Ugonna Ubaka · Follow
Oct 3, 2021 · 8 min read · Listen

Securing your Spring-Boot Microservice with Keycloak and Spring Security

Photo by Jose Fontano on Unsplash

Keycloak is an open-source product developed by the RedHat Community which provides identity and access management solutions to modern applications. It requires little to no code to secure your applications and services.

Let's dive right in

Set Up Keycloak

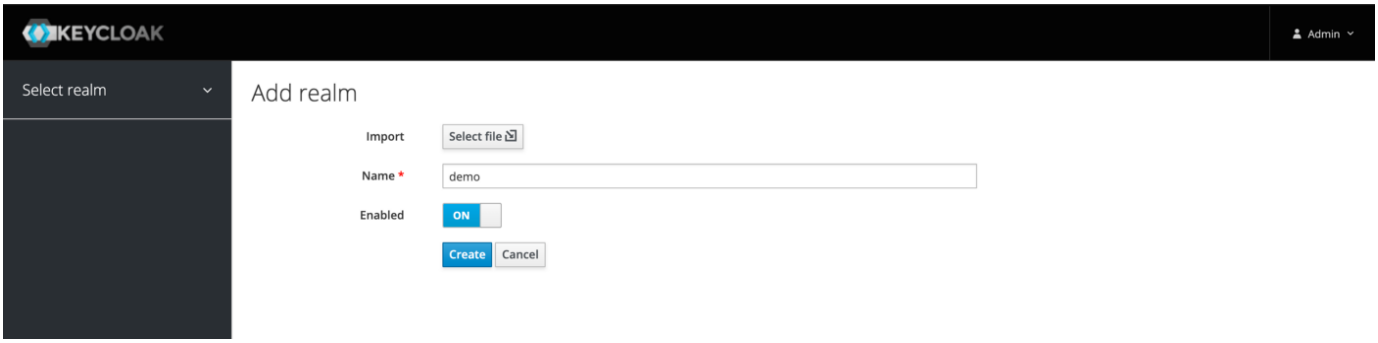
Head over to the Keycloak's getting-started-guide , download and set up keycloak on your local.

Getting Started Guide
This section describes how to install and start a Keycloak server in standalone mode, set up the initial admin user...
www.keycloak.org

Create a Realm

A realm manages a set of users, credentials, roles, and groups. The users you create in a realm belong to that realm. So, when they log in to Keycloak, they log in to the specified realm

1. Click on the upper left corner and choose **Add realm**.
2. Name it `demo` and click **Create**.



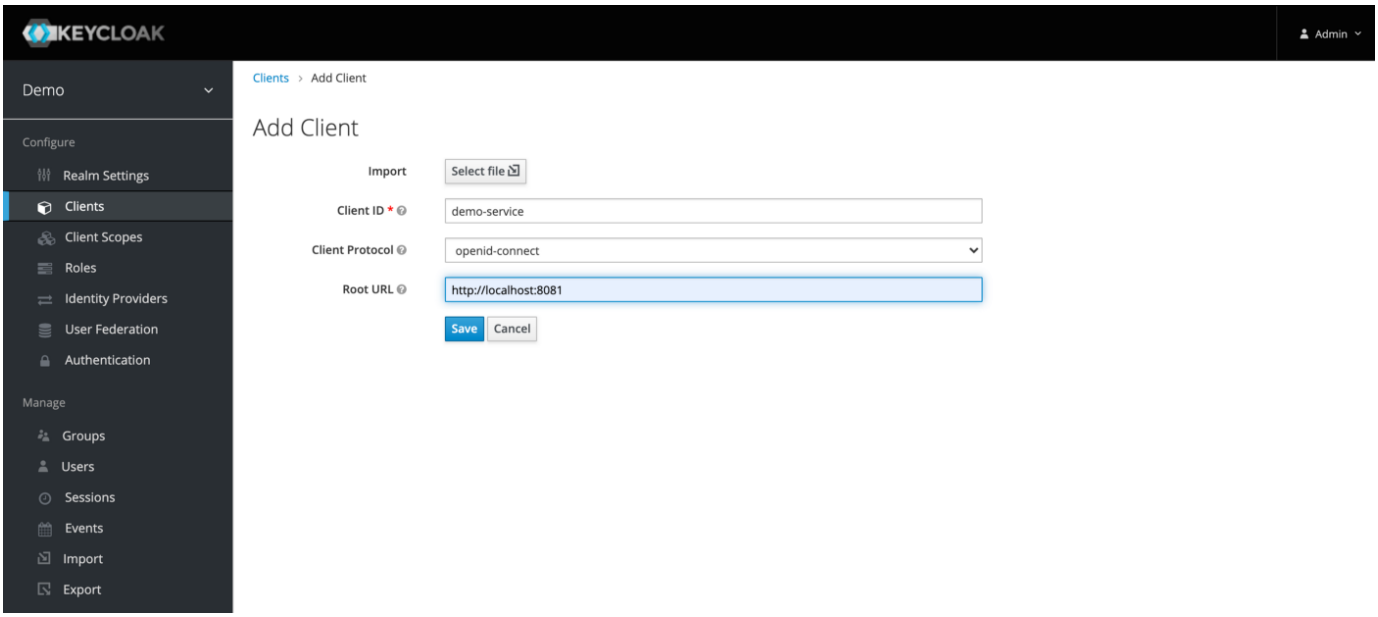
Create Realm

Create a client

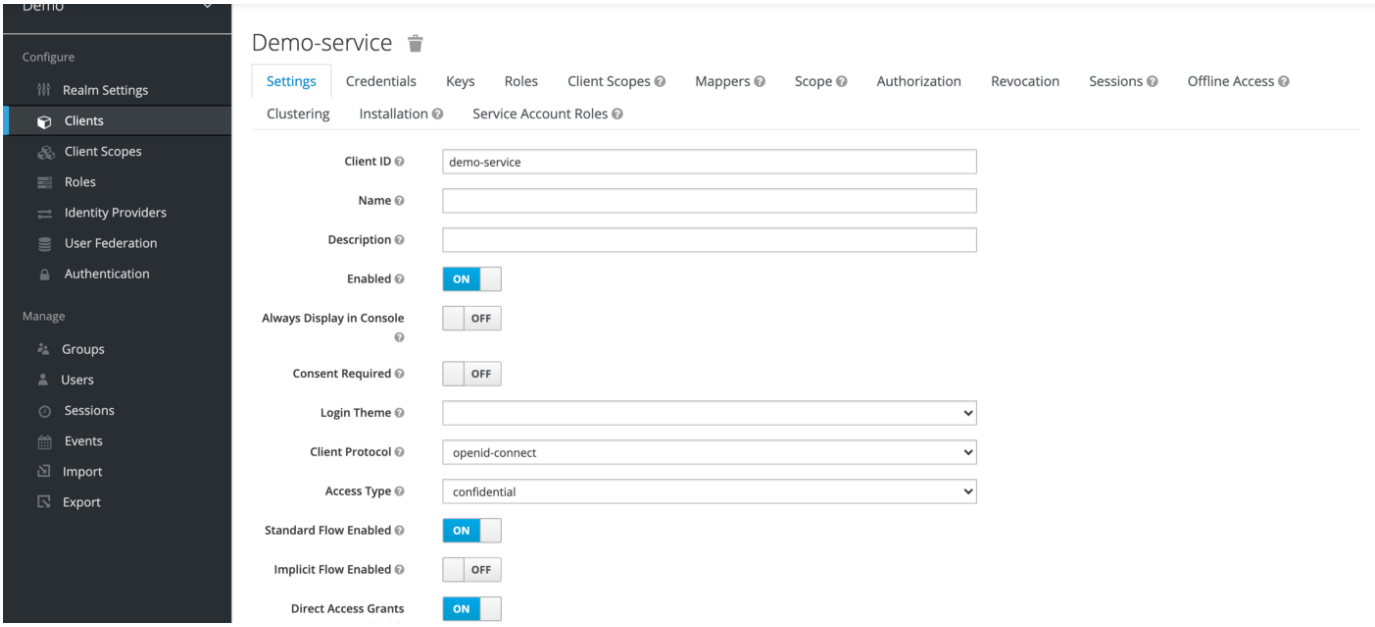
A client in keycloak, is a resource server i.e the server that hosts the resources that needs to be secured. In our case, it'll be the Spring Boot app we're going to create shortly.

Let's add a client to our realm.

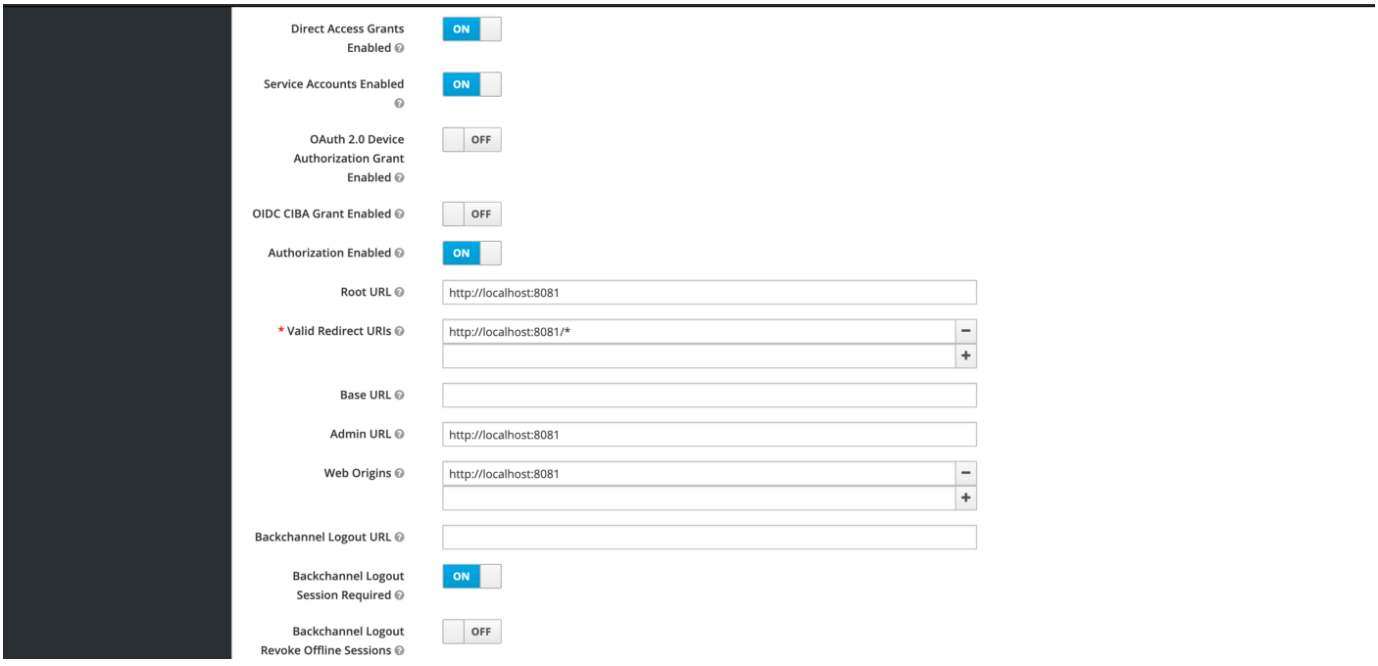
1. Click the **Clients** menu item.
2. Select **Create** from the upper right corner.
3. Call it `demo-service`.
4. Enter the root/base url of our microservice. Our Spring Boot app will be running on `http://localhost:8081`. So, let's enter `http://localhost:8081`
5. Save the client.
6. Click the **Settings** tab. change the **Access Type** from `public` to `confidential` and switch on **Authorization Enabled.**



Add Client



Settings for Demo-service client



Settings for Demo-service client

Create a role

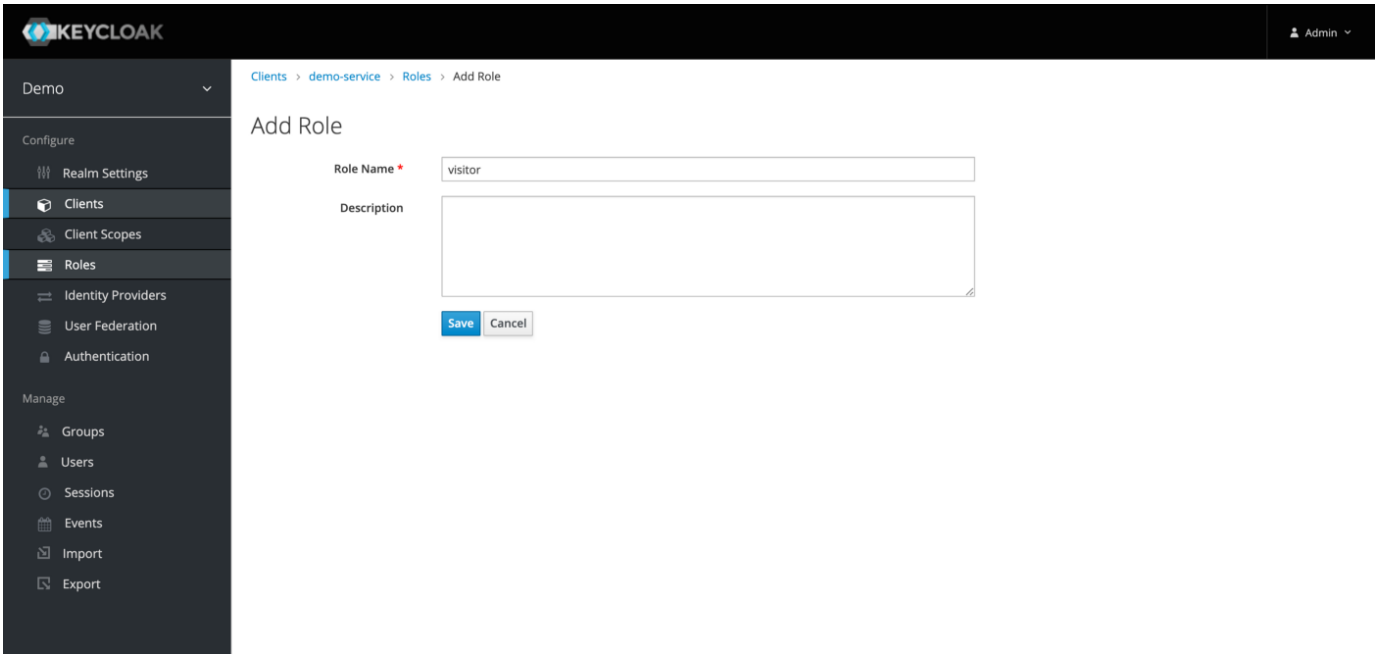
Currently, there are three types of roles in Keycloak:

- realm roles — they're global roles that are specific to the realm.
- client roles — they're specific to each client.
- composite roles — as the name implies, they're roles that have multiple roles associated with them.

In our example, let's create a client role dedicated to our Spring Boot app:

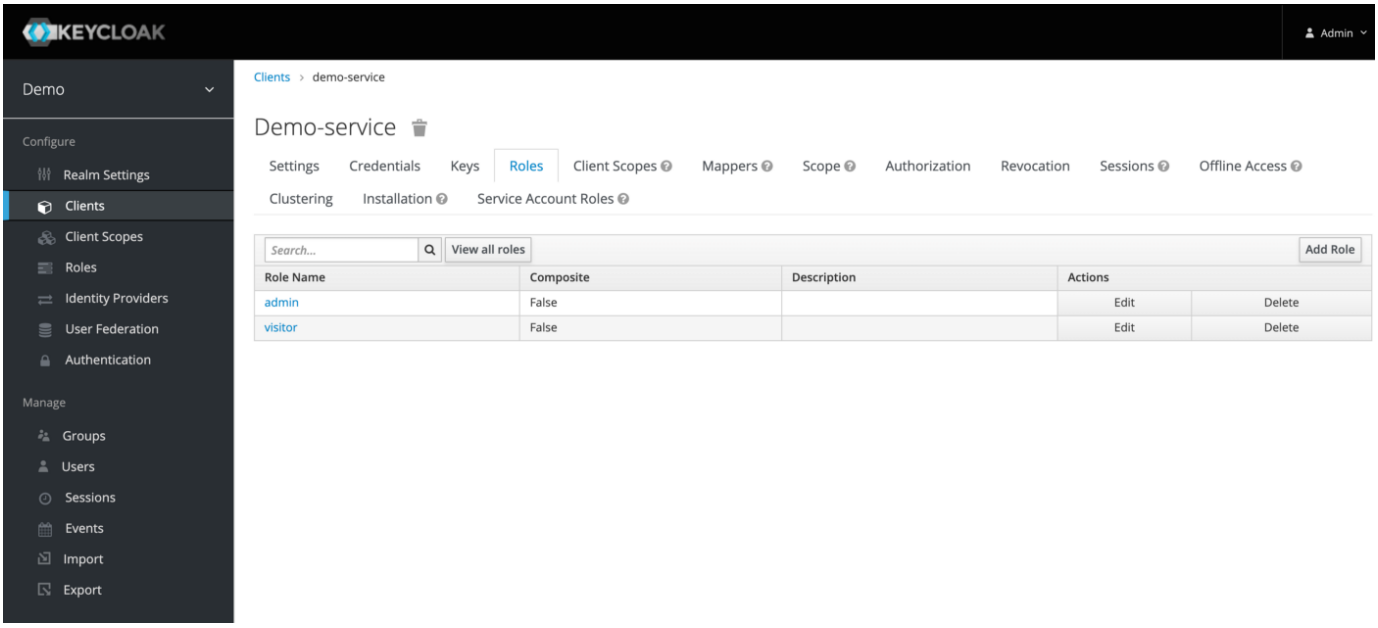
1. Select the `demo-service` client from the **Clients** menu item.
2. Click the **Roles** tab.
3. Click **Add Role**

5. Create another one called `admin` and save.



Add Role to Demo-service client

To verify that the roles have been successfully created, click the `demo-service` client, then the **Roles** tab and select **View all roles**:

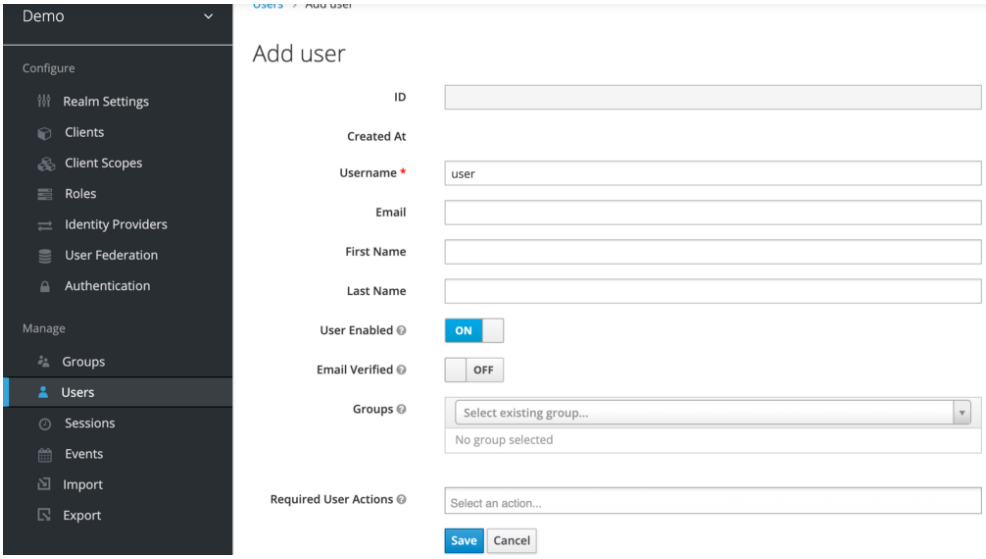


All roles for Demo-service

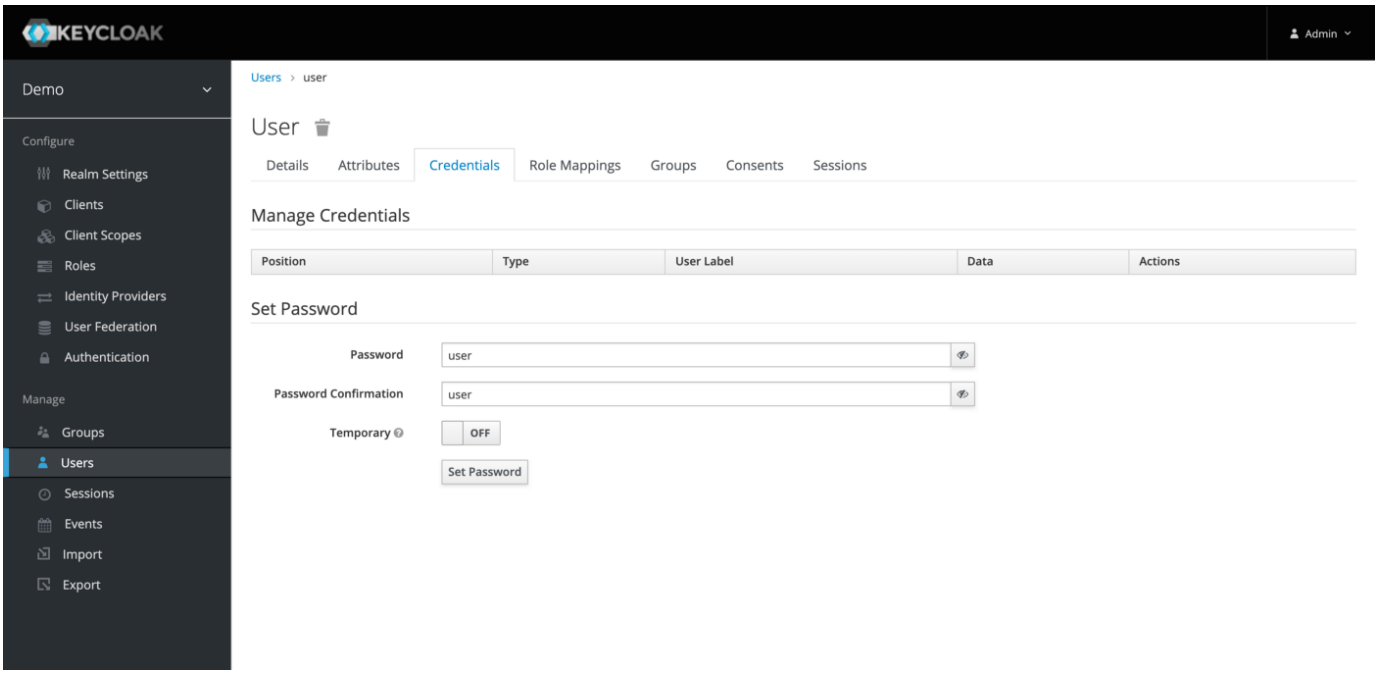
Create users

Now we need to assign users to the roles we've just created.

1. Click the **Users** menu item.
2. Select **Add User**.
3. Create a user called `user` and save it.
4. Click on the Credentials tab for `user` and create a password, To avoid asking the user to change their password after the first login, switch the **Temporary** option off and set the password.



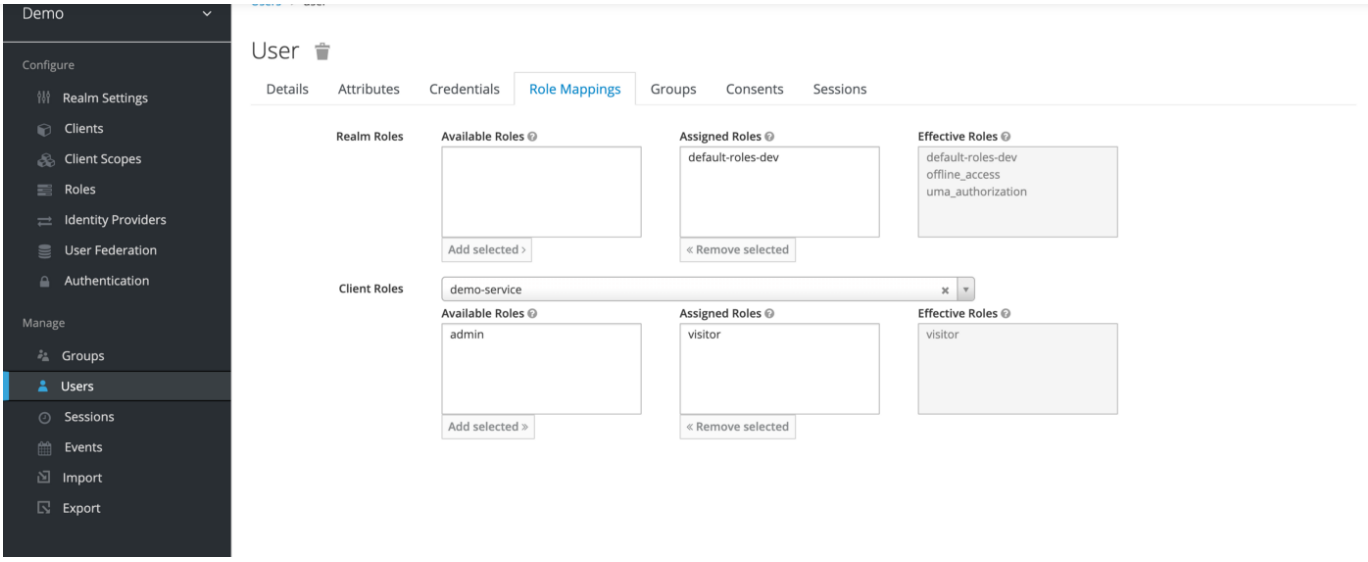
Add User



create password for user

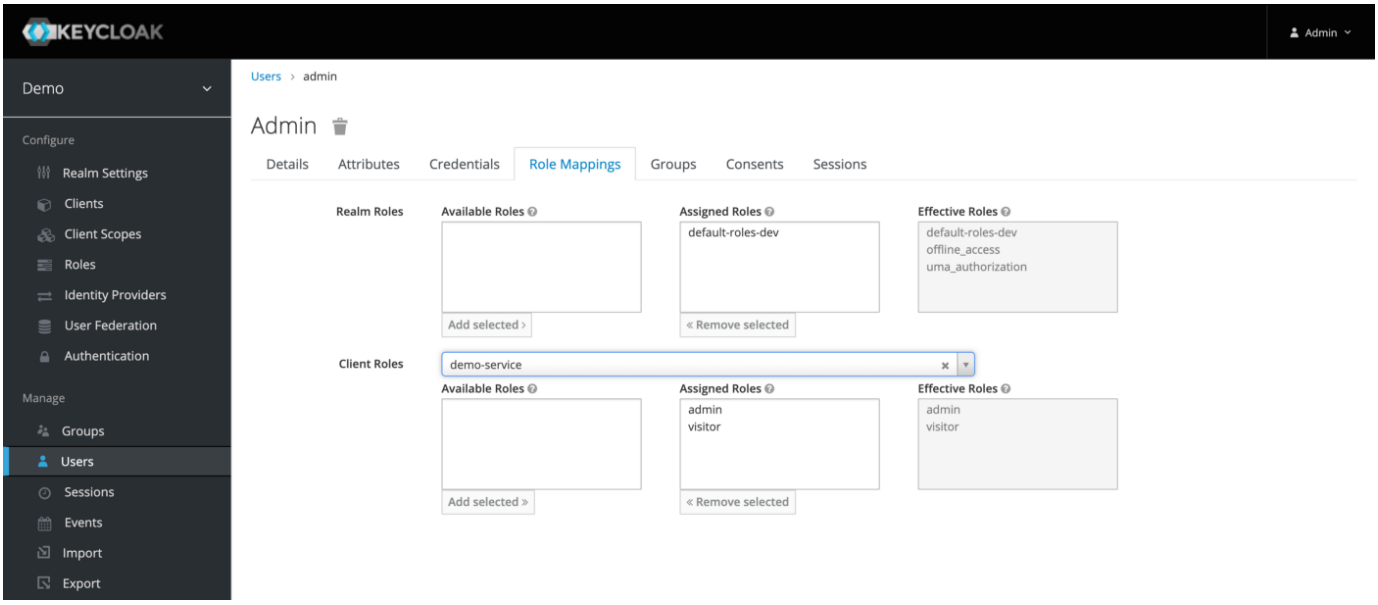
5. Add another user called `admin` and create `admin` credentials following same steps in 3 & 4

6. To demonstrate how permissions work, let's assign different roles to the users. For `user`, go to the `Role Mappings` tab. From the **Client Roles** drop-down menu, select `demo-service` and assign `visitor`.



Assign visitor role to user ‘user’

For `admin`, assign both `visitor` and `admin` roles.



Assign visitor and admin role to user ‘admin’

Create the Spring Boot App

We’ll create a very simple app that will display custom text to the user based on their role. In our case, we’ll distinguish `visitors` from `admins`

I’m using Maven to build this project. Create a New Maven Project in your favorite IDE. I’m Using IntelliJ. Alternatively, go to the Spring Initializer page to create a template Spring-Boot app.

We’ll be using the **Spring-Boot keycloak adapter** dependencies for authorizations with our authorization server already running on `http://localhost:8080`. **lombok** is also used here, to avoid dummy getters and setters and for faster development with less code. Add the following dependencies to your `pom.xml`

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

```
        <version>2.5.4</version>
        <relativePath/>
    </parent>
    <groupId>org.pcs</groupId>
    <artifactId>springboot-keycloak</artifactId>
    <version>1.0</version>
    <name>springboot-keycloak</name>
    <description>Spring Boot for basic integration with Keycloak</description>

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

    <dependencies>
        <!--            lombok-->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <!--        keycloak adapter-->
        <dependency>
            <groupId>org.keycloak</groupId>
            <artifactId>keycloak-spring-boot-starter</artifactId>
            <version>15.0.2</version>
        </dependency>
        <!--        Spring Security-->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
    </dependencies>

    <dependencyManagement>
      <!--        keycloak adapter-->
        <dependencies>
            <dependency>
                <groupId>org.keycloak.bom</groupId>
                <artifactId>keycloak-adapter-bom</artifactId>
                <version>15.0.2</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

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

</project>
```

pom.xml hosted with ❤ by GitHub view raw

Create the main application

Create a new class called `Application.java` and add a **RestTemplate** Bean method, which we will use in making http calls. Create as follows:

```java
package org.ugonna.springboot.keycloak;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.client.RestTemplateBuilder;
import org.springframework.context.annotation.Bean;
import org.springframework.web.client.RestTemplate;

@SpringBootApplication
public class Application {

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @Bean
    public RestTemplate restTemplate(RestTemplateBuilder builder) {
        return builder.build();
    }

}
```

Application.java hosted with ❤ by GitHub view raw

Set-up Keycloak properties

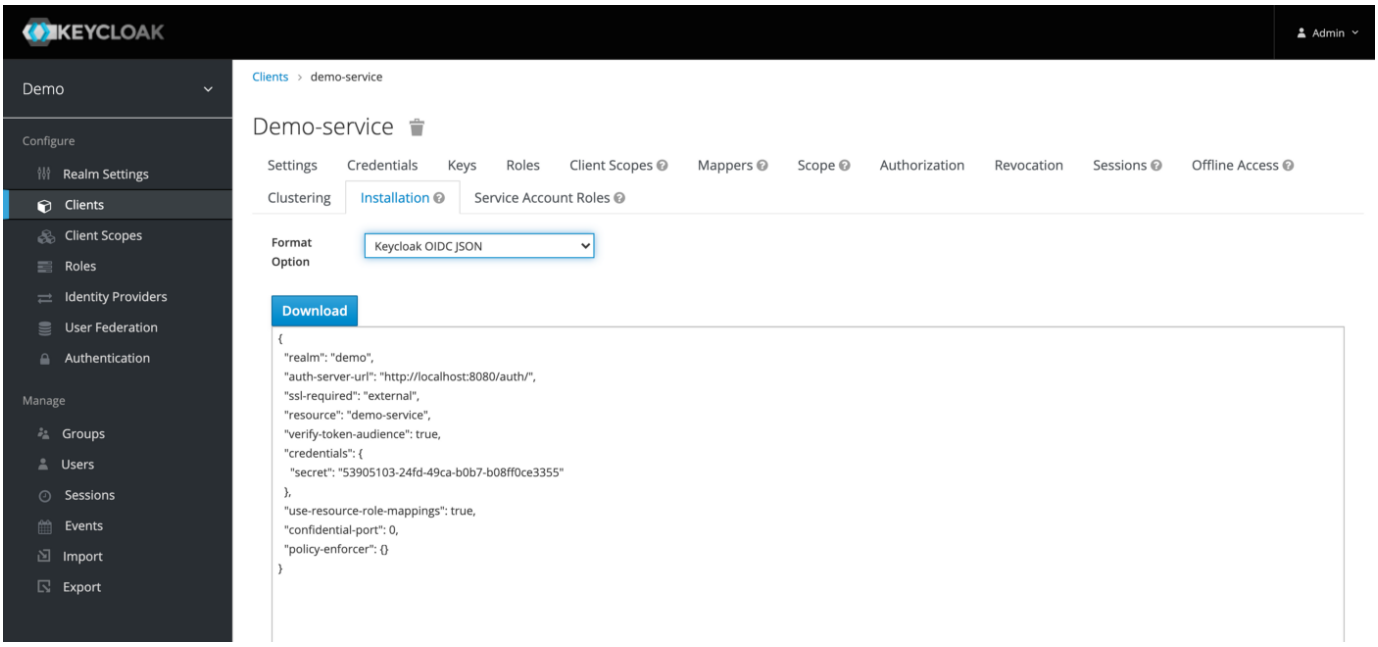
To integrate Keycloak authentication, we need to define a few settings.

Create an `application.properties` file under the `resources` folder and define the following properties. The application is explicitly set

```
keycloak.resource=demo-service
keycloak.auth-server-url=http://localhost:8080/auth
keycloak.ssl-required=external
keycloak.use-resource-role-mappings=true

#keycloak authentication properties
app.keycloak.login.url=http://localhost:8080/auth/realms/demo/protocol/openid-connect/token
app.keycloak.grant-type=password
app.keycloak.client-id = ${keycloak.resource}
app.keycloak.client-secret=53905103-24fd-49ca-b0b7-b08ff0ce3355

#Define authorization rules
#keycloak.security-constraints[0].authRoles[0]=visitor
#keycloak.security-constraints[0].securityCollections[0].patterns[0]=/visitor/*
#keycloak.security-constraints[0].authRoles[1]=admin
#keycloak.security-constraints[0].securityCollections[1].patterns[1]=/admin/*

server.port=8081
```

application.properties hosted with ❤ by GitHub view raw

The keycloak authorization properties are specific properties Keycloak expects from our application in order to do authorizations for us. This properties can be found on keycloak's dashboard : Clients > demo-service > installation > keycloak OIDC JSON. The `keycloak.use-resource-role-mappings` property when set to `true`, tells keycloak to use the defined client roles for authorizations, otherwise, keycloak uses the realm roles instead. The authentication properties are required to authenticate users (in-app) within our realm and provide an `AccessToken` upon successful Authentication. We can also decide to set up route rules here, or in the `SecurityConfig.java` class, as we'll see later on. The part is commented out as we will be using the latter.



Keycloak-OIDC-JSON

Create the Login Service

```java
package org.ugonna.springboot.keycloak.service;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.http.*;
import org.springframework.stereotype.Service;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;
import org.springframework.web.client.RestTemplate;
import org.ugonna.springboot.keycloak.dto.LoginRequest;
import org.ugonna.springboot.keycloak.dto.LoginResponse;


@Service
public class LoginService {

    @Value("${app.keycloak.login.url}")
    private String loginUrl;
    @Value("${app.keycloak.client-secret}")
    private String clientSecret;
    @Value("${app.keycloak.grant-type}")
    private String grantType;
    @Value("${app.keycloak.client-id}")
    private String clientId;

    @Autowired
    RestTemplate restTemplate;

    public ResponseEntity<LoginResponse> login (LoginRequest request)  {

        HttpHeaders headers = new HttpHeaders();
        headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);

        MultiValueMap<String, String> map = new LinkedMultiValueMap<>();
        map.add("username", request.getUsername());
        map.add("password", request.getPassword());
        map.add("client_id", clientId);
        map.add("client_secret", clientSecret);
        map.add("grant_type", grantType);

        HttpEntity<MultiValueMap<String, String>> httpEntity = new HttpEntity<>(map, headers);
        ResponseEntity<LoginResponse> loginResponse = restTemplate.postForEntity(loginUrl, httpEntity, LoginResponse.class);

        return ResponseEntity.status(200).body(loginResponse.getBody());

    }
}
```

LoginService.java hosted with ❤ by GitHub view raw

Create the Login Request and Response DTOs

Create a new class called `LoginRequest.java` and `LoginRespoonse.java` as follows :

```java
package org.ugonna.springboot.keycloak.dto;

import lombok.Data;

@Data
public class LoginRequest {
    private String username;
    private String password;
}
```

LoginRequest.java hosted with ❤ by GitHub view raw

```java
package org.ugonna.springboot.keycloak.dto;

import lombok.Data;

@Data
public class LoginResponse {
    private String access_token;
    private String refresh_token;
    private String expires_in;
    private String refresh_expires_in;
    private String token_type;
}
```

LoginResponse.java hosted with ❤ by GitHub view raw

Create the Controllers

Create a new class called `LoginController.java` for our login operation and `HelloController.java` for other operations within our application.

```java
package org.ugonna.springboot.keycloak.controller;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.servlet.HandlerMapping;
```

```
17
18  @RestController
19  public class LoginController {
20
21      Logger log = LoggerFactory.getLogger(LoginController.class);
22
23      @Autowired
24      LoginService loginService;
25
26      @PostMapping("login")
27      public ResponseEntity<LoginResponse> login (HttpServletRequest request,
28                                                  @RequestBody LoginRequest loginRequest) throws Exception {
29          log.info("Executing login");
30
31          ResponseEntity<LoginResponse> response = null;
32          response = loginService.login(loginRequest);
33
34          return response;
35      }
36  }
```

LoginController.java hosted with ❤ by GitHub view raw

```
1   package org.ugonna.springboot.keycloak.controller;
2   import org.springframework.web.bind.annotation.*;
3
4   @RestController
5   public class HelloController {
6
7       @GetMapping("/visitor")
8       public String getVisitor(@RequestHeader String Authorization) {
9           return "Hello visitor";
10      }
11
12      @GetMapping("/admin")
13      public String getAdmin(@RequestHeader String Authorization) {
14          return "Hello admin";
15      }
16
17      @GetMapping("/user")
18      public String getUsers(@RequestHeader String Authorization) {
19          return "Hello user";
20      }
21
22      @GetMapping("/random")
23      public String getRandomUser() {
```

We have 5 routes : `/login` , which will be used for authentication, `/visitor` , which will only be accessed by authenticated users with `visitor` role/privilege , `/admin` , which will only be accessed by authenticated users with `admin` role, `/user` , which will be accessed by any authenticated user (regardless of role) and `/random` , which will be accessed by anyone without authentication

Configure Keycloak and Route rules

Create a new class `SecurityConfig.java` , to set-up keycloak's spring security configurations and enable role-based authorizations to routes and other route rules.

```
1  package org.ugonna.springboot.keycloak.config;
2
3  import org.keycloak.adapters.springboot.KeycloakSpringBootConfigResolver;
4  import org.keycloak.adapters.springsecurity.KeycloakConfiguration;
5  import org.keycloak.adapters.springsecurity.authentication.KeycloakAuthenticationProvider;
6  import org.keycloak.adapters.springsecurity.config.KeycloakWebSecurityConfigurerAdapter;
7  import org.keycloak.adapters.springsecurity.filter.KeycloakAuthenticationProcessingFilter;
8  import org.springframework.beans.factory.annotation.Autowired;
9  import org.springframework.context.annotation.Bean;
10 import org.springframework.http.HttpMethod;
11 import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
12 import org.springframework.security.config.annotation.web.builders.HttpSecurity;
13 import org.springframework.security.core.authority.mapping.SimpleAuthorityMapper;
14 import org.springframework.security.core.session.SessionRegistryImpl;
15 import org.springframework.security.web.authentication.session.RegisterSessionAuthenticationStrategy;
16 import org.springframework.security.web.authentication.session.SessionAuthenticationStrategy;
17 import org.ugonna.springboot.keycloak.exception.CustomKeycloakAuthenticationHandler;
18 import org.ugonna.springboot.keycloak.exception.RestAccessDeniedHandler;
19
20 // Defines all annotations that are needed to integrate Keycloak in Spring Security
21 @KeycloakConfiguration
22 class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter {
23
24     @Autowired
25     RestAccessDeniedHandler restAccessDeniedHandler;
26
27     @Autowired
28     CustomKeycloakAuthenticationHandler customKeycloakAuthenticationHandler;
29
30     @Override
31     protected void configure(HttpSecurity http) throws Exception {
32         super.configure(http);
33         http.csrf().disable().cors().disable()
34                 .authorizeRequests()
35                 .antMatchers("/login", "/random").permitAll()
36                 .antMatchers("/visitor").hasRole("visitor")
37                 .antMatchers("/admin").hasRole("admin")
38                 .anyRequest()
39                 .authenticated();
40
41         //Custom error handler
```

```
47      public void configureGlobal( AuthenticationManagerBuilder auth) throws Exception {
48          KeycloakAuthenticationProvider keycloakAuthenticationProvider = keycloakAuthenticationProvider();
49          keycloakAuthenticationProvider.setGrantedAuthoritiesMapper(new SimpleAuthorityMapper());
50          auth.authenticationProvider(keycloakAuthenticationProvider);
51      }
52
53      // Use Spring Boot property files instead of default keycloak.json
54      @Bean
55      public KeycloakSpringBootConfigResolver KeycloakConfigResolver() {
56          return new KeycloakSpringBootConfigResolver();
57      }
58
59      // Register authentication strategy for public or confidential applications
60      @Bean
61      @Override
62      protected SessionAuthenticationStrategy sessionAuthenticationStrategy() {
63          return new RegisterSessionAuthenticationStrategy(new SessionRegistryImpl());
64      }
65
66      //Keycloak auth exception handler
67      @Bean
68      @Override
69      protected KeycloakAuthenticationProcessingFilter keycloakAuthenticationProcessingFilter() throws Exception {
70          KeycloakAuthenticationProcessingFilter filter = new KeycloakAuthenticationProcessingFilter(this.authenticationManagerBean());
71          filter.setSessionAuthenticationStrategy(this.sessionAuthenticationStrategy());
72          filter.setAuthenticationFailureHandler(customKeycloakAuthenticationHandler);
73          return filter;
74      }
75
76
77  }
```

SecurityConfig.java hosted with ❤ by GitHub view raw

We set up the route rules in the `configure` method. Our `/login` and `/random` routes will be accessed by any unauthenticated user, `/visitor` will be accessed by authenticated users with `visitor` role, `/admin` will be accessed by authenticated users with `admin` role. Finally every other route(s) within our application will require the user to just be authenticated (without necessarily having any roles assigned to them). In this case, `/user` route

The `configureGlobal` method tells spring security to use keycloak as the auth provider

The `keycloakConfigResolver` tells keycloak to look into our `application.properties` file for keycloak configurations (prefixed with keycloak).

We can handle Authentication and Authorizaton exceptions, and return custom error messages to our users via the RestAccessDeniedHandler and CustomKeycloakAuthenticationHandler classes (link to complete code will be shared at the end).

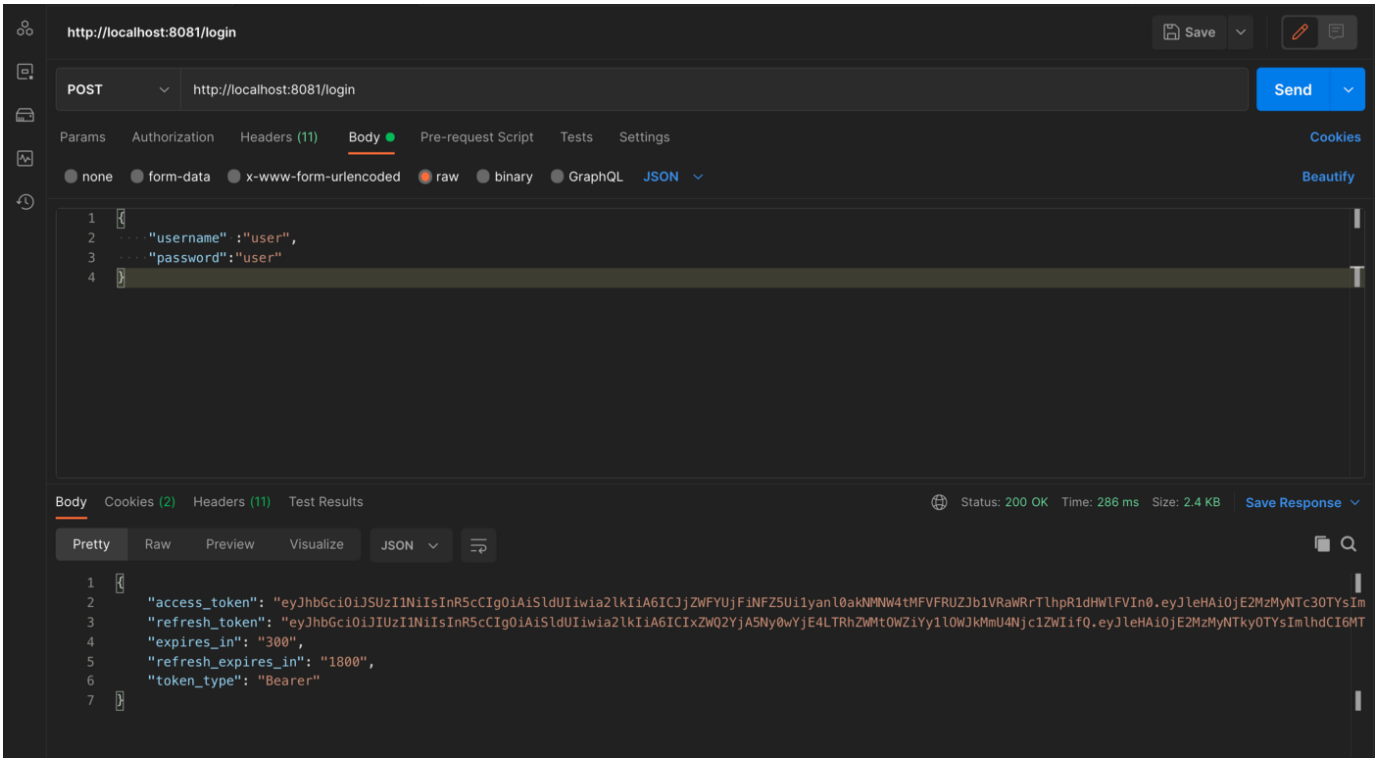
Test the Application

Start the app by executing this command in your Terminal:

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

Or by clicking on the play button at the top right of intellij or from the main `Application.java` class

Open up postman, and make the call to `/login` with the credentials for **user**. retrieve the `access_token`, which will be used to try and access routes within our application.

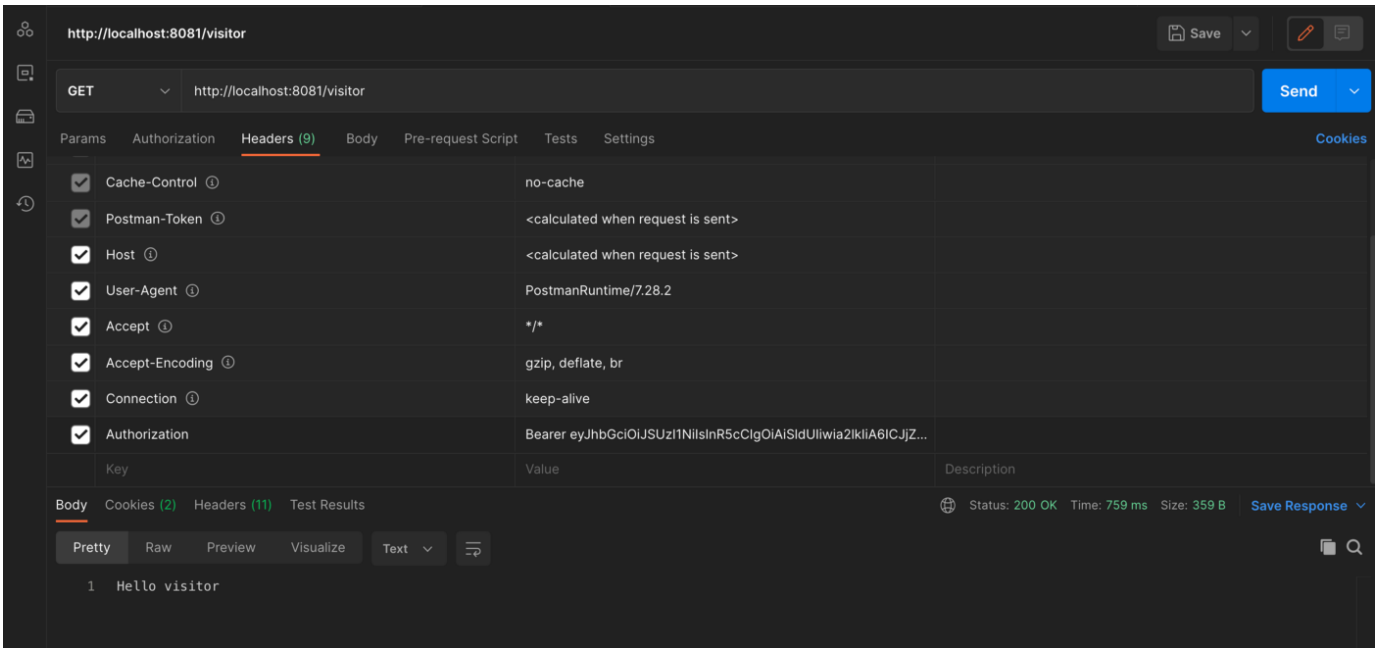


`access_token` : is a jwt used as a Bearer token when accessing protected routes.

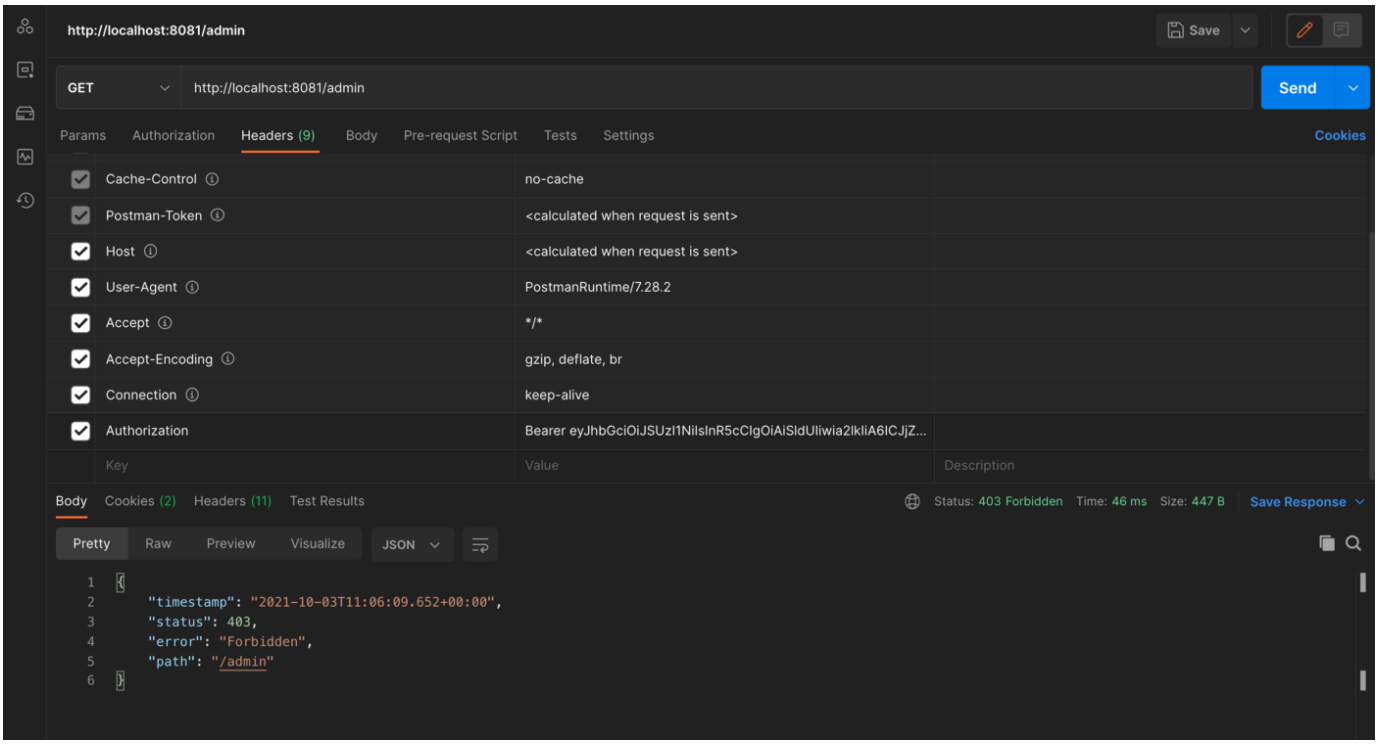
`refresh_token` : is also a jwt used to generate new token credentials without having the user login again (atleast till the refresh token expires).

`expires_in` : gives the expiry time of the access_token in seconds. This can be configured either at the realm level or client level or both, on the Keycloak dashboard. However, configuration at the client level takes precedence if set. For Realm level, From the

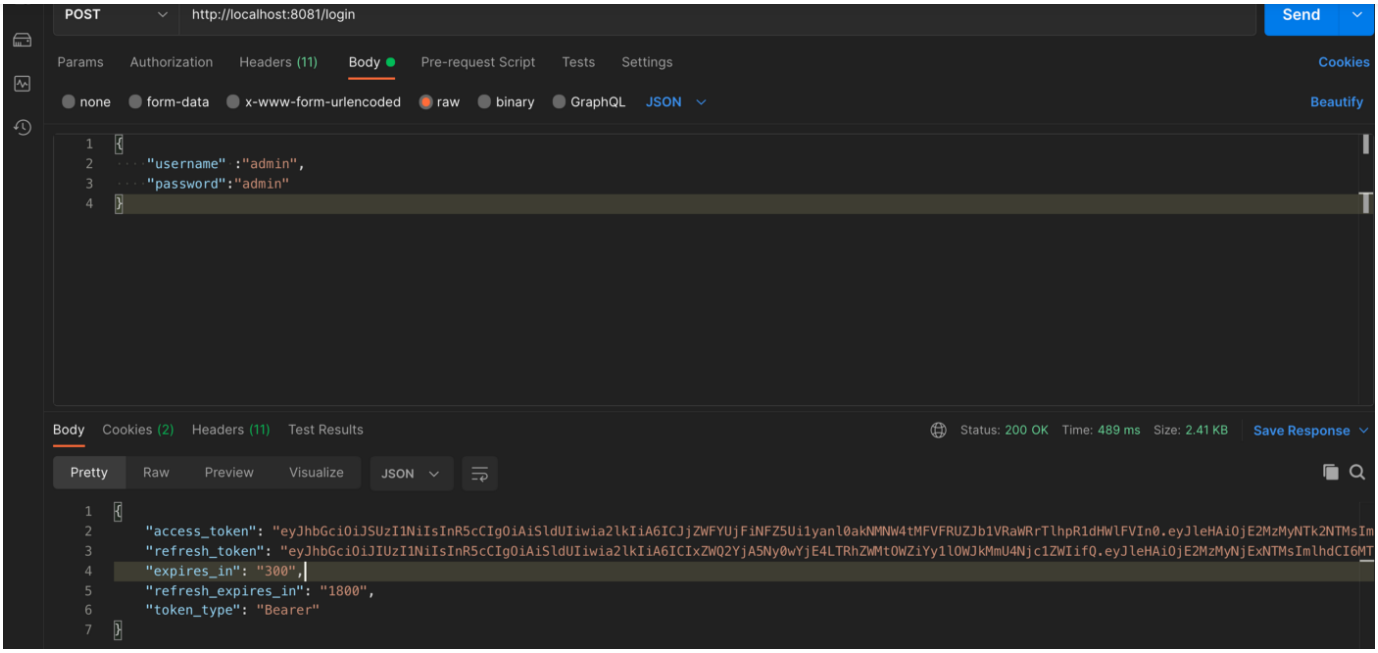
Let's try accessing the `/visitor` route, setting the Authorization header to Bearer ${access_token}



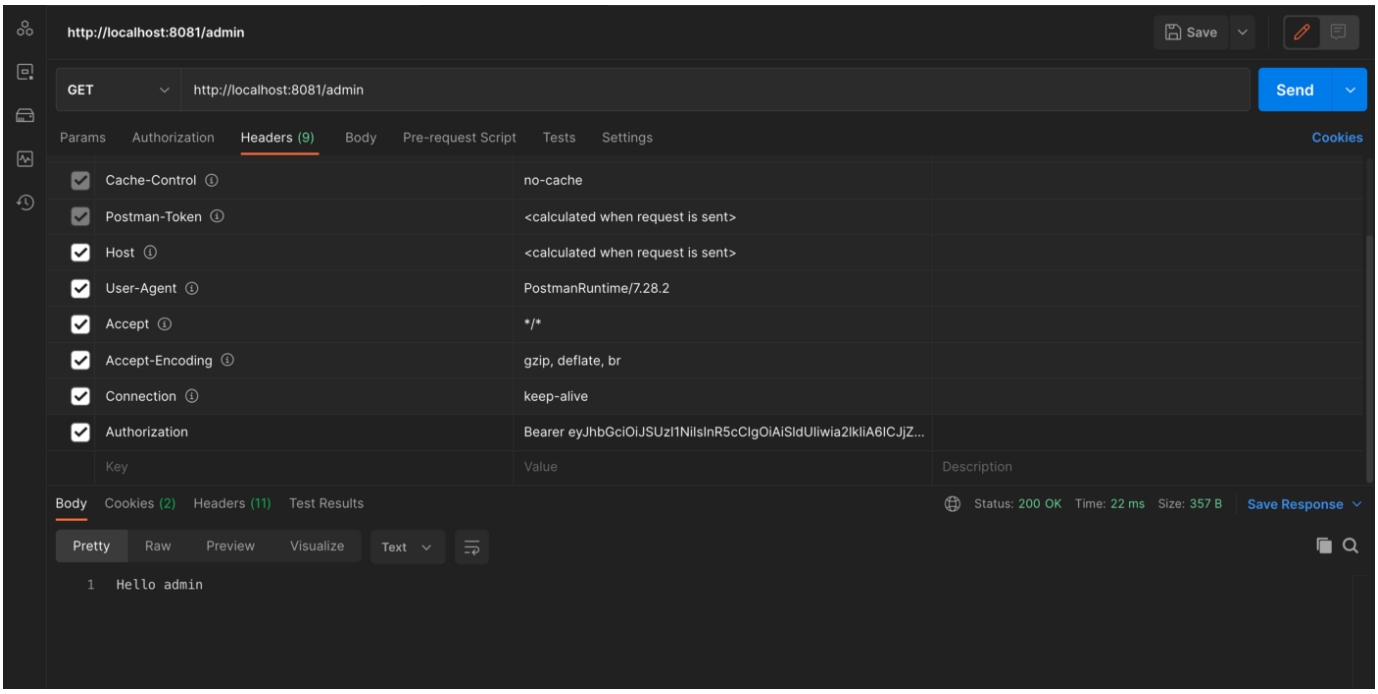
Accessing the `/admin` route with same access_token for **user** throws a 403 error because the user doesn't have `admin` privileges.



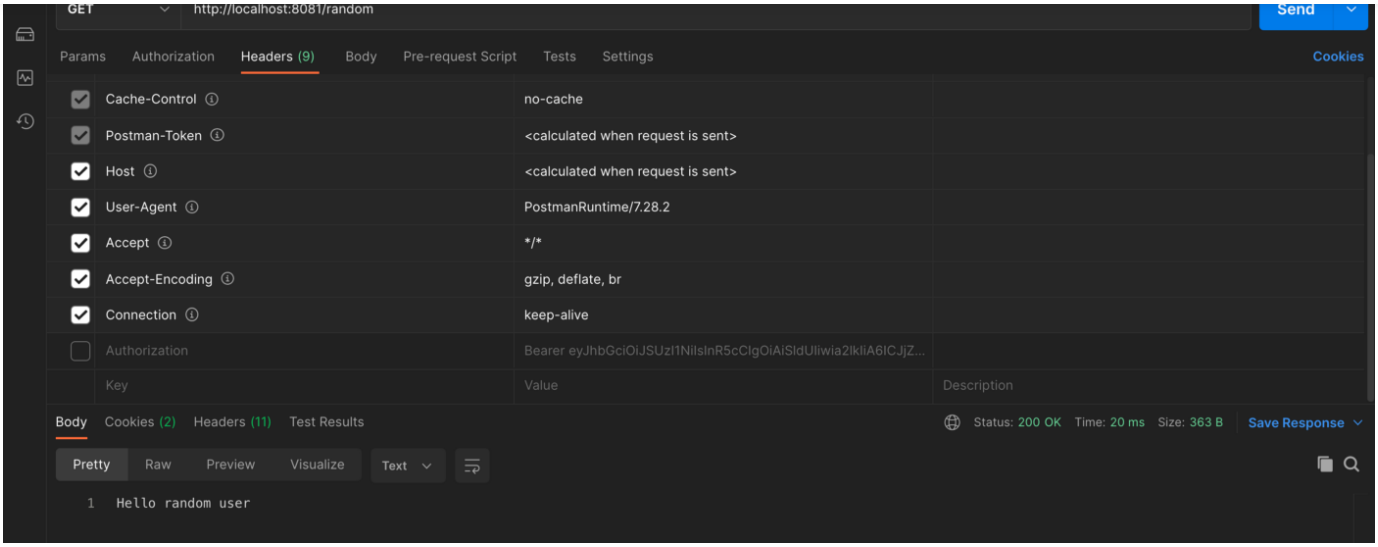
Now lets's login as **admin** and retrieve the admin access_token



We can now successfully access the `/admin` and also the `/visitor` routes because both `admin` and `visitor` roles have been assigned to the **admin** user.

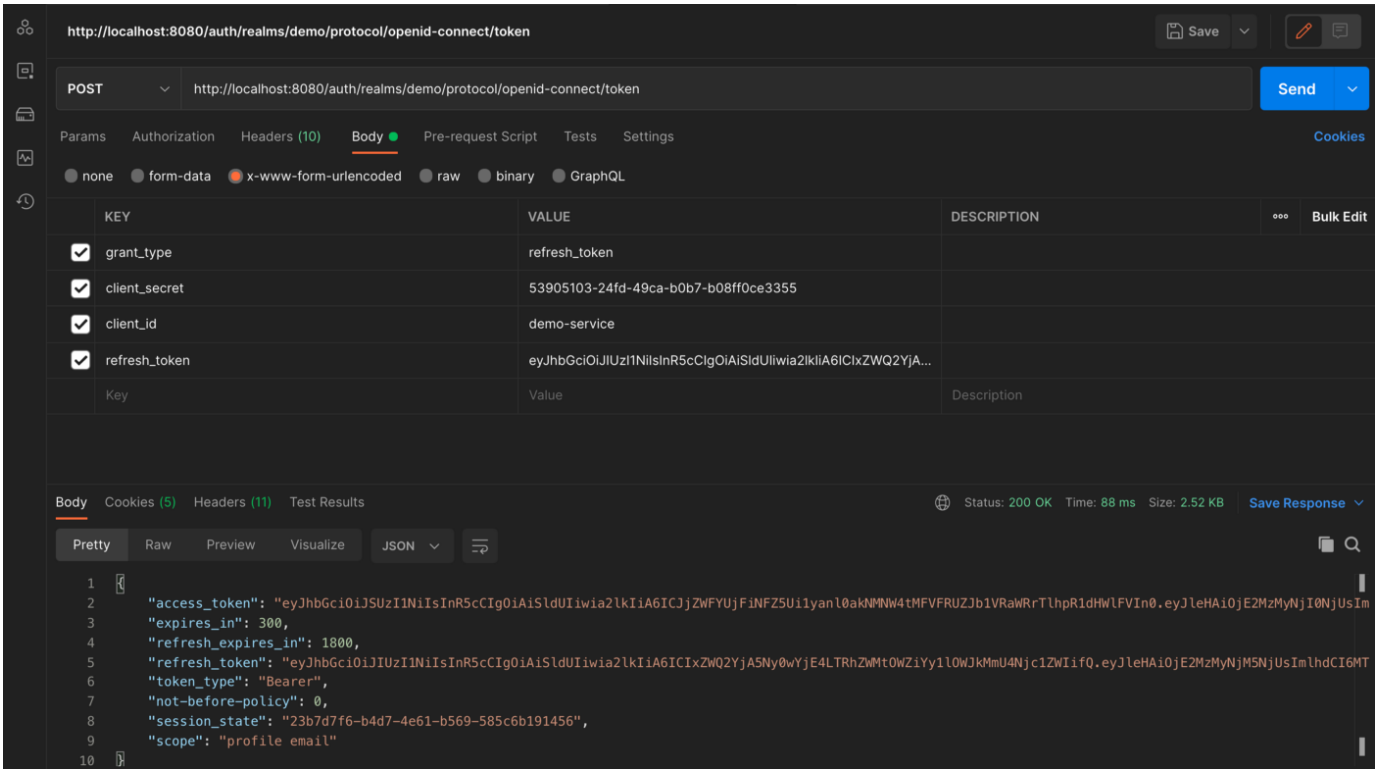


Both **user** and **admin** users will be able to access the `/user` route so far they have a valid access_token. Meanwhile `/random` route can be accessed by anyone without authentication.



We can see that our app works as expected.

SideNote : to use `refresh_token` to generate new token credentials, here's how it's done. You can decide to integrate a second login endpoint in the `LoginController.java` class say `/loginRefresh` and add the corresponding method in the `LoginService.java` class. However, I am calling the token endpoint directly from our authorization server to demonstrate how it's done.



login/authenticate via refresh_token

Conclusion

Congratulations, for making it to the end of this tutorial :). Now you should have a basic understanding of keycloak concepts and Spring Security and how to use it in securing your spring-boot microservice.

References

- https://betterprogramming.pub/how-to-authenticate-your-spring-boot-application-with-keycloak-1e9ccb5f2478
- https://www.keycloak.org/docs/latest/getting_started/