Arquitecturas Orientadas a Servicios

OAuth 2.0 y JWT

Grado en Ingeniería Informática en Ingeniería del Software

Curso 2024-2025

Pablo Fernández González

Sara Guillén Torrado

Índice

Listings

Índice de figuras

Objetivo del proyecto. Entregables

El principal objetivo de este proyecto es **comprender el funcionamiento del estándar OAuth**. Para ello, se realizará un desarrollo teórico sobre OAuth, que se complementará con una explicación del estándar JWT (*JSON Web Token*), un tipo de *token* que se suele utilizar junto a OAuth. Posteriormente, se realizará una Demo de OAuth 2.0 paso a paso para probar el funcionamiento del protocolo con un ejemplo de juguete.

Se hace entrega de esta **memoria** del proyecto, que contiene tanto el desarrollo teórico como la guía de las prácticas. El material necesario para la realización de la demo y su solución se encuentran en el siguiente repositorio de GitHub: `https://github.com/saguit03/oauth-demo`.

Asimismo, se realizó una presentación teórico-práctica en clase, el pasado 16 de diciembre de 2024. Se adjunta un enlace a dicha presentación: `https://www.canva.com/design/DAGZAILOgfo/66ntjtxlcby11hS2LTRl6A/edit?utm_content=DAGZAILOgfo&utm_campaign=designshare&utm_medium=link2&utm_source=sharebutton`.

Durante esta presentación, se realizó una actividad con los compañeros. Se adjunta el enunciado de esta actividad en el Anexo. Kahoot: OAuth y JWT.

1. Introducción

OAuth 2.0 (*Open Authorization*), en español «autorización abierta», es un **estándar de seguridad** definido en el RFC 6749 ([IETF, 2012]) y diseñado para permitir que un sitio web o una aplicación accedan a recursos alojados por otras aplicaciones web en nombre de un usuario. ([Auth0, 2024]).

Su función es **otorgar acceso consentido** a recursos, y restringe las acciones que la aplicación del cliente puede realizar en los recursos en nombre del usuario, **nunca compartiendo las credenciales** del usuario.

Puede ser desplegado en plataformas web, aplicaciones basadas en el navegador, aplicaciones web del lado del servidor, aplicaciones nativas/móviles, dispositivos conectados, etc.

De ahora en adelante, se trabajará sobre **OAuth 2.0**, por lo que, si no se especifica la versión, esta será la predeterminada.

OAuth 1.0, el antecesor de OAuth 2.0, era más complejo de implementar, lo que limitaba sus posibilidades. Además, la terminología usada, como los roles, difería bastante con la actual ([OAuth, 2024]).

OAuth 2.0 destaca por su **ligereza y flexibilidad** en una gran variedad de arquitecturas. Utiliza *tokens* [1] de acceso y puede manejar múltiples flujos de autorización.

Una herramienta similar es el estándar SAML (*Security Assertion Markup Language*), más enfocado a la autenticación a través de un **proveedor de identidades**. OAuth a veces se ve forzado a desempeñar el rol de autenticación, aunque no soporta **SSO** (*Single Sign-On*), como sí hace SAML. ([Auth0, sfa]).

SAML es más complejo que OAuth y se basa en XML, por lo que es bastante más pesado que OAuth y menos adecuado para API (*Application Programming Interface*) modernas o sistemas distribuidos.

Hay que destacar OpenID Connect (OIDC), un protocolo que se basa en OAuth que suple la falta de autenticación de OAuth. Tanto OIDC como SAML son estándares que pueden **complementar a OAuth**. ([Auth0, sfb]).

[1] *Token* en español: identificador, evidencia o prueba)

2. OAuth 2.0

2.1. Motivos de uso

En la autenticación tradicional de cliente-servidor, el cliente solicita un recurso de acceso restringido autenticándose en el servidor con las credenciales del propietario del recurso.

Sin embargo, requiere compartir a terceros las credenciales del propietario de los recursos, lo que puede acarrear ciertos problemas y limitaciones:

- Las aplicaciones de terceros deben almacenar las credenciales del credenciales del propietario del recurso para su uso futuro, normalmente en formato de **contraseña en texto claro**.
- Se exige a los servidores que admitan la **autenticación por contraseña**, a pesar de las debilidades de seguridad inherentes a las contraseñas.
- Los propietarios **no tienen capacidad de restringir la duración o el acceso** por parte de los terceros a un subconjunto limitado de recursos.
- Los propietarios de recursos **no pueden revocar el acceso a un tercero individual** sin revocarlo a todos los terceros, y se debe hacer cambiando la contraseña del tercero.
- El **compromiso de cualquier aplicación de terceros** resulta en el **compromiso de contraseña del usuario final** y todos aquellos **datos protegidos** por esa contraseña.

OAuth, por otro lado, aborda estos problemas introduciendo una **capa de autorización** y separando el papel del cliente del del propietario del recurso: el cliente solicita acceso a los recursos controlados por el propietario de los recursos y alojados en el servidor de recursos, y recibe un **conjunto de credenciales distinto al del propietario** de los recursos.

2.2. Principios

OAuth 2.0 es un **protocolo de autorización**, diseñado para conceder **acceso a recursos**, como API o datos de usuario, en nombre del usuario final. Utiliza ***tokens* de acceso**, los cuales representan la autorización para acceder a los recursos en nombre del usuario final esta autorización.

Aunque OAuth 2.0 no especifica un formato para los *tokens*, el formato **JWT** (*JSON Web Token*) es comúnmente empleado para incluir datos en ellos. Más adelante se explicarán los JWT con más detalle en JSON Web Token (JWT). Por motivos de seguridad, los *tokens* suelen tener una fecha de expiración. ([Auth0, 2024]).

2.3. Roles de OAuth 2.0

Los siguientes **roles** toman un papel clave durante el funcionamiento del protocolo OAuth ([Auth0, 2024], [OAuth, 2024]):

- **Propietario del recurso**: El usuario o sistema que **posee** los recursos protegidos y **puede conceder acceso** a ellos.
- **Cliente**: El sistema que **solicita acceso** a los recursos protegidos, requiriendo un *token* de acceso válido.
- **Servidor de autorización**: Gestiona la **emisión de *tokens*** de acceso, autenticando al propietario del recurso y recibiendo su consentimiento. Proporciona dos puntos de conexión: uno para la autorización (autenticación y consentimiento) y otro para la emisión de *tokens*.
- **Servidor de recursos**: **Protege** los recursos del usuario y **valida** los *tokens* de acceso presentados por el cliente para permitir el acceso a los recursos.

2.4. Funcionamiento de OAuth 2.0

Como paso preliminar, el protocolo OAuth 2.0 requiere que la aplicación cliente obtenga sus **credenciales específicas** (identificador de cliente y secreto del cliente) **del servidor de autorización**. Estas credenciales son esenciales para la posterior autenticación durante las solicitudes de *tokens* de acceso ([Auth0, 2024], [Hardt, 2012]).

El protocolo inicia cuando una aplicación (ya sea móvil, web u otra plataforma) requiere acceso a los recursos de un servidor. Se sigue una secuencia estructurada ([Hardt, 2012]):

1. Inicialmente, **la aplicación cliente emite una solicitud de autorización al servidor de autorización**, presentando sus credenciales identificativas. Adicionalmente, especifica los alcances requeridos y proporciona una URI (*Uniform Resource Identifier*) de redirección donde recibirá la respuesta.

2. Posteriormente, **el servidor de autorización procede a autenticar la solicitud del cliente** y valida que los alcances solicitados sean apropiados.

3. En la siguiente fase, se establece una **interacción entre el propietario del recurso y el servidor de autorización para confirmar la concesión de acceso**.

4. A continuación, **el servidor de autorización efectúa una redirección hacia el cliente**, transmitiendo ya sea un código de autorización o un *token* de acceso, dependiendo del método de concesión implementado. En ciertos casos, también puede incluirse un *token* de actualización.

5. Finalmente, una vez obtenido el *token* de acceso, **la aplicación cliente puede proceder a solicitar acceso a los recursos específicos** alojados en el servidor de recursos.

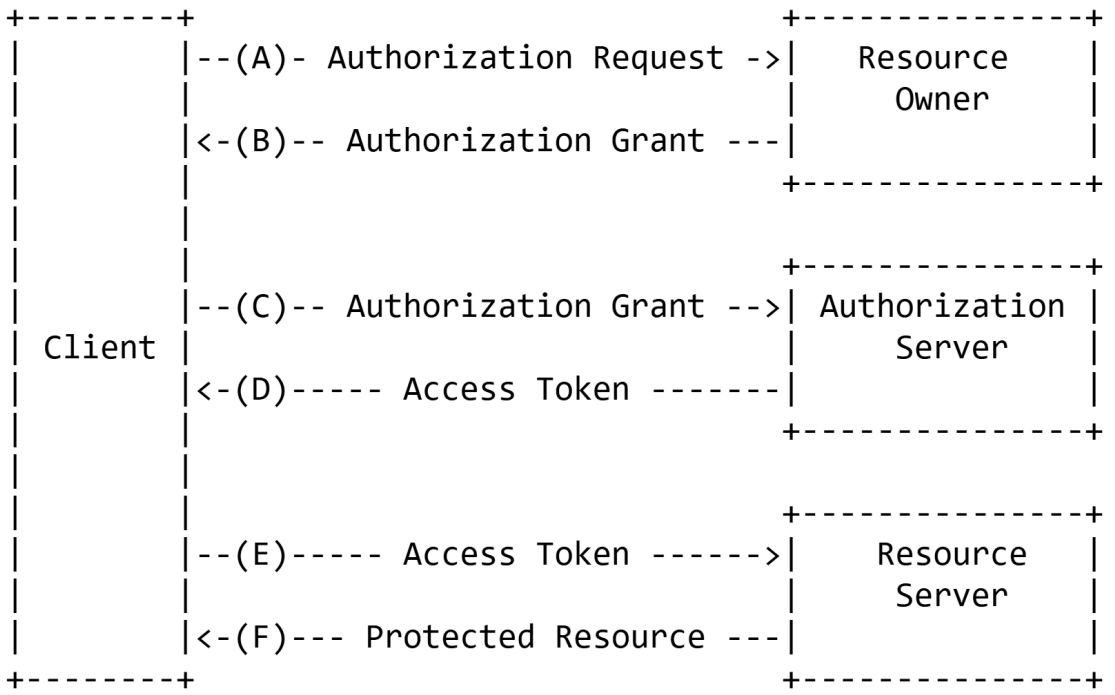

Figura 1: Flujo de autorización en OAuth ([Hardt, 2012]).

Refrescar token

Una forma alternativa de funcionamiento consiste en **renovar** periódicamente el **token de acceso a recursos**, con el fin de limitar el tiempo que se tiene acceso a estos.

Se sigue un proceso similar al anterior, con ciertas modificaciones ([Hardt, 2012]):

1. El cliente solicita un **token de acceso** autenticándose con el servidor de autorización, y presentando una concesión de autorización.
2. El servidor de autorización autentica al cliente y valida la concesión de autorización y, si es válida, no solo emite un **token de acceso** como se describió anteriormente, sino que se incluye un **token de actualización**, que permite renovarlo.
3. El cliente realiza una solicitud de recurso protegido al servidor de recursos presentando el *token* de acceso.
4. El servidor de recursos valida el *token* de acceso y, si es válido, devuelve el recurso solicitado.
5. Los pasos 3 y 4 se repiten hasta que que el *token* de acceso **expira** y se invalida.
6. El cliente, de nuevo, solicita un *token* de acceso autenticándose con el servidor de autorización y presentando el *token* correspondiente.
7. Finalmente, el servidor de autorización autentica al cliente y valida el *token* de actualización. Si es aceptado, emite un **nuevo *token* de acceso** (y, **opcionalmente**, un nuevo **token de actualización**).

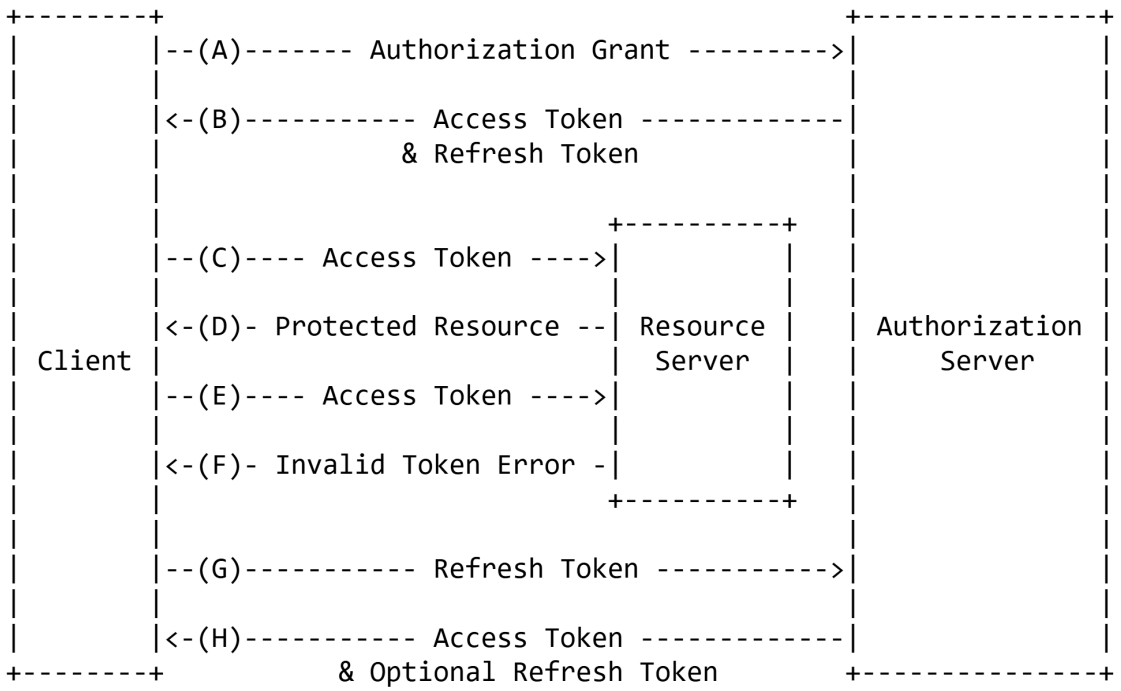

Figura 2: Refrescar un *token* ([Hardt, 2012]).

2.5. Métodos de concesión de autorización

Concesión implícita

Se devuelve el *token* de acceso directamente al cliente, ya sea como un parámetro de URI o en respuesta a la publicación de un formulario. Recomendable con *Single Page Applications* (SPA) sin necesidad de *tokens* de acceso.

Si bien mejora la **eficiencia** de algunos clientes al reducir el número de redirecciones necesarias en la obtención de *tokens* de acceso, pueden contraerse riesgos de **seguridad** con su uso, tales como la obtención del *token* por parte de terceros a partir de la URI enviada como respuesta a la petición.

Código de autorización

El servidor de autorización devuelve al cliente un código de autorización de un único uso, que se intercambia por un *token* de acceso.

Supone la opción más recomendada para **aplicaciones web tradicionales** en las que el intercambio puede realizarse de forma segura en el lado del servidor. El flujo del código de autorización puede ser utilizado por aplicaciones de página única (*Single Page Apps*, SPA) y aplicaciones móviles/nativas.de un único uso, utilizado en aplicaciones web (SPA).

Precisamente se utilizará este método de concesión en la realización de la Demo de OAuth 2.0 paso a paso.

Sin embargo, en este caso, el ***client secret*** no puede almacenarse de forma segura, por lo que la autenticación durante el intercambio se limita al uso del id del cliente únicamente

Código de autorización con concesión PKCE

La PKCE (*Proof Key for Code Exchange*) es similar a la concesión del Código de autorización, pero con pasos adicionales que lo hacen más seguro para las aplicaciones móviles/nativas y SPA.

Esta mejora previene falsificaciones de peticiones por terceros (CSFR, *Cross-Site Request Forgery*) e ataques de inyecciones sobre códigos de autorización.

Credenciales de contraseña del propietario

Las credenciales del propietario pueden ser utilizadas como *tokens* de autorización para solicitar *tokens* de acceso a los recursos.

Este tipo de concesión de autorización se limita a los **clientes de mayor confianza** del proveedor, o al no estar disponibles otros tipos de concesión.

Si bien se requiere que el cliente acceda directamente a las credenciales del propietario de los recursos, estas son utilizadas para una única petición a cambio de un *token* de acceso de larga duración o la posibilidad de renovarlo una vez expire.

Credenciales de contraseña del usuario

Se utiliza para aplicaciones que necesitan acceder a sus propios recursos en un servidor o solicita acceso a recursos protegidos basados en una autorización previamente otorgada por el servidor de autorización. Estos se autentican por sí mismos mediante el uso de su id de cliente y su *client secret*

Se reserva el uso de este tipo de concesión a **clientes de máxima confidencialidad**, dada la facilidad de solicitud de *tokens* de acceso.

3. JSON Web Token (JWT)

JSON Web Token (JWT a partir de ahora) es un estándar (RFC 7519, [IETF, 2015b]) que define un método compacto y autocontenido para la transmisión segura de información entre dos partes. Para ello, se intercambian información (a través de «conjuntos de afirmaciones», o *Claims Set*) codificada como objeto JSON. Estas *claims* son fiables y pueden verificarse porque están firmadas digitalmente; y también pueden cifrarse y descifrarse.

La información se codifica como objeto JSON, que se puede transmitir como parte de un *JSON Web Signature* (**JWS**) o como texto plano de un *JSON Web Encryption* (**JWE**); lo que permite tanto la firma como el cifrado y descifrado.

Los JWT pueden firmarse utilizando un **secreto** compartido (con el algoritmo HMAC) o un **par de claves públicas-privadas** (con RSA).

Se puede dar el caso de **firma y cifrado o descifrado a la vez**, lo que produce un **JWT anidado** (*nested JWT*). En estos casos, se pueden encontrar JWT dentro de otro JWT; JWE o JWE. Por ejemplo, un JWT dentro de un JWE, que a su vez está dentro de un JWS.

El formato JWT está pensado para entornos con **limitaciones de espacio**, como encabezados HTTP de autorización o parámetros de consulta en URIs; por eso se dice que es un **método compacto**. Además, debido a su tamaño, la **transmisión es rápida**.

Se representan utilizando **serializaciones** compactas específicas:

- *JWS Compact Serialization* para la firma.
- *JWE Compact Serialization* para cifrado y descifrado.

3.1. Afirmaciones JWT

Cada afirmación (*claim*) es una pieza de información que se confirma sobre algo o alguien específico. Se representa como un **par clave-valor**, consistente del **nombre de la afirmación** (una cadena de texto o *string*) y su valor (que puede ser cualquier valor JSON). A estos se les denomina *Claim Name* y *Claim Value*, respectivamente. Si alguna de las afirmaciones del JWT no tiene integridad o no está cifrada, entonces el JWT no es seguro.

Los conjuntos de afirmaciones JWT (*JWT CLaims Set*) representan un objeto JSON que contiene cada uno de los pares *Claim Name*/*Claim Value* del JWT. Los nombres de las afirmaciones deben ser únicos y, en caso de que haya duplicados, se puede decidir si rechazar esa afirmación o solamente escoger la que se ha definido en último lugar, lo cual depende del analizador utilizado.

Dependiendo del contexto, un conjunto de afirmaciones será considerado válido o no; por lo que queda fuera del alcance del estándar RFC 7519. Aun así, todas las afirmaciones que no se entiendan deben ser ignoradas.

Hay tres clases de nombres para las afirmaciones JWT:

- **Nombres registrados** (*Registered Claim Names*). Constan en el registro «JSON Web Toekn Claims» de la IANA (*Internet Assigned Numbers Authority*), establecido por el RFC 7519 en su sección 10.1 ([IETF, 2015b]). Cada aplicación puede decidir cuáles usar, y si algunas de estas serán obligatorias u opcionales.
  - «iss» (Issuer). Identifica al emisor del JWT. Suele ser específico para cada aplicación.
  - «sub» (Subject). Identifica el tema del JWT. Debe ser único para el emisor o de manera global.

- «aud» (Audience). Identifica los destinatarios del JWT.
- «exp» (Expiration Time). El tiempo de expiración tras el cual el JWT no debe ser procesado.
- «nbf» (Not Before). El tiempo que debe pasar para que un JWT pueda ser procesado.
- «iat» (Issued At). La fecha y hora en la que se emitió.
- «jti» (JWT ID). Identificador único para el JWT. Se debe asignar de tal forma que apenas haya probabilidades de que el mismo identificador se asigne a otro objeto. Se puede utilizar para evitar que se reutilicen JWT en un ataque por repetición.

- **Nombres públicos** (*Public Claim Names*). Se pueden definir nuevos nombres, aunque se recomienda registrarlos en el registro de la IANA. También se puede realizar de tal forma que el valor contenga un nombre resistente a colisiones. En cualquier caso, quien defina el nombre o el valor debe tomar las medidas pertinentes para asegurar el control del nombre de la afirmación.
- **Nombres privados** (*Private Claim Names*). Entre un productor y un consumidor de JWT se pueden acordar nombres de afirmaciones que no aparezcan en el registro de la IANA. Esto los hace más propensos a colisiones, por lo que se debe tener precaución.

Estos conjuntos de afirmaciones se incluyen en el cuerpo del JSON como *payload* o carga.

3.2. Cabeceras JOSE

Los JWT tienen una cabecera denominada *JavaScript Object Signing and Encryption* (JOSE), que contiene información sobre el tipo de *token* y los algoritmos criptográficos utilizados para proteger las afirmaciones, entre otras propiedades. En Cabecera JOSE, `typ` indica que el tipo de token es un JWT, y `alg` especifica que el algoritmo utilizado para firmar el token es HMAC SHA-256].

```
{
  "typ": "JWT",
  "alg": "HS256"
}
```

Listing 1: Cabecera JOSE

La cabecera JOSE describe las operaciones que se han aplicado al conjunto de afirmaciones del JWT. Aquí es donde se distingue entre JWS (si el JWT está firmado) y JWE (si el JWT está cifrado).

Cada JWT se representa como una secuencia de partes URL seguras separadas por puntos («.»). Cada parte está codificada como una URL en base 64, y el número de partes depende de la serialización compacta utilizada (de JWS o JWE).

Algunos de los parámetros de la cabecera son:

- «typ» (Type). Define el tipo de objeto (*media type*) del JWT, ya que no siempre hay únicamente objetos JWT. Si se utiliza, se recomienda que su valor sea «JWT» (en mayúsculas) para mejorar la compatibilidad con distintas implementaciones.
- «cty» (Content Type). Se **debe utilizar cuando hay JWT anidados** para dar información de su estructura, en cuyo caso su valor debe ser «JWT» para indicar que hay un JWT anidado. Se recomienda que su valor sea «JWT» (en mayúsculas) para mejorar la compatibilidad con distintas implementaciones.

No se recomienda añadir «cty» si no hay anidación de JWT.

En algunas aplicaciones, puede ser útil tener una representación sin cifrar de algunas afirmaciones para, por ejemplo, procesamiento de reglas antes de descifrar los JWT. En estos casos, se pueden replicar algunos parámetros en la cabecera. Si se repiten afirmaciones, **se debe comprobar** que los valores de los campos replicados sean idénticos.

3.3. Creación y validación de JWT

Los JWT se convierten a base 64 para su transmisión. Cada una de las partes que lo componen se separan por puntos («.»). En el ejemplo de la figura 3, el JWT tiene cabecera, el conjunto de afirmaciones (*payload*) y firma; cada una de las partes en base 64 y separadas por puntos («.»).

Figura 3: Estructura de un JWT. [Kintali, 2024]

3.3.1. Pasos para la creación de un JWT

1. Crear un conjunto de afirmaciones JWT con las afirmaciones deseadas, y representarlo de la forma adecuada.
2. Crear una cabecera JOSE con los parámetros deseados.
3. Crear el mensaje JWT acorde a la especificación JWS o JWE, dependiendo del caso específico.
4. Si es un JWT anidado, se ha de incluir el «cty» (Content Type) en la cabecera JOSE.

3.3.2. Pasos para la validación de un JWT

Si cualquiera de los pasos falla, el JWT **debe rechazarse**, es decir, invalidarlo.

1. Verificar que el JWT contiene al menos un caracter de punto («.»). La parte que precede al primer punto («.») es la cabecera JOSE.
2. Decodificar la cabecera JOSE (URL base 64).
3. Verificar que la cabecera JOSE es un objeto JSON válido.
4. Verificar que la cabecera JOSE solo contiene parámetros y valores cuya sintaxis y semántica sean entendibles y estén soportadas, o que se haya indicado que se deban ignorar cuando no se entiendan.
5. Determinar si el JWT es JWS o JWE.
   - Si es un JWS, se el mensaje será resultado de URL base 64.
   - Si es JWE, el mensaje será el texto plano.

6. Si la cabecera contiene «cty» (Content Type), entonces el JWT es anidado y hay que repetir los pasos para procesar el JWT contenido.

7. Decodificar el mensaje del JWT.

8. Verificar que el mensaje del JWT es un objeto JSON válido. Este mensaje será el conjunto de afirmaciones JWT.

Dependiendo de la aplicación y de los acuerdos que se hayan establecido, algunos valores de las afirmaciones serán válidos o no. Por ejemplo, un JWT podría validarse, pero si incluye un algoritmo no soportado por la aplicación, debería rechazarse.

3.3.3. Ejemplo de JWT

En la página web de JWT (https://jwt.io/), se pueden codificar y decodificar *tokens* de manera fácil y sencilla ([JWT.io,]).

Introduciendo la cabecera (1), el conjunto de afirmaciones (2) y una firma (3), se puede generar el JWT codificado: 4 (se han añadido saltos de línea para su correcta visualización). Se puede elegir el algoritmo (por defecto «HS256»).

```
{
  "sub": "1234567890",
  "name": "John Doe",
  "iat": 1516239022
}
```

Listing 2: Conjunto de afirmaciones (payload)

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  clave
)
```

Listing 3: Firma

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9
.
eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6I
kpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ
.
rH9lhA8v9DDPaJtatHwzrn13FN9HNv3aUDWEoWZYJRU
```

Listing 4: JWT codificado en base 64

También permite pegar un JWT codificado. De esta forma, se pueden recuperar automáticamente la cabecera, el conjunto de afirmaciones y la firma que se han utilizado para el JWT. Se puede introducir la clave en la firma (**??**) para verificar si es válida o no.

Sobra mencionar que, si el JWT contiene información sensible, debería ser procesado con más cautela.

3.4. Requisitos de implementación

Las aplicaciones deben implementar algunos algoritmos y características de forma obligatoria, y añadir opcionalmente otros requisitos en función de sus necesidades.

A continuación se nombrarán algunos algoritmos comúnmente utilizados en el ámbito de la ciberseguridad y la criptografía. Algunos de estos algoritmos se han estudiado en la optativa de *Seguridad en Redes Telemáticas*, por lo que se ha considerado interesante su mención; aunque no se explicará ninguno de ellos, ya que queda fuera del alcance de la asignatura actual de *Arquitecturas Orientadas a Servicios*.

De los algoritmos MAC y de firma especificados en la JWA (*JSON Web Algorithms*, RFC 7518, [IETF, 2015a]), **se deben implementar «HS256» (HMAC SHA-256) y «none»**.

Sin embargo, se recomienda que también se añadan otros, como «RS256» (RSASSA-PKCS1-v1_5 SH6A-25) y «ES256» (ECDSA P-256 curve SHA-256). Los demás algoritmos y los distintos tamaños de clave son opcionales.

El **cifrado de JWT es opcional**. No obstante, si se implementa, se deben implementar:

- «RSA1_5» (RSAES-PKCS1-v1_5 with 2048-bit keys)
- «A128KW» y «A256KW» (AES Key Wrap with 128-bit and 256-bit keys)
- «A128CBC-HS256» y «A256CBC-HS512» (AES-CBC and HMAC SHA-2)

Se recomienda dar soporte a «ECDH-ES» (Elliptic Curve Diffie-Hellamn Ephemeral Static), entre otros algoritmos; pero es completamente opcional, así como no es necesario incluir otros tamaños de clave.

Tampoco es necesario soportar JWT anidados.

3.4.1. JWT inseguros

A veces, los JWT se pueden crear sin que se firmen o se cifren: es un JWT con el parámetro «alg» como «none» y con cadenas vacías en los valores de los algoritmos de firma o cifrado.

```
{"alg":"none"}
```

Listing 5: JWT inseguro

Cuando se trabaja con JWT inseguros se suele deber a que se toman medidas que van más allá del propio JWT para garantizar su integridad y seguridad.

3.5. Algunas consideraciones

Un JWT por sí solo no es de confianza, sino que se debe verificar que las claves de firma o cifrado del JWT han sido emitidas de una fuente confianza, y que son íntegras.

Para los JWT anidados no hay un orden establecido, pero lo habitual es primero firmar y luego cifrar el mensaje firmado. Esto aumenta la privacidad del firmante y puede evitar algunos ataques, al no dejar la firma al descubierto.

3.6. Flujo de funcionamiento de JWT en una aplicación

1. Inicio de sesión del usuario a través de credenciales (u otro método de autenticación).
2. La aplicación verifica las credenciales de inicio de sesión. Si son válidas, se genera el JWT.
3. Envío del JWT al usuario, quien lo almacena localmente.
4. Verificación del *token*. Cuando el usuario realiza peticiones para acceder al recurso, el servidor verificará la firma del JWT y las afirmaciones (*claims*) para comprobar que el usuario tiene los permisos necesarios para acceder al recurso que solicita.

5. Si el JWT es válido, se concederá el acceso al recurso solicitado.

Un resumen del funcionamiento se puede ver en la figura 4.

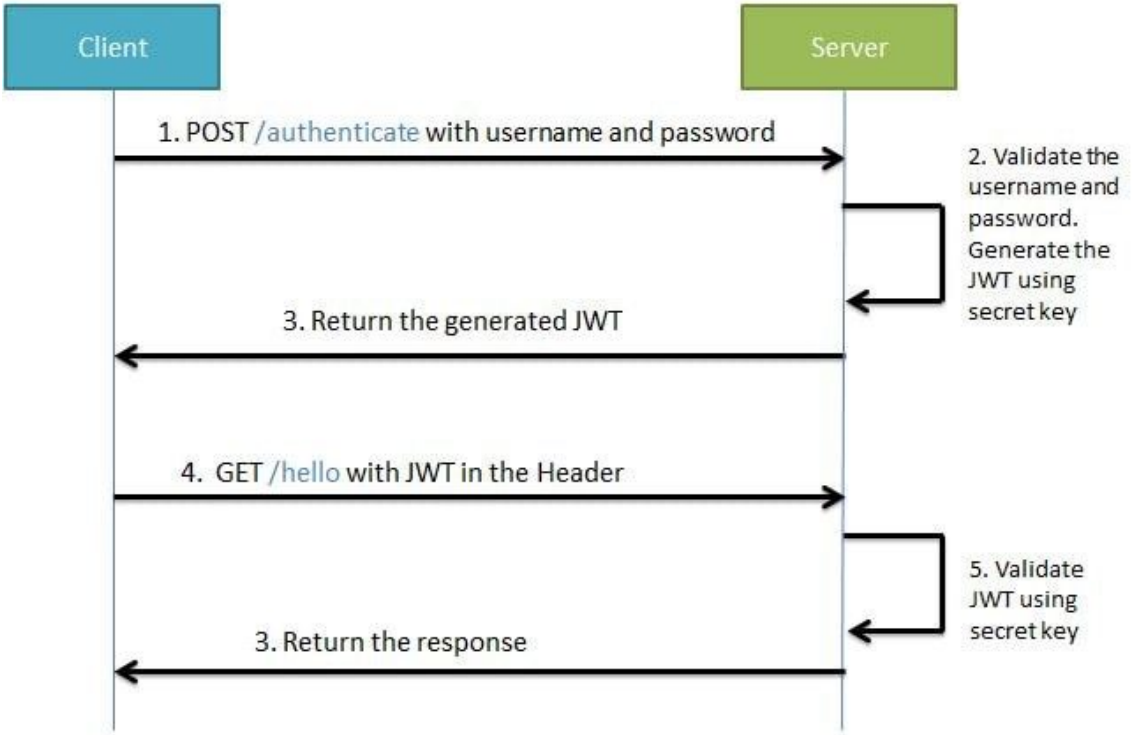

Figura 4: Flujo de JWT. [Fadatare, 2024]

3.7. Beneficios de JWT

Tras iniciar sesión, los usuarios pueden recibir un JWT con el que podrán acceder a los recursos de la aplicación. De esta manera, se reduce el número de veces que tienen que introducir sus credenciales. Además, como el servidor emite los JWT y los usuarios los guardan, no supone un incremento de espacio en las bases de datos.

- Los JWT son **muy ligeros** debido a su formato como objeto JSON. Debido a esto, son muy **compactos** e ideales para entornos con limitaciones de espacio.
- Los JWT se pueden firmar tanto simétrica como asimétricamente, por lo que ofrecen una **seguridad robusta**.
- Se puede establecer un tiempo de expiración.
- Es fácil trabajar con JWT, ya que suelen implementarse en soluciones de SSO (inicio de sesión único).

4. Diferencias entre JWT y OAuth

OAuth es un **protocolo de autorización** definido en la RFC 6749, mientras que JWT es un **formato de *token*** definido en la RFC 7519. Debido a esto, OAuth se utiliza en varias capas (web, navegador, API y aplicaciones) y JWT se centra principalmente en las API ([Geekflare,]).

OAuth es más complejo que los JWT, que son más fáciles de comprender, aunque tienen un alcance más limitado que OAuth. Cuando se requiere de más flexibilidad, se recomienda recurrir a OAuth por tener una gama más amplia de casos de uso, por lo que se puede adaptar mejor a diversos escenarios.

Además, OAuth puede permitir que una aplicación de terceros acceda a los datos de sus usuarios en otra plataforma sin revelar sus datos de acceso, lo que puede resultar muy útil para, por ejemplo, iniciar sesión.

Como JWT es un formato de *token*, se pueden implementar los JWT en OAuth, atendiendo a los Requisitos de implementación mencionados anteriormente.

Es común que en un mismo servidor se combinen OAuth y JWT, ya que se pueden emitir dos *tokens* (uno de JWT y otro de acceso a OAuth) para tener más control sobre el acceso y los datos del usuario.

Depende de de la aplicación específica utilizar OAuth o JWT (o ambos a la vez). Para proyectos más complejos, lo ideal sería utilizar OAuth; pero puede ser demasiado para otros más sencillos, en los que JWT sería mejor. Aun así, se debe considerar la posibilidad de utilizar los dos combinados para una mayor seguridad.

5. Conclusiones

OAuth 2.0 es un **protocolo de autorización** que separa las credenciales del propietario del recurso de las del cliente a la hora de solicitar recursos de un servidor. Esta filosofía aborda los problemas de seguridad y limitaciones relacionadas con la implementación tradicional mediante cliente-servidor.

Los roles clave del protocolo incluyen el propietario del recurso, el cliente, el servidor de autorización y el servidor de recursos. Su funcionamiento sigue una secuencia estructurada donde el cliente obtiene **tokens de acceso** mediante la interacción con el servidor de autorización y el propietario del recurso. Estos pueden poseer una fecha de expiración, renovable mediante el uso de **tokens de actualización**.

Se describen diversos métodos de concesión de autorización, adaptados a la **confianza** que posea el proveedor de los recursos con el cliente que los solicita, así como el **tipo de aplicación** en el que se utiliza el protocolo.

OAuth 2.0, gracias a estas características, destaca por su **flexibilidad**, **escalabilidad** y **seguridad**, siendo fundamental en escenarios donde se requiere acceso controlado a recursos protegidos.

Los JWT (*JSON Web Tokens*) son objetos JSON que se utilizan en el protocolo OAuth debido a su ligereza, portabilidad y seguridad, centrados en las API. Constan de tres partes fundamentales: la cabecera JOSE, el cuerpo (*payload*) y la firma, aunque esta última es prescindible. En el cuerpo del JWT, se encuentran un conjunto de afirmaciones como pares clave-valor que se utilizan para intercambiar y verificar información.

Para seguir ahondando en el funcionamiento de OAuth, a continuación se realizará un ejemplo práctico que utiliza este protocolo utilizando JWT: Demo de OAuth 2.0 paso a paso.

Referencias

[Auth0, 2024] Auth0 (2024). ¿qué es oauth 2.0? https://auth0.com/es/intro-to-iam/what-is-oauth-2. Último acceso: 19 de octubre de 2024.

[Auth0, sfa] Auth0 (s.f.a). Comparación entre saml y oauth. https://auth0.com/es/intro-to-iam/saml-vs-oauth. Último acceso: 19 de octubre de 2024.

[Auth0, sfb] Auth0 (s.f.b). What's the difference between oauth, openid connect, and saml? https://www.okta.com/identity-101/whats-the-difference-between-oauth-openid-connect-and-saml/. Último acceso: 19 de octubre de 2024.

[Fadatare, 2024] Fadatare, R. (2024). What is jwt (json web token) and how it works? https://www.javaguides.net/2021/03/what-is-jwt-json-web-token-and-how-it.html. Último acceso: 17 de noviembre de 2024.

[Geekflare,] Geekflare. Diferencias entre oauth y jwt. https://geekflare.com/es/jwt-vs-oauth/. Último acceso: 17 de noviembre de 2024.

[Hardt, 2012] Hardt, D. (2012). The oauth 2.0 authorization framework (rfc 6749). https://www.rfc-editor.org/rfc/rfc6749.html. Último acceso: 19 de octubre de 2024.

[IETF, 2012] IETF (2012). Oauth 2.0 authorization framework. https://datatracker.ietf.org/doc/html/rfc6749. Último acceso: 17 de noviembre de 2024.

[IETF, 2015a] IETF (2015a). Json web algorithms (jwa). https://www.rfc-editor.org/rfc/rfc7518. Último acceso: 17 de noviembre de 2024.

[IETF, 2015b] IETF (2015b). Json web token (jwt). https://datatracker.ietf.org/doc/html/rfc7519. Último acceso: 17 de noviembre de 2024.

[JWT.io,] JWT.io. Json web token debugger y generador. https://jwt.io/. Último acceso: 17 de noviembre de 2024.

[Kintali, 2024] Kintali, K. (2024). What is jwt? why it is used for? https://medium.com/@koushikkintali/what-is-jwt-why-it-is-used-for-15e267b70f9b. Último acceso: 17 de noviembre de 2024.

[OAuth, 2024] OAuth (2024). Differences between oauth 1.0 and oauth 2.0. https://www.oauth.com/oauth2-servers/differences-between-oauth-1-2/. Último acceso: 19 de octubre de 2024.

6. Demo de OAuth 2.0 paso a paso

En esta práctica, se realizarán tres proyectos en **Spring Boot**, el *framework* que se ha aprendido en la asignatura; para demostrar el flujo de OAuth 2.0 paso a paso. Para ello, se proporcionan los materiales necesarios en el siguiente repositorio de GitHub: https://github.com/saguit03/oauth-demo.

Estos proyectos son acordes a lo explicado durante la parte teórica:

- Servidor de autorización (`autorizacion`): gestiona la autorización de los usuarios.
- Servidor de recursos (`recursos`): gestiona los permisos de los recursos a los que se accede.
- Cliente (`cliente`): solicita los permisos para acceder los recursos.

La práctica se desarrollará en tres partes, cada una de ellas dedicada a un proyecto. En cada una de ellas, se explicará cómo configurar el proyecto y se mostrará el código necesario para realizar la demostración. El cliente se creará al final, por lo que, hasta entonces, se utilizará una página web que hará la función de cliente: https://oauthdebugger.com/.

Para la realización de la práctica, se utilizó como referencia el siguiente vídeo: https://youtu.be/oHiIBkSv3nw.

Si no se desea implementar todo el código, también se puede ejecutar directamente utilizando el código resuelto. Para ello, se pueden leer las instrucciones de los siguientes apartados:

- Flujo de autorización
- Flujo del servidor de recursos
- Flujo del cliente

Un resumen de todas las peticiones utilizadas durante estos flujos se encuentra en el Anexo. Resumen peticiones en Postman.

Si se opta por implementar todo el código, conviene leer con atención todos los apartados. El código se explica paso a paso en cada uno de los siguientes puntos:

- Guía de implementación del servidor de autorización
- Guía de implementación del servidor de recursos
- Guía de implementación del cliente

Dado que en Java los nombres de los paquetes son muy extensos, se han suprimido de los ejemplos anteriores para facilitar la lectura. Se incluyen todos los paquetes omitidos en el Anexo. Paquetes de los proyectos.

6.1. Preparación previa. Creación de los proyectos

Se utilizarán Visual Studio Code y Postman para la implementación y pruebas.

Se puede descargar el repositorio con los materiales base y resueltos de la práctica desde el repositorio previamente mencionado (https://github.com/saguit03/oauth-demo). Este contiene cuatro carpetas:

- **autorizacion**: proyecto resuelto del servidor de autorización.
- **recursos**: proyecto resuelto del servidor de recursos.

- **cliente**: proyecto resuelto del cliente.
- **Base**: proyectos descargados directamente desde Spring Boot Initializr con la configuración necesaria.

Otra opción sería acceder a *Spring Boot Initializr*, donde configurar cada uno de los proyectos y descargarlos. Dicha página se encuentra en el siguiente enlace: `https://start.spring.io`.

En cualquier caso, las dependencias necesarias para la práctica se pueden consultar en el Anexo. Dependencias del pom.xml.

La configuración de los proyectos es la siguiente:

- **Project**: Maven
- **Language**: Java
- **Spring Boot**: 3.4.0 (por defecto, la más reciente)
- **Group**: es.unex.aos
- **Artifact**: `autorizacion`, `recursos` o `cliente`, según el proyecto
- **Description**: Cualquiera
- **Packaging**: Jar
- **Java**: 17
- **Dependencies**:
  - **Spring Web**
  - **Spring Boot DevTools**
  - **Spring Security**
  - **OAuth Authorization Server** (solo para el servidor de autorización)
  - **OAuth2 Resource Server** (solo para el servidor de recursos)
  - **OAuth2 Client** (solo para el cliente)

Como se ve en la figura 5, se ha configurado un proyecto con las dependencias para los tres proyectos, puramente para la demostración del ejemplo. Durante la práctica, se deben elegir únicamente las dependencias que se vayan a utilizar, cambiando también el nombre del *Artifact* para que sea acorde a cada proyecto (autorización, recursos o cliente).

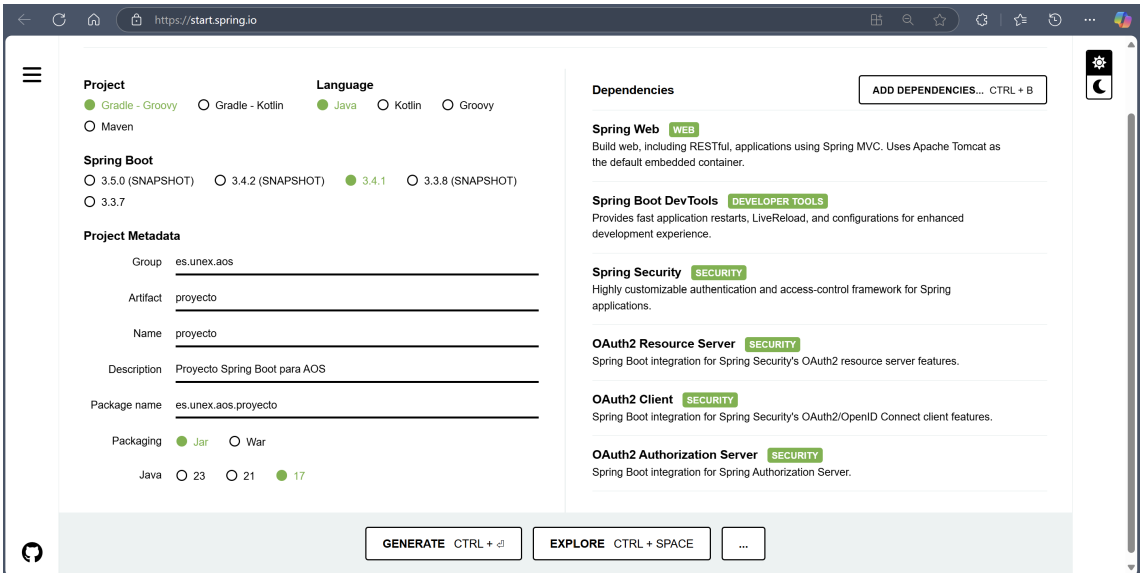

Figura 5: Configuración de un proyecto en Spring Boot Initializr.

6.2. Servidor de autorización

6.2.1. Guía de implementación del servidor de autorización

Antes de nada, se configurará el `application.properties` del proyecto de la siguiente manera:

```
spring.application.name=autorizacion
server.port=9000
logging.level.org.springframework.security=DEBUG
```

Listing 6: application.properties del servidor de autorización.

Desde la documentación oficial, se puede obtener la clase `SecurityConfig.java`, que se encargará de la configuración de la seguridad del servidor de autorización. Esta clase se podrá definir en cualquier paquete del proyecto, siempre y cuando se anote correctamente:

```
package es.unex.aos.autorizacion.config; // Por ejemplo
// Dependencias (incluidas todas en un anexo)
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    // Código
}
```

Listing 7: Clase `SecurityConfig.java` del servidor de autorización.

Se puede copiar directamente desde la documentación, aunque hay que eliminar los números adyacentes a los *Bean* (en la página sirven para ver explicaciones del código, pero produce fallos porque no forman parte de la sintaxis de Java): https://docs.spring.io/spring-authorization-server/reference/getting-started.html#defining-required-components.

Se modificarán algunos de los métodos proporcionados, mientras que los demás se quedarán tal y como están. Aquellos que no se modifiquen no se explicarán, ya que no se consideran importantes para el buen entendimiento de la práctica en el ámbito de la asignatura de AOS. No obstante, se anima al lector a leer el resto de la documentación si se desea profundizar más en el tema; especialmente si siente inquietud por cuestiones de seguridad más específicas.

Dado que es un servidor de juguete para una demostración básica, se va a deshabilitar la protección CSRF, ya que no se va a implementar un formulario de inicio de sesión personalizado. Para ello, se añadirá la línea `.csrf(csrf ->csrf.disable()` en el método `defaultSecurityFilterChain` (véase el código 8).

Por lo demás, la función de este filtro es autorizar peticiones HTTP a través de un formulario de inicio de sesión por defecto.

```
@Bean
@Order(2)
public SecurityFilterChain defaultSecurityFilterChain(HttpSecurity
    http) throws Exception {
     http.authorizeHttpRequests((authorize) -> authorize
          .anyRequest().authenticated())
          .csrf(csrf -> yellowcsrf.disable())
          .formLogin(Customizer.withDefaults());
     return http.build();
}
```

Listing 8: Filtro de seguridad del servidor de autorización.

En esta demo, se pueden mantener las credenciales del usuario de forma estática en el código. NUNCA debería realizarse esta práctica en un entorno de producción real.

Un usuario se crea a partir de su nombre de usuario, su contraseña y sus roles. En este caso, se ha creado un usuario con nombre `aos`, contraseña `aos` y rol `USER` (véase el código 9). Es importante añadir en la contraseña el prefijo `{noop}`, ya que indica que la contraseña no está codificada.

Al final del método, se devuelve el usuario recién construido y se almacena en memoria.

```
@Bean
public UserDetailsService userDetailsService() {
    UserDetails userDetails = User.builder()
        .username("aos")
        .password("{noop}aos")
        .roles("USER")
        .build();

    return new InMemoryUserDetailsManager(userDetails);
}
```

Listing 9: Detalles de un nuevo usuario en el servidor de autorización.

Por último, se debe configurar el cliente registrado en el servidor de autorización. En este caso, se ha creado un cliente con ID `oidc-client` y contraseña `secreto`, que se utilizarán para la autenticación durante las peticiones en Postman. También se añade el método de autenticación, en este caso `CLIENT_SECRET_BASIC`.

Como se ve en el código 10, se permiten dos tipos de autorización: `AUTHORIZATION_CODE` y `REFRESH_TOKEN`. Además, se ha añadido un *scope* de tipo `OPENID` y `PROFILE`, que se utilizarán en las peticiones del cliente *OAuth Debugger* durante el Flujo de autorización.

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
    RegisteredClient oidcClient =
   RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("oidc-client")
        .clientSecret("{noop}secreto")

   .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)

   .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)

   .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
        .redirectUri("https://oauthdebugger.com/debug")
        .scope(OidcScopes.OPENID)
        .scope(OidcScopes.PROFILE)
        .build();
    return new InMemoryRegisteredClientRepository(oidcClient);
}
```

Listing 10: Datos del cliente registrado en el servidor de autorización.

Ya debería estar todo preparado. Si se ha generado el proyecto con Spring Boot Initializr, la clase principal ya estaría creada por defecto. Si no, se debe crear una clase principal con la anotación `@SpringBootApplication` y ejecutarla.

6.2.2. Flujo de autorización

Tras ejecutar el servidor de autorización (en el directorio raíz del proyecto, en la terminal con el comando: `mvn spring-boot:run`), se accede a la siguiente página, que hará la función de cliente:

`https://oauthdebugger.com/`

Se deben introducir los siguientes datos (véase la parte izquierda de la figura 6):

- **Authorize URI**: `http://localhost:9000/oauth2/authorize`
- **Redirect URI**: `https://oauthdebugger.com/debug`
- **Client ID**: oidc-client
- **Scope**: profile
- **Response type**: code

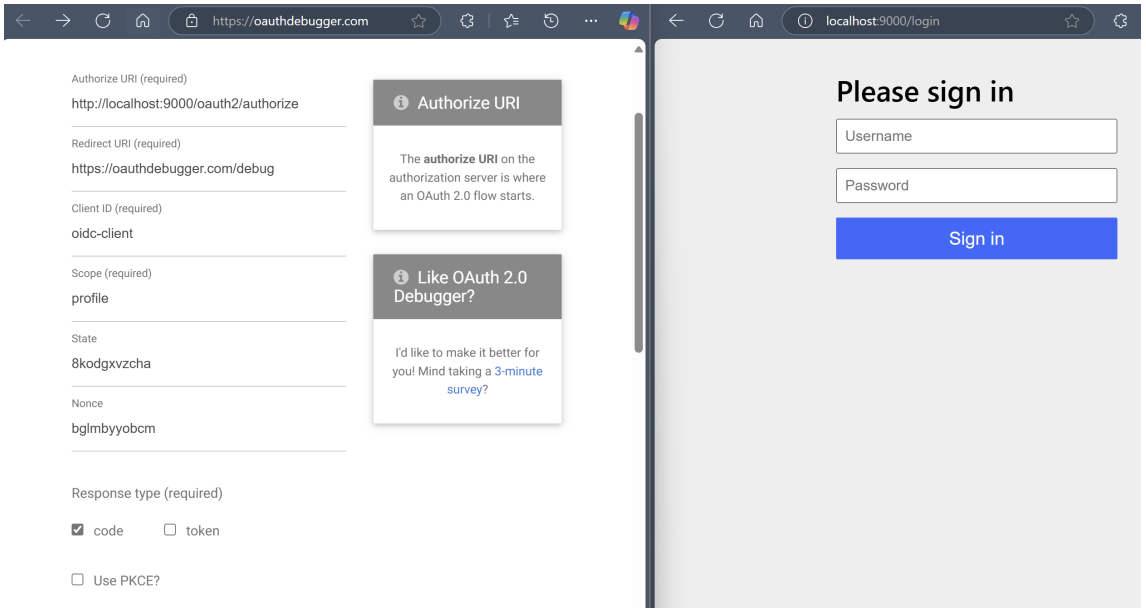

Figura 6: OAuth Debugger y servidor de autorización.

El tipo de respuesta *code* se utiliza en el flujo de autorización de OAuth 2.0. En este flujo, el cliente solicita autorización al servidor de autorización para acceder a los recursos protegidos. El servidor de autorización responderá con un código, que el cliente podrá intercambiar por *tokens* en un canal seguro. Este flujo se debe utilizar cuando el código de la aplicación se ejecuta en un servidor seguro (común para aplicaciones de páginas MVC y renderizadas en el servidor).

Si es necesario, se desplazará la página hacia abajo para enviar la petición. En este momento, si es la primera vez que se accede al servidor de autorización, solicitará las credenciales de usuario establecidas anteriormente (véase la parte derecha de la figura 6).

Si todo es correcto, se redirigirá a la página de *OAuth Debugger* con un código en la URL, como se ve en la figura 7. Este código es el *token* que se utilizará para obtener el *token* de acceso.

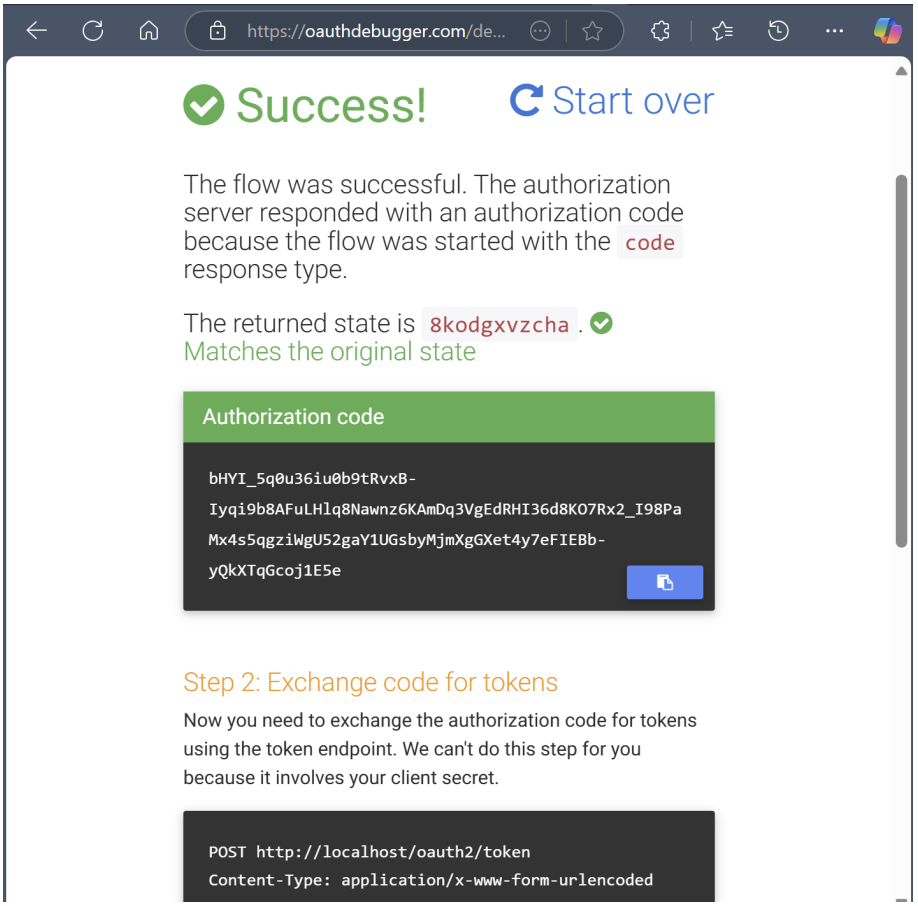

Figura 7: Respuesta en OAuth Debugger del servidor de autorización.

Se debe copiar el código devuelto por el servidor de autorización. A continuación, desde Postman se realizá una petición POST a `http://localhost:9000/oauth2/token` con los siguientes datos:

- **URL**: `http://localhost:9000/oauth2/token`
- **Tipo de petición**: POST
- **Authorization** (véase figura 8):
  - **Type**: Basic Auth (porque antes se ha configurado el cliente del servidor con `CLIENT_SECRET_BASIC`)
  - **Username**: oidc-client
  - **Password**: oidc-secret
- **Body** (véase figura 9):
  - **code**: código devuelto por el debugger
  - **grant_type**: authorization_code
  - **redirect_uri**: `https://oauthdebugger.com/debug`

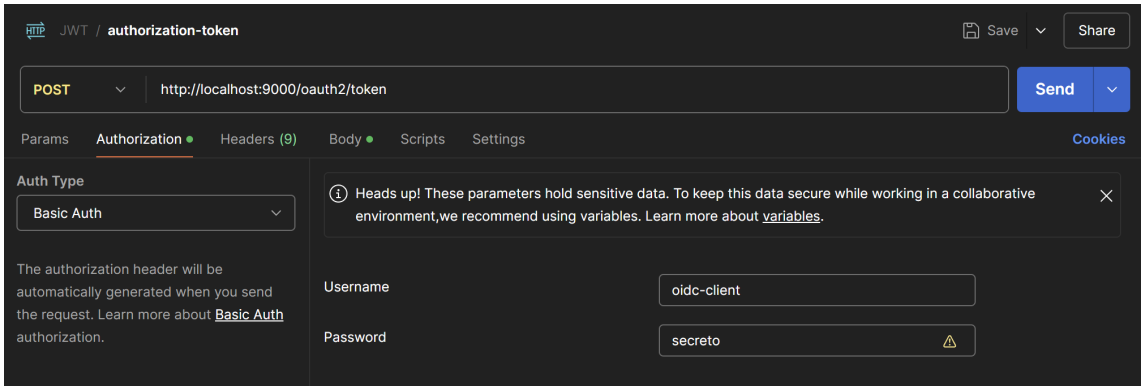

Figura 8: Petición POST a `http://localhost:9000/oauth2/token` con Postman: `Authorization`

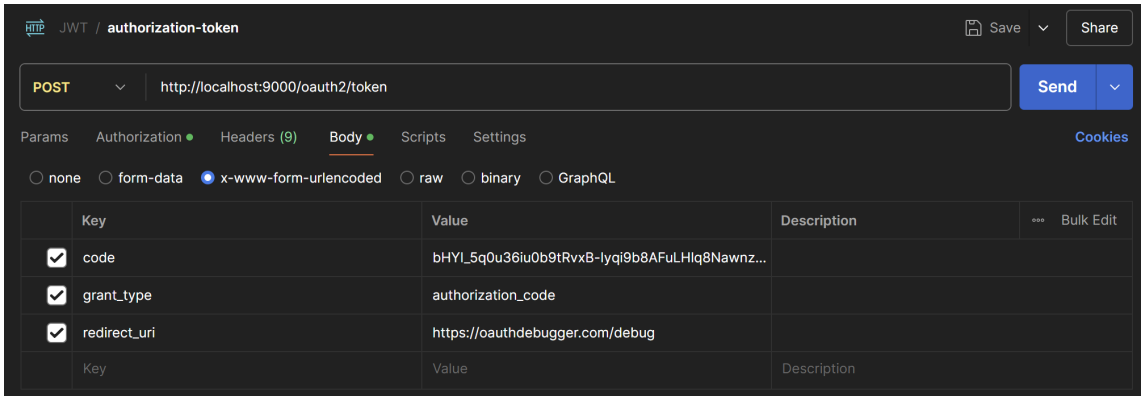

Figura 9: Petición POST a `http://localhost:9000/oauth2/token` con Postman: `Body`

Si todo va bien, se obtendrá un *token* de acceso, como se ve en la figura 10. Este *token* se podría utilizar para acceder a los recursos protegidos del servidor de recursos, pero todavía no tenemos implementado este servicio.

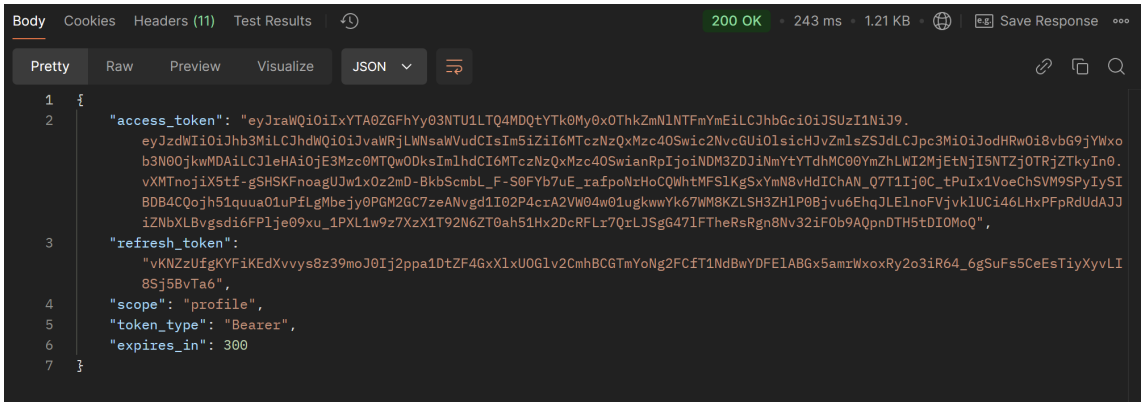

Figura 10: Respuesta de la petición POST a `http://localhost:9000/oauth2/token`

Para un mayor entendimiento, se puede comprobar el *token* en `https://jwt.io/`, como se ve en la figura 11.

La cabecera y la firma del *token* se han generado por defecto. Lo más interesante es el cuerpo del *token*, donde se puede ver la información del usuario autenticado y los permisos que tiene.

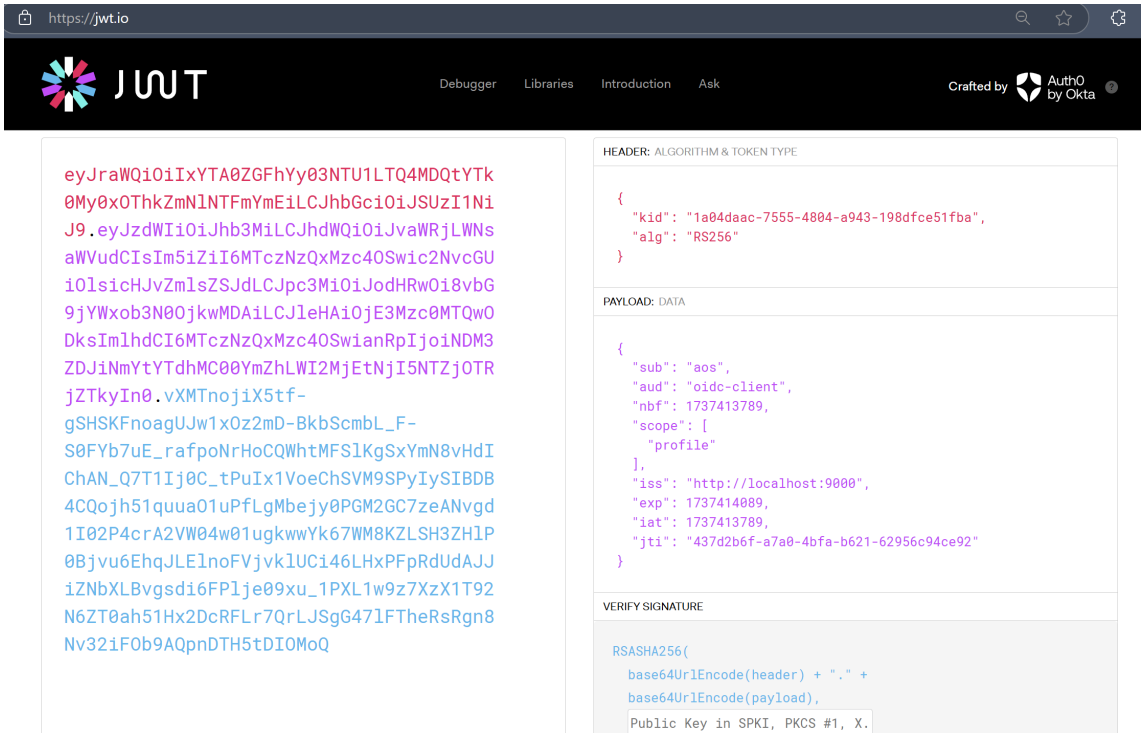

Figura 11: Comprobación del *token* en jwt.io.

En este caso, estos datos se han introducido manualmente en el primer paso del flujo (véase la figura 6). Asimismo, se ven otros como el `iss` (emisor), `exp` (fecha de expiración), `iat` (hora de emisión) y `jti` (identificador único del *token*).

6.3. Servidor de recursos

6.3.1. Guía de implementación del servidor de recursos

El `application.properties` del servidor de recursos luce así:

```
spring.application.name=recursos
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:9000
logging.level.org.springframework.security=DEBUG
server.port=8081
```

Listing 11: application.properties del servidor de recursos.

Desde la documentación oficial, se puede copiar el `application.yml` para complementar la configuración anterior:

https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/jwt.html

```
spring:
    security:
        oauth2:
            resourceserver:
                jwt:
                    jws-algorithms: RS512
                    jwk-set-uri: http://localhost:9000
```

Listing 12: application.yml del servidor de recursos.

Se podría utilizar únicamente uno de los dos ficheros de configuración (ya sea `application.properties` o `application.yml`), pero se han mostrado ambos para que se vea que se pueden utilizar ambos formatos.

A continuación, se crea la clase `SecurityConfig.java` en el paquete deseado, que se encargará de la configuración de la seguridad del servidor de recursos. Esta clase se podrá definir en cualquier paquete del proyecto, siempre y cuando se anote correctamente:

```
package es.unex.aos.recursos.config; // Por ejemplo
// Dependencias (incluidas todas en un anexo)
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    final static String issuerUri = "http://localhost:9000";

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http)
   throws Exception {
        http.authorizeHttpRequests(authorize -> authorize
            .requestMatchers(HttpMethod.GET, "/resources/**")
            .hasAnyAuthority("SCOPE_read", "SCOPE_write")
            .requestMatchers(HttpMethod.POST, "/resources/**")
            .hasAuthority("SCOPE_write")
            .anyRequest().authenticated())
            .oauth2ResourceServer((oauth2) ->
   oauth2.jwt(Customizer.withDefaults()));
        return http.build();
    }
```

```
    @Bean
    public JwtDecoder jwtDecoder() {
        NimbusJwtDecoder jwtDecoder = (NimbusJwtDecoder)
    JwtDecoders.fromIssuerLocation(issuerUri);

        OAuth2TokenValidator<Jwt> withClockSkew = new
    DelegatingOAuth2TokenValidator<>(
            new JwtTimestampValidator(Duration.ofSeconds(60)),
            new JwtIssuerValidator(issuerUri));

        jwtDecoder.setJwtValidator(withClockSkew);

        return jwtDecoder;
    }
}
```

Listing 13: Clase `SecurityConfig.java` del servidor de recursos.

Dentro de esta clase, se debe definir el filtro de seguridad, `filterChain`, que se encargará de autorizar las peticiones HTTP. En este caso, se ha definido un filtro que permite las peticiones GET a `/resources/**` con los *scopes* `read` y `write`, y las peticiones POST con el *scope* `write`. Además, se ha añadido un filtro de seguridad para validar el *token* JWT, `jwtDecoder`, que se encargará de validar el *token* JWT con el emisor y la hora de emisión.

- **requestMatchers**: se encarga de definir las peticiones que se van a autorizar. En este caso, se han definido dos tipos de peticiones: GET y POST.
- **hasAnyAuthority** y **hasAuthority**: se encargan de definir los *scopes* necesarios para acceder a las peticiones GET y POST, respectivamente.
  - **GET**: se ha definido para las peticiones GET a `/resources/**` con los *scopes* `read` y `write`. Se define con **hasAnyAuthority**, ya que se puede tener cualquiera de los dos *scopes* para acceder a la petición.
  - **POST**: se ha definido para las peticiones POST a `/resources/**` con el *scope* `write`. Se define con **hasAuthority**, ya que solamente permite a este *scope* acceder a la petición.
- **anyRequest().authenticated()**: se encarga de definir que cualquier otra petición que no sea GET o POST debe estar autenticada.
- **oauth2ResourceServer**: se encarga de definir el servidor de recursos OAuth 2.0.
- **jwt**: se encarga de definir el decodificador de *tokens* JWT.

Ahora bien, para poder acceder a los recursos se deben definir los *endpoints* para las peticiones GET y POST. Para ello, se crea la clase `ResourceController.java` en el paquete correspondiente. En esta clase, se definen dos métodos, uno para las peticiones GET y otro para las peticiones POST, como se ve en el código 14.

```
@RestController
@RequestMapping("/resources")
public class ResourceController {

    @GetMapping("/user")
    public ResponseEntity<String> read_user(Authentication
    authentication) {
        return ResponseEntity.ok("The user can read." +
    authentication.getName() + authentication.getAuthorities());
```

```
    }

    @PostMapping("/user")
    public ResponseEntity<String> write_user(Authentication
   authentication) {
        return ResponseEntity.ok("The user can write." +
   authentication.getName() + authentication.getAuthorities());
    }
}
```

Listing 14: Clase `ResourceController.java` del servidor de recursos.

Si se desean copiar los ejemplos durante el flujo, es importante que no se cambien las rutas de los *endpoints*. Si se cambiasen, el lector debe tener la precaución de adaptar las direcciones utilizadas para que el ejemplo funcione correctamente.

Este controlador es muy simple y únicamente devuelve mensajes con el nombre del usuario autenticado y los permisos que tiene. En un entorno más realista, debería conceder acceso a los recursos que se solicitan.

Para que se puedan otorgar los permisos de escritura o lectura, se debe modificar el cliente registrado en el servidor de autorización. Para ello, se deben añadir los *scopes* de `read` y `write` al método `registeredClientRepository`, como se ve en el código 15.

```
@Bean
public RegisteredClientRepository registeredClientRepository() {
    RegisteredClient oidcClient =
   RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("oidc-client")
        .clientSecret("{noop}secreto")

   .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)

   .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)

   .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)
        .redirectUri("https://oauthdebugger.com/debug")
        .scope(OidcScopes.OPENID)
        .scope(OidcScopes.PROFILE)
        yellow.scope('read') // sustituir ' por "
        yellow.scope('write') // sustituir ' por "
        .build();
    return new InMemoryRegisteredClientRepository(oidcClient);
}
```

Listing 15: Modificación del cliente registrado en el servidor de autorización.

Sin más dilación, con estas dos clases creadas y la configuración previamente enseñada, se puede ejecutar el servidor de recursos. Si hubiera algún fallo, convendría asegurarse de que existe una clase principal que ejecute la aplicación anotada con `@SpringBootApplication` (generada automáticamente con Spring Boot Initializr).

6.3.2. Flujo del servidor de recursos

Una vez desplegado el servidor de recursos (`mvn spring-boot:run`), y mientras también está el servidor de autorización en ejecución, se pueden repetir todos los pasos explicados en Flujo

de autorización, pero cambiando el *scope* a `write` o `read`. Dependiendo de este parámetro, se tendrá permisos para algunas acciones u otras. Se verán ambos casos.

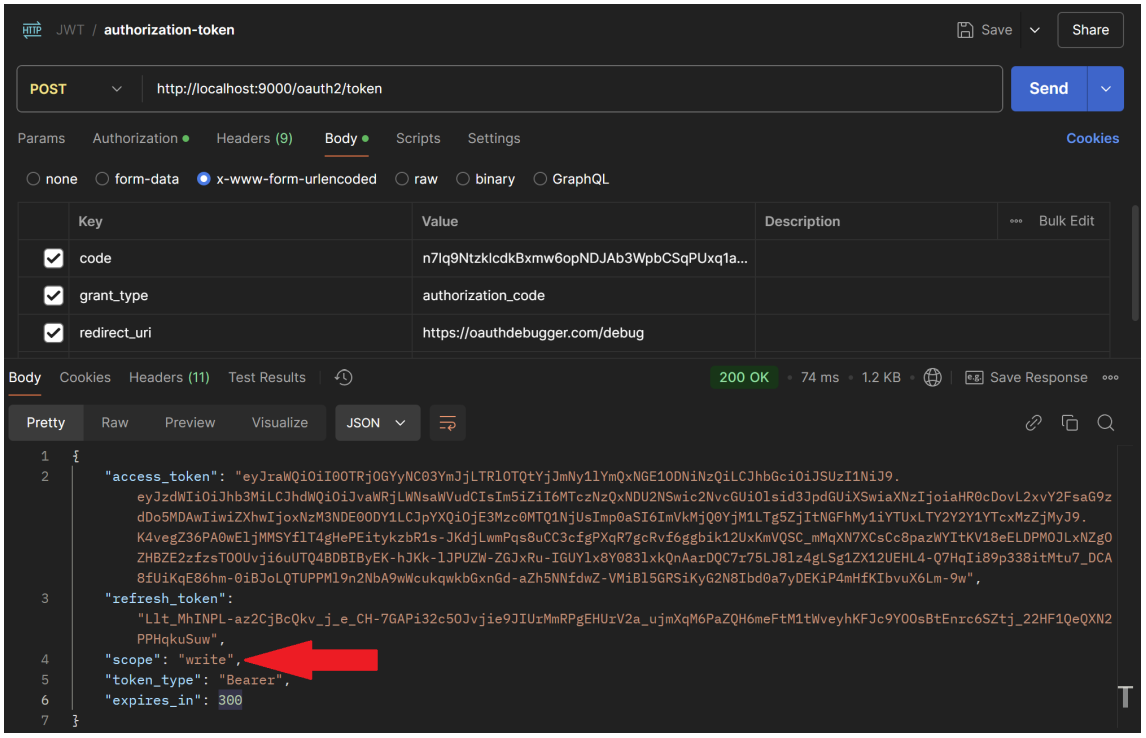

Figura 12: Petición POST a `http://localhost:9000/oauth2/token` con Postman: `scope` para `write`.

Tras cambiar los parámetros en la petición del código en **OAuth Debugger**, se envía la petición POST a `http://localhost:9000/oauth2/token`. Si todo es correcto, se obtendrá un *token* de acceso con el *scope* `write` esperado, como se ve en la figura 12.

Se puede consultar el *token* en `https://jwt.io/`, igual que se hizo en el flujo anterior, por lo que ahora no es necesario repetirlo.

Lo que sí se debe hacer es realizar una petición POST a `http://localhost:8081/resources/user` con el *token* obtenido como *Bearer Token* (en el apartado de *Authorization*), para consultar si se posee el permiso necesario para acceder a este método, como se muestra en la figura 13.

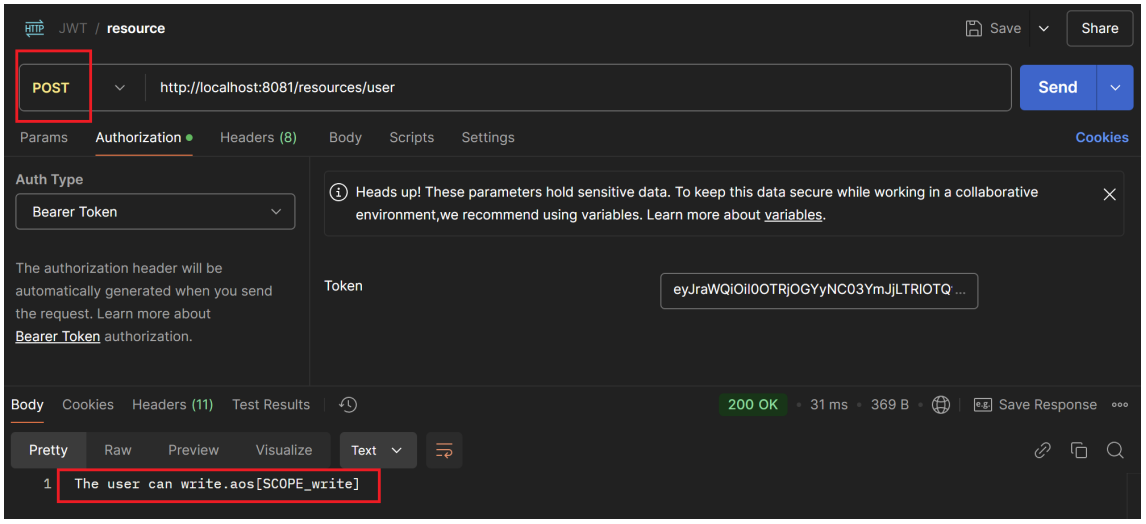

Figura 13: Petición POST a `http://localhost:8081/resources/user`, `scope` para `write`.

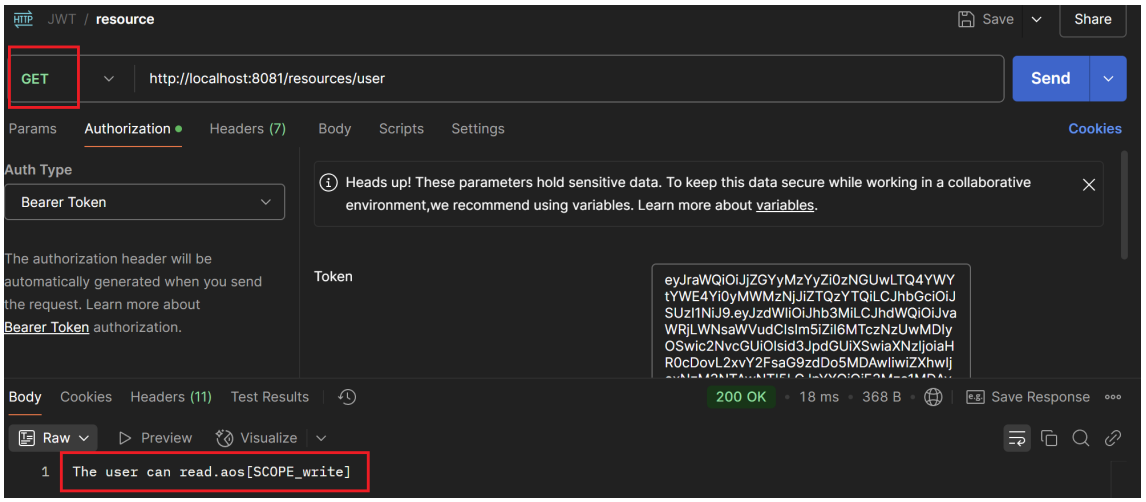

Figura 14: Petición GET a `http://localhost:8081/resources/user`, `scope` para `write`.

De igual forma, después se puede realizar una petición GET. Nótese la importancia del filtro definido en `SecurityConfig` para que se puedan realizar peticiones GET con ambos `SCOPE`, tanto `read` como `write`, como se ve en el código 16, en el que se resalta que se puede tener cualquier permiso (escritura o lectura) para acceder a las peticiones GET.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws
    Exception {
     http.authorizeHttpRequests(authorize -> authorize
         yellow.requestMatchers(HttpMethod.GET, "/resources/**")
         yellow.hasAnyAuthority("SCOPE_read", "SCOPE_write")
         .requestMatchers(HttpMethod.POST, "/resources/**")
         .hasAuthority("SCOPE_write")
         .anyRequest().authenticated())
         .oauth2ResourceServer((oauth2) ->
    oauth2.jwt(Customizer.withDefaults()));
     return http.build();
}
```

Listing 16: Filtro de seguridad en el servidor de recursos (I).

Si en vez de utilizar el método `hasAnyAuthority`, se hubiera implementado con `hasAuthority` (como en el código 17) para cada únicacmente un *scope*, la petición GET de la figura 14 con `write` no se habría podido realizar.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws
    Exception {
     http.authorizeHttpRequests(authorize -> authorize
         yellow.requestMatchers(HttpMethod.GET, "/resources/**")
         yellow.hasAuthority("SCOPE_write")
         .requestMatchers(HttpMethod.POST, "/resources/**")
         .hasAuthority("SCOPE_write")
         .anyRequest().authenticated())
         .oauth2ResourceServer((oauth2) ->
    oauth2.jwt(Customizer.withDefaults()));
     return http.build();
}
```

Listing 17: Filtro de seguridad en el servidor de recursos (II)

Ahora se probará el funcionamiento del servidor de recursos con el *scope* `read`, repitiendo el proceso de obtención del *token* previamente explicado, hasta llegar al paso de la figura 15.

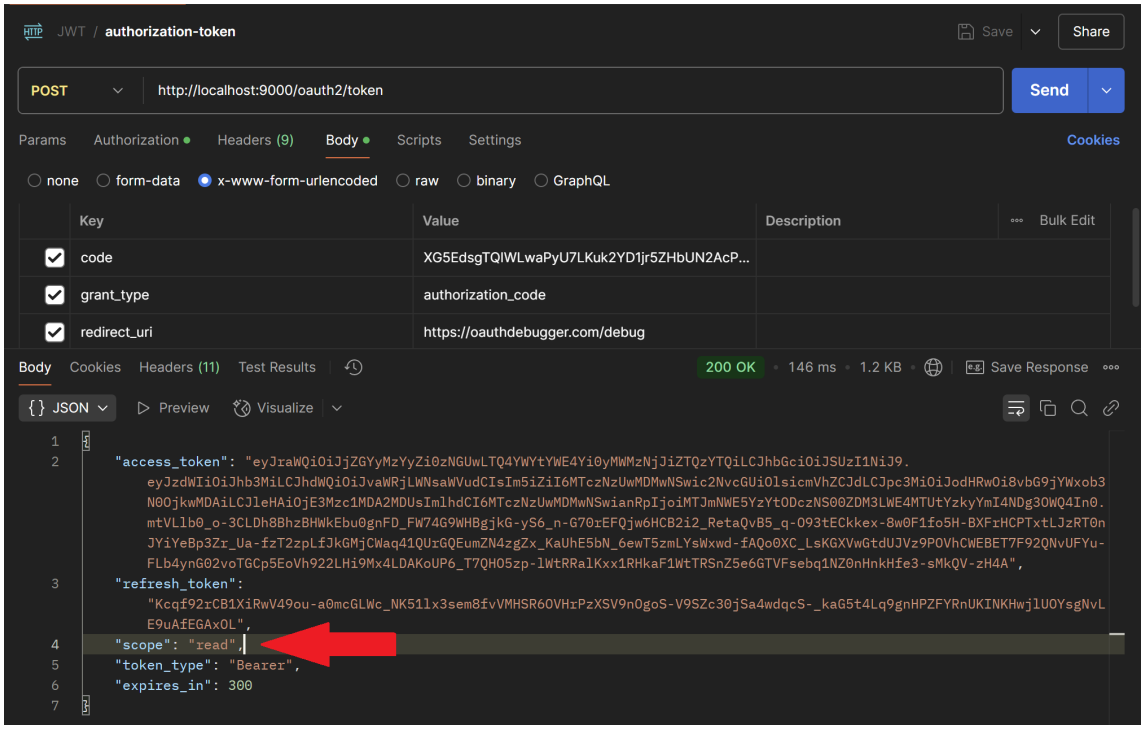

Figura 15: Petición POST a `http://localhost:9000/oauth2/token` con Postman: `scope` para `read`.

Una vez conseguido el *token* de acceso, se puede realizar una petición GET a `http://localhost:8081/resources/user`, como se ve en la figura 16. Si todo es correcto, se obtendrá una respuesta exitosa, ya que este *scope* sí tiene el permiso para acceder a esta petición.

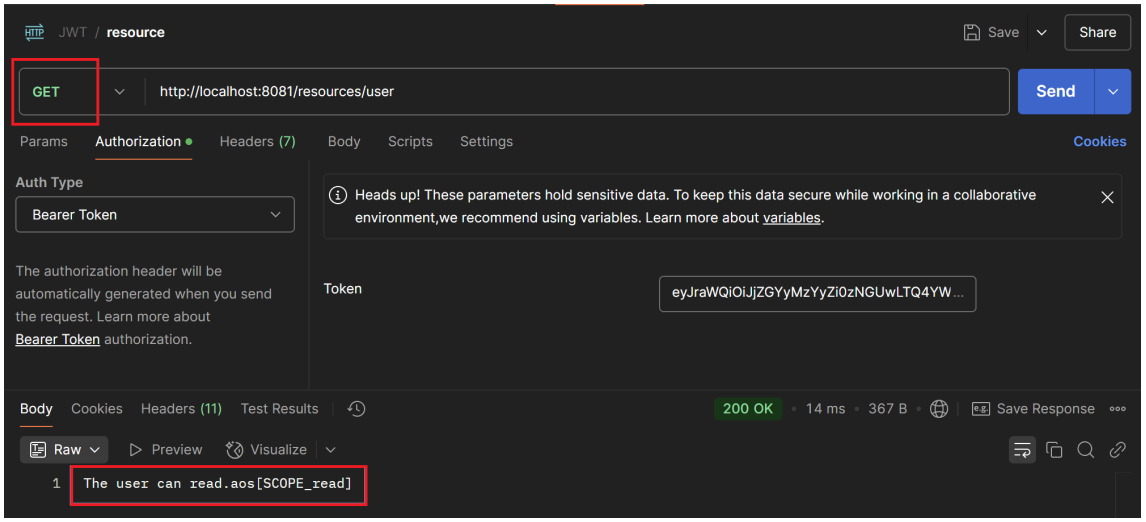

Figura 16: Petición GET a `http://localhost:8081/resources/user`, `scope` para `read`.

No obstante, si se intenta realizar una petición POST a http://localhost:8081/resources/user, como se ve en la figura 17, se obtendrá un error 403, ya que este *scope* no tiene permiso para acceder a esta petición. Únicamente se puede realizar la petición POST con el permiso de escritura, como se vio en la figura 13. Esto se explica por el código 18, en el que se resaltan las líneas que controlan este filtro para las peticiones POST.

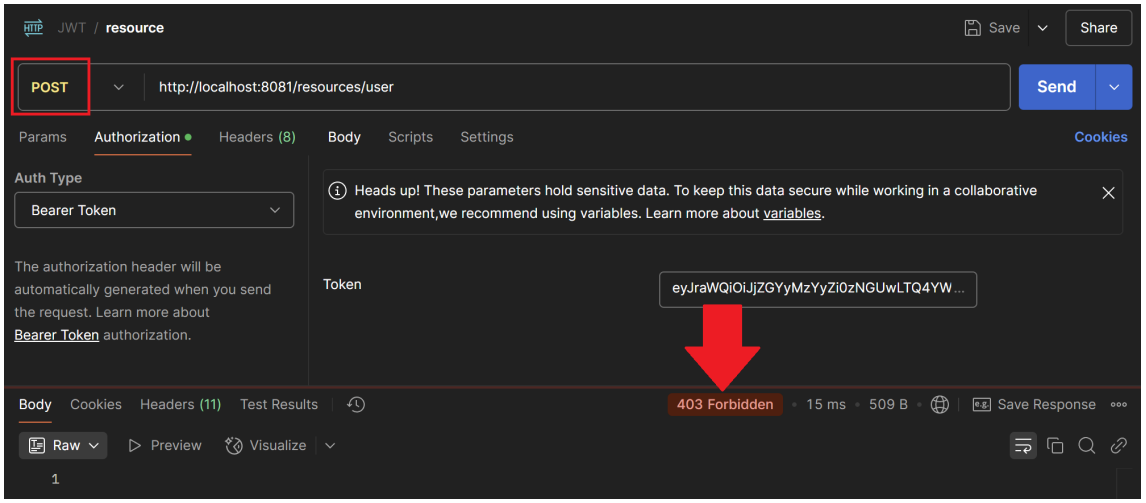

Figura 17: Petición POST a http://localhost:8081/resources/user, `scope` para `read`.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws
    Exception {
     http.authorizeHttpRequests(authorize -> authorize
          .requestMatchers(HttpMethod.GET, "/resources/**")
          .hasAuthority("SCOPE_write")
          yellow.requestMatchers(HttpMethod.POST, "/resources/**")
          yellow.hasAuthority("SCOPE_write")
          .anyRequest().authenticated())
          .oauth2ResourceServer((oauth2) ->
    oauth2.jwt(Customizer.withDefaults()));
     return http.build();
}
```

Listing 18: Filtro de seguridad en el servidor de recursos (III)

6.4. Cliente

6.4.1. Guía de implementación del cliente

Para finalizar la demostración, se creará un cliente en Spring Boot que solicitará los permisos para acceder a los recursos. Se utilizará de referencia la documentación oficial:

https://docs.spring.io/spring-security/reference/servlet/oauth2/index.html#oauth2-client

En el `application.properties` del cliente, se añadirán las siguientes líneas:

```
spring.application.name=cliente
logging.level.org.springframework.security=DEBUG
server.port=8080
```

Listing 19: application.properties del cliente.

Igual que en el servidor de recursos, se definirá el `application.yml` para complementar la configuración anterior con todas las cuestiones de seguridad:

```
spring:
  security:
    oauth2:
      client:
        registration:
          oauth-client:
            provider: servidor-autorizacion
            client-id: oauth-client
            client-name: oauth-client
            client-secret: 12345678910
            authorization-grant-type: authorization_code
              redirect-uri: "http://localhost:8080/authorized"
              scope: openid,profile,read,write
        provider:
          servidor-autorizacion:
            issuer-uri: http://localhost:9000
```

Listing 20: application.yml del cliente.

Con esta configuración, se está definiendo el cliente Spring Boot con su nombre, ID, secretos, tipo de autorización, URI de redirección y *scopes* que soporta. Al ser un ejemplo de juguete, se debe añadir el cliente en el servidor de autorización manualmente, modificando el método método `registeredClientRepository` para añadir este nuevo cliente, como se ve en el código 21.

```
// En el servidor de autorización
@Bean
public RegisteredClientRepository registeredClientRepository() {
    RegisteredClient oidcClient = ... // Código anterior

    RegisteredClient oauthClient =
   RegisteredClient.withId(UUID.randomUUID().toString())
        .clientId("oauth-client")
        .clientSecret("{noop}12345678910")

   .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)

   .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)

   .authorizationGrantType(AuthorizationGrantType.REFRESH_TOKEN)

   .redirectUri("http://localhost:8080/login/oauth2/code/oauth-client")
        .redirectUri("http://localhost:8080/authorized")
        .postLogoutRedirectUri("http://localhost:8080/logout")
```

```
            .scope(OidcScopes.OPENID)
            .scope(OidcScopes.PROFILE)
            .scope("read")
            .scope("write")
            .build();

        return new InMemoryRegisteredClientRepository(oidcClient,
        oauthClient);
    }
```

Listing 21: Modificación del cliente registrado en el servidor de autorización.

Las equivalencias entre el `application.yml` del cliente y el método `registeredClientRepository` son las siguientes:

- **client-id**: `clientId`: `oauth-client`
- **client-secret**: `clientSecret`: `12345678910`
- **authorization-grant-type**: `authorizationGrantType`: `authorization_code`
- **redirect-uri**: `redirectUri`:
  - Para el inicio de sesión: `http://localhost:8080/login/oauth2/code/oauth-client`
  - Tras el inicio de sesión: `http://localhost:8080/authorized`
- **scope**: `scope`: `openid,profile,read,write`
- **issuer-uri**: URL del servidor de autorización

En el cliente, también hay que definir un `SecurityConfig.java`, como se ve en el código 22, que se encargará de la configuración de la seguridad del cliente. Esta clase se podrá definir en cualquier paquete del proyecto, siempre y cuando se anote correctamente.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {
  @Bean
  public SecurityFilterChain securityFilterChain(HttpSecurity
   http) throws Exception {
    http.authorizeHttpRequests((oauthHttp) -> oauthHttp
            .requestMatchers(HttpMethod.GET,
   "/authorized").permitAll()
            .anyRequest().authenticated())
            .oauth2Login((login) ->
   login.loginPage("/oauth2/authorization/oauth-client"))
            .oauth2Client(Customizer.withDefaults());
    return http.build();
  }
}
```

Listing 22: Clase `SecurityConfig.java` del cliente.

Esta configuración autorizará al *endpoint* `/authorized`, que se definirá en el siguiente controlador mostrado en el código 23, cuando el usuario haya iniciado sesión correctamente. Además, se ha definido el *login* con OAuth 2.0, que redirigirá al servidor de autorización para iniciar sesión, y se ha definido el cliente OAuth 2.0 con los valores por defecto.

```java
@RestController
public class ClientController {
    @GetMapping("/authorized")
    public Map<String, String> authorize(@RequestParam String
    code) {
        return Collections.singletonMap("authorizationCode", code);
    }
}
```

Listing 23: Clase `ClientController.java` del cliente.

En el ejemplo resuelto, también hay un *endpoint* con el clásico *Hello* que se puede utilizar para comprobar el funcionamiento del cliente, dentro del mismo controlador `ClientController.java`.

Con todos estos cambios, ya estaría terminado el cliente Spring Boot y adaptado el servidor de autorización para que pueda funcionar con este cliente. Ahora se debe reiniciar el servidor de autorización y ejecutar el cliente (`mvn spring-boot:run`) para realizar el último flujo de esta práctica.

6.4.2. Flujo del cliente

Se comienza la demostración accediendo al cliente: `http://localhost:8080`, donde se redigirá al servidor de autorización, quien solicitará iniciar sesión con los datos de usuario, como se ve en la figura 18.

Figura 18: Inicio de sesión en el cliente.

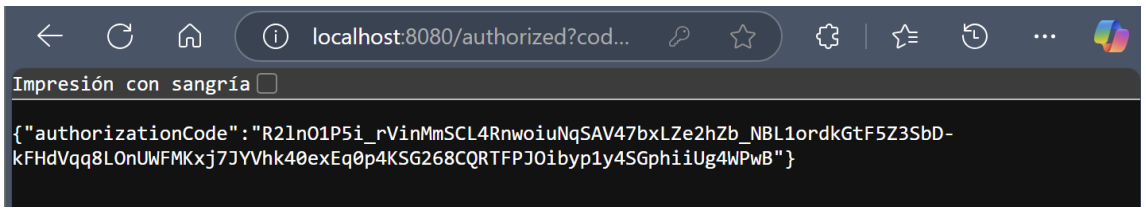

Figura 19: Código de acceso devuelto por el servidor de autorización.

Si las credenciales son correctas, el servidor de autorización devolverá directamente el código de acceso, como se ve en la figura 19. Básicamente, es lo mismo que se realizaba en el Flujo de autorización, pero utilizando el cliente Spring Boot en vez de *OAuth Debugger*.

El siguiente paso es crear una nueva petición en Postman, o modificar la que se realizó en el Flujo de autorización, a la dirección `http://localhost:8081/resources/user`:

- **Authorization**:
  - **Type**: Basic Auth
  - **Username**: yellow**oauth-client**
  - **Password**: yellow**12345678910**
- **Body** (x-www-form-urlencoded):
  - **code**: código devuelto por el cliente
  - **grant_type**: authorization_code
  - **redirect_uri**: `http://localhost:8080/authorized` (yellow**¡ES DISTINTA A LA PETICIÓN ANTERIOR!**)

Se debe introducir el *token* obtenido del servidor de autorización en el paso anterior, como se ve en la figura 20.

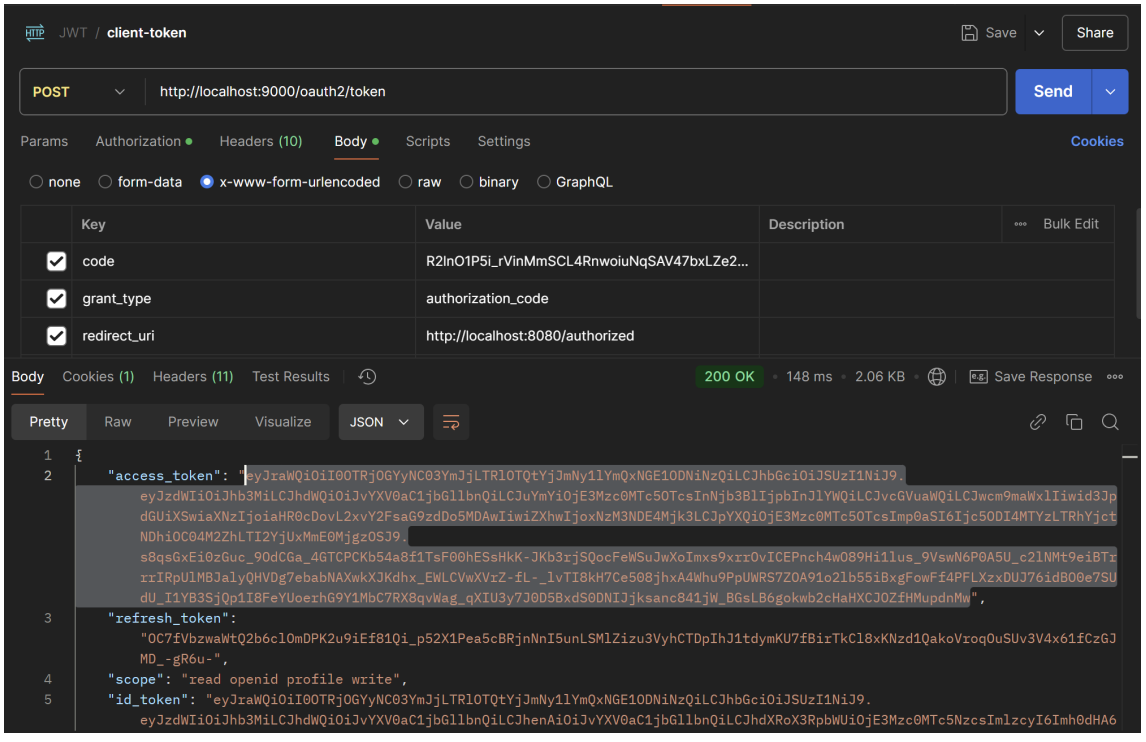

Figura 20: Petición POST a `http://localhost:8081/resources/user` con el código del cliente Spring Boot.

De igual forma, se puede comprobar el *token* en `https://jwt.io/`, como se ve en la figura 21.

Por último, se comprobarán los permisos de acceso realizando una petición POST al servidor

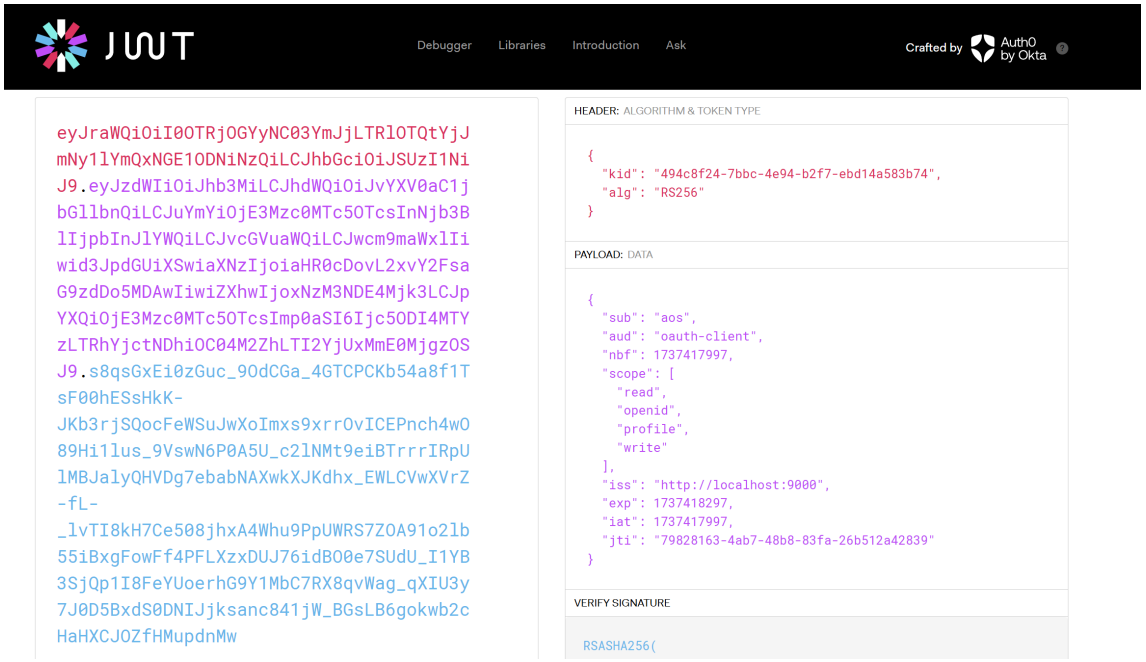

Figura 21: Comprobación del *token* en jwt.io.

de recursos, como se ve en la figura 22. Aquí se puede ver que este cliente contaba con varios *scopes* diferentes, lo que se ve reflejado en la respuesta del servidor de recursos (22), aunque ya se podía ver en la representación del JWT (21).

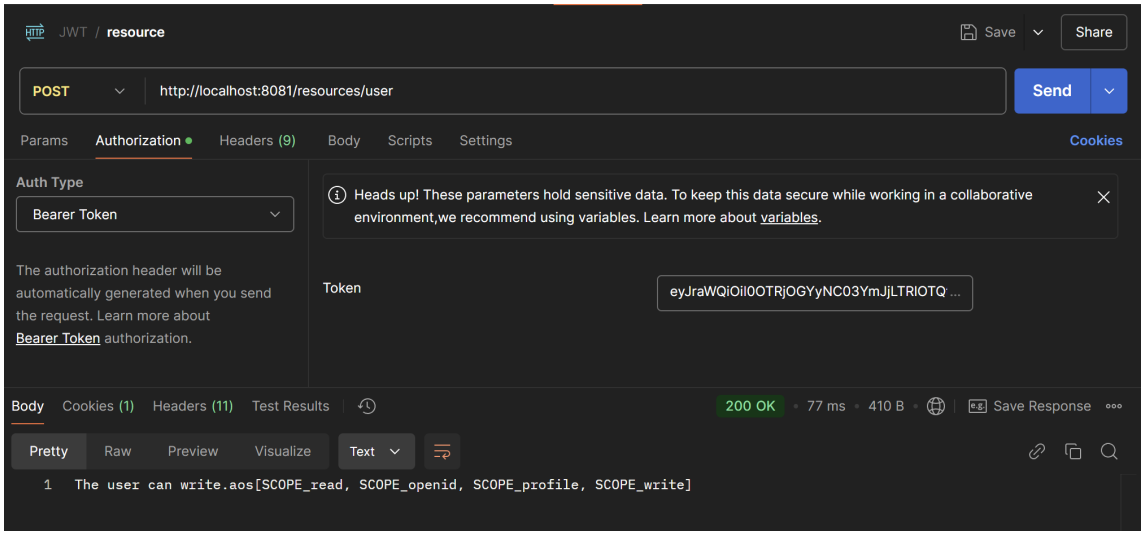

Figura 22: Petición POST a `http://localhost:8081/resources/user` con el *token* del cliente Spring Boot.

6.5. Resumen de las prácticas

En esta práctica, se han implementado tres proyectos para entender el funcionamiento de OAuth 2.0 con Spring Boot. Se ha creado un servidor de autorización, un servidor de recursos y un cliente, que han interactuado entre sí para solicitar y conceder permisos de acceso a los recursos.

Estos proyectos se han implementado paso a paso, siguiendo la documentación oficial de Spring Boot y OAuth. Tras completar cada uno de los proyectos, se ha probado su funcionamiento, siguiendo el flujo explicado en la parte teórica. Puede haber resultado algo repetitivo, pero se cree que de esta forma se consigue entender claramente el funcionamiento básico de OAuth, así como se han visto ejemplos de *tokens* como JWT.

Los servidores de autorización y de recursos, así como el cliente OAuth, son muy sencillos, como de juguete. Aun así, se espera que, si en un futuro se desea implementar OAuth en un proyecto real a tan bajo nivel, se tenga una base sólida para hacerlo. No obstante, seguramente en un entorno real, no se implementen estas funcionalidades de esta forma tan manual, sino que se haga uso de otras librerías o servicios para facilitar el trabajo y no tener que picar tanto código.

Esperamos que haya sido una práctica interesante, y que haya servido para asentar los conocimientos básicos del flujo de OAuth y el uso de JWT.

Gracias por su atención.

6.6. Referencias de las prácticas

- **Spring Boot Initializr**: https://start.spring.io
- **Repositorio de la práctica**: https://github.com/saguit03/oauth-demo
- **Vídeo tutorial**: https://youtu.be/oHiIBkSv3nw
- **OAuth 2.0**: https://oauth.net/2/
- **Debugger de OAuth**: https://oauthdebugger.com/
- **Documentación Spring Authorization Server**: https://docs.spring.io/spring-authorization-server/reference/
- **Documentación Spring Resource Server**: https://docs.spring.io/spring-security/reference/servlet/oauth2/resource-server/
- **Documentación Spring OAuth2 Client**: https://docs.spring.io/spring-security/reference/servlet/oauth2/index.html#oauth2-client
- **JWT**: https://jwt.io/

7. Conclusiones personales

Con este proyecto hemos aprendido a utilizar de forma básica el protocolo OAuth. Hemos adquirido algunos conocimientos teóricos básicos, que hemos reforzado con el ejemplo práctico; logrando así un mayor entendimiento de su funcionamiento. Además, se ha visto un tipo específico de *token*: JWT, que es bastante sencillo de utilizar.

La demo se ha implementado a muy bajo nivel, siguiendo la línea de los laboratorios de Spring Boot durante la primera parte de la asignatura. Entendiendo esta práctica, será fácil trasladar el flujo de OAuth a otras aplicaciones, ya que la esencia del protocolo es la misma.

La experiencia de organizar una clase teórico-práctica para el resto de compañeros fue emocionante. Consideramos que transmitir lo que hemos estudiado sirve para reforzar todavía más el aprendizaje. Esperamos que nuestra clase fuera entretenida y amena, a pesar de las complicaciones que hubo durante la parte práctica. Ojalá pudiéramos repetir la experiencia en un futuro, ¡y mejor preparados!

Estudiantes

Ambos estudiantes han participado en la redacción de la memoria, la presentación en clase y la realización del *Kahoot*.

Pablo Fernández González

Email: pfernandzoq@alumnos.unex.es

Participante. Ha redactado parte del contenido teórico y las conclusiones, y se ha encargado del diseño de las transparencias de la presentación.

Sara Guillén Torrado

Email: sguillenl@alumnos.unex.es. Todas las comunicaciones se establecerán con este autor.

Líder del proyecto y participante. Ha redactado la sección de JWT y la guía de las prácticas. También ha coordinado el proyecto LaTeXde la memoria y supervisó la creación de las transparencias de la presentación.

8. Anexo. Resumen peticiones en Postman

8.1. Servidor de autorización (cliente OAuth Debugger)

- **URL**: http://localhost:9000/oauth2/token
- **Tipo de petición**: POST
- **Authorization**:
  - **Type**: Basic Auth
  - **Username**: oidc-client
  - **Password**: oidc-secret
- **Body** (x-www-form-urlencoded):
  - **code**: código devuelto por el debugger
  - **grant_type**: authorization_code
  - **redirect_uri**: https://oauthdebugger.com/debug

8.2. Servidor de recursos

- **URL**: http://localhost:9000/oauth2/token
- **Tipo de petición**: POST
- **Authorization**:
  - **Type**: Bearer Token
  - **Token**: *token* devuelto por el servidor de autorización
- **Body**: vacío

8.3. Servidor de autorización (cliente Spring Boot)

- **URL**: http://localhost:9000/oauth2/token
- **Tipo de petición**: POST
- **Authorization**:
  - **Type**: Basic Auth
  - **Username**: oauth-client
  - **Password**: 12345678910
- **Body** (x-www-form-urlencoded):
  - **code**: código devuelto por el cliente
  - **grant_type**: authorization_code
  - **redirect_uri**: http://localhost:8080/authorized

9. Anexo. Dependencias del pom.xml

```
<dependencies>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-security</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <scope>runtime</scope>
        <optional>true</optional>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-test</artifactId>
        <scope>test</scope>
    </dependency>
    <dependency>
        <groupId>org.springframework.security</groupId>
        <artifactId>spring-security-test</artifactId>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Listing 24: Dependencias comunes del `pom.xml` de los proyectos.

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

Listing 25: Dependencia del servidor de autorización

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

Listing 26: Dependencia del servidor de recursos

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

Listing 27: Dependencia del cliente OAuth

10. Anexo. Paquetes de los proyectos

10.1. Paquetes del servidor de autorización

```
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.util.UUID;

import com.nimbusds.jose.jwk.JWKSet;
import com.nimbusds.jose.jwk.RSAKey;
import com.nimbusds.jose.jwk.source.ImmutableJWKSet;
import com.nimbusds.jose.jwk.source.JWKSource;
import com.nimbusds.jose.proc.SecurityContext;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.annotation.Order;
import org.springframework.http.MediaType;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
    org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.oauth2.core.AuthorizationGrantType;
import org.springframework.security.oauth2.core.ClientAuthenticationMethod;
import org.springframework.security.oauth2.core.oidc.OidcScopes;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.server.authorization.client
    .InMemoryRegisteredClientRepository;
import org.springframework.security.oauth2.server.authorization.client
    .RegisteredClient;
import org.springframework.security.oauth2.server.authorization.client
    .RegisteredClientRepository;
import org.springframework.security.oauth2.server.authorization.config
    .annotation.web.configuration.OAuth2AuthorizationServerConfiguration;
import org.springframework.security.oauth2.server.authorization.config
    .annotation.web.configurers.OAuth2AuthorizationServerConfigurer;
import org.springframework.security.oauth2.server.authorization.settings
    .AuthorizationServerSettings;
import org.springframework.security.oauth2.server.authorization.settings
    .ClientSettings;
import org.springframework.security.provisioning.InMemoryUserDetailsManager;
import org.springframework.security.web.SecurityFilterChain;
import
    org.springframework.security.web.authentication.LoginUrlAuthenticationEntryPoint;
import org.springframework.security.web.util.matcher.MediaTypeRequestMatcher;
```

Listing 28: Paquetes del SecurityConfig del servidor de autorización.

10.2. Paquetes del servidor de recursos

```
import java.time.Duration;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
    org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.oauth2.core.DelegatingOAuth2TokenValidator;
import org.springframework.security.oauth2.core.OAuth2TokenValidator;
import org.springframework.security.oauth2.jwt.Jwt;
import org.springframework.security.oauth2.jwt.JwtDecoder;
import org.springframework.security.oauth2.jwt.JwtDecoders;
import org.springframework.security.oauth2.jwt.JwtIssuerValidator;
import org.springframework.security.oauth2.jwt.JwtTimestampValidator;
import org.springframework.security.oauth2.jwt.JwtValidators;
import org.springframework.security.oauth2.jwt.NimbusJwtDecoder;
import org.springframework.security.web.SecurityFilterChain;
```

Listing 29: Paquetes del SecurityConfig del servidor de recursos.

```
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

Listing 30: Paquetes del ResourceController del servidor de recursos.

10.3. Paquetes del cliente OAuth

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.http.HttpMethod;
import org.springframework.security.config.Customizer;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
    org.springframework.security.config.annotation.web.configuration.EnableWebSecurity;
import org.springframework.security.web.SecurityFilterChain;
```

Listing 31: Paquetes del SecurityConfig del cliente OAuth.

```
import java.util.Collections;
import java.util.Map;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.RequestParam;
```

Listing 32: Paquetes del ClientController del cliente OAuth.

11. Anexo. Kahoot: OAuth y JWT

Durante la presentación, se jugó a un Kahoot! (https://kahoot.it/). Dado que está en privado, en este anexo se incluirán las preguntas, las posibles opciones y la respuesta correcta.

1. **¿Qué es OAuth?**
   - Un estándar abierto de autorización [CORRECTA]
   - Un estándar abierto de autenticación
2. **¿Cuáles son las versiones de OAuth?**
   - OAuth 3.5, 4 y o1
   - OAuth 1.0, 2.0 y 3.0
   - OAuth 1.0 y 2.0 [CORRECTA]
   - Solo existe OAuth 2.0
3. **¿Qué roles forman parte del ujo de comunicación del protocolo OAuth 2.0?**
   - Cliente, propietario, servidor de recursos y servidor de autorización [CORRECTA]
   - Cliente, servidor de autorización y servidor de recursos
   - Cliente, propietario y servidor de recursos
   - Cliente y servidor
4. **¿Cuál es la función principal del servidor de autorización?**
   - Da acceso al recurso
   - Otorga autorización sobre el recurso
   - Otorga *token* de acceso [CORRECTA]
   - Autentica al propietario del recurso
5. **¿Qué rol se encarga de emitir *tokens* de acceso?**
   - Cliente
   - Propietario de los recursos
   - Servidor de recursos
   - Servidor de autorización [CORRECTA]
6. **¿Qué paso sucede primero durante el ujo de comunicación de OAuth?**
   - Se emite el *token* de acceso
   - Se autoriza al cliente [CORRECTA]
   - Se da acceso al recurso
   - Se refresca el *token*

7. **¿Qué significa JOSE en JWT?**
   - Java Object Signing and Encryption [CORRECTA]
   - Javascript Object Signing and Encryption
   - Java Object Signature and Entity
   - El nombre del creador
8. **¿Es válido este JWT?** (Véase figura 23)
   - Sí
   - No, falta la firma
   - No, está en base 32
   - No, por otro motivo [CORRECTA, el JWT va separado por «.», no por «:»]
9. **¿Cuál es una característica de OAuth?**
   - Centrado en las API
   - Almacenamiento solo en el lado del cliente
   - Formato de token
   - Todas son incorrecta [CORRECTA para OAuth, ya que son características de JWT]
10. **¿Cuál NO es uno de los beneficios de JWT?**
    - Es compacto
    - Es robusto
    - Es seguro
    - Mucho tiempo de vida [CORRECTA]

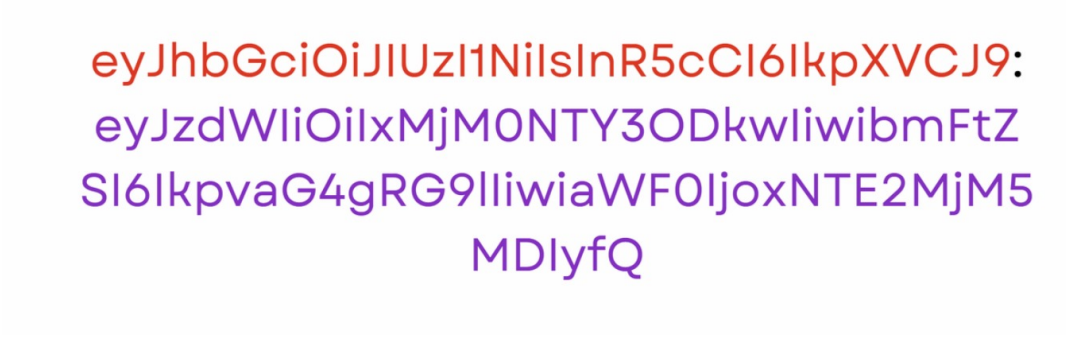

Figura 23: ¿Es válido este JWT?