Sécurité : authentification et autorisation sous Quarkus

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

2026-11-07

Table des matières

1 Objectifs 1
2 Introduction : pourquoi sécuriser une API ? 2
3 Authentification vs autorisation 2
4 Le modèle de sécurité de Quarkus 3
4.1 Permissions HTTP 3
5 Authentification de base (Basic) 4
6 Sécurité par méthode 5
7 JWT avec SmallRye 7
7.1 Structure d'un JWT 7
7.2 Validation par Quarkus 7
7.3 Consommer les claims 8
8 Test en mode développement 9
9 Bonnes pratiques 10
10 Ressources et liens 10

1 Objectifs

À la fin de cette leçon, vous serez capable de :

- **Distinguer l'authentification de l'autorisation** et situer chaque mécanisme dans le cycle de vie d'une requête HTTP.
- **Décrire le modèle de sécurité de Quarkus** : mécanismes d'authentification, identité de sécurité et permissions HTTP.
- **Configurer l'authentification de base** (`quarkus.security.users`) pour un usage local et de test.
- **Protéger des méthodes** de ressource grâce à la sécurité par méthode (`@RolesAllowed`, `@PermitAll`, `@DenyAll`).

- **Expliquer la structure et la validation d'un JWT** (JSON Web Token) avec SmallRye.
- **Tester la sécurité en mode développement** (`quarkus:dev`) sans avoir à provisionner de clés.

Comment suivre cette leçon : ce document est le **cœur** conceptuel. Il s'accompagne d'un TP pratique (`05_java_Security_P_securing-quarkus.qmd`) et d'une partie avancée optionnelle (`05_java_Security_L_advanced.qmd`) pour les approfondissements (OAuth2 / OpenID Connect, clés de production).

2 Introduction : pourquoi sécuriser une API ?

Une API REST exposée sur le réseau ne peut pas faire confiance à n'importe quel appelant. Avant de répondre, le serveur doit répondre à deux questions fondamentales, dans cet ordre :

1. **Qui êtes-vous ?** — vérifier l'identité de l'appelant.
2. **Avez-vous le droit de faire cela ?** — vérifier que cette identité est autorisée à accéder à cette ressource.

Quarkus encapsule ces deux responsabilités dans un **modèle de sécurité** unifié : des mécanismes s'occupent de l'authentification, tandis que des permissions et des annotations s'occupent de l'autorisation. Le reste de cette leçon détaille chaque brique, en partant du plus simple (authentification de base) vers le plus courant en microservices (JWT).

3 Authentification vs autorisation

Ces deux notions sont souvent confondues, alors qu'elles sont distinctes et complémentaires :

Notion	Question posée	Moment	Exemples de mécanismes
Authentification	« Qui êtes-vous ? »	Avant l'accès à la ressource	mot de passe (Basic), JWT, OpenID Connect
Autorisation	« Avez-vous le droit ? »	Une fois l'identité établie	rôles (`@RolesAllowed`), permissions par chemin

Règle de lecture : l'authentification produit une **identité** ; l'autorisation **décide** à partir de cette identité. On ne peut pas autoriser un appelant qu'on n'a pas d'abord authentifié.

4 Le modèle de sécurité de Quarkus

Quarkus sépare clairement les rôles dans son modèle de sécurité :

- **Mécanismes d'authentification** : Basic, Form login, Bearer token (JWT), OAuth2 / OpenID Connect. Un mécanisme lit les identifiants (en-tête, corps de formulaire, cookie) et les transforme en identité.
- `SecurityIdentity` : l'objet qui représente l'utilisateur authentifié pour une requête — son nom (subject) et l'ensemble de ses **rôles**. C'est lui que les autorisations consultent.
- **Permissions HTTP** : des règles déclaratives, par chemin, qui décident quelle identité (et quels rôles) est requise pour atteindre un endpoint.

Le flux type d'une requête est le suivant :

1. La requête HTTP arrive sur le serveur Vert.x.
2. Le security handler tente l'authentification via le mécanisme configuré (par exemple l'en-tête `Authorization`).
3. Si l'identité est valide, une `SecurityIdentity` est associée au contexte de la requête.
4. Les **permissions HTTP** et la **sécurité par méthode** vérifient ensuite que cette identité est autorisée.
5. En cas d'échec : `401 Unauthorized` (non authentifié) ou `403 Forbidden` (authentifié mais non autorisé).

Comportement par défaut : lorsque la sécurité est en place, Quarkus protège l'ensemble des endpoints HTTP. Une requête sans identité valide reçoit une réponse `401 Unauthorized`. On assouplit ce comportement via des permissions HTTP (par exemple pour laisser un endpoint public).

4.1 Permissions HTTP

Les permissions HTTP se décrivent par chemin, avec une politique :

```
# Exiger un utilisateur authentifié sur /api/*
quarkus.http.auth.permission.api.paths=/api/*
quarkus.http.auth.permission.api.policy=authenticated

# Exiger le rôle "admin" sur /admin/*
quarkus.http.auth.permission.admin.paths=/admin/*
```

```
quarkus.http.auth.permission.admin.policy=role:admin

# Rendre un endpoint public (aucune identité requise)
quarkus.http.auth.permission.public.paths=/public/*
quarkus.http.auth.permission.public.policy=permit
```

Les politiques courantes : `permit` (tout le monde), `authenticated` (toute identité valide), `deny` (personne), et `role:<nom>` (un rôle précis).

5 Authentification de base (Basic)

L'authentification **Basic** transmet un identifiant et un mot de passe dans l'en-tête `Authorization: Basic`, encodés en base64. C'est le mécanisme le plus simple, adapté au **développement** et aux tests — jamais à la production sans TLS, car le couple identifiant/mot de passe est alors transporté en clair.

Pour l'activer, on configure le mécanisme `basic` et, pour le test, le realm embarqué de Quarkus :

```
# Activer le mécanisme Basic
quarkus.http.auth.basic=true

# Désactiver l'authentification proactive : le navigateur
# affiche la boîte de dialogue au premier accès protégé
quarkus.http.auth.proactive=false

# Realm embarqué (usage local / test uniquement)
quarkus.security.users.embedded.enabled=true
quarkus.security.users.embedded.plain-text=true
quarkus.security.users.embedded.users.alice=alice
quarkus.security.users.embedded.users.bob=bob
quarkus.security.users.embedded.roles.alice=admin
quarkus.security.users.embedded.roles.bob=user
```

Cet exemple déclare deux utilisateurs : `alice` (mot de passe `alice`, rôle `admin`) et `bob` (mot de passe `bob`, rôle `user`). Le realm embarqué est fourni par l'extension `quarkus-elytron-security-properties-file` et ne doit **jamais** être utilisé en production : il existe pour faciliter le développement et les tests.

On peut alors tester directement avec `curl` :

```
# alice est authentifiée et porte le rôle admin
curl -u alice:alice http://localhost:8080/api/whoami
```

```
# bob est authentifié et porte le rôle user
curl -u bob:bob http://localhost:8080/api/whoami

# sans identifiants : 401 Unauthorized
curl -i http://localhost:8080/api/whoami
```

À retenir : `quarkus.http.auth.basic` active le mécanisme ; `quarkus.security.users.embedded.*` fournit la source d'identités de test. En production, on remplace le realm embarqué par un vrai annuaire (base de données, LDAP, fournisseur d'identité).

6 Sécurité par méthode

Plutôt que de protéger par chemin, on peut annoter directement les **méthodes** de ressource avec les annotations Jakarta de sécurité. C'est le moyen le plus lisible d'exprimer une autorisation fine.

Les trois annotations principales :

- `@RolesAllowed("rôle")` : la méthode n'est accessible qu'aux identités possédant au moins un des rôles listés.
- `@PermitAll` : la méthode est accessible sans contrainte de rôle (l'authentification reste toutefois appliquée par défaut).
- `@DenyAll` : la méthode est inaccessible à tous.
- `@DeclareRoles("rôle")` : ne restreint pas l'accès, mais déclare l'ensemble des rôles utilisés par la classe (utile pour la validation et la lisibilité).

```
package org.acme.security;

import jakarta.annotation.security.DenyAll;
import jakarta.annotation.security.PermitAll;
import jakarta.annotation.security.RolesAllowed;
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;
import org.eclipse.microprofile.jwt.JsonWebToken;

@Path("/api")
public class WhoAmIResource {

    @Inject
```

```
    JsonWebToken jwt;

    @GET
    @Path("/whoami")
    @Produces(MediaType.TEXT_PLAIN)
    public String whoAmI() {
        return "connecté en tant que " + jwt.getName();
    }

    @GET
    @Path("/admin")
    @RolesAllowed("admin")
    @Produces(MediaType.TEXT_PLAIN)
    public String admin() {
        return "zone admin";
    }

    @GET
    @Path("/public")
    @PermitAll
    @Produces(MediaType.TEXT_PLAIN)
    public String publicZone() {
        return "zone publique";
    }

    @GET
    @Path("/secret")
    @DenyAll
    @Produces(MediaType.TEXT_PLAIN)
    public String secret() {
        return "inaccessible";
    }
}
```

La sécurité par méthode s'applique **en complément** de l'authentification : l'annotation ne produit pas d'identité, elle filtre l'identité déjà établie. Un appelant non authentifié reçoit `401` ; un appelant authentifié mais sans le rôle exigé reçoit `403`.

> **Chemin vs méthode** : les permissions HTTP (`quarkus.http.auth.permission.*`) sont commodes pour des règles grossières par préfixe ; la sécurité par méthode est préférable dès que l'autorisation dépend de la logique métier. On peut combiner les deux.

7 JWT avec SmallRye

Les **JWT** (JSON Web Tokens) sont le format dominant pour l'authentification stateless en microservices : un token signé est émis par un fournisseur d'identité, puis présenté à chaque requête. Le serveur ne le stocke pas ; il le **vérifie**.

7.1 Structure d'un JWT

Un token est une chaîne de **trois parties** séparées par des points (`.`), chacune encodée en `base64url` :

1. **En-tête** : les métadonnées du token (algorithme de signature, type).
2. **Charge** (payload) : les claims, c'est-à-dire les affirmations sur le sujet (identifiant, rôles, expiration, émetteur).
3. **Signature** : la preuve que le token n'a pas été altéré.

L'en-tête décodé d'un token signé avec RSA + SHA-256 ressemble à :

```
{
  "alg": "RS256",
  "typ": "JWT"
}
```

La charge décodée porte les claims, dont les plus importants :

```
{
  "sub": "alice",
  "upn": "alice",
  "iss": "https://example.com/issuer",
  "groups": ["users", "admin"],
  "exp": 1798761600
}
```

Les claims clés :

- `iss` (issuer) : l'émetteur du token — Quarkus peut l'exiger pour relier le token à un fournisseur connu.
- `sub` (subject) : l'identifiant unique du sujet.
- `upn` (user principal name) : le claim préféré pour nommer le `Principal` dans l'API de sécurité.
- `groups` : les groupes / rôles du porteur — c'est ce que `@RolesAllowed` consulte.
- `exp` (expiration) : la date de péremption — un token expiré est rejeté.

7.2 Validation par Quarkus

Quand un token est présenté (en-tête `Authorization: Bearer <token>`), Quarkus le valide en plusieurs étapes :

1. **Extraction** du token depuis l'en-tête `Authorization` (mécanisme bearer).
2. **Vérification de la signature** avec la **clé publique** configurée. Si la signature ne correspond pas, le token est rejeté.
3. **Vérification des claims** : `exp` (non expiré), `iss` (émetteur attendu, si configuré), et éventuellement l'audience (`aud`).
4. **Construction** d'une `SecurityIdentity` / `JsonWebToken` à partir des claims : `sub` devient le subject, `upn` le nom du principal, `groups` les rôles.

La configuration minimale de validation :

```
# Clé publique utilisée pour vérifier la signature
mp.jwt.verify.publickey.location=META-INF/resources/publicKey.pem

# Émetteur attendu (le claim "iss" doit correspondre)
mp.jwt.verify.issuer=https://example.com/issuer

# Inclure la clé dans le binaire natif (GraalVM)
quarkus.native.resources.includes=META-INF/resources/publicKey.pem
```

Privilégiez le standard MicroProfile `mp.jwt.verify.publickey.location` (bloc ci-dessus) : c'est la voie la plus portable pour indiquer la clé publique de vérification.

> **Confiance asymétrique** : seul l'émetteur détient la **clé privée** pour signer ; n'importe quel service qui détient la **clé publique** peut vérifier. C'est ce qui permet à des microservices de vérifier un token sans communiquer avec le fournisseur d'identité.

7.3 Consommer les claims

Une fois le token validé, on injecte l'interface MicroProfile `JsonWebToken` pour lire les claims dans la ressource :

```
package org.acme.security;

import jakarta.annotation.security.RolesAllowed;
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;
import org.eclipse.microprofile.jwt.JsonWebToken;

@Path("/secure")
```

```java
public class SecureResource {

    @Inject
    JsonWebToken jwt;

    @GET
    @Path("/whoami")
    @RolesAllowed("user")
    @Produces(MediaType.TEXT_PLAIN)
    public String whoAmI() {
        return "sujet=" + jwt.getSubject() + " ; groupes=" + jwt.getGroups();
    }
}
```

`jwt.getSubject()` renvoie le claim `sub`, `jwt.getGroups()` renvoie le claim `groups`, et `jwt.getClaim("...")` permet de lire n'importe quel claim personnalisé.

8 Test en mode développement

Tester la sécurité JWT sans provisionner de clés est possible grâce à un confort dédié de Quarkus : **en mode développement (et de test)**, si aucune clé de vérification n'est configurée, l'extension SmallRye JWT **génère automatiquement une paire de clés RSA** (2048 bits). La clé publique configure la validation ; la clé privée est mise à disposition pour signer des tokens de test.

On peut donc exposer temporairement un endpoint qui émet un token signé :

```java
package org.acme.security;

import io.smallrye.jwt.build.Jwt;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

@Path("/token")
public class TokenResource {

    @GET
    public String token() {
        return Jwt.issuer("https://example.com/issuer")
                .upn("alice")
                .groups(java.util.Set.of("users", "admin"))
                .sign();
    }
}
```

Émetteur par défaut : en mode développement, si `mp.jwt.verify.issuer` n'est pas défini, SmallRye JWT applique l'émetteur par défaut `https://quarkus.io/issuer`. On peut le neutraliser en le positionnant à `NONE`.

On teste ensuite la boucle complète avec `curl` :

```
# 1. Récupérer un token signé (dev mode, clé auto-générée)
TOKEN=$(curl -s http://localhost:8080/token)

# 2. L'utiliser sur une ressource protégée
curl -H "Authorization: Bearer $TOKEN" http://localhost:8080/secure/whoami

# 3. Sans token : 401 Unauthorized
curl -i http://localhost:8080/secure/whoami
```

À retenir : la génération automatique de clés n'est active qu'en **dev** et **test**. En production, on fournit toujours une vraie clé publique (et l'émetteur signe avec la clé privée correspondante).

Pour un test automatisé, Quarkus fournit l'annotation `@MockJwt` (package `io.quarkus.test.security.mock.token`) : appliquée à une méthode de test `@QuarkusTest`, elle injecte une identité JWT portant les groupes indiqués, sans avoir à construire ni signer un token. C'est le complément naturel des tests manuels `curl` du mode dev.

9 Bonnes pratiques

- **Toujours TLS** en production : Basic auth et JWT circulent en clair autrement que par le chiffrement TLS.
- **Préférer le stateless** (JWT / OIDC) aux sessions pour les microservices ; la Basic auth reste un outil de développement.
- **Ne jamais utiliser le realm embarqué** (`quarkus.security.users.embedded`) en production.
- **Dénier par défaut** : sécuriser par défaut et ouvrir explicitement (`permit` / `@PermitAll`) les seuls endpoints publics.
- **Distinguer `401` et `403`** : non authentifié vs non autorisé — utile pour le débogage et pour le client.
- **Faire tourner les clés** et émettre des tokens à **durée de vie courte** (`exp` proche).

10 Ressources et liens

- Guide Quarkus Security : https://quarkus.io/guides/security
- Guide Quarkus Basic Authentication : https://quarkus.io/guides/security-basic-authentication-howto

- Guide Quarkus JWT (SmallRye) : https://quarkus.io/guides/security-jwt
- Spécification MicroProfile JWT : https://microprofile.io
- SmallRye JWT : https://smallrye.io