Andr0

Zeste de savoir

Introduction à l'injection de dépendances en Java

12 février 2021

Table des matières

Le contenu de ce tutoriel est devenu obsolète et devrait être archivée et/ou supprimée.

L'injection de dépendances demande des compétences confirmées dans la programmation orientée objet. Dans la mesure du possible, les bases seront expliquées mais certaines notions seront considérées comme acquises, notamment l'utilisation des interfaces.

Les prérequis pour ce tutoriel sont:

— la programmation orientée objet;
— des connaissances basiques en java;
— un gestionnaire de dépendances comme Maven est un plus si vous désirez exécuter les exemples.

Tous les codes sources de ce tutoriel sont disponibles sur ce projet GitHub afin que vous puissiez consulter des exemples fonctionnels directement exécutables dans un terminal.

1. Qu'est ce que l'injection de dépendances ?

1.1. Mais une dépendance, c'est quoi ?

Un développeur qui se soucie de la qualité logicielle et de la maintenance de son application va tenter d'élaborer une architecture logicielle adaptée. Il existe plusieurs solutions, parmi lesquelles on trouve l'injection de dépendances. Reste à savoir ce qu'est une dépendance…

Pour illustrer ce concept, prenons un cas simple: une classe voudrait invoquer une ou plusieurs méthodes d'une autre classe. Une implémentation pourrait correspondre au code ci-dessous.

```
1  public class PizzaManagerImpl implements PizzaManager {
2    private final PizzaDao pizzaDao;
3
4    public PizzaManagerImpl(PizzaDao pizzaDao) {
5      this.pizzaDao = pizzaDao;
6    }
7
8    @Override public List<Pizza> menu() {
9      return pizzaDao.getAll();
10   }
11 }
```

La responsabilité de `PizzaManagerImpl` est de renvoyer toutes les pizzas du menu d'une pizzeria. Pour ce faire, cette classe a besoin d'une autre classe pour faire l'intermédiaire entre la base de donnée et elle-même (ceci dans un souci d'architecturer convenablement le code). La classe possède donc une **dépendance** vers la classe `PizzaDao`. Cette dépendance entraine un couplage fort puisque le manager ne peut pas se passer du dao des pizzas.

Le dao est un concept bien connu dans les applications, notamment celles qui communiquent avec une base de données. Sa responsabilité est d'abstraire les opérations usuelles sur une base de données (ou tout autre système de persistance des données) pour une ressource donnée. Ainsi, partout ailleurs dans l'application, il suffit d'utiliser le dao adéquat. Dans cet exemple, le dao est consacré à la récupération d'une liste de pizzas.

Pour éviter de rajouter la responsabilité d'instanciation des classes et pour réduire les dépendances entre les classes, l'injection de dépendances se place comme une solution adaptée. Raison pour laquelle ce patron de conception fait l'objet de ce tutoriel.

1.2. Donnons une définition à l'injection de dépendances

Pour être efficaces, prenons une définition mûrement réfléchie par la communauté de Wikipedia comme point de départ:

> L'injection de dépendances (Dependency Injection) est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle. Il consiste à créer dynamiquement (injecter) les dépendances entre les différentes classes en s'appuyant sur une description (fichier de configuration ou métadonnées) ou de manière programmatique. Ainsi les dépendances entre composants logiciels ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement à l'exécution.
>
> *Version de l'article Injection de dépendances de Wikipedia datée du 4 juin 2014*

Cette définition utilise des termes génériques pour toucher le plus de domaines possible. Dans le cas ci-présent, le tutoriel aborde l'injection de dépendances dans le langage java et par ses bibliothèques. Il est possible d'adapter un poil cette définition comme suit:

> L'injection de dépendances (Dependency injection) est un mécanisme qui permet d'implémenter le principe de l'inversion de contrôle (souvent abrégé par l'acronyme IoC). Elle consiste à **injecter dynamiquement les dépendances** pour différentes classes en s'appuyant sur un ou plusieurs **modules**. Ainsi, les **dépendances entre des classes d'implémentation et une interface** ne sont plus exprimées dans le code de manière statique mais déterminées dynamiquement.

Il se dégage de cette définition un concept important: pour pouvoir utiliser l'injection de dépendances, il est obligatoire de penser l'architecture de son logiciel avec des interfaces et de coupler leurs implémentations grâce à des modules. En plus de permettre l'injection, c'est une bonne pratique de conception pour éviter des couplages forts entre différents modules d'un logiciel.

Bien entendu, l'injection de dépendances ne se limite pas à l'injection par interface. Il existe 4 types d'injections de dépendances:

- injection par constructeur;
- injection par interface;
- injection par mutateur;
- injection par champs.

Toutes ces injections ne seront pas abordées puisque les bibliothèques Java ne supportent pas tous les types d'injection. Si vous êtes intéressés par une injection spécifique, des nombreuses ressources sont disponibles sur internet.

2. Présentation de bibliothèques d'injection de dépendances

Des bibliothèques sur l'injection de dépendances, il en existe des tas dans quasiment tous les langages. Par conséquent, il serait difficile de toutes les couvrir. Sachant toutefois que les différences entre ces bibliothèques sont minimes et se jouent plutôt sur la syntaxe qu'au niveau des performances vis-à-vis du compilateur par exemple.

Notez que certaines bibliothèques sont plus performantes que d'autres dans des utilisations bien précises. Ce niveau de performance ne sera pas abordé dans le cadre de ce tutoriel puisqu'il s'agit là d'un cadre plus avancé de l'injection de dépendances.

Devant les innombrables bibliothèques traitant de notre sujet, un premier parti portera sur le langage. Les exemples seront illustrés par le java, étant donné son aspect pédagogique et qu'il s'agit d'un des langages les plus connus par les développeurs à travers le monde à ce jour (où du moins, un des plus faciles à prendre en main).

- Google Guice : À l'époque, cette bibliothèque avait été développée par Google pour tirer parti de toute la puissance de Java, notamment sur les annotations et les génériques. Ceci pour faire face à une alternative vieillissante, Spring . Encore aujourd'hui, c'est

l'une des solutions les plus utilisées dans les applications, par son développement actif et sa communauté qui a mené à d'autres projets comme Robo Guice .

— Dagger : Développé par une boite très active dans la conception de bibliothèques open-sources, Square , Dagger ne se distingue pas par son utilisation, hormis sur quelques points, mais par son support natif d'Android et par ses meilleures performances. Cependant, cette bibliothèque souffre de limitations techniques lorsque des choses plus complexes sont nécessaires.

Les différences sont minimes et se jouent à pas grand chose. Raison pour laquelle aucune distinction ne sera faite dans les exemples de la section suivante. Si un choix doit être fait pour l'une de ces librairies, tout est une question de besoin. Si un conseil devait vous être donné, ce serait le suivant: si votre projet est conséquent, privilégiez Google Guice. Sinon, privilégiez la légèreté et la facilité de prise en main de Dagger.

3. Utiliser l'injection de dépendances

Pour illustrer l'injection de dépendances, Google Guice sera utilisé pour sa maturité et sa fiabilité. Pour consulter le code source compilable et testable, rendez-vous sur le dépôt git du tutoriel .

3.1. Déclarer une dépendance

Google Guice construit les instances des classes d'une application et satisfait leurs dépendances. Il utilise les annotations du package `com.google.inject` ou, plus standard, `javax.inject`. Leurs annotations `Inject` identifient les constructeurs ou les attributs à injecter.

Utiliser l'annotation `@Inject` sur le constructeur ou l'attribut d'une classe permet de créer une nouvelle instance des paramètres du constructeur ou de l'attribut. Lorsqu'une nouvelle instance est requise par l'application, Google Guice satisfera les dépendances nécessaires en invoquant les constructeurs voulus. Donc, tout se passe à l'exécution!

Pour illustrer l'injection sur le constructeur, `PizzaManagerImpl` désire une instance pour `pizzaDao` du type `PizzaDao`. Attention, il faut quand même sauvegarder (si c'est le comportement désiré) les paramètres du constructeur dans les attributs, pour pouvoir les utiliser dans la classe. Comme le nom de l'injection l'indique, Google Guice va faire l'injection au niveau du constructeur et pas sur les attributs:

```
1 public class PizzaManagerImpl implements PizzaManager {
2   private final PizzaDao pizzaDao;
3
4   @Inject public PizzaManagerImpl(PizzaDao pizzaDao) {
5     this.pizzaDao = pizzaDao;
6   }
7
8   @Override public List<Pizza> menu() {
9     return pizzaDao.getAll();
```

```
10    }
11    //...
12  }
```

— Google Guice est compatible avec les annotations `com.google.inject` et `javax.inject` alors que Dagger n'est compatible qu'avec ces dernières. Dans le doute, utilisez `javax.inject`, puisque cette solution a le mérite d'être la plus standard et la plus utilisée.
— Contrairement à Google Guice, lorsqu'une annotation est placée au niveau du constructeur plutôt que sur un attribut avec Dagger, les dépendances sont résolues au moment de la compilation et non plus à l'exécution. Les performances à l'usage seront bien meilleures.

Une solution équivalente à l'exemple précédent pour l'injection sur les attributs, `PizzaManagerImpl` peut placer l'annotation d'injection `@Inject` sur l'attribut pour se le faire injecter automatiquement (sans passer par le constructeur). Pour rappel, même si cette solution semble plus simple, élégante et facile, il ne faut pas oublier qu'aucune optimisation ne sera faite ni pour Dagger ni pour Google Guice avec cette solution. Elle est donc plus coûteuse.

```
1  public class PizzaManagerImpl implements PizzaManager {
2    @Inject PizzaDao pizzaDao;
3    //...
4  }
```

3.2. Satisfaire les dépendances

Pour satisfaire les dépendances d'une application, des modules sont développés afin de renseigner les classes d'implémentation. Les bibliothèques peuvent alors utiliser ce module pour invoquer le constructeur voulu, pour le constructeur ou l'attribut annoté par une annotation d'injection.

La seule réelle différence dans l'usage des bibliothèques Google Guice et Dagger réside dans cette satisfaction des dépendances. Alors que Google Guice utilisera une classe abstraite de sa bibliothèque, Dagger modernisera le concept en n'utilisant que des annotations. Puisque les solutions sont vraiment différentes, les deux bibliothèques seront expliquées dans cette section.

3.2.1. Google Guice

Une nouvelle classe doit étendre la classe `AbstractModule` de la bibliothèque Google Guice. Cette dernière classe nous oblige à mettre en œuvre la méthode `configure()`. C'est dans cette méthode qu'une interface ou une classe abstraite renseigne sa classe concrète pour pouvoir être instanciée.

Par exemple, la classe abstraite `AbstractPayment` est, par définition, non instanciable. C'est pourquoi, le module spécifie une de ses classes filles comme classe d'implémentation pour que l'application puisse connaître la classe qu'il devra instancier à l'exécution. Ce procédé fonctionne de la même manière avec les interfaces. Ceci se fait très simplement avec des méthodes de la classe `AbstractModule`: `bind(...)` pour renseigner l'interface ou la classe abstraite, suivi de `to(...)` pour renseigner la classe concrète.

```
1 public class PizzaManagerModuleGuice extends AbstractModule {
2   @Override protected void configure() {
3     bind(AbstractPayment.class).to(CreditCardPayment.class);
4   }
5 }
```

Pour utiliser ce module, dans la classe principale de l'application avec la méthode `main`, il faut créer l'injecteur avec la méthode statique `Guice.createInjector(...)` pour pouvoir injecter toutes les instances nécessaires dans une classe et l'utiliser. Dans cet exemple, Guice fournit une instance de la classe `AppPizzaManagerGuice` pour pouvoir injecter les classes voulues.

```
1  public class AppPizzaManagerGuice implements Runnable {
2    @Inject PizzaManager pizzaManager;
3    @Inject PaymentManager paymentManager;
4
5    public static void main(String[] args) {
6      final Injector injector = Guice.createInjector(new
           PizzaManagerModuleGuice());
7      final AppPizzaManagerGuice app =
           injector.getInstance(AppPizzaManagerGuice.class);
8      app.run();
9    }
10
11   @Override public void run() {
12
           System.out.println(paymentManager.payWithPayPal(pizzaManager.menu()));
13   }
14 }
```

3.2.2. Dagger

Un module avec Dagger ne doit étendre aucune classe parent mais doit faire bon usage des annotations. La classe du module va être annotée par `@Module` et renseigner en paramètre la classe principale de l'application. Puis, toutes les dépendances seront résolues par des méthodes. Leurs types de retour correspondent à l'interface ou la classe abstraite, la valeur retournée dans le corps de la méthode à la classe d'implémentation pour le type de retour et la méthode est annotée par l'annotation `@Provides`.

```
@Module(
        injects = AppPizzaManagerDagger.class
)
public class PizzaManagerModuleDagger {
  @Provides AbstractPayment provideCreditCard() {
    return new CreditCardPayment();
  }
}
```

Pour utiliser ce module, il faut obtenir le graphe des injections avec l'invocation de la méthode statique `ObjectGraph.create(...)` qui accepte un ou plusieurs modules.

```
final ObjectGraph objectGraph = ObjectGraph.create(new
    PizzaModule());
```

Afin d'utiliser ce graphe, il faut amorcer les injections. Habituellement, cela requiert l'injection de la classe principale. Pour l'exemple, la classe `AppPizzaManagerDagger` est utilisée pour démarrer l'injection de dépendances. Nous demandons au graphe de fournir une instance injectée de la classe:

```
public class AppPizzaManagerDagger implements Runnable {
  @Inject PizzaManager pizzaManager;
  @Inject PaymentManager paymentManager;

  public static void main(String[] args) {
    final ObjectGraph objectGraph = ObjectGraph.create(new
        PizzaManagerModuleDagger());
    final AppPizzaManagerDagger app =
        objectGraph.get(AppPizzaManagerDagger.class);
    app.run();
  }

  @Override public void run() {

        System.out.println(paymentManager.payWithPayPal(pizzaManager.menu()));
  }
}
```

4. Des fonctionnalités supplémentaires

Jusqu'à maintenant, il a été présenté la fonctionnalité principale de l'injection de dépendances mais il en existe bien d'autres, dont certaines qui peuvent varier d'une bibliothèque à l'autre

ou d'un langage à l'autre. L'idée n'est pas de toutes les énumérer ni de toutes les expliquer. Par contre, il en existe quelques unes qui sont assez génériques et présentes dans la plupart des solutions.

4.1. Nommer les injections

Imaginez deux attributs du même type (par exemple, d'une classe abstraite) mais pour lesquels vous désirez des implémentations différentes. La fonctionnalité de base consiste à lier une classe abstraite (ou une interface) à une classe concrète (ou d'implémentation). À partir de ce constat, il ne serait pas possible de fournir deux implémentations différentes. Si vous tentez de faire:

```
1 bind(AbstractPayment.class).to(PayPalPayment.class);
2 bind(AbstractPayment.class).to(CreditCardPayment.class);
```

La bibliothèque prendra en compte le dernier «binding». Toutes les injections pour `AbstractPayment` seront instanciées par `CreditCardPayment`. Pour pallier ce problème, une des solutions les plus communes est la possibilité de nommer ses injections grâce à l'annotation `@Named("your-name")`. Le mode de payement par PayPal pourrait être nommé par "paypal" et la carte de crédit par "creditcard". Ainsi, `PaymentManager` peut disposer de deux attributs typés de `AbstractPayment` avec deux implémentations différentes.

```
1 public class PaymentManagerImpl implements PaymentManager {
2   @Inject @Named("paypal") AbstractPayment payPal;
3   @Inject @Named("creditcard") AbstractPayment creditCard;
4   //...
5 }
```

Petite modification aussi du côté du module puisqu'il faut spécifier à Google Guice que toutes les injections du type `AbstractPayment` et annotées par le nom «paypal» ou «creditcard» doivent posséder l'implémentation `PayPalPayment` ou `CreditCardPayment` (réciproquement).

```
1 public class PizzaManagerModuleGuice extends AbstractModule {
2   @Override protected void configure() {
3
          bind(AbstractPayment.class).annotatedWith(Names.named("paypal")).to(Pa
4
          bind(AbstractPayment.class).annotatedWith(Names.named("creditcard")).to
5   }
6   //...
7 }
```

Pour Dagger, il suffit de créer deux nouvelles méthodes annotées par `@Provides` et `@Named("name")` en renvoyant la classe d'implémentation voulue.

```
public class PizzaManagerModuleDagger {
  @Provides @Named("paypal") AbstractPayment providePayPal() {
    return new PayPalPayment();
  }

  @Provides @Named("creditcard") AbstractPayment
      provideCreditCard() {
    return new CreditCardPayment();
  }
  //...
}
```

4.2. Ses propres annotations pour nommer une injection

Une solution qui peut paraitre un poil plus élégante, en fonction du développeur, est la possibilité de créer ses propres annotations pour nommer une injection. Ainsi, cela consiste à créer une nouvelle annotation en spécifiant que c'est une annotation de type «binding», qu'elle peut s'appliquer sur des attributs, des paramètres et des méthodes et qu'elle est interprétée à l'exécution du programme.

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

L'exemple du mode de payement va légèrement évoluer pour PayPal puisqu'il faut dorénavant annoter le champ par `@PayPal` plutôt que par '@Named("paypal").

```
public class PaymentManagerImpl implements PaymentManager {
  @Inject @PayPal AbstractPayment payPal;
  @Inject @Named("creditcard") AbstractPayment creditCard;
  //...
}
```

Par contre, réelle nouveauté dans le module de Google Guice puisqu'il n'est plus possible de renseigner l'injection dans la méthode `configure()`. La mise en œuvre se rapproche fortement de Dagger puisqu'il est nécessaire de créer une nouvelle méthode annotée par `@Provides` et `@PayPal` en renvoyant une instance de la classe d'implémentation voulue pour le type retourné de la méthode.

```java
public class PizzaManagerModuleGuice extends AbstractModule {
  @Override protected void configure() {

        bind(AbstractPayment.class).annotatedWith(Names.named("creditcard")).to
  }

  @Provides @PayPal public AbstractPayment providePayPal() {
    return new PayPalPayment();
  }
  //...
}
```

4.3. Fournir des instances différentes

Parfois, il est nécessaire de vouloir retourner plusieurs instances différentes pour une injection. Il existe aussi plusieurs possibilités, une de ces options étant l'injection d'un attribut grâce à `Provider<T>`. `Provider<T>` crée une nouvelle instance de `T` à chaque invocation de la méthode `get()` sur l'attribut.

```java
public class PaymentManagerImpl implements PaymentManager {
  @Inject @PayPal AbstractPayment payPal;
  @Inject @Named("creditcard") AbstractPayment creditCard;
  @Inject Provider<Payment> paymentProvider;

  @Override public Payment payWithPayPal(List<Pizza> pizzas) {
    return getPayment(payPal.pay(pizzas));
  }

  @Override public Payment payWithCreditCard(List<Pizza> pizzas) {
    return getPayment(creditCard.pay(pizzas));
  }

  private Payment getPayment(double total) {
    final Payment payment = paymentProvider.get();
    payment.setTotal(total);
    return payment;
  }
}
```

Ce tutoriel aborde les bases de l'injection de dépendances, mais des bases tout de même assez solides pour une utilisation courante dans des petits et moyens projets. Pour en savoir plus sur l'injection de dépendances, de nombreuses ressources existent sur internet, notamment la documentation officielle des bibliothèques mises en œuvre dans ce tutoriel.

Sachez aussi que toutes les sources des exemples de ce tutoriel sont disponibles sur ce dépôt git . Libre à vous de vous inspirer de ce projet pour vos projets personnels, ou d'y apporter des améliorations si vous tombez sur des petites trouvailles à force de pratique; toute contribution directe de votre part est la bienvenue.

Dans des tutoriels ouverts comme celui-ci, chacun peut apporter sa pierre à l'édifice.

4.4. Remerciements

— L'icône de ce tutoriel est distribuée sous licence CC BY-SA 2.0, par tamasrepus et distribuée via la plateforme Flickr .
— Merci à la communauté de Zeste de Savoir pour leurs retours sur ce tutoriel lorsqu'il était en bêta.
— Merci à Coyote pour la relecture de ce tutoriel.