Dependency Injection and Mocking

AMT

9 - Dependency Injection and Mocking

Résumé du document

Cette note présente les concepts de l'injection de dépendances avec Jakarta CDI et du mocking dans le cadre des tests unitaires. Elle explique comment Jakarta CDI facilite la création, l'injection et la gestion du cycle de vie des dépendances tout en favorisant la modularité et la testabilité. Les notions de stubbing et mocking sont également détaillées, montrant comment isoler les classes sous test et vérifier leurs interactions avec des dépendances simulées. Ces pratiques permettent d'améliorer la qualité, la maintenabilité et la flexibilité du code.

Table des matières

1. Injection de dépendance avec Jakarta CDI

Jakarta CDI (Contexts and Dependency Injection) est une spécification pour l'injection de dépendances (DI) et l'inversion de contrôle (IoC) dans les applications Jakarta EE. Les environnements qui implémentent cette spécification gèrent la création, l'injection des dépendances et leur cycle de vie.

1.1. Avantages de Jakarta CDI

- Flexible et sans opinion préconçue.
- Indépendant de l'environnement d'exécution.
- Favorise la modularité.
- Améliore la testabilité.
- Permet un code plus clair et mieux organisé.

Cependant, certains développeurs critiquent sa complexité et sa verbosité, comme pour la spécification JPA.

1.2. Concepts clés de Jakarta CDI

- **Beans**
- **Scopes**
- **Producers**
- **Qualifiers**
- **Cycle de vie**
- **Intercepteurs**
- **Décorateurs**
- **Événements et observateurs**, etc.

1.3. Beans

Un **bean** est une classe gérée par le runtime. **Attention** à ne pas confondre avec les **JavaBeans** qui sont des classes Java avec des propriétés et des méthodes d'accès. C'est pourquoi nous parlons plus de **CDI Bean** pour bien faire la distinction. Une instance de **CDI Bean** est appelée une instance managée ou en anglais **managed instance**.

1.4. Scopes

Les **scopes** définissent la durée de vie d'une instance de **CDI Bean**. Il existe plusieurs **scopes** prédéfinis dans Jakarta CDI, tels que :

- **ApplicationScoped**: une seule instance pour toute l'application.
- **SessionScoped**: une instance par session utilisateur.
- **RequestScoped**: une instance par requête.
- **Dependent**: une instance par injection (scope par défaut).

Les instances managées sont généralement crée de manière **lazily** donc que en cas d'utilisation et sont détruites à la fin de leur **scope**. De manière générale, les instances sont exposées comme des **proxy** (le runtime injecte un **proxy** à la place de l'instance réelle).

- **Singleton**: une instance unique pour toute l'application.
  - L'instance est créée lors du démarrage de l'application (**eagerly**).
  - L'instance est détruite lors de l'arrêt de l'application.
  - L'instance n'est **pas** injectée en tant que **proxy**.

1.5. Producers and Qualifiers

1.5.1. Producers

Un producteur peut-être défini en utilisant l'annotation `@Produces`. Un producteur peut:

- Produire une instance d'une méthode ou d'un champ
- Des ambiguités peuvent apparaître si plusieurs producteurs produisent le même type

1.5.2. Qualifiers

Un qualificateur est une annotation qui permet de distinguer les différentes instances d'un même type. Par exemple, `@Named` est un qualificateur qui permet de distinguer les différentes instances d'un même type.

1.5.3. Exemple

```
@ApplicationScoped
public class Producers {
    @Produces
    @Named("name")
    public String name() {
        return "Edouard";
    }
}

@ApplicationScoped
public class HelloService {
    @Inject
    @Named("name")
    private String name;

    public String sayHello() {
        return "Hello " + name + "!";
    }
}
```

L'annotation `@Named` est un **qualifier** qui permet de distinguer les différentes instances d'un même type.

1.6. Life Cycle

Le cycle de vie d'une instance de **CDI Bean** est défini par l'interface `Contextual`. Cela permet de donner accès aux annontations:

- `@PostConstruct`: méthode appelée après la création de l'instance.
- `@PreDestroy`: méthode appelée avant la destruction de l'instance.

```
@ApplicationScoped
public class HelloService {

    @PostConstruct
    public void init() {
        System.out.println("The managed instance is initialized.");
    }

    public String sayHello() {
        return "Hello World!";
    }
}
```

La méthode `init` est appelée après la création de l'instance.

1.7. Interceptors

Un intercepteur est une classe qui permet d'intercepter les appels à une méthode. Les annotations suivantes sont utilisées pour définir un intercepteur:

- `@InterceptorBinding`: annotation qui permet de définir un intercepteur.
- `@Interceptor`: annotation qui permet de définir une classe comme intercepteur.
- `@AroundInvoke`: annotation qui permet de définir une méthode qui sera appelée avant et après l'appel de la méthode interceptée.
- `@AroundConstruct`: annotation qui permet de définir une méthode qui sera appelée avant et après l'initialisation de l'instance.

Les intercepteurs sont utile pour implémenter des fonctionnalités transversales, comme le logging, la sécurité, etc..

Il ne faut pas les utiliser pour implémenter de la logique métier.

1.7.1. Exemple

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

@Interceptor
@Logged
public class LoggedInterceptor {
    @AroundInvoke
    public Object log(InvocationContext context) throws Exception {
        System.out.println("Before method " + context.getMethod().getName());
        Object result = context.proceed();
        System.out.println("After method " + context.getMethod().getName());
        return result;
    }
}

@ApplicationScoped
@Logged
public class HelloService {
    public String sayHello() {
        return "Hello World!";
    }
}
```

1.8. Decorators

Un décorateur est une classe qui permet de décorer le comportement d'une autre classe. Les annotations suivantes sont utilisées pour définir un décorateur:

- `@Decorator`: annotation qui permet de définir une classe comme décorateur.
- `@Delegate` and `@Inject`: injecte l'instance décorée dans un champ.
- `@Priority`: permet de définir la priorité du décorateur.
  - Plus la valeur est basse, plus le décorateur est prioritaire.

Les décorateurs sont utiles pour ajouter des fonctionnalités à une classe sans modifier son code. Ils peuvent être utilisé pour ajouter de la logique métier.

1.8.1. Exemple

```
@Decorator
public class HelloDecorator implements HelloService {
    @Inject
    @Delegate
    private HelloService helloService;
    @Override
    public String sayHello() {
        return helloService.sayHello() + " from decorator!";
    }
}

public interface HelloService {
    String sayHello();
}

@ApplicationScoped
public class HelloServiceImpl implements HelloService {
    public String sayHello() {
        return "Hello World!";
    }
}
```

```
@ApplicationScoped
public class HelloTrigger {
    @Inject
    private HelloService helloService;
    public void sayHello() {
        System.out.println(helloService.sayHello());
    }
}
```

1.9. Events and Observers

Les événements et les observateurs permettent de communiquer entre les différentes parties de l'application sans les couplers directement.

- La classe `Event` permet de publier un événement grâce à la méthode `fire`.
- `@Observes` permet de définir une méthode comme observateur d'un événement.

1.9.1. Exemple

```
@ApplicationScoped
public class HelloTrigger {
    @Inject
    private Event<String> event;

    public void sayHello() {
        event.fire("Hello World!");
    }
}

@ApplicationScoped
public class HelloObserver {
    public void observe(@Observes String message) {
        System.out.println(message);
    }
}
```

Le type de l'événement est défini par le type générique de la classe `Event` et la methode observant peut avoir le nom qu'on souhaite.

2. Mocking

2.1. Concepts clés

- **Classe sous test** La classe principale dont le comportement est testé.
- **Dépendance** Une classe dont la classe sous test a besoin pour fonctionner.
- **Unit tests** Vérifient le comportement et les interactions de la classe sous test avec ses dépendances.
- **Stub** Remplacement simplifié d'une dépendance, retournant des valeurs contrôlées pour isoler la classe sous test.
- **Mock** Remplacement avancé d'une dépendance, permettant de vérifier les interactions avec la classe sous test.
- **Patron AAA** Structure typique pour écrire des tests unitaires :
  - **Arrange** Préparation des objets nécessaires.
  - **Act** Invocation de la méthode à tester.
  - **Assert** Vérification des résultats.

2.2. Stubbing

- **Définition** Technique consistant à remplacer une dépendance par un stub.
- **Objectif** Isoler la classe sous test en contrôlant ses dépendances via des valeurs prédéfinies.
- **Exemple**
  - La classe `GreetingService` dépend de `HelloService`.
  - Lors du test, `HelloService` est remplacée par un stub qui retourne "Hello".
  - Cela permet de tester `GreetingService` en isolation.

2.3. Mocking

- **Définition** Technique consistant à remplacer une dépendance par un mock.
- **Objectif** Vérifier les interactions entre la classe sous test et ses dépendances.
- **Exemple**
  - La classe `NotificationService` dépend de `EmailSender`.
  - Lors du test, `EmailSender` est remplacé par un mock.
  - Le test vérifie que `NotificationService` appelle correctement la méthode `sendEmail` du mock.