

CDI

EMSI - Université Côte d'Azur
Richard Grin
Version 2.6 – 2/11/23

Plan du support

- Injection de dépendance – CDI
- Remplacement des EJB
- Exécution d'une méthode au démarrage ou à l'arrêt d'une application
- Gestion des transactions
- Transactions et exceptions
- Appel asynchrone de méthode

Richard Grin

CDI

page 2

1

2

Injection de dépendance

-
CDI

(Contexts and Dependency Injection)

Richard Grin

CDI

page 3

Définition

- Au lieu de créer un objet, on peut demander à l'environnement d'exécution (container) de le fournir, de l'« injecter »

Richard Grin

CDI

page 4

3

4

Injection CDI

- La spécification CDI permet d'injecter des beans avec l'annotation `@Inject`
- Les objets injectés par CDI ont une portée

Richard Grin

CDI

page 5

Autres injections de Jakarta EE

- La spécification CDI a été introduite tardivement dans Java EE 6
- Il reste encore dans Jakarta EE des injections qui sont définies par d'autres spécifications (EJB, JPA,...), en particulier l'injection de ressource liée à l'annotation `@Resource`
- Un des objectifs des futures versions de Jakarta EE est de ne garder que CDI pour les injections

Richard Grin

CDI

page 6

5

6

Injection de ressource (non CDI)

- @Resource injecte une source de données, mais aussi d'autres types de ressources comme un serveur d'emails

Exemples :

```
@Resource(lookup="jdbc/mabd")  
private DataSource source;
```

```
@Resource(lookup="java:app/mail/free")  
private Session sessionEmail;
```

A quoi peuvent correspondre les valeurs de lookup ?

Où sont définis les objets qui correspondent à ces noms JNDI ?

7

Injection avec CDI - @Inject

@Inject

```
A a;
```

fournit une instance d'une classe qui a le type A ;
A est une classe ou une interface Java

- IMPORTANT : Si une instance de type A existe déjà (voir « portée »), elle est fournie au code
- Sinon, une nouvelle instance est créée par le container CDI, et fournie au code
- Si plusieurs classes de type A, ou aucune classe, une exception est lancée au déploiement ; un « *qualifier* » permet de lever l'ambiguïté

8

Bean CDI

- Un bean CDI est une instance Java gérée par CDI (pour la création, injection, suppression)
- Instance Java injectée par CDI ou dans laquelle une injection CDI est faite
- Injection au déploiement, selon le mode de découverte indiqué dans beans.xml
- Le mode de découverte par défaut est annotated, ce qui signifie que le type d'un bean CDI doit être annoté, essentiellement avec @Named ou une des portées CDI

9

Types dans lesquelles on peut faire une injection CDI

- Presque toutes les classes Java connues de CDI (annotées pour une portée CDI, EJB, Servlet,...)

10

Types injectables par CDI ?

- Presque toutes les classes Java non abstraites (pas une classe interne non static)
- La classe doit avoir un constructeur sans paramètre (ou annoté par @Inject) qui n'est pas private

11

Portée CDI

- Un objet injecté a une portée (requête, session,...) qui définit la durée de vie de l'objet
- Par exemple, un objet injecté, de portée « requête »
 - est créé pendant le traitement d'une requête HTTP d'un client, la 1^{ère} fois qu'il est utilisé
 - est détruit quand la réponse à la requête est envoyée au client détruit par qui ?
- La portée est définie par une annotation de la classe de l'objet injecté

12

Portées CDI

- Requête HTTP : `@RequestScoped`
- Vue : `@ViewScoped` ; utilisateur reste sur la même page
- Session : `@SessionScoped`
- Application : `@ApplicationScoped`
- Conversation : `@ConversationScoped` ; portée définie pendant l'exécution du code (`begin()` pour commencer, `end()` pour finir)
- Flot : `@FlowScoped` ; ensemble de pages JSF
- Dépendant (portée par défaut) : `@Dependent` ; une nouvelle instance est toujours injectée

Richard Grin

CDI

page 13

13

Exemple

```
public class Ordinateur {
    @Inject
    private Imprimante imprimante;
    // Injection

    public void imprimer(Document document) {
        imprimante.imprimer(document);
    }
    // Utilisation
}

@RequestScoped
public class Imprimante {
    ...
    public void imprimer(Document document) { ... }
}
// Pourquoi ?
```

imprimante supprimée à la fin du traitement de la requête

Richard Grin

CDI

page 14

14

Utilité de la portée CDI

- Le bean importé est automatiquement supprimé à la fin de sa portée et donc n'encombre pas la mémoire du serveur
- Plusieurs classes (ou pages JSF) peuvent partager les informations contenues dans un bean injecté tant que sa portée est en cours

Richard Grin

CDI

page 15

15

Exemple

```
public class A {
    @Inject
    private Bean bean;
    public void m1(int n) { bean.setP(n); }
    ...
}

public class B {
    @Inject
    private Bean bean;
    private int v;
    public void m2() { this.v = bean.getP(); }
    ...
}

@RequestScope
public class Bean {
    ... // getP et setP
}
```

- Si les classes A et B interviennent dans cet ordre pour traiter une même requête, la valeur mise par A dans la propriété P pourra être récupérée dans B

Richard Grin

CDI

page 16

16

Création d'un bean CDI

1. Le container crée le bean avec le constructeur sans paramètre
 2. Ensuite, si le bean dépend d'autres beans CDI, CDI les injecte
- Donc, on ne peut pas utiliser dans le constructeur un bean injecté

Pourquoi ?

Quelle erreur aurait-on ?

Richard Grin

CDI

page 17

17

@PostConstruct

- Solution pour initialiser un bean CDI avec des objets injectés : écrire dans le bean une méthode annotée par `@PostConstruct`
- Cette méthode sera appelée par le container juste après le constructeur du bean et *après que toutes les injections ont été faites*
- A retenir : Si un bean doit utiliser des instances injectées pendant son initialisation, il faut utiliser ces instances dans une méthode annotée par `@PostConstruct`, pas dans un constructeur

Richard Grin

CDI

page 18

18

Exemple - Erreur

```
@RequestScope
public class Bean1 {
    @Inject
    private Bean2 autreBean;

    public Bean1(){
        // Initialisation qui utilise autreBean
        ...
        autreBean.m(); // NullPointerException !
    }
}
```

Richard Grin

CDI

page 19

19

Exemple - Solution

```
@RequestScope
public class Bean1 {
    @Inject
    private Bean2 autreBean;

    @PostConstruct
    public void init(){
        // Initialisation qui utilise autreBean
        ...
        autreBean.m();
    }
}
```

Richard Grin

CDI

page 20

20

@PreDestroy

- Peut annoter une méthode d'un bean CDI
- Cette méthode « fera le ménage » avant la suppression du bean

Qui va supprimer le bean ?

Qui va lancer l'exécution de la méthode annotée par @PreDestroy ?

Richard Grin

CDI

page 21

21

Architecture application Web JSF

- Les pages JSF (*Jakarta Faces*) se chargent de l'interface avec l'utilisateur
- Des beans CDI (les *backing beans*), injectés dans les pages JSF, aident au bon fonctionnement des pages
- Les backing beans délèguent à d'autres classes tout ce qui n'est pas lié directement aux pages JSF ; en particulier les traitements métier et les interfaces avec les bases de données sont souvent délégués à des beans CDI
- Les interfaces avec les bases de données utilisent des classes Java entités, si JPA

Richard Grin

CDI

page 22

22

Un peu plus loin avec CDI

- Pour lever l'ambiguïté lors d'une injection, un « qualifier », annotation spéciale, peut être utilisé
- Des méthodes productrices peuvent être utilisées pour injecter des valeurs qui ne sont pas des beans CDI ou pour configurer les instances injectées en tenant compte du point d'injection
- Les intercepteurs CDI peuvent intervenir juste avant ou juste après l'exécution d'une méthode d'un bean CDI

Richard Grin

CDI

page 23

23

Exemple d'ambiguïté

```
□ // Injection dans une classe
@Inject
private Salut salut; Instance de quelle classe sera injectée ?
...
// Dans le code d'une méthode :
String s = salut.bonjour();

□ public class SalutFormel implements Salut {
    public String bonjour() { return "Bonjour"; }
    ...
}

□ public class SalutAmical implements Salut {
    public String bonjour() { return "Hello"; }
    ...
}
```

Richard Grin

CDI

page 24

24

Lever l'ambiguïté avec un Qualifier

- // Injection dans une classe
@Inject
@Formel
private Salut salut;
...
// Dans le code d'une méthode :
String s = salut.bonjour();
- @Formel
public class SalutFormel implements Salut;

25

Exemple définition d'un qualifieur

```
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.FIELD, ElementType.TYPE,
        ElementType.PARAMETER})
public @interface Formel {
}
```

26

Remplacement des EJBs

27

EJB

- Un EJB est un type particulier de composant de Jakarta EE
- Correspond à des classes Java dont les instances sont gérées par un container d'EJB
- Existents depuis les 1^{ères} versions de Java EE, et sont utilisés pour exécuter des processus métiers et des accès aux bases de données
- Le container d'EJB permet d'offrir de nombreux services aux développeurs

28

Services offerts par un container d'EJB

- Gestion du cycle de vie : création, suppression des EJB
- Gestion des transactions : démarrer et terminer des transactions
- Gestion des threads : création, appels asynchrones de méthodes
- Gestion de la concurrence : protéger un EJB contre des accès concurrents
- Gestion de la sécurité : seuls les clients autorisés peuvent appeler les méthodes des EJB

29

Avantages

- Le développeur peut se concentrer sur les traitements « métier »
- Il n'a (presque) plus à s'occuper des problèmes non fonctionnels liés aux transactions, à la concurrence, à la sécurité,...

30

Remplacement des EJBs

- A ce stade, il existe peu de services rendus par l'utilisation des EJB, qui ne peuvent pas être rendus par l'utilisation des beans CDI
- L'injection d'un bean CDI est plus souple et plus puissante que l'injection d'un EJB
- Un objectif pour Jakarta EE est de retirer la spécification EJB de Jakarta EE (pas pour demain...); chacun des services offerts par les EJBs doit donc être rendu par une autre spécification de Jakarta EE, et en particulier par CDI

Richard Grin

CDI

page 31

31

Exemples de remplacement

- Nous allons étudier en particulier comment
 - Les beans CDI peuvent fournir une gestion simple des transactions
 - Lancer l'exécution d'une méthode au démarrage ou à l'arrêt d'une application
 - Une méthode d'un bean CDI peut être lancée en parallèle au fil d'exécution principal (appel asynchrone)

Richard Grin

CDI

page 32

32

Gestion des transactions

Richard Grin

CDI

page 33

33

2 possibilités pour les transactions

- Dans Jakarta EE la gestion des transactions peut être faite
 - par le code des classes, en utilisant l'interface `UserTransaction`
 - par le serveur d'application, en utilisant `@Transactional`

Richard Grin

CDI

page 34

34

Contraintes de la gestion par le serveur

- Une transaction doit
 - commencer au début d'une méthode `public`
 - se terminer à la fin de la méthode qui a commencé la transaction
- Le plus souvent ces contraintes ne sont pas gênantes et la gestion par le serveur d'application est choisie car elle est plus simple

Richard Grin

CDI

page 35

35

UserTransaction

- Pour les rares cas où on doit
 - mettre la transaction dans une méthode non `public`
 - commencer ou terminer une transaction au milieu d'une méthode
 - terminer une transaction dans une autre méthode que la méthode qui démarré la transaction

il faut injecter une instance de l'interface `jakarta.transaction.UserTransaction`

Richard Grin

CDI

page 36

36

Utilisation de UserTransaction

- Injecter dans une variable une instance de type UserTransaction

```
@Resource  
UserTransaction tx;
```

- Dans le code d'une méthode, début transaction
tx.begin()
- Fin de transaction
tx.commit() ou tx.rollback()

Richard Grin

CDI

page 37

37

Exemple

```
@RequestScoped  
public class MaClasse {  
    @Resource  
    private UserTransaction userTransaction;  
    public void performTransaction() {  
        try {  
            userTransaction.begin();  
            ... // Actions transactionnelles  
            userTransaction.commit();  
        } catch (Exception e) {  
            try {  
                userTransaction.rollback();  
            } catch (Exception rollbackException) { ... }  
        }  
    }  
}
```

Richard Grin

CDI

page 38

38

- **La suite de cette section illustre la gestion des transactions par le serveur d'application**

Richard Grin

CDI

page 39

39

@Transactional

- jakarta.transaction.Transactional annote une méthode public d'une classe gérée par Jakarta EE (bean CDI, servlet,...) pour indiquer comment la méthode se comportera avec les transactions
- Peut annoter une classe pour donner une valeur par défaut à ses méthodes

Richard Grin

CDI

page 40

40

Implémentation de @Transactional

- CDI offre des intercepteurs qui peuvent être exécutés avant ou après les appels de méthode
- Ce mécanisme peut être implémenté avec des clients proxy générés par CDI
- Un proxy pour une classe C est une instance d'une classe P qui a le même type que C et qui est utilisée à la place d'une instance de C
- Pour @Transactional, le proxy utilisé par CDI contient la logique liée aux transactions

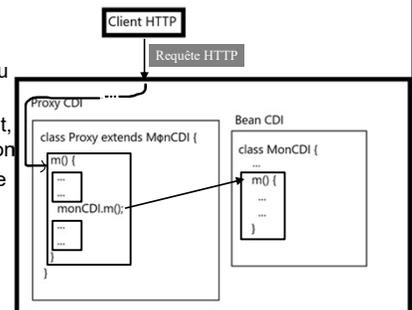
Richard Grin

CDI

page 41

41

- Le code qui traite la requête appelle la méthode m() du bean CDI
- C'est la méthode m() du proxy qui est appelée
- Cette méthode m() peut, démarrer une transaction
- puis appeler la méthode m() du bean CDI,
- puis, au retour de cette méthode, lancer un commit ou un rollback



Richard Grin

CDI

page 42

42

Quelles méthodes annoter avec @Transactional ?

- Celles qui doivent être exécutées dans une transaction : celles qui créent, modifient ou suppriment des valeurs dans une base de données
- Celles qui retournent des valeurs (de type « select ») peuvent ne pas être annotées ; choix dépend du contexte

Richard Grin

CDI

page 43

43

Attributs de @Transactional

- Les attributs optionnels de l'annotation sont
 - value (type TxType) ; indique si une méthode doit être exécutée dans le contexte d'une transaction
 - dontRollbackOn (type Class[])
 - rollbackOn (type Class[])
- Valeurs par défaut : REQUIRED pour value et un tableau vide pour les 2 autres éléments

Richard Grin

CDI

page 44

44

value

- Indique au container ce qu'il doit faire au début et à la fin de l'exécution d'une méthode
- Exemple :

```
import jakarta.transaction.Transactional.TxType;  
  
@Transactional(TxType.SUPPORTS)  
public List<Client> getAllClients() { ... }
```

Richard Grin

CDI

page 45

45

Enumération Transactional.TxType

- NEVER
- NOT_SUPPORTED
- SUPPORTS
- REQUIRED
- REQUIRES_NEW
- MANDATORY
- La suite explique ces valeurs ; une méthode method-A appelle une méthode method-B qui est annotée par @Transactional avec un paramètre qui prend chacune de ces valeurs

Richard Grin

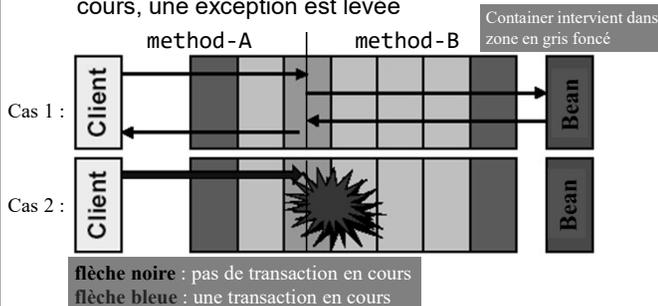
CDI

page 46

46

NEVER

- Si method-B est appelée avec une transaction en cours, une exception est levée



Richard Grin

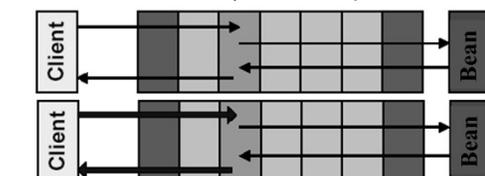
CDI

page 47

47

NOT_SUPPORTED

- Toute transaction en cours est suspendue pendant l'exécution de method-B
- Pour une méthode qui ne comporte que des « select » pour des entités qui ne sont pas modifiées ensuite (meilleures performances)



Richard Grin

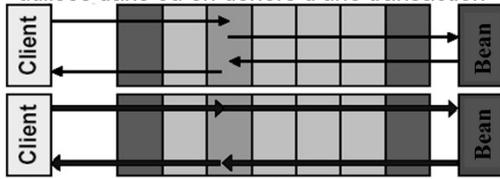
CDI

page 48

48

SUPPORTS

- Le bean peut supporter une transaction s'il y en a une mais peut fonctionner sans
- Pour une méthode qui ne comporte que des « select » ; à utiliser avec prudence car les résultats des « select » peuvent être différents si utilisés dans ou en dehors d'une transaction



Richard Grin

CDI

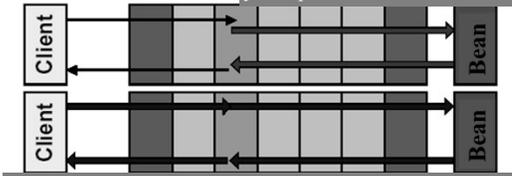
page 49

49

REQUIRED

- Une nouvelle transaction est créée par le container si aucune n'est en cours ; si une transaction est en cours, elle est utilisée par method-B
- Valeur par défaut

Pourquoi fin de la nouvelle transaction juste après exécution de method-B ?



flèche rouge : une nouvelle transaction créée par le container

Richard Grin

CDI

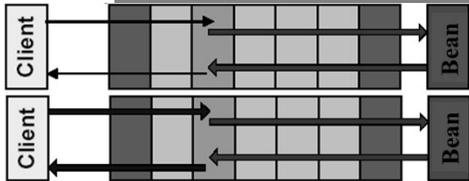
page 50

50

REQUIRES_NEW

- Une nouvelle transaction est toujours créée par le container pour cette méthode
- Pour une méthode qui veut être sûre que ses modifications sont validées ou non à la fin de son exécution

Exemple de différence avec @Requies ?



Richard Grin

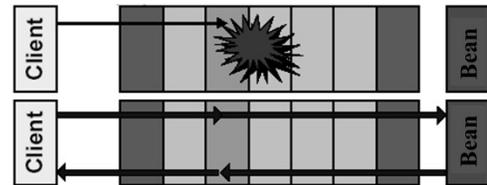
CDI

page 51

51

MANDATORY

- Une exception est levée si aucune transaction n'est en cours au moment de l'appel de la méthode



Richard Grin

CDI

page 52

52

Transactions et exceptions

Richard Grin

CDI

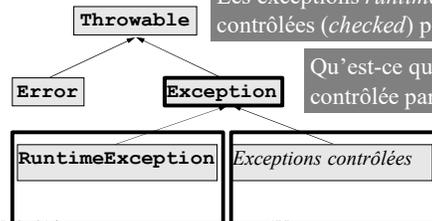
page 53

53

Rappel de Java

- Une exception hérite de la classe Exception
- Une exception est « runtime » quand elle hérite (directement ou indirectement) de RuntimeException
- Une exception est « checked » quand elle n'hérite pas de RuntimeException

Les exceptions runtime ne sont pas contrôlées (checked) par le compilateur.



Qu'est-ce qu'une exception contrôlée par le compilateur ?

Richard Grin

CDI

page 54

54

Prise en compte des exceptions

- Par défaut, si une exception « runtime » est lancée par une méthode pendant une transaction, la transaction est « marquée pour un rollback » : un rollback sera effectué par le serveur d'application quand la transaction se terminera
- Si aucune exception « runtime » n'est lancée, la transaction n'est pas marquée pour un rollback, la transaction se terminera par un commit (même si une exception non runtime a été lancée)
- Les attributs `rollbackOn` et `dontRollbackOn` peuvent modifier ce comportement

Richard Grin

CDI

page 55

55

rollbackOn et dontRollbackOn

- Si une classe est mentionnée, les classes filles sont aussi concernées
- Si une classe est concernée par les 2 attributs, l'attribut `dontRollbackOn` l'emporte

Richard Grin

CDI

page 56

56

Exemple

- Les exceptions contrôlée `SQLException`, sauf les `SQLWarning`, provoqueront un rollback :

```
@Transactional(  
    rollbackOn = { SQLException.class,  
                  JMSEException.class }  
    dontRollbackOn = { SQLWarning.class })  
public void m(...) {  
    ...  
}
```

Richard Grin

CDI

page 57

57

Remarque

- Pour que le container ait le comportement décrit, il faut qu'il puisse prendre connaissance des exceptions
- Par exemple, si une `RuntimeException` lancée dans une méthode est attrapée tout de suite dans cette même méthode, cette exception ne provoquera pas de rollback à la fin de la transaction
- De même, si une méthode `m1` d'un bean CDI appelle une méthode `m2` *du même bean* annotée avec `@Transactional`, il ne sera pas tenu compte de cette annotation

Richard Grin

CDI

page 58

58

Usage fréquent

- Une transaction est démarrée au début d'une méthode `m1` d'un bean CDI
- `m1` appelle une méthode `m2` d'un autre bean CDI
- La méthode `m2` ne peut pas exécuter ce qui lui est demandé ; elle lance une exception contrôlée pour le signaler
- La méthode `m1` attrape cette exception (catch) et essaie de réparer le problème
- Si elle n'y parvient pas, elle doit provoquer un rollback de la transaction ; comment faire ?

Richard Grin

CDI

page 59

59

Provoquer un rollback

- Il suffit de lancer une exception runtime qui provoque un rollback automatique ; la méthode qui a appelé la méthode `m1` contient un catch de l'exception pour savoir s'il y a eu commit ou rollback
- On peut lancer, par exemple, une `TransactionException` qui est une exception runtime, en passant comme cause en paramètre l'exception initiale non runtime

Richard Grin

CDI

page 60

60

Exécution d'une méthode au démarrage ou à l'arrêt d'une application

Richard Grin

CDI

page 61

61

Evénements CDI

- Un type d'événement peut être écouté par des méthodes (les méthodes sont exécutées lorsque l'événement survient) ; on dit que la méthode observe ce type d'événement
- Le développeur peut créer ses propres types événements, mais ce support ne détaillera pas cette possibilité
- Il existe aussi des événements intégrés à CDI, en particulier ceux qui sont déclenchés par le début ou la fin d'une portée CDI

Richard Grin

CDI

page 62

62

Faire exécuter une méthode au démarrage d'une application

- Il faut écrire la méthode dans un bean CDI de portée `ApplicationScoped` et annoter ainsi un des paramètres de la méthode :

```
@Observes  
@Initialized(ApplicationScoped.class)
```

- Ces annotations indiquent que la méthode observe (`@Observes`) le démarrage (`@Initialized`) de la portée « application », donc de l'application (`ApplicationScoped.class`)

Richard Grin

CDI

page 63

63

Arrêt de l'application

- Si on remplace `@Initialized` par `@Destroyed`, la méthode sera exécutée à l'arrêt de l'application

Richard Grin

CDI

page 64

64

Exemple @Initialized

```
public void init(  
    @Observes  
    @Initialized(ApplicationScoped.class)  
    ServletContext context) {  
    users.ajouter(new User("bob", 35));  
    ... // ajoute d'autres utilisateurs  
}
```

Richard Grin

CDI

page 65

65

Appel asynchrone de méthode

Richard Grin

CDI

page 66

66

Traitements asynchrones

- Intéressant de lancer en asynchrone une méthode longue à exécuter si elle ne nécessite pas l'intervention de l'utilisateur
- Exemples : envois d'emails, impressions, traitements qui utilisent des bases de données
- Avec Jakarta EE, pas recommandé de créer explicitement un thread pour exécuter la méthode
- Pour que l'appel d'une méthode se fasse en asynchrone il suffit de l'annoter avec `@Asynchronous`

Richard Grin

CDI

page 67

67

@Asynchronous

- Paquetage `jakarta.enterprise.concurrent`
- La méthode annotée est exécutée en // ; **elle retourne immédiatement** ; son éventuel résultat pourra être récupéré plus tard quand la méthode terminera son exécution, en utilisant un `CompletableFuture`
- La méthode doit retourner `void`, `CompletableFuture` (ou `CompletionStage`)

Richard Grin

CDI

page 68

68

CompletableFuture

- Classe de `java.util.concurrent`, qui représente un résultat potentiel ou une exception résultant d'une opération asynchrone
- Une méthode qui est lancée en parallèle peut retourner immédiatement une instance de `CompletableFuture` qui peut être utilisée pour récupérer le résultat quand il sera disponible, ou pour enchaîner d'autres opérations asynchrones
- L'utilisation de `CompletableFuture` n'est pas au programme de ce cours mais l'exemple suivant donne un aperçu de son utilisation

Richard Grin

CDI

page 69

69

Exemple (méthode)

```
@Asynchronous
public CompletableFuture<Double> heuresTravail(LocalDate depuis,
LocalDate jusqu) {
    try (Connection con = ((DataSource) InitialContext.doLookup(
        "java:comp/env/jdbc/tempsTravailDB")).getConnection()) {
        ... // Calcule le total des heures travaillées
        return Asynchronous.Result.complete(total);
    } catch (NamingException | SQLException x) {
        throw new CompletionException(x);
    }
}
```

L'accès à la base de données des heures travaillées utilise JDBC. La méthode retourne immédiatement ; l'accès à la base de données et le calcul seront effectués en arrière-plan

Richard Grin

CDI

page 70

70

Exemple (utilisation méthode)

```
heuresTravail(lundi, vendredi).thenAccept(total -> {
    DataSource ds = InitialContext.doLookup(
        "java:comp/env/jdbc/salairesDB");
    ... // Utilisation de total
});
```

Lorsque la méthode retournera, le résultat sera utilisé pour calculer le salaire et l'enregistrer dans la base de données des salaires

Richard Grin

CDI

page 71

71