

# Quarkus - Using OpenID Connect and Keycloak to Centralize Authorizations

This guide demonstrates how your Quarkus application can authorize access to protected resources using [Keycloak Authorization Services](#).

The `quarkus-keycloak-authorization` extension is based on `quarkus-oidc` and provides a policy enforcer that enforces access to protected resources based on permissions managed by Keycloak. It provides a flexible and dynamic authorization capability based on Resource-Based Access Control. In other words, instead of explicitly enforce access based on some specific access control mechanism (e.g.: RBAC), you just check whether or not a request is allowed to access a resource based on its name, identifier or URL.

By externalizing authorization from your application, you are allowed to protect your applications using different access control mechanisms as well as avoid re-deploying your application every time your security requirements change, where Keycloak will be acting as a centralized authorization service from where your protected resources and their associated permissions are managed.

If you are already familiar with Keycloak, you'll notice that the extension is basically another adapter implementation but specific for Quarkus applications. Otherwise, you can find more information in the Keycloak [documentation](#).

## Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.6.3
- [jq tool](#)
- [Keycloak](#)
- Docker

## Architecture

In this example, we build a very simple microservice which offers two endpoints:

- `/api/users/me`
- `/api/admin`

These endpoints are protected and can only be accessed if a client is sending a bearer token along

with the request, which must be valid (e.g.: signature, expiration and audience) and trusted by the microservice.

The bearer token is issued by a Keycloak Server and represents the subject to which the token was issued for. For being an OAuth 2.0 Authorization Server, the token also references the client acting on behalf of the user.

The `/api/users/me` endpoint can be accessed by any user with a valid token. As a response, it returns a JSON document with details about the user where these details are obtained from the information carried on the token. This endpoint is protected with RBAC (Role-Based Access Control) and only users granted with the `user` role can access this endpoint.

The `/api/admin` endpoint is protected with RBAC (Role-Based Access Control) and only users granted with the `admin` role can access it.

This is a very simple example using RBAC policies to govern access to your resources. However, Keycloak supports other types of policies that you can use to perform even more fine-grained access control. By using this example, you'll see that your application is completely decoupled from your authorization policies with enforcement being purely based on the accessed resource.

## Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can go right to the completed example.

Clone the Git repository: `git clone https://github.com/quarkusio/quarkus-quickstarts.git`, or download an [archive](#).

The solution is located in the `security-keycloak-authorization-quickstart` directory.

## Creating the Project

First, we need a new project. Create a new project with the following command:

```
mvn io.quarkus:quarkus-maven-plugin:1.4.0.CR1:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=security-keycloak-authorization-quickstart \
  -Dextensions="oidc, keycloak-authorization, resteasy-jsonb"
cd security-keycloak-authorization-quickstart
```

This command generates a Maven project, importing the `keycloak-authorization` extension which is an implementation of a Keycloak Adapter for Quarkus applications and provides all the necessary capabilities to integrate with a Keycloak Server and perform bearer token authorization.

Let's start by implementing the `/api/users/me` endpoint. As you can see from the source code below it is just a regular JAX-RS resource:

```

package org.acme.security.keycloak.authorization;;

import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import org.jboss.resteasy.annotations.cache.NoCache;

import io.quarkus.security.identity.SecurityIdentity;

@Path("/api/users")
public class UsersResource {

    @Inject
    SecurityIdentity identity;

    @GET
    @Path("/me")
    @Produces(MediaType.APPLICATION_JSON)
    @NoCache
    public User me() {
        return new User(identity);
    }

    public static class User {

        private final String userName;

        User(SecurityIdentity identity) {
            this.userName = identity.getPrincipal().getName();
        }

        public String getUserName() {
            return userName;
        }
    }
}

```

The source code for the `/api/admin` endpoint is also very simple:

```

package org.acme.security.keycloak.authorization;;

import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

import io.quarkus.security.Authenticated;

@Path("/api/admin")
@Authenticated
public class AdminResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String admin() {
        return "granted";
    }
}

```

Note that we did not define any annotation such as `@RoleAllowed` to explicitly enforce access to a resource. The extension will be responsible to map the URIs of the protected resources you have in Keycloak and evaluate the permissions accordingly, granting or denying access depending on the permissions that will be granted by Keycloak.

## Configuring the application

The OpenID Connect extension allows you to define the adapter configuration using the `application.properties` file which should be located at the `src/main/resources` directory.

```

# OIDC Configuration
quarkus.oidc.auth-server-
url=http://localhost:8180/auth/realms/quarkus
quarkus.oidc.client-id=backend-service
quarkus.oidc.credentials.secret=secret

# Enable Policy Enforcement
quarkus.keycloak.policy-enforcer.enable=true

```



By default, applications using the `quarkus-oidc` extension are marked as a `service` type application (see `quarkus.oidc.application-type`). This extension currently supports only such `service` type applications.

# Starting and Configuring the Keycloak Server

To start a Keycloak Server you can use Docker and just run the following command:

```
docker run --name keycloak -e KEYCLOAK_USER=admin -e  
KEYCLOAK_PASSWORD=admin -p 8180:8080  
quay.io/keycloak/keycloak:9.0.2
```

You should be able to access your Keycloak Server at [localhost:8180/auth](http://localhost:8180/auth).

Log in as the **admin** user to access the Keycloak Administration Console. Username should be **admin** and password **admin**.

Import the [realm configuration file](#) to create a new realm. For more details, see the Keycloak documentation about how to [create a new realm](#).

## Running and Using the Application

### Running in Developer Mode

To run the microservice in dev mode, use `./mvnw clean compile quarkus:dev`.

### Running in JVM Mode

When you're done playing with "dev-mode" you can run it as a standard Java application.

First compile it:

```
./mvnw package
```

Then run it:

```
java -jar ./target/security-keycloak-authorization-quickstart-  
runner.jar
```

### Running in Native Mode

This same demo can be compiled into native code: no modifications required.

This implies that you no longer need to install a JVM on your production environment, as the runtime technology is included in the produced binary, and optimized to run with minimal resource overhead.

Compilation will take a bit longer, so this step is disabled by default; let's build again by enabling the **native** profile:

```
./mvnw package -Pnative
```

After getting a cup of coffee, you'll be able to run this binary directly:

```
./target/security-keycloak-authorization-quickstart-runner
```

## Testing the Application

The application is using bearer token authorization and the first thing to do is obtain an access token from the Keycloak Server in order to access the application resources:

```
export access_token=$(\  
  curl -X POST\  
  http://localhost:8180/auth/realms/quarkus/protocol/openid-  
  connect/token \  
    --user backend-service:secret \  
    -H 'content-type: application/x-www-form-urlencoded' \  
    -d 'username=alice&password=alice&grant_type=password' | jq\  
  --raw-output '.access_token' \  
)
```

The example above obtains an access token for user **alice**.

Any user is allowed to access the <http://localhost:8080/api/users/me> endpoint which basically returns a JSON payload with details about the user.

```
curl -v -X GET \  
  http://localhost:8080/api/users/me \  
  -H "Authorization: Bearer "$access_token
```

The <http://localhost:8080/api/admin> endpoint can only be accessed by users with the **admin** role. If you try to access this endpoint with the previously issued access token, you should get a **403** response from the server.

```
curl -v -X GET \  
  http://localhost:8080/api/admin \  
  -H "Authorization: Bearer "$access_token
```

In order to access the admin endpoint you should obtain a token for the **admin** user:

```
export access_token=$(\
  curl -X POST
  http://localhost:8180/auth/realms/quarkus/protocol/openid-
  connect/token \
    --user backend-service:secret \
    -H 'content-type: application/x-www-form-urlencoded' \
    -d 'username=admin&password=admin&grant_type=password' | jq
  --raw-output '.access_token' \
)
```

## Checking Permissions Programmatically

In some cases, you may want to programmatically check whether or not a request is granted to access a protected resource. By injecting a `SecurityIdentity` instance in your beans, you are allowed to check permissions as follows:

```
@Path("/api/protected")
public class ProtectedResource {

    @Inject
    SecurityIdentity identity;

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public CompletionStage<List<Permission>> get() {
        return identity.checkPermission(new AuthPermission(
            "{resource_name}"))
            .thenCompose(granted -> {
                if (granted) {
                    return CompletableFuture.completedFuture
(doGetState());
                }
                throw new ForbiddenException();
            });
    }
}
```

## Mapping Protected Resources

By default, the extension loads all the protected resources managed by Keycloak where their `URI` are used to map the resources in your application that should be protected. This is done during the application startup.

If you want to disable this behavior and map resources on-demand, you can use the following

configuration:

```
quarkus.keycloak.policy-enforcer.lazy-load-paths=true
```

## More About Configuring Protected Resources

In the default configuration, Keycloak is responsible for managing the roles and deciding who can access which routes.

To configure the protected routes using the `@RolesAllowed` annotation or the `application.properties` file, check the [Using OpenID Connect Adapter to Protect JAX-RS Applications](#) guide. For more details, check the [Security guide](#).

## References

- [Keycloak Documentation](#)
- [Keycloak Authorization Services Documentation](#)
- [OpenID Connect](#)
- [JSON Web Token](#)