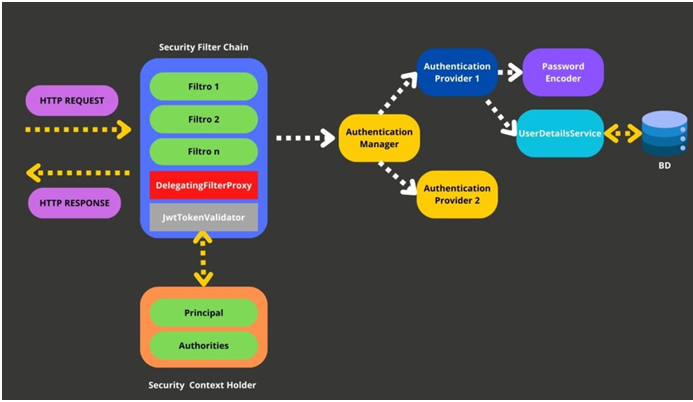
**Trabajo Práctico Nro. 7 Autenticación basada en JWT**

**Paso 0: Introducción**

 Para llevar a cabo la implementación de JWT vamos a agregar un nuevo filtro a nuestra cadena de filtros. Si nos guiamos por la arquitectura de Spring Security con la que venimos trabajando, se vería más o menos así:



Como podemos ver, nuestro filtro personalizado va a llamarse JwtTokenValidator y lo vamos a ir implementando paso a paso a continuación.

**Paso 1: Dependencias**

Como primer paso vamos a agregar las dependencias necesarias. Si accedemos a la documentación oficial de [JWT.io](https://jwt.io/libraries?language=Java) podemos ver que nos recomienda distintas librerías/bibliotecas para trabajar con JWT. Nosotros vamos a seleccionar una de las más usadas **Java JWT**.

Para incorporarla debemos agregar la siguiente dependencia a nuestro **pom.xml**:

<**dependency**>

<**groupId**>com.auth0</**groupId**>

<**artifactId**>java-jwt</**artifactId**>

<**version**>4.4.0</**version**>

</**dependency**>

Nota: Recordá que la versión puede variar en base al momento en que estés llevando a cabo este paso a paso. Si querés asegurarte de siempre usar la versión más actualizada, podés consultar la documentación oficial de [Java JWT](https://github.com/auth0/java-jwt)

**Paso 2: Paquete utils**

Crearemos un nuevo paquete en nuestro proyecto y lo llamaremos utils (nombre que se usa como estándar para las utilidades de una aplicación).

Una vez creado el paquete agregaremos una clase llamada JwtUtils y la marcaremos con la annotation **@Component**.

@Component

public class JwtUtils {…

**@Component:** es una de las anotaciones básicas en el framework Spring de Java. Indica que una clase es un componente de Spring y se debe cargar automáticamente en el ApplicationContext.

El **ApplicationContext** es una interfaz central dentro del framework Spring que proporciona un mecanismo para administrar la configuración de una aplicación Spring. Básicamente, es un contenedor que contiene y administra todos los componentes y beans de una aplicación Spring.

**Paso 3: Clave privada**

Para poder generar nuestros tokens necesitamos una clave privada para firmar los mismos. La finalidad de la **clave privada** es garantizar la integridad y autenticidad de los tokens JWT generados por el servidor.

Cuando se firma un token JWT con la clave privada, cualquier persona con acceso a la clave pública puede verificar la firma del token y estar seguro de que el token no ha sido manipulado o falsificado.

Nota: En este código se está usando el algoritmo HMAC256 para firmar el JWT. HMAC256 es un algoritmo simétrico, lo que significa que usa la misma clave secreta tanto para firmar como para verificar el token. No hay clave pública en este caso, solo una clave secreta compartida.

¿Cómo se verifica el token?

En el método `validateToken`, el token se verifica así:

Algorithm algorithm = Algorithm.HMAC256(this.privateKey);

JWTVerifier verifier = JWT.require(algorithm)

.withIssuer(this.userGenerator)

.build();

DecodedJWT decodedJWT = verifier.verify(token);

Aquí, `this.privateKey` es la clave secreta que utilizamos para firmar y verificar.

Si el token fue manipulado o la firma no coincide, lanzará una excepción.

Para generar nuestra clave privada vamos a buscar en Google el término «SHA generator key» y encontraremos diferentes opciones para generar claves SHA, vamos a utilizar particularmente esta: <https://tools.keycdn.com/sha256-online-generator>

**SHA Key:** Se refiere a una clave generada utilizando una función hash segura (Secure Hash Algorithm, SHA). La familia de algoritmos de hash SHA es ampliamente utilizada para generar valores hash únicos e irreversibles a partir de datos de entrada.

Vamos a colocar el contenido que queremos que tenga nuestra firma. En este caso vamos a probar ProgramacionSeguraCentro8 tal como se ve en el ejemplo (por supuesto, vos podes elegir la frase o código que quieras) y vamos a hacerlo mediante **SHA256**.



Una vez generada nuestra clave privada, vamos a incluirla en nuestro **application.properties**junto con un **user generator**.

El parámetro «user generator» va a ser el nombre de usuario que va a ser utilizado para la creación de tokens.

#Config de JWT

security.jwt.private.key=3dd76886d75471fcc8422c68a8980ab692d452b29424d6a481b49b2c8e46ce8e

#Acá puedo "inventar" el "nombre de usuario" que quiera

security.jwt.user.generator=CENTRO8-SEC

Nota**:**  no es una buena práctica dejar estos datos en texto plano en nuestro archivo, por lo que vamos a incluirlos en forma de variables de entorno.

#Config de JWT

security.jwt.private.key=${PRIVATE\_KEY}

#Acá puedo "inventar" el "nombre de usuario" que quiera

security.jwt.user.generator=${USER\_GENERATOR}

**Paso 4: Configuración JwtUtils**

Vamos a asociar dos variables en nuestra clase JwtUtils con los respectivos datos que generamos en el paso 3 en nuestro archivo application.properties de la siguiente manera:

import org.springframework.beans.factory.annotation.Value;

@Component

public class JwtUtils {

    //Con estas configuraciones aseguramos la autenticidad del token a crear

    @Value("${security.jwt.private.key}")

    private String privateKey;

    @Value("${security.jwt.user.generator}")

    private String userGenerator;

Nota**:** El **@Value** NO DEBE SER el de Lombok, sino el de **Spring** mediante **import org.springframework.beans.factory.annotation.Value;**

**Paso 5: Método para crear Tokens**

Dentro de la clase de utilidades de JWT que estamos armando vamos a crear ahora un método para crear nuestros Tokens mediante el siguiente código:

public String createToken (Authentication authentication) {

        Algorithm algorithm = Algorithm.HMAC256(this.privateKey);

        //esto está dentro del security context holder

        String username = authentication.getPrincipal().toString();

        //también obtenemos los permisos/autorizaciones

        //la idea es traer los permisos separados por coma

        String authorities = authentication.getAuthorities()

                .stream()

                .map(GrantedAuthority::getAuthority)

                .collect(Collectors.joining(","));

        //a partir de esto generamos el token

        String jwtToken = JWT.create()

                .withIssuer(this.userGenerator) //acá va el usuario que genera el token

                .withSubject(username) // a quien se le genera el token

                .withClaim("authorities", authorities) //claims son los datos contraidos en el JWT

                .withIssuedAt(new Date()) //fecha de generación del token

                .withExpiresAt(new Date(System.currentTimeMillis() + 1800000)) //fecha de expiración, tiempo en milisegundos

                .withJWTId(UUID.randomUUID().toString()) //id al token - que genere una random

                .withNotBefore(new Date (System.currentTimeMillis())) //desde cuando es válido (desde ahora en este caso)

                .sign(algorithm); //nuestra firma es la que creamos con la clave secreta

        return jwtToken;

    }

**Conceptos de programación funcional:**

1. **.stream()**: Este método se llama en el objeto Collection authentication.getAuthorities() para convertir la colección de autoridades en un Stream. Un Stream es una secuencia de elementos que permite realizar operaciones de forma secuencial o paralela.
2. **.map(GrantedAuthority::getAuthority)**: Se aplica la operación map en el Stream. La expresión GrantedAuthority::getAuthority es una referencia a un método. En este caso, se está llamando al método getAuthority() de cada objeto GrantedAuthority en el Stream. Esto transforma cada objeto GrantedAuthority en su autoridad (generalmente un rol) representado como una cadena de texto.
3. **.collect(Collectors.joining(","))**: Finalmente, se usa collect para combinar todos los elementos del Stream en una sola cadena. Collectors.joining(",") es un colector que concatena los elementos del Stream usando una coma como delimitador.

Estos **métodos de Streams** y referencias a métodos permiten una forma más concisa y funcional de trabajar con colecciones en Java, evitando bucles tradicionales y facilitando la escritura de código más legible y mantenible.

**Paso 6: Método para validar y decodificar tokens**

Dentro de la clase de utilidades de JWT que estamos armando también crearemos un método para decodificar y validar nuestros tokens que generemos mediante el siguiente código:

public DecodedJWT validateToken(String token) {

        try {

            Algorithm algorithm = Algorithm.HMAC256(this.privateKey); //algoritmo + clave privada

            JWTVerifier verifier = JWT.require(algorithm)

                    .withIssuer(this.userGenerator)

                    .build(); //usa patrón builder

            //si está todo ok, no genera excepción y hace el return

            DecodedJWT decodedJWT = verifier.verify(token);

             return decodedJWT;

        }

        catch (JWTVerificationException exception) {

            throw new JWTVerificationException("Invalid token. Not authorized");

        }

    }

1. **Algorithm algorithm = Algorithm.HMAC256(this.privateKey);**: Se crea un objeto Algorithm utilizando el algoritmo HMAC con SHA-256 y la clave privada proporcionada. Esta clave privada se utiliza para firmar y verificar el token JWT. La clase Algorithm es parte de la librería com.auth0:java-jwt y proporciona métodos para crear algoritmos de encriptación para JWT.
2. **JWTVerifier verifier = JWT.require(algorithm)**: Se crea un objeto JWTVerifier utilizando el patrón de constructor de tipo Builder. Este verificador se configura para requerir un algoritmo específico (el que acabamos de crear) para la verificación del token JWT.
3. **.withIssuer(this.userGenerator)**: Se especifica el emisor esperado del token JWT. El valor this.userGenerator hace referencia al usuario que genera el token.
4. **.build();**: Se finaliza la construcción del verificador JWT. Este método es común en patrones de diseño Builder. Después de configurar todas las opciones necesarias, se llama a este método para crear el objeto JWTVerifier listo para usar en la validación de tokens.

**Paso 7: Método para obtener nombre de usuario del token**

Dentro de la clase de utilidades de JWT que estamos armando también crearemos un método para obtener el nombre de usuario (username) de nuestro JWT en caso de que lo necesitemos:

public String extractUsername (DecodedJWT decodedJWT) {

        //el subject es el usuario según establecimos al crear el token

        return decodedJWT.getSubject().toString();

    }

**Paso 8: Método para obtener claims**

Dentro de la clase de utilidades de JWT que estamos armando también crearemos un método para obtener los valores de los **claims** (propiedades) que hayamos establecido en nuestro JWT.

*Vamos a crear dos métodos:*

1. Uno para obtener un claim específico
2. Otro para obtener todos los claims

//devuelvo un claim en particular

    public Claim getSpecificClaim (DecodedJWT decodedJWT, String claimName) {

        return decodedJWT.getClaim(claimName);

    }

    //devuelvo todos los claims

    public Map<String, Claim> returnAllClaims (DecodedJWT decodedJWT){

        return decodedJWT.getClaims();

    }

**Paso 9: Creación del JWT Filter**

Luego de estas configuraciones en nuestra clase utils, ya estamos en condiciones de empezar a crear nuestro propio filtro para JWT. Para ello vamos a crear, dentro del paquete config que creamos anteriormente, un nuevo paquete llamado filter.

Dentro de este paquete crearemos una nueva clase llamada **JwtTokenValidator** (el filtro que identificamos en el Paso 0 que íbamos a agregar) que contenga el siguiente código:

public class JwtTokenValidator extends OncePerRequestFilter {

    private JwtUtils jwtUtils;

    public JwtTokenValidator(JwtUtils jwtUtils) {

        this.jwtUtils = jwtUtils;

    }

    @Override

    //importante: el nonnull debe ser de sringframework, no lombok

    protected void doFilterInternal(@NonNull HttpServletRequest request,

                                    @NonNull HttpServletResponse response,

                                    @NonNull FilterChain filterChain) throws ServletException, IOException {

            String jwtToken = request.getHeader(HttpHeaders.AUTHORIZATION);

            if(jwtToken != null) {

                //en el encabezado antes del token viene la palabra bearer (esquema de autenticación)

                //por lo que debemos sacarlo

                 jwtToken = jwtToken.substring(7); //son 7 letras + 1 espacio

               DecodedJWT decodedJWT = jwtUtils.validateToken(jwtToken);

               //si el token es válido, le concedemos el acceso

                String username = jwtUtils.extractUsername(decodedJWT);

                //me devuelve claim, necesito pasarlo a String

                String authorities = jwtUtils.getSpecificClaim(decodedJWT, "authorities").asString();

                //Si todo está ok, hay que setearlo en el Context Holder

                //Para eso tengo que convertirlos a GrantedAuthority

                Collection<? extends GrantedAuthority> authoritiesList = AuthorityUtils.commaSeparatedStringToAuthorityList(authorities);

                //Si se valida el token, le damos acceso al usuario en el context holder

                SecurityContext context = SecurityContextHolder.getContext();

                Authentication authentication = new UsernamePasswordAuthenticationToken(username, null, authoritiesList);

                context.setAuthentication(authentication);

                SecurityContextHolder.setContext(context);

            }

            // si no viene token, va al siguiente filtro

            //si no viene el token, eso arroja error

            filterChain.doFilter(request,response);

    }

}

**¿Qué hace éste código?:**Básicamente el filtro JwtTokenValidator que creamos mediante este código se encarga de validar y procesar los tokens JWT en las solicitudes HTTP, extrayendo información de autenticación del token y configurando la autenticación en el contexto de seguridad de Spring.

**Paso 10: Agregar filtro a la FilterChain**

Ahora debemos agregar el filtro que acabamos de crear dentro de nuestra cadena de filtros en la clase SecurityConfig, para ello haremos:

* Inyectamos en el método securityFilterChain la dependencia a jwtUtils:

@Autowired

private JwtUtils jwtUtils;

* Luego, dentro de los filtros y antes del **.build**, agregamos lo siguiente:

         .addFilterBefore(new JwtTokenValidator(jwtUtils), BasicAuthenticationFilter.class)

**Paso 11: Creamos un controller de autenticación**

Para poder validar el inicio de sesión de un usuario y que éste obtenga su correspondiente token mediante JWT vamos a crear un controlador, con sus respectivos end-points para esta tarea:

package ar.edu.centro8.ps.jwt.controller;

import ar.edu.centro8.ps.jwt.dto.AuthLoginRequestDTO;

import ar.edu.centro8.ps.jwt.dto.AuthResponseDTO;

import ar.edu.centro8.ps.jwt.service.UserDetailsServiceImp;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.http.HttpStatus;

import org.springframework.http.ResponseEntity;

import org.springframework.web.bind.annotation.PostMapping;

import org.springframework.web.bind.annotation.RequestBody;

import org.springframework.web.bind.annotation.RequestMapping;

import org.springframework.web.bind.annotation.RestController;

import jakarta.validation.Valid;

@RestController

@RequestMapping("/auth")

public class AuthenticationController {

    @Autowired

    private UserDetailsServiceImp userDetailsService;

    //Todas estas requests y responses vamos a tratarlas como dto

    @PostMapping("/login")

    public ResponseEntity<AuthResponseDTO> login(@RequestBody @Valid AuthLoginRequestDTO userRequest) {

        return new ResponseEntity<>(this.userDetailsService.loginUser(userRequest), HttpStatus.OK);

    }

}

En este código el método: **public ResponseEntity login(@RequestBody @Valid AuthLoginRequestDTOuserRequest)**maneja la solicitud de inicio de sesión. Devuelve una **ResponseEntity** y acepta un cuerpo de solicitud (**@RequestBody**) que se debe mapear a un objeto **AuthLoginRequestDTO** (que crearemos más tarde. La anotación **@Valid** indica que el objeto debe ser validado antes de que el método procese la solicitud.

Por otro lado, return new **ResponseEntity<>(this.userDetailsService.loginUser(userRequest), HttpStatus.OK);** crea y devuelve una nueva **ResponseEntity** con el resultado de **userDetailsService.loginUser(userRequest)** y un estado **HTTP 200 OK**.

Por su parte **this.userDetailsService.loginUser(userRequest)** es una llamada al método **loginUser** en el servicio **UserDetailsServiceImp**, que procesa la lógica de inicio de sesión y devuelve el resultado (puede ser un token **JWT** o información de usuario autenticado).

*Para que este código funcione correctamente, debemos tener la siguiente librería agregada:*

<dependency>

            <groupId>jakarta.validation</groupId>

            <artifactId>jakarta.validation-api</artifactId>

            <version>3.1.1</version>

        </dependency>

**Paso 12: Creación de los DTO**

Una vez listo nuestro controller, procederemos a crear los DTO que mencionamos en el. Para ello, crearemos un nuevo paquete dto en nuestro proyecto y dentro dos clases: **AuthResponseDTO** y **AuthLoginRequestDTO** de la siguiente manera:

package ar.edu.centro8.ps.jwt.dto;

import com.fasterxml.jackson.annotation.JsonPropertyOrder;

/\*Cuando una clase se declara como un registro, el compilador de Java genera automáticamente

ciertos métodos como el constructor, los métodos equals(), hashCode() y toString(),

basados en los componentes de datos declarados en la clase.\*/

@JsonPropertyOrder({"username", "message", "jwt", "status"})

public record AuthResponseDTO (String username, String message, String jwt, boolean status) {

}

**Uso de Record**

Como podemos ver, hacemos uso de un nuevo concepto que es la palabra reservada **record**.

En Java, un record es una característica introducida en Java 14 como parte de un proyecto más amplio para simplificar la creación de clases que son principalmente contenedores de datos.

Un **record** automáticamente proporciona muchas de las funcionalidades básicas que usualmente se requieren en una clase de datos, como constructores, getters, equals, hashCode, y toString, reduciendo así la cantidad de código boilerplate (el término «boilerplate» en programación se refiere a las secciones de código que deben escribirse en múltiples lugares con poca o ninguna variación).

Un record es una clase especial que actúa como un contenedor de datos inmutable. Cuando declaras un record, el compilador de Java automáticamente genera:

1. Un constructor público que toma todos los parámetros especificados en la declaración del record.
2. Métodos de acceso (getter) para cada campo.
3. Implementaciones de equals, hashCode, y toString.
4. Los campos privados y finales para almacenar los datos.

**Uso de Jackson**

Jackson es una biblioteca popular para la serialización y deserialización de objetos en Java. Esto significa que se puede utilizar para convertir objetos Java en formatos de texto como JSON (JavaScript Object Notation) y viceversa.

La anotación **@JsonPropertyOrder** se aplica a nivel de clase y toma un array de nombres de propiedades en el orden en que querés que aparezcan en el JSON.

**Paso 13: Creación de método login en userDetailsService**

Como vimos en el controller, tenemos un llamado a un método login en nuestro userDetailsService que todavía no implementamos, por lo cual, vamos a implementarlo ahora mediante el siguiente código:

public AuthResponseDTO loginUser (AuthLoginRequestDTO authLoginRequest){

        //recuperamos nombre de usuario y contraseña

        String username = authLoginRequest.username();

        String password = authLoginRequest.password();

        Authentication authentication = this.authenticate (username, password);

        //si todo sale bien

        SecurityContextHolder.getContext().setAuthentication(authentication);

        String accessToken =jwtUtils.createToken(authentication);

        AuthResponseDTO authResponseDTO = new AuthResponseDTO(username, "login ok", accessToken, true);

        return authResponseDTO;

}

A partir de esto, creamos el método **authenticate** al que hacemos referencia, mediante el siguiente código:

public Authentication authenticate (String username, String password) {

        //con esto debo buscar el usuario

        UserDetails userDetails = this.loadUserByUsername(username);

        if (userDetails==null) {

            throw new BadCredentialsException("Ivalid username or password");

        }

        // si no es igual

        if (!passwordEncoder.matches(password, userDetails.getPassword())) {

            throw new BadCredentialsException("Invalid password");

        }

        return new UsernamePasswordAuthenticationToken(username, userDetails.getPassword(), userDetails.getAuthorities());

}

Con todo esto ya configurado, estamos listos para realizar las pruebas.

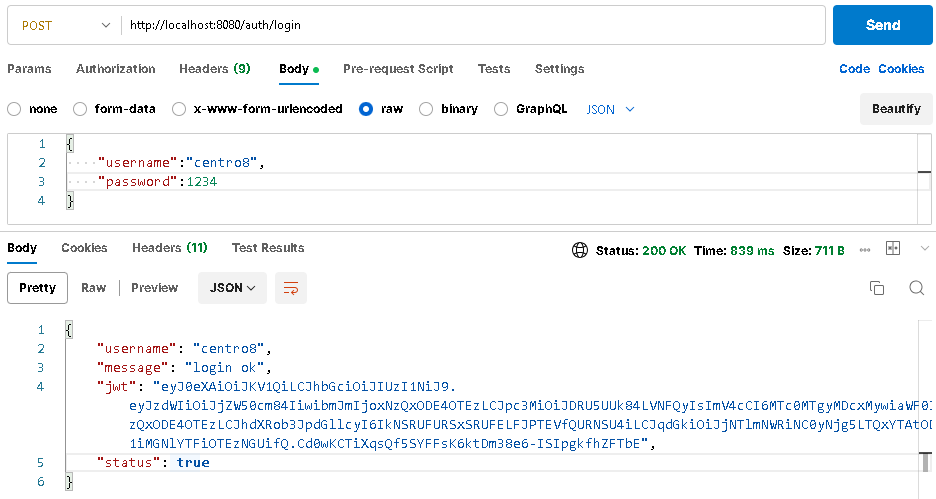
**Paso 14: Pruebas**

Para corroborar la correcta configuración de todo lo que logramos a partir de este paso a paso vamos a llevar a cabo 2 pruebas.

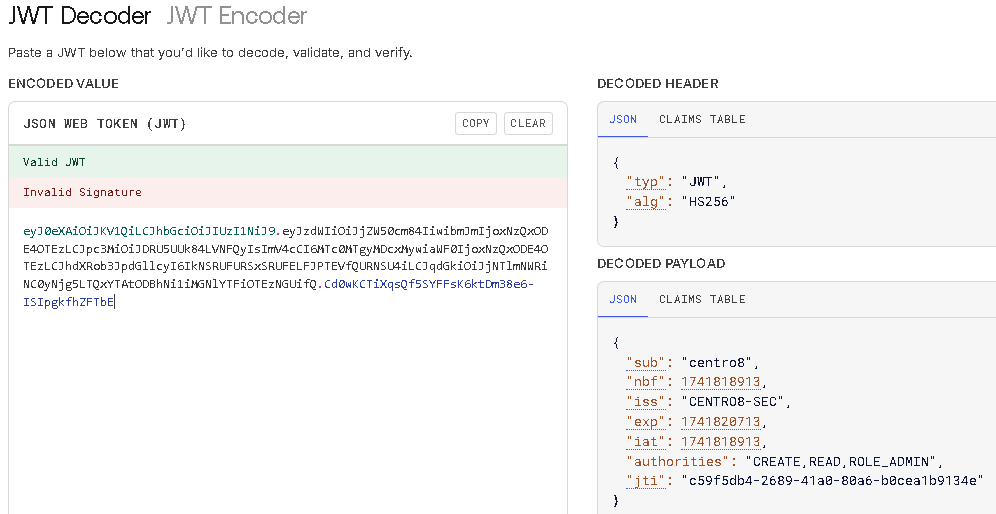
**Prueba Nº 1: Login y obtención de token**

Vamos a probar el controller de login que acabamos de desarrollar para ver si obtenemos nuestro JWT como respuesta.

Para ello vamos a hacer lo siguiente en postman:



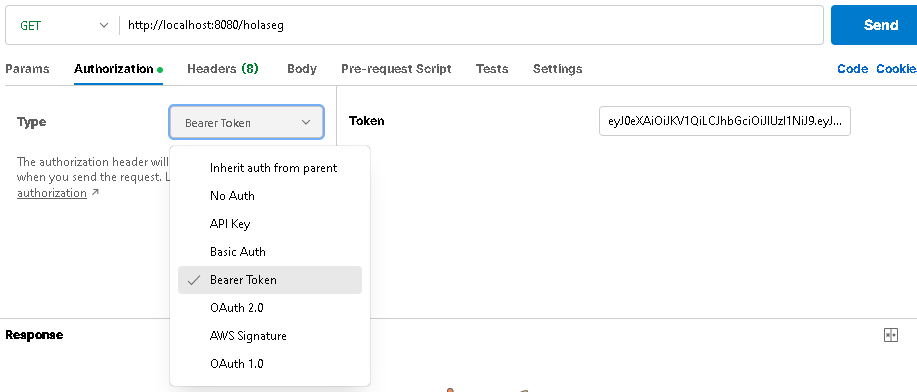
Como podemos ver, si nuestro usuario y contraseña están ok, obtenemos como respuesta nuestro token correspondiente de inicio de sesión. Si lo copiamos en [JWT.io](https://jwt.io/) tendremos como resultado esto:



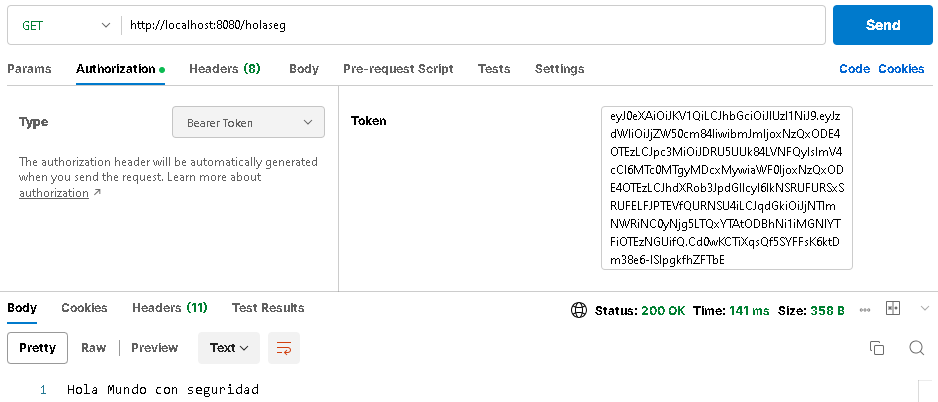
Podemos observar que están nuestros claims.

**Prueba Nº 2: Probando acceso a endpoints securizados mediante JWT**

Vamos ahora a probar ingresar a nuestros endpoints securizados mediante JWT. Para ello, en Postman, en lugar de seleccionar **Basic Auth** en el apartado “**Authorization**” vamos a elegir **Bearer Token**:



Una vez hecho esto, vamos a colocar el token que nos devolvió nuestro login:



Si todo sale bien, y el usuario está autorizado, obtendremos el acceso sin problema alguno.