JEE : Première application Quarkus — TP

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 Étapes 2
3.1 1. Créer le projet `message-app` 2
3.2 2. L'entité `Message` 3
3.3 3. Le repository `MessageRepository` 4
3.4 4. La ressource REST `MessageResource` 5
3.5 5. La configuration `application.properties` 6
4 Exécution en mode dev 6
5 Tests / Vérification 7
5.1 (Optionnel) Exécuter les tests générés 8
6 À retenir 8
6.1 Différences clés avec Jakarta EE traditionnel 9

1 Objectifs

Dans ce **travail pratique (TP)**, vous concevez, développez une première application **Jakarta EE** construite avec **Quarkus** et vérifiez le fonctionnement de bout en bout (endpoints testés avec `curl`). À l'issue de ce TP, vous serez en mesure de manipuler les briques fondamentales du cloud-native Java : une ressource REST, une entité JPA persistée, un repository, l'injection de dépendances et la gestion des transactions.

Ce que vous allez construire

- Une application REST `message-app` de gestion de messages
- Une entité JPA `Message` persistée avec **Hibernate ORM Panache**
- Un repository `MessageRepository` et une ressource REST `MessageResource`
- Une base de données **H2** en mémoire, configurée de façon centralisée

Ce que vous serez capable de faire

- Générer un projet Quarkus à partir de la CLI
- Modéliser une entité JPA avec Panache (sans boilerplate)
- Injecter des dépendances avec **CDI** et déclarer des transactions avec `@Transactional`
- Exécuter l'application en **Dev Mode** et la vérifier avec `curl`

> **Positionnement** : ce TP est le prolongement pratique de la leçon JEE : Introduction aux serveurs d'application et aux plateformes Java d'entreprise (`01_java_jee_L_Introduction.qmd`). Les leçons suivantes approfondissent **CDI** (`02_java_CDI_L_Introduction.qmd`) et les **Transactions** (`03_java_Transactions_L_Introduction.qmd`).

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.

3 Étapes

3.1 1. Créer le projet `message-app`

Générez le projet à l'aide du Quarkus CLI, avec les trois extensions nécessaires : RESTEasy Reactive Jackson, Hibernate ORM Panache et le pilote JDBC H2.

Avec le Quarkus CLI

```
# Avec Quarkus CLI
quarkus create app com.example:message-app \
    --extension=rest-jackson,hibernate-orm-panache,jdbc-h2

cd message-app
```

Ou via code.quarkus.io

Rendez-vous sur https://code.quarkus.io et sélectionnez les extensions :

- RESTEasy Reactive Jackson
- Hibernate ORM with Panache
- JDBC Driver - H2

Le projet généré présente la structure suivante :

```
message-app/
├── src/main/java/
│   └── com/example/
│       ├── entity/
│       │   └── Message.java          # Entité JPA Panache
│       ├── repository/
│       │   └── MessageRepository.java # Repository Panache
│       └── resource/
│           └── MessageResource.java   # Ressource REST
├── src/main/resources/
│   ├── application.properties        # Configuration centralisée
│   └── import.sql                    # Données de test (optionnel)
└── pom.xml
```

> **Point de contrôle** : le répertoire `message-app/` est créé et le fichier `pom.xml` contient les dépendances `rest-jackson`, `hibernate-orm-panache` et `jdbc-h2`.

3.2 2. L'entité `Message`

Créez l'entité JPA `Message` dans `src/main/java/com/example/entity/Message.java`. Elle étend `PanacheEntity`, ce qui fournit l'identifiant `id` et les opérations de base sans boilerplate.

```java
package com.example.entity;

import io.quarkus.hibernate.orm.panache.PanacheEntity;
import jakarta.persistence.*;
import java.time.LocalDateTime;

@Entity
@Table(name = "messages")
public class Message extends PanacheEntity {
    // 'id' hérité de PanacheEntity

    @Column(nullable = false, length = 500)
    public String content;

    @Column(name = "created_at")
```

```
    public LocalDateTime createdAt;

    public Message() {}

    public Message(String content) {
        this.content = content;
        this.createdAt = LocalDateTime.now();
    }
}
```

Note Panache : pas de getters/setters nécessaires ! Les champs publics sont transformés en propriétés.

3.3 3. Le repository `MessageRepository`

Créez le repository dans `src/main/java/com/example/repository/MessageRepository.java`. En implémentant `PanacheRepository<Message>`, vous héritez des opérations courantes (`persist()`, `delete()`, `findById()`, `listAll()`, etc.).

```
package com.example.repository;

import com.example.entity.Message;
import io.quarkus.hibernate.orm.panache.PanacheRepository;
import jakarta.enterprise.context.ApplicationScoped;
import java.util.List;

@ApplicationScoped  // CDI Bean
public class MessageRepository implements PanacheRepository<Message> {

    // Méthodes héritées : persist(), delete(), findById(), listAll()

    public List<Message> findRecent(int limit) {
        return list("SELECT m FROM Message m ORDER BY createdAt DESC", limit);
    }

    public List<Message> searchByContent(String keyword) {
        return list("SELECT m FROM Message m WHERE LOWER(content) LIKE LOWER(?
1)", "%" + keyword + "%");
    }
}
```

3.4 4. La ressource REST `MessageResource`

Créez la ressource dans `src/main/java/com/example/resource/MessageResource.java`. Elle expose les opérations `GET` / `POST` / `DELETE` sur `/messages` et injecte le repository via CDI. Les écritures sont encapsulées dans des transactions avec `@Transactional`.

```
package com.example.resource;

import com.example.entity.Message;
import com.example.repository.MessageRepository;
import jakarta.inject.Inject;
import jakarta.transaction.Transactional;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;
import java.util.List;

@Path("/messages")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class MessageResource {

    @Inject
    MessageRepository messageRepository;

    @GET
    public List<Message> getAllMessages() {
        return messageRepository.listAll();
    }

    @GET
    @Path("/{id}")
    public Response getMessageById(@PathParam("id") Long id) {
        Message message = messageRepository.findById(id).orElse(null);
        if (message == null) {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
        return Response.ok(message).build();
    }

    @POST
    @Transactional
    public Response createMessage(Message message) {
        messageRepository.persist(message);
        return Response.status(Response.Status.CREATED)
```

```
                        .entity(message)
                        .build();
    }

    @DELETE
    @Path("/{id}")
    @Transactional
    public Response deleteMessage(@PathParam("id") Long id) {
        boolean deleted = messageRepository.deleteById(id);
        if (!deleted) {
            return Response.status(Response.Status.NOT_FOUND).build();
        }
        return Response.noContent().build();
    }
}
```

3.5 5. La configuration `application.properties`

Complétez `src/main/resources/application.properties` avec la source de données H2, la génération du schéma et les réglages HTTP / Swagger.

```
# Configuration DataSource H2 (dev)
quarkus.datasource.db-kind=h2
quarkus.datasource.jdbc.url=jdbc:h2:mem:testdb
quarkus.hibernate-orm.database.generation=drop-and-create
quarkus.hibernate-orm.log.sql=true

# Configuration HTTP
quarkus.http.port=8080

# Swagger UI (disponible en dev)
quarkus.swagger-ui.always-include=true
```

Si le port `8080` est déjà occupé, modifiez `quarkus.http.port` et ajustez en conséquence les URLs `curl`, Swagger et Dev UI utilisées plus loin dans ce TP.

> **Point de contrôle** : les trois classes compilent sans erreur et le fichier `application.properties` est complet. La base H2 est en mémoire (`mem:testdb`) et le schéma est recréé à chaque démarrage (`drop-and-create`).

4 Exécution en mode dev

Lancez l'application en **Dev Mode** depuis la racine du projet :

```
# Démarrer en mode dev
quarkus dev
```

Quarkus démarre en moins d'une seconde et met à disposition deux interfaces :

Dev UI — http://localhost:8080/q/dev

- **Config Editor** : modifier `application.properties` en temps réel
- **Datasources** : visualiser les connexions et le schéma DB
- **Hibernate** : voir les entités et les requêtes SQL
- **Swagger** : tester les endpoints REST
- **Continuous Testing** : lancer les tests automatiquement

Swagger UI — http://localhost:8080/q/swagger-ui

- Documentation OpenAPI interactive
- Exécution directe des opérations `GET`, `POST`, `DELETE`
- Vérification des codes de réponse et des corps JSON

> **Point de contrôle** : la console affiche le démarrage réussi et l'URL `http://localhost:8080` répond. Le hot reload est actif : toute modification du code est rechargée à la volée. À noter : la base H2 est en mémoire et le schéma est recréé à chaque (re)démarrage, donc les messages créés disparaissent après toute modification du code qui déclenche un rechargement en Dev Mode.

5 Tests / Vérification

Avec l'application en Dev Mode, testez les endpoints REST à l'aide de `curl`.

1. Créer un message (`POST /messages`)

```
curl -X POST http://localhost:8080/messages \
     -H "Content-Type: application/json" \
     -d '{"content":"Hello Quarkus!"}'
```

Réponse attendue : **`201 Created`**, avec le message créé retourné dans le corps JSON, y compris l'identifiant `id` généré par Panache.

> **Point de contrôle** : le code HTTP est `201` et le JSON renvoyé contient `id`, `content` (valeur `Hello Quarkus!`) et `createdAt`.

2. Récupérer tous les messages (`GET /messages`)

```
curl http://localhost:8080/messages
```

Réponse attendue : `200 OK`, avec un tableau JSON contenant le message créé.

> **Point de contrôle** : la liste renvoyée contient bien le message précédent.

3. (Optionnel) Récupérer / supprimer un message (`GET` / `DELETE /messages/{id}`)

```
# Récupérer le message d'identifiant 1
curl http://localhost:8080/messages/1

# Supprimer le message d'identifiant 1
curl -X DELETE http://localhost:8080/messages/1
```

> **Point de contrôle** : `GET /messages/1` renvoie `200 OK` et le message ; `DELETE /messages/1` renvoie `204 No Content`, puis `GET /messages/1` renvoie `404 Not Found`. (Remplacez `1` par l'identifiant renvoyé lors de la création si votre schéma a été recréé.)

5.1 (Optionnel) Exécuter les tests générés

Le projet généré inclut un test `GreetingResourceTest` annoté `@QuarkusTest`, qui démarre l'application en mode test et vérifie le bon fonctionnement de l'endpoint REST. Lancez la suite de tests depuis la racine du projet :

```
quarkus test
```

> **Point de contrôle** : la commande se termine par `BUILD SUCCESS` et le test `GreetingResourceTest` est en vert.

6 À retenir

- **Panache** simplifie drastiquement JPA : entités et repositories sans boilerplate (moins de code, les champs publics suffisent).
- **CDI** est activé par défaut : `@ApplicationScoped` et `@Inject` sans `beans.xml`.
- **Transactions** gérées de façon déclarative avec l'annotation `@Transactional`.
- **Configuration centralisée** dans un unique fichier `application.properties` (plus de `persistence.xml` / `web.xml`).
- **Dev Mode** : hot reload instantané, Dev UI puissant et tests continus.
- **Standards Jakarta EE** implémentés : les parties fondées sur les API Jakarta (REST, JPA, CDI, `@Transactional`) restent portables vers d'autres serveurs ; **Panache est spécifique à Quarkus**.

6.1 Différences clés avec Jakarta EE traditionnel

Aspect	Jakarta EE Classique	Quarkus + Panache
Entité JPA	Getters/setters (souvent dans les anciens exemples)	Champs publics suffisent
DAO/Repository	EntityManager manuel	PanacheRepository automatique
Queries	JPQL verbeux	API fluide simplifiée
Configuration	persistence.xml + web.xml	application.properties unique
CDI	beans.xml (facultatif depuis CDI 1.1)	Activé par défaut
Démarrage	~10s	<1s en mode dev

Source : ce TP s'appuie sur le quickstart officiel Hibernate ORM Panache de Quarkus — https://github.com/quarkusio/quarkus-quickstarts/tree/main/hibernate-orm-panache-quickstart.