Keycloak Workshop

Table of Content

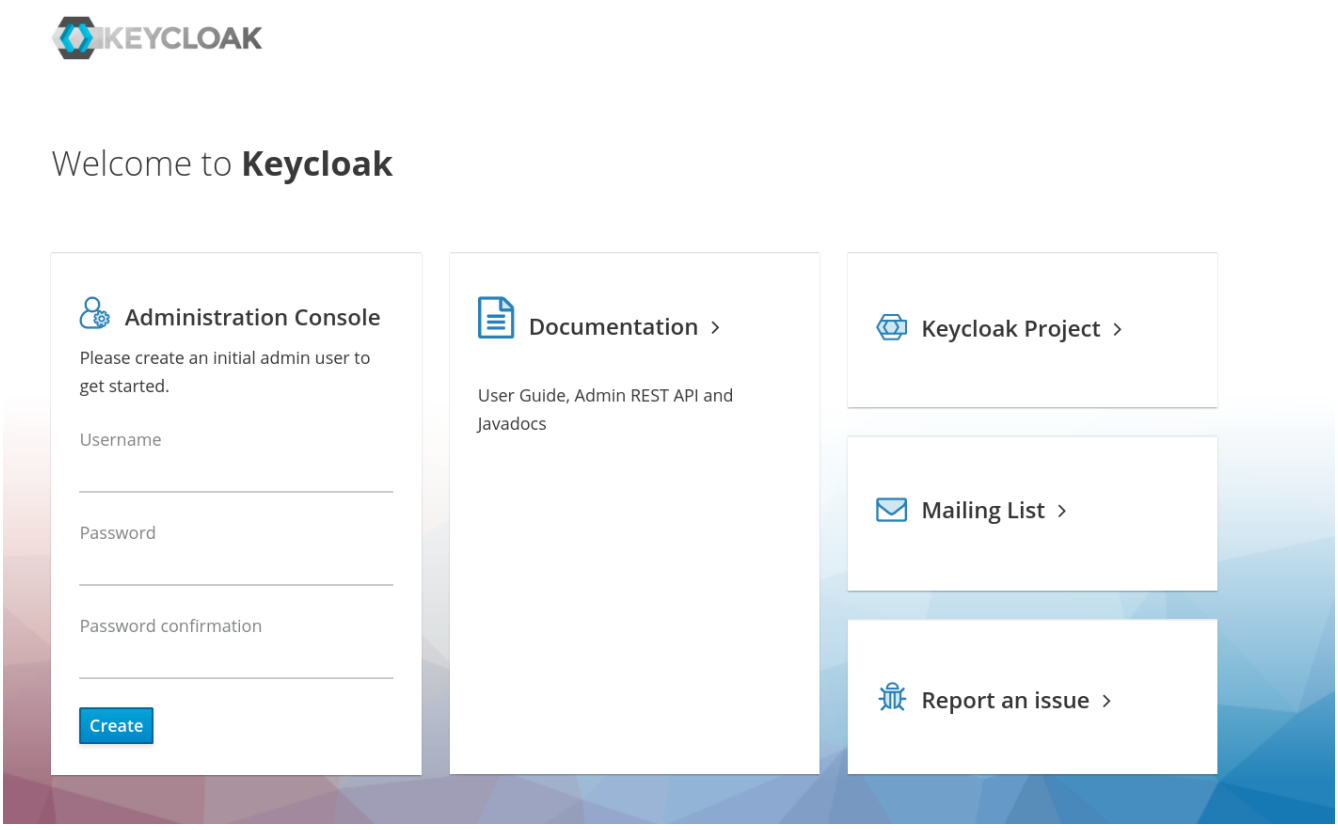
1. Installing the Keycloak Server

- Grab a server binary here (Standalone Server)
- Unzip it
- Run it `KC_LOCATION/bin/standalone.sh -Djboss.socket.binding.port-offset=100`

We run it with a port offset of 100 to make sure we don't conflict with 8080 for the apps we will be deploying.

2. Creating the initial Admin user

Browse to `http://localhost:8180/auth` and you will see this screen :



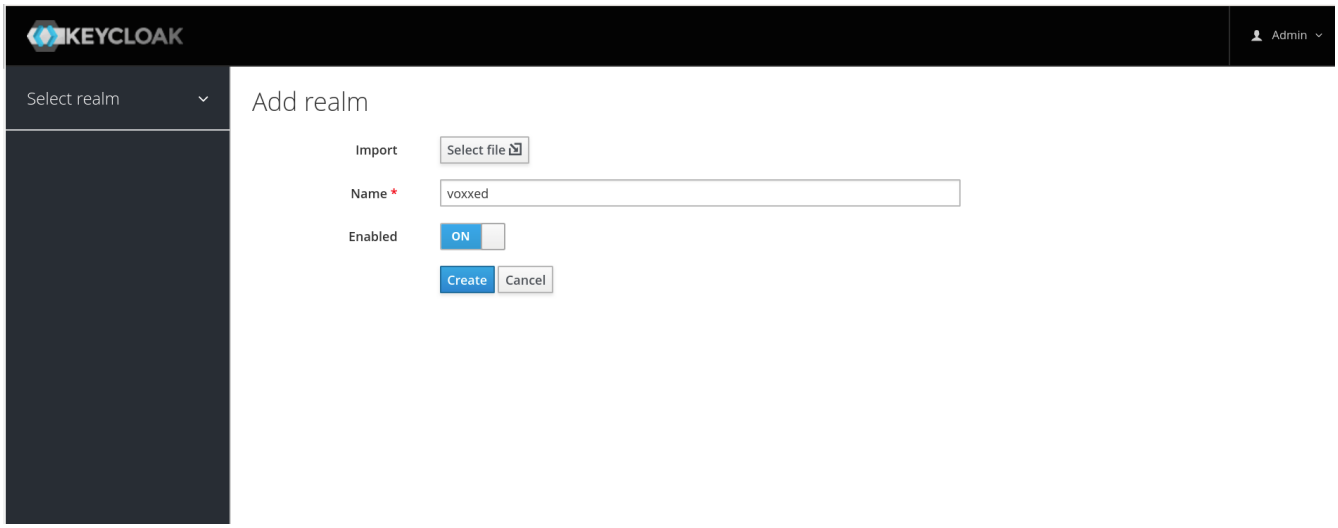
Create a new `admin` user with password `admin`.

Now you can access the Keycloak Web Console

3. Configuring a new Realm

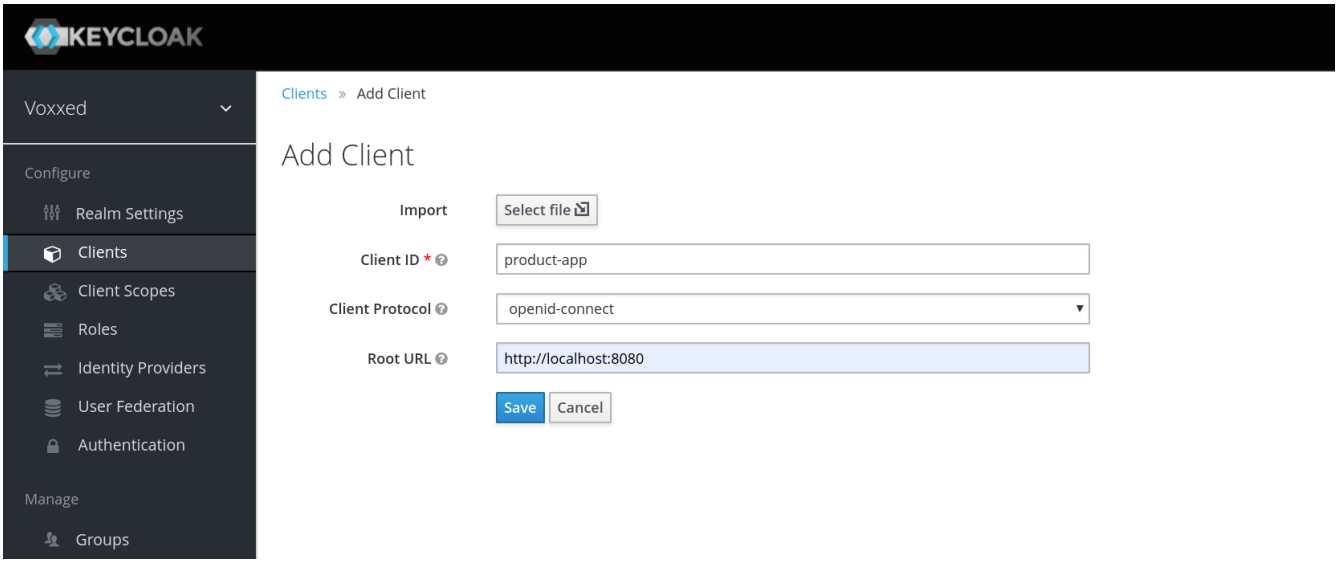
A good practice is to create a new Realm but you could keep doing this workshop on the `master` Realm.

You can name the realm as you want, in this workshop I will use `voxxed`



3.1. Creating the client, the role, and the user

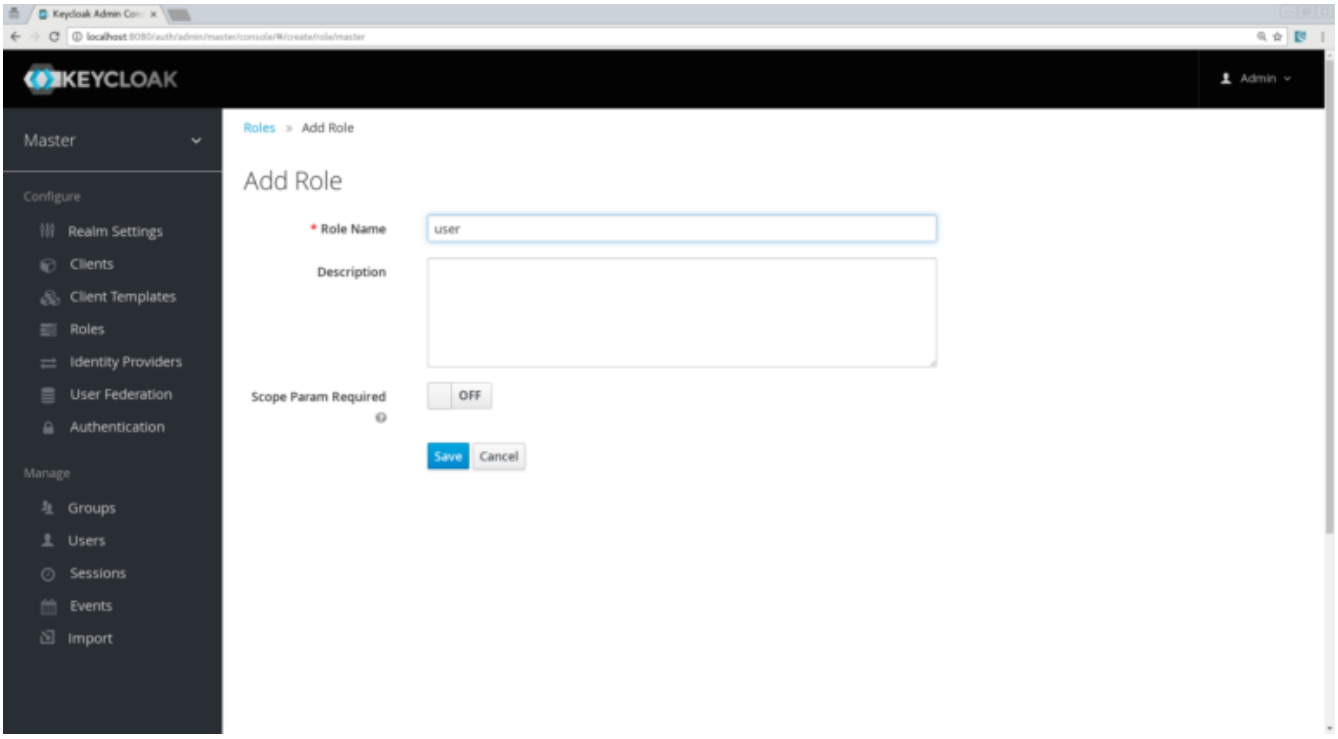
- Now we need to define a client, which will be our Spring Boot app. Go to the “Clients” section and click the “create” button. We will call our client 'product-app'



- On the next screen, we can keep the defaults and save.

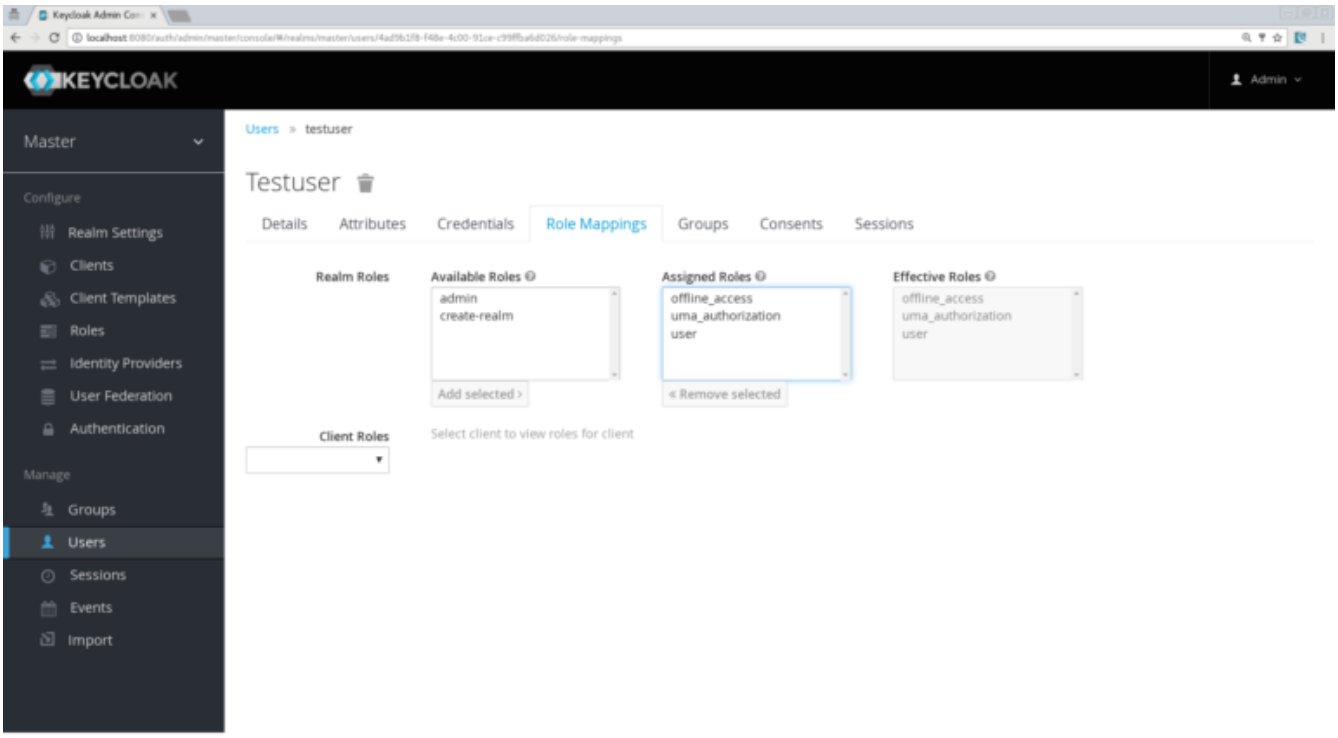
Don’t forget to save !

Now, we will define a role that will be assigned to our users, let’s create a simple role called `user` :



Now, we need to set his credentials, so go to the credentials tab of your user and choose a password, I will be using “password” for the rest of this workshop, make sure to turn off the “Temporary” flag unless you want the user to have to change his password the first time he authenticates.

Finally proceed to the `Role Mappings` tab and assign the role “user” :



4. Running the Spring Boot Product-app

Okay, we are ready to secure our first application, the `product-app` a simple Spring Boot Application using Spring MVC and Freemarker.

And let's create it from scratch !

Browse to The Spring Boot Initialzr and add the following dependencies :

- Web
- Freemarker

Import the application in your favorite IDE, I will be using IntelliJ.

Let's add one dependency in the `pom.xml` :

```
<dependency>
   <groupId>org.keycloak</groupId>
   <artifactId>keycloak-spring-boot-starter</artifactId>
   <version>6.0.1</version>
</dependency>
```

Our app will contain only 2 pages:

- An `index.html` which will be the landing page containing just a link to the product page.
- `products.ftl` which will be our product page template and will be only accessible for authenticated user.

Let's start by creating in simple `index.html` file in `/src/resources/static`:

```
<html>
<head>
    <title>My awesome landing page</title>
</head>
 <body>
   <h1>Landing page</h1>
   <a href="/products">My products</a>
 </body>
</html>
```

Let's create our Controller and Service classes now

You can create all the classes (Controller, etc) in the same Main file of your Spring Boot.

```
@Component
class ProductService {
   public List<String> getProducts() {
      return Arrays.asList("iPad","iPod","iPhone");
   }
}

@Controller
class ProductController {

   @Autowired ProductService productService;

   @GetMapping(path = "/products")
   public String getProducts(Model model){
      model.addAttribute("products", productService.getProducts());
      return "products";
   }

   @GetMapping(path = "/logout")
   public String logout(HttpServletRequest request) throws ServletException {
      request.logout();
      return "/";
   }
}
```

The final missing piece before you configure Keycloak is the product template (`products.ftl`), create this file in `sec/resources/templates` :

```
<#import "/spring.ftl" as spring>
<html>
<h1>My products</h1>
<ul>
<#list products as product>
    <li>${product}</li>
</#list>
</ul>
<p>
    <a href="/logout">Logout</a>
</p>
</html>
```

4.1. Configuring the Keycloak Adapter

Let’s start by adding the mandatory fields :

```
keycloak.auth-server-url=http://localhost:8180/auth
keycloak.realm=voxxed
keycloak.public-client=true
keycloak.resource=product-app
```

Now, in this same property file, let's add some security constraints :

```
keycloak.security-constraints[0].authRoles[0]=user
keycloak.security-constraints[0].securityCollections[0].patterns[0]=/products/*
```

Now we can run the app !

`mvn clean spring-boot:run` or directly from your IDE.

4.2. The Login Page

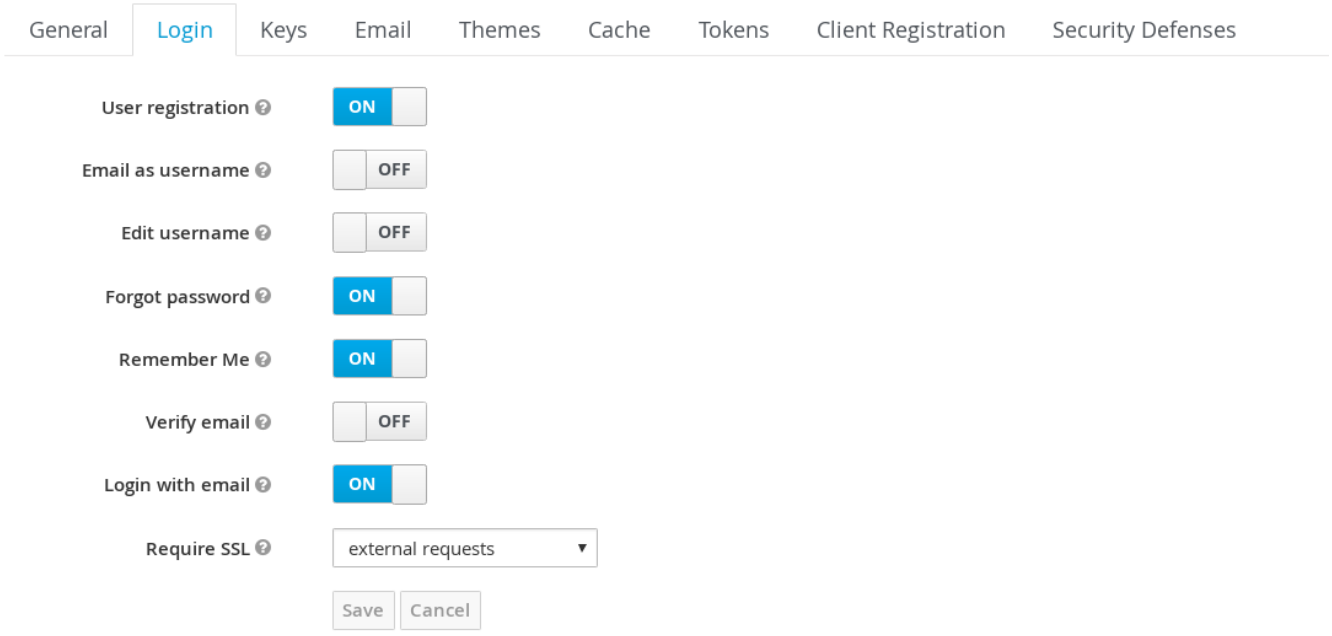
Browse to `http://localhost:8080` and click the `products` link, you should be redirected to the Keycloak Login Page.

Login with the user you create in the first step and after Keycloak should redirect you back to your application showing the list of products.

4.2.1. Enabling user registration

Click the `logout` link and go back to the Login page. Let's tweak our Login page using the Keycloak Web Console.

In the `Realm Settings` screen select the `Login` tab :



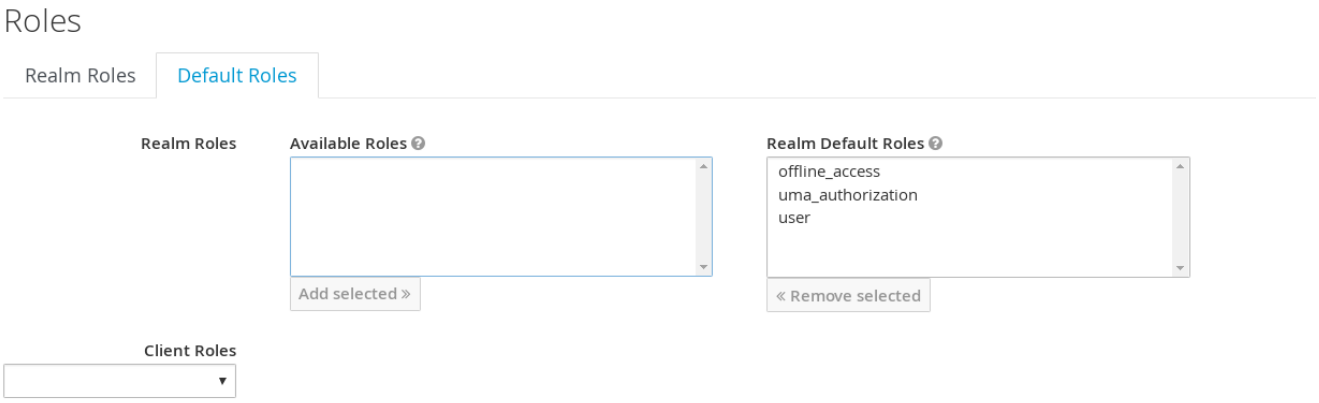
Turn on `User Registration`, go back to the Login page and refresh.

You can also "play" with the other options like `Remember me` etc ...

Click the `Register new user` link and fill in the form.

Notice that when you will be redirect to the application you will have an error. That's because you new user don't have the role `user`.

Make sure you add the role to your newly created user and let's also make sure the role user is added by default when an user is created :



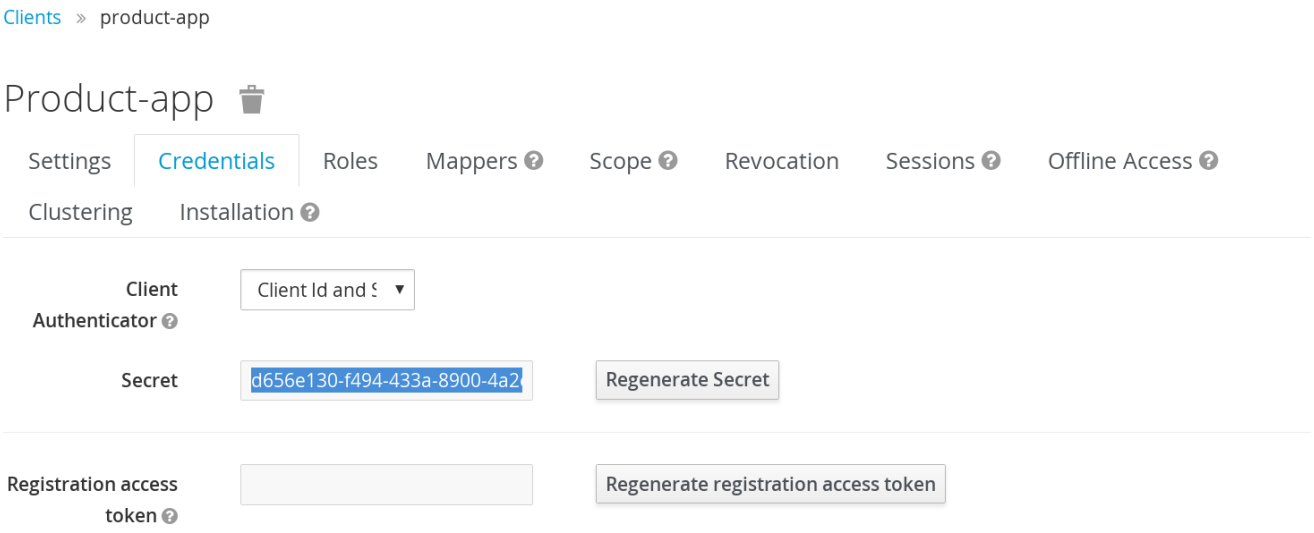
In the Role section, you have a `Default Roles` tab, from there you can choose the default roles.

4.2.2. Making the Client Confidential

Since we have a Server Side Application we can make it Confidential to add an extra security.

Go to your Keycloak Web Console and select your `product-app` client and change the `access type` to `confidential` and save.

You will notice now that you have an extra tab `Credentials` , go there are copy your `secret` :



Add this `secret` to your application properties :

```
keycloak.credentials.secret=your_secret
```

Also remove the property `keycloak.public-client=true`

Instead of using a `secret` it is also possible to use a signed JWT but this needs a bit more configuration. Check the documentation for more details.

4.2.3. Enabling Spring Security

Keycloak has also support for Spring Security and fits perfectly with the Spring Boot Adapter.

Let's start by adding the Spring Security bits :

```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-security</artifactId>
</dependency>
```

Creating a SecurityConfig class

Like any other project that is secured with Spring Security, a configuration class extending WebSecurityConfigurerAdapter is needed. Keycloak provides its own subclass that you can again subclass:

```
@KeycloakConfiguration
 class SecurityConfig extends KeycloakWebSecurityConfigurerAdapter
{
   /**
    * Registers the KeycloakAuthenticationProvider with the authentication manager.
    */
   @Autowired
   public void configureGlobal(AuthenticationManagerBuilder auth) throws Exception {
      KeycloakAuthenticationProvider keycloakAuthenticationProvider =
keycloakAuthenticationProvider();
      keycloakAuthenticationProvider.setGrantedAuthoritiesMapper(new
SimpleAuthorityMapper());
      auth.authenticationProvider(keycloakAuthenticationProvider);
   }

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

   /**
    * Defines the session authentication strategy.
    */
   @Bean
   @Override
   protected SessionAuthenticationStrategy sessionAuthenticationStrategy() {
      return new RegisterSessionAuthenticationStrategy(new SessionRegistryImpl());
   }

   @Override
   protected void configure(HttpSecurity http) throws Exception
   {
      super.configure(http);
      http
            .authorizeRequests()
            .antMatchers("/products*").hasRole("user")
            .anyRequest().permitAll();
   }

    @Bean
    @Override
    @ConditionalOnMissingBean(HttpSessionManager.class)
    protected HttpSessionManager httpSessionManager() {
        return new HttpSessionManager();
    }
}
```

In the property file we can now remove the security constraint (all properties that starts with` keycloak.security-constraints`) that we defined since it’s Spring Security that handles this now.

Restart the app and it should just work as before.

Injecting the Principal

Just like any other app secured with Spring Security you can easily inject the `Principal` in your controller :

```
@GetMapping(path = "/products")
public String getProducts(Principal principal, Model model){
   model.addAttribute("principal",principal);
   model.addAttribute("products", productService.getProducts());
   return "products";
}
```

And add this to your property file :

```
keycloak.principal-attribute=preferred_username
```

And in your template :

```
<h1>Hello ${principal.getName()}</h1>
```

5. Decomposing the monolith

Instead of returning a hard coded product list, let's create a new application that will serve this list.

5.1. Creating a quarkus app

```
mvn io.quarkus:quarkus-maven-plugin:0.15.0:create
```

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< org.apache.maven:standalone-pom >-------------------
[INFO] Building Maven Stub Project (No POM) 1
[INFO] --------------------------------[ pom ]---------------------------------
[INFO]
[INFO] --- quarkus-maven-plugin:0.14.0:create (default-cli) @ standalone-pom ---
Set the project groupId [org.acme.quarkus.sample]: org.sebi
Set the project artifactId [my-quarkus-project]: product-service
Set the project version [1.0-SNAPSHOT]:
Do you want to create a REST resource? (y/n) [no]: y
Set the resource classname [org.sebi.HelloResource]: org.sebi.ProductResource
Set the resource path  [/hello]: /products
Creating a new project in /home/sblanc/voxxedminsk/product-service
Configuration file created in src/main/resources/META-INF/application.properties
[INFO]
[INFO] Maven Wrapper version 0.5.3 has been successfully set up for your project.
[INFO] Using Apache Maven: 3.6.0
[INFO] Repo URL in properties file: https://repo.maven.apache.org/maven2
[INFO]
[INFO]
[INFO] ========================================================================================
[INFO] Your new application has been created in /home/sblanc/voxxedminsk/product-service
[INFO] Navigate into this directory and launch your application with mvn compile quarkus:dev
[INFO] Your application will be accessible on http://localhost:8080
[INFO] ========================================================================================
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  49.401 s
[INFO] Finished at: 2019-05-12T17:05:07+02:00
[INFO] ------------------------------------------------------------------------
➜  voxxedminsk
```

5.2. Adding the dependencies

Open your Quarkus application and add the following dependencies to your `pom.xml` :

```
<dependency>
   <groupId>io.quarkus</groupId>
   <artifactId>quarkus-resteasy-jsonb</artifactId>
</dependency>
<dependency>
   <groupId>io.quarkus</groupId>
   <artifactId>quarkus-keycloak</artifactId>
</dependency>
```

5.3. Implementing the Rest Endpoint

```
@Path("/products")
public class ProductResource {

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @RolesAllowed("user")
    public List<String> hello() {
        return Arrays.asList("Android", "Nokia");
    }
}
```

5.4. Creating a new Keycloak Client

- Create a new client and call it `product-service` as with root URL `http://localhost:8081`
- In the next screen , in `Access Type` , select `Confidential`
- In the `Credentials` tab copy the `secret` we will need it for the `application.properties`

5.5. Adding the properties

Go to `src/main/resources/` and open the `application.properties` file :

```
quarkus.keycloak.realm=voxxed
quarkus.keycloak.auth-server-url=http://localhost:8180/auth
quarkus.keycloak.resource=product-service
quarkus.keycloak.bearer-only=true
quarkus.keycloak.credentials.secret=secret_of_your_client
quarkus.http.port=8081
```

5.6. Update the test class

```
  @Test
    public void testHelloEndpoint() {
        given()
          .when().get("/products")
          .then()
             .statusCode(401);
    }
```

5.7. Run the product service

```
mvn clean compile quarkus:dev
```

5.7.1. Bonus - Run it as Native

```
mvn clean package -Pnative
cd target
./product-service-1.0-SNAPSHOT-runner
```

5.8. Updating the Product App

Now we need to modify our initial application so that it calls the `product-rest` service. We have to make sure it will pass the authorization bearer in the headers.

Luckily, the Keycloak Spring Security Adapter ships a really useful class, the `KeycloakRestTemplate` :

Let's update our Security Config class by adding this :

```
@Autowired
public KeycloakClientRequestFactory keycloakClientRequestFactory;

@Bean
@Scope(ConfigurableBeanFactory.SCOPE_PROTOTYPE)
public KeycloakRestTemplate keycloakRestTemplate() {
  return new KeycloakRestTemplate(keycloakClientRequestFactory);
}
```

Then, we can inject this bean in our service class :

```
@Autowired
private KeycloakRestTemplate template;

public List<String> getProducts() {
  ResponseEntity<String[]> response =
template.getForEntity("http://localhost:8081/products", String[].class);
  return Arrays.asList(response.getBody());
}
```

Restart the app and it should just work as before.

6. Adding a Web Application

Clone this repo : https://github.com/sebastienblanc/voxxed-apps

Now that we have a separate rest service, we can built a third application that will consume this service.

Let's see how a Pure Web App can be secured with Keycloak and consume a secured rest service.

From the apps repo browse to the `quarkus-front` folder.

For convenience, this app has been wrapped inside a Quarkus Application. But you can put this in any Web Server (Apache, Node, etc...)

6.1. Creating the `product-web` client

Again, we need to create a new client in the Keycloak Web Console with as base URL `http://localhost:8082`.

From the `Installation` tab grab the `keycloak.json` and add it to `src/resources/`

6.2. Enabling CORS support in the `product-rest` services

Before running our Web Application, we need first to enable CORS support in our `product-service`. We do that by adding a filter :

```
package org.sebi;

import javax.ws.rs.container.ContainerRequestContext;
import javax.ws.rs.container.ContainerResponseContext;
import javax.ws.rs.container.ContainerResponseFilter;
import javax.ws.rs.ext.Provider;
import java.io.IOException;

@Provider
public class CORSFilter implements ContainerResponseFilter {
    @Override
    public void filter(ContainerRequestContext requestContext,
ContainerResponseContext responseContext) throws IOException {
        responseContext.getHeaders().add("Access-Control-Allow-Origin", "*");
        responseContext.getHeaders().add("Access-Control-Allow-Headers", "origin,
content-type, accept, authorization");
        responseContext.getHeaders().add("Access-Control-Allow-Credentials", "true");
        responseContext.getHeaders().add("Access-Control-Allow-Methods", "GET, POST,
PUT, DELETE, OPTIONS, HEAD");
        responseContext.getHeaders().add("Access-Control-Max-Age", "1209600");
    }
}
```

6.3. Running the Web Application

Running the Web Application can be done with this command : `mvn clean compile quarkus:dev`

Access the Web App : `http://localhost:8082` , note that if you open this in a tab of a browser where you were already connected with the `product-app` you won't need to authenticate ;)

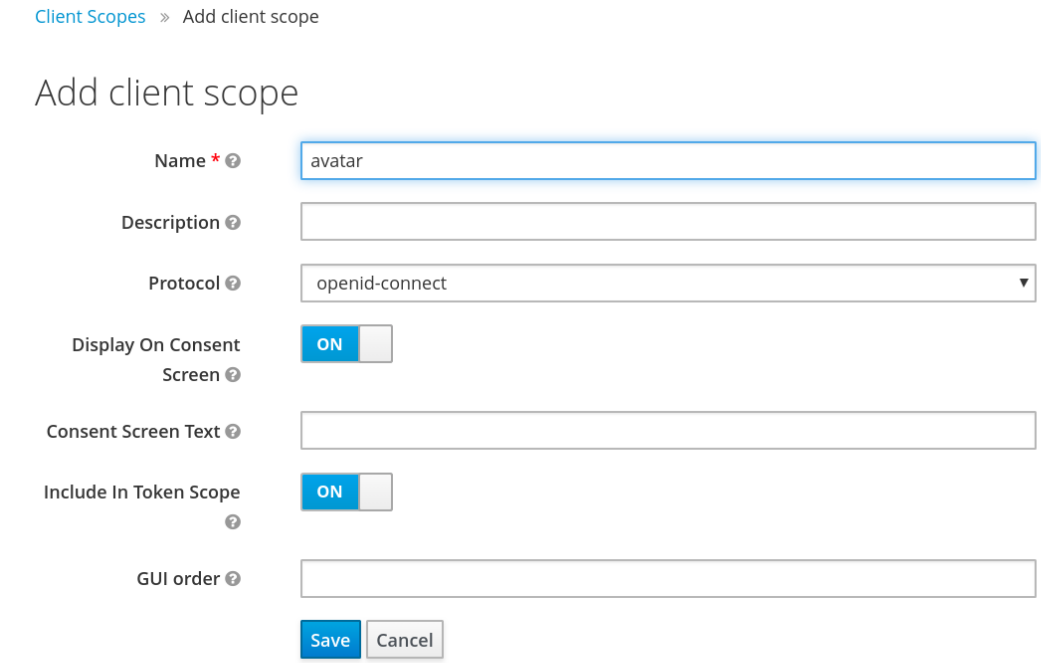
6.4. Discoveing the user account pages

You might have noticed an `account` button on the web app.

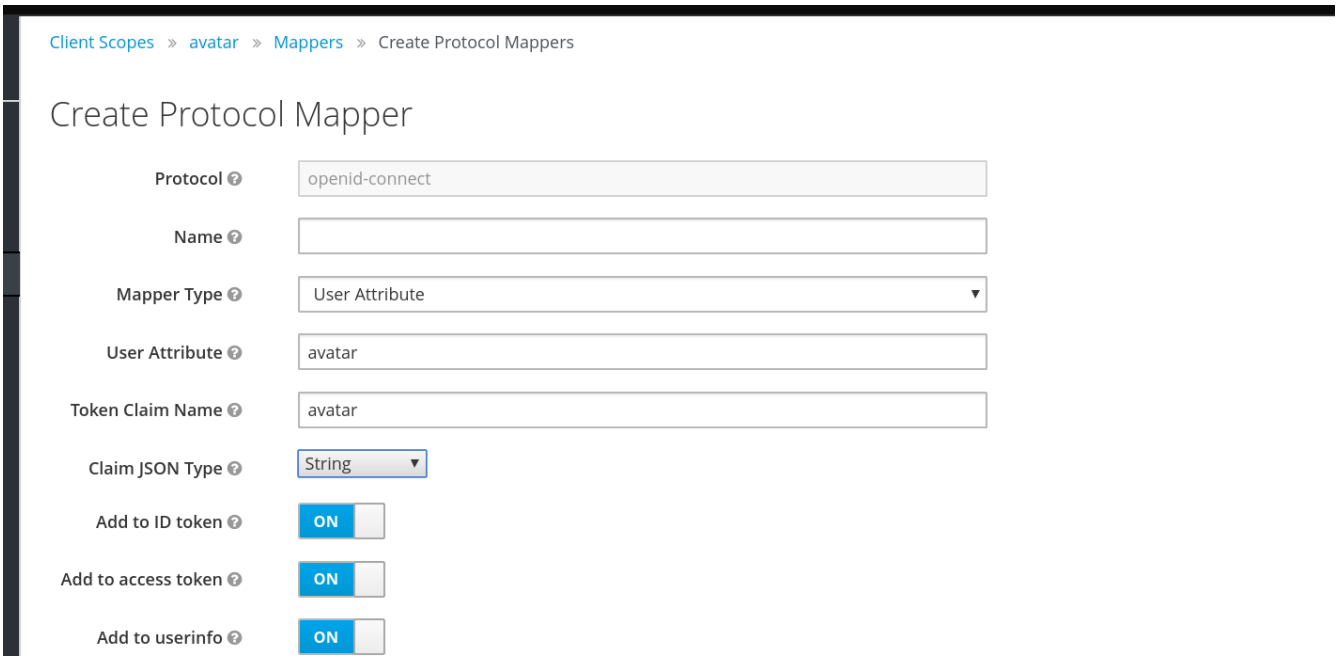
On this page you cam add some details to your profile but also set up 2FA for instance

6.5. Adding a new Client Scope

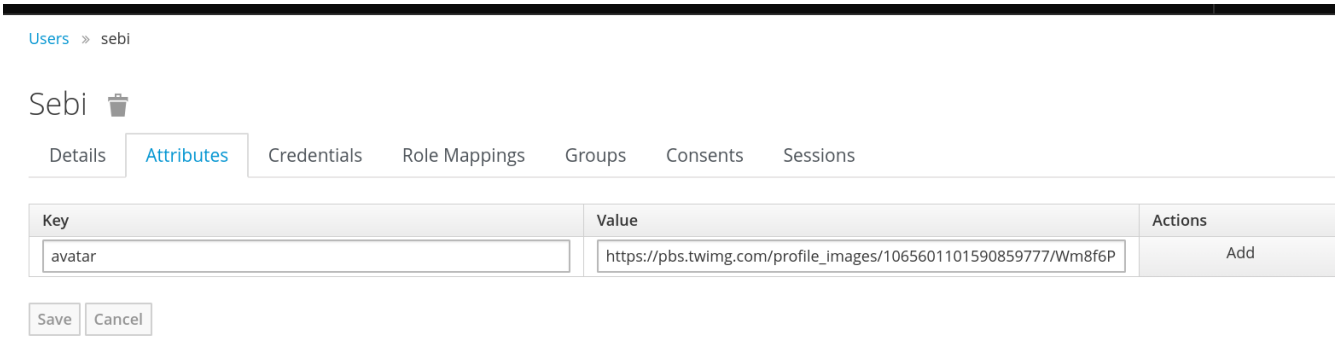
Let's add our avatar url to the token. And let's use a `Client Scope` for this since we might reuse it for other client :



Then create a mapper for this Client Scope :



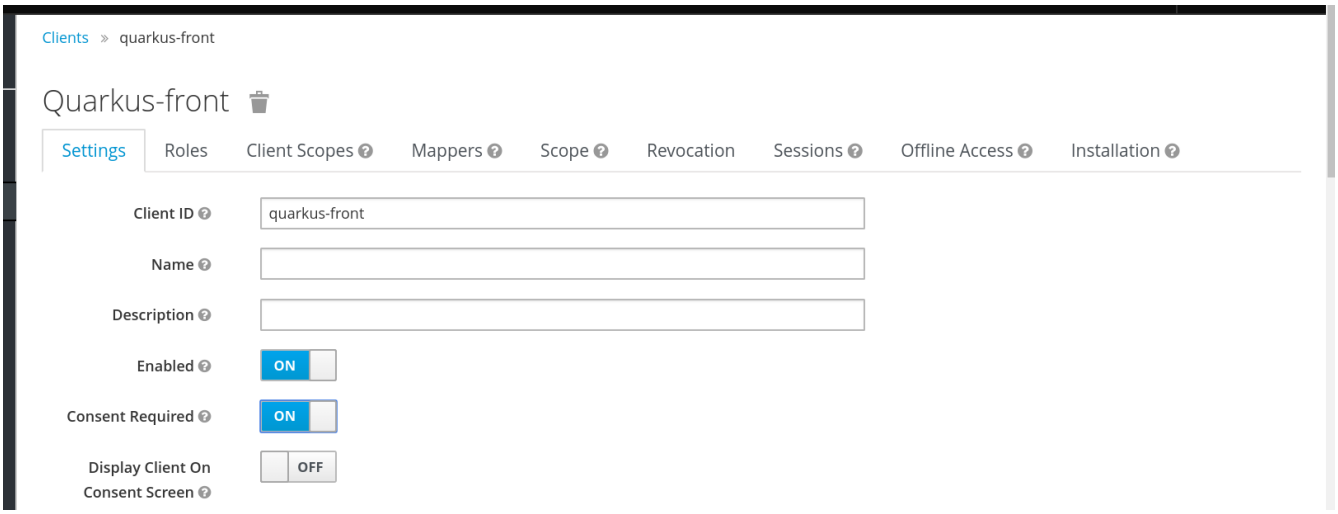
And inally add this attrivute to your user :



Don’t forget to save !

6.5.1. adding consent

Let’s add a consent screen :



Log out and Log in again !

7. Adding a third service with NodeJS

Let’s add now a third Microservice, build with NodeJS this time.

In the app repo, go to `service-nodejs`.

Install the app : `npm install`

Start the app : `npm start`

7.1. Adding a REST client to the `product-service`

Start by adding the rest client dependency :

```
 <dependency>
   <groupId>io.quarkus</groupId>
   <artifactId>quarkus-smallrye-rest-client</artifactId>
</dependency>
```

Now we define an interface for our client :

```
package org.sebi;

import org.eclipse.microprofile.rest.client.annotation.RegisterClientHeaders;
import org.eclipse.microprofile.rest.client.inject.RegisterRestClient;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;

@Path("/premium/products")
@RegisterRestClient
@RegisterClientHeaders
public interface PremiumService {

    @GET
    @Produces("application/text")
    String getPremium();
}
```

Now we can inject the client in our rest resource :

```
@Inject
@RestClient
PremiumService premiumService;
```

We update the method for retrieving the products :

```
@GET
@Produces(MediaType.APPLICATION_JSON)
@RolesAllowed("user")
public List<String> hello() {
   String premiumProduct = premiumService.getPremium();
   return Arrays.asList("Android", "Nokia", premiumProduct);
}
```

And finally we add some configuration to our `application.properties` :

```
org.sebi.PremiumService/mp-rest/url=http://localhost:3000
org.eclipse.microprofile.rest.client.propagateHeaders=Authorization
```

Notice the second property `propagateHeaders` , this make sure our rest client will reuse the `jwt access token` seamlessly.

Restart `product-service` and use the web app or the `product-app` to call the service again. You should see a new produt coming from the NodeJS service.

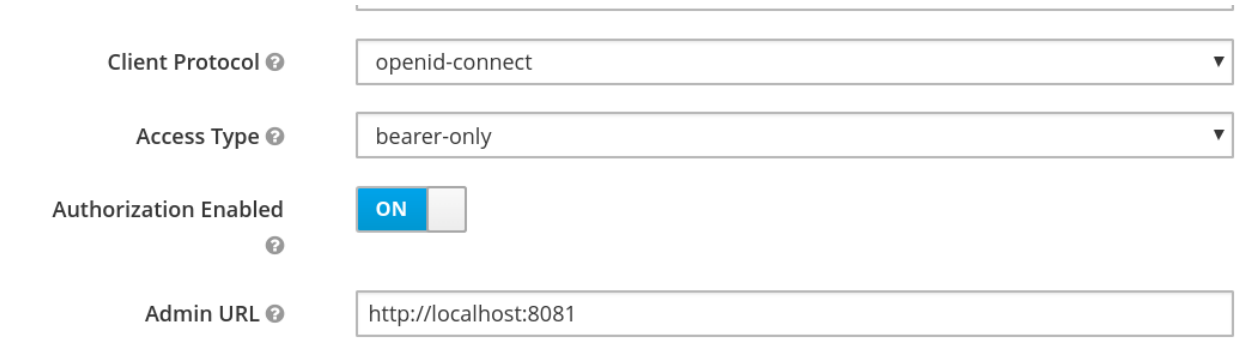
8. Adding the Authorization layer (AuthZ)

Until now we have done authentication with some basic RBAC (Role Based Access Control) but Keycloak also comes with a really complete authz layer.

To start, let's move the RBAC from the app to Keycloak.

8.1. Enabling authorization in Keycloak

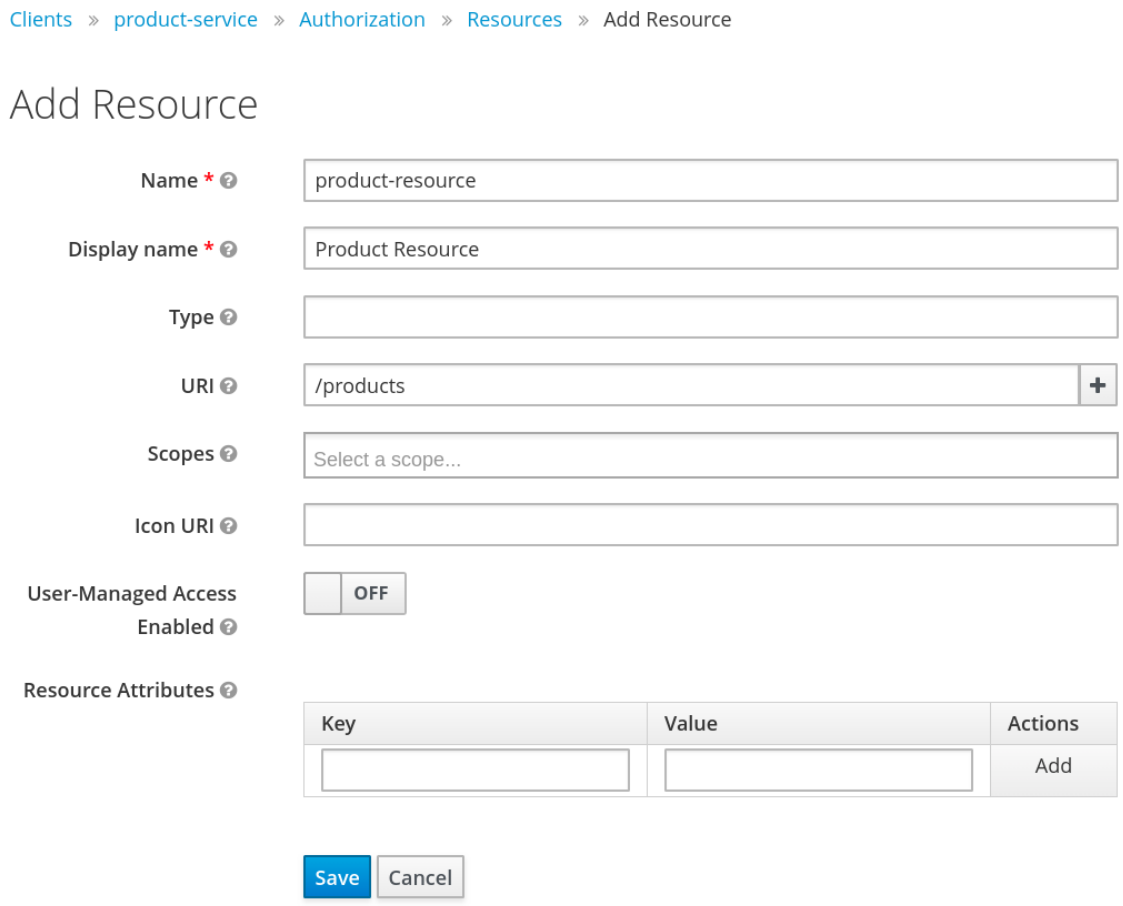
Go to the `product-service` and switch the authorization :



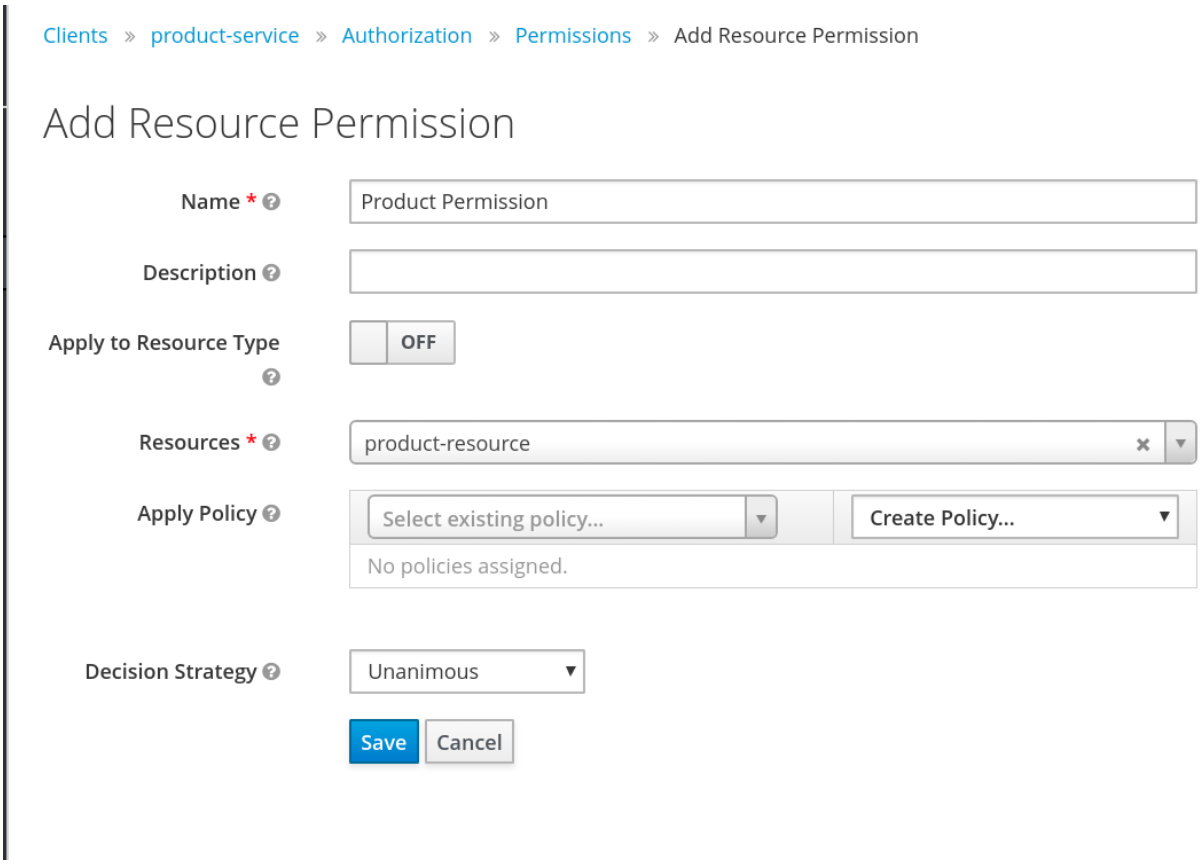
Now go to the `Authorization` tab and hen select `resources`.

8.2. Creating a new resource

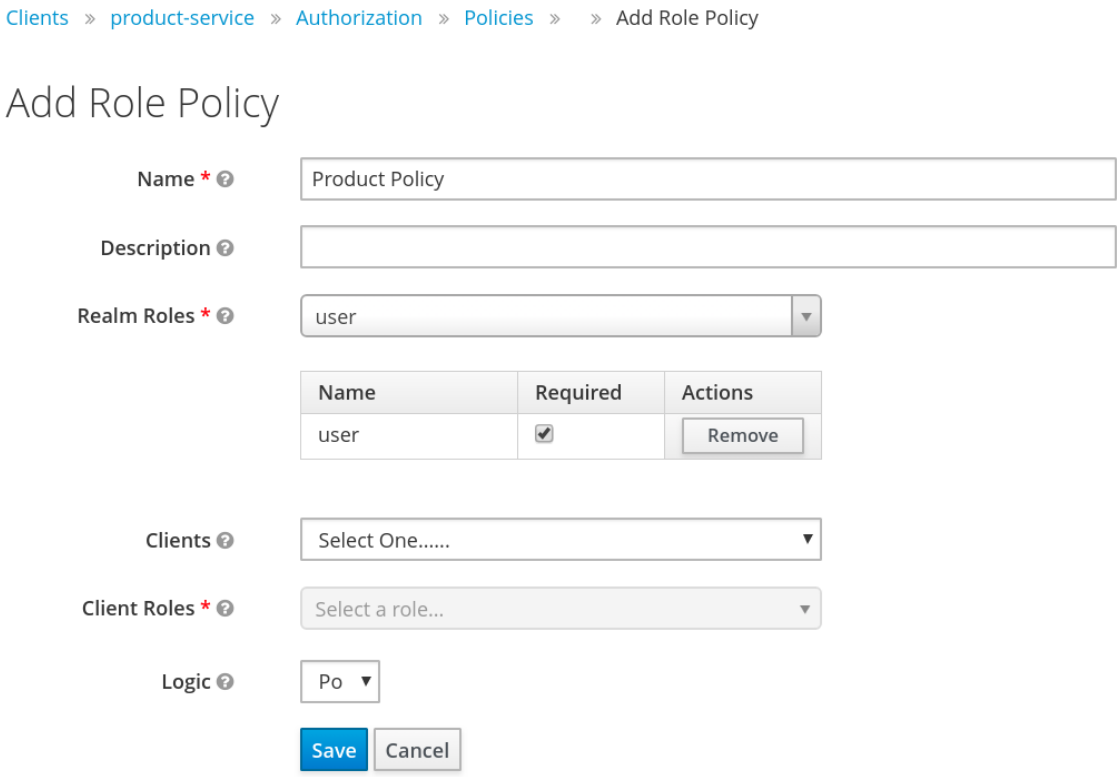
Create a new resource :



Now create a permission :



From this same screen, you can also create the new policy of type `role` :



Save your policy and your permission and you're good on the Keycloak side.

Last thing to do is to switch your service from `bearer-only` to confidential, also copy your secret for the next step.

8.3. Enabling Authz on the service side

Remove all the keycloak properties from `application.properties` and create a new file in the same folder called `keycloak.json`

```
{
  "realm": "voxxed",
  "auth-server-url": "http://localhost:8180/auth",
  "ssl-required": "external",
  "resource": "product-service",
  "credentials": {
    "secret": "you_client_secret"
  },
  "confidential-port": 0,
  "policy-enforcer": {
    "path-cache" : {
      "lifespan" : 0
    }
  },
  "enable-cors" : true
}
```

We can now remove the `@RoleAllowed` annotation from our method since the RBAC is handled on the Keycloak side.

Try the service, it should just work as before.

8.4. Adding a new Role to the policy

To see the power of centralized RBAC, let's create a new realm role , for instance : `superuser` and let's update the policy we created before :

Log in again with your user, you should not be able to access the service unless you assign the role `superuser` to your user.

8.5. Adding a Time Policy

You can add a lot of different authorization policies, for instance let's add a policy based on time :

[timepolicy] | *images/timepolicy.png*

9. Limiting th Scope

10. The Gatekeeper

Download the Keycloak gatekeeper

10.1. Start the PHP server

10.2. Start the Gatekeeper

```
./keycloak-gatekeeper --resources="uri=/public|white-listed=true"
--resources="uri=/secured|roles=user|methods=GET"
--resources="uri=/admin|roles=admin|methods=GET" --client-id=php-rest --client
-secret=b9bd100a-db63-404b-810c-f5abe8f8ae6b --discovery
-url=http://localhost:8180/auth/realms/voxxed/.well-known/openid-configuration
--listen=0.0.0.0:9090 --enable-logging=true --enable-json-logging=true --upstream
-url=http://localhost:8080 --cors-origins='*' --skip-openid-provider-tls-verify=true
--no-redirects=true --cors-methods="GET" --cors-methods="POST" --cors-headers="*"
--add-claims=preferred_username
```

10.3. Update the web client to call the PHP service instead

The Gatekeeper is listing on port `9090`