

TP : IOC

Framework d'injection des Dépendances

sommaire

Objectives :	2
Rappel :	2
Conception :	3
Bibliothèques Java requises	3
Structure de projet	4
Créer des annotations définies par l'utilisateur	4
Classe d'injecteur	6
Classe d'injecteurXml	6
Exemple d'exécution :	7
Version annotations :	7
Version xml :	9
Fichier de configuration xml	9
Le résultat :	10

Objectives :

Concevoir et créer un mini Framework d'injection des dépendances similaire à Spring IOC

Le Framework doit permettre à un programmeur de faire l'injection des dépendances entre les différents composants 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)

Rappel :

L'injection de dépendance est un modèle de conception utilisé pour implémenter IoC, dans lequel des variables d'instance (c'est-à-dire des dépendances) d'un objet ont été créées et attribuées par le framework.

Pour utiliser DI, il suffit d'ajouter des annotations prédéfinies par l'infrastructure pour utiliser une classe et ses variables d'instance.

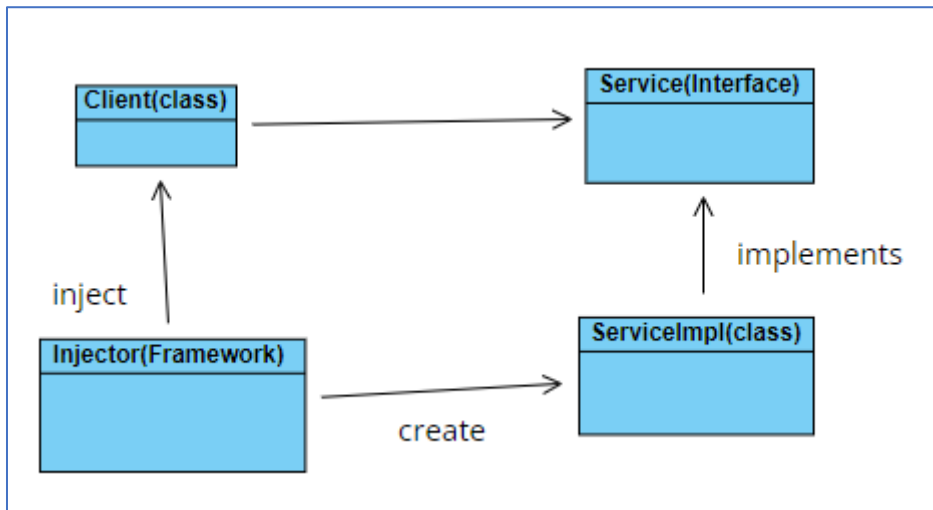
Le modèle d'injection de dépendance implique 3 types de classes.

- **Classe client** : la classe client (classe dépendante) dépend de la classe de service.
- **Classe de service** : classe de service (classe de dépendance) qui fournit un service à la classe cliente.
- **Classe Injector** : la classe injecteur injecte l'objet classe service dans la classe cliente.

De cette façon, le modèle DI sépare la responsabilité de la création d'un objet de la classe de service à partir de la classe cliente. Vous trouverez ci-dessous quelques autres termes utilisés dans DI.

- **Interfaces** qui définissent la façon dont le client peut utiliser les services.
- **L'injection** fait référence au passage d'une dépendance (un service) dans l'objet (un client), également appelé fil automatique.

Conception :



Dans le diagramme de classes ci-dessus, la classe Client qui requiert les objets Service n'instancie pas directement la classe ServiceImpl

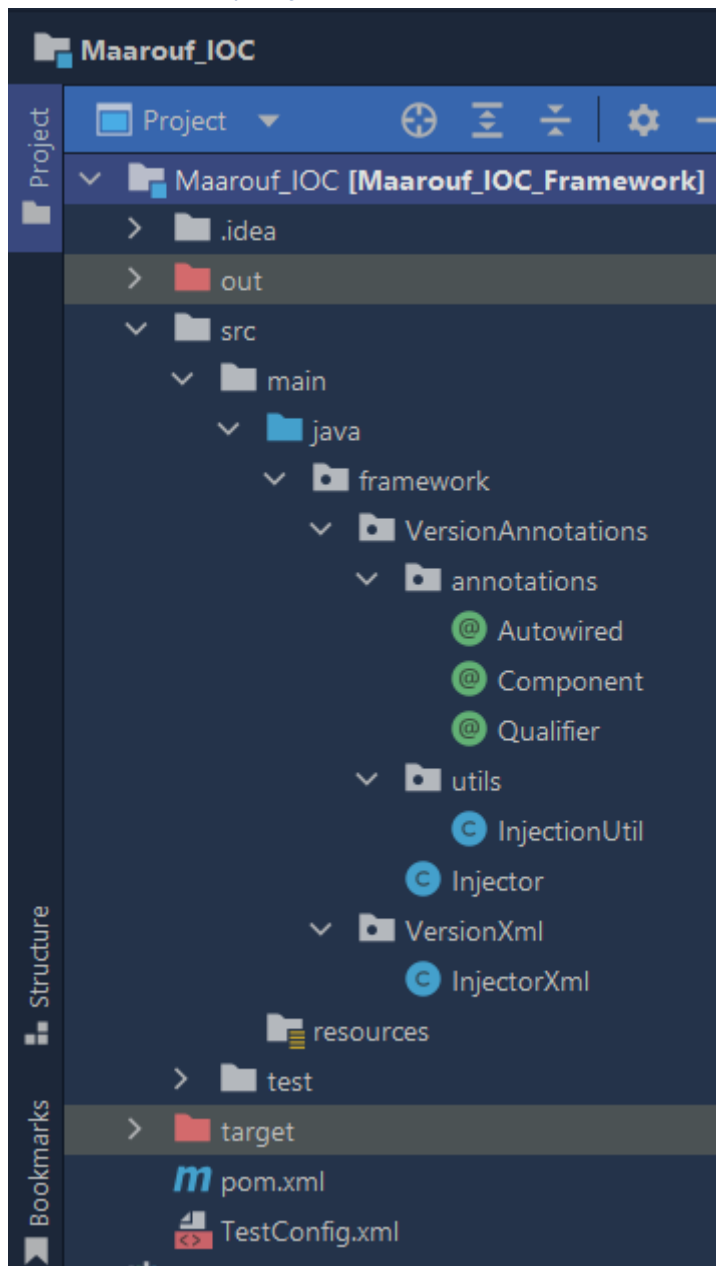
Au lieu de cela, une classe Injector crée les objets et les injecte dans le client, ce qui rend le client indépendant de la façon dont les objets sont créés.

Bibliothèques Java requises

Avant de commencer les étapes de codage, vous pouvez créer un nouveau projet maven dans intelli et ajouter la dépendance Burningwave Core dans *pom.xml* :

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

Structure de projet



Créer des annotations définies par l'utilisateur

Comme décrit ci-dessus, l'implémentation de l'ID doit fournir des annotations prédéfinies, qui peuvent être utilisées lors de la déclaration de variables de classe client et de service à l'intérieur d'une classe client.

Ajoutons des annotations de base, qui peuvent être utilisées :

```
@ Autowired.java x
1 package framework.VersionAnnotations.annotations;
2 import ...
10 @Target({ METHOD, CONSTRUCTOR, FIELD })
11 @Retention(RUNTIME)
12 @Documented
13 public @interface Autowired {
14
15 }
```

```
@ Component.java x
1 package framework.VersionAnnotations.annotations;
2 import ...
6 /**
7  * La classe client doit utiliser cette annotation
8  */
9 @Retention(RetentionPolicy.RUNTIME)
10 @Target(ElementType.TYPE)
11 public @interface Component {
12
13 }
```

```
@ Qualifier.java x
1 package framework.VersionAnnotations.annotations;
2 import java.lang.annotation.*;
3 /**
4  * Les variables de champ de service doivent utiliser cette annotation
5  * Cette annotation peut être utilisée pour éviter les conflits s'il existe plusieurs
6  */
7 @Target({ ElementType.FIELD, ElementType.METHOD,
8           ElementType.PARAMETER, ElementType.TYPE, ElementType.ANNOTATION_TYPE })
9 @Retention(RetentionPolicy.RUNTIME)
10 @Inherited
11 @Documented
12 public @interface Qualifier {
13     String value() default "";
14 }
```

Classe d'injecteur

La classe injector joue un rôle majeur dans le cadre DI. Parce qu'il est responsable de créer des instances de tous les clients et des instances de câblage automatique pour chaque service dans les classes clientes :

1. Analyser tous les clients sous le package racine et tous les sous-packages
2. Créez une instance de classe client.
3. Analyser tous les services utilisés dans la classe cliente (variables membres, paramètres constructeur, paramètres de méthode)
4. Rechercher tous les services déclarés à l'intérieur du service lui-même (dépendances imbriquées), de manière récursive
5. Créer une instance pour chaque service renvoyé aux étapes 3 et 4
6. Autowire : injecter (c'est-à-dire initialiser) chaque service avec l'instance créée à l'étape 5
7. Créer une carte de toutes les classes clientes Carte
8. Exposez l'API pour obtenir `getBean(Class classz)/getService(Class classz)`.
9. Valider s'il existe plusieurs implémentations de l'interface ou s'il n'y a pas d'implémentation
10. Gérer le qualificateur pour les services ou le câblage automatique par type en cas d'implémentations multiples.

RQ : voir le code source de cette classe dans le projet

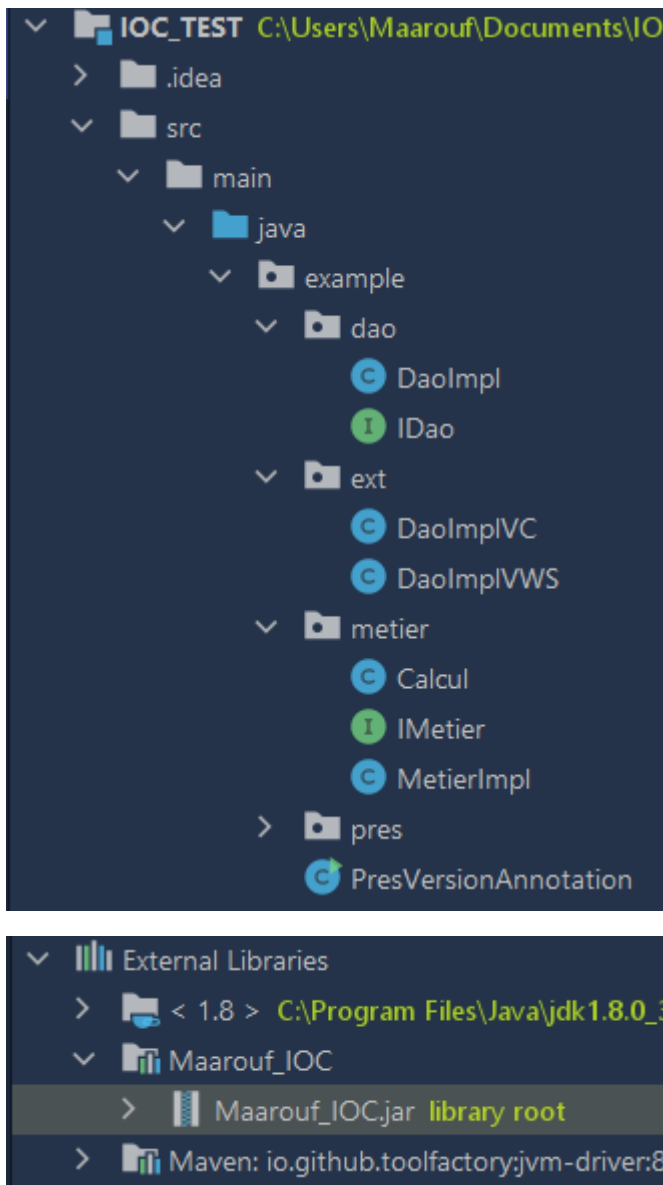
Classe d'injecteurXml

La classe **InjectorXml** permet l'injection via un fichier xml

RQ : voir le code source de cette classe dans le projet

Exemple d'exécution :

Version annotations : on prend le même projet de chapitre 1 :



On doit ajouter notre Framework comme in jar dans le projet

```
DaoImpl.java x
1 package example.dao;
2 import framework.VersionAnnotations.annotations.Component;
3 @Component
4 public class DaoImpl implements IDao {
5     @Override
6     public double getData() {
7         /*
8          * se connecter a la BD pour recuperer la temperature
9          * */
10        System.out.println("version BD");
11        double temp = Math.random()*40;
12        return temp;
13    }
14 }
```

```
MetierImpl.java x
1 package example.metier;
2 import ...
3 @Component
4 public class MetierImpl implements IMetier{
5     @Autowired
6     private IDao dao;
7     @Override
8     public double calcul() {
9         double temp = dao.getData(); //n'importe quel source:classe
10        double res=temp*540/Math.cos(temp*Math.PI);
11        return res;
12    }
13    public MetierImpl() {}
14    public MetierImpl(IDao dao) { this.dao = dao; }
15    //permet d'injecter dans la variable dao un obj d'une classe qui implemente l'interface IDao
16    public void setDao(IDao dao) { this.dao = dao; }
17 }
18
19
20
21
22
23
24
25
```



```

PresVersionAnnotation.java x
1 package example;
2 import example.metier.IMetier;
3 import framework.VersionAnnotations.Injector;
4 public class PresVersionAnnotation {
5     public static void main(String[] args) {
6         Injector.startApplication(PresVersionAnnotation.class);
7         System.out.println(Injector.getService(IMetier.class).calcul());
8     }
9 }

```

```

2022-04-03 09:25:15.415 [main] -
version BD
1428.7238304295538

```

Version xml :

Fichier de configuration xml

```

TestConfig.xml x
1 <?xml version="1.0" encoding="UTF-8"?>
2 <Beans>
3     <bean id="dao" class="example.dao.DaoImpl"></bean>
4     <bean id="metier" class="example.metier.MetierImpl">
5         <constructor-arg index="0" value="dao"/>
6         <!-- <property name="dao" ref="dao"></property> -->
7     </bean>
8 </Beans>

```

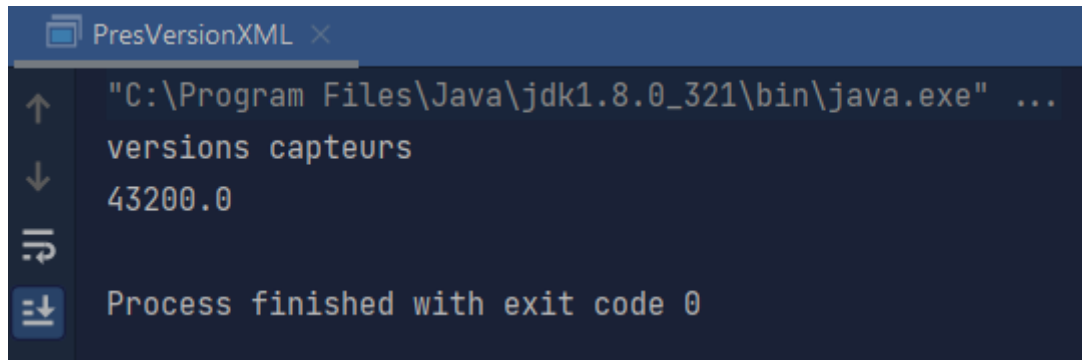
La fonction main

```

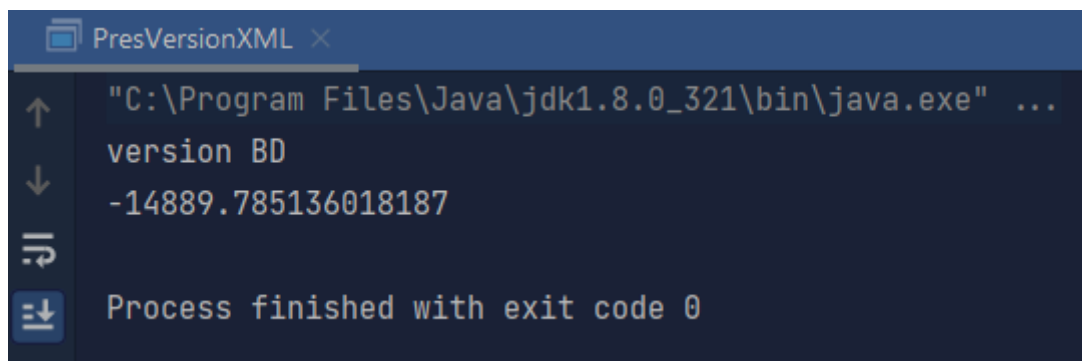
package example.pres;
import ...
public class PresVersionXML {
    public static void main(String[] args) throws Exception{
        InjectorXml context =new InjectorXml( fileName: "TestConfig.xml");
        IMetier metier=(IMetier) context.getBean( BeanName: "metier");
        System.out.println(metier.calcul());
    }
}

```

Le résultat :



```
PresVersionXML x
"C:\Program Files\Java\jdk1.8.0_321\bin\java.exe" ...
versions capteurs
43200.0
Process finished with exit code 0
```



```
PresVersionXML x
"C:\Program Files\Java\jdk1.8.0_321\bin\java.exe" ...
version BD
-14889.785136018187
Process finished with exit code 0
```