**Lab Oauth 2.0**

**Refactoring the ShoppingService**

In the previous labs you have developed a shopping service and a product service. Up untill now both services are unsecured. So, everyone is able to invoke the endpoints that query, update and delete drinks.

Let’s make adjustjusytment to the shoppingservice to control who is allowed to make changes to the data in the repository. For now, you will only make these changes to the shopping service and leave the product service as is (assuming this is a 3rd party service).

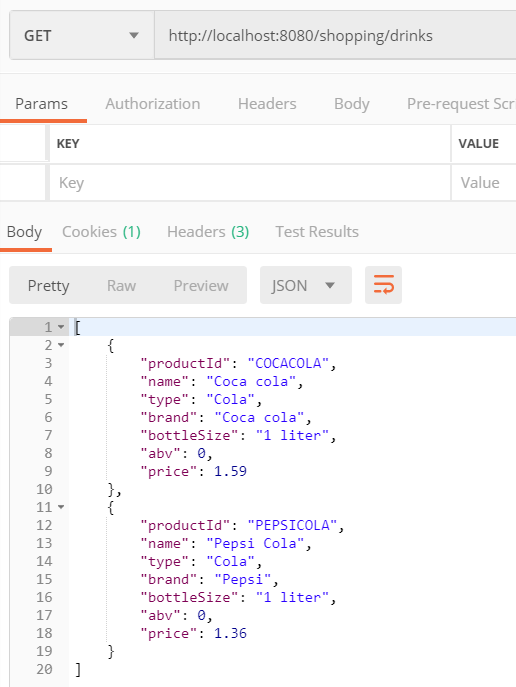
If you do not have the shopping service and product service available, you can make use of the ones in the starter folder.

**Test the ShoppingService app**

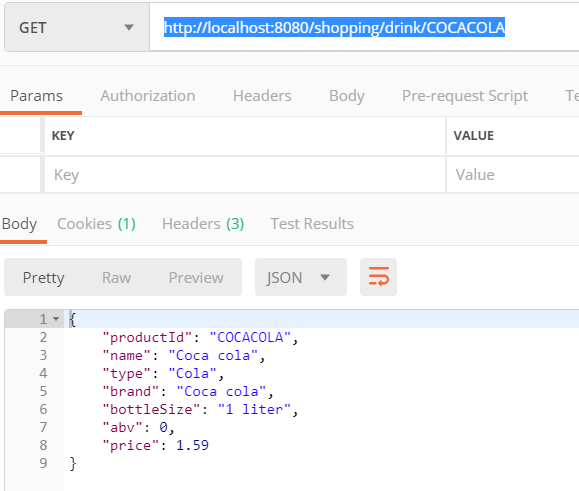
Start the shopping service and the product service.

Test that things are working by entering the following URIs in your browser, curl or postman.

<http://localhost:8080/shopping/drinks>



<http://localhost:8080/shopping/drink/COCACOLA>

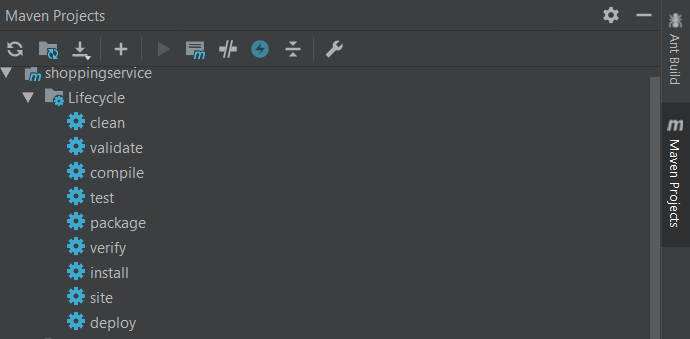


**Add dependencies**

<dependency>  
 <groupId>org.springframework.boot</groupId>  
 <artifactId>spring-boot-starter-security</artifactId>  
</dependency>  
  
<dependency>  
 <groupId>org.springframework.security.oauth</groupId>  
 <artifactId>spring-security-oauth2</artifactId>  
 <version>2.0.13.RELEASE</version>  
</dependency>

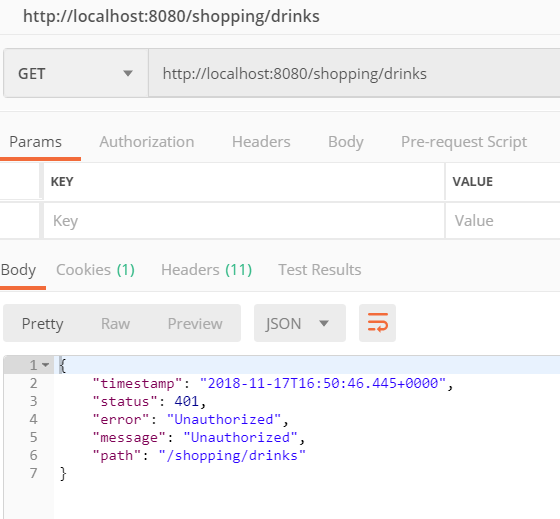
* Add the above dependencies to the pom.xml file for the shopping service.
* Do a maven clean + compile

In IntelliJ : upper right corner :



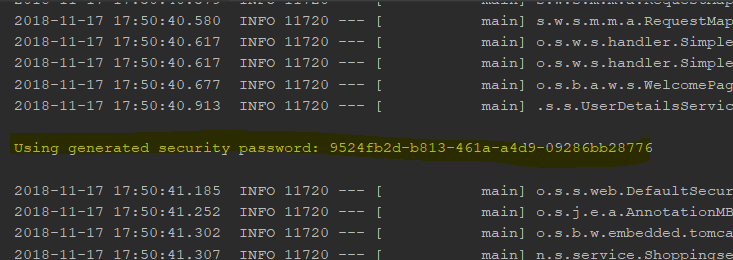
* Restart the shopping service.

Can you still retrieve data by invoking: <http://localhost:8080/shopping/drinks> ?



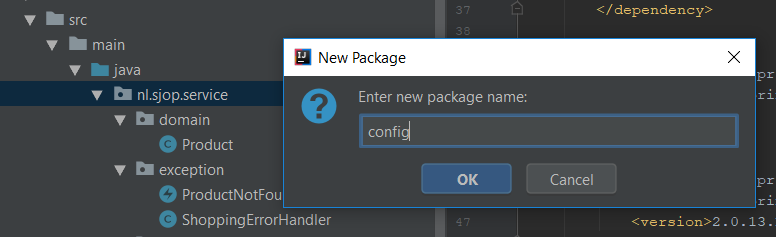
If things went well, you should get a message that you are not authorized query the data through this URL.

By adding the security dependencies to the pom.xml, the application gets secured out of the box. Another indication that this is the case, can be seen in the log files:



**Add folder**

With that in place, let’s move on and add some code to the shopping service. Place this code in a new folder named **config,** located under (src.main.java) nl.sjop.service.



**Add configuration for the authorization server**

Add a new file to the config folder and name it AuthorizationServerConfig.java. Have it extend the AuthorizationServerConfigurerAdapter, which provides empty implementation methods for the interface AuthorizationServerConfigurer. Also annotate the class with @EnableAuthorizationServer.

@Configuration  
@EnableAuthorizationServer  
public class AuthorizationServerConfig extends AuthorizationServerConfigurerAdapter {

}

The clients that can access the authorization server will be setup here. For the demo purpose this is done by means of a in memory client details service.

@Override  
 public void configure(final ClientDetailsServiceConfigurer clients) throws Exception {  
 clients.inMemory()   
 .withClient("client")  
 .secret("client-secret")  
 .scopes("read")  
 .authorizedGrantTypes("password")  
 .accessTokenValiditySeconds(3600)  
 .and()  
 .withClient("client2")  
 .secret("client2-secret")  
 .scopes("read", "write", "trusted")  
 .authorizedGrantTypes("password")  
 .accessTokenValiditySeconds(3600);  
 }

Add the above configure method to the AuthorizationConfig file. As you can see, 2 (or more) clients can be defined each with it’s own scope(s).

Before a client will be authorized to access some (protected) resource, it must be authenticated first. This is done by an authentication mechanism that is managed by an authentication manager. A default manager is injected into the configuration.

As a proof of authorization, a token will be send to the client and cached in the server for validation purposes further on in the process. The tokenstore is instantiated by the container as a bean.

Both the authentication manager and the tokenstore are registered as endpoints in the configure method.

@Autowired  
private AuthenticationManager authenticationManager;  
  
@Bean  
public TokenStore tokenStore() {  
 return new InMemoryTokenStore();  
}  
  
@Override  
public void configure(final AuthorizationServerEndpointsConfigurer endpoints) {  
 endpoints.tokenStore(tokenStore())  
 .authenticationManager(authenticationManager);  
}

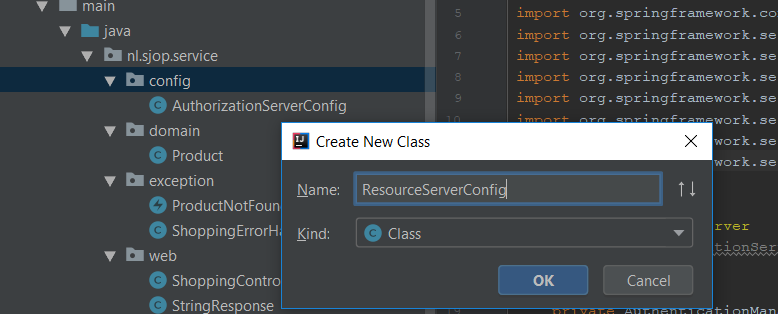
Add the above code to the AuthorizationServerConfig class.

**Add configuration for the resource server**

The resource server serves the resources that are protected by the OAuth2 token.

You wil now add a configuration for this to the shopping service.

Add a new file ResourceServerConfig.java to the config folder.



Have the configuration class extend the ResourceServerConfigurerAdapter and give it the following annotations. As expected, the annotation @EnableResourceServer lets **our application to behave as a**Resource Server.

@Configuration  
@EnableResourceServer  
@EnableWebSecurity  
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {

The resource server needs to know about the users and their roles. In this example you will access the application with 2 users, that are configured in memory. Normally these users will be registered in LDAP (or database). For this kind of implementation, have a look on the internet.

Add the following code for configuring the 2 users:

@Autowired  
 public void configureGlobal(final AuthenticationManagerBuilder auth) throws Exception {  
 auth.inMemoryAuthentication()  
 .withUser("user").password("user").roles("USER")  
 .and().withUser("admin").password("admin").roles("ADMIN");  
 }

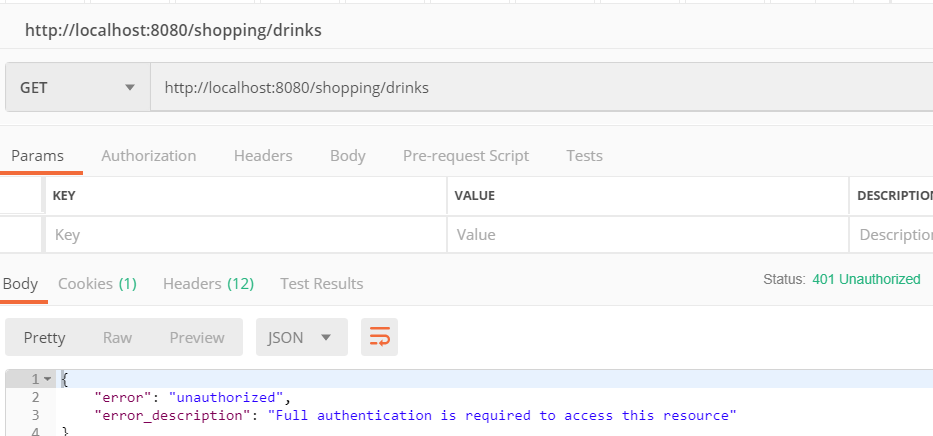
Finally, you have to configure how the endpoints to the resources wil be (un)protected. Add the code below. Through it, all the requests to the shopping service must be authenticated.

@Override  
public void configure(HttpSecurity http) throws Exception {  
 http  
 .authorizeRequests()  
 .anyRequest().authenticated().and()  
 .sessionManagement()  
 .sessionCreationPolicy(SessionCreationPolicy.*STATELESS*).and()  
 .csrf().disable();  
}

**Test the shopping service**

Restart the shopping service.

Invoke the url: <http://localhos:8080/shopping/drinks> .



As expected, you are not authorized to invoke this (yet). For this you need a valid access token. So, let’s ask for one, using Postman.

URI : [http://localhost:8080**/oauth/token**](http://localhost:8080/oauth/token) (this path is standard for OAUTH 2.0)

Method : POST

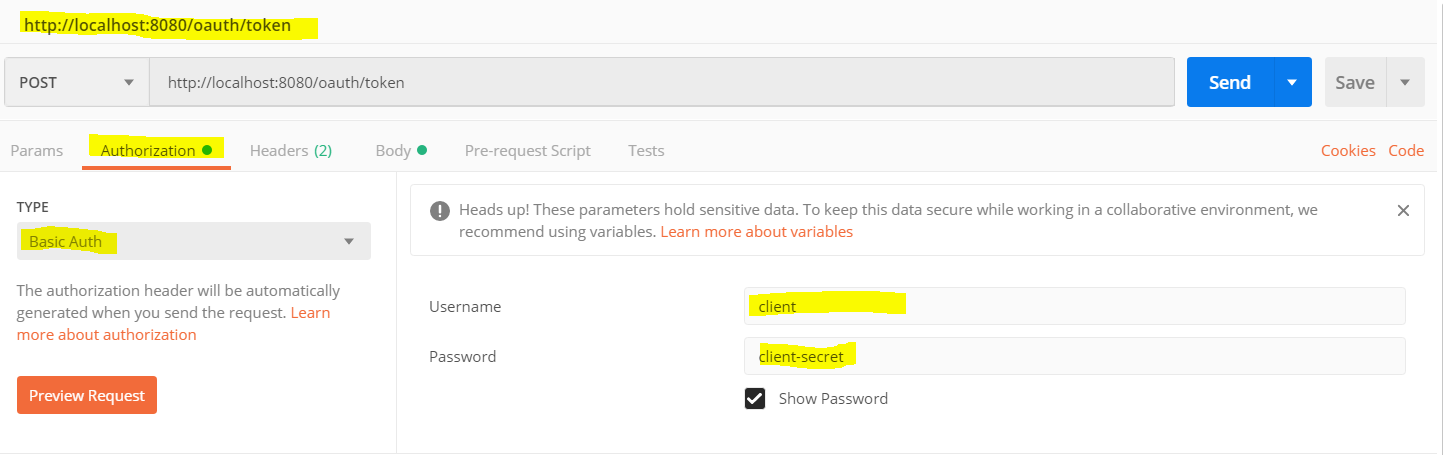
Authorization tab :

Type : Basic Authentication

Username: client

Password: client-secret

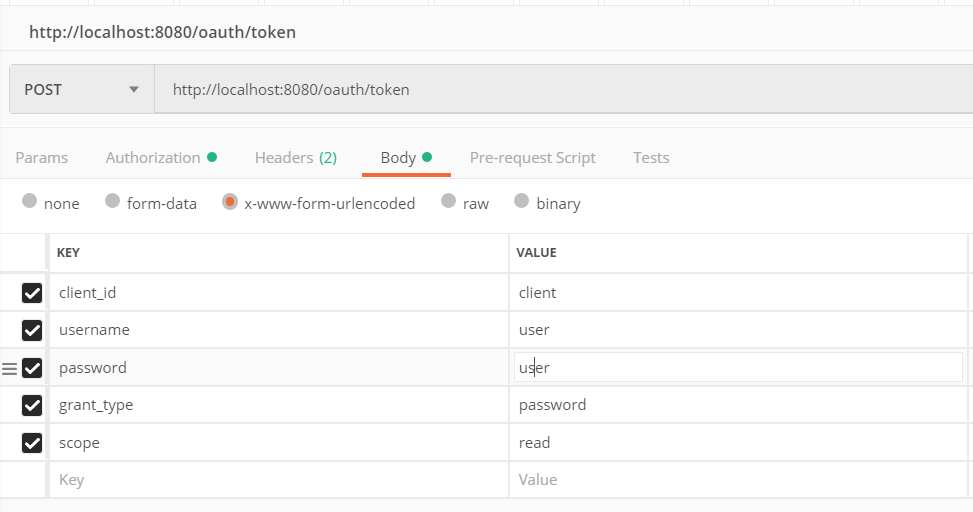
(so, these must match with a configured clients in the AuthorizationConfig)



Body tab :

Type : x-www-form-urlencoded

Fill in the key-tab values as below



**Note,**

the client\_id value must match with the username value in the Basic Auth setting:

the username and password must match with one configured in the ResourceServerConfig:

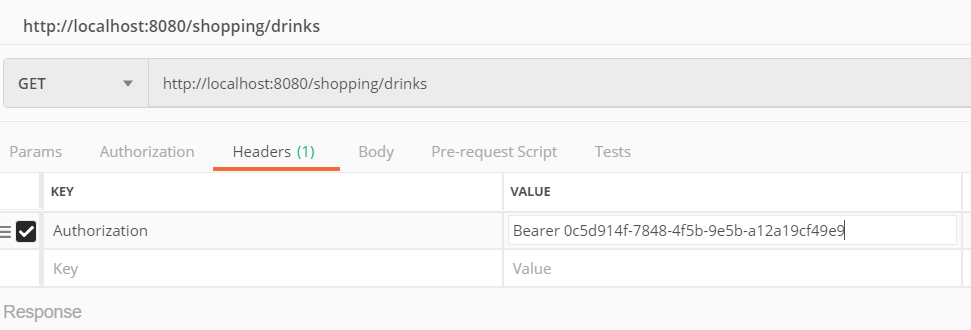
the scope value must match with one of the values registered with the client in the AuthorizationConfig.

After pressing the Send button, you shoul get a response like below:

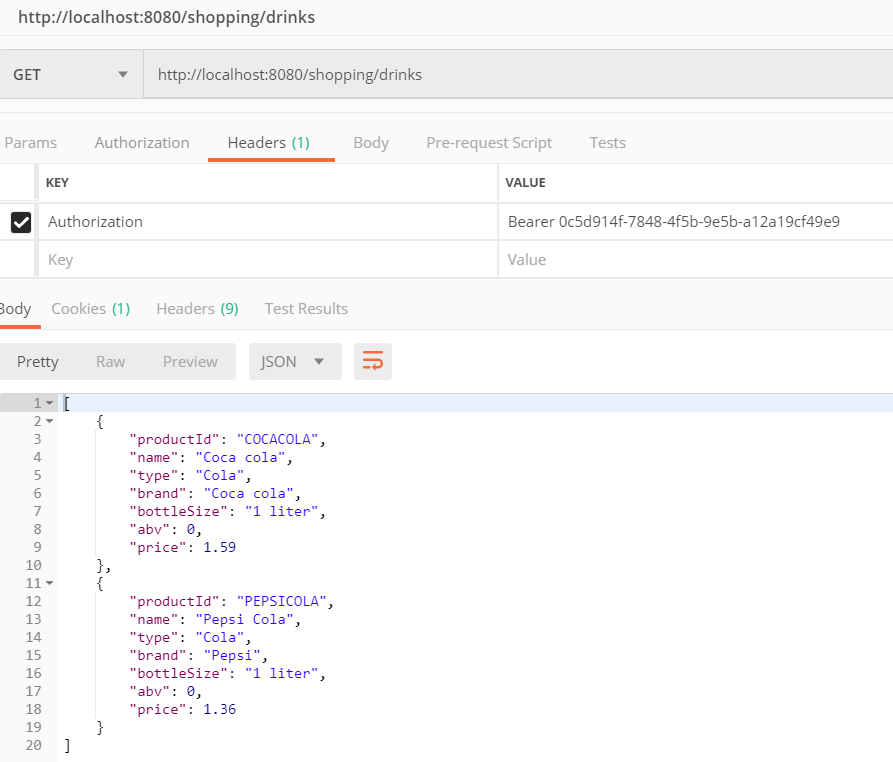


Copy the access value you received in the response (copying the value from the example will NOT work because it is a unique to your server and the value expires)

Make another request to the shopping service : <http://localhost:8080/shopping/drinks> . Before sending the request, fill in an Authorization header with type Bearer and your access token value as follows:



The response should now be:



Nice to see things are working, but you are not quite there yet. At this stage, every authorized user is able to invoke all the methods of the protected resource. So, you might want to restrict things further.

For testing purposes, let’s first restrict the endpoint [http://localhost:8080/shopping/drink/{id}](http://localhost:8080/shopping/drink/%7bid%7d) so that it is only available to users with role admin.

Open the file ShoppingController.java (which is already in the project) and add the annotation @EnableGlobalMethodSecury(prePostEnabled) to it.

@RestController  
@EnableGlobalMethodSecurity(prePostEnabled = true)  
@RequestMapping(value="shopping")  
public class ShoppingController {

Annotate the method retrieveDrinks with: @PreAuthorize("hasRole('ROLE\_USER') or hasRole('ROLE\_ADMIN')") . It allows users with the role USER or ADMIN to invoke : http://localhost:8080/shopping/drinks

@RequestMapping(value="drinks", method=RequestMethod.*GET*)  
@PreAuthorize("hasRole('ROLE\_USER') or hasRole('ROLE\_ADMIN')")  
public List<Product> retrieveDrinks() {

For testing purposes, allow only users with role admin to query a specific drink: @PreAuthorize("hasRole('ROLE\_ADMIN')")

@RequestMapping(value="drink/{id}", method=RequestMethod.*GET*, produces = "application/json")  
@PreAuthorize("hasRole('ROLE\_ADMIN')")  
public Product retrieveDrink(@PathVariable("id") String productId) {

Likewise, annotate the method for create and update (and delete if implemented)

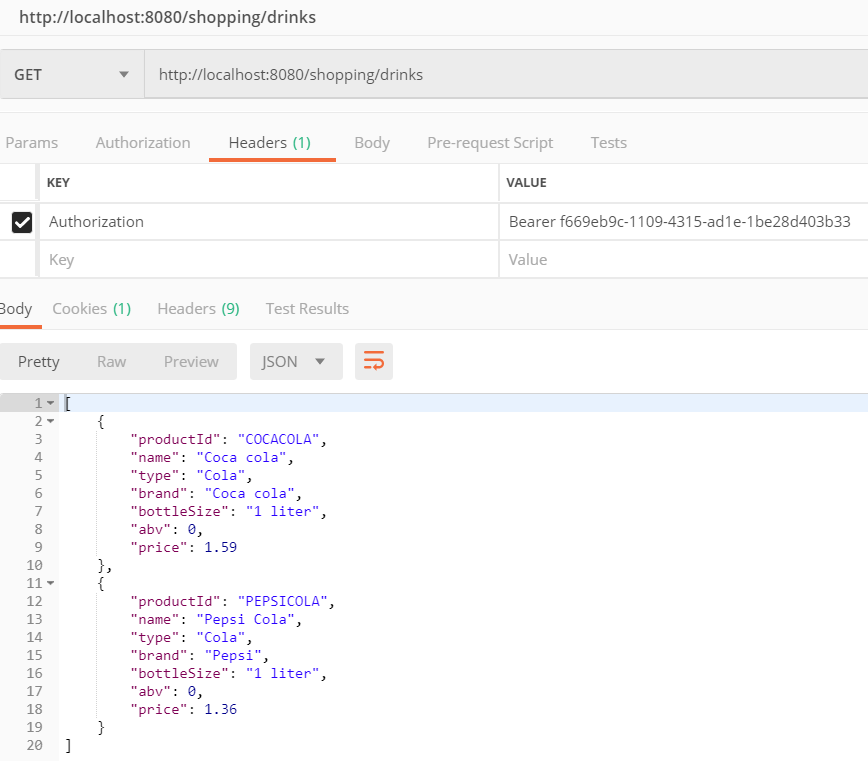
@RequestMapping(value = "drink/{id}", method=RequestMethod.*POST*)  
@PreAuthorize("hasRole('ROLE\_ADMIN')")  
@ResponseBody  
public ResponseEntity<Product> addDrink(@PathVariable("id") String productId, @RequestBody Product product) {

@RequestMapping(value = "drink/{id}", method=RequestMethod.*PUT*)  
@PreAuthorize("hasRole('ROLE\_ADMIN')")   
@ResponseBody  
public ResponseEntity<Product> modifyDrink(@PathVariable("id") String productId, @RequestBody Product product) {

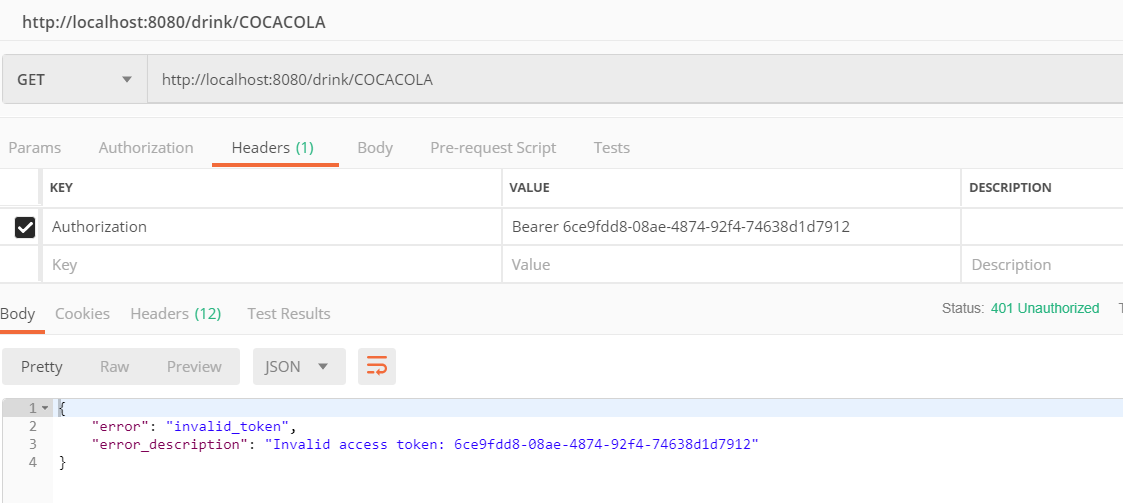
Rebuild and (re)start the shoppingservice.

Get a new access token for the user with role user (same way as described earlier above) : <http://localhost:8080/oauth/token> .

Copy the value for the access token to the Authorization header type Bearer in the request for <http://localhost:8080/shopping/drinks> (as shown below).

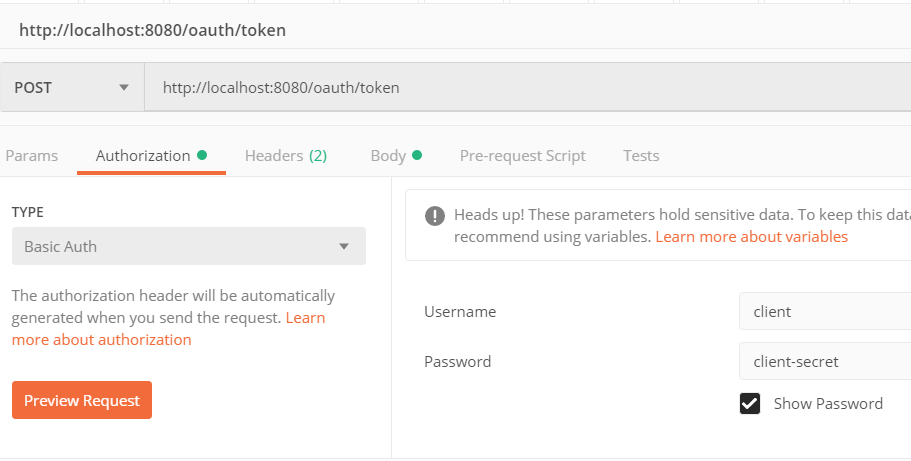


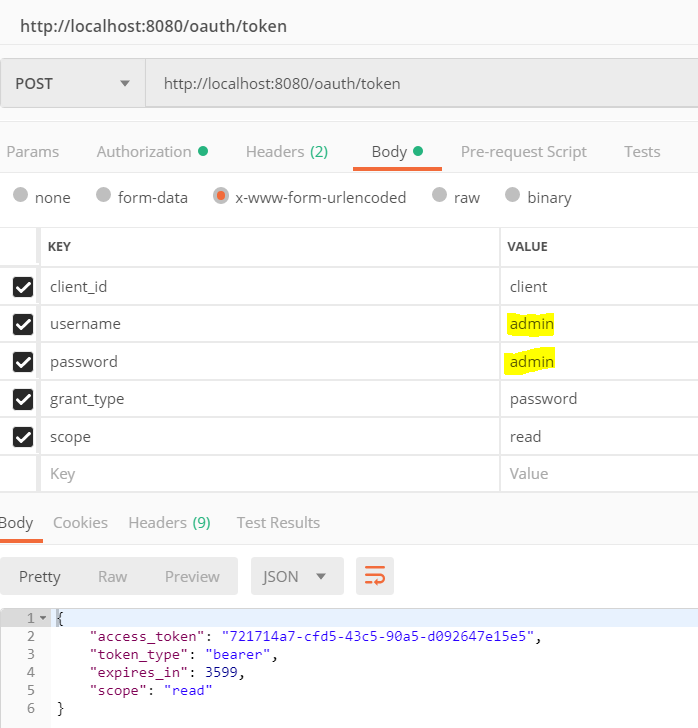
Make a new request to <http://localhost:8080/shopping/drink/COCACOLA> with the same Authorization header used for <http://localhost:8080/shopping/drinks> .



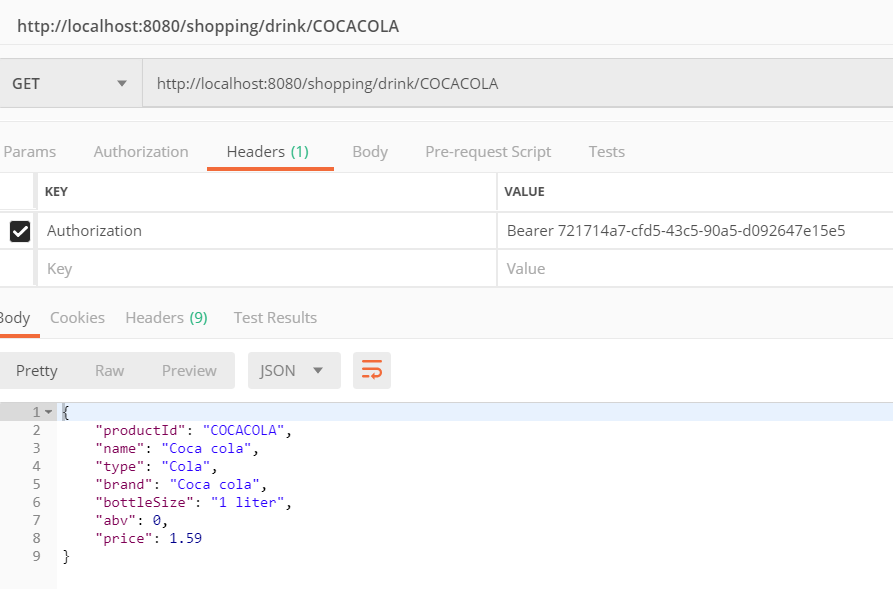
Result should be: 401 Unauthorized

Now get a new access token for the admin user:





Again, copy the access token value and place it in the Authorization header for the call to the protected resource, e.g. <http://localhost:8080/shopping/drinks/COCACOLA> .



Nice, now methods can be role based protected.

More fine grained control is possible in a similar way, as documented in: <https://docs.spring.io/spring-security/site/docs/3.0.x/reference/el-access.html>

