TP : Premiers pas avec CDI (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 Prérequis 2
3 Étapes 2
3.1 1. Créer le projet `cdi-tp` 2
3.2 2. Un bean simple : `Greeter` 3
3.3 3. Injection par constructeur : `HelloResource` 3
3.4 4. Scopes : un compteur par requête 4
3.5 5. Qualifiers : deux `PaymentService` 5
3.6 6. Événements : `OrderCreatedEvent` 7
4 Vérification 9
5 À vous de jouer 10
6 À retenir 10

1 Objectifs

Dans ce **travail pratique (TP)**, vous saisissez vous-même chaque commande dans le **devcontainer** (Quarkus + Java) et vous construisez, pas à pas, une petite application **CDI** avec **Quarkus** : un bean, une ressource qui l'injecte par constructeur, un bean par requête, deux services distingués par des qualifiers et un événement observé. 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 :

- Générer un projet Quarkus avec la CLI et le lancer en **Dev Mode**
- Écrire un bean CDI et l'injecter par constructeur
- Distinguer `@ApplicationScoped` (instance partagée) de `@RequestScoped` (instance par requête)
- Lever l'ambiguïté entre implémentations grâce aux **qualifiers**
- Découpler des composants via des **événements** CDI (`@Observes` / `@ObservesAsync`)

> **Positionnement** : ce TP prolonge la leçon Inversion de contrôle en Java (`02_java_CDI_L_Introduction.qmd`), et s'inscrit dans la continuité du TP Première application Quarkus de la leçon 01 (JEE). Il est **autonome** : il crée son propre projet `cdi-tp` et ne dépend pas des fichiers de la leçon JEE.

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` sont à **créer dans votre éditeur** (le hot reload de Dev Mode recharge les modifications automatiquement).

3 Étapes

3.1 1. Créer le projet `cdi-tp`

Générez le projet à l'aide du Quarkus CLI, avec l'extension RESTEasy Reactive Jackson, puis rentrez dans le répertoire :

```
quarkus create app fr.ebruno.cdi:cdi-tp --extension=rest-jackson
cd cdi-tp
```

Le projet généré présente la structure suivante :

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

Lancez l'application en **Dev Mode** et **laissez-la tourner** : le hot reload chargera chaque fichier créé aux étapes suivantes sans redémarrage.

```
quarkus dev
```

> **Point de contrôle** : le projet est créé, `quarkus dev` démarre en moins d'une seconde et la console affiche l'URL `http://localhost:8080`. Aucune configuration n'est requise : **CDI est activé par défaut** dans Quarkus, il n'y a rien à ajouter dans `application.properties`.

3.2 2. Un bean simple : `Greeter`

Créez le fichier `src/main/java/fr/ebruno/cdi/bean/Greeter.java` :

```
package fr.ebruno.cdi.bean;

import jakarta.enterprise.context.ApplicationScoped;

@ApplicationScoped
public class Greeter {
    public String greet(String name) {
        return "Bonjour, " + name + " !";
    }
}
```

Une classe annotée d'un **scope** (`@ApplicationScoped` ici) est un **bean** : CDI la découvre automatiquement au démarrage. Contrairement à un serveur d'application classique, **Quarkus n'exige aucun fichier `beans.xml`** : le mode de découverte est « annotated » et CDI est activé par défaut.

3.3 3. Injection par constructeur : `HelloResource`

Supprimez d'abord la ressource générée (et son test) — nous créons nos propres ressources :

```
rm src/main/java/fr/ebruno/cdi/GreetingResource.java
rm src/test/java/fr/ebruno/cdi/GreetingResourceTest.java
```

Créez `src/main/java/fr/ebruno/cdi/resource/HelloResource.java` :

```
package fr.ebruno.cdi.resource;

import fr.ebruno.cdi.bean.Greeter;
```

```java
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.PathParam;

@Path("/hello")
public class HelloResource {

    private final Greeter greeter;

    @Inject
    public HelloResource(Greeter greeter) {
        this.greeter = greeter;
    }

    @GET
    @Path("/{name}")
    public String hello(@PathParam("name") String name) {
        return greeter.greet(name);
    }
}
```

La ressource JAX-RS est elle-même un bean CDI : CDI résout `Greeter` et l'**injecte par constructeur** — le **défaut recommandé**, qui rend le bean immuable (champ `final`), facile à instancier et simple à tester ; c'est le constructeur qui est annoté `@Inject`, pas le champ.

3.4 4. Scopes : un compteur par requête

Créez un bean dont le cycle de vie est lié à la requête HTTP, `src/main/java/fr/ebruno/cdi/bean/RequestCounter.java` :

```java
package fr.ebruno.cdi.bean;

import jakarta.enterprise.context.RequestScoped;

@RequestScoped
public class RequestCounter {
    private int hits = 0;

    public int hit() {
        return ++hits;
    }
}
```

Puis une ressource qui l'exploite, `src/main/java/fr/ebruno/cdi/resource/CounterResource.java` :

```
package fr.ebruno.cdi.resource;

import fr.ebruno.cdi.bean.RequestCounter;
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;

@Path("/counter")
public class CounterResource {

    @Inject
    RequestCounter counter;

    @GET
    public String count() {
        int first = counter.hit();
        int second = counter.hit();
          return "dans cette requête : appel 1 = " + first + " ; appel 2 = "
+ second;
    }
}
```

Un bean `@RequestScoped` vit **une instance par requête HTTP**. Les deux appels `hit()` de `count()` tombent sur la **même instance** (donc `1` puis `2`), mais une requête suivante repart de `0`. Avec `@ApplicationScoped`, l'instance serait **partagée** entre toutes les requêtes (le compteur monterait en continu, `3`, `4`, …). La ressource étant longue-vie, CDI fournit une **proxy client** qui résout l'instance de la requête en cours à chaque appel.

3.5 5. Qualifiers : deux `PaymentService`

Définissez d'abord l'interface, `src/main/java/fr/ebruno/cdi/payment/PaymentService.java` :

```
package fr.ebruno.cdi.payment;

public interface PaymentService {
    String process(double amount);
}
```

Deux implémentations de même type rendraient l'injection **ambiguë** (`AmbiguousResolutionException`). Les **qualifiers** lèvent cette ambiguïté. Créez `src/main/java/fr/ebruno/cdi/payment/PayPal.java` :

```
package fr.ebruno.cdi.payment;

import jakarta.inject.Qualifier;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Qualifier
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE,        ElementType.FIELD,        ElementType.METHOD,
ElementType.PARAMETER})
public @interface PayPal {
}
```

Le fichier `CreditCard.java` (même dossier) est strictement identique, à l'exception du nom de l'annotation : `public @interface CreditCard { }`.

Ensuite les deux implémentations, chacune **qualifiée**. Créez `src/main/java/fr/ebruno/cdi/payment/PayPalPaymentService.java` :

```
package fr.ebruno.cdi.payment;

import jakarta.enterprise.context.ApplicationScoped;

@PayPal
@ApplicationScoped
public class PayPalPaymentService implements PaymentService {
    @Override
    public String process(double amount) {
        return "Paiement PayPal de " + amount + " € accepté";
    }
}
```

Le fichier `CreditCardPaymentService.java` (même dossier) est identique, à l'exception du qualifier (`@CreditCard`), du nom de la classe et du texte renvoyé : `return "Paiement par carte de " + amount + " € accepté";`.

Enfin, la ressource qui injecte **les deux** en les distinguant par leur qualifier, `src/main/java/fr/ebruno/cdi/resource/PayResource.java` :

```java
package fr.ebruno.cdi.resource;

import fr.ebruno.cdi.payment.CreditCard;
import fr.ebruno.cdi.payment.PayPal;
import fr.ebruno.cdi.payment.PaymentService;
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.QueryParam;

@Path("/pay")
public class PayResource {

    private final PaymentService paypal;
    private final PaymentService creditCard;

    @Inject
    public PayResource(@PayPal PaymentService paypal,
                       @CreditCard PaymentService creditCard) {
        this.paypal = paypal;
        this.creditCard = creditCard;
    }

    @GET
    public String pay(@QueryParam("method") String method,
                      @QueryParam("amount") double amount) {
        PaymentService service = "card".equals(method) ? creditCard : paypal;
        return service.process(amount);
    }
}
```

Sans qualifier explicite, CDI utilise le qualifier implicite `@Default`. Ici, chaque implémentation porte un qualifier personnalisé (`@PayPal`, `@CreditCard`) et n'a donc plus `@Default` : l'injection ne passe que par le qualifier correspondant, y compris sur les **paramètres du constructeur**.

3.6 6. Événements : `OrderCreatedEvent`

Un **événement** permet à un émetteur de notifier des observateurs **sans les connaître** (couplage faible). Créez l'objet événement — un **record** immuable, `src/main/java/fr/ebruno/cdi/event/OrderCreatedEvent.java` :

```java
package fr.ebruno.cdi.event;
```

```
public record OrderCreatedEvent(String orderId, double amount) {
}
```

Puis un bean observateur, avec un observateur **synchrone** et un **asynchrone**, `src/main/java/fr/ebruno/cdi/event/OrderObservers.java` :

```
package fr.ebruno.cdi.event;

import jakarta.enterprise.context.ApplicationScoped;
import jakarta.enterprise.event.Observes;
import jakarta.enterprise.event.ObservesAsync;

@ApplicationScoped
public class OrderObservers {

    // Synchrone : exécuté dans le thread de l'émetteur
    public void onOrderCreated(@Observes OrderCreatedEvent event) {
        System.out.println("[SYNC]  " + event.orderId()
                + " : notification client (" + event.amount() + " €)");
    }

    // Asynchrone : exécuté dans un thread séparé
    public void onOrderCreatedAsync(@ObservesAsync OrderCreatedEvent event) {
        System.out.println("[ASYNC] " + event.orderId()
                + " : mise à jour analytics (thread séparé)");
    }
}
```

Enfin, l'émetteur, `src/main/java/fr/ebruno/cdi/resource/OrderResource.java` :

```
package fr.ebruno.cdi.resource;

import fr.ebruno.cdi.event.OrderCreatedEvent;
import jakarta.enterprise.event.Event;
import jakarta.inject.Inject;
import jakarta.ws.rs.GET;
import jakarta.ws.rs.Path;
import jakarta.ws.rs.PathParam;

@Path("/order")
public class OrderResource {

    @Inject
```

```
    Event<OrderCreatedEvent> orderEvent;

    @GET
    @Path("/{id}")
    public String createOrder(@PathParam("id") String id) {
        orderEvent.fire(new OrderCreatedEvent(id, 42.5));
        return "Commande " + id + " créée (événement émis).";
    }
}
```

L'émetteur injecte `Event<OrderCreatedEvent>` et appelle `fire(...)`. L'observateur `@Observes` est invoqué **dans le même thread** que l'émetteur ; l'observateur `@ObservesAsync`, **dans un thread séparé** (traitements longs qui ne doivent pas bloquer la requête). L'émetteur ne référence aucun observateur : en ajouter un ne modifie jamais son code.

4 Vérification

Avec `quarkus dev` toujours en marche (étape 1), ouvrez un **autre terminal** et testez les endpoints avec `curl` :

```
curl http://localhost:8080/hello/Marie
curl http://localhost:8080/counter
curl "http://localhost:8080/pay?method=paypal&amount=19.99"
curl "http://localhost:8080/pay?method=card&amount=19.99"
curl http://localhost:8080/order/ORD-123
```

Réponses attendues :

```
$ curl http://localhost:8080/hello/Marie
Bonjour, Marie !
$ curl http://localhost:8080/counter
dans cette requête : appel 1 = 1 ; appel 2 = 2
$ curl "http://localhost:8080/pay?method=paypal&amount=19.99"
Paiement PayPal de 19.99 € accepté
$ curl "http://localhost:8080/pay?method=card&amount=19.99"
Paiement par carte de 19.99 € accepté
$ curl http://localhost:8080/order/ORD-123
Commande ORD-123 créée (événement émis).
```

Et, dans la **console** de `quarkus dev`, après l'appel `/order/ORD-123` :

```
[SYNC]  ORD-123 : notification client (42.5 €)
[ASYNC] ORD-123 : mise à jour analytics (thread séparé)
```

Points de contrôle : `GET /hello/Marie` renvoie la salutation (injection par constructeur) ; `GET /counter` renvoie **toujours** `appel 1 = 1 ; appel 2 = 2` (scope par requête) ; les deux `method` de `/pay` renvoient la bonne implémentation (qualifiers) ; `/order/ORD-123` déclenche les **deux** observateurs en console (l'asynchrone peut apparaître un instant après, dans un autre thread).

5 À vous de jouer

Niveau : le défi Scope se fait avec le seul cœur de leçon. Les défis Producer et Alternative sont des **bonus** qui mobilisent la **leçon avancée** (`02_java_CDI_L_advanced.qmd` : producers, beans alternatifs).

- **Producer** : créez un bean producteur qui fournit une `java.util.Random` (méthode `@Produces @ApplicationScoped`), puis injectez-la dans une nouvelle ressource qui renvoie un « numéro de loterie » (`random.nextInt(100)`) à chaque appel.
- **Alternative** : écrivez une `MockPaymentService` annotée `@Alternative` (qui simule le paiement) et activez-la avec `@Priority(100)` ; observez que l'injection bascule vers le mock **sans modifier** `PayResource`. Retirez `@Priority` pour revenir au défaut.
- **Scope** : changez `@RequestScoped` en `@ApplicationScoped` sur `RequestCounter` ; relancez `curl http://localhost:8080/counter` plusieurs fois et observez que le compteur **s'accumule d'une requête à l'autre** au lieu de repartir de `0`.

6 À retenir

- **CDI activé par défaut** dans Quarkus : `@ApplicationScoped` + `@Inject`, **sans** `beans.xml` (découverte « annotated »).
- **Injection par constructeur** : le défaut recommandé (bean immuable, champ `final`, testable).
- **Scopes** : `@ApplicationScoped` (instance partagée) vs `@RequestScoped` (instance par requête, résolue via une proxy client).
- **Qualifiers** : lèvent l'ambiguïté entre implémentations d'un même type ; sans qualifier explicite, c'est `@Default`.
- **Événements** : l'émetteur ne connaît pas ses observateurs ; `@Observes` (synchrone, même thread) et `@ObservesAsync` (thread séparé).

Source : ce TP s'appuie sur la spécification **Jakarta CDI** et le guide officiel Quarkus CDI — https://quarkus.io/guides/cdi-reference.