

Inversion de contrôle en Java

Emmanuel BRUNO

Université de Toulon, LIS UMR CNRS 7020

emmanuel.bruno@univ-tln.fr

2026-10-14

Table des matières

1	Introduction à CDI (Contexts and Dependency Injection)	2	9.2	Application du Qualifier	10
1.1	Objectifs de CDI	2	9.3	Injection Qualifiée	10
2	Problèmes Résolus par CDI	2	9.4	Qualifier <code>@Default</code>	10
2.1	Avantages Clés	3	9.5	Qualifiers avec Valeurs	11
3	Historique de CDI	3	9.6	Combinaison de Qualifiers	11
4	Qu'est-ce qu'un Bean CDI ?	3	10	Beans Alternatifs	11
4.1	Vue Conceptuelle	3	10.1	Définition d'un Bean Alternatif ...	12
4.2	Vue Technique	4	10.2	Activation dans beans.xml	12
4.3	Résumé : Contrat d'un Bean CDI ...	4	10.3	Alternatives avec Priorité	12
5	Scopes et Cycle de Vie	4	10.4	Cas d'Usage	12
5.1	Scopes Principaux	4	11	Producers	12
5.2	Scope par Défaut	5	11.1	Producer Method	13
5.3	Exemple de Scopes	5	11.2	Producer Field	13
5.4	Passivation et Activation	5	11.3	Producer avec Qualifier	13
6	Injection de Dépendances	6	11.4	Disposer Method	13
6.1	Types d'Injection	6	11.5	Cas d'Usage	14
6.2	Annotation <code>@Named</code>	6	12	Intercepteurs	14
6.3	Injection par Type	7	12.1	Design Pattern d'Interception ...	14
6.4	Ce qui est Injecté	7	12.2	Création d'un Intercepteur	14
6.5	Initializer Methods	7	12.3	Annotations de Cycle de Vie	15
7	Exemple Progressif	8	12.4	Exemple : Intercepteur de Transaction	15
7.1	Étape 1 : Bean Simple	8	12.5	Intercepteurs Multiples	16
7.2	Étape 2 : Avec Interface	8	13	Décorateurs	16
7.3	Étape 3 : Avec Scope Approprié ...	9	13.1	Design Pattern Décorateur	16
8	Conflits de Dépendances	9	13.2	Création d'un Décorateur	16
8.1	Problème : Ambiguïté	9	13.3	Activation des Décorateurs	17
8.2	Solutions	9	13.4	Exemple : Décorateur de Validation	17
9	Qualifiers	9	13.5	Décorateurs en Chaîne	18
9.1	Création d'un Qualifier	10	14	Événements	18
			14.1	Principe	18

14.2	Émission d'Événements	18
14.3	Observation Synchrone	18
14.4	Observation Asynchrone	19
14.5	Événements Qualifiés	19
14.6	Événements Conditionnels	20
14.7	Transaction Phases (Java EE)	20
15	Stereotypes	20
15.1	Création d'un Stereotype	20
15.2	Utilisation d'un Stereotype	20
15.3	Stereotypes Avancés	20
15.4	Stereotypes Prédéfinis	21
16	Implémentations CDI	21
16.1	Weld - Implémentation de Référence	21
16.1.a	Weld dans Java SE	21
16.1.b	Weld dans Jakarta EE	22
16.2	Quarkus CDI (ArC)	22
17	Résumé	23
17.1	Concepts Fondamentaux	23
17.2	Gestion des Dépendances	23
17.3	Production de Beans	23
17.4	Programmation Orientée Aspect ..	24
17.5	Concepts Avancés	24
17.6	Implémentations	24
18	Bonnes Pratiques	24
18.1	1. Préférez l'Injection par Constructeur	24
18.2	2. Utilisez les Scopes Appropriés ..	24
18.3	3. Nommez les Qualifiers de Manière Descriptive	24
18.4	4. Utilisez <code>@Alternative</code> pour les Tests	25
18.5	5. Gérez les Ressources avec Producer/Disposer	25
18.6	6. Évitez les Cycles de Dépendances	25
18.7	7. Documentez les Qualifiers et Stereotypes	25
19	Ressources	25
19.1	Documentation Officielle	25
19.2	Exemples	25
19.3	Tutoriels	25
19.4	Outils	25

1 Introduction à CDI (Contexts and Dependency Injection)

La spécification **CDI (Contexts and Dependency Injection)**, au cœur de *Jakarta EE*, fournit un cadre puissant pour la gestion déclarative des dépendances et des cycles de vie des objets au sein des applications Java.

1.1 Objectifs de CDI

- **Injection de Dépendances** : Éliminer le besoin de code de configuration explicite en permettant l'injection automatique de dépendances entre composants.
- **Gestion des Contextes** : Contrôler finement le cycle de vie des objets selon leur portée (application, session, requête, etc.).
- **Programmation Orientée Aspect** : Ajouter des fonctionnalités transversales (logging, sécurité, transactions) via intercepteurs et décorateurs.
- **Communication Découplée** : Permettre aux composants de communiquer via des événements sans couplage direct.

Principe Fondamental : Couplage faible avec un typage fort

Un exemple de code simple utilisant CDI : <https://github.com/ebpro/sample-cdi>

2 Problèmes Résolus par CDI

CDI résout des problèmes récurrents dans le développement d'applications Java :

Sans CDI

- **Accès aux objets** : Références manuelles, couplage fort entre composants
- **Gestion du cycle de vie** : Création/destruction manuelle des objets
- **Multi-threading** : Gestion complexe de la concurrence

- **Mémoire** : Risque de fuites mémoire par gestion manuelle des références
- **Implémentations alternatives** : Configuration manuelle et rigide au déploiement
- **Partage d'objets** : Gestion manuelle des références partagées

Avec CDI

- **Accès aux objets** : Injection automatique avec `@Inject`
- **Gestion du cycle de vie** : Scopes automatiques (`@RequestScoped`, `@ApplicationScoped`, etc.)
- **Multi-threading** : Gestion automatique par les contextes
- **Mémoire** : Destruction automatique selon le scope
- **Implémentations alternatives** : Qualifiers (`@Qualifier`) et alternatives (`@Alternative`)
- **Partage d'objets** : Scopes facilitent le partage sécurisé

2.1 Avantages Clés

- ☑ **Faible couplage** : Composants indépendants et interchangeables
- ☑ **Testabilité** : Injection de mocks/stubs simplifiée
- ☑ **Extensibilité** : Ajout de fonctionnalités sans modifier le code source
- ☑ **Maintenabilité** : Code plus clair et mieux organisé

3 Historique de CDI

Avant CDI

- **EJB (Enterprise JavaBeans)**
 - Premières versions Java EE
 - Gestion des composants d'entreprise
 - ✗ Complexe et lourd (XML verbeux)
- **Spring Framework (2004)**
 - Alternative légère à EJB
 - Injection de dépendances simplifiée
 - ☑ Adoption massive

Évolution CDI

- **2009 : CDI 1.0** (Java EE 6, JSR 299)
 - Simplification gestion dépendances
 - Inspiration de Spring mais standardisé
- **2013 : CDI 1.1** (Java EE 7)
 - Événements améliorés
 - Beans transactionnels
- **2017 : CDI 2.0** (Java EE 8)
 - Événements asynchrones (`@ObservesAsync`)
 - Support Java SE
- **2020 : CDI 3.0** (Jakarta EE 9)
 - Namespace `javax.*` → `jakarta.*`
- **2022 : CDI 4.0** (Jakarta EE 10)
 - CDI Lite (subset pour compilation native)
 - Build-compatible extensions

Impact : CDI a standardisé l'injection de dépendances dans l'écosystème Jakarta EE, offrant une alternative officielle à Spring.

4 Qu'est-ce qu'un Bean CDI ?

Un **Bean CDI** est un composant géré par le conteneur CDI qui respecte certaines règles et conventions.

4.1 Vue Conceptuelle

Un bean CDI est défini par un **contrat d'injection** comprenant :

1. **Un ensemble de types**
 - Classes et interfaces implémentées/étendues
 - Exemple : `class PayPalService implements PaymentService`
 - Permet l'injection par type : `@Inject PaymentService service;`
2. **Un ensemble de qualifiers**
 - Annotations distinguant les implémentations
 - Exemple : `@PayPal`, `@CreditCard`

- Permet l'injection spécifique : `@Inject @PayPal PaymentService service;`
3. **Un scope (contexte)**
 - Définit le cycle de vie du bean
 - Exemple : `@ApplicationScoped, @RequestScoped`
 - Gère automatiquement création/destruction
 4. **Une implémentation**
 - Code métier concret du bean
 5. **Optionnellement : un nom**
 - Permet la référence dans Expression Language (EL)
 - Exemple : `@Named("paypal")` → `#{paypal}` dans JSF
 6. **Optionnellement : des intercepteurs**
 - Comportements transversaux (logging, sécurité)

4.2 Vue Technique

CDI définit techniquement un bean comme :

☒ **Bean valide si :**

- Classe avec constructeur sans paramètre

```
public class MyService {
    public MyService() {}
}
```

- Classe avec constructeur annoté `@Inject`

```
public class MyService {
    @Inject
    public MyService(Logger logger) {}
}
```

- Produit par méthode `@Produces`

```
@Produces
public Logger createLogger() {
    return Logger.getLogger("app");
}
```

✗ **Pas un bean si :**

- Classe abstraite (sans `@Produces`)
- Interface (sans `@Produces`)
- Classe sans constructeur approprié
- Inner class non-static
- Classe avec constructeurs multiples non annotés

4.3 Résumé : Contrat d'un Bean CDI

```
Bean CDI = Types + Qualifiers + Scope
+ Implémentation
          [+ Nom EL] [+ Intercepteurs]
          [+ Alternative]
```

5 Scopes et Cycle de Vie

Les **scopes** définissent la durée de vie et la visibilité des beans CDI.

5.1 Scopes Principaux

Scope	Durée de vie	Cas d'usage	Exemple
@Dependency	Aucune durée que le bean consommateur	Bean utilitaire, pas de partage	Calculateur, validateur
@RequestScoped	Une requête HTTP	Données formulaire, requête REST	Formulaire JSF, DTO REST
@SessionScoped	Un utilisateur	Données utilisateur persistantes	Panier e-commerce, préférences
@ApplicationScoped	L'application entière	Configuration, serveur	Cache, confi-

Scope	Durée de vie	Cas d'usage	Exemple
		vices partagés	guration globale
<code>@ConversationScoped</code>	Procession utilisateur multi-requêtes	Proces- sus mé- tier multi-étapes	Tun- nel de commande, wizard

```
Serializable {
    private List<Item> items = new
    ArrayList<>();

    public void addItem(Item item) {
        items.add(item);
    }
}
```

5.2 Scope par Défaut

- Si aucun scope n'est spécifié, le bean est `@Dependent`
- Un bean `@Dependent` n'est jamais partagé, chaque injection crée une nouvelle instance

5.3 Exemple de Scopes

```
// Service partagé dans toute
// l'application
@ApplicationScoped
public class ConfigurationService {
    private Properties config;

    @PostConstruct
    public void init() {
        config = loadConfiguration();
    }
}

// Bean lié à une requête HTTP
@RequestScoped
public class OrderFormBean {
    private String productId;
    private int quantity;
    // getters/setters
}

// Bean lié à la session utilisateur
@SessionScoped
public class ShoppingCart implements
```

5.4 Passivation et Activation

i Note

Les beans `@SessionScoped` et `@ConversationScoped` peuvent être **passivés** (sérialisés sur disque) pour économiser la mémoire, puis **activés** (désérialisés) à la demande.

Prérequis : Le bean doit implémenter `Serializable`

```
@SessionScoped
public class UserSession
implements Serializable {

    @PrePassivate
    public void
    beforePassivation() {
        // Nettoyage avant
        // sérialisation (fermer connexions,
        // etc.)
    }

    @PostActivate
    public void afterActivation() {
        // Restauration après
        // désérialisation
    }
}
```

6 Injection de Dépendances

L'injection de dépendances est le mécanisme central de CDI, permettant de fournir automatiquement les dépendances d'un bean.

6.1 Types d'Injection

1. Injection par Champ (Field)

```
@ApplicationScoped
public class OrderService {
    @Inject
    private PaymentService
    paymentService;

    @Inject
    private Logger logger;
}
```

- ☒ Syntaxe concise
- ✗ Difficile à tester (pas d'accès direct)
- ✗ Dépendances non finales

2. Injection par Constructeur

```
@ApplicationScoped
public class OrderService {
    private final PaymentService
    paymentService;
    private final Logger logger;

    @Inject
    public OrderService(
        PaymentService paymentService,
        Logger logger) {
        this.paymentService =
        paymentService;
        this.logger = logger;
    }
}
```

- ☒ Dépendances finales (immutabilité)
- ☒ Facilite les tests (injection manuelle)
- ☒ Dépendances explicites
- ☒ **Recommandé**

3. Injection par Setter

```
@ApplicationScoped
public class OrderService {
    private PaymentService
    paymentService;

    @Inject
    public void
    setPaymentService(PaymentService
    paymentService) {
        this.paymentService =
        paymentService;
    }
}
```

- ☒ Dépendances optionnelles
- ✗ Bean peut être dans un état incomplet
- ⚠ Rarement utilisé

6.2 Annotation @Named

L'annotation `@Named` permet de donner un nom à un bean pour le référencer dans Expression Language (EL), notamment dans les pages JSF.

```
@Named("cart") // Accessible via
#{cart} dans JSF
@SessionScoped
public class ShoppingCartBean
implements Serializable {
    private List<Item> items = new
    ArrayList<>();

    public void addItem(Item item) {
        items.add(item);
    }

    public List<Item> getItems() {
        return items;
    }
}
```

Utilisation dans JSF :

```
<h:dataTable value="#{cart.items}"
var="item">
    <h:column>#{item.name}</h:column>
</h:dataTable>
```

Nom par défaut :

- Si `@Named` sans valeur : nom = nom de la classe avec première lettre minuscule
- `@Named` sur `ShoppingCartBean` → nom = `shoppingCartBean`

6.3 Injection par Type

CDI injecte par défaut en se basant sur le **type** de l'attribut ou du paramètre.

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

@ApplicationScoped
public class
CreditCardPaymentService implements
PaymentService {
    @Override
    public void processPayment(double
amount) {
        System.out.println("Paielement
par carte : " + amount);
    }
}

@ApplicationScoped
public class OrderService {
    @Inject
    private PaymentService
paymentService; // Type =
PaymentService

    public void checkout(double
amount) {
        paymentService.processPayment(amount)
    }
}
```

```
}
}
```

Règle CDI : Le conteneur cherche un bean dont le type correspond à `PaymentService`

6.4 Ce qui est Injecté

Lorsqu'on injecte une dépendance avec `@Inject`, CDI recherche un bean correspondant au **contrat d'injection** :

```
@Inject
@PayPal // Qualifier
private PaymentService
paymentService; // Type
```

Contrat d'injection = Type + Qualifiers

Le bean injecté doit : 1. ☑ Implémenter ou étendre le type spécifié (`PaymentService`) 2. ☑ Posséder tous les qualifiers spécifiés (`@PayPal`) 3. ☑ Avoir un scope approprié 4. ☑ Être activé (pas désactivé par `@Alternative` non activé)

Cas particulier : Si aucun qualifier n'est spécifié - CDI utilise le qualifier implicite `@Default` - `@Inject private PaymentService service;` = `@Inject @Default private PaymentService service;`

6.5 Initializer Methods

Les **initializer methods** sont des méthodes annotées `@Inject` permettant d'initialiser un bean après sa création, avec injection de dépendances dans les paramètres.

```
@ApplicationScoped
public class ReportService {
    private Logger logger;
    private Database database;

    // Méthode d'initialisation avec
injection
    @Inject
```

```

    public void initialize(Logger
logger, Database database) {
        this.logger = logger;
        this.database = database;
        logger.info("ReportService
initialized");
    }
}

```

Différence avec @PostConstruct :

- @Inject sur méthode : Permet l'injection de paramètres
- @PostConstruct : Callback après injection complète, pas de paramètres injectables

Ordre d'exécution :

1. Constructeur
2. Injection des champs (@Inject sur fields)
3. Initializer methods (@Inject sur méthodes)
4. @PostConstruct

7 Exemple Progressif

Construisons progressivement une application de gestion de commandes.

7.1 Étape 1 : Bean Simple

```

@ApplicationScoped
public class Logger {
    public void log(String message) {
        System.out.println("[LOG] "
+ message);
    }
}

@ApplicationScoped
public class OrderService {
    @Inject
    private Logger logger;

    public void createOrder(String
productId) {
        logger.log("Commande créée

```

```

pour : " + productId);
    }
}

```

7.2 Étape 2 : Avec Interface

```

public interface NotificationService
{
    void notify(String message);
}

@ApplicationScoped
public class EmailNotificationService implements
NotificationService {
    @Override
    public void notify(String message)
    {
        System.out.println("[] Email :
" + message);
    }
}

@ApplicationScoped
public class OrderService {
    private final Logger logger;
    private final NotificationService
notificationService;

    @Inject
    public OrderService(Logger
logger, NotificationService
notificationService) {
        this.logger = logger;
        this.notificationService =
notificationService;
    }

    public void createOrder(String
productId) {
        logger.log("Commande créée
pour : " + productId);
        notificationService.notify("Votre

```

```

    commande est confirmée");
    }
}

```

7.3 Étape 3 : Avec Scope Approprié

```

@RequestScoped
public class OrderFormBean {
    @Inject
    private OrderService
    orderService; // @ApplicationScoped
    injecté

    private String productId;
    private int quantity;

    public void submitOrder() {
        orderService.createOrder(productId);
    }

    // getters/setters
}

```

```

    }
}

@ApplicationScoped
public class
CreditCardPaymentService implements
PaymentService {
    @Override
    public void processPayment(double
    amount) {
        System.out.println("💳 Paiement
        Carte : " + amount);
    }
}

@ApplicationScoped
public class OrderService {
    @Inject
    private PaymentService
    paymentService; // ✕ ERREUR :
    Ambiguïté!
}

```

8 Conflits de Dépendances

Que se passe-t-il si **plusieurs implémentations** d'une même interface existent ?

8.1 Problème : Ambiguïté

```

public interface PaymentService {
    void processPayment(double
    amount);
}

@ApplicationScoped
public class PayPalPaymentService
implements PaymentService {
    @Override
    public void processPayment(double
    amount) {
        System.out.println("💳 Paiement
        PayPal : " + amount);
    }
}

```

CDI lève une exception : `AmbiguousResolutionException`

- Deux beans correspondent au type `PaymentService`
- CDI ne sait pas lequel injecter

8.2 Solutions

1. **Qualifiers** : Distinguer les implémentations (recommandé)
2. **Alternatives** : Activer/désactiver des implémentations via configuration
3. **Producers** : Contrôler finement la création

9 Qualifiers

Les **qualifiers** permettent de distinguer différentes implémentations d'une même interface.

9.1 Création d'un Qualifier

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

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

9.2 Application du Qualifier

```
@PayPal
@ApplicationScoped
public class PayPalPaymentService
implements PaymentService {
    @Override
    public void processPayment(double
amount) {
        System.out.println("💳 Paiement
PayPal : " + amount);
    }
}

@CreditCard
@ApplicationScoped
public class CreditCardPaymentService implements
PaymentService {
    @Override
    public void processPayment(double
amount) {
        System.out.println("💳 Paiement
Carte : " + amount);
    }
}
```

9.3 Injection Qualifiée

```
@ApplicationScoped
public class OrderService {
    @Inject
    @PayPal
    private PaymentService
paypalService;

    @Inject
    @CreditCard
    private PaymentService
creditCardService;

    public void processPayment(String
method, double amount) {
        if ("paypal".equals(method)) {
            paypalService.processPayment(amount);
        } else {
            creditCardService.processPayment(amount);
        }
    }
}
```

9.4 Qualifier @Default

Lorsqu'aucun qualifier n'est spécifié, CDI utilise le qualifier implicite **@Default**.

```
// Ces deux injections sont
équivalentes :

@Inject
private PaymentService service;

@Inject
@Default
private PaymentService service;
```

Règle importante :

- Si un bean a un qualifier personnalisé (@PayPal), il n'a **pas** @Default
- Pour qu'un bean ait @Default, il ne doit avoir aucun qualifier explicite

```

@ApplicationScoped           // A
implicitement @Default
public class DefaultPaymentService
implements PaymentService {
    // ...
}

@PayPal
@ApplicationScoped           // N'a PAS
@Default
public class PayPalPaymentService
implements PaymentService {
    // ...
}

```

9.5 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;
}

```

9.6 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!
}

```

10 Beans Alternatifs

Les **beans alternatifs** permettent de remplacer une implémentation par une autre selon

l'environnement (développement, test, production) **sans modifier le code source**.

10.1 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]
Paielement simulé : " + amount);
    }
}
```

10.2 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

10.3 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

10.4 Cas d'Usage

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

11 Producers

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

11.1 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
    }
}
```

11.2 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
}
```

11.3 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
}
```

11.4 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();
        }
    }
}

```

11.5 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

12 Intercepteurs

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

12.1 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

12.2 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)

```

12.3 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, tran-

Annotation	Point d'interception	Usage
		sac-tions
@PreDestroy	Avant destruction	Nettoyage
@AroundTimeout	Autour d'un timeout (EJB)	Ges-tion timeouts

12.4 Exemple : Intercepteur de Transaction

```

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

@Interceptor
@Transactional
@Priority(Interceptor.Priority.PLATFORM_BEFORE
+ 200)
public class TransactionInterceptor {

    @Inject
    private EntityManager em;

    @AroundInvoke
    public Object
manageTransaction(InvocationContext
ctx) throws Exception {
        EntityTransaction tx =
em.getTransaction();
        boolean newTx = !tx.isActive();

        if (newTx) {
            tx.begin();
        }

        try {
            Object result =
ctx.proceed();
            if (newTx) {

```

```

        tx.commit();
    }
    return result;
} catch (Exception e) {
    if (newTx && tx.isActive()) {
        tx.rollback();
    }
    throw e;
}
}
}

```

12.5 Intercepteurs Multiples

Plusieurs intercepteurs peuvent être appliqués :

```

@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)

13 Décorateurs

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

13.1 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
Implémentée 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

13.2 Création d'un Décorateur

```

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;
    }
}

```

13.3 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
}

```

13.4 Exemple : Décorateur de Vali- dation

```

@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);
    }
}

```

13.5 Décorateurs en Chaîne

Plusieurs décorateurs peuvent être chaînés :

```

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

```

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

14 Événements

Les **événements** permettent une communication découplée entre composants : un émetteur envoie un événement, des observateurs réagissent, sans que l'émetteur connaisse les observateurs.

14.1 Principe

```

Émetteur    →    Event<T>    →    CDI
Container → @Observes/@ObservesAsync
→ Observateurs

```

Avantages : - ☒ Couplage faible (émetteur ≠ observateurs) - ☒ Extensibilité (ajouter observateurs sans modifier émetteur) - ☒ Communication asynchrone possible

14.2 Émission d'Événements

```

// Classe d'événement (POJO)
public class OrderCreatedEvent {
    private final String orderId;
    private final double amount;

    public OrderCreatedEvent(String
orderId, double amount) {
        this.orderId = orderId;
        this.amount = amount;
    }

    // getters
}

@ApplicationScoped
public class OrderService {

    @Inject
    private Event<OrderCreatedEvent>
orderEvent;

    public void createOrder(String
orderId, double amount) {
        // Logique métier
        System.out.println("Création
commande " + orderId);

        // Émission événement
        orderEvent.fire(new
OrderCreatedEvent(orderId, amount));
    }
}

```

14.3 Observation Synchrone

Les observateurs synchrones sont invoqués dans le **même thread** que l'émetteur :

```

@ApplicationScoped
public class
EmailNotificationService {

```

```

        public
        void onOrderCreated(@Observes
        OrderCreatedEvent event) {
            System.out.println("[]
        Envoi email pour commande " +
        event.getOrderId());
        }
    }

    @ApplicationScoped
    public class InventoryService {

        public
        void onOrderCreated(@Observes
        OrderCreatedEvent event) {
            System.out.println("📦
        Mise à jour stock pour commande " +
        event.getOrderId());
        }
    }

```

Résultat :

```

Création commande ORD-123
[] Envoi email pour commande ORD-123
📦 Mise à jour stock pour commande
ORD-123

```

⚠ **Attention** : Si un observateur lève une exception, les suivants ne sont pas exécutés.

14.4 Observation Asynchrone

Les observateurs asynchrones sont invoqués dans un **thread séparé** :

```

@ApplicationScoped
public class AnalyticsService {

        public
        void onOrderCreated(@ObservesAsync
        OrderCreatedEvent event) {
            // Traitement long (envoi
        analytics)
    }

```

```

        System.out.println("[]
        Traitement analytics (thread
        séparé)");
    }
}

```

Émission asynchrone :

```

@Inject
private Event<OrderCreatedEvent>
orderEvent;

public void createOrder(String
orderId, double amount) {
    // Émission asynchrone
    orderEvent.fireAsync(new
    OrderCreatedEvent(orderId, amount));

    // Continue immédiatement (ne
    bloque pas)
    System.out.println("Commande
    créée, traitement en cours en
    arrière-plan");
}

```

14.5 Événements Qualifiés

Les événements peuvent être qualifiés pour filtrer les observateurs :

```

@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!");
}

```

14.6 Événements Conditionnels

Les observateurs peuvent filtrer les événements :

```

@ApplicationScoped
public class LargeOrderService {

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

```

14.7 Transaction Phases (Java EE)

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

```

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
}

```

15 Stereotypes

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

15.1 Création d'un Stereotype

```

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

```

15.2 Utilisation d'un Stereotype

Au lieu de :

```

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

```

On peut écrire :

```

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

```

15.3 Stereotypes Avancés

Les stereotypes peuvent inclure interceptors :

```

@Stereotype
@ApplicationScoped
@Transactional
@Logged
@Retention(RUNTIME)
@Target(TYPE)
public @interface Service {}

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

```

15.4 Stereotypes Prédéfinis

CDI définit quelques stereotypes standards :

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

```

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

```

16 Implémentations CDI

16.1 Weld - Implémentation de Référence

Weld est l'implémentation de référence de la spécification CDI, développée par Red Hat.

Fonctionnalités : - ☑ Implémentation complète de CDI 4.0+ - ☑ Support Java SE et Jakarta EE - ☑ Utilisé par WildFly, Payara, GlassFish

16.1.a Weld dans Java SE

```

<!-- pom.xml -->
<dependency>

```

```

<groupId>org.jboss.weld.se</groupId>
<artifactId>weld-se-core</artifactId>
<version>5.1.0.Final</version>
</dependency>

```

Fichier beans.xml dans META-INF/ :

```

<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="all">
</beans>

```

Point d'entrée :

```

public class Main {

    public static void main(String[] args) {
        // Démarrage conteneur Weld
        try (SeContainer container = SeContainerInitializer.newInstance().getSeContainer()) {

            // Récupération bean
            OrderService orderService = container.select(OrderService.class).get();

            orderService.createOrder("ORD-123", 99.99);

        }
    }
}

```

Avec écouteur d'événement :

```

public class AppStartup {

```

```

@Inject
private OrderService orderService;

public void onStartUp(@Observes
ContainerInitialized event) {
    System.out.println("[]
Application démarrée");
    orderService.createOrder("STARTUP-001"
0.0);
}
}

```

16.1.b Weld dans Jakarta EE

- **Intégration automatique** dans WildFly, Payara, Open Liberty, GlassFish
- **beans.xml** dans `WEB-INF/` (applications web) ou `META-INF/` (modules EJB)
- **Aucun code de démarrage** nécessaire (géré par le serveur)

```

<!-- WEB-INF/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">
</beans>

```

Bean Discovery Modes :

- **all** : Tous les types sont des beans (même sans annotations)
- **annotated** : Seulement les classes avec annotations CDI (recommandé)
- **none** : CDI désactivé

16.2 Quarkus CDI (ArC)

ArC (Quarkus DI) est une implémentation CDI optimisée pour Quarkus, axée sur la performance et la compilation native.

Différences avec Weld :

Aspect	Weld (Jakarta EE)	ArC (Quarkus)
Initialisation	Runtime (réflexion)	Build-time (code généré)
beans.xml requis (ou annotations)	Requis (ou annotations)	✗ Pas nécessaire
Performance	Standard	↗ Optimisée
Compilation native	△ Limitée	☑ Complète (GraalVM)
Scope support	Tous les scopes	Subset (pas de <code>@ConversationScope</code>)
Découverte beans	Runtime	Build-time

Configuration Quarkus :

```

# application.properties

# Pas de beans.xml nécessaire!
# CDI activé par défaut

# Pour activer tous les beans (équivalent bean-discovery-mode="all")
quarkus.arc.unremovable-types=**

```

```
# Logs CDI (debug)
quarkus.log.category."io.quarkus.arc".level=DEBUG
```

Exemple Quarkus :

```
@ApplicationScoped
public class GreetingService {
    public String greet(String name) {
        return "Hello, " + name;
    }
}

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

    @Inject
    GreetingService greetingService;

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return
greetingService.greet("Quarkus");
    }
}
```

Dev Mode Quarkus :

```
mvn quarkus:dev
# Hot reload instantané, pas de
redémarrage!
```

Optimisations ArC : -  **Build-time DI** : Résolution des dépendances à la compilation -  **Code generation** : Pas de réflexion, code natif généré -  **Removal unused beans** : Beans non utilisés supprimés automatiquement -  **Fast startup** : <0.1s avec GraalVM native

Limitations ArC : -  Pas de `@ConversationScoped` (pas de notion de conversation en microservices) -  Pas de

Portable Extensions (remplacé par Build Compatible Extensions) -  Decorators limités

Quand utiliser Quarkus CDI ? -  Microservices cloud-native -  Performance critique (startup, mémoire) -  Compilation native GraalVM -  Containers/Kubernetes

Quand utiliser Weld ? -  Applications Jakarta EE complètes -  Besoin de tous les scopes CDI -  Portable Extensions -  Compatibilité maximale

17 Résumé

17.1 Concepts Fondamentaux

- **Bean CDI** = Types + Qualifiers + Scope + Implémentation
- **Injection** = `@Inject` (field, constructor, setter)
- **Scopes** = `@ApplicationScoped`, `@RequestScoped`, `@SessionScoped`, `@Dependent`

17.2 Gestion des Dépendances

- **Qualifiers** : Distinguer les implémentations (`@Qualifier`)
- **@Default** : Qualifier implicite si aucun qualifier personnalisé
- **@Named** : Nommer un bean pour EL (JSF)
- **Alternatives** : Remplacer implémentations selon environnement (`@Alternative`)

17.3 Production de Beans

- **Producers** : Créer beans programmatiquement (`@Produces`)
- **Producer methods** : Logique de création complexe
- **Producer fields** : Cas simples
- **Disposers** : Nettoyer ressources (`@Disposes`)

17.4 Programmation Orientée Aspect

- **Intercepteurs** : Ajouter comportements transversaux (@Interceptor, @AroundInvoke)
- **Décorateurs** : Étendre comportement métier (@Decorator, @Delegate)
- **Événements** : Communication découplée (@Observes, @ObservesAsync)

17.5 Concepts Avancés

- **Stereotypes** : Combiner annotations (@Stereotype)
- **Initializer methods** : Initialisation avec injection (@Inject sur méthode)

17.6 Implémentations

- **Weld** : Implémentation de référence, complète, Java SE + Jakarta EE
- **Quarkus CDI (ArC)** : Optimisée, build-time, compilation native

18 Bonnes Pratiques

18.1 1. Préférez l'Injection par Constructeur

```
// ☐ BON
@ApplicationScoped
public class OrderService {
    private final PaymentService
    paymentService;

    @Inject
    public
    OrderService(PaymentService
    paymentService) {
        this.paymentService =
        paymentService;
    }
}

// ✕ ÉVITER
```

```
@ApplicationScoped
public class OrderService {
    @Inject
    private PaymentService
    paymentService; // Pas final,
    difficile à tester
}
```

18.2 2. Utilisez les Scopes Appropriés

```
// ☐ BON : Service partagé =
@ApplicationScoped
@ApplicationScoped
public class ConfigurationService { }

// ☐ BON : Données requête =
@RequestScoped
@RequestScoped
public class OrderFormBean { }

// ✕ ÉVITER : Service en
@RequestScoped (recréé à chaque
requête!)
@RequestScoped
public class EmailService { } //
Devrait être @ApplicationScoped
```

18.3 3. Nommez les Qualifiers de Manière Descriptive

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

// ✕ ÉVITER
@Qualifier
@Retention(RUNTIME)
@Target({FIELD, TYPE, METHOD})
public @interface Q1 { } // Nom peu
clair
```

18.4 4. Utilisez @Alternative pour les Tests

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

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

18.5 5. Gérez les Ressources avec Producer/Disposer

```
@ApplicationScoped
public class DatabaseProducer {

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

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

18.6 6. Évitez les Cycles de Dépendances

```
// ✗ ERREUR : Cycle A → B → A
@ApplicationScoped
public class ServiceA {
    @Inject
    private ServiceB serviceB; // B
```

```
dépend de A!
}
```

```
// ☐ SOLUTION : Refactorer ou utiliser
@Produces
```

18.7 7. 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 { }
```

19 Ressources

19.1 Documentation Officielle

- [Jakarta CDI Specification](#)
- [Weld Documentation](#)
- [Quarkus CDI Guide](#)

19.2 Exemples

- [Sample CDI Project](#)
- [Quarkus Quickstarts](#)

19.3 Tutoriels

- [Jakarta EE Tutorial - CDI](#)
- [Quarkus CDI Extensions](#)

19.4 Outils

- [CDI TCK - Test Compatibility Kit](#)
- [Arquillian - Testing CDI applications](#)