Reading and understanding the official documentation is essential to installing and using Keycloak in a secure manner, we highly recommend you follow the detailed information there to tune the installation and implementation to your specific use case.

Please use this document as a guide.

Some prerequisites for one to utilize this document effectively is basic understanding of Java, Spring Boot and REST concepts.
The reader should also have a basic understanding of OAuth2.0.
One good reference for OAuth2.0 is https://auth0.com/intro-to-iam/what-is-oauth-2

1. What is Keycloak and How can It help me?

Keycloak is an open-source identity and access management solution that provides authentication, authorization, and single sign-on capabilities for web applications and services. It allows you to secure your applications by managing user identities, enforcing access controls, and facilitating seamless user authentication across multiple systems.

More details about keycloak and its capabilities are on Keycloak's website
https://www.keycloak.org/

Keycloak itself is written in Java and is completely open source. Its code is hosted on Github and is present here on https://github.com/keycloak/keycloak

Javadocs are available for those who are interested on the URL
https://www.keycloak.org/docs-api/21.1.1/javadocs/index.html

Keycloak also exposes a REST based Admin API via which one can manage all activities of Keycloak. Reference: https://www.keycloak.org/docs-api/21.1.1/rest-api/index.html

2. How do I use keycloak to secure my application the Merce way?

One approach to secure **Spring/Spring Boot** applications is what we'll discuss here.

There are a few pieces that we need to understand before we begin with this journey.

Keycloak supports multiple authorization frameworks including OpenID Connect, OAuth 2.0 and SAML 2.0. (Ref: https://www.keycloak.org/)

OAuth2.0 (Ref: https://auth0.com/intro-to-iam/what-is-oauth-2) is an authorization framework that allows applications to access and use resources on behalf of a user without requiring the user to share their credentials (such as username and password) with the application.
It provides a secure and standardized way for users to grant permissions to third-party applications to access their protected resources.

Spring Security is a powerful and highly customizable security framework for Java applications, specifically those built on the Spring framework.
It provides a comprehensive set of features and APIs to handle authentication, authorization, and other security-related tasks in a Java application.
It is the de-facto standard for securing Spring-based applications.
(Ref: https://spring.io/projects/spring-security)

So, we'll now use our Spring boot based code with Spring Security using the OAuth2 Framework and Keycloak Server to secure our application.

Keycloak is a separate server that is managed on our network. Applications are configured to point to and be secured by this server.

Browser applications redirect a user's browser from the application to the Keycloak authentication server where they enter their credentials. This redirection is important because users are completely isolated from applications and applications never see a user's credentials.

Applications instead are given an identity token or assertion that is cryptographically signed. These tokens can have identity information like username, address, email, and other profile data. They can also hold permission data so that applications can make authorization decisions. These tokens can also be used to make secure invocations on REST-based services.

2.1 Keycloak

2.1.1 Keycloak Core Concepts and Terms:

- **Users**
  - Users are entities that are able to log into your system.
  - They can have attributes associated with themselves like email, username, address, phone number, and birthday.
  - They can be assigned group membership and have specific roles assigned to them.
- **Roles**
  - Roles identify a type or category of user.
  - Admin, user, manager, and employee are all typical roles that may exist in an organization.
  - Applications often assign access and permissions to specific roles rather than individual users as dealing with users can be too fine-grained and hard to manage.
- **User role mapping**
  - A user role mapping defines a mapping between a role and a user. A user can be associated with zero or more roles.
  - This role mapping information can be encapsulated into tokens and assertions so that applications can decide access permissions on various resources they manage.
- **Realms**
  - A realm manages a set of users, credentials, roles.
  - A user belongs to and logs into a realm.
  - Realms are isolated from one another and can only manage and authenticate the users that they control.
- **Clients**
  - Clients are entities that can request Keycloak to authenticate a user.
  - Most often, clients are applications and services that want to use Keycloak to secure themselves and provide a single sign-on solution.

- Clients can also be entities that just want to request identity information or an access token so that they can securely invoke other services on the network that are secured by Keycloak.

There are more concepts which are good to know.
These can be found in the official documentation of Keycloak on
https://www.keycloak.org/docs/latest/server_admin/#core-concepts-and-terms

2.1.2 Downloading Keycloak

Keycloak works on almost all Linux based distribution and windows.
For our case, since most of us developers are on Ubuntu, we'll proceed with Basic JDK based setup.
The minimum system and software requirements are updated on the Keycloak website. Please refer to it before proceeding with installation.
https://www.keycloak.org/getting-started/getting-started-zip

Basic steps are:

1. Download the keycloak zip file.
2. Extract the zip file to some folder. (e.g. unzip keycloak-21.1.1.zip). Note: at the time of writing this doc, the latest version was 21.1.1, so unzip file keycloak zip file accordingly.
3. Start Keycloak. (bin/kc.sh start-dev)

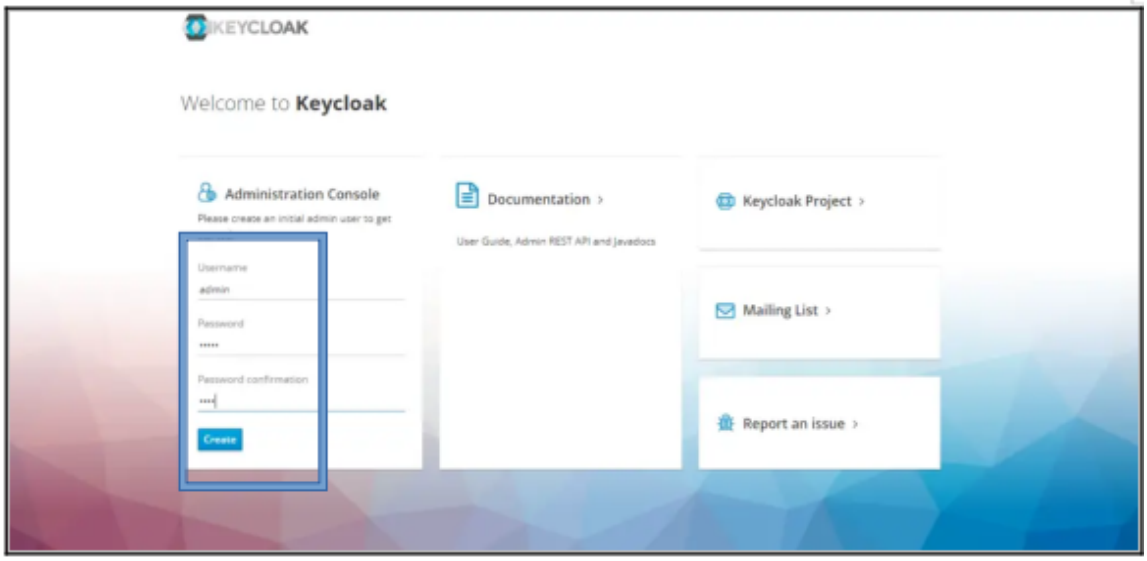
Note that Keycloak by default starts on port **8080**. Ensure it's available.

Note: There are container images also available if you are comfortable with containers.
Docker: https://www.keycloak.org/getting-started/getting-started-docker
Kubernetes: https://www.keycloak.org/getting-started/getting-started-kube

2.1.3 Setting up Keycloak

2.1.2.1 Creating Administrator user

1) Open : http://localhost:8080/



2) Fill in the form with your preferred username and password.
3) Now go to the default admin console http://localhost:8080/admin
   And Login with username and password created earlier.

2.1.2.2 Creating a Realm.

A realm is a space where you manage objects, including users, applications, roles, and groups. A user belongs to and logs into a realm. One Keycloak deployment can define, store, and manage as many realms as there is space for in the database.

Realms are isolated from one another and can only manage and authenticate the users that they control. Following this security model helps prevent accidental changes and follows the tradition of permitting user accounts access to only those privileges and powers necessary for the successful completion of their current task.

You create a realm to provide a management space where you can create users and give them permissions to use applications. At first login, you are typically in the master realm, the top-level realm from which you create other realms.

You can also consider Realm to be a *'Tenant'*

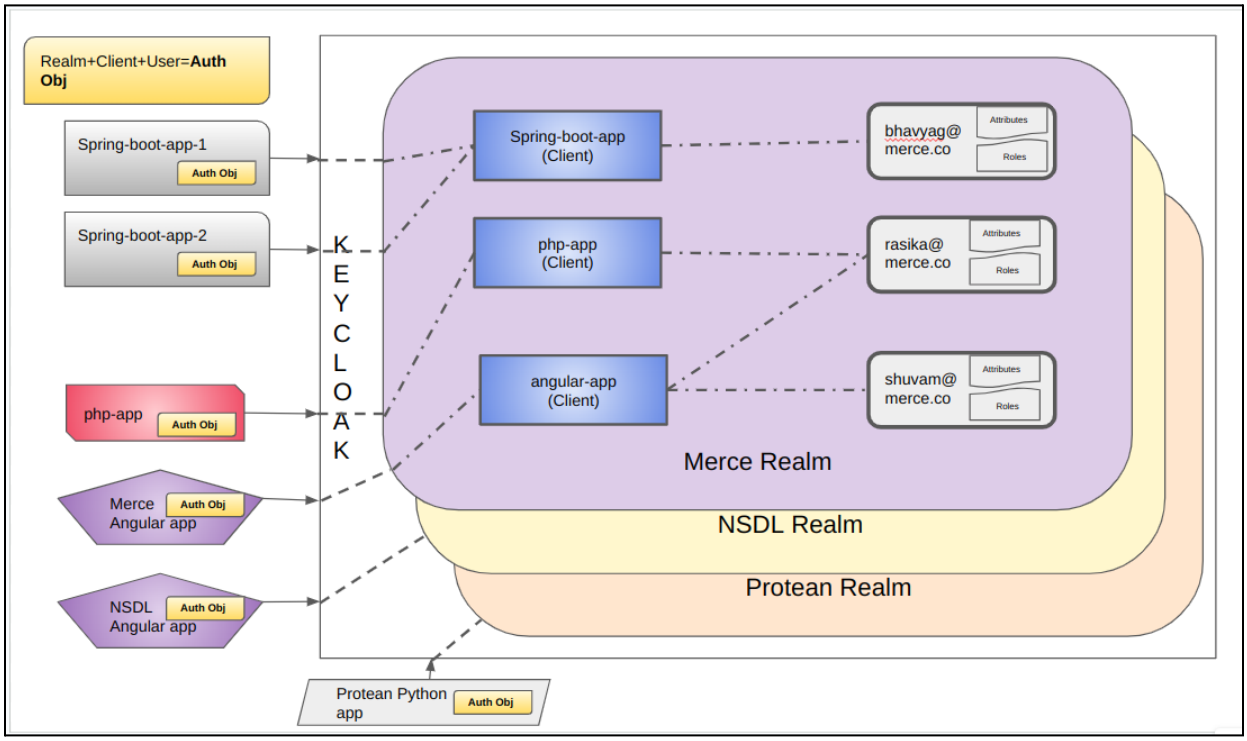
When deciding what realms you need, *consider the kind of* ***isolation*** you want to have for your users and applications.
For example, you might create a realm for the employees of your company and a separate realm for your customers. Your employees would log into the employee realm and only be able to visit internal company applications. Customers would log into the customer realm and only be able to interact with customer-facing apps.

Another way we can think of creating a realm is by Business Entity for example:

MerceRealm : for Merce internal applications.
NSDLRealm : for NSDL applications.
CDSLRealm : for CDSL applications.
ProteanRealm : for Protean applications.

Key idea is to decide the *isolation* needed for the use case.

Example of how this can look:



Out of the box, Keycloak includes a single realm, called '*Master*' realm.
`Master realm` - This (default) realm is created during the first Keycloak installation. It contains the administrator account you created at the first login. By convention, we'll use the master realm only to create and manage the realms in our system.
`Other realms` - These realms are created by the administrator in the master realm. In these realms, administrators manage the users in your organization and the applications they need. The applications are owned by the users.

Use the following steps to create the first realm.

1) Open the Keycloak Admin Console.
2) Click the word master in the top-left corner, then click Create realm.
3) Enter myrealm in the Realm name field.
4) Click Create.

2.1.2.3 Creating a Keycloak Client.

A Keycloak client refers to an application or service that interacts with the Keycloak server to obtain authentication and authorization services.
It represents a registered entity that wants to utilize Keycloak's features, such as user authentication, access control, and single sign-on.

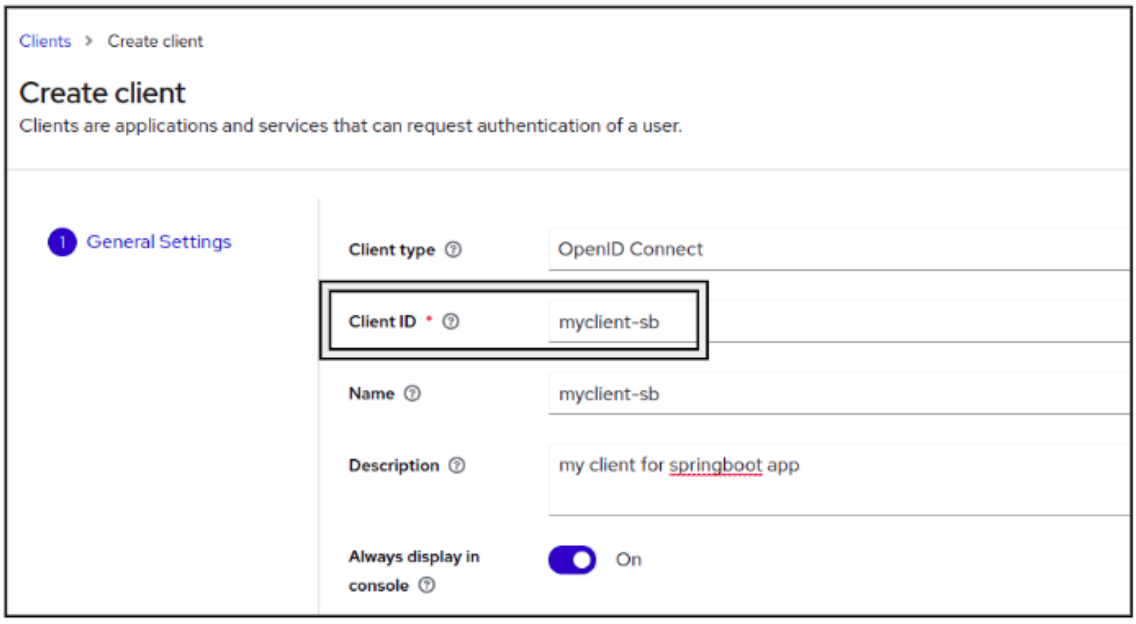
So, we can have a keycloak client for our Spring boot application, another client for say our PHP application and another client for say our Front end application.

Again, like Realm, there is no defined way to use a Client in Keycloak. Since it is like a framework, you can decide to use Client in ways you may think is more feasible.

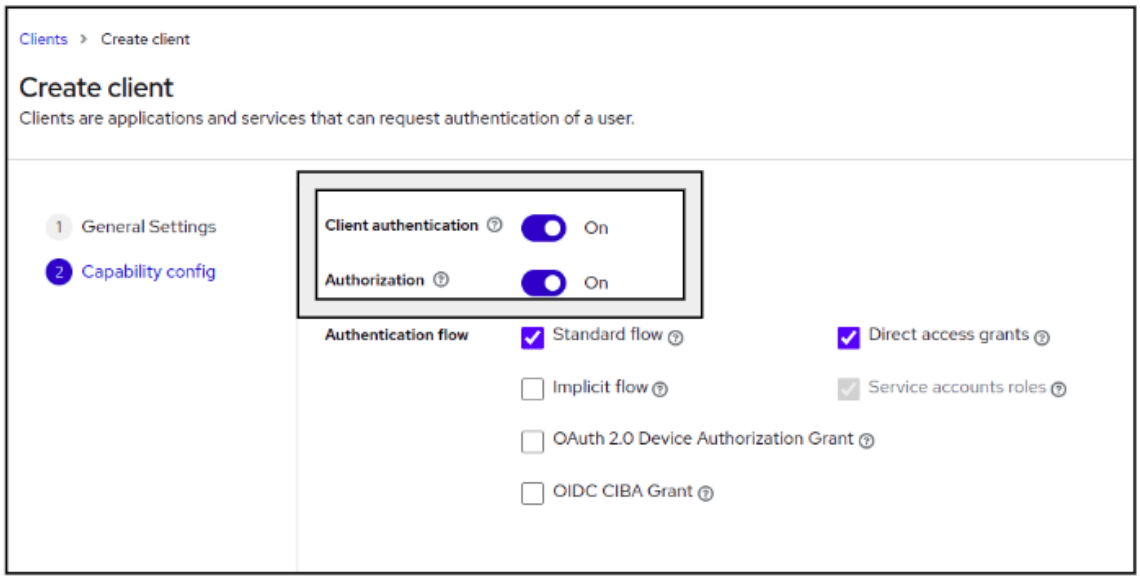
One way to use a Client is to pair it with the type of application since there could be different requirements of a Front-end application as compared to a backend application.

Steps to create Keycloak Client are as follows:

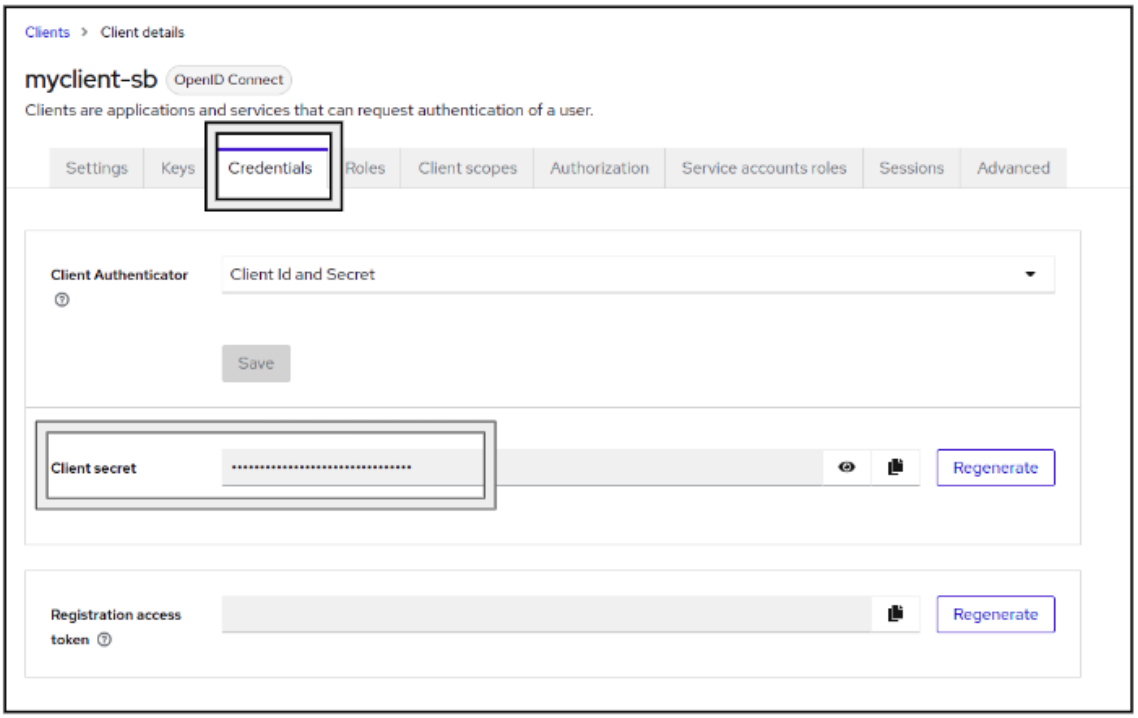
1) Open the Keycloak Admin Console.
2) Click on master on top right corner and select Realm name, in our case it is `myrealm`
3) Click on 'Clients' in the menu bar on right
4) Click on 'Create Client' button
5) Fill in the details as following



6) Click on Next
7) Enable Client Authentication, Authorization as following screenshot:



8) Click on Save.
9) Now, if you go to the 'Credentials' tab you will see the client secret as follows:

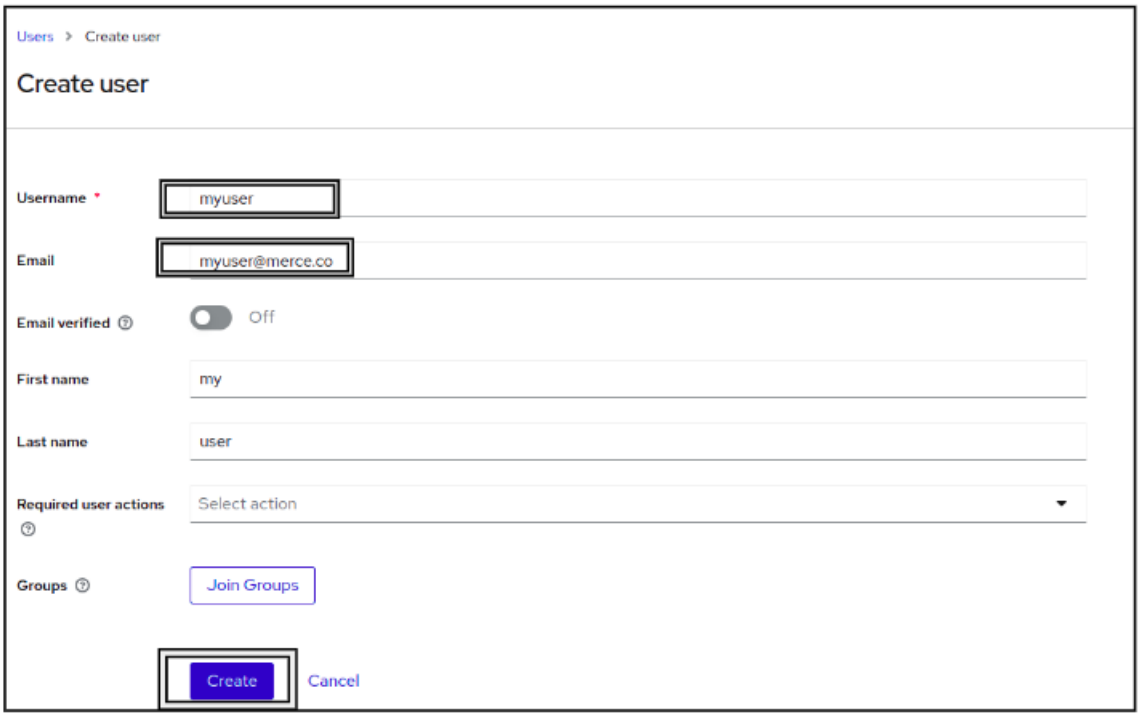


We'll use this **'Client secret'** while connecting to Keycloak.
You have now created a Keycloak Client for the spring boot app.

2.1.2.4 Creating a Keycloak User

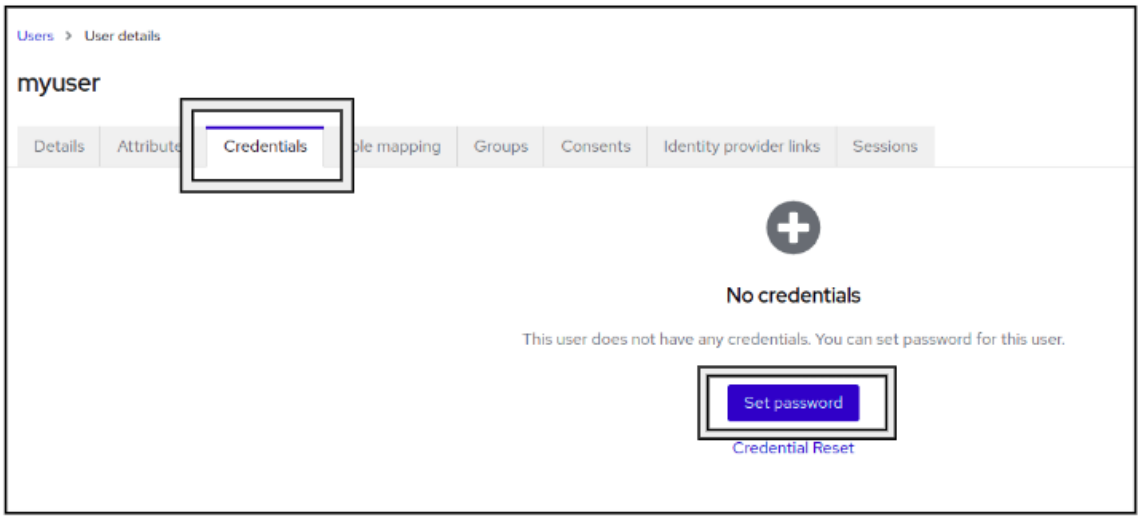
A keycloak user is the user who uses your application. Any user that will use your application, will have to be created in Keycloak. Keycloak will manage the user lifecycle.
Following are the steps to create a user in Keycloak:

1) Open the Keycloak Admin Console.
2) Click on master on top right corner and select myrealm
3) Click on ‘Users’ in the menu bar on the right
4) Click on the ‘Add user’ button.
5) Fill in details as following:

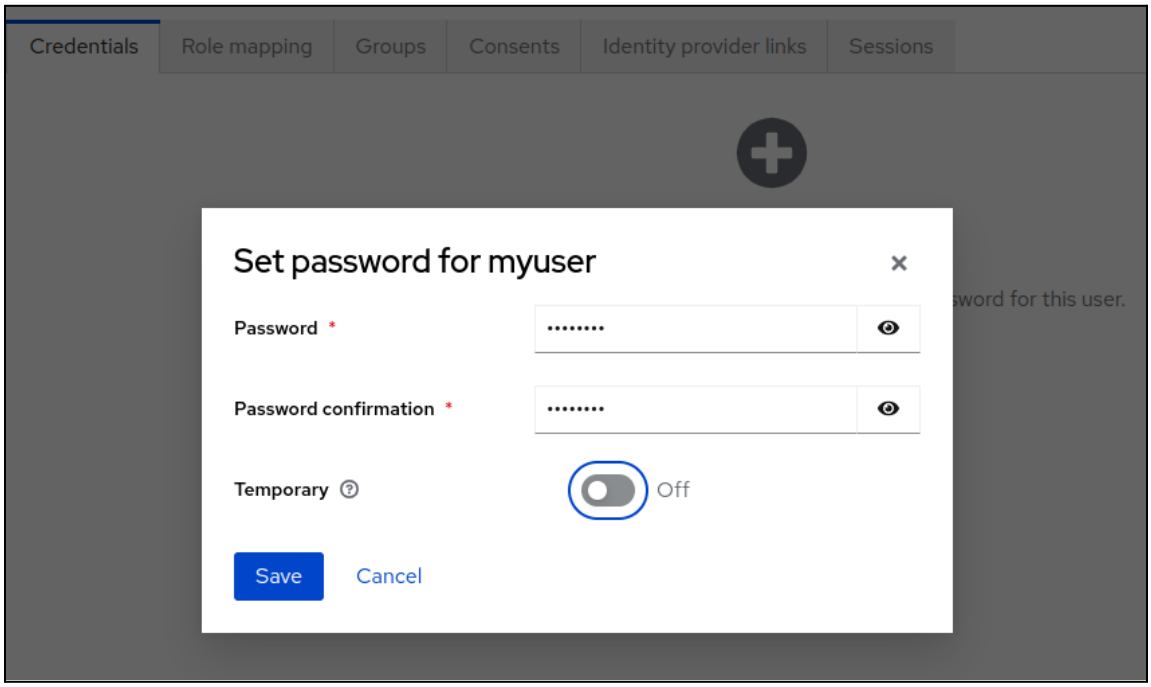


And Click on ‘Create’.

6) Now go to Credentials tab and click on ‘Set password’

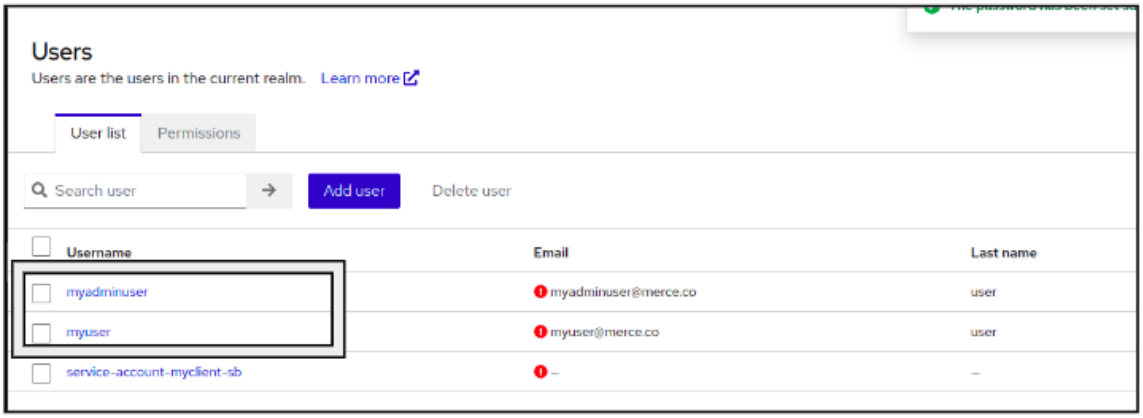


7) Create a password for this user and click on Save



You have now created a user called ‘myuser’ in Keycloak.

8) We’ll also create another user ‘myAdminUser’ using the same step as above.
9) So now, we will see two users:



2.1.2.5 Setting Roles in Keycloak

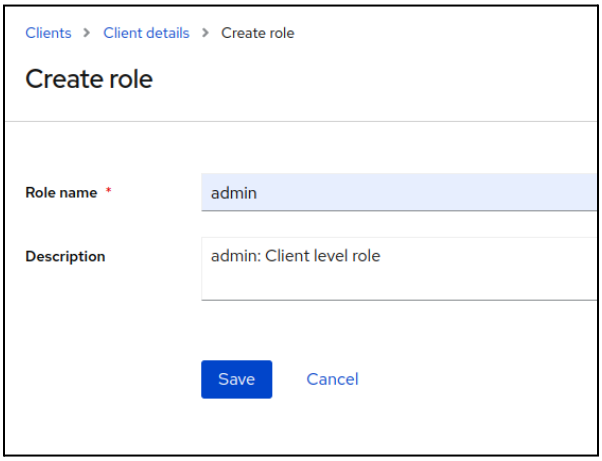
For the authorization part in our spring boot application, we’ll need to create different roles.

A role refers to a predefined set of permissions or access rights that can be assigned to users or clients. Roles are used to control and enforce authorization policies within the OAuth2 framework.
By assigning roles to users or clients, you can determine what actions they are allowed to perform and what resources they can access.

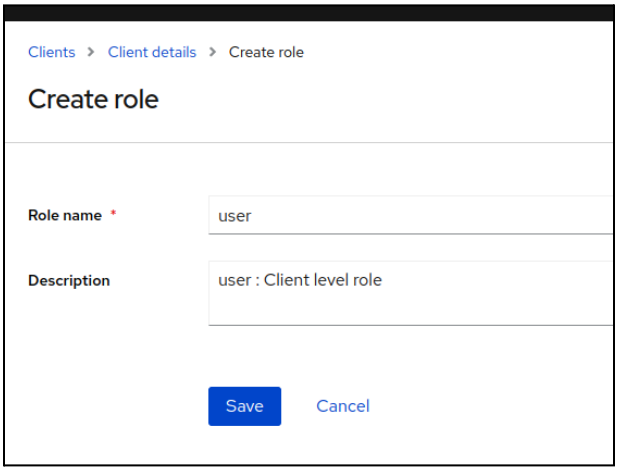
There are roles to be created at two levels, Keycloak Client level and Keycloak Realm level. We'll create roles at "myclient-sb" client which we created above and another role at our "myrealm" realm level.

Then we'll convert our realm level role to a composite role so that whenever we create a user, we'll just need to add one realm level role.

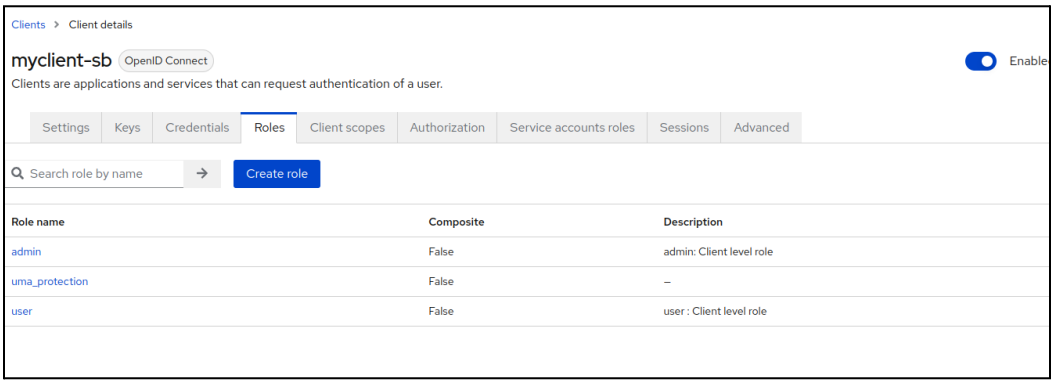
1. Create Client level role
    a. Open the Keycloak Admin Console.
    b. Click on master on top right corner and select myrealm
    c. Click on 'Clients' in the menu bar on right
    d. Now click on "myclient-sb" Client we previously created.
    e. Click on the 'Roles' tab.
    f. Click on the 'Create role' button.
    g. Add role name as 'admin'



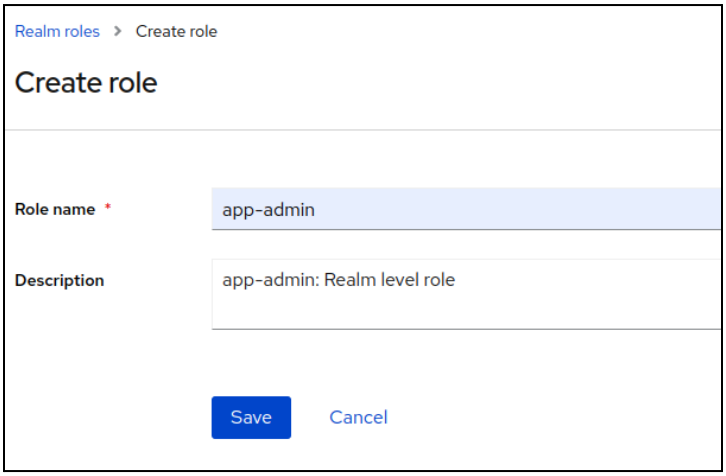
    h. Click on Save.
    i. Now create another role 'user' using the same step as above.



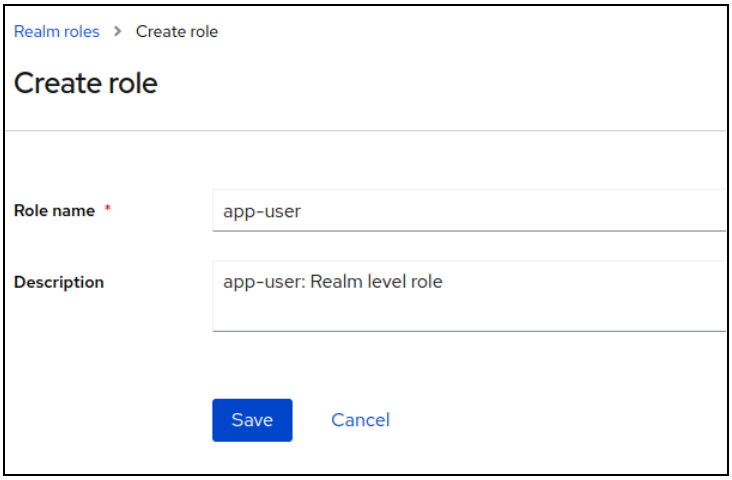
    j. Now under the client -> Roles tab we can see 2 custom roles we created as follows:



2. Create Realm level role
    a. Open the Keycloak Admin Console.
    b. Click on master on top right corner and select myrealm
    c. Click on ‘Realm roles’ in the menu bar on right
    d. Click on ‘Create role’ button
    e. Create a role ‘app-admin’ as follows:



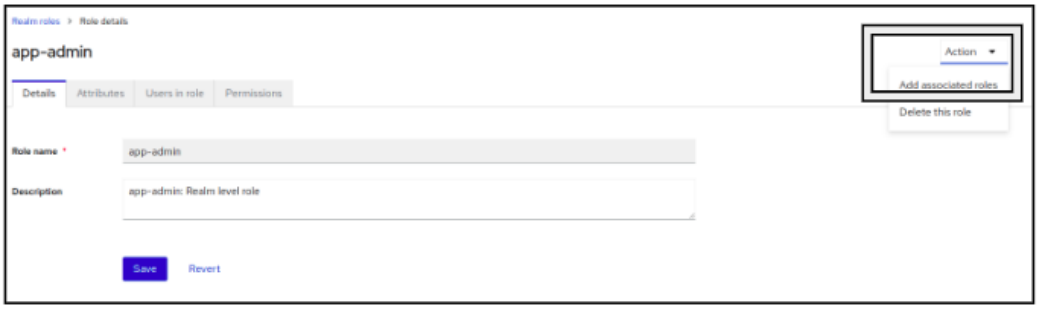
    f. Click on save.
    g. Create a role ‘app-user’ as follows:



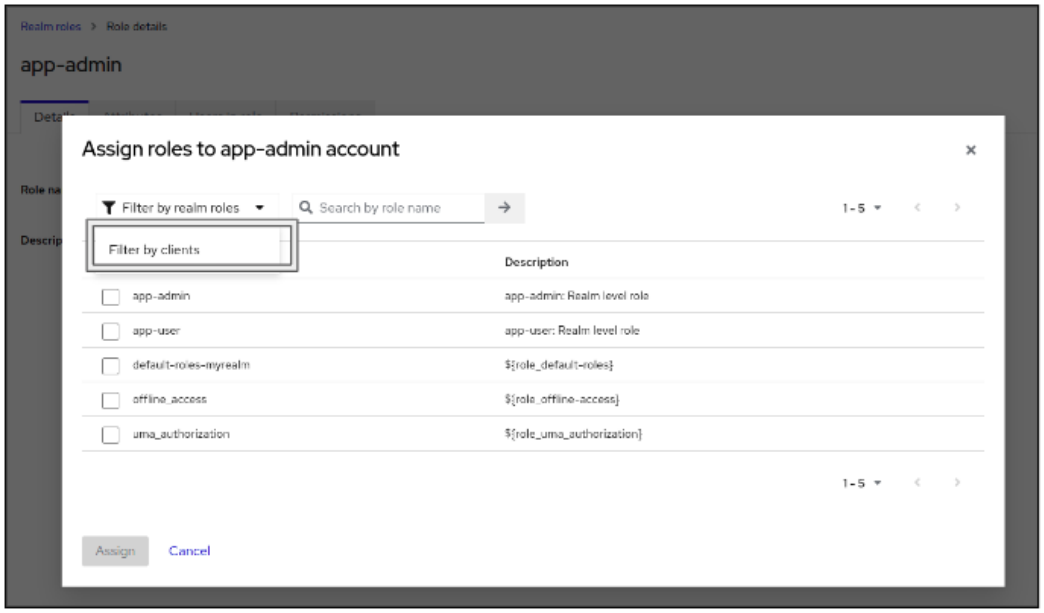
h. Now, under Realm Roles, we can see 2 custom roles we created above as follows:

Note in above screenshot, we can see that under the “Composite” column, roles are termed as ‘False’, which means they are not a composite role at this point.

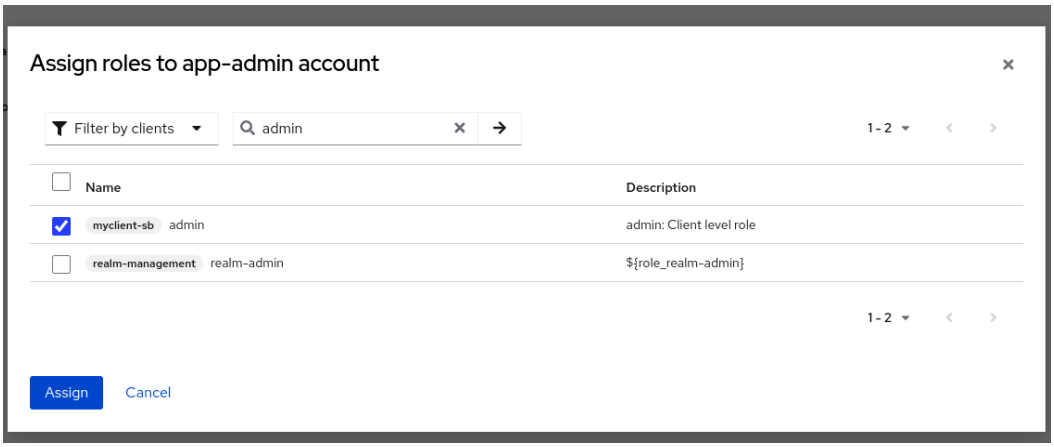
3. Convert Realm level role to a Composite role
   a. To convert Realm role to Composite role, we’ll select a role ‘app-admin’
   b. Click on Action on top right corner and click on ‘Add associated roles’ as follows:



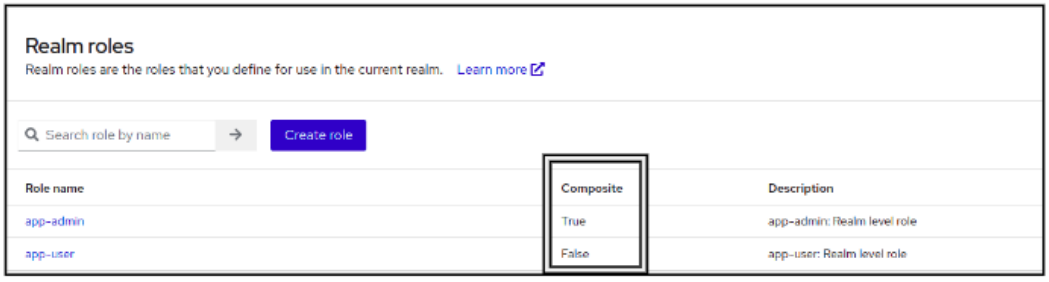
   c. Here, from the drop down, select ‘Filter by clients’ as follows:



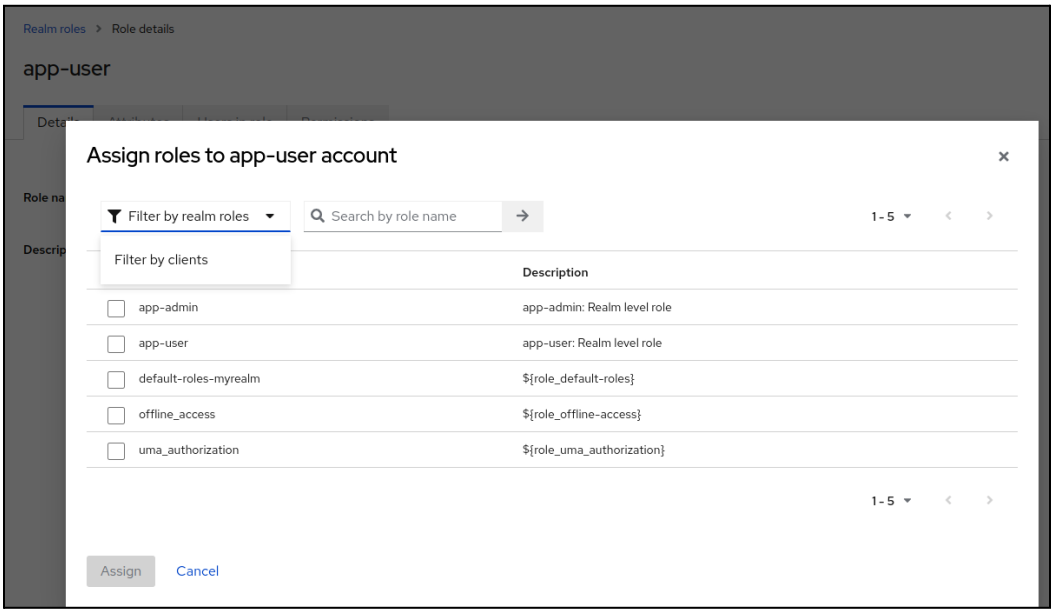
d. And select ‘admin’ client level role we previously created as follows and click on ‘Assign’ button:



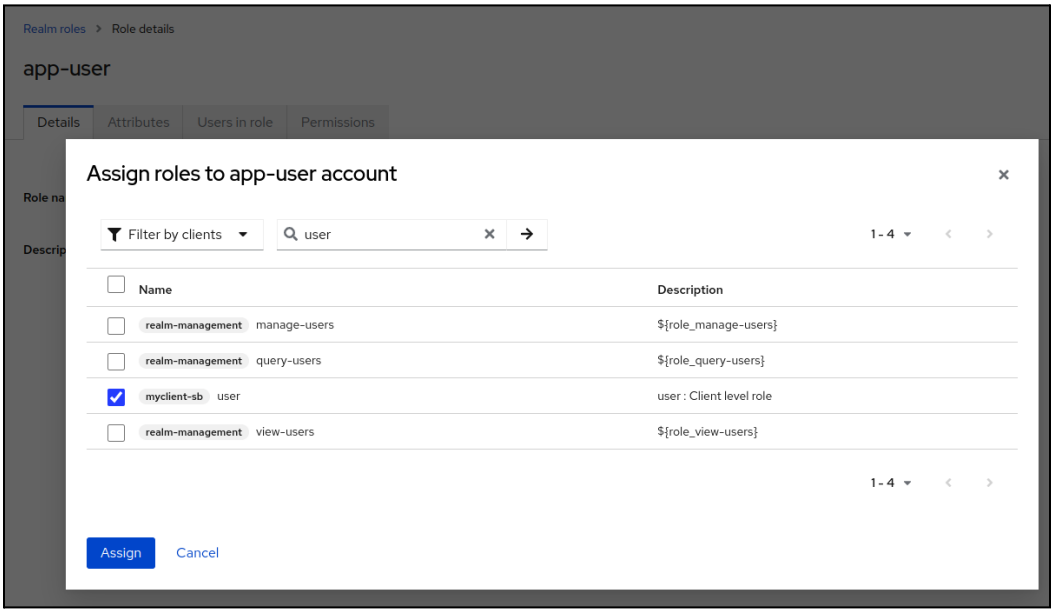
e. You can now see the role is now a Composite role, whereby the ‘Composite’ column is visible as ‘True’.



f. We’ll repeat the steps for ‘app-user role to convert it to a composite role
g. Click on the role ‘app-user’, Click on ‘Action’ on top right corner and select ‘Add associated roles’
h. Select ‘Filter by clients’ in the drop down

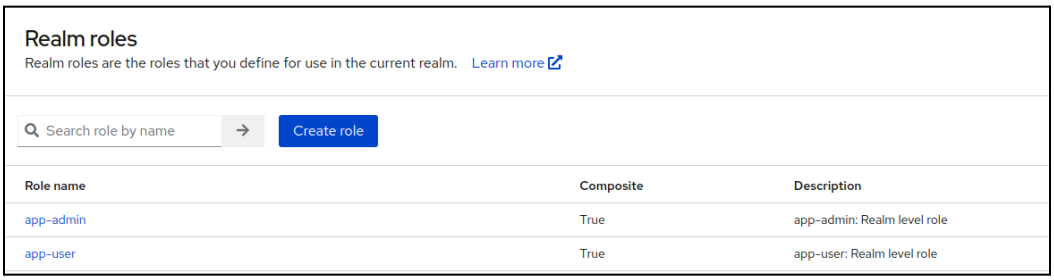


i. Select client level role ‘user’



j. Click on Assign to save the role

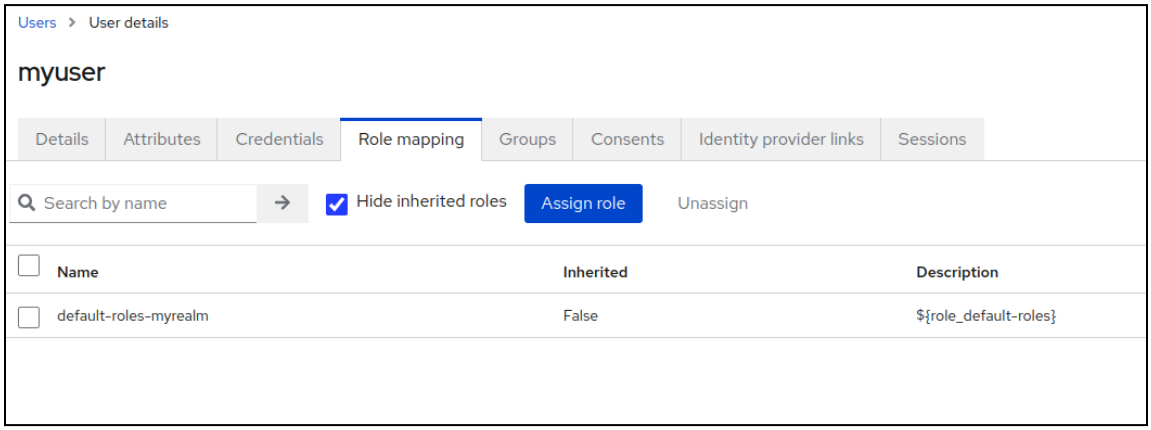
k. Now we can see both roles are composite role as follows:



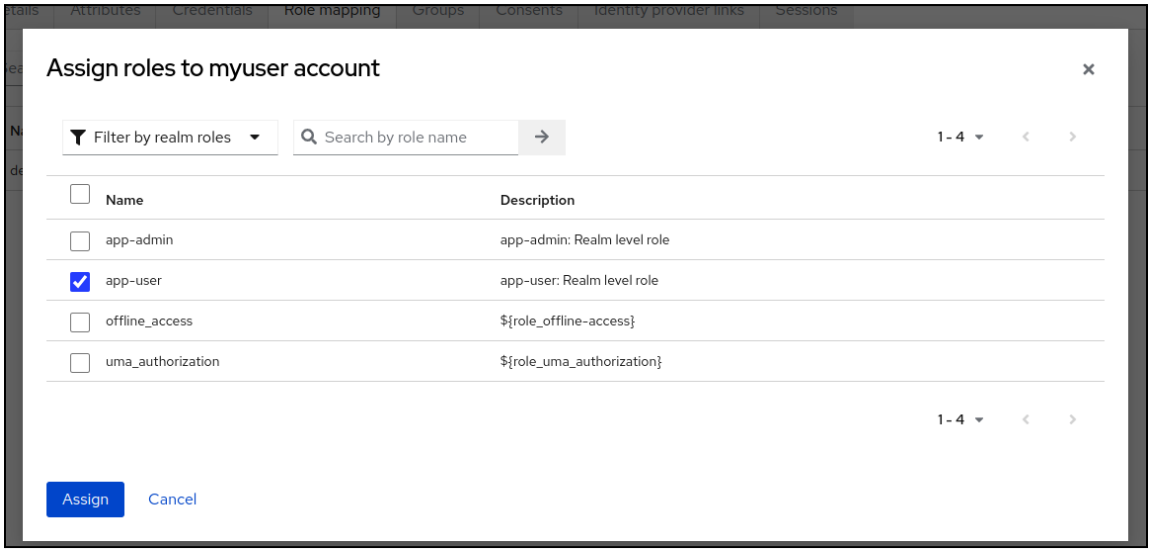
2.1.2.6 Adding Role to the User

We'll now add the composite role we created to the user so that role will be a part of the user's authorization parameters i.e. it'll be a part of users Access Tokens.

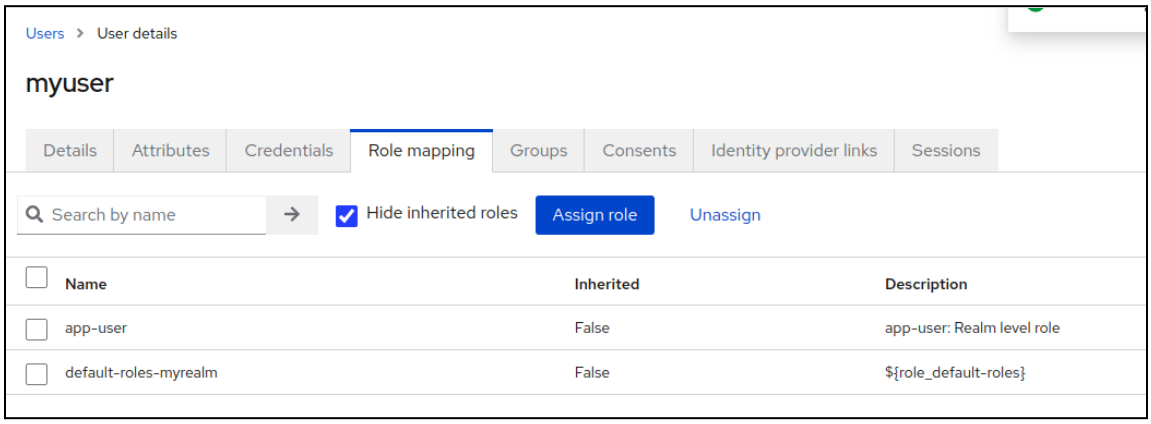
1) Open the Keycloak Admin Console.
2) Click on master on top right corner and select myrealm
3) Click on 'Users' in the menu bar on the right
4) Select 'myuser' the user we previously created
5) Click on 'Role mapping' tab
6) Click on 'Assign role' button as follows



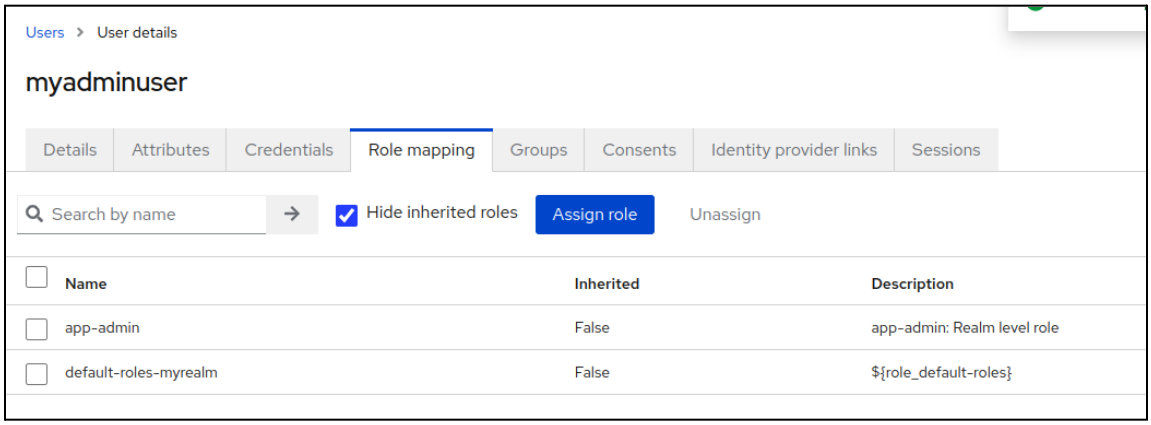
7) Select 'app-admin' composite role we created in the previous step as follows:



8) Click on 'Assign' button to assign the role
9) Now you can see the role is assigned to the user 'myuser' under the 'Role mapping' tab



10) Now similarly using the same steps as above, we'll add 'app-admin' role to user 'myadminuser'

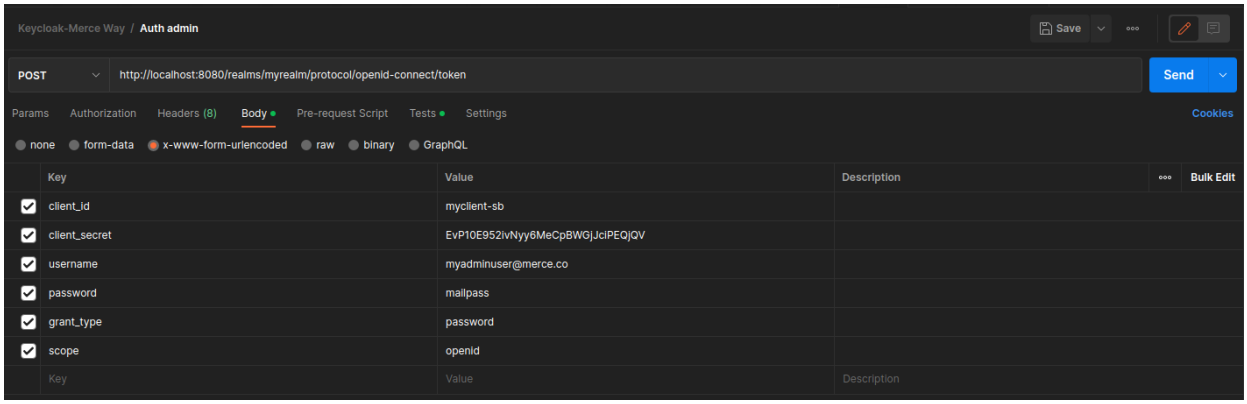


We have now successfully configured Keycloak Users with associated roles.

2.2 Testing the Keycloak setup using Postman

I am assuming at this point that you have the 'Postman' app installed on your local machine. If not, please google the step to install Postman based on your device.

We'll connect to Keycloak to fetch 'Access token'.
I am attaching the Postman collection here for the reference. However, we'll create a new connection as follows:

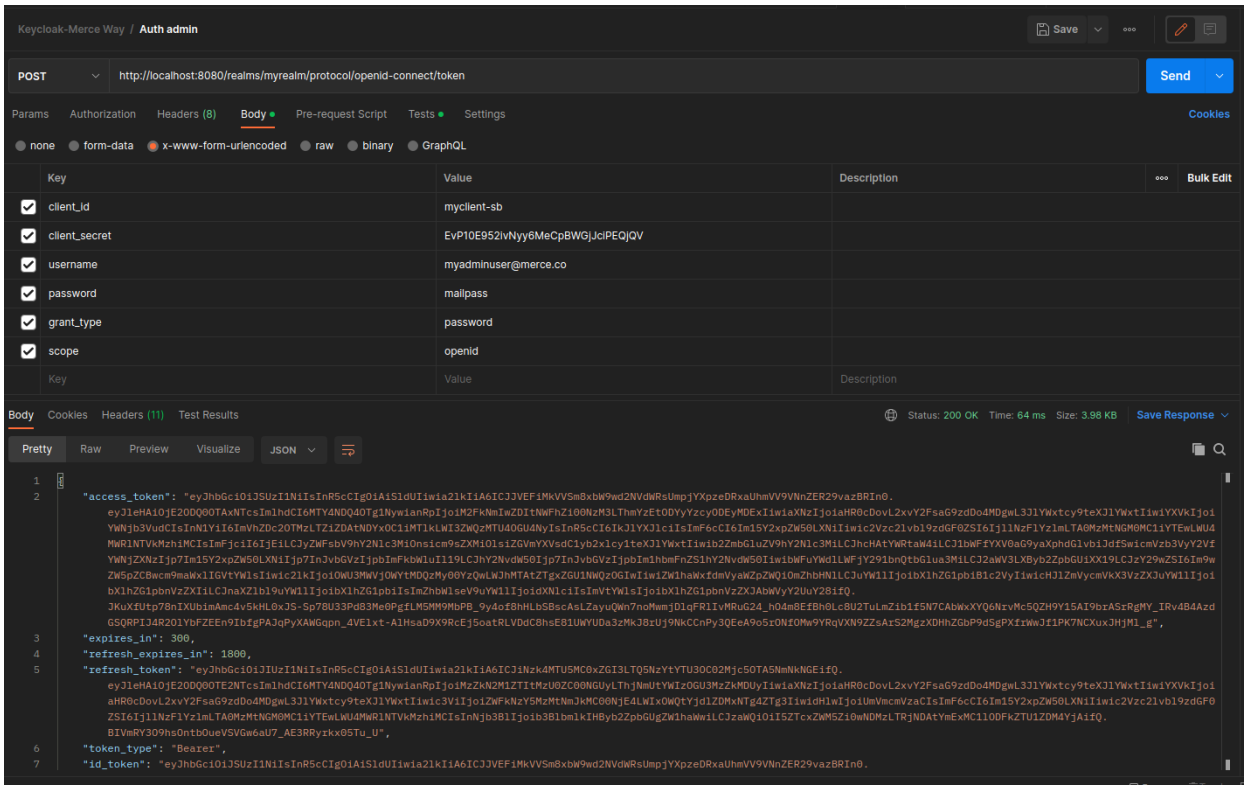


Note the URL: http://localhost:8080/realms/myrealm/protocol/openid-connect/token
Here: localhost : It is the host where Keycloak is running
8080: Port on which Keycloak is listening
'Myrealm' is the realm we created above.

Following are the parameters which we add in the request body:

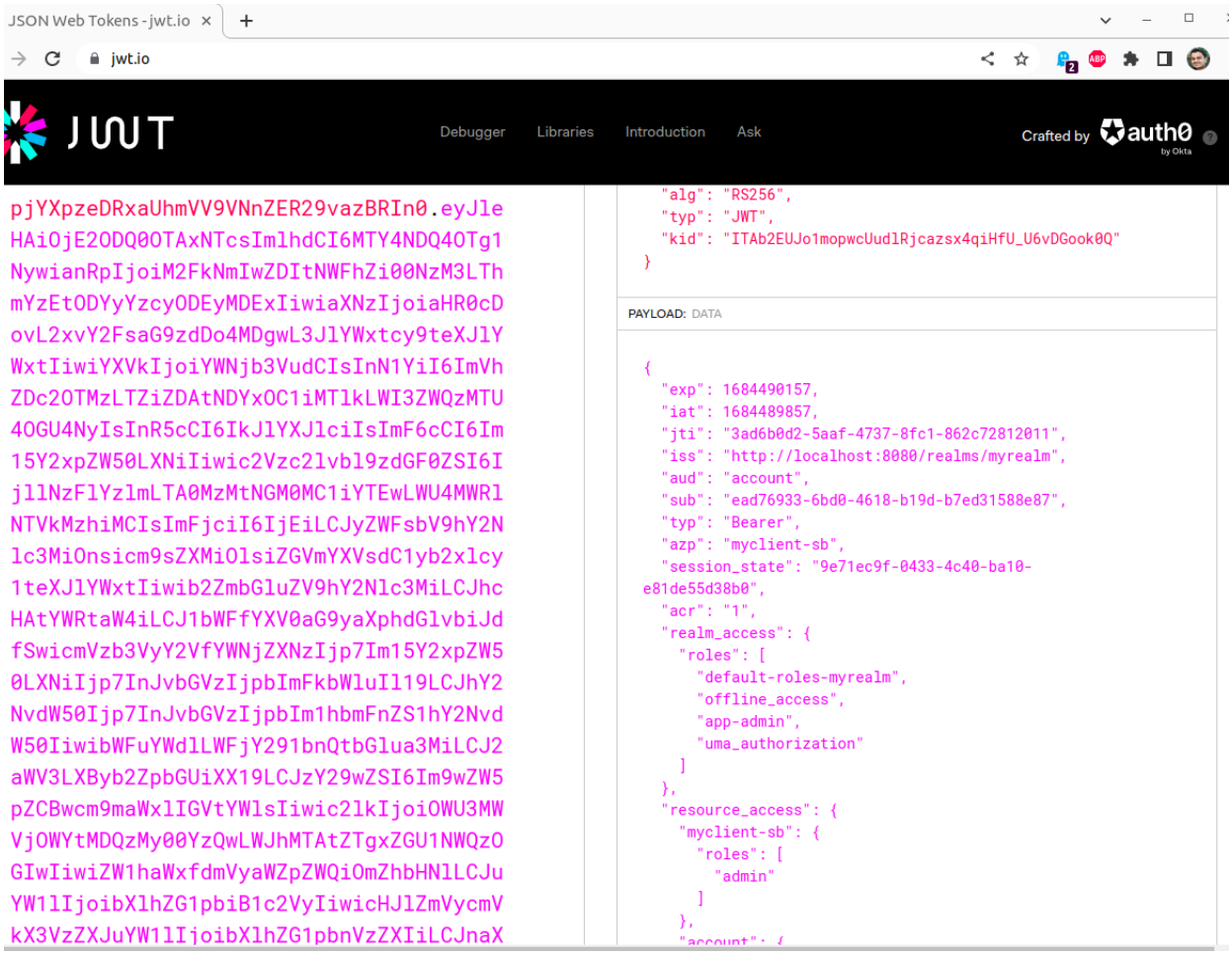
- client_id: ID of the keycloak client we created above
- client_secret : client secret generated by Keycloak [as seen in step 9 of 2.1.2.3]
- username : username of the userid trying to login to keycloak
- password : password of the userid
- grant_type : this can be 'client_credentials' or 'password'. We'll use 'password'
- scope : 'openid'

If all configuration is correct, on sending this request, keycloak will respond with 'access_token' and 'refresh_token' alongwith expiry and other parameters.



We'll use this 'access_token' for all consecutive requests to authenticate yourself.

Pro-tip: You can check contents of this JWT token, using a site like 'http://jwt.io' Following is the example:




```
  "alg": "RS256",
  "typ": "JWT",
  "kid": "ITAb2EUJo1mopwcUudlRjcazsx4qiHfU_U6vDGook0Q"
}
```

PAYLOAD: DATA

```
{
  "exp": 1684490157,
  "iat": 1684489857,
  "jti": "3ad6b0d2-5aaf-4737-8fc1-862c72812011",
  "iss": "http://localhost:8080/realms/myrealm",
  "aud": "account",
  "sub": "ead76933-6bd0-4618-b19d-b7ed31588e87",
  "typ": "Bearer",
  "azp": "myclient-sb",
  "session_state": "9e71ec9f-0433-4c40-ba10-
e81de55d38b0",
  "acr": "1",
  "realm_access": {
    "roles": [
      "default-roles-myrealm",
      "offline_access",
      "app-admin",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "myclient-sb": {
      "roles": [
        "admin"
      ]
    },
    "account": {
```

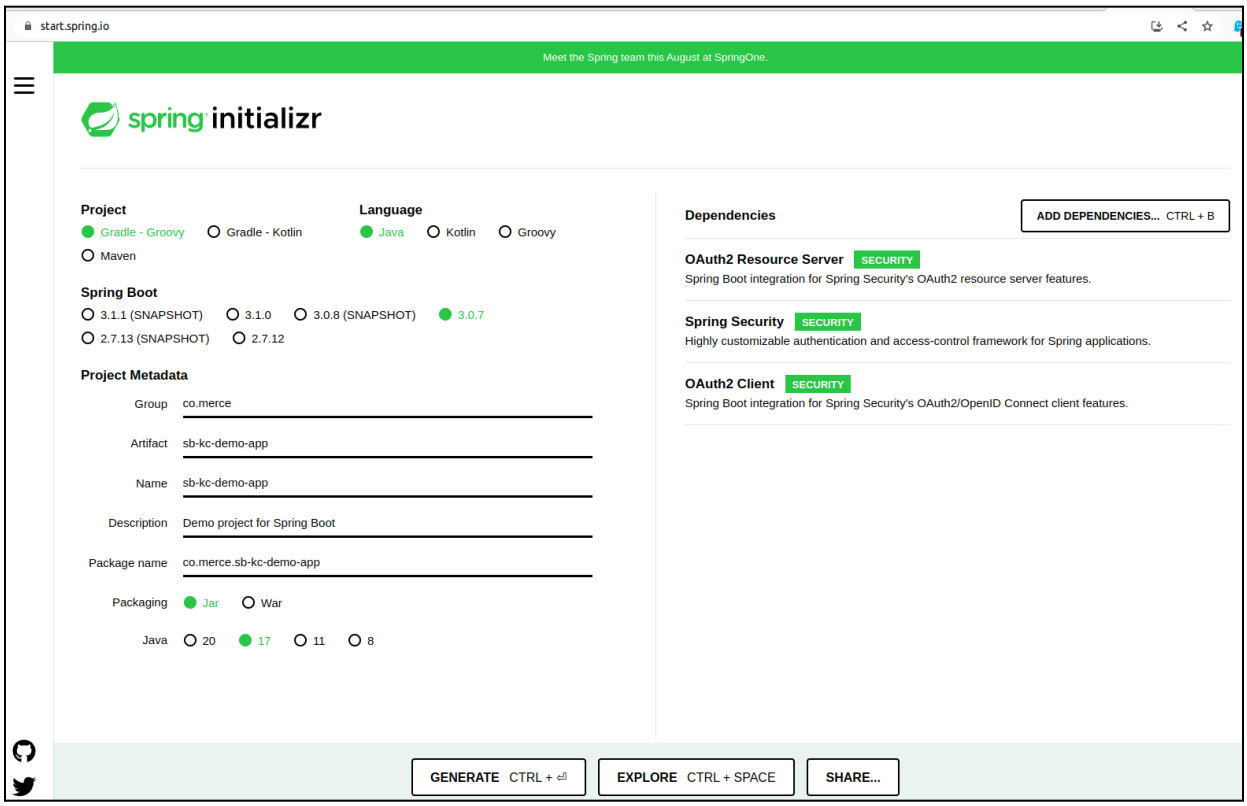
As you can see in the above screenshot, under ‘resource_access’>’myclient-sb’>’roles’>’admin’ Here we can see the client name we created in Keycloak, along with the role of the user ‘admin’ We’ll use this role for the authorization part in the steps ahead.

2.3 Business application

We’ll now create a Spring boot application that will use Spring Security to secure the application via OAuth and it’ll work with Keycloak for Authentication and Role level Authorization.

You can find the entire working code on Github link
[https://github.com/merce-bhavyag/sb-kc-demo]

Go to https://start.spring.io/ and we’ll get a new spring boot application.
Note: You can also use the latest Spring Boot 3.1.0 as well

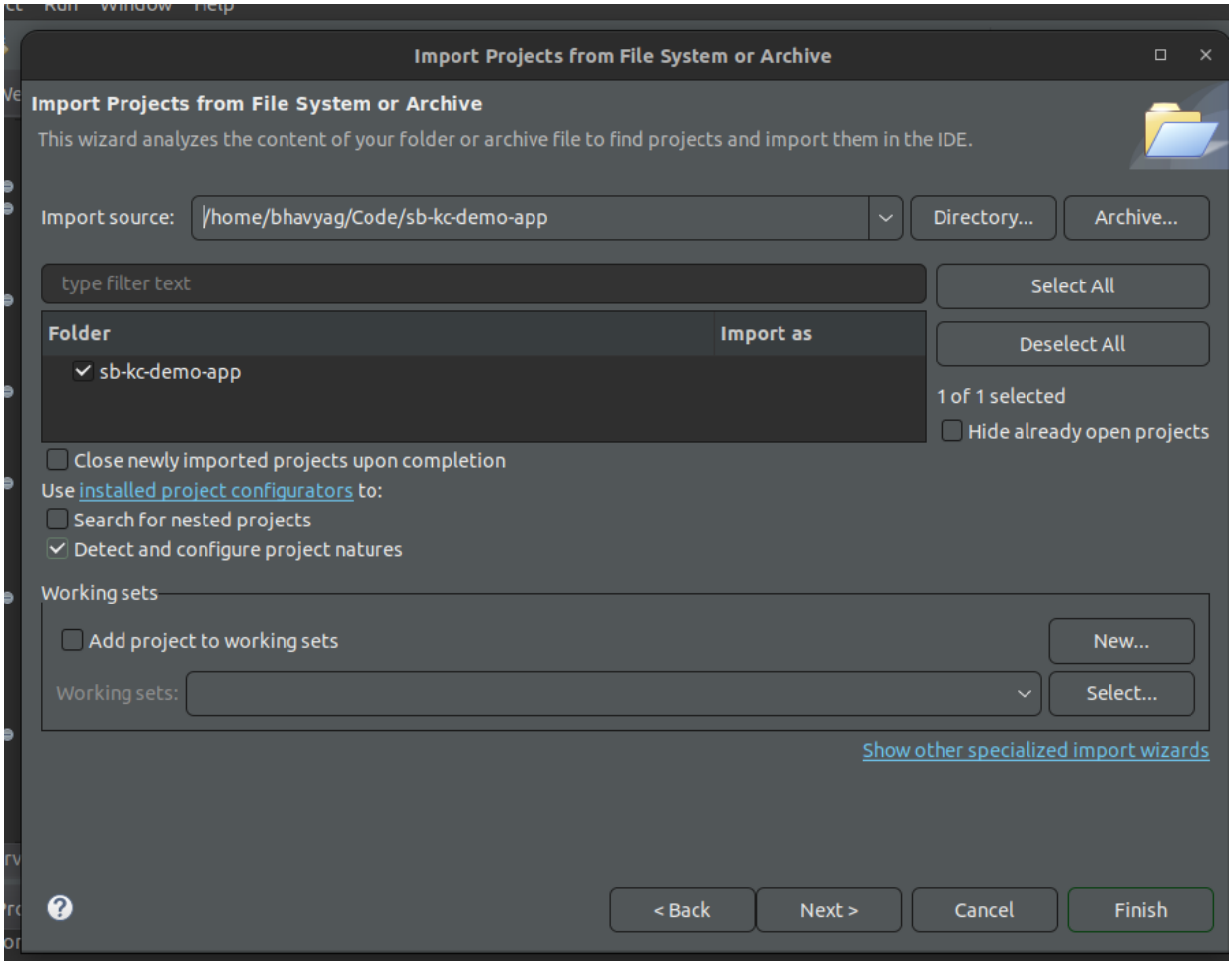


And we'll click on "GENERATE"
This will download a zip archive "sb-kc-demo-app.zip"

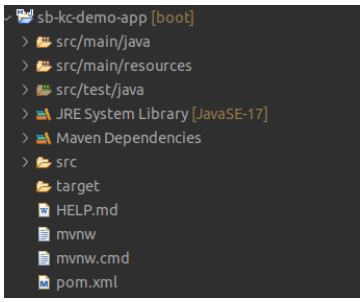
We'll now use Either Eclipse or STS(Spring Tool Suite) to use this downloaded application.

For this guide, I'm using STS, but steps should be the same for Eclipse.

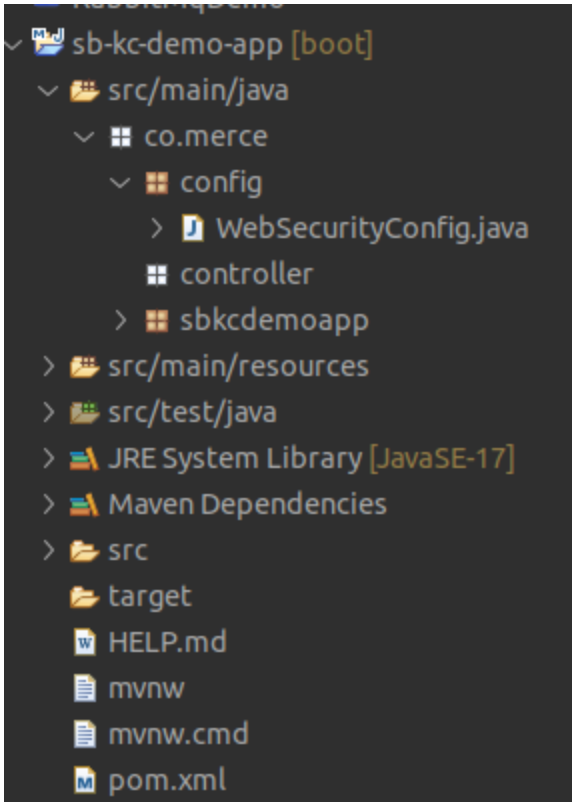
Extract the ZIP file to a folder
Open STS, Click on File>'Open Projects from File System' > Browse for the Zip file folder we downloaded from Spring Initilizr



So now, the application will open in the STS and will look as follows:



We'll create two packages 'config' and 'controller' within the 'co.merce' package
We'll create config and controllers files under the respective packages



We'll create a new class 'WebSecurityConfig.java' under the config package.

This 'WebSecurityConfig' file will host all the configuration required for securing the springboot application.
We'll add the following annotations to the class:

```
@Configuration

@EnableWebSecurity

@EnableMethodSecurity(securedEnabled = true, jsr250Enabled = true)
```

@Configuration : Will mark the class as a configuration for spring boot.
@EnableWebSecurity : This will enable Spring Web Security for the application
@EnableMethodSecurity(securedEnabled = true, jsr250Enabled = true) : This will enable Method based security annotations and Spring will now look for "@Secured" annotation on methods and will secure the method accordingly.

Here we'll add a bean to manage the HTTP requests the spring boot application will receive.

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
http.authorizeHttpRequests(requests -> requests
            /*
```

```
* We can set Global Request-Role Matchers here. Following are some of the
examples.
*/
            //.requestMatchers(HttpMethod.GET, "/test/admin",
"/test/admin/**").hasRole("admin")
            //.requestMatchers(HttpMethod.GET, "/test/user").hasAnyRole("user",
"admin")
            //.requestMatchers(HttpMethod.GET, "/test/anonymous",
"/test/anonymous/**").permitAll()

            /*
* Alternatively, these can be set directly on the methods. In case you want to
* use Roles defined on methods, then you have to add @EnableMethodSecurity
* annotation
*/
            .anyRequest()
            .authenticated()
            );
/*
* Add JWT Auth Converter to extract the Roles from JWT
*/
http.oauth2ResourceServer(oauth2 -> oauth2
.jwt(jwt -> jwt.jwtAuthenticationConverter(jwtAuthConverter))
);
/*
* Set HTTP session management policy
*/
http.sessionManagement(sessionManagement ->

sessionManagement.sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                  /**
                   * Following policies can be set if need be
                   */
                        //.sessionConcurrency(sessionConcurrency ->
                        //    sessionConcurrency
                        //          .maximumSessions(1)
                        //          .expiredUrl("/login?expired")
                        //)
            );
/*
* Return the HTTP object
*/
return http.build();
}
```

Here, if you note, we have our custom ‘jwtAuthConverter’ class that will extract the role information from the JWT token of the request.

Payload of a Decoded JWT token looks like this:


```
{
  "exp": 1684504093,
  "iat": 1684503793,
  "jti": "47fd2933-be33-4fba-bc98-2a83db11a80a",
  "iss": "http://localhost:8080/realms/myrealm",
  "aud": "account",
```

```
 "sub": "ead76933-6bd0-4618-b19d-b7ed31588e87",
 "typ": "Bearer",
 "azp": "myclient-sb",
 "session_state": "f1c96b22-70eb-4f4c-9e06-f10d28c2cd94",
 "acr": "1",
 "realm_access": {
  "roles": [
   "default-roles-myrealm",
   "offline_access",
   "app-admin",
   "uma_authorization"
  ]
 },
 "resource_access": {
  "myclient-sb": {
   "roles": [
    "admin"
   ]
  },
  "account": {
   "roles": [
    "manage-account",
    "manage-account-links",
    "view-profile"
   ]
  }
 },
 "scope": "openid profile email",
 "sid": "f1c96b22-70eb-4f4c-9e06-f10d28c2cd94",
 "email_verified": false,
 "name": "myadmin user",
 "preferred_username": "myadminuser",
 "given_name": "myadmin",
 "family_name": "user",
 "email": "myadminuser@merce.co"
}
```


From the above token, we need to extract roles of our client “myclient-sb” which falls under “resource_access”

Following is the method within the jwtAuthConverter that extracts roles as per above logic:

```
private Collection<? extends GrantedAuthority> extractResourceRoles(Jwt jwt) {
Map<String, Object> resourceAccess;
Map<String, Object> resource;
Collection<String> resourceRoles;
      resourceAccess = jwt.getClaim("resource_access");
       if (resourceAccess == null
|| (resource = (Map<String, Object>)
resourceAccess.get(properties.getResourceId())) == null
```

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

Of course there are other methods which we'll need in this JwtConverter as follows:

```
public AbstractAuthenticationToken convert(Jwt jwt) {
      Collection<GrantedAuthority> a =
jwtGrantedAuthoritiesConverter.convert(jwt);
      Collection<? extends GrantedAuthority> b = extractResourceRoles(jwt);
      Collection<GrantedAuthority> authorities;
      if(a!=null) {
            authorities =
Stream.concat(a.stream(),b.stream()).collect(Collectors.toSet());
      }else {
            authorities=b.stream().collect(Collectors.toSet());
      }
return new JwtAuthenticationToken(jwt, authorities,
getPrincipalClaimName(jwt));
}
```

And

```
private String getPrincipalClaimName(Jwt jwt) {
String claimName = JwtClaimNames.SUB;
if (properties.getPrincipalAttribute() != null) {
claimName = properties.getPrincipalAttribute();
}
return jwt.getClaim(claimName);
}
```

We'll set the application.yml to set up the configuration as follows:

```yaml
server:
  port: 8090                                ## Spring boot app Server port

spring:
  application:
    name: springboot-keycloak               ## Spring boot app name
  security:
    oauth2:
      resourceserver:
        jwt:
          issuer-uri: ${keycloak.auth-server-url}realms/${keycloak.realm}                                  ## Do Not Change
          jwk-set-uri: ${spring.security.oauth2.resourceserver.jwt.issuer-uri}/protocol/openid-connect/certs  ## Do Not Change

jwt:
  auth:
    converter:
      resource-id: ${keycloak.resource}       ## Do Not Change
      principal-attribute: preferred_username ## Do Not Change
      use-realm-role: false                   ## TRUE(If you want to use REALM ROLES). FALSE(If you want to use keycloak client level roles)

keycloak:
  realm: myrealm                            ## Keycloak Realm name
  auth-server-url: http://localhost:8080/   ## Keycloak server URL. Ensure "/" at end of URL
  ssl-required: external                    ## Do Not Change
  resource: myclient-sb                     ## Keycloak Client Name
  credentials:
    secret: EvP10E952ivNyy6MeCpBWGjJciPEQjQV  ## Keycloak Client Secret
  use-resource-role-mappings: true          ## Do Not Change
  bearer-only: true                         ## Do Not Change
```

Ensure you set the correct client secret [as seen in step 9 of 2.1.2.3] in “credentials.secret” parameter of application.yml

We’ll now create a TestController and secure the methods via the roles as follows:

```java
@RestController
@RequestMapping("/test")
public class TestController {
	private static final Logger logger =
LoggerFactory.getLogger(TestController.class);
	//@PreAuthorize("hasRole('ROLE_USER')")
	@Secured("user")
	@GetMapping(value = "/user")
	public ResponseEntity<String> getUser(Principal principal) {
		logger.info("Hello form method User to user{}
",principal.getName());
		return ResponseEntity.ok("Hello form method User to User
"+principal.getName());
	}
	@Secured("admin")
	@GetMapping(value = "/admin")
	public ResponseEntity<String> getAdmin(Principal principal) {
		logger.info("Hello from Admin");
		return ResponseEntity.ok("Hello from method Admin To user
"+principal.getName());
	}
	@Secured({ "user", "admin" })
	@GetMapping(value = "/all-user")
	public ResponseEntity<String> getAllUser(Principal principal) {
		logger.info("Hello from All User");
		return ResponseEntity.ok("Hello from method All User to User
"+principal.getName());
	}
}
```

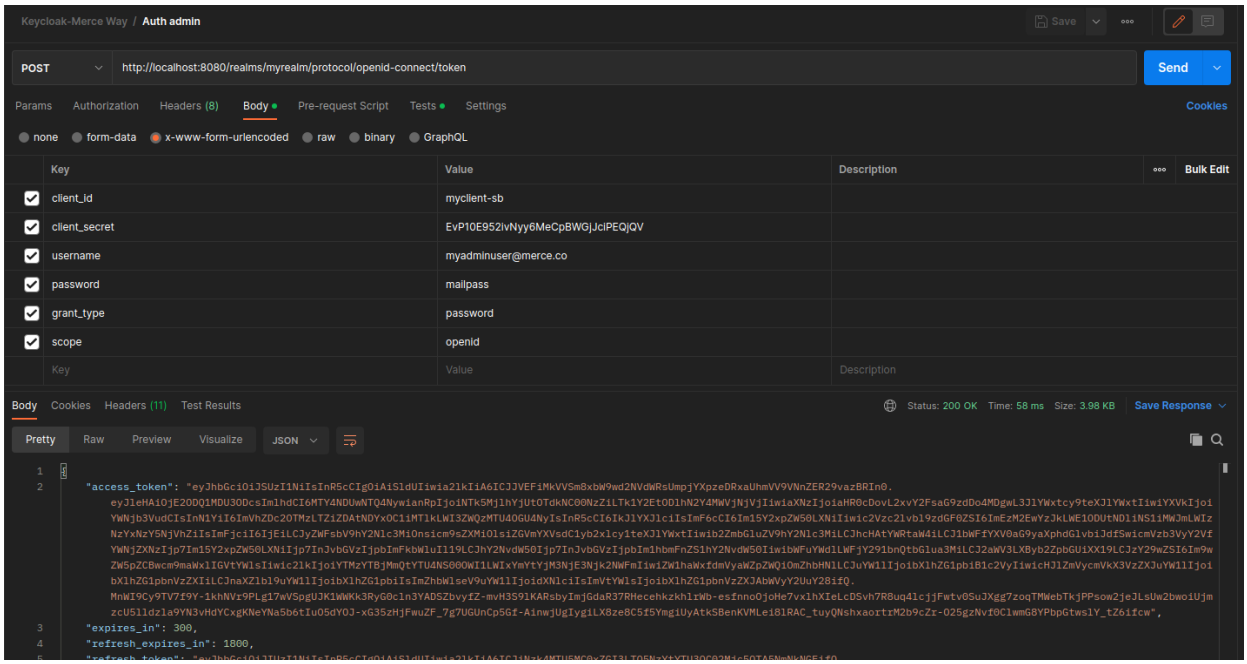
Now once we have everything set up, we can run the application and test if our authentication and authorization works.

Again, the entire code can be downloaded from the github repository
[https://github.com/merce-bhavyag/sb-kc-demo]

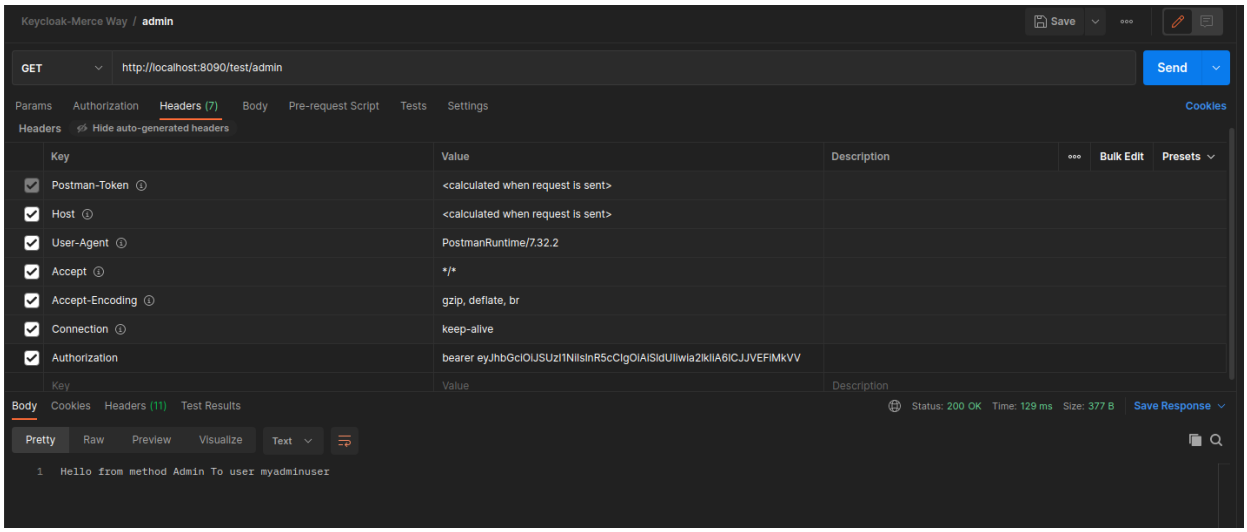
2.4 Test Authentication and Authorization

We'll use the Postman application to test this as we did in Step 2 before.
You can use the postman collection from the link below to test the setup.
[https://github.com/merce-bhavyag/sb-kc-demo/blob/main/postman/Keycloak-Merce-Way.postman_collection.json]

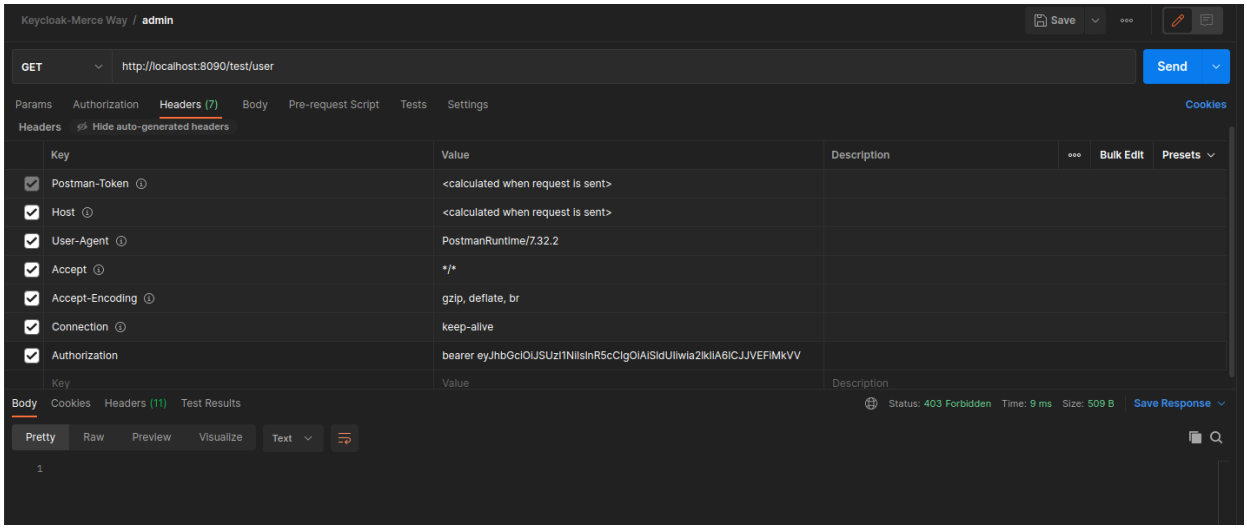
This is how the test looks:



We'll pass the access_token to subsequent requests
So request to our endpoint "/admin" from myadminuser will work.

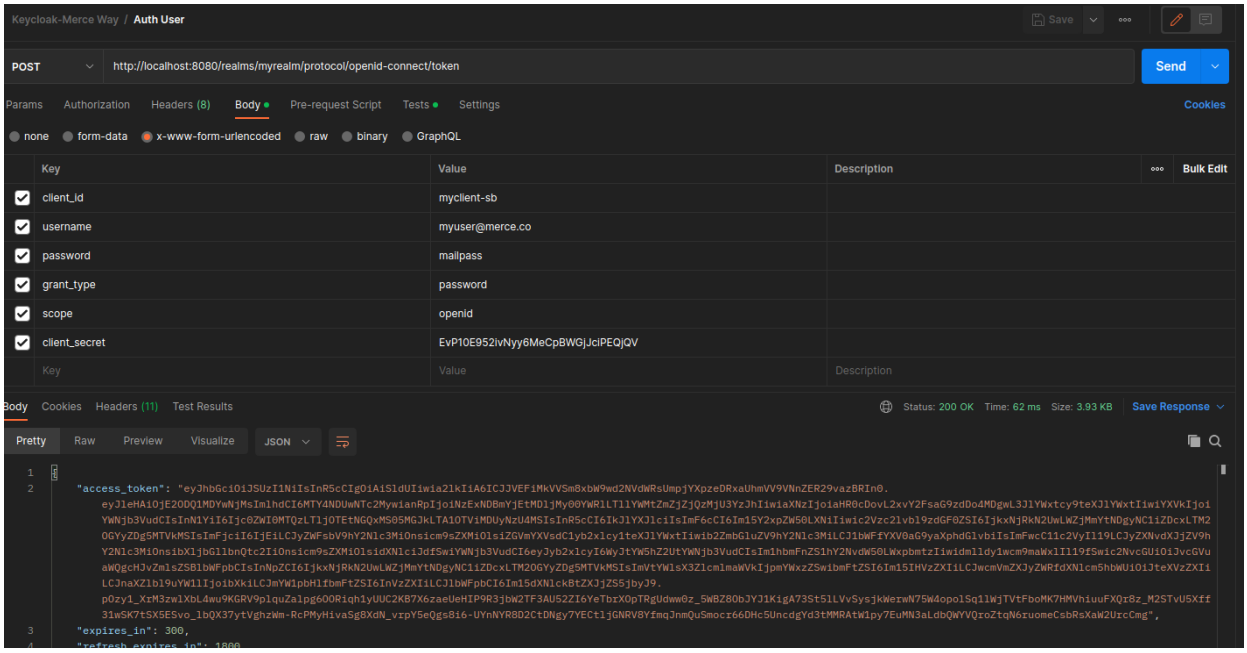


However, the same request to endpoint “/user” will not work

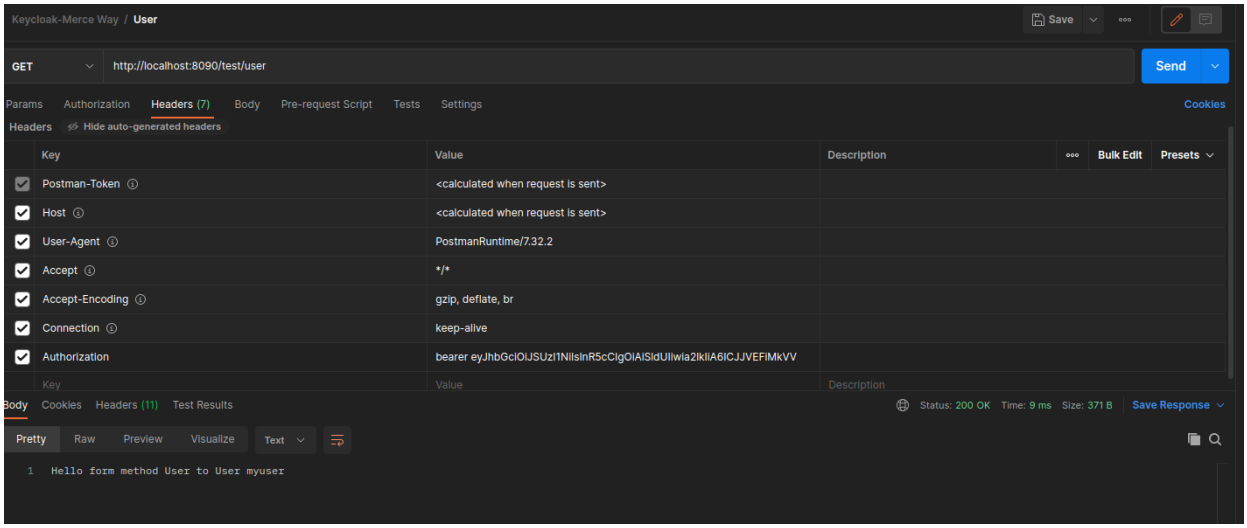


As we can see in the response as “403 Forbidden”

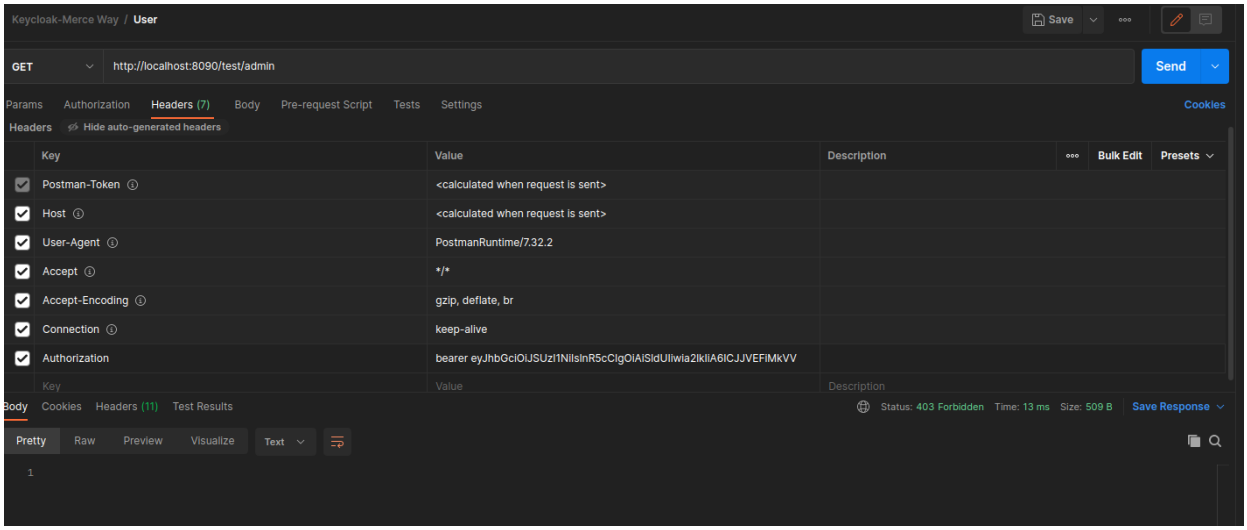
Similarly, access_token with user “myuser” will work with endpoint “/user” and will not work with endpoint “/admin”



We'll use this access_token to our request to endpoint "/user"



We'll get Forbidden error for endpoint "/admin"



So, we have now secured our endpoints with correct roles along with authentication based on Oauth2 framework using Keycloak.