

You have **2 free member-only stories left** this month. [Sign up](#) for Medium and get an extra one.

◆ Member-only story

# Using Keycloak with Spring Boot 3.0

Hands-on guidance to integrate Keycloak with Spring Boot v3



Yasar 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.

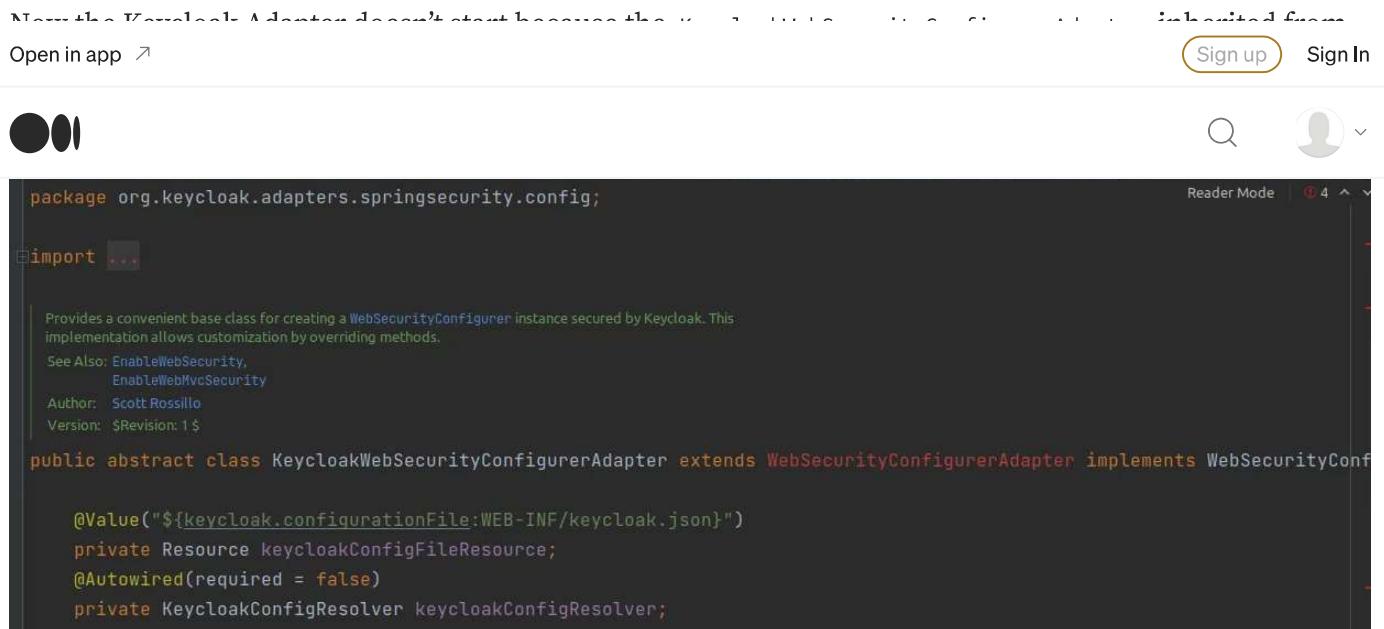
It introduced so many features/changes such as Java 17 baseline, enhanced support for native compilation with GraalVM, improved observability with Micrometer Tracing, support for Jakarta EE etc. Visit the [Release Notes](#) for more info.

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



WebSecurityConfigurerAdapter 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.

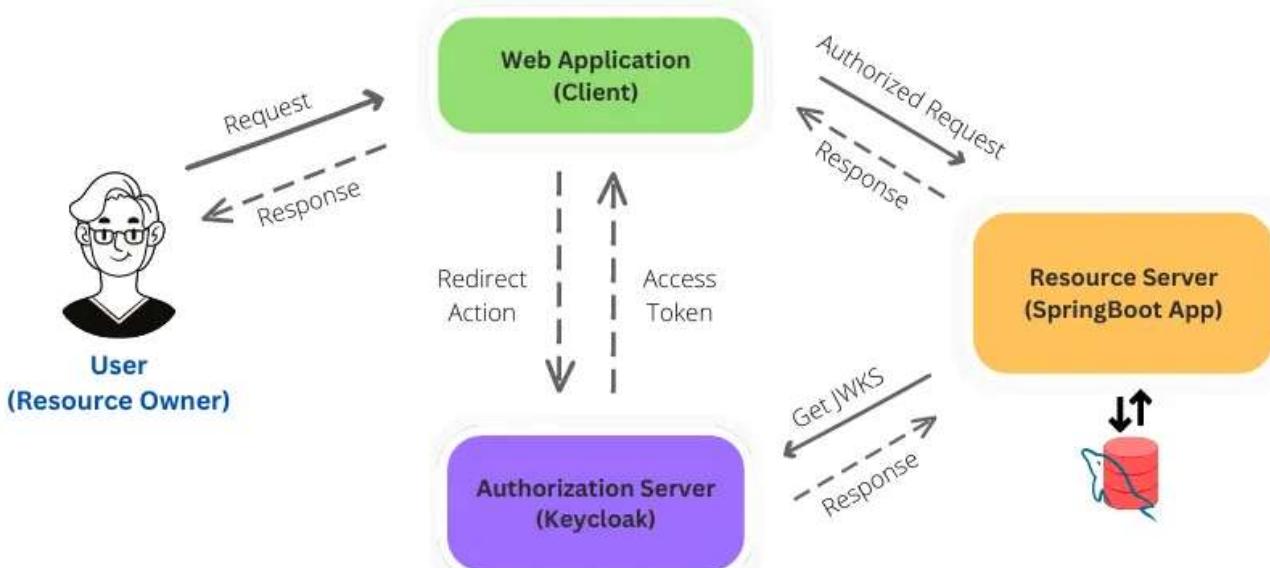
## Keycloak Authorization Flow

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](https://github.com)

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

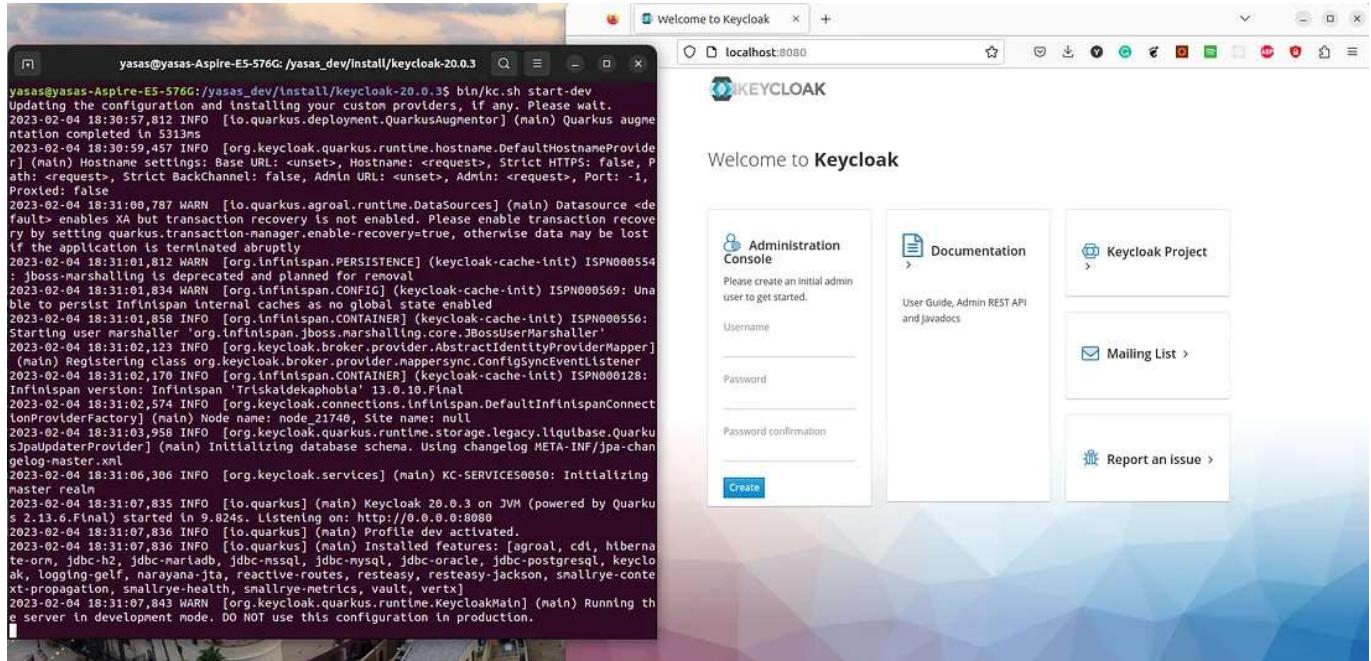
Extract the downloaded Keycloak zip/tar.gz file.

```

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.

master

Create realm

A realm manages a set of users, credentials, roles, and groups. 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.

Resource file

Drag a file here or browse to upload

Browse... Clear

1

Upload a JSON file

Realm name \*

SpringBootKeycloak

Enabled

On

Create Cancel

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.

SpringBootKeycloak

Clients

Clients are applications and services that can request authentication of a user. [Learn more](#)

Clients list Initial access token

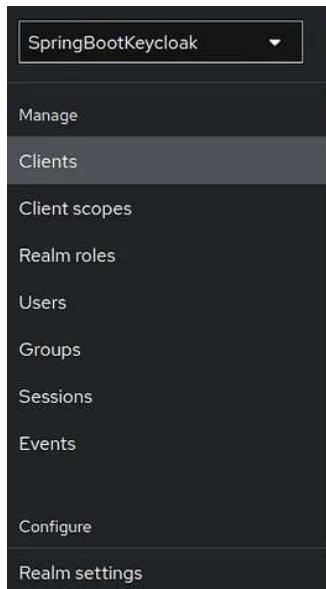
Search for client Create client Import client

Client ID	Type	Description	Home URL
account	OpenID Connect	–	<a href="http://localhost:8080/realms/SpringBootKeycloak/account/">http://localhost:8080/realms/SpringBootKeycloak/account/</a>
account-console	OpenID Connect	–	<a href="http://localhost:8080/realms/SpringBootKeycloak/account-console/">http://localhost:8080/realms/SpringBootKeycloak/account-console/</a>
admin-cli	OpenID Connect	–	–
broker	OpenID Connect	–	–
realm-management	OpenID Connect	–	–
security-admin-console	OpenID Connect	–	<a href="http://localhost:8080/admin/SpringBootKeycloak/console/">http://localhost:8080/admin/SpringBootKeycloak/console/</a>

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.



Clients > Create client

## Create client

Clients are applications and services that can request authentication of a user.

**General Settings**

Client type	OpenID Connect
Client ID *	springboot-keycloak-client
Name	
Description	
Always display in console	<input checked="" type="checkbox"/> Off

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.

SpringBootKeycloak

Settings Roles Client scopes Sessions Advanced

Manage Clients Client scopes Realm roles Users Groups Sessions Events Configure Realm settings Authentication Identity providers User federation

**General Settings**

Client ID *	springboot-keycloak-client
Name	
Description	
Always display in console	<input checked="" type="checkbox"/> Off

**Access settings**

Root URL	
Home URL	
Valid redirect URIs	<code>http://localhost:8081/*</code>

**Jump to section**

- General Settings
- Access settings
- Capability config
- Login settings
- Logout settings

**Save** **Revert**

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.

The screenshot shows the Keycloak interface for managing clients. On the left, a sidebar menu includes options like 'Manage', 'Clients', 'Client scopes', 'Realm roles', 'Users', 'Groups', 'Sessions', 'Events', 'Configure', 'Realm settings', 'Authentication', and 'Identity providers'. The 'Clients' option is selected. In the main content area, the title is 'Clients > Client details' for the client 'springboot-keycloak-client' (OpenID Connect). A message states: 'Clients are applications and services that can request authentication of a user.' Below this, tabs for 'Settings', 'Roles', 'Client scopes', 'Sessions', and 'Advanced' are present, with 'Roles' being the active tab. A large blue '+' button is centered above the message 'No roles for this client'. Below the message, a sub-message says: 'You haven't created any roles for this client. Create a role to get started.' A prominent blue 'Create role' button is located at the bottom right of this section.

Creating Client Roles (Image by author)

Create the two roles previously mentioned here.

This screenshot shows the same Keycloak interface as the previous one, but now with two roles listed in the 'Roles' tab. The 'Clients' tab is still selected in the sidebar. The main content area shows the 'springboot-keycloak-client' client details. The 'Roles' tab is active. At the top of the roles list, there is a search bar labeled 'Search role by name' and a blue 'Create role' button. Below this, a table lists the roles: 'Role name' (admin, user), 'Composite' (False), and 'Description' (empty). The table has a footer with the number '1 - 2'.

Role name	Composite	Description
admin	False	-
user	False	-

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.

The screenshot shows the 'Realm roles' section of the Keycloak interface. On the left is a sidebar with options like Manage, Clients, Client scopes, Realm roles (which is selected), Users, Groups, Sessions, and Events. The main area has a search bar 'Search role by name' and a 'Create role' button. A table lists three roles:

Role name	Composite	Description
default-roles-springbootkeycloak	True	\${role_default-roles}
offline_access	False	\${role_offline-access}
uma_authorization	False	\${role_uma_authorization}

Creating Realm Roles (Image by author)

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

The screenshot shows the 'Realm roles' section after adding two new roles, `app_admin` and `app_user`. The table now includes these new entries:

Role name	Composite	Description
app_admin	False	-
app_user	False	-
default-roles-springbootkeycloak	True	\${role_default-roles}
offline_access	False	\${role_offline-access}
uma_authorization	False	\${role_uma_authorization}

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.

The screenshot shows the 'Role details' page for the `app_admin` role. The sidebar on the left is identical to the previous screenshots. The main area shows the `app_admin` role details with a 'Details' tab selected. An 'Action' dropdown menu is open, showing the option `Add associated roles`. Other options in the menu include `Delete this role`. The role details form includes fields for 'Role name \*' (set to `app_admin`) and 'Description'. At the bottom are 'Save' and 'Revert' buttons.

Composite Roles (Image by author)

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

The screenshot shows the 'Assign roles to app\_admin account' dialog. On the left, there's a sidebar with various management options like Manage, Clients, Client scopes, Realm roles, Users, Groups, Sessions, Events, Configure, Realm settings, Authentication, Identity providers, and User federation. The 'Realm roles' option is currently selected. The main area has a search bar with 'admin' and a list of roles. One role, 'springboot-keycloak-client admin', is checked. Another role, 'realm-management realm-admin', is also listed. At the bottom are 'Assign' and 'Cancel' buttons.

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 screenshot shows the 'Realm roles' page. The sidebar has the 'Realm roles' option selected. The main area displays a table of roles. The 'app\_admin' role is marked as 'Composite: True'. The 'app\_user' role is also marked as 'Composite: True'. Other roles like 'default-roles-springbootkeycloak', 'offline\_access', and 'uma\_authorization' are listed with 'Composite' status set to 'False'. A search bar and a 'Create role' button are located at the top of the table area.

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.

SpringBootKeycloak

Users > Create user

Create user

Enabled

Username \* globeAdmin

Email admin@mail.com

Email verified On

First name Admin

Last name

Required user actions Select action

Groups

Create Cancel

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.

SpringBootKeycloak

Users > User details

globeadmin

Enabled Action

Details Attributes Credentials Role mapping Groups Consents Identity provider links Sessions

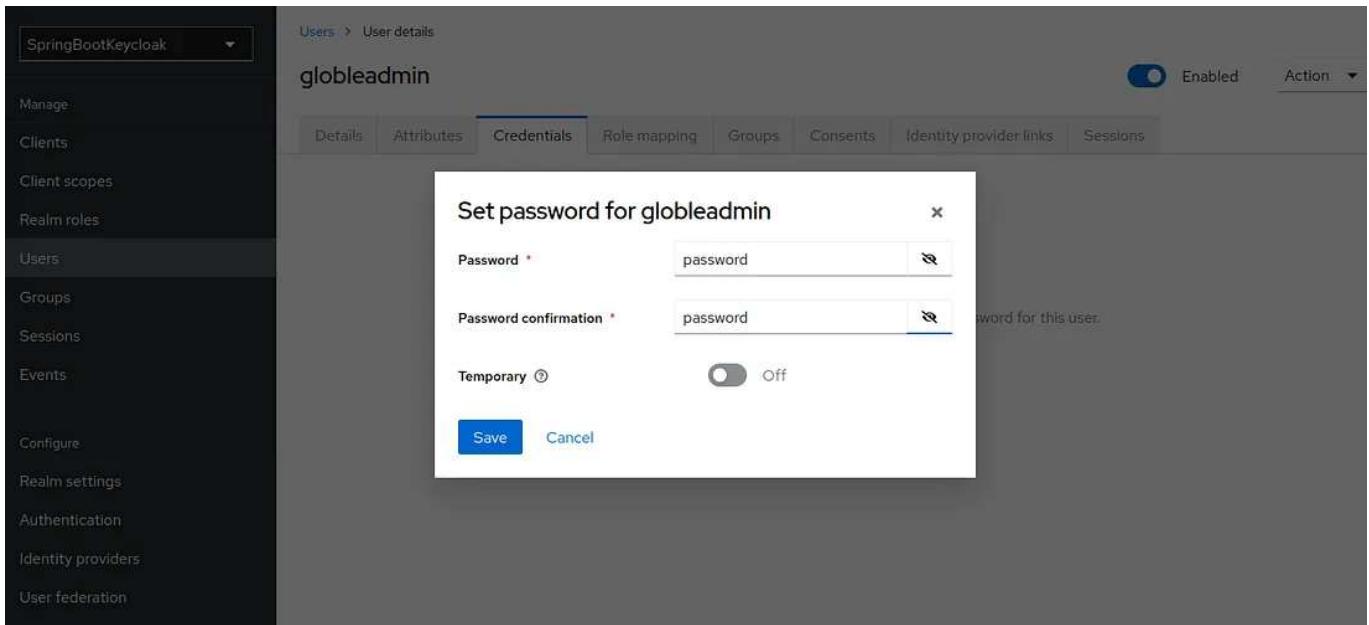
No credentials

This user does not have any credentials. You can set password for this user.

Set password Credential Reset

Adding user credentials (Image by author)

Here you need to 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.



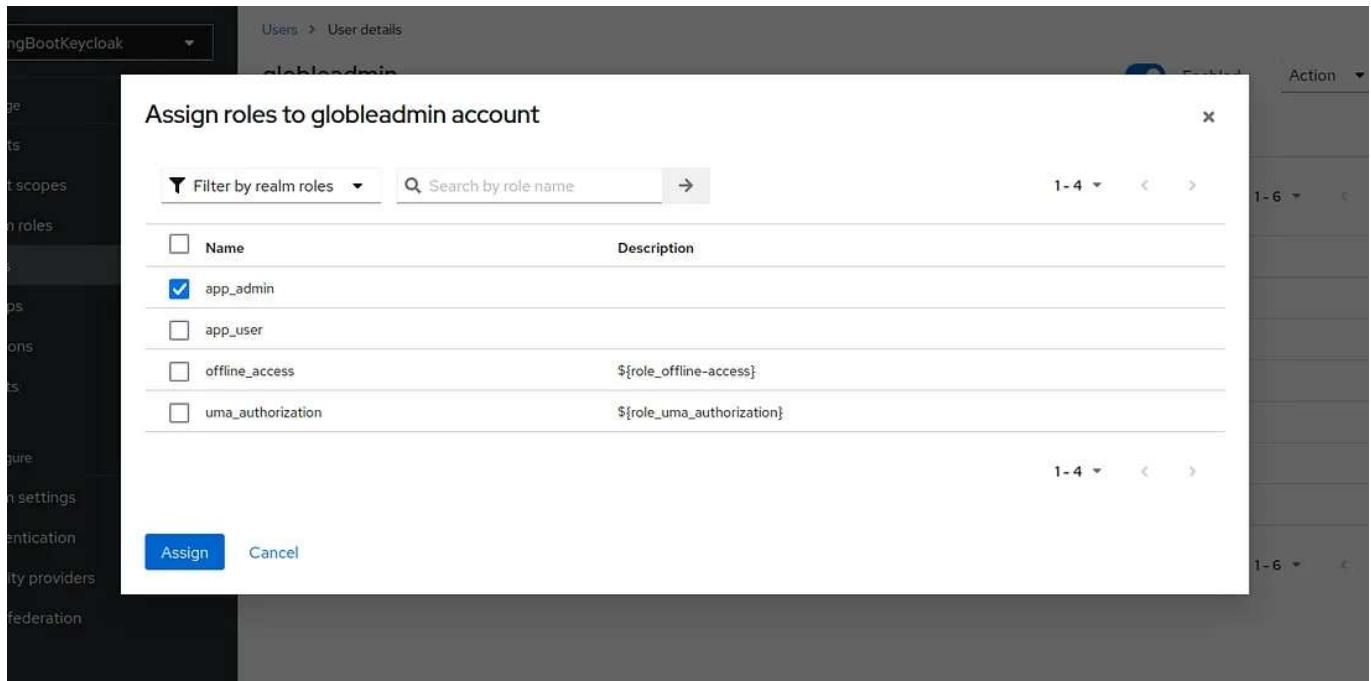
Setting up a user password (Image by author)

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

<input type="checkbox"/>	Name	Inherited	Description
<input type="checkbox"/>	account:view-profile	True	\${role_view-profile}
<input type="checkbox"/>	account:manage-account	True	\${role_manage-account}
<input type="checkbox"/>	account:manage-account-links	True	\${role_manage-account-links}
<input type="checkbox"/>	default-roles-springbootkeycloak	False	\${role_default-roles}
<input type="checkbox"/>	uma_authorization	True	\${role_uma_authorization}
<input type="checkbox"/>	offline_access	True	\${role_offline-access}

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 screenshot shows the "Users" page. On the left is a sidebar with navigation links: Manage, Clients, Client scopes, Realm roles, **Users** (selected), Groups, Sessions, Events, Configure, Realm settings, Authentication, and Identity providers. The main area has a title "Users" and a subtitle "Users are the users in the current realm. Learn more". It features tabs "User list" (selected) and "Permissions". Below is a search bar, an "Add user" button, and a "Delete user" button. A table lists users with columns: Username, Email, Last name, First name, and Status. The users listed are: globleadmin (Email: admin@mail.com, Last name: Admin, First name: Yasar, Status: -), user1 (Email: user1@gmail.com, Last name: Sandeepa, First name: Yasar, Status: -), and user2 (Email: sandun@mail.com, Last name: Fernando, First name: Sandun, Status: -).

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.

```

{
  "issuer": "http://localhost:8080/realm/SpringBootKeycloak",
  "authorization_endpoint": "http://localhost:8080/realm/SpringBootKeycloak/protocol/openid-connect/auth",
  "token_endpoint": "http://localhost:8080/realm/SpringBootKeycloak/protocol/openid-connect/token",
  "introspection_endpoint": "http://localhost:8080/realm/SpringBootKeycloak/protocol/openid-connect/introspect",
  "userinfo_endpoint": "http://localhost:8080/realm/SpringBootKeycloak/protocol/openid-connect/userinfo",
  "end_session_endpoint": "http://localhost:8080/realm/SpringBootKeycloak/protocol/openid-connect/logout",
  "frontchannel_logout_session_supported": true,
  "frontchannel_logout_supported": true,
  "jwks_uri": "http://localhost:8080/realm/SpringBootKeycloak/protocol/openid-connect/certs",
  "check_session_iframe": "http://localhost:8080/realm/SpringBootKeycloak/protocol/openid-connect/login-status-iframe.html",
  "grant_types_supported": [
    "authorization_code",
    "implicit",
    "refresh_token",
    "password",
    "client_credentials",
    "urn:ietf:params:oauth:grant-type:device_code",
    "urn:openid:params:grant-type:ciba"
  ],
  "acr_values_supported": [
    "0",
    "1"
  ],
  "response_types_supported": [
    "code",
    "none",
    "id_token",
    "token",
    "id_token token",
    "code id_token",
    "code token",
    "code id_token token"
  ],
  "subject_types_supported": [
    "public",
    "pairwise"
  ]
}

```

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/realm/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.

POST <http://localhost:8080/realms/SpringBootKeycloak/protocol/openid-connect/token>

**Body** (9) **Body** **Pre-request Script** **Tests** **Settings** **Cookies**

KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/> grant_type	password			
<input checked="" type="checkbox"/> client_id	springboot-keycloak-client			
<input checked="" type="checkbox"/> username	user1			
<input checked="" type="checkbox"/> password	password			

Key Value Description

**Body** **Cookies** **Headers (11)** **Test Results** Status: 200 OK Time: 1825 ms Size: 2.92 KB Save Response

Pretty Raw Preview Visualize JSON

```

1 "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCIgOiAiSldUiIwia2lKIiA6ICJOUXVYRGRVSGRzbVZiSGZQVWpDZ19WdmpBU2ZsM3JRbThjVzFRVV4ZWdNIn0.
eyJleHAiOjE2NzU1NjU0MzYsImIhdCI6MTY3NTU2NTezNiwianRpIjo1NTA2NTN1NTetMGE2MS00NzY0LWJmNmQt0WNjOTQ0YTk5MjIyIiwiiaXNzIjoiaHR0cDovL2xvY2FsaG9
zdDo4MDgwL3J1YWxty9CtHJpbmdCb290S2V5Y2xvYWsilCJhdwQiOjIjH2NvdW50Iiwiic3ViIjoiZWVvMTg4MTUtm2Q2Ny00MTkxLThjZDEtZmiyMTliMmJiYTM1IiwidHlwIj
oiqmVhcmVylIiwiYXpwIjoiic3Byaw5nYm9vdC1zX1jb61bnq0iLCjzZXNza9uX3N0YXR1IjoiNjk3N212NzctNzca4YS00NjJ1LWFjMjTktYTh1Njg1ZjciZmI4IiwiY
WNyIjoiMSIsInJ1YwxtX2FjY2VzcycI6eyJyb2xlcycI6WyJhcHBFdXN1ciIsImR1ZmF1bHQtc9sZXMtc3Byaw5nYm9vdGt1ewNsB2FrIiwb2ZmbGluZV9hY2N1c3MiLCJ1bWFf
YXVoagyaXphdGlvb1dfSwicmVzb3VyY2VfYWNjZXNzIjp7InNwcmLu22Jvb3Qta2V5Y2xvVWstY2xpZw50Ij7InJvbGVzIjpbInVzZXi1XX0sImfjY291bnQ10nsicm9sZXM
i0lsibWFuYwd1LWFjY291bnQ1LCjtYW5hZ2UtYWNb3VudC1saW5rcyIsInZpZ2XctcHjvZm1sZSJdfX0sInNjb3B1IjoiZw1haWwgCHjvZm1sZSiIsInNpZCI6IjY5NzdiNjc3LT
c30GETNDYyZ51hYzE5LWE4ZTY4NWY3NWZ1OCIsImVtYwlsX3Z1cm1maWVkJp0cnV1LCJuW1IjoiWWFzYXMuU2FuZGV1cGEiLCJwcVmZKJyZwRfdXN1cm5hbWU0iJ1c2Vym
SIg1mdpmVuX25hbwU10iJ2YXNhcylsImZhbw1se9uyW1IjoiU2FuZGV1cGEiLCJ1bWFpbCI6InVzZKJxQGdtYw1sLmNvbSj9.
DwQU1RVbkGFBBPkw-8SRsLVV8pTKLZPxIVKbf0pupzJpjPEug598Nqenqu7RCd1ZukvoWBZUp1bdsMRWiSe64wTpWE2Epc7jAcDduixG9knFQK5fsu8f7V07G2zIecZ2Lpi025r
htstcdczTF1VJTHBnwRb0EEdcv3_8Eq152mEsYX1NyvpKYPLflJ3614PSwW1oTQHoOTHOoo1SvjSGJRsSaF_F3sPrCbs5xL-4fZsgI11nLD_5ndMINIK-J-mEE-pL22oyX1Tzb
nPCZf0wM0jOLjeB5A5icNNn2LWwWkukhDZaN_zr_ACtnSJLwDN23ByTMZ3SWCbA9A1X_SjA",
2
3 "expires_in": 300,
4 "refresh_expires_in": 1800,
5 "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cCIgOiAiSldUiIwia2lKIiA6ICJmZGRmOGNkMii1Mjf1LTQ2YWItoTkxOC0yOTMyMzM5MTJmODQifQ."

```

Testing Keycloak token endpoint (Image by author)

Now you can copy that `access_token` and decode it using the [jwt.io](https://jwt.io) website to see the information inside that.

The screenshot shows the jwt.io interface. On the left, under 'Encoded', is a long string of characters representing a JWT token. On the right, under 'Decoded', is the JSON structure of the token. The 'Header' section shows the algorithm (RS256), type (JWT), and kid (a UUID). The 'Payload' section contains the token's expiration (exp), issued at (iat), and issued to (jti). It also includes the audience (aud), subject (sub), type (typ), and access token (azp). The most interesting part is the 'resource\_access' field, which maps client IDs to roles. Two arrows point to the 'springboot-keycloak-client' entry, which has 'app\_user' and 'user' roles assigned. Other clients listed include 'springboot-keycloak-client' and 'account'.

```

{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "NQuXDDUHdYmVbHfPUjCg_VvjASF13rQm8cW1QaUxegM"
}

{
  "exp": 1675535811,
  "iat": 1675535511,
  "jti": "89fbc6d3-81ef-4bee-9e51-5f800f7fe3af",
  "iss": "http://localhost:8080/realm",
  "aud": "account",
  "sub": "eed18815-3d67-4191-8cd1-fb219b2bba35",
  "typ": "Bearer",
  "azp": "springboot-keycloak-client",
  "session_state": "39817291-4c51-4825-ac06-41e2ef05d6de",
  "acr": "1",
  "realm_access": {
    "roles": [
      "app_user", ←
      "default-roles-springbootkeycloak",
      "offline_access",
      "uma_authorization"
    ]
  },
  "resource_access": {
    "springboot-keycloak-client": { ←
      "roles": [
        "user"
      ]
    },
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "email profile"
}

```

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 shows 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.

Groups

Sessions

Events

Configure

**Realm settings**

Authentication

Identity providers

User federation

## Access tokens

Access Token Lifespan

Minutes

It is recommended for this value to be shorter than the SSO session idle timeout: 30 minutes

Access Token Lifespan

Minutes

For Implicit Flow

Client Login Timeout

Minutes

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`

POST http://localhost:8080/realm/SpringBootKeycloak/protocol/openid-connect/token

Params Authorization Headers (9) Body **Pre-request Script** Tests Settings Cookies

none  form-data  x-www-form-urlencoded  raw  binary  GraphQL

KEY	VALUE	DESCRIPTION	...	Bulk Edit
<input checked="" type="checkbox"/> grant_type	refresh_token			
<input checked="" type="checkbox"/> client_id	springboot-keycloak-client			
<input checked="" type="checkbox"/> refresh_token	eyJhbGciOiJIUzI1NiIsInR5cC1gOiAiSldUiwia2lkIiA6ICJOUXVYRGRVSGRzbVzISGZQVWpDZ19WdmpBU2zsM3JRBThjVzFRYVV4ZWdNIn0.			
<input type="checkbox"/> password	password			
Key	Value	Description		

Body Cookies Headers (9) Test Results Status: 200 OK Time: 54 ms Size: 2.64 KB Save Response

Pretty Raw Preview Visualize JSON

```

1 "access_token": "eyJhbGciOiJSUzI1NiIsInR5cC1gOiAiSldUiwia2lkIiA6ICJOUXVYRGRVSGRzbVzISGZQVWpDZ19WdmpBU2zsM3JRBThjVzFRYVV4ZWdNIn0.
2     eyJleHAiOiE2NzU1NjY0NTAsImhlhdC16MTY3NTU2NjE1M CiwanRpIjoiYTg5YjViZDctMDg5MjY0ONTlmlWxEzNzAtZD1jY2U1YjYTY4IiwiuaXNzIjoiaHR0cDovL2xvY2FsaG9
3     zdDo4MDgwL3J1YwXtcy9TchpbmDcb29052V5Y2xvWls1lCjhdWQj0iJh2Vndw5050iwi c3ViIjoiZWKmtg4MTUM2Q9Ny00MTkxLThjZDtzMzIyMT1iMjY1YTM1IiwidHlwIj
4     oiqVmhcmVyiIwIYXpwIjoiicByaw5nYm9vdC1zX1jbG9hay1jbG1lnQ1lCzZXNzaIu9uX3N0YXR1jioiNjk3N2IzNzctNzc4Ys08Nj1LWFjMktYTh1Njg1Zjc1ZmI4iwbwIY
5     WNjYIjoiMSIsInJ1Ywxt2FjY2VzcI6eyJyb2x1cyI6WyJhcBfdXN1ciisImR1zmF1bHQtcm9sZXm tc3Byaw5nYm9vdGtleWNsb2Friiwb2ZmbGlzUv9hY2NlC3MILCj1ibWff
6     YXV0aG9yaXphdGlvbildfSwicmVzb3VY2vFyWnJzXNzIjp0InwmcmluZ2Jvb3Qta2V5Y2xvYwstY2xpZw50Ijp7InJvbGzVjpbInVzXliXX0sInFjY291bnQoNsicm9sZXm
7     i01sibWfUwYd1LWfjY291bnq0ilCjtYw5hZ2U1YnNj3Bv3Ud1saW5rcyIsInZpXctcHjvZm1sZSjdxOsInnjbs2B1ljIz1whawgcHjVzH1sZsIsInNpZC16IjY5Nzd1NjC3LT
8     c30GETNDYyZS1hYzE5LWE4ZT4NWY3Nz10icIsImVtWysX3Z1clmaWVkj1p0inV1LCljUy11joi1WFWzyXmgU2ZvG1cGe1LcJwcmVmZKJyZwRfIdXN1cm5hbWU10i1j1c2Vym
9     SIsImdpdmwUx25hbWU10i1jZYXNhcyIsImZhbw1seV9uYw11jioi2FuZvg1cGe1LcJ1bwFpbC16InVzXixQGdtYw1sLmNvbSj9.
10    i_4jkXhm7iUw-F3nV7sTr4Hw0AbcIFCnsvXvMT9iUhFqE1Fjgq88JEd230qYTPCg9uFThNEdg2v3n1bpI-W - f6LDKBNz0weULzbJbMny07eKVPqAa2yMK300013pUutQm1Kqt
11    Ic4F1lPwR3w52Ane06_HpQqZgwGq8L736JAfw5K_cbmXg-Q+HiHaMEH1B65zINJ389hMcoczqoeauWgakLNg0039_hCnqN6Xim-v075XBllizKm1mNid0BaqsjjuH_gIn3Gr071
12    hoFwN9X7XJUzmzzP9CFZpGZ31oyqwOp9uY0_gj9k0ZKXGJIP_SHvt2kIwMhgWboHt8RsQsfw",
13
14    "expires_in": 300,
15    "refresh_expires_in": 1800,
16    "refresh_token": "eyJhbGciOiJIUzI1NiIsInR5cC1gOiAiSldUiwia2lkIiA6ICJmZGRm0GNKMi1iMjFlLTQ2YWItoTkxOC0yOTMyMzMSMTJm0DQifQ."
  
```

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.

The screenshot shows the Spring Initializer web interface. On the left, there's a sidebar with a menu icon and the 'spring initializr' logo. The main area has several sections:

- Project**: Options for Project (Gradle - Groovy, Gradle - Kotlin, Maven), Language (Java, Kotlin, Groovy), and Spring Boot version (3.0.3 (SNAPSHOT), 3.0.2, 2.7.9 (SNAPSHOT), 2.7.8).
- Project Metadata**: Fields for Group (com.example), Artifact (springboot-keycloak), Name (springboot-keycloak), Description (Demo project for integrating Keycloak with Spring Boot), Package name (com.example.springboot-keycloak), Packaging (Jar, War), and Java version (19, 17, 11, 8).
- Dependencies**: A button to "ADD DEPENDENCIES..." (CTRL + B). Below it are sections for "Spring Web" (WEB), "Spring Security" (SECURITY), "Lombok" (DEVELOPER TOOLS), and "OAuth2 Resource Server" (SECURITY).
- Bottom Buttons**: "GENERATE" (CTRL + ⌘), "EXPLORE" (CTRL + SPACE), and "SHARE...".

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.

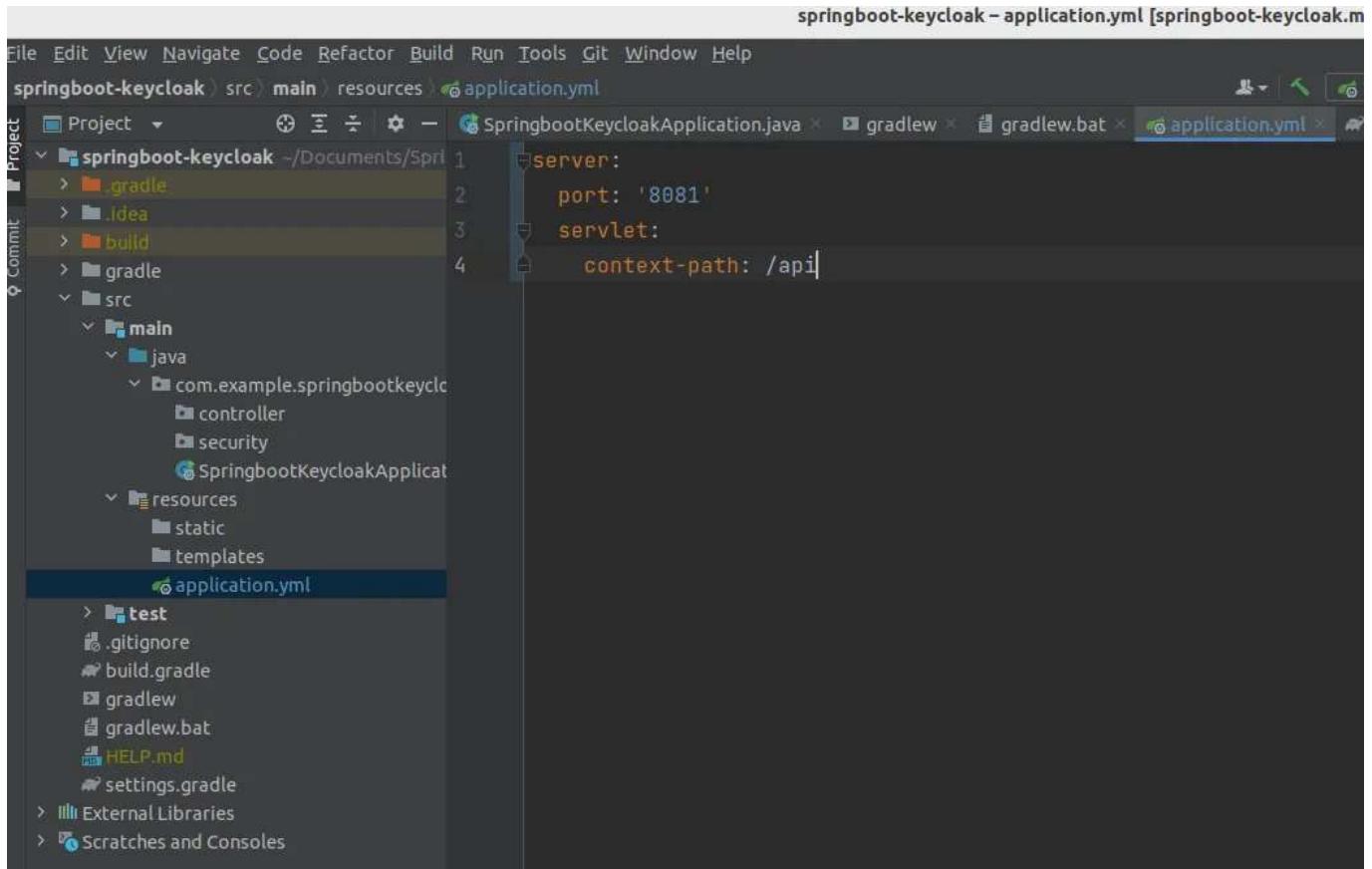
1. `spring-boot-starter-oauth2-client` : You can use this if your app serves UI with Thymeleaf or alike template engines.
2. `spring-boot-starter-oauth2-resource-server` : Use this if the app is a REST API server.

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/realmms/SpringBootKeycloak
          jwk-set-uri: ${spring.security.oauth2.resource-server.jwt.issuer-uri}/protocol/openid-connect/certs
      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 JwtGrantedAutho

    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())) == null
            || (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();
    }

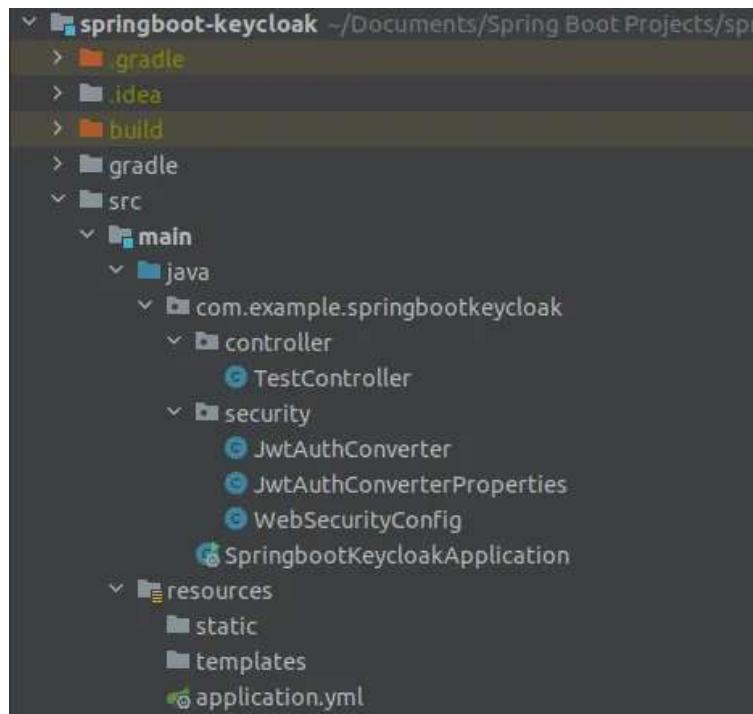
}

```

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 : " + userEmail);
    }

    @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 : " + userEmail);
    }
}
```

We can access the token from `Principal` class. The `principal.getName()` method retrieves the Principal Claim name, which we set as `preferred_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!

```
/home/yasas/.jdks/corretto-17.0.5-1/bin/java ...
.
.
.
:: Spring Boot ::      (v3.0.2)

2023-02-05T10:28:46.756+05:30  INFO 21434 --- [           main] c.e.s.SpringbootKeycloakApplication      : Starting SpringbootKeycloakApplication using Java 17.0.5 with
2023-02-05T10:28:46.761+05:30  INFO 21434 --- [           main] c.e.s.SpringbootKeycloakApplication      : No active profile set, falling back to 1 default profile: "de
2023-02-05T10:28:47.988+05:30  INFO 21434 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat initialized with port(s): 8081 (http)
2023-02-05T10:28:47.994+05:30  INFO 21434 --- [           main] o.apache.catalina.core.StandardService   : Starting service [Tomcat]
2023-02-05T10:28:47.995+05:30  INFO 21434 --- [           main] o.apache.catalina.core.StandardEngine    : Starting Servlet engine: [Apache Tomcat/10.1.5]
2023-02-05T10:28:48.101+05:30  INFO 21434 --- [           main] o.a.c.c.C.[Tomcat].[localhost].[/api]     : Initializing Spring embedded WebApplicationContext
2023-02-05T10:28:48.182+05:30  INFO 21434 --- [           main] w.s.c.ServletWebServerApplicationContext  : Root WebApplicationContext: initialization completed in 1279 ms
2023-02-05T10:28:48.285+05:30  DEBUG 21434 --- [           main] swordEncoderAuthenticationManagerBuilder : No authenticationProviders and no parentAuthenticationManager
2023-02-05T10:28:48.613+05:30  INFO 21434 --- [           main] o.s.s.web.DefaultSecurityFilterChain      : Will secure any request with [org.springframework.security.web
2023-02-05T10:28:48.805+05:30  INFO 21434 --- [           main] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8081 (http) with context path '/ap
2023-02-05T10:28:48.821+05:30  INFO 21434 --- [           main] c.e.s.SpringbootKeycloakApplication      : Started SpringbootKeycloakApplication in 2.673 seconds (process=
```

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.

The screenshot shows a Postman request to `http://localhost:8081/api/test/admin`. The `Authorization` tab is selected, showing a `Bearer Token` type with a token value. The response status is `403 Forbidden` with the message `Invalid grant`.

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.

The screenshot shows the Postman interface with a successful API call. The URL is `http://localhost:8081/api/test/user`. The response body contains:

```

1 Hello User
2 User Name : Yasar Sandeepa
3 User Email : user1@gmail.com

```

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.

The screenshot shows the Postman interface with an unauthorized API call. The URL is `http://localhost:8081/api/test/user`. The response body is empty, and the status bar indicates:

Status: 401 Unauthorized Time: 48 ms Size: 523 B

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.