

Integrate Keycloak With Spring Boot Step by Step

Updated on February 22, 2023 by [Đạt Trần](#)

Table of Contents [hide]

[1 API Requirements](#)

[2 Code repository](#)

[3 Quick Keycloak setup with docker compose](#)

[4 Keycloak client configuration](#)

[5 Spring boot application configuration](#)

[6 Guarding API with RolesAllowed](#)

[7 Create account and login using Keycloak Admin Client](#)

[8 Conclusion](#)

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

If you need help [integrating Keycloak with Spring Boot 3, check this tutorial out.](#)

API Requirements

There are some requirements as follow:

The spring boot API can create and login (return access token) to the caller

There are three roles (`admin`, `moderator`, `member`) in the app and each API endpoint can be guarded by a specific role require. There are also endpoints that don't require authorization (publicly available)

Endpoints that are accessible by `member` are also accessible by `admin` and `moderator`. Endpoints that are accessible by `moderator` are accessible by `admin`

Code repository

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

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

Quick Keycloak setup with docker compose

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

YAML



```
1 version: '3'  
2  
3 services:  
4   keycloak:  
5     container_name: keycloak  
6     image: jboss/keycloak:15.0.2  
7     restart: always  
8     env_file: ./keycloak.env  
9     depends_on:  
10       - keycloak_db
```

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

The `keycloak.env` contains some environment variables:

Textile

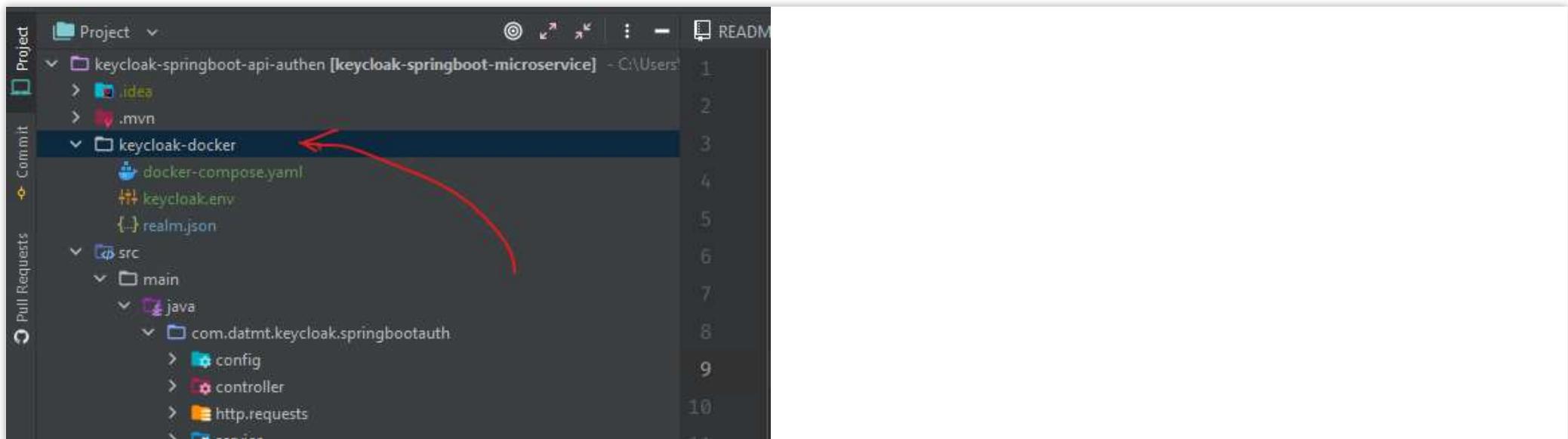


```
1 KEYCLOAK_USER=kc_dev  
2 KEYCLOAK_PASSWORD=kc_dev1231232  
3 KEYCLOAK_IMPORT="-Dkeycloak.profile.feature.upload_scripts=enabled"  
4 DB_VENDOR=mariadb  
5 DB_ADDR=keycloak_db:3306  
6 DB_DATABASE=keycloak_1
```

7

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

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



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

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

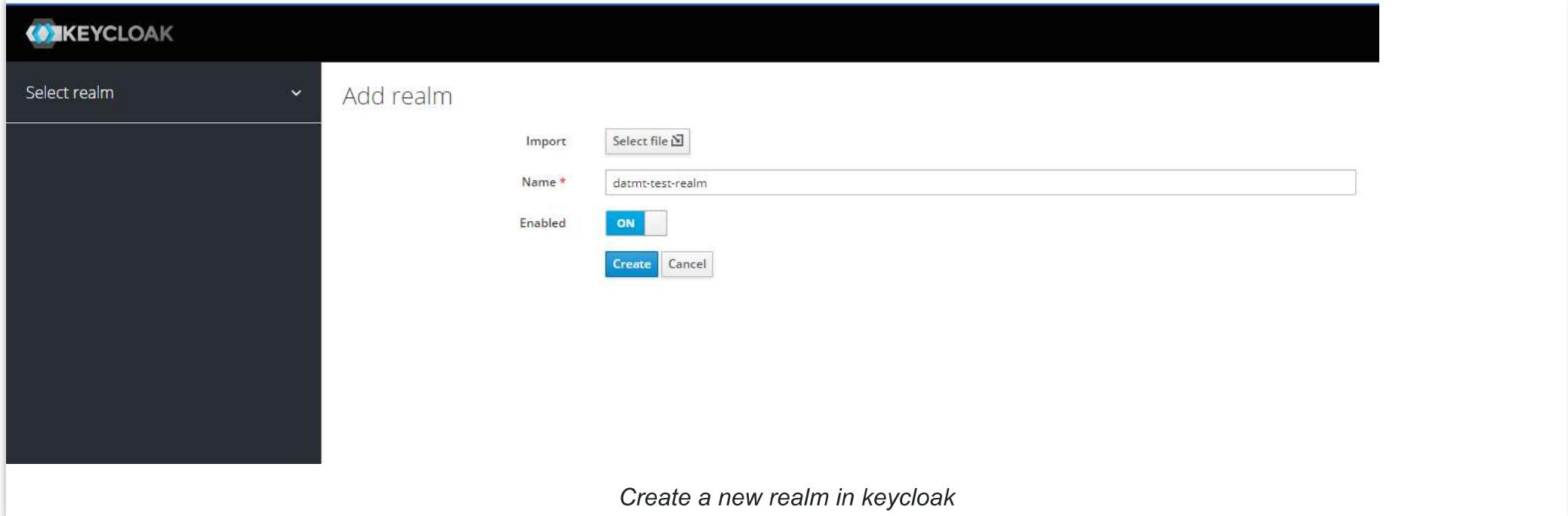
The screenshot shows the Keycloak Admin UI interface. On the left is a sidebar with a dark theme containing the following sections:

- Clients**
- Client Scopes**
- Roles**
- Identity Providers**
- User Federation**
- Authentication**
- Manage**
 - Groups**
 - Users**
 - Sessions**
 - Events**
 - Import**
 - Export**

At the top right of the main content area, there is a "General" tab selected, along with other tabs: Login, Keys, Email, Themes, Localization, Cache, Tokens, Client Registration, Client Policies, and Security Defenses. The main content area displays the configuration for the "Master" realm, which has the following settings:

- Name:** master
- Display name:** Keycloak
- HTML Display name:** <div class="kc-logo-text">Keycloak</div>
- Frontend URL:** (empty input field)
- Enabled:** ON (blue button)
- User-Managed Access:** OFF (gray button)
- Endpoints:** OpenID Endpoint Configuration, SAML 2.0 Identity Provider Metadata

At the bottom of the configuration form are "Save" and "Cancel" buttons.



The screenshot shows the Keycloak interface for adding a new realm. On the left, there's a sidebar with a 'Select realm' dropdown. The main area is titled 'Add realm'. It contains three fields: 'Import' with a 'Select file' button, 'Name *' with the value 'datmt-test-realm', and 'Enabled' with a switch set to 'ON'. At the bottom are 'Create' and 'Cancel' buttons. Below the form, a subtext reads 'Create a new realm in keycloak'.

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

Keycloak client configuration

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

At the beginning, there are some default clients:

The screenshot shows the Keycloak administration interface for the 'Datmt-test-realm'. The left sidebar has a dark theme with navigation links like 'Realm Settings', 'Clients' (which is selected), 'Client Scopes', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. The main content area is titled 'Clients' and contains a table with the following data:

| Client ID | Enabled | Base URL | Actions | | |
|------------------------|---------|--|---------|--------|--------|
| account | True | http://localhost:18080/auth/realm/account/ | Edit | Export | Delete |
| account-console | True | http://localhost:18080/auth/realm/account/console/ | Edit | Export | Delete |
| admin-cli | True | Not defined | Edit | Export | Delete |
| broker | True | Not defined | Edit | Export | Delete |
| realm-management | True | Not defined | Edit | Export | Delete |
| security-admin-console | True | http://localhost:18080/auth/admin/realm/console/ | Edit | Export | Delete |

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

[Clients](#) > Add Client

Add Client

Import

Client ID *

Client Protocol

Root URL

And configure the client like this:

The screenshot shows the Keycloak administration interface. On the left, a sidebar menu includes 'Configure', 'Realm Settings', 'Clients' (which is selected and highlighted in blue), 'Client Scopes', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. Under 'Manage', there are links for 'Groups', 'Users', 'Sessions', 'Events', 'Import', and 'Export'. The main content area is titled 'Spring-boot-client' and shows the 'Settings' tab selected. The configuration fields include:

- Client ID: spring-boot-client
- Name: (empty)
- Description: (empty)
- Enabled: ON
- Always Display in Console: OFF
- Consent Required: OFF
- Login Theme: (dropdown menu)
- Client Protocol: openid-connect
- Access Type: confidential
- Standard Flow Enabled: ON
- Implicit Flow Enabled: OFF
- Direct Access Grants Enabled: ON
- Service Accounts Enabled: ON
- OAuth 2.0 Device Authorization Grant Enabled: ON
- OIDC CIBA Grant Enabled: OFF
- Authorization Enabled: ON
- Root URL: (empty)
- * Valid Redirect URIs: (list box with a '+' button)
- Base URL: (empty)

Below the main configuration, there is a section titled 'Authentication Flow Overrides' with two dropdown menus: 'Browser Flow' set to 'browser' and 'Direct Grant Flow' set to 'browser'. At the bottom are 'Save' and 'Cancel' buttons.

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

admin

moderator

member

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

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

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



The screenshot shows the Keycloak interface for adding a new role. The left sidebar is for the 'Datmt-test-realm' client, with 'Configure' and 'Clients' sections expanded. Under 'Clients', the 'spring-boot-client' is selected. The 'Roles' section is also expanded. The main page title is 'Add Role'. It has two input fields: 'Role Name *' containing 'member' and 'Description' which is empty. At the bottom are 'Save' and 'Cancel' buttons.

After creating the role, you will see there is a switch to turn that role into a composite role.

For the **member** role, we don't need that.

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

The screenshot shows the Keycloak admin interface for the 'Datmt-test-realm'. The left sidebar is collapsed. The main page shows the 'Moderator' role configuration under the 'spring-boot-client' client. The 'Details' tab is selected. The 'Composite Roles' switch is turned 'ON' (indicated by a red arrow). In the 'Composite Roles' section, the 'Client Roles' tab is selected, showing the 'spring-boot-client' role. Red arrows point from the 'Available Roles' and 'Associated Roles' sections of the 'Client Roles' tab towards the 'Add selected >' button.

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

Let's do the same for the admin role.

The screenshot shows the Keycloak Admin UI for the 'Datmt-test-realm'. The left sidebar is dark-themed and includes sections for 'Clients' (selected), 'Realm Settings', 'Roles', 'Identity Providers', 'User Federation', and 'Authentication'. The main content area shows the 'Admin' role configuration for the 'spring-boot-client' client. The 'Details' tab is selected, showing the 'Role Name' as 'admin', an empty 'Description' field, and a 'Composite Roles' switch set to 'ON'. Below the form are 'Save' and 'Cancel' buttons. A 'Composite Roles' section is expanded, showing two tabs: 'Realm Roles' and 'Client Roles'. Under 'Realm Roles', there is a list of 'Available Roles': 'default-roles-datmt-test-realm', 'offline_access', and 'uma_authorization'. An 'Add selected >' button is present. Under 'Client Roles', there is a list of 'Available Roles': 'admin', 'member', and 'uma_protection'. An 'Add selected >' button is also present. To the right of these lists are 'Associated Roles' sections for both tabs, which are currently empty. Buttons for 'Remove selected' are available in both sections.

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

Spring boot application configuration

Let's configure the `application.properties` file.

Textile

```
1 #keycloak
2 keycloak.realm=datmt-test-realm
3 keycloak.auth-server-url=http://localhost:18080/auth/
4 #use external in production
5 keycloak.ssl-required=none
6
7 #name of the client
```



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

The screenshot shows the Keycloak administration interface. On the left, a sidebar menu is open for the realm 'Datmt-test-realm'. Under the 'Clients' section, a client named 'Spring-boot-client' is selected. The main content area shows the 'Credentials' tab of the client configuration. The 'Client Authenticator' dropdown is set to 'Client Id and Secret', and the 'Secret' field contains the value 'ee923b32-a4f1-4435-8066-9eb4541e816c'. There is also a 'Regenerate Secret' button. Below this, there is a 'Registration access token' input field and a 'Regenerate registration access token' button.

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

Java

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

2

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

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

32

```
.anyRequest()
```

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

Paths begins with `/admin` require `admin` role

Paths begin with `/member` require `member` role

Paths begin with `/moderator` require `moderator` role

Paths begin with `/public` is accessible by all

Let's create a controller and test our config:

Java

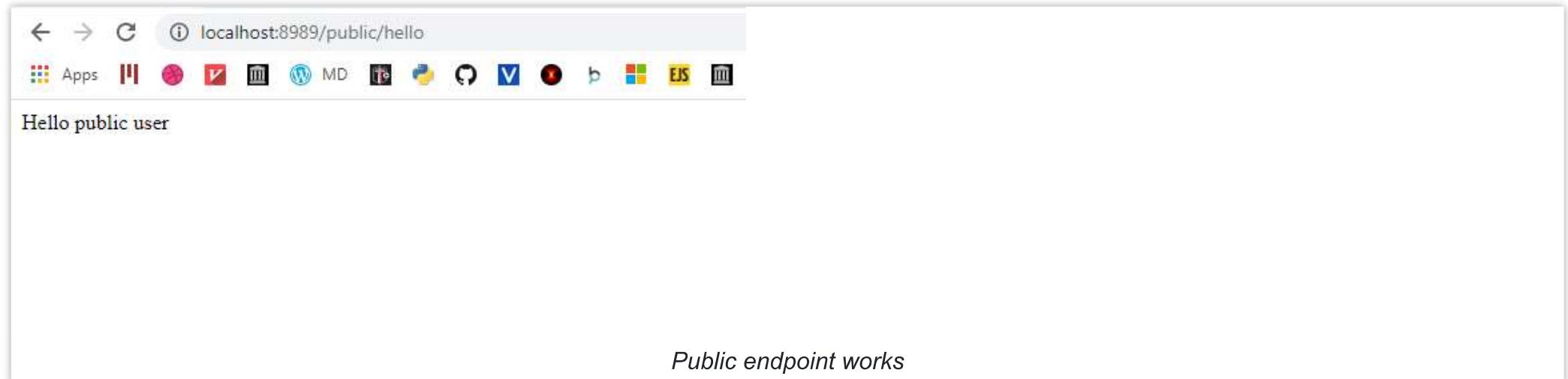


```
1 package com.datmt.keycloak.springbootauth.controller;  
2  
3 import org.springframework.http.ResponseEntity;  
4 import org.springframework.web.bind.annotation.GetMapping;  
5 import org.springframework.web.bind.annotation.RequestHeader;  
6 import org.springframework.web.bind.annotation.RequestMapping;
```

```
7 import org.springframework.web.bind.annotation.RequestMapping;  
8 import org.springframework.web.bind.annotation.RestController;  
9 import javax.annotation.security.RolesAllowed;  
10  
11 @RestController  
12 public class HelloController {  
13  
14     @GetMapping("/public/hello")  
15     public ResponseEntity<String> helloPublic() {  
16         return ResponseEntity.ok("Hello public user");  
17     }  
18  
19     @GetMapping("/member/hello")  
20     public ResponseEntity<String> helloMember() {  
21         return ResponseEntity.ok("Hello dear member");  
22     }  
23 }
```

22

Let's try the public endpoint:



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

Now let's try the path for member:

← → C ⓘ localhost:8989/member/hello

Apps MD EJS

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Nov 21 11:09:57 ICT 2021

There was an unexpected error (type=Unauthorized, status=401).

Unauthorized

and also admin:

Whitelabel Error Page

This application has no explicit mapping for /error, so you are seeing this as a fallback.

Sun Nov 21 11:10:18 ICT 2021

There was an unexpected error (type=Unauthorized, status=401).

Unauthorized

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

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

Let's first create an user with **member** role:

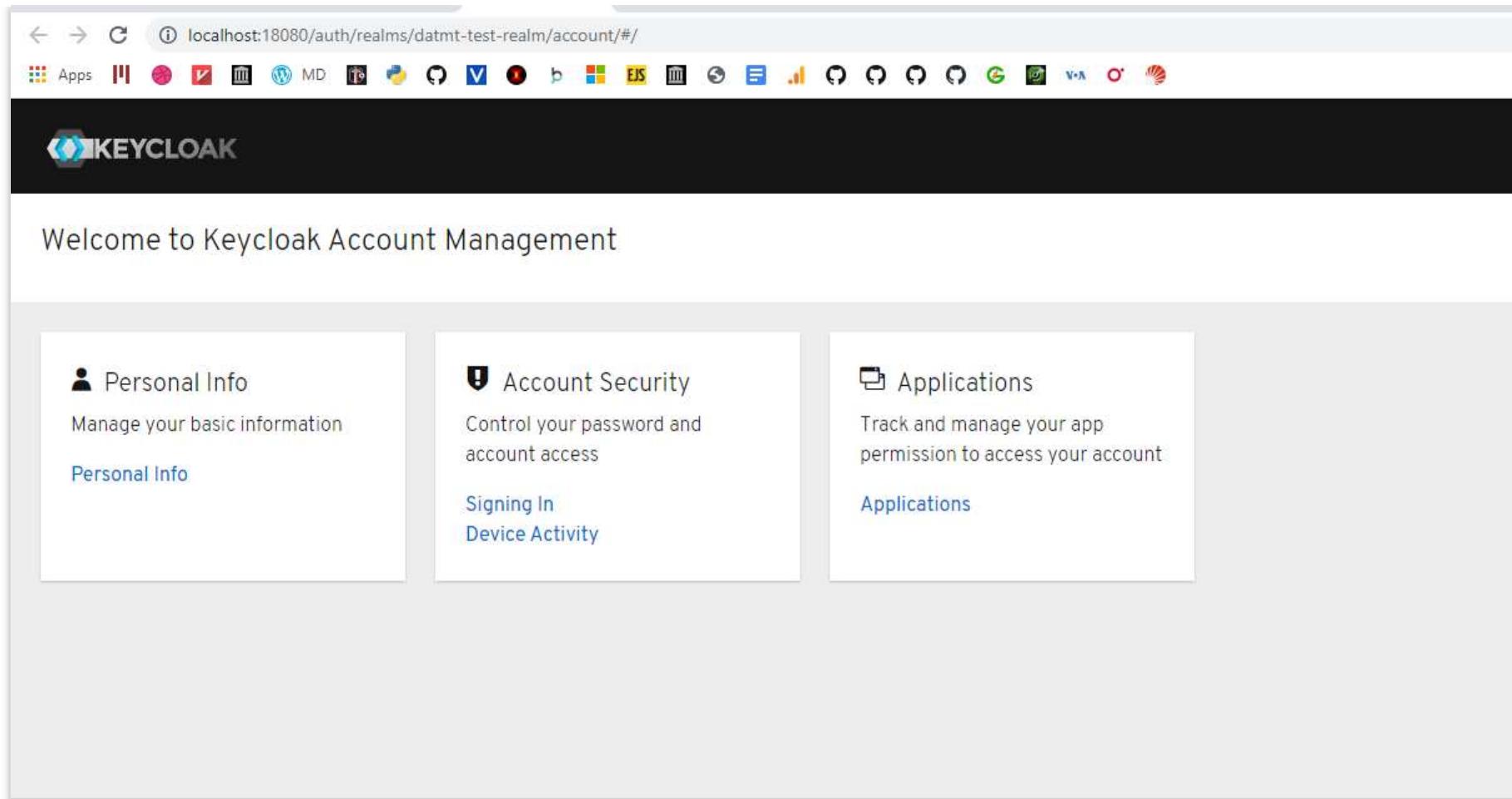
The screenshot shows the Keycloak administration interface for a realm named 'Datmt-test-realm'. The left sidebar has 'Configure' and 'Manage' sections. Under 'Manage', the 'Users' tab is selected, indicated by a blue bar. The main content area is titled 'Add user' and contains fields for ID (empty), Created At (empty), Username (set to 'jane_the_user'), Email (empty), First Name (empty), Last Name (empty), User Enabled (set to 'ON'), Email Verified (set to 'ON'), Groups (a dropdown menu showing 'Select existing group...' and 'No group selected'), and Required User Actions (a dropdown menu showing 'Select an action...'). At the bottom are 'Save' and 'Cancel' buttons.

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

The screenshot shows the Keycloak Admin UI for the 'Datmt-test-realm'. On the left, the navigation sidebar is visible with the 'Users' tab selected. The main page displays the 'Role Mappings' tab for the user 'jane_the_user'. There are two sections: 'Realm Roles' and 'Client Roles'. In the 'Realm Roles' section, the 'Assigned Roles' list contains 'default-roles-datmt-test-realm'. In the 'Client Roles' section, for the client 'spring-boot-client', the 'Assigned Roles' list contains 'member'. A success message at the top right says 'Success! Role mappings updated.'.

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

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



Welcome to Keycloak Account Management

Personal Info
Manage your basic information
[Personal Info](#)

Account Security
Control your password and account access
[Signing In](#)
[Device Activity](#)

Applications
Track and manage your app permission to access your account
[Applications](#)

The screenshot shows the Keycloak Account Management interface. At the top, there's a navigation bar with icons for various applications. Below it is the Keycloak logo. The main content area has three cards: "Personal Info", "Account Security", and "Applications". The "Personal Info" card contains links to "Personal Info" and "Device Activity". The "Account Security" card contains links to "Signing In" and "Device Activity". The "Applications" card contains a link to "Applications". At the bottom of the page, a developer tools network tab is open, showing a timeline of requests and a detailed view of a selected token named "token". The token details include:
access_token: eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJIN0NWjRIMgtIZWFKQm1LUTY1V0dHanVXY1Zyd1ZDZjRmQ1NRYm1wUzYwIn0.eyJleHAiOjE
expires_in: 300
id_token: eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICJIN0NWjRIMgtIZWFKQm1LUTY1V0dHanVXY1Zyd1ZDZjRmQ1NRYm1wUzYwIn0.eyJleHAiOjE2Mzc
not-before-policy: 0
refresh_expires_in: 1800
refresh_token: eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUIiwia2lkIiA6ICIxNzM0GE4Mc02MNniLTrmYzItYjY3NC11ZmE5MDA3NWzNNiifQ.eyJleHAiOjE2Mzc0ODI
scope: openid email profile"

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

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

The screenshot shows a Postman request configuration and its response. The request is a GET to `http://localhost:8989/member/hello`. The Authorization tab is selected, showing a Bearer Token type with the value `eyJhbGciOiJSUzI1NiIsInR5cC IgOiAiSldUiwi...`. The response status is 403 Forbidden, with a timestamp of 2021-11-21T07:42:36.227+0000, status 403, error Forbidden, message Forbidden, and path /member/hello.

```
1  "timestamp": "2021-11-21T07:42:36.227+0000",  
2  "status": 403,  
3  "error": "Forbidden",  
4  "message": "Forbidden",  
5  "path": "/member/hello"
```

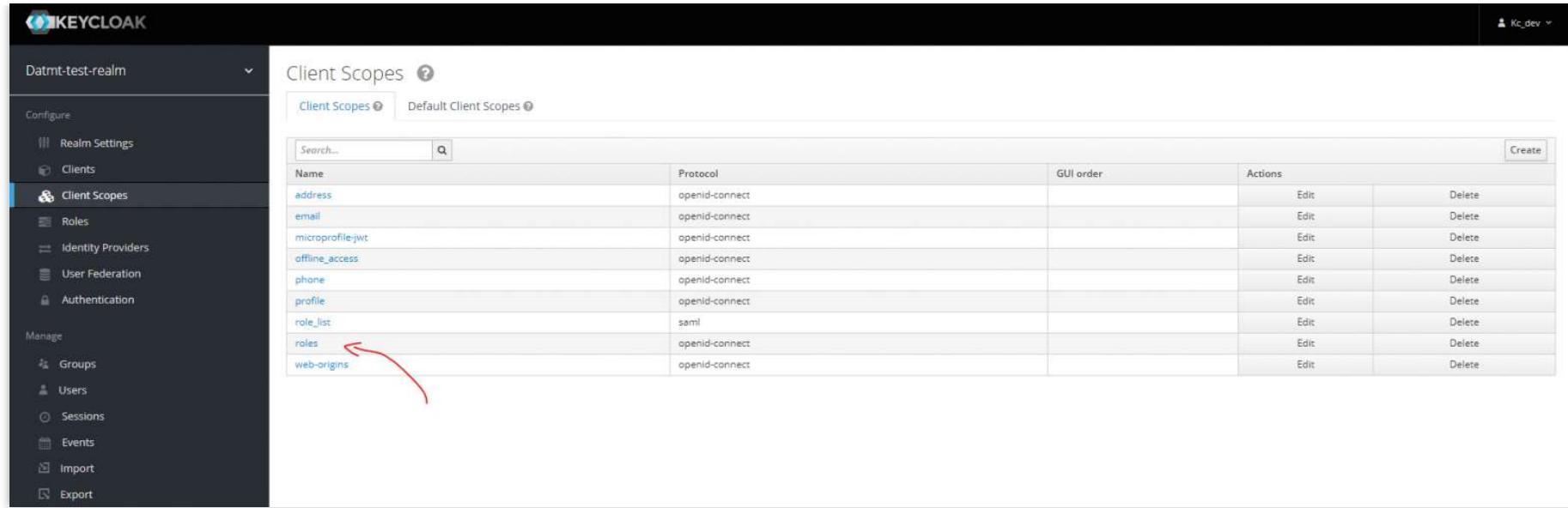
You expected that to be a 200 response, right? But what I got was 403. That's better than 401 since we can authenticate with the token but somehow, the user was not authorized to access the endpoint.

What's the missing configuration?

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

```
{  
  "exp": 1637480705, Sun Nov 21 2021 14:45:05 GMT+0700 (Indochina Time)  
  "iat": 1637480405,  
  "jti": "8de803a2-f154-47b4-ab54-90e0a215cc14",  
  "iss": "http://localhost:18080/auth/realms/datmt-  
test-realm",  
  "aud": "account",  
  "sub": "4e14de83-f723-4bfd-9bc0-ab9489e59cf1",  
  "typ": "Bearer",  
  "azp": "account-console",  
  "nonce": "b3eb849b-4119-47b3-9163-0760855e418e",  
  "session_state": "5f5ded8c-71cd-4eea-86d6-  
1be2b772f620",  
  "acr": "0",  
  "resource_access": {  
    "account": {  
      "roles": [  
        "manage-account",  
        "manage-account-links"  
      ]  
    }  
  },  
  "scope": "openid email profile",  
  "sid": "5f5ded8c-71cd-4eea-86d6-1be2b772f620",  
  "email_verified": true,  
  "preferred_username": "jane_the_user"  
}
```

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



The screenshot shows the Keycloak administration interface for the 'Datmt-test-realm'. The left sidebar is a navigation menu with sections like 'Realm Settings', 'Clients', 'Client Scopes' (which is currently selected), 'Roles', 'Identity Providers', 'User Federation', 'Authentication', 'Groups', 'Users', 'Sessions', 'Events', 'Import', and 'Export'. The main content area is titled 'Client Scopes' and contains two tabs: 'Client Scopes' (selected) and 'Default Client Scopes'. A search bar at the top of the table allows filtering by name. The table lists various client scopes with columns for 'Name', 'Protocol', 'GUI order', and 'Actions' (Edit and Delete). One row, 'roles', is highlighted with a red arrow pointing to it from the bottom-left.

| Name | Protocol | GUI order | Actions |
|------------------|----------------|-----------|-------------|
| address | openid-connect | | Edit Delete |
| email | openid-connect | | Edit Delete |
| microprofile-jwt | openid-connect | | Edit Delete |
| offline_access | openid-connect | | Edit Delete |
| phone | openid-connect | | Edit Delete |
| profile | openid-connect | | Edit Delete |
| role_list | saml | | Edit Delete |
| roles | openid-connect | | Edit Delete |
| web Origins | openid-connect | | Edit Delete |

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

The screenshot shows the Keycloak Admin UI for the 'Datmt-test-realm'. The left sidebar is dark-themed and includes sections for 'Configure' (Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication), 'Manage' (Groups, Users, Sessions, Events, Import, Export), and the current 'Client Scopes' section which is highlighted.

The main content area is titled 'Client Scopes > roles'. It shows two sets of role mappings:

- Realm Roles:** Available Roles include 'default-roles-datmt-test-realm', 'offline_access', and 'uma_authorization'. The 'Assigned Roles' and 'Effective Roles' boxes are empty.
- Client Roles:** A client named 'spring-boot-client' is selected. Its available roles are 'uma_protection'. In the 'Assigned Roles' box, there is one entry: 'admin'. In the 'Effective Roles' box, there are three entries: 'admin', 'member', and 'moderator'.

A green success message at the top right says 'Success! Scope mappings updated.' with a close button.

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

PAYLOAD: DATA

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



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

The screenshot shows a Postman request configuration for a GET request to `http://localhost:8989/member/hello`. The **Authorization** tab is selected, showing a **Type** of `Bearer Token`. A warning message states: `Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables.` Below the type dropdown, it says: `The authorization header will be automatically generated when you send the request. Learn more about authorization ↗`. The **Token** field contains a long JWT token: `eyJhbGciOiJSUzI1NiIsInR5cCtgOiAiSldUiwi...`. The response body is displayed as `Hello dear member`.

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

Guarding API with RolesAllowed

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

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

Java

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

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

What if the path is prefix but `/public`?

Java

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

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

http://localhost:8989/public/hello-fake-public

GET http://localhost:8989/public/hello-fake-public

Params Authorization Headers (7) Body Pre-request Script Tests Settings

Type No Auth

This request does not use any authorization. Learn more at

Body Cookies (1) Headers (12) Test Results

Pretty Raw Preview Visualize JSON

```
1 "timestamp": "2021-11-21T07:57:08.948+0000",
2 "status": 401,
3 "error": "Unauthorized",
4 "message": "Unauthorized",
5 "trace": "org.springframework.security.access.AccessDeniedException: Access is denied\r\n\tat org.springframework.security.access.vote
```

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

http://localhost:8989/public/hello-fake-public

Save

GET http://localhost:8989/public/hello-fake-public Send

Params Authorization ● Headers (8) Body Pre-request Script Tests Settings Cookies

Type Bearer Token

⚠ Heads up! These parameters hold sensitive data. To keep this data secure while working in a collaborative environment, we recommend using variables.
Learn more about variables

The authorization header will be automatically generated when you send the request. Learn more about authorization

Token eyJhbGciOiJSUzI1NiIsInR5cCigOiAiSldUiwi...

Body Cookies (1) Headers (11) Test Results Status: 200 OK Time: 17 ms Size: 387 B Save Response

Pretty Raw Preview Visualize Text

1 Nice day, it appears to be public but not

Create account and login using Keycloak Admin Client

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

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

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

The screenshot shows the Keycloak administration interface for the 'spring-boot-client' client within the 'Datmt-test-realm'. The left sidebar contains navigation links for realm settings, clients, roles, identity providers, user federation, authentication, groups, users, sessions, events, import, and export. The main content area displays two sections for role configuration:

- Realm Roles:** Shows three columns: Available Roles (empty), Assigned Roles (containing 'default-roles-datmt-test-realm'), and Effective Roles (containing 'default-roles-datmt-test-realm', 'offline_access', and 'uma_authorization'). Buttons for 'Add selected' and 'Remove selected' are present.
- Client Roles:** Shows three columns: Available Roles (empty), Assigned Roles (containing 'admin' and 'uma_protection'), and Effective Roles (containing 'admin', 'member', 'moderator', and 'uma_protection'). Buttons for 'Add selected' and 'Remove selected' are present.

A banner at the bottom of the page reads: *Configure role for the service account*.

The screenshot shows the Keycloak admin interface for the 'spring-boot-client' client under the 'spring-boot-client Service Accounts' tab. It displays two sets of role assignments:

- Realm Roles:**
 - Available Roles:** default-roles-datmt-test-realm
 - Assigned Roles:** default-roles-datmt-test-realm
 - Effective Roles:** default-roles-datmt-test-realm, offline_access, ume_authorization
- Client Roles:**
 - Available Roles:** realm-management, create-client, impersonation, manage-authorization, manage-clients, manage-events
 - Assigned Roles:** manage-users, query-users
 - Effective Roles:** manage-users, query-users

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

Now, the client is able to manage users.

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

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

Next add a **KeycloakProvider** class:

Java

```
1 @Configuration
2 @Getter
3 public class KeycloakProvider {
4
5   @Value("${keycloak.auth-server-url}")
6   public String serverURL;
7
8   @Value("${keycloak.realm}")
9   public String realm;
10
11  @Value("${keycloak.resource}")
```

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

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

Then a service to create and log in user:

Java

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

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

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

Finally, create a controller to create and login user:

Java



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

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

23

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

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

The screenshot shows a Postman interface with the following details:

- Request URL:** `http://localhost:8989/user/create`
- Method:** POST
- Body (JSON):**

```
1
2   "username": "test_user",
3   "email": "test_user@gmail.com",
4   "password": "nopass"
```
- Response Status:** 201 Created

A red arrow points to the "Status: 201 Created" message in the response area.

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

The screenshot shows the Keycloak administration interface. On the left, a sidebar menu is open for the realm 'Datmt-test-realm'. The 'Users' option is selected under the 'Manage' section. The main content area displays the details for a user named 'Test_user@gmail.com'. The 'Details' tab is active, showing the following fields:

| | |
|-----------------------|--|
| ID | 56a17c6d-853d-4364-b3e1-5a5d9613caed |
| Created At | 11/21/21 5:03:31 PM |
| Username | test_user@gmail.com |
| Email | test_user@gmail.com |
| First Name | (empty) |
| Last Name | (empty) |
| User Enabled | <input checked="" type="checkbox"/> ON |
| Email Verified | <input type="checkbox"/> OFF |
| Required User Actions | Select an action... |
| Impersonate user | Impersonate |

At the bottom right of the form are 'Save' and 'Cancel' buttons.

Simply switch **email verified** on then you can get the tokens:

The screenshot shows the Postman interface with a POST request to `http://localhost:8989/user/login`. The request body is set to `JSON` and contains the following data:

```

1
2   ...
3     "username": "test_user@gmail.com",
4     "password": "nopass"

```

The response status is `200 OK` with a time of `122 ms` and a size of `2.33 KB`. The response body is a large JSON object representing the access token, which is partially visible:

```

1
2   "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldeUiwia2lkIiA6ICJIN0NWWjRIMGTIZWFKQmlLUTY1V0dHanVXY1ZydlZDzjRmQ1NRym1wUzYwIn0...
eyJleHAiOjE2Mzc00Dk1MTEsImlhCI6MTYzNzQ40TiXMsSwianRpIjoimNmZmRkNGYtZTk6Yy00ZjhilTgxNTgtMTA50GiwNGF1MjgzIiwiAxNzIjoi...
...
3
4   "expires_in": 300,
5   "refresh_expires_in": 1800,
6   "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldeUiwia2lkIiA6ICIxNzM00GE4MC02MWNiLTrmYzItYjY3NC1lZmE5MDA3NWEzNW...
eyJleHAiOjE2Mzc00TEwMTEsImlhCI6MTYzNzQ40TiXMsSwianRpIjoimTjJjNWU40WMnjlZ1yS00ZDUyLWI20TktZWM50GNm0Nv10DlkIiwi...
...
10
11   "token_type": "Bearer",
12   "id_token": null,
13   "not-before-policy": 0,
14   "session_state": "eaf076ab-4ab9-4a13-af1e-3618367d151a",
15   "scope": "email profile",
16   "error": null,
17   "error_description": null.

```

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

The screenshot shows the 'Login' configuration page for the 'Datmt-test-realm' in Keycloak. The 'Login' tab is active. The configuration includes:

- User registration: OFF
- Edit username: OFF
- Forgot password: OFF
- Remember Me: OFF
- Verify email: OFF
- Login with email: ON
- Require SSL: external requests

At the bottom are 'Save' and 'Cancel' buttons.

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

Conclusion

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

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