

Quarkus - Using JWT RBAC

This guide explains how your Quarkus application can utilize MicroProfile Json Web Token (JWT) Role-Based Access Control (RBAC) to provide secured access to the JAX-RS endpoints.

Solution

We recommend that you follow the instructions in the next sections and create the application step by step. However, you can skip 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-jwt-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.Final:create \
  -DprojectId=org.acme \
  -DprojectId=security-jwt-quickstart \
  -DclassName="org.acme.security.jwt.TokenSecuredResource" \
  -Dpath="/secured" \
  -Dextensions="resteasy-jsonb, jwt"
cd security-jwt-quickstart
```

This command generates the Maven project with a REST endpoint and imports the `smallrye-jwt` extension, which includes the MicroProfile JWT RBAC support.

Examine the JAX-RS resource

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

Basic REST Endpoint

```
package org.acme.security.jwt;

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 Smallrye JWT specific features, so let's add some.



The MicroProfile JWT RBAC 1.1.1 specification details the annotations and behaviors we will make use of in this quickstart. See [HTML](#) and [PDF](#) versions of the specification for the details.

```

package org.acme.security.jwt;

import java.security.Principal;

import javax.annotation.security.PermitAll;
import javax.enterprise.context.RequestScoped;
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;

import org.eclipse.microprofile.jwt.JsonWebToken;

/**
 * Version 1 of the TokenSecuredResource
 */
@Path("/secured")
@RequestScoped ①
public class TokenSecuredResource {

    @Inject
    JsonWebToken jwt; ②

    @GET()
    @Path("permit-all")
    @PermitAll ③
    @Produces(MediaType.TEXT_PLAIN)
    public String hello(@Context SecurityContext ctx) { ④
        Principal caller = ctx.getUserPrincipal(); ⑤
        String name = caller == null ? "anonymous" : caller.
getName();
        boolean hasJWT = jwt.getClaimNames() != null;
        String helloReply = String.format("hello + %s, isSecure:
%s, authScheme: %s, hasJWT: %s", name, ctx.isSecure(), ctx
.getAuthenticationScheme(), hasJWT);
        return helloReply; ⑥
    }
}

```

- ① Add a `RequestScoped` as Quarkus uses a default scoping of `ApplicationScoped` and this will produce undesirable behavior since JWT claims are naturally request scoped.
- ② Here we inject the `JsonWebToken` interface, an extension of the `java.security.Principal` interface that provides access to the claims associated with the current authenticated token.

- ③ `@PermitAll` is a JSR 250 common security annotation that indicates that the given endpoint is accessible by any caller, authenticated or not.
- ④ Here we inject the JAX-RS `SecurityContext` to inspect the security state of the call.
- ⑤ 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.
- ⑥ The reply we build up makes use of the caller name, the `isSecure()` and `getAuthenticationScheme()` states of the request `SecurityContext`, and whether a non-null `JsonWebToken` was injected.

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 compile quarkus:dev
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.acme:security-jwt-quickstart
>-----
[INFO] Building security-jwt-quickstart 1.0-SNAPSHOT
[INFO] -----[ jar
]------
...
Listening for transport dt_socket at address: 5005
2019-03-03 07:23:06,988 INFO [io.qua.dep.QuarkusAugmentor] (main)
Beginning quarkus augmentation
2019-03-03 07:23:07,328 INFO [io.qua.dep.QuarkusAugmentor] (main)
Quarkus augmentation completed in 340ms
2019-03-03 07:23:07,493 INFO [io.quarkus] (main) Quarkus started
in 0.769s. Listening on: http://127.0.0.1:8080
2019-03-03 07:23:07,493 INFO [io.quarkus] (main) Installed
features: [cdi, resteasy, resteasy-jsonb, security, smallrye-jwt,
vertx, vertx-web]
```

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, hasJWT: false
```

We have not provided any JWT 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
- hasJWT is false

Use Ctrl-C to stop the Quarkus server.

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

REST Endpoint V2

```
package org.acme.security.jwt;

import java.security.Principal;

import javax.annotation.security.PermitAll;
import javax.annotation.security.RolesAllowed;
import javax.enterprise.context.RequestScoped;
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;

import org.eclipse.microprofile.jwt.JsonWebToken;

/**
 * Version 2 of the TokenSecuredResource
 */
@Path("/secured")
@RequestScoped
public class TokenSecuredResource {

    @Inject
    JsonWebToken jwt;

    @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();
    boolean hasJWT = jwt.getClaimNames() != null;
    String helloReply = String.format("hello + %s, isSecure:
%s, authScheme: %s, hasJWT: %s", name, ctx.isSecure(), ctx
.getAuthenticationScheme(), hasJWT);
    return helloReply;
}
}

```

- ① This new endpoint will be located at `/secured/roles-allowed`
- ② `@RolesAllowed` is a JSR 250 common security annotation that 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`, rerun the `./mvnw compile quarkus:dev` command, and then 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 JWT 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 JWT to access that endpoint. There are two steps to this, 1) configuring our Smallrye JWT extension with information on how to validate a JWT, and 2) generating a matching JWT with the appropriate claims.

Configuring the Smallrye JWT Extension Security Information

In the [Configuration Reference](#) section we introduce the `application.properties` file that affect the Smallrye JWT extension.

Setting up application.properties

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

application.properties for TokenSecuredResource

```
mp.jwt.verify.publickey.location=META-INF/resources/publicKey.pem ①
mp.jwt.verify.issuer=https://quarkus.io/using-jwt-rbac ②
quarkus.smallrye-jwt.enabled=true ③
```

① We are setting public key location to point to a classpath `publicKey.pem` resource location. We will add this key in part B, [Adding a Public Key](#).

- ② We are setting the issuer to the URL string <https://quarkus.io/using-jwt-rbac>.
- ③ We are enabling the Smallrye JWT. Also not required since this is the default, but we are making it explicit.

Adding a Public Key

The [JWT specification](#) defines various levels of security of JWTs that one can use. The MicroProfile JWT RBAC specification requires that JWTs that are signed with the RSA-256 signature algorithm. This in turn requires a RSA public key pair. On the REST endpoint server side, you need to configure the location of the RSA public key to use to verify the JWT sent along with requests. The `mp.jwt.verify.publickey.location=publicKey.pem` setting configured previously expects that the public key is available on the classpath as `publicKey.pem`. To accomplish this, copy the following content to a `security-jwt-quickstart/src/main/resources/META-INF/resources/publicKey.pem` file.

RSA Public Key PEM Content

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAlivFI8qB4D0y2jy0CfEq
Fyy46R0o7S8TKpsx5xbHKoU1VWg6QkQm+ntyIv1p4kE1sPEQ073+HY8+Bzs75XwR
TYL1BmR1w8J5hmjVWjc6R2BTBGAYRPFRRhor3kpM6ni2SPmNNhurEAHw7TaqszP5e
UF/F9+KEBkwVta+PZ37bwqSE4sCb1soZFrVz/UT/LF4tYpuVYt3YbqToZ3pZ0Z9
AX2o1GCG3xw0jkc4x0W7ezbQZdC9iftPxVHR8ir0ijJRRjcPDtA6vPKpzL16CyYn
sIYPd991twxTHjr3npfv/3Lw50bAkbT4HeLFxTx4f1EoZLK0/g0bAoV2uqBhkA9x
nQIDAQAB
-----END PUBLIC KEY-----
```

Generating a JWT

Often one obtains a JWT from an identity manager like [Keycloak](#), but for this quickstart we will generate our own using the JWT generation API provided by `smallrye-jwt` (see [Generate JWT tokens with Smallrye JWT](#) for more information) and the `TokenUtils` class shown in the following listing. Take this source and place it into `security-jwt-quickstart/src/test/java/org/acme/security/jwt/TokenUtils.java`.

JWT utility class

```
package org.acme.security.jwt;

import java.io.InputStream;
import java.security.KeyFactory;
import java.security.KeyPair;
import java.security.KeyPairGenerator;
import java.security.NoSuchAlgorithmException;
import java.security.PrivateKey;
import java.security.PublicKey;
import java.security.spec.PKCS8EncodedKeySpec;
```

```

import java.security.spec.X509EncodedKeySpec;
import java.util.Base64;
import java.util.Map;

import org.eclipse.microprofile.jwt.Claims;

import io.smallrye.jwt.build.Jwt;
import io.smallrye.jwt.build.JwtClaimsBuilder;
/**
 * Utilities for generating a JWT for testing
 */
public class TokenUtils {

    private TokenUtils() {
        // no-op: utility class
    }

    /**
     * Utility method to generate a JWT string from a JSON resource
     file that is signed by the privateKey.pem
     * test resource key, possibly with invalid fields.
     *
     * @param jsonResName - name of test resources file
     * @param timeClaims - used to return the exp, iat, auth_time
     claims
     * @return the JWT string
     * @throws Exception on parse failure
     */
    public static String generateTokenString(String jsonResName,
Map<String, Long> timeClaims)
        throws Exception {
        // Use the test private key associated with the test public
        key for a valid signature
        PrivateKey pk = readPrivateKey("/privateKey.pem");
        return generateTokenString(pk, "/privateKey.pem",
jsonResName, timeClaims);
    }

    public static String generateTokenString(PrivateKey privateKey,
String kid,
        String jsonResName, Map<String, Long> timeClaims) throws
Exception {

        JwtClaimsBuilder claims = Jwt.claims(jsonResName);
        long currentTimeInSecs = currentTimeInSecs();
        long exp = timeClaims != null && timeClaims.containsKey
(Claims.exp.name())
            ? timeClaims.get(Claims.exp.name()) : currentTimeInSecs
+ 300;

```

```

        claims.issuedAt(currentTimeInSecs);
        claims.claim(Claims.auth_time.name(), currentTimeInSecs);
        claims.expiresAt(exp);

        return claims.jws().signatureKeyId(kid).sign(privateKey);
    }

    /**
     * Read a PEM encoded private key from the classpath
     *
     * @param pemResName - key file resource name
     * @return PrivateKey
     * @throws Exception on decode failure
     */
    public static PrivateKey readPrivateKey(final String
pemResName) throws Exception {
        try (InputStream contentIS = TokenUtils.class
.getResourceAsStream(pemResName)) {
            byte[] tmp = new byte[4096];
            int length = contentIS.read(tmp);
            return decodePrivateKey(new String(tmp, 0, length,
"UTF-8"));
        }
    }

    /**
     * Decode a PEM encoded private key string to an RSA PrivateKey
     *
     * @param pemEncoded - PEM string for private key
     * @return PrivateKey
     * @throws Exception on decode failure
     */
    public static PrivateKey decodePrivateKey(final String
pemEncoded) throws Exception {
        byte[] encodedBytes = toEncodedBytes(pemEncoded);

        PKCS8EncodedKeySpec keySpec = new PKCS8EncodedKeySpec
(encodedBytes);
        KeyFactory kf = KeyFactory.getInstance("RSA");
        return kf.generatePrivate(keySpec);
    }

    private static byte[] toEncodedBytes(final String pemEncoded) {
        final String normalizedPem = removeBeginEnd(pemEncoded);
        return Base64.getDecoder().decode(normalizedPem);
    }

    private static String removeBeginEnd(String pem) {
        pem = pem.replaceAll("-----BEGIN (.*)-----", "");
    }

```

```

        pem = pem.replaceAll("-----END (.*)-----", "");
        pem = pem.replaceAll("\r\n", "");
        pem = pem.replaceAll("\n", "");
        return pem.trim();
    }

    /**
     * @return the current time in seconds since epoch
     */
    public static int currentTimeInSecs() {
        long currentTimeMS = System.currentTimeMillis();
        return (int) (currentTimeMS / 1000);
    }
}

```

Next take the code from the following listing and place into `security-jwt-quickstart/src/test/java/org/acme/security/jwt/GenerateToken.java`:

GenerateToken main Driver Class

```
package org.acme.security.jwt;

import java.util.HashMap;

import org.eclipse.microprofile.jwt.Claims;

/**
 * A simple utility class to generate and print a JWT token string
 * to stdout. Can be run with:
 * mvn exec:java
 * -Dexec.mainClass=org.acme.security.jwt.GenerateToken
 * -Dexec.classpathScope=test
 */
public class GenerateToken {
    /**
     * @param args - [0]: optional name of classpath resource for
     * json document of claims to add; defaults to "/JwtClaims.json"
     * [1]: optional time in seconds for expiration of
     * generated token; defaults to 300
     * @throws Exception
     */
    public static void main(String[] args) throws Exception {
        String claimsJson = "/JwtClaims.json";
        if (args.length > 0) {
            claimsJson = args[0];
        }
        HashMap<String, Long> timeClaims = new HashMap<>();
        if (args.length > 1) {
            long duration = Long.parseLong(args[1]);
            long exp = TokenUtils.currentTimeInSecs() + duration;
            timeClaims.put(Claims.exp.name(), exp);
        }
        String token = TokenUtils.generateTokenString(claimsJson,
            timeClaims);
        System.out.println(token);
    }
}
```

Now we need the content of the RSA private key that corresponds to the public key we have in the `TokenSecuredResource` application. Take the following PEM content and place it into `security-jwt-quickstart/src/test/resources/privateKey.pem`.

```
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBAcwggSjAgEAAoIBAQCWK8UjyoHgPTLa
PLQJ8SoXLLjpHSjtLxMqzmHnFscqHTVvADpCRcb6e3Ii/WniQTWw8RA7vf4dz4H
OzvlfBFNgvUGZHXDwnmGaNvaNzPHYFMEYBhE8VGGiveSkzqeLZI+Y02G6sQAfdtN
qqzM/15QX8X34oQFaTBW1r49nftvCpITiWjvWyhkWtXP9RP8sXi1im5Vi3dhup0h
nelk5n0BfajUYIbfHA60RzjHRbt7NtB10L2J+0/FUdHyKs6KMLFGNw800Dq88qnM
uXoLJiewhg9332W3DFMe0veel+//cvDnRsCRtPgd4sXFPHh+USHks07+DRsChXa6
oGGQD3GdAgMBAAECggEAAjftSZwMHwvIXIDZB+yP+pemg4ryt84iMlbofclQV8hv
6TsI4UGwcbKxFO5VSYxbN0isb80qasb929gixsyBjsQ8284bhPJR7r0q8h1C+jY
URA6S4pk8d/LmFakXwG9Tz6YPo3pJziuh481zkFTk0xW2Dp4SLwtAptZY/+ZXyJ6
96QXDrZKSSM99Jh9s7a0ST66WoxSS0UC51ak+Keb0KJ1jz4bIJ2C3r4rY1Su4hHB
Y73GfkWORTQuyUDa9yD0em0/z0nr6pp+pBSXPLHADsqvZiIhxD/00Xk5I6/zVHB3
zuoQqLERk0WvA8FXz2o8AYwcQRY2g30eX9kU4uDQAQKBgQDmf7KGImUGitsEPepF
KH5yLWYWqghHx6wfV+fdbBxoqn9WlwcQ7JbynIiVx8MX8/11LCCe8v41ypu/eLtp
iY1ev2IKdrUStvYRSsFigRkuPHUo1ajsGHQd+ucTdf58mn7kRLW1JGMeGxo/t32B
m96Af6AiPWPEJuVfgGV0iwg+HQKBgQCmyPzL9M2rhYZn1AozRUgUvlpMJHU2DpqS
34Q+7x2Ghf7MgBUhqE0t3FA0xEC7IYBwHmeY0vFR8ZkVRKNF4gbnF9RtLdz0DMEG
5qsMnvJUSQbNB1yVjUCnDAteLqiFRlQ/k0LgYkjKDY7Lfcizl9uJR100SYeX/qG2
tRW09t0pgQKBgBSGkpM3RN/MRayfBtmZvYjvVwH3yjkI2GbHA1jj1g6IebLB9SnfL
WbXJErCj1U+wvoPf5hfBc7m+jRgD3Eo86YXibQyZfY5pFIh9q7L15CQ15hj4zc4Y
b16sFR+xQ1Q9Pcd+BuBWmSz5J0E/qcF869dthgkGhnfVlt/0QzqZluZRAoGAXQ09
nT0TkmKIvlza5Af/YbTqEpg8m1BDhTYXPlWCD4+qvMWpBII1rSSBtftgCGca9XLB
MXmRmbqtQeRtg4u7dishZVh1MeP7vbHsNLppUQT90l6lFPsd2xUpJDc6BkFat62d
Xjr3iWNPC9E9nhPPdCNBv7reX7q81obpeXFMXgECgYEAmk2Q1us30V0tfoNRqNpe
Mb0teduf2+h3xaI1XDIzPVtZF35ELY/RkAHlmWRT4PCdR0zXDide67L6XdJyecSt
Fd0UH8z5qUraVVeBfVJqf/oGsXc4+ex1ZKUTbY0wqY1y9E39yvB3MaTmZFuuqk8
f3cg+fr8aou7pr9SHhJlZCU=
-----END PRIVATE KEY-----
```

And finally, we need to define what claims to include in the JWT. The `TokenUtils` class uses a json resource on the classpath to define the non-time sensitive claims, so take the content from the following listing and place it into `security-jwt-quickstart/src/test/resources/JwtClaims.json`:

JwtClaims.json claims document

```
{
  "iss": "https://quarkus.io/using-jwt-rbac",
  "jti": "a-123",
  "sub": "jdoe-using-jwt-rbac",
  "upn": "jdoe@quarkus.io",
  "preferred_username": "jdoe",
  "aud": "using-jwt-rbac",
  "birthdate": "2001-07-13",
  "roleMappings": {
    "group1": "Group1MappedRole",
    "group2": "Group2MappedRole"
  },
  "groups": [
    "Echoer",
    "Tester",
    "Subscriber",
    "group2"
  ]
}
```

Let's explore the content of this document in more detail to understand how the claims will affect our application security.

JwtClaims.json claims document

```
{
  "iss": "https://quarkus.io/using-jwt-rbac", ①
  "jti": "a-123",
  "sub": "jdoe-using-jwt-rbac",
  "upn": "jdoe@quarkus.io", ②
  "preferred_username": "jdoe",
  "aud": "using-jwt-rbac",
  "birthdate": "2001-07-13",
  "roleMappings": { ③
    "group1": "Group1MappedRole",
    "group2": "Group2MappedRole"
  },
  "groups": [ ④
    "Echoer",
    "Tester",
    "Subscriber",
    "group2"
  ]
}
```

① The `iss` claim is the issuer of the JWT. This needs to match the server side

`mp.jwt.verify.issuer` in order for the token to be accepted as valid.

- ② The `upn` claim is defined by the MicroProfile JWT RBAC spec as preferred claim to use for the `Principal` seen via the container security APIs.
- ③ The `roleMappings` claim can be used to map from a role defined in the `groups` claim to an application level role defined in a `@RolesAllowed` annotation. We won't use this feature in this quickstart, but it can be useful when the IDM providing the token has roles that do not directly align with those defined by the application.
- ④ The `group` claim provides the groups and top-level roles associated with the JWT bearer. In this quickstart we are only using the top-level role mapping which means the JWT will be seen to have the roles "Echoer", "Tester", "Subscriber" and "group2". The full set of roles would also include a "Group2MappedRole" due to the `roleMappings` claim having a mapping from "group2" to "Group2MappedRole".

Now we can generate a JWT to use with `TokenSecuredResource` endpoint. To do this, run the following command:

Command to Generate JWT

```
mvn exec:java -Dexec.mainClass=org.acme.security.jwt.GenerateToken
-Dexec.classpathScope=test
```



You may need to run `./mvnw test-compile` before this if you are working strictly from the command line and not an IDE that automatically compiles code as you write it.

Sample JWT Generation Output

```
$ mvn exec:java
-Dexec.mainClass=org.acme.security.jwt.GenerateToken
-Dexec.classpathScope=test
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.acme:security-jwt-quickstart
>-----
[INFO] Building security-jwt-quickstart 1.0-SNAPSHOT
[INFO] -----[ jar
]-----
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ security-
jwt-quickstart ---
Setting exp: 1551659976 / Sun Mar 03 16:39:36 PST 2019
    Added claim: sub, value: jdoe-using-jwt-rbac
    Added claim: aud, value: [using-jwt-rbac]
    Added claim: upn, value: jdoe@quarkus.io
    Added claim: birthdate, value: 2001-07-13
    Added claim: auth_time, value: 1551659676
    Added claim: iss, value: https://quarkus.io/using-jwt-rbac
```


oVM3ZUrBz57JKtr0e9jv0HjPQWyvbx1HuxZd6eA8ow8xzvooKXFxoSFCMnxotd3wagvYQ9ysBa89bgzL-1hjWtusuMFDUVYwFqADE7o0S0D4Vtclgq8svznBQ-YpfTHfb9QEcofMlpyjNA

If you start playing around with the code and/or the solution code, you will only be able to use a given token for 5-6 minutes because that is the default expiration period + grace period. To use a longer expiration, pass in the lifetime of the token in seconds as the second argument to the `GenerateToken` class using `-Dexec.args=...`. The first argument is the classpath resource name of the json document containing the claims to add to the JWT, and should be `/JwtClaims.json` for this quickstart.

Example Command to Generate JWT with Lifetime of 3600 Seconds

```
$ mvn exec:java
-Dexec.mainClass=org.acme.security.jwt.GenerateToken
-Dexec.classpathScope=test -Dexec.args="/JwtClaims.json 3600"
[INFO] Scanning for projects...
[INFO]
[INFO] -----< org.acme: >-----
[INFO] Building security-jwt-quickstart 1.0-SNAPSHOT
[INFO] -----[ jar
]-----
[INFO]
[INFO] --- exec-maven-plugin:1.6.0:java (default-cli) @ security-
jwt-quickstart ---
    Added claim: iss, value: https://quarkus.io/using-jwt-rbac
    Added claim: jti, value: a-123
    Added claim: sub, value: jdoe-using-jwt-rbac
    Added claim: upn, value: jdoe@quarkus.io
    Added claim: preferred_username, value: jdoe
    Added claim: aud, value: using-jwt-rbac
    Added claim: birthdate, value: 2001-07-13
    Added claim: roleMappings, value: {group1=Group1MappedRole,
group2=Group2MappedRole}
    Added claim: groups, value: [Echoer, Tester, Subscriber,
group2]
    Added claim: iat, value: 1571329458
    Added claim: auth_time, value: NumericDate{1571329458 -> Oct
17, 2019 5:24:18 PM IST}
    Added claim: exp, value: 1571333058
eyJraWQiOiIvcHJpdmF0ZUtleS5wZW0iLCJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1NiJ
9.eyJpc3MiOiJodHRwczovL3F1YXJrdXMuaW8vdXNpbmctand0LXJiYWMiLCJqdGkiOi
iJhLTEyMyIsInN1YiI6Impkb2UtdXNpbmctand0LXJiYWMiLCJ1c2UiOiJqZG91QHF1
YXJrdXMuaW8iLCJwcmVudXNpbmctand0LXJiYWMiLCJ1c2UiOiJqZG91IiwiaXVkiOiJj
oidXNpbmctand0LXJiYWMiLCJiaXN0aGRhdGUiOiIyMDAxLTA3LTEzIiwicm9sZU1hcHBp
bmdzIjp7Imdyb3VwMSI6Imdyb3VwMU1hcHB1ZiJvbnV0eXNpbmctand0LXJiYWMiLC
J1c2UiOiJqZG91IiwiaXV0aF90aW11IjoiTnVtZXJpY0RhZGV7MTU3MTMyOTQ1OCAtPi
BPY3QgMTcsIDIwMTkgNToyNDoxOj0CBQTSBJU1R9IiwiaXhwIjoxNTcxMzMzMDU4fQ.
Hn6f0qSk6wbbqOM-
```


curl Command for /secured/roles-allowed With JWT

```
$ curl -H "Authorization: Bearer eyJraWQ..."
http://127.0.0.1:8080/secured/roles-allowed; echo
hello + jdoe@quarkus.io, isSecure: false, authScheme: MP-JWT,
hasJWT: true
```

Success! We now have:

- a non-anonymous caller name of jdoe@quarkus.io
- an authentication scheme of Bearer
- a non-null `JsonWebToken`

Using the `JsonWebToken` and Claim Injection

Now that we can generate a JWT to access our secured REST endpoints, let's see what more we can do with the `JsonWebToken` interface and the JWT claims. The `org.eclipse.microprofile.jwt.JsonWebToken` interface extends the `java.security.Principal` interface, and is in fact the type of the object that is returned by the `javax.ws.rs.core.SecurityContext#getUserPrincipal()` call we used previously. This means that code that does not use CDI but does have access to the REST container `SecurityContext` can get hold of the caller `JsonWebToken` interface by casting the `SecurityContext#getUserPrincipal()`.

The `JsonWebToken` interface defines methods for accessing claims in the underlying JWT. It provides accessors for common claims that are required by the MicroProfile JWT RBAC specification as well as arbitrary claims that may exist in the JWT.

Let's expand our `TokenSecuredResource` with another endpoint `/secured/winners`. The `winners()` method, some hypothetical lottery winning number generator, whose code is shown in the following list:

TokenSecuredResource#winners Method Addition

```
package org.acme.security.jwt;

import java.security.Principal;
import java.time.LocalDate;
import java.util.ArrayList;

import javax.annotation.security.PermitAll;
import javax.annotation.security.RolesAllowed;
import javax.enterprise.context.RequestScoped;
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;

import org.eclipse.microprofile.jwt.Claims;
import org.eclipse.microprofile.jwt.JsonWebToken;

/**
 * Version 3 of the TokenSecuredResource
 */
@Path("/secured")
@RequestScoped
public class TokenSecuredResourceV3 {

    @Inject
    JsonWebToken jwt;

    ...

    @GET
    @Path("winners")
    @Produces(MediaType.TEXT_PLAIN)
    @RolesAllowed("Subscriber")
    public String winners() {
        int remaining = 6;
        ArrayList<Integer> numbers = new ArrayList<>();

        // If the JWT contains a birthdate claim, use the day of
        the month as a pick
        if (jwt.containsClaim(Claims.birthdate.name())) { ①
            String bdayString = jwt.getClaim(Claims.birthdate.name
()); ②
            LocalDate bday = LocalDate.parse(bdayString);
            numbers.add(bday.getDayOfMonth()); ③
            remaining --;
        }
        // Fill remaining picks with random numbers
        while(remaining > 0) { ④
            int pick = (int) Math rint(64 * Math.random() + 1);
            numbers.add(pick);
            remaining --;
        }
        return numbers.toString();
    }
}

```

① Here we use the injected `JsonWebToken` to check for a `birthdate` claim.

② If it exists, we obtain the claim value as a `String`, and then convert it to a `LocalDate`.


```

package org.acme.security.jwt;

import java.security.Principal;
import java.time.LocalDate;
import java.util.ArrayList;
import java.util.Optional;

import javax.annotation.security.PermitAll;
import javax.annotation.security.RolesAllowed;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import javax.json.JsonString;
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;

import org.eclipse.microprofile.jwt.Claim;
import org.eclipse.microprofile.jwt.Claims;
import org.eclipse.microprofile.jwt.JsonWebToken;

/**
 * Version 4 of the TokenSecuredResource
 */
@Path("/secured")
@RequestScoped
public class TokenSecuredResource {

    @Inject
    JsonWebToken jwt;
    @Inject ①
    @Claim(standard = Claims.birthdate) ②
    Optional<JsonString> birthdate; ③

    ...

    @GET
    @Path("winners2")
    @Produces(MediaType.TEXT_PLAIN)
    @RolesAllowed("Subscriber")
    public String winners2() {
        int remaining = 6;
        ArrayList<Integer> numbers = new ArrayList<>();

        // If the JWT contains a birthdate claim, use the day of
        the month as a pick
        if (birthdate.isPresent()) { ④
            String bdayString = birthdate.get().getString(); ⑤

```

```

        LocalDate bday = LocalDate.parse(bdayString);
        numbers.add(bday.getDayOfMonth());
        remaining --;
    }
    // Fill remaining picks with random numbers
    while(remaining > 0) {
        int pick = (int) Math rint(64 * Math.random() + 1);
        numbers.add(pick);
        remaining --;
    }
    return numbers.toString();
}
}

```

- ① We use CDI `@Inject` along with...
- ② an MicroProfile JWT RBAC `@Claim(standard = Claims.birthdate)` qualifier to inject the `birthdate` claim directly as
- ③ an `Optional<JsonString>` value.
- ④ Now we check whether the injected `birthdate` field is present
- ⑤ and if it is, get its value.

The remainder of the code is the same as before. Update your `TokenSecuredResource` to either add or replace the current `winners()` method, and then invoke the following command with `YOUR_TOKEN` replaced:

curl command for /secured/winners2

```

curl -H "Authorization: Bearer YOUR_TOKEN"
http://localhost:8080/secured/winners2; echo

```



```

Scotts-iMacPro:security-jwt-quickstart starksm$ ./mvnw clean
package -Pnative
[INFO] Scanning for projects...
...
[security-jwt-quickstart-runner:25602]    universe:      493.17 ms
[security-jwt-quickstart-runner:25602]    (parse):      660.41 ms
[security-jwt-quickstart-runner:25602]    (inline):     1,431.10 ms
[security-jwt-quickstart-runner:25602]    (compile):    7,301.78 ms
[security-jwt-quickstart-runner:25602]    compile:     10,542.16 ms
[security-jwt-quickstart-runner:25602]    image:       2,797.62 ms
[security-jwt-quickstart-runner:25602]    write:       988.24 ms
[security-jwt-quickstart-runner:25602]    [total]:     43,778.16 ms
[INFO]
-----
[INFO] BUILD SUCCESS
[INFO]
-----
[INFO] Total time: 51.500 s
[INFO] Finished at: 2019-03-28T14:30:56-07:00
[INFO]
-----

Scotts-iMacPro:security-jwt-quickstart starksm$ ./target/security-
jwt-quickstart-runner
2019-03-28 14:31:37,315 INFO [io.quarkus] (main) Quarkus 0.12.0
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, smallrye-jwt]

```

Explore the Solution

The solution repository located in the [security-jwt-quickstart directory](#) contains all of the versions we have worked through in this quickstart guide as well as some additional endpoints that illustrate subresources with injection of [JsonWebTokens](#) and their claims into those using the CDI APIs. We suggest that you check out the quickstart solutions and explore the [security-jwt-quickstart](#) directory to learn more about the Smallrye JWT extension features.

Configuration Reference

Quarkus configuration

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

Configuration property	Type	Default
 <code>quarkus.smallrye-jwt.enabled</code> The MP-JWT configuration object	boolean	<code>true</code>
 <code>quarkus.smallrye-jwt.rsa-sig-provider</code> The name of the <code>java.security.Provider</code> that supports SHA256withRSA signatures	string	<code>SunRsaSign</code>

MicroProfile JWT configuration

Property Name	Default	Description
<code>mp.jwt.verify.publickey</code>	<code>none</code>	The <code>mp.jwt.verify.publickey</code> config property allows the Public Key text itself to be supplied as a string. The Public Key will be parsed from the supplied string in the order defined in section Supported Public Key Formats .
<code>mp.jwt.verify.publickey.location</code>	<code>none</code>	Config property allows for an external or internal location of Public Key to be specified. The value may be a relative path or a URL. If the value points to an HTTPS based JWK set then, for it to work in native mode, the <code>quarkus.ssl.native</code> property must also be set to <code>true</code> , see Using SSL With Native Executables for more details.
<code>mp.jwt.verify.issuer</code>	<code>none</code>	Config property specifies the value of the <code>iss</code> (issuer) claim of the JWT that the server will accept as valid.

Supported Public Key Formats

Public Keys may be formatted in any of the following formats, specified in order of precedence:

- Public Key Cryptography Standards #8 (PKCS#8) PEM
- JSON Web Key (JWK)
- JSON Web Key Set (JWKS)
- JSON Web Key (JWK) Base64 URL encoded
- JSON Web Key Set (JWKS) Base64 URL encoded

Additional Smallrye JWT configuration

Smallrye JWT provides more properties which can be used to customize the token processing:

Property Name	Default	Description
<code>smallrye.jwt.verify.algorithm</code>	<code>RS256</code>	Signature algorithm. Set it to <code>ES256</code> to support the Elliptic Curve signature algorithm.
<code>smallrye.jwt.token.header</code>	<code>Authorization</code>	Set this property if another header such as <code>Cookie</code> is used to pass the token.
<code>smallrye.jwt.token.cookie</code>	<code>none</code>	Name of the cookie containing a token. This property will be effective only if <code>smallrye.jwt.token.header</code> is set to <code>Cookie</code> .
<code>smallrye.jwt.always-check-authorization</code>	<code>false</code>	Set this property to true for Authorization header be checked even if the <code>smallrye.jwt.token.header</code> is set to <code>Cookie</code> but no cookie with a <code>smallrye.jwt.token.cookie</code> name exists.
<code>smallrye.jwt.token.schemes</code>	<code>Bearer</code>	Comma-separated list containing an alternative single or multiple schemes, for example, <code>DPoP</code> .
<code>smallrye.jwt.token.kid</code>	<code>none</code>	Key identifier. If it is set then the verification JWK key as well every JWT token must have a matching <code>kid</code> header.
<code>smallrye.jwt.time-to-live</code>	<code>none</code>	The maximum number of seconds that a JWT may be issued for use. Effectively, the difference between the expiration date of the JWT and the issued at date must not exceed this value.
<code>smallrye.jwt.require.named-principal</code>	<code>false</code>	If an application relies on <code>java.security.Principal</code> returning a name then a token must have a <code>upn</code> or <code>preferred_username</code> or <code>sub</code> claim set. Setting this property will result in Smallrye JWT throwing an exception if none of these claims is available for the application code to reliably deal with a non-null <code>Principal</code> name.
<code>smallrye.jwt.path.sub</code>	<code>none</code>	Path to the claim containing the subject name. It starts from the top level JSON object and can contain multiple segments where each segment represents a JSON object name only, example: <code>realms/subject</code> . This property can be used if a token has no 'sub' claim but has the subject set in a different claim. Use double quotes with the namespace qualified claims.

Property Name	Default	Description
<code>smallrye.jwt.claims.sub</code>	<code>none</code>	This property can be used to set a default sub claim value when the current token has no standard or custom <code>sub</code> claim available. Effectively this property can be used to customize <code>java.security.Principal</code> name if no <code>upn</code> or <code>preferred_username</code> or <code>sub</code> claim is set.
<code>smallrye.jwt.path.groups</code>	<code>none</code>	Path to the claim containing the groups. It starts from the top level JSON object and can contain multiple segments where each segment represents a JSON object name only, example: <code>realm/groups</code> . This property can be used if a token has no 'groups' claim but has the groups set in a different claim. Use double quotes with the namespace qualified claims.
<code>smallrye.jwt.groups-separator</code>	<code>' '</code>	Separator for splitting a string which may contain multiple group values. It will only be used if the <code>smallrye.jwt.path.groups</code> property points to a custom claim whose value is a string. The default value is a single space because a standard OAuth2 <code>scope</code> claim may contain a space separated sequence.
<code>smallrye.jwt.claims.groups</code>	<code>none</code>	This property can be used to set a default groups claim value when the current token has no standard or custom groups claim available.
<code>smallrye.jwt.jwks.refresh-interval</code>	<code>60</code>	JWK cache refresh interval in minutes. It will be ignored unless the <code>mp.jwt.verify.publickey.location</code> points to the HTTPS URL based JWK set and no HTTP <code>Cache-Control</code> response header with a positive <code>max-age</code> parameter value is returned from a JWK HTTPS endpoint.
<code>smallrye.jwt.expiration.grace</code>	<code>60</code>	Expiration grace in seconds. By default an expired token will still be accepted if the current time is no more than 1 min after the token expiry time.
<code>smallrye.jwt.verify.aud</code>	<code>none</code>	Comma separated list of the audiences that a token <code>aud</code> claim may contain.

Generate JWT tokens with Smallrye JWT

JWT claims can be signed or encrypted or signed first and the nested JWT token encrypted. Signing the claims is used most often to secure the claims. What is known today as a JWT token is typically produced by signing the claims in a JSON format using the steps described in the [JSON Web Signature](#) specification. However, when the claims are sensitive, their confidentiality can be guaranteed by

following the steps described in the [JSON Web Encryption](#) specification to produce a JWT token with the encrypted claims. Finally both the confidentiality and integrity of the claims can be further enforced by signing them first and then encrypting the nested JWT token.

SmallRye JWT provides an API for securing the JWT claims using all of these options.

Create JwtClaimsBuilder and set the claims

The first step is to initialize a `JwtClaimsBuilder` using one of the options below and add some claims to it:

```
import java.util.Collections;
import io.smallrye.jwt.build.Jwt;
import io.smallrye.jwt.build.JwtClaimsBuilder;
...
// Create an empty builder and add some claims
JwtClaimsBuilder builder1 = Jwt.claims();
builder1.claim("customClaim", "custom-value").issuer(
    "https://issuer.org");

// Builder created from the existing claims
JwtClaimsBuilder builder2 = Jwt.claims("/tokenClaims.json");

// Builder created from a map of claims
JwtClaimsBuilder builder3 = Jwt.claims(Collections.singletonMap(
    "customClaim", "custom-value"));
```

The API is fluent so the builder initialization can be done as part of the fluent API sequence. The builder will also set `iat` (issued at) to the current time, `exp` (expires at) to 5 minutes away from the current time and `jti` (unique token identifier) claims if they have not already been set, so one can skip setting them when possible.

The next step is to decide how to secure the claims.

Sign the claims

The claims can be signed immediately or after the [JSON Web Signature](#) headers have been set:

```

import io.smallrye.jwt.build.Jwt;
...

// Sign the claims using the private key loaded from the location
set with a 'smallrye.jwt.sign.key-location' property.
// No 'jws()' transition is necessary.
String jwt1 = Jwt.claims("/tokenClaims.json").sign();

// Set the headers and sign the claims with an RSA private key
loaded in the code (the implementation of this method is omitted).
Note a 'jws()' transition to a 'JwtSignatureBuilder'.
String jwt2 = Jwt.claims("/tokenClaims.json").jws().signatureKeyId
("kid1").header("custom-header", "custom-value").sign(
getPrivateKey());

```

Note the `alg` (algorithm) header is set to `RS256` by default.

Encrypt the claims

The claims can be encrypted immediately or after the `JSON Web Encryption` headers have been set the same way as they can be signed. The only minor difference is that encrypting the claims always requires a `jwe()` `JwtEncryptionBuilder` transition:

```

import io.smallrye.jwt.build.Jwt;
...

// Encrypt the claims using the public key loaded from the location
set with a 'smallrye.jwt.encrypt.key-location' property.
String jwt1 = Jwt.claims("/tokenClaims.json").jwe().encrypt();

// Set the headers and encrypt the claims with an RSA public key
loaded in the code (the implementation of this method is omitted).
String jwt2 = Jwt.claims("/tokenClaims.json").jwe().header("custom-
header", "custom-value").encrypt(getPublicKey());

```

Note the `alg` (key management algorithm) header is set to `RSA-OAEP-256` (it will be changed to `RSA-OAEP` in a future version of `smallrye-jwt`) and the `enc` (content encryption header) is set to `A256GCM` by default.

Sign the claims and encrypt the nested JWT token

The claims can be signed and then the nested JWT token encrypted by combining the sign and encrypt steps.

```
import io.smallrye.jwt.build.Jwt;
...

// Sign the claims and encrypt the nested token using the private
// and public keys loaded from the locations set with the
// 'smallrye.jwt.sign.key-location' and 'smallrye.jwt.encrypt.key-
// location' properties respectively.
String jwt = Jwt.claims("/tokenClaims.json").innerSign().encrypt();
```

References

- [MP JWT 1.1.1](#)
- [Smallrye JWT](#)
- [JSON Web Token](#)
- [JSON Web Signature](#)
- [JSON Web Encryption](#)
- [JSON Web Algorithms](#)