Table of Contents

[Step 1: Getting Started With Keycloak 2](#_Toc64494361)

[Step 2: Create dev Realm 3](#_Toc64494362)

[Step 3: Create a Client (Micro-Service) 3](#_Toc64494363)

[Step 4: Configure Client 4](#_Toc64494364)

[Step 5: Create Client Role 5](#_Toc64494365)

[Step 6: Create a Mapper (To get user\_name in access token) 6](#_Toc64494366)

[Step 7: Create User 7](#_Toc64494367)

[Step 8: Map Client Role To User 8](#_Toc64494368)

[Step 9: Get Configuration From OpenID Configuration Endpoint 9](#_Toc64494369)

[To Get Access Token Using Postman (For Testing) 10](#_Toc64494370)

[Step 10: Create a Spring Boot Application 11](#_Toc64494371)

[Step 11: Configure application.properties 11](#_Toc64494372)

[Step 12: JWT Access Token Customizer 14](#_Toc64494373)

[Step 13: OAuth2 Resource Server Configurer 14](#_Toc64494374)

[Step 14: Secure REST Endpoints 14](#_Toc64494375)

[For Complete Code 15](#_Toc64494376)

[For Complete Articles 15](#_Toc64494377)

[Benefits Of Using Spring OAuth2 Over Keycloak Adapters: 16](#_Toc64494378)

**Let us see how we can use Keycloak OIDC support and Spring OAuth2 library to secure REST APIs**.

# Step 1: Getting Started With Keycloak

Refer Keycloak getting started [documentation](https://www.keycloak.org/docs/latest/getting_started/index.html) to run and setup keycloak admin user.

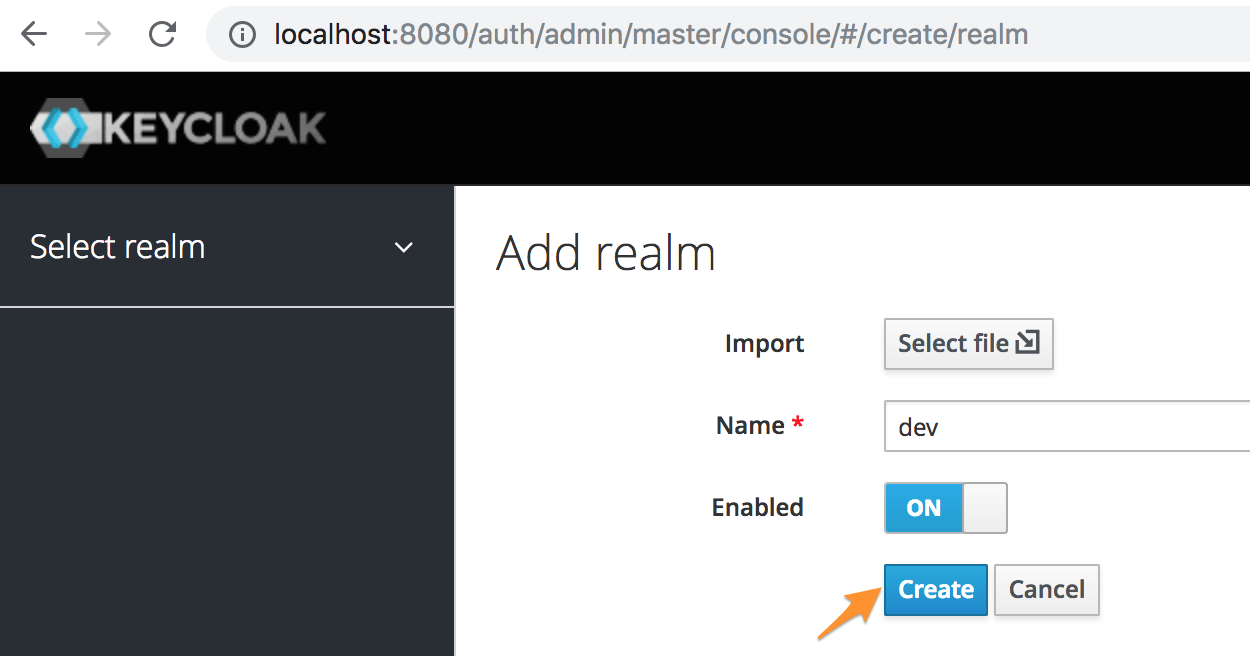
After running Keycloak, access keycloak admin console using <http://localhost:8080/auth>

Setup keycloak username=admin, password=admin.

**Note:**Standalone Keycloak runs on Wildfly server. Don’t worry about configuring a user to manage Wildfly server. We need a Keycloak admin user to create realm, client, user, role etc in Keycloak.

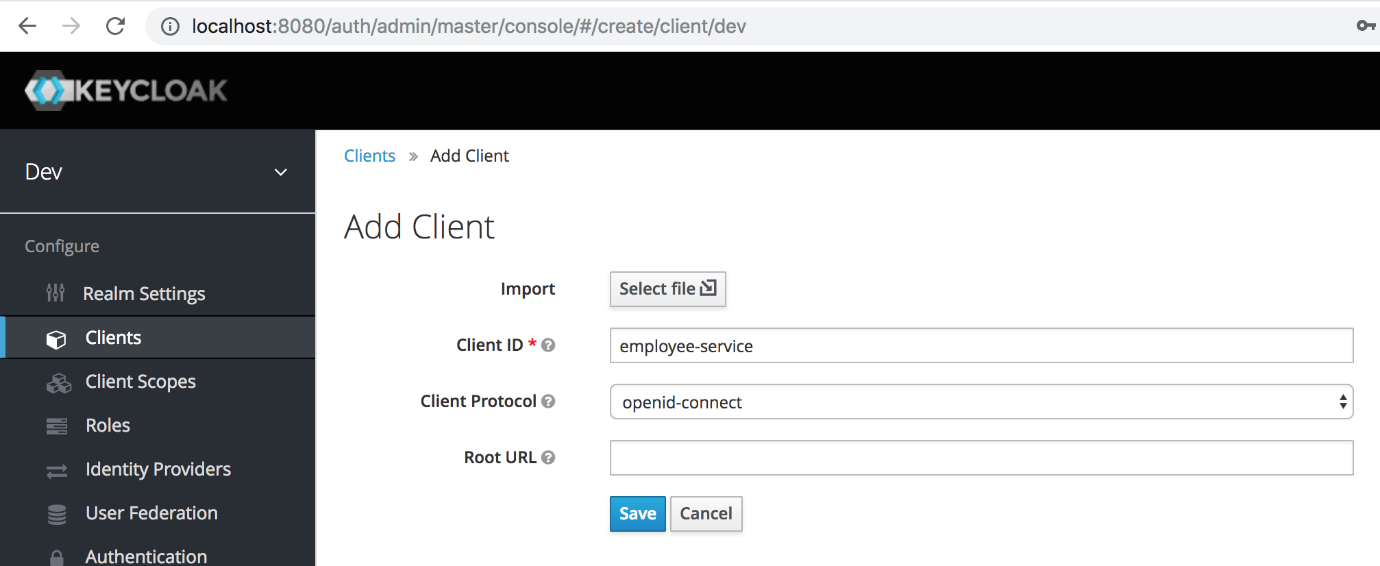
# ****Step 2: Create dev Realm****

Name: dev



Step 3: Create a Client (Micro-Service)

Client ID : **employee-service**  
Client Protocol : **openid-connect**

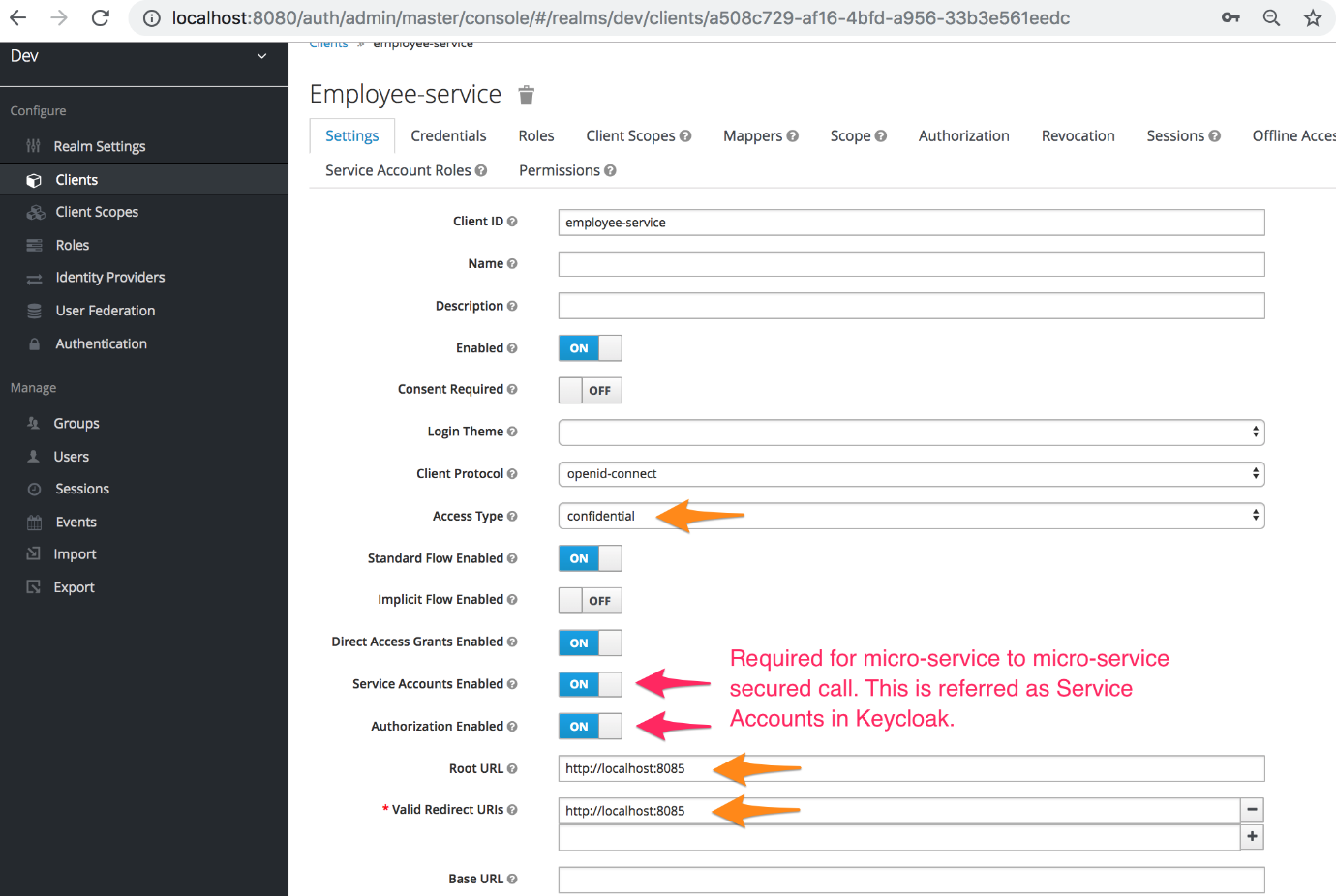


# Step 4: Configure Client

If Keycloak runs on Port 8080, make sure your microservice runs on another port. In the example, micro-service is configured to run on 8085.

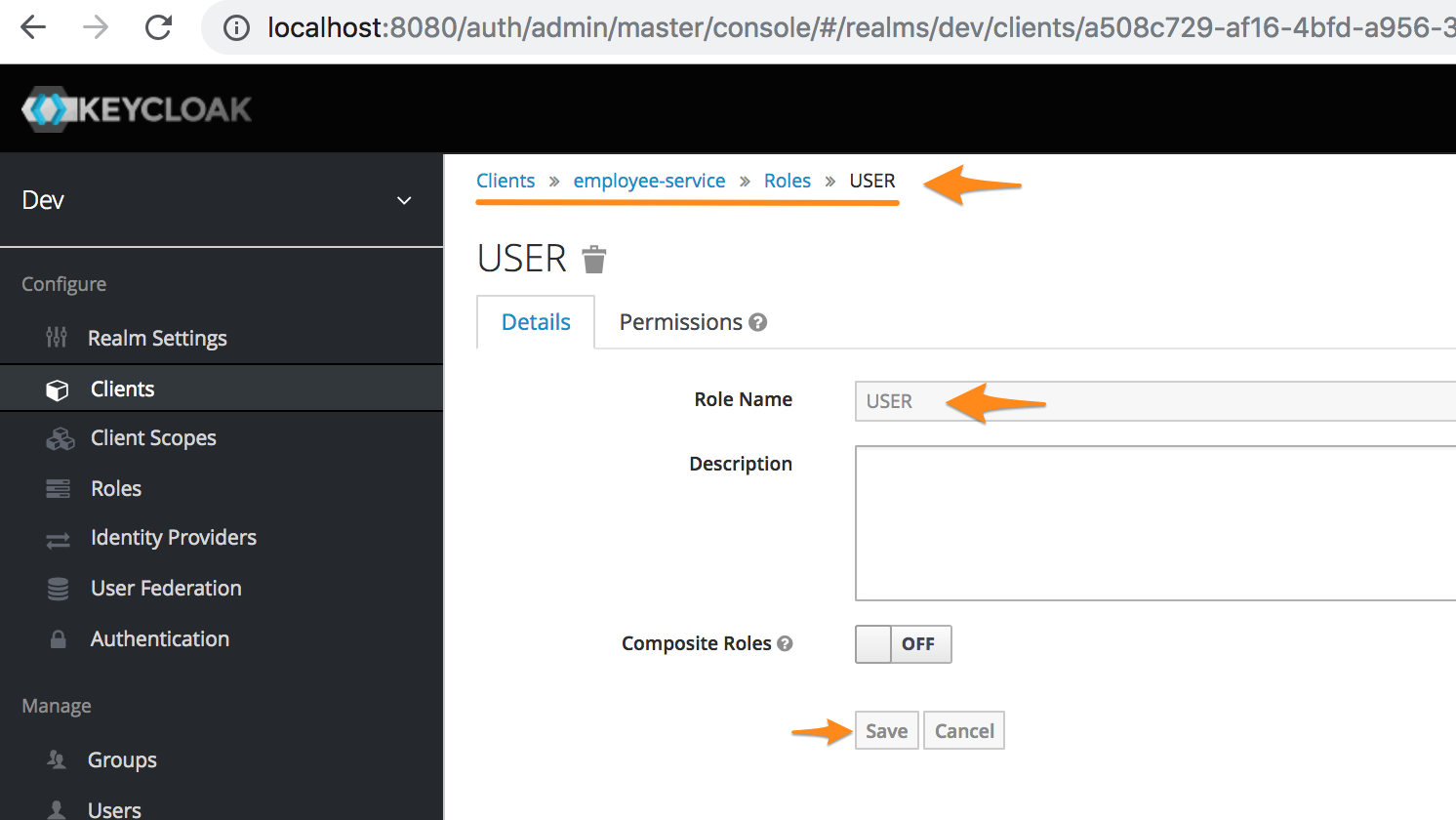
Access Type : **confidential**  
Valid Redirect URIs : [**http://localhost:8085**](http://localhost:8085/)**# Required for micro-service to micro-service secured calls**  
Service Accounts Enabled : **On**Authorization Enabled : **On**

**Note:** Access Type confidential supports getting access token using client credentials grant as well as authorization code grant. If a micro-service need to call another micro-service, caller will be ‘confidential’ and callee will be ‘bearer-only’.



# Step 5: Create Client Role

Create a role under the client. In this case, role USER is created under employee-service.



# Step 6: Create a Mapper (To get ****user\_name**** in access token)

Keycloak access token is a [JWT](https://jwt.io/). It is a JSON and each field in that JSON is called a claim. By default, logged in username is returned in a claim named “preferred\_username” in access token. Spring Security OAuth2 Resource Server expects username in a claim named **“user\_name”**. Hence, we had to create below mapper to map logged in username to a new claim named user\_name.

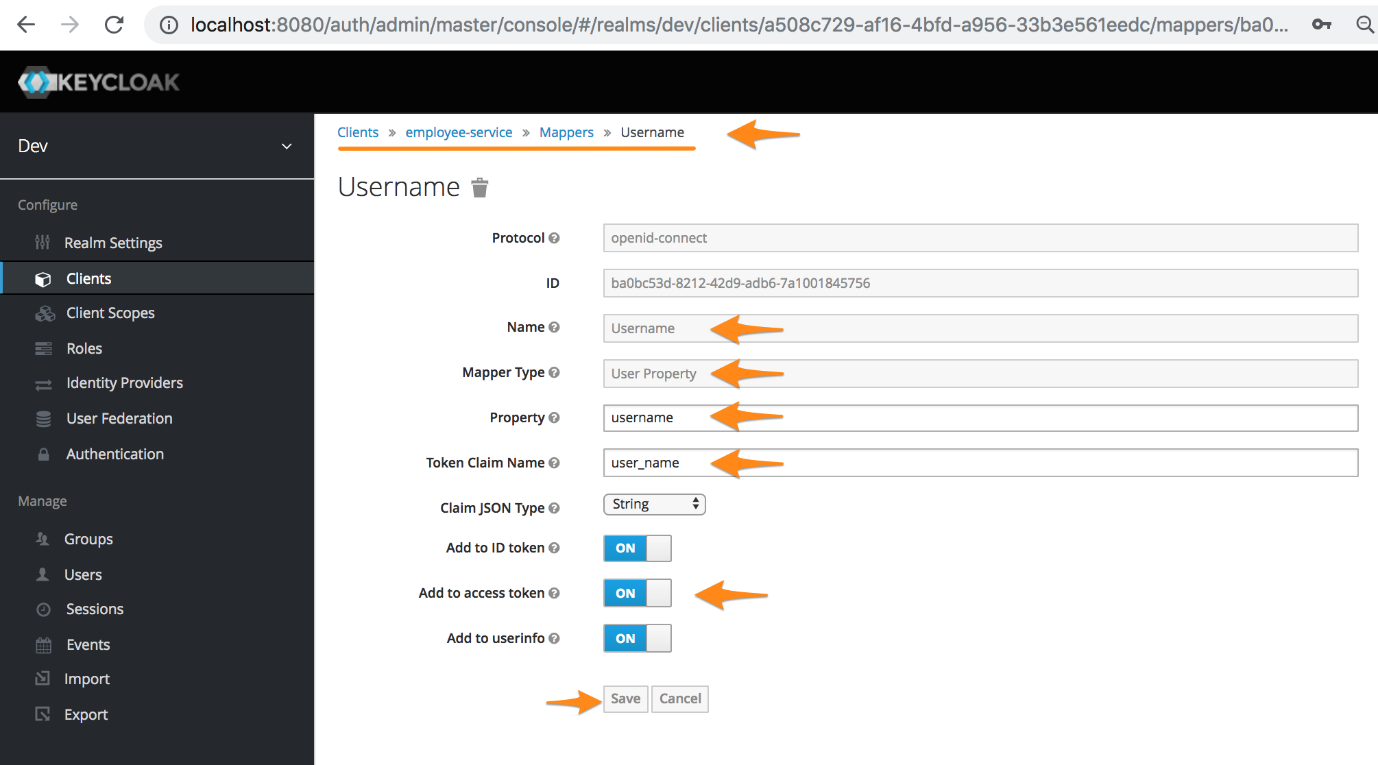


Figure 4: Create Mapper

# Step 7: Create User

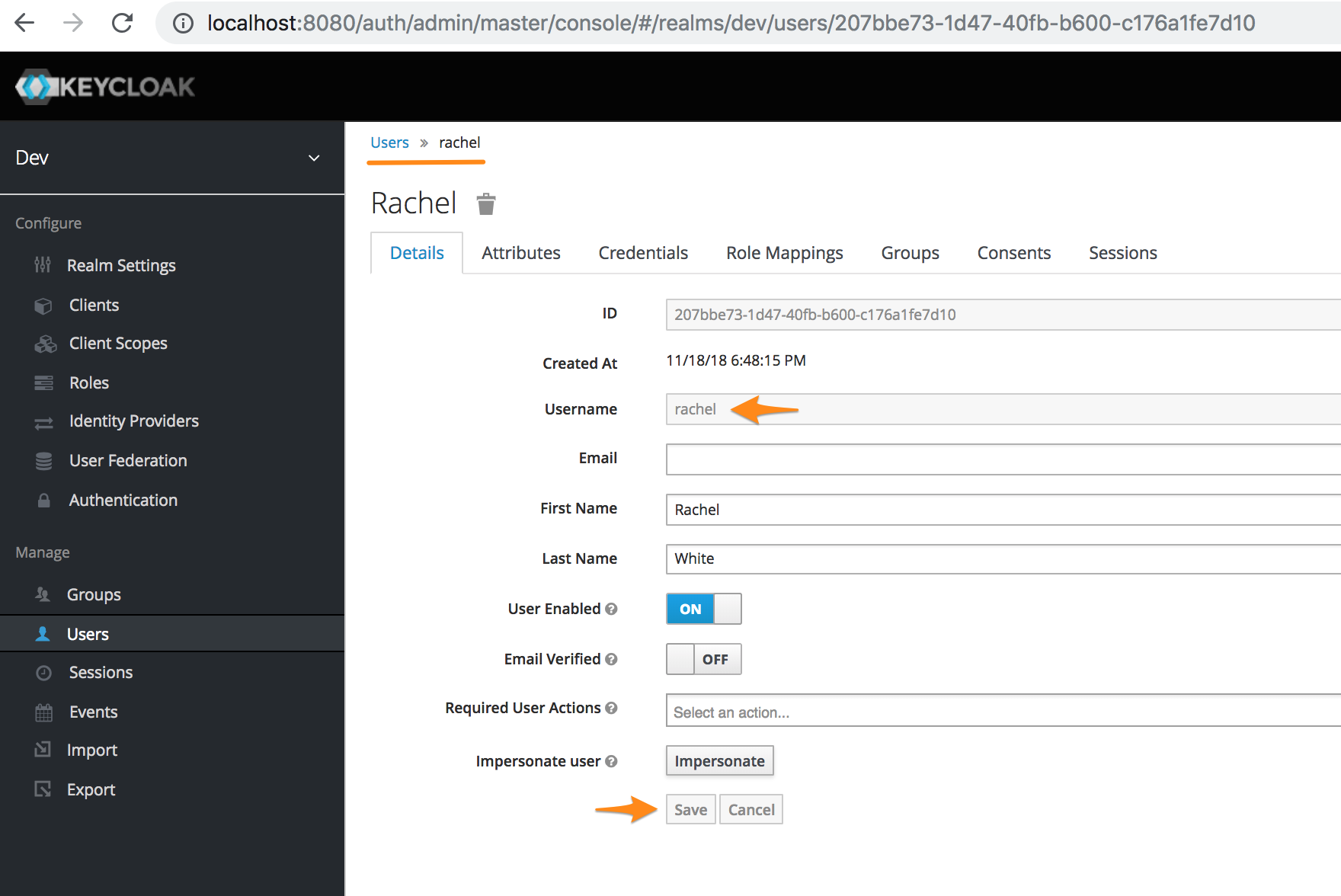


Figure 5: Create User

# Step 8: Map Client Role To User

In order to provide access to client (micro-service), respective role needs to be assigned/mapped to user.

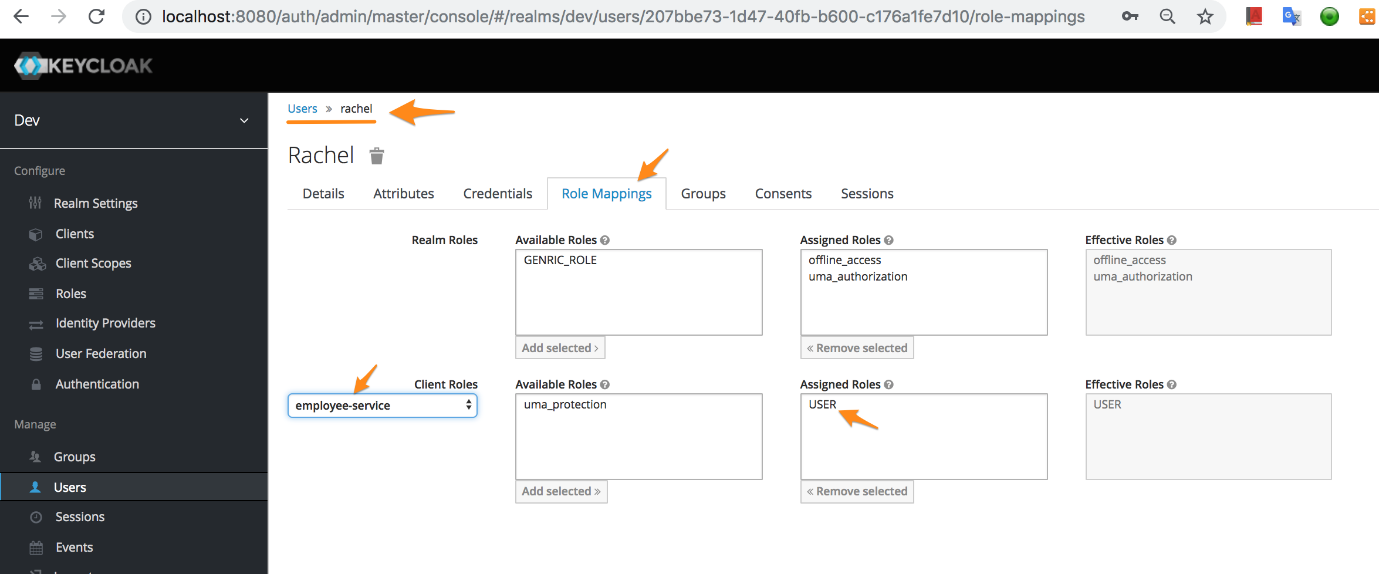


Figure 6: Assign role to user

# Step 9: Get Configuration From OpenID Configuration Endpoint

Because Keycloak is OpenID Connect and OAuth2 complaint, below is OpenID Connection configuration URL to get details about all security endpoints,

GET [**http://localhost:8080/auth/realms/dev/.well-known/openid-configuration**](http://localhost:8080/auth/realms/dev/.well-known/openid-configuration)

**Important URLS to be copied from response:**

**issuer** : <http://localhost:8080/auth/realms/dev>

**authorization\_endpoint:** ${**issuer**}/protocol/openid-connect/auth

**token\_endpoint:** ${**issuer**}/protocol/openid-connect/token  
  
**token\_introspection\_endpoint:** ${**issuer**}[/protocol/openid-connect/token/introspect](http://localhost:8080/auth/realms/dev/protocol/openid-connect/token/introspect)

**userinfo\_endpoint:** ${**issuer**}[/protocol/openid-connect/userinfo](http://localhost:8080/auth/realms/dev/protocol/openid-connect/userinfo)

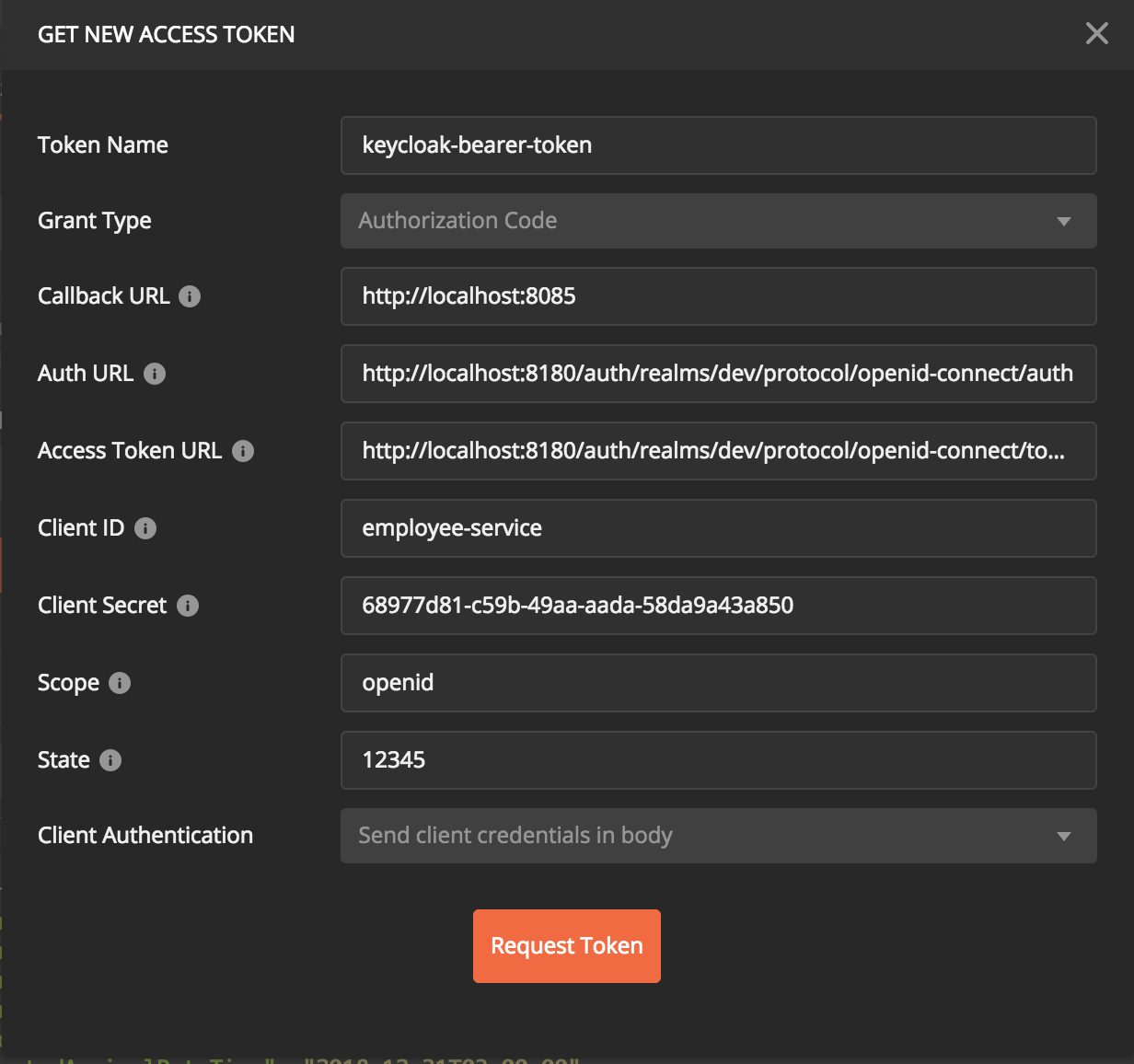
**Response also contains grant types and scopes supported**

**grant\_types\_supported:** ["client\_credentials", …]

**scopes\_supported:** ["openid", …]

# To Get Access Token Using Postman (For Testing)

Select [Authorization](https://learning.getpostman.com/docs/postman/sending_api_requests/authorization/) Type as OAuth 2.0, click on ‘Get New Access Token’ and enter following details.



Postman tool screenshot: To get access token from keycloak for a client

* Make sure you select client authentication as “Send client credentials in body” while requesting token.
* Callback URL is redirect URL configured in Keycloak.
* Client secret may be different for you, copy the one from client configuration in keycloak.
* You may also use [https://jwt.io](https://jwt.io/) to inspect the contents of token received.

# Step 10: Create a Spring Boot Application

**Spring Boot**

<**parent**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-parent</**artifactId**>  
 <**version**>2.0.1.RELEASE</**version**>  
 <**relativePath**/> <!-- lookup parent from repository --></**parent**>

**Dependencies**

<**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter</**artifactId**>  
</**dependency**><**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-security</**artifactId**>  
</**dependency**><**dependency**>  
 <**groupId**>org.springframework.security.oauth.boot</**groupId**>  
 <**artifactId**>spring-security-oauth2-autoconfigure</**artifactId**>  
 <**version**>2.0.1.RELEASE</**version**>  
</**dependency**><**dependency**>  
 <**groupId**>org.springframework.boot</**groupId**>  
 <**artifactId**>spring-boot-starter-web</**artifactId**>  
</**dependency**>

# Step 11: Configure application.properties

**General Security Properties**

**# Can be set to false to disable security during local development**  
rest.security.enabled=truerest.security.api-matcher=/api/\*\*  
rest.security.cors.allowed-origins=\*  
rest.security.cors.allowed-headers=\*  
rest.security.cors.allowed-methods=GET,POST,PUT,PATCH,DELETE,OPTIONS  
rest.security.cors.max-age=3600

**Properties to secure REST Endpoints using OAuth2 Resource Server**

**rest.security.issuer-uri**=http://localhost:8080/auth/realms/devsecurity.oauth2.resource.id=employee-servicesecurity.oauth2.resource.token-info-uri=${**rest.security.issuer-uri**}/protocol/openid-connect/token/introspectsecurity.oauth2.resource.user-info-uri=${**rest.security.issuer-uri**}/protocol/openid-connect/userinfosecurity.oauth2.resource.jwt.key-value=-----BEGIN PUBLIC KEY-----\nMIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAhWOcKAVAwt+5FF/eE2hLaMVD5zQBBr+RLdc7HFUrlvU9Pm548rnD+zRTfOhnl5b6qMjtpLTRe3fG+8chjPwQriRyFKCzg7eYNxuR/2sK4okJbfQSZFs16TFhXtoQW5tWnzK6PqcB2Bpmy3x7QN78Hi04CjNrPz2BX8U+5BYMavYJANpp4XzPE8fZxlROmSSyNeyJdW30rJ/hsWZJ5nnxSZ685eT4IIUHM4g+sQQTZxnCUnazNXng5B5yZz/sh+9GOXDGT286fWdGbhGKU8oujjSJLOHYewFZX5Jw8aMrKKspL/6glRLSiV8FlEHbeRWxFffjZs/D+e9A56XuRJSQ9QIDAQAB\n-----END PUBLIC KEY-----

Image for post

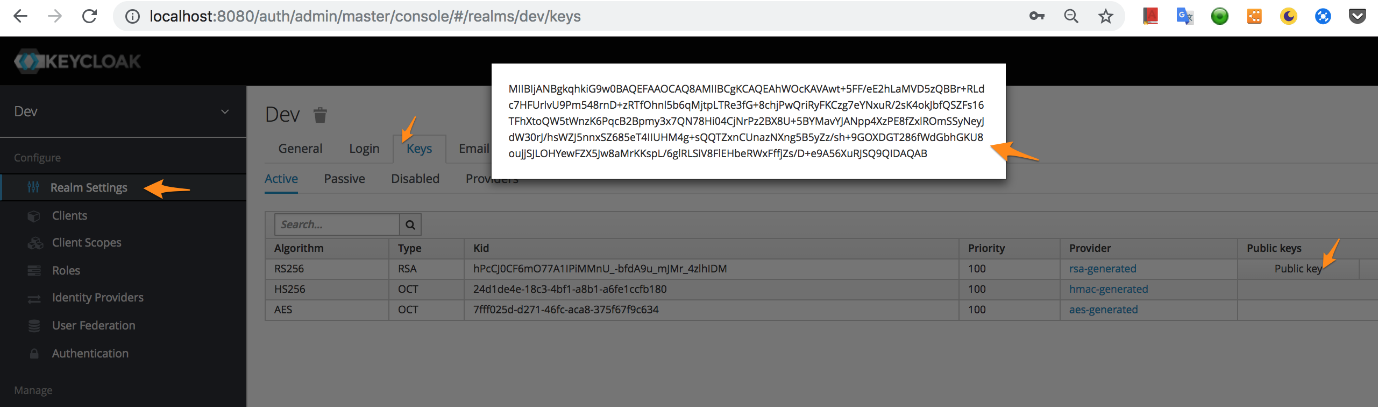


Figure 7: Copy jwt public key value

**Note 1:** security.oauth2.resource.jwt.key-value property value can be copied from public key at realm level. This is very important and this property is what uses **JwtAccessTokenCustomizer** which we will see later.

**Note 2:** Property values will be different based on your configuration, care should be take to use correct values.

**Properties to call another micro-service (Service Accounts)**

**# If this micro-services that needs to call another # secured micro-service**security.oauth2.client.client-id=employee-servicesecurity.oauth2.client.client-secret=68977d81-c59b-49aa-aada-58da9a43a850security.oauth2.client.user-authorization-uri=${rest.security.issuer-uri}/protocol/openid-connect/authsecurity.oauth2.client.access-token-uri=${rest.security.issuer-uri}/protocol/openid-connect/tokensecurity.oauth2.client.scope=openid  
security.oauth2.client.grant-type=client\_credentials

**Note:** Above properties are required for **OAuth2RestTemplate** that is used to make secure service account calls.

# Step 12: JWT Access Token Customizer

In order for Spring OAuth2 to parse and set SecurityConextHolder, it needs the roles or authorities from token. Also, in order to determine the list of clients/application/micro-service a user has access, it needs the list of client ids from token. This is the only setup that needs some special handling.

# Step 13: OAuth2 Resource Server Configurer

**Note:** OAuth2RestTemplate is required if this micro-service needs to call another micro-service.

# Step 14: Secure REST Endpoints

**PreAuthorize** annotation is use to secure REST endpoints with appropriate roles. Refer below example.

**Step 15: Disable Basic Auth if not required**

In order to disable default security, **SecurityAutoConfiguration** and **UserDetailsServiceAutoConfiguration** can be excluded.

# For Complete Code

Refer: <https://github.com/bcarun/spring-oauth2-keycloak-connector>

# For Complete Articles

Refer <https://medium.com/@bcarunmail/securing-rest-api-using-keycloak-and-spring-oauth2-6ddf3a1efcc2>

Refer <https://medium.com/@ravthiru/rest-service-protected-using-keycloak-authorization-services-a6ad2d8ecb9f>

# Benefits Of Using Spring OAuth2 Over Keycloak Adapters:

1. Most of the time, Keycloak Server upgrade requires upgrading Keycloak adapter as well. If Keycloak adapters were used in 100s of micro-services, then all these micro-services need to be upgraded to use newer version of Keycloak adapter. This requires regression testing of all micro-services and it is time consuming. This can be avoided by using Spring OAuth2 which integrates with Keycloak at protocol level. As these protocols don’t change often and Keycloak Server upgrade will continue to support the respective protocol version, no change is required in micro-services when Keycloak Server is upgraded.
2. Sometimes, Spring Boot version upgrade requires Keycloak and Keycloak Spring Boot adapter or Keycloak Spring Security adapter to be upgraded. This can be avoided if we use Spring OAuth2.
3. If organization decides to migrate from Keycloak to another OAuth2 OpenID Authx Provider, all the micro-services that used Keycloak Spring Boot adapter and Keyclock Spring Security needs to be refactored. Using Spring OAuth2 + Spring Security can significantly simplify the migrations.
4. [OAuth2RestTemplate](https://docs.spring.io/spring-security/oauth/apidocs/org/springframework/security/oauth2/client/OAuth2RestTemplate.html) class available in Spring OAuth2 takes care of refreshing on need and caching access tokens using OAuth2Context. This is a great benefit when there is a need to securely interact between micro-services.

To understand accessing secured micro-service from another micro-service using **OAuth2RestTemplate**, refer my other article stated below.