**Securing Spring Boot REST APIs with Keycloak**

**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](https://medium.com/@ddezoysa/keycloak-for-identity-and-access-management-9860a994bf0)

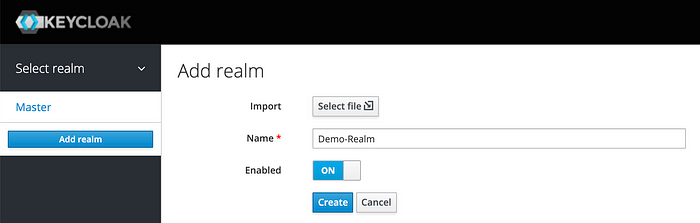
**Keycloak Configuration**

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**.



Add Realm in Keycloak Admin Console

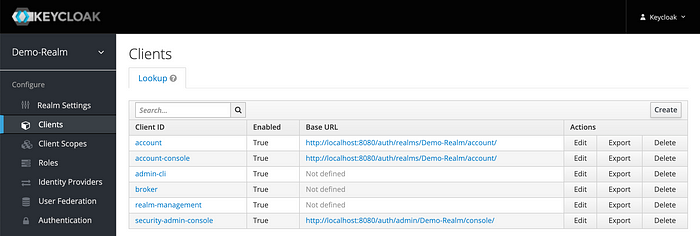
When the realm is created, the main admin console page opens. Notice the current realm is now set to Demo-Realm. Switch between managing the master realm and the realm you just created by clicking entries in the **Select realm** drop-down menu.

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**

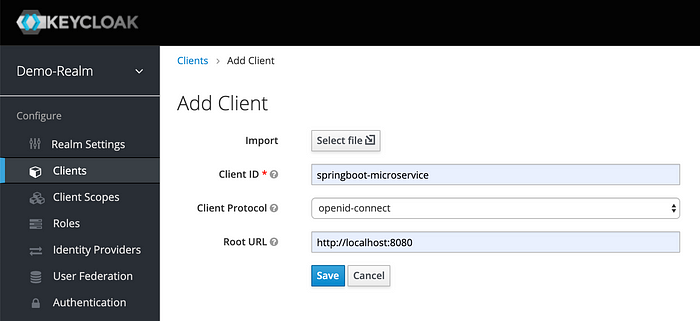
Clients are entities that can request Keycloak to authenticate a user. Most often, clients are applications and services that want to use Keycloak to secure themselves and provide a single sign-on solution. Clients can also be entities that just want to request identity information or an access token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

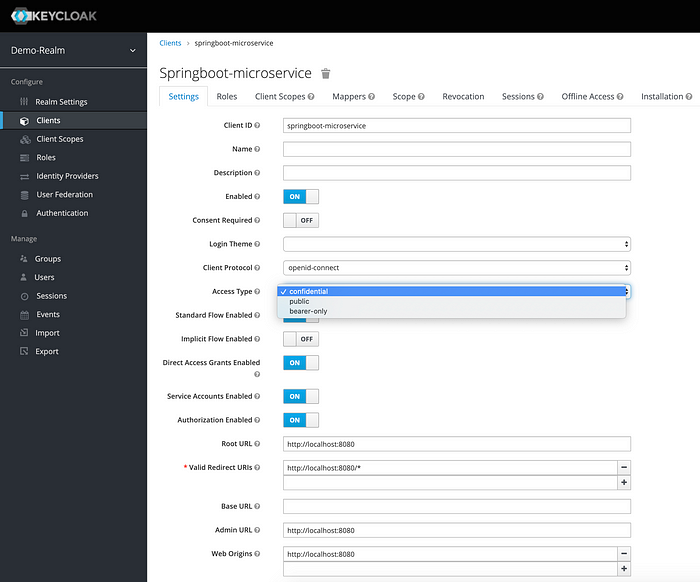
2. To create a new client, click **Create**. You will be prompted for a **Client ID**, a **Client Protocol** and a **Root URL**. A good choice for the client ID is the name of your application (springboot-microservice), the client protocol should be set to openid-connectand the root URL should be set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



Add Client in Keycloak Admin Console

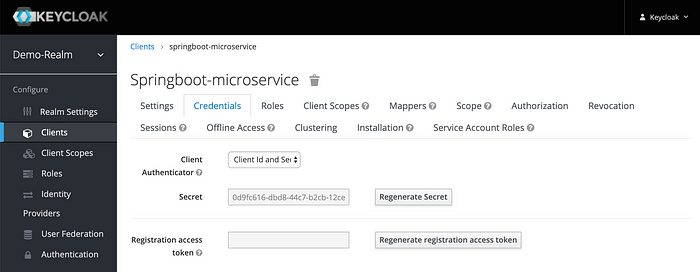
3. After saving you will be presented with the client configuration page where you can assign a name and description to the client if desired.

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



Configure client with Access Type: ‘confidential’

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

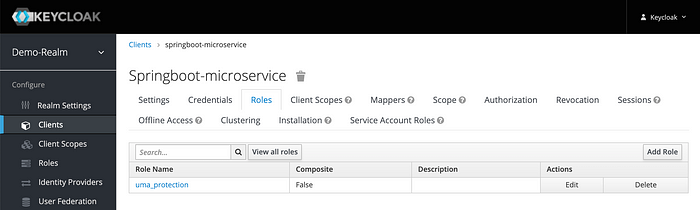


Client Credentials Tab

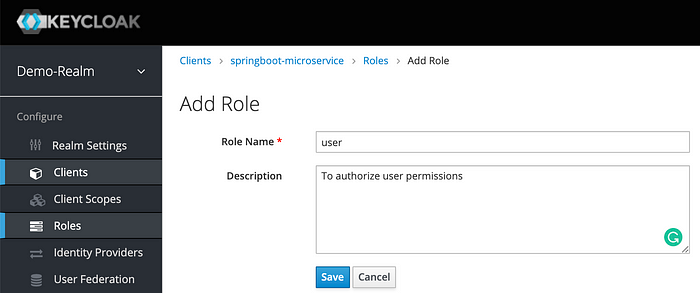
4. Go to **Client Roles**tab to create the springboot-microservice role definitions. Imagine the Application that you are building with have different types of users with different user permissions. Ex: users and administrators.

* 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.

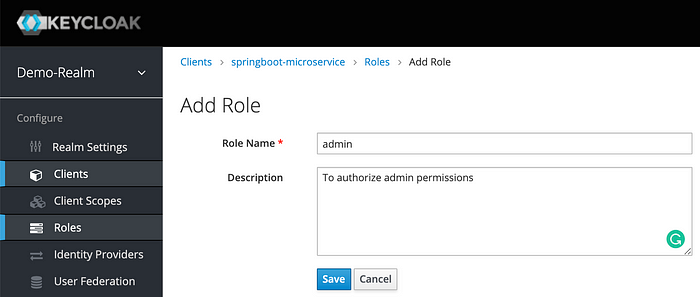
As per the example, let’s create two roles: user and adminby clicking **Add Role** button.



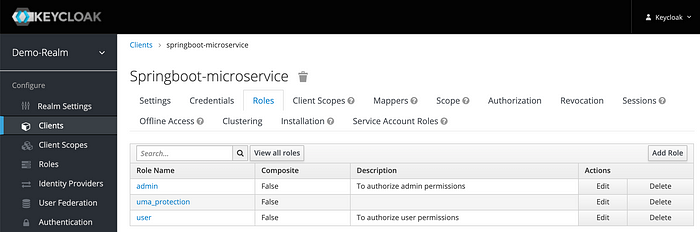
‘springboot-microservice’ Client Roles



Add ‘user’ role and Save



Add ‘admin’ role and Save



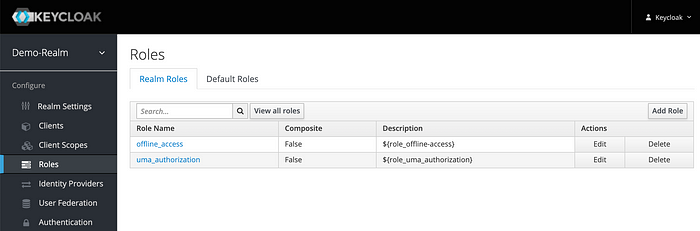
‘springboot-microservice’ Client Roles after adding ‘user’, ‘admin’ roles

**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.

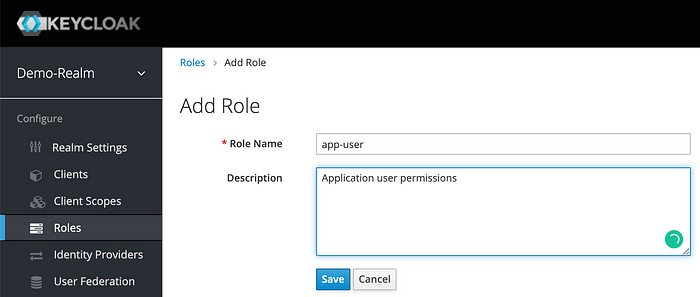
Let’s create app-user and app-admin Realm roles by assigning correspondingspringboot-microservice roles (user, admin).

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



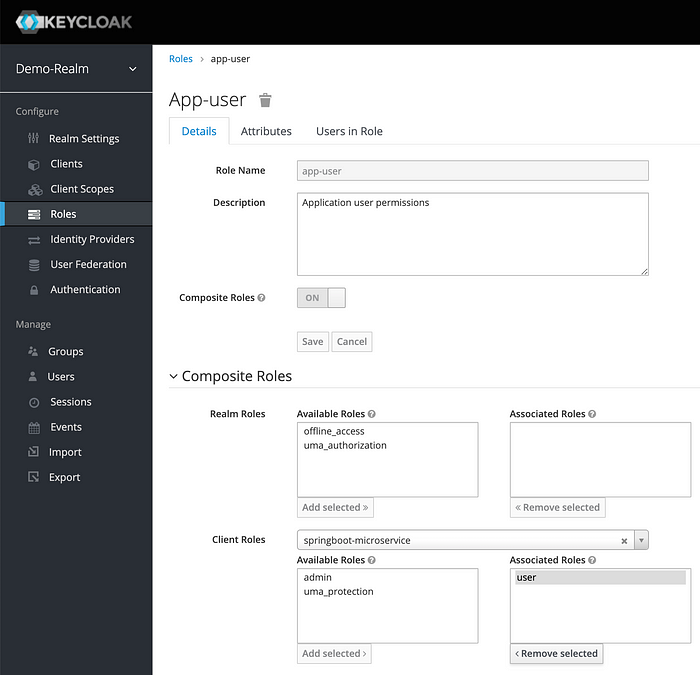
Realm Roles in Keycloak Admin Console

2. To createapp-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

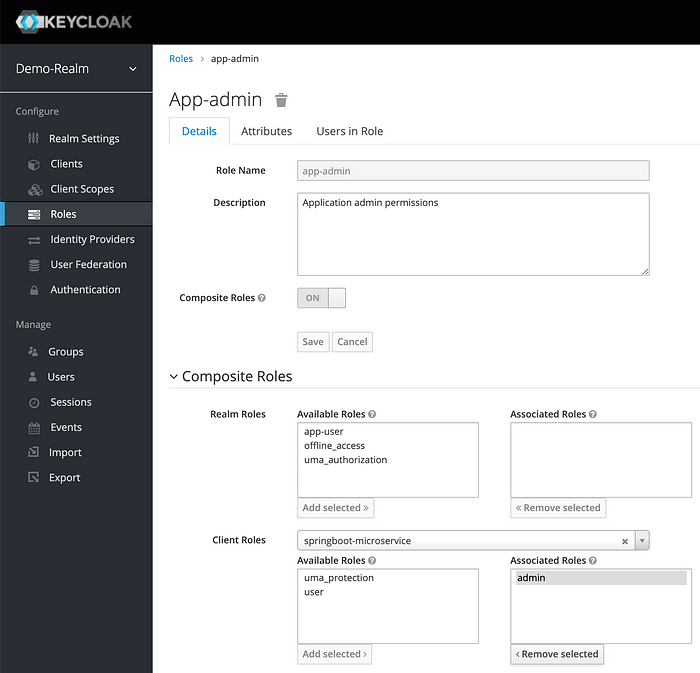
After **Save**, enabled **Composite Roles**and Search for springboot-microservice under **Client Roles** field. Select user role of the springboot-microservice and Click **Add Selected >**.



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.



Assign ‘admin’ Client Role to ‘app-admin’ Realm Role

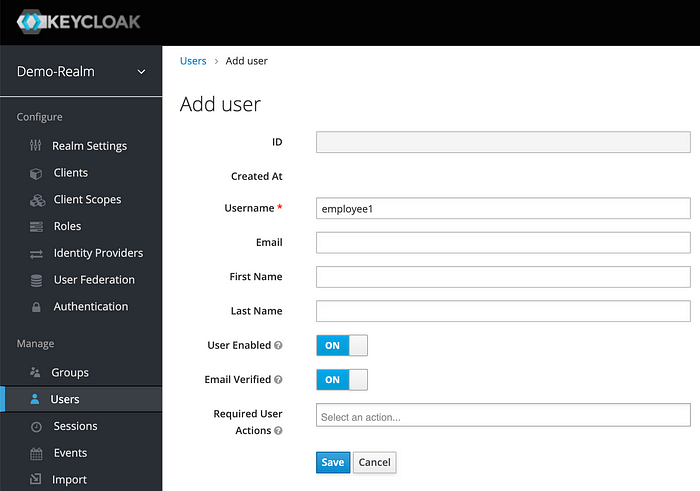
**Create Users**

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-userrealm role
* employee2 with app-adminrealm role
* employee3 with app-user& app-adminrealm roles

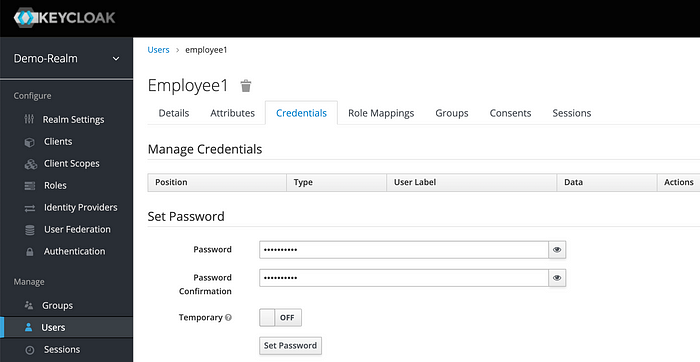
1. From the menu, click **Users** to open the user list page.
2. On the right side of the empty user list, click **Add User** to open the add user page.
3. Enter a name in the Username field; this is the only required field. Flip the **Email Verified** switch from **Off** to **On** and click **Save** to save the data and open the management page for the new user.



Add New User to ‘Demo-Realm‘

4. Click the **Credentials** tab to set a temporary password for the new user.

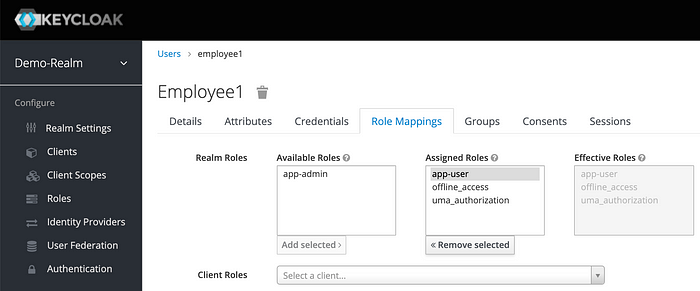
5. Type a new password and confirm it. Flip the **Temporary** switch from **On to**Off and click **Reset Password** to set the user password to the new one you specified. For simplicity let’s set the password to mypassword for all the users.



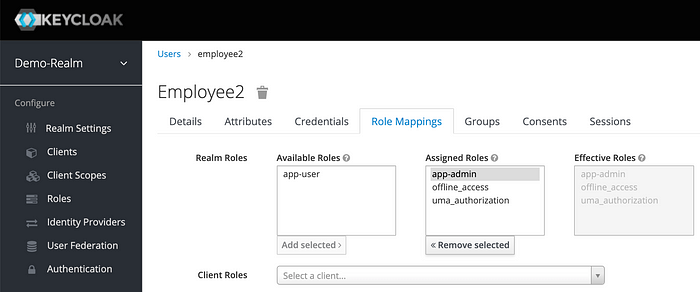
Setting Credentials to Users

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.

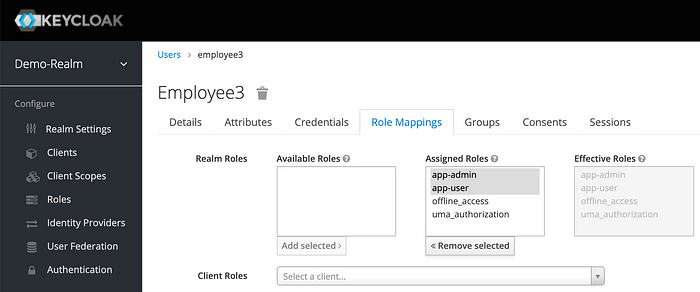
After role assignment, assigned roles will be available under **Assigned Roles** list. Role assignments for employee1, employee2, and employee3 would be as below.



`employee1` Role Assignment



`employee2` Role Assignment



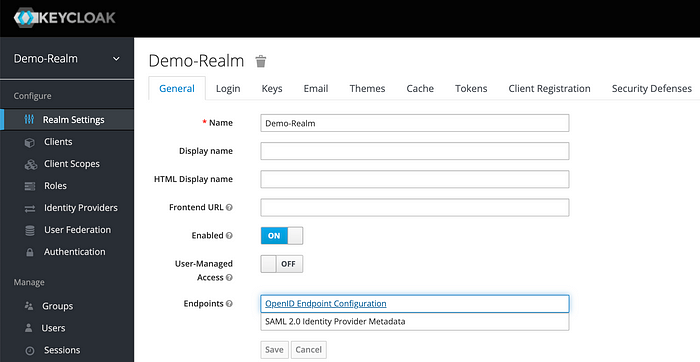
`employee3` Role Assignment

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.

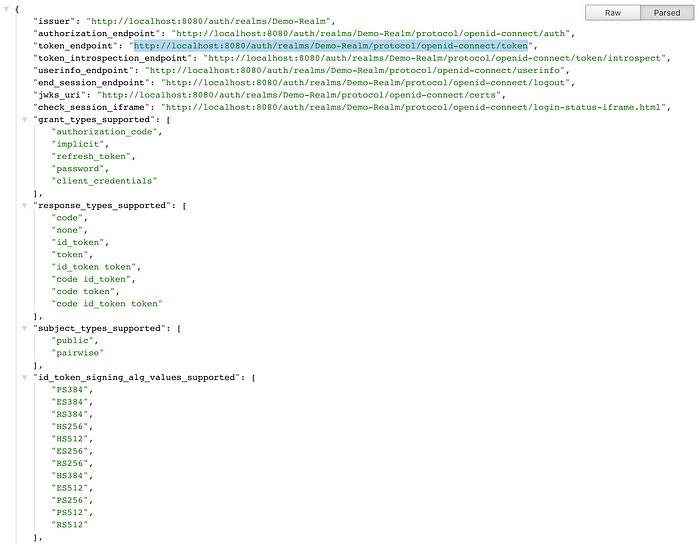
**Generate Tokens**

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.



Realm Settings of ‘Demo-Realm’



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>

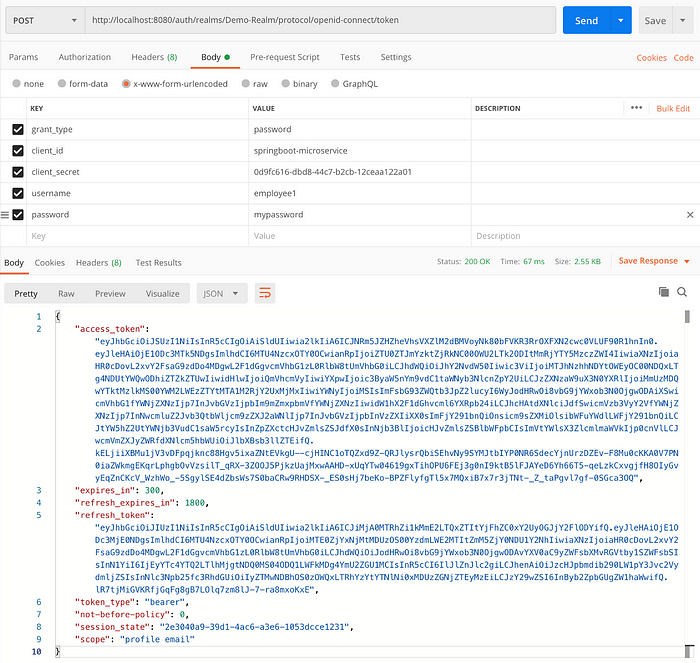
3. Use the following CURL command to generate user credentials. Replace KEYCLOAK\_SERVER\_URL, REALM\_NAME, CLIENT\_ID, CLIENT\_SECRET, USERNAME, PASSWORD with correct values.

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.

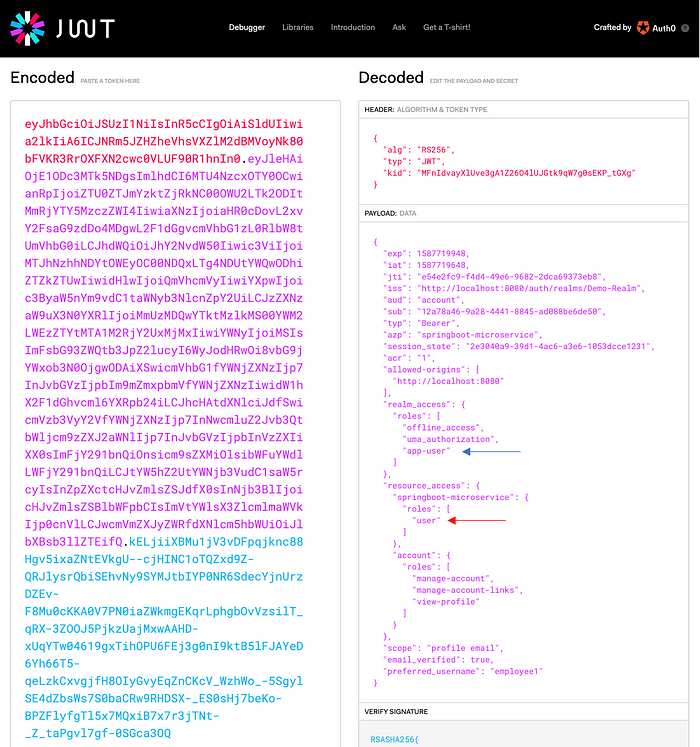


Get Token using Postman

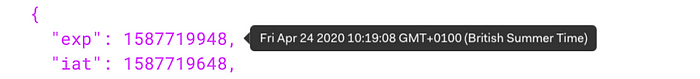
Let’s decode the **access\_token** JWT token issued for employee1 using [https://jwt.io](https://jwt.io/).

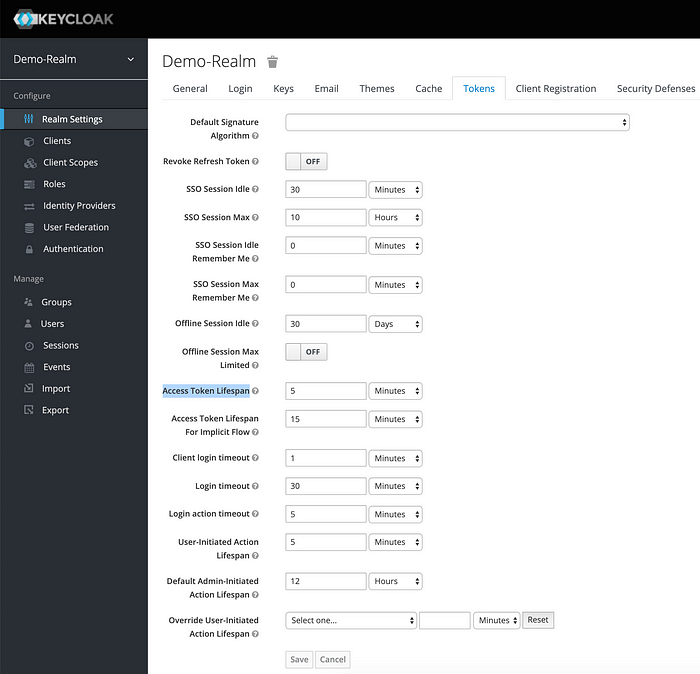
**access\_token**includes the permission details.

* **realm\_access.roles** includes app\_user realm role.
* **resource\_access.springboot-microservice.roles** include the userclient role.
* **preferred\_username** includes the username of the user (employee1)



* **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.





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**

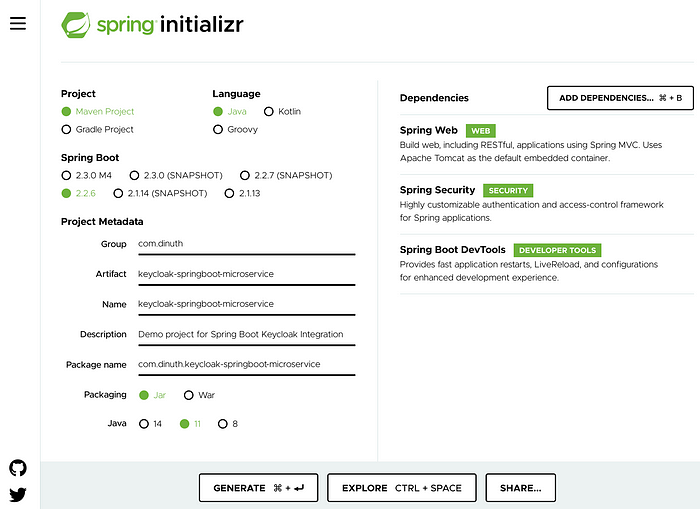
Let’s build a new Spring Boot application and configure it with Keycloak Spring Boot Adaptor.

**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



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-adapter-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 = XXXXXXXXXXXXXXXXXXXXXXXXX  
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](https://docs.spring.io/spring-security/site/docs/4.0.x/apidocs/org/springframework/security/config/annotation/web/WebSecurityConfigurer.html" \t "_blank) 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.

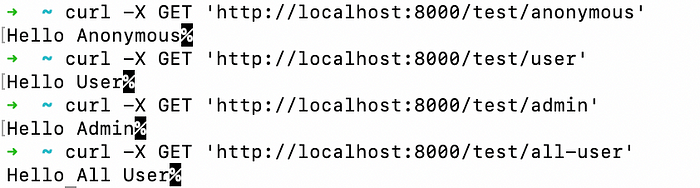
mvn spring-boot:run



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:



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'](http://localhost:8000/test/admin') \  
--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'](http://localhost:8000/test/admin') \  
--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](https://github.com/ddezoysa/keycloak-springboot-microservice).

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