Authentication and authorization using the Keycloak REST API

November 24, 2020 | Muhammad Edwin

Related topics: DevSecOps, Java, Linux, Security

Related products: Red Hat Single sign-on

Share:

Enabling authentication and authorization involves complex functionality beyond a simple login API. In a previous article, I described the Keycloak REST login API endpoint, which only handles some authentication tasks. In this article, I describe how to enable other aspects of authentication and authorization by using Keycloak REST API functionality out of the box.

Authentication versus authorization

But first, what is the difference between authentication and authorization? Simply stated, authentication means **who you are**, while authorization means **what can you do**, with each approach using separate methods for validation. For example, authentication uses the user management and login form, and authorization uses role-based access control (RBAC) or an access control list (ACL).

Fortunately, these validation methods are provided in Red Hat's single sign-on (SSO) tools, or in their upstream open source project, Keycloak's REST API.

Keycloak SSO case study

To better understand using Keycloak for authentication and authorization, let's start with a simple case study. Suppose that Indonesia's Ministry of Education is planning to create a single sign-on integration with multiple schools. They plan to maintain their students' and teachers' single account IDs across multiple schools using a centralized platform. This lets each user have the same role, but with different access and privileges at each school, as shown in Figure 1.

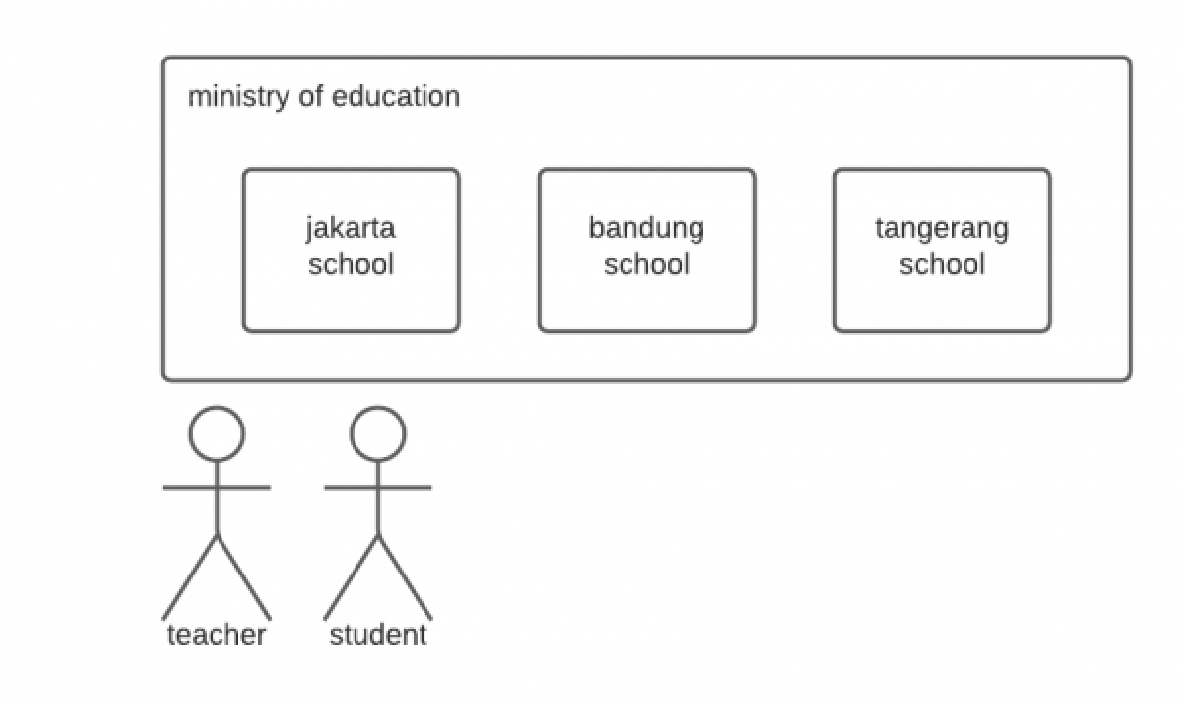
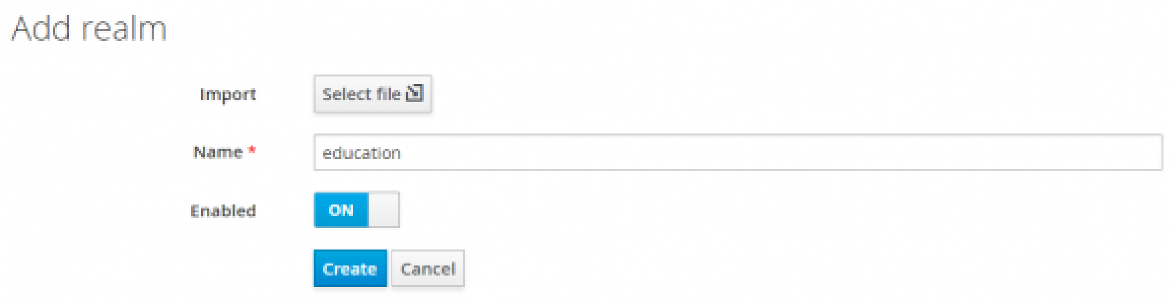


Figure 1: Each user can use the same role, but with different access and privileges at each school.">

Keycloak SSO demo

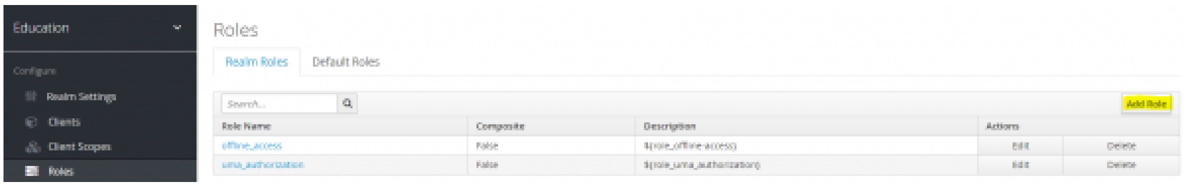
Let's start the demo by creating a Keycloak realm. Using the **Add realm** dialog box for this ministry (as shown in Figure 2). Name the realm `education`, set **Enabled** to **ON**, and click **Create**.


Add realm

Import	Select file
Name *	education
Enabled	ON

Figure 2: Create a Keycloak realm for the Ministry of Education named "education."

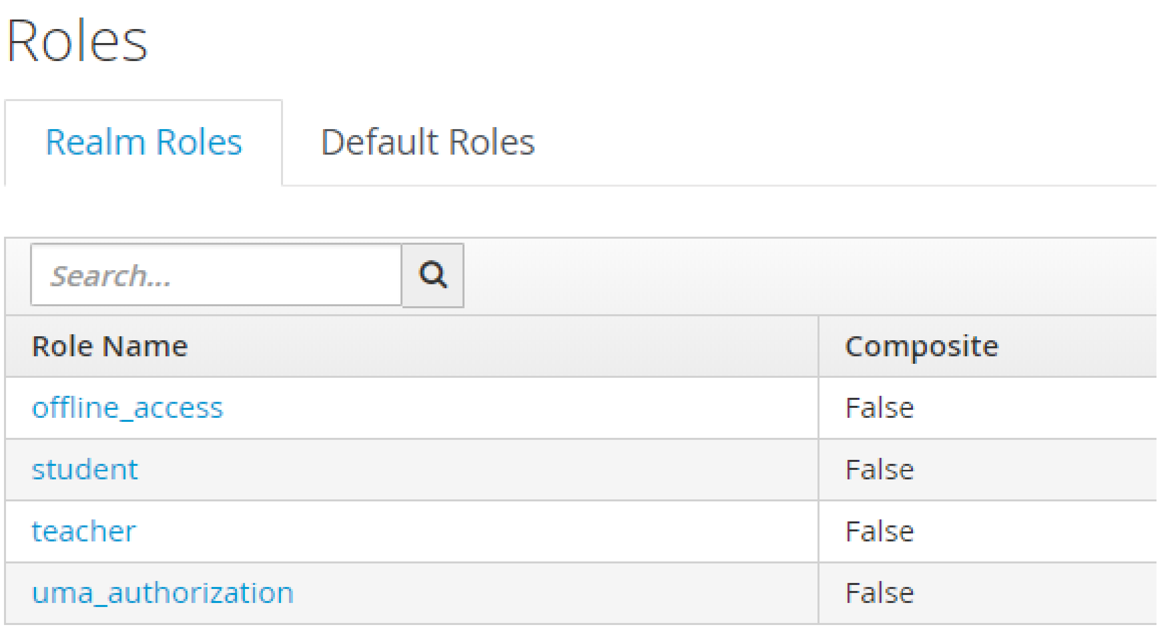
Next, go to the Roles page and make sure the **Realm Roles** tab is selected, as shown in Figure 3.



Role Name	Composite	Description	Actions	
offline_access	False	${role_offline-access}	Edit	Delete
uma_authorization	False	${role_uma_authorization}	Edit	Delete

Figure 3: Create two separate roles for the education realm: teacher and student.

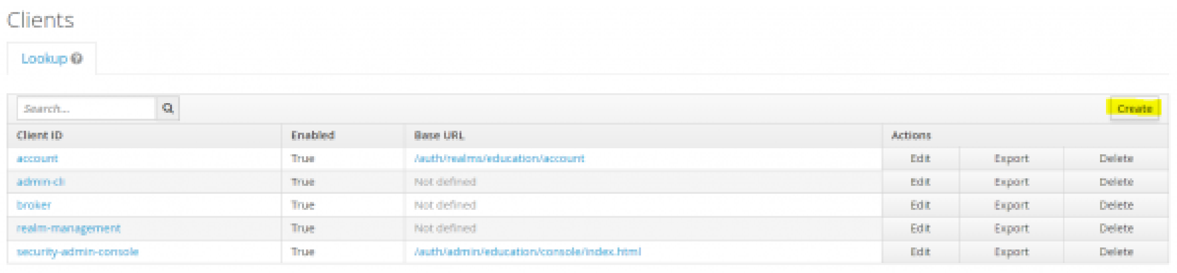
Click **Add Role** to create two separate roles for this realm called "teacher" and "student." These new roles will then appear in the **Realm Roles** tab as shown in Figure 4.



Role Name	Composite
offline_access	False
student	False
teacher	False
uma_authorization	False

Figure 4: Add the teacher and student roles.

Then, using the Clients page, click **Create** to add a client, as shown in Figure 5.

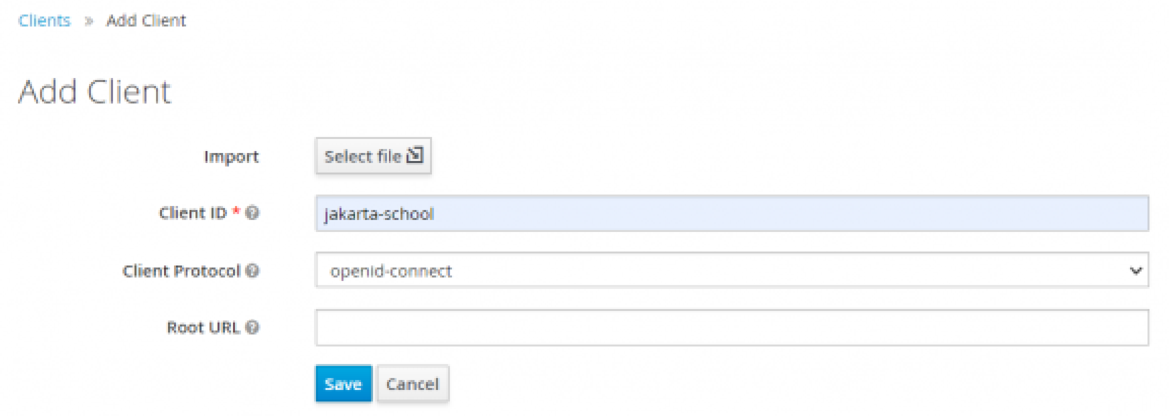


Clients

Client ID	Enabled	Base URL	Actions		
account	True	/auth/realms/education/account	Edit	Export	Delete
admin-cli	True	Not defined	Edit	Export	Delete
broker	True	Not defined	Edit	Export	Delete
realm-management	True	Not defined	Edit	Export	Delete
security-admin-console	True	/auth/admin/education/console/index.html	Edit	Export	Delete

Figure 5: Add a client.

On the **Add Client** page, create a client named `jakarta-school`, and click **Save** to add this client, as shown in Figure 6.



Clients » Add Client

Add Client

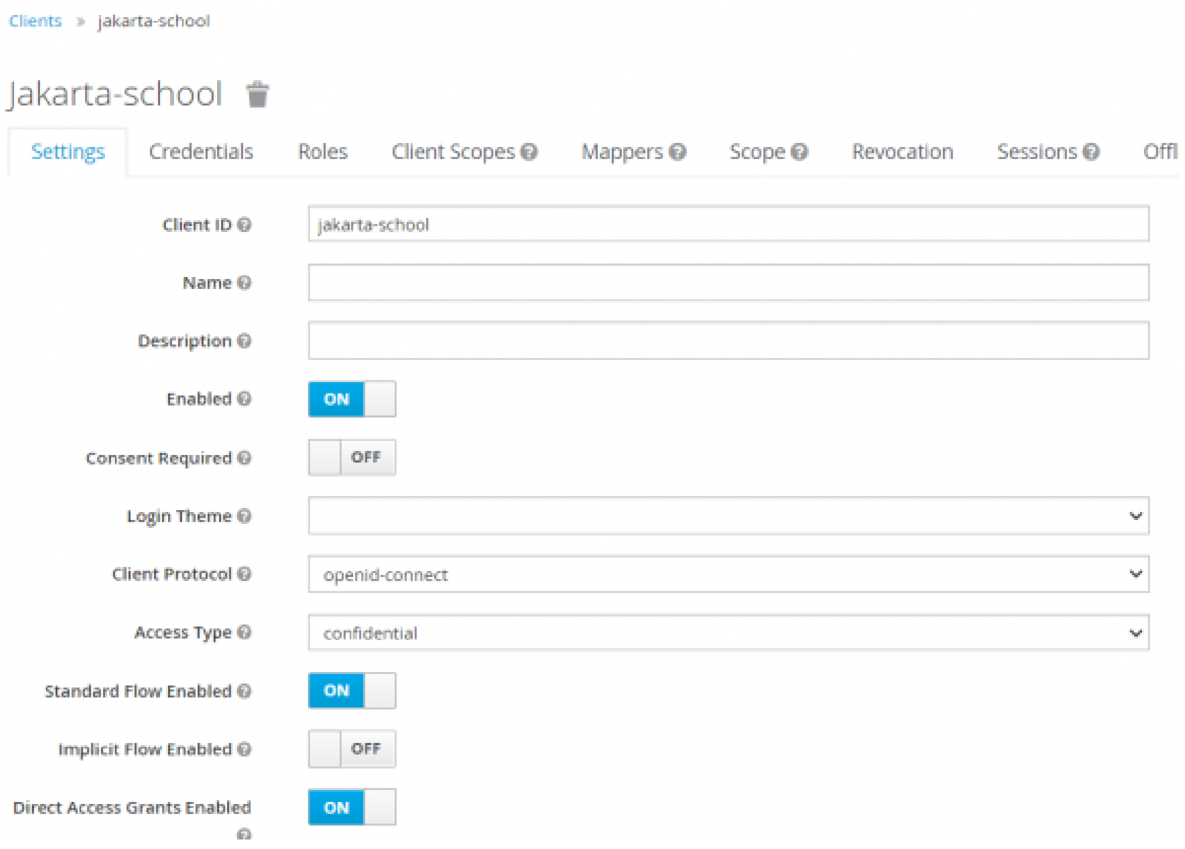
Client ID *: jakarta-school

Client Protocol: openid-connect

Root URL:

On the jakarta-school details page, go to the **Settings** tab and enter the following client configuration, as shown in Figure 7:

- **Client ID**: jakarta-school
- **Enabled**: ON
- **Consent Required**: OFF
- **Client Protocol**: openid-connect
- **Access Type**: confidential
- **Standard Flow Enabled**: ON
- **Impact Flow Enabled**: OFF
- **Direct Access Grants Enabled**: ON



Clients » jakarta-school

Jakarta-school

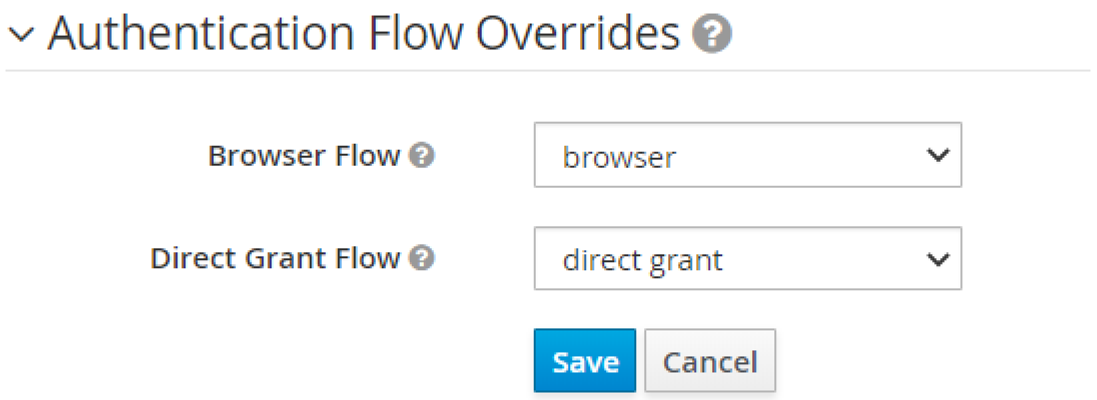
Settings | Credentials | Roles | Client Scopes | Mappers | Scope | Revocation | Sessions | Offl

Client ID	jakarta-school
Name	
Description	
Enabled	ON
Consent Required	OFF
Login Theme	
Client Protocol	openid-connect
Access Type	confidential
Standard Flow Enabled	ON
Implicit Flow Enabled	OFF
Direct Access Grants Enabled	ON

Figure 7: On the jakarta-school details page, enter the client configuration.

At the bottom of the same page, on the Authentication Flow Overrides part, we can set to the following as shown in Figure 8:

- **Browser Flow**: browser
- **Direct Grant Flow**: direct grant

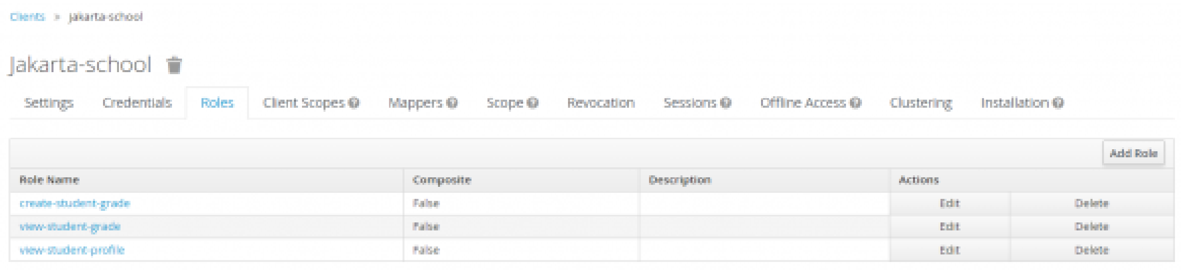


Authentication Flow Overrides

Browser Flow	browser
Direct Grant Flow	direct grant

Figure 8: Configure the authentication flow overrides.

Go to the **Roles** tab, click **Add Role**, and create the create-student-grade, view-student-grade, and view-student-profile roles for this client as shown in Figure 9. Each should be set to **Composite** False.


Clients » jakarta-school

Jakarta-school

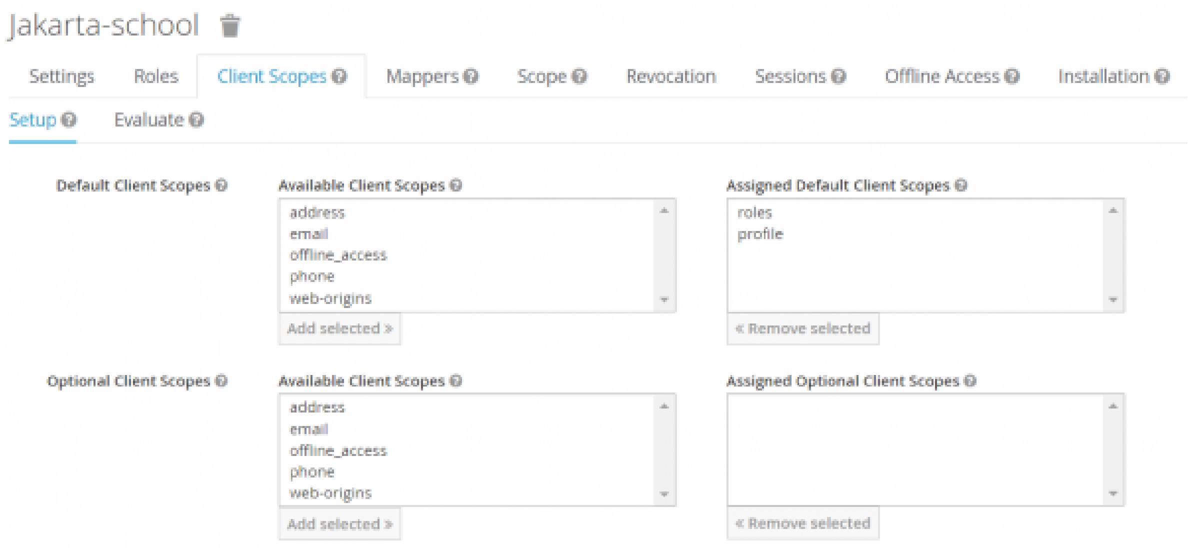
Settings | Credentials | Roles | Client Scopes | Mappers | Scope | Revocation | Sessions | Offline Access | Clustering | Installation

Add Role

Role Name	Composite	Description	Actions	
create-student-grade	False		Edit	Delete
view-student-grade	False		Edit	Delete
view-student-profile	False		Edit	Delete

Figure 9: Create roles for this client.

Next, go to the **Client Scopes** tab and in the **Default Client Scopes** section, add "roles" and "profile" to the **Assigned Default Client Scopes**, as shown in Figure 10.


Jakarta-school

Settings | Roles | Client Scopes | Mappers | Scope | Revocation | Sessions | Offline Access | Installation

Setup | Evaluate

Default Client Scopes

Available Client Scopes: address, email, offline_access, phone, web-origins

Add selected »

Assigned Default Client Scopes: roles, profile

« Remove selected

Optional Client Scopes

Available Client Scopes: address, email, offline_access, phone, web-origins

Add selected »

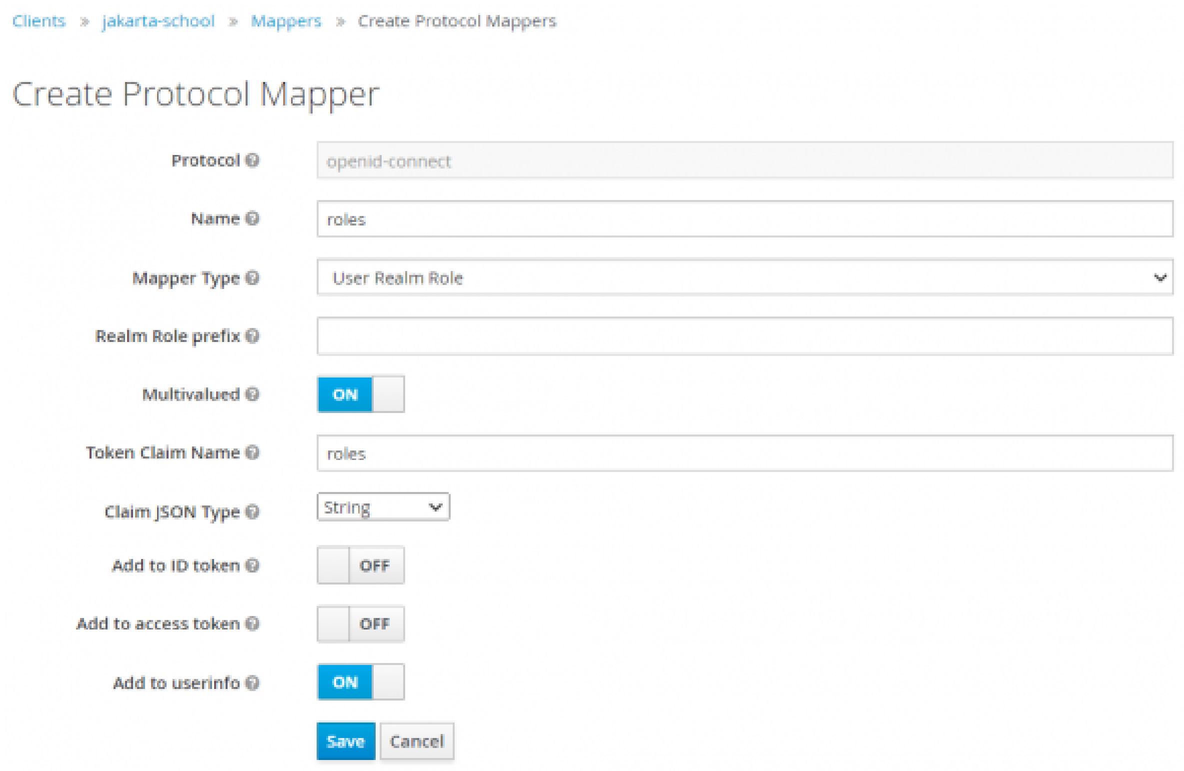
Assigned Optional Client Scopes

« Remove selected

Figure 10: Set the client scopes.

On the jakarta-school details page, select **Mappers** and then **Create Protocol Mappers**, and set mappers to display the client roles on the Userinfo API, as shown in Figure 11:

- **Name**: roles
- **Mapper Type**: User Realm Role
- **Multivalued**: ON
- **Token Claim Name**: roles
- **Claim JSON Type**: String
- **Add to ID token**: OFF
- **Add to access token**: OFF
- **Add to userinfo**: ON


Clients » jakarta-school » Mappers » Create Protocol Mappers

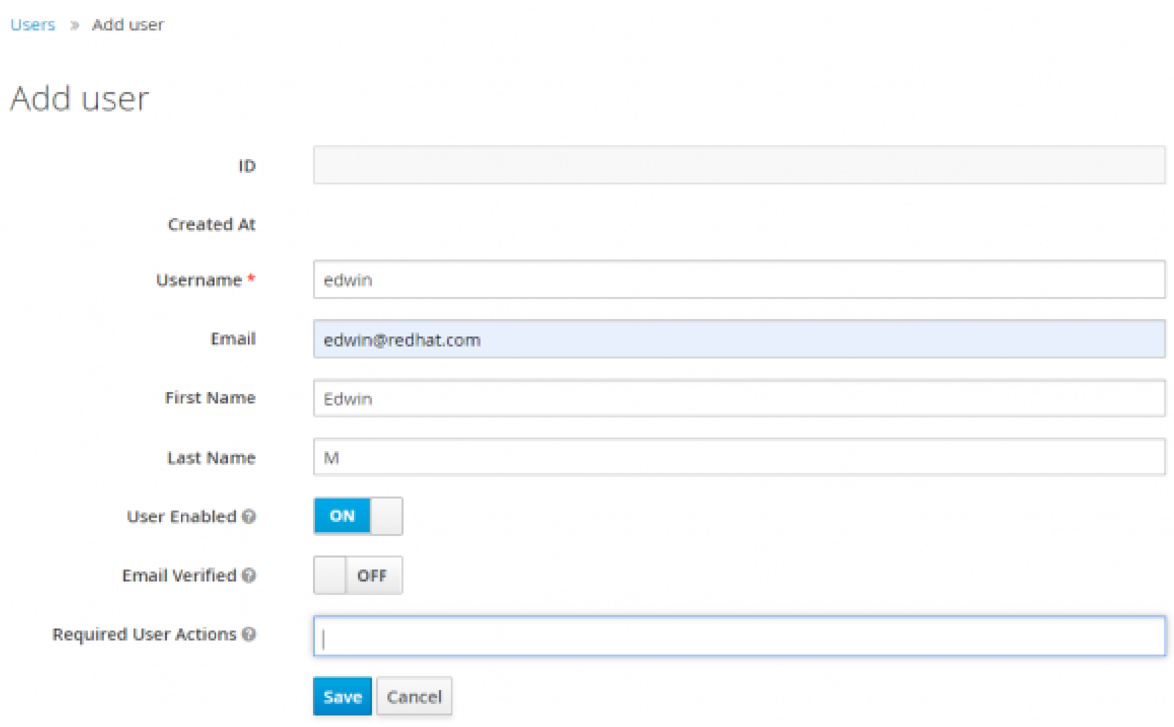
Create Protocol Mapper

Protocol	openid-connect
Name	roles
Mapper Type	User Realm Role
Realm Role prefix	
Multivalued	ON
Token Claim Name	roles
Claim JSON Type	String
Add to ID token	OFF
Add to access token	OFF
Add to userinfo	ON

Figure 11: Set the mappers to display the client roles.

Next, go to the **Users** page, select **Add user**, create the new users, and click **Save** as shown in Figure 12:

- **Username**: edwin
- **Email**: edwin@redhat.com
- **First Name**: Edwin
- **Last Name**: M
- **User Enabled**: ON
- **Email Verified**: OFF

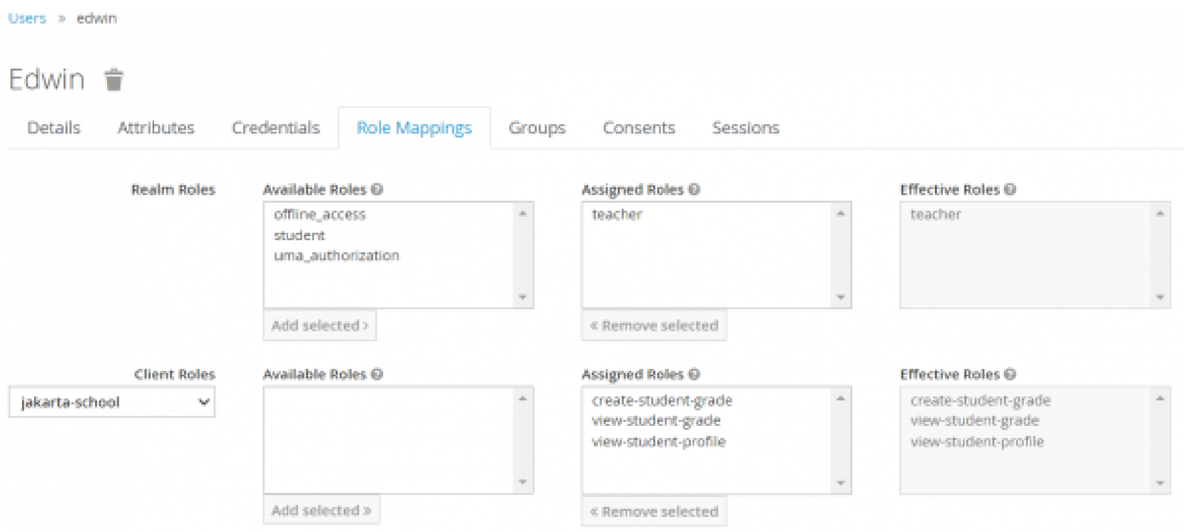

Users » Add user

Add user

ID	
Created At	
Username *	edwin
Email	edwin@redhat.com
First Name	Edwin
Last Name	M
User Enabled	ON
Email Verified	OFF
Required User Actions	

Figure 12: Create the new users.

And finally, in the **Role Mappings** tab, select the **Client Roles** for each user in jakarta-school, as shown in Figure 13. These should be create-student-grade, view-student-grade, and view-student-profile.


Users » edwin

Edwin

	Available Roles	Assigned Roles	Effective Roles
Realm Roles	offline_access student uma_authorization	teacher	teacher
Client Roles: jakarta-school		create-student-grade view-student-grade view-student-profile	create-student-grade view-student-grade view-student-profile

Figure 13: Select the Client Roles for each user in jakarta-school.

This concludes my demo of the Keycloak configuration.

Keycloak connection using a Java application

Now I want to demonstrate how to develop a very simple Java application. This application connects to your Keycloak instances and uses Keycloak's authentication and authorization capability through its REST API.

First, develop the Java application starting with a pom.xml file, as shown in the following sample:

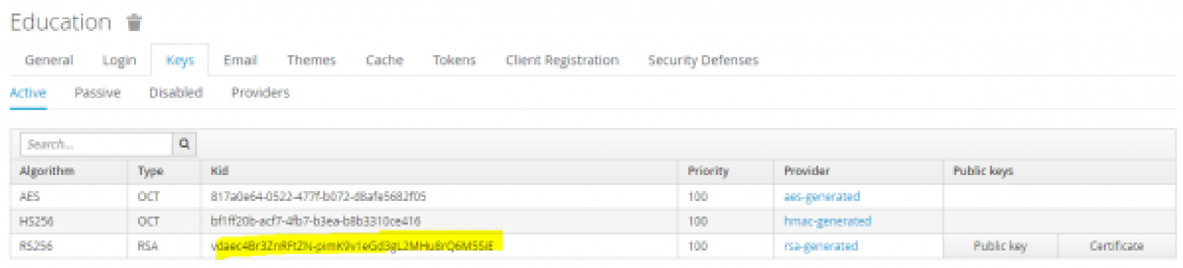
```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>EducationService</groupId>
    <artifactId>com.edw</artifactId>
    <version>1.0-SNAPSHOT</version>

    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.2.1.RELEASE</version>
        <relativePath/>
    </parent>

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

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-
web</artifactId>
        </dependency>

        <dependency>
            <groupId>com.auth0</groupId>
            <artifactId>jwks-rsa</artifactId>
            <version>0.12.0</version>
        </dependency>
        <dependency>
            <groupId>com.auth0</groupId>
            <artifactId>java-jwt</artifactId>
            <version>3.8.3</version>
        </dependency>

    </dependencies>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-
plugin</artifactId>
            </plugin>
        </plugins>
    </build>

</project>
```

The Java application also requires you to develop a simple properties file:

```
server.error.whitelabel.enabled=false
spring.mvc.favicon.enabled=false
server.port = 8082

keycloak.client-id=jakarta-school
keycloak.client-secret=197bc3b4-64b0-452f-9bdb-fcaea0988e90
keycloak.scope=openid, profile
keycloak.authorization-grant-type=password

keycloak.authorization-
uri=http://localhost:8080/auth/realms/education/protocol/op
enid-connect/auth
keycloak.user-info-
uri=http://localhost:8080/auth/realms/education/protocol/op
enid-connect/userinfo
keycloak.token-
uri=http://localhost:8080/auth/realms/education/protocol/op
enid-connect/token
keycloak.logout=http://localhost:8080/auth/realms/education
/protocol/openid-connect/logout
keycloak.jwk-set-
```

```
uri=http://localhost:8080/auth/realms/education/protocol/op
enid-connect/certs
keycloak.certs-id=vdaec4Br3ZnRFtZN-
pimK9v1eGd3gL2MHu8rQ6M5SiE

logging.level.root=INFO
```

Next, get the Keycloak certificate ID from the form shown in Figure 14.



Education

General Login Keys Email Themes Cache Tokens Client Registration Security Defenses

Active Passive Disabled Providers

Algorithm	Type	Kid	Priority	Provider	Public keys	
AES	OCT	817a0e64-0522-477f-b072-d8afe5682f05	100	aes-generated		
HS256	OCT	bf1ff20b-acf7-4fb7-b3ea-b8b3310ce416	100	hmac-generated		
RS256	RSA	vdaec4Br3ZnRFtZN-pimK9v1eGd3gL2MHu8rQ6M5SiE	100	rsa-generated	Public key	Certificate

Figure 14: Find the Keycloak certificate ID.

After that, and most importantly, your next task is to develop the integration code; several Keycloak APIs are involved in this action. Note that I did not go into detail about the Keycloak login API as it is already described in my previous article.

Start with a simple logout API:

```
        @Value("${keycloak.logout}")
        private String keycloakLogout;
```

```
    public void logout(String refreshToken) throws
Exception {
        MultiValueMap<String, String> map = new
LinkedMultiValueMap<>();
        map.add("client_id",clientId);
        map.add("client_secret",clientSecret);
        map.add("refresh_token",refreshToken);

        HttpEntity<MultiValueMap<String, String>> request =
new HttpEntity<>(map, null);
        restTemplate.postForObject(keycloakLogout, request,
String.class);
    }
```

First, I want to point out that, for logging out, it's critical that you use your `refresh_token` parameter and not `access_token`. Now, use the

API to check for whether a bearer token is valid and active or not, in order to validate whether a request is bringing a valid credential.

```
    @Value("${keycloak.user-info-uri}")
    private String keycloakUserInfo;
```

```
    private String getUserInfo(String token) {
        MultiValueMap<String, String> headers = new
LinkedMultiValueMap<>();
        headers.add("Authorization", token);

        HttpEntity<MultiValueMap<String, String>> request =
new HttpEntity<>(null, headers);
        return restTemplate.postForObject(keycloakUserInfo,
request, String.class);
    }
```

For authorization, you can use two approaches to decide whether a given role is eligible to access a specific API. The first approach is to determine what role a bearer token brings by verifying it against Keycloak's userinfo API, and the next approach is to validate a role within the bearer token.

For the first approach, you can expect the following response from Keycloak:

```
{
    "sub": "ef2cbe43-9748-40e5-aed9-fe981e3082d5",
    "roles": [
        "teacher"
    ],
    "name": "Edwin M",
    "preferred_username": "edwin",
    "given_name": "Edwin",
    "family_name": "M"
}
```

As you can see, there is a `roles` tag there and one approach is to validate the access right based on that. The drawback is the multiple roundtrip request between your application and Keycloak for each request, which results in higher latency.

Another approach is to read the contents of the JWT token, which are sent through each request. In order to successfully decode your JWT token, you must know what public key is used for signing it. That's why Keycloak provides a JWKS endpoint. You can view its content by using the curl command, as shown in the following sample:

```
$ curl -L -X GET
'http://localhost:8080/auth/realms/education/protocol/openi
d-connect/certs'
```

For this previous sample, the result is as follows:

```
{
    "keys": [
        {
            "kid": "vdaec4Br3ZnRFtZN-
pimK9v1eGd3gL2MHu8rQ6M5SiE",
            "kty": "RSA",
            "alg": "RS256",
            "use": "sig",
            "n":
"4OPCc_LDhU6ADQj7cEgRei4VUf4PZH8GYsxsR6RSdeKmDvZ48hCSEFiEgf
c3FIfh-
gC4r9PtKucc_nkRofrAKR4qL8KNNoSuzQAOC92Yz6r7Ao4HppHJ8-
QVdo5H-
d9wfNSlDLBSo5My4b4EnHb1HLuFxDqyhFpGvsoUJdgbt3m_Q3WAVb2yrM83
S6HX__vrQvqUk2e7z5RNrI7LSsW3ZOz9fU7pvm8-
kFFAIPJ7fOJIC7UQ9wBWg3YdwQ0B2b24jXjVr0QCGzqJ6o1G_UZYSJCDMGQ
DpDcEuYnvSKBLfVR-
0EcAjolRhcSPjHlW0Cp0YU8qwWDHpjkbrMrFmxlO4Q",
            "e": "AQAB"
        }
    ]
}
```

Note that, in the previous sample, `kid` means key id, `alg` is the algorithm, and `n` is the public key used for this realm. You can use this public key to easily decode our JWT token, and read `roles` from the JWT claim. The sample decoded JWT token is shown next:

```
{
  "jti": "85edca8c-a4a6-4a4c-b8c0-356043e7ba7d",
  "exp": 1598079154,
  "nbf": 0,
  "iat": 1598078854,
  "iss": "http://localhost:8080/auth/realms/education",
  "sub": "ef2cbe43-9748-40e5-aed9-fe981e3082d5",
  "typ": "Bearer",
  "azp": "jakarta-school",
  "auth_time": 0,
  "session_state": "f8ab78f8-15ee-403d-8db7-7052a8647c65",
  "acr": "1",
  "realm_access": {
    "roles": [
      "teacher"
    ]
  },
  "resource_access": {
    "jakarta-school": {
      "roles": [
        "create-student-grade",
        "view-student-profile",
        "view-student-grade"
      ]
    }
  },
  "scope": "profile",
  "name": "Edwin M",
  "preferred_username": "edwin",
  "given_name": "Edwin",
  "family_name": "M"
}
```

You can read the `roles` tag by using the code shown in the following sample:

```
    @GetMapping("/teacher")
    public HashMap teacher(@RequestHeader("Authorization")
String authHeader) {
        try {
            DecodedJWT jwt =
JWT.decode(authHeader.replace("Bearer", "").trim());

            // check JWT is valid
            Jwk jwk = jwtService.getJwk();
            Algorithm algorithm =
Algorithm.RSA256((RSAPublicKey) jwk.getPublicKey(), null);
```

```
            algorithm.verify(jwt);

            // check JWT role is correct
            List<String> roles =
((List)jwt.getClaim("realm_access").asMap().get("roles"));
            if(!roles.contains("teacher"))
                throw new Exception("not a teacher role");

            // check JWT is still active
            Date expiryDate = jwt.getExpiresAt();
            if(expiryDate.before(new Date()))
                throw new Exception("token is expired");

            // all validation passed
            return new HashMap() {{
                put("role", "teacher");
            }};
        } catch (Exception e) {
            logger.error("exception : {} ",
e.getMessage());
            return new HashMap() {{
                put("status", "forbidden");
            }};
        }
    }
```

The best part of this approach is that you can place the public key from Keycloak in a cache, which reduces the round-trip request, and this practice eventually increases application latency and performance. The full code for this article can be found in my GitHub repository.

Conclusion

In conclusion, I prepared this article first to explain that enabling authentication and authorization involves complex functionality, beyond just a simple login API. Then I demonstrated how to enable many aspects of authentication and authorization using Keycloak REST API functionality out of the box.

In addition, I demonstrated how to develop a simple Java application that connects to your Keycloak instances, and uses Keycloak's authentication and authorization capability through its REST API.