

Quarkus - Using OAuth2 RBAC

This guide explains how your Quarkus application can utilize OAuth2 tokens to provide secured access to the JAX-RS endpoints.

OAuth2 is an authorization framework that enables applications to obtain access to an HTTP resource on behalf of a user. It can be used to implement an application authentication mechanism based on tokens by delegating to an external server (the authentication server) the user authentication and providing a token for the authentication context.

This extension provides a light-weight support for using the opaque Bearer Tokens and validating them by calling an introspection endpoint.

If the OAuth2 Authentication server provides JWT Bearer Tokens then you should consider using either [OpenId Connect](#) or [MicroProfile JWT RBAC](#) extensions instead. OpenId Connect extension has to be used if the Quarkus application needs to authenticate the users using OIDC Authorization Code Flow, please read [Using OpenID Connect to Protect Web Applications](#) guide for more information.



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](#).

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-oauth2-quickstart` directory. It contains a very simple UI to use the JAX-RS resources created here, too.

Creating the Maven project

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

```
mvn io.quarkus:quarkus-maven-plugin:1.7.2.Final:create \
  -DprojectId=org.acme \
  -DprojectArtifactId=security-oauth2-quickstart \
  -DclassName="org.acme.security.oauth2.TokenSecuredResource" \
  -Dpath="/secured" \
  -Dextensions="resteasy-jsonb, security-oauth2"
cd security-oauth2-quickstart
```

This command generates the Maven project with a REST endpoint and imports the `elytron-security-oauth2` extension, which includes the OAuth2 opaque token support.

If you don't want to use the Maven plugin, you can just include the dependency in your pom.xml:

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-elytron-security-oauth2</artifactId>
</dependency>
```

Examine the JAX-RS resource

Open the `src/main/java/org/acme/security/oauth2/TokenSecuredResource.java` file and see the following content:

Basic REST Endpoint

```
package org.acme.security.oauth2;

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

@Path("/secured")
public class TokenSecuredResource {

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

This is a basic REST endpoint that does not have any of the Elytron Security OAuth2 specific features, so let's add some.

We will use the JSR 250 common security annotations, they are described in the [Using Security](#) guide.

```

package org.acme.security.oauth2;

import java.security.Principal;

import javax.annotation.security.PermitAll;
import javax.inject.Inject;
import javax.ws.rs.GET;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.core.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.SecurityContext;

@Path("/secured")
@ApplicationScoped
public class TokenSecuredResource {

    @GET()
    @Path("permit-all")
    @PermitAll ①
    @Produces(MediaType.TEXT_PLAIN)
    public String hello(@Context SecurityContext ctx) { ②
        Principal caller = ctx.getUserPrincipal(); ③
        String name = caller == null ? "anonymous" :
caller.getName();
        String helloReply = String.format("hello + %s, isSecure:
%s, authScheme: %s", name, ctx.isSecure(),
ctx.getAuthenticationScheme());
        return helloReply; ④
    }
}

```

- ① **@PermitAll** indicates that the given endpoint is accessible by any caller, authenticated or not.
- ② Here we inject the JAX-RS **SecurityContext** to inspect the security state of the call.
- ③ Here we obtain the current request user/caller **Principal**. For an unsecured call this will be null, so we build the user name by checking **caller** against null.
- ④ The reply we build up makes use of the caller name, the **isSecure()** and **getAuthenticationScheme()** states of the request **SecurityContext**.

Run the application

Now we are ready to run our application. Use:

```
./mvnw compile quarkus:dev
```

and you should see output similar to:

quarkus:dev Output

```
$ ./mvnw clean compile quarkus:dev
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.acme:security-oauth2-quickstart
>-----
[INFO] Building security-oauth2-quickstart 1.0-SNAPSHOT
[INFO] -----[ jar
]-----
...
[INFO] --- quarkus-maven-plugin:999-SNAPSHOT:dev (default-cli) @
security-oauth2-quickstart ---
Listening for transport dt_socket at address: 5005
2019-07-16 09:58:09,753 INFO [io.qua.dep.QuarkusAugmentor] (main)
Beginning quarkus augmentation
2019-07-16 09:58:10,884 INFO [io.qua.dep.QuarkusAugmentor] (main)
Quarkus augmentation completed in 1131ms
2019-07-16 09:58:11,385 INFO [io.quarkus] (main) Quarkus 0.20.0
started in 1.813s. Listening on: http://[::]:8080
2019-07-16 09:58:11,391 INFO [io.quarkus] (main) Installed
features: [cdi, resteasy, resteasy-jsonb, security, security-
oauth2]
```

Now that the REST endpoint is running, we can access it using a command line tool like curl:

curl command for /secured/permit-all

```
$ curl http://127.0.0.1:8080/secured/permit-all; echo
hello + anonymous, isSecure: false, authScheme: null
```

We have not provided any token in our request, so we would not expect that there is any security state seen by the endpoint, and the response is consistent with that:

- user name is anonymous
- `isSecure` is false as https is not used
- `authScheme` is null

So now let's actually secure something. Take a look at the new endpoint method `helloRolesAllowed` in the following:

```

package org.acme.security.oauth2;

import java.security.Principal;

import javax.annotation.security.PermitAll;
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.Context;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.SecurityContext;

@Path("/secured")
@ApplicationScoped
public class TokenSecuredResource {

    @GET()
    @Path("permit-all")
    @PermitAll
    @Produces(MediaType.TEXT_PLAIN)
    public String hello(@Context SecurityContext ctx) {
        Principal caller = ctx.getUserPrincipal();
        String name = caller == null ? "anonymous" :
caller.getName();
        String helloReply = String.format("hello + %s, isSecure:
%s, authScheme: %s", name, ctx.isSecure(),
ctx.getAuthenticationScheme());
        return helloReply;
    }

    @GET()
    @Path("roles-allowed") ①
    @RolesAllowed({"Echoer", "Subscriber"}) ②
    @Produces(MediaType.TEXT_PLAIN)
    public String helloRolesAllowed(@Context SecurityContext ctx) {
        Principal caller = ctx.getUserPrincipal();
        String name = caller == null ? "anonymous" :
caller.getName();
        String helloReply = String.format("hello + %s, isSecure:
%s, authScheme: %s", name, ctx.isSecure(),
ctx.getAuthenticationScheme());
        return helloReply;
    }
}

```

① This new endpoint will be located at `/secured/roles-allowed`

② `@RolesAllowed` indicates that the given endpoint is accessible by a caller if they have either a "Echoer" or "Subscriber" role assigned.

After you make this addition to your `TokenSecuredResource`, try `curl -v http://127.0.0.1:8080/secured/roles-allowed; echo` to attempt to access the new endpoint. Your output should be:

curl command for /secured/roles-allowed

```
$ curl -v http://127.0.0.1:8080/secured/roles-allowed; echo
* Trying 127.0.0.1...
* TCP_NODELAY set
* Connected to 127.0.0.1 (127.0.0.1) port 8080 (#0)
> GET /secured/roles-allowed HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/7.54.0
> Accept: */*
>
< HTTP/1.1 401 Unauthorized
< Connection: keep-alive
< Content-Type: text/html; charset=UTF-8
< Content-Length: 14
< Date: Sun, 03 Mar 2019 16:32:34 GMT
<
* Connection #0 to host 127.0.0.1 left intact
Not authorized
```

Excellent, we have not provided any OAuth2 token in the request, so we should not be able to access the endpoint, and we were not. Instead we received an HTTP 401 Unauthorized error. We need to obtain and pass in a valid OAuth2 token to access that endpoint. There are two steps to this, 1) configuring our Elytron Security OAuth2 extension with information on how to validate the token, and 2) generating a matching token with the appropriate claims.

Configuring the Elytron Security OAuth2 Extension Security Information

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
 <code>quarkus.oauth2.enabled</code> Determine if the OAuth2 extension is enabled. Enabled by default if you include the <code>elytron-security-oauth2</code> dependency, so this would be used to disable it.	boolean	<code>true</code>

 <code>quarkus.oauth2.role-claim</code>		
The claim that is used in the introspection endpoint response to load the roles.	string	scope
<code>quarkus.oauth2.client-id</code>		
The OAuth2 client id used to validate the token. Mandatory if the extension is enabled.	string	
<code>quarkus.oauth2.client-secret</code>		
The OAuth2 client secret used to validate the token. Mandatory if the extension is enabled.	string	
<code>quarkus.oauth2.introspection-url</code>		
The OAuth2 introspection endpoint URL used to validate the token and gather the authentication claims. Mandatory if the extension is enabled.	string	
<code>quarkus.oauth2.ca-cert-file</code>		
The OAuth2 server certificate file. Warning: this is not supported in native mode where the certificate must be included in the truststore used during the native image generation, see Using SSL With Native Executables .	string	

Setting up application.properties

For part A of step 1, create a `security-oauth2-quickstart/src/main/resources/application.properties` with the following content:

application.properties for TokenSecuredResource

```
quarkus.oauth2.client-id=client_id
quarkus.oauth2.client-secret=secret
quarkus.oauth2.introspection-url=http://oauth-server/introspect
```

You need to specify the introspection URL of your authentication server and the `client-id` / `client-secret` that your application will use to authenticate itself to the authentication server.

The extension will then use this information to validate the token and recover the information associate with it.

Generating a token

You need to obtain the token from a standard OAuth2 authentication server ([Keycloak](#) for example) using the token endpoint.

You can find below a curl example of such call for a `client_credential` flow:

```
curl -X POST "http://oauth-server/token?grant_type=client_credentials" \
-H "Accept: application/json" -H "Authorization: Basic Y2xpZW50X2lkOmNsaWVudF9zZWNYZXQ="
```

It should respond something like that...

```
{"access_token": "60acf56d-9daf-49ba-b3be-7a423d9c7288", "token_type": "bearer", "expires_in": 1799, "scope": "READER"}
```

Finally, Secured Access to `/secured/roles-allowed`

Now let's use this to make a secured request to the `/secured/roles-allowed` endpoint

curl Command for `/secured/roles-allowed` With a token

```
$ curl -H "Authorization: Bearer 60acf56d-9daf-49ba-b3be-7a423d9c7288" http://127.0.0.1:8080/secured/roles-allowed; echo
hello + client_id isSecure: false, authScheme: OAuth2
```

Success! We now have:

- a non-anonymous caller name of `client_id`
- an authentication scheme of `OAuth2`

Roles mapping

Roles are mapped from one of the claims of the introspection endpoint response. By default, it's the `scope` claim. Roles are obtained by splitting the claim with a space separator. If the claim is an array, no splitting is done, the roles are obtained from the array.

You can customize the name of the claim to use for the roles with the `quarkus.oauth2.role-claim` property.

Package and run the application

As usual, the application can be packaged using `./mvnw clean package` and executed using the `-runner.jar` file: `.Runner.jar Example`

```
$ ./mvnw clean package
[INFO] Scanning for projects...
...
$ java -jar target/security-oauth2-quickstart-runner.jar
2019-03-28 14:27:48,839 INFO [io.quarkus] (main) Quarkus 0.20.0
started in 0.796s. Listening on: http://[::]:8080
2019-03-28 14:27:48,841 INFO [io.quarkus] (main) Installed
features: [cdi, resteasy, resteasy-jsonb, security, security-
oauth2]
```

You can also generate the native executable with `./mvnw clean package -Pnative`. [Native Executable Example](#)

```

$ ./mvnw clean package -Pnative
[INFO] Scanning for projects...
...
[security-oauth2-quickstart-runner:25602]    universe:      493.17
ms
[security-oauth2-quickstart-runner:25602]    (parse):      660.41
ms
[security-oauth2-quickstart-runner:25602]    (inline):    1,431.10
ms
[security-oauth2-quickstart-runner:25602]    (compile):   7,301.78
ms
[security-oauth2-quickstart-runner:25602]    compile:    10,542.16
ms
[security-oauth2-quickstart-runner:25602]    image:      2,797.62
ms
[security-oauth2-quickstart-runner:25602]    write:      988.24
ms
[security-oauth2-quickstart-runner:25602]    [total]:   43,778.16
ms
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 51.500 s
[INFO] Finished at: 2019-06-28T14:30:56-07:00
[INFO]
-----

$ ./target/security-oauth2-quickstart-runner
2019-03-28 14:31:37,315 INFO [io.quarkus] (main) Quarkus 0.20.0
started in 0.006s. Listening on: http://[::]:8080
2019-03-28 14:31:37,316 INFO [io.quarkus] (main) Installed
features: [cdi, resteasy, resteasy-jsonb, security, security-
oauth2]

```

Integration testing

If you don't want to use a real OAuth2 authorization server for your integration tests, you can use the [Properties based security](#) extension for your test, or mock an authorization server using Wiremock.

First of all, Wiremock needs to be added as a test dependency. For a Maven project that would happen like so:

```
<dependency>
  <groupId>com.github.tomakehurst</groupId>
  <artifactId>wiremock-jre8</artifactId>
  <scope>test</scope>
  <version>${wiremock.version}</version> ①
</dependency>
```

① Use a proper Wiremock version. All available versions can be found [here](#).

In Quarkus tests when some service needs to be started before the Quarkus tests are ran, we utilize the `@io.quarkus.test.common.QuarkusTestResource` annotation to specify a `io.quarkus.test.common.QuarkusTestResourceLifecycleManager` which can start the service and supply configuration values that Quarkus will use.



For more details about `@QuarkusTestResource` refer to [this part of the documentation](#).

Let's create an implementation of `QuarkusTestResourceLifecycleManager` called `MockAuthorizationServerTestResource` like so:

```

import com.github.tomakehurst.wiremock.WireMockServer;
import com.github.tomakehurst.wiremock.client.WireMock;
import io.quarkus.test.common.QuarkusTestResourceLifecycleManager;

import java.util.Collections;
import java.util.Map;

public class MockAuthorizationServerTestResource implements
QuarkusTestResourceLifecycleManager { ①

    private WireMockServer wireMockServer;

    @Override
    public Map<String, String> start() {
        wireMockServer = new WireMockServer();
        wireMockServer.start(); ②

        // define the mock for the introspect endpoint

WireMock.stubFor(WireMock.post("/introspect").willReturn(WireMock.a
Response() ③
                .withBody(

"{\"active\":true,\"scope\":\"Echoer\",\"username\":null,\"iat\":15
62315654,\"exp\":1562317454,\"expires_in\":1458,\"client_id\":\"my_
client_id\"}"));

        return
Collections.singletonMap("quarkus.oauth2.introspection-url",
wireMockServer.baseUrl() + "/introspect"); ④
    }

    @Override
    public void stop() {
        if (null != wireMockServer) {
            wireMockServer.stop(); ⑤
        }
    }
}

```

- ① The **start** method is invoked by Quarkus before any test is run and returns a **Map** of configuration properties that apply during the test execution.
- ② Launch Wiremock.
- ③ Configure Wiremock to stub the calls to **/introspect** by returning an OAuth2 introspect response. You need to customize this line to return what's needed for your application (at least the scope property as roles are derived from the scope).

- ④ As the `start` method returns configuration that applies for tests, we set the `quarkus.oauth2.introspection-url` property that controls the URL of the introspect endpoint used by the OAuth2 extension.
- ⑤ When all tests have finished, shutdown Wiremock.

Your test class needs to be annotated like with `@QuarkusTestResource(MockAuthorizationServerTestResource.class)` to use this `QuarkusTestResourceLifecycleManager`.

Below is an example of a test that uses the `MockAuthorizationServerTestResource`.

```
@QuarkusTest
@QuarkusTestResource(MockAuthorizationServerTestResource.class) ①
class TokenSecuredResourceTest {
    // use whatever token you want as the mock OAuth server will
    // accept all tokens
    private static final String BEARER_TOKEN = "337aab0f-b547-489b-
    9dbd-a54dc7bdf20d"; ②

    @Test
    void testPermitAll() {
        RestAssured.given()
            .when()
            .header("Authorization", "Bearer: " + BEARER_TOKEN)
            .get("/secured/permit-all")
            .then()
            .statusCode(200)
            .body(containsString("hello"));
    }

    @Test
    void testRolesAllowed() {
        RestAssured.given()
            .when()
            .header("Authorization", "Bearer: " + BEARER_TOKEN)
            .get("/secured/roles-allowed")
            .then()
            .statusCode(200)
            .body(containsString("hello"));
    }
}
```

- ① Use the previously created `MockAuthorizationServerTestResource` as a Quarkus test resource.
- ② Define whatever token you want, it will not be validated by the OAuth2 mock authorization server.
- ③ Use this token inside the `Authorization` header to trigger OAuth2 authentication.



`@QuarkusTestResource` applies to all tests, not just `TokenSecuredResourceTest`.

References

- [OAuth2](#)
- [Quarkus Security](#)