture*. Available: https://www.redhat.com/files/summit/session-assets/2019/T42224.pdf

[22] Spring. *Projects*. Available: https://spring.io/projects

[23] Spring. *Architecture*. Available: https://docs.spring.io/spring-security/reference/servlet/architecture.html

[24] StackOverflow. (2019). *Difference between Role and GrantedAuthority in Spring Security*. Available: https://stackoverflow.com/questions/19525380/difference-between-role-and-grantedauthority-in-spring-security

[25] Spring. (2020). *Feature List*. Available: https://docs.spring.io/spring-authorization-server/docs/current/reference/html/overview.html#feature-list

[26] Wikipedia. (2022). *OAuth*. Available: https://es.wikipedia.org/wiki/OAuth

[27] E. D. Hardt. (2012). *The OAuth 2.0 Authorization Framework*. Available: https://datatracker.ietf.org/doc/html/rfc6749

[28] Oracle. *Step-by-Step Workflow of the Resource Owner Password Credentials Grant*. Available: https://docs.oracle.com/en/cloud/get-started/subscriptions-cloud/csimg/step-step-workflow-resource-owner-password-credentials-grant.html

[29] Okta. (2018). *What is the OAuth 2.0 Implicit Grant Type?* Available: https://developer.okta.com/blog/2018/05/24/what-is-the-oauth2-implicit-grant-type

[30] E. N. Sakimura. (2015). *Proof Key for Code Exchange by OAuth Public Clients*. Available: https://www.rfc-editor.org/rfc/rfc7636

[31] Okta. *Differences Between OAuth 1 and 2*. Available: https://www.oauth.com/oauth2-servers/differences-between-oauth-1-2/

[32] A. Parecki, *OAuth 2.0 Simplified: A Guide to Building OAuth 2.0 Servers*. Lulu Press, Inc, 2018.

7. APÉNDICE A. MANUAL DE USO DE LA APLICACIÓN

Para acceder a la web creada es necesario primeramente iniciar todos y cada uno de los componentes de la arquitectura descritos anteriormente. Luego de iniciados , el servidor de eureka, el Gateway, el servidor de autorización , los servicios o servidores de recursos y la aplicación cliente, entonces se procede a acceder a ella. Para acceder a la web se introduce en el navegador la siguiente dirección URL http://servicio-cliente-peliculas:8080/ y se muestra el listado de películas de la página inicial:

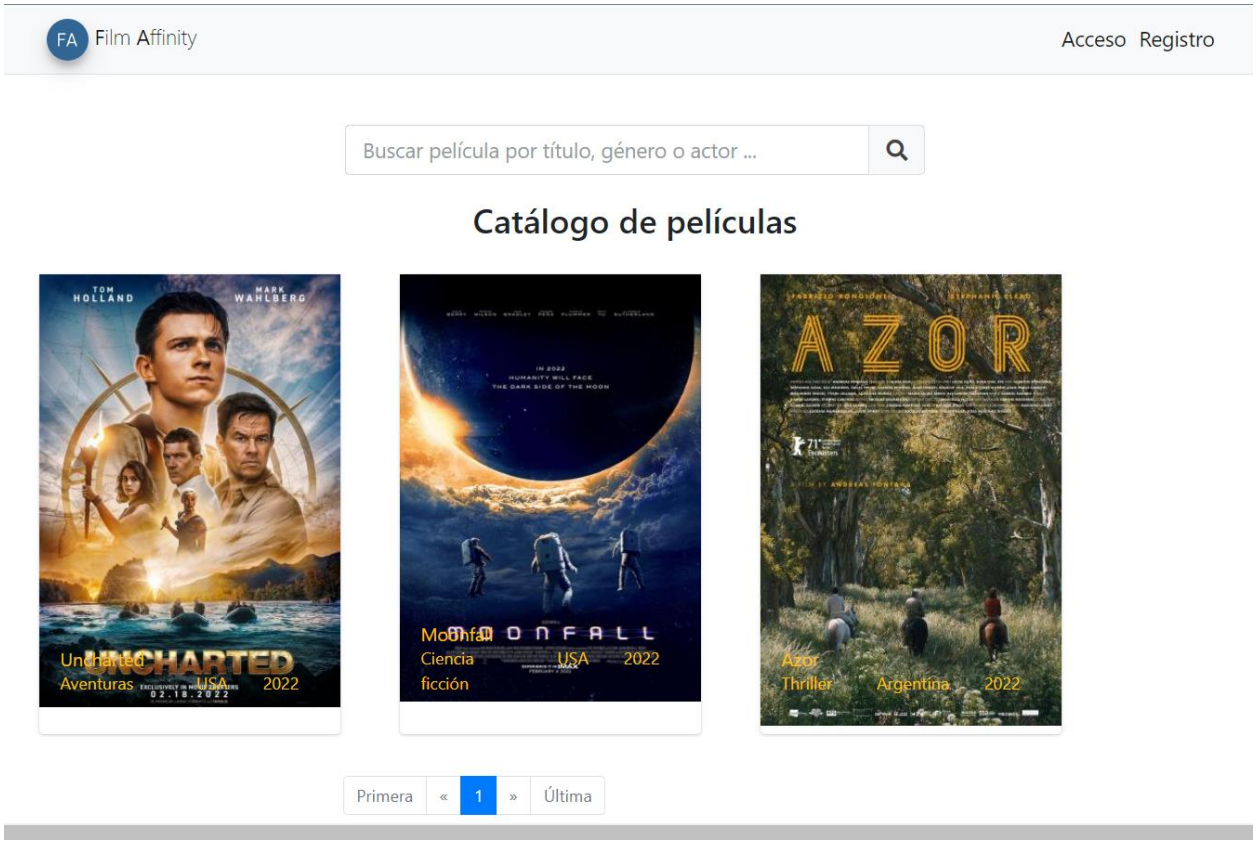


Figura 57. Listado películas de la página inicial

Se puede seleccionar una película para ver un conjunto de detalles como su Sinopsis, Dirección, Duración, Género, País , Año y Actores además de que se puede hacer una crítica a cada una de las películas otorgándole una puntuación.

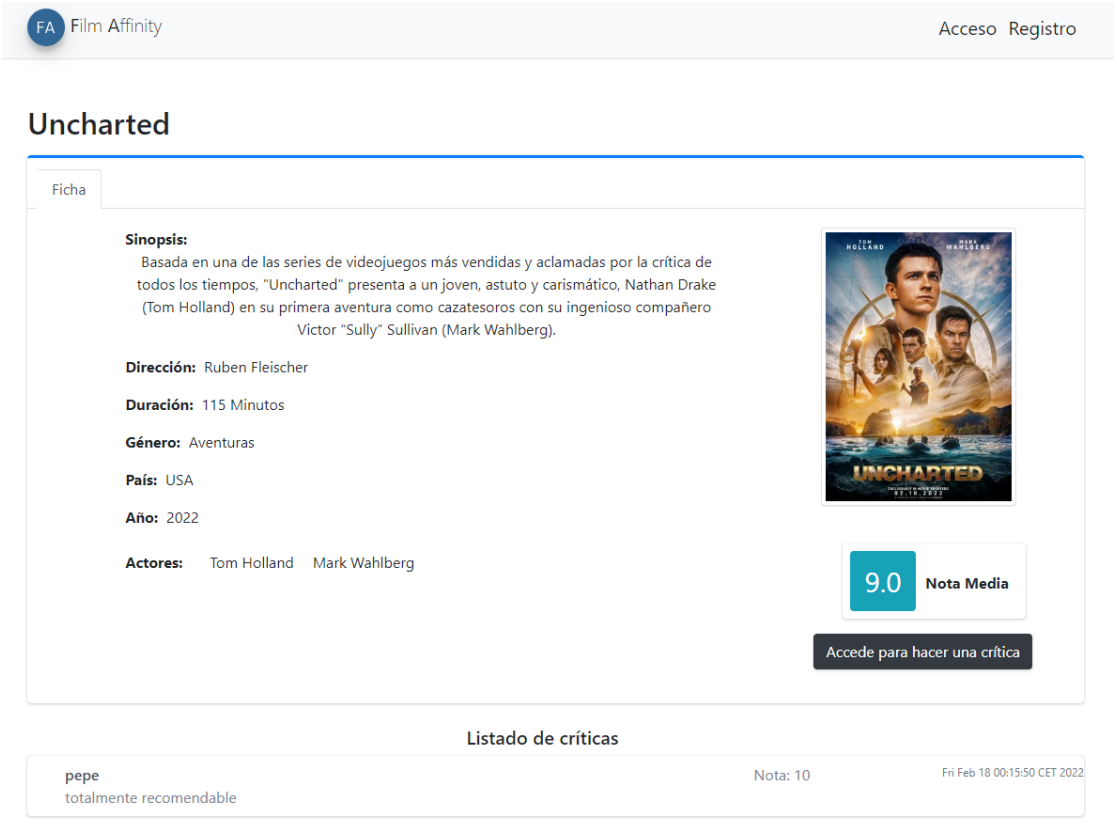


Figura 58. Detalles de la película

El usuario en la página principal, donde se muestra el catálogo de películas, tiene la opción de registrarse a la web o acceder a ella si ya tiene creada una cuenta.

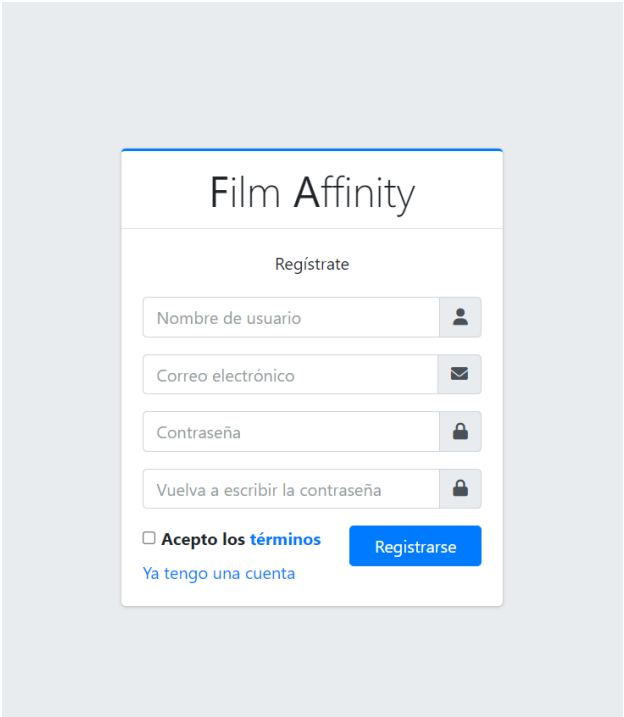


Figura 59. Formulario de registro

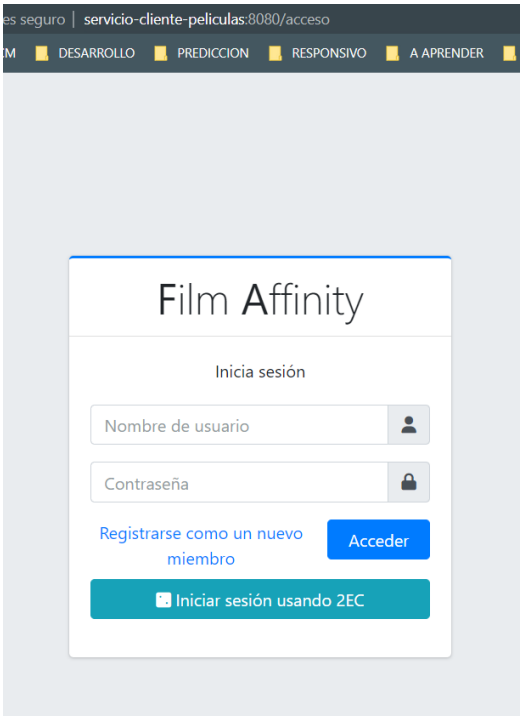


Figura 60. Formulario de acceso

El usuario puede acceder al sitio en el formulario anterior, proporcionando sus credenciales o a través del botón que dice "**Iniciar sesión usando 2EC**". Al accionar el botón de acceder con 2EC, se daría inicio al flujo de OAuth2 **Authorization code** que se implementó en la aplicación. Al accionarlo se visualiza que se redirige de http://servicio-cliente-peliculas:8080/acceso a otra página http://auth-server:9000/login, que sería la URL de acceso dentro del propio servidor de autorización, como se muestra a continuación.

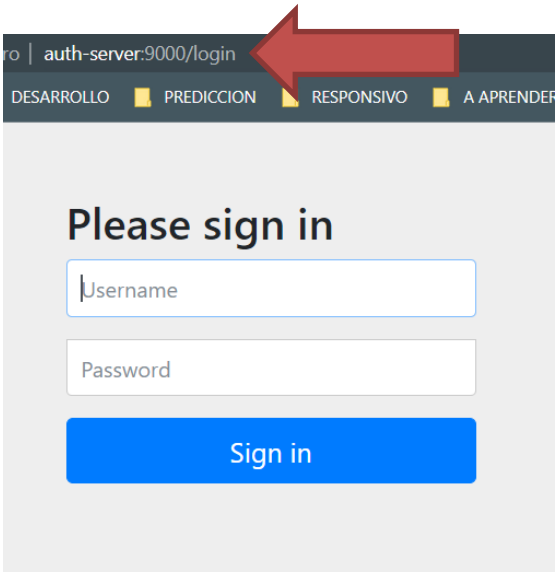


Figura 61. URL de acceso a través del servidor de autorización.

Si se habilita el modo desarrollador del navegador para observar las peticiones que se están realizando, podemos observar que según el flujo de OAuth2, se envía una petición y se espera que el tipo de respuesta sea un código (**response_type=code**), a través de una redirección de la propia URL.

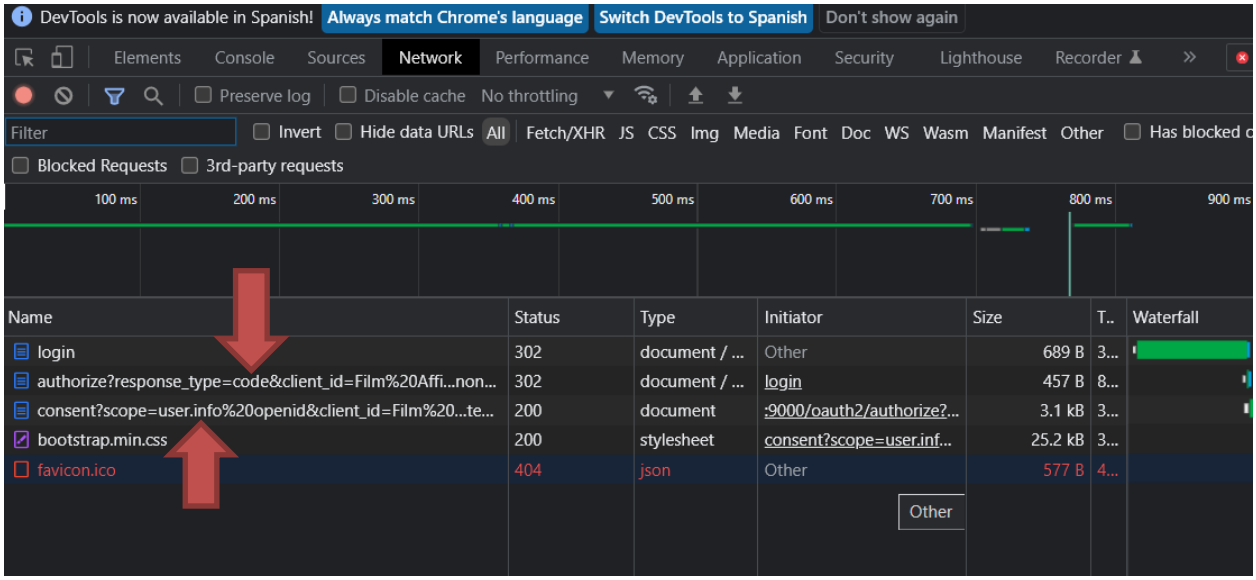


Figura 62. Tipo de solicitud que espera una respuesta con un código

Al poner las credenciales de acceso en el sitio, se observa que a la par de que se observa la petición que espera el Código que se intercambia por el token, también se hace una segunda petición. Esta segunda petición es la de la ventana de consentimiento, donde se le pregunta al propietario del recurso (usuario) si desea dar permisos a la aplicación de gestión de películas, según el scope definido para el cliente. Ver la siguiente figura.

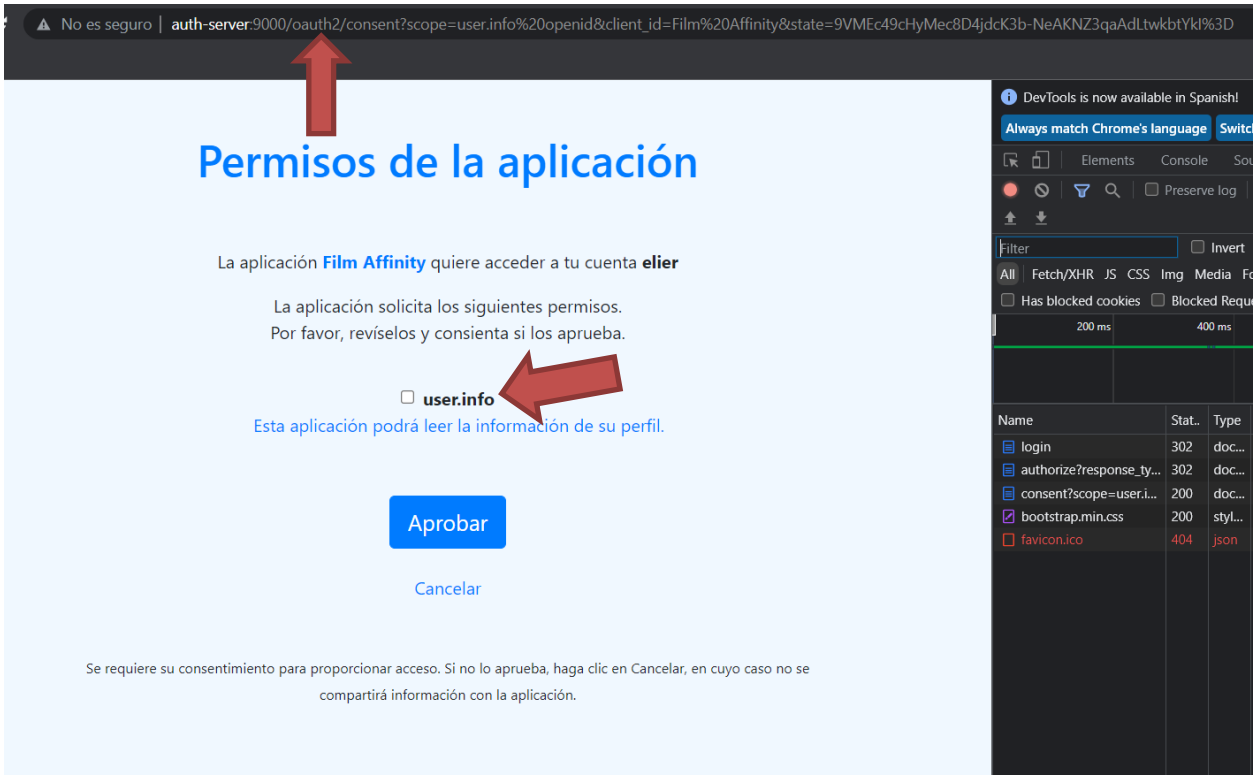


Figura 63. Ventana de consentimiento de la aplicación de gestión de películas

Si el usuario aprueba con su consentimiento, pues entonces, se obtiene un código que es el que se utiliza para intercambiar por el token de acceso que emite el servidor de autorización. Con este código lo

intercambiamos por un token que es que se utiliza para acceder a un recurso protegido. Hay que destacar que este token se almacena en el navegador como una cookie.

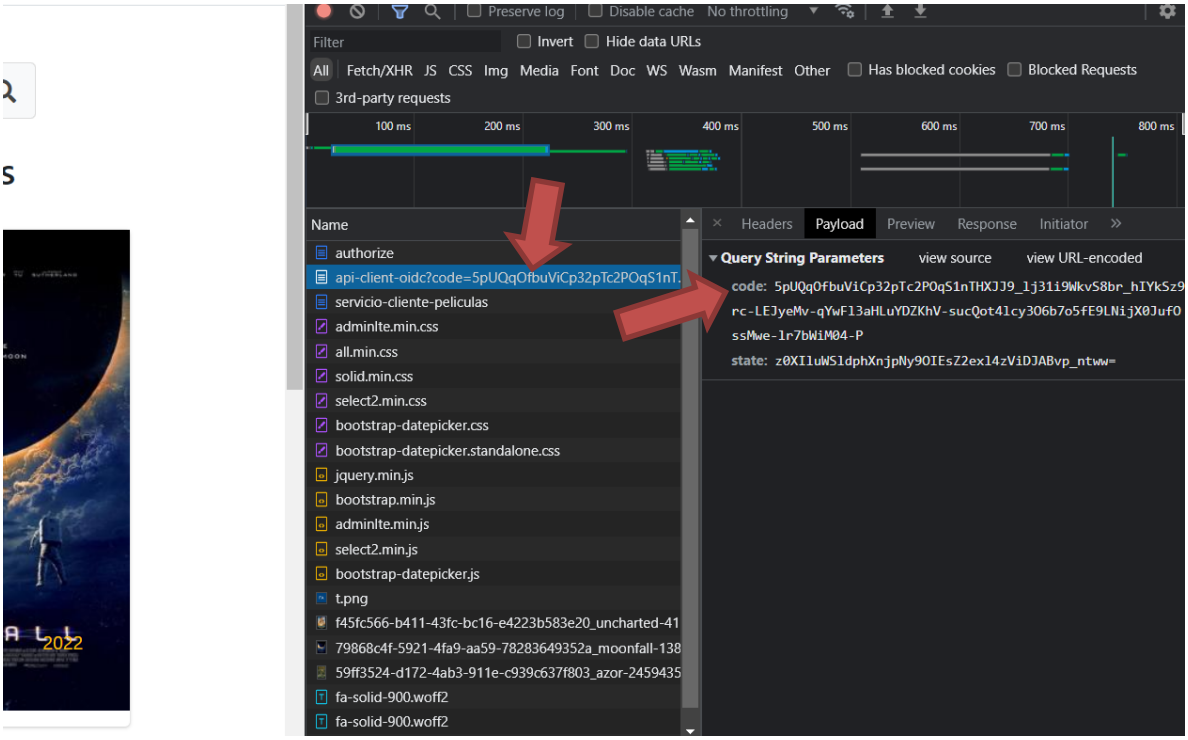


Figura 64. Código intercambiable por el token

Como se comentaba anteriormente, el token no es capaz de apreciarse porque es almacenado como una cookie dentro del navegador. Al realizar la misma petición utilizando el postman, se puede observar el token que se obtiene de realizar el flujo completo de OAuth2 Authorization code.

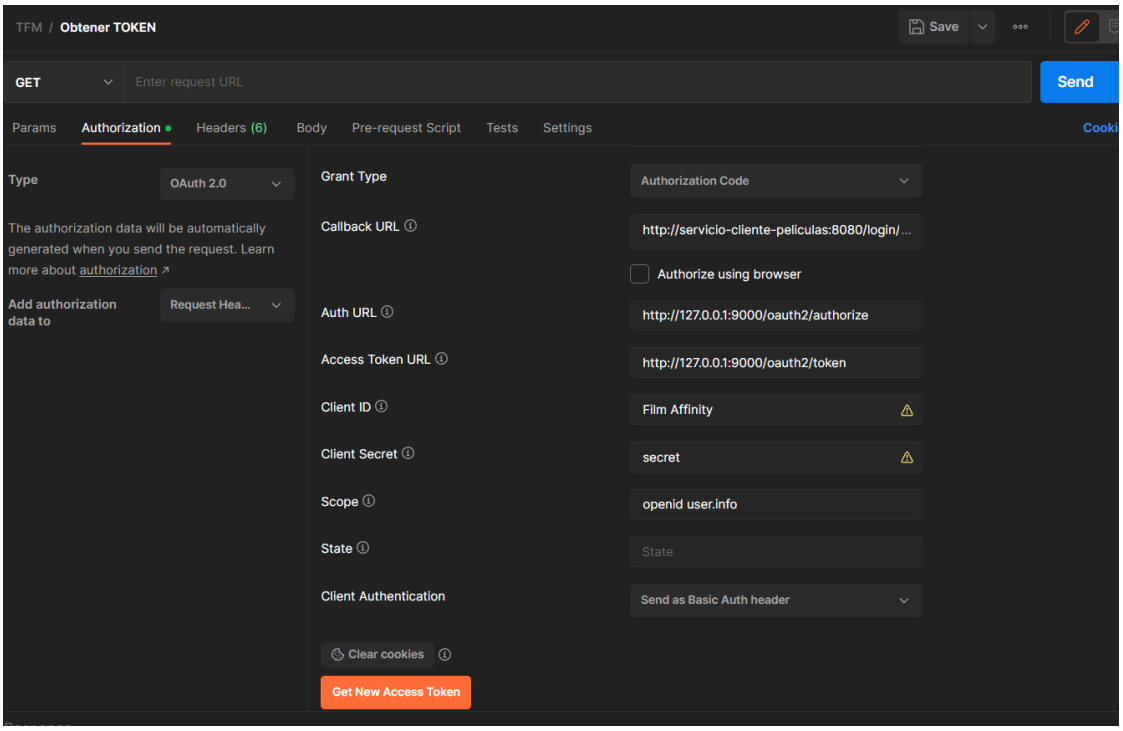


Figura 65. Obtener un token de acceso

Luego de hacer la petición para obtener el token de acceso, se abre una ventana nueva dentro del postman para solicitar las credenciales de acceso del usuario.

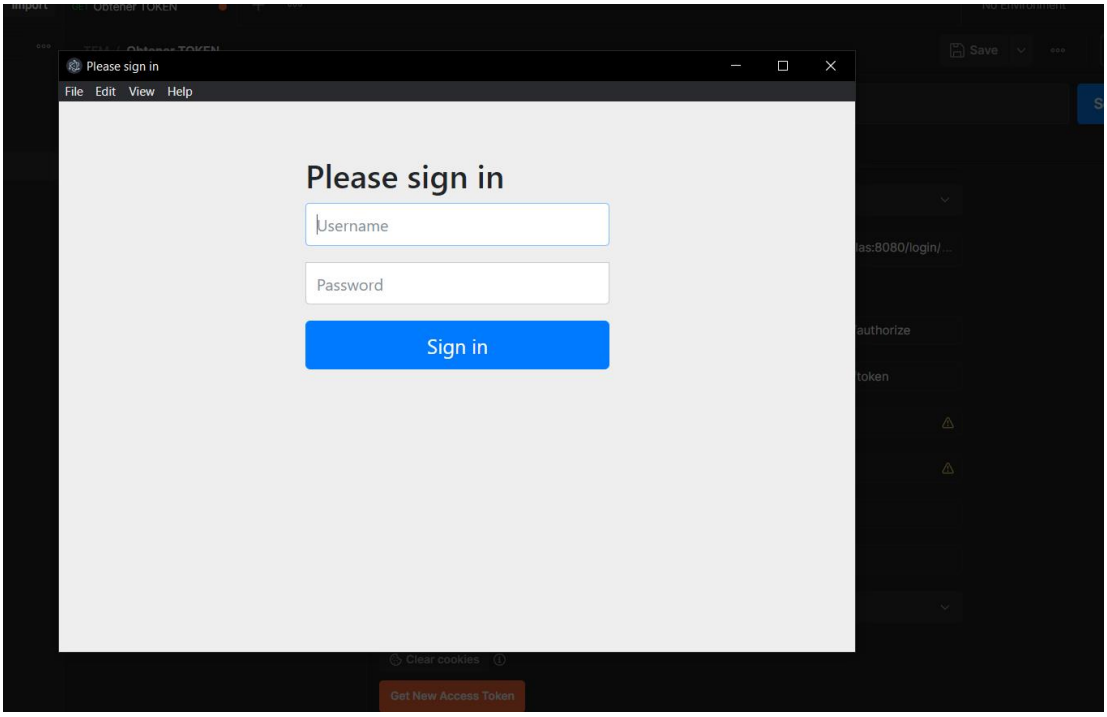


Figura 66. Credenciales del usuario

Luego de enviar las credenciales de acceso sale la ventana de consentimiento mostrada en figuras anteriores, al aprobar el permiso que se le otorga al cliente para acceder al recurso entonces la autenticación se ha completado correctamente y el servidor de autorización nos ha enviado el token de acceso.

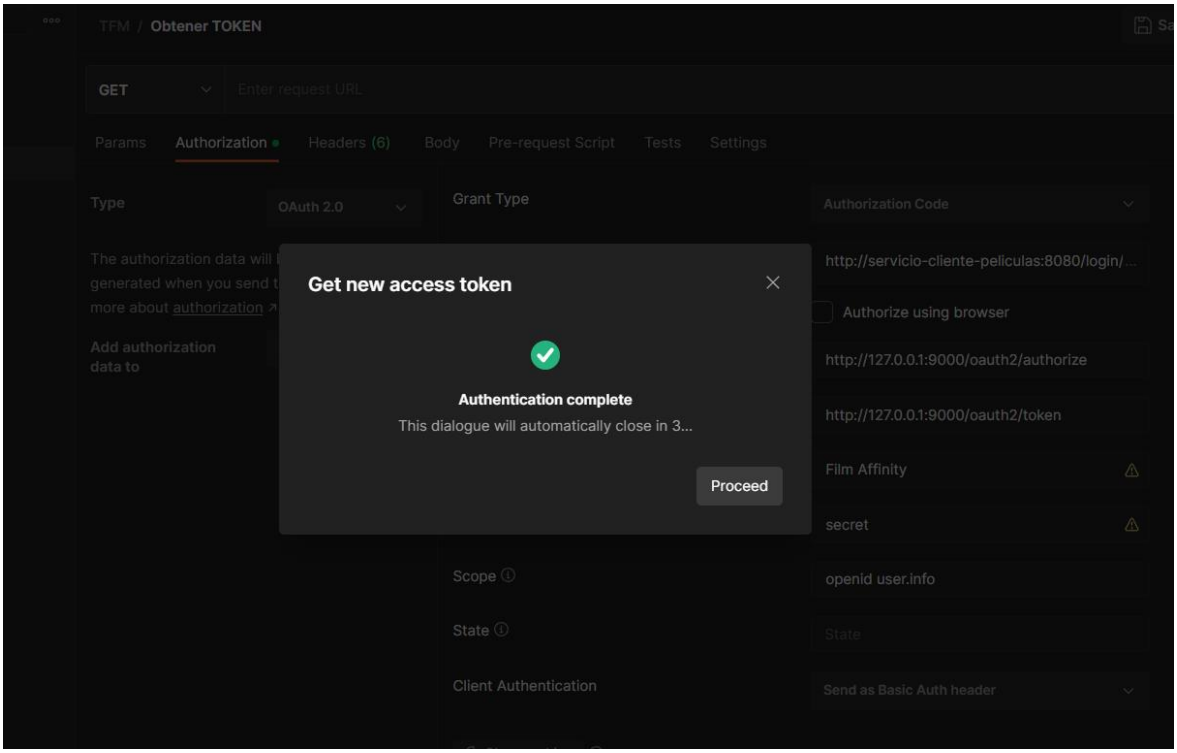


Figura 67. Autenticación correcta

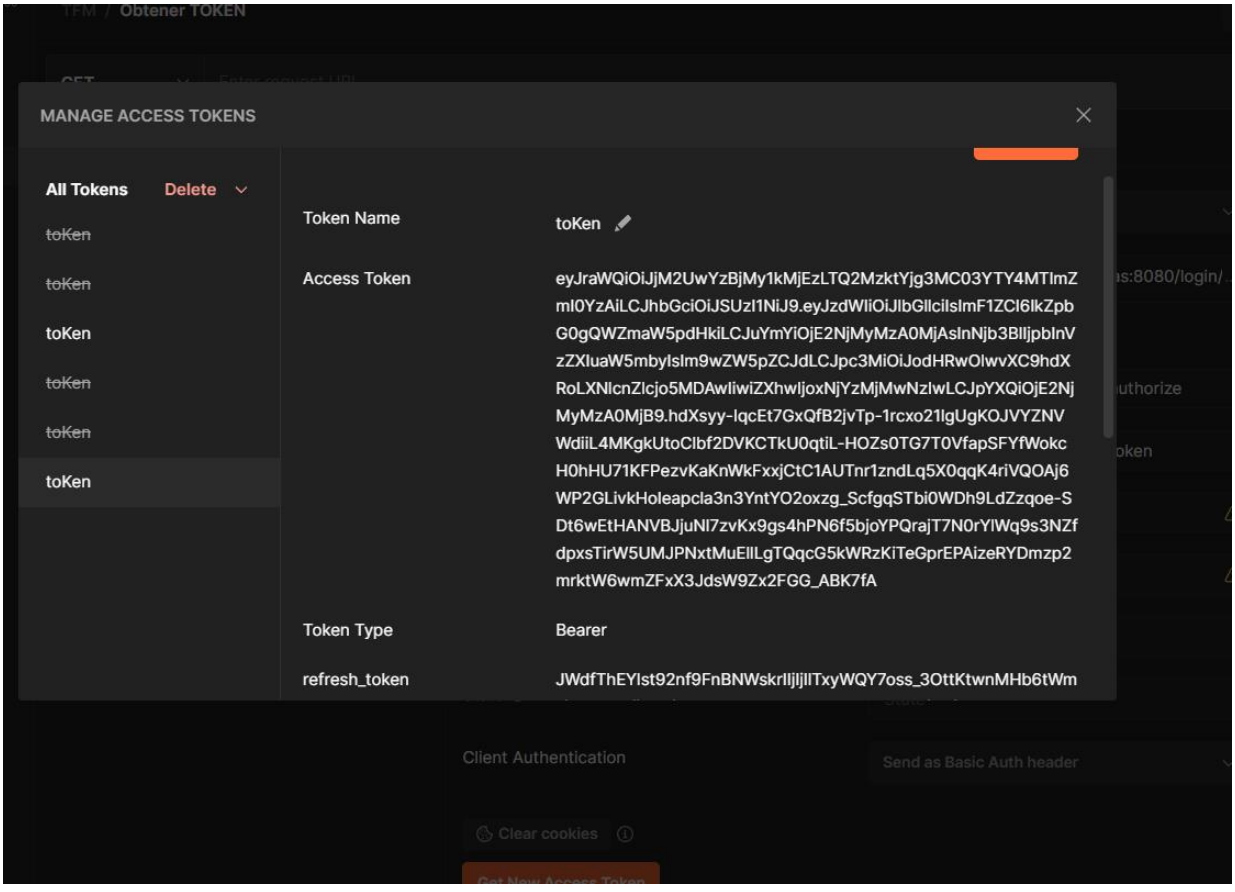


Figura 68. Token de acceso

Al proceder a copiar el token y dirigirnos a la página https://jwt.io/ para decodificar el token se puede observar la información dentro del token que fue emitida por el servidor de autorización.

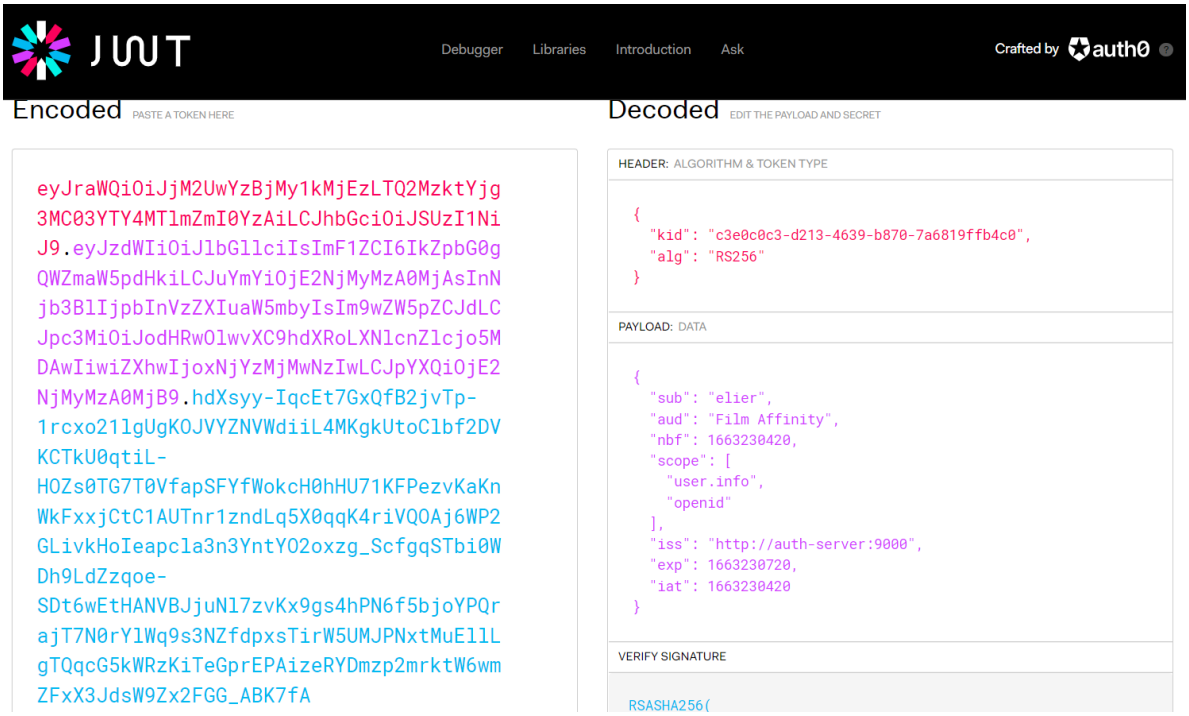


Figura 69. Token de acceso decodificado