Quarkus - Using JWT RBAC

This guide explains how your Quarkus application can utilize MicroProfile JWT (MP JWT) to verify JSON Web Tokens, represent them as MP JWT `org.eclipse.microprofile.jwt.JsonWebToken` and provide secured access to the Quarkus HTTP endpoints using Bearer Token Authorization and Role-Based Access Control.

> Quarkus OpenId Connect extension also supports Bearer Token Authorization and uses `smallrye-jwt` to represent the bearer tokens as `JsonWebToken`, please read the Using OpenID Connect to Protect Service Applications guide for more information. 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.

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:1.8.0.Final:create \
    -DprojectGroupId=org.acme \
    -DprojectArtifactId=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.

If you already have your Quarkus project configured, you can add the `smallrye-jwt` extension to your project by running the following command in your project base directory:

```
./mvnw quarkus:add-extension -Dextensions="smallrye-jwt"
```

This will add the following to your `pom.xml`:

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

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.

REST Endpoint V1

```
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.InternalServerErrorException;
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;

@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) {
        return getResponseString(ctx); ④
    }

    private String getResponseString(SecurityContext ctx) {
        String name;
        if (ctx.getUserPrincipal() == null) { ⑤
            name = "anonymous";
        } else if
(!ctx.getUserPrincipal().getName().equals(jwt.getName())) { ⑥
            throw new InternalServerErrorException("Principal and
JsonWebToken names do not match");
        } else {
            name = ctx.getUserPrincipal().getName(); ⑦
        }
        return String.format("hello + %s,"
            + " isHttps: %s,"
            + " authScheme: %s,"
            + " hasJWT: %s",
            name, ctx.isSecure(), ctx.getAuthenticationScheme(),
hasJwt()); ⑧
    }

    private boolean hasJwt() {
    return jwt.getClaimNames() != null;
    }
}
```

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 and use a `getResponseString()` function to populate a response string.

⑤ Here we check if the call is insecured by checking the request user/caller `Principal` against null.

⑥ Here we check that the Principal and JsonWebToken have the same name since JsonWebToken does represent the current Principal.

⑦ Here we get the Principal name.

⑧ 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
2020-07-15 16:09:50,883 INFO  [io.quarkus] (Quarkus Main Thread)
security-jwt-quickstart 1.0-SNAPSHOT on JVM (powered by Quarkus
999-SNAPSHOT) started in 1.073s. Listening on: http://0.0.0.0:8080
2020-07-15 16:09:50,885 INFO  [io.quarkus] (Quarkus Main Thread)
Profile dev activated. Live Coding activated.
2020-07-15 16:09:50,885 INFO  [io.quarkus] (Quarkus Main Thread)
Installed features: [cdi, mutiny, resteasy, resteasy-jsonb,
security, smallrye-context-propagation, 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, isHttps: 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
- isHttps 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 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.InternalServerErrorException;
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;

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

    @Inject
    JsonWebToken jwt; ①

    @GET
    @Path("permit-all")
    @PermitAll
    @Produces(MediaType.TEXT_PLAIN)
    public String hello(@Context SecurityContext ctx) {
        return getResponseString(ctx);
    }

    @GET
    @Path("roles-allowed") ②
    @RolesAllowed({ "User", "Admin" }) ③
    @Produces(MediaType.TEXT_PLAIN)
    public String helloRolesAllowed(@Context SecurityContext ctx) {
        return getResponseString(ctx) + ", birthdate: " +
jwt.getClaim("birthdate").toString(); ④
    }

    private String getResponseString(SecurityContext ctx) {
        String name;
        if (ctx.getUserPrincipal() == null) {
            name = "anonymous";
        } else if
(!ctx.getUserPrincipal().getName().equals(jwt.getName())) {
            throw new InternalServerErrorException("Principal and
JsonWebToken names do not match");
        } else {
            name = ctx.getUserPrincipal().getName();
        }
        return String.format("hello + %s,"
            + " isHttps: %s,"
            + " authScheme: %s,"
            + " hasJWT: %s",
            name, ctx.isSecure(), ctx.getAuthenticationScheme(),
hasJwt());
    }

    private boolean hasJwt() {
        return jwt.getClaimNames() != null;
    }
}
```

① Here we inject `JsonWebToken`

② This new endpoint will be located at /secured/roles-allowed

③ @RolesAllowed is a JSR 250 common security annotation that indicates that the given endpoint is accessible by a caller if they have either a "User" or "Admin" role assigned.

④ Here we build the reply the same way as in the `hello` method but also add a value of the JWT `birthdate` claim by directly calling the injected `JsonWebToken`.

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

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
```

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`.

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.

Adding `publicKey.pem` to `resources/META-INF/resources` ensures that it is available in the native image without having to provide a GraalVM resource file.

RSA Public Key PEM Content

```
-----BEGIN PUBLIC KEY-----
MIIBIjANBgkqhkiG9w0BAQEFAAOCAQ8AMIIBCgKCAQEAlivFI8qB4D0y2jy0CfEq
Fyy46R0o7S8TKpsx5xbHKoU1VWg6QkQm+ntyIv1p4kE1sPEQO73+HY8+Bzs75XwR
TYL1BmR1w8J5hmjVWjc6R2BTBGAYRPFRhor3kpM6ni2SPmNNhurEAHw7TaqszP5e
UF/F9+KEBWkwVta+PZ37bwqSE4sCb1soZFrVz/UT/LF4tYpuVYt3YbqToZ3pZOZ9
AX2o1GCG3xwOjkc4x0W7ezbQZdC9iftPxVHR8irOijJRRjcPDtA6vPKpzLl6CyYn
sIYPd99ltwxTHjr3npfv/3Lw50bAkbT4HeLFxTx4flEoZLKO/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).

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

GenerateToken main Driver Class

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

import java.util.Arrays;
import java.util.HashSet;

import org.eclipse.microprofile.jwt.Claims;

import io.smallrye.jwt.build.Jwt;

public class GenerateToken {
    /**
     * Generate JWT token
     */
    public static void main(String[] args) {
        String token =
           Jwt.issuer("https://quarkus.io/using-jwt-rbac") ①
             .upn("jdoe@quarkus.io") ②
             .groups(new HashSet<>(Arrays.asList("User", "Admin")))
③
             .claim(Claims.birthdate.name(), "2001-07-13") ④
           .sign();
        System.out.println(token);
    }
}
```

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 `group` claim provides the groups and top-level roles associated with the JWT bearer.

④ The `birthday` claim. It can be considered to be a sensitive claim so you may want to consider encrypting the claims, see Generate JWT tokens with SmallRye JWT.

Note for this code to work 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/main/resources/META-INF/resources/privateKey.pem`:

RSA Private Key PEM Content

```
-----BEGIN PRIVATE KEY-----
MIIEvQIBADANBgkqhkiG9w0BAQEFAASCBKcwggSjAgEAAoIBAQCWK8UjyoHgPTLa
PLQJ8SoXLLjpHSjtLxMqmzHnFscqhTVVaDpCRCb6e3Ii/WniQTWw8RA7vf4djz4H
OzvlfBFNgvUGZHXDwnmGaNVaNzpHYFMEYBhE8VGGiveSkzqeLZI+Y02G6sQAfDtN
qqzM/l5QX8X34oQFaTBW1r49nftvCpITiwJvWyhkWtXP9RP8sXi1im5Vi3dhupOh
nelk5n0BfajUYIbfHA6ORzjHRbt7NtBl0L2J+0/FUdHyKs6KMlFGNw8O0Dq88qnM
uXoLJiewhg9332W3DFMeOveel+//cvDnRsCRtPgd4sXFPHh+UShkso7+DRsChXa6
oGGQD3GdAgMBAAECggEAAjfTSZwMHwvIXIDZB+yP+pemg4ryt84iMlbofclQV8hv
6TsI4UGwcbKxFOM5VSYxbNOisb80qasb929gixsyBjsQ8284bhPJR7r0q8h1C+jY
URA6S4pk8d/LmFakXwG9Tz6YPo3pJziuh48lzkFTk0xW2Dp4SLwtAptZY/+ZXyJ6
96QXDrZKSSM99Jh9s7a0ST66WoxSS0UC51ak+Keb0KJ1jz4bIJ2C3r4rYlSu4hHB
Y73GfkWORtQuyUDa9yDOem0/z0nr6pp+pBSXPLHADsqvZiIhxD/O0Xk5I6/zVHB3
zuoQqLERk0WvA8FXz2o8AYwcQRY2g30eX9kU4uDQAQKBgQDmf7KGImUGitsEPepF
KH5yLWYWqghHx6wfV+fdbBxoqn9WlwcQ7JbynIiVx8MX8/1lLCCe8v41ypu/eLtP
iY1ev2IKdrUStvYRSsFigRkuPHUo1ajsGHQd+ucTDf58mn7kRLW1JGMeGxo/t32B
m96Af6AiPWPEJuVfgGV0iwg+HQKBgQCmyPzL9M2rhYZn1AozRUguvlpmJHU2DpqS
34Q+7x2Ghf7MgBUhqE0t3FAOxEC7IYBwHmeYOvFR8ZkVRKNF4gbnF9RtLdz0DMEG
5qsMnvJUSQbNB1yVjUCnDAtElqiFRlQ/k0LgYkjKDY7LfciZl9uJRl0OSYeX/qG2
tRW09tOpgQKBgBSGkpM3RN/MRayfBtmZvYjVWh3yjkI2GbHA1jj1g6IebLB9SnfL
WbXJErCj1U+wvoPf5hfBc7m+jRgD3Eo86YXibQyZfY5pFIh9q7Ll5CQl5hj4zc4Y
b16sFR+xQ1Q9Pcd+BuBWmSz5JOE/qcF869dthgkGhnfVLt/OQzqZluZRAoGAXQ09
nT0TkmKIvlza5Af/YbTqEpq8mlBDhTYXPlWCD4+qvMWpBII1rSSBtftgcgca9XLB
MXmRMbqtQeRtg4u7dishZVh1MeP7vbHsNLppUQT9Ol6lFPsd2xUpJDc6BkFat62d
Xjr3iWNPC9E9nhPPdCNBv7reX7q81obpeXFMXgECgYEAmk2Qlus3OV0tfoNRqNpe
Mb0teduf2+h3xaI1XDIzPVtZF35ELY/RkAHlmWRT4PCdR0zXDidE67L6XdJyecSt
FdOUH8z5qUraVVebRFvJqf/oGsXc4+ex1ZKUTbY0wqY1y9E39yvB3MaTmZFuuqk8
f3cg+fr8aou7pr9SHhJlZCU=
-----END PRIVATE KEY-----
```

We will use a `smallrye.jwt.sign.key-location` property to point to this private signing key.

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

Sample JWT Generation Output

```
$ mvn exec:java
-Dexec.mainClass=org.acme.security.jwt.GenerateToken
-Dexec.classpathScope=test -Dsmallrye.jwt.sign.key
-location=privateKey.pem

eyJraWQiOiJcL3ByaXZhdGVLZXkucGVtIiwidHlwIjoiSldUIiwiYWxnIjoiUlMyNTY
ifQ.eyJzdWIiOiJqZG9lLXVzaW5nLWp3dC1yYmFjIiwiYXVkIjoidXNpbmctand0LXJ
iYWMiLCJ1cG4iOiJqZG9lQHF1YXJrdXMuaW8iLCJiaXJ0aGRhdGUiOiIyMDAxLTA3LT
EzIiwiYXV0aF90aW1lIjoxNTUxNjU5Njc2LCJpc3MiOiJodHRwczpcL1wvcXVhcmt1c
y5pb1wvdXNpbmctand0LXJiYWMiLCJyb2xlTWFwcGluZ3MiOnsiZ3JvdXAyIjoiR3Jv
dXAyTWFwcGVkUm9sZSIsImdyb3VwMSI6Ikdyb3VwMU1hcHBlZFJvbGUifSwiZ3JvdXB
zIjpbIkVjaG9lciIsIlRlc3RlciIsIlN1YnNjcmliZXIiLCJncm91cDIiXSwicHJlZm
VycmVkX3VzZXJuYW1lIjoiamRvZSIsImV4cCI6MTU1MTY1OTk3NiwiaWF0IjoxNTUxN
jU5Njc2LCJqdGkiOiJhLTEyMyJ9.O9tx_wNNS4qdpFhxeD1e7v4aBNWz1FCq0UV8qmX
d7dW9xM4hA5TO-ZREk3ApMrL7_rnX8z81qGPIo_R8IfHDyNaI1SLD56gVX-
NaOLS2OjfcbO3zOWJPKR_BoZkYACtMoqlWgIwIRC-
wJKUJU025dHZiNL0FWO4PjwuCz8hpZYXIuRscfFhXKrDX1fh3jDhTsOEFfu67ACd85f
3BdX9pe-ayKSVLh_RSbTbBPeyoYPE59FW7H5-i8IE-Gqu838Hz0i38ksEJFI25eR-
AJ6_PSUD0_-TV3NjXhF3bFIeT4VSaIZcpibekoJg0cQm-4ApPEcPLdgTejYHA-
mupb8hSwg
```

The JWT string is the Base64 URL encoded string that has 3 parts separated by '.' characters. First part - JWT headers, second part - JWT claims, third part - JWT signature.

Finally, Secured Access to /secured/roles-allowed

Now let’s use this to make a secured request to the /secured/roles-allowed endpoint. Make sure you have the Quarkus server running using the `./mvnw compile quarkus:dev` command, and then run the following command, making sure to use your version of the generated JWT from the previous step:

```
curl -H "Authorization: Bearer
eyJraWQiOiJcL3ByaXZhdGVLZXkucGVtIiwidHlwIjoiSldUIiwiYWxnIjoiUlMyNTY
ifQ.eyJzdWIiOiJqZG9lLXVzaW5nLWp3dC1yYmFjIiwiYXVkIjoidXNpbmctand0LXJ
iYWMiLCJ1cG4iOiJqZG9lQHF1YXJrdXMuaW8iLCJiaXJ0aGRhdGUiOiIyMDAxLTA3LT
EzIiwiYXV0aF90aW1lIjoxNTUxNjUyMDkxLCJpc3MiOiJodHRwczpcL1wvcXVhcmt1c
y5pb1wvdXNpbmctand0LXJiYWMiLCJyb2xlTWFwcGluZ3MiOnsiZ3JvdXAyIjoiR3Jv
dXAyTWFwcGVkUm9sZSIsImdyb3VwMSI6Ikdyb3VwMU1hcHBlZFJvbGUifSwiZ3JvdXB
zIjpbIkVjaG9lciIsIlRlc3RlciIsIlN1YnNjcmliZXIiLCJncm91cDIiXSwicHJlZm
VycmVkX3VzZXJuYW1lIjoiamRvZSIsImV4cCI6MTU1MTY1MjM5MSwiaWF0IjoxNTUxN
jUyMDkxLCJqdGkiOiJhLTEyMyJ9.aPA4Rlc4kw7n_OZZRRk25xZydJy_J_3BRR8ryYL
yHTO1o68_aNWWQCgpnAuOW64svPhPnLYYnQzK-
l2vHX34B64JySyBD4y_vRObGmdwH_SEufBAWZV7mkG3Y4mTKT3_4EWNu4VH92IhdnkG
I4GJB6yHAEzlQI6EdSOa4Nq8Gp4uPGqHsUZTJrA3uIW0TbNshFBm47-
oVM3ZUrBz57JKtr0e9jv0HjPQWyvbzx1HuxZd6eA8ow8xzvooKXFxoSFCMnxotd3wag
vYQ9ysBa89bgzL-lhjWtusuMFDUVYwFqADE7oOSOD4Vtclgq8svznBQ-
YpfTHfb9QEcofMlpyjNA" http://127.0.0.1:8080/secured/roles-allowed;
echo
```

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, isHttps: false, authScheme: MP-JWT,
hasJWT: true, birthdate: 2001-07-13
```

Success! We now have:

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

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.

All the JWT claims can also be injected. Let’s expand our `TokenSecuredResource` with another endpoint /secured/roles-allowed-admin which users the injected `birthdate` claim (as opposed to getting it from `JsonWebToken`):

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

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.InternalServerErrorException;
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;

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

    @Inject
    JsonWebToken jwt;
    @Inject
    @Claim(standard = Claims.birthdate)
    String birthdate; ①

    @GET
    @Path("permit-all")
    @PermitAll
    @Produces(MediaType.TEXT_PLAIN)
    public String hello(@Context SecurityContext ctx) {
        return getResponseString(ctx);
    }

    @GET
    @Path("roles-allowed")
    @RolesAllowed({ "User", "Admin" })
    @Produces(MediaType.TEXT_PLAIN)
    public String helloRolesAllowed(@Context SecurityContext ctx) {
        return getResponseString(ctx) + ", birthdate: " +
jwt.getClaim("birthdate").toString();
```

```
    }

    @GET
    @Path("roles-allowed-admin")
    @RolesAllowed("Admin")
    @Produces(MediaType.TEXT_PLAIN)
    public String helloRolesAllowedAdmin(@Context SecurityContext
ctx) {
        return getResponseString(ctx) + ", birthdate: " +
birthdate; ②
    }

    private String getResponseString(SecurityContext ctx) {
        String name;
        if (ctx.getUserPrincipal() == null) {
            name = "anonymous";
        } else if
(!ctx.getUserPrincipal().getName().equals(jwt.getName())) {
            throw new InternalServerErrorException("Principal and
JsonWebToken names do not match");
        } else {
            name = ctx.getUserPrincipal().getName();
        }
        return String.format("hello + %s,"
            + " isHttps: %s,"
            + " authScheme: %s,"
            + " hasJWT: %s",
            name, ctx.isSecure(), ctx.getAuthenticationScheme(),
hasJwt());
    }

    private boolean hasJwt() {
        return jwt.getClaimNames() != null;
    }
}
```

① Here we use the injected `birthday` claim.

② Here we use the injected `birthday` claim to build the final reply.

Now generate the token again and run:

```
curl -H "Authorization: Bearer
eyJraWQiOiJcL3ByaXZhdGVLZXkucGVtIiwidHlwIjoiSldUIiwiYWxnIjoiUlMyNTY
ifQ.eyJzdWIiOiJqZG9lLXVzaW5nLWp3dC1yYmFjIiwiYXVkIjoidXNpbmctand0LXJ
iYWMiLCJ1cG4iOiJqZG9lQHF1YXJrdXMuaW8iLCJiaXJ0aGRhdGUiOiIyMDAxLTA3LT
EzIiwiYXV0aF90aW1lIjoxNTUxNjUyMDkxLCJpc3MiOiJodHRwczpcL1wvcXVhcmt1c
y5pb1wvdXNpbmctand0LXJiYWMiLCJyb2xlTWFwcGluZ3MiOnsiZ3JvdXAyIjoiR3Jv
dXAyTWFwcGVkUm9sZSIsImdyb3VwMSI6Ikdyb3VwMU1hcHBlZFJvbGUifSwiZ3JvdXB
zIjpbIkVjaG9lciIsIlRlc3RlciIsIlN1YnNjcmliZXIiLCJncm91cDIiXSwicHJlZm
VycmVkX3VzZXJuYW1lIjoiamRvZSIsImV4cCI6MTU1MTY1MjM5MSwiaWF0IjoxNTUxN
jUyMDkxLCJqdGkiOiJhLTEyMyJ9.aPA4Rlc4kw7n_OZZRRk25xZydJy_J_3BRR8ryYL
yHTO1o68_aNWWQCgpnAuOW64svPhPnLYYnQzK-
l2vHX34B64JySyBD4y_vRObGmdwH_SEufBAWZV7mkG3Y4mTKT3_4EWNu4VH92IhdnkG
I4GJB6yHAEzlQI6EdSOa4Nq8Gp4uPGqHsUZTJrA3uIW0TbNshFBm47-
oVM3ZUrBz57JKtr0e9jv0HjPQWyvbzx1HuxZd6eA8ow8xzvooKXFxoSFCMnxotd3wag
vYQ9ysBa89bgzL-lhjWtusuMFDUVYwFqADE7oOSOD4Vtclgq8svznBQ-
YpfTHfb9QEcofMlpyjNA" http://127.0.0.1:8080/secured/roles-allowed-
admin; echo
```

```
$ curl -H "Authorization: Bearer eyJraWQ..."
http://127.0.0.1:8080/secured/roles-allowed-admin; echo
hello + jdoe@quarkus.io, isHttps: false, authScheme: MP-JWT,
hasJWT: true, birthdate: 2001-07-13
```

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

```
Scotts-iMacPro:security-jwt-quickstart starksm$ ./mvnw clean
package
[INFO] Scanning for projects...
...
[INFO] [io.quarkus.creator.phase.runnerjar.RunnerJarPhase] Building
jar: /Users/starksm/Dev/JBoss/Protean/starksm64-quarkus-
quickstarts/security-jwt-quickstart/target/security-jwt-quickstart-
runner.jar

Scotts-iMacPro:security-jwt-quickstart starksm$ java -jar
target/security-jwt-quickstart-runner.jar
2019-03-28 14:27:48,839 INFO  [io.quarkus] (main) Quarkus 0.12.0
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, smallrye-jwt]
```

You can also generate the native executable with `./mvnw clean package -Pnative`. .Native Executable Example

```
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 `JsonWebToken`s 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 runtime

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

MicroProfile JWT configuration

Property Name	Default	Description
`mp.jwt.verify.publickey`	`none`	The `mp.jwt.verify.publickey` 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.
`mp.jwt.verify.publickey.location`	`none`	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 `quarkus.ssl.native` property must also be set to `true`, see Using SSL With Native Executables for more details.
`mp.jwt.verify.issuer`	`none`	Config property specifies the value of the `iss` (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
`smallrye.jwt.verify.algorithm`	`RS256`	Signature algorithm. Set it to `ES256` to support the Elliptic Curve signature algorithm.
`smallrye.jwt.verify.key-format`	`ANY`	Set this property to a specific key format such as `PEM_KEY`, `PEM_CERTIFICATE`, `JWK` or `JWK_BASE64URL` to optimize the way the verification key is loaded.
`smallrye.jwt.verify.relax-key-validation`	`false`	Relax the validation of the verification keys, setting this property to `true` will allow public RSA keys with the length less than 2048 bit.
`smallrye.jwt.verify.certificate-thumbprint`	`false`	If this property is enabled then a signed token must contain either 'x5t' or 'x5t#S256' X509Certificate thumbprint headers. Verification keys can only be in JWK or PEM Certificate key formats in this case. JWK keys must have a 'x5c' (Base64-encoded X509Certificate) property set.
`smallrye.jwt.token.header`	`Authorization`	Set this property if another header such as `Cookie` is used to pass the token.
`smallrye.jwt.token.cookie`	`none`	Name of the cookie containing a token. This property will be effective only if `smallrye.jwt.token.header` is set to `Cookie`.
`smallrye.jwt.always-check-authorization`	`false`	Set this property to true for Authorization header be checked even if the smallrye.jwt.token.header is set to Cookie but no cookie with a smallrye.jwt.token.cookie name exists.
`smallrye.jwt.token.schemes`	`Bearer`	Comma-separated list containing an alternative single or multiple schemes, for example, `DPoP`.
`smallrye.jwt.token.kid`	`none`	Key identifier. If it is set then the verification JWK key as well every JWT token must have a matching `kid` header.
`smallrye.jwt.time-to-live`	`none`	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.

Property Name	Default	Description
smallrye.jwt.require.named-principal	false	If an application relies on java.security.Principal returning a name then a token must have a upn or preferred_username or sub 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 Principal name.
smallrye.jwt.path.sub	none	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: realms/subject. 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.
smallrye.jwt.claims.sub	none	This property can be used to set a default sub claim value when the current token has no standard or custom sub claim available. Effectively this property can be used to customize java.security.Principal name if no upn or preferred_username or sub claim is set.
smallrye.jwt.path.groups	none	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: realm/groups. 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.
smallrye.jwt.groups-separator	' '	Separator for splitting a string which may contain multiple group values. It will only be used if the smallrye.jwt.path.groups property points to a custom claim whose value is a string. The default value is a single space because a standard OAuth2 scope claim may contain a space separated sequence.
smallrye.jwt.claims.groups	none	This property can be used to set a default groups claim value when the current token has no standard or custom groups claim available.

Property Name	Default	Description
`smallrye.jwt.jwks.refresh-interval`	`60`	JWK cache refresh interval in minutes. It will be ignored unless the `mp.jwt.verify.publickey.location` points to the HTTPS URL based JWK set and no HTTP `Cache-Control` response header with a positive `max-age` parameter value is returned from a JWK HTTPS endpoint.
`smallrye.jwt.jwks.forced-refresh-interval`	`30`	Forced JWK cache refresh interval in minutes which is used to restrict the frequency of the forced refresh attempts which may happen when the token verification fails due to the cache having no JWK key with a kid property matching the current token's kid header. It will be ignored unless the mp.jwt.verify.publickey.location points to the HTTPS URL based JWK set.
`smallrye.jwt.expiration.grace`	`60`	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.
`smallrye.jwt.verify.aud`	`none`	Comma separated list of the audiences that a token `aud` claim may contain.
`smallrye.jwt.required.claims`	`none`	Comma separated list of the claims that a token must contain.
`smallrye.jwt.decrypt.key-location`	`none`	Config property allows for an external or internal location of Private Decryption Key to be specified.
`smallrye.jwt.decrypt.algorithm`	`RSA_OAEP`	Decryption algorithm.
`smallrye.jwt.token.decryption.kid`	`none`	Decryption Key identifier. If it is set then the decryption JWK key as well every JWT token must have a matching `kid` header.

Create JsonWebToken with JWTParser

If the JWT token can not be injected, for example, if it is embedded in the service request payload or the service endpoint acquires it out of band, then one can use `JWTParser`:

```
import org.eclipse.microprofile.jwt.JsonWebToken;
import io.smallrye.jwt.auth.principal.JWTParser;
...
@Inject JWTParser parser;

String token = getTokenFromOidcServer();

// Parse and verify the token
JsonWebToken jwt = parser.parse(token);
```

You can also use it to customize the way the token is verified or decrypted. For example, one can supply a local `SecretKey`:

```
import javax.crypto.SecretKey;
import javax.ws.rs.GET;
import javax.ws.rs.core.NewCookie;
import javax.ws.rs.core.Response;
import org.eclipse.microprofile.jwt.JsonWebToken;
import io.smallrye.jwt.auth.principal.JWTParser;
import io.smallrye.jwt.build.Jwt;

@Path("/secured")
public class SecuredResource {
 @Inject JWTParser parser;
 private String secret = "AyM1SysPpbyDfgZld3umj1qzKObwVMko";

 @GET
 @Produces("text/plain")
 public Response getUserName(@CookieParam("jwt") String jwtCookie)
{
    Response response = null;
    if (jwtCookie == null) {
        // Create a JWT token signed using the 'HS256' algorithm
        String newJwtCookie =
Jwt.upn("Alice").signWithSecret(secret);
        // or create a JWT token encrypted using the 'A256KW'
algorithm
        // Jwt.upn("alice").encryptWithSecret(secret);
        return Response.ok("Alice").cookie(new NewCookie("jwt",
newJwtCookie)).build();
    else {
        // All mp.jwt and smallrye.jwt properties are still
effective, only the verification key is customized.
        JsonWebToken jwt = parser.verify(jwtCookie, secret);
        // or jwt = parser.decrypt(jwtCookie, secret);
        return Response.ok(jwt.getName()).build();
    }
}
```

Token Decryption

If your application needs to accept the tokens with the encrypted claims or with the encrypted inner signed claims then all you have to do is to set `smallrye.jwt.decrypt.key-location` pointing to the decryption key.

If this is the only key property which is set then the incoming token is expected to contain the encrypted claims only. If either `mp.jwt.verify.publickey` or `mp.jwt.verify.publickey.location` verification properties are also set then the incoming token is expected to contain the encrypted inner-signed token.

See Generate JWT tokens with SmallRye JWT and learn how to generate the encrypted or inner-signed and then encrypted tokens fast.

How to check the errors in the logs

Set `quarkus.log.category."io.quarkus.smallrye.jwt.runtime.auth.MpJwtValidator".level=TRACE` to see more details about the token verification or decryption errors.

Custom Factories

`io.smallrye.jwt.auth.principal.DefaultJWTCallerPrincipalFactory` is used by default to parse and verify JWT tokens and convert them to `JsonWebToken` principals. It uses `MP JWT` and `smallrye-jwt` properties listed in the `Configuration` section to verify and customize JWT tokens.

If you need to provide your own factory, for example, to avoid verifying the tokens again which have already been verified by the firewall, then you can either use a `ServiceLoader` mechanism by providing a `META-INF/services/io.smallrye.jwt.auth.principal.JWTCallerPrincipalFactory` resource or simply have an `Alternative` CDI bean implementation like this one:

```
import java.nio.charset.StandardCharsets;
import java.util.Base64;
import javax.annotation.Priority;
import javax.enterprise.context.ApplicationScoped;
import javax.enterprise.inject.Alternative;
import org.jose4j.jwt.JwtClaims;
import org.jose4j.jwt.consumer.InvalidJwtException;
import io.smallrye.jwt.auth.principal.DefaultJWTCallerPrincipal;
import io.smallrye.jwt.auth.principal.JWTAuthContextInfo;
import io.smallrye.jwt.auth.principal.JWTCallerPrincipal;
import io.smallrye.jwt.auth.principal.JWTCallerPrincipalFactory;
import io.smallrye.jwt.auth.principal.ParseException;

@ApplicationScoped
@Alternative
@Priority(1)
public class TestJWTCallerPrincipalFactory extends
JWTCallerPrincipalFactory {

    @Override
    public JWTCallerPrincipal parse(String token,
JWTAuthContextInfo authContextInfo) throws ParseException {
        try {
            // Token has already been verified, parse the token
claims only
            String json = new
String(Base64.getUrlDecoder().decode(token.split("\\.")[1]),
StandardCharsets.UTF_8);
            return new
DefaultJWTCallerPrincipal(JwtClaims.parse(json));
        } catch (InvalidJwtException ex) {
            throw new ParseException(ex.getMessage());
        }
    }
}
```

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 javax.json.Json;
import javax.json.JsonObject;
import io.smallrye.jwt.build.Jwt;
import io.smallrye.jwt.build.JwtClaimsBuilder;
import org.eclipse.microprofile.jwt.JsonWebToken;
...
// Create an empty builder and add some claims
JwtClaimsBuilder builder1 = Jwt.claims();
builder1.claim("customClaim", "custom-
value").issuer("https://issuer.org");
// Or start typing the claims immediately:
// JwtClaimsBuilder builder1 = Jwt.upn("Alice");

// 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"));

// Builder created from JsonObject
JsonObject userName = Json.createObjectBuilder().add("username",
"Alice").build();
JsonObject userAddress = Json.createObjectBuilder().add("city",
"someCity").add("street", "someStreet").build();
JsonObject json = Json.createObjectBuilder(userName).add("address",
userAddress).build();
JwtClaimsBuilder builder4 = Jwt.claims(json);

// Builder created from JsonWebToken
@Inject JsonWebToken token;
JwtClaimsBuilder builder5 = Jwt.claims(token);
```

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 (it can be customized with the `smallrye.jwt.new-token.lifespan` property) and `jti` (unique token identifier) claims if they have not already been set. One can also configure `smallrye.jwt.new-token.issuer` property and skip setting the issuer directly with the builder API.

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().keyId("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();
```

Fast JWT Generation

If `smallrye.jwt.sign.key-location` or/and `smallrye.jwt.encrypt.key-location` properties are set then one can secure the existing claims (resources, maps, JsonObjects) with a single call:

```
// More compact than Jwt.claims("/claims.json").sign();
Jwt.sign("/claims.json");

// More compact than Jwt.claims("/claims.json").jwe().encrypt();
Jwt.encrypt("/claims.json");

// More compact than
Jwt.claims("/claims.json").innerSign().encrypt();
Jwt.signAndEncrypt("/claims.json");
```

As mentioned above, `iat`, `exp`, `jti` and `iss` claims will be added if needed.

SmallRye JWT Builder configuration

Smallrye JWT supports the following properties which can be used to customize the way claims are signed and encrypted:

Property Name	Default	Description
`smallrye.jwt.sign.key-location`	`none`	Location of a private key which will be used to sign the claims when either a no-argument `sign()` or `innerSign()` method is called.
`smallrye.jwt.encrypt.key-location`	`none`	Location of a public key which will be used to encrypt the claims or inner JWT when a no-argument `encrypt()` method is called.

Property Name	Default	Description
`smallrye.jwt.new-token.lifespan`	`300`	Token lifespan in seconds which will be used to calculate an exp (expiry) claim value if this claim has not already been set.
`smallrye.jwt.new-token.issuer`	`none`	Token issuer which can be used to set an iss (issuer) claim value if this claim has not already been set.

References

- MP JWT 1.1.1 HTML
- MP JWT 1.1.1 PDF
- SmallRye JWT
- JSON Web Token
- JSON Web Signature
- JSON Web Encryption
- JSON Web Algorithms
- Quarkus Security