Profesor-P

Explicando la informática esa.

Menú

Securing REST services with Oauth2 in SpringBoot

19 octubre, 2018 por El Profe

Good, students. In this post I will explain how we can provide security to REST services in Spring Boot. The example application is the same as the previous WEB security entry (in Spanish), so the source code is in: https://github.com/chuchip/OAuthServer .

– Explaining the Oauth2 technology

As I said, we will use the OAuth2 protocol, so the first thing will be to explain how this protocol works.

OAuth2 has some variants but I am going to explain what I will use in the program and, for this, I will give you an example so that you understand what we intend to do.

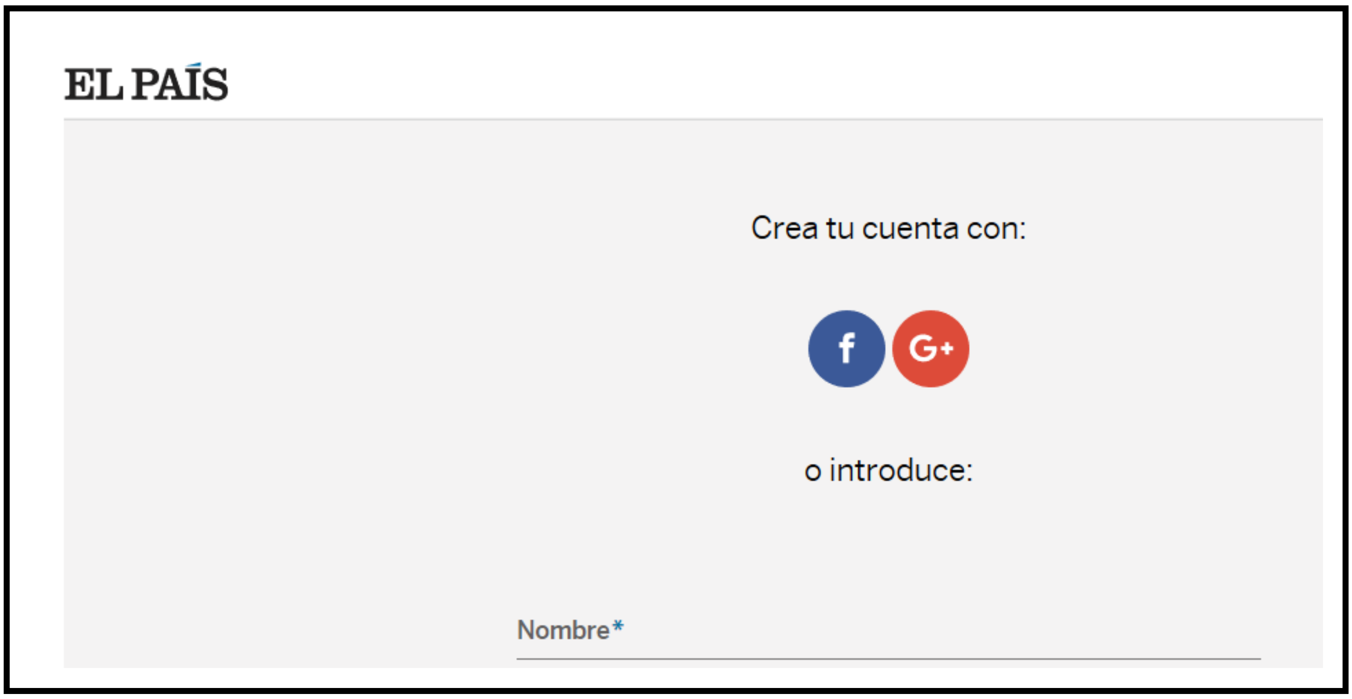
I will put a daily scene: Payment with a credit card in a store. In this case there are three partners: The store, the bank and us. Something similar happens in the Oauth2 protocol. These are the steps:

1. The client, that is, the buyer, asks the bank for a credit card, so that the bank will give us, verify who we are, and give us a credit depending on the mone

have in the account or it tells us not to let's waste time ;-). In the OAuth2 protocol to which it grants the cards, it is called the Authentication Server.

2. If the bank has given us the card, we can go to the store, ie the web server, and we present the credit card. The store does not know us anything, but he can ask the bank, through the card reader, if he can trust us and to what extent (the credit balance). The store are the Resource Server.
3. The store depending on the money that the bank says we have will allow us to buy some products or others. In the OAuth2 analogy, the web server will allow us to access some pages or others depending on whether we are very rich, rich, average or poor.

As a comment, say, in case you have not noticed, that authentication servers are usually used. When you go to a web page and ask you to register, but as an option lets you do it through Facebook or Google, you are using this technology. Google or Facebook becomes the 'bank' that issues that 'card', the web page that asks you to register, will use it to verify that you have 'credit' and let you enter. I hope you understand the example ;-).

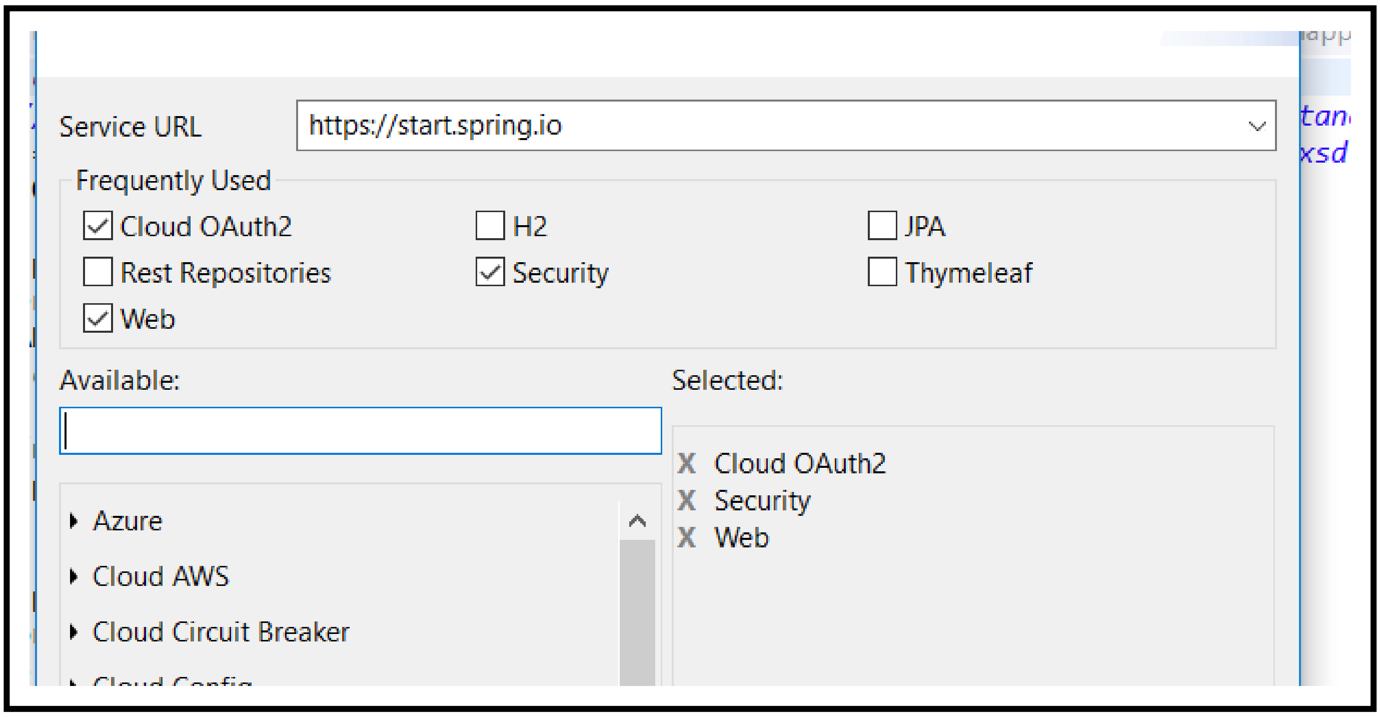


Here you can see the website of the newspaper "El Pais', creating an account. If we use Google or Facebook, the newspaper (the store) will rely on what those authentication providers tell them. In this case the only thing the website needs is that you have a credit card, regardless of the balance 😉

– Creating an Authorization Server

It is understood ?. OK, so let's see how to create the bank, the store and everything you need 😉

First, in our project, we need to have the appropriate dependencies, we will need the starters **Cloud OAuth2, Security and Web**



Service URL: https://start.spring.io

Frequently Used
- [✓] Cloud OAuth2
- [ ] H2
- [ ] JPA
- [ ] Rest Repositories
- [✓] Security
- [ ] Thymeleaf
- [✓] Web

Available:
- Azure
- Cloud AWS
- Cloud Circuit Breaker
- Cloud Config

Selected:
- X Cloud OAuth2
- X Security
- X Web

Well, let's start by defining the bank, this is what we do in the class: **AuthorizacionServerConfiguration**

```
 @Configuration
@EnableAuthorizationServer
public class AuthorizacionServerConfiguration extends AuthorizationServerConfigu

        @Autowired
        @Qualifier ("authenticationManagerBean")
        private AuthenticationManager authenticationManager;

        @Autowired
        private TokenStore tokenStore;
@Override
public void configure (ClientDetailsServiceConfigurer clients) throws Exception {
clients.inMemory ()
                .withClient ("client")
         .authorizedGrantTypes ("password", "authorization_code", "refresh_toke
                .authorities ("ROLE_CLIENT", "ROLE_TRUSTED_CLIENT", "USER")
                .scopes ("read", "write")
                .autoApprove (true)
                .secret (passwordEncoder (). encode ("password"));
            }

            @Bean
              public PasswordEncoder passwordEncoder () {
                 return new BCryptPasswordEncoder ();
              }
            @Override
            public void configure (AuthorizationServerEndpointsConfigurer endpoints) thr
               endpoints
                     .authenticationManager (authenticationManager)
                     .tokenStore (tokenStore);
            }

             @Bean
             public TokenStore tokenStore () {
                return new InMemoryTokenStore ();
             }
     }
```

We start the class by entering it as a configuration with the @ **Configuration** label and then use the **@EnableAuthorizationServer** tag to tell Spring to activate the authorization server. To define the server properties, we specify that our class extends the **AuthorizationServerConfigurerAdapter** , which implements the **AuthorizationServerConfigurerAdapter** interface, so Spring will use this class to parameterize the server.

We define an **AuthenticationManager** type bean that Spring provides automatically and that we will collect with the **@Autowired** tag. We also define a **TokenStore** object **,** but to be able to inject it we must define it, which we do in the **public** function **TokenStore tokenStore ().**

The **AuthenticationManager**, as I said, is provided by Spring but we will have to configure it ourselves. Later I will explain how it is done. The **TokenStore** or Identifier Store is where the identifiers that our authentication server is supplying will be stored, so that when the resource server (the store) asks for the credit on a credit card it can respond to it. In this case we use the **InMemoryTokenStore** class that will **store** the identifiers in memory. In a real application we could use a **JdbcTokenStore** to save them in a database, so that if the application goes down the clients do not have to renew their credit cards 😉

In the function **configure (ClientDetailsServiceConfigurer clients)** we specify the credentials of the bank, I say of the administrator of authentications, and the services offered. Yes, in plural, because to access the bank we must have a username and password for each of the services offered. This is a very important concept: *The user and password is from the bank, not the customer,* for each service offered by the bank there will be a single authentication, although it may be the same for different services.

I will detail the lines:

- **clients.inMemory ()** Specifies that we are going to store the services in memory. In a 'real' application we would save it in a database, an LDAP server, etc.
- **withClient ("client")** Is the user with whom we will identify in the bank. In this case it will be called 'client'. Would it have been better to call him 'user'?
- to **uthorizedGrantTypes ("password", "authorization_code", "refresh_token", "implicit")** . We specify the services that we are configuring for the defined user, for ' **client** '. In our example we will only use the **password** service.
- **authorities ("ROLE_CLIENT", "ROLE_TRUSTED_CLIENT", "USER").** Specify roles or groups that the service offered has. We will not use it in our example either, so let's let it run for the time being.
- **scopes ("read", "write").** The scope of the service. Nor will we use it in our application.
- **autoApprove (true)** If you must automatically approve the client's requests. We'll say yes to make the application simpler.
- **secret (passwordEncoder (). encode ("password")).** Password of the client. Note that the **encode** function that we have defined a little below is called, to specify with what type of encryption the password will be saved. The **encode** function is annotated with the **@Bean** tag because spring, when we supply the password in an HTTP request, will look for a **PasswordEncoder** object to check the validity of the delivered password.

And finally we have the function **configure (AuthorizationServerEndpointsConfigurer endpoints)** where we define which authentication controller and which store of identifiers should use the end points. Clarify that the end points are the URLs where 'we will talk to our bank', to request the piggy cards 😉

Ok, we have our authentication server created but we still need the way that he kr who we are and puts us in different groups, according to the credentials introduce

Well, to do this we will use the same class that we use to protect a web page. If you have read the previous article: http://www.profesor-p.com/2018/10/17/seguridad-web-en-spring-boot/ (in Spanish) remember that we created a class that inherited from **WebSecurityConfigurerAdapter** , where we **overwrote** the function **UserDetailsService userDetailsService ().**

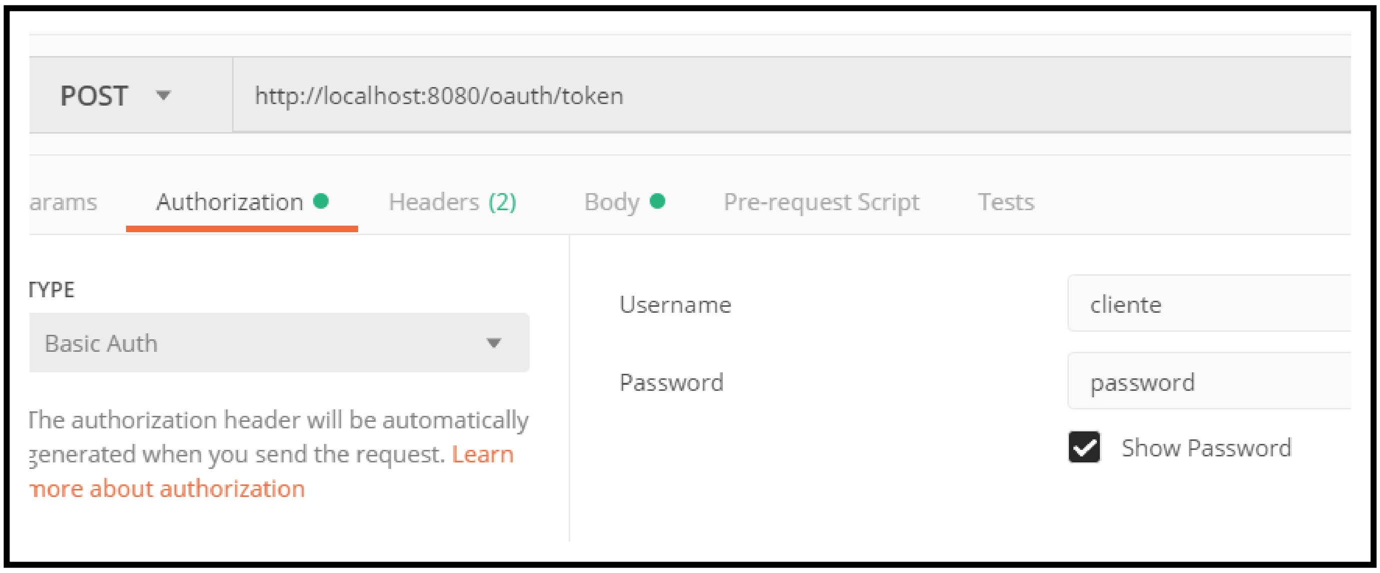
```
 @EnableWebSecurity
public class WebSecurityConfiguration extends WebSecurityConfigurerAdapter {
 ....
   @Bean
   @Override
   public UserDetailsService userDetailsService () {

       UserDetails user = User.builder (). Username ("user"). Password (passwordEnc
                   roles ("USER"). build ();
       UserDetails userAdmin = User.builder (). Username ("admin"). Password (pass
                   roles ("ADMIN"). build ();
      return new InMemoryUserDetailsManager (user, userAdmin);
   }
....
}
```

Well, users with their roles or groups are defined in the same way. We should have a class that extends **WebSecurityConfigurerAdapter** and define our users.

Now and we can check if our authorizations server works. Let's see how, using the excellent **PostMan** program **.**

To speak with the 'bank' to request our credentials, and as we have not defined otherwise, we must go to the URI "/oauth/token". This is one of the end points I spoke about earlier. There is more but in our example and since we are only going to use the service 'password' we will not use more.

We will use a HTTP request type POST, indicating that we want to use basic validation, we will put the user and password, which will be those of the "bank", in our example: 'client' and 'password' respectively.



POST http://localhost:8080/oauth/token

arams | Authorization ● | Headers (2) | Body ● | Pre-request Script | Tests

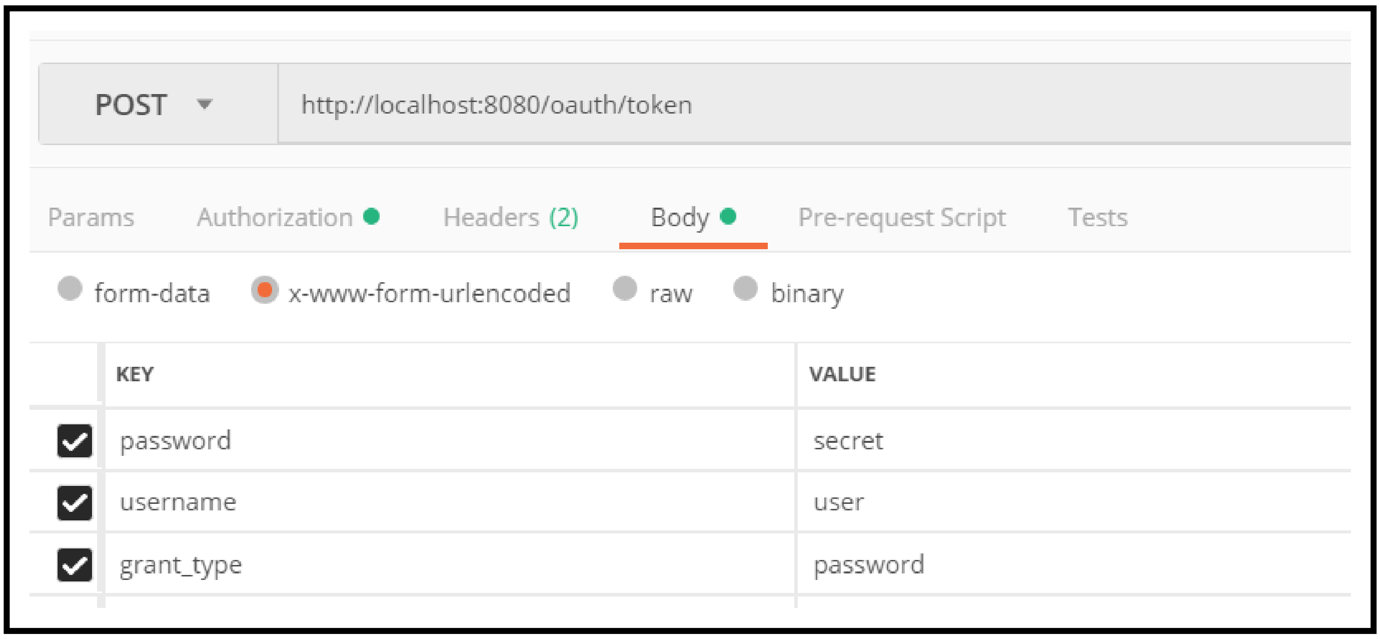
TYPE: Basic Auth

The authorization header will be automatically generated when you send the request. Learn more about authorization

Username	cliente
Password	password

✓ Show Password

In the body of the request, in form-url-encoded format we will introduce the service to request, our username and our password.



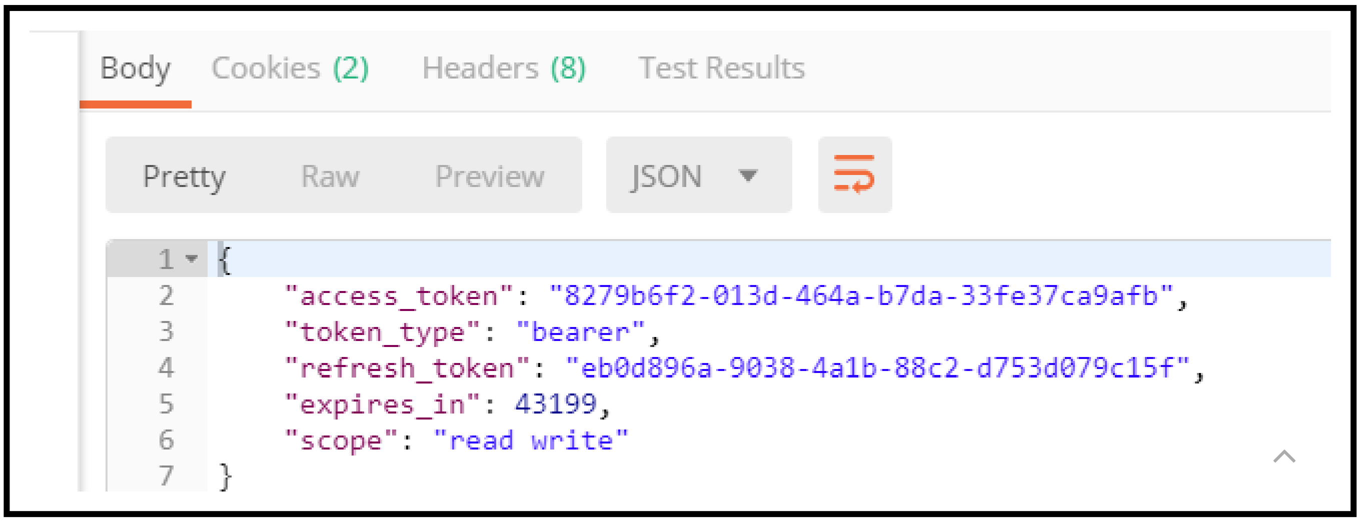
POST http://localhost:8080/oauth/token

Params | Authorization ● | Headers (2) | Body ● | Pre-request Script | Tests

form-data [✓ x-www-form-urlencoded] raw binary

	KEY	VALUE
✓	password	secret
✓	username	user
✓	grant_type	password

and we launched the petition, which should get us an exit like this:



Body | Cookies (2) | Headers (8) | Test Results

Pretty | Raw | Preview | JSON

```
{
    "access_token": "8279b6f2-013d-464a-b7da-33fe37ca9afb",
    "token_type": "bearer",
    "refresh_token": "eb0d896a-9038-4a1b-88c2-d753d079c15f",
    "expires_in": 43199,
    "scope": "read write"
}
```

That 'access_token' " **8279b6f2-013d-464a-b7da-33fe37ca9afb** " is our credit card and is the one that we must present to our resource server (the store) in order to see pages (resources) that are not public.

– Creating a Resource Server (ResourceServer)

Now that we have our credit card, we will create the store that accepts that card 😉

In our example we are going to create the server of resources and of authentication in the same program, with which Spring Boot, will be in charge to do that I trusted a part in another, without having to configure anything. If, as usual in real life, the resource server is in one place and the authentication server in another, we should indicate to the resource server, which is our 'bank' and how to talk to it. But we'll leave that for another entry.

The only class of the resource server is **ResourceServerConfiguration**

```
 @EnableResourceServer
@RestController
public class ResourceServerConfiguration extends ResourceServerConfigurerAdapt
{
.....
}
```

Observe the @ **EnableResourceServer** annotation that will cause Spring to activate the resource server. The tag **@RestController** is because in this same class we will have the resources, but they could be perfectly in another class.

Finally, note that the class extends **ResourceServerConfigurerAdapter** this is because we are going to overwrite methods of that class to configure our resource server.

As I said before, since the authentication and resources server is in the same program, we only have to configure the security of our resource server. This is done in the function:

```
 @Override
public void configure (HttpSecurity http) throws Exception {
http
.authorizeRequests (). antMatchers ("/ oauth / token", "/ oauth / authorize **", "
// .anyRequest (). authenticated ();
```

```
        http.requestMatchers (). antMatchers ("/ private") // Deny access to "/ private"
    .and (). authorizeRequests ()
    .antMatchers ("/ private"). access ("hasRole ('USER')")
        .and (). requestMatchers (). antMatchers ("/ admin") // Deny access to "/ admir
    .and (). authorizeRequests ()
    .antMatchers ("/ admin"). access ("hasRole ('ADMIN')");
    }
```

In the previous entry when we defined the security on the web, explained a function called **configure (HttpSecurity http)** , how is it much like this? Well, if it is basically the same, and in fact it receives an HttpSecurity object that we must configure.

I explain line to line the sentences:

- **`http.authorizeRequests().antMatchers("/oauth/token", "/oauth/authorize**", "/publica").permitAll()`** allow all requests to "/ oauth / token", "/ oauth / authorize ** "," / publishes "without any validation.
- **`anyRequest().authenticated()`** This line is commented, if not all the resources would be accessible only if the user has been validated.
- **`requestMatchers().antMatchers("/privada")`** Deny access to the url "/ private"
- **`authorizeRequests().antMatchers("/privada").access("hasRole('USER')")`** allow access to "/ private" if the validated user has the role 'USER'
- **`requestMatchers().antMatchers("/admin")`** Deny access to the url "/ admin"
- **`authorizeRequests().antMatchers("/admin").access("hasRole('ADMIN')")`** allow access to "/ admin" if the validated user has the role 'ADMIN'

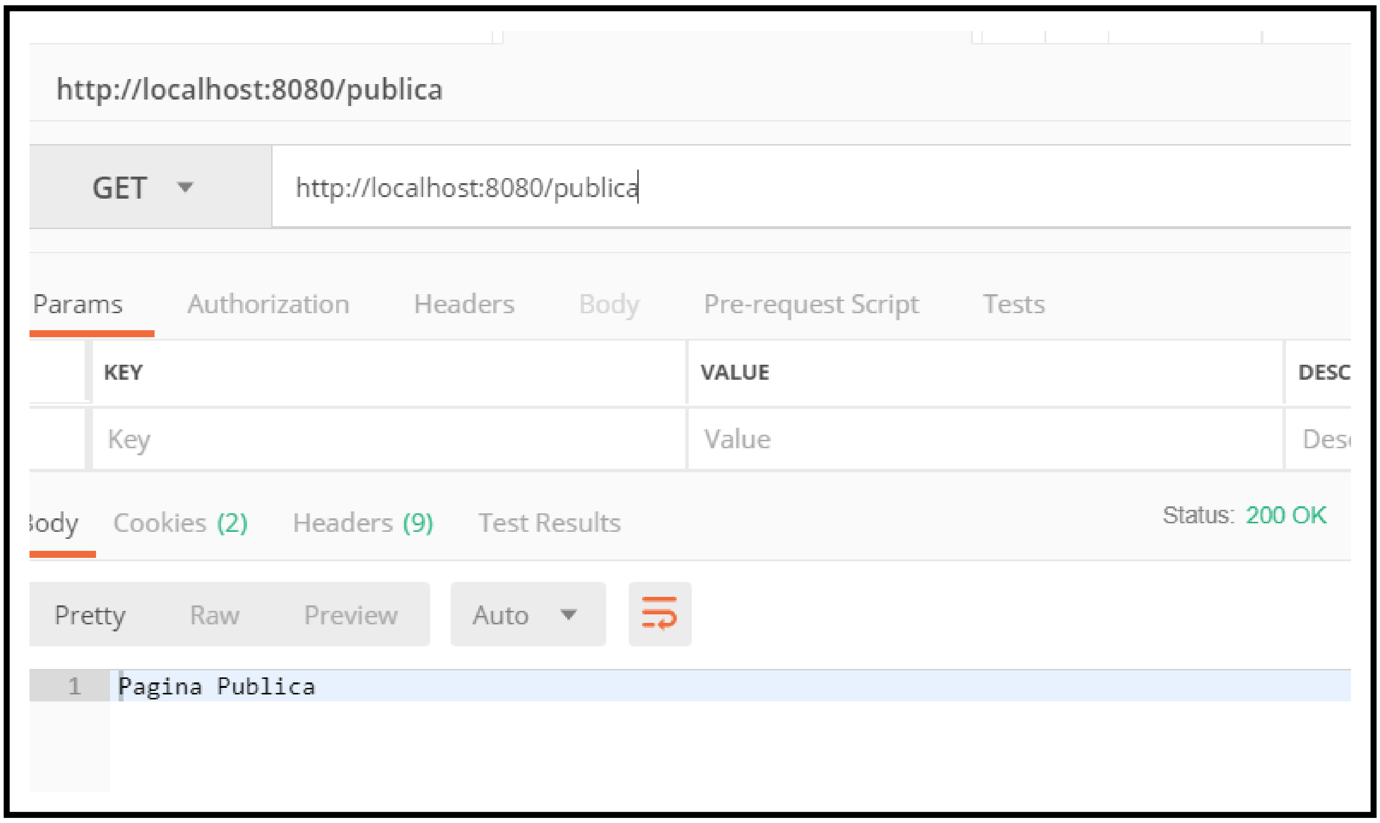
Once we have our resource server created we must only create the services which is done with these lines:

```
@RequestMapping ("/ publishes")
 public String publico () {
        return "Public Page";
 }
 @RequestMapping ("/ private")
 public String private () {
      return "Private Page";
 }
 @RequestMapping ("/ admin")
 public String admin () {
         return "Administrator Page";
 }
```

As you can see, there are 3 basic functions that only return their corresponding Strings.

Let’s see now how validation works.

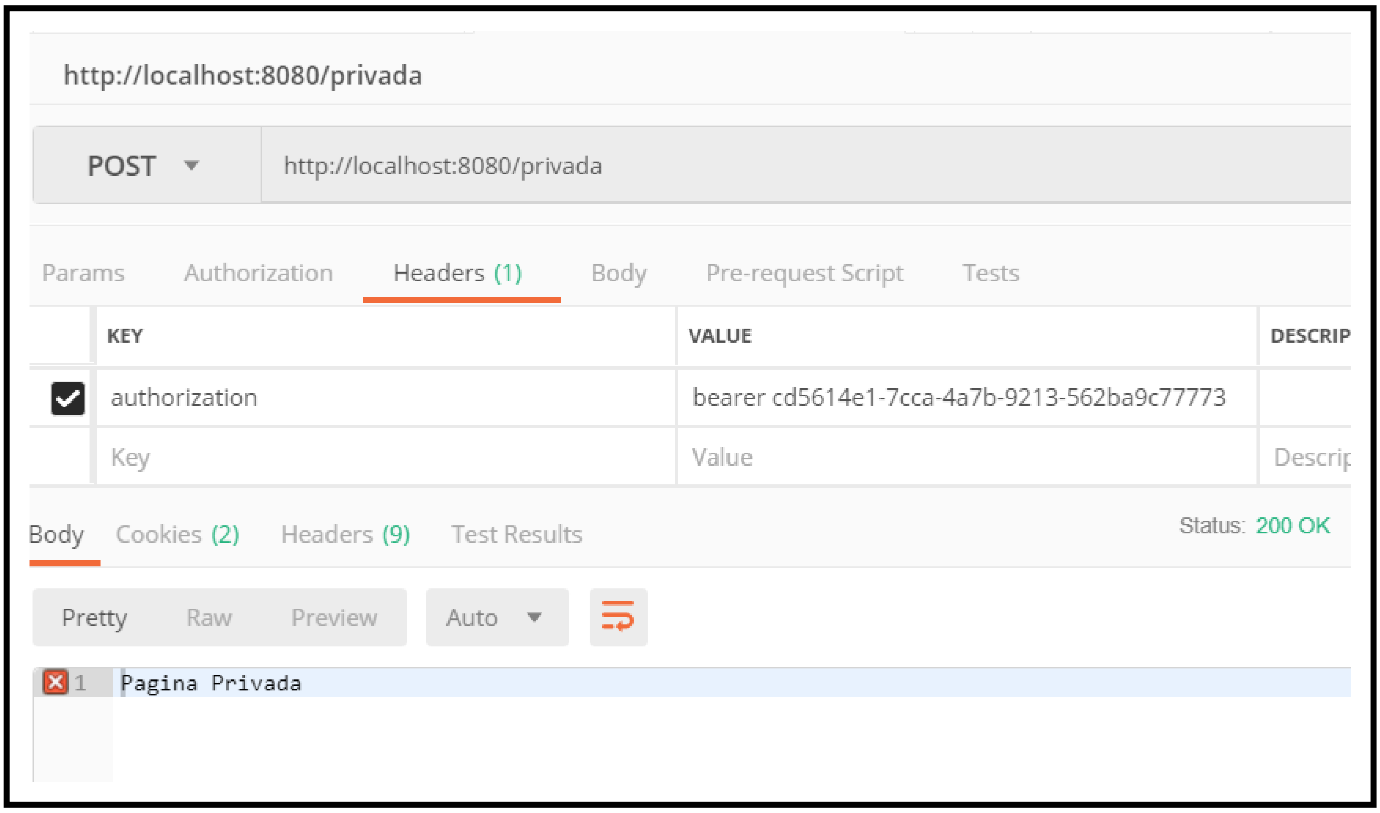
First we check that we can access “/publica” without any validation:



```
Pagina Publica
```

Right. This works!!

If I try to access the page “/private” I receive an error “401 unauthorized”, which indicates that we do not have permission to see that page, so we will use the token issued by our authorizations server, for the user ‘*user*‘, to see what happens 😉



http://localhost:8080/privada

POST http://localhost:8080/privada

Params | Authorization | Headers (1) | Body | Pre-request Script | Tests

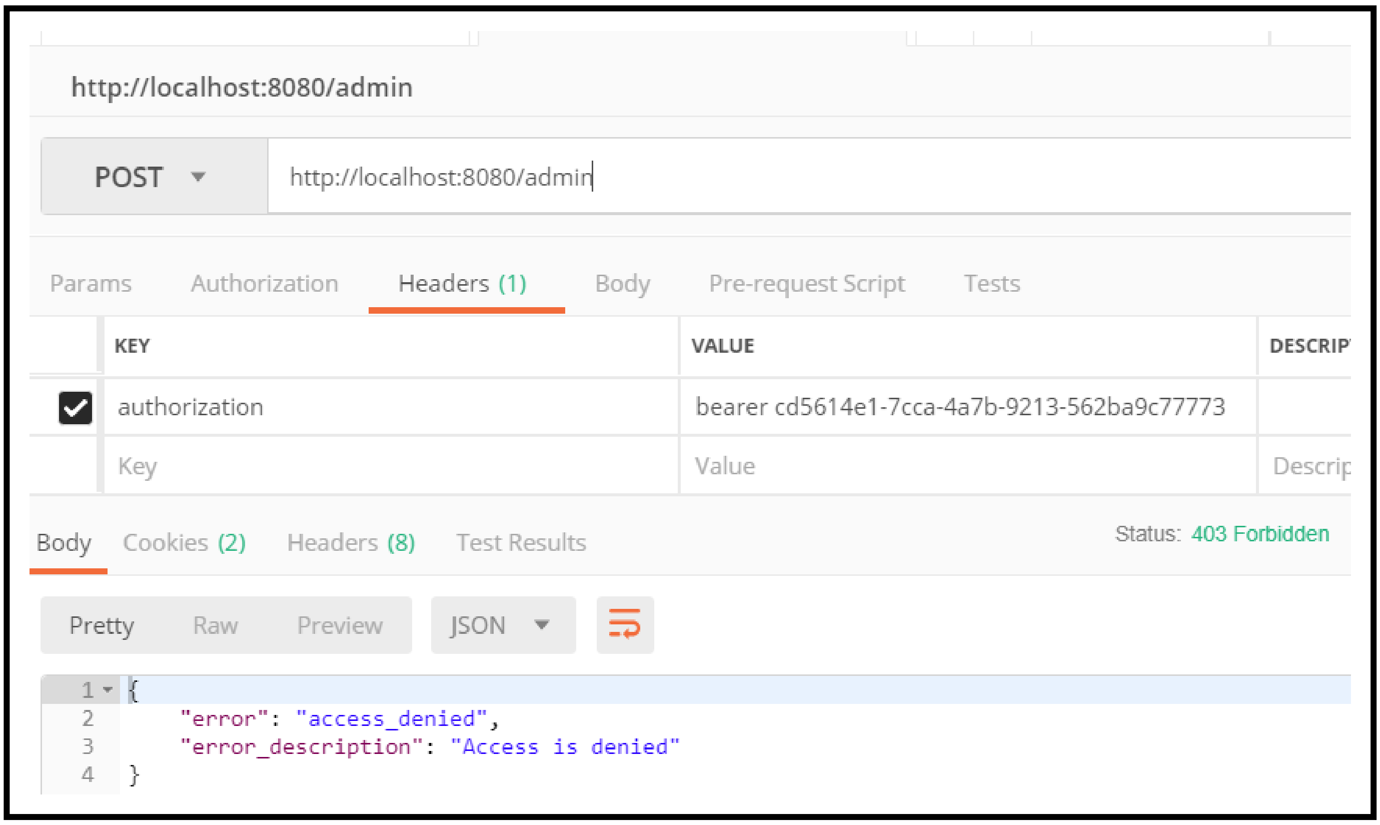
	KEY	VALUE	DESCRIP
✓	authorization	bearer cd5614e1-7cca-4a7b-9213-562ba9c77773	
	Key	Value	Descrip

Body | Cookies (2) | Headers (9) | Test Results — Status: 200 OK

Pretty | Raw | Preview | Auto

```
Pagina Privada
```

Come on, if we can see our private page. Let's try the administrator's page ...



http://localhost:8080/admin

POST http://localhost:8080/admin

Params | Authorization | Headers (1) | Body | Pre-request Script | Tests

	KEY	VALUE	DESCRIP
✓	authorization	bearer cd5614e1-7cca-4a7b-9213-562ba9c77773	
	Key	Value	Descrip

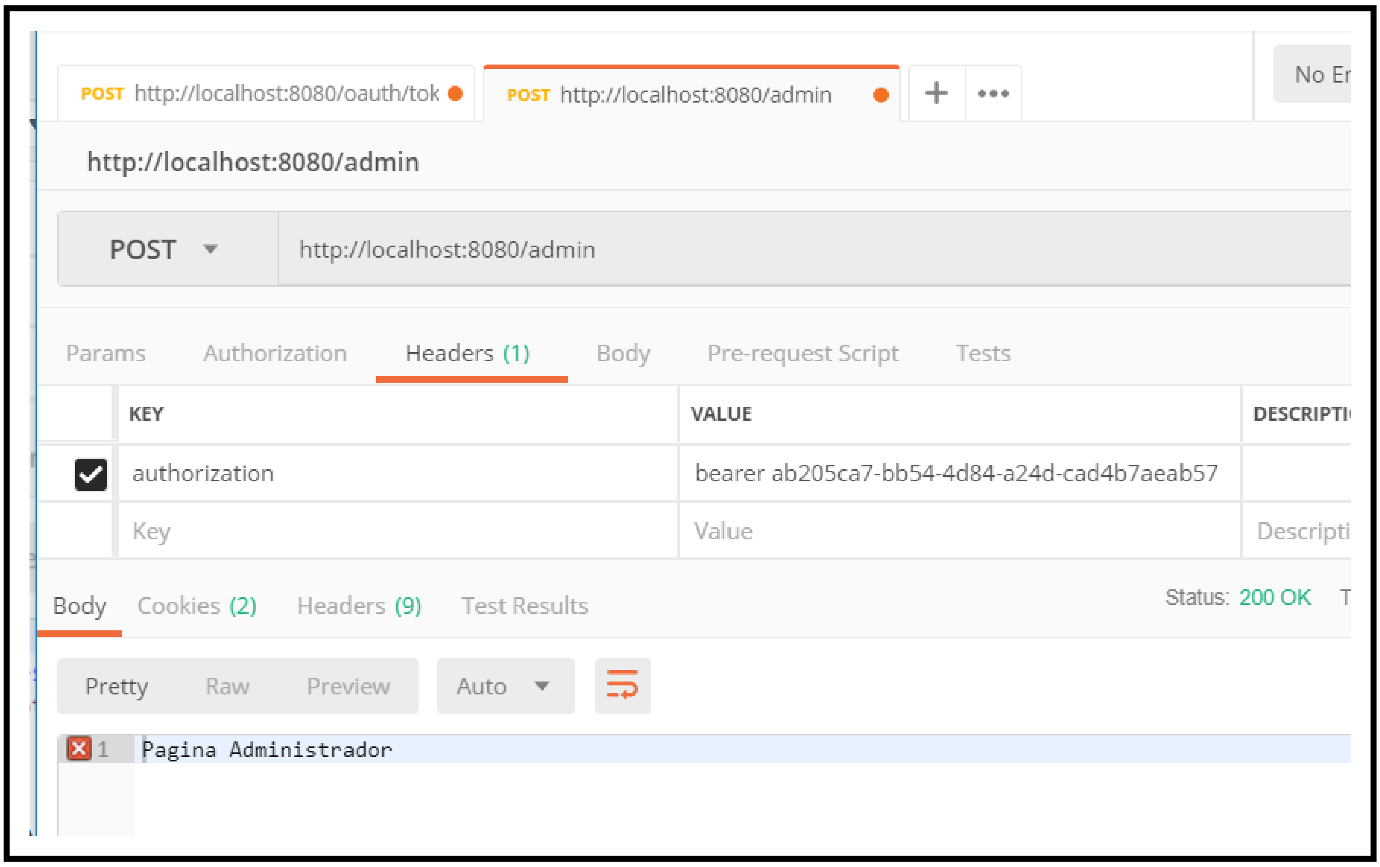
Body | Cookies (2) | Headers (8) | Test Results — Status: 403 Forbidden

Pretty | Raw | Preview | JSON

```
{
    "error": "access_denied",
    "error_description": "Access is denied"
}
```

Right, we can not see it. So we're going to request a new token from the credential administrator, but identify ourselves with the user 'admin'.

The token returned is: "" ab205ca7-bb54-4d84-a24d-cad4b7aeab57 ". We use i see what happens:



http://localhost:8080/admin

POST http://localhost:8080/admin

KEY	VALUE	DESCRIPTI
✓ authorization	bearer ab205ca7-bb54-4d84-a24d-cad4b7aeab57	
Key	Value	Descripti

Status: 200 OK

```
Pagina Administrador
```

Well, that's it, we can go shopping safely! Now we just need to set up the store and have the products 😉

And that's it. See you in the next article, students. 🙂

java, security, seguridad, spring boot, thymeleaf
java, mvc, rest, security, spring boot

‹ Securizando servicios REST con Oauth2 en SpringBoot
› Spring WebFlow con JSP – Configuración

Deja un comentario

Nombre *

Correo electrónico *

Sitio web

Publicar comentario

Búsqueda

Buscar ...

Entradas recientes

Beans avanzados en Spring

Optimizando relaciones entre entidades en Hibernate

Accediendo facilmente a los datos con Spring Rest Data

Categorías

Elegir categoría

Archivos

abril 2019 (2)

marzo 2019 (3)
febrero 2019 (2)
enero 2019 (2)
diciembre 2018 (2)
noviembre 2018 (2)
octubre 2018 (10)
septiembre 2018 (14)
agosto 2018 (10)

Etiquetas

ajax angular angular 6 angular6 beans bootstrap cloud crud curso formularios frontend glassfish h2 hibernate html i18n internacionalizacion java Java ee javaee jdbc jndi jpa jquery json junit kotlin lambda linux mvc netbeans netbeans9 pooper profiles reactivo rest routes security seguridad spring spring boot test tomcat web webflow