CDI : Concepts Avancés

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

2026-11-07

Table des matières

1 Qualifiers avancés 2
1.1 Qualifiers avec Valeurs 2
1.2 Combinaison de Qualifiers 3
2 Beans Alternatifs 4
2.1 Définition d'un Bean Alternatif 4
2.2 Activation dans beans.xml 4
2.3 Alternatives avec Priorité 5
2.4 Cas d'Usage 5
3 Producers 5
3.1 Producer Method 5
3.2 Producer Field 6
3.3 Producer avec Qualifier 6
3.4 Disposer Method 7
3.5 Cas d'Usage 7
4 Intercepteurs 7
4.1 Design Pattern d'Interception 7
4.2 Création d'un Intercepteur 8
4.3 Annotations de Cycle de Vie 9
4.4 Intercepteurs Multiples 9
5 Décorateurs 10
5.1 Design Pattern Décorateur 10
5.2 Création d'un Décorateur 10
5.3 Activation des Décorateurs 11
5.4 Exemple : Décorateur de Validation 12
5.5 Décorateurs en Chaîne 12
6 Événements avancés 12
6.1 Événements Qualifiés 12
6.2 Événements Conditionnels 13

6.3 Transaction Phases (Java EE) 13
7 Stereotypes 14
7.1 Création d'un Stereotype 14
7.2 Utilisation d'un Stereotype 14
7.3 Stereotypes Avancés 14
7.4 Stereotypes Prédéfinis 15
8 Implémentations CDI 15
9 Bonnes Pratiques avancées 15
9.1 1. Utilisez `@Alternative` pour les Tests 15
9.2 2. Gérez les Ressources avec Producer/Disposer 16
9.3 3. Documentez les Qualifiers et Stereotypes 16
10 Ressources 16

Cette leçon rassemble les mécanismes CDI qui dépassent le socle de base : qualifiers avancés, beans alternatifs, producers, intercepteurs, décorateurs, événements avancés et stereotypes. Elle prolonge directement Introduction à CDI, où le modèle d'injection, les scopes et les qualifiers de base ont été posés.

> **i** Note
>
> **Partie optionnelle.** Cette leçon approfondit les mécanismes CDI au-delà du socle de base. Elle suppose que vous avez lu Introduction à CDI et n'est **pas requise** pour la suite du cours.

Les sections ci-dessous sont largement indépendantes les unes des autres : lisez-les dans l'ordre, ou accédez directement à celle qui vous intéresse.

1 Qualifiers avancés

Les qualifiers de base (`@Qualifier`, `@Default`) ont été posés dans l'introduction ; cette section exploite leur expressivité : qualifiers porteurs d'une valeur, et combinaison de qualifiers pour une injection fine.

1.1 Qualifiers avec Valeurs

Les qualifiers peuvent avoir des valeurs pour plus de flexibilité :

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD, PARAMETER})
public @interface PaymentMethod {
    PaymentType value();
```

```
    enum PaymentType {
        CREDIT_CARD, PAYPAL, BANK_TRANSFER
    }
}

@PaymentMethod(PaymentType.PAYPAL)
@ApplicationScoped
public class PayPalPaymentService implements PaymentService {
    // ...
}

@ApplicationScoped
public class OrderService {
    @Inject
    @PaymentMethod(PaymentType.PAYPAL)
    private PaymentService paypalService;
}
```

1.2 Combinaison de Qualifiers

Plusieurs qualifiers peuvent être combinés pour une injection encore plus précise :

```
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface Secure {}

@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface FastProcessing {}

@PayPal
@Secure
@FastProcessing
@ApplicationScoped
public class SecureFastPayPalService implements PaymentService {
    // ...
}

@ApplicationScoped
public class PremiumOrderService {
    @Inject
    @PayPal
```

```
    @Secure
    @FastProcessing
    private PaymentService paymentService;  // Injection précise!
}
```

2 Beans Alternatifs

Les **beans alternatifs** permettent de remplacer une implémentation par une autre selon l'environnement (développement, test, production) **sans modifier le code source**.

2.1 Définition d'un Bean Alternatif

```
// Bean par défaut (production)
@ApplicationScoped
public class ProductionPaymentService implements PaymentService {
    @Override
    public void processPayment(double amount) {
        // Appel réel à l'API de paiement
        realPaymentGateway.charge(amount);
    }
}

// Bean alternatif (test/développement)
@Alternative
@ApplicationScoped
public class MockPaymentService implements PaymentService {
    @Override
    public void processPayment(double amount) {
        System.out.println("🧪 [MOCK] Paiement simulé : " + amount);
    }
}
```

2.2 Activation dans beans.xml

Pour activer un bean alternatif, déclarez-le dans `beans.xml` :

```
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                           https://jakarta.ee/xml/ns/jakartaee/beans_4_0.xsd"
       version="4.0" bean-discovery-mode="annotated">
    <alternatives>
        <class>com.example.MockPaymentService</class>
```

```
    </alternatives>
</beans>
```

Résultat : `MockPaymentService` remplace `ProductionPaymentService` partout

2.3 Alternatives avec Priorité

CDI 4.0+ permet d'utiliser `@Priority` pour activer les alternatives sans `beans.xml` :

```
@Alternative
@Priority(100)
@ApplicationScoped
public class MockPaymentService implements PaymentService {
    // ...
}
```

Plus la priorité est élevée, plus l'alternative est prioritaire

2.4 Cas d'Usage

- 🔨 **Tests** : Remplacer services externes par des mocks
- 🔧 **Développement** : Simuler des services coûteux
- 🌍 **Environnements** : Basculer entre configurations (dev/staging/prod)

3 Producers

Les **producers** permettent de créer des beans de manière programmatique, utile pour : - Objets non-CDI (bibliothèques tierces) - Logique de création complexe - Configuration dynamique

3.1 Producer Method

```
@ApplicationScoped
public class LoggerProducer {

    @Produces
    public Logger createLogger(InjectionPoint injectionPoint) {
        // Création personnalisée basée sur le point d'injection
        Class<?> targetClass = injectionPoint.getMember().getDeclaringClass();
        return Logger.getLogger(targetClass.getName());
    }
}

@ApplicationScoped
public class OrderService {
    @Inject
```

```
    private Logger logger;  // Logger créé par producer avec le bon nom

    public void createOrder() {
        logger.info("Commande créée");  // [OrderService] Commande créée
    }
}
```

3.2 Producer Field

Pour les cas simples, un champ peut produire un bean :

```
@ApplicationScoped
public class ConfigurationProducer {

    @Produces
   private final String apiUrl = System.getProperty("api.url", "http://localhost:
8080");

    @Produces
    @Named("appVersion")
    private final String version = "1.0.0";
}

@ApplicationScoped
public class ApiClient {
    @Inject
    private String apiUrl;  // Injecté depuis producer field
}
```

3.3 Producer avec Qualifier

```
public class RandomProducer {
    private Random random = new Random();

    @Produces
    @RandomNumber
    public int produceRandomInt() {
        return random.nextInt(100);
    }
}

@ApplicationScoped
public class LotteryService {
    @Inject
```

```
    @RandomNumber
    private int luckyNumber;  // Nombre aléatoire injecté
}
```

3.4 Disposer Method

Un **disposer** permet de nettoyer les ressources créées par un producer :

```
@ApplicationScoped
public class DatabaseProducer {

    @Produces
    @ApplicationScoped
    public EntityManager createEntityManager() {
        return Persistence.createEntityManagerFactory("myPU")
                          .createEntityManager();
    }

    public void closeEntityManager(@Disposes EntityManager em) {
        if (em.isOpen()) {
            em.close();
        }
    }
}
```

3.5 Cas d'Usage

- 📦 **Bibliothèques tierces** : Créer des beans pour des classes non-CDI
- ⚙️ **Configuration** : Injecter des valeurs de configuration
- 🔧 **Création complexe** : Logique de création nécessitant plusieurs étapes
- 🎲 **Objets dynamiques** : Valeurs calculées à chaque injection

4 Intercepteurs

Les **intercepteurs** permettent d'ajouter des comportements transversaux (cross-cutting concerns) à des méthodes sans modifier leur code source.

4.1 Design Pattern d'Interception

L'interception insère du code **avant**, **après** ou **autour** de l'exécution d'une méthode :

```
Client → Intercepteur → Méthode Cible → Intercepteur → Client
         [Avant]        [Exécution]      [Après]
```

Cas d'usage typiques : - 📝 Journalisation (logging) - 🔒 Sécurité (vérification permissions) - 💾 Transactions - ⏱️ Mesure de performance - 📦 Validation

4.2 Création d'un Intercepteur

Étape 1 : Définir l'annotation d'interception

```
@InterceptorBinding
@Retention(RUNTIME)
@Target({TYPE, METHOD})
public @interface Logged {}
```

Étape 2 : Implémenter l'intercepteur

```
@Interceptor
@Logged
@Priority(Interceptor.Priority.APPLICATION)
public class LoggingInterceptor {

    @Inject
    private Logger logger;

    @AroundInvoke
    public Object logMethod(InvocationContext ctx) throws Exception {
        String methodName = ctx.getMethod().getName();

        logger.info("→ Entrée dans " + methodName);
        long start = System.currentTimeMillis();

        try {
            Object result = ctx.proceed();  // Appel méthode cible
            return result;
        } finally {
            long duration = System.currentTimeMillis() - start;
            logger.info("← Sortie de " + methodName + " (" + duration + "ms)");
        }
    }
}
```

Étape 3 : Appliquer l'intercepteur

```
@ApplicationScoped
public class OrderService {

    @Logged  // Méthode interceptée
    public void createOrder(String productId) {
        // Logique métier
        System.out.println("Création commande : " + productId);
```

```
    }
}
```

Résultat :

```
→ Entrée dans createOrder
Création commande : ABC123
← Sortie de createOrder (15ms)
```

4.3 Annotations de Cycle de Vie

CDI offre plusieurs points d'interception :

Annotation	Point d'interception	Usage
`@AroundConstruct`	Autour de la construction	Initialisation custom
`@PostConstruct`	Après construction	Setup initial
`@AroundInvoke`	Autour d'un appel de méthode	Logging, transactions
`@PreDestroy`	Avant destruction	Nettoyage
`@AroundTimeout`	Autour d'un timeout (EJB)	Gestion timeouts

4.4 Intercepteurs Multiples

Plusieurs intercepteurs peuvent être appliqués :

```java
@Logged
@Transactional
@Secured
@ApplicationScoped
public class OrderService {
    public void createOrder(String productId) {
        // Ordre d'exécution : Secured → Transactional → Logged → méthode
    }
}
```

Ordre d'exécution déterminé par `@Priority` (plus petit = plus tôt)

5 Décorateurs

Les **décorateurs** permettent d'étendre dynamiquement le comportement d'un bean en implémentant la même interface.

5.1 Design Pattern Décorateur

Le décorateur **enveloppe** l'objet original et peut modifier son comportement :

```
Client → Décorateur → Bean Original
         [Logique +]   [Logique métier]
```

Différence avec les intercepteurs :

Aspect	Intercepteur	Décorateur
Implémente interface	✕ Non	✅ Oui
Peut modifier params/résultat	⚠ Limité	✅ Oui
Application	Annotation sur méthode/classe	Délégation explicite
Use case	Aspects transversaux (logging, sécurité)	Extension comportement métier

5.2 Création d'un Décorateur

```java
public interface PaymentService {
    boolean processPayment(double amount);
}

@ApplicationScoped
public class BasicPaymentService implements PaymentService {
    @Override
    public boolean processPayment(double amount) {
        System.out.println("💳 Paiement de " + amount + "€");
        return true;
    }
}

@Decorator
@Priority(Interceptor.Priority.APPLICATION)
public abstract class PaymentLoggingDecorator implements PaymentService {
```

```
    @Inject
    @Delegate
    @Any
    private PaymentService delegate;  // Bean original

    @Inject
    private Logger logger;

    @Override
    public boolean processPayment(double amount) {
        logger.info("🔔 Tentative de paiement : " + amount + "€");

       boolean result = delegate.processPayment(amount);  // Appel bean original

        if (result) {
            logger.info("✅ Paiement réussi");
        } else {
            logger.warning("❌ Paiement échoué");
        }

        return result;
    }
}
```

5.3 Activation des Décorateurs

Avec beans.xml :

```
<beans xmlns="https://jakarta.ee/xml/ns/jakartaee"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="https://jakarta.ee/xml/ns/jakartaee
                           https://jakarta.ee/xml/ns/jakartaee/beans_4_0.xsd"
       version="4.0" bean-discovery-mode="annotated">
    <decorators>
        <class>com.example.PaymentLoggingDecorator</class>
    </decorators>
</beans>
```

Ou avec `@Priority` (CDI 2.0+) :

```
@Decorator
@Priority(Interceptor.Priority.APPLICATION)
public abstract class PaymentLoggingDecorator implements PaymentService {
```

```
    // Activé automatiquement
}
```

5.4 Exemple : Décorateur de Validation

```
@Decorator
@Priority(Interceptor.Priority.APPLICATION + 10)
public abstract class PaymentValidationDecorator implements PaymentService {

    @Inject
    @Delegate
    @Any
    private PaymentService delegate;

    @Override
    public boolean processPayment(double amount) {
        // Validation avant délégation
        if (amount ⩽ 0) {
            throw new IllegalArgumentException("Montant invalide : " + amount);
        }

        if (amount > 10000) {
           throw new IllegalArgumentException("Montant trop élevé : " + amount);
        }

        return delegate.processPayment(amount);
    }
}
```

5.5 Décorateurs en Chaîne

Plusieurs décorateurs peuvent être chaînés :

```
Client → ValidationDecorator → LoggingDecorator → BasicPaymentService
         [Valide]               [Log]              [Traite]
```

Ordre déterminé par `@Priority` ou ordre dans `beans.xml`

6 Événements avancés

Au-delà de l'émission et de l'observation de base, CDI raffine les événements : qualifiers, filtrage conditionnel et phases transactionnelles.

6.1 Événements Qualifiés

Les événements peuvent être qualifiés pour filtrer les observateurs :

```java
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, PARAMETER})
public @interface Important {}

// Émission qualifiée
@Inject
@Important
private Event<OrderCreatedEvent> importantOrderEvent;

public void createVIPOrder(String orderId, double amount) {
    importantOrderEvent.fire(new OrderCreatedEvent(orderId, amount));
}

// Observation qualifiée
public void onImportantOrder(@Observes @Important OrderCreatedEvent event) {
    System.out.println("🔔 Commande VIP importante!");
}
```

6.2 Événements Conditionnels

Les observateurs peuvent filtrer les événements :

```java
@ApplicationScoped
public class LargeOrderService {

    public void onLargeOrder(@Observes OrderCreatedEvent event) {
        if (event.getAmount() > 1000) {
             System.out.println("⚠ Commande importante : " + event.getAmount()
+ "€");
        }
    }
}
```

6.3 Transaction Phases (Java EE)

Dans un contexte transactionnel (EJB/JTA), les événements peuvent être observés à différentes phases :

```java
public void afterSuccess(@Observes(during = TransactionPhase.AFTER_SUCCESS)
                         OrderCreatedEvent event) {
    // Exécuté seulement si transaction réussit
}
```

```
public void afterFailure(@Observes(during = TransactionPhase.AFTER_FAILURE)
                         OrderCreatedEvent event) {
    // Exécuté seulement si transaction échoue
}
```

7 Stereotypes

Les **stereotypes** permettent de combiner plusieurs annotations CDI en une seule annotation réutilisable, simplifiant le code et standardisant les patterns.

7.1 Création d'un Stereotype

```
@Stereotype
@Named
@RequestScoped
@Retention(RUNTIME)
@Target(TYPE)
public @interface WebController {}
```

7.2 Utilisation d'un Stereotype

Au lieu de :

```
@Named
@RequestScoped
public class OrderController {
    // ...
}
```

On peut écrire :

```
@WebController
public class OrderController {
    // ...
}
```

7.3 Stereotypes Avancés

Les stereotypes peuvent inclure interceptors :

```
@Stereotype
@ApplicationScoped
@Transactional
@Logged
@Retention(RUNTIME)
```

```java
@Target(TYPE)
public @interface Service {}

// Usage
@Service
public class OrderService {
    // Automatiquement @ApplicationScoped + @Transactional + @Logged
}
```

7.4 Stereotypes Prédéfinis

CDI définit quelques stereotypes standards :

- `@Model` = `@Named` + `@RequestScoped` (pour controllers MVC)

```java
@Model
public class ProductFormController {
    // Équivalent à @Named @RequestScoped
}
```

8 Implémentations CDI

Deux implémentations méritent d'être nommées :

- **Weld** — l'implémentation de référence de la spécification CDI, fournie pour Java SE et Jakarta EE (utilisée par WildFly, Payara, GlassFish).
- **ArC** — le mécanisme CDI de Quarkus : l'injection y est résolue au build-time (code généré, sans réflexion) et aucun fichier `beans.xml` n'est requis.

9 Bonnes Pratiques avancées

Quelques règles qui prennent tout leur sens une fois les mécanismes avancés en place :

9.1 1. Utilisez `@Alternative` pour les Tests

```java
// Production
@ApplicationScoped
public class RealEmailService implements EmailService { }

// Test
@Alternative
@Priority(100)
@ApplicationScoped
public class MockEmailService implements EmailService { }
```

9.2 2. Gérez les Ressources avec Producer/Disposer

```
@ApplicationScoped
public class DatabaseProducer {

    @Produces
    @ApplicationScoped
    public DataSource createDataSource() {
        // Création
    }

    public void closeDataSource(@Disposes DataSource ds) {
        // Nettoyage
    }
}
```

9.3 3. Documentez les Qualifiers et Stereotypes

```
/**
 * Qualifier pour les services de paiement rapide.
 * Ces services garantissent un traitement en < 1s.
 */
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface FastProcessing { }
```

10 Ressources

- 📘 Jakarta CDI Specification
- 📚 Weld Documentation
- 🔧 CDI TCK — Test Compatibility Kit
- 📘📘 Arquillian — Testing CDI applications