Quarkus - Using OpenID Connect Adapter to Protect JAX-RS Applications

This guide demonstrates how your Quarkus application can use an OpenID Connect Adapter to protect your JAX-RS applications using bearer token authorization, where these tokens are issued by OpenId Connect and OAuth 2.0 compliant Authorization Servers such as Keycloak.

Bearer Token Authorization is the process of authorizing HTTP requests based on the existence and validity of a bearer token representing a subject and his access context, where the token provides valuable information to determine the subject of the call as well whether or not a HTTP resource can be accessed.

We are going to give you a guideline on how to use OpenId Connect and OAuth 2.0 in your JAX-RS applications using the Quarkus OpenID Connect Extension.

> This technology is considered preview.
>
> In *preview*, backward compatibility and presence in the ecosystem is not guaranteed. Specific improvements might require to change configuration or APIs and plans to become *stable* are under way. Feedback is welcome on our mailing list or as issues in our GitHub issue tracker.
>
> For a full list of possible extension statuses, check our FAQ entry.

Prerequisites

To complete this guide, you need:

- less than 15 minutes
- an IDE
- JDK 1.8+ installed with `JAVA_HOME` configured appropriately
- Apache Maven 3.5.3+
- jq tool
- Docker

Architecture

In this example, we build a very simple microservice which offers three 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.

The `/api/admin` endpoint is protected with RBAC (Role-Based Access Control) where only users granted with the `admin` role can access. At this endpoint, we use the `@RolesAllowed` annotation to declaratively enforce the access constraint.

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-openid-connect-quickstart` directory.

Creating the Maven Project

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

```
mvn io.quarkus:quarkus-maven-plugin:1.3.0.Alpha1:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=security-openid-connect-quickstart \
    -Dextensions="oidc, resteasy-jsonb"
cd security-openid-connect-quickstart
```

This command generates a Maven project, importing the `keycloak` 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.

Writing the application

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:

```java
package org.acme.keycloak;

import javax.annotation.security.RolesAllowed;
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 securityIdentity;

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

    public static class User {

        private final String userName;

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

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

The source code for the /api/admin endpoint is also very simple. The main difference here is that we are using a @RolesAllowed annotation to make sure that only users granted with the admin role can access the endpoint:

```
package org.acme.keycloak;

import javax.annotation.security.RolesAllowed;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.MediaType;

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

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

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.

Configuring using the application.properties file

🗄 Configuration property fixed at build time - ⚙ Configuration property overridable at runtime

Configuration property	Type	Default
🗄 `quarkus.oidc.enabled` If the OIDC extension is enabled.	boolean	`true`
🗄 `quarkus.oidc.application-type` The application type, which can be one of the following values from enum `ApplicationType`.	`web-app`, `service`	`service`
⚙ `quarkus.oidc.connection-delay` The maximum amount of time the adapter will try connecting to the currently unavailable OIDC server for. For example, setting it to '20S' will let the adapter keep requesting the connection for up to 20 seconds.	Duration ❓	

⚙ `quarkus.oidc.auth-server-url` The base URL of the OpenID Connect (OIDC) server, for example, 'https://host:port/auth'. All the other OIDC server page and service URLs are derived from this URL. Note if you work with Keycloak OIDC server, make sure the base URL is in the following format: 'https://host:port/auth/realms/{realm}' where '{realm}' has to be replaced by the name of the Keycloak realm.	string	
⚙ `quarkus.oidc.introspection-path` Relative path of the RFC7662 introspection service.	string	
⚙ `quarkus.oidc.jwks-path` Relative path of the OIDC service returning a JWK set.	string	
⚙ `quarkus.oidc.public-key` Public key for the local JWT token verification.	string	
⚙ `quarkus.oidc.client-id` The client-id of the application. Each application has a client-id that is used to identify the application	string	
⚙ `quarkus.oidc.roles.role-claim-path` Path to the claim containing an array of groups. It starts from the top level JWT JSON object and can contain multiple segments where each segment represents a JSON object name only, example: "realm/groups". This property can be used if a token has no 'groups' claim but has the groups set in a different claim.	string	
⚙ `quarkus.oidc.roles.role-claim-separator` Separator for splitting a string which may contain multiple group values. It will only be used if the "role-claim-path" property points to a custom claim whose value is a string. A single space will be used by default because the standard 'scope' claim may contain a space separated sequence.	string	
⚙ `quarkus.oidc.token.issuer` Expected issuer 'iss' claim value	string	
⚙ `quarkus.oidc.token.audience` Expected audience `aud` claim value which may be a string or an array of strings	list of string	

	Type	Default
⚙ `quarkus.oidc.token.principal-claim` Name of the claim which contains a principal name. By default, the 'upn', 'preferred_username' and `sub` claims are checked.	string	
⚙ `quarkus.oidc.credentials.secret` The client secret	string	
⚙ `quarkus.oidc.authentication.redirect-path` Relative path for calculating a "redirect_uri" parameter. It set it will be appended to the request URI's host and port, otherwise the complete request URI will be used. It has to start from the forward slash, for example: "/service"	string	
⚙ `quarkus.oidc.authentication.scopes` List of scopes	list of string	
Additional named tenants	**Type**	**Default**
⚙ `quarkus.oidc."tenant".connection-delay` The maximum amount of time the adapter will try connecting to the currently unavailable OIDC server for. For example, setting it to '20S' will let the adapter keep requesting the connection for up to 20 seconds.	Duration ❓	
⚙ `quarkus.oidc."tenant".auth-server-url` The base URL of the OpenID Connect (OIDC) server, for example, 'https://host:port/auth'. All the other OIDC server page and service URLs are derived from this URL. Note if you work with Keycloak OIDC server, make sure the base URL is in the following format: 'https://host:port/auth/realms/{realm}' where '{realm}' has to be replaced by the name of the Keycloak realm.	string	
⚙ `quarkus.oidc."tenant".introspection-path` Relative path of the RFC7662 introspection service.	string	
⚙ `quarkus.oidc."tenant".jwks-path` Relative path of the OIDC service returning a JWK set.	string	
⚙ `quarkus.oidc."tenant".public-key` Public key for the local JWT token verification.	string	

⚙ `quarkus.oidc."tenant".client-id` The client-id of the application. Each application has a client-id that is used to identify the application	string	
⚙ `quarkus.oidc."tenant".roles.role-claim-path` Path to the claim containing an array of groups. It starts from the top level JWT JSON object and can contain multiple segments where each segment represents a JSON object name only, example: "realm/groups". This property can be used if a token has no 'groups' claim but has the groups set in a different claim.	string	
⚙ `quarkus.oidc."tenant".roles.role-claim-separator` Separator for splitting a string which may contain multiple group values. It will only be used if the "role-claim-path" property points to a custom claim whose value is a string. A single space will be used by default because the standard 'scope' claim may contain a space separated sequence.	string	
⚙ `quarkus.oidc."tenant".token.issuer` Expected issuer 'iss' claim value	string	
⚙ `quarkus.oidc."tenant".token.audience` Expected audience `aud` claim value which may be a string or an array of strings	list of string	
⚙ `quarkus.oidc."tenant".token.principal-claim` Name of the claim which contains a principal name. By default, the 'upn', 'preferred_username' and `sub` claims are checked.	string	
⚙ `quarkus.oidc."tenant".credentials.secret` The client secret	string	
⚙ `quarkus.oidc."tenant".authentication.redirect-path` Relative path for calculating a "redirect_uri" parameter. It set it will be appended to the request URI's host and port, otherwise the complete request URI will be used. It has to start from the forward slash, for example: "/service"	string	
⚙ `quarkus.oidc."tenant".authentication.scopes` List of scopes	list of string	

> *About the Duration format*
>
> The format for durations uses the standard `java.time.Duration` format. You can learn more about it in the Duration#parse() javadoc.
>
> You can also provide duration values starting with a number. In this case, if the value consists only of a number, the converter treats the value as seconds. Otherwise, `PT` is implicitly prepended to the value to obtain a standard `java.time.Duration` format.

Example configuration:

```
quarkus.oidc.auth-server-
url=http://localhost:8180/auth/realms/quarkus
quarkus.oidc.client-id=backend-service
```

Configuring CORS

If you plan to consume this application from another application running on a different domain, you will need to configure CORS (Cross-Origin Resource Sharing). Please read the HTTP CORS documentation for more details.

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:8.0.1
```

You should be able to access your Keycloak Server at 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-openid-connect-quickstart-runner.jar
```

Runing 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-openid-connect-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' \
 )
```

References

- Keycloak Documentation
- OpenID Connect
- JSON Web Token