Sécurité : Approfondissement

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-11-07

Table des matières

1 OAuth2 et OpenID Connect (OIDC) 2
1.1 OAuth 2.0 : la délégation d'accès 2
1.2 OpenID Connect : la couche identité 2
1.3 Quarkus et l'extension `quarkus-oidc` 2
2 Intégration Keycloak 2
2.1 Mode développement 3
2.2 Mode production 3
3 Approfondissement JWT 3
3.1 Structure et claims 3
3.2 Vérification du token côté serveur 4
3.3 Le resource server 4
4 Autorisation avancée 5
4.1 Sécurité au niveau méthode 5
4.2 Security policies (configuration) 5
4.3 Politiques personnalisées 6
5 Pour aller plus loin 7

> → **Leçon de base :** `05_java_Security_L_Introduction.qmd` — pose les notions fondamentales de la sécurité applicative : authentification, rôles et protection d'endpoints HTTP. → **TP associé :** `05_java_Security_P_securing-quarkus.qmd` — exercice pratique de sécurisation d'une application Quarkus avec un fournisseur d'identité externe.

Cette leçon d'approfondissement reprend le socle de la leçon de base et pousse plus loin les mécanismes de sécurité : **OAuth2 / OpenID Connect (OIDC)**, l'intégration **Keycloak** (dev et prod), l'anatomie et la vérification des **JWT**, et l'autorisation avancée (sécurité par méthode et security policies). Elle suppose que vous avez lu la le-

çon de base et manipulé les primitives `@RolesAllowed` / `quarkus.http.auth.permission.*` de l'introduction.

1 OAuth2 et OpenID Connect (OIDC)

1.1 OAuth 2.0 : la délégation d'accès

OAuth 2.0 est un protocole de **délégation d'accès** : un client obtient, au nom d'un utilisateur ou d'une application, un **jeton d'accès** (access token) pour atteindre une ressource protégée. Il ne définit ni la méthode d'authentification de l'utilisateur, ni le format du jeton. Les rôles clés : la **ressource** (l'API protégée, resource server), le **client** (l'application consommatrice) et le **serveur d'autorisation** (émet les jetons, ex. Keycloak). Les grants usuels : authorization code (flux navigateur), client credentials (application-à-application) et refresh token (renouvellement).

1.2 OpenID Connect : la couche identité

OpenID Connect (OIDC) ajoute à OAuth 2.0 une **couche d'identité** : il standardise un **id_token** (un JWT signé) qui décrit l'utilisateur authentifié. Concrètement, OIDC permet à l'application de savoir **qui** s'est connecté (identité) en plus de savoir **ce** qu'il est autorisé à faire (accès). C'est la combinaison des deux qui fait OIDC, et c'est ce que Quarkus consomme via l'extension `quarkus-oidc`.

1.3 Quarkus et l'extension `quarkus-oidc`

L'extension `quarkus-oidc` fait de Quarkus un **resource server** OIDC : elle vérifie les jetons Bearer reçus sur l'en-tête `Authorization` et les traduit en identité injectable. Le minimum pour la mettre en route :

```
# quarkus-oidc : le serveur d'identité
quarkus.oidc.auth-server-url=http://localhost:8180/realms/quarkus
quarkus.oidc.client-id=quarkus
# service = resource server pur (jeton Bearer uniquement)
quarkus.oidc.application-type=service
```

`application-type` distingue deux postures : `service` (resource server, jeton Bearer) et `web-app` (ajoute la redirection navigateur et la connexion OIDC avec session).

2 Intégration Keycloak

Keycloak est le serveur d'autorisation OIDC de référence. On l'intègre différemment selon l'environnement.

2.1 Mode développement

Quarkus embarque des commandes dédiées pour ne pas avoir à installer Keycloak à la main. En mode développement, on démarre un Keycloak local puis on configure l'application pour y pointer :

```
# démarre un Keycloak local dédié au dev (realm `quarkus`, client `quarkus`)
quarkus keycloak dev

# génère des scripts de création du realm (clients, rôles) — utile au 1er boot
quarkus keycloak generate-dev-scripts
```

Le realm par défaut fourni par `quarkus keycloak dev` est `quarkus` (client `quarkus`), ce qui correspond à la configuration `quarkus.oidc.auth-server-url` / `client-id` montrée plus haut. En dev, on peut aussi se passer d'un client secret pour un client public (browser).

2.2 Mode production

En production, l'application pointe vers le Keycloak d'entreprise (URL stable, TLS), et le secret du client n'est **jamais** versionné : il est injecté via la configuration externalisée (vault, variables d'environnement, CI/CD). On utilise typiquement le grant client credentials pour un service backend :

```
quarkus.oidc.auth-server-url=https://auth.example.com/realms/quarkus
quarkus.oidc.client-id=quarkus-service
quarkus.oidc.application-type=service
# secret injecté à l'exécution, jamais en dépôt
quarkus.oidc.client-secret=${QUARKUS_OIDC_CLIENT_SECRET}
```

Pour administrer le realm (créer des clients, des rôles, des groupes) depuis l'application, Quarkus propose l'admin client `quarkus-keycloak-admin-client`, piloté par la famille de propriétés `quarkus.keycloak.admin-client.*` (`auth-server-url`, `realm`, `client-id`, `username`, `password`).

3 Approfondissement JWT

3.1 Structure et claims

> **Rappel** — voir l'Introduction, section « JWT avec SmallRye » : un jeton est `header.payload.signature`, chaque partie encodée en Base64URL. Le payload porte ici des claims standard et des claims propres au fournisseur OIDC :

```
{
  "sub": "12345678-1234-1234-1234-123456789abc",
  "iss": "http://localhost:8180/realms/quarkus",
  "aud": "account",
  "exp": 1736937600,
  "iat": 1736934000,
  "azp": "quarkus",
  "preferred_username": "alice",
  "email": "alice@example.com",
  "name": "Alice Martin",
  "groups": ["user", "admin"]
}
```

Lecture des claims principaux :

- `sub` — identifiant unique de l'utilisateur (subject).
- `iss` — émetteur du jeton (URL du realm).
- `aud` — audience à laquelle le jeton est destiné.
- `exp` / `iat` — expiration et date d'émission (en secondes Unix).
- `azp` — client ayant demandé le jeton (authorized party).
- `preferred_username`, `email`, `name` — claims d'identité OIDC.
- `groups` — claim custom Keycloak contenant les groupes/rôles de l'utilisateur.

3.2 Vérification du token côté serveur

Quand Quarkus reçoit un jeton Bearer, l'extension `quarkus-oidc` le **vérifie** avant de l'accepter : elle valide la **signature** (contre le jeu de clés publiques JWKS de l'émetteur), puis les **claims** d'identité attendus. Ces attentes se configurent explicitement :

```
# issuer et audience attendus — tout écart fait échouer la vérification
quarkus.oidc.issuer=http://localhost:8180/realms/quarkus
quarkus.oidc.audience=quarkus-service
# claim d'où sont tirés les rôles (défaut Keycloak : `groups`)
quarkus.oidc.roles.role-claim-path=groups
```

Un jeton dont l'`iss` ne correspond pas, dont l'`aud` n'inclut pas l'audience attendue, ou dont l'`exp` est dépassé, est **rejeté** (401). C'est ce qui distingue une vérification sérieuse d'un simple « le header est présent ».

3.3 Le resource server

Côté application, l'identité vérifiée s'injecte. Le type MicroProfile JWT `org.eclipse.microprofile.jwt.JsonWebToken` donne accès aux claims ; les annotations `jakarta.annotation.security.*` appliquent l'autorisation :

```java
import jakarta.annotation.security.RolesAllowed;
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.core.Response;
import org.eclipse.microprofile.jwt.JsonWebToken;

@Path("/api/profile")
public class ProfileResource {

    @Inject
    JsonWebToken jwt;

    @GET
    @RolesAllowed("user")
    public Response me() {
        String username = jwt.getClaim("preferred_username");
        return Response.ok("Bonjour " + username).build();
    }
}
```

4 Autorisation avancée

4.1 Sécurité au niveau méthode

> **Rappel** — voir l'Introduction, section « Sécurité par méthode » : les annotations Jakarta `@RolesAllowed`, `@PermitAll`, `@DenyAll` restreignent chaque méthode JAX-RS.

Pour qu'un endpoint **sans** annotation devienne refusé par défaut (et non implicitement public), on active :

```
quarkus.security.jaxrs.deny-unannotated-endpoints=true
```

4.2 Security policies (configuration)

Les security policies associent des **chemins** (expressions régulières) à une **politique** nommée. C'est le levier déclaratif pour protéger des groupes d'endpoints sans toucher au code :

```
# le jeton OIDC est requis sur tout /api/*
quarkus.http.auth.permission.api.paths=/api/*
```

```
quarkus.http.auth.permission.api.policy=oidc

# /api/admin/* exige en plus le rôle `admin`
quarkus.http.auth.permission.admin.paths=/api/admin/*
quarkus.http.auth.permission.admin.policy=roles:admin
```

Les politiques intégrées usuelles : `http-basic`, `oidc`, `role-group`, ainsi que les préfixes `role:` / `roles:` (p. ex. `roles:admin`) pour l'autorisation par rôle. La politique `role:` fonctionne aussi bien pour les utilisateurs de base (Basic auth) que pour les utilisateurs OIDC — dans le second cas, les rôles sont dérivés du claim du JWT via `quarkus.oidc.roles.role-claim-path`. Un chemin qui ne correspond à aucune politique reste, par défaut, accessible — d'où l'intérêt de coupler cela avec `deny-unannotated-endpoints` pour une posture fail-safe.

4.3 Politiques personnalisées

Pour une règle d'autorisation qui dépasse « un rôle parmi N », on enregistre une politique **custom** : un bean CDI implémentant `io.quarkus.security.identity.SecurityPolicy`, référencé par son nom dans la configuration :

```
import io.quarkus.security.identity.SecurityIdentity;
import io.quarkus.security.identity.SecurityPolicy;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.inject.Named;

@ApplicationScoped
@Named("admin-only")
public class AdminOnlyPolicy implements SecurityPolicy {

    @Override
    public boolean check(SecurityIdentity identity) {
        return identity.hasRole("admin");
    }
}
```

```
quarkus.http.auth.permission.admin.paths=/api/admin/*
quarkus.http.auth.permission.admin.policy=admin-only
```

La méthode `check` reçoit l'`SecurityIdentity` vérifiée et décide de l'autorisation ; elle peut combiner plusieurs conditions (rôle, claim, origine) que les politiques intégrées ne couvrent pas seules.

5 Pour aller plus loin

Pour relier les parties : **OAuth2** délègue l'accès, **OIDC** y ajoute l'identité (`id_token`), **Keycloak** joue le serveur d'autorisation (dev : `quarkus keycloak dev`, prod : URL stable + secret externalisé), le **JWT** transporte les claims vérifiés (signature, `iss`, `aud`, `exp`), et l'autorisation combine annotations Jakarta par méthode, security policies par chemin et politiques custom pour les règles composites. La leçon de base (`05_java_Security_L_Introduction.qmd`) fournit les notions de départ ; le TP (`05_java_Security_P_securing-quarkus.qmd`) permet de mettre ces mécanismes en pratique.