Integrate Keycloak With Spring Boot Step by Step

Updated on February 22, 2023 by Đạt Trần

Table of Contents [hide]

Recently, I had to configure an API to use Keycloak to authenticate and authorize users of a web application. After spending almost two weeks, I finally got the app up and running as required. If you are on the same boat, read on since I'll provide step by step (with pictures!) to help you save time on this setup so you can spend more time with actual development.

If you need help integrating Keycloak with Spring Boot 3, check this tutorial out.

API Requirements

There are some requirements as follow:

The spring boot API can create and login (return access token) to the caller
There are three roles (admin, moderator, member) in the app and each API endpoint can be guarded by a specific role require. There are also endpoints that don't require authorization (publicly available)
Endpoints that are accessible by member are also accessible by admin and moderator. Endpoints that are accessible by moderator are accessible by admin

Code repository

If you are the impatient kind (like me), you can skip the post and go directly to the source code (available on Github) to start tinkering with the application here.

However, when you are stuck, feel free to go back to this post to find the missing pieces.

Quick Keycloak setup with docker compose

If you have a Keycloak instance up and running, you can skip this part. However, in case you haven't, use the docker-compose file below to quickly set up Keycloak:

YAML

```yaml
version: '3'

services:
  keycloak:
    container_name: keycloak
    image: jboss/keycloak:15.0.2
    restart: always
    env_file: ./keycloak.env
    depends_on:
      - keycloak_db
```

```
11    volumes:
12      - ./realm.json:/tmp/realm.json
13    ports:
14      - "18080:8080"
15
16
```

The keycloak.env contains some environment variables:

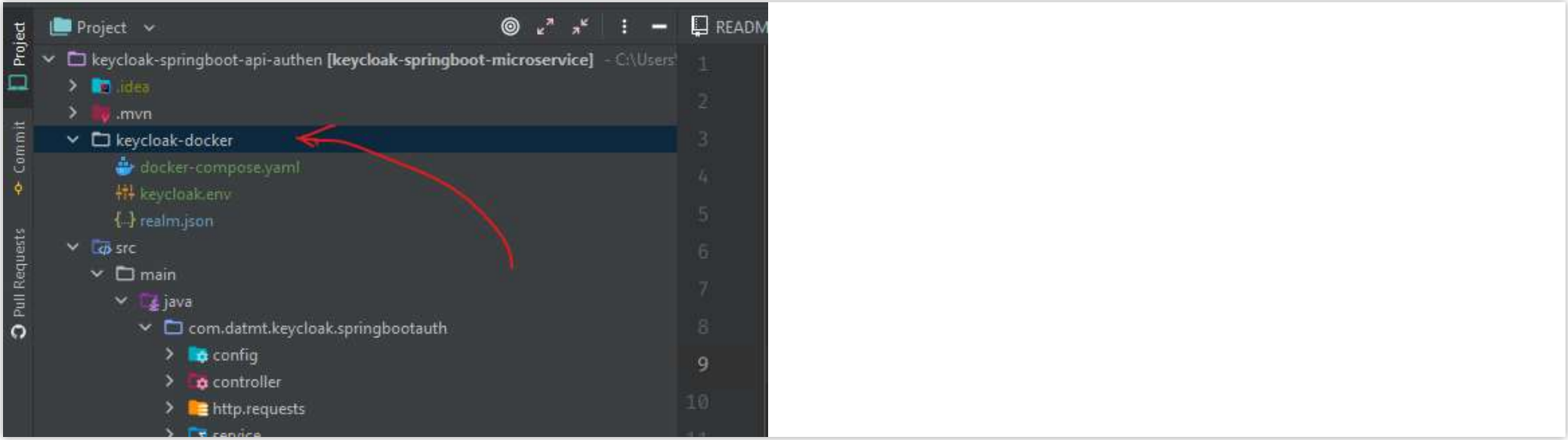
Textile

```
1 KEYCLOAK_USER=kc_dev
2 KEYCLOAK_PASSWORD=kc_dev1231232
3 KEYCLOAK_IMPORT="-Dkeycloak.profile.feature.upload_scripts=enabled"
4 DB_VENDOR=mariadb
5 DB_ADDR=keycloak_db:3306
6 DB_DATABASE=keycloak_1
```

7

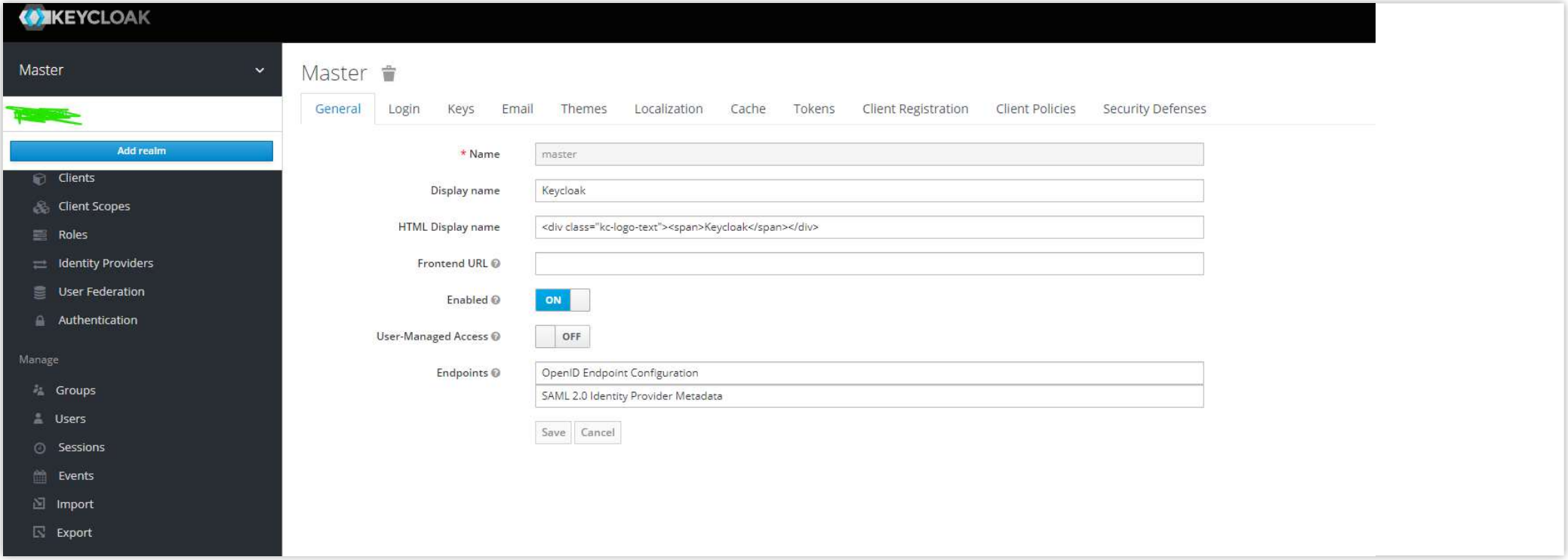
Now run docker-compose up -d then you should be able to access keycloak at http://localhost:18080 in a few minutes (docker may need to pull the images from its registry, which depends largely on the speed of your connection).

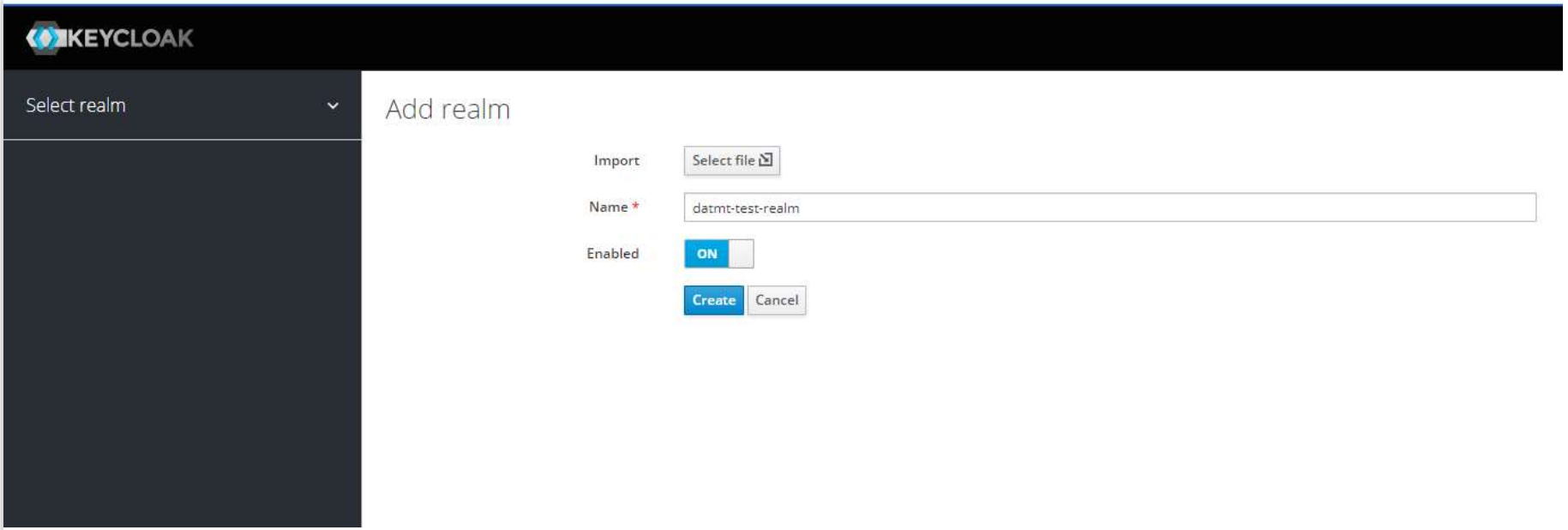
For the lazy guys, please go to this location in the repo and hit docker-compose up -d



Once, you have access to Keycloak, login with the id and password specified in the environment file (kc_dev and kc_dev1231232).

In the beginning, there should be only one realm, you need to click on Add realm to create a new realm.





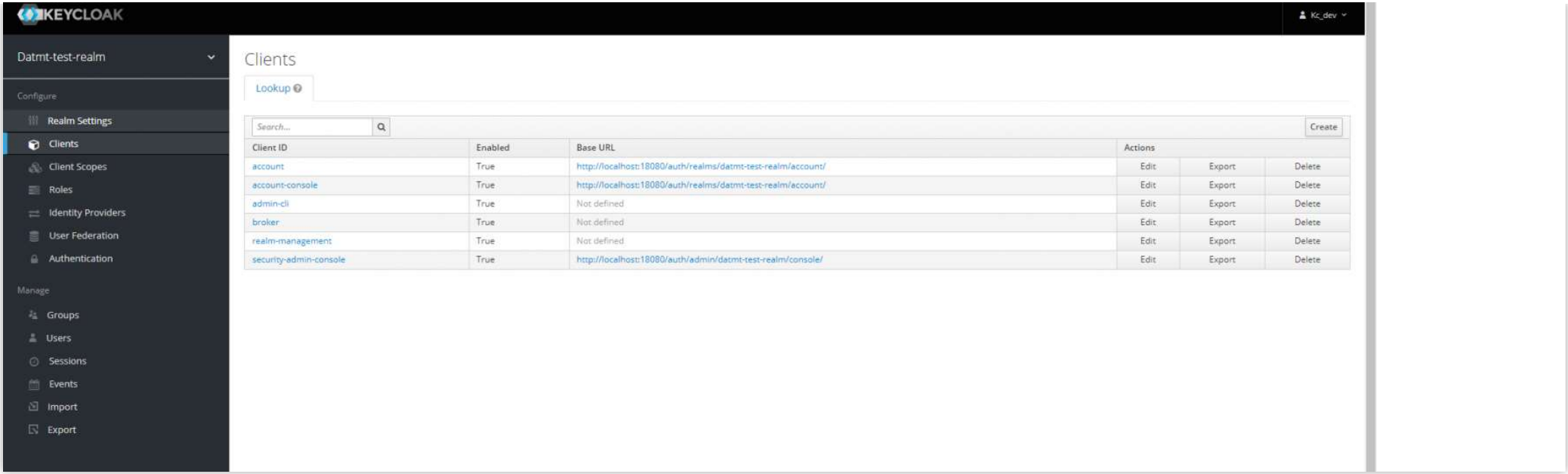
Create a new realm in keycloak

Now the keycloak instance is ready. It’s time to create and configure a client.

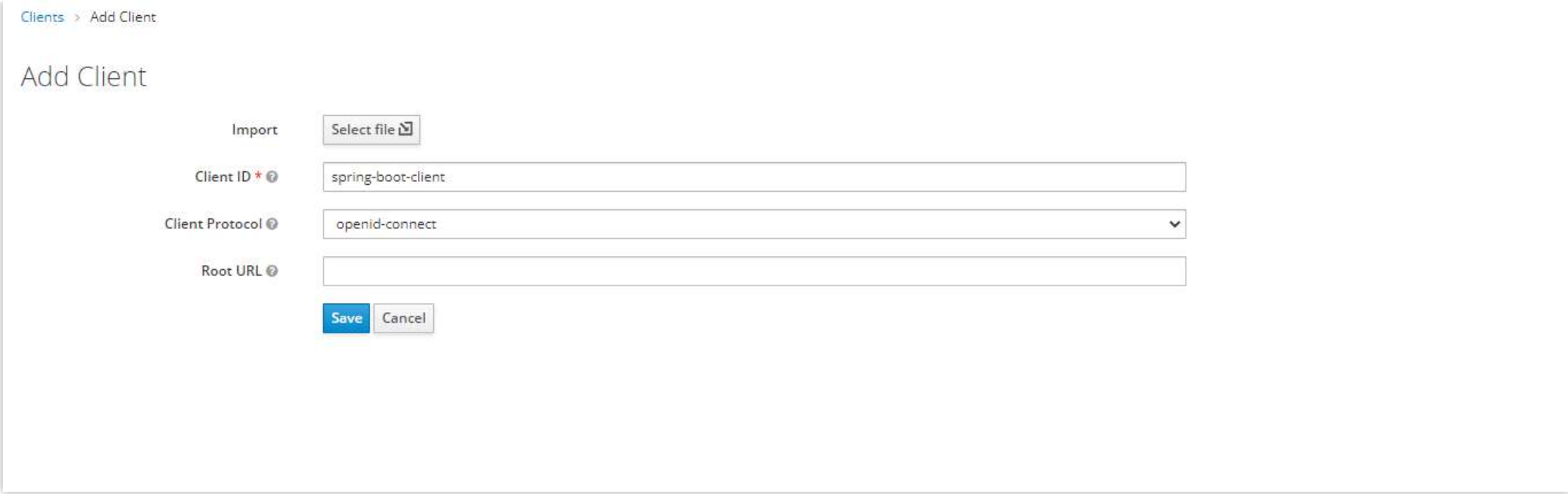
Keycloak client configuration

The client’s configuration is very important so you need to pay close attention to this part.

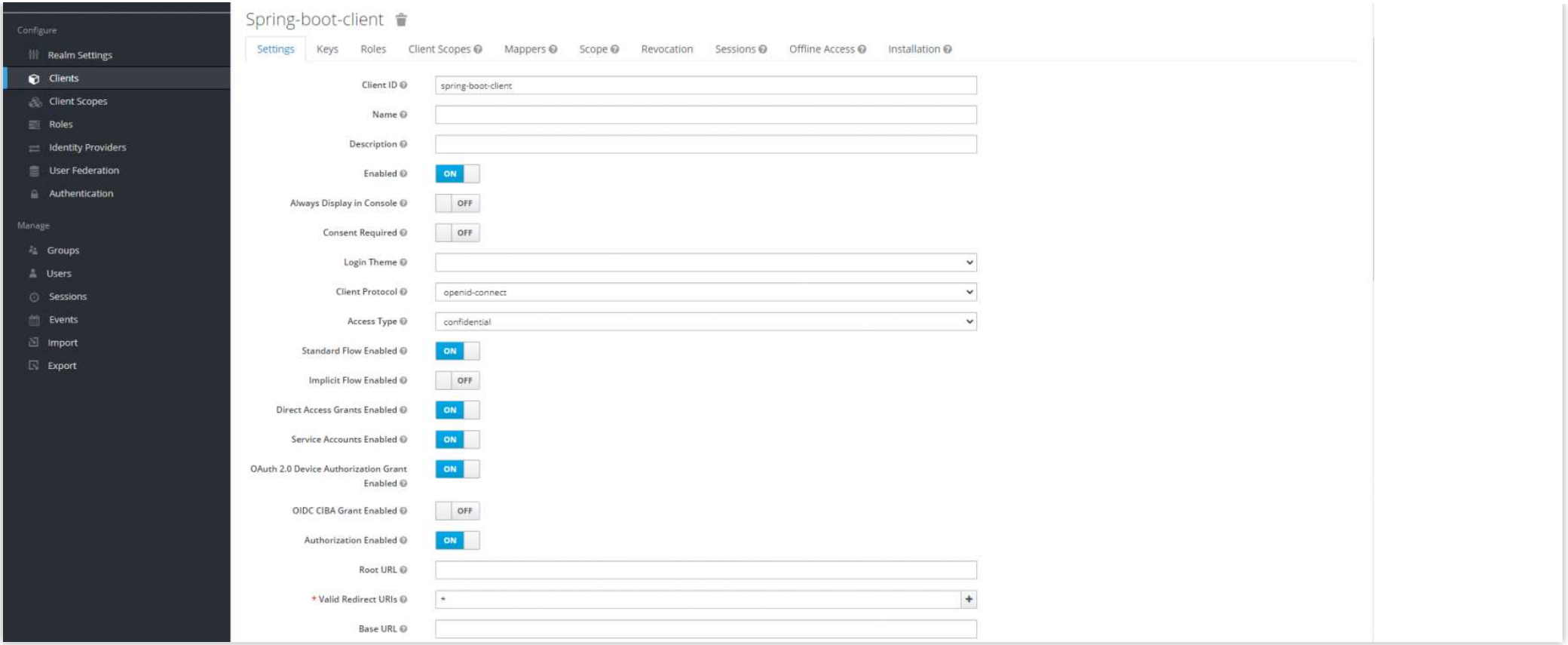
At the beginning, there are some default clients:

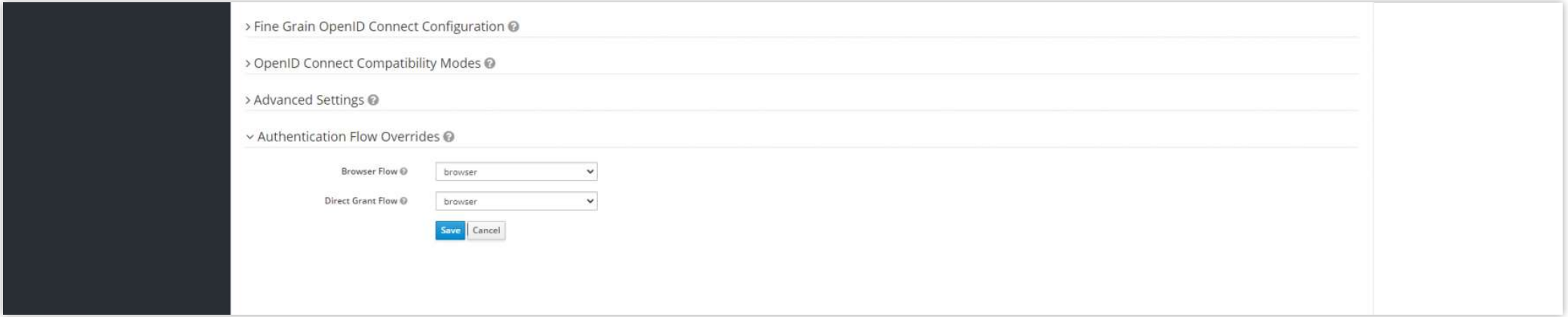


You can click on the Create button on the right to create a new client like this:



And configure the client like this:





Now, it’s time to create roles for the client. As you can remember, we have three roles:

admin

moderator

member

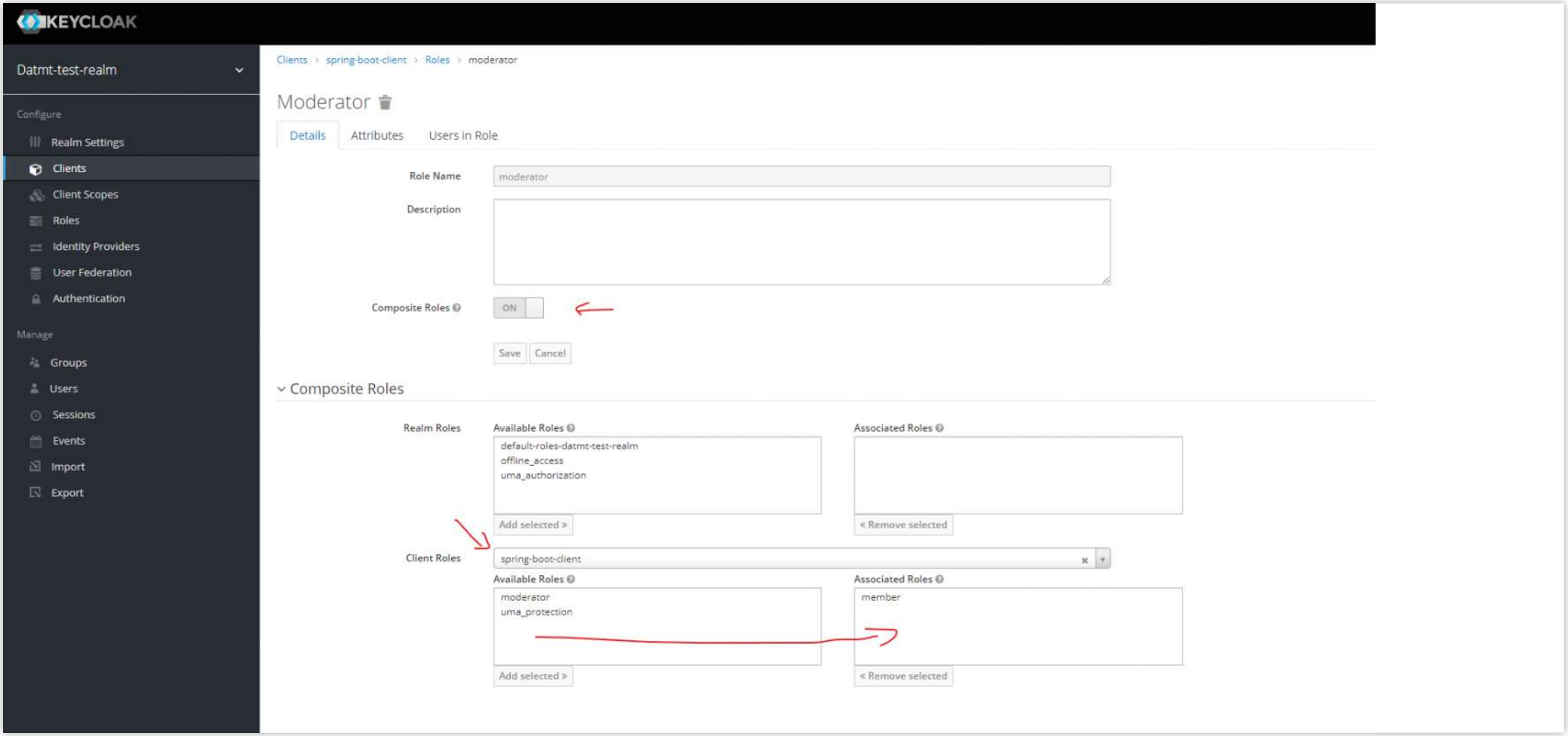
From the requirement at the beginning, we know that admin and moderator are composite roles while member is not a composite role.

A composite role is a role that consists of one or more other roles

Let’s switch to the Roles tab and click on Add role

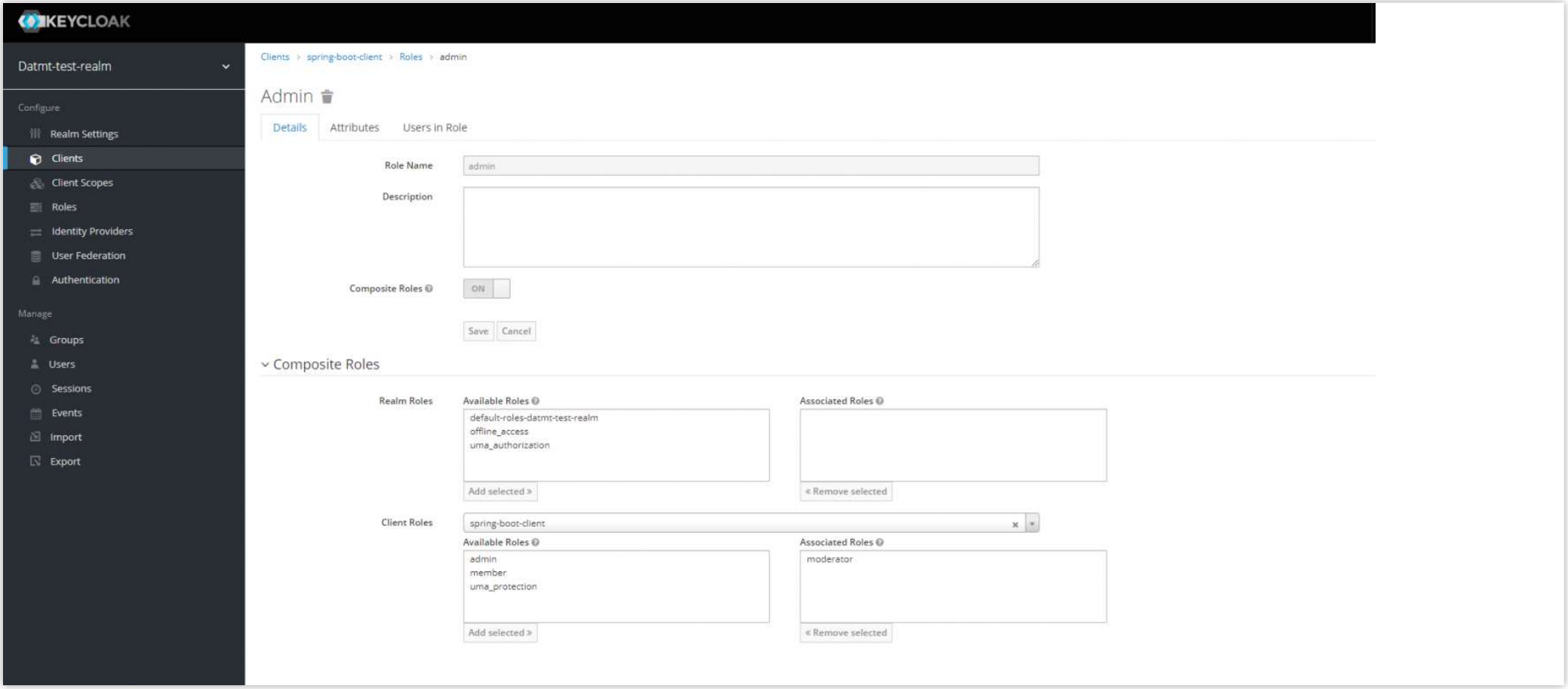
After creating the role, you will see there is a switch to turn that role into a composite role. For the member role, we don’t need that.

However, as mentioned above, admin and moderator are composite roles, we need to turn that option on:



As you can see in this case, when creating moderator role, I turned the composite role on. In the Client roles select box, I select this client (spring-boot-client).

Let’s do the same for the admin role.



You don’t need to add the member role to the associated roles box since moderator includes member already.

Spring boot application configuration

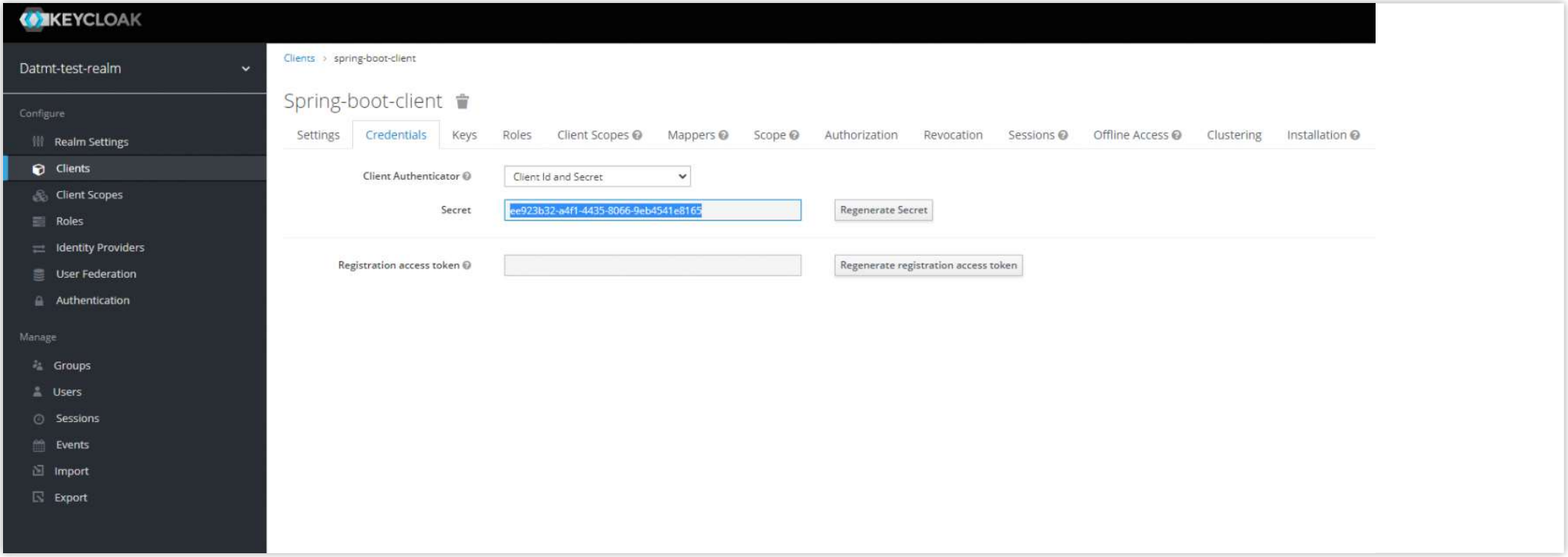
Let’s configure the application.properties file.

Textile

```
#keycloak
keycloak.realm=datmt-test-realm
keycloak.auth-server-url=http://localhost:18080/auth/
#use external in production
keycloak.ssl-required=none

#name of the client
```

All the fields in this configuration file are self-explanatory. You may wonder where to get the secret? It's in the Credentials tab of the client:



Also, for Keycloak to guard your endpoints, you need to create a configuration file like this:

Java

```
1 package com.datmt.keycloak.springbootauth.config;
```

```
2
3 import org.keycloak.adapters.KeycloakConfigResolver;
4 import org.keycloak.adapters.springboot.KeycloakSpringBootConfigResolver
5 import org.keycloak.adapters.springsecurity.authentication.KeycloakAuthe
6 import org.keycloak.adapters.springsecurity.config.KeycloakWebSecurityCo
7 import org.springframework.beans.factory.annotation.Autowired;
8 import org.springframework.context.annotation.Bean;
9 import org.springframework.context.annotation.Configuration;
10import org.springframework.security.config.annotation.authentication.bui
11import org.springframework.security.config.annotation.method.configurati
12import org.springframework.security.config.annotation.web.builders.HttpS
13import org.springframework.security.config.annotation.web.configuration.
14import org.springframework.security.core.authority.mapping.SimpleAuthori
15import org.springframework.security.core.session.SessionRegistryImpl;
16import org.springframework.security.web.authentication.session.RegisterS
```

```
17import org.springframework.security.web.authentication.session.SessionAu
18
19@Configuration
20@EnableWebSecurity
21@EnableGlobalMethodSecurity(prePostEnabled = true, securedEnabled = true
22public class KeycloakSecurityConfig extends KeycloakWebSecurityConfigure
23
24    @Override
25    protected void configure(HttpSecurity http) throws Exception {
26        super.configure(http);
27        http.authorizeRequests()
28            .antMatchers("/public/**").permitAll()
29            .antMatchers("/member/**").hasAnyRole("member")
30            .antMatchers("/moderator/**").hasAnyRole("moderator")
31            .antMatchers("/admin/**").hasAnyRole("admin")
```

```
32                .anyRequest()
```

Here, you can see that I've configured that:

Paths begins with /admin requrie admin role

Paths begin with /member require member role

Paths begin with /moderator require moderator role

Paths begin with /public is accessible by all

Let's create a controller and test our config:

Java

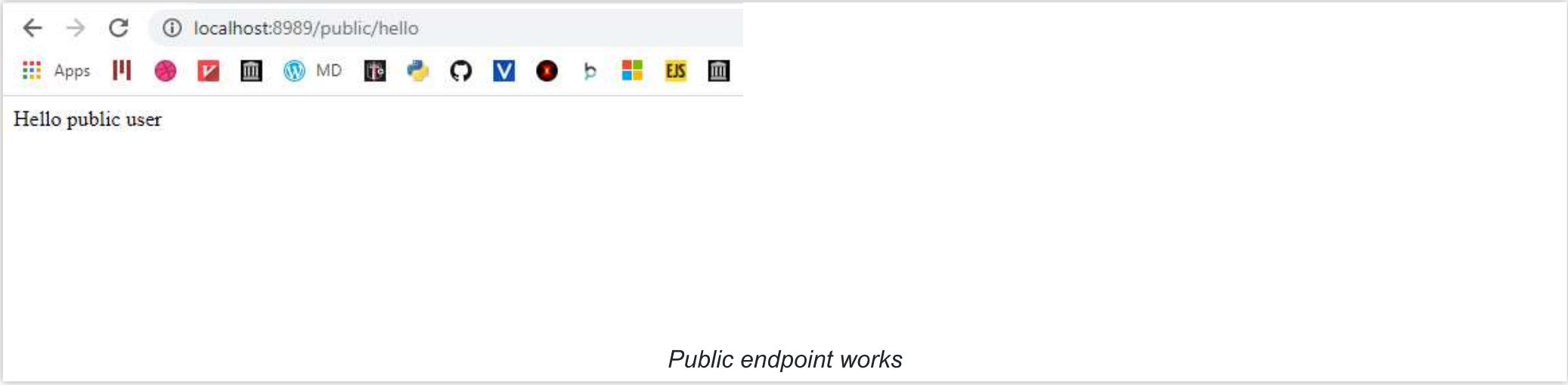
```java
package com.datmt.keycloak.springbootauth.controller;

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestHeader;
import org.springframework.web.bind.annotation.RequestMapping;
```

```
7 import org.springframework.web.bind.annotation.RequestMethod;
8 import org.springframework.web.bind.annotation.RestController;
9 import javax.annotation.security.RolesAllowed;
10
11@RestController
12public class HelloController {
13
14    @GetMapping("/public/hello")
15    public ResponseEntity<String> helloPublic() {
16        return ResponseEntity.ok("Hello public user");
17    }
18
19    @GetMapping("/member/hello")
20    public ResponseEntity<String> helloMember() {
21        return ResponseEntity.ok("Hello dear member");
```

22

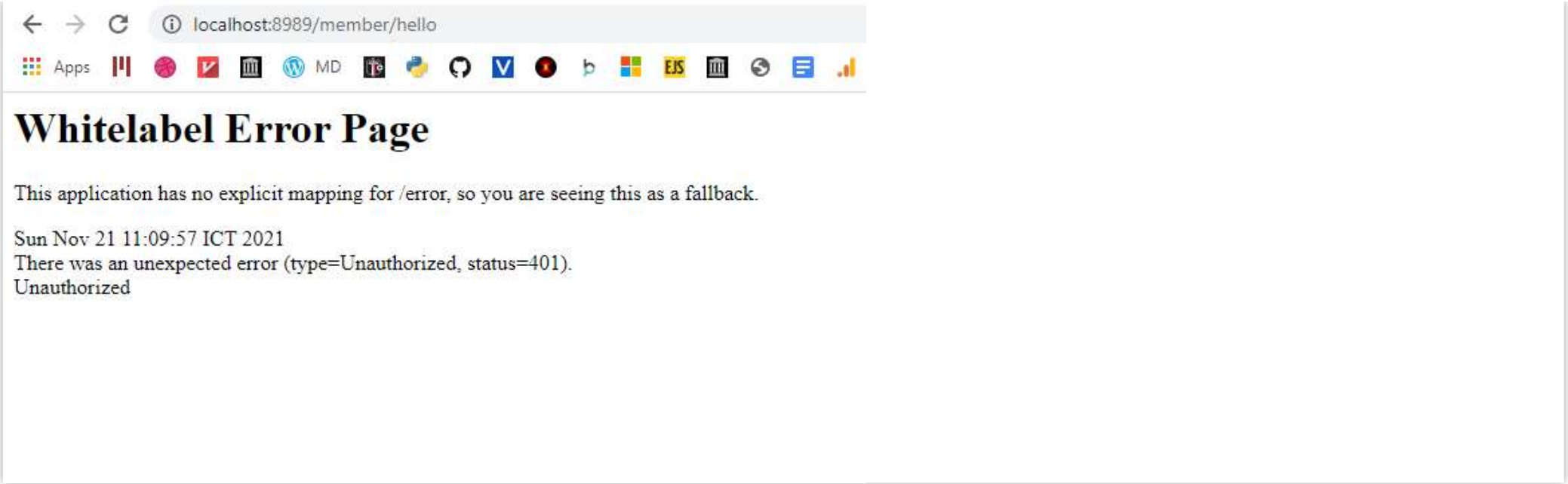
Let’s try the public endpoint:



Public endpoint works

As you can see, it works as expected. The path begins with /public so anyone can access without any authentication.

Now let’s try the path for member:

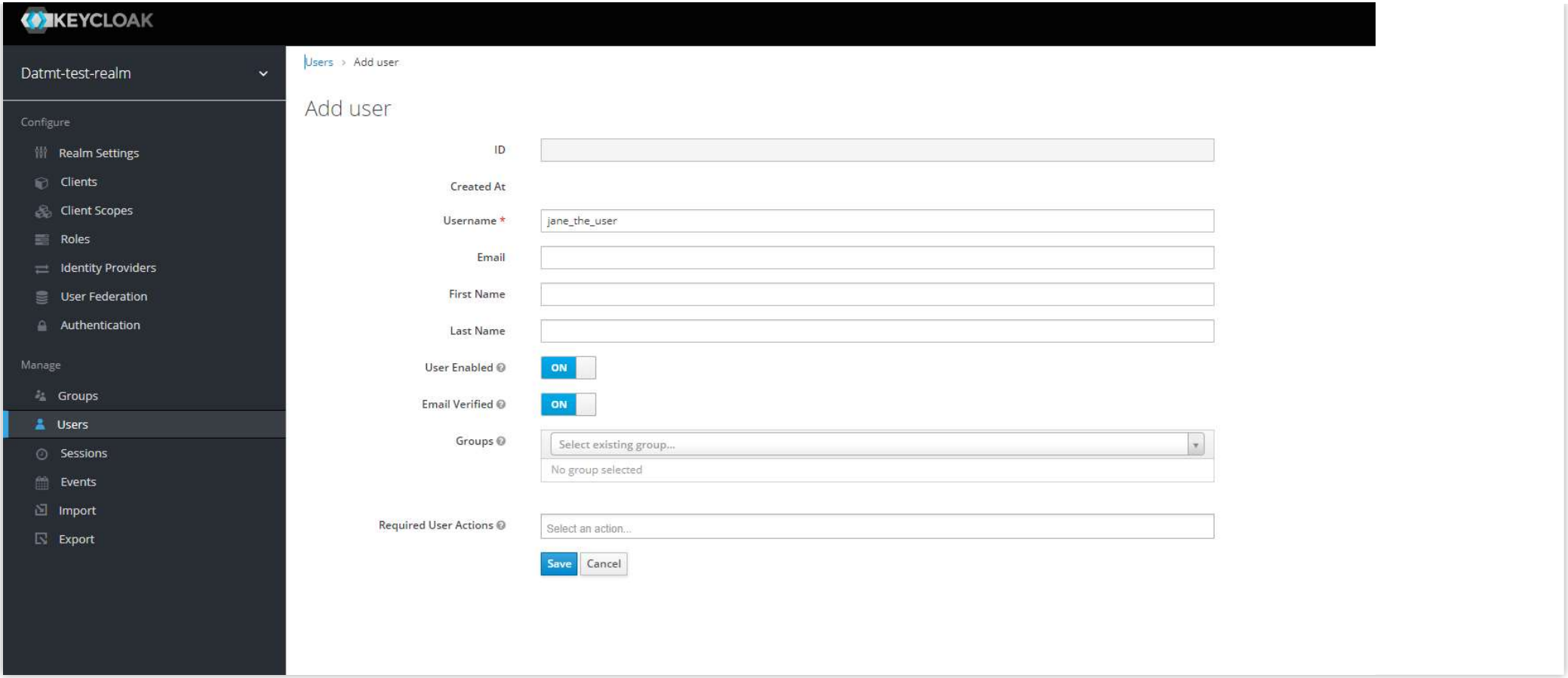


and also admin:

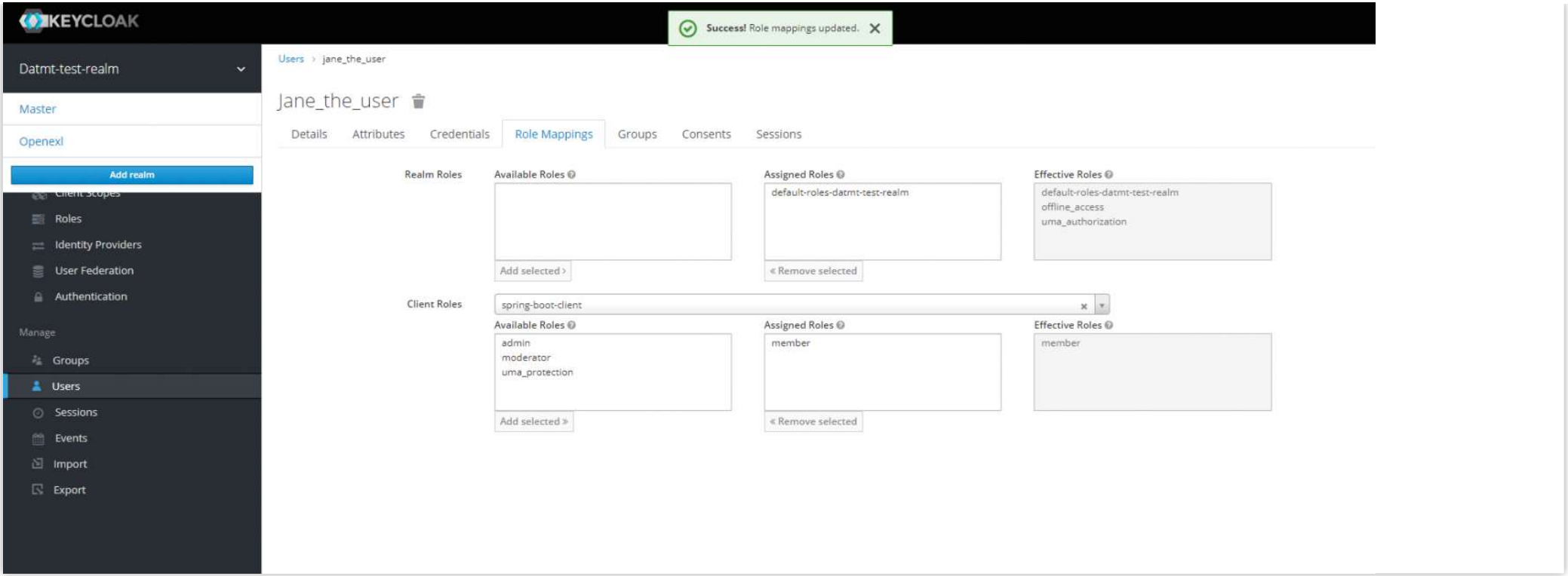
You can see that since the request didn't present any kind of credentials, the response is 401.

To make sure this works, let's create users with specific roles and test if they can access the endpoints with his token.

Let's first create an user with member role:

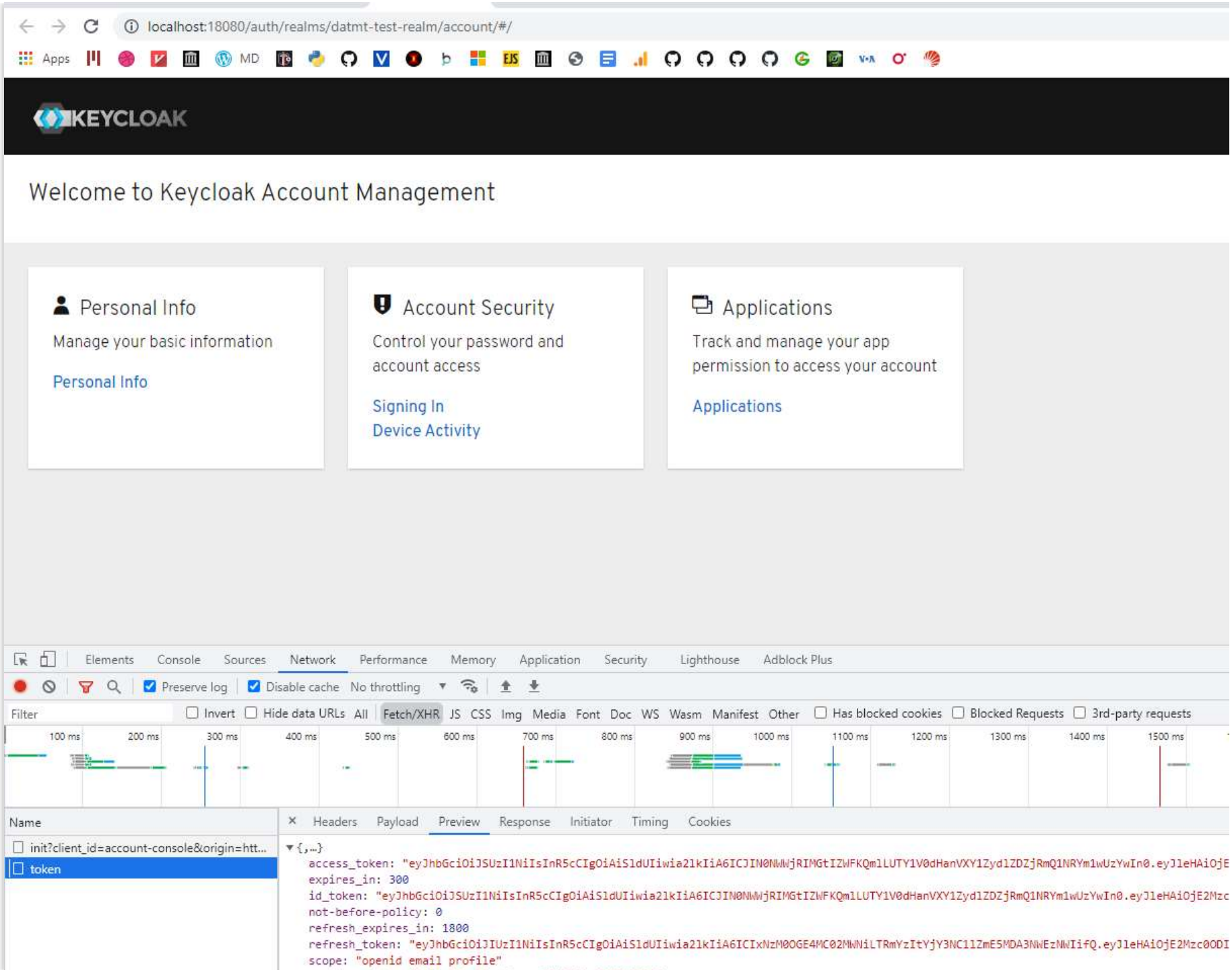


After creating the user, go to Role Mappings tab, select spring-boot-client and add member to user's list of roles



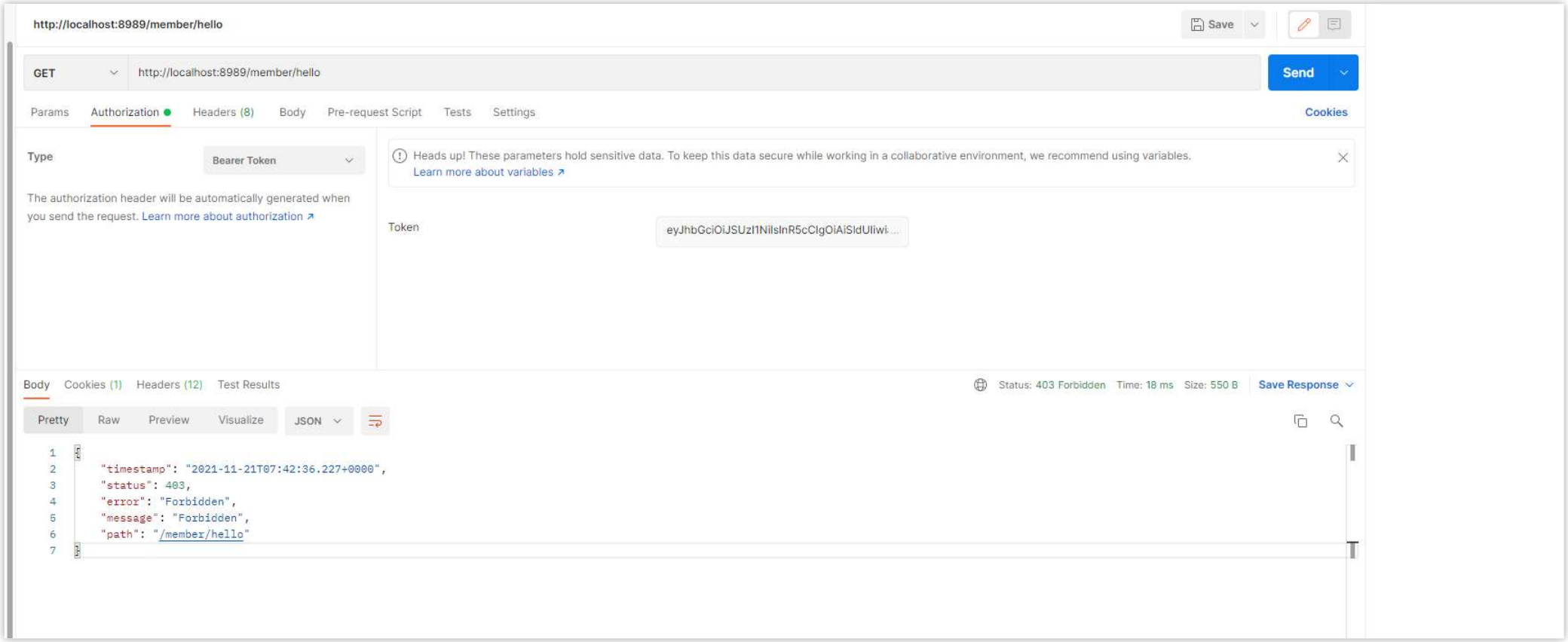
Now, to quickly get the token for this user, go back to the details tab and click on Impersonate:

In the new tab that appears, open the network tab of the developer console, reload and copy the access token:


localhost:18080/auth/realms/datmt-test-realm/account/#/
KEYCLOAK
Welcome to Keycloak Account Management
Personal Info
Manage your basic information
Personal Info
Account Security
Control your password and account access
Signing In
Device Activity
Applications
Track and manage your app permission to access your account
Applications
Elements Console Sources Network Performance Memory Application Security Lighthouse Adblock Plus
Preserve log Disable cache No throttling
Filter Invert Hide data URLs All Fetch/XHR JS CSS Img Media Font Doc WS Wasm Manifest Other Has blocked cookies Blocked Requests 3rd-party requests
Name
init?client_id=account-console&origin=htt...
token
Headers Payload Preview Response Initiator Timing Cookies
{,…}
access_token: "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJIN0NWWjRIMGtIZWFKQmlLUTY1V0dHanVXY1ZydlZDZjRmQ1NRYm1wUzYwIn0.eyJleHAiOjE
expires_in: 300
id_token: "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJIN0NWWjRIMGtIZWFKQmlLUTY1V0dHanVXY1ZydlZDZjRmQ1NRYm1wUzYwIn0.eyJleHAiOjE2Mzc
not-before-policy: 0
refresh_expires_in: 1800
refresh_token: "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICIxNzM0OGE4MC02MWNiLTRmYzItYjY3NC11ZmE5MDA3NWEzNWIifQ.eyJleHAiOjE2Mzc0ODI
scope: "openid email profile"

```
session_state: "5f5ded8c-71cd-4eea-86d6-1be2b772f620"
token_type: "Bearer"
```


Now, use that token to make a request in postman:



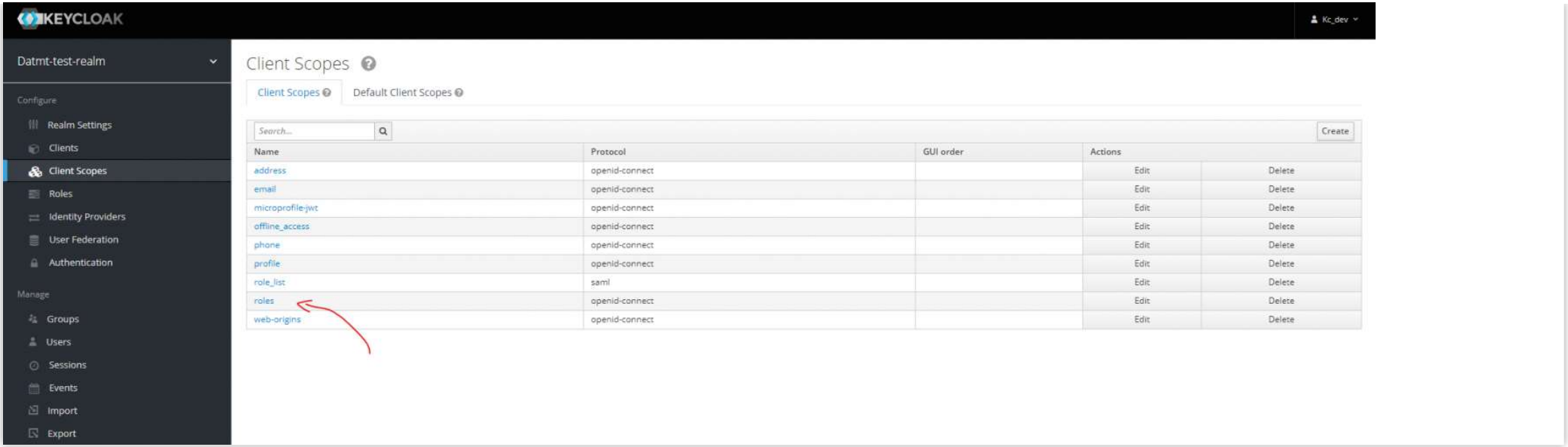
You expected that to be a 200 response, right? But what I got was 403. That's better than 401 since we can authenticate with the token but somehow, the user was not authorized to access the endpoint.

What’s the missing configuration?

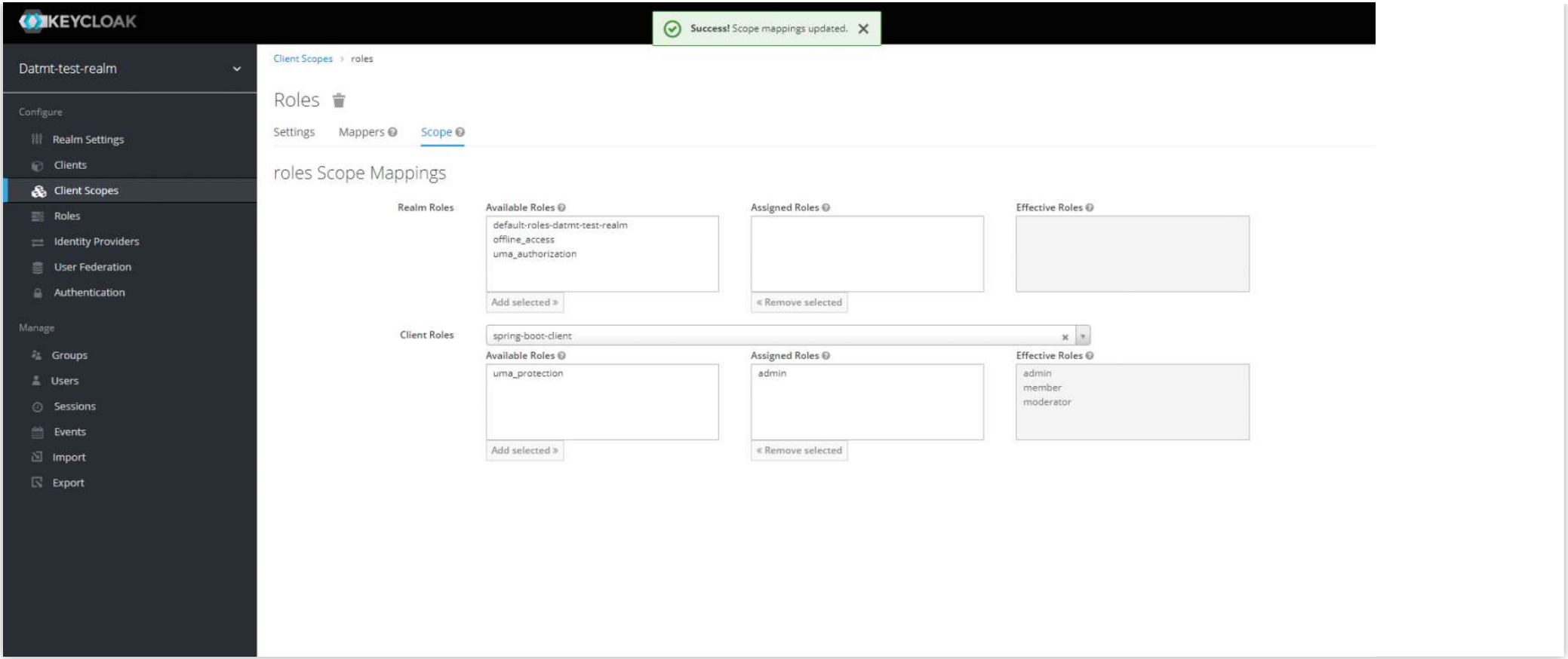
If you decode the token using sites like jwt.io, you will see that even though the user has role member, it didn’t show in the token:

```
{
  "exp": 1637480705,
  "iat": 1637480405,
  "jti": "8de803a2-f154-47b4-ab54-90e0a215cc14",
  "iss": "http://localhost:18080/auth/realms/datmt-
test-realm",
  "aud": "account",
  "sub": "4e14de83-f723-4bfd-9bc0-ab9489e59cf1",
  "typ": "Bearer",
  "azp": "account-console",
  "nonce": "b3eb849b-4119-47b3-9163-0760855e418e",
  "session_state": "5f5ded8c-71cd-4eea-86d6-
1be2b772f620",
  "acr": "0",
  "resource_access": {
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links"
      ]
    }
  },
  "scope": "openid email profile",
  "sid": "5f5ded8c-71cd-4eea-86d6-1be2b772f620",
  "email_verified": true,
  "preferred_username": "jane_the_user"
}
```


It turned out, you also need to configure client scopes so the roles appear on the token.



Then go to Scopes and add all roles of the client spring-boot-client

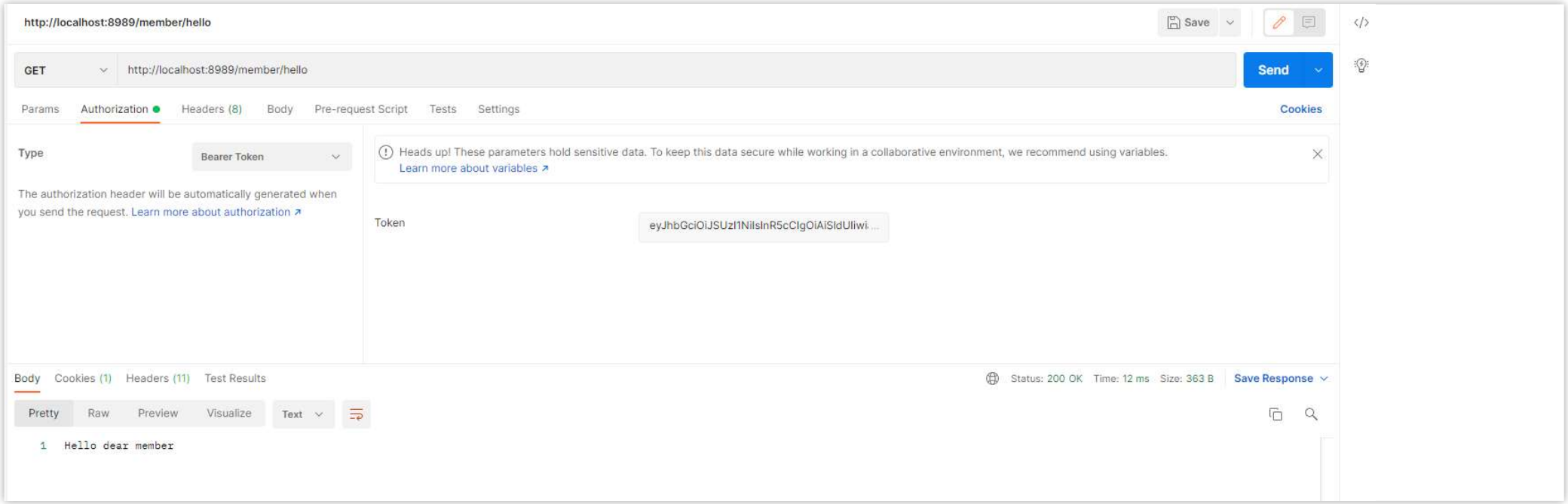


Now, let’s impersonate the user again. Now, her roles should be visible on the token decoding page:

PAYLOAD: DATA


```
{
  "exp": 1637481275,
  "iat": 1637480975,
  "jti": "4238c38d-8b1b-45f4-91b7-befb8329d3a2",
  "iss": "http://localhost:18080/auth/realms/datmt-
test-realm",
  "aud": [
    "spring-boot-client",
    "account"
  ],
  "sub": "4e14de83-f723-4bfd-9bc0-ab9489e59cf1",
  "typ": "Bearer",
  "azp": "account-console",
  "nonce": "f994383f-ea81-44ea-846d-06c60c4b8ce3",
  "session_state": "5f5ded8c-71cd-4eea-86d6-
1be2b772f620",
  "acr": "0",
  "resource_access": {
    "spring-boot-client": {
      "roles": [
        "member"
      ]
    },
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links"
      ]
    }
  },
  "scope": "openid email profile",
  "sid": "5f5ded8c-71cd-4eea-86d6-1be2b772f620",
  "email_verified": true,
  "preferred_username": "jane_the_user"
}
```

Let’s make the same request to them member’s endpoint again but with the new token, sure enough, it worked:



You can try with other roles, they should work too.

Guarding API with RolesAllowed

Guarding endpoint with path prefixes is OK and sometimes that's a clean solution. However, there are times you may need a finer tune to your access policy.

In such case, you can use RolesAllowed to specify which role can access a specific endpoint, no matter what its path prefix is.

Java

```java
@RolesAllowed("member")
@GetMapping("/other/hello")
public ResponseEntity<String> helloCustom() {
```

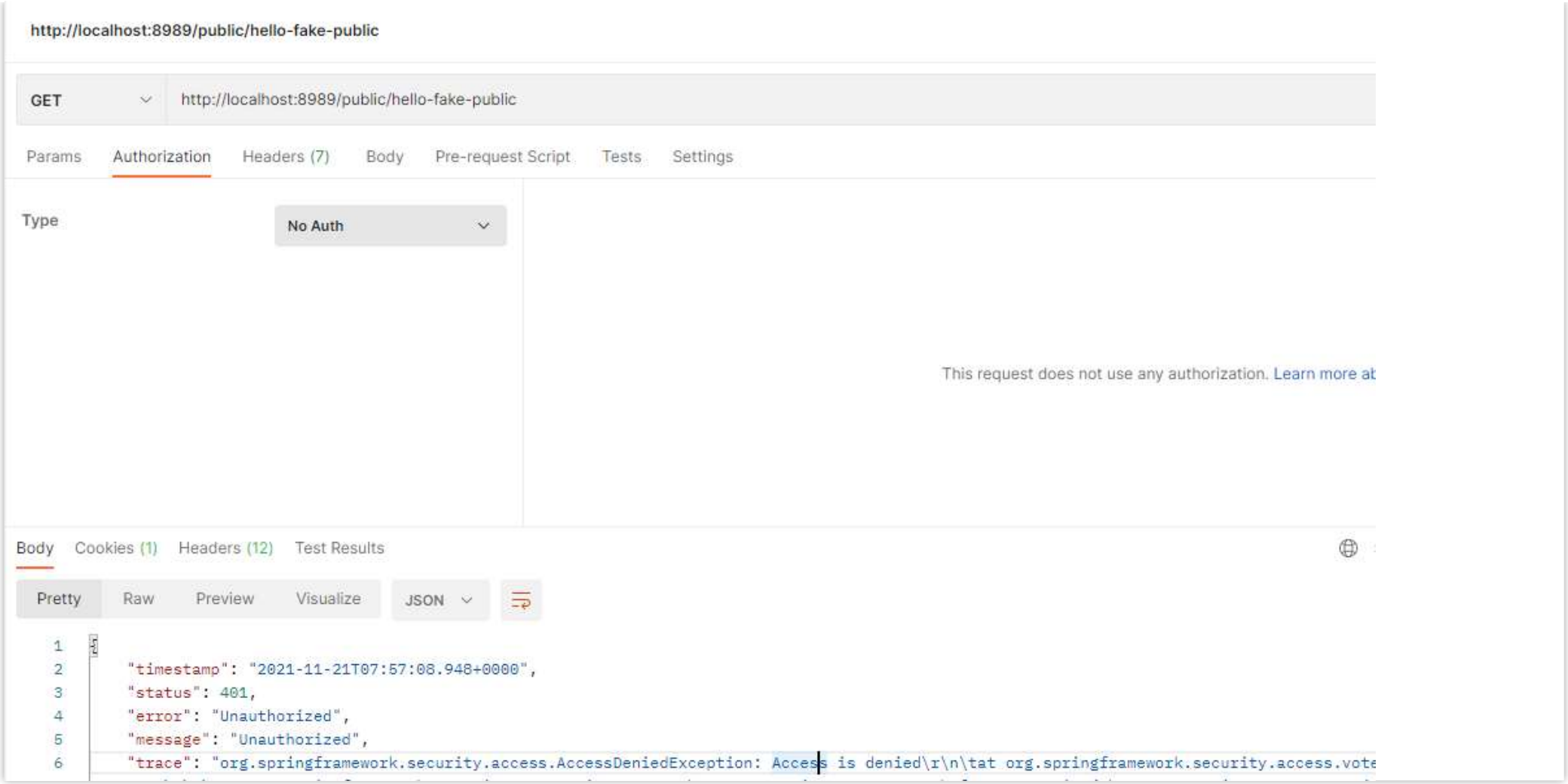
You can guess from the annotation, anyone with role member can access this endpoint.

What if the path is prefix but /public?

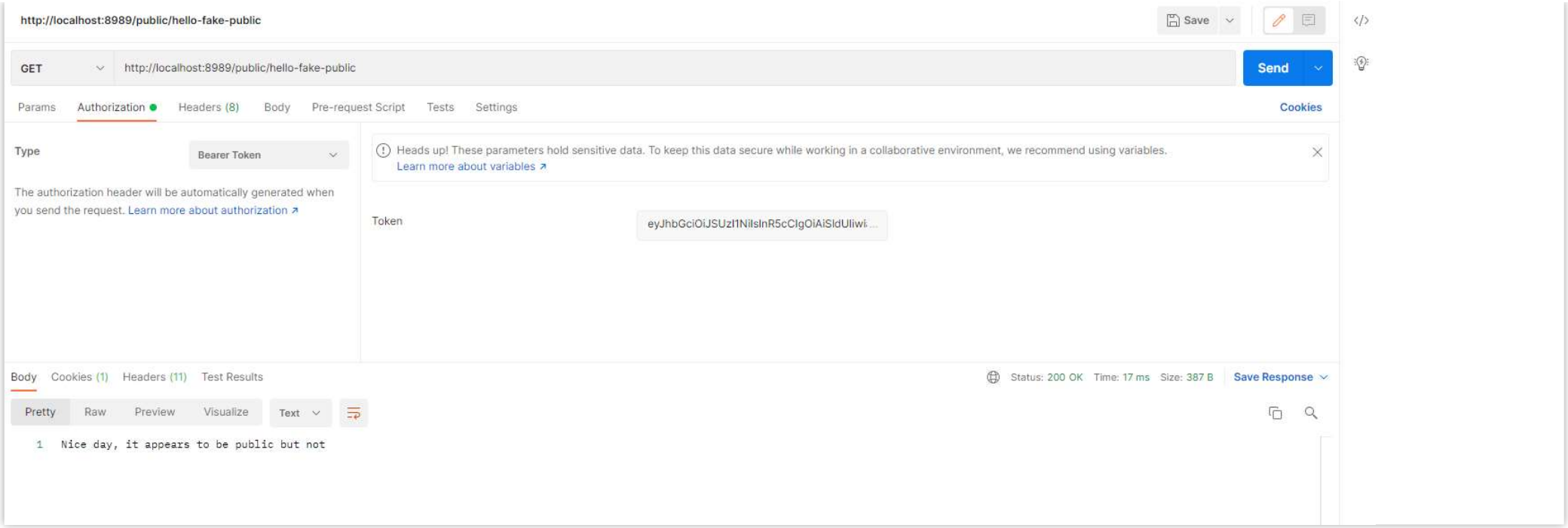
Java

```
1    @RolesAllowed("member")
2    @GetMapping("/public/hello-fake-public")
```

If you try to access the URL, even though the prefix is public, you will get a 401 without a token:



But if a user with member role, the access is granted:

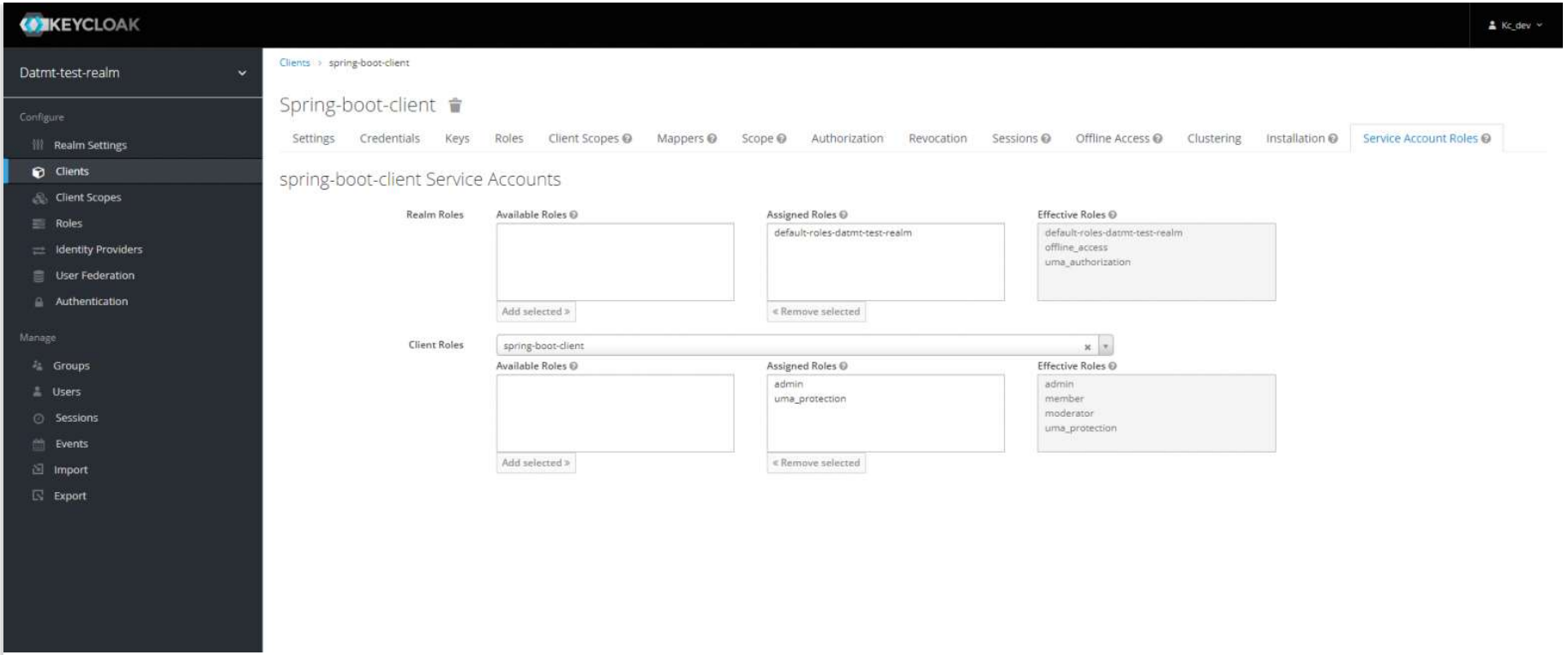


Create account and login using Keycloak Admin Client

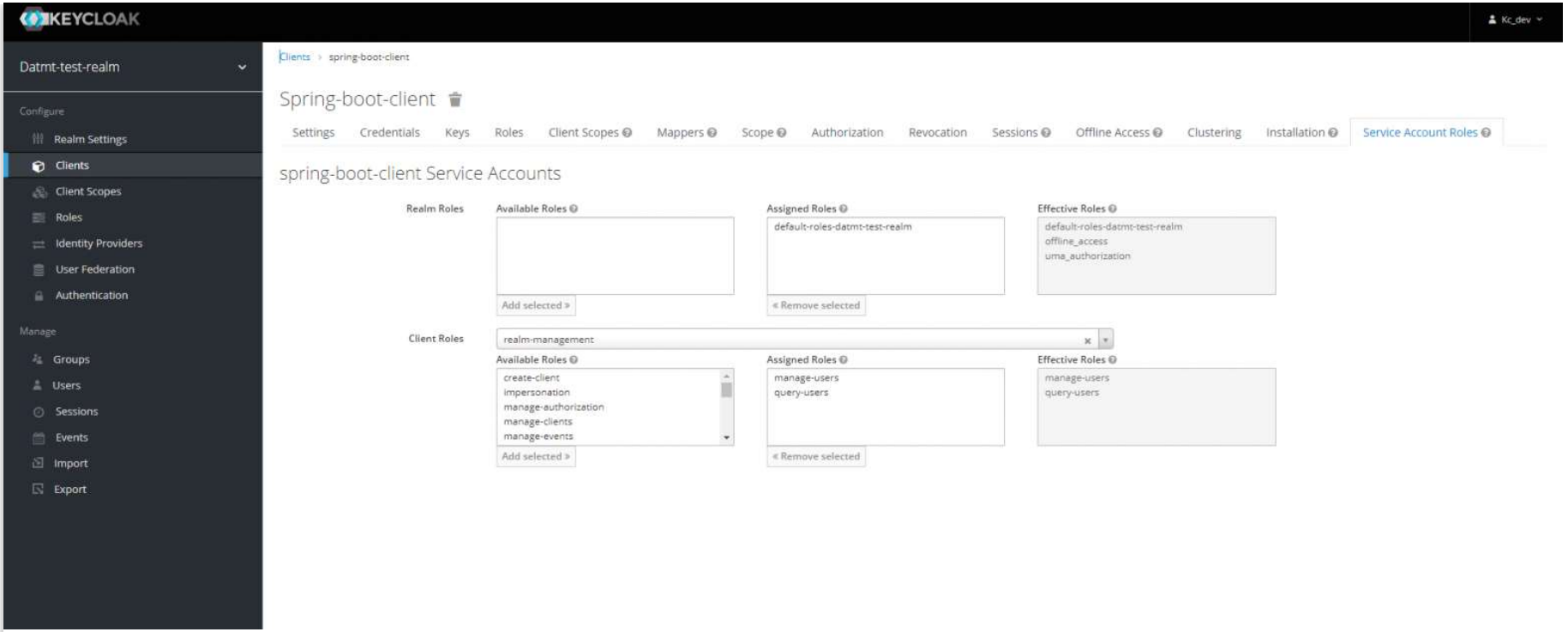
Now the API can talk to keycloak to authenticate and authorize users, let's also integrate user login and creation to the API so we can get the token via the API.

Before doing anything with the code, let's add some more configuration to the client so it has what it needs to manage user.

First, go to the client's Service Account Role tab and configure as follow:



Configure role for the service account



Add manager users, query users role for the client's service account

Now, the client is able to manage users.

Let's add the keycloak-admin-client package to maven. You need this to enable spring boot to operate Keycloak's admin-related tasks.

XML

```
1         <dependency>
2             <groupId>org.keycloak</groupId>
3             <artifactId>keycloak-admin-client</artifactId>
```

Next add a KeycloakProvider class:

Java

```java
@Configuration
@Getter
public class KeycloakProvider {

    @Value("${keycloak.auth-server-url}")
    public String serverURL;
    @Value("${keycloak.realm}")
    public String realm;
    @Value("${keycloak.resource}")
```

```
10    public String clientID;
11    @Value("${keycloak.credentials.secret}")
12    public String clientSecret;
13
14    private static Keycloak keycloak = null;
15
16    public KeycloakProvider() {
17    }
18
19    public Keycloak getInstance() {
20        if (keycloak == null) {
21
22            return KeycloakBuilder.builder()
23                    .realm(realm)
24                    .serverUrl(serverURL)
```

```
25                      .clientId(clientID)
26                      .clientSecret(clientSecret)
27                      .grantType(OAuth2Constants.CLIENT_CREDENTIALS)
28                      .build();
29          }
30          return keycloak;
31      }
32
```

Then a service to create and log in user:

Java

```
1 @Service
2 public class KeycloakAdminClientService {
3     @Value("${keycloak.realm}")
4     public String realm;
```

```
5
6     private final KeycloakProvider kcProvider;
7
8
9     public KeycloakAdminClientService(KeycloakProvider keycloakProvider)
10        this.kcProvider = keycloakProvider;
11    }
12
13    public Response createKeycloakUser(CreateUserRequest user) {
14        UsersResource usersResource = kcProvider.getInstance().realm(rea
15        CredentialRepresentation credentialRepresentation = createPasswo
16
17        UserRepresentation kcUser = new UserRepresentation();
18        kcUser.setUsername(user.getEmail());
19        kcUser.setCredentials(Collections.singletonList(credentialRepres
```

```
20          kcUser.setFirstName(user.getFirstname());
21          kcUser.setLastName(user.getLastname());
22          kcUser.setEmail(user.getEmail());
23          kcUser.setEnabled(true);
24          kcUser.setEmailVerified(false);
```

Finally, create a controller to create and login user:

Java

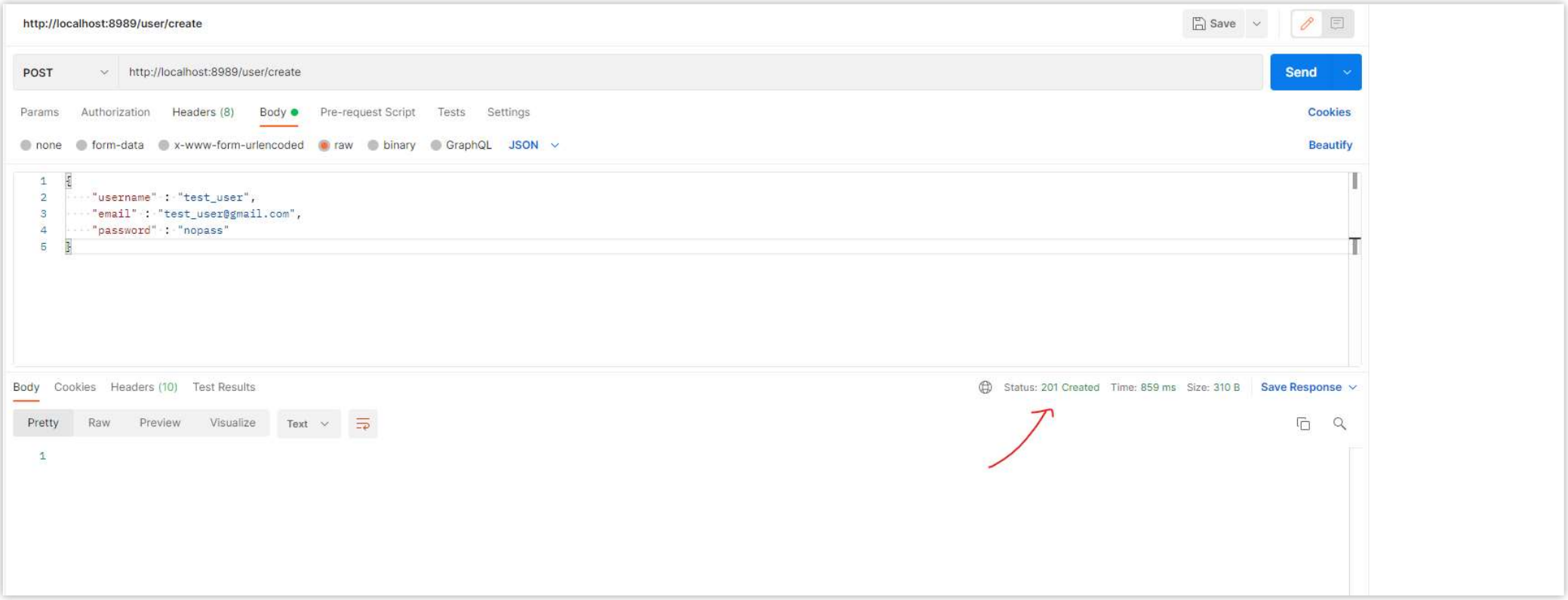
```
1 @RestController
2 @RequestMapping("/user")
3 public class UserController {
4     private final KeycloakAdminClientService kcAdminClient;
5
6     private final KeycloakProvider kcProvider;
7
```

```
8     private static final Logger LOG = org.slf4j.LoggerFactory.getLogger(
9
10
11    public UserController(KeycloakAdminClientService kcAdminClient, Keyc
12        this.kcProvider = kcProvider;
13        this.kcAdminClient = kcAdminClient;
14    }
15
16
17    @PostMapping(value = "/create")
18    public ResponseEntity<Response> createUser(@RequestBody CreateUserRe
19        Response createdResponse = kcAdminClient.createKeycloakUser(user
20        return ResponseEntity.ok(createdResponse);
21
22    }
```

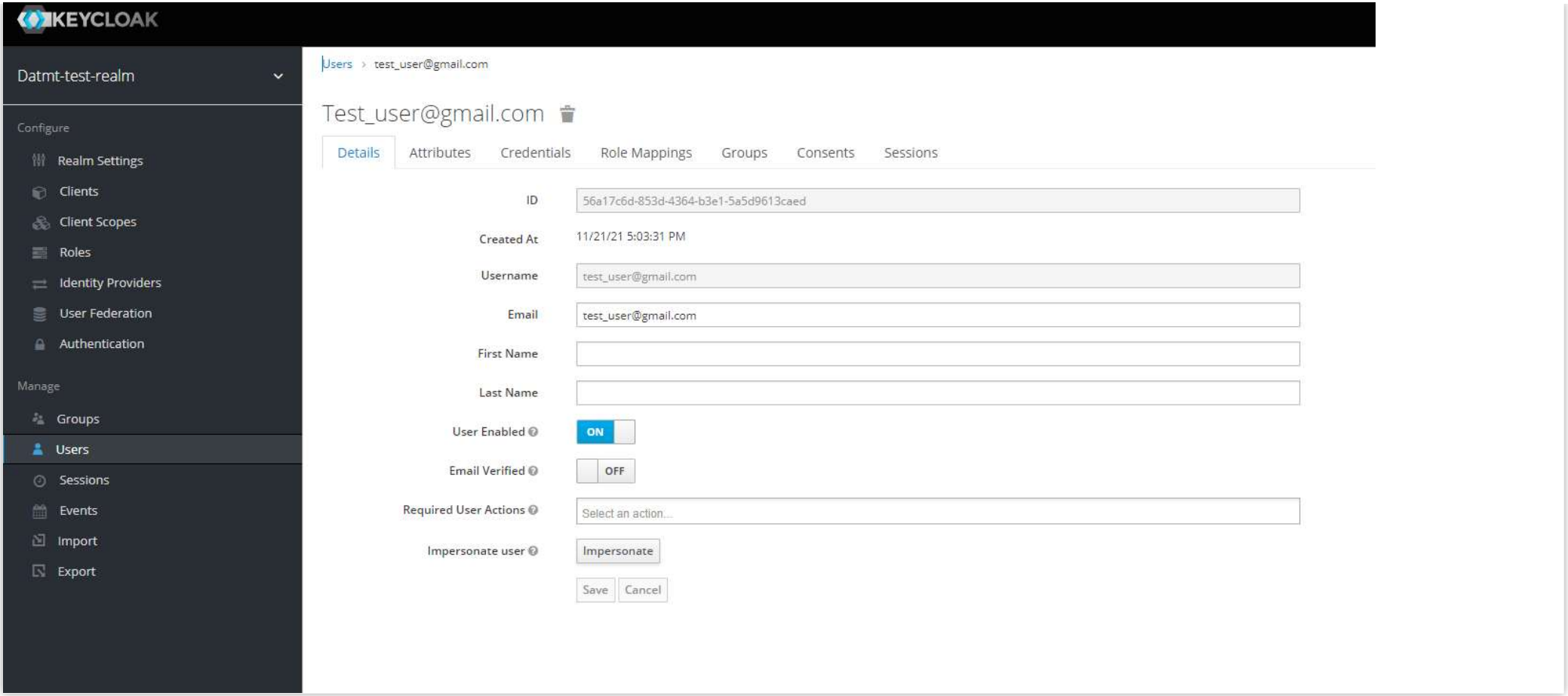
```
23
```

Too much code? No worries, they are all available on Github.

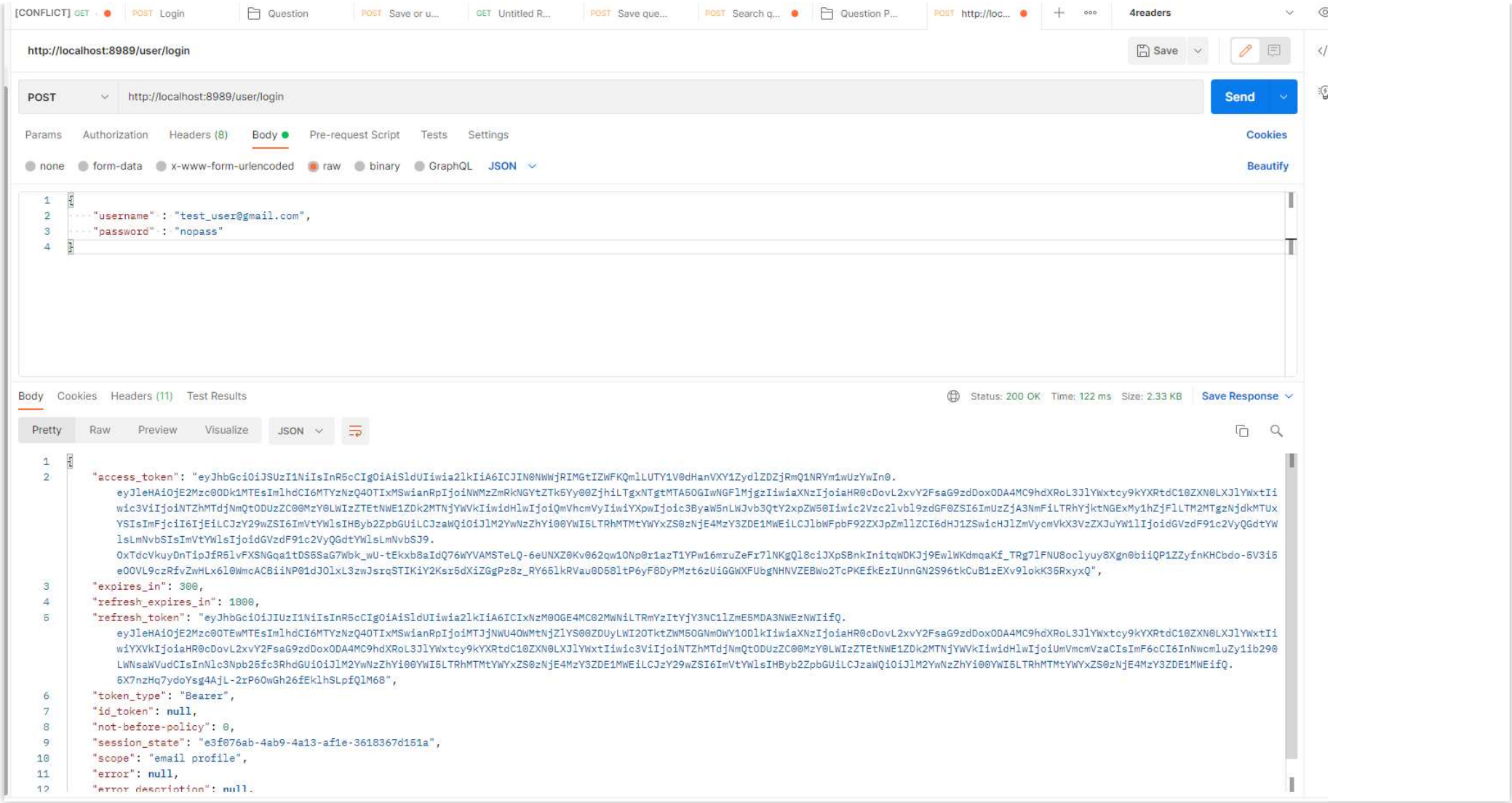
Now, open postman and call these endpoints. First, create a user:



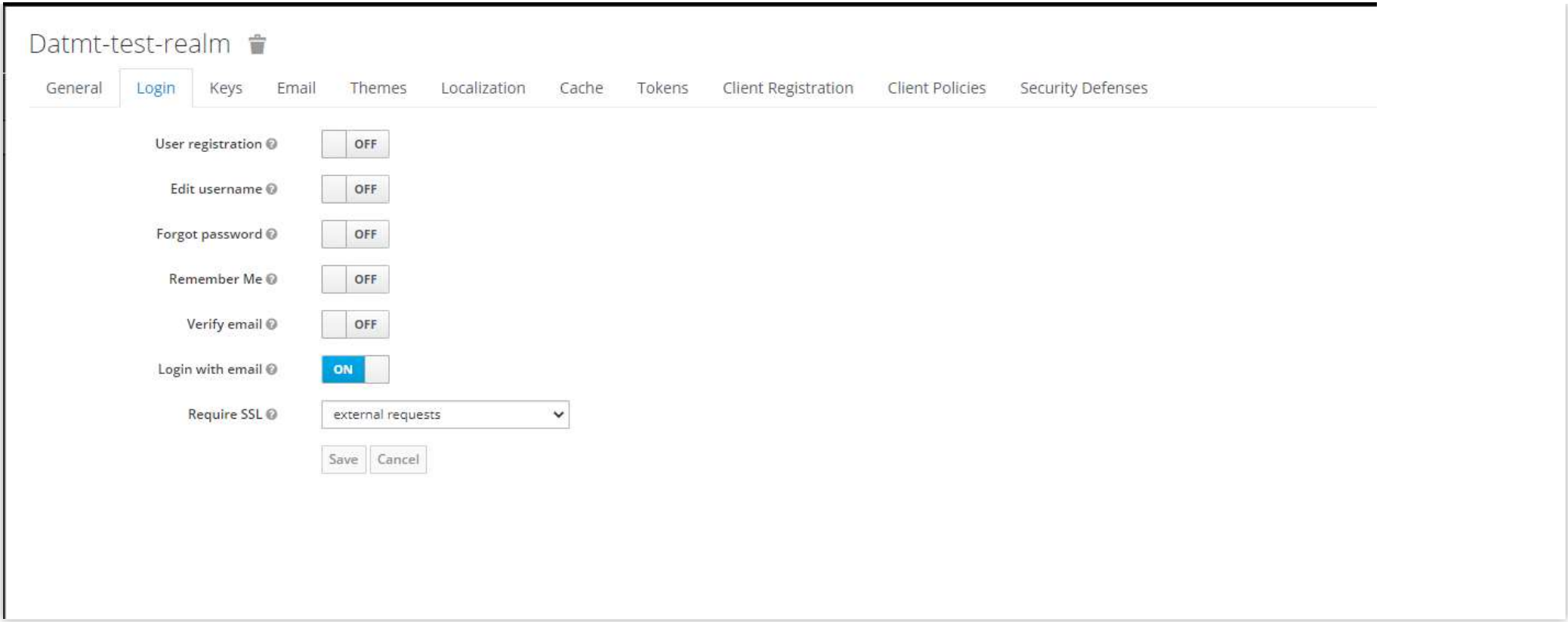
If you login with that user now, you will get an error since the user hasn’t got email verified:



Simply switch email verified on then y ou can get the tokens:



Notice that you need to put email in the place of user name because, by default, email is used as username:



Obviously, you can turn this off to log in with the username.

Conclusion

In this super long post, I created a project that help you to quickly configure keycloak with your spring boot app. You can now use this project as a base and start creating your app's business logic.

Code repo is here: https://github.com/datmt/Keycloak-Spring-Boot-Login-Create-User