TP : Sécuriser une application 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 Consigne 2
3 Étapes 2
3.1 Étape 1 — Authentification de base 2
3.2 Étape 2 — Sécurité par méthode 2
3.3 Étape 3 — JWT SmallRye 4
4 Pièges courants 5
5 Vérification 6
6 Aller plus loin 8

1 Objectifs

Dans ce **travail pratique (TP)**, vous sécurisez pas à pas une petite application REST **Quarkus** : vous activez l'authentification par utilisateurs embarqués, vous protégez une méthode de ressource par rôle, puis vous validez un jeton **JWT** émis via **SmallRye**.

À l'issue de ce TP, vous serez capable de :

- **Activer l'authentification de base** (`quarkus.security.users.embedded`) et présenter des identifiants avec `curl`.
- **Protéger une méthode de ressource** par rôle grâce à l'annotation `@RolesAllowed`.
- **Consulter un endpoint protégé par JWT** en transmettant un en-tête `Authorization: Bearer`.

> **Positionnement** : ce TP prolonge la leçon `05_java_Security_L_Introduction.qmd` — le modèle de sécurité (mécanismes, `SecurityIdentity`, permissions) y est présenté conceptuellement ; ici vous le mettez en œuvre dans une application réelle, du plus simple (Basic) au plus courant en microservices (JWT).

2 Consigne

À partir de votre **devcontainer** (Quarkus + Java), générez un projet avec les extensions REST et JWT, puis lancez-le en **Dev Mode** avant de commencer les étapes.

```
# Générer le projet (groupId fr.ebruno, artifactId secure)
quarkus create app fr.ebruno:secure --extensions rest-jackson,smallrye-jwt
cd secure
# Lancer en Dev Mode : le serveur démarre sur http://localhost:8080
./mvnw quarkus:dev
```

Gardez le serveur ouvert : chaque étape se teste en envoyant une requête `curl` vers `http://localhost:8080`. Le projet généré expose déjà l'endpoint `GET /api/greet/{name}`.

3 Étapes

3.1 Étape 1 — Authentification de base

Activez l'authentification par **utilisateurs embarqués** dans `application.properties`, puis présentez les identifiants `alice`/`admin` pour vérifier que le mécanisme est opérationnel.

```
# src/main/resources/application.properties
# Mécanisme d'authentification par utilisateurs embarqués (usage local / test)
quarkus.security.users.embedded.enabled=true
# Mots de passe stockés en clair (à ne JAMAIS utiliser en production)
quarkus.security.users.embedded.plain-text=true
# Déclaration de l'utilisateur alice (mot de passe : admin)
quarkus.security.users.embedded.users.alice=admin
# Rôles portés par alice (consultés par @RolesAllowed)
quarkus.security.users.embedded.roles.alice=admin
```

```
# Présenter les identifiants Basic ; le mécanisme reconnaît alice (rôle admin)
# Réponse attendue :
#   HTTP/1.1 200 OK
#   content-type: text/plain
#   Hello from alice
curl -i -u alice:admin http://localhost:8080/api/greet/alice
```

3.2 Étape 2 — Sécurité par méthode

Écrivez une ressource qui illustre les trois annotations de sécurité par méthode : un endpoint public (`@PermitAll`), un endpoint réservé au rôle `admin` (`@RolesAllowed`) et un endpoint interdit (`@DenyAll`).

```java
// src/main/java/fr/ebbruno/secure/SecureResource.java
package fr.ebruno.secure;

import jakarta.annotation.security.DenyAll;
import jakarta.annotation.security.PermitAll;
import jakarta.annotation.security.RolesAllowed;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.core.MediaType;

/**
 * Ressource de démonstration : un endpoint public, un endpoint protégé par rôle
 * et un endpoint interdit.
 */
@Path("/api/secure")
public class SecureResource {

    @GET
    @Path("/ping")
    @Produces(MediaType.TEXT_PLAIN)
    @PermitAll
    public String ping() {
        return "ping ok (public)";
    }

    @GET
    @Path("/admin")
    @Produces(MediaType.TEXT_PLAIN)
    @RolesAllowed("admin")
    public String admin() {
        return "Accès réservé au rôle admin : ok";
    }

    @GET
    @Path("/restricted")
    @Produces(MediaType.TEXT_PLAIN)
    @DenyAll
    public String restricted() {
        return "Jamais atteint (refusé)";
    }
}
```

```
# alice porte le rôle admin → 200 ; sans identifiants → 401 Unauthorized
# Réponse attendue (authentifié) :
#   HTTP/1.1 200 OK
#   Accès réservé au rôle admin : ok
curl -i -u alice:admin http://localhost:8080/api/secure/admin
# Réponse attendue (non authentifié) :
#   HTTP/1.1 401 Unauthorized
curl -i http://localhost:8080/api/secure/admin
```

3.3 Étape 3 — JWT SmallRye

Pour valider l'authentification par porteur, il faut d'abord **obtenir un jeton JWT**. En mode développement, l'extension SmallRye JWT génère automatiquement une paire de clés RSA : la clé publique configure la validation, la clé privée signe les jetons. On s'en sert pour exposer temporairement un endpoint d'émission, `TokenResource`, qui émets un jeton signé pour `alice` (porteur du rôle `admin`) — la même ressource que celle présentée dans la leçon `05_java_Security_L_Introduction.qmd`.

```
// src/main/java/fr/ebbruno/secure/TokenResource.java
package fr.ebruno.secure;

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

/**
 * Endpoint d'émission de jetons, réservé au mode développement : signe un
JWT pour
 * alice (groupes users, admin) avec la clé privée auto-générée.
 */
@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();
    }
}
```

En mode développement, si `mp.jwt.verify.issuer` n'est pas défini, SmallRye JWT applique par défaut l'émetteur `https://quarkus.io/issuer` — or le jeton ci-dessus est

signé avec `https://example.com/issuer`. On déclare donc cet émetteur pour que la vérification le reconnaisse (la clé publique de validation, elle, est auto-générée en dev mode) :

```
# src/main/resources/application.properties
# Émetteur attendu : doit correspondre au claim "iss" du jeton émis par /token
mp.jwt.verify.issuer=https://example.com/issuer
```

On peut alors dérouler la boucle complète : émettre le jeton, puis le présenter sur l'endpoint protégé par rôle.

```
# 1. Émettre un jeton signé pour alice (rôle admin) via l'endpoint /token
TOKEN=$(curl -s http://localhost:8080/token)

# 2. Le présenter sur l'endpoint protégé (en-tête Bearer) : accès autorisé
# Réponse attendue (jeton valide, rôle admin) :
#   HTTP/1.1 200 OK
#   Accès réservé au rôle admin : ok
curl -i -H "Authorization: Bearer $TOKEN" http://localhost:8080/api/secure/admin

# 3. Sans jeton : aucune identité présentée → HTTP/1.1 401 Unauthorized
# (un jeton valide mais dépourvu du rôle admin donnerait 403 — identité présente,
#  autorisation manquante ; cf. les pièges courants)
curl -i http://localhost:8080/api/secure/admin
```

4 Pièges courants

Quatre écueils récurrents lors de la mise en place de la sécurité Quarkus. Chacun renvoie à un comportement réel du framework que ce TP vient de manipuler ; les connaître évite la plupart des fausses pistes de débogage — en particulier les confusions entre les codes `401` (non authentifié) et `403` (authentifié mais refusé). Chaque piège est rattaché à une étape du TP : les trois premiers à Basic, rôle et JWT, le dernier au passage en production.

- **Authentification proactive par défaut — `quarkus.http.auth.proactive`.** Par défaut, la valeur de `quarkus.http.auth.proactive` est `true` : Quarkus authentifie de façon proactive toute requête qui **porte des identifiants**, même à destination d'un endpoint public. Des identifiants invalides sont donc rejetés (`401`) même sur un endpoint public, tandis qu'une requête **sans** identifiants passe anonymement, et le `401 Unauthorized` n'apparaît alors qu'au contact d'une ressource protégée. C'est une source classique de surprise : l'absence de `401` sur un endpoint public interrogé sans identifiants ne prouve pas que la sécurité est désactivée. → À retenir :

`401` = aucune identité valide présentée ; `403` = identité présente mais permission (ou rôle) insuffisante.

- **`@RolesAllowed("admin")` renvoie `403`, pas `401`.** Si l'utilisateur est authentifié (Basic ou JWT valide) mais ne porte pas le rôle exigé, Quarkus répond `403 Forbidden` : l'identité est reconnue, c'est l'autorisation qui manque. Le `401` ne s'observe que lorsque la requête n'arrive pas authentifiée du tout — la distinction entre les deux codes guide directement le diagnostic. → `403` sur `/api/secure/admin` alors qu'`alice` est bien authentifiée → vérifier les rôles déclarés dans la configuration ; `401` → vérifier la présentation des identifiants.
- **Le couple de clés de Dev Mode n'est pas embarqué en production.** En Dev Mode, Quarkus génère automatiquement une paire de clés (signature et vérification des JWT) qui n'existe qu'à l'exécution locale. L'UI de développement s'appuie d'ailleurs sur ce couple de clés pour émettre les jetons de test. En mode production, cette clé n'est pas fournie : vous devez déclarer la clé publique de vérification via `mp.jwt.verify.publickey.location`, et la signature du jeton est du ressort de l'émetteur. → Déployer sans cette clé : le démarrage peut réussir, mais toute vérification de jeton échoue dès la première requête `Bearer`.
- **`quarkus.security.users.embedded` est un mécanisme DEV/TEST.** Les utilisateurs embarqués — d'autant avec `plain-text=true` — sont prévus pour le développement local et les tests d'intégration, pas pour porter les comptes réels d'un service exposé. Le TP s'en sert volontairement : il rend chaque étape testable en quelques secondes. En production, branchez un fournisseur de sécurité adapté (LDAP, OIDC, base de données) avec des mots de passe hachés. → Tout code qui s'appuie sur les utilisateurs embarqués doit être neutralisé avant le déploiement.

Règle de poche : `401` = « qui êtes-vous ? » (identité manquante), `403` = « vous êtes bien identifié, mais non » (autorisation manquante) ; et rien de ce que le Dev Mode génère automatiquement ne doit survivre au packaging de production.

5 Vérification

L'application est considérée comme sécurisée lorsque l'endpoint `/api/secure/admin` renvoie `401` sans identité, `200` avec l'utilisateur `alice` (Basic) **et** `200` avec un jeton **JWT** valide portant le rôle `admin`.

Cette section rend chaque critère observable. Elle ajoute d'abord un cas intermédiaire — « authentifié mais sans le rôle », source du `403` décrit dans les pièges courants —, puis récapitule les trois réponses attendues en une seule passe de `curl`, avant de clore sur une liste de contrôle à cocher avant tout déploiement.

Prérequis. Le serveur tourne en Dev Mode sur `http://localhost:8080`, la ressource `SecureResource` est chargée, et l'utilisateur `alice` (rôle `admin`) est déclaré conformément à l'étape 1.

1. Préparer le cas « authentifié sans le rôle ». Ajoutez un second utilisateur embarqué `bob`, dépourvu du rôle `admin`, dans `application.properties` :

```
# src/main/resources/application.properties
# Second utilisateur embarqué : authentifié, mais sans le rôle admin
# (permet d'observer le 403 — identité présente, autorisation manquante)
quarkus.security.users.embedded.users.bob=bob
```

2. Mesurer les trois réponses attendues. Trois commandes `curl` interrogent `/api/secure/admin` ; chacune illustre un cas distinct de l'arbitrage Quarkus :

```
# Émettre le jeton JWT via l'endpoint d'émission /token (étape 3, ressource TokenResource)
TOKEN=$(curl -s http://localhost:8080/token)

# 1. Identifiants Basic valides (alice porte le rôle admin) : accès autorisé → HTTP/1.1 200
curl -i -u alice:admin http://localhost:8080/api/secure/admin
# 2. Utilisateur authentifié mais sans le rôle admin (bob) : refus → HTTP/1.1 403
curl -i -u bob:bob http://localhost:8080/api/secure/admin
# 3. Jeton JWT valide portant le rôle admin : accès autorisé → HTTP/1.1 200
curl -i -H "Authorization: Bearer $TOKEN" http://localhost:8080/api/secure/admin
```

où `$TOKEN` est le jeton émis par l'endpoint d'émission `/token` (étape 3, ressource `TokenResource`) pour un porteur portant le rôle `admin`.

3. Interpréter chaque statut.

- **`200` pour `alice` (Basic)** : l'identité est reconnue **et** le rôle `admin` est porté — l'autorisation est accordée.
- **`403` pour `bob`** : l'identité est reconnue mais le rôle `admin` lui manque — l'autorisation est refusée ; c'est la distinction `401`/`403` qui fonde le diagnostic.
- **`200` pour le JWT** : le même endpoint s'ouvre par l'authentification par porteur, sans aucun changement de code de la ressource.

4. Liste de contrôle. La vérification est complète lorsque :

- ☐ `alice` (Basic) obtient `200` sur `/api/secure/admin` ;
- ☐ `bob` (authentifié, sans rôle) obtient `403` ;
- ☐ un jeton JWT valide portant `admin` obtient `200` ;
- ☐ une requête sans identité obtient `401` (rappel du comportement par défaut).

En Dev Mode, toute modification d'`application.properties` (par exemple l'ajout de `bob`) est prise en charge automatiquement : relancez les `curl` sans redémarrer manuellement le serveur.

Ces quatre points recouvrent l'essentiel de la sécurité de base Quarkus : l'authentification de base, la sécurité par méthode et le JWT. Au-delà — délégation de l'authentification à un annuaire, OAuth2, OIDC — la section finale indique où poursuivre.

6 Aller plus loin

Poursuivre : la leçon `05_java_Security_L_advanced.qmd` approfondit les mécanismes avancés de sécurité Quarkus — **OAuth2**, **OpenID Connect (OIDC)** et l'annuaire **Keycloak** — au-delà de l'authentification embarquée et des JWT manipulés dans ce TP.