Développement d'une API REST avec Quarkus

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

2026-11-01

Table des matières

1 Objectifs 1
2 Présentation de Quarkus 1
3 Partie 1: API REST basique 3
3.1 Configuration du projet 3
3.2 Première ressource REST 3
3.3 Endpoints à implémenter 7
3.4 Points à respecter 7
4 Partie 2: Connexion des entités avec JPA 8
4.1 2.1 Entité JPA 8
5 Partie 3 (Optionnel) : Version V4 avec DTOs et mapping 12
6 Partie 3: Intégration de Panache 14
6.1 3.1 Configuration de la base de données 14
6.2 3.2 Entités et repository Panache 15
6.3 Panache REST 16
7 Points d'attention 17
8 Conteneurs 17

1 Objectifs

Développer une API REST pour gérer un catalogue de produits, permettant de:

1. Créer des endpoints REST basiques
2. Manipuler différents types de paramètres
3. Intégrer la persistance avec Panache

2 Présentation de Quarkus

Quarkus est un framework Java innovant conçu pour le développement d'applications cloud-natives et serverless. Créé par Red Hat, il se distingue par son approche « container-first » et ses performances exceptionnelles, notamment un temps de démarrage ultra-rapide et une faible consommation mémoire. Quarkus

utilise GraalVM pour la compilation native, permettant de créer des applications Java optimisées pour les environnements cloud. Il propose une expérience de développement moderne avec le hot reload, une configuration simplifiée et une excellente intégration avec les technologies cloud comme Kubernetes. Son architecture extensible via les extensions Quarkus facilite l'intégration de nombreuses bibliothèques et frameworks Java populaires (Hibernate, RESTEasy, etc.). Pour le développement d'APIs REST, Quarkus offre un modèle de programmation réactif et impératif, avec une excellente intégration de standards comme JAX-RS et des outils modernes comme Panache pour simplifier la persistance.

En plus des libraires, Quarkus propose un outils en ligne de commande pour la création de projets, la gestion des dépendances, la compilation et l'exécution des applications. Il fournit également une interface de développement web (Dev UI) pour visualiser les métriques, les logs, les configurations et les endpoints de l'application. Cette outils gère les projets java avec Maven ou Gradle.

Le démarrage est simple: https://quarkus.io/get-started/ et la documentation est très complète: https://quarkus.io/guides/

```
quarkus create app --maven --java=21 --wrapper --no-code --no-dockerfiles \
    org.acme:product-catalog:1.0.0-SNAPSHOT \
    --extension="quarkus-jdbc-postgresql" \
    --extension="quarkus-hibernate-orm-rest-data-panache" \
    --extension="resteasy-jackson" \
    --extension="quarkus-smallrye-openapi" \
    --extension="quarkus-smallrye-health"
```

```
src/main/java/com/example
 ├─ domain
 │    └─ Product.java               # Modèle métier (règles liées au produit
lui-même)
 ├─ persistence
 │    ├─ ProductEntity.java       # Représentation base de données (JPA)
 │    └─ ProductRepository.java   # Accès aux données (CRUD, requêtes)
 ├─ service
 │     └─ ProductService.java        # Cas d'usage métier (orchestration +
règles globales)
 ├─ api
 │    └─ ProductResource.java     # API REST (HTTP / JSON)
 └─ dto
      └─ ProductDTO.java          # Objets échangés avec l'API
```

Domain = « Qu'est-ce qu'un produit et comment il se comporte ? » : Règles métier liées à l'objet Repository = « Comment je stocke ? » : En mémoire, fichier, base de données (JDBC, JPA, ...) Service = « Comment faire ? est-ce que j'ai le droit ? » : Règles métier, validation, orchestration API (Resource) = « Comment j'expose ça au monde extérieur ? » : HTTP, REST, JSON

```
mkdir -p src/main/java/org/acme/{domain,persistence,service,api,dto}
```

3 Partie 1: API REST basique

3.1 Configuration du projet

```
quarkus create app \
    org.acme:product-catalog:1.0.0-SNAPSHOT \
    --extension="io.quarkus:quarkus-hibernate-orm-rest-data-panache" \
    --extension="io.quarkus:quarkus-jdbc-postgresql" \
    --extension="io.quarkus:quarkus-rest-jackson" \
    --gradle=false
```

Cette commande crée un projet Quarkus avec les extensions et options suivantes:

- `quarkus-hibernate-orm-rest-data-panache`: pour la persistance des données avec Panache
- `quarkus-jdbc-postgresql`: pour la connexion à une base de données PostgreSQL
- `quarkus-rest-jackson`: pour la sérialisation/désérialisation JSON
- `gradle=false`: pour utiliser Maven comme système de build

entrez dans le répertoire du projet et ouvrez-le dans votre IDE préféré.

3.2 Première ressource REST

Créez une classe `Product` pour représenter un produit (sku - Stock Keeping Unit (identifiant unique du produit) , nom, prix, stock)

```
package org.acme.domain;

import java.math.BigDecimal;

/**
 * Product métier immuable avec factory et méthodes withX pour mise à jour
 * fonctionnelle
 */
public record Product(String sku, String name, BigDecimal price, int stock) {

    // Factory method pour validation
```

```
     public static Product of(String sku, String name, BigDecimal price, int
stock) {
        if (sku == null || sku.isBlank()) {
            throw new IllegalArgumentException("SKU cannot be null or empty");
        }
        if (name == null || name.isBlank()) {
            throw new IllegalArgumentException("Name cannot be null or empty");
        }
        if (price == null || price.compareTo(BigDecimal.ZERO) < 0) {
         throw new IllegalArgumentException("Price cannot be null or negative");
        }
        if (stock < 0) {
            throw new IllegalArgumentException("Stock cannot be negative");
        }
        return new Product(sku, name, price, stock);
    }

    // Méthodes withX pour “modifier” l’objet de manière immuable
    public Product withName(String newName) {
        return Product.of(this.sku, newName, this.price, this.stock);
    }

    public Product withPrice(BigDecimal newPrice) {
        return Product.of(this.sku, this.name, newPrice, this.stock);
    }

    public Product withStock(int newStock) {
        return Product.of(this.sku, this.name, this.price, newStock);
    }

    // Méthodes métier
    public boolean isInStock() {
        return stock > 0;
    }

    @Override
    public String toString() {
        return String.format("Product[sku=%s, name=%s, price=%s, stock=%d]",
                sku, name, price, stock);
    }
}
```

et une classe `ProductResource` pour gérer les opérations CRUD sur les produits.

```java
package org.acme.api;

import org.acme.model.Product;
import jakarta.ws.rs.*;
import jakarta.ws.rs.core.MediaType;
import jakarta.ws.rs.core.Response;

import java.util.ArrayList;
import java.util.List;
import java.util.Optional;

/**
 * Resource REST minimale pour Product
 * - GET all
 * - POST create
 */
@Path("/api/v1/products")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class ProductResourceV1 {

    // Stockage en mémoire pour la démo V1
    private static final List<Product> products = new ArrayList<>();

    /**
     * GET /api/v1/products
     */
    @GET
    public List<Product> getAllProducts() {
        return products;
    }

    /**
     * POST /api/v1/products
     */
    @POST
    public Response createProduct(Product product) {
        // Vérification SKU unique
        Optional<Product> existing = products.stream()
                .filter(p -> p.sku().equals(product.sku()))
                .findFirst();
        if (existing.isPresent()) {
            return Response.status(Response.Status.CONFLICT)
                    .entity("Product with this SKU already exists").build();
```

```
        }

        products.add(product);
       return Response.status(Response.Status.CREATED).entity(product).build();
    }
}
```

Démarrez l'application avec la commande `./mvnw quarkus:dev` et accédez à l'URL `http://localhost:8080/api/v1/products` pour tester les endpoints. Vous pouvez utiliser un client REST comme Postman ou curl pour envoyer des requêtes POST et GET.

```
curl -iX GET http://localhost:8080/api/v1/products

curl -iX POST http://localhost:8080/api/v1/products \
     -H "Content-Type: application/json" \
     -d '{"sku":"SKU1","name":"Laptop","price":1000,"stock":10}'
```

Exercice : Enrichissement de l'API REST — Version v2 (M1)

Dans une nouvelle Resource `ProductResourceV2` associée au path `/api/v2/products`, complétez l'API pour gérer les produits en mémoire.

3.3 Endpoints à implémenter

- `GET /api/v2/products` → Récupère tous les produits existants. Tip : pas besoin d'exception ici, renvoyez simplement une liste vide si aucun produit n'est présent.
- `GET /api/v2/products/{sku}` → Récupère un produit spécifique par son SKU. → Retournez **404 Not Found** si le produit n'existe pas. Astuce : utilisez `jakarta.ws.rs.NotFoundException` pour renvoyer automatiquement 404.
- `POST /api/v2/products` → Ajoute un nouveau produit. → Vérifiez que le SKU est unique et retournez **409 Conflict** si le produit existe déjà. → Retournez **201 Created** si l'ajout est réussi. Astuce : utilisez `jakarta.ws.rs.ConflictException` pour gérer le conflit de SKU. Tip : assurez-vous de valider les champs métier (SKU non vide, stock >= 0, prix >= 0).
- `DELETE /api/v2/products/{sku}` → Supprime un produit existant. → Retournez **204 No Content** si la suppression est réussie et **404 Not Found** si le produit n'existe pas. Astuce : utilisez `NotFoundException` pour gérer les SKU inexistants.

3.4 Points à respecter

- Conservez l'immutabilité du record `ProductV2`.
- Validez les données d'entrée côté métier (SKU non vide, stock >= 0, prix >= 0).
- Utilisez les codes HTTP appropriés pour chaque situation.
- Testez vos endpoints avec `curl -i` ou Postman pour vérifier les headers et les codes HTTP.
- Bonus pédagogique : utilisez `Optional` et les méthodes fonctionnelles (`stream().findFirst().orElseThrow(...)`) pour gérer la recherche des produits par SKU.
- Tip : structurer votre code pour que la logique métier reste séparée de la logique REST, même en mémoire, facilite la transition vers JPA/Panache plus tard.

Le fichier `products.http` contient des exemples de requêtes HTTP pour tester votre API. Vous pouvez aussi utiliser les commandes `curl` ou les plugins REST de votre IDE (comme REST Client pour VSCode).

4 Partie 2: Connexion des entités avec JPA

4.1 2.1 Entité JPA

Créez une entité JPA `ProductEntity` pour représenter un produit dans la base de données.

```java
package org.acme.persistence;

import jakarta.persistence.*;
import java.math.BigDecimal;

@Entity
@Table(name = "products")
public class ProductEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(unique = true, nullable = false)
    private String sku;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private BigDecimal price;

    @Column(nullable = false)
    private int stock;

    // Constructeurs
    public ProductEntity() {
    }

    public ProductEntity(String sku, String name, BigDecimal price, int stock) {
        this.sku = sku;
        this.name = name;
        this.price = price;
        this.stock = stock;
    }

    // Getters / Setters
    public Long getId() {
```

```
        return id;
    }

    public String getSku() {
        return sku;
    }

    public void setSku(String sku) {
        this.sku = sku;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public BigDecimal getPrice() {
        return price;
    }

    public void setPrice(BigDecimal price) {
        this.price = price;
    }

    public int getStock() {
        return stock;
    }

    public void setStock(int stock) {
        this.stock = stock;
    }
}
```

Créez un repository JPA `ProductRepository` pour gérer les opérations de base de données.

```
package org.acme.persistence;

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
```

```java
import java.util.List;

@ApplicationScoped
public class ProductRepository {

    @PersistenceContext
    private EntityManager em;

    public ProductEntity save(ProductEntity product) {
        em.persist(product);
        return product;
    }

    public List<ProductEntity> findAll() {
      return em.createQuery("SELECT p FROM ProductEntity p", ProductEntity.class)
                .getResultList();
    }

    public ProductEntity findBySku(String sku) {
            return em.createQuery("SELECT p FROM ProductEntity p WHERE p.sku
= :sku", ProductEntity.class)
                .setParameter("sku", sku)
                .getResultStream()
                .findFirst()
                .orElse(null);
    }

    public void delete(ProductEntity product) {
        if (em.contains(product)) {
            em.remove(product);
        } else {
            em.remove(em.merge(product));
        }
    }
}
```

Il est très important de distinguer :

- les contraintes métier : Objets métiers (Domain) qui contiennent les règles liées à l'objet lui-même.
- les techniques de persistance : Objets JPA (Entities) et Repository qui gèrent la sauvegarde et la récupération des données.
- les règles globales : Services qui orchestrent les opérations métier et de persistance. Mais indépendamment des détails de stockage.

Supposez que l'on veuille changer la persistance d'une base de données relationnelle vers une base NoSQL, on ne devrait pas avoir à modifier les objets métiers ni la logique métier dans les services.

Nous allons donc introduire une couche de service `ProductService` pour orchestrer les opérations métier.

```
package org.acme.service;

import org.acme.persistence.ProductRepository;
import org.acme.persistence.ProductEntity;
import jakarta.enterprise.context.ApplicationScoped;
import jakarta.ws.rs.NotFoundException;
import jakarta.ws.rs.WebApplicationException;
import jakarta.ws.rs.core.Response;

import java.util.List;

@ApplicationScoped
public class ProductService {

    private final ProductRepository repository;

    public ProductService(ProductRepository repository) {
        this.repository = repository;
    }

    public List<ProductEntity> getAll() {
        return repository.findAll();
    }

    public ProductEntity getBySku(String sku) {
        ProductEntity p = repository.findBySku(sku);
        if (p == null) {
            throw new NotFoundException("Product not found");
        }
        return p;
    }

    public ProductEntity create(ProductEntity product) {
        if (repository.findBySku(product.getSku()) != null) {
              throw new WebApplicationException("Product with this SKU already
exists", Response.Status.CONFLICT);
        }
```

```
        return repository.save(product);
    }

    public void delete(String sku) {
        ProductEntity p = repository.findBySku(sku);
        if (p == null) {
            throw new NotFoundException("Product not found");
        }
        repository.delete(p);
    }
}
```

```
# GET tous les produits (liste vide au démarrage)
curl -iX GET http://localhost:8080/api/v3/products

# POST pour créer un nouveau produit
curl -iX POST http://localhost:8080/api/v3/products \
     -H "Content-Type: application/json" \
     -d '{"sku":"SKU1","name":"Laptop","price":1000,"stock":10}'

# POST avec SKU existant (devrait renvoyer 409 Conflict)
curl -iX POST http://localhost:8080/api/v3/products \
     -H "Content-Type: application/json" \
     -d '{"sku":"SKU1","name":"Laptop 2","price":1200,"stock":5}'

# GET produit par SKU
curl -iX GET http://localhost:8080/api/v3/products/SKU1

# GET produit inexistant (devrait renvoyer 404)
curl -iX GET http://localhost:8080/api/v3/products/SKU999

# DELETE produit existant
curl -iX DELETE http://localhost:8080/api/v3/products/SKU1

# DELETE produit inexistant (devrait renvoyer 404)
curl -iX DELETE http://localhost:8080/api/v3/products/SKU999
```

5 Partie 3 (Optionnel) : Version V4 avec DTOs et mapping

Dans les versions précédentes (V1 → V3), notre API exposait directement les **entités JPA**. Si cela fonctionne pour une petite application ou un prototype, ce n'est pas idéal pour un projet évolutif :

- **Couplage fort entre API et base de données** : tout changement dans l'entité JPA impacte directement l'API.
- **Exposition de détails internes** : certains champs de l'entité ne devraient pas être envoyés à l'utilisateur.
- **Difficulté à appliquer des règles métier indépendamment du stockage**.

Pour répondre à ces besoins, on introduit les **DTOs (Data Transfer Objects)**.

Un DTO est un objet **simple, immuable et orienté API**, qui contient uniquement les données que l'on souhaite exposer. Il peut être utilisé en entrée (requêtes) et en sortie (réponses) de l'API.

```
// DTO exposé par l'API
public record ProductDTO(String sku, String name, BigDecimal price, int stock) { }

// DTO utilisé pour créer un produit
public record CreateProductRequest(String sku, String name, BigDecimal price,
int stock) { }
```

- `ProductDTO` est retourné par les endpoints GET.
- `CreateProductRequest` est reçu par les endpoints POST.
- Les DTOs ne contiennent **aucune logique métier ni référence à la base**.

On appelle mapping, l'action de transformer :

```
* Une **entité JPA** → un **DTO** pour l'API
* Un **DTO reçu** → une **entité JPA** pour la persistance
```

Dans la version propre (V4), le mapping est réalisé **dans le Service** :

- La Resource REST reste **simple et centrée sur HTTP**.
- Le Service gère :
  - Les règles métier globales (SKU unique, validation, existence).
  - Le mapping Entity ↔ DTO.
  - La persistance via le Repository.

Exemple :

```
private ProductDTO toDTO(ProductEntity entity) {
    return new ProductDTO(
        entity.getSku(),
        entity.getName(),
        entity.getPrice(),
        entity.getStock()
```

```
    );
}
```

Flux de données V4

1. Le client envoie un `CreateProductRequest` via POST.
2. La **Resource V4** transmet la requête au **Service V4**.
3. Le **Service V4** :
   - Vérifie les règles métier.
   - Crée une `ProductEntity` à partir du DTO.
   - Sauvegarde via le Repository.
   - Retourne un `ProductDTO` à la Resource.
4. La Resource renvoie le DTO au client.

Avec la V4, on peut facilement :

- Faire évoluer l'API sans impacter la base de données.
- Appliquer des règles métier indépendamment du stockage.
- Contrôler précisément les données exposées via les DTOs.

Mais cela ajoute un peu de code boilerplate pour le mapping. Comme c'est très classique, il existe des bibliothèques comme MapStruct pour automatiser ce mapping.

6 Partie 3: Intégration de Panache

Panache est une extension Quarkus qui simplifie drastiquement la persistance des données en proposant une approche active record ou des repositories pour JPA/Hibernate. En étendant PanacheEntity ou en implémentant PanacheRepository, les développeurs obtiennent automatiquement un ensemble complet d'opérations CRUD et de méthodes de requêtage avec une syntaxe fluide et intuitive. Par exemple, `Person.findAll()` ou `Person.find("name", "John")` remplacent les traditionnelles requêtes JPA verbeuses. Panache offre deux modèles de programmation : l'approche active record où les entités contiennent leur logique de persistance, et l'approche repository qui sépare cette logique dans des classes dédiées. L'extension quarkus-hibernate-orm-rest-data-panache va encore plus loin en générant automatiquement une API REST CRUD complète à partir de vos entités, réduisant considérablement le code boilerplate nécessaire.

6.1 3.1 Configuration de la base de données

Ajoutez les propriétés suivantes dans le fichier `application.properties` pour configurer la connexion à une base de données PostgreSQL (à vous de créer la base de données PostgreSQL avec docker, adpattez les paramètres de connexion si besoin):

```
docker run -d \
  --name postgres-tpjparest \
  -e POSTGRES_USER=tpuser \
  -e POSTGRES_PASSWORD="Tp@2026" \
  -e POSTGRES_DB=products \
  -p 15432:5432 \
  -v postgres-tpjparest-data:/var/lib/postgresql/data \
  --restart unless-stopped \
  postgres:16
```

```
quarkus.datasource.db-kind=postgresql
quarkus.datasource.username=tpuser
quarkus.datasource.password="Tp@2026!"
quarkus.datasource.jdbc.url=jdbc:postgresql://localhost:15432/products
quarkus.hibernate-orm.database.generation=drop-and-create
```

Il ne sera plus nécessaire de configurer le persistence.xml, ni de créer des entity managers, Panache se charge de tout.

Plus de détails sur la configuration de la base de données: https://quarkus.io/guides/datasource

6.2 3.2 Entités et repository Panache

Transformez la classe `Product` en une entité JPA.

La classe suivante crée un repository Panache pour gérer les opérations de base de données en définissant des méthodes personnalisées associées à un ensemble de méthode génériques fournies par PanacheRepository.

Le détail est expliqué ici : https://quarkus.io/guides/rest-data-panache#hr-hibernate-orm

particulierement le repository pattern : https://quarkus.io/guides/hibernate-orm-panache#solution-2-using-the-repository-pattern

```
package org.acme.repository

import io.quarkus.hibernate.orm.panache.PanacheRepository;
import io.quarkus.panache.common.Page;
import jakarta.enterprise.context.ApplicationScoped;
import java.math.BigDecimal;
import java.util.List;
import org.acme.entity.ProductEntity;
```

```java
@ApplicationScoped
public class ProductRepository implements PanacheRepository<Product> {

    // Méthodes personnalisées
   public List<ProductEntity> findByPriceRange(BigDecimal min, BigDecimal max) {
        return find("price ⩾ ?1 and price ⩽ ?2", min, max).list();
    }

    public List<ProductEntity> findInStock() {
        return find("stock > 0").list();
    }

    public List<ProductEntity> searchByName(String name, Page page) {
        return find("name like ?1", "%" + name + "%")
            .page(page)
            .list();
    }

    public ProductEntity findBySku(String sku) {
        return find("sku", sku).firstResult();
    }

    public List<ProductEntity> findByCategory(String categoryName) {
        return find("category.name", categoryName).list();
    }
}
```

6.3 Panache REST

Quarkus propose une extension `quarkus-hibernate-orm-rest-data-panache` qui permet de générer automatiquement une API REST CRUD complète à partir de vos entités. Cette approche, inspirée de Spring Data REST, réduit considérablement le code boilerplate en générant automatiquement les endpoints REST standards à partir de vos repositories Panache.

En résumé, voici comment utiliser un repository Panache dans une ressource REST:

```java
package org.acme.resource;

import io.quarkus.hibernate.orm.rest.data.panache.PanacheRepositoryResource;
import io.quarkus.rest.data.panache.ResourceProperties;

@ResourceProperties(path = "products", hal = true)
public              interface              ProductResource              extends
```

```
PanacheRepositoryResource<ProductRepository, ProductEntity, Long> {
    // Les endpoints CRUD sont automatiquement générés:
    // GET /products
    // GET /products/{id}
    // POST /products
    // PUT /products/{id}
    // DELETE /products/{id}
    // GET /products?page=0&size=20
}
```

Cette simple interface génère automatiquement: - Une API REST complète avec les opérations CRUD - La pagination - Le support du format HAL pour l'hypermedia - La documentation OpenAPI - La validation des données - La gestion des erreurs

Vous pouvez personnaliser le comportement par défaut avec des annotations comme `@MethodProperties` ou en implémentant vos propres méthodes.

Le détail est expliqué ici : https://quarkus.io/guides/rest-data-panache

7 Points d'attention

- Implémentation correcte des endpoints REST
- Gestion appropriée des erreurs et codes HTTP
- Validation des données
- Documentation (OpenAPI/Swagger)

8 Conteneurs

JIB

./mvnw install -Dquarkus.container-image.build=true