Índice

Listings

Índice de figuras

1. Demo de OAuth 2.0 paso a paso

En esta práctica, se realizarán tres proyectos en Spring Boot 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.

1.1. Preparación previa. Creación de los proyectos

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 1, 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).

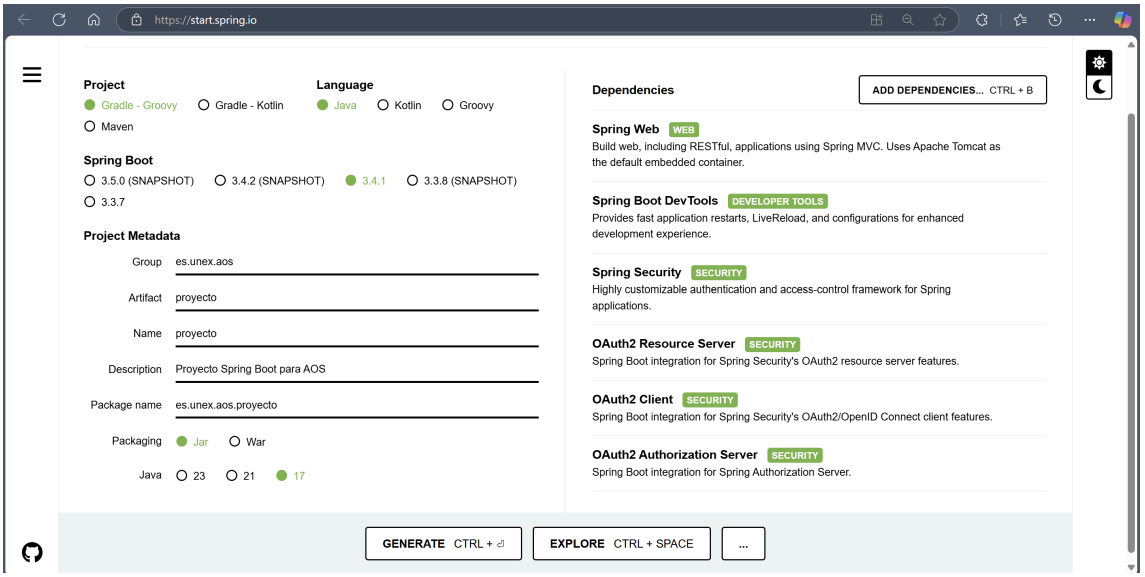

Figura 1: Configuración de un proyecto en Spring Boot Initializr.

1.2. Servidor de autorización

1.2.1. 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 1: 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 2: 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 3).

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 -> csrf.disable())
          .formLogin(Customizer.withDefaults());
     return http.build();
}
```

Listing 3: 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 4). 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 4: 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 5, 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 5: 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.

1.2.2. 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 2):

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

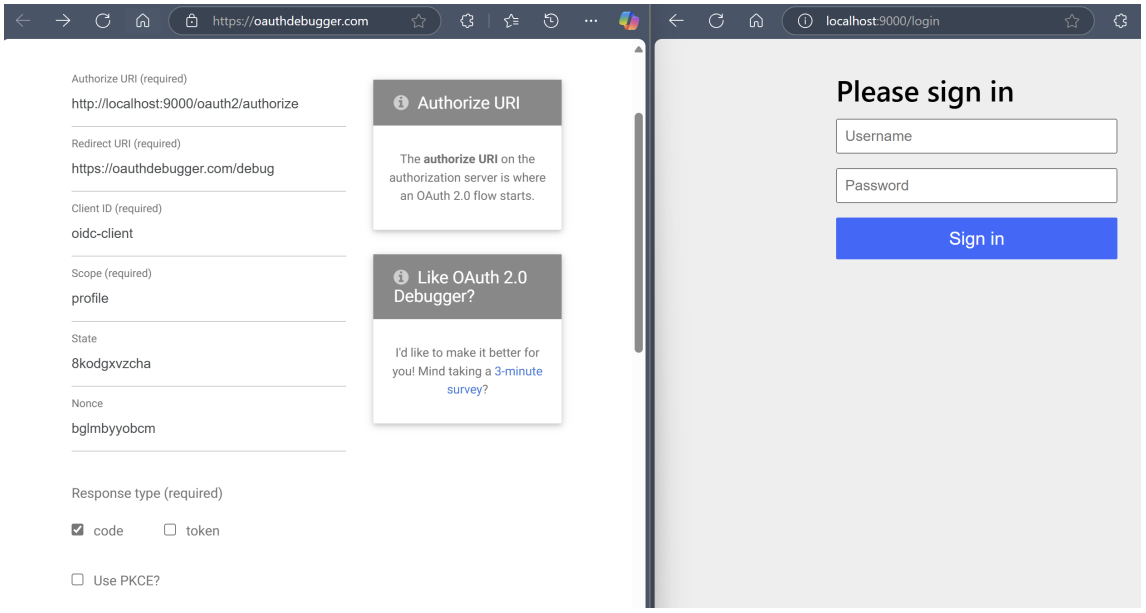

Figura 2: OAuth Debugger y servidor de autorización.

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 2).

Si todo es correcto, se redirigirá a la página de *OAuth Debugger* con un código en la URL, como se ve en la figura 3. Este código es el *token* que se utilizará para obtener el *token* de acceso.

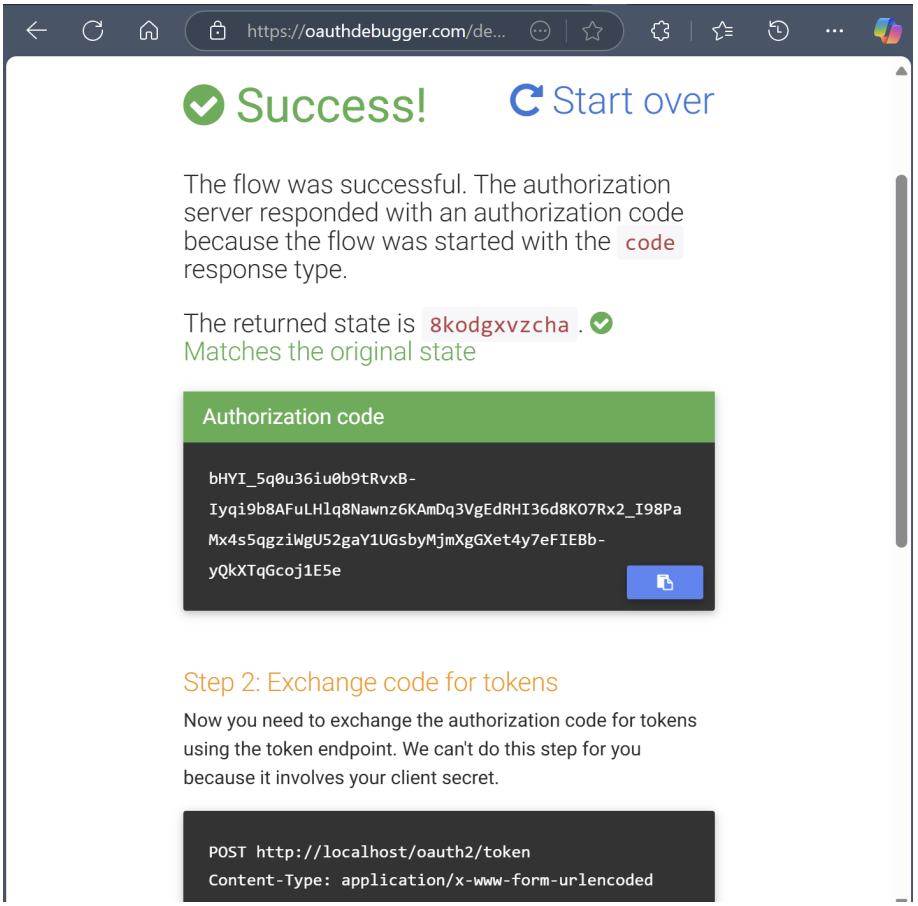

Figura 3: Respuesta en OAuth Debugger del servidor de autorización.

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 4):
  - **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 5):
  - **code**: código devuelto por el debugger
  - **grant_type**: authorization_code
  - **redirect_uri**: `https://oauthdebugger.com/debug`

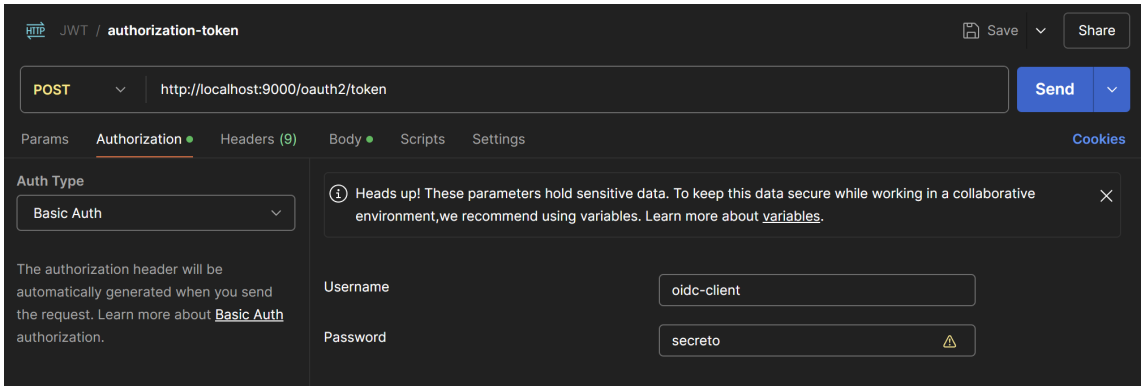

Figura 4: Petición POST a `http://localhost:9000/oauth2/token` con Postman: `Authorization`

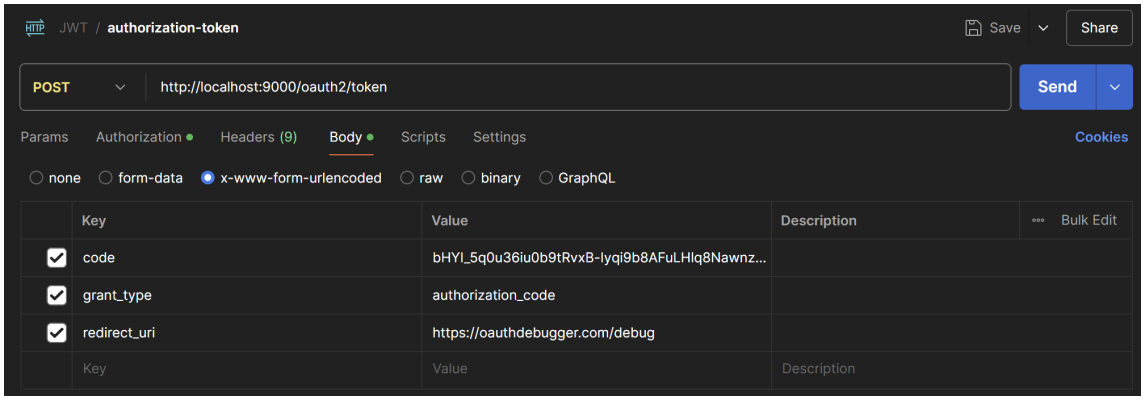

Figura 5: Petición POST a `http://localhost:9000/oauth2/token` con Postman: `Body`

Si todo va bien, se obtendrá un *token* de acceso, como se ve en la figura 6. Este *token* se podría utilizar para acceder a los recursos protegidos del servidor de recursos, pero todavía no tenemos implementado este servicio.

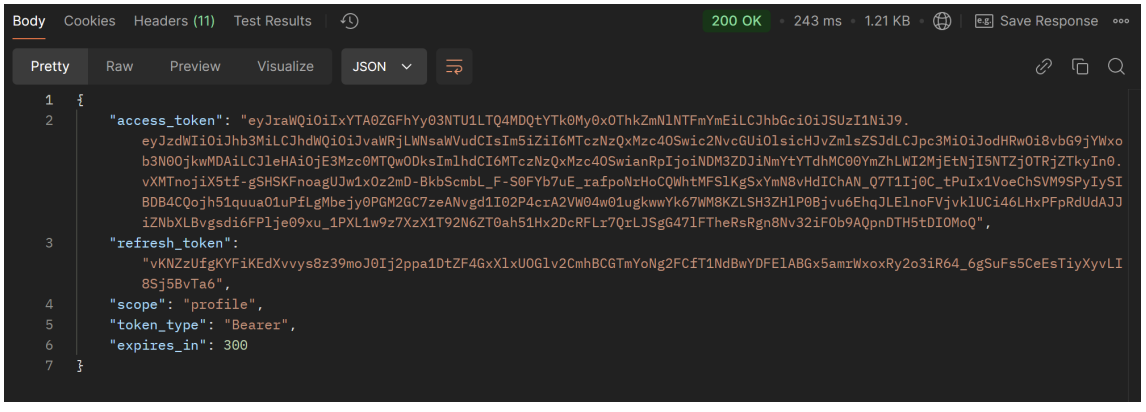

Figura 6: Respuesta de la petición POST a `http://localhost:9000/oauth2/token`

Para un mayor entendimiento, se puede comprobar el *token* en `https://jwt.io/`, como se ve en la figura 7.

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.

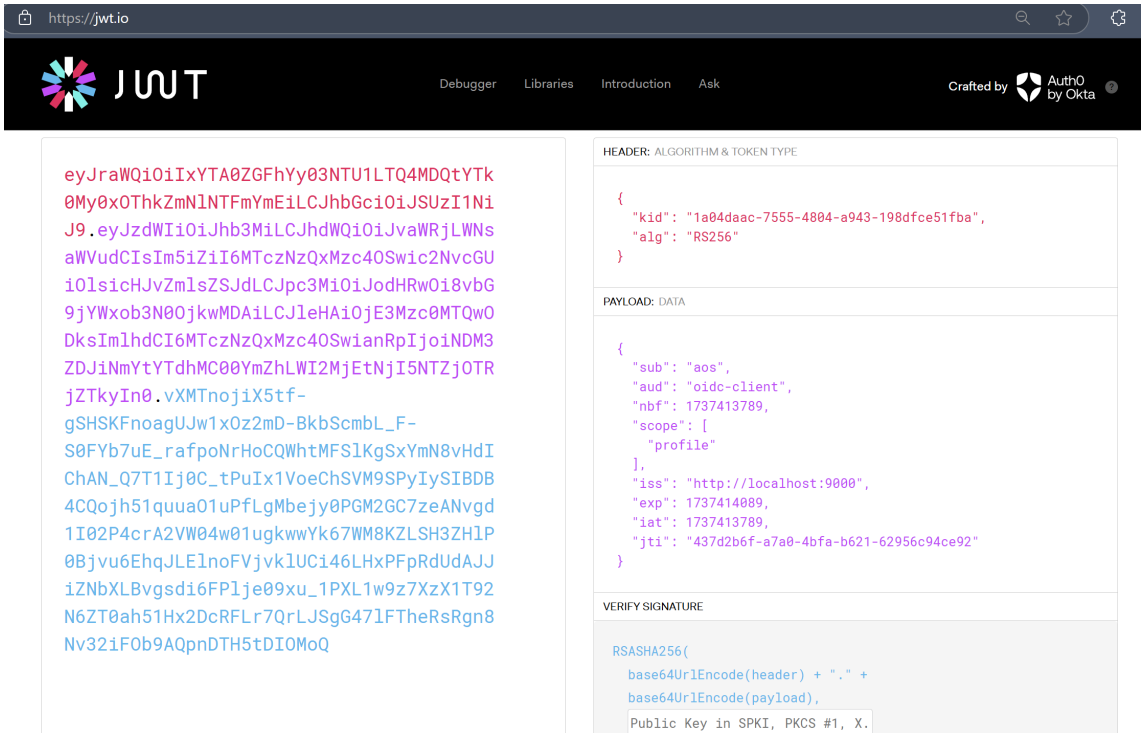

Figura 7: Comprobación del *token* en jwt.io.

En este caso, estos datos se han introducido manualmente en el primer paso del flujo (véase la figura 2). Asimismo, se ven otros como el `iss` (emisor), `exp` (fecha de expiración), `iat` (hora de emisión) y `jti` (identificador único del *token*).

1.3. Servidor de recursos

1.3.1. 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 6: 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 7: 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 8: 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 9.

```
@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 9: 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 10.

```
@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)
        .scope('read') // sustituir ' por "
        .scope('write') // sustituir ' por "
        .build();
    return new InMemoryRegisteredClientRepository(oidcClient);
}
```

Listing 10: 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).

1.3.2. 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.

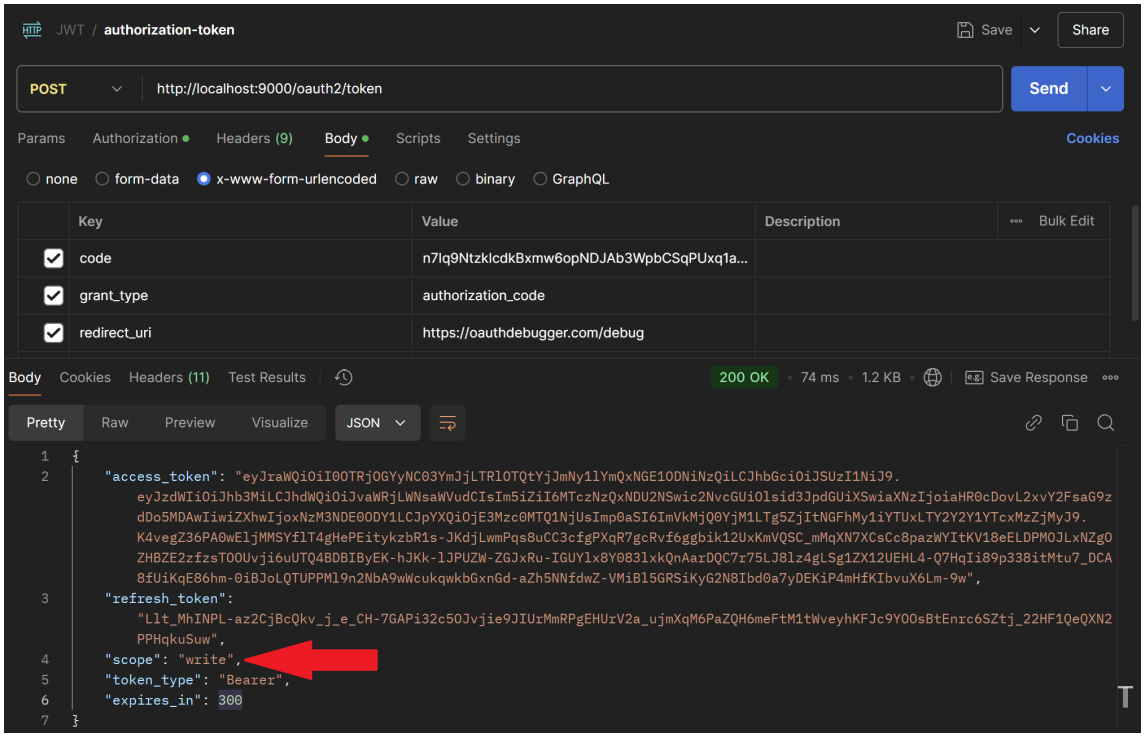

Figura 8: Petición POST a `http://localhost:9000/oauth2/token` con Postman: `scope` para `write`.

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 8.

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 9.

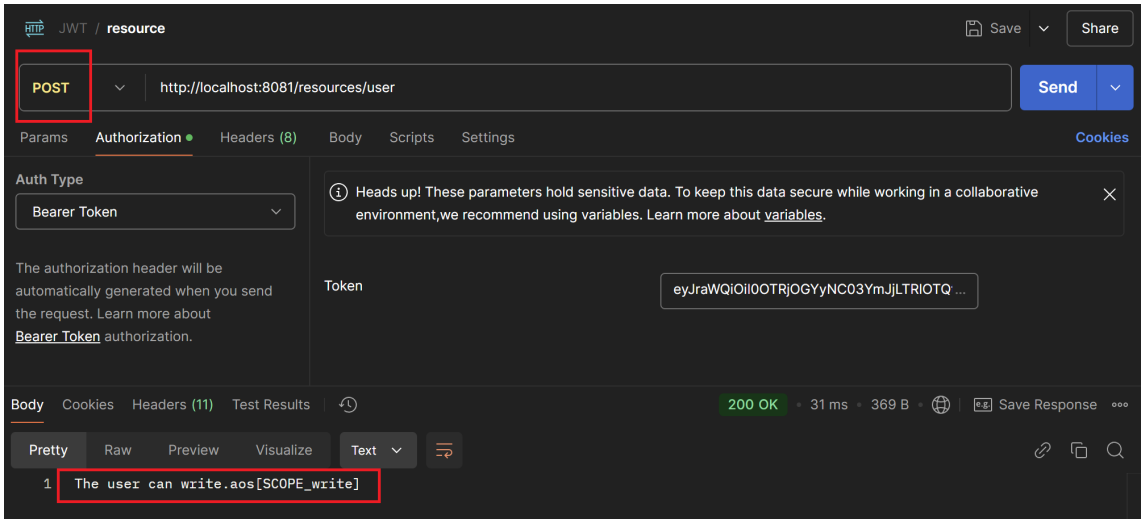

Figura 9: Petición POST a `http://localhost:8081/resources/user`, `scope` para `write`.

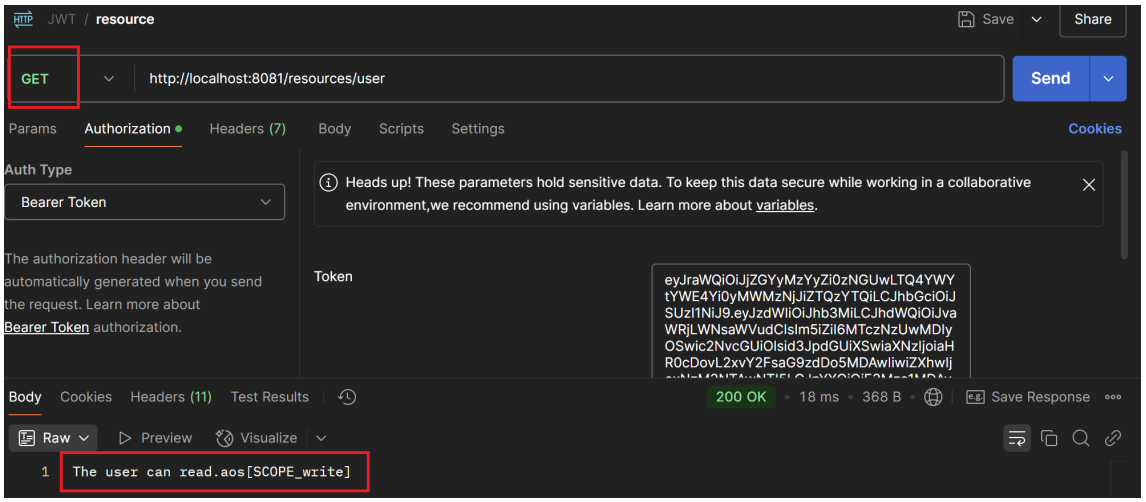

Figura 10: Petición GET a `http://localhost:8081/resources/user`, `scope` para `write`.

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 11, 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
          .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();
}
```

Listing 11: 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 12) para cada únicacmente un *scope*, la petición GET de la figura 10 con `write` no se habría podido realizar.

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

Listing 12: 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 11.

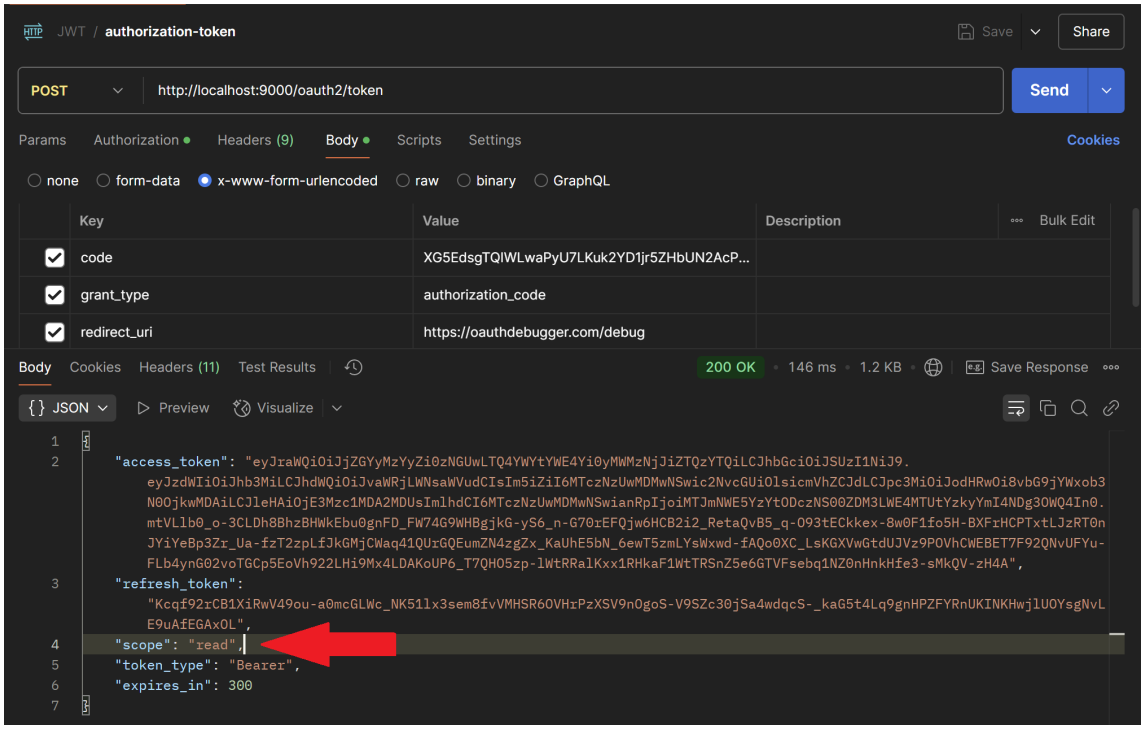

Figura 11: Petición POST a `http://localhost:9000/oauth2/token` con Postman: `scope` para `read`.

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 12. Si todo es correcto, se obtendrá una respuesta exitosa, ya que este *scope* sí tiene el permiso para acceder a esta petición.

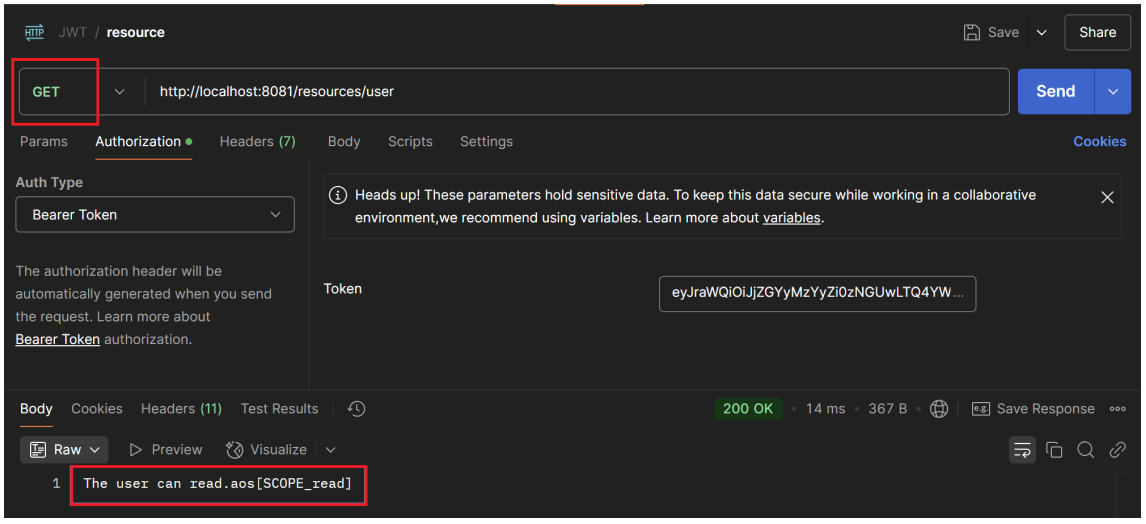

Figura 12: Petición GET a `http://localhost:8081/resources/user`, `scope` para `read`.

No obstante, si se intenta realizar una petición POST a http://localhost:8081/resources/user, como se ve en la figura 13, 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 9. Esto se explica por el código 13, en el que se resaltan las líneas que controlan este filtro para las peticiones POST.

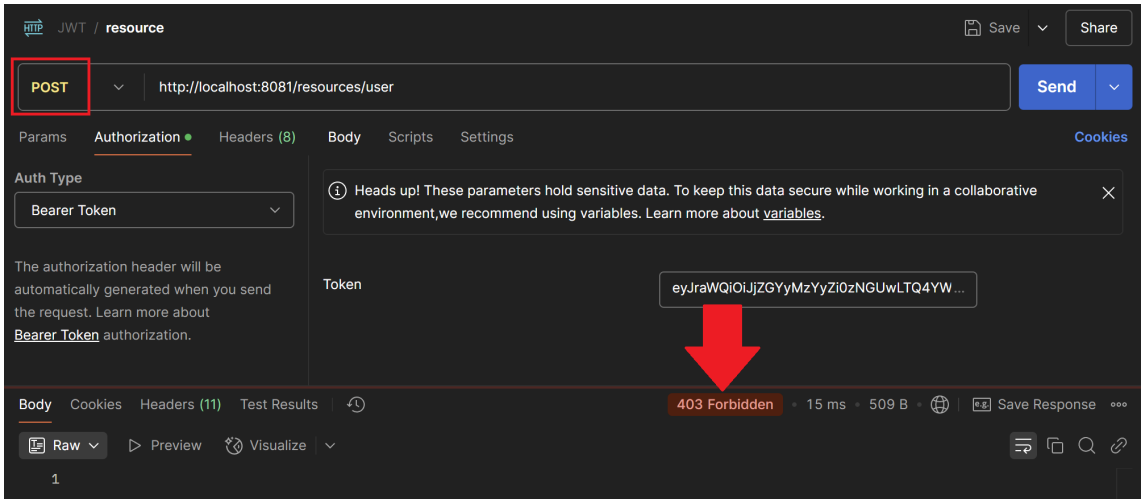

Figura 13: Petición POST a http://localhost:8081/resources/user, `scope` para `read`.

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

Listing 13: Filtro de seguridad en el servidor de recursos (III)

1.4. Cliente

1.4.1. 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 14: 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 15: 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 16.

```
// 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 16: 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 17, 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 17: Clase `SecurityConfig.java` del cliente.

Esta configuración autorizará al *endpoint* `/authorized`, que se definirá en el siguiente controlador mostrado en el código 18, 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.

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

Listing 18: 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.

1.4.2. 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 14.

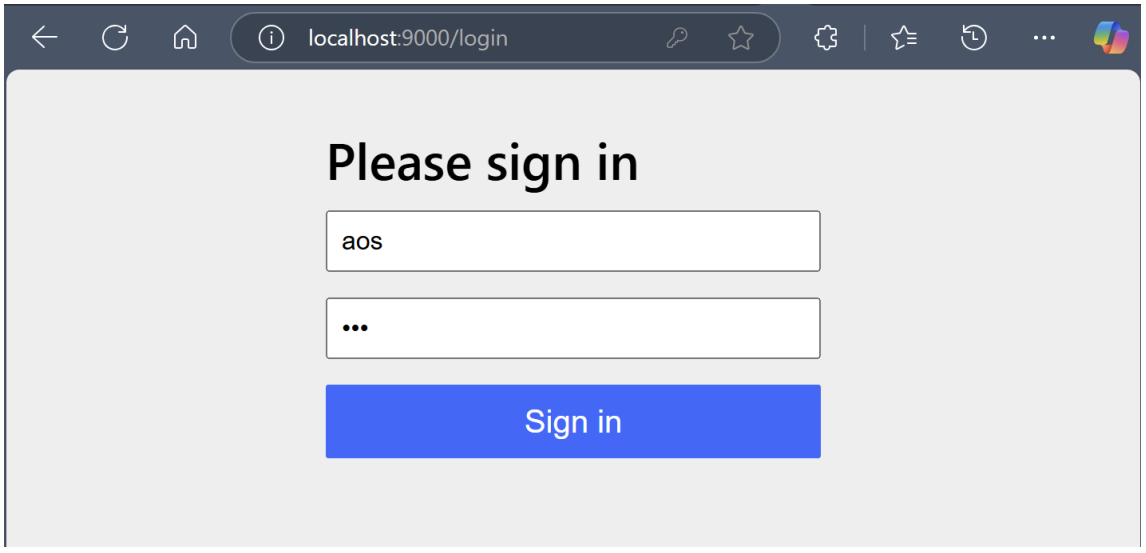

Figura 14: Inicio de sesión en el cliente.

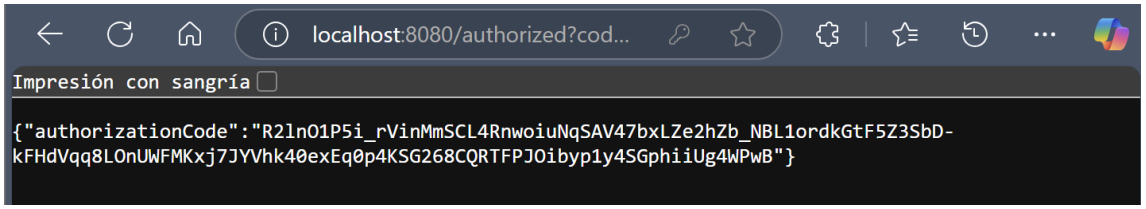

Figura 15: Código de acceso devuelto por el servidor de autorización.

Si las credenciales son correctas, el servidor de autorización devolverá directamente el código de acceso, como se ve en la figura 15. 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**: **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` (**¡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 16.

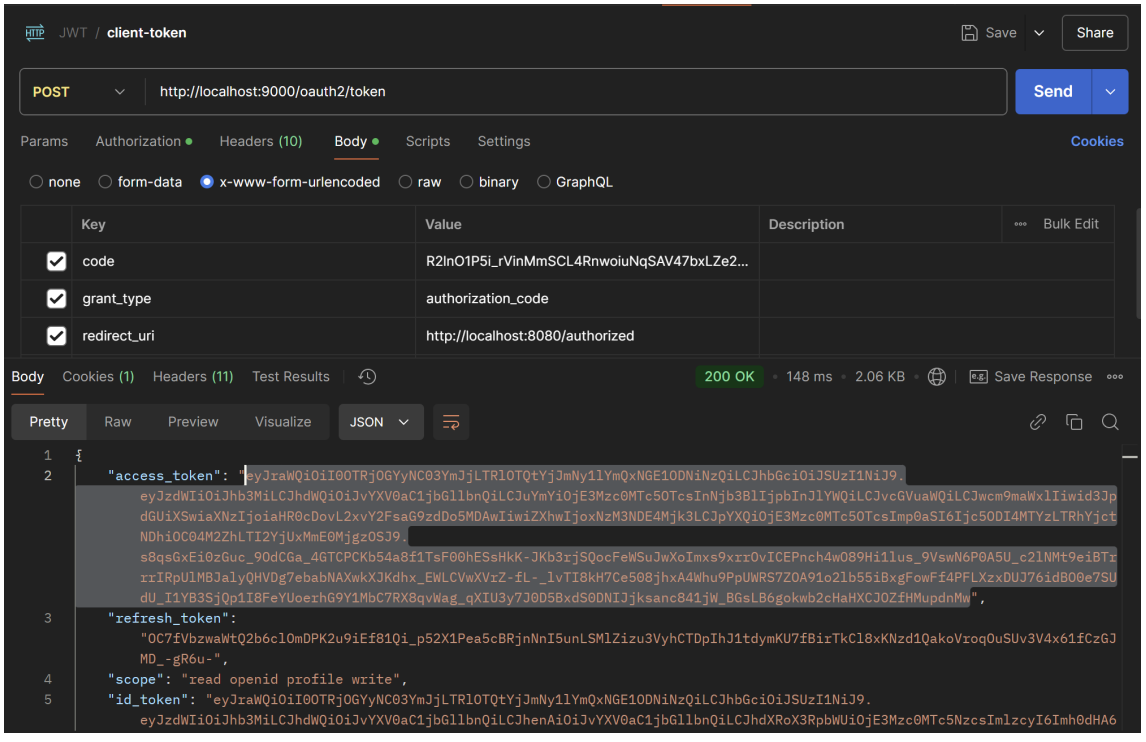

Figura 16: Petición POST a `http://localhost:8081/resources/user` con el código del cliente Spring Boot.

De igual forma, se puede comprobar el *token* en `https://jwt.io/`, como se ve en la figura 17.

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

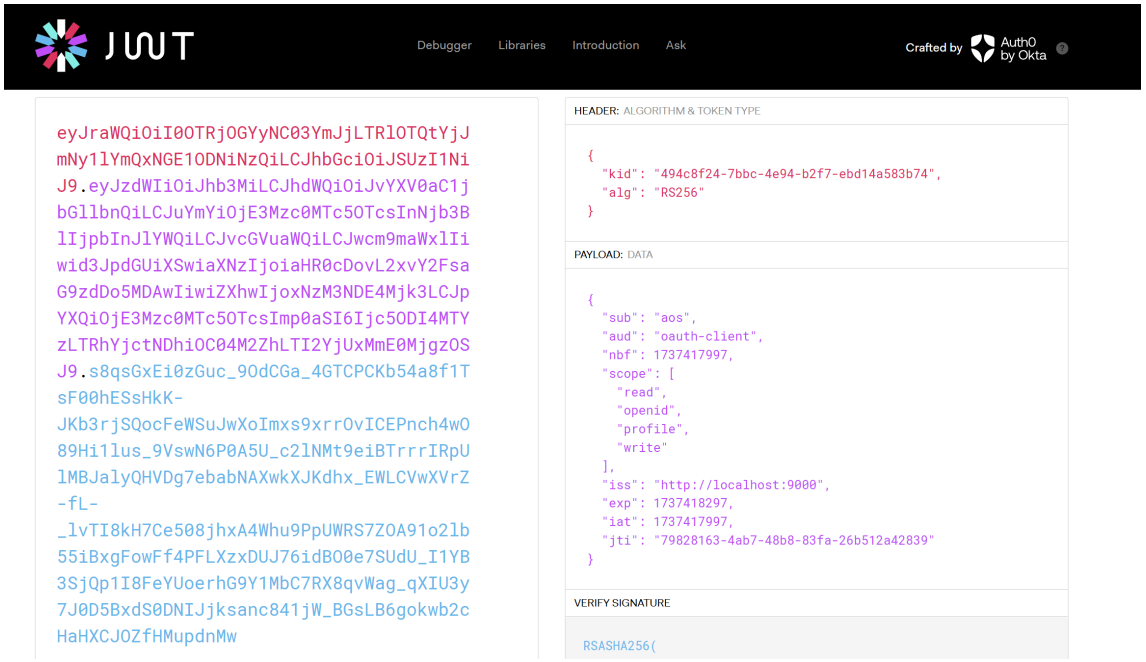

Figura 17: Comprobación del *token* en jwt.io.

de recursos, como se ve en la figura 18. Aquí se puede ver que este cliente contaba con varios *scopes* diferentes, lo que se ve reflejado en la respuesta del servidor de recursos (18), aunque ya se podía ver en la representación del JWT (17).

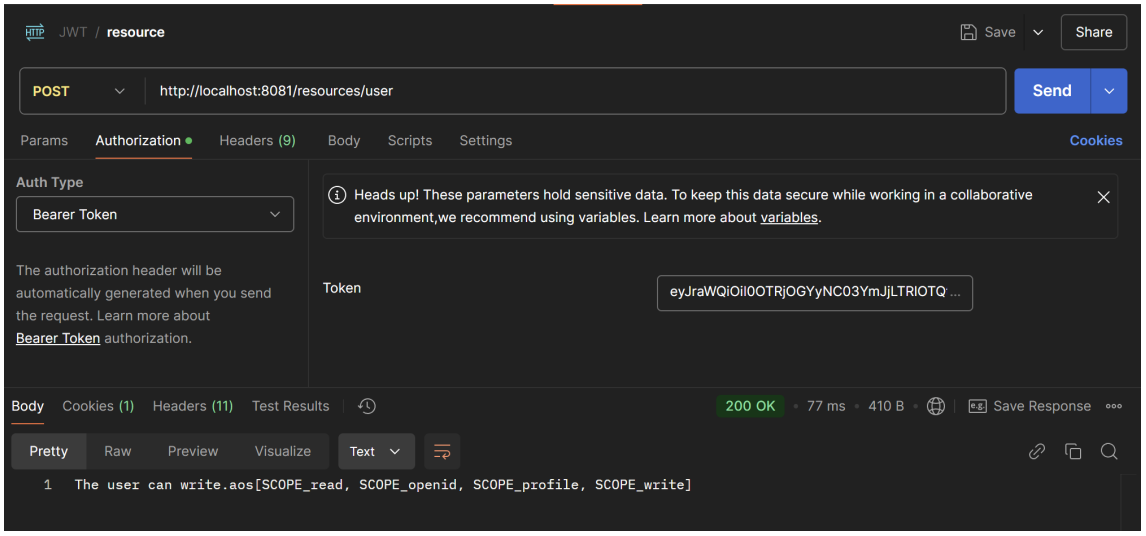

Figura 18: Petición POST a `http://localhost:8081/resources/user` con el *token* del cliente Spring Boot.

1.5. 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.

1.6. Referencias

- **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/

1.7. Anexo. Resumen peticiones en Postman

1.7.1. 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

1.7.2. 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

1.7.3. 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

1.8. 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 19: Dependencias comunes del `pom.xml` de los proyectos.

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

Listing 20: Dependencia del servidor de autorización

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

Listing 21: Dependencia del servidor de recursos

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

Listing 22: Dependencia del cliente OAuth

1.9. Anexo. Paquetes de los proyectos

1.9.1. 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 23: Paquetes del SecurityConfig del servidor de autorización.

1.9.2. 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 24: 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 25: Paquetes del ResourceController del servidor de recursos.

1.9.3. 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 26: 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 27: Paquetes del ClientController del cliente OAuth.