Ecole Normale Supérieure de
l'Enseignement Technique Mohammedia
Université Hassan II de Casablanca

ENSET

ا لمدرسة العليا لأساتذة التعليم التقني
المحمدية
جامعة الحسن الثاني بالدار البيضاء

DEPARTEMENT MATHEMATIQUES ET INFORMATIQUE

Compte rendu du Mini Projet Framework d'Injection des dépendances

Filière :
« Génie du Logiciel et des Systèmes Informatiques Distribués »
GLSID2

Module : Architecture Distribuée et Middlewares

Élaboré par :

ELMAJNI Khaoula

Encadré par :

M. YOUSSFI Mohammed

Année Universitaire : 2021-2022

Contents

Introduction

Ce travail a pour objectif de créer un mini framework d'injection des dépendances, afin de l'utiliser en créant des applications Java légère à l'aide de notre propre implémentation d'injection de dépendance.

Dans ce travail on va concevoir et créer un mini Framework d'injection des dépendances similaire à Spring IOC.
Le mini Framework va permettre de faire l'injection des dépendances entre les différents composant de son application respectant les possibilités suivantes :

1- A travers un fichier XML de configuration
2- En utilisant les annotations
3- Possibilité d'injection via :
   a- Le constructeur
   b- Le Setter
   c- Attribut (accès direct à l'attribut : Field)

Injection de dépendance

L'injection de dépendances est un modèle de conception utilisé pour implémenter IoC (Inversion de Controle), dans lequel les variables d'instance (c'est-à-dire les dépendances) d'un objet sont créées et affectées par le framework.

Autrement dit l'injection de dépendances est une technique de programmation qui rend une classe indépendante de ses dépendances. Il y parvient en dissociant l'utilisation d'un objet de sa création.

Objectif :

L'injection de dépendance prend en charge l'amélioration de la réutilisabilité du code en découplant la création de l'utilisation d'un objet.

Cela nous permet de remplacer les dépendances sans changer la classe qui les utilise, et réduit également le risque de modification d'une classe simplement parce que l'une de ses dépendances a changé.

Les 4 rôles dans l'injection de dépendances

Pour utiliser cette technique, nous avons besoin de classes qui remplissent quatre rôles de base. Ceux-ci sont:

- Le service que nous souhaitons utiliser.
- Le client qui utilise le service.
- Interface utilisée par le client et implémentée par le service.
- L'injecteur qui crée une instance de service et l'injecte dans le client.

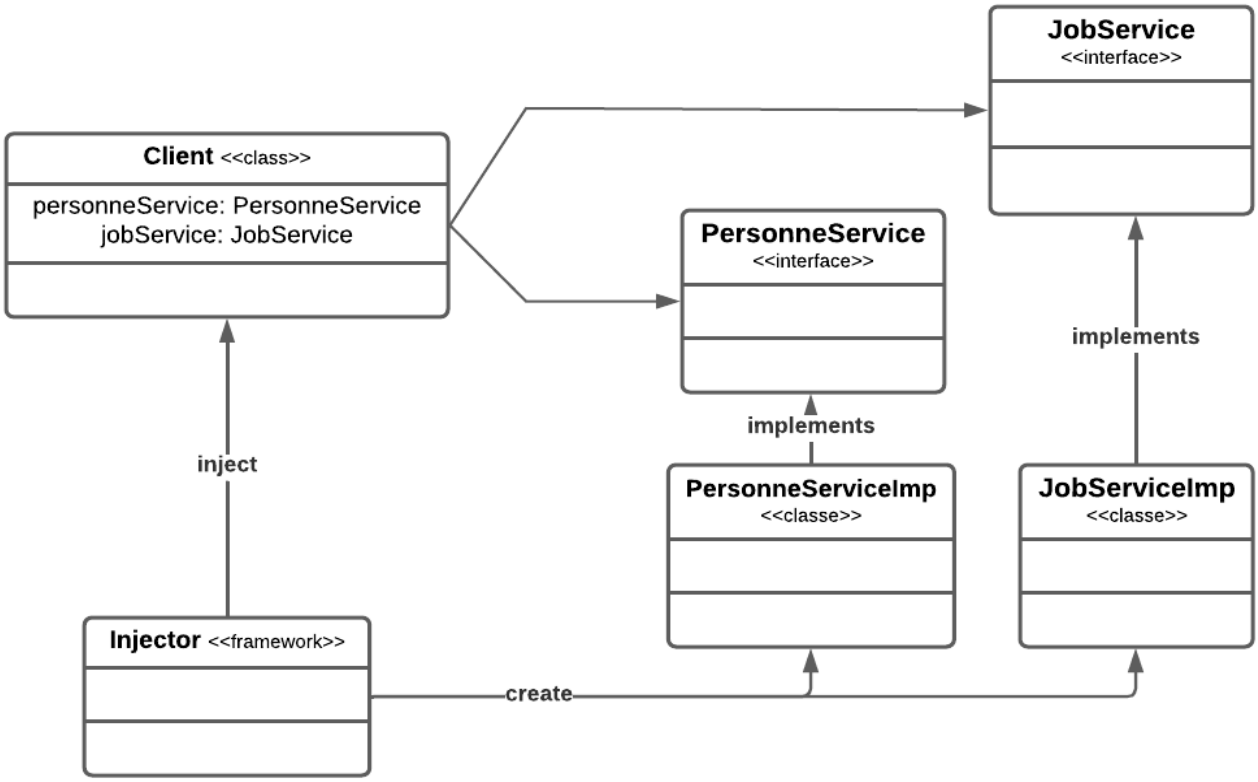
On peut introduire des interfaces pour rompre les dépendances entre les classes de niveau supérieur et inférieur. De ce fait les classes dépendent de l'interface et non plus l'une de l'autre.

Inversion de contrôle

C'est un principe de conception qui conciste sur la séparation de tous ce qui métier à tous ce qui est technique, il permet aux développeurs de s'occuper uniquement du code métier alors que le code technique va être occuper par un framework, en utilisant le paradigme AOP (Aspect Oriented Programming).

Il est utilisé pour inverser différents types de contrôles (c'est-à-dire la création d'objets ou la création et la liaison d'objets dépendants), et il est l'une des approches pour mettre en œuvre l'IoC.

Diagramme de classe d'injection de dépendance de notre framework et une petite application pour le test



Dans ce diagramme de classe, la classe Client qui requiert les objets PersonneService et JobService n'instancie pas directement les classes PersonneServiceImp et JobServiceImp.

Au lieu de cela, une classe Injector crée les objets et les injecte dans le client, ce qui rend le client indépendant de la façon dont les objets sont créés.

Types d'injection de dépendance

Injection à travers le constructeur : l'injecteur fournit le service (dépendance) via le constructeur de classe client. Dans ce cas, annotation Autowired ajoutée sur le constructeur.

Injection à travers l'attribut : l'injecteur fournit le service (dépendance) via une propriété publique de la classe client. Dans ce cas, l'annotation **@Autowired** a été ajoutée lors de la déclaration de la variable membre.

Injection à travers la méthode setter : la classe client implémente une interface qui déclare la ou les méthodes pour fournir le service (dépendance) et l'injecteur utilise cette interface pour fournir la dépendance à la classe client.

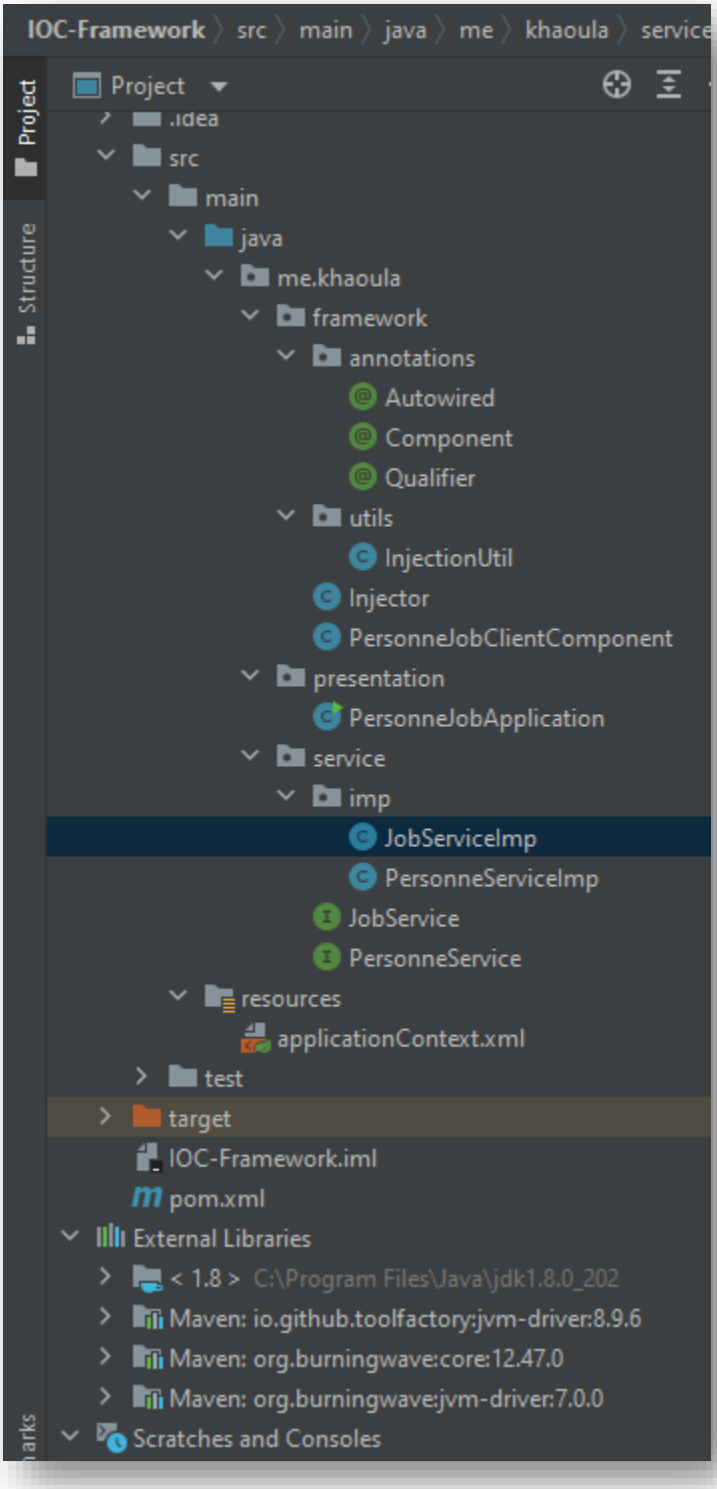
Les librairies utilisées

Les étapes de codage :

j'ai créé un nouveau projet maven dans IntellijIdea et ajouté la dépendance Burningwave Core dans pom.xml :

```
<dependency>
   <groupId>org.burningwave</groupId>
   <artifactId>core</artifactId>
   <version>12.47.0</version>
</dependency>
```

La structure :



Créer des annotations (@Autowired - @Qualifier - @Component)

Annotation Autowired:

Les objets de service doivent utiliser cette annotation

```
import java.lang.annotation.*;
import static java.lang.annotation.ElementType.*;
import static java.lang.annotation.RetentionPolicy.RUNTIME;

@Target({ METHOD, CONSTRUCTOR, FIELD })
@Retention(RUNTIME)
@Documented
public @interface Autowired {

}
```

annotation Component

la classe client doit utiliser cette annotation

```
import java.lang.annotation.*;

@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.TYPE)
public @interface Component {

}
```

annotation Qualifier

Les objets de service doivent utiliser cette annotation, elle peut être utilisée pour éviter les conflits s'il existe plusieurs implémentations de la même interface

```
import java.lang.annotation.*;

@Target({ ElementType.FIELD, ElementType.METHOD,
ElementType.PARAMETER, ElementType.TYPE,
ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Qualifier {
```

```
    String value() default "";
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd" >
<beans>
    <bean id="job" class="me.khaoula.service.imp.JobServiceImp"></bean>
    <bean id="personne" class="me.khaoula.service.imp.PersonneServiceImp">
        <property name="jobService" ref="job"/>
    </bean>
</beans>
```

Basé sur le constructeur:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd" >
<beans>
    <bean id="job" class="me.khaoula.service.imp.JobServiceImp"></bean>
    <bean id="personne" class="me.khaoula.service.imp.PersonneServiceImp">
        <constructor-arg index="0" value="id1"/>
    </bean>
</beans>
```

Interfaces de service

```
public interface JobService {
    Double getJobSalary(String userName);
}
```

```
public interface PersonneService {
    String getPersonneName();
}
```

classes de service

```
import me.khaoula.framework.annotations.Component;
import me.khaoula.service.JobService;

@Component
public class JobServiceImp implements JobService {
    @Override
    public Double getJobSalary(String userName) {
        return null;
    }
}
```

```
import me.khaoula.framework.annotations.Component;
import me.khaoula.service.JobService;
import me.khaoula.service.PersonneService;


@Component
public class PersonneServiceImp implements PersonneService {
    JobService jobService;
    public PersonneServiceImp(Object o) {
    }
    public void setJobService(JobServiceImp jobService) {
    }
    @Override
    public String getPersonneName() {
        return "PersonneName";
    }
}
```

Classe de client

Pour utiliser l'injection des dépendances, la classe client doit utiliser les annotations qu'on a déjà créées pour le client et la classe de service.

```
import me.khaoula.framework.annotations.Autowired;
import me.khaoula.framework.annotations.Component;
import me.khaoula.framework.annotations.Qualifier;
```

```
import me.khaoula.service.JobService;
import me.khaoula.service.PersonneService;
@Component
public class PersonneJobClientComponent {
   @Autowired
   private PersonneService personneService;
   @Autowired
   @Qualifier(value = "jobServiceImp")
   private JobService jobService;

   public void showPersonneJob() {
      String name = personneService.getPersonneName();
      double jobSalary = jobService.getJobSalary(name);
      System.out.println("\n\tPersonne Name: " + name + "\n\tJob salary: " +
jobSalary);
   }
}
```

Classe Injecteur

La classe d'injecteur joue un rôle majeur dans l'injection des dépendances. Parce qu'il est responsable de créer des instances de tous les clients et des instances de connexion automatique pour chaque service dans les classes de clients.

Dans cet injecteur, il peut:

- ✓ Analyser de tous les clients sous le package racine et tous les sous-packages
- ✓ Créer une instance de la classe client.
- ✓ Analyser tous les services utilisant dans la classe client (variables membres, paramètres de constructeur, paramètres de méthode)
- ✓ Rechercher tous les services déclarés à l'intérieur du service lui-même (dépendances imbriquées), de manière récursive
- ✓ Créer une instance pour chaque service renvoyé par les étapes 3 et 4
- ✓ Autowire : injecter (c'est-à-dire initialiser) chaque service avec l'instance créée à l'étape 5
- ✓ Créer Map toutes les classes de clients Map
- ✓ Exposez l'API pour obtenir le getBean(Class classz)/getService(Class classz).

- ✓ Valider s'il y a plusieurs implémentations de l'interface ou s'il n'y a pas d'implémentation
- ✓ Handle Qualifier pour les services ou connexion automatique par type en cas d'implémentations multiples.

```java
public class Injector {
    private Map<Class<?>, Class<?>> diMap;
    private Map<Class<?>, Object> applicationScope;

    private static Injector injector;

    private Injector() {
        super();
        diMap = new HashMap<>();
        applicationScope = new HashMap<>();
    }

    /**
     * Start application
     * @param mainClass
     */
    public static void startApplication(Class<?> mainClass) {
        try {
            synchronized (Injector.class) {
                if (injector == null) {
                    injector = new Injector();
                    injector.initFramework(mainClass);
```

Le test d'application avec une application main:

```java
import me.khaoula.framework.Injector;
import me.khaoula.framework.PersonneJobClientComponent;

public class PersonneJobApplication {
   public static void main(String[] args) {
      long startTime = System.currentTimeMillis();
      Injector.startApplication(PersonneJobApplication.class);
      Injector.getService(PersonneJobClientComponent.class).showPersonneJob();
```

```
        long endime = System.currentTimeMillis();
    }
}
```

avec le fichier xml d'injection basé sur le Setter:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd" >
<beans>
    <bean id="job" class="me.khaoula.service.imp.JobServiceImp"></bean>
    <bean id="personne" class="me.khaoula.service.imp.PersonneServiceImp">
        <property name="jobService" ref="job"/>
    </bean>
</beans>
```

avec le fichier xml d'injection basé sur le Constructeur:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
        "http://www.springframework.org/dtd/spring-beans-2.0.dtd" >
<beans>
    <bean id="job" class="me.khaoula.service.imp.JobServiceImp"></bean>
    <bean id="personne" class="me.khaoula.service.imp.PersonneServiceImp">
        <constructor-arg index="0" value="id1"/>
    </bean>
</beans>
```

déclaration de la classe avec l'annotation Component créée :

```
@Component
public class PersonneServiceImp implements PersonneService {}
```

avec le fichier xml d'injection basé sur l'annotation:

```
@Autowired
JobService jobService;
```

Conclusion

Le principe d'injection des dépendances introduit des interfaces entre une interface et ses dépendances. Cela dissocie la classe de niveau supérieur de ses dépendances afin que nous puissions modifier le code d'une classe de niveau inférieur sans

modifier le code qui l'utilise. Le seul code qui utilise directement une dépendance est celui qui instancie un objet d'une classe spécifique qui implémente l'interface.

La technique d'injection de dépendances nous permet de faire tous celà, aussi il fournit un moyen de séparer la création d'un objet de son utilisation. En faisant cela, nous pouvons remplacer une dépendance sans changer de code et cela réduit également le code dans la logique métier.