

**Proyecto:**

SEGURIDAD CON SPRING BOOT Y APLICACIONES WEB

**Integrantes:**

U201723461-Nelson Dennis Sánchez Esparza

U201716528-Sandro Enrique Hurtado León

**Profesor:**

Carlos Alberto Flores Orihuela

Índice

[1 Seguridad con Spring Boot y Aplicaciones Web 3](#_Toc120598391)

[1.1 Describir la Seguridad de Autorización y Autenticación basada en Tokens 3](#_Toc120598392)

[1.2 ¿Qué es OAuth2.0? 5](#_Toc120598393)

[1.3 CORS: qué es y cómo funciona el cross-origin resource sharing 9](#_Toc120598394)

[1.4 Estructura de la CORS header 10](#_Toc120598395)

[1.5 ¿Qué es la autorización, Roles? 13](#_Toc120598396)

[1.6 Ejemplo de Protección de API REST con JWT FrontEnd y Backend 14](#_Toc120598397)

[1.6.1 Desarrollo Backend 14](#_Toc120598398)

[1.6.2 Derrollo Frontend 14](#_Toc120598399)

[1.7 Pruebas con Postman y Angular Frontend 15](#_Toc120598400)

[1.7.1 El servicio responde status 401 15](#_Toc120598401)

[1.7.2 El servicio responde con status 200 15](#_Toc120598402)

[1.7.3 Servicio que genera el token 16](#_Toc120598403)

[1.8 ¿Qué es lo que más les ha impresionado o gustado? 16](#_Toc120598404)

[1.9 Conclusiones y Recomendaciones 16](#_Toc120598405)

[1.10 Bibliografía 17](#_Toc120598406)

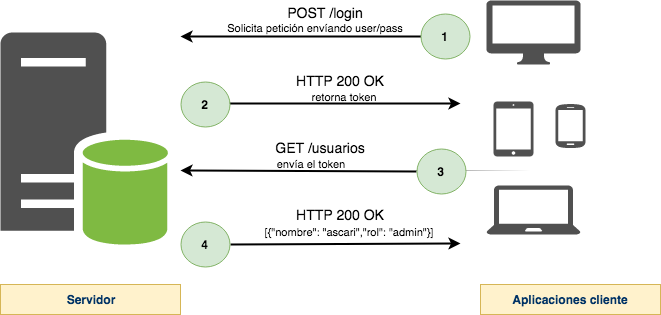
# Seguridad con Spring Boot y Aplicaciones Web

## Describir la Seguridad de Autorización y Autenticación basada en Tokens

Una de las tendencias en el desarrollo web moderno es tener un API RESTful como back-end y como front-end una aplicación desarrollada en Angular 2, además que mediante un API RESTful podemos desarrollar también aplicaciones móviles que utilicen estos mismos servicios.

La autenticación entre el API RESTful y sus consumidores es conveniente realizarla mediante tokens, específicamente usando el estándar JSON Web Token. La autenticación basada en tokens proporciona varias ventajas de las cuales no hablaremos aquí.

El siguiente diagrama muestra el flujo general de un proceso de autenticación basada en token.



1. El cliente envía sus credenciales (usuario y password) al servidor.
2. Si las credenciales son válidas, el servidor devuelve al cliente un token de acceso.
3. El cliente solicita un recurso protegido. En la petición, se envía el token de acceso.
4. El servidor valida el token y en caso de ser válido, devuelve el recurso solicitado.

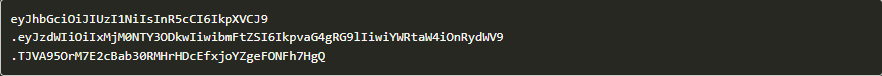
**JSON Web Token (JWT)**

Es un estándar abierto (RFC 7519) que define un modo compacto y autónomo para transmitir de forma segura la información entre las partes como un objeto JSON. Esta información puede ser verificada y es confiable porque está firmada digitalmente. Los JWT se pueden firmar usando un secreto (con el algoritmo HMAC) o utilizando un par de claves públicas / privadas usando RSA.

Consta de 3 partes separadas por un punto ( . )

1. Header
2. Payload
3. Signature

Cada una de estas partes se codifica en base64 de tal forma que el token generado tiene una apariencia como esta:



**Header**

El header consta de dos partes, el tipo de token y el algoritmo de hash.



Si al JSON anterior lo codificamos en base64 tendremos nuestra primera parte del JWT.

**Payload**

El payload contiene datos como: iss (issuer), exp (expiration time) y sub (subject).

1. iss es quien emitió el token
2. exp contiene la fecha de expiración del token
3. sub indica el usuario del token

Además, podemos indicar otros campos como el nombre, roles, etc.

Forma

Descripción generada automáticamente con confianza media

Al codificar el json anterior en base 64, obtenemos la segunda parte de nuestro JWT.

**Signature**

Para crear la signature, se debe tomar el header codificado, el payload codificado, una palabra o clave secreta, el algoritmo especificado en el header y firmarlo.

Por ejemplo, si desea utilizar el algoritmo HMAC SHA256, la signature se creará de la siguiente manera:

Un conjunto de letras blancas en un fondo blanco

Descripción generada automáticamente con confianza media

La signature se usa para verificar que el mensaje no se modificó en el camino y, en el caso de tokens firmados con una clave privada, también puede verificar que el remitente del JWT es quien dice ser.

## ¿Qué es OAuth2.0?

OAuth 2.0 es un estándar abierto para la autorización de APIs, que nos permite compartir información entre sitios sin tener que compartir la identidad.

Es un mecanismo utilizado hoy en día por grandes compañías como Google, Facebook, Microsoft, Twitter, GitHub o LinkedIn, entre otras muchas.

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

**Por qué surge OAuth**

Surge para paliar la necesidad que se establece del envío continuo de credenciales entre cliente y servidor.

Si nos planteamos desarrollar una API REST y una aplicación cliente que pueda consumir de nuestros servicios, si queremos tener un mínimo de mecanismos de seguridad, antes de OAuth2 necesitábamos que el usuario nos enviara las credenciales.

Si la aplicación cliente es nuestra y la API REST también, posiblemente no existan grandes dificultades. El problema está cuando se quiere hacer una integración con aplicaciones de terceros, ahí es donde aparece la dificultad.

Por ejemplo, si queremos que nuestra aplicación sea capaz de generar contenido para Twitter y que aparezca en el timeline de un usuario, tendríamos que pedirle el usuario y contraseña al mismo para poder hacer este proceso, y la verdad es que no es algo razonable ni aceptable.

Con OAuth2 el usuario delega la capacidad de realizar ciertas acciones, no todas, a las cuales da su consentimiento para hacerlas en su nombre.

Podríamos delegar en una aplicación como Runtastic, la posibilidad de escribir por nosotros en Twitter cada vez que terminemos una carrera. De esta forma, la aplicación no tiene por qué tener nuestras credenciales de Twitter, pero le permitiríamos hacer algo en Twitter por nosotros.

De esta forma, cuando desarrollamos una aplicación no tenemos por qué almacenar el nombre de usuario y contraseña de este, sin embargo, vamos a poder hacer algún determinado conjunto de acciones en su nombre.

**Roles**

Dentro de OAuth 2.0 encontramos diferentes roles que van a participar en el proceso:

* Dueño del recurso (Owner).
* Cliente (Client).
* Servidor de recursos protegidos (Resource Server).
* Servidor de autorización (Authorization Server).

**Dueño del recurso**

El propietario del recurso es el usuario que da autorización a una determinada aplicación para acceder a su cuenta y poder hacer algunas cosas en su nombre.

El conjunto de cosas que puede hacer la aplicación en su nombre se denomina alcance, y podría ser, por ejemplo, solamente acceso de lectura y no poder crear ningún tipo de elemento de ningún nuevo recurso en su nombre.

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

Se le llama dueño del recurso porque, si bien la API no es suya, los datos que maneja sí lo son.

**Cliente**

El cliente sería la aplicación que desea acceder a esa cuenta de usuario.

Antes de que pueda hacerlo debe ser autorizada por el usuario, y dicha autorización debe ser validada por la API.

El cliente puede ser una aplicación web, una aplicación móvil, una aplicación de escritorio, una aplicación para smartTV, un dispositivo de IoT, otra API que consumiera de esta API, etcétera.

**Servidor de autorización**

El servidor de autorización es el responsable de gestionar las peticiones de autorización.

Verifica la identidad de los usuarios y emite una serie de tokens de acceso a la aplicación cliente.

En muchas ocasiones, podemos implementar el servidor de autorización nosotros mismos, o podemos delegar en un tercero (Facebook, Twitter, Github, Google, etcétera) y tener solamente un servidor de recursos. Incluso podemos desarrollar el cliente y delegar la autorización en un tercero.

**Servidor de recursos**

El servidor de recursos será la API propiamente, es decir, el servidor que aloja el recurso protegido al cual queremos acceder.

Puede que también forme parte de la misma aplicación que el propio servidor de autenticación.

**Flujo de OAuth2**

El flujo más o menos genérico de este protocolo el que vemos en la siguiente imagen:



Una aplicación cliente, ya sea otra API o cualquier otra aplicación, quiere acceder a un recurso, y para ello lo que hace es pedir autorización al dueño del recurso. Si todo va bien, este se la concede.

Al concederle la autorización pasaría al servidor de autorización para obtener el token de acceso, una vez obtenido el mismo, podríamos ir al servidor de recursos y obtener el recurso protegido.

Como ya comentamos antes, puede darse el caso que el servidor de autorización y el servidor de recursos no sean de aplicaciones separadas, sino que estén dentro del mismo servidor.

## CORS: qué es y cómo funciona el cross-origin resource sharing

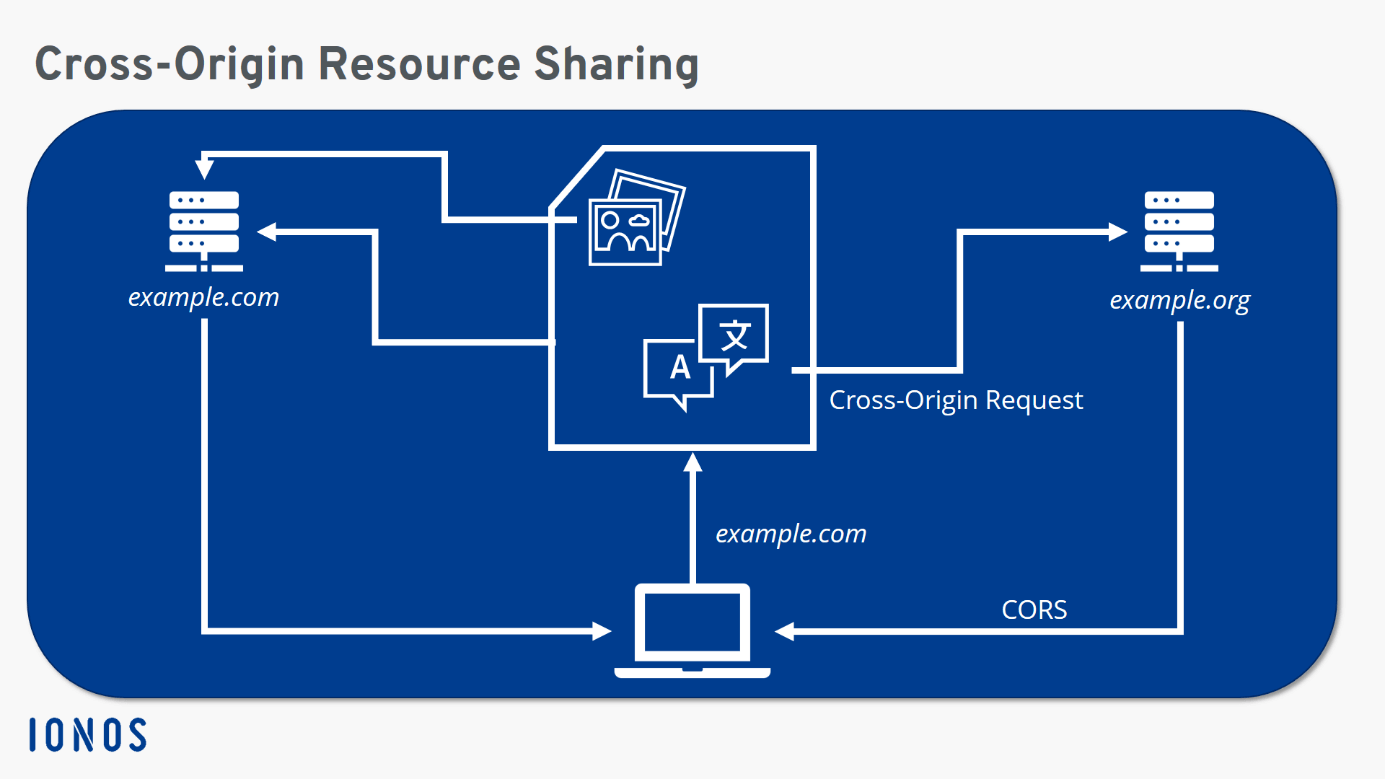
Cuando se abre una página web, cargar datos de servidores ajenos está, en teoría, estrictamente prohibido. Sin embargo, puede haber excepciones: si los administradores de ambas webs han acordado trabajar juntos, no hay por qué evitar el intercambio. En estos casos, el llamado cross-origin resource sharing (CORS) regula la colaboración.

**¿Cómo funciona el CORS?**

La same-origin policy (SOP o política de seguridad del mismo origen) prohíbe que se carguen datos de servidores ajenos al acceder a una página web. Todos los datos deben provenir de la misma fuente, es decir, corresponder al mismo servidor. Se trata de una medida de seguridad, ya que JavaScript y CSS podrían cargar, sin que el usuario lo supiese, contenido de otros servidores (y, con este, también contenido malicioso). Tales intentos son denominados “cross-origin requests”. Si, por el contrario, ambos administradores web saben del intercambio de contenido y lo aprueban, no tiene sentido impedir este proceso. El servidor solicitado (es decir, aquel del que se quiere cargar contenido) puede permitir entonces el acceso mediante cross-origin resource sharing, en castellano, intercambio de recursos de origen cruzado.

Este permiso se da, no obstante, únicamente a clientes concretos, es decir, el CORS no es un comodín para realizar cualquier cross-origin request. En lugar de eso, el segundo servidor permite al primero un acceso exclusivo mediante una cabecera HTTP. En dicha cabecera de la respuesta HTTP está indicado específicamente qué servidores pueden cargar datos y ponerlos a disposición del usuario. El acceso generalizado a todos los clientes se permite únicamente mediante una “wildcard” o certificado comodín. Esta solución, sin embargo, solo es conveniente para servidores cuyo contenido debe estar a disposición del público general, como es el caso, por ejemplo, de las tipografías web.

Si todo sale bien, el usuario no se percatará en absoluto del intercambio entre ambos servidores. Todos los navegadores actuales soportan el CORS, y el envío de solicitudes y respuestas sucede rápidamente al solicitar una página web sin que el usuario lo note.



## Estructura de la CORS header

De acuerdo con la política de seguridad del mismo origen, en una conexión entre servidores, los datos referentes al origen se componen de tres elementos: ***host, puerto y protocolo***. De este modo, y tomando el ejemplo de la imagen, la directriz prohíbe que ’https://example.com’ acceda a ’http://example.com’ o a ’https://example.org’. En el primer caso, el protocolo no es el mismo y, en el segundo, los datos de host no coinciden.

Una petición de origen cruzado es, en teoría, una petición HTTP. Los métodos específicos no suelen dar problemas. GET y HEAD no pueden alterar datos y, por lo tanto, no suelen considerarse como un riesgo para la seguridad. No se puede decir lo mismo de PATCH, PUT o DELETE: con ellos sí se puede llevar a cabo acciones maliciosas, por lo que en estos casos también hay que activar el cross-origin resource sharing, ya que CORS no solo puede contener información sobre el origen permitido, sino también acerca de qué peticiones HTTP están permitidas por la fuente.

Si se trata de métodos HTTP de seguridad, el cliente envía en primer lugar una solicitud preflight (preflight request) en la que solo se indica qué método HTTP se piensa transmitir al servidor a continuación y se pregunta si la solicitud es considerada segura. Para ello, se usa la cabecera OPTIONS (OPTIONS header). Una vez se haya recibido una respuesta positiva, ya se puede realizar la solicitud propiamente dicha.

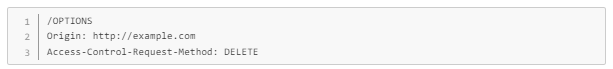
Existen diferentes cabeceras o CORS headers y cada una aborda un aspecto distinto. Ya hemos mencionado dos cabeceras importantes para identificar orígenes seguros y métodos permitidos, pero hay más:

* Access-Control-Allow-Origin: ¿qué origen está permitido?
* Access-Control-Allow-Credentials: ¿también se aceptan solicitudes cuando el modo de credenciales es incluir (include)?
* Access-Control-Allow-Headers: ¿qué cabeceras pueden utilizarse?
* Access-Control-Allow-Methods: ¿qué métodos de petición HTTP están permitidos?
* Access-Control-Expose-Headers: ¿qué cabeceras pueden mostrarse?
* Access-Control-Max-Age: ¿cuándo pierde su validez la solicitud preflight?
* Access-Control-Request-Headers: ¿qué header HTTP se indica en la solicitud preflight?
* Access-Control-Request-Method: ¿qué método de petición HTTP se indica en la solicitud preflight?
* Origin: ¿de qué origen proviene la solicitud?

El primer header es especialmente importante, ya que especifica desde qué otro host se puede acceder al servidor solicitado. Además de una dirección concreta, en dicha cabecera también se puede incluir una wildcard en forma de asterisco. De esta manera, el servidor permitirá cross-origin requests de cualquier origen.

**Ejemplo de cross-origin resource sharing**

En el siguiente ejemplo, suponemos que el host A (example.com) quiere enviar una petición DELETE (DELETE request) al host B (example.org). Para ello, el servidor A envía, en primer lugar, una preflightrequest:



Si el host B no se opone a esta cross-origin request, responderá con los CORS headers correspondientes:



Si las cabeceras de la respuesta no correspondieran con las especificaciones de la solicitud, o si el servidor solicitado no respondiese, la cross-origin request no se podría realizar.

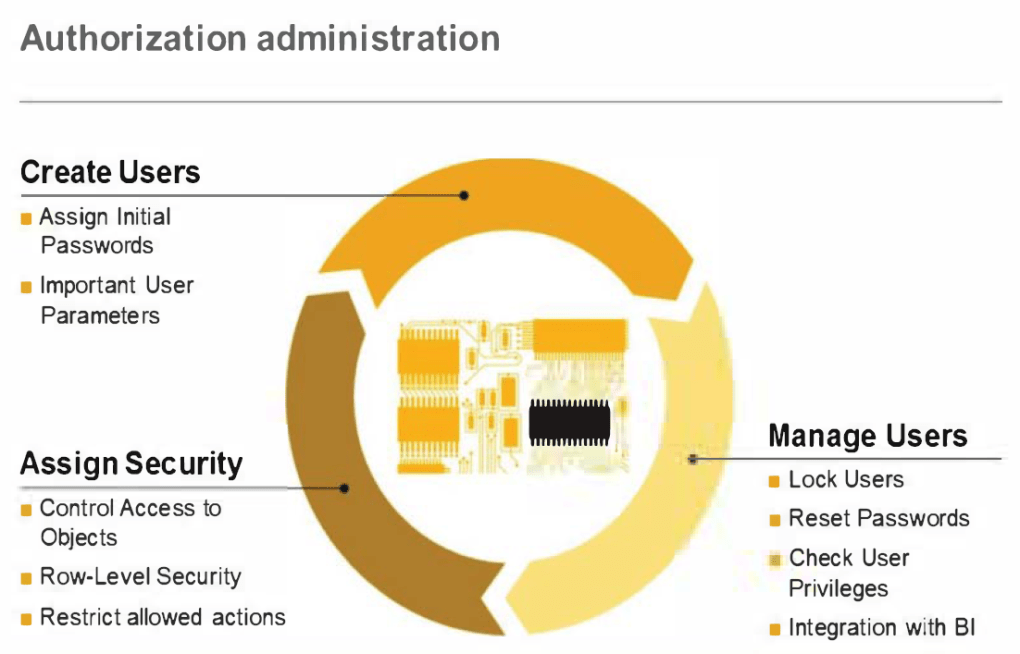
**Ventajas e inconvenientes del CORS**

El propósito del CORS es eludir la medida de seguridad establecida como configuración predeterminada (la política de seguridad del mismo origen). Dicha política es, de hecho, un medio muy eficaz para bloquear conexiones potencialmente peligrosas. Internet, sin embargo, se basa a menudo precisamente en este tipo de cross-origin requests, ya que muchas de las conexiones entre hosts sí son deseadas.

Por eso, el CORS ofrece una solución intermedia, permitiendo hacer excepciones a la prohibición en aquellas situaciones en que las solicitudes de origen cruzado son expresamente requeridas. No obstante, se corre el riesgo de que los administradores web se aprovechen de las wildcards por comodidad, haciendo que la protección de la SOP sea en vano. Por eso, es importante utilizar el CORS solo en casos especiales y configurarlo de la manera más restrictiva posible.

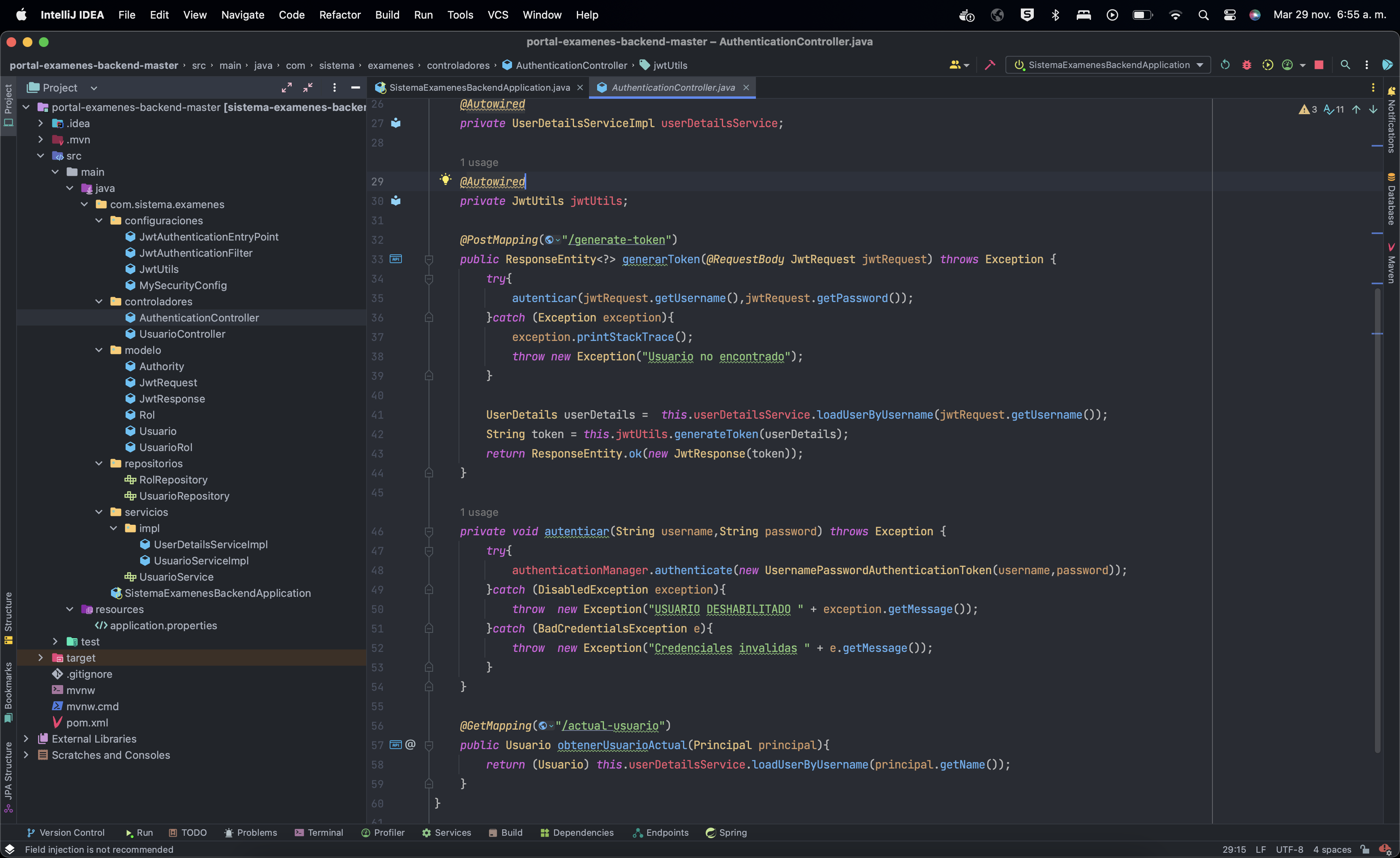
## ¿Qué es la autorización, Roles?

El role based access control, cuyas siglas son RBAC, se traduce como “control de acceso basado en roles”. Este modelo de seguridad permite asignar funciones y autorizaciones en la infraestructura informática de una organización. El término “basado en roles” es clave para entender cómo funciona el RBAC, ya que lo distingue de otros conceptos de seguridad, como el mandatory access control. En este modelo, el administrador del sistema asigna un nivel y una categoría de seguridad a cada usuario y objeto en dependencia del rol que cumple. El sistema operativo enlaza automáticamente los dos niveles y luego concede o deniega el acceso.



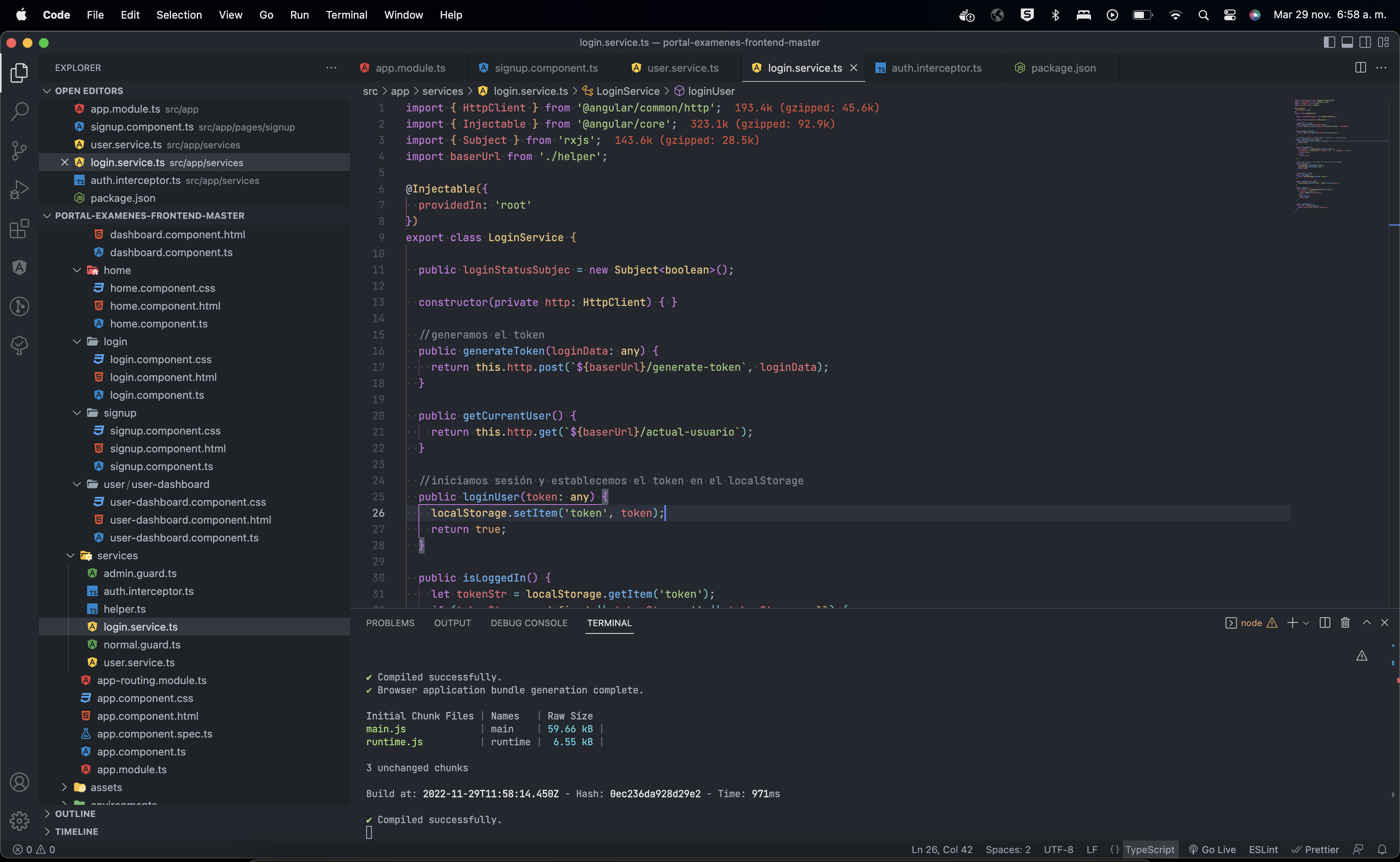
## Ejemplo de Protección de API REST con JWT FrontEnd y Backend

### Desarrollo Backend



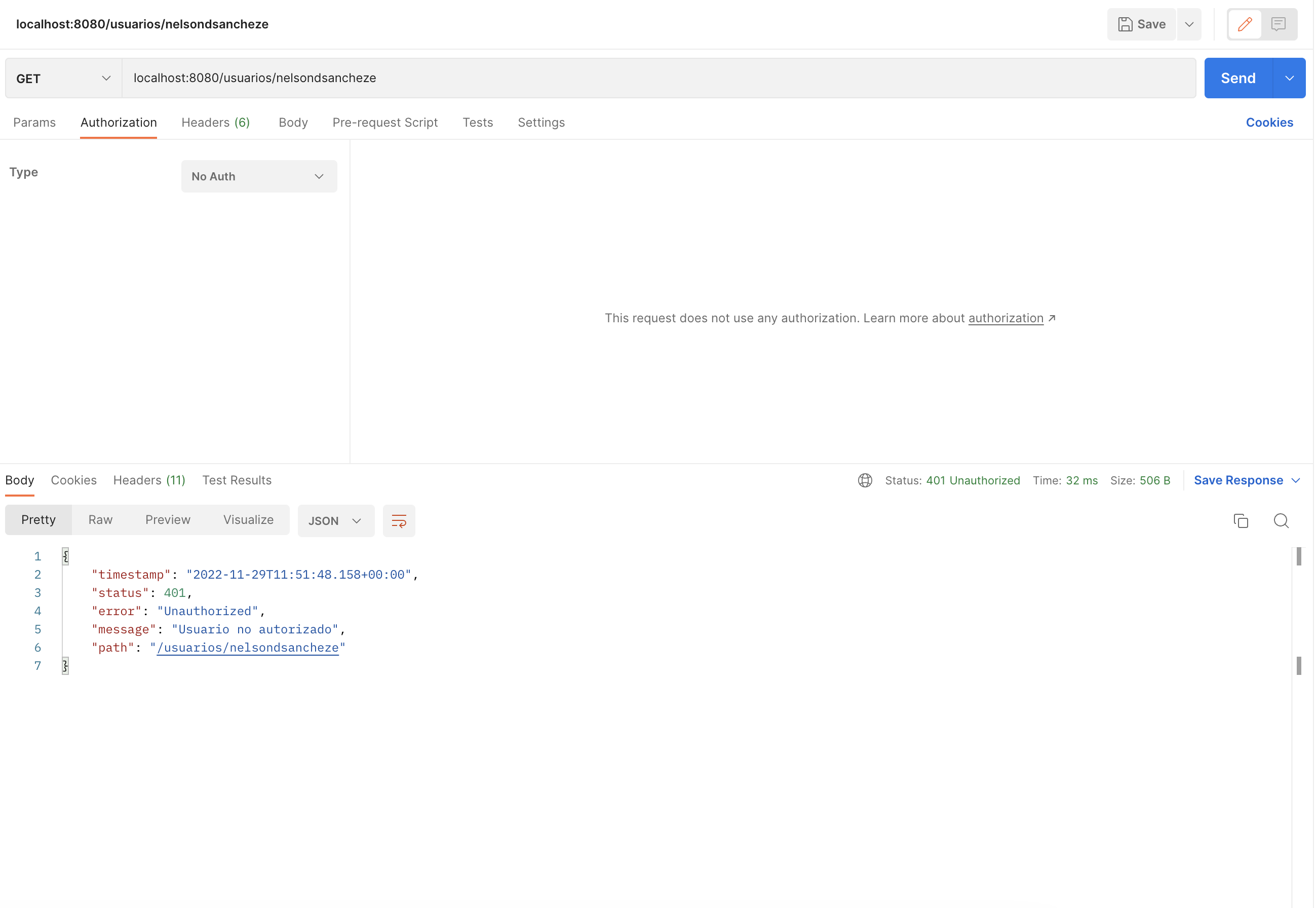
Obs: Se adjunto el proyecto de backend conectado a la base de datos MySql

### Derrollo Frontend

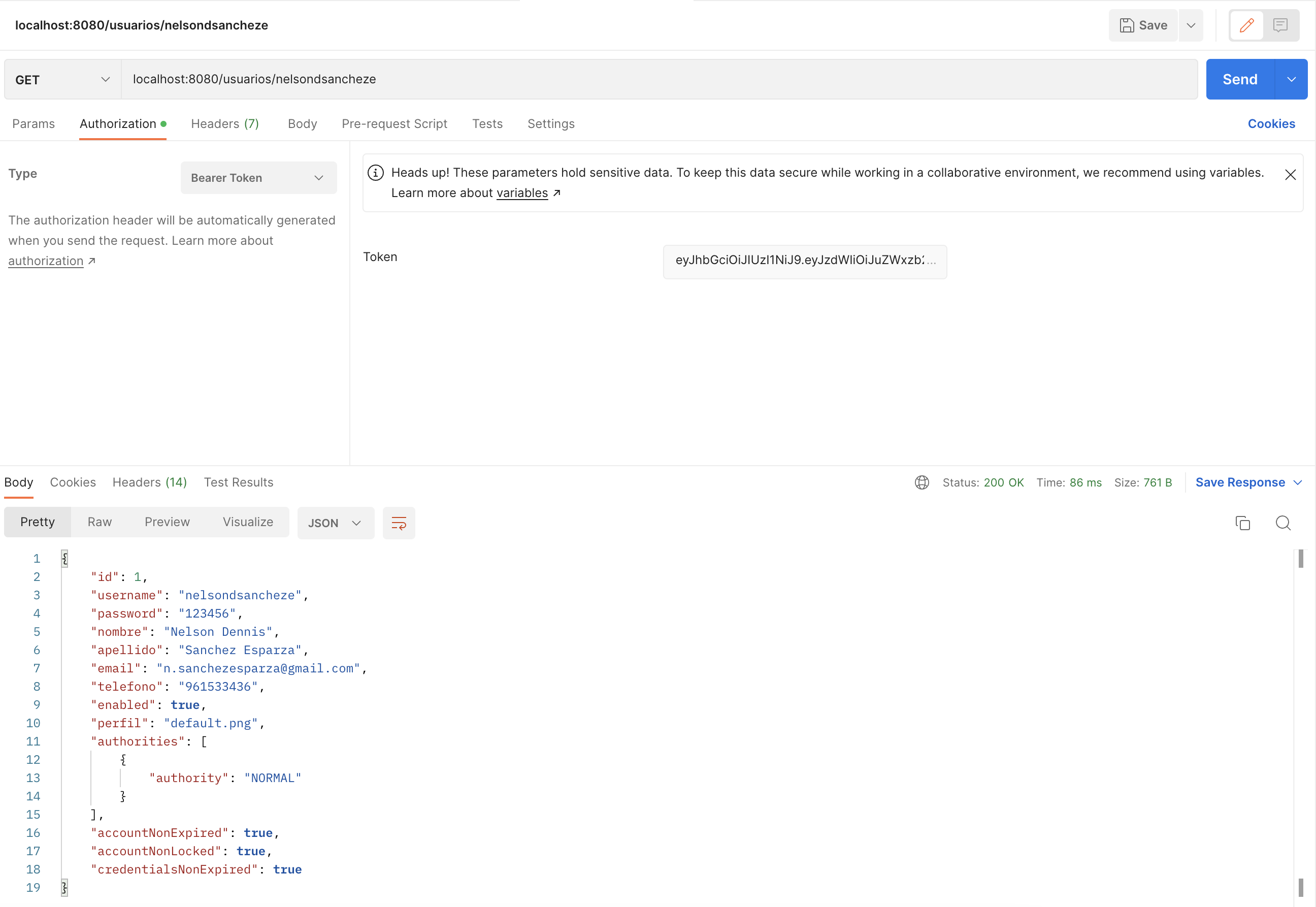


## Pruebas con Postman y Angular Frontend

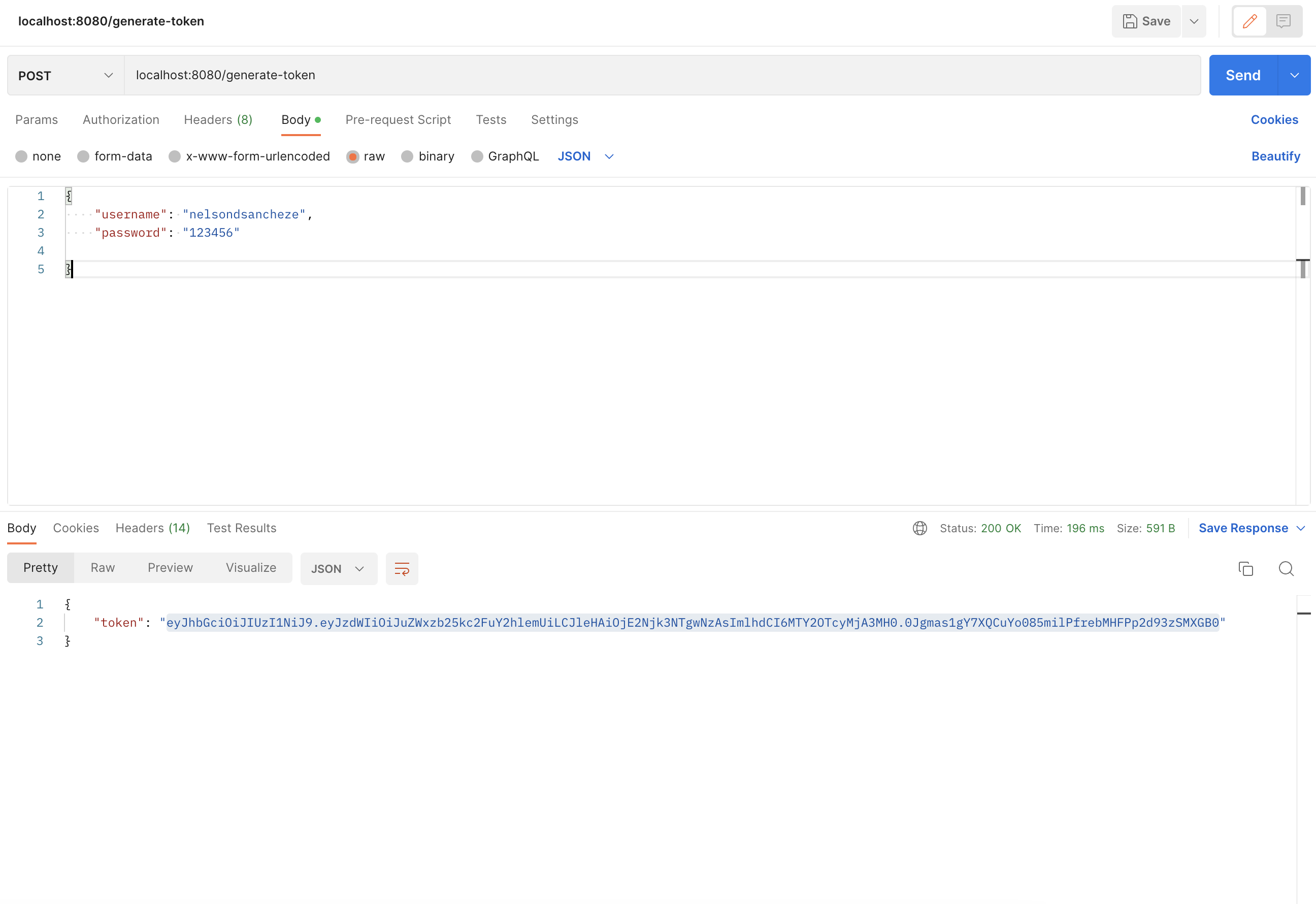
### El servicio responde status 401



### El servicio responde con status 200



### Servicio que genera el token



## ¿Qué es lo que más les ha impresionado o gustado?

Lo que más nos ha gustado son las clases que componen toda la configuración para generar los tokens que protegeran nuestros servicios, como se construye, como se realizo los filtros para especificar que servicios no requieren dicha autenticación y cuales si. Todo esto a nivel de backend. Con respecto a la integración con frontend desde un login y un servicio que pide que se inyecte el token, lo que más nos gusto que en angular se utiliza los interceptores con cual se puede inspeccionar todos los llamados a los servicios o peticiones y a partir de ello se puede modificar estas peticiones para realizar la inserción del Bearer token.

## Conclusiones y Recomendaciones

Nosotros como desarrolladores siempre debemos estar actualizados con la tecnología, aplicar siempre las buenas prácticas para tener un código de calidad y que estos seas escalables. Siempre se debe tener en cuenta los temas de seguridad y protección de nuestros servicios que desarrollamos es por ello que el usos de los token son una medida de protección que nos da la seguridad a nuestras APIS.

## Bibliografía

* Romo, A. (2017, 28 abril). *Seguridad con Token en Spring Boot*. windoctor7. <https://windoctor7.github.io/spring-jwt.html>
* Ionos.es (2019, 12 diciembre). *CORS: qué es y cómo funciona el cross-origin resource sharing*. Digital Guide Ionos. <https://www.ionos.es/digitalguide/paginas-web/desarrollo-web/cross-origin-resource-sharing/>
* López, L. (2020, 17 enero). *Qué es OAuth 2*. Openwebinars.net. <https://openwebinars.net/blog/que-es-oauth2/>