✦ Member-only story

Using Keycloak with Spring Boot 3.0

Hands-on guidance to integrate Keycloak with Spring Boot v3

Yasas Sandeepa · Follow
Published in Geek Culture
13 min read · Feb 16

Listen Share

Cover Image (Designed by Author)

Spring Boot 3 is the newest version of the Spring Boot family. This is the first major release of Spring Boot after the release of Spring Boot 2.0 around 4.5 years ago.

It introduced so many features/changes such as Java 17 baseline, enhanced support for native compilation with GraalVM, improved observability with Micrometer Tracing, support for Jakarta EE etc. Visit the Release Notes for more info.

Undoubtedly, one of the major turns in the new release was the dropping of support for older versions of Java. Certain prior libraries have become obsolete and are no longer compatible with the latest release.

The same situation applied to Keycloak as well. We previously used Keycloak Client Adapters to easily connect our applications.

But unfortunately, some of the classes, methods, properties and annotations that were deprecated in Spring Boot 2.x have been removed in this new release.

For example, `WebSecurityConfigureAdapter` class and the `@EnableGlobalWebSecurity` annotation has been removed. Also, methods like `authosizeRequests()` and `antMatchers()` are changed to `authosizeHttpRequests()` and `requestMatchers()` respectively.

Now the Keycloak Adapter doesn't start because the [illegible] inherited from


```
package org.keycloak.adapters.springsecurity.config;

import ...

Provides a convenient base class for creating a WebSecurityConfigurer instance secured by Keycloak. This
implementation allows customization by overriding methods.
See Also: EnableWebSecurity,
          EnableWebMvcSecurity
Author:   Scott Rossillo
Version:  $Revision: 1 $
public abstract class KeycloakWebSecurityConfigurerAdapter extends WebSecurityConfigurerAdapter implements WebSecurityConf

    @Value("${keycloak.configurationFile:WEB-INF/keycloak.json}")
    private Resource keycloakConfigFileResource;
    @Autowired(required = false)
    private KeycloakConfigResolver keycloakConfigResolver;
```

WebSecurityConfigureAdapter removed from Spring Boot 3.0 (Image by author)

But now you don't need to worry about it!

This article will help you in integrating Keycloak into Spring Boot 3.0 application, whether you are migrating from Spring Boot 2.x or starting anew.

I'll start with the Keycloak authorization flow to teach you the concepts behind this.

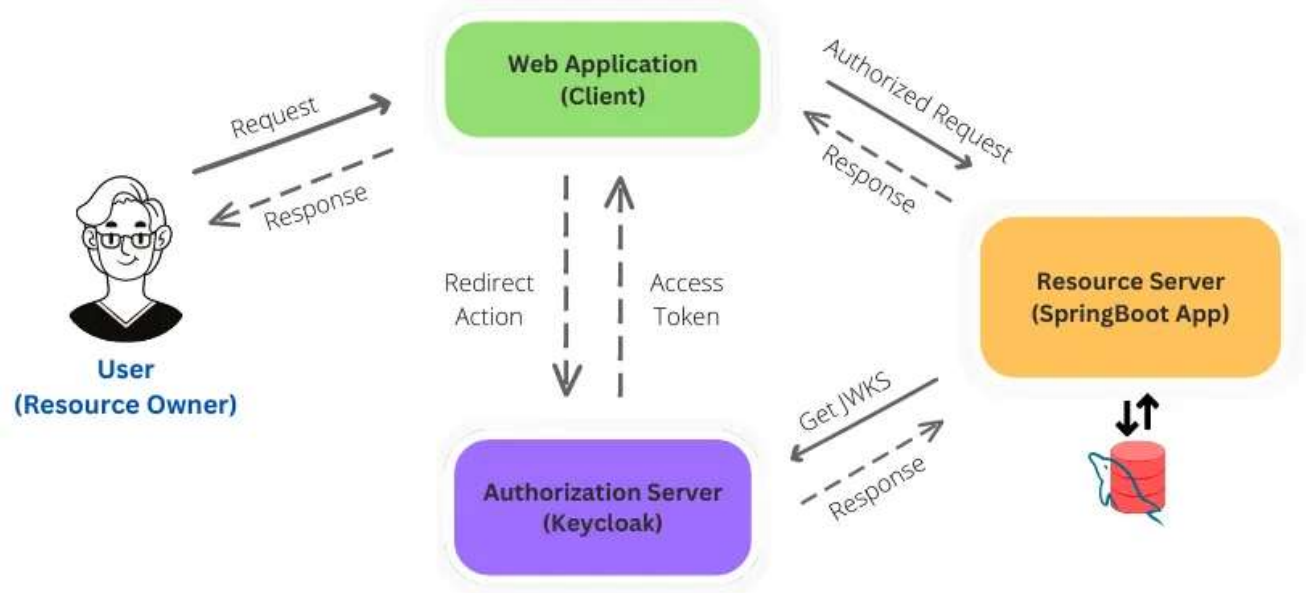
Keycloak Authorization Flow

As you all know, Keycloak is an open-source identity and access management (IAM) tool. It secures our applications and services with little to no code.

Keycloak implements almost all standard IAM protocols like OAuth 2.0, OpenID, and SAML. So we can use one of these protocols to connect with Keycloak.

The best way is directly using an OAuth library to integrate Keycloak with our application. Then you don't need any Keycloak specific client libraries or adapters to communicate with Keycloak.

You can understand the standard three-step OAuth 2 authentication scheme from the below diagram.



OAuth2 Authentication scheme (Image by author)

The added advantage of this approach is, we can easily move from Keycloak to another authorization server/service like OneLogin, Auth0, Asgardeo, Okta etc.

Let's see how to implement this architecture.

Setting up a Keycloak server

As the first thing you will have to set up a Keycloak server. Either you can install it locally or deploy a server in a cloud platform.

The official Keycloak website provides comprehensive and well-organized documentation. So you can refer to it and configure this easily.

All Keycloak releases are available in the following GitHub repository. I used the latest Keycloak-20.0.3 Standalone server distribution for my demonstration.

Releases · keycloak/keycloak

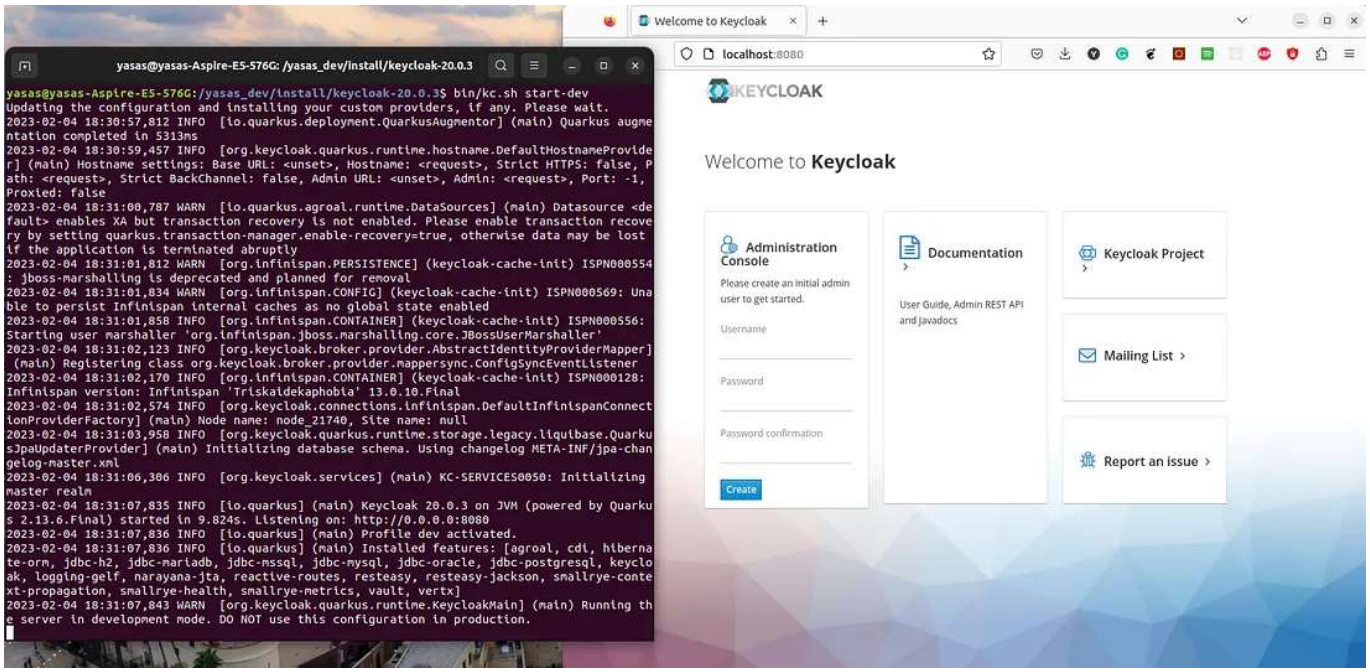
github.com

> *Make sure you have OpenJDK 11 or newer installed to run the latest version of Keycloak.*

Extract the downloaded Keycloak zip/tar.gz file.

```
unzip keclock.ss
cd keycloak-20.0.3
bin/kc.sh start-dev
```

If you are a Windows user run `bin/kc.bat start-dev` to start the server.

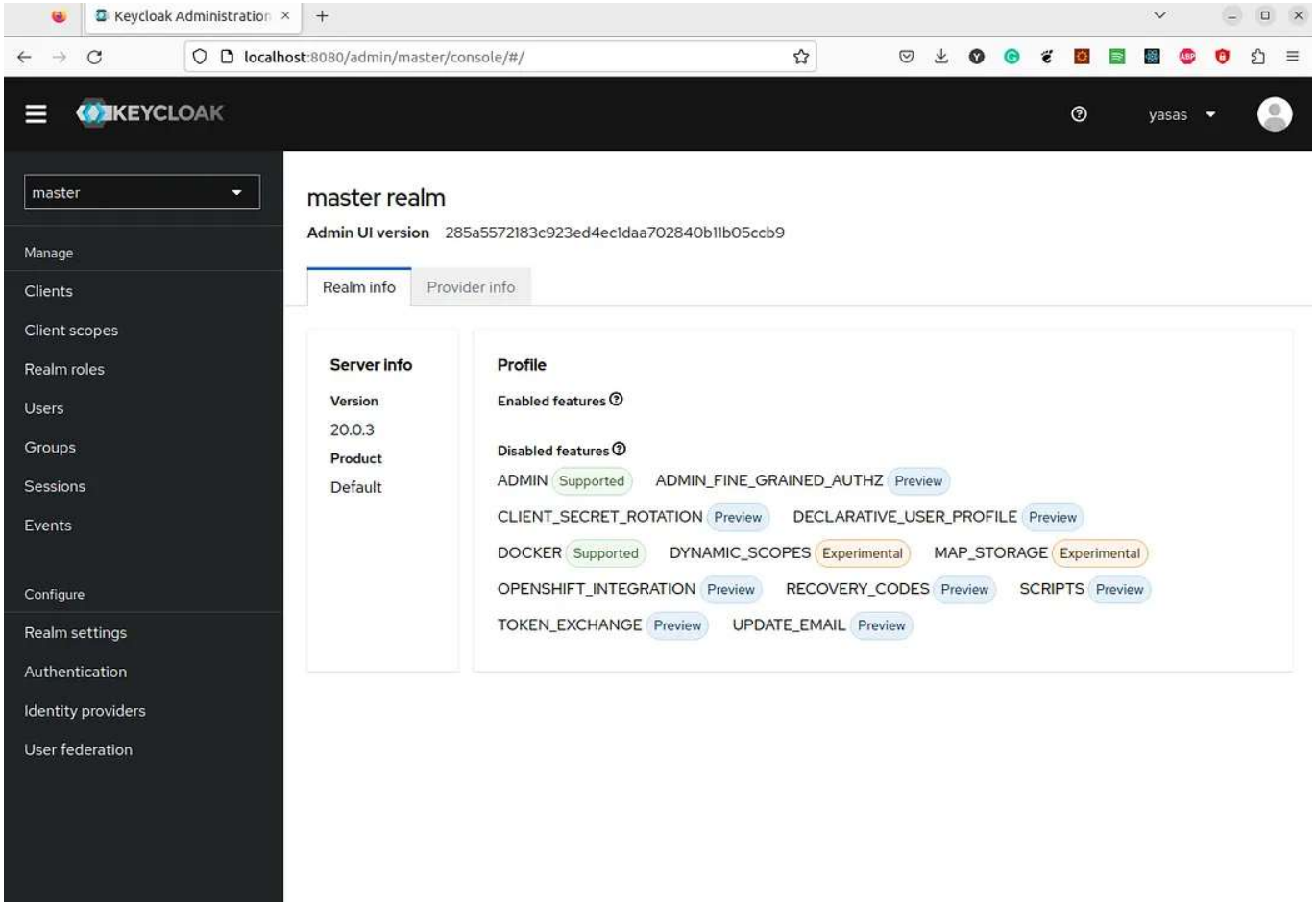


Keycloak server running in localhost (Image by author)

Once all the services started, navigate to http://localhost:8080 using the browser. You can see the Welcome screen of the Keycloak server.

To begin with, you need to create an initial admin user by providing Username and a Password. Then click on the Administration Console link to access the Keycloak admin console.

Now you are connected to the Keycloak Admin Console.



Keycloak Admin Console (Image by author)

There are a few more steps to configure the Keycloak server.

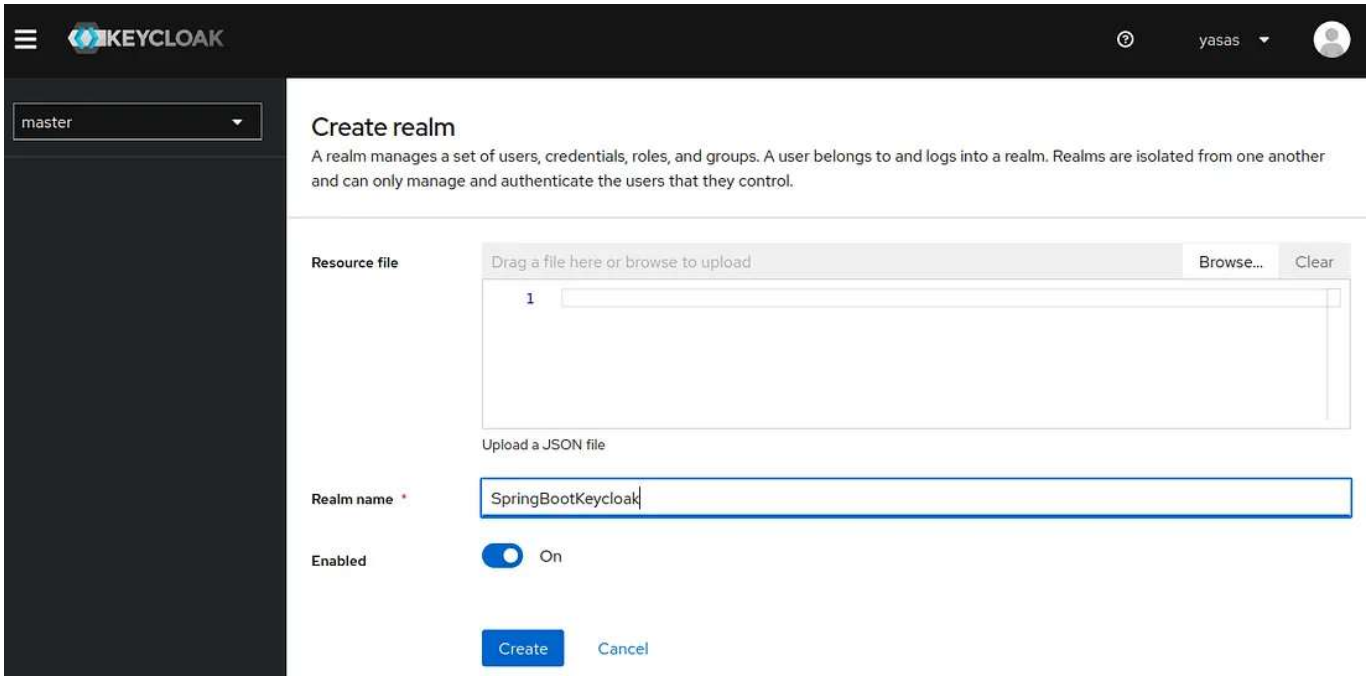
1. Creating a Realm

As the first thing in the Keycloak configuration, we need to create a Realm.

What is a Realm? It is a management entity that controls a set of users, their credentials, roles, and groups. A user belongs to and logs into a realm.

Imagine you step into a large shopping mall with many stores. This mall represents Keycloak, and the individual stores correspond to your realm.

Let's create a new Realm. From the **Master** drop-down menu, click **Add Realm** and type **SpringBootKeycloak** in the **name** field and click **Create** button.



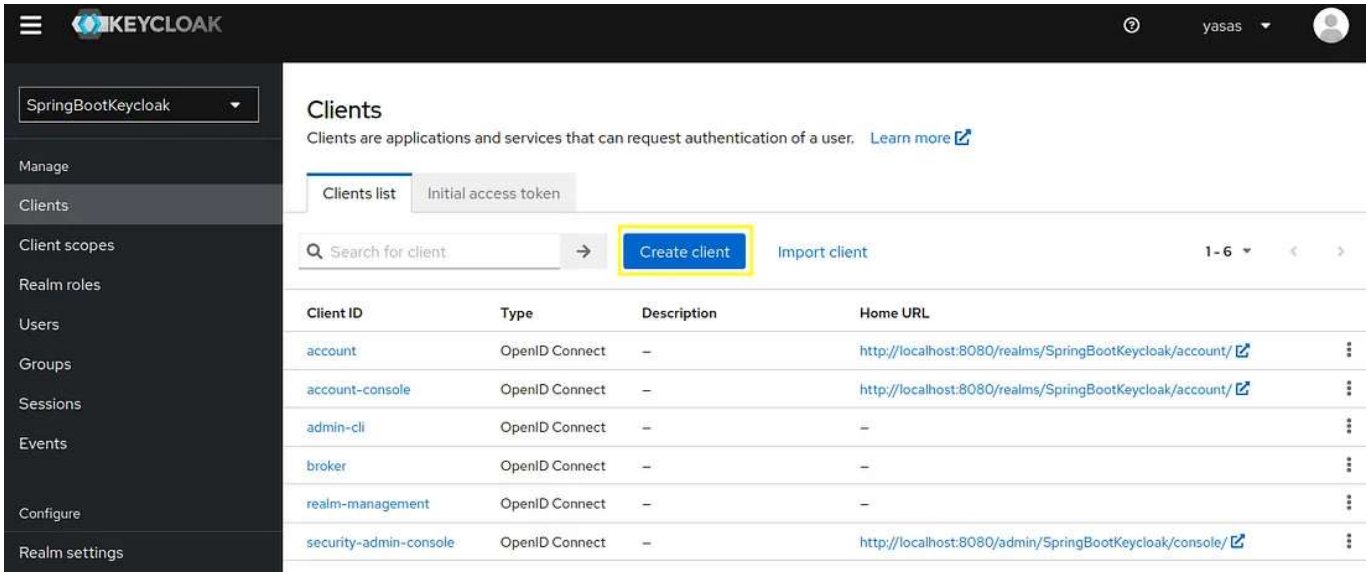
Creating a Realm (Image by author)

Make sure `SpringBootKeycloak` is selected for the below configurations. Avoid using the master realm.

2. Creating a client

As per the official doc, clients are entities that can request Keycloak to authenticate a user. A client represents a resource that can be accessed by certain users. Keycloak comes with a set of inbuilt clients that are for its internal use.

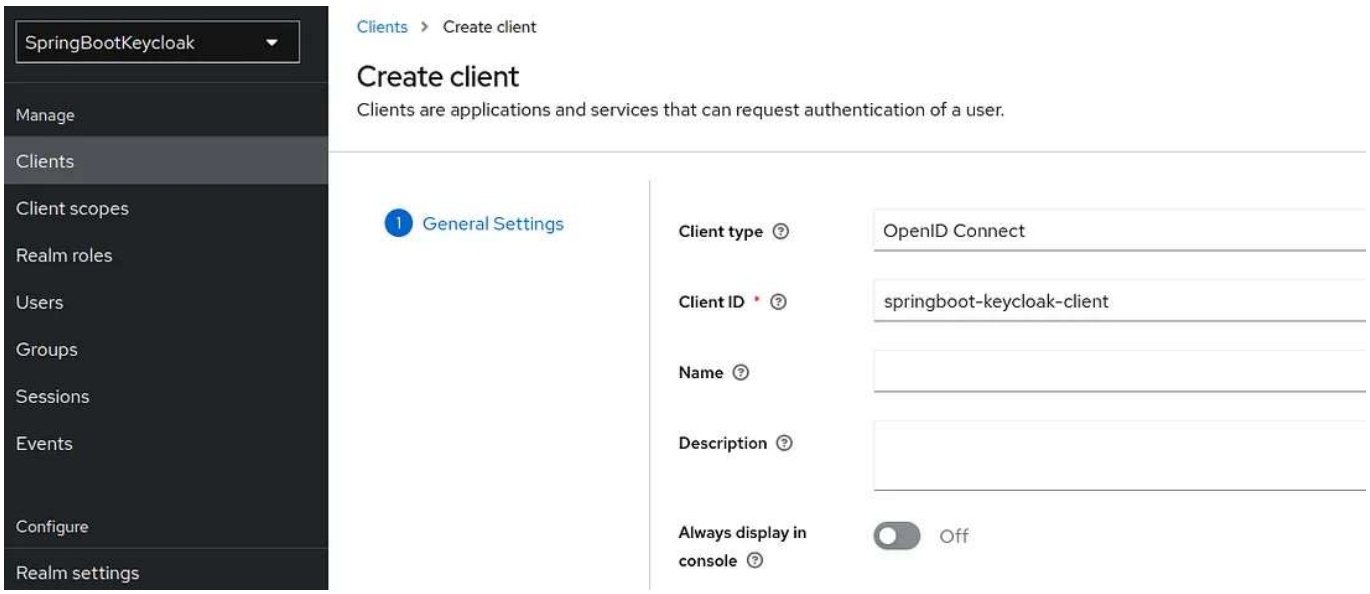
To create a new client, click on the **Clients** menu from the left pane and click the **Create client** button.



Creating a Client (Image by author)

Then you will be prompted for a **Client type**, a **Client ID** and other basic info. Keep the client type as same as OpenID Connect.

I'll create the client as `springboot-keycloak-client` You can give a name and description to the client if needed.

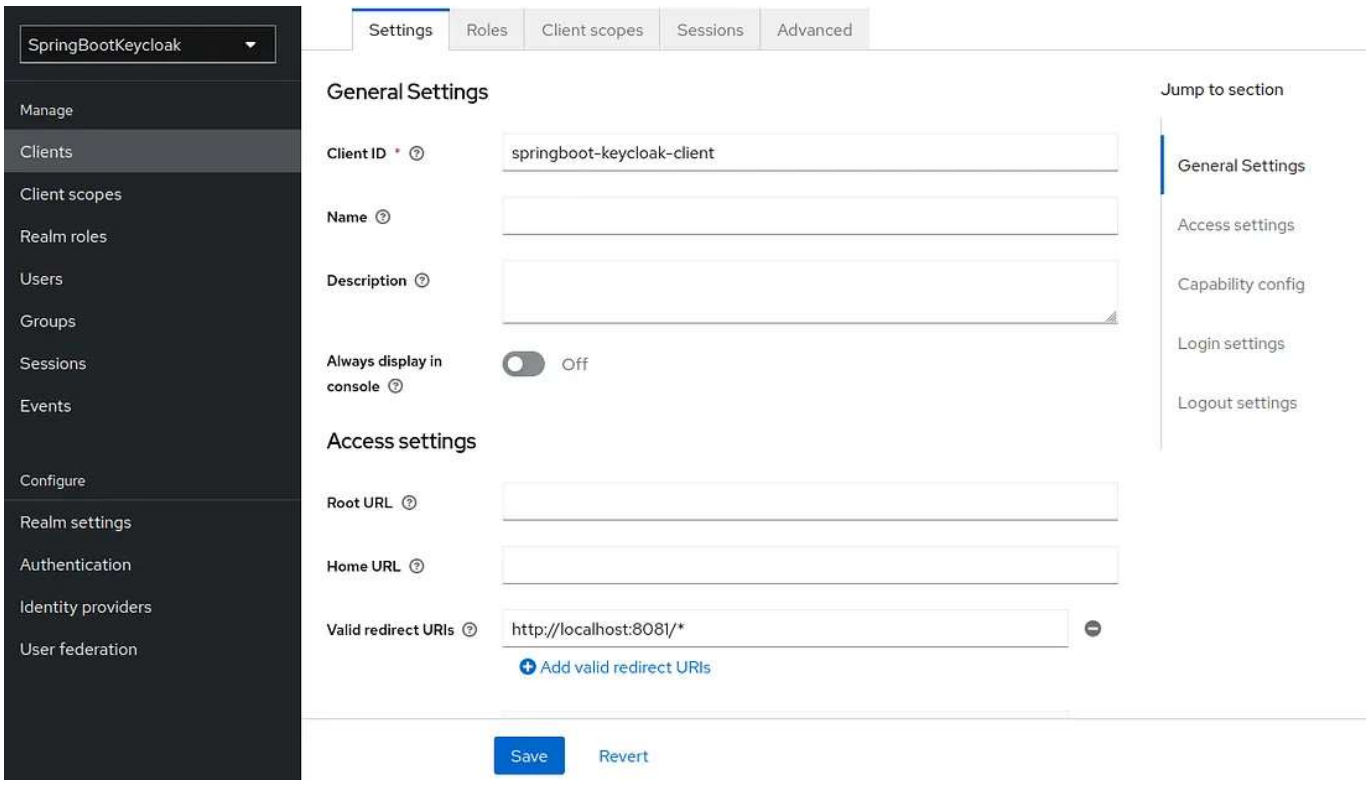


Adding client details (Image by author)

In the next screen, keep the default configs and save. Now the client is successfully created.

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

Our Keycloak server is running on port 8080. So I intend to run my Spring Boot application on port 8081. Let's give `http://localhost:8081/*` as redirect URIs and click the save button.



Configuring additional client settings (Image by author)

3. Creating Roles

Every user must have a role as Keycloak uses Role-Based Access. This allows us to have different types of users with different user permissions.

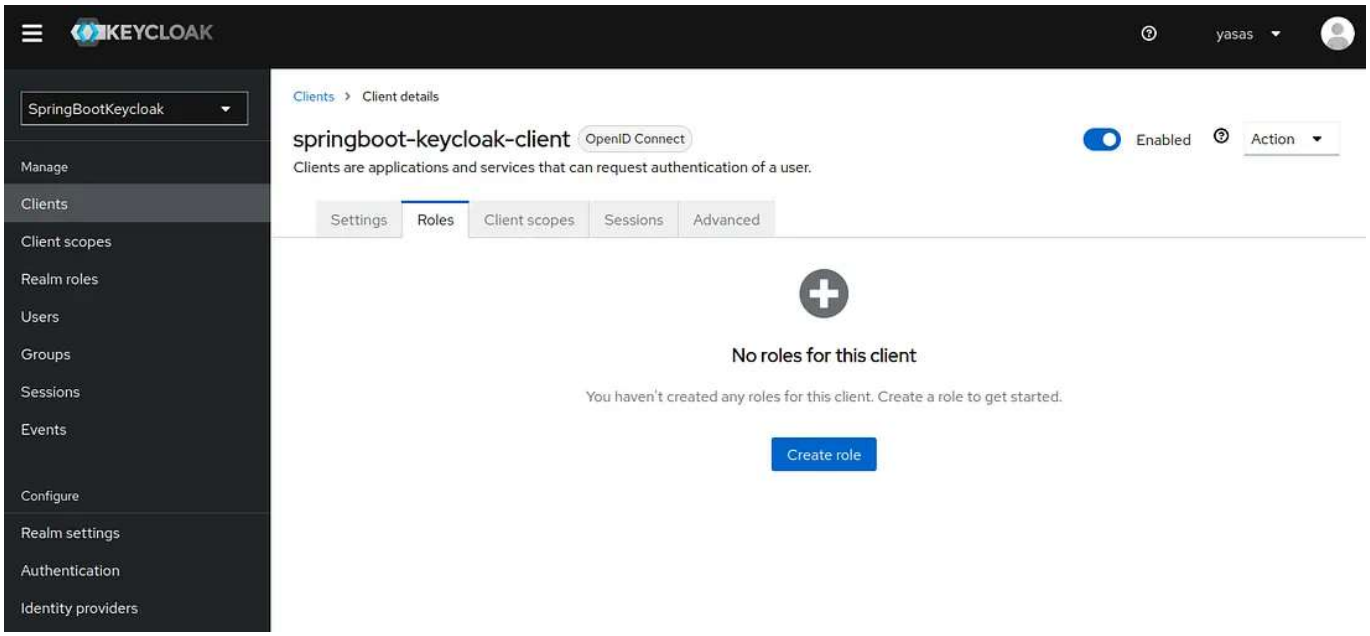
Mainly there are 2 role types in Keycloak.

1. **Realm Role:** It is a global role, belonging to that specific realm. This can access from any client and map it to any user. Global Admin and Admin roles can be considered examples of this.
2. **Client Role:** It is a role which belongs only to that specific client. These roles cannot be accessed from a different client. This can only map to the users from that client. *Example Roles:* Employee, User etc.

A role can be combined from multiple roles. Then it becomes a **Composite Role.**

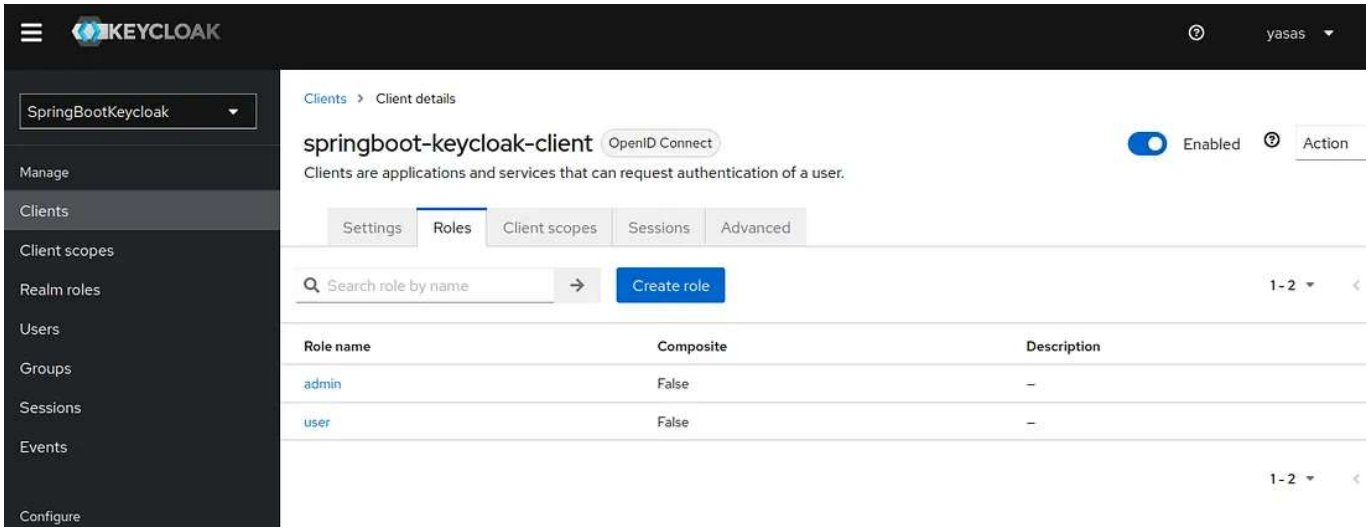
Let's create client roles first. I'll create user and admin as the client roles.

Navigate to clients from the sidebar. Then select the client `springboot-keycloak-client` and click `Roles` Press the Create role button.



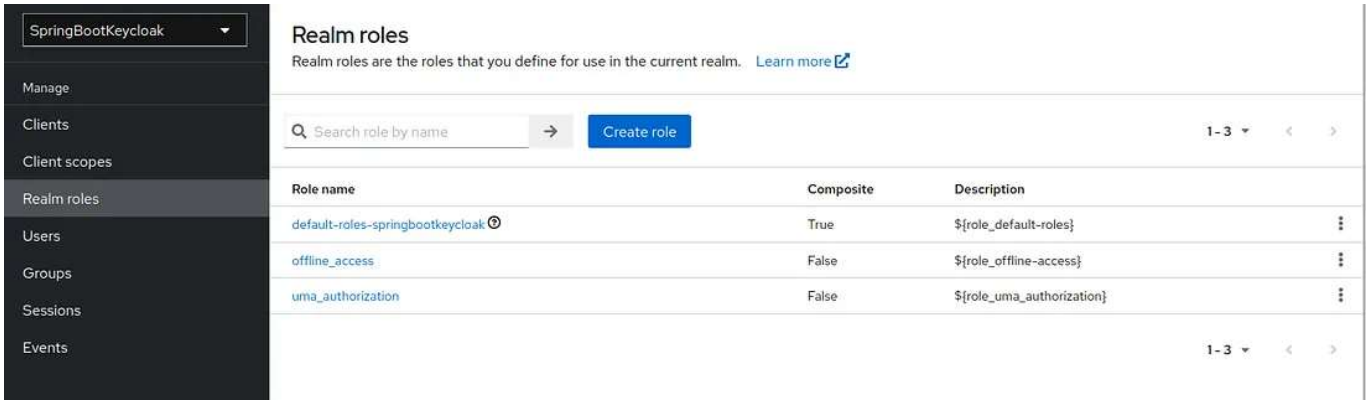
Creating Client Roles (Image by author)

Create the two roles previously mentioned here.



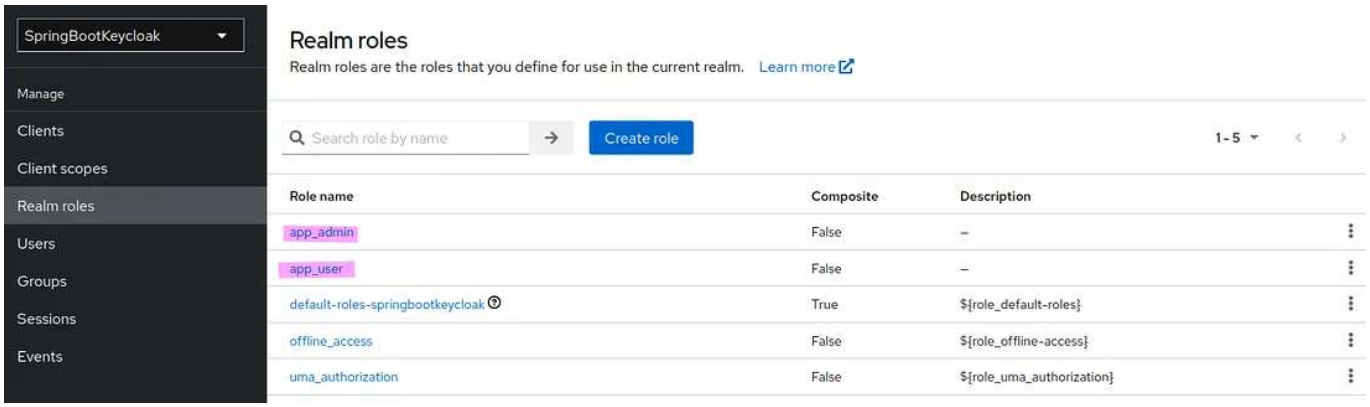
Client Roles after adding user and admin roles (Image by author)

Alright! Now we need to create Realm roles. Navigate to the *Realm Roles* page to create roles.



Creating Realm Roles (Image by author)

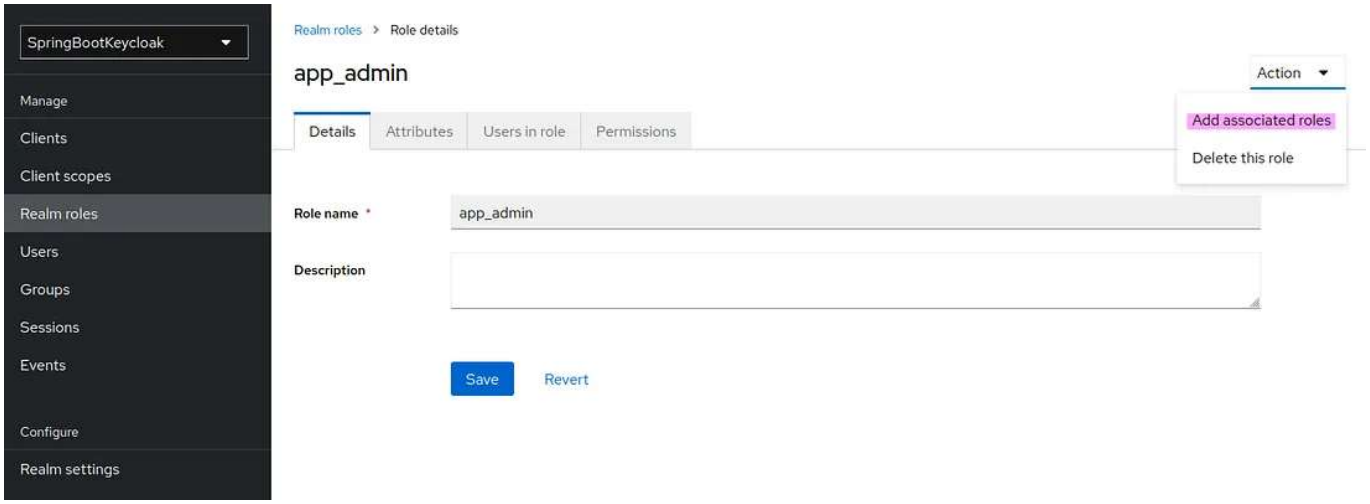
Let's create two Realm roles named `app_admin` and `app_user`



Realm Roles after adding app_user and app_admin roles (Image by author)

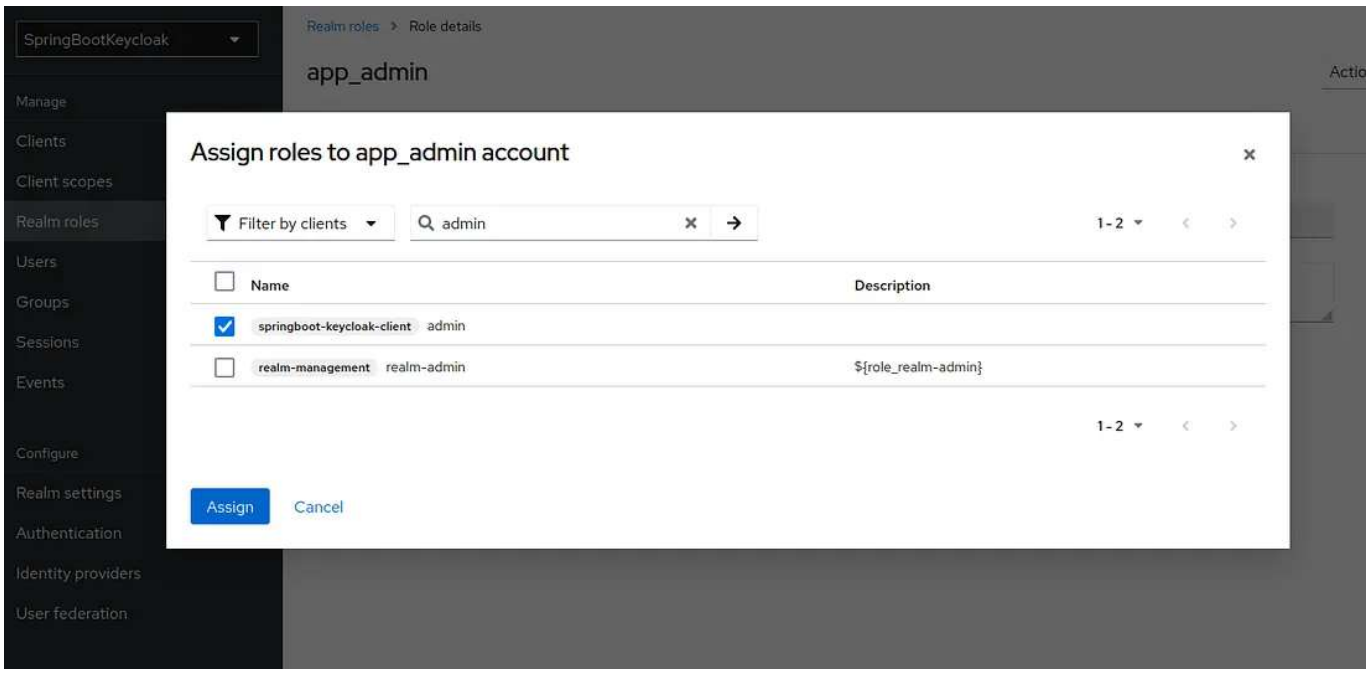
Then `app_user` Realm role needs to composite with `user` Client role and `app_admin` needs to composite with `admin` role.

Click one of the roles and select `Add associated roles` in the Action drop down.



Composite Roles (Image by author)

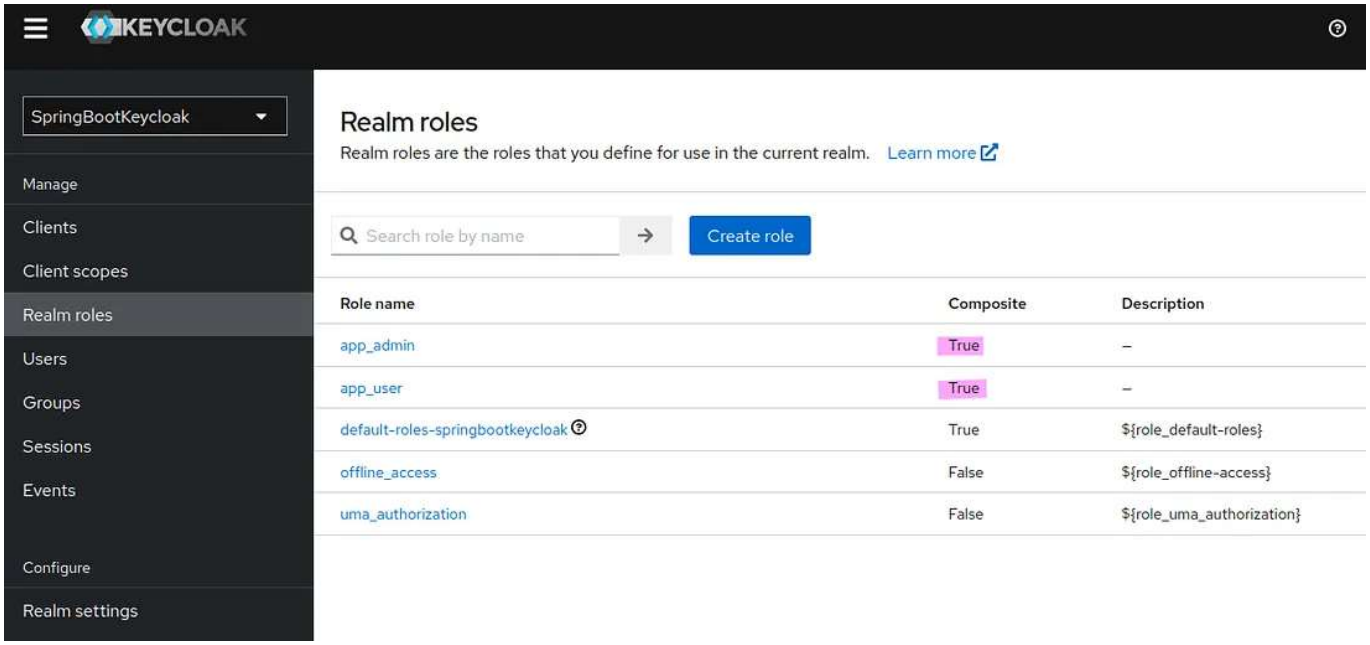
Change the filter to Filter by clients and search the admin role. Then select the admin and click assign.



Assigning roles (Image by author)

Follow the same steps to composite the `app_user` but assign `user` role instead of `admin` role.

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



The final outcome of Realm roles (Image by author)

4. Creating Users

We created Realms, Clients and Roles. The only part that needs to be added is the Users.

Let's head on to the **Users** page and add some users.

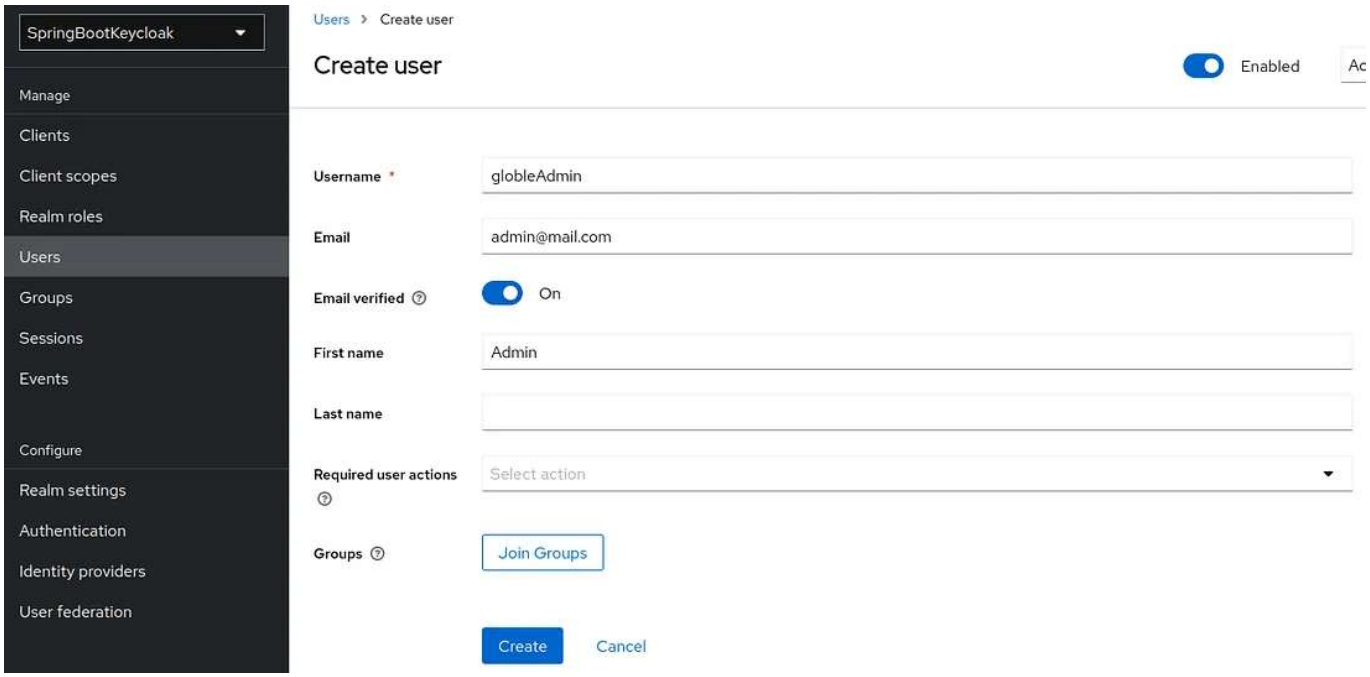
Users are entities that are able to log into your system. Users are created in the sign-up process. This can be performed either through API requests or manually. For this demonstration, I add the users manually.

Let's create the following users and grant them `app_user` and `app_admin` roles as mentioned.

- globalAdmin with `app_admin` realm role.

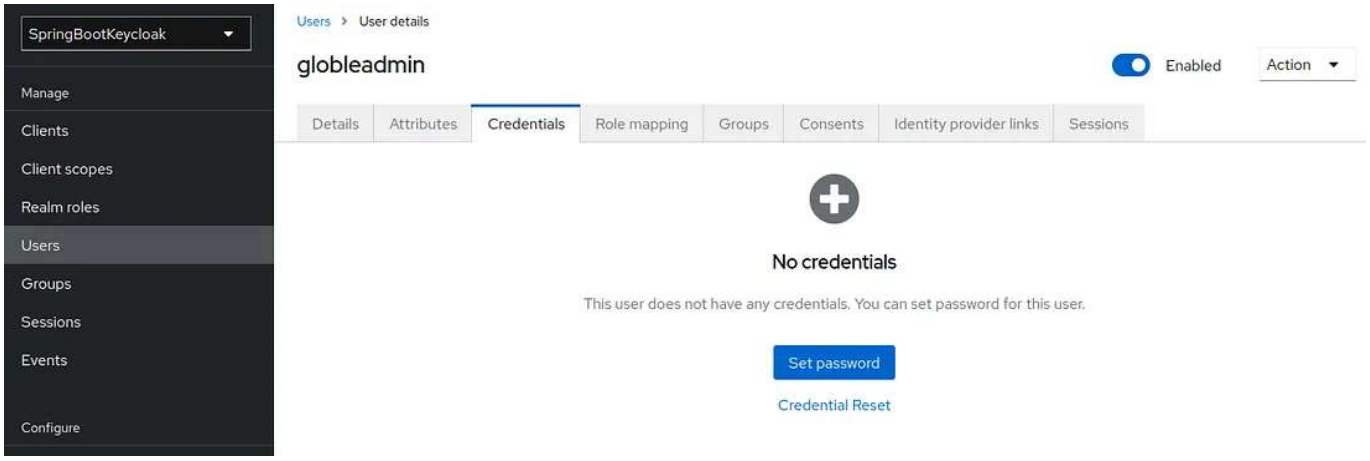
- user1 with `app_user` realm role.
- user2 with `app_user` & `app_admin` realm roles.

In the Users page, click on Create user. Now you can add the user details. Make sure to turn on the email-verified slider.



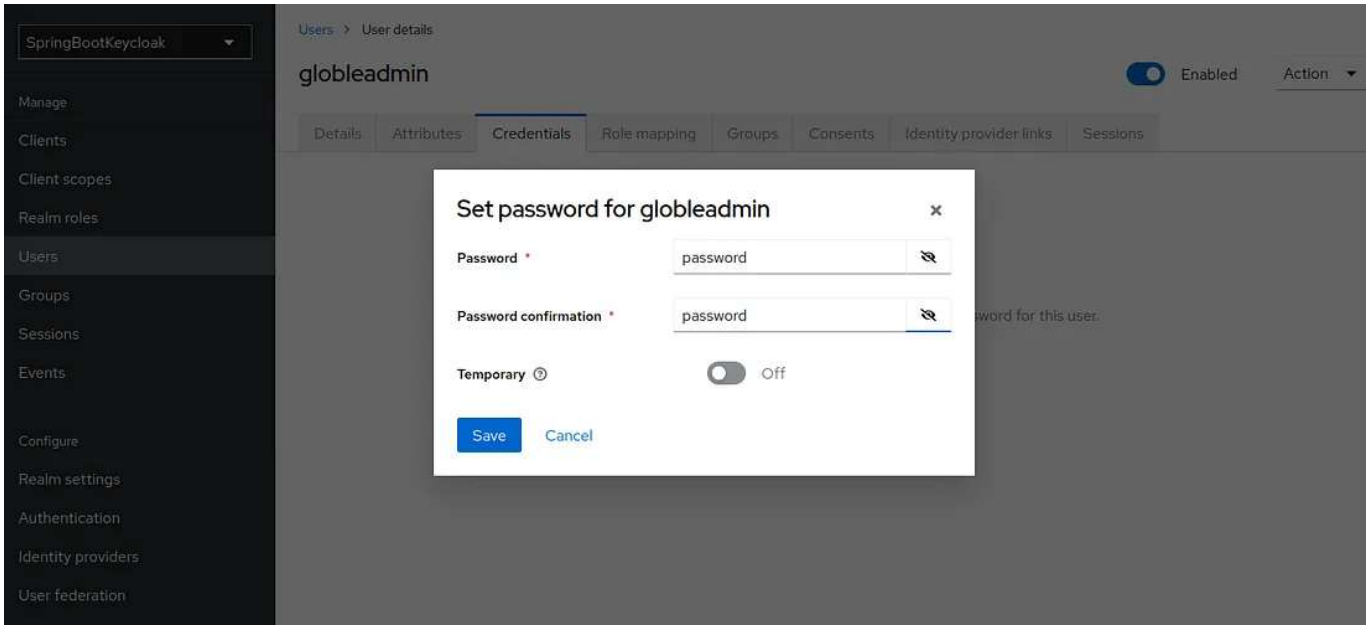
Creating a user (Image by author)

Now we need to set a password for the user. Click the **Credentials** tab and press **Set password** button.



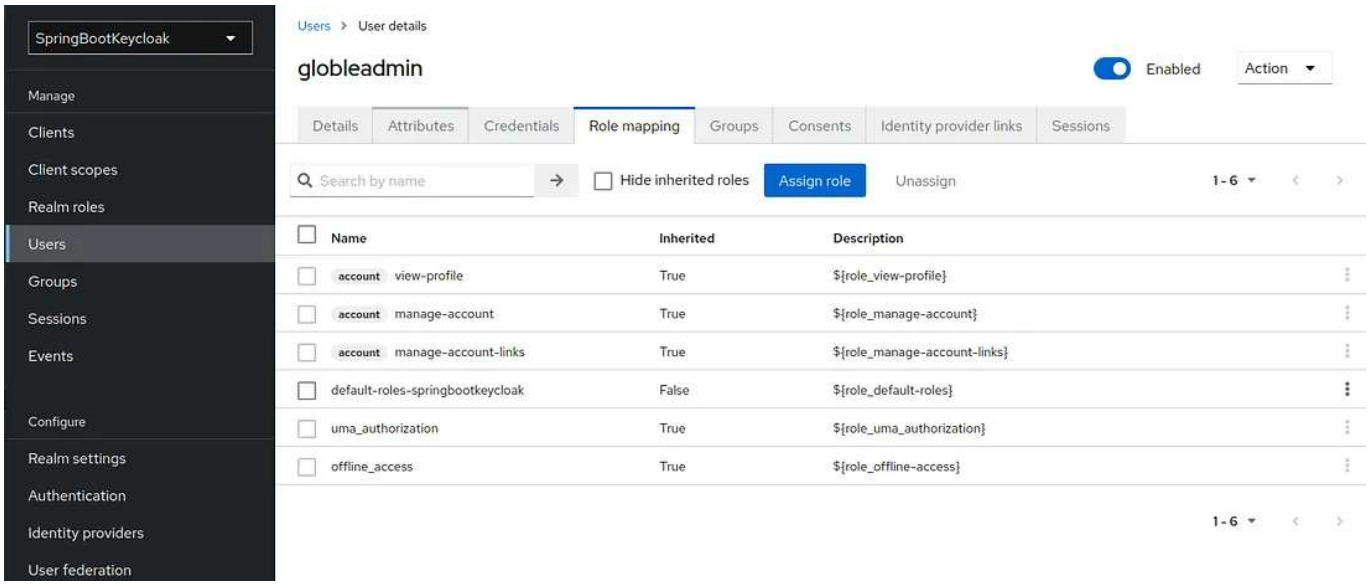
Adding user credentials (Image by author)

Here you need a give a password and confirm it. You also need to turn off the **Temporary** switch. For simplicity let's set the password to `password` for all the users.



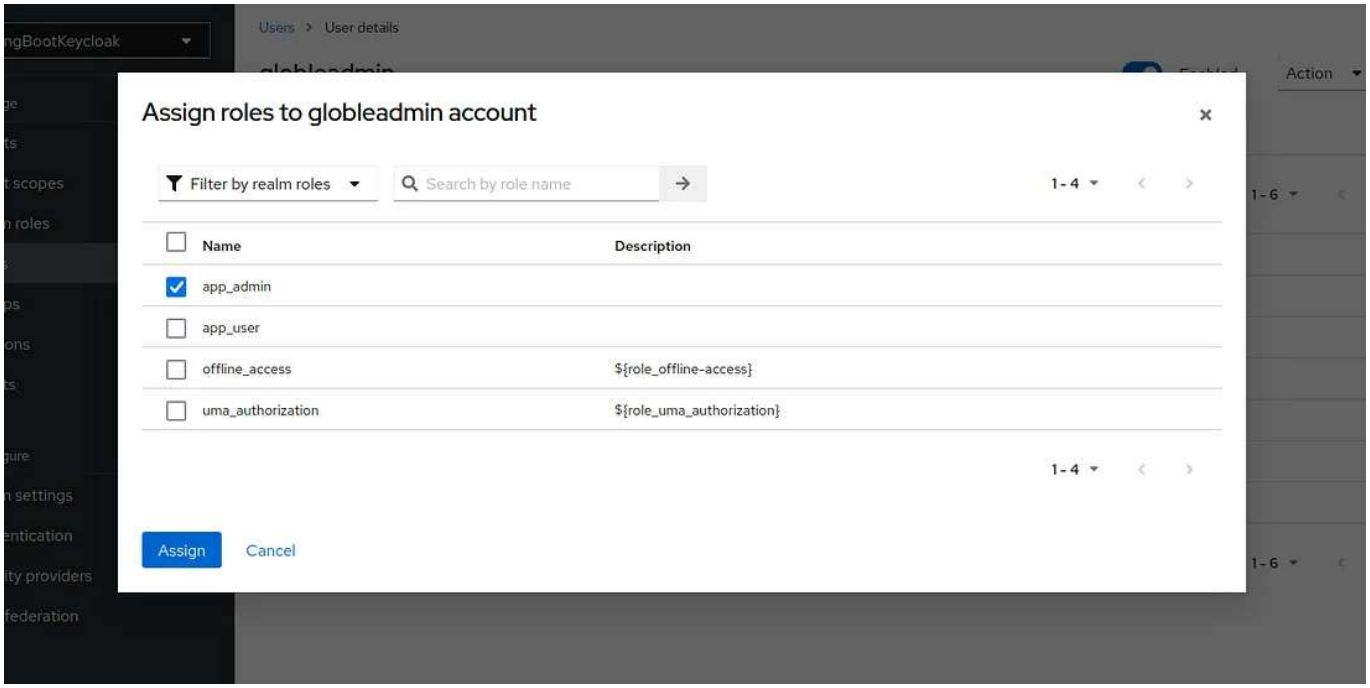
Setting up a user password (Image by author)

Now we need to set a role for the user. Click the **Role mapping** tab and press **Assign role**.



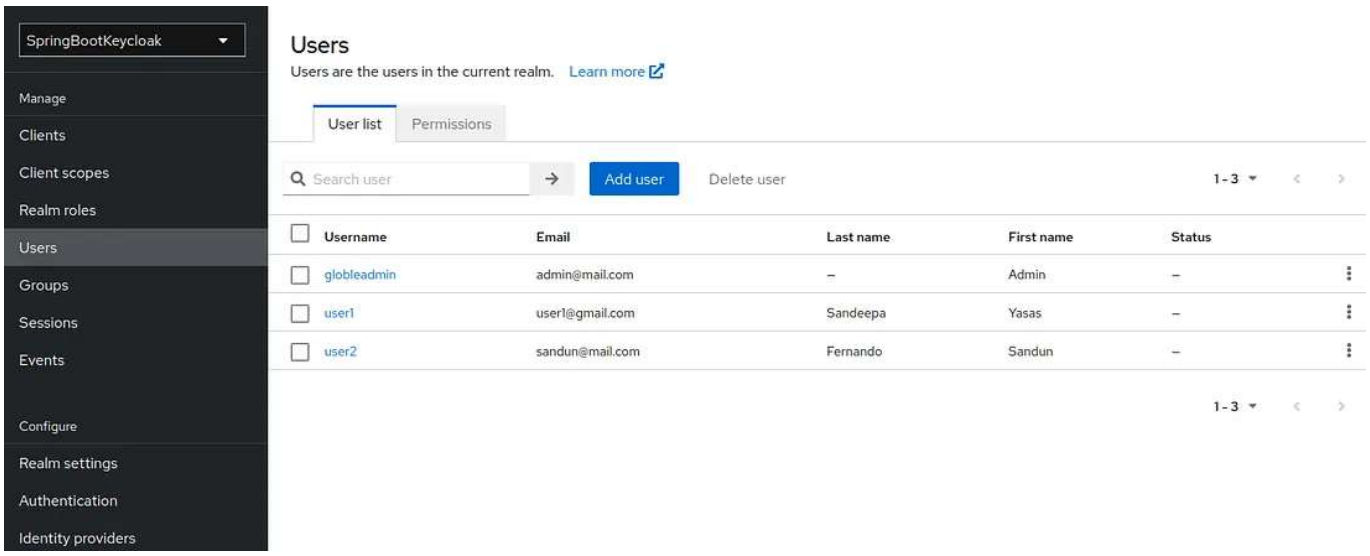
Role mapping (Image by author)

Now the Realm roles list will be available in the table. Check the required role and click **Assign** to map it to the respective user.



Assigning roles (Image by author)

One user is successfully created. You can create the other two users as same as this. Once all users are created, you can see the list of them on the **Users** page.



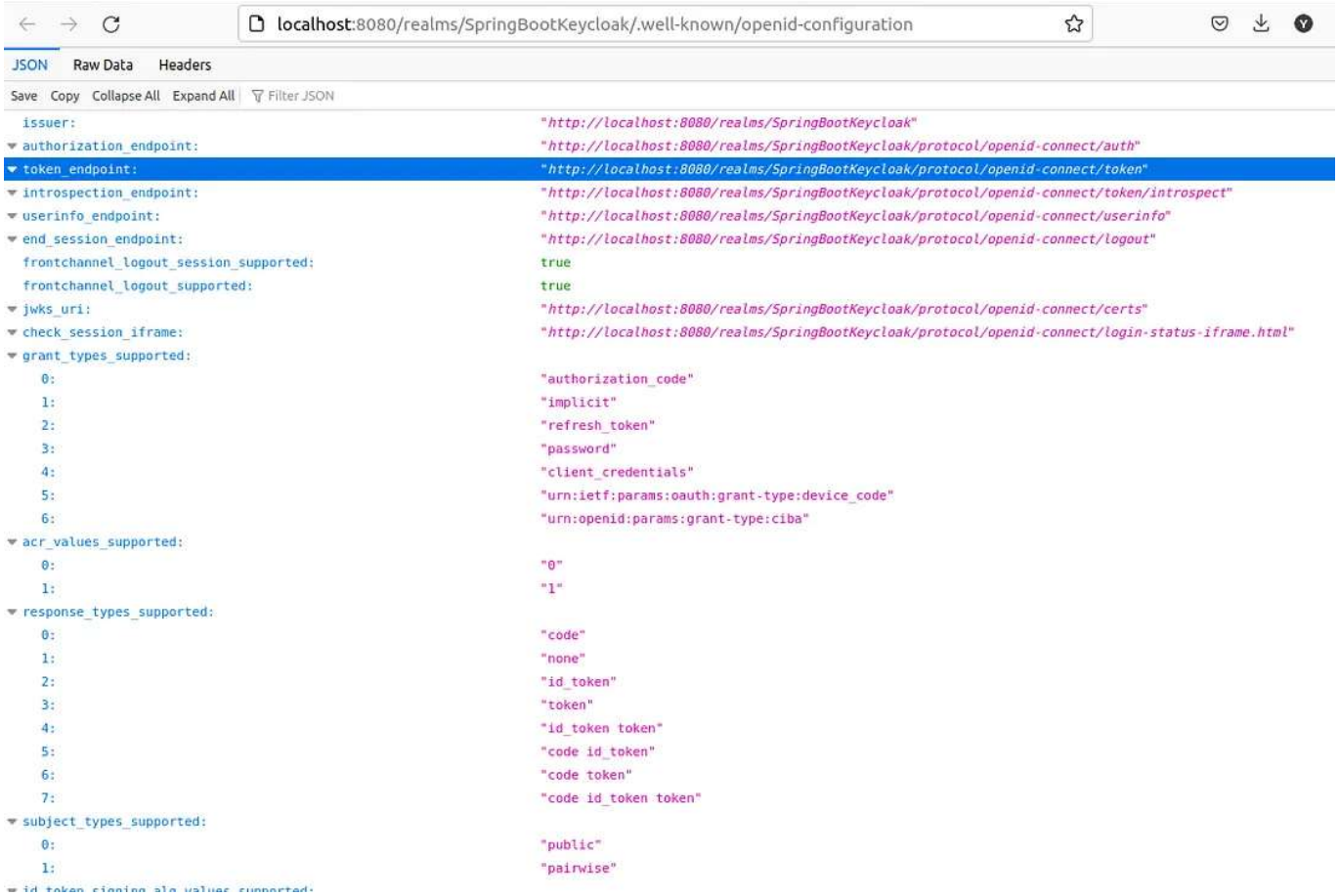
The final result of the users (Image by author)

That's it from the Keycloak configurations. Yes, it was a bit cumbersome to go through all the settings during the initial setup. But when you keep using Keycloak, these configurations will become much easier and more intuitive.

Generating Access tokens with Keycloak API

Now we can test our Keycloak server with its APIs.

You can go to Realm settings and click on the OpenID Endpoint Configuration to see the available endpoints.



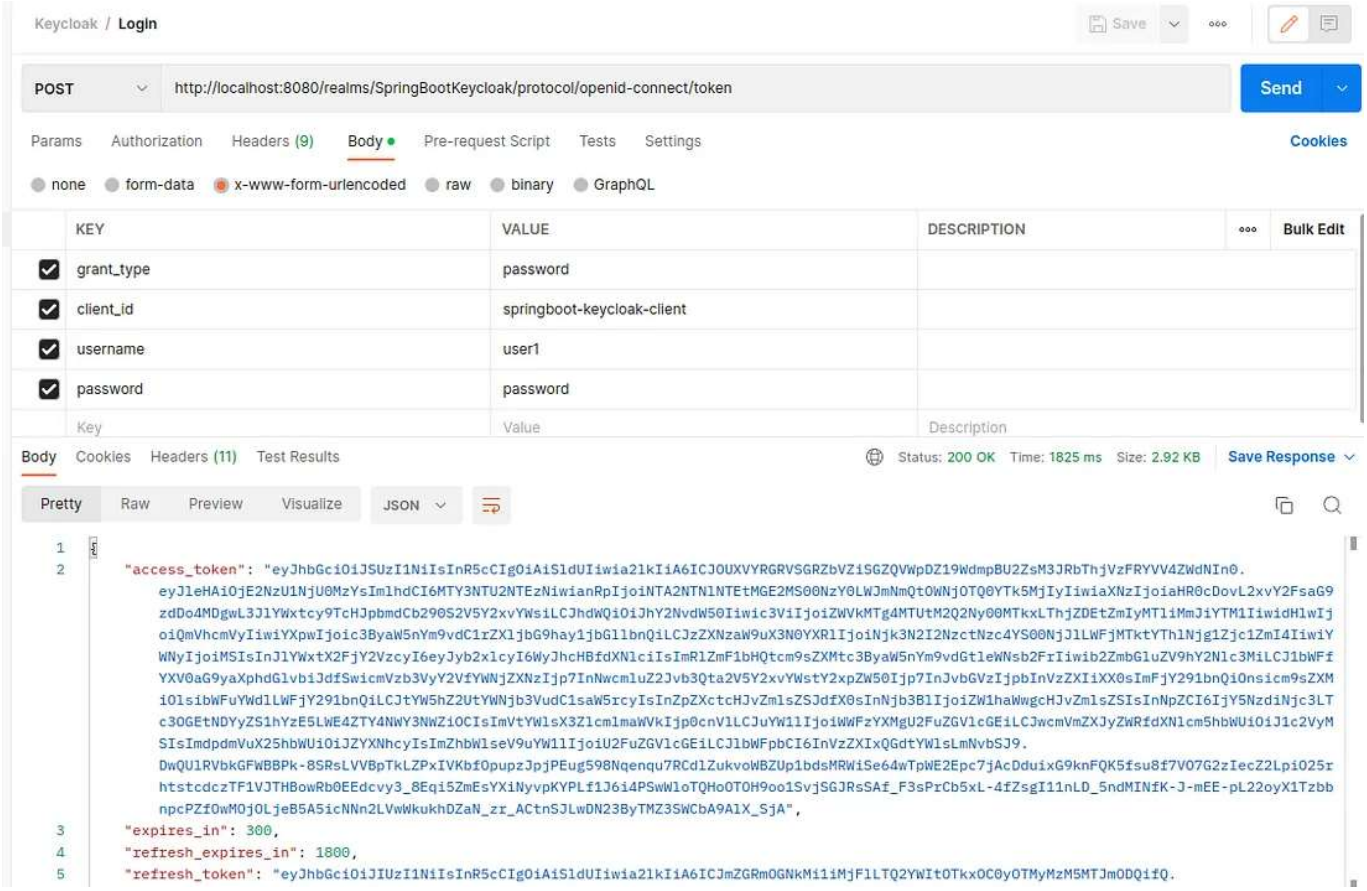
Keycloak available endpoints (Image by author)

You can send a post request to this token endpoint to get a token. You can use one of the users we have already created (`user1` , `user2` or `globalAdmin`) as credentials.

The request body should in a `x-www-form-urlencoded` format. You can use this curl request as an example.

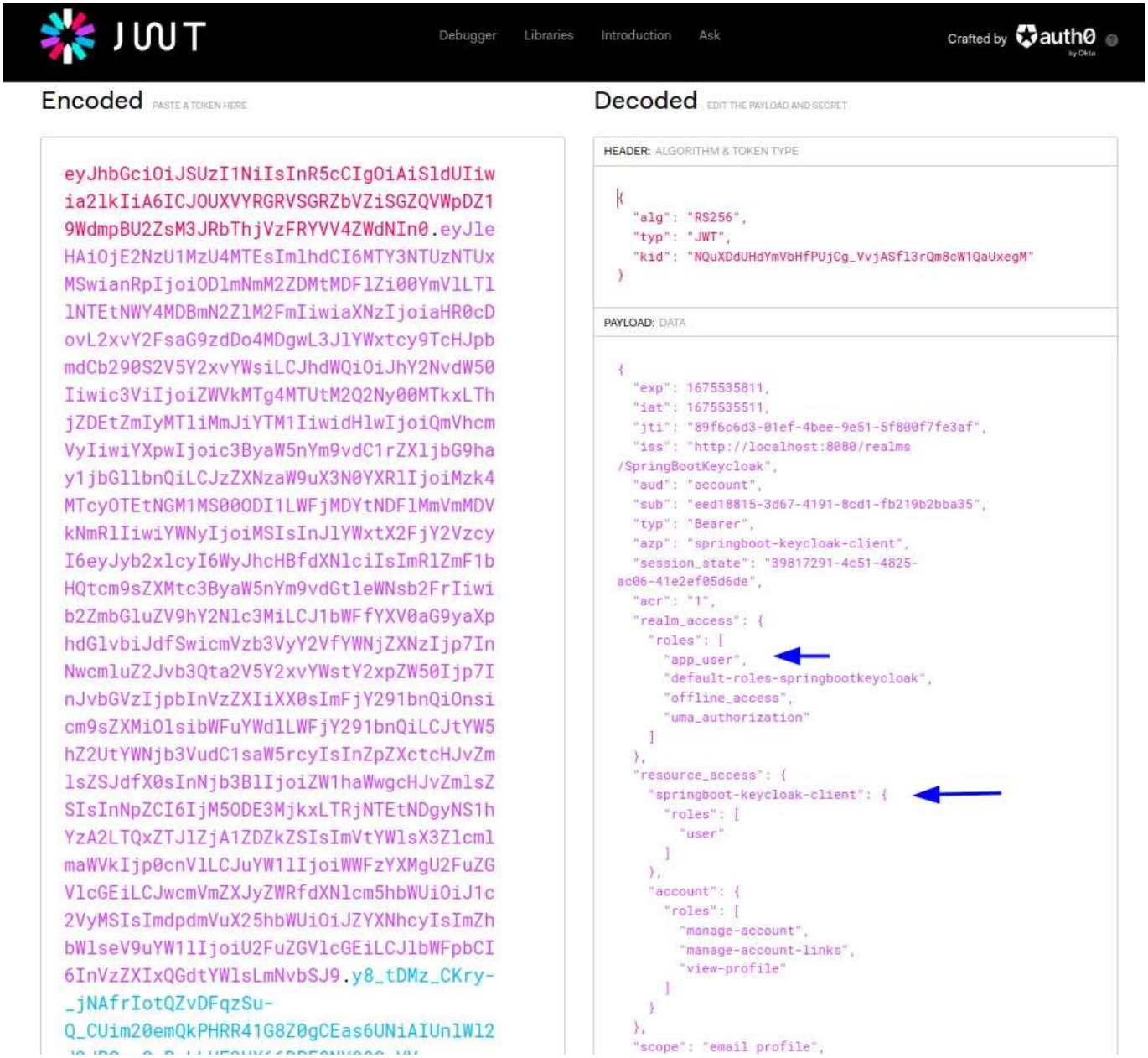
```
curl -X POST 'http://localhost:8080/realms/SpringBootKeycloak/protocol/openid-connect/token' \
 --header 'Content-Type: application/x-www-form-urlencoded' \
 --data-urlencode 'grant_type=password' \
 --data-urlencode 'client_id=springboot-keycloak-client' \
 --data-urlencode 'username=user1' \
 --data-urlencode 'password=password'
```

Or else, you can test it using Postman. The response would look like the one below.



Testing Keycloak token endpoint (Image by author)

Now you can copy that `access_token` and decode it using the jwt.io website to see the information inside that.

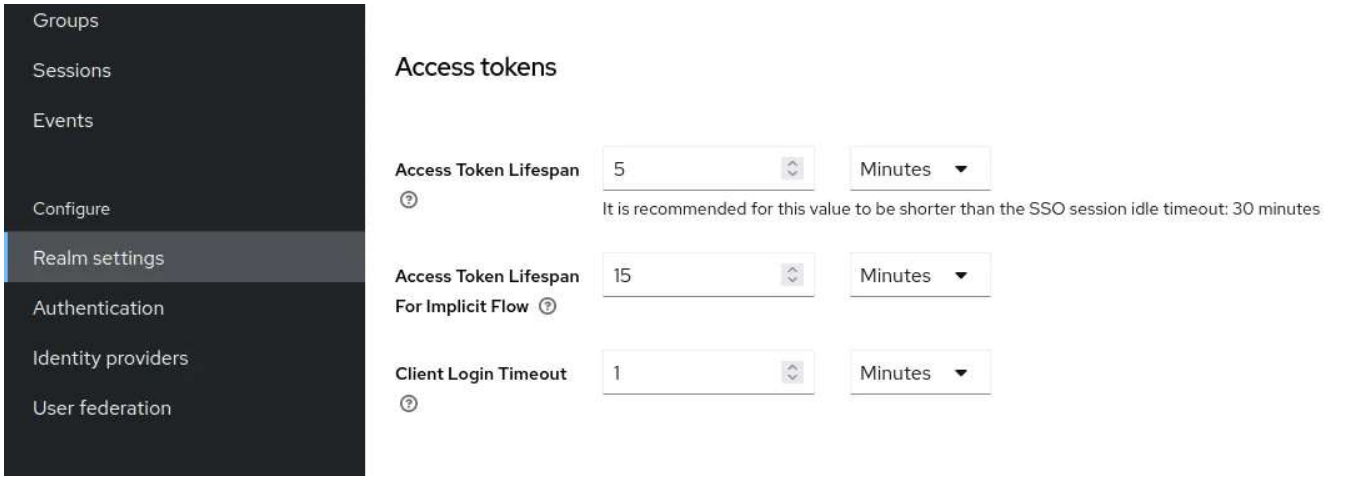


Screenshot of decoded token in https://jwt.io/ (Image by author)

As you can see, there is a specific field called `resource_access` to our client.

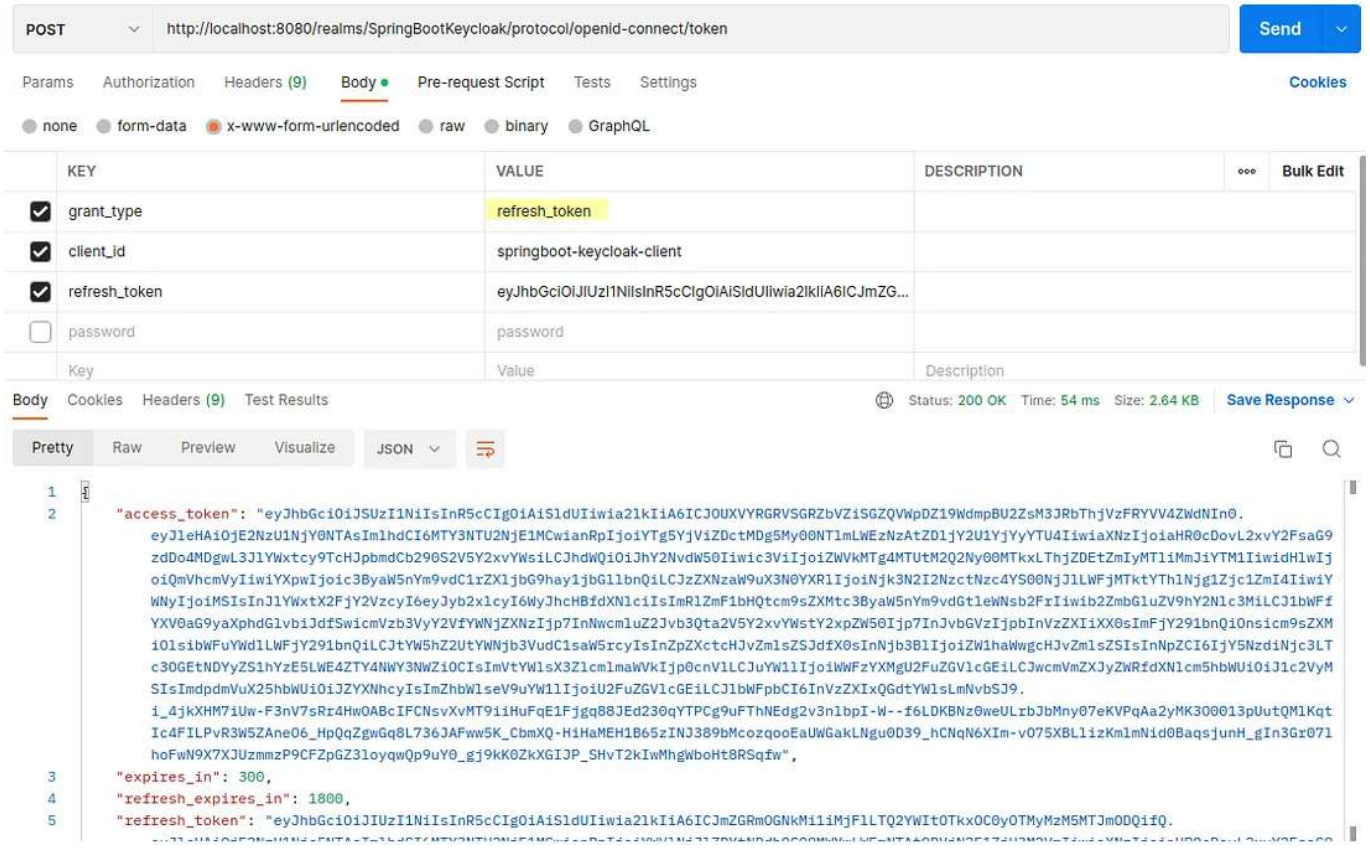
It show's the client and the roles that are assigned to the signed user. The resources can only be accessed based on these roles. Other user-related details can be seen in the latter part of the token.

Furthermore, you can change the access token expiration time under the token tab in Realm settings.



Changing the access token expiration time (Image by author)

If the `access_token` expired, you can send a request to the same endpoint alongside with previous refresh token to get a new `access_token` and a `refresh_token`

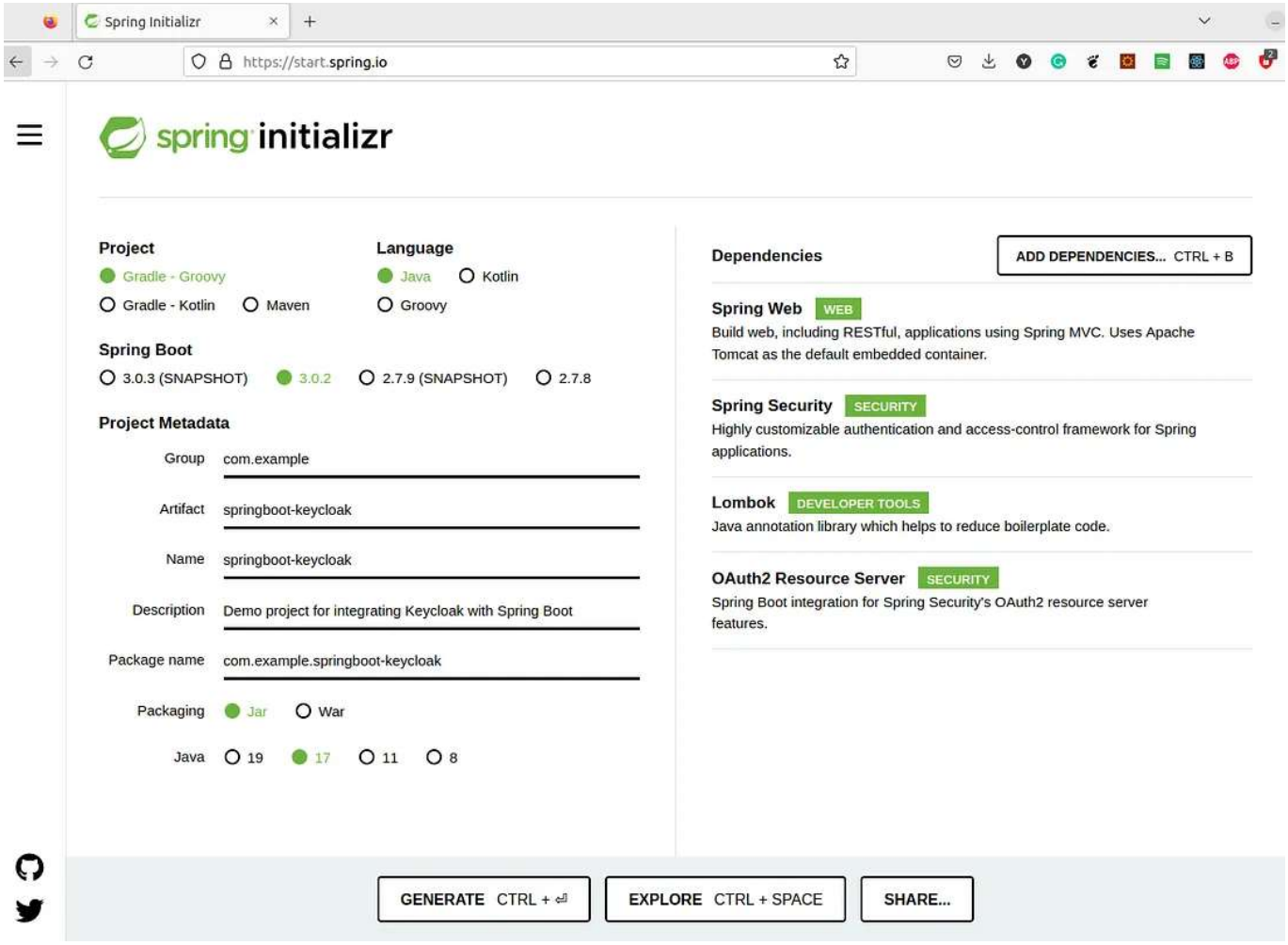


Getting the new access token with the previous refresh token (Image by author)

Creating the Spring Boot 3 Application

Here, I use Spring Initializr: https://start.spring.io/ to generate the initial project structure.

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



Creating a new Spring boot project using Spring Initializer -https://start.spring.io/ (Image by author)

We need Spring web, Spring Security and Lombok dependencies. Additionally, we need one more dependency called OAuth2 Resource Server.

There are two Spring Boot inbuilt libraries to ease the integration of OAuth2.

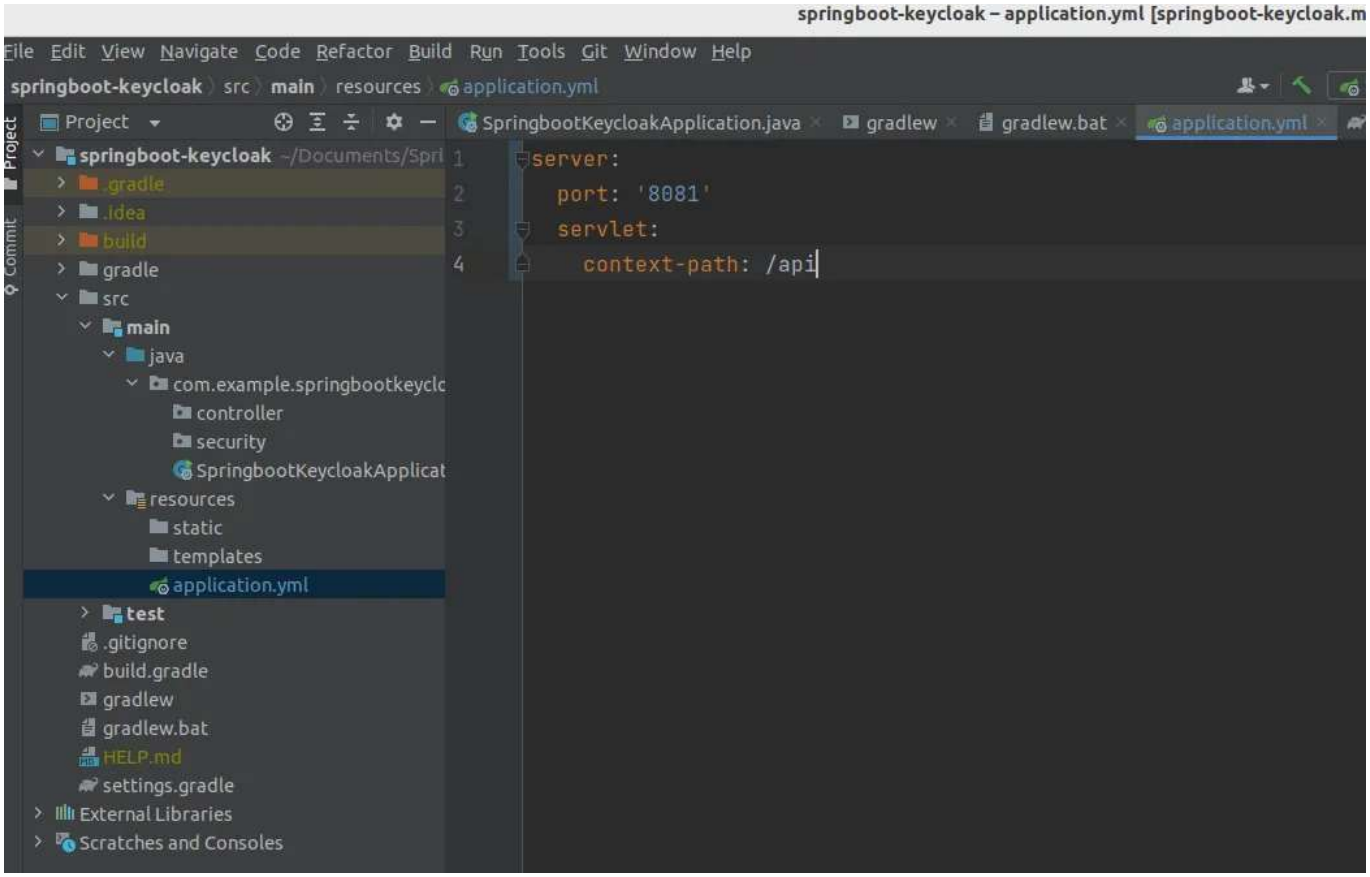
1. `spring-boot-starter-oath2-client` : You can use this if your app serves UI with Thymeleaf or alike template engines.
2. `spring-boot-starter-oath2-resource-server` : Use this if the app is a REST API server.

Here I will be using the OAuth2 Resource Server as I'm going to build a REST API server.

After adding all the dependencies, you can generate the bundle and Open it in your preferred Java development IDE such as IntelliJ IDEA.

Let's first change our server port to 8081 as the Keycloak is already running on port 8080.

I changed the `application.properties` file to `application.yml` file as the YAML version is more concise and more human-readable.



Spring Boot application.yml configurations (Image by author)

I'm creating some dummy REST APIs to test our application. Create `TestController.java` in `controller` package.

```
@RestController
@RequestMapping("/test")
public class TestController {

    @GetMapping("/anonymous")
    public ResponseEntity<String> getAnonymous() {
        return ResponseEntity.ok("Hello Anonymous");
    }

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

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

Alright! Now we can integrate Keycloak into our application.

Keycloak Integration

Let's change the `application.yml` file as follows.

```
spring:
  application:
    name: springboot-keycloak
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: http://localhost:8080/realms/SpringBootKeycloak
          jwk-set-uri: ${spring.security.oauth2.resourceserver.jwt.issuer-uri}/protocol/openid-connec

jwt:
  auth:
    converter:
      resource-id: springboot-keycloak-client
      principal-attribute: preferred_username

logging:
  level:
    org.springframework.security: DEBUG

server:
  port: '8081'
  servlet:
    context-path: /api
```

Here we need to provide resource server configurations.

As you have seen previously, JWT access token includes all the necessary information about the signed user. So the Resource Server needs to verify the signature of the token to ensure the data has not been modified.

The `jwk-set-uri` property contains the public key that the server can use for this purpose.

In JWT auth converter properties, the `resource-id` specifies the identifier for the resource that the token is intended to access. The `principal-attribute` refers to the attribute in the JWT (access token) that the client will use to identify the authenticated user.

Here I used the `preferred_username` field which means that the client will extract the value of this field from the access token and use it as the identifier for the authenticated user.

I have created a class named `JwtAuthConverterProperties` to get those config properties into the Spring Boot application inside the security package.

```
import lombok.Data;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Configuration;
import org.springframework.validation.annotation.Validated;

@Data
@Validated
@Configuration
```

```
@ConfigurationProperties(prefix = "jwt.auth.converter")
public class JwtAuthConverterProperties {

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

Now we want to create custom claims on the Resource Server side. For that, we need to add a class that implements the Converter interface. I created `JwtAuthConverter` class inside the security package.

```
@Component
public class JwtAuthConverter implements Converter<Jwt, AbstractAuthenticationToken> {

    private final JwtGrantedAuthoritiesConverter jwtGrantedAuthoritiesConverter = new JwtGrantedAuthc

    private final JwtAuthConverterProperties properties;

    public JwtAuthConverter(JwtAuthConverterProperties 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, getPrincipalClaimName(jwt));
    }

    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(properties.getResourceId()))
                || (resourceRoles = (Collection<String>) resource.get("roles")) == null) {
            return Set.of();
        }
        return resourceRoles.stream()
                .map(role -> new SimpleGrantedAuthority("ROLE_" + role))
                .collect(Collectors.toSet());
    }
}
```

Here we extract the principle claim and roles of the JWT (access token).

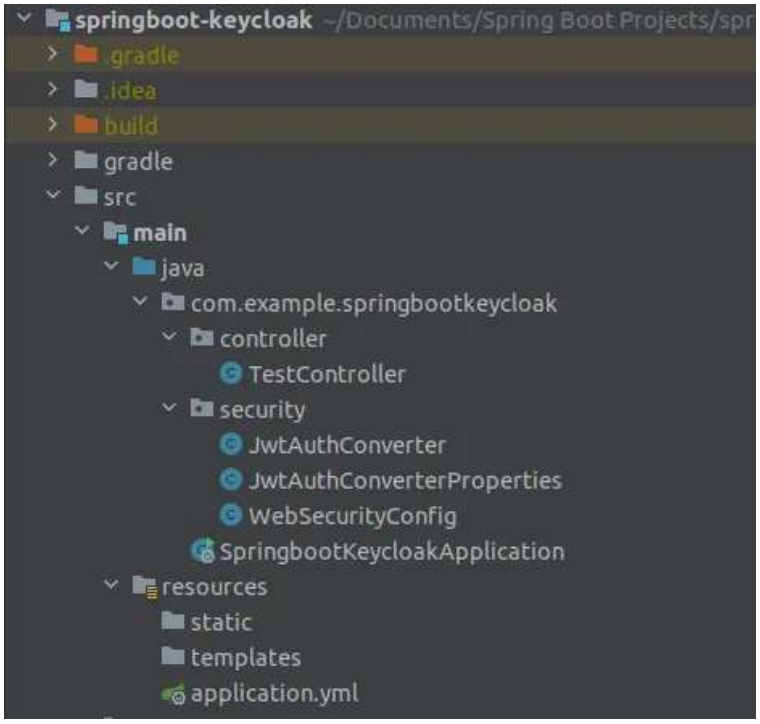
Finally, we need to add a `WebSecurityConfig` class as follows to configure `HttpSecurity` with OAuth2 Resource Server's JWT authentication.

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

    public static final String ADMIN = "admin";
    public static final String USER = "user";
    private final JwtAuthConverter jwtAuthConverter;

    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        http.authorizeHttpRequests()
                .requestMatchers(HttpMethod.GET, "/test/anonymous", "/test/anonymous/**").permitAll()
                .requestMatchers(HttpMethod.GET, "/test/admin", "/test/admin/**").hasRole(ADMIN)
                .requestMatchers(HttpMethod.GET, "/test/user").hasAnyRole(ADMIN, USER)
                .anyRequest().authenticated();
        http.oauth2ResourceServer()
                .jwt()
                .jwtAuthenticationConverter(jwtAuthConverter);
        http.sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);
        return http.build();
    }

}
```

Here we are overriding the default HTTP Security configuration. We need to specify explicitly that we want this to behave as a Resource Server. This is achieved through the use of the `SecurityFilterChain` method.

You can use the `requestMatchers` to protect all the routes. Additionally, you can add CORS configurations here as well.

Now the project structure will look like the below.



Project structure (Image by author)

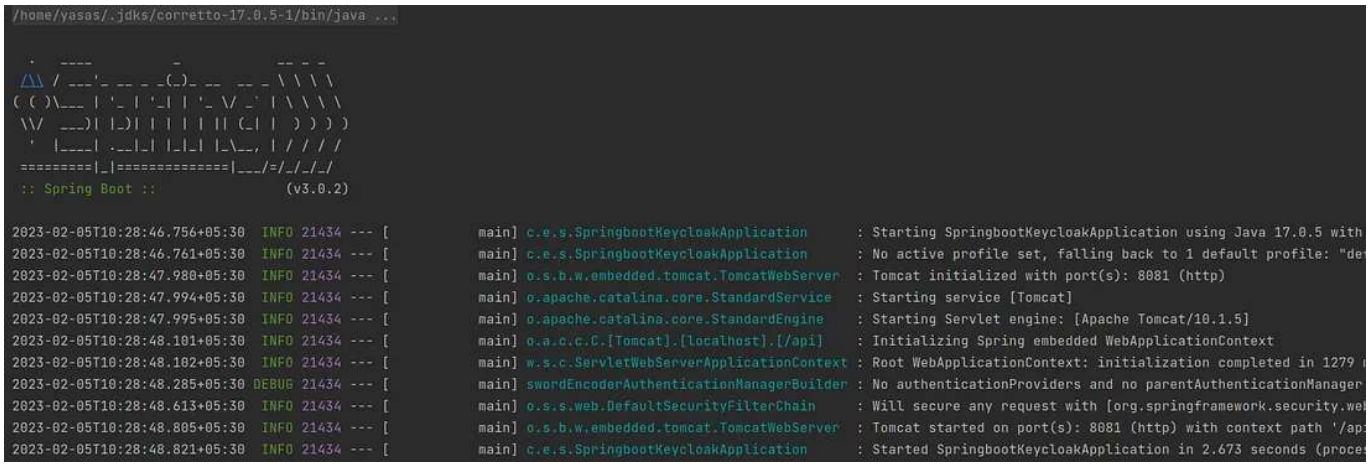
Let's change the test controller routes as well.

```
@RestController
@RequestMapping("/test")
public class TestController {

    @GetMapping("/anonymous")
    public ResponseEntity<String> getAnonymous() {
        return ResponseEntity.ok("Hello Anonymous");
    }

    @GetMapping("/admin")
    public ResponseEntity<String> getAdmin(Principal principal) {
        JwtAuthenticationToken token = (JwtAuthenticationToken) principal;
        String userName = (String) token.getTokenAttributes().get("name");
        String userEmail = (String) token.getTokenAttributes().get("email");
        return ResponseEntity.ok("Hello Admin \nUser Name : " + userName + "\nUser Email : " + userEm
    }

    @GetMapping("/user")
    public ResponseEntity<String> getUser(Principal principal) {
        JwtAuthenticationToken token = (JwtAuthenticationToken) principal;
        String userName = (String) token.getTokenAttributes().get("name");
        String userEmail = (String) token.getTokenAttributes().get("email");
        return ResponseEntity.ok("Hello User \nUser Name : " + userName + "\nUser Email : " + userEma
    }

}
```

We can access the token from `Principal` class. The `principal.getName()` method retrieves the Principal Claim name, which we set as `preffered_username` Additional information we can get from casting it to a `JwtAuthenticationToken`

Testing the Application

All the configs are completed now. Let's run this to see how it works!



Testing the spring boot application (Image by author)

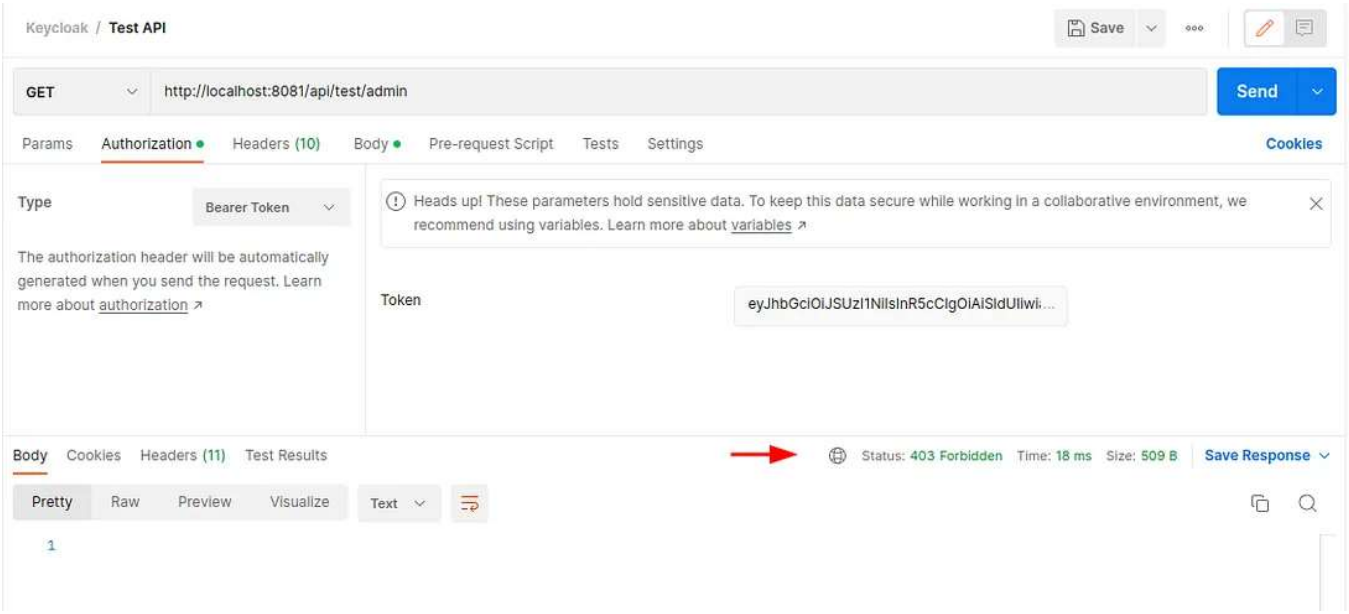
We can test our endpoints through Postman.

Here we need to send a valid access token obtained from the Keycloak. You can get a token as I mentioned in the generating access token section.

```
headers: {
    'Authorization': 'Bearer access_token'
}
```

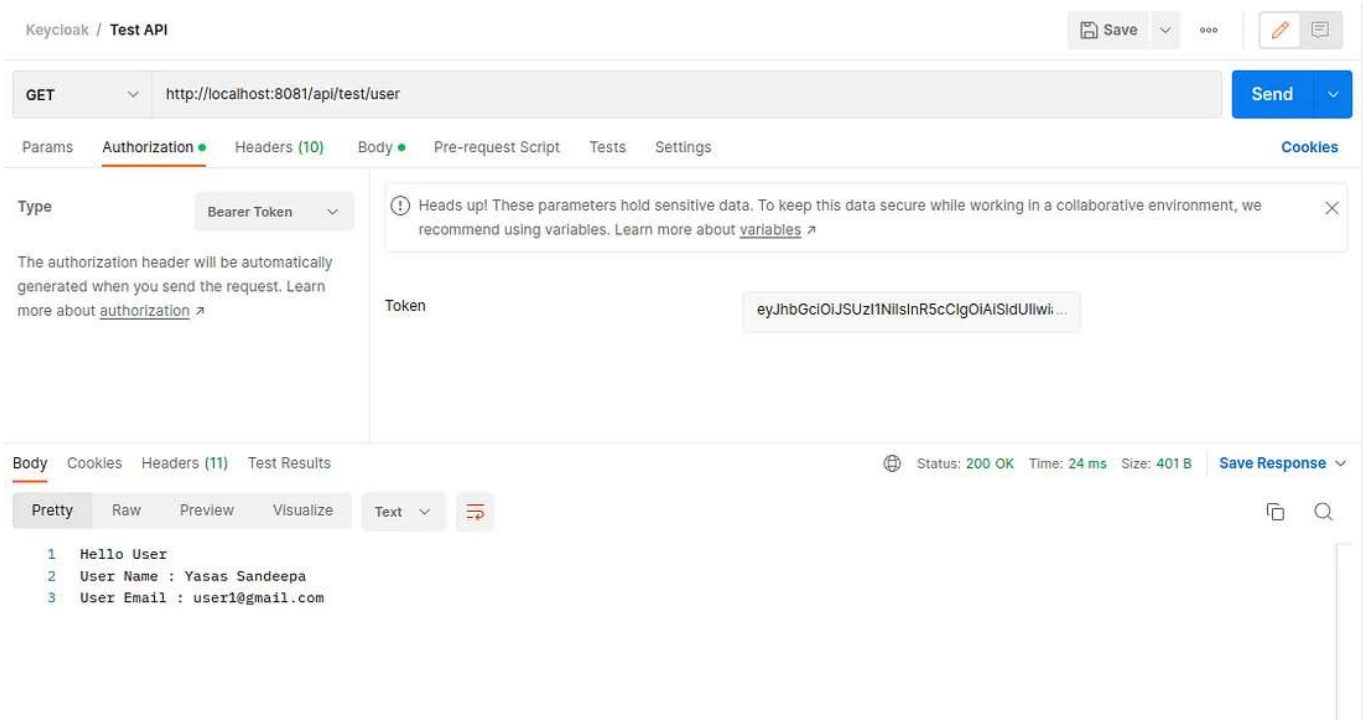
Earlier, we created an endpoint at http://localhost:8081/api/test/admin that is only accessible to users with the `admin` role.

Let's call this endpoint with `user1` credentials. If you remember, `user1` has only the `user` role. So he can't access this endpoint.



Invalid grant (Image by author)

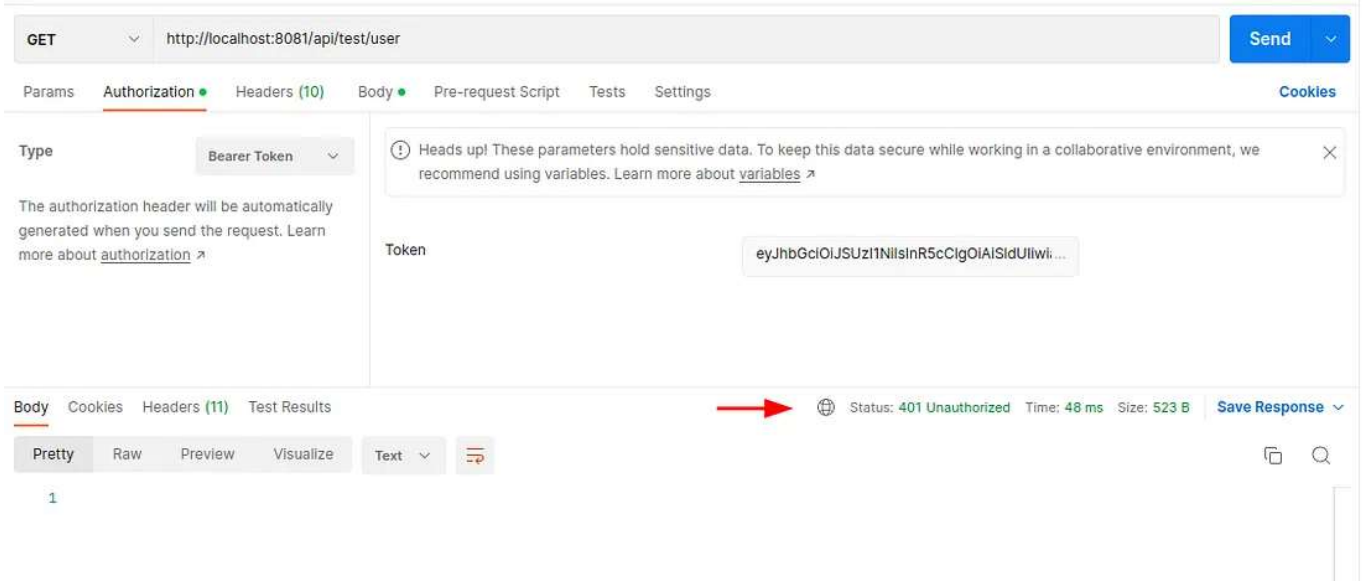
See, he is getting 403 Forbidden. Now let's check the http://localhost:8081/api/test/user endpoint. He should be able to access this as he owned the user role.



A valid grant (Image by author)

Boom! He can access the endpoint.

You can test with other users as well. Also, check the functionality with expired token and the refresh token scenarios.



After access token expiration (Image by author)

That's a wrap! We have successfully connected our Spring Boot 3 application with Keycloak using inbuilt OAuth2 Resource Server configurations.

The source code for this article is available in this repository.

Thank you for reading the article. I hope you have learned something new here.