

Securing Spring Boot REST APIs with Keycloak

M medium.com/devops-dudes/securing-spring-boot-rest-apis-with-keycloak-id760b2004e

April 24, 2020



Overview

Keycloak is an open-source identity and access management solution which makes it easy to secure modern applications and services with little to no code.

Keycloak comes with its own adapters for selected platforms, but it is also possible to use generic OpenID Connect Relying Party and SAML Service Provider libraries. But using the Keycloak Client Adaptors would be much simpler, easy to use and they require less boilerplate code than what is typically required by a library.

The primary focus of this article is to secure Spring Boot REST APIs with Keycloak Spring Boot Adaptor.

To follow through this tutorial, you need to have a running Keycloak instance. If you don't have, follow my previous Medium article.

[Keycloak for Identity and Access Management & High Availability Deployment with Kubernetes](#)

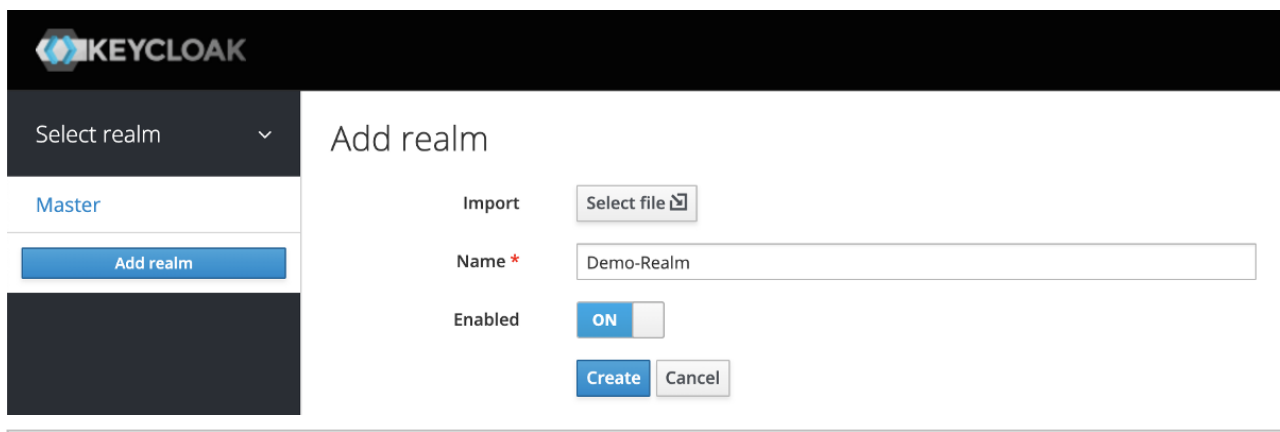
Keycloak Configuration

First, let's make the required configurations in Keycloak.

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.

1. Go to <http://localhost:8080/auth/admin/> and log in to the Keycloak Admin Console using the admin credentials.
2. From the **Master** drop-down menu, click **Add Realm**. When you are logged in to the master realm this drop-down menu lists all existing realms.
3. Type **Demo-Realm** in the **Name** field and click **Create**.

The screenshot shows the Keycloak Admin Console interface. At the top, there's a dark header with the Keycloak logo. Below it, on the left, is a sidebar with a 'Select realm' dropdown menu currently showing 'Master' and an 'Add realm' button. The main area is titled 'Add realm' and contains an 'Import' section with a 'Select file' button, a 'Name' field with the value 'Demo-Realm', an 'Enabled' toggle switch set to 'ON', and 'Create' and 'Cancel' buttons at the bottom.

Add Realm in Keycloak Admin Console

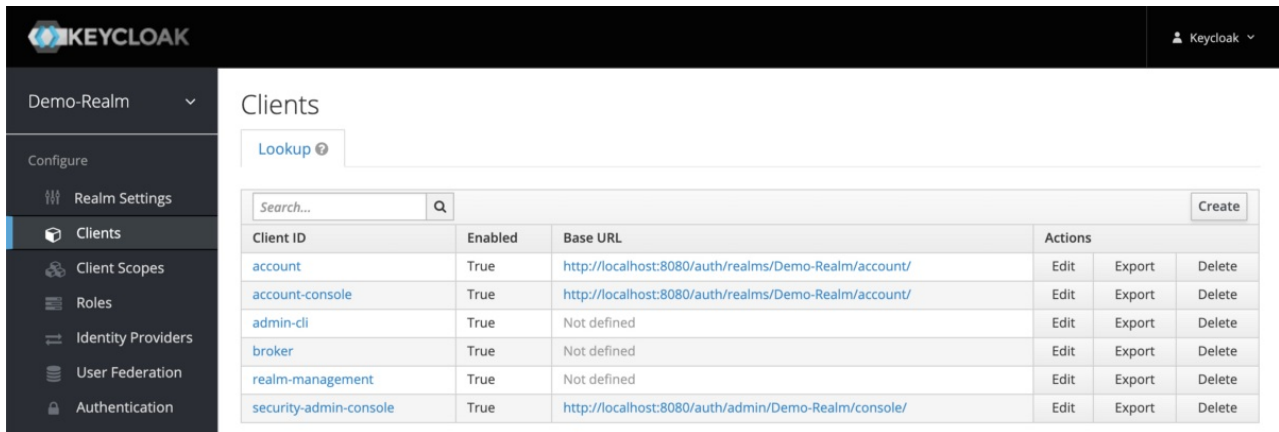
When the realm is created, the main admin console page opens. Notice the current realm is now set to **Demo-Realm**. Switch between managing the **master** realm and the realm you just created by clicking entries in the **Select realm** drop-down menu.

Make sure **Demo-Realm** is selected for the below configurations. Avoid using the master realm. You don't have to create the realm every time. It's a one time process.

Create a Client

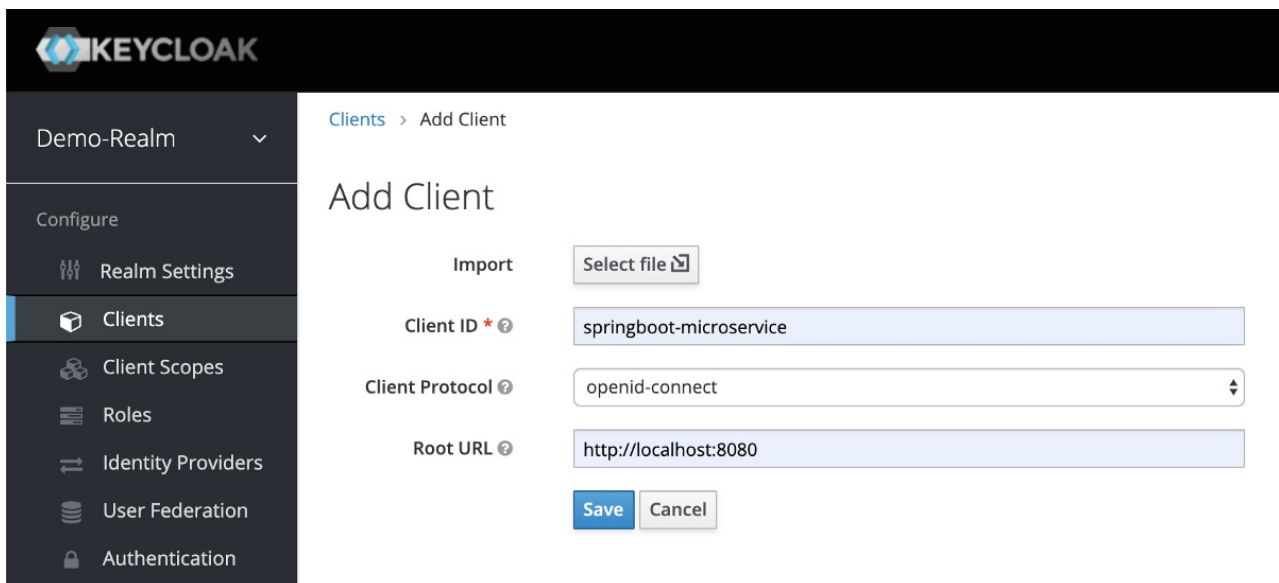
Clients are entities that can request Keycloak to authenticate a user. Most often, clients are applications and services that want to use Keycloak to secure themselves and provide a single sign-on solution. Clients can also be entities that just want to request identity information or an access token so that they can securely invoke other services on the network that are secured by Keycloak.

1. Click on the **Clients** menu from the left pane. All the available clients for the selected Realm will get listed here.



Client Management in Keycloak Admin Console

2. To create a new client, click **Create**. You will be prompted for a **Client ID**, a **Client Protocol** and a **Root URL**. A good choice for the client ID is the name of your application (`springboot-microservice`), the client protocol should be set to `openid-connect` and the root URL should be set to the application URL.



Add Client in Keycloak Admin Console

3. After saving you will be presented with the client configuration page where you can assign a name and description to the client if desired.

Set the **Access Type** to `confidential` , **Authorization Enabled** to `ON` , **Service Account Enabled** to `ON` and click **Save**.

The screenshot shows the Keycloak administration interface for a client named 'springboot-microservice'. The left sidebar contains navigation links for 'Configure' (Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events, Import, Export). The main panel is titled 'Springboot-microservice' and has tabs for 'Settings', 'Roles', 'Client Scopes', 'Mappers', 'Scope', 'Revocation', 'Sessions', 'Offline Access', and 'Installation'. The 'Settings' tab is active, displaying various configuration fields. The 'Access Type' dropdown is open, showing three options: 'confidential' (selected), 'public', and 'bearer-only'. Other visible settings include 'Client ID' (springboot-microservice), 'Name', 'Description', 'Enabled' (ON), 'Consent Required' (OFF), 'Login Theme', 'Client Protocol' (openid-connect), 'Standard Flow Enabled', 'Implicit Flow Enabled' (OFF), 'Direct Access Grants Enabled' (ON), 'Service Accounts Enabled' (ON), 'Authorization Enabled' (ON), 'Root URL' (http://localhost:8080), 'Valid Redirect URIs' (http://localhost:8080/*), 'Base URL', 'Admin URL' (http://localhost:8080), and 'Web Origins' (http://localhost:8080).

Configure client with Access Type: 'confidential'

Credentials tab will show the **Client Secret** which is required for the Spring Boot Application Keycloak configurations.

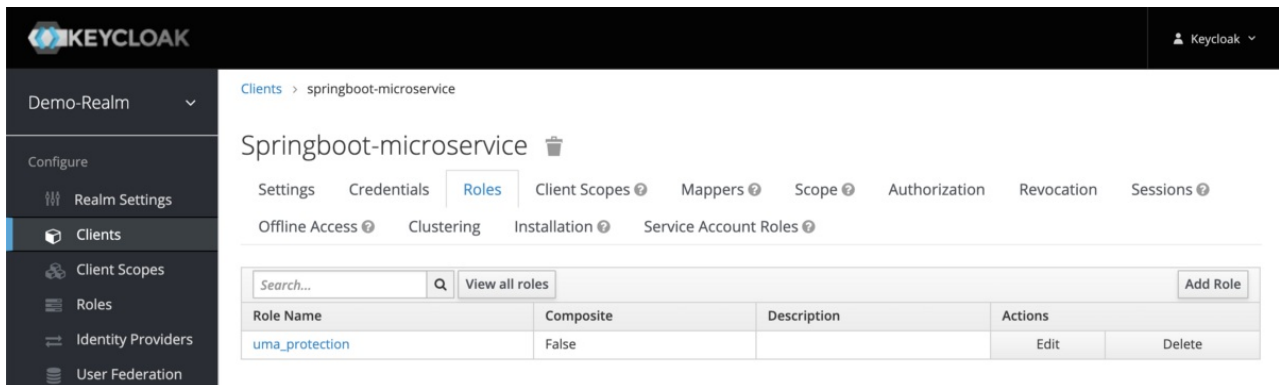
The screenshot shows the 'Credentials' tab for the 'springboot-microservice' client. The left sidebar is the same as in the previous image. The main panel has tabs for 'Settings', 'Credentials', 'Roles', 'Client Scopes', 'Mappers', 'Scope', 'Authorization', and 'Revocation'. The 'Credentials' tab is active, showing the 'Client Authenticator' set to 'Client Id and Secret'. Below this, the 'Secret' is displayed as '0d9fc616-dbd8-44c7-b2cb-12ce' with a 'Regenerate Secret' button. At the bottom, there is a 'Registration access token' field and a 'Regenerate registration access token' button.

Client Credentials Tab

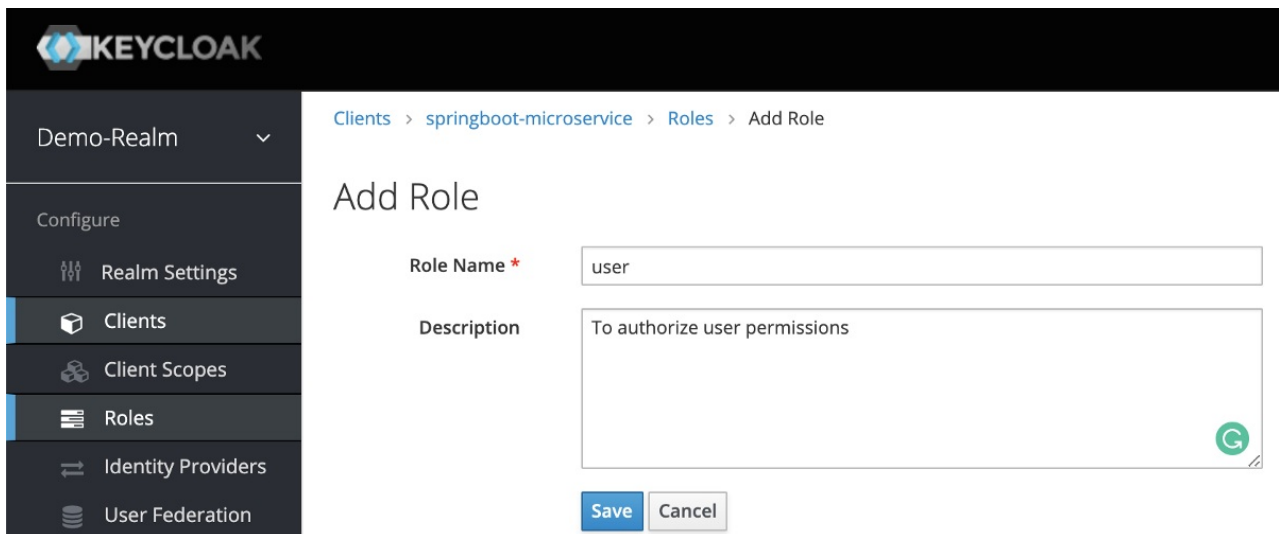
4. Go to **Client Roles** tab to create the `springboot-microservice` role definitions. Imagine the Application that you are building with have different types of users with different user permissions. Ex: users and administrators.

- Some APIs would only be accessible to users only.
- Some APIs would be accessible to administrators only.
- Some APIs would be accessible to both users and administrators.

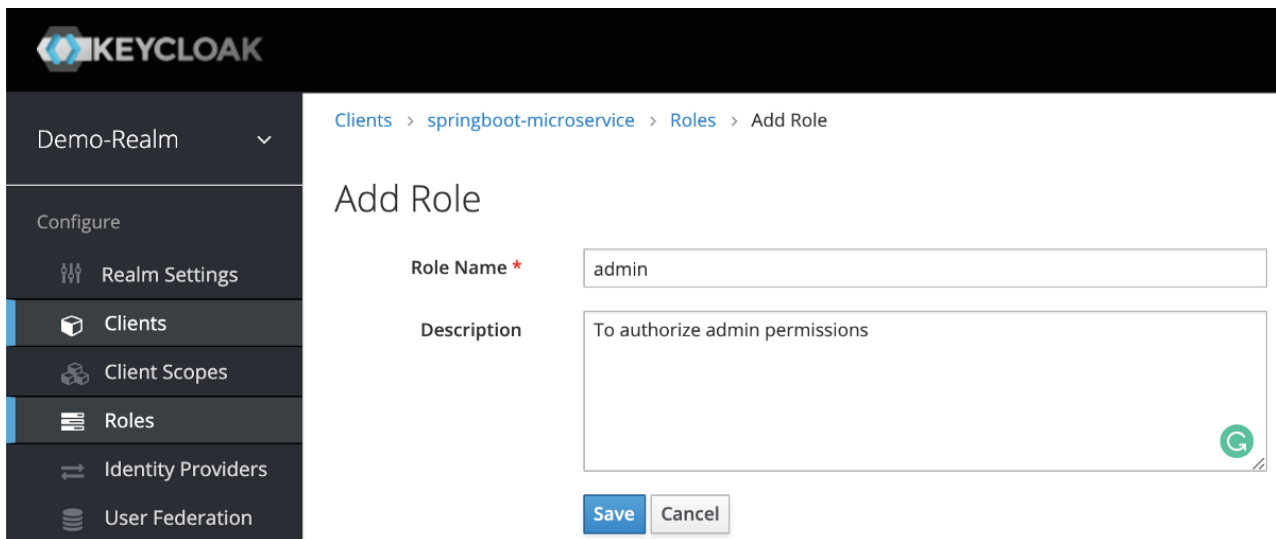
As per the example, let's create two roles: `user` and `admin` by clicking **Add Role** button.



'springboot-microservice' Client Roles

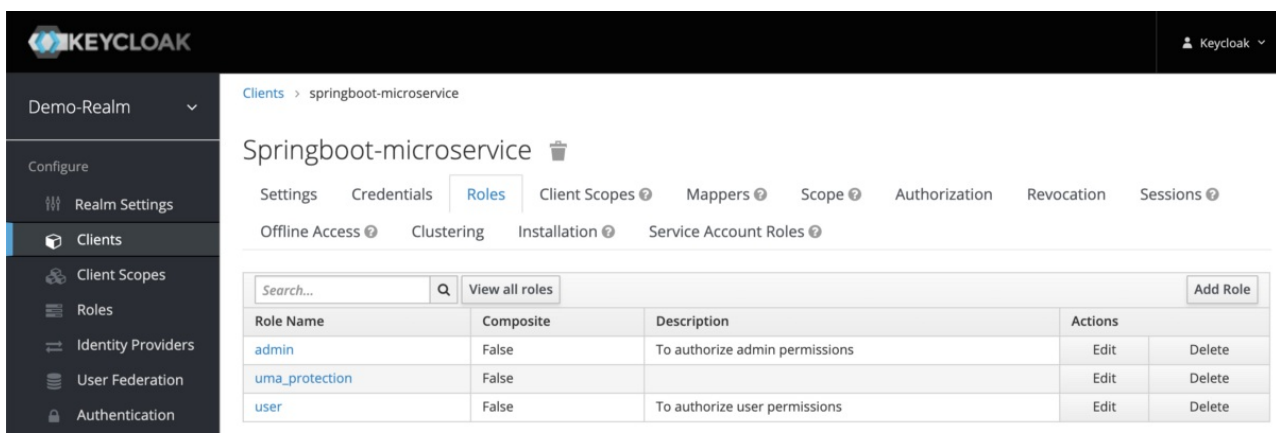


Add 'user' role and Save



The screenshot shows the Keycloak Admin Console interface. On the left is a sidebar with the 'Demo-Realm' dropdown and a 'Configure' menu containing 'Realm Settings', 'Clients', 'Client Scopes', 'Roles', 'Identity Providers', and 'User Federation'. The 'Roles' option is selected. The main area shows the breadcrumb 'Clients > springboot-microservice > Roles > Add Role'. The 'Add Role' form has two fields: 'Role Name' with the value 'admin' and 'Description' with the value 'To authorize admin permissions'. At the bottom are 'Save' and 'Cancel' buttons.

Add 'admin' role and Save



The screenshot shows the Keycloak Admin Console with the 'Roles' tab selected for the 'springboot-microservice' client. The table lists the following roles:

Role Name	Composite	Description	Actions
admin	False	To authorize admin permissions	Edit Delete
uma_protection	False		Edit Delete
user	False	To authorize user permissions	Edit Delete

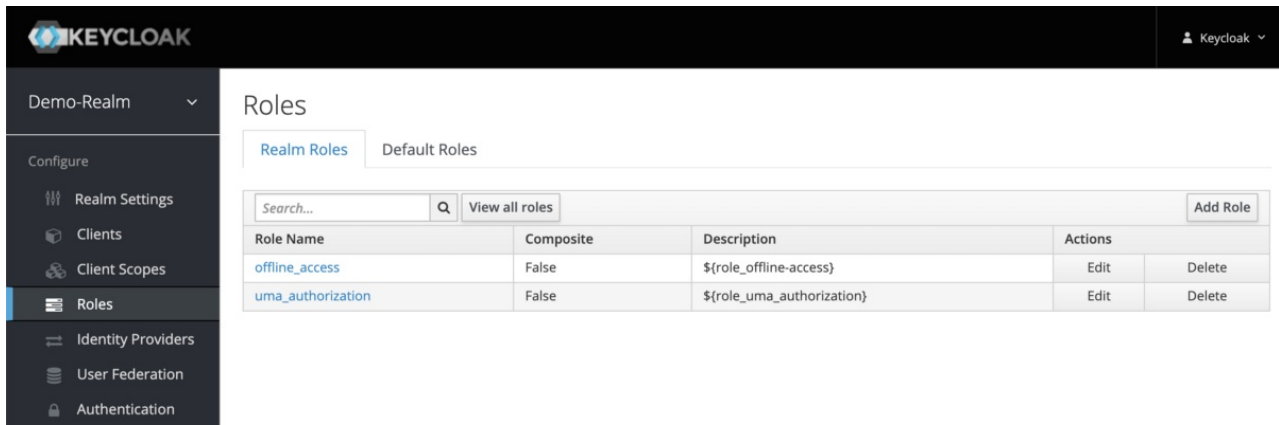
'springboot-microservice' Client Roles after adding 'user', 'admin' roles

Create Realm Roles

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.

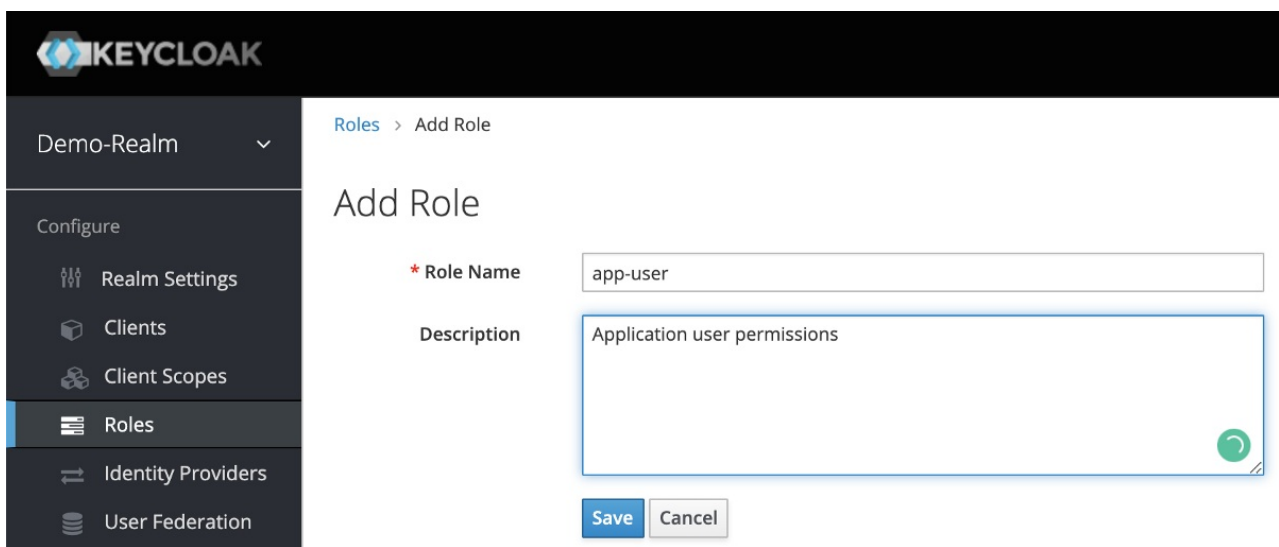
Let's create `app-user` and `app-admin` Realm roles by assigning corresponding `springboot-microservice` roles (`user` , `admin`).

1. Click on the **Roles** menu from the left pane. All the available roles for the selected Realm will get listed here.



Realm Roles in Keycloak Admin Console

2. To create **app-user** realm role, click **Add Role**. You will be prompted for a **Role Name**, and a **Description**. Provide the details as below and **Save**.



Adding 'app-user' Realm Role

After **Save**, enabled **Composite Roles** and Search for **springboot-microservice** under **Client Roles** field. Select **user** role of the **springboot-microservice** and Click **Add Selected >**.

The screenshot shows the Keycloak Admin Console interface. On the left is a dark sidebar with a navigation menu. The top of the sidebar has the 'Demo-Realm' dropdown. Below it are two sections: 'Configure' (containing Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, and Authentication) and 'Manage' (containing Groups, Users, Sessions, Events, Import, and Export). The 'Roles' option in the 'Configure' section is highlighted. The main content area on the right shows the configuration for the 'app-user' role. At the top, there's a breadcrumb 'Roles > app-user'. Below it is the role name 'App-user' with a trash icon. There are three tabs: 'Details' (active), 'Attributes', and 'Users in Role'. The 'Details' tab shows the role name 'app-user' and a description 'Application user permissions'. Below this is a 'Composite Roles' toggle set to 'ON', and 'Save' and 'Cancel' buttons. A section titled 'Composite Roles' is expanded, showing two parts: 'Realm Roles' and 'Client Roles'. Under 'Realm Roles', there are 'Available Roles' (offline_access, uma_authorization) and an empty 'Associated Roles' box. Under 'Client Roles', the 'springboot-microservice' client is selected in a dropdown. Below it, 'Available Roles' (admin, uma_protection) and 'Associated Roles' (user) are shown. The 'user' role is listed in the 'Associated Roles' box.

Assign 'user' Client Role to 'app-user' Realm Role

This configuration will assign `springboot-microservice` `user` client role to the `app-user` realm role. If you have multiple clients with multiple roles, pick and choose the required roles from each client to create realm roles based on the need.

3. Follow the same steps to create the `app-admin` user but assign `admin` client role instead of `user` role.

The screenshot shows the Keycloak Admin Console interface. On the left is a dark sidebar with a menu. The 'Demo-Realm' is selected at the top. Under 'Configure', 'Roles' is highlighted. Under 'Manage', 'Users' is visible. The main content area is titled 'Roles > app-admin'. It shows the 'App-admin' role configuration page with tabs for 'Details', 'Attributes', and 'Users in Role'. The 'Details' tab is active, showing the 'Role Name' as 'app-admin' and the 'Description' as 'Application admin permissions'. There are 'Save' and 'Cancel' buttons. Below this is a section for 'Composite Roles' which is expanded. It contains two sub-sections: 'Realm Roles' and 'Client Roles'. The 'Realm Roles' section has an 'Available Roles' list with 'app-user', 'offline_access', and 'uma_authorization', and an 'Associated Roles' list which is empty. The 'Client Roles' section has a dropdown for 'springboot-microservice', an 'Available Roles' list with 'uma_protection' and 'user', and an 'Associated Roles' list with 'admin' selected.

Assign 'admin' Client Role to 'app-admin' Realm Role

Create Users

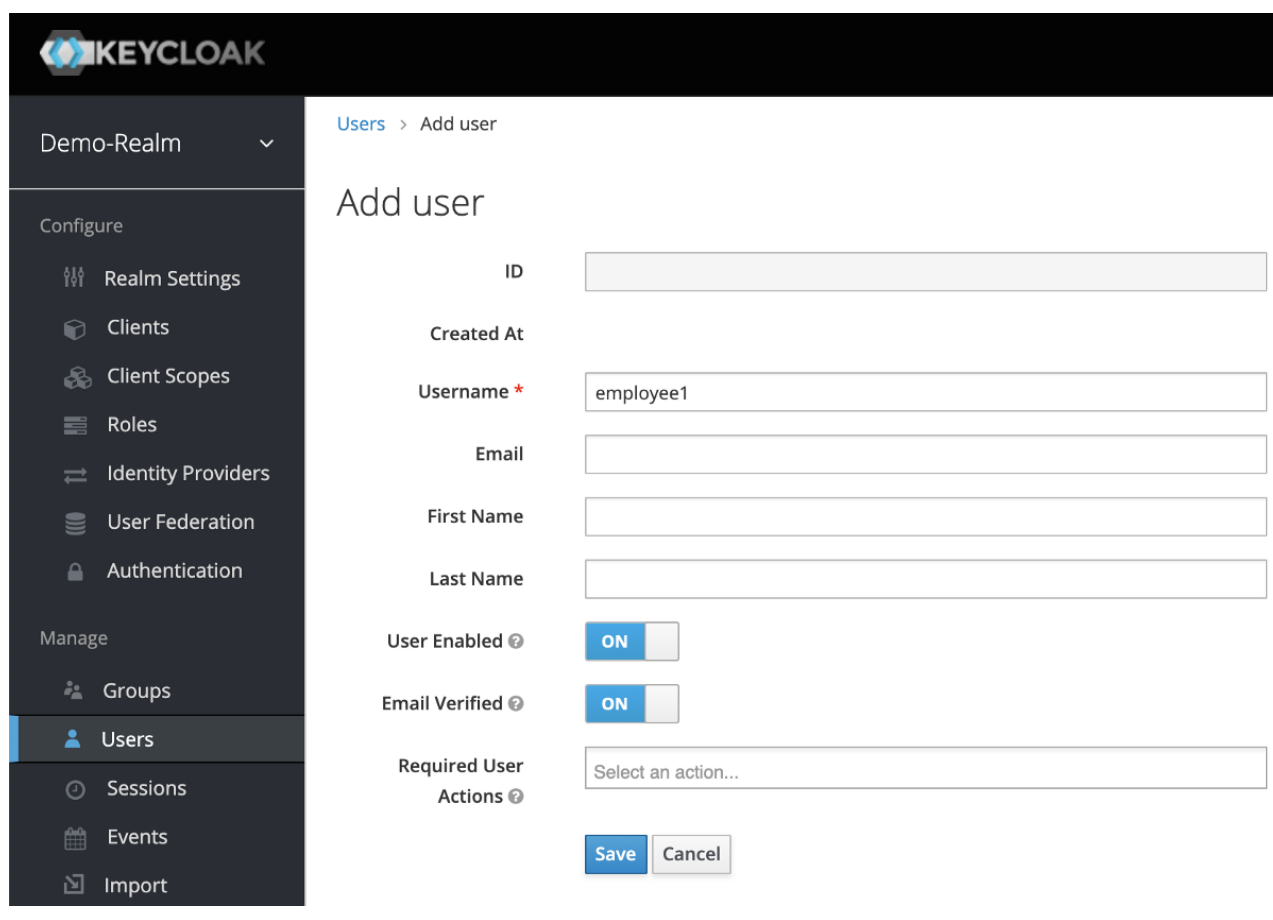
Users are entities that are able to log into your system. They can have attributes associated with themselves like email, username, address, phone number, and birth day. They can be assigned group membership and have specific roles assigned to them.

Let's create following users and grant them `app-user` and `app-admin` roles for testing purposes.

- employee1 with `app-user` realm role
- employee2 with `app-admin` realm role
- employee3 with `app-user` & `app-admin` realm roles

1. From the menu, click **Users** to open the user list page.
2. On the right side of the empty user list, click **Add User** to open the add user page.

3. Enter a name in the **Username** field; this is the only required field. Flip the **Email Verified** switch from **Off** to **On** and click **Save** to save the data and open the management page for the new user.



The screenshot shows the Keycloak administration interface. On the left is a dark sidebar with the 'Demo-Realm' dropdown and a menu with 'Configure' (Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions, Events, Import) sections. The 'Users' option is highlighted. The main content area is titled 'Add user' and contains the following fields and controls:

- ID**: An empty text input field.
- Created At**: A label with no input field.
- Username ***: A text input field containing 'employee1'.
- Email**: An empty text input field.
- First Name**: An empty text input field.
- Last Name**: An empty text input field.
- User Enabled ?**: A toggle switch currently set to 'ON'.
- Email Verified ?**: A toggle switch currently set to 'ON'.
- Required User Actions ?**: A dropdown menu showing 'Select an action...'.
- Buttons**: 'Save' and 'Cancel' buttons at the bottom.

Add New User to 'Demo-Realm'

4. Click the **Credentials** tab to set a temporary password for the new user.
5. Type a new password and confirm it. Flip the **Temporary** switch from **On** to **Off** and click **Reset Password** to set the user password to the new one you specified. For simplicity let's set the password to **mypassword** for all the users.

The screenshot shows the Keycloak administration console for the 'Demo-Realm'. The left sidebar contains navigation options under 'Configure' (Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication) and 'Manage' (Groups, Users, Sessions). The 'Users' option is selected. The main content area shows the 'Employee1' user profile with tabs for Details, Attributes, Credentials (selected), Role Mappings, Groups, Consents, and Sessions. The 'Manage Credentials' section is active, displaying a table with columns: Position, Type, User Label, Data, and Actions. Below the table is the 'Set Password' form, which includes two password input fields (one for the new password and one for confirmation), a 'Temporary' toggle switch set to 'OFF', and a 'Set Password' button.

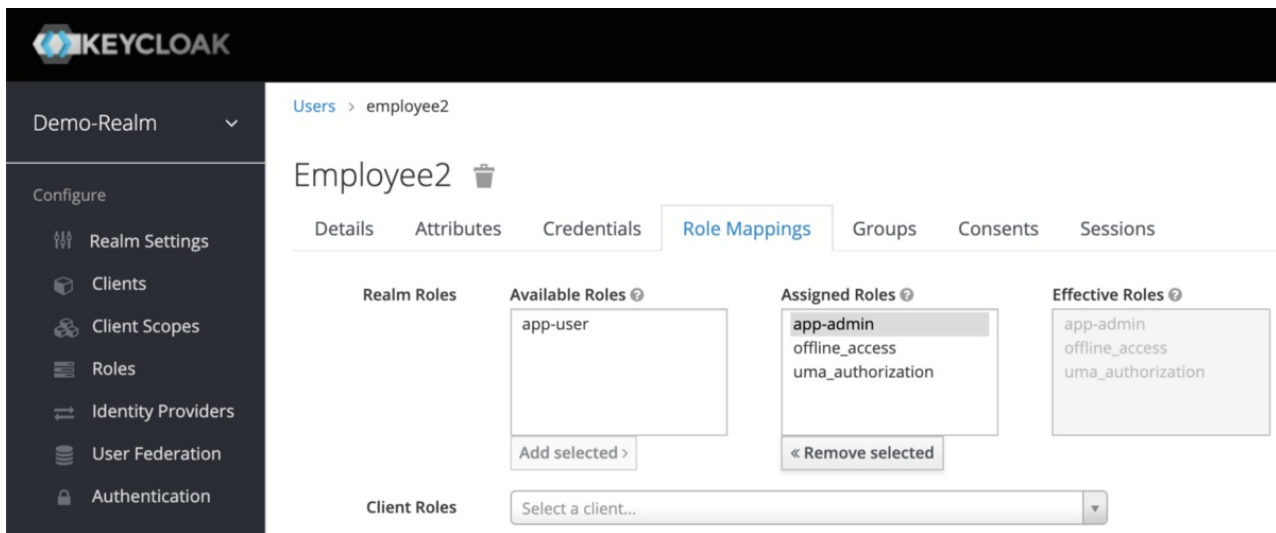
Setting Credentials to Users

6. Click the **Role Mappings** tab to assign realm roles to the user. Realm roles list will be available in **Available Roles** list. Select one required role and click on the **Add Selected >** to assign it to the user.

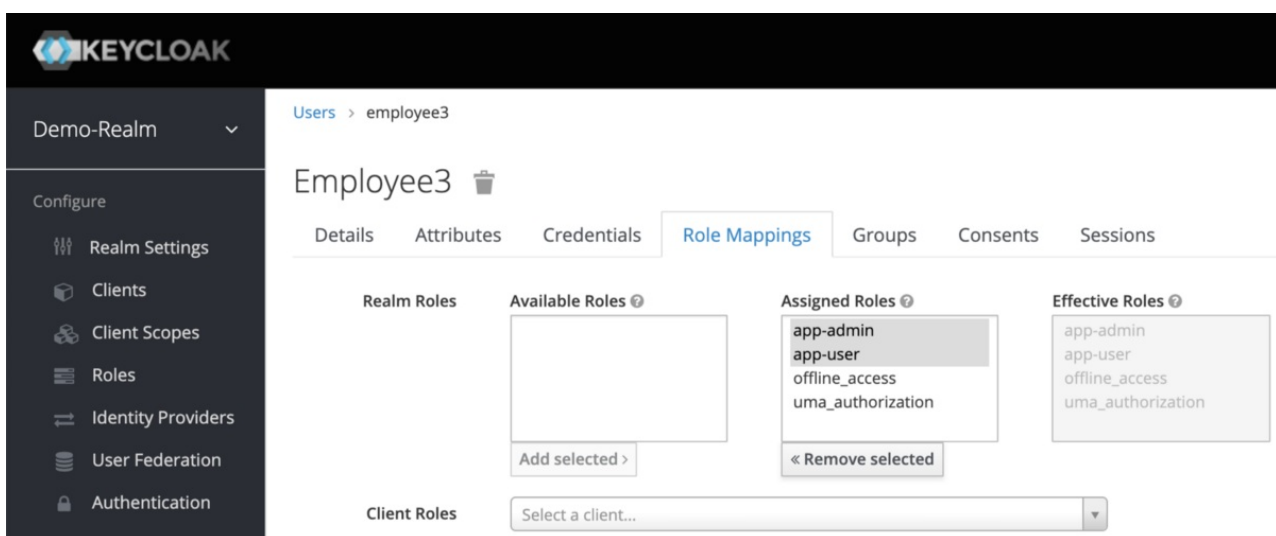
After role assignment, assigned roles will be available under **Assigned Roles** list. Role assignments for `employee1` , `employee2` , and `employee3` would be as below.

The screenshot shows the Keycloak administration console for the 'Demo-Realm', specifically the 'Role Mappings' tab for user 'Employee1'. The interface is divided into four main sections: 'Realm Roles', 'Available Roles', 'Assigned Roles', and 'Effective Roles'. The 'Available Roles' list contains 'app-admin'. The 'Assigned Roles' list contains 'app-user', 'offline_access', and 'uma_authorization'. The 'Effective Roles' list contains 'app-user', 'offline_access', and 'uma_authorization'. Below the 'Available Roles' list is an 'Add selected >' button. Below the 'Assigned Roles' list is a '<< Remove selected' button. At the bottom, there is a 'Client Roles' section with a dropdown menu labeled 'Select a client...'. The left sidebar is identical to the previous screenshot, with 'Users' selected.

`employee1` Role Assignment



`employee2` Role Assignment



`employee3` Role Assignment

Yes, it was a bit of a hassle to go through all the configurations. But when you keep using Keycloak, these configurations will become a piece of cake. For new microservices getting added, you don't need to do all of the above. You just need to add a new client with client roles and assign the client roles to corresponding realm roles.

Generate Tokens

Let's learn how to generate an access token for Keycloak users.

1. Go to **Realm Settings** of the **Demo-Realm** from the left menu and click on **OpenID Endpoint Configuration** to view OpenID Endpoint details.

Demo-Realm

Configure

Realm Settings

Clients

Client Scopes

Roles

Identity Providers

User Federation

Authentication

Groups

Users

Sessions

Demo-Realm

General

Login

Keys

Email

Themes

Cache

Tokens

Client Registration

Security Defenses

Name

Demo-Realm

Display name

HTML Display name

Frontend URL

Enabled

ON

User-Managed Access

OFF

Endpoints

OpenID Endpoint Configuration

SAML 2.0 Identity Provider Metadata

Save

Cancel

Realm Settings of 'Demo-Realm'

Raw

Parsed

```

{
  "issuer": "http://localhost:8080/auth/realms/Demo-Realm",
  "authorization_endpoint": "http://localhost:8080/auth/realms/Demo-Realm/protocol/openid-connect/auth",
  "token_endpoint": "http://localhost:8080/auth/realms/Demo-Realm/protocol/openid-connect/token",
  "token_introspection_endpoint": "http://localhost:8080/auth/realms/Demo-Realm/protocol/openid-connect/token/introspect",
  "userinfo_endpoint": "http://localhost:8080/auth/realms/Demo-Realm/protocol/openid-connect/userinfo",
  "end_session_endpoint": "http://localhost:8080/auth/realms/Demo-Realm/protocol/openid-connect/logout",
  "jwks_uri": "http://localhost:8080/auth/realms/Demo-Realm/protocol/openid-connect/certs",
  "check_session_iframe": "http://localhost:8080/auth/realms/Demo-Realm/protocol/openid-connect/login-status-iframe.html",
  "grant_types_supported": [
    "authorization_code",
    "implicit",
    "refresh_token",
    "password",
    "client_credentials"
  ],
  "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"
  ],
  "id_token_signing_alg_values_supported": [
    "PS384",
    "ES384",
    "RS384",
    "HS256",
    "HS512",
    "ES256",
    "RS256",
    "HS384",
    "ES512",
    "PS256",
    "PS512",
    "RS512"
  ]
}

```

Keycloak Realm OpenID Endpoint Configuration

2. Copy **token_endpoint** from the **OpenID Endpoint Configuration**. URL would look like:

<KEYCLOAK_SERVER_URL>/auth/realms/<REALM_NAME>/protocol/openid-connect/tokenEx:
<http://localhost:8080/auth/realms/Demo-Realm/protocol/openid-connect/token>

3. Use the following CURL command to generate user credentials. Replace **KEYCLOAK_SERVER_URL** , **REALM_NAME** , **CLIENT_ID** , **CLIENT_SECRET** , **USERNAME** , **PASSWORD** with correct values.

```
curl -X POST '<KEYCLOAK_SERVER_URL>/auth/realms/<REALM_NAME>/protocol/openid-connect/token' \  
--header 'Content-Type: application/x-www-form-urlencoded' \  
--data-urlencode 'grant_type=password' \  
--data-urlencode 'client_id=<CLIENT_ID>' \  
--data-urlencode 'client_secret=<CLIENT_SECRET>' \  
--data-urlencode 'username=<USERNAME>' \  
--data-urlencode 'password=<PASSWORD>'
```

Example:

```
curl -X POST 'http://localhost:8080/auth/realms/Demo-Realm/protocol/openid-connect/token' \  
--header 'Content-Type: application/x-www-form-urlencoded' \  
--data-urlencode 'grant_type=password' \  
--data-urlencode 'client_id=springboot-microservice' \  
--data-urlencode 'client_secret=xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxx' \  
--data-urlencode 'username=employee1' \  
--data-urlencode 'password=mypassword'
```

Execute the CURL from Terminal or use Postman. The response would look like below.

[illegible]

Get Token using Postman

Let's decode the **access_token** JWT token issued for `employee1` using <https://jwt.io>.

access_token includes the permission details.

- **realm_access.roles** includes `app_user` realm role.
- **resource_access.springboot-microservice.roles** include the `user` client role.
- **preferred_username** includes the username of the user (`employee1`)

Encoded PASTE A TOKEN HERE

PASTE A TOKEN HERE

[illegible]Decoded EDIT THE PAYLOAD AND SECRET

EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "RS256",
  "typ": "JWT",
  "kid": "MFnIdvayXlUve3gA1Z2604lUJGtK9qW7g8sEKP_tGXg"
}
```

PAYLOAD: DATA

```
{
  "exp": 1587719948,
  "iat": 1587719648,
  "jti": "e54e2fc9-f4d4-49e6-9682-2dca69373eb8",
  "iss": "http://localhost:8080/auth/realms/Demo-Realm",
  "aud": "account",
  "sub": "12a78a46-9a28-4441-8845-ad088be6de50",
  "typ": "Bearer",
  "azp": "springboot-microservice",
  "session_state": "2e3040a9-39d1-4ac6-a3e6-1053dcce1231",
  "acr": "1",
  "allowed-origins": [
    "http://localhost:8080"
  ],
  "realm_access": {
    "roles": [
      "offline_access",
      "uma_authorization",
      "app-user"
    ]
  },
  "resource_access": {
    "springboot-microservice": {
      "roles": [
        "user"
      ]
    },
    "account": {
      "roles": [
        "manage-account",
        "manage-account-links",
        "view-profile"
      ]
    }
  },
  "scope": "profile email",
  "email_verified": true,
  "preferred_username": "employee1"
}
```

VERIFY SIGNATURE

RSASHA256(

iat, **exp** includes the token issued time as well as the token expiry time. Access Token expiry times can be customizable under **Realm Settings, Tokens** tab. By default, **Access Token Lifespan** would be set to 5 minutes which can be customized based on your security requirements.

```
{
  "exp": 1587719948,
  "iat": 1587719648,
```


The screenshot displays the Keycloak Admin Console interface. On the left is a dark sidebar with navigation options: Demo-Realm, Configure, Realm Settings, Clients, Client Scopes, Roles, Identity Providers, User Federation, Authentication, Manage, Groups, Users, Sessions, Events, Import, and Export. The main content area is titled 'Demo-Realm' and contains tabs for General, Login, Keys, Email, Themes, Cache, Tokens (selected), Client Registration, and Security Defenses. The Tokens tab shows various configuration settings for tokens, including Default Signature Algorithm, Revoke Refresh Token (OFF), SSO Session Idle (30 Minutes), SSO Session Max (10 Hours), SSO Session Idle Remember Me (0 Minutes), SSO Session Max Remember Me (0 Minutes), Offline Session Idle (30 Days), Offline Session Max Limited (OFF), Access Token Lifespan (5 Minutes), Access Token Lifespan For Implicit Flow (15 Minutes), Client login timeout (1 Minutes), Login timeout (30 Minutes), Login action timeout (5 Minutes), User-Initiated Action Lifespan (5 Minutes), Default Admin-Initiated Action Lifespan (12 Hours), and Override User-Initiated Action Lifespan (Select one... Minutes). At the bottom are Save and Cancel buttons.

In the testing phase of the Spring Boot Application, use the above steps to generate access tokens for multiple users with corresponding user credentials. Further, if the token expired, generate a new token with the same process.

Spring Boot Application Configuration

Let's build a new Spring Boot application and configure it with Keycloak Spring Boot Adaptor.

Creating the Spring Boot Application

To generate the initial project structure, visit Spring Initializr: <https://start.spring.io/>

Provide details as below.

- Project: Maven Project
- Language: Java
- Spring Boot: Select the latest stable version or keep the default selection as it is.
- Project Metadata: Provide an artifact name and select your preferred Java version. Make sure your local environment has the selected Java version available. If not download and install.
- Dependencies: Add Spring Web, Spring Security and Spring Boot DevTools



Project

- ☒ Maven Project
☐ Gradle Project

Language

- ☒ Java ☐ Kotlin
☐ Groovy

Spring Boot

- ☐ 2.3.0 M4 ☐ 2.3.0 (SNAPSHOT) ☐ 2.2.7 (SNAPSHOT)
☒ 2.2.6 ☐ 2.1.14 (SNAPSHOT) ☐ 2.1.13

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☒ Jar ☐ War

Java ☐ 14 ☒ 11 ☐ 8

Dependencies

ADD DEPENDENCIES... ⌘ + B

Spring Web

WEB

Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Spring Security

SECURITY

Highly customizable authentication and access-control framework for Spring applications.

Spring Boot DevTools

DEVELOPER TOOLS

Provides fast application restarts, LiveReload, and configurations for enhanced development experience.



GENERATE ⌘ + ↵

EXPLORE CTRL + SPACE

SHARE...

Click **Generate** to download the project structure zip bundle and extract it.

Open it in your preferred Java development IDE such as IntelliJ IDEA, Eclipse.

pom.xml Changes

Open pom.xml and make the following changes

1. Add Keycloak Version Property

Find the <properties> section and add <keycloak.version> property. Property values should match with the Keycloak version. In my case **9.0.2**.

```

<properties>
  <java.version>11</java.version>
  <keycloak.version>9.0.2</keycloak.version>
</properties>

```

2. Add Keycloak Dependency

Find the <dependencies> section and add `keycloak-spring-boot-starter` .

```

<dependencies>
  <dependency>
    <groupId>org.keycloak</groupId>
    <artifactId>keycloak-spring-boot-starter</artifactId>
    <version>${keycloak.version}</version>
  </dependency>
  ...
</dependencies>

```

3. Add Dependency Management

Below the <dependencies> section add below section.

```

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.keycloak.bom</groupId>
      <artifactId>keycloak-adaptor-bom</artifactId>
      <version>${keycloak.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

```

Check the updated pom.xml changes here.

application.properties

Open `application.properties` under src/main/resources and provide required Keycloak Configurations.

```

server.port                = 8000

keycloak.realm              = <REALM_NAME>
keycloak.auth-server-url    = <KEYCLOAK_SERVER_URL>/auth
keycloak.ssl-required       = external
keycloak.resource           = <CLIENT_ID>
keycloak.credentials.secret = <CLIENT_SECRET>
keycloak.use-resource-role-mappings = true
keycloak.bearer-only        = true

```

Example

```
server.port                = 8000

keycloak.realm              = Demo-Realm
keycloak.auth-server-url    = http://localhost:8080/auth
keycloak.ssl-required       = external
keycloak.resource           = springboot-microservice
keycloak.credentials.secret = XXXXXXXXXXXXXXXXXXXXXXXXXXXXX
keycloak.use-resource-role-mappings = true
keycloak.bearer-only        = true
```

KeycloakSecurityConfig.java

Keycloak provides a `KeycloakWebSecurityConfigurerAdapter` as a convenient base class for creating a `WebSecurityConfigurer` instance. The implementation allows customization by overriding methods. While its use is not required, it greatly simplifies your security context configuration.

Let's create `KeycloakSecurityConfig.java` in `config` package.

```

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(jsr250Enabled = true)
public class KeycloakSecurityConfig extends KeycloakWebSecurityConfigurerAdapter {

```

```

    @Override
    protected void configure(HttpSecurity http) throws Exception {
    super.configure(http);
        http.authorizeRequests()
            .anyRequest()
            .permitAll();
        http.csrf().disable();
    }

```

```

    @Autowired
    public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
        KeycloakAuthenticationProvider keycloakAuthenticationProvider =
        keycloakAuthenticationProvider();
        keycloakAuthenticationProvider.setGrantedAuthoritiesMapper(new
        SimpleAuthorityMapper());
        auth.authenticationProvider(keycloakAuthenticationProvider);
    }

```

```

    @Bean
    @Override
    protected SessionAuthenticationStrategy sessionAuthenticationStrategy() {
    return new RegisterSessionAuthenticationStrategy(new SessionRegistryImpl());
    }

```

```

    @Bean
    public KeycloakConfigResolver KeycloakConfigResolver() {
        return new KeycloakSpringBootConfigResolver();
    }
}

```

configureGlobal: Registers the KeycloakAuthenticationProvider with the authentication manager.

sessionAuthenticationStrategy: Defines the session authentication strategy.

KeycloakConfigResolver : By Default, the Spring Security Adapter looks for a `keycloak.json` configuration file. You can make sure it looks at the configuration provided by the Spring Boot Adapter by adding this bean

@EnableGlobalMethodSecurity: The *jsr250Enabled* property allows us to use the *@RoleAllowed* annotation. We'll explore more about this annotation in the next section.

TestController.java

We need some dummy APIs to test the API security.

Create `TestController.java` in `controller` package.

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

    @RequestMapping(value = "/anonymous", method = RequestMethod.GET)
    public ResponseEntity<String> getAnonymous() {
        return ResponseEntity.ok("Hello Anonymous");
    }

    @RequestMapping(value = "/user", method = RequestMethod.GET)
    public ResponseEntity<String> getUser() {
        return ResponseEntity.ok("Hello User");
    }

    @RequestMapping(value = "/admin", method = RequestMethod.GET)
    public ResponseEntity<String> getAdmin() {
        return ResponseEntity.ok("Hello Admin");
    }

    @RequestMapping(value = "/all-user", method = RequestMethod.GET)
    public ResponseEntity<String> getAllUser() {
        return ResponseEntity.ok("Hello All User");
    }
}
```

Run the Spring Boot Application. Make sure Maven is installed and configured.

```
mvn spring-boot:run
```

[illegible]

As defined in the `TestController`, let's invoke the REST APIs with CURL one by one.

```
curl -X GET 'http://localhost:8000/test/anonymous'
curl -X GET 'http://localhost:8000/test/user'
curl -X GET 'http://localhost:8000/test/admin'
curl -X GET 'http://localhost:8000/test/all-user'
```

Outputs for each curl would be as below:

```
→ ~ curl -X GET 'http://localhost:8000/test/anonymous'
[Hello Anonymous%
→ ~ curl -X GET 'http://localhost:8000/test/user'
[Hello User%
→ ~ curl -X GET 'http://localhost:8000/test/admin'
[Hello Admin%
→ ~ curl -X GET 'http://localhost:8000/test/all-user'
Hello All User%
```

As you see all APIs don't require any authentication or authorization. Now let's try to secure these API endpoints.

Define Role-Based Access with @RolesAllowed Annotation

Use `@RolesAllowed` annotation can be used to define the allowed user roles.

/test/anonymous:

This API should be accessible without any Authorization token with no restrictions. It already meets our requirements and needs no additional changes.

/test/user:

This API should be accessible to users with `springboot-microservice` `user` role. This can be defined by changing the code as below.

```
@RolesAllowed("user")
@RequestMapping(value = "/user", method = RequestMethod.GET)
public ResponseEntity<String> getUser(@RequestHeader String Authorization) {
    return ResponseEntity.ok("Hello User");
}
```

/test/admin:

This API should be accessible to users with `springboot-microservice` `admin` role. This can be defined by changing the code as below.

```
@RolesAllowed("admin")
@RequestMapping(value = "/admin", method = RequestMethod.GET)
public ResponseEntity<String> getAdmin(@RequestHeader String Authorization) {
    return ResponseEntity.ok("Hello Admin");
}
```

/test/all-user:

This API should be accessible to users with `springboot-microservice` `user` & `admin` roles. This can be defined by changing the code as below.

```
@RolesAllowed({ "admin", "user" })
@RequestMapping(value = "/all-user", method = RequestMethod.GET)
public ResponseEntity<String> getAllUser(@RequestHeader String Authorization) {
    return ResponseEntity.ok("Hello All User");
}
```

Now TestController would look like below.


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

```
    @RequestMapping(value = "/anonymous", method = RequestMethod.GET)
    public ResponseEntity<String> getAnonymous() {
    return ResponseEntity.ok("Hello Anonymous");
    }
```

```
    @RolesAllowed("user")
    @RequestMapping(value = "/user", method = RequestMethod.GET)
    public ResponseEntity<String> getUser(@RequestHeader String Authorization) {
    return ResponseEntity.ok("Hello User");
    }
```

```
    @RolesAllowed("admin")
    @RequestMapping(value = "/admin", method = RequestMethod.GET)
    public ResponseEntity<String> getAdmin(@RequestHeader String Authorization) {
    return ResponseEntity.ok("Hello Admin");
    }
```

```
    @RolesAllowed({ "admin", "user" })
    @RequestMapping(value = "/all-user", method = RequestMethod.GET)
    public ResponseEntity<String> getAllUser(@RequestHeader String Authorization) {
    return ResponseEntity.ok("Hello All User");
    }

}
```

Restart the Spring Boot Application and test above APIs by passing tokens from `employee1` , `employee2` , `employee3` access tokens in the `Authorization` header with the `bearer` prefix (`bearer <ACCESS_TOKEN>`).

```

curl -X GET 'http://localhost:8000/test/user' \
--header 'Authorization: bearer <ACCESS_TOKEN>'Outputs:
anonymous: 403 Forbidden
employee1: Hello User
employee2: 403 Forbidden
employee3: Hello Usercurl -X GET 'http://localhost:8000/test/admin' \
--header 'Authorization: bearer <ACCESS_TOKEN>'Outputs:
anonymous: 403 Forbidden
employee1: 403 Forbidden
employee2: Hello Admin
employee3: Hello Admincurl -X GET 'http://localhost:8000/test/all-user' \
--header 'Authorization: bearer <ACCESS_TOKEN>'Outputs:
anonymous: 403 Forbidden
employee1: Hello All User
employee2: Hello All User
employee3: Hello All User

```

If the token is expired, you will receive 401 Unauthorized error.

Define Role-Based Access with Security Configuration

Rather than using `@RolesAllowed` annotation, the same configuration can be made in `KeycloakSecurityConfig` class as below.

```

@Override
protected void configure(HttpSecurity http) throws Exception {
    super.configure(http);
    http.authorizeRequests()
        .antMatchers("/test/anonymous").permitAll()
        .antMatchers("/test/user").hasAnyRole("user")
        .antMatchers("/test/admin").hasAnyRole("admin")
        .antMatchers("/test/all-user").hasAnyRole("user", "admin")
        .anyRequest()
        .permitAll();
    http.csrf().disable();
}

```

Hope you enjoyed the article. Final code can be found [here](#).

In the next article let's discuss Securing Node.js based REST APIs with Keycloak.