

JEE: Introduction à Jakarta Enterprise Edition

Emmanuel BRUNO

Université de Toulon, LIS UMR CNRS 7020

emmanuel.bruno@univ-tln.fr

2026-10-05

Table des matières

1 Applications Complexes	1
2 Serveurs d'Application	2
3 Architecture Multi-tiers	2
4 Introduction à Jakarta EE	2
5 Spécifications — Partie 1 : Web & Données	3
5.1 Spécifications — Partie 2 : Infrastructure & Sécurité	3
5.2 Types de Conteneurs	3
6 Évolution Historique	4
7 L'Écosystème Java Enterprise : Vue d'Ensemble	4
8 Comparaison : Spring Boot vs Jakarta EE (Quarkus/Payara)	4
8.1 Tableau Comparatif Détaillé	5
8.2 Quel Framework Choisir ?	6
9 Pourquoi Quarkus dans ce Cours ?	6
10 Installation et Configuration Quarkus ...	6
10.1 Installation via CLI (Recommandée)	6
10.2 Outils de Développement	7
10.3 Modes d'Exécution Quarkus	7
11 Première Application Jakarta EE avec Quarkus	7
11.1 Architecture du Projet	7
11.2 Composants Principaux	7
11.2.a 1. Entité JPA avec Panache (Message.java)	7
11.2.b 2. Repository Panache (MessageRepository.java)	8
11.2.c 3. Ressource REST (MessageResource.java)	8
11.3 Configuration (application.properties)	9
11.4 Tester l'Application	9
11.5 Différences Clés avec Jakarta EE Traditionnel	10
11.6 Dev UI Quarkus - Fonctionnalités ..	10
11.7 Points Clés à Retenir	10

1 Applications Complexes

- **Gestion des ressources :**
 - ▶ Bases de données, systèmes de fichiers, services externes, etc.
 - ▶ Utilisation efficace et gestion des connexions, des pools de connexions, etc.
- **Accès distant :**
 - ▶ Protocoles et technologies : TCP/IP, RMI, CORBA, REST, WebSocket, etc.
 - ▶ Communication entre services et intégration de systèmes distribués.
- **Transactions :**
 - ▶ Gestion des transactions, y compris les transactions applicatives.
 - ▶ Assurer l'intégrité des données et la cohérence des opérations.
- **Gestion et répartition de la charge :**
 - ▶ Clustering et équilibrage de charge pour améliorer la disponibilité et la performance.

- Répartition des requêtes et gestion des sessions.
- **Arrêt des serveurs sans arrêt de l'application :**
 - Techniques pour mettre à jour ou redémarrer les serveurs sans interrompre le service.
 - Gestion des déploiements continus et des mises à jour en direct.
- **Sécurité :**
 - Authentification, autorisation, chiffrement des données, etc.
 - Protection contre les attaques et les vulnérabilités.

2 Serveurs d'Application

- **Focus sur la logique métier :**
 - Permet aux développeurs de se concentrer sur la partie « métier » de l'application.
 - Les services techniques sont fournis par le serveur d'application (middleware).
- **Services fournis par le serveur d'application :**
 - Gestion des transactions, sécurité, gestion des connexions, etc.
 - Facilite le développement en fournissant des services prêts à l'emploi.
- **Déploiement d'applications :**
 - L'application est « déployée » sous forme de briques logicielles sur le serveur.
 - Simplifie la gestion et la maintenance des applications.
- **Spécifications « parapluie » pour Java :**
 - S'appuient sur des « standards » pour les services courants.
 - Définissent une approche spécifique pour coordonner les services.
- **Solutions les plus utilisées :**
 - Jakarta EE : Une plateforme robuste pour le développement d'applications d'entreprise.

- Spring : Un framework flexible et modulaire pour le développement d'applications Java.

• Infrastructure Middleware

- Couche intermédiaire entre OS et applications
- Gestion des ressources système
- Orchestration des services techniques

3 Architecture Multi-tiers

• Tier Client

- Navigateurs web (HTML5/JS), Applications mobiles, Clients Java SE, Applications REST

• Tier Web

- *Présentation* : Jakarta Faces (JSF), Jakarta Pages (JSP)
- *Contrôleurs* : Jakarta Servlet, Jakarta REST (JAX-RS), Jakarta WebSocket

• Tier Enterprise

- *Services Métier* : Enterprise JavaBeans (EJB), Context & Dependency Injection (CDI), Jakarta Transactions (JTA)
- *Messaging* : Jakarta Messaging (JMS), Message-Driven Beans

• Tier Données

- *Persistence* : Jakarta Persistence (JPA), Jakarta NoSQL
- *Transaction* : Jakarta Transactions (JTA)

4 Introduction à Jakarta EE

Jakarta EE (anciennement Java EE / J2EE) est une plateforme standardisée pour le développement d'applications d'entreprise Java.

• Objectifs :

- Fournir des APIs standardisées pour les besoins enterprise
- Garantir la portabilité entre serveurs d'applications
- Faciliter le développement par des services prêts à l'emploi

- Supporter les architectures modernes (cloud, microservices)
- **Gouvernance :**
 - Maintenu par l'Eclipse Foundation depuis 2017
 - Spécifications ouvertes et communautaires
 - Certifications de compatibilité des serveurs

5 Spécifications — Partie 1 : Web & Données

Web & Services

- **Servlet 6.1** - Conteneur web & gestion HTTP
- **RESTful Web Services 4.0** (JAX-RS) - Services web REST
- **WebSocket 2.2** - Communication bidirectionnelle temps réel
- **Faces 4.1** - Framework web MVC
- **Pages 4.0** (JSP) - Pages web dynamiques

Données & Persistance

- **Persistence 3.2** (JPA) - Mapping objet-relationnel
 - Support Java Records (`@Embeddable`, `@IdClass`)
 - Mapping automatique `java.time.Instant` et `java.time.Year`
 - `EntityManager` injectable avec scope CDI
- **Data 1.0** ☆ NEW - Repositories simplifiés
- **NoSQL 1.0** - Accès bases NoSQL

5.1 Spécifications — Partie 2 : Infrastructure & Sécurité

Infrastructure & Services

- **CDI 4.1** - Injection de dépendances & contextes
- **Dependency Injection 2.0** - IoC de base
- **Transactions 2.0** (JTA) - Gestion transactions

- **Enterprise Beans 4.0** (EJB) - Composants métier

Sécurité & Validation

- **Security 4.0** - Authentification & autorisation
- **Authentication 3.1** - Mécanismes d'authentification
- **Authorization 3.0** - Gestion des autorisations
- **Validation 3.1** - Validation de données
 - Support Java Records

Formats & Messaging

- **JSON Binding 3.0** - Sérialisation JSON
- **JSON Processing 2.1** - Traitement JSON
- **Messaging 3.1** (JMS) - Messagerie asynchrone

☑ **Documentation complète** : Jakarta EE
11 Specifications | 🎓 Tutorial

5.2 Types de Conteneurs

- **Web Container**
 - *Composants Web*
 - Servlets (Jakarta Servlet)
 - Pages JSP/Facelets (Jakarta Faces)
 - WebSocket Endpoints
 - Resources REST (Jakarta RESTful WS)
 - *Services*
 - HTTP request/response
 - Cookie/Session management
 - Web security & HTTPS
 - Content negotiation
- **EJB Container**
 - *Enterprise Beans*
 - Stateless/Stateful Session Beans
 - Singleton Session Beans
 - Message-Driven Beans
 - *Services*
 - Transaction management (JTA)
 - Security (JAAS)

- Resource pooling
- Asynchronous messaging
- Timer services

- **Application Client Container** : Java SE clients, Remote EJB access, JNDI naming, Security context

6 Évolution Historique

Ère J2EE (1999-2006)

- **1999** : J2EE 1.2 - Première version (EJB, Servlet, JSP)
- **Problème** : Complexité excessive (XML verbeux, EJB 2.x difficiles)
- **2004** : Spring Framework 1.0 émerge comme alternative légère

Ère Java EE (2006-2017)

- **2006** : Java EE 5 - Révolution (Annotations, JPA, simplification)
- **2009** : Java EE 6 - CDI & Web Profile
- **2013** : Java EE 7 - Cloud & HTML5
- **2014** : Spring Boot 1.0 devient game changer
- **2017** : Java EE 8 - Dernière version Oracle

Ère Jakarta EE (2017-présent)

- **2017** : Transfert à Eclipse Foundation
- **2020** : Jakarta EE 9 - Nouveau namespace (`javax.*` → `jakarta.*`)
- **2022** : Jakarta EE 10 - Cloud-Native (Core Profile, CDI Lite)
- **2024** : Jakarta EE 11 - Java 21 LTS, Developer Experience

L'Émergence Cloud-Native (2015-2020)

- Problème : Serveurs traditionnels inadaptés au cloud
- **2018** : Quarkus (Red Hat) - Compilation native GraalVM
- **2021** : Spring Native - Réponse à Quarkus
- Résultat : Innovation accélérée, choix variés

Évolution clé : De J2EE complexe → Spring simple → Jakarta EE modernisé → Quarkus cloud-native

7 L'Écosystème Java Enterprise : Vue d'Ensemble

Plateformes Standards Jakarta EE

- *Serveurs Traditionnels (Certifiés Jakarta EE)*
 - **WildFly** - JBoss/Red Hat
 - **Payara** - Dérivé GlassFish
 - **Open Liberty** - IBM
 - **GlassFish** - Implémentation de référence
 - **Apache TomEE** - Extension Tomcat
- *Serveurs Cloud-Native (Utilisent des specs Jakarta EE)*
 - **Quarkus** - Red Hat, GraalVM - Implémente des spécifications Jakarta EE individuelles (non certifié Jakarta EE Full/Web Profile)
 - **Payara Micro** - Microservices (Certifié Web Profile)
 - **Open Liberty** - Mode conteneur (Certifié)

Écosystèmes Alternatifs

- *Spring Framework/Boot*
 - Le plus populaire (~87%)
 - Écosystème complet et mature
 - N'implémente pas Jakarta EE mais utilise des APIs similaires
- *MicroProfile*
 - Standard pour microservices
 - Basé sur un sous-ensemble de Jakarta EE
 - APIs: Health, Metrics, Fault Tolerance

Liste complète des serveurs certifiés : Jakarta EE Compatibility

8 Comparaison : Spring Boot vs Jakarta EE (Quarkus/Payara)

Spring Framework & Spring Boot

- **Forces**
 - Écosystème le plus riche
 - Adoption massive dans l'industrie
 - Documentation exhaustive
 - Convention over configuration
- **Philosophie**
 - Framework « opinionated »
 - Pas de serveur d'application requis
 - Serveur embarqué (Tomcat/Jetty)

Jakarta EE avec Quarkus

- **Forces**
 - Implémente de nombreuses spécifications Jakarta EE
 - Portabilité du code utilisant les specs standard
 - Performance native exceptionnelle
 - Consommation mémoire minimale
- **Philosophie**
 - Utilise les spécifications Jakarta EE individuelles
 - Container-managed services
 - « Build Time » optimization

8.1 Tableau Comparatif Détaillé

Cri- tère	Spring Boot	Quarkus	Jakarta (Payara/Liber- ty)	EE
Standards	Pro- prié- taire	Spécifica- tions Ja- akarta EE + MicroPro- file	Jakarta EE Full Profile	Full
Certi- fica- tion	Non certi- fié	Non certi- fié des specs indi- viduelles)	cert- (uti- des Certifié Jakarta EE	
In- jec- tion	Spring DI	CDI (Ja- akarta)	CDI (Jakarta)	

Cri- tère	Spring Boot	Quarkus	Jakarta (Payara/Liber- ty)	EE
REST	Spring MVC	JAX-RS (RESTEa- sy)	JAX-RS	
Per- sis- tence	Spring Data JPA	JPA (Hi- bernate Panache)	JPA (Eclipse- Link/Hiber- nate)	
Confi- gu- ra- tion	application.properties/ application.yml/ domain.xml			
Pa- cka- ging	JAR exé- cu- table	JAR/Na- tive binary	WAR/EAR	
Dé- mar- rage	~3-5s	<0.1s (na- tif) / ~1s (JVM)	~8-10s	
Mé- moir	~100-150 MB	~20 MB (natif) / ~70 MB (JVM)	~150-200 MB	
Hot Re- load	Spring Dev- Tools	Dev Mode (instanta- né)	Redéploiement	
Po- pu- la- ri- té	☆☆☆☆☆☆ (87%)	☆☆☆☆☆☆ (8%)	☆☆ (5%)	
Use Case	Tout type	Micro- services, Cloud, Ser- verless	Enterprise tra- ditionnelle	

Vérification

```
quarkus --version
# Quarkus CLI 3.x.x
```

10.2 Outils de Développement

- **Quarkus CLI** (recommandé) : Génération code, extensions, dev mode
- **Code.quarkus.io** : Générateur de projets en ligne, sélection extensions graphique
- **VS Code** : Extension Quarkus officielle, Red Hat Java Extension Pack
- **IntelliJ IDEA** : Plugin Quarkus (Ultimate & Community), Support dev mode intégré

10.3 Modes d'Exécution Quarkus

Mode Développement (Dev Mode)

- Live reload automatique
- Console Dev UI : <http://localhost:8080/q/dev>
- Hot deployment instantané
- Debugging intégré
- Tests continus

Mode Production

- Compilation native GraalVM (optionnel)
- Packaging JAR uber-jar
- Optimisations build-time
- Docker/Kubernetes ready

Ports par défaut

- **8080** : Application
- **8443** : HTTPS
- **5005** : Debug

11 Première Application Jakarta EE avec Quarkus

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

```
h2
```

```
cd message-app
```

```
# Lancer en mode dev
quarkus dev
```

Ou via <https://code.quarkus.io> avec les extensions : - RESTEasy Reactive Jackson - Hibernate ORM with Panache - JDBC Driver - H2

11.1 Architecture du Projet

```
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
```

11.2 Composants Principaux

11.2.a 1. Entité JPA avec Panache (Message.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.

11.2.b 2. Repository Panache (MessageRepository.java)

```

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("ORDER BY createdAt
DESC", limit);
}

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

```

11.2.c 3. Ressource REST (MessageResource.java)

```

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);
    if (message == null) {
        return Response.status(Response.S
    }

    return
Response.ok(message).build();
}

@POST
@Transactional
public Response
createMessage(Message message) {
messageRepository.persist(message);
    return Response.status(Response.Stat
        .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.S
    }

    return
Response.noContent().build();
}
}

```

11.3 Configuration (application.properties)

```

# 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

```

11.4 Tester l'Application

```

# Démarrer en mode dev
quarkus dev

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

# Récupérer tous les messages
curl http://localhost:8080/messages

# Dev UI : http://localhost:8080/q/
dev
# Swagger UI : http://localhost:8080/
q/swagger-ui

```

11.5 Différences Clés avec Jakarta EE Traditionnel

Aspect	Jakarta EE Classique	Quarkus + Panache
Entité JPA	Getters/setters obligatoires	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 requis	Activé par défaut
Démarrage	~10s	<1s en mode dev

- **CDI** activé par défaut (pas de beans.xml nécessaire)
- **Transactions** gérées avec `@Transactional`
- **Configuration centralisée** dans `application.properties`
- **Dev Mode** offre hot reload instantané et Dev UI puissant
- **Standards Jakarta EE** implémentés (portabilité possible vers autres serveurs pour code basé sur les specs standard)

Repository GitHub : <https://github.com/quarkusio/quarkus-quickstarts/tree/main/hibernate-orm-panache-quickstart>

11.6 Dev UI Quarkus - Fonctionnalités

Le Dev UI (<http://localhost:8080/q/dev>) offre :

- **Config Editor** : Modifier `application.properties` en temps réel
- **Datasources** : Visualiser les connexions et schéma DB
- **Hibernate** : Voir les entités et requêtes SQL
- **Swagger** : Tester les endpoints REST
- **Continuous Testing** : Lancer les tests automatiquement

11.7 Points Clés à Retenir

- **Panache** simplifie drastiquement JPA (moins de boilerplate)