Javarevisited · Following

Member-only story

Keycloak Integration with Spring Security 6

Aziz Kale · Following
Published in Javarevisited
10 min read · Jan 25, 2024

Listen Share More

As the security needs of modern web applications continue to grow, reliable identity authentication and authorization systems become imperative. In this context, ***Keycloak's*** *open-source, flexible, and trusted identity management solutions provide a solution to the security challenges.* ***Keycloak*** *makes it easy to work with many applications, not just* ***Spring Boot*** *projects. However, with the deprecation of* `WebSecurityConfigurerAdapter`

> *and the migration of **Spring Security** to 6.x, things got a bit confusing. But no worries, in this article, I will cover how to integrate **Keycloak** with **Spring Security,** using newer versions of both.*

I will be using

- Java 17
- Spring Boot 3.2.2
- Keyclaok 23.0.4

and as the IDE, **IntelliJ IDEA.**

Keycloak Configurations

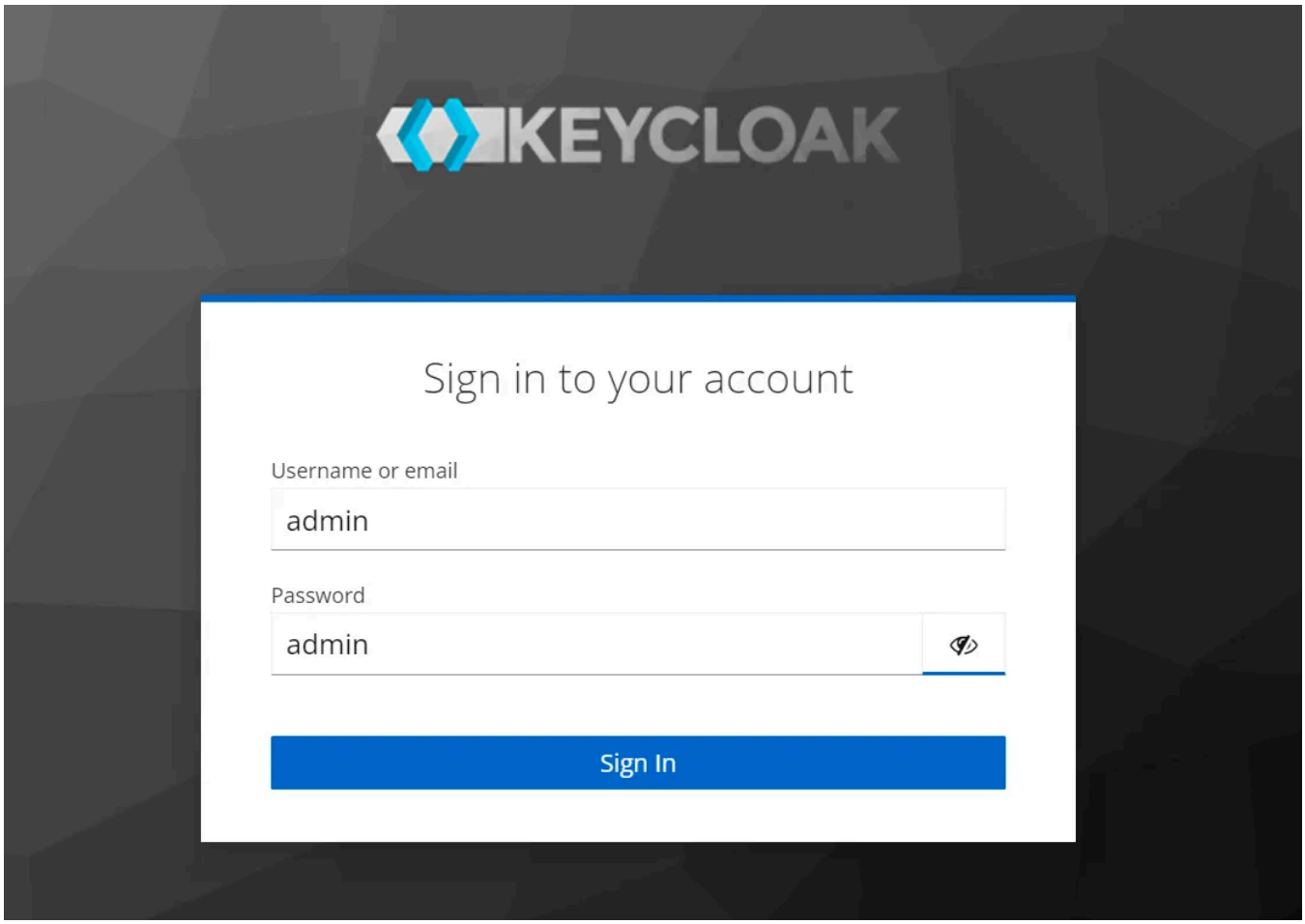
Note: If you already have an idea about Keycloak, you can skip this step and go to the Spring Boot Application Configurations setting by this link.

Keycloak is an open-source platform (server) designed to centrally manage user identities and access to applications. To use Keycloak, you can download it through this **link**. But if you have **Docker** on your PC, I recommend using the *docker command* to start Keycloak instead of setting up a Keycloak server on the computer.

```
docker run -p 8080:8080 -e KEYCLOAK_ADMIN=admin -e KEYCLOAK_ADMIN_PASSWORD=admi
```

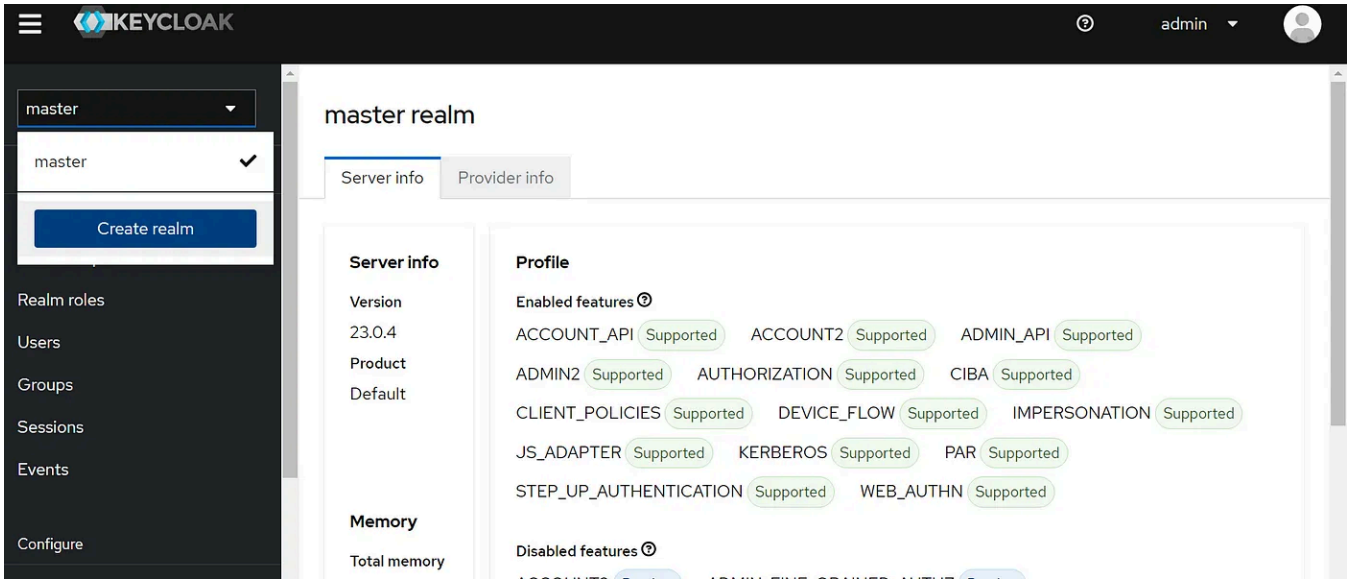
From the terminal, once you enter this command, it runs a Keycloak container with the specified parameters, making Keycloak accessible at *http://localhost:8080*. The admin console can be accessed through *http://localhost:8080/admin.*

Your initial **username** and **password** will both be "***admin***" as specified in the command.



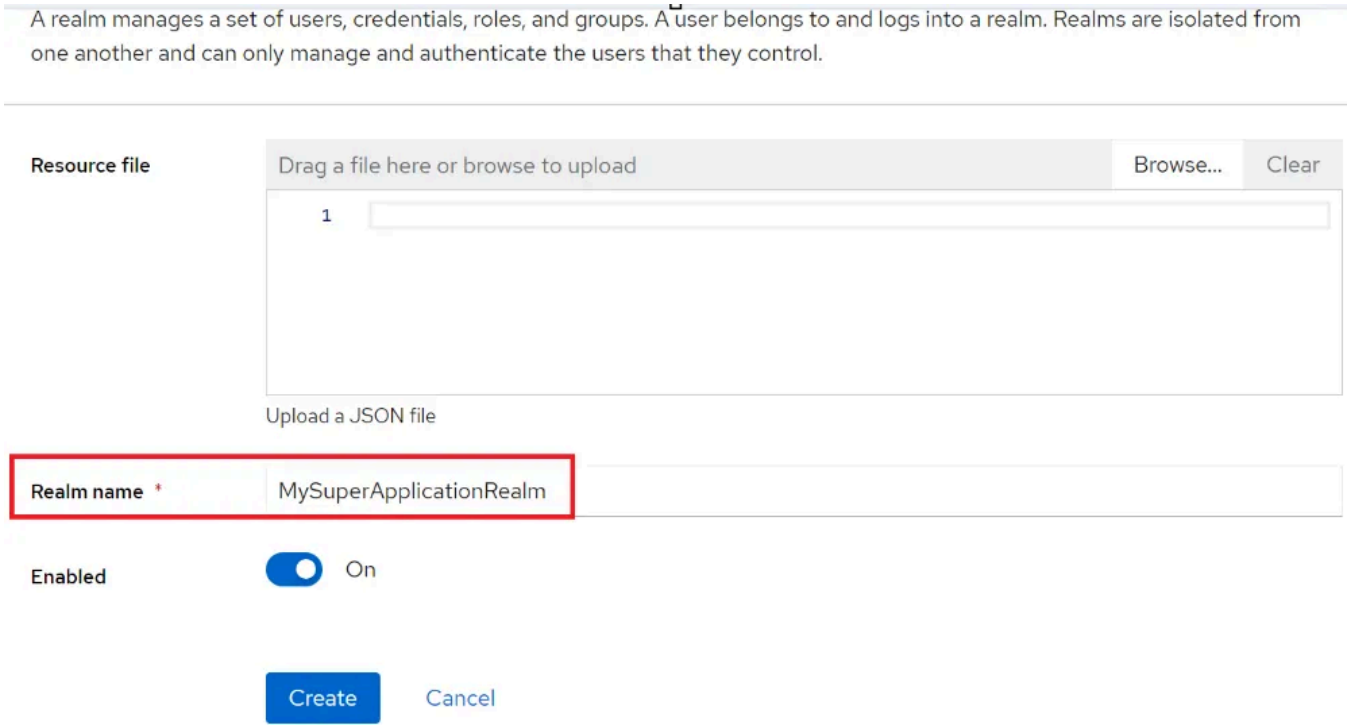
1. Creating Realm

After login, you should create a realm:



Realms are the structures that allow an administrator to create isolated groups of applications and users. Keycloak describes the realms as *tenants* in Keycloak.

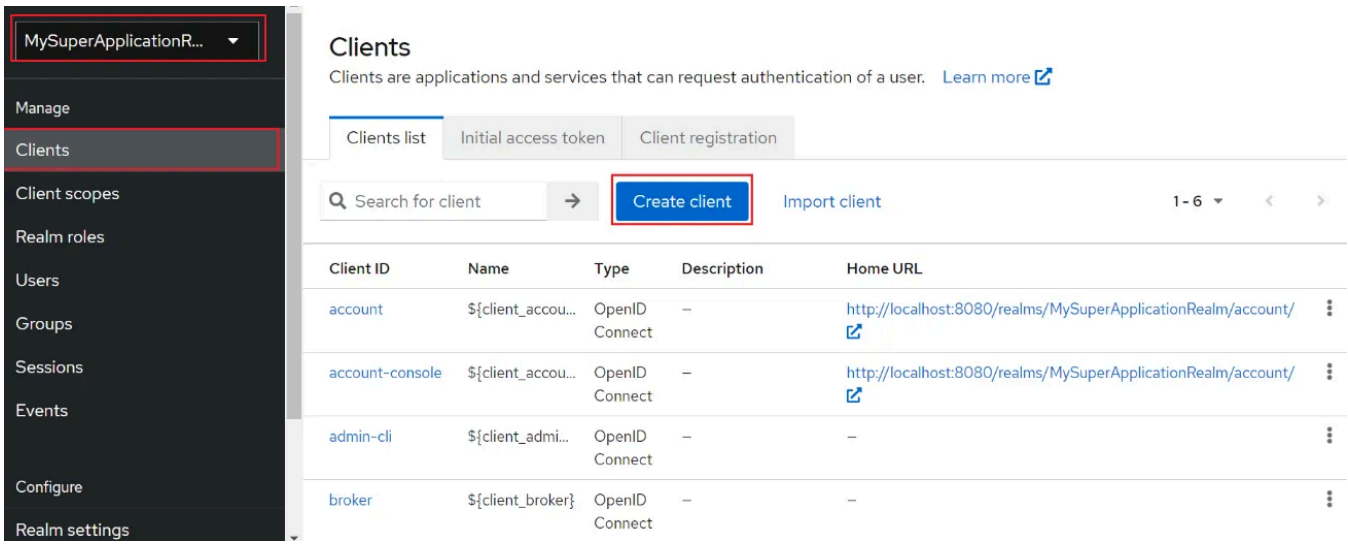
The master realm is just to manage Keycloak. Therefore I create my realm:



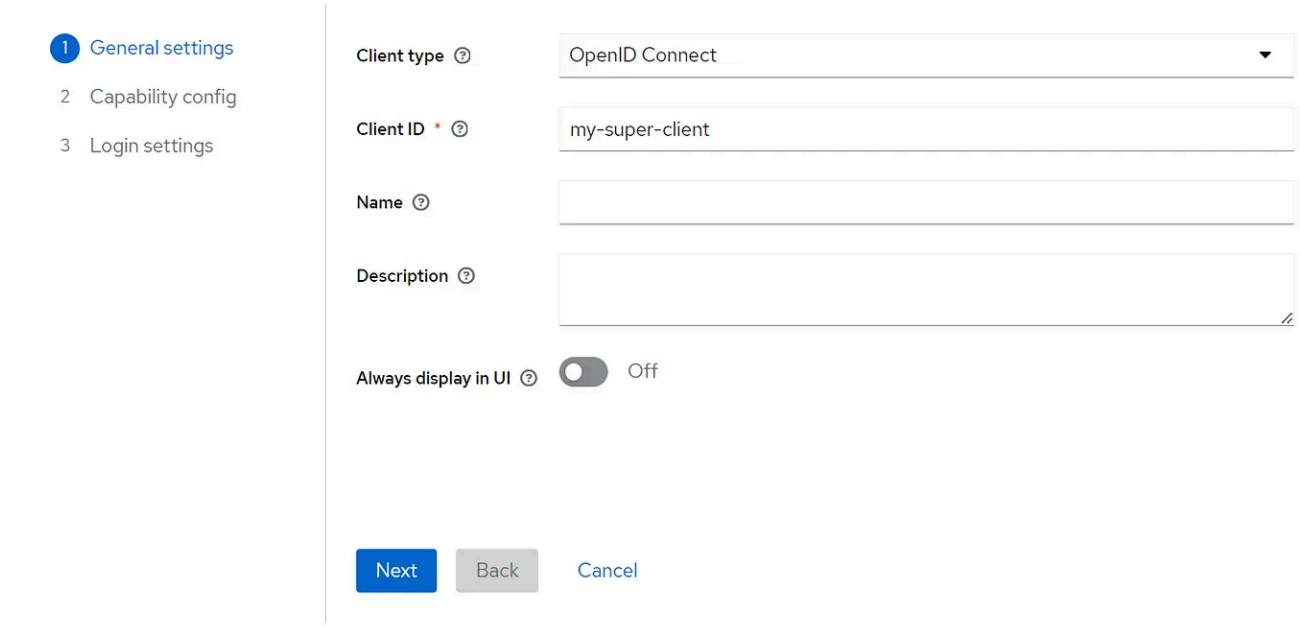
2. Creating Client

And now we need a client to protect our application. According to the official documentation of Keycloak, clients are applications and services that can request authentication of a user.

To create a new client in `MySuperApplicationRealm`, I click the "*Clients*" button on the left menu and then click on the "*Create client*" button:



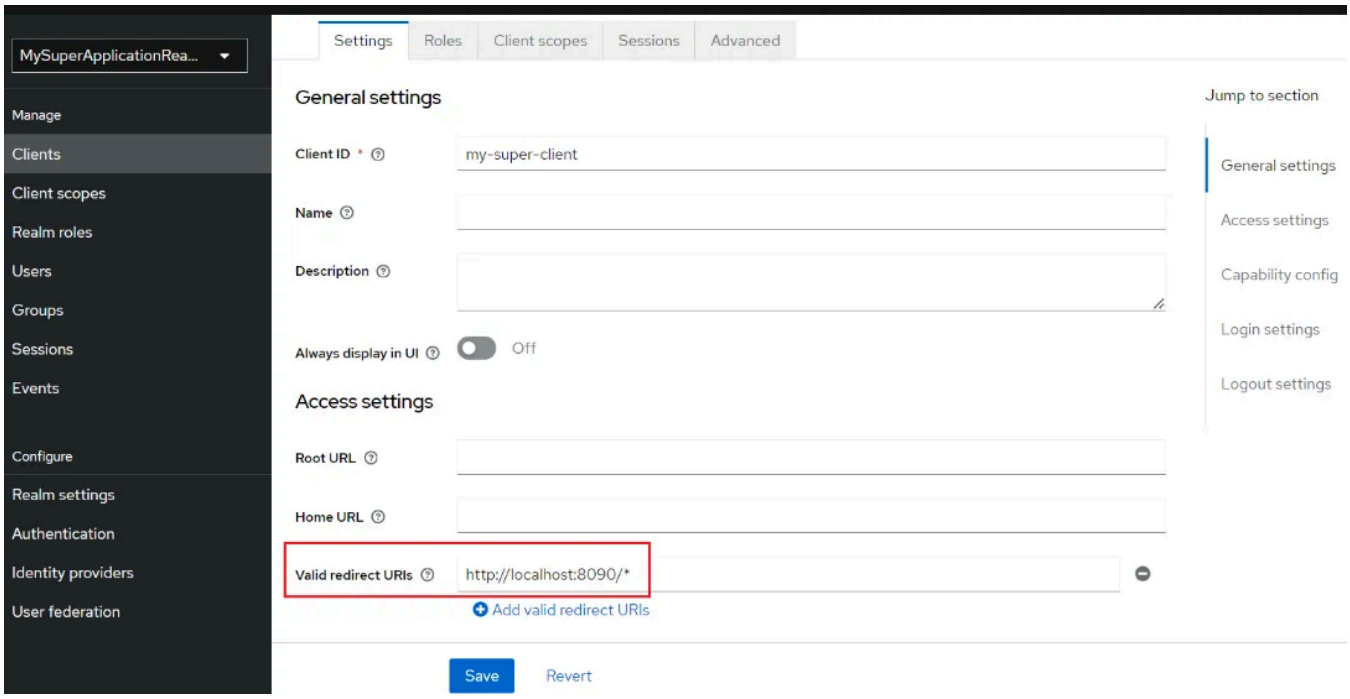
And then on the next page, I give just the "*Client ID*":



In the next screens, keep the default configurations and save. Now the client was successfully created.

Then we need to provide a **Valid Redirect URIs.**

The Keycloak server runs on port **8080.** So I want to run my Spring Boot application on port **8090.** So I give `http://localhost:8090/*` as redirect URIs and click on the '*save*' button.



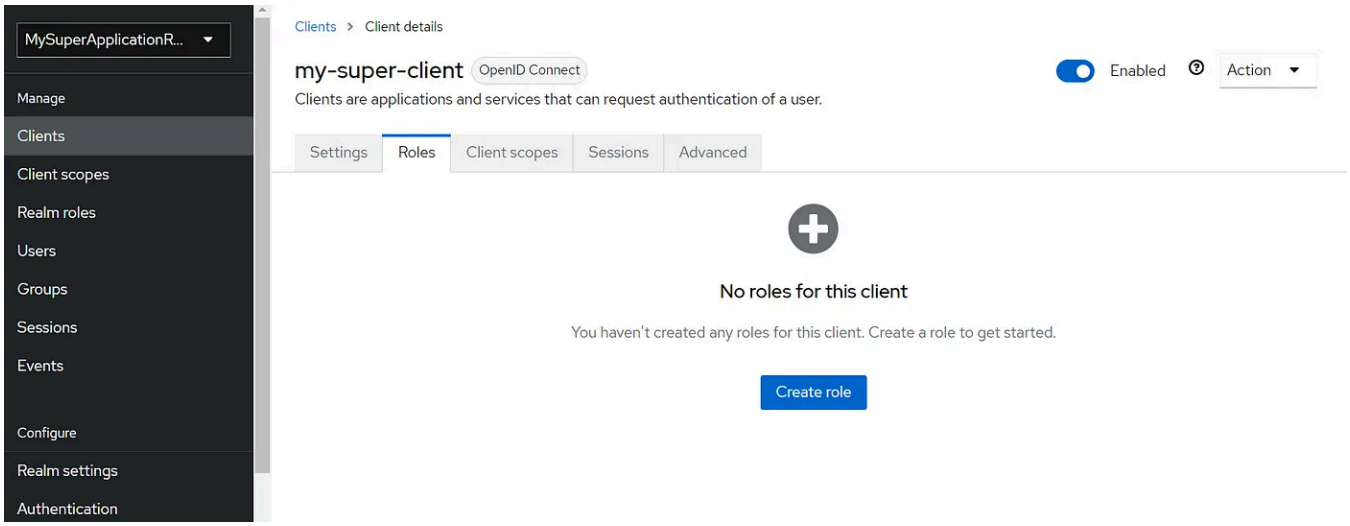
3. Creating Roles

Keycloak provides a convenient way to create roles, assign them to users, and manage application roles effectively. In Keycloak mainly there are two types of roles.

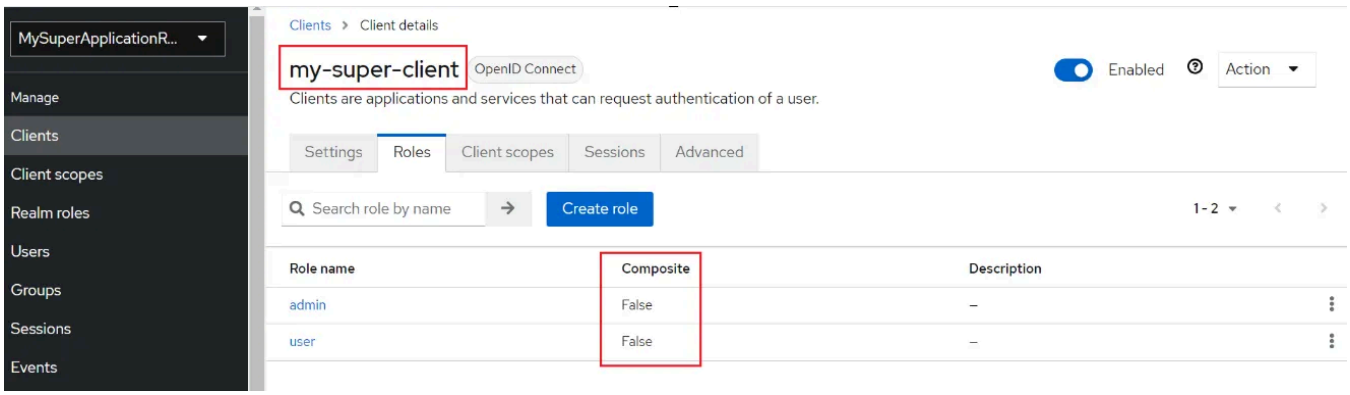
1. **Client Roles:** These roles are specific to a particular client application. Clients are created in the Keycloak administration interface and can be assigned to users within a specific client.
2. **Realm Roles:** These roles represent a realm, which can be a real or virtual application domain. Realm roles are applicable to all clients within a realm and can be assigned to users across the entire realm.

And in addition to them, there are **composite roles**. A composite role is not a role type but a special role that includes multiple authorities.

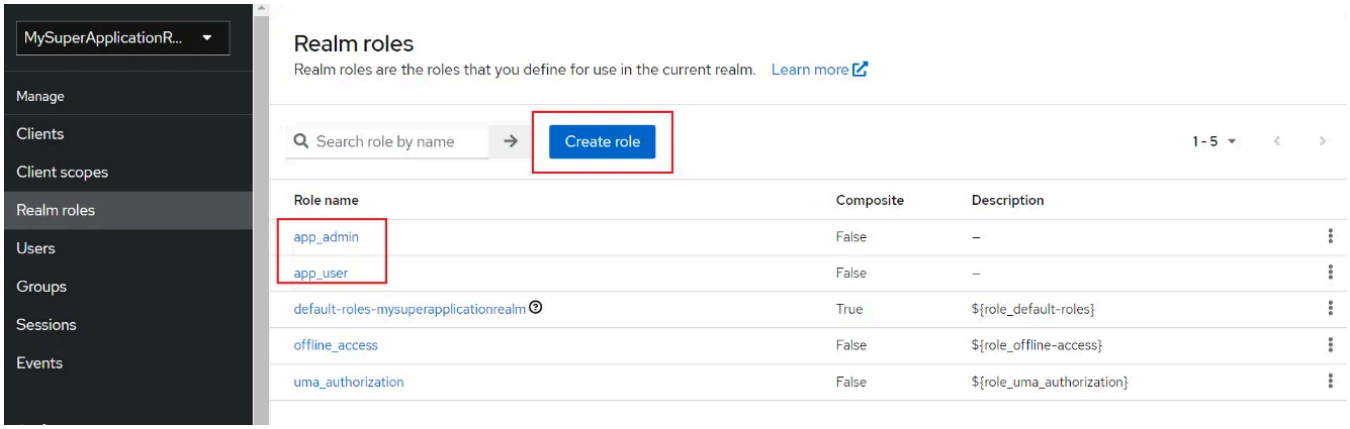
First I create two client roles called `user` and `admin`. To do this, I click on the client that I created and then click on *"Create role"* :



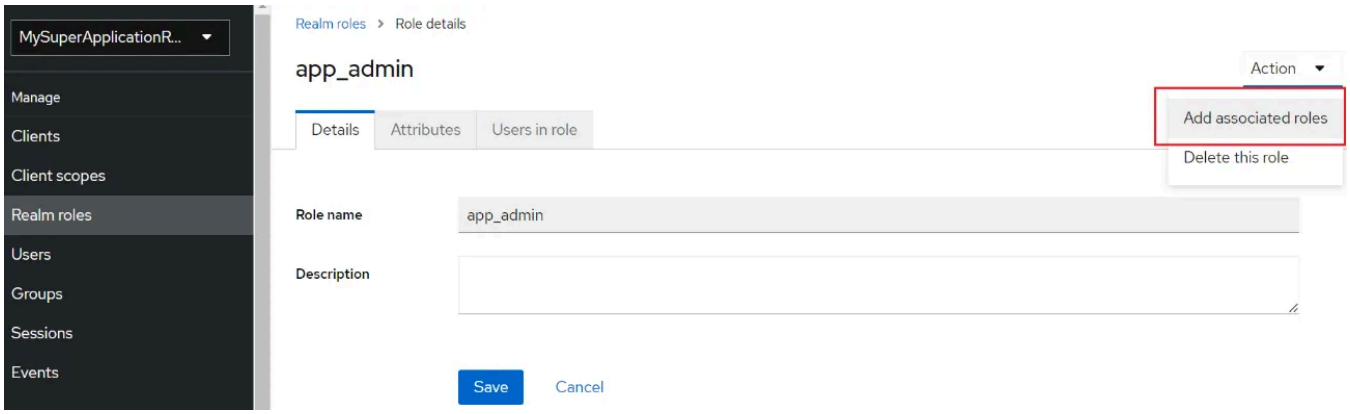
It should now look like this:



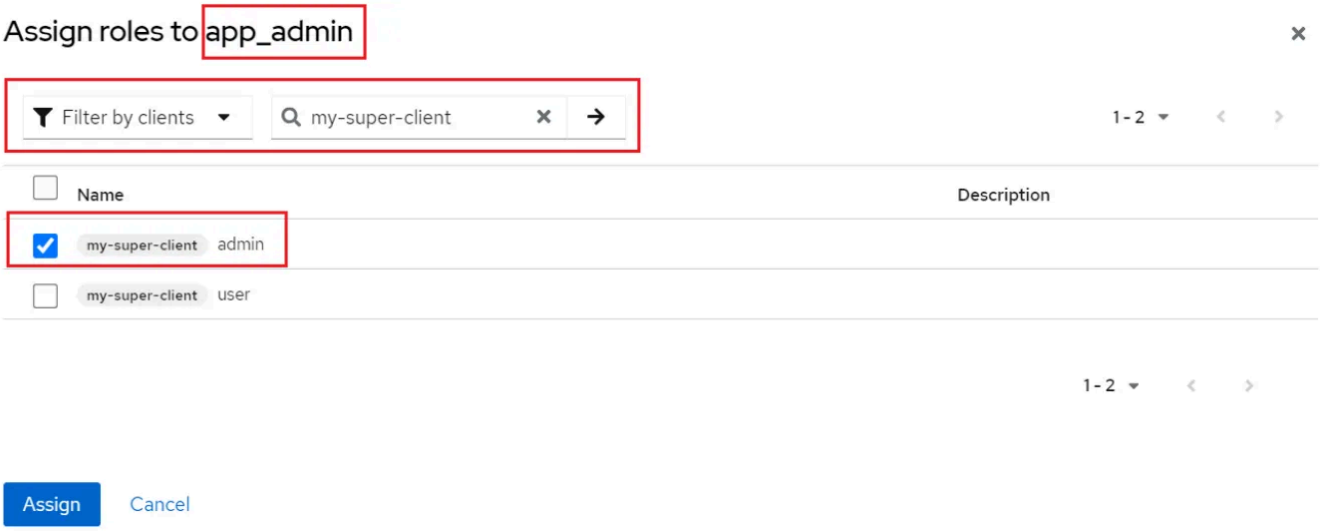
As you can see, they are not composite roles yet, and they don't need to be composite by default — only when necessary. I will make them composite just for the sake of this example. So I create two realm roles. To do this, navigate to the *Realm Roles* page to create roles.



I created `app_admin` and `app_user` by clicking on the button in the rectangle. And then click on one and on the prompted page click on the right-top dropdown menu and select the *"Add associated roles"* option:

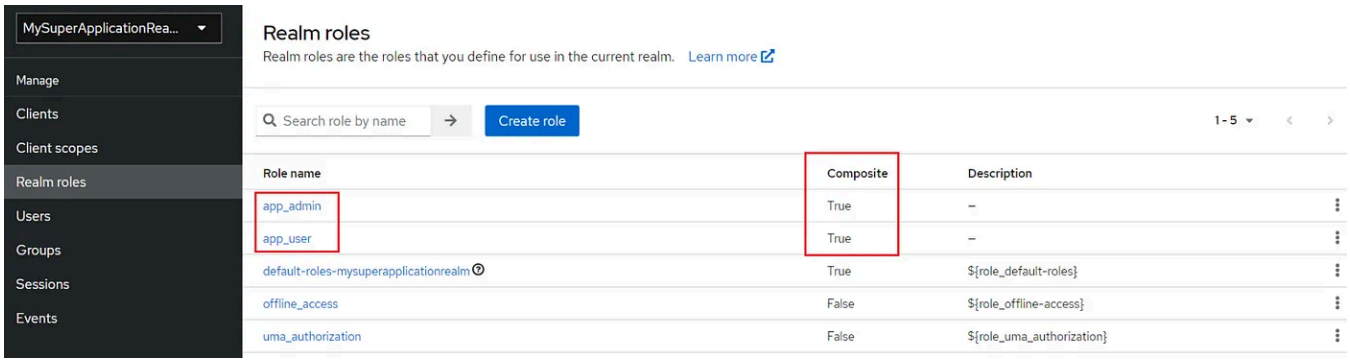


and on the page that came up, change the filter to "*Filter by clients*". You will see your client roles:



I assign the client role `admin` to the realm role `app_admin`. And follow the same steps for the `app_user` role.

If you added those correctly, Realm roles will be composite.

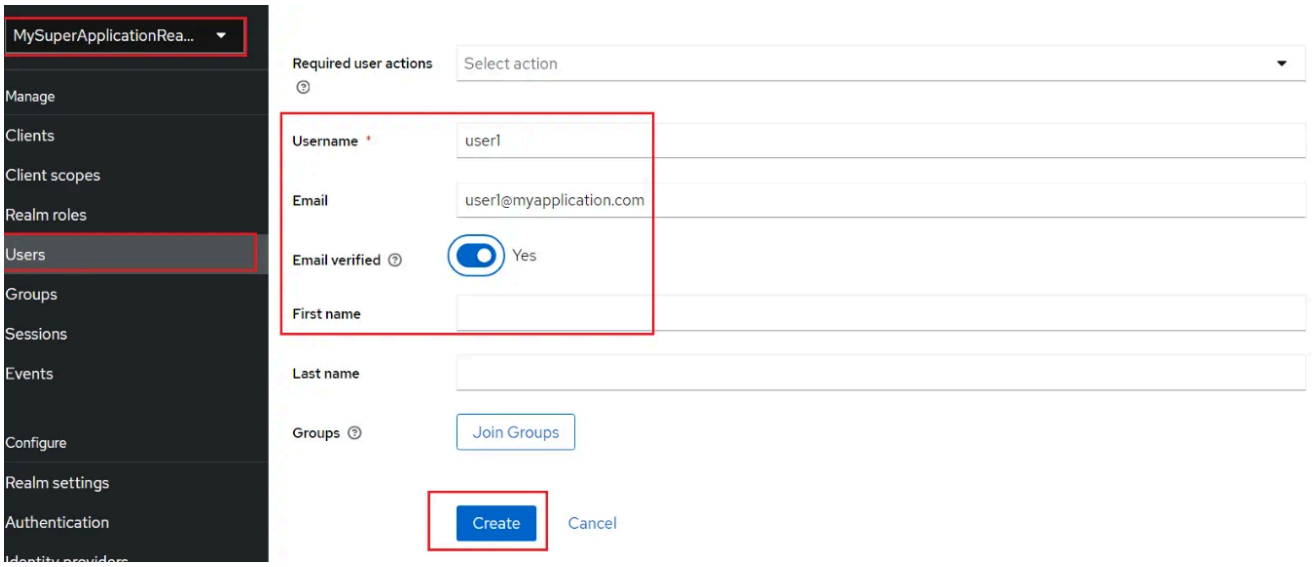


4. Creating Users

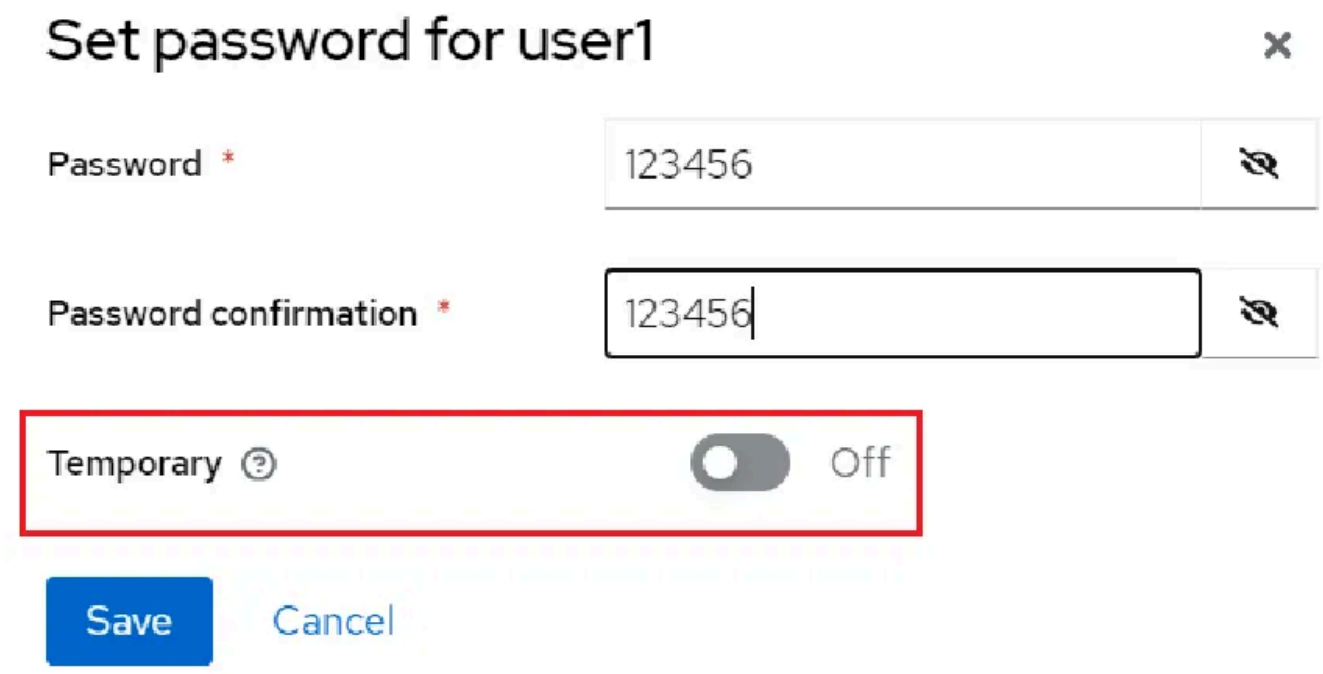
The last part that needs to be added is the users. I go to the *Users* page and create three users with these roles:

- user1 with `app_admin` role
- user2 with `app_user` role
- user3 with `app_user` and `app_admin` roles

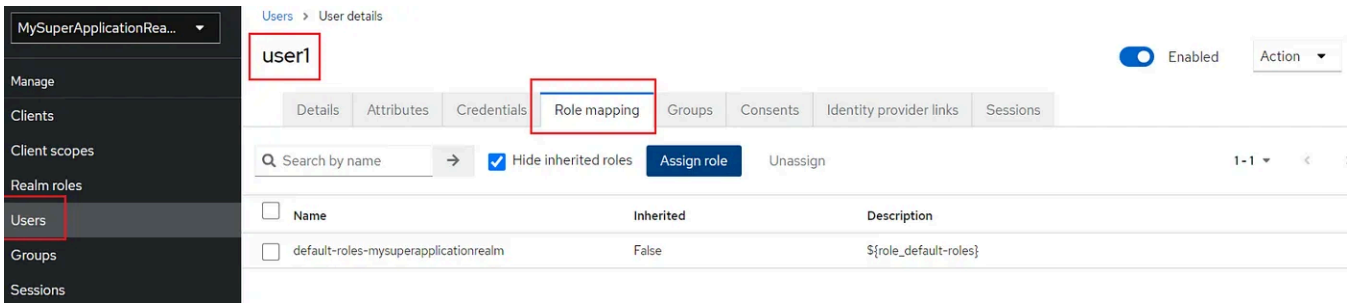
I go to the *"Users"* page and click on *"Add user"*:



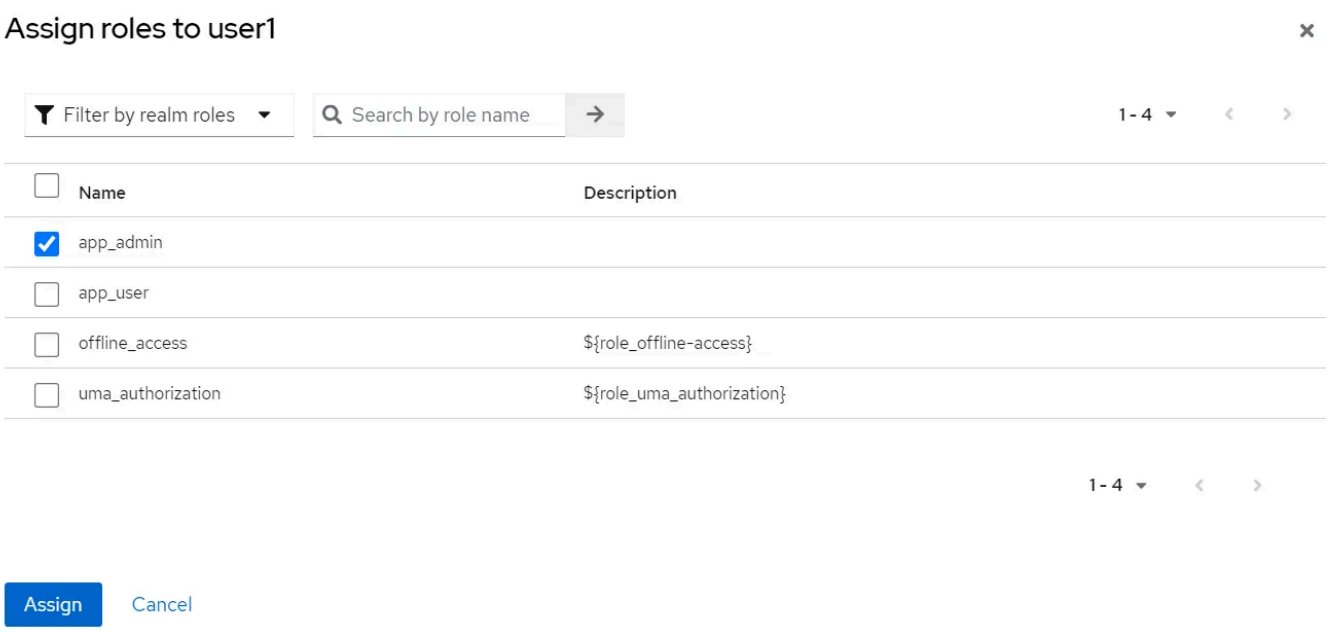
Now I need to set passwords for users. I go to the *"Users"* page and click on `user1` and select the *"Credentials"* tab on the page that came and then click on "*Set password*" button:



After saving credentials I click on the *"Role mapping"* tab:

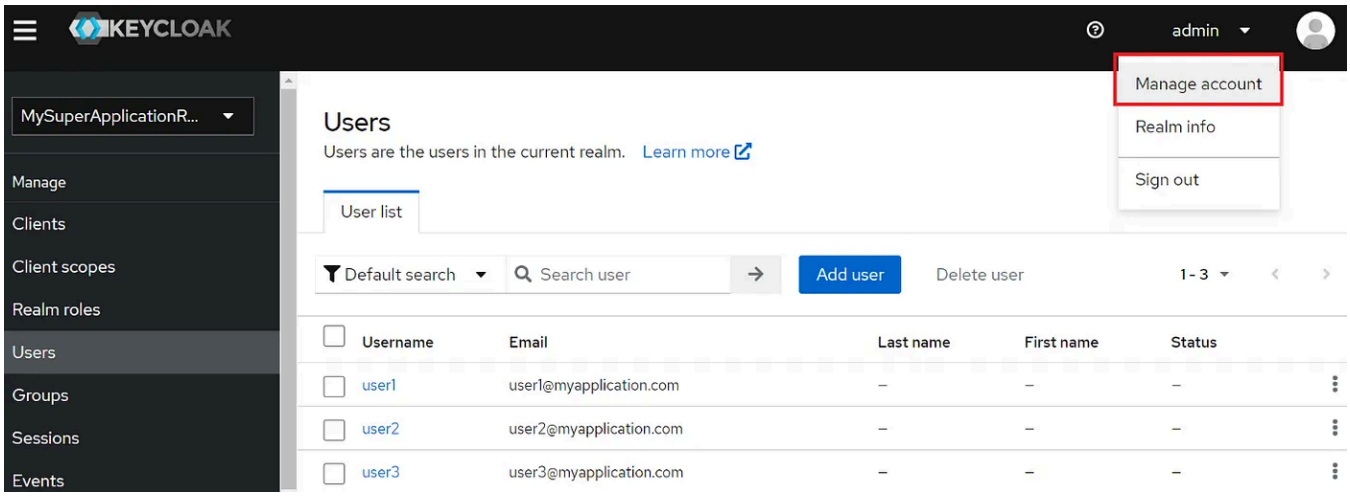


When the *'Assign Role'* button is clicked, the list of realm roles will be available:



I set the same password to other users as well, and I assigned the roles I mentioned above to the relevant users.

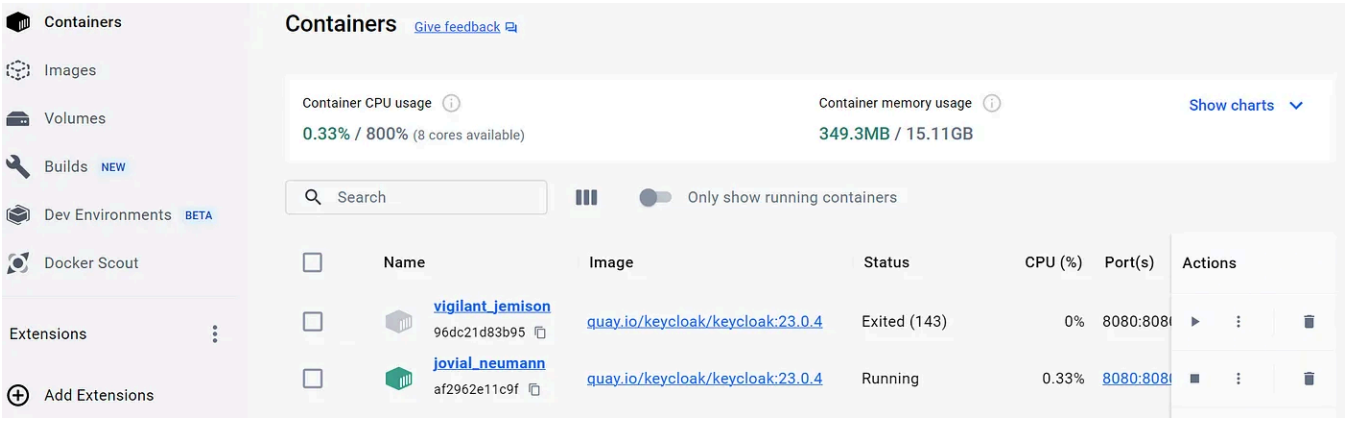
Up to this point, authentication and authorization processes are complete. Remember to customize your Keycloak account credentials for the safety of the application. You can use the "*Manage account*" menu to do this:



By the

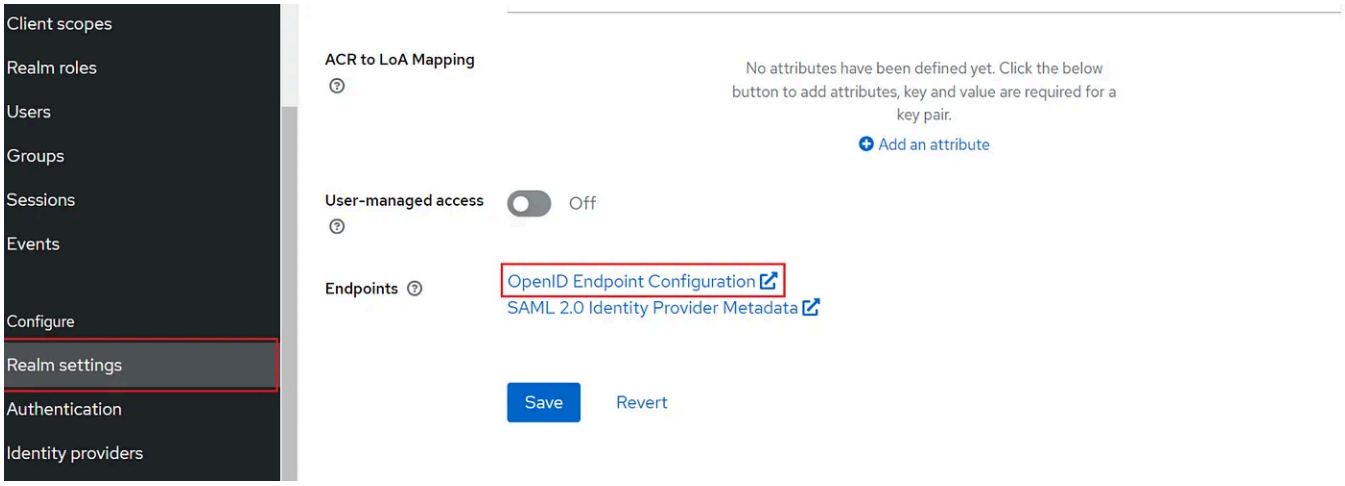
```
-v ${HOME}/docker_volumes/keycloak:/opt/keycloak/data
```

flag in the docker command we used at the beginning, we copied our credentials data from the docker container to our PC. And remember please, If you have already this kind of file in the path on the PC, Keycloak reads these credentials to run. Or you can just run specific containers for specific projects and their credentials by using **Docker Desktop**.



Now let's move on to obtaining the token.

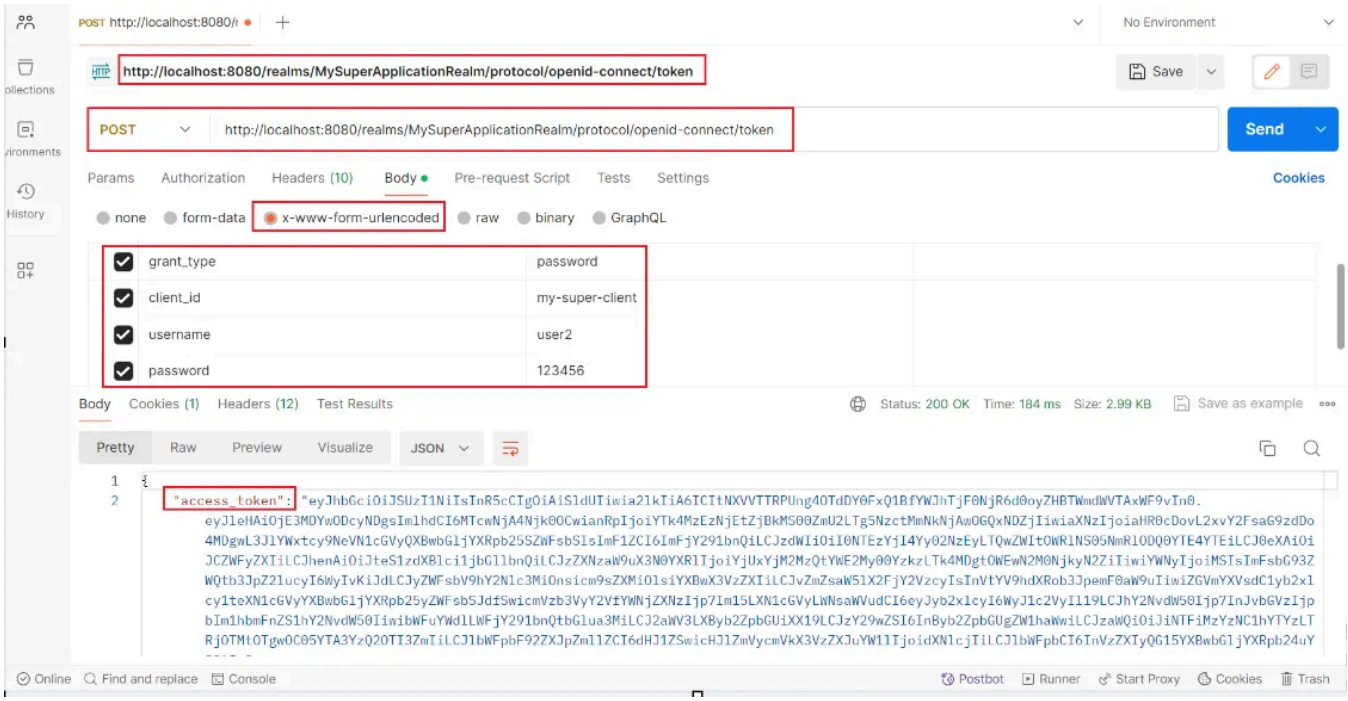
When you go to the "*Realm settings*" section on the sidebar you will see the *"OpenID Endpoint Configuration"* link:



When you click on this link, you will see a list of available endpoints:

```
{
  "issuer": "http://localhost:8080/realms/MySuperApplicationRealm",
  "authorization_endpoint": "http://localhost:8080/realms/MySuperApplicationRealm/protocol/openid-connect/auth",
  "token_endpoint": "http://localhost:8080/realms/MySuperApplicationRealm/protocol/openid-connect/token",
  "introspection_endpoint": "http://localhost:8080/realms/MySuperApplicationRealm/protocol/openid-connect/token/introspect",
  "userinfo_endpoint": "http://localhost:8080/realms/MySuperApplicationRealm/protocol/openid-connect/userinfo",
  "end_session_endpoint": "http://localhost:8080/realms/MySuperApplicationRealm/protocol/openid-connect/logout",
  "frontchannel_logout_session_supported": true,
  "frontchannel_logout_supported": true,
  "jwks_uri": "http://localhost:8080/realms/MySuperApplicationRealm/protocol/openid-connect/certs",
  "check_session_iframe": "http://localhost:8080/realms/MySuperApplicationRealm/protocol/openid-connect/login-status-iframe.html",
  "grant_types_supported": [
    "authorization_code",
    "implicit",
    "refresh_token",
    "password",
    "client_credentials",
    "urn:openid:params:grant-type:ciba",
    "urn:ietf:params:oauth:grant-type:device_code"
  ],
  "acr_values_supported": [
    "0",
    "1"
  ],
  "response_types_supported": [
    "code",
```


We can send now a **post** request to this endpoint to get a token. We can use one of the users we created as the credentials. I use postman for this request:



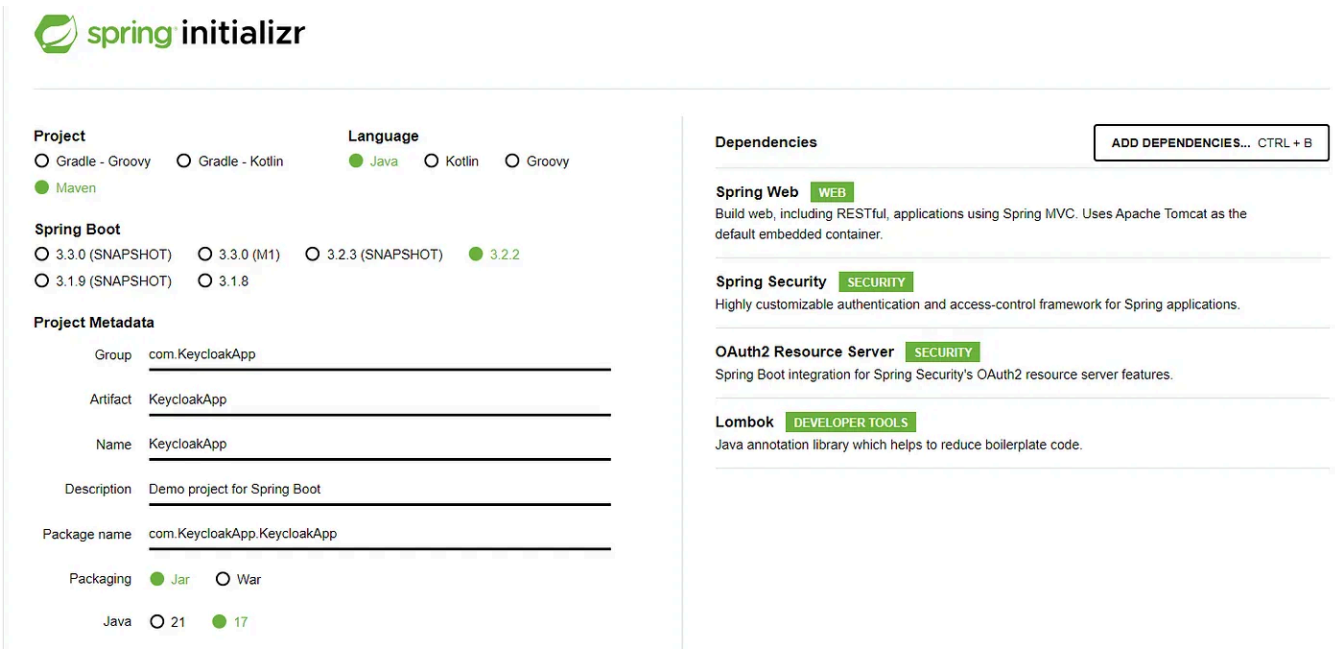
When you configure the settings indicated in the image, you will be able to obtain the access token.

Spring Boot Application Configurations

The Keycloak configuration concludes here. Now, we can proceed to integrate it with our Spring Boot application.

Creating Spring Boot Project

I'm using Spring Boot version 3, Java 17, and Maven as the package manager.



After settings, adding necessary dependencies, generating and downloading the project I open it by IntelliJ IDEA and change the port from the default value `8080` to `8081` on which the project will run, in the `application.properties` file as **Keycloak** runs on port `8080`.

```
server.port=8081
```

I am creating now a simple REST API to test the application. I am just adding the class below under the package `controller` I created.

```
@RestController
@RequestMapping("/api")
public class ControllerHello{

    @GetMapping("/hello")
    public ResponseEntity<String> sayHello() {
        return ResponseEntity.ok("Hello");
    }

    @GetMapping("/admin")
    public ResponseEntity<String> sayHelloToAdmin() {
        return ResponseEntity.ok("Hello Admin");
    }
```

```
    @GetMapping("/user")
    public ResponseEntity<String> sayHelloToUser() {
        return ResponseEntity.ok("Hello User");
    }
}
```

Keycloak Integration

In the `application.properties` file, I add the following settings:

```
spring.application.name=KeycloakSpringBootApplication

# Security Configuration
spring.security.oauth2.resourceserver.jwt.issuer-uri=http://localhost:8080/real
spring.security.oauth2.resourceserver.jwt.jwk-set-uri=${spring.security.oauth2.

# JWT Configuration
jwt.auth.converter.resource-id=my-super-client
jwt.auth.converter.principal-attribute=principal_username

# Logging Configuration
logging.level.org.springframework.security=DEBUG
```

- `spring.application.name=KeycloakSpringBootApplication`: This setting defines the name of the Spring application. It represents the overall name of the application.
- `spring.security.oauth2.resourceserver.jwt.issuer-uri`: This setting specifies the **Issuer URI** for JWT authentication. It includes the URI of the Keycloak realm.
- `spring.security.oauth2.resourceserver.jwt.jwk-set-uri`: Specifies the JWK set URI at the location where JWT will be used. This URI is used to fetch public keys.
- `jwt.auth.converter.resource-id`: Specifies the resource ID of the JWT. Typically, it includes a client identifier.
- `jwt.auth.converter.principal-attribute`: Specifies a specific attribute name in the JWT's header (such as `principal_username`) where principals (users) are indicated.

Now, I need security to protect my API. To achieve this, I will create a `SecurityConfig` class using Spring Security. However, as I aim to integrate my project with Keycloak, I initially create two classes to decode the JWT obtained from the Keycloak server.

Under the `security` package:

```
@Data
@Validated
@Configuration
@ConfigurationProperties(prefix = "jwt.auth.converter")
public class JwtConverterProperties {

    private String resourceId;
    private String principalAttribute;
}
```

The `JwtConverterProperties` class is a configuration class for **JWT** authentication conversion. It holds properties like "***resourceId***" (Keycloak resource identifier) and "***principalAttribute***" (preferred principal attribute). These properties play a crucial role in decoding and processing JWTs within the Keycloak integration.

```
@Component
public class JwtConverter implements Converter<Jwt, AbstractAuthenticationToker

    private final JwtGrantedAuthoritiesConverter jwtGrantedAuthoritiesConverter

    private final JwtConverterProperties properties;

    public JwtConverter(JwtConverterProperties properties) {
        this.properties = properties;
    }

    @Override
    public AbstractAuthenticationToken convert(Jwt jwt) {
        Collection<GrantedAuthority> authorities = Stream.concat(
                jwtGrantedAuthoritiesConverter.convert(jwt).stream(),
                extractResourceRoles(jwt).stream()).collect(Collectors.toSet())
        return new JwtAuthenticationToken(jwt, authorities, getPrincipalClaimNa
    }

    private String getPrincipalClaimName(Jwt jwt) {
        String claimName = JwtClaimNames.SUB;
```

```
            if (properties.getPrincipalAttribute() != null) {
                claimName = properties.getPrincipalAttribute();
            }
            return jwt.getClaim(claimName);
        }

        private Collection<? extends GrantedAuthority> extractResourceRoles(Jwt jwt
            Map<String, Object> resourceAccess = jwt.getClaim("resource_access");
            Map<String, Object> resource;
            Collection<String> resourceRoles;

            if (resourceAccess == null
                    || (resource = (Map<String, Object>) resourceAccess.get(propert
                    || (resourceRoles = (Collection<String>) resource.get("roles"))
                return Set.of();
            }
            return resourceRoles.stream()
                    .map(role -> new SimpleGrantedAuthority("ROLE_" + role))
                    .collect(Collectors.toSet());
        }
    }
```

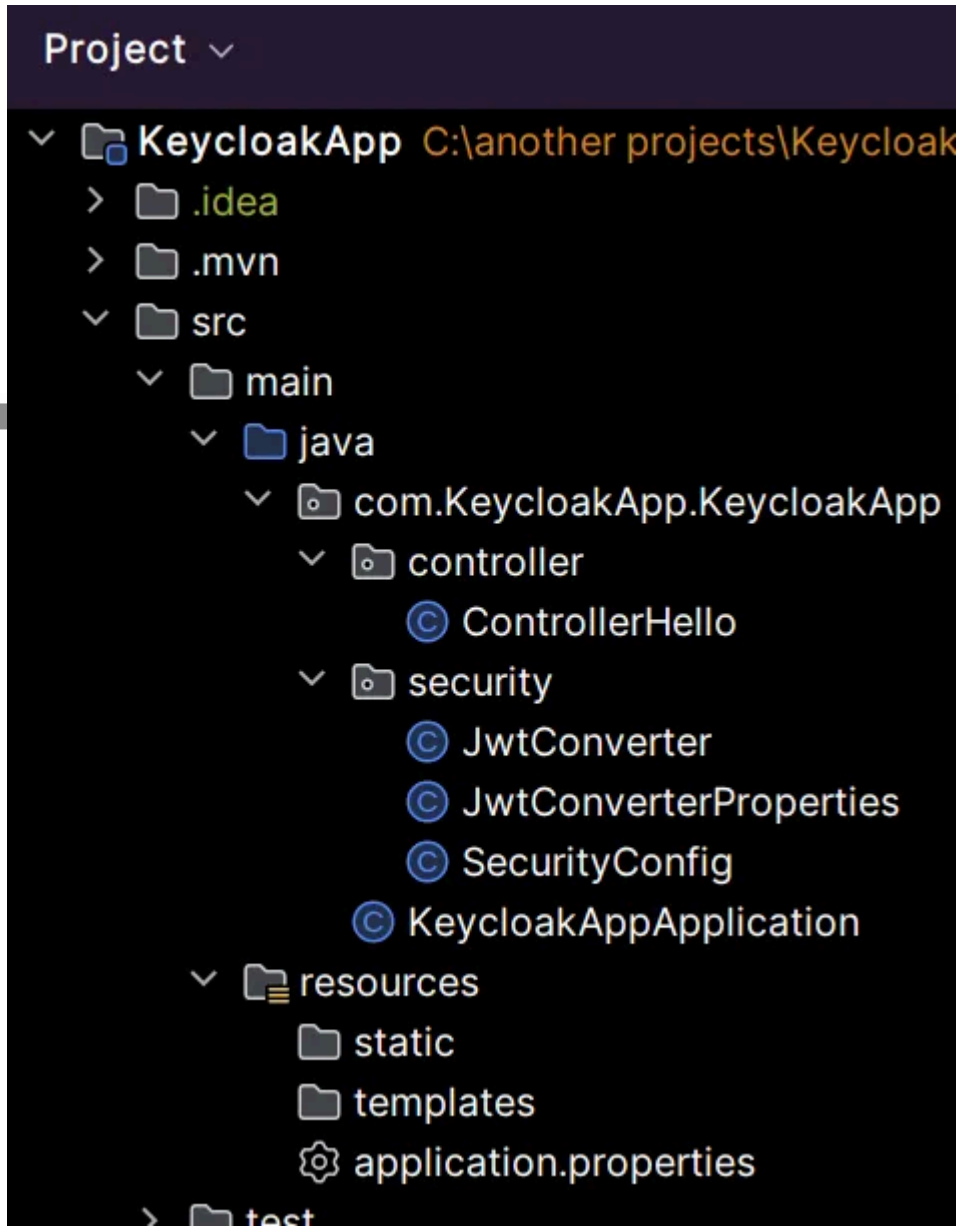
`JwtConverter` decodes **JWTs** from **Keycloak**, enabling role-based endpoint access and user information extraction in **Spring Security**. It implements the **Converter** interface, extracting roles and details from JWTs, including resource-specific roles. Configurable via **JwtConverterProperties**, it plays a dual role: access control and JWT deciphering.

And the `SecurityConfig` class(*for more info about new spring security you can follow this* ***link***):

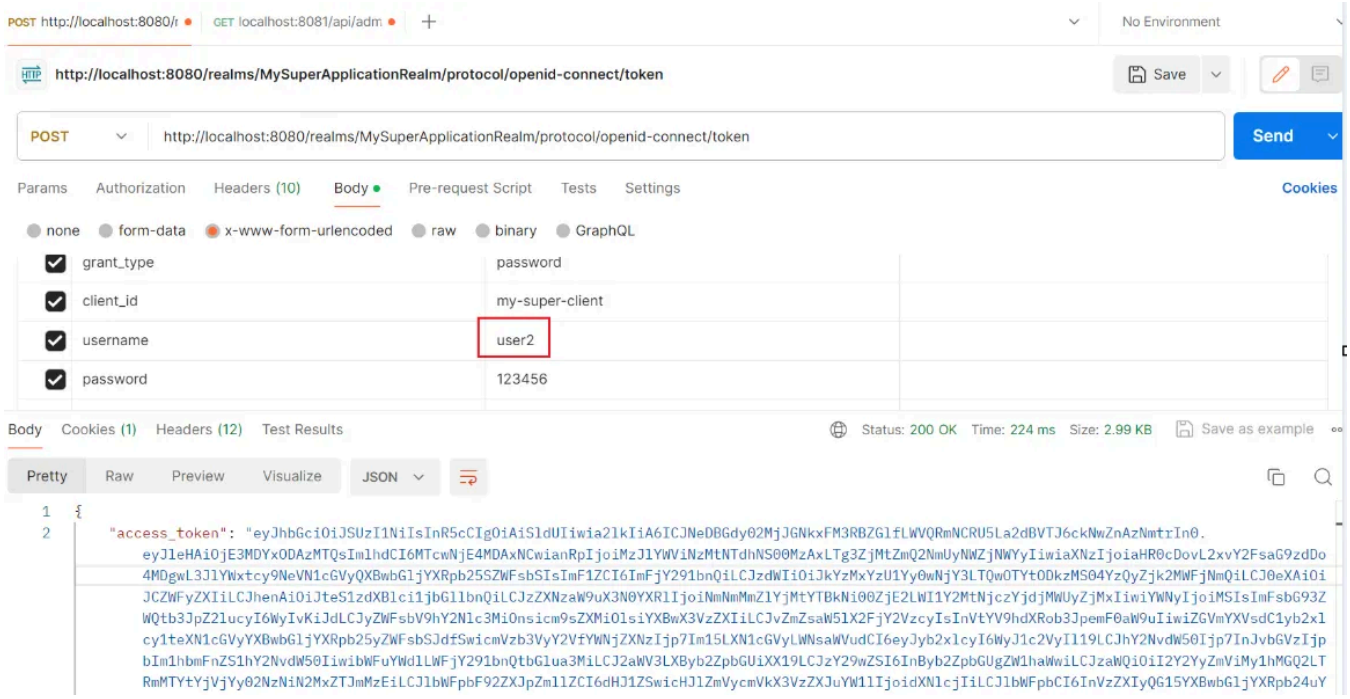
```
@RequiredArgsConstructor
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    public static final String ADMIN = "admin";
    public static final String USER = "user";
    private final JwtConverter jwtConverter;
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Ex
        http.authorizeHttpRequests((authz) ->
                authz.requestMatchers(HttpMethod.GET, "/api/hello").permitAll()
                .requestMatchers(HttpMethod.GET, "/api/admin/**").hasRole(ADMIN
                .requestMatchers(HttpMethod.GET, "/api/user/**").hasRole(USER)
```

```
                .requestMatchers(HttpMethod.GET, "/api/admin-and-user/**").hasA
                .anyRequest().authenticated());

        http.sessionManagement(sess -> sess.sessionCreationPolicy(
                SessionCreationPolicy.STATELESS));
        http.oauth2ResourceServer(oauth2 -> oauth2.jwt(jwt -> jwt.jwtAuthentica

        return http.build();
    }
}
```

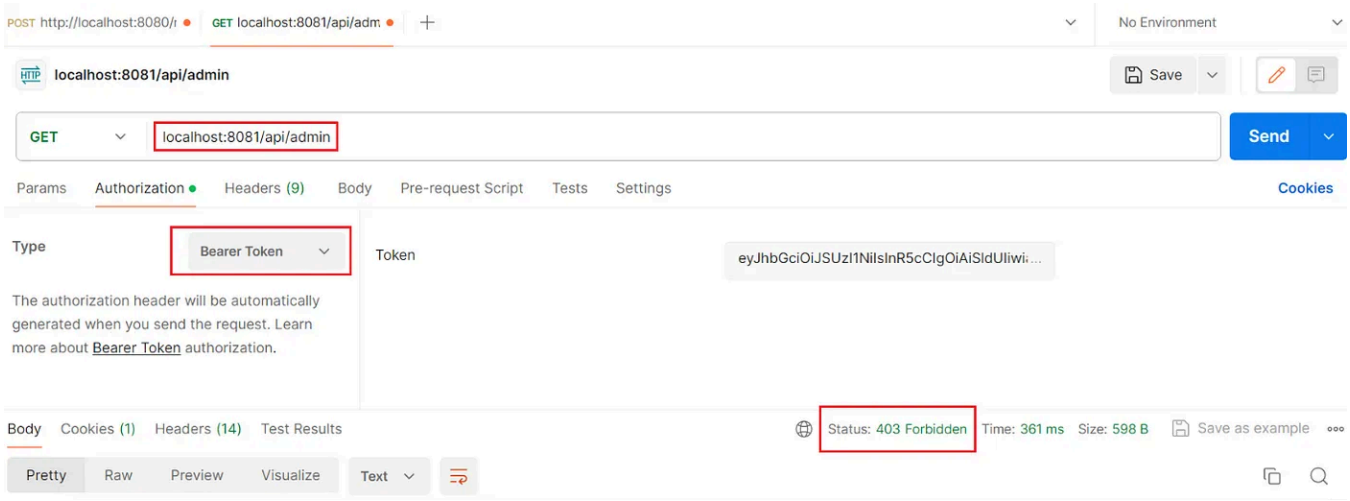
Now the project structure looks like the below:



After Spring application configurations, we can now run our project and test it. I use **postman** to do this:

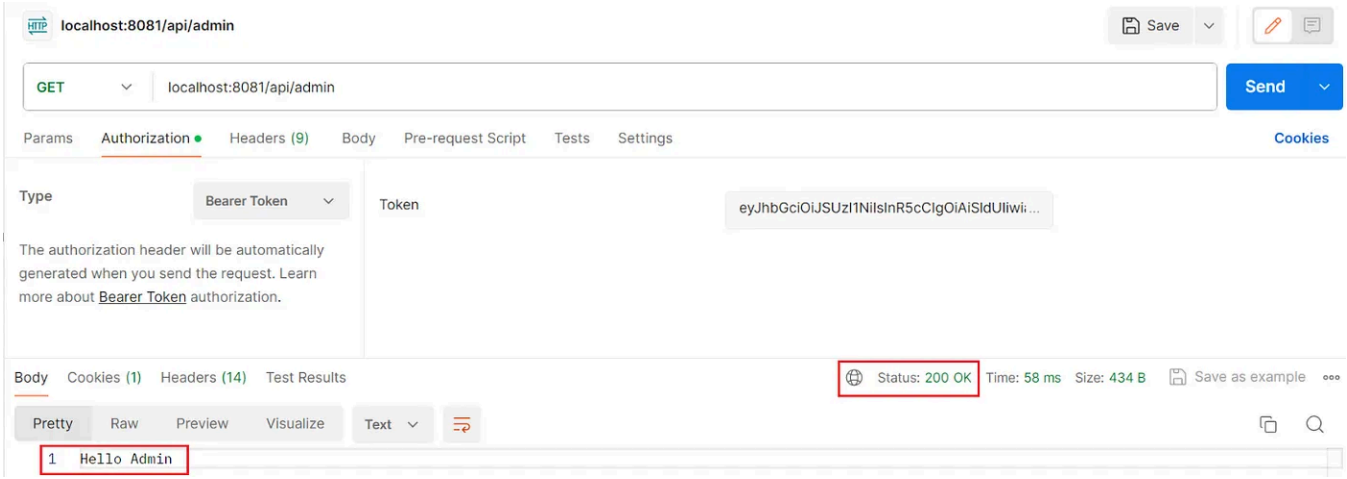


I picked up `user2` as the credentials. I get the token I've achieved and go to the get request:



I set up the **authorization type** to `Bareer Token` , pasted the token into the token box, and sent a **GET** request to the `admin` endpoint. And it got `403 Forbidden` . Because `user2` has just the `user` role not `admin` .

Now I get a token by the credentials of `user1` as he has the `admin` role and send a request to the same endpoint. The result:



That's all!

Conclusion

We set up a **Keycloak** server using the **Docker** command. We used the `-v` flag in this command to store the data, such as **realms**, **clients**, and **users**, **credentials** on the local machine. We covered the concepts of **realms** and **clients** in Keycloak, as well as the types of roles available. Users were created and assigned roles. Subsequently, we developed a Spring Boot **RESTful API** application and integrated it with our Keycloak server. We secured endpoints using tokens obtained from Keycloak.

You can access the Spring Boot project at this **link**.

Thank you for your time!

Spring Security | Spring Boot | Keycloak | Spring Security 6 | Spring Boot 3

Following

Published in Javarevisited

37K Followers · Last published 3 hours ago

A humble place to learn Java and Programming better.

Following

Written by Aziz Kale

360 Followers · 14 Following

Highly Motivated, Passionate Full Stack Developer | EMM-IT Co. | Web: azizkale.com

Responses (5)

Prakash

What are your thoughts?

Herman Essoungou
Jul 6, 2024

Merci beaucoup

51 Reply

Tabrez Shaikh
May 11, 2024 (edited)

This is well detailed. Thanks!

2 Reply

G

Ganesh Nawle
Feb 13, 2024

WebSecurityConfigurerAdapter

Deprecated Class

2 Reply

See all responses

More from Aziz Kale and Javarevisited

In Javarevisited by Aziz Kale

RESTful API Testing in Java with Mockito (Controller Layer)

In this article, I will continue covering Mockito in Java projects. In the previous article, we discussed how to create a unit test for the...

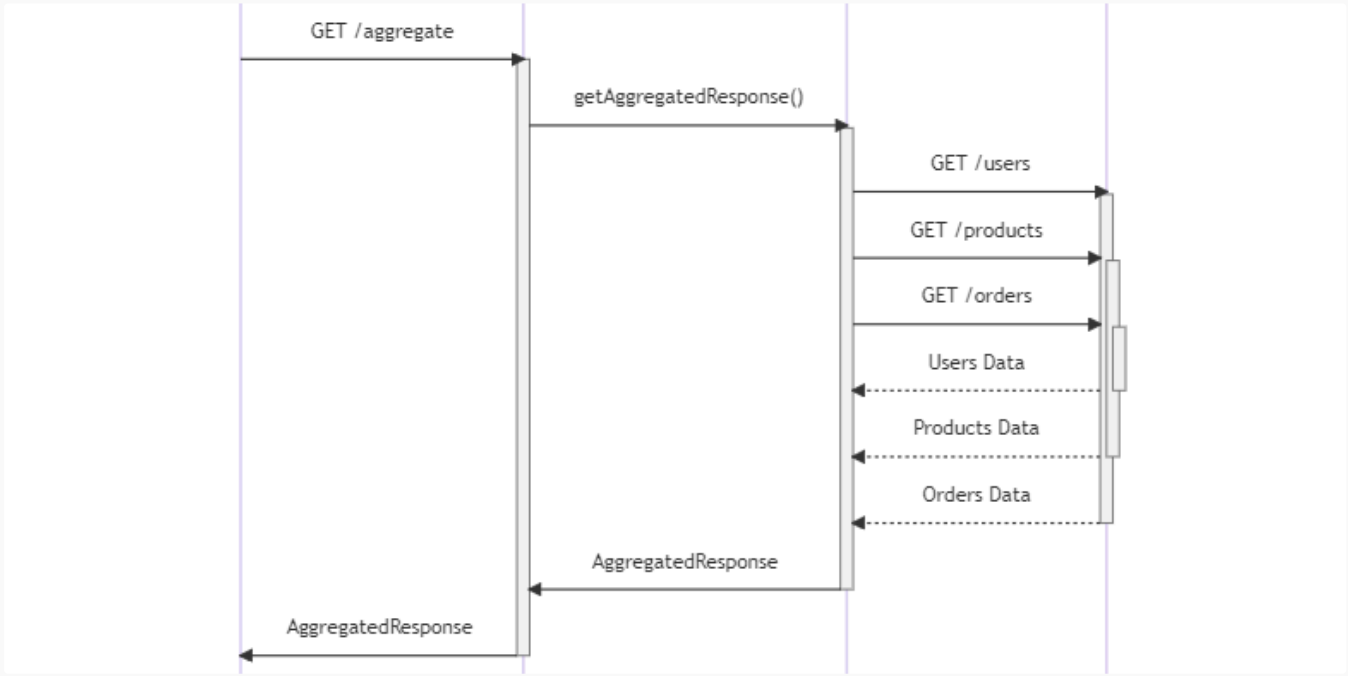
Oct 31, 2023 137 2

In Javarevisited by Sivaram Rasathurai

16 Common REST API Status Code Mistakes to Avoid in 2025

Did you know 70+% API bugs come from misused status codes?

Feb 26 483 9



In Javarevisited by Srikanth Dannarapu

Java CompletableFuture

CompletableFuture is a class introduced in Java 8 that allows us to write asynchronous, non-blocking code. It is a powerful tool that can...

Mar 14, 2023 985 10

In Javarevisited by Aziz Kale

RESTful API Testing in Java with Mockito (Service Layer)

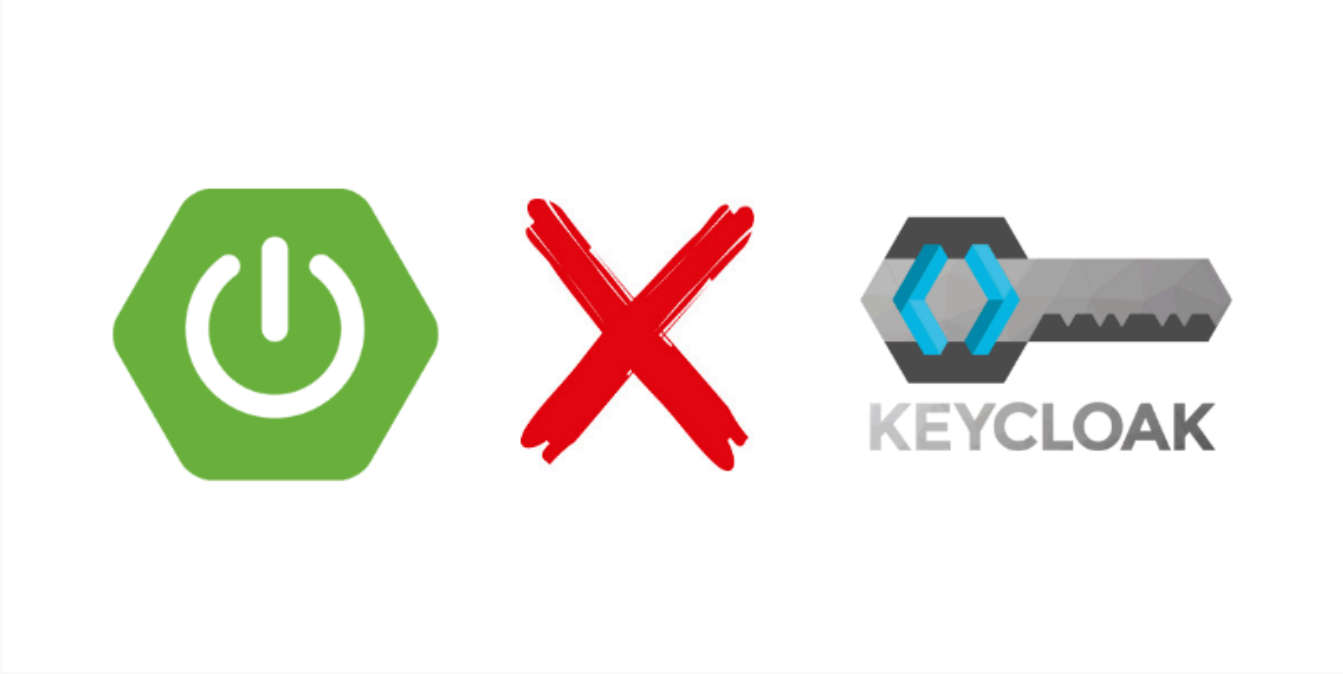
Mockito is a popular Java library used for creating and working with mock objects in unit testing. It allows you to simulate the behavior...

Oct 19, 2023 133 2

See all from Aziz Kale

See all from Javarevisited

Recommended from Medium



MO72

Securing Spring Boot Applications with Keycloak: A Step-by-Step Guide to User Authentication and...

Oct 18, 2024 2

```
>> '
{"id":1,"email":"james.bond@gmail.com","password":"$2a$10$yLiLQFZjqaXUoRn2b6lOz.S9RZ
TRZmuUBJBqOpNqxVoMYRm52KIU2","fullName":"james bond","enabled":true,"authorities":[]
,"username":"james.bond@gmail.com","accountNonLocked":true,"accountNonExpired":true,
"credentialsNonExpired":true}
PS C:\Users\User> curl --location 'http://localhost:8080/auth/login' `
>> --header 'Content-Type: application/json' `
>> --data-raw '{
>>     "email":"james.bond@gmail.com",
>>     "password":"abcd"
>> }
>> '
{"token":"eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJqYW1lcy5ib25kQGdtYWlsLmNvbSIsImlhdCI6MTcyN
jAxNTA4NiwiZXhwIjoxNzI2MDE2ODg2fQ.RD53fXqpEU5XMTKu0h2cQuYfdgEZfex60mKcixzmvyw","expi
resIn":1800000}
PS C:\Users\User> curl --location 'http://localhost:8080/users/me' `
>> --header 'Authorization: Bearer eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJqYW1lcy5ib25kQGdt
YWlsLmNvbSIsImlhdCI6MTcyNjAxNTA4NiwiZXhwIjoxNzI2MDE2ODg2fQ.RD53fXqpEU5XMTKu0h2cQuYfd
gEZfex60mKcixzmvyw'
```

In Dev Genius by Mohamad Mahmood

Spring Boot 3 Auth REST API with JWT Bearer Token

Spring Boot is a powerful framework that simplifies the development of Java applications, particularly for creating robust RESTful APIs...

Sep 10, 2024 8

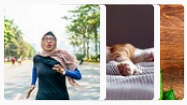
Lists

Staff picks

822 stories · 1645 saves

Stories to Help You Level-Up at Work

19 stories · 947 saves



Self-Improvement 101

20 stories · 3348 saves

Productivity 101

20 stories · 2815 saves

Victor Onu S.

Implementing JWT Authentication in a Simple Spring Boot Application with Java

Let's create a Spring Boot project demonstrating JWT (JSON Web Token) authentication. This example will show how to secure your REST APIs...

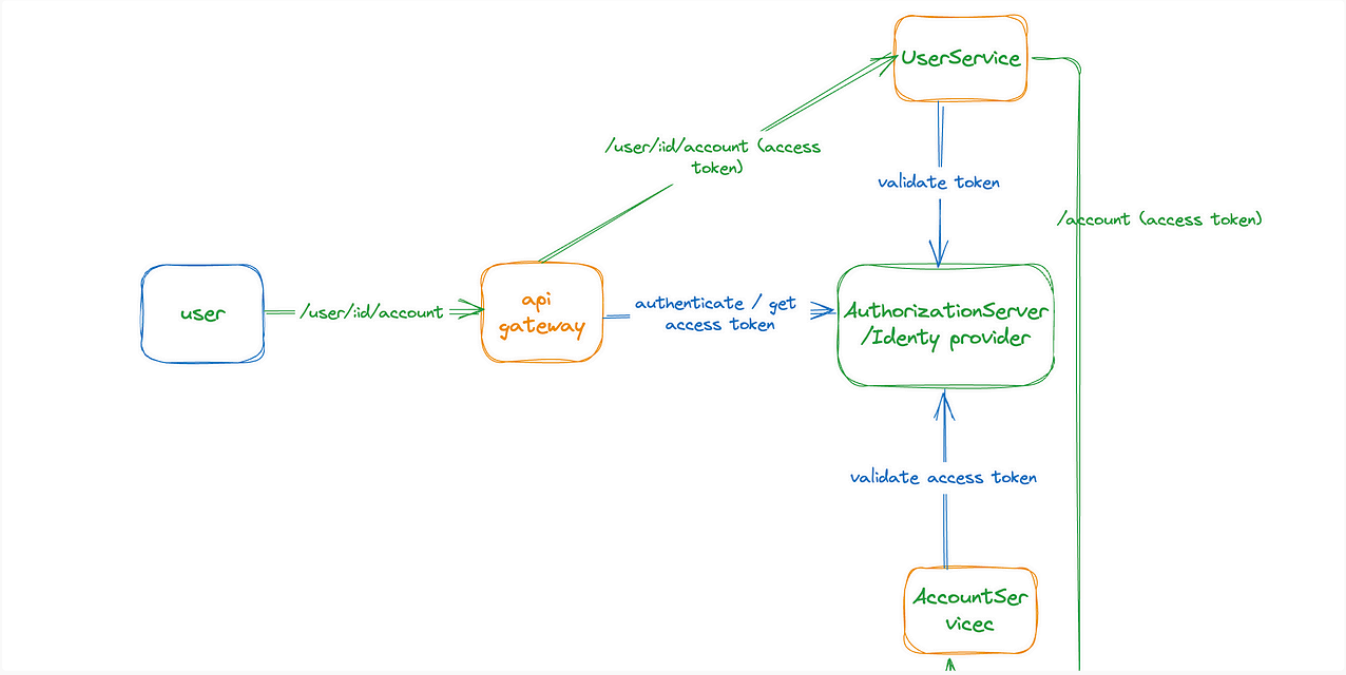
Oct 3, 2024 51

Max Di Franco

User Registration and JWT Authentication with Spring Boot 3: Part 1 — Registration & Login

In this tutorial, we will build a user authentication service using Spring Boot, JWT (JSON Web Tokens), and PostgreSQL. The application...

Sep 23, 2024 89 1



Arun Chalise

Easy OAuth2 in Microservices: Quick Setup with Spring Cloud Gateway & Spring Security

In a MicroServices architecture, key security requirements include:

Oct 21, 2024 14 1

Alon Cohn

Implementing Secure IAM Authentication Between ECS Spring Boot Tasks and RDS

Passwords pose several security vulnerabilities, such as susceptibility to phishing, brute force attacks, and social engineering. Even with...

Oct 28, 2024 2

See more recommendations