TP : Mettre en œuvre MicroProfile sous Quarkus (Health, Metrics, Config, OpenAPI)

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

2026-11-07

Table des matières

1 Objectifs 1
2 Prérequis 2
3 Consigne 2
4 Étapes 3
4.1 1. Créer le projet `microprofile-demo` 3
4.2 2. Externaliser la configuration (MicroProfile Config) 3
4.3 3. Instrumenter le service (MicroProfile Metrics) 4
4.4 4. Exposer la santé (MicroProfile Health) 5
4.5 5. Câbler la ressource et documenter l'API (OpenAPI) 6
4.6 6. Lancer en Dev Mode 7
5 Vérification 7
5.1 Le service répond (Config + Metrics) 7
5.2 La santé est exposée (Health) 7
5.3 Les métriques sont exposées (Metrics) 8
5.4 La documentation est exposée (OpenAPI) 8
5.5 Grille de vérification 9
6 À retenir 9
7 Prolongements 9

1 Objectifs

Dans ce **travail pratique (TP)**, vous saisissez vous-même chaque commande dans le **devcontainer** (Quarkus + Java) et vous enrichissez, pas à pas, un petit service **Quarkus** avec quatre APIs **MicroProfile** : la **configuration** externalisée (`@ConfigProperty`), la **santé** (liveness / readiness), les **métriques** applicatives (`@Counted` / `@Timed`) et la **documentation** OpenAPI. Rien n'est exécuté à votre place : ce sont vous qui lancez `quarkus dev` et qui testez les endpoints avec `curl`.

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

- Créer un projet Quarkus avec les extensions SmallRye appropriées et le lancer en **Dev Mode**
- **Externaliser la configuration** avec MicroProfile Config (`@ConfigProperty` + `application.properties`)
- **Exposer des health checks** de liveness et de readiness et les lire sur `/q/health`
- **Mesurer un service** avec les annotations `@Counted` et `@Timed` et récupérer les métriques sur `/q/metrics`
- **Documenter l'API** avec OpenAPI et consulter la spécification sur `/q/openapi`

> **Positionnement** : ce TP prolonge la leçon MicroProfile pour les microservices (`04_java_MicroProfile_L_Introduction.qmd`), qui développe le cœur conceptuel (Health, Metrics, Config, Fault Tolerance, OpenAPI, Rest Client). La sécurité JWT est traitée dans la leçon 05 (à venir) (`05_java_Security_L_Introduction.qmd`). Ce TP est **autonome** : il crée son propre projet `microprofile-demo` et ne dépend pas des autres leçons.

2 Prérequis

- **Java 21 LTS** et le **Quarkus CLI** doivent être disponibles. Le **devcontainer fourni** avec le cours (dossier `.devcontainer/`) les pré-installe : ouvrez le projet dans ce conteneur avant de commencer.
- Vérifiez le toolchain avec les deux commandes suivantes :

```
java --version
quarkus --version
```

La commande `java --version` doit afficher `21.x` et `quarkus --version` doit renvoyer la version du Quarkus CLI. En l'absence de devcontainer, installez Java 21 LTS et le Quarkus CLI localement avant de poursuivre.

> **Posture** : les commandes `bash` sont à **saisir par vous** dans un terminal ; les fichiers `java` et `properties` sont à **créer ou éditer dans votre éditeur** (le hot reload de Dev Mode recharge les modifications automatiquement).

3 Consigne

À partir d'un projet Quarkus vierge, vous allez construire un service de « salutation » (`greet`) qui :

1. lit un **message** de salutation depuis la **configuration** (avec une valeur par défaut),
2. est **instrumenté** : chaque appel au service est compté et chronométré,

3. expose sa **santé** (liveness et readiness),
4. est **documenté** par une spécification OpenAPI générée automatiquement.

Vous terminerez le TP en **vérifiant** chaque endpoint (`/greet`, `/q/health`, `/q/metrics`, `/q/openapi`) avec `curl` pendant que l'application tourne en Dev Mode.

4 Étapes

4.1 1. Créer le projet `microprofile-demo`

Générez le projet avec le Quarkus CLI. Les extensions `smallrye-health`, `smallrye-metrics` et `smallrye-openapi` apportent les implémentations SmallRye des APIs MicroProfile correspondantes ; `rest-jackson` fournit le runtime REST. La **configuration** (MicroProfile Config) est **intégrée** à Quarkus : aucune extension supplémentaire n'est nécessaire pour utiliser `@ConfigProperty`.

```
quarkus create app fr.ebruno.microprofile:microprofile-demo \
  --extension=rest-jackson,smallrye-health,smallrye-metrics,smallrye-openapi
cd microprofile-demo
```

Le projet généré présente la structure suivante :

```
microprofile-demo/
├── src/main/java/fr/ebruno/microprofile/
│   └── GreetingResource.java        # ressource générée (nous la remplaçons)
├── src/main/resources/
│   └── application.properties       # vide par défaut
├── src/test/java/fr/ebruno/microprofile/
│   └── GreetingResourceTest.java    # test généré (nous le supprimons)
└── pom.xml
```

Lancez l'application en **Dev Mode** et **laissez-la tourner** : le hot reload chargera vos modifications automatiquement.

```
quarkus dev
# hot-reload, accessible à http://localhost:8080
```

> **Astuce** : gardez ce terminal ouvert et ouvrez un **second terminal** dans le même répertoire pour exécuter les `curl` des étapes suivantes.

4.2 2. Externaliser la configuration (MicroProfile Config)

Ajoutez la clé `greeting.message` au fichier `src/main/resources/application.properties` :

```
greeting.message=Bonjour
```

Créez ensuite le bean `GreetingService` (`src/main/java/fr/ebruno/microprofile/GreetingService.java`). Il injecte le message via `@ConfigProperty`, en précisant une **valeur par défaut** (`Hello`) pour le cas où la clé serait absente de la configuration :

```
package fr.ebruno.microprofile;

import jakarta.enterprise.context.ApplicationScoped;
import org.eclipse.microprofile.config.inject.ConfigProperty;

@ApplicationScoped
public class GreetingService {

    @ConfigProperty(name = "greeting.message", defaultValue = "Hello")
    String message;

    public String greet(String name) {
        return message + ", " + name + "!";
    }
}
```

L'application redémarre en Dev Mode. La valeur effective provient désormais du fichier `application.properties` et non plus du code.

4.3 3. Instrumenter le service (MicroProfile Metrics)

Dans le même `GreetingService`, posez les annotations `@Counted` (compteur incrémenté à chaque appel) et `@Timed` (chronomètre de la durée d'exécution) sur la méthode `greet` pour produire des **métriques applicatives** :

```
package fr.ebruno.microprofile;

import jakarta.enterprise.context.ApplicationScoped;
import org.eclipse.microprofile.config.inject.ConfigProperty;
import org.eclipse.microprofile.metrics.annotation.Counted;
import org.eclipse.microprofile.metrics.annotation.Timed;

@ApplicationScoped
public class GreetingService {

    @ConfigProperty(name = "greeting.message", defaultValue = "Hello")
    String message;
```

```
    @Counted(name = "greet_calls_count")
    @Timed(name = "greet_call_time")
    public String greet(String name) {
        return message + ", " + name + "!";
    }
}
```

Les métriques sont désormais produites à chaque appel. Elles seront lues plus loin sur `/q/metrics` (brut) et `/q/metrics/application` (métriques applicatives uniquement).

4.4 4. Exposer la santé (MicroProfile Health)

Créez une **liveness** check (`src/main/java/fr/ebruno/microprofile/LivenessCheck.java`). Elle indique que le processus est **vivant** (l'application tourne), indépendamment de l'état de ses dépendances :

```
package fr.ebruno.microprofile;

import jakarta.enterprise.context.ApplicationScoped;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Liveness;

@Liveness
@ApplicationScoped
public class LivenessCheck implements HealthCheck {
    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.up("application");
    }
}
```

Ajoutez une **readiness** check (`src/main/java/fr/ebruno/microprofile/ReadinessCheck.java`). Elle indique que le service est **prêt** à recevoir du trafic ; c'est ici qu'on verrait un `HealthCheckResponse.down` si une dépendance (base de données, file d'attente) n'était pas disponible :

```
package fr.ebruno.microprofile;

import jakarta.enterprise.context.ApplicationScoped;
import org.eclipse.microprofile.health.HealthCheck;
import org.eclipse.microprofile.health.HealthCheckResponse;
import org.eclipse.microprofile.health.Readiness;
```

```java
@Readiness
@ApplicationScoped
public class ReadinessCheck implements HealthCheck {
    @Override
    public HealthCheckResponse call() {
        return HealthCheckResponse.up("application-ready");
    }
}
```

La distinction liveness / readiness est importante pour l'orchestration : un liveness en échec conduit à **redémarrer** le conteneur, un readiness en échec le retire simplement du pool de trafic sans le tuer.

4.5 5. Câbler la ressource et documenter l'API (OpenAPI)

Remplacez le contenu de `GreetingResource.java` (`src/main/java/fr/ebruno/microprofile/GreetingResource.java`) : la ressource injecte le service et expose `GET /greet`. L'extension `smallrye-openapi` génère **automatiquement** la spécification OpenAPI à partir de ces annotations JAX-RS, sans aucune configuration supplémentaire :

```java
package fr.ebruno.microprofile;

import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.Produces;
import jakarta.ws.rs.QueryParam;
import jakarta.ws.rs.core.MediaType;

@Path("/greet")
public class GreetingResource {

    @Inject
    GreetingService service;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
     public String hello(@QueryParam(value = "name", defaultValue = "world")
String name) {
        return service.greet(name);
    }
}
```

Supprimez le test généré (il référençait l'ancienne ressource) ; il est remplacé par les vérifications `curl` de la section Vérification ci-dessous :

```
rm src/test/java/fr/ebruno/microprofile/GreetingResourceTest.java
```

4.6 6. Lancer en Dev Mode

Au premier terminal, assurez-vous que l'application tourne (relancez-la si nécessaire) :

```
quarkus dev
# accessible à http://localhost:8080
```

Attendez la ligne `Started` dans la console. Toutes les modifications apportées aux étapes précédentes sont rechargées automatiquement ; vous n'avez pas besoin de relancer manuellement l'application.

5 Vérification

Au **second terminal**, testez chaque endpoint avec `curl`. Les URL contenant `?` ou `&` sont entre guillemets afin que le terminal ne les interprète pas.

5.1 Le service répond (Config + Metrics)

```
curl "http://localhost:8080/greet?name=Quarkus"
```

Réponse attendue (le message vient de la configuration `greeting.message=Bonjour`) :

```
Bonjour, Quarkus!
```

Appelez l'endpoint à nouveau pour faire avancer le compteur et le chronomètre :

```
curl "http://localhost:8080/greet?name=MicroProfile"
curl "http://localhost:8080/greet"
```

5.2 La santé est exposée (Health)

```
curl http://localhost:8080/q/health
```

La réponse agrège liveness et readiness. Exemple indicatif (le contenu exact dépend de l'ordre et du format de rendu) :

```
{
  "status": "UP",
```

```
  "liveness": {
    "status": "UP",
    "checks": [
      { "name": "application", "status": "UP" }
    ]
  },
  "readiness": {
    "status": "UP",
    "checks": [
      { "name": "application-ready", "status": "UP" }
    ]
  }
}
```

Vous pouvez aussi interroger les sous-endpoints séparés :

```
curl http://localhost:8080/q/health/live
curl http://localhost:8080/q/health/ready
```

5.3 Les métriques sont exposées (Metrics)

```
curl http://localhost:8080/q/metrics
```

Pour n'afficher que les métriques **applicatives** (celles produites par `@Counted` / `@Timed`), utilisez le sous-endpoint dédié :

```
curl http://localhost:8080/q/metrics/application
```

Exemple indicatif :

```
# HELP greet_calls_count greet_calls_count
# TYPE greet_calls_count counter
greet_calls_count 3.0
# HELP greet_call_time greet_call_time
# TYPE greet_call_time summary
```

Plus l'endpoint `/greet` a été appelé, plus `greet_calls_count` est élevée : c'est la preuve que l'instrumentation fonctionne.

5.4 La documentation est exposée (OpenAPI)

```
curl http://localhost:8080/q/openapi
```

La réponse est la **spécification OpenAPI** au format JSON, générée à partir de `GreetingResource`. Pour une consultation plus lisible, ouvrez l'interface Swagger dans un navigateur :

```
http://localhost:8080/q/swagger-ui
```

5.5 Grille de vérification

Endpoint	API MicroProfile	Résultat attendu
`GET /greet?name=...`	Config (+ Metrics)	`Bonjour, ...!`
`GET /q/health`	Health	JSON `"status": "UP"`
`GET /q/health/live`	Health (liveness)	JSON `"status": "UP"`
`GET /q/health/ready`	Health (readiness)	JSON `"status": "UP"`
`GET /q/metrics`	Metrics	métriques brutes (format Prometheus)
`GET /q/metrics/application`	Metrics (applicatives)	`greet_calls_count`, `greet_call_time`
`GET /q/openapi`	OpenAPI	spécification JSON
`GET /q/swagger-ui`	OpenAPI	interface Swagger (navigateur)

6 À retenir

- **Config** : `@ConfigProperty` lit une clé de `application.properties` (ou l'environnement) avec une valeur par défaut — la configuration est externalisée du code.
- **Health** : `@Liveness` / `@Readiness` + `HealthCheck` alimentent `/q/health`, `/q/health/live` et `/q/health/ready` pour l'orchestrateur.
- **Metrics** : `@Counted` / `@Timed` produisent des métriques applicatives lues sur `/q/metrics` et `/q/metrics/application` (format compatible Prometheus).
- **OpenAPI** : `smallrye-openapi` documente l'API automatiquement sur `/q/openapi` (JSON) et `/q/swagger-ui` (interface).

7 Prolongements

- **Fault Tolerance** : appliquez `@Retry`, `@Timeout` et `@Fallback` à un appel externe ; ces APIs sont développées dans la leçon cœur (`04_java_MicroProfile_L_Introduction.qmd`).
- **Rest Client** : écrivez un client REST typé (`@RegisterRestClient`) pour appeler l'endpoint `/greet` depuis un autre service.

- **Sécurité** : protégez les endpoints avec JWT (MicroProfile JWT / `@RolesAllowed`) ; voir la leçon 05 (à venir) (`05_java_Security_L_Introduction.qmd`).
- **Production** : exposez `/q/metrics` à Prometheus et `/q/health` à l'orchestrateur (Kubernetes) ; consultez la leçon cœur pour les bonnes pratiques d'observabilité.