

2021-2022

**Département Mathématiques et Informatique
GLSID 2 – S4**

Architecture JEE et Middlewares

Rapport de mini projet
Framework d'Injection des dépendances

**Préparé par : Fatima Zahra HASBI
Encadré par : Mr Mohamed YOUSSEFI**

Sommaire

Introduction	2
Qu'est-ce que l'injection de dépendance ?.....	3
Qu'est-ce que l'inversion de contrôle ?.....	3
Types d'injection de dépendance ?.....	4
Diagramme de classes du modèle de conception d'injection de dépendance	4
Initialisation du projet	5
Bibliothèques Java requises ?	5
Création des annotations user-defined	6
Interfaces de service.....	7
Classes de Service	8
Classe Injector.....	8
Classe Client.....	10
Application Classe main	11
Conclusion.....	12

Introduction

Ce rapport porte sur le mini projet Framework d'injection de dépendances, il consiste à 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 composant de son application respectant les possibilités suivantes :

- 1- A travers un fichier XML de configuration en utilisant Jax Binding (OXM : Mapping Objet XML)
- 2- En utilisant les annotations
- 3- Possibilité d'injection via :
 - Le constructeur
 - Le Setter
 - Attribut (accès direct à l'attribut : Field)

Qu'est-ce que l'injection de dépendance ?

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

Pour utiliser la fonctionnalité DI, une classe et ses variables d'instance ont juste besoin d'ajouter des annotations prédéfinies par le Framework.

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 : La classe de service (classe de dépendance) qui fournit le service à la classe client.
- Classe d'injecteur : la classe d'injecteur injecte l'objet de classe de service dans la classe client.

De cette manière, le modèle DI sépare la responsabilité de la création d'un objet de la classe de service de la classe client.

Plus de termes utilisés dans DI :

- Interfaces : qui définissent comment le client peut utiliser les services.
- Injection : fait référence au passage d'une dépendance (un service) dans l'objet (un client), on l'appelle aussi auto Wire.

Qu'est-ce que l'inversion de contrôle ?

En bref, "Ne nous appelez pas, nous vous appellerons."

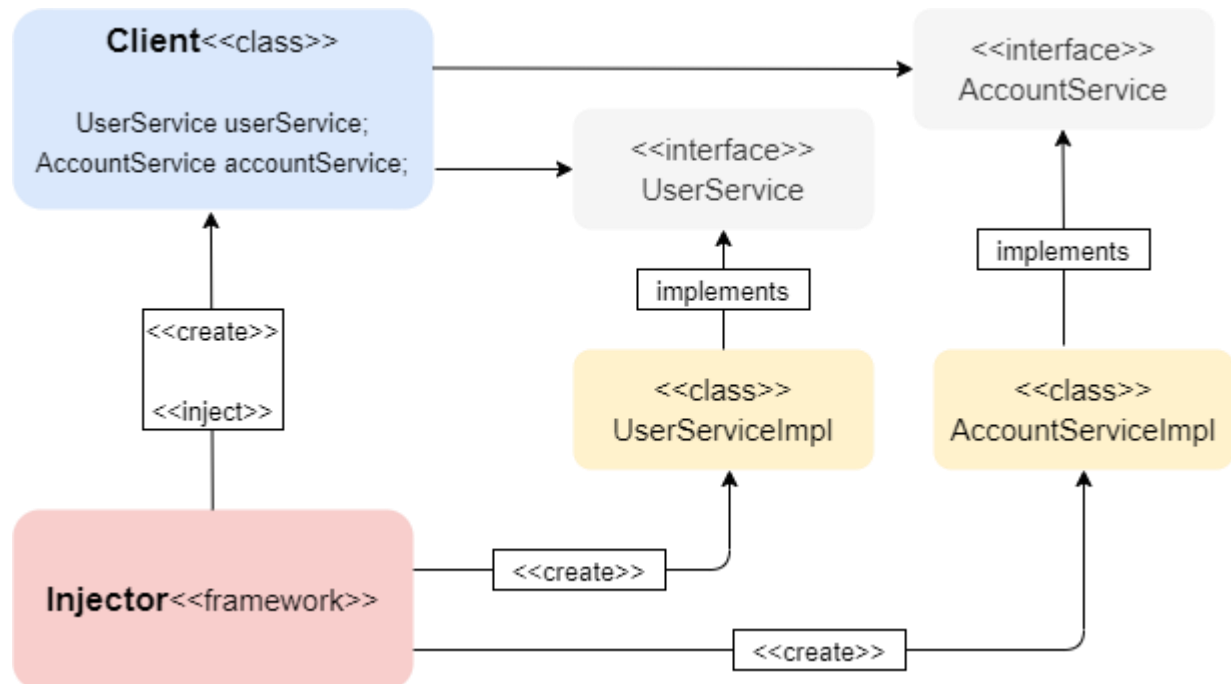
- L'inversion de contrôle (Ioc) est un principe de conception. Il est utilisé pour inverser différents types de contrôles (c'est-à-dire la création d'objets ou la création et la liaison d'objets dépendants) dans la conception orientée objet pour obtenir un couplage faible.
- L'injection de dépendances est l'une des approches pour mettre en œuvre l'Ioc.
- Ioc aide à dissocier l'exécution d'une tâche de sa mise en œuvre.
- Ioc l'aide à concentrer un module sur la tâche pour laquelle il est conçu.
- Ioc évite les effets secondaires lors du remplacement d'un module.

Types d'injection de dépendance ?

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

Dans ce cas, l'annotation Autowired a été ajoutée lors de la déclaration de la méthode.

Diagramme de classes du modèle de conception d'injection de dépendance

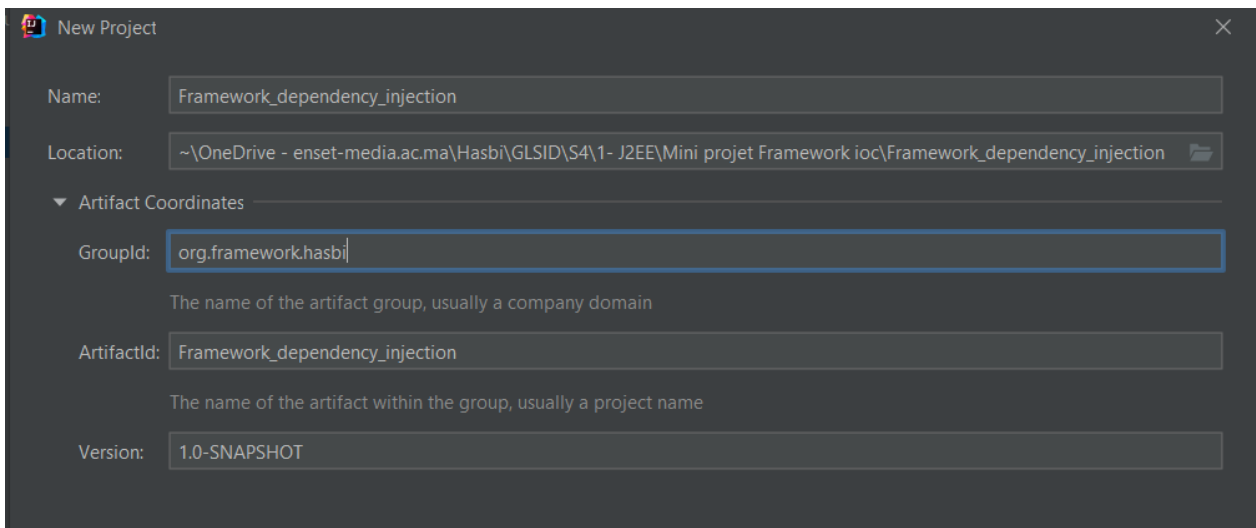


Dans le diagramme de classes ci-dessus, la classe Client qui requiert les objets UserService et AccountService n'instancie pas directement les classes UserServiceImpl et AccountServiceImpl.

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.

Initialisation du projet

Création d'un nouveau projet maven.



The screenshot shows the 'New Project' dialog box with the following fields and values:

- Name: Framework_dependency_injection
- Location: ~\OneDrive - enset-media.ac.ma\Hasbi\GLSID\S4\1- J2EE\Mini projet Framework ioc\Framework_dependency_injection
- Artifact Coordinates section:
 - GroupId: org.framework.hasbi (with a tooltip: 'The name of the artifact group, usually a company domain')
 - ArtifactId: Framework_dependency_injection (with a tooltip: 'The name of the artifact within the group, usually a project name')
 - Version: 1.0-SNAPSHOT

Bibliothèques Java requises ?

Ajouter dans le fichier pom.xml la dépendance Burningwave Core.

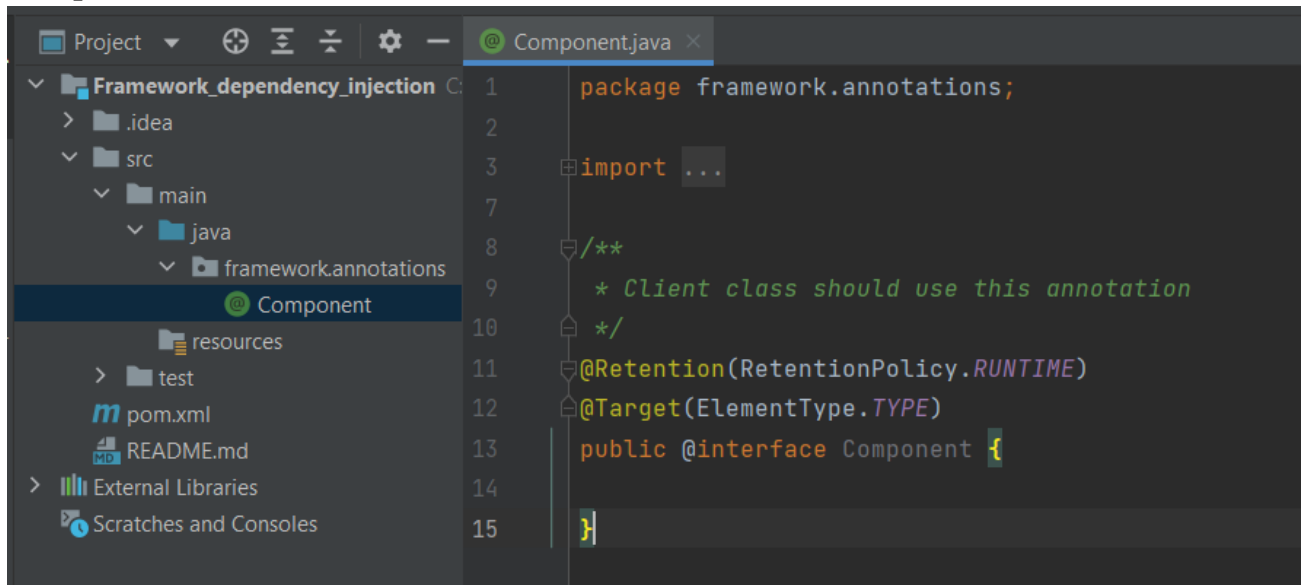
```
<dependencies>
  <dependency>
    <groupId>org.burningwave</groupId>
    <artifactId>core</artifactId>
    <version>12.47.0</version>
  </dependency>
</dependencies>
```

Création des annotations user-defined

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

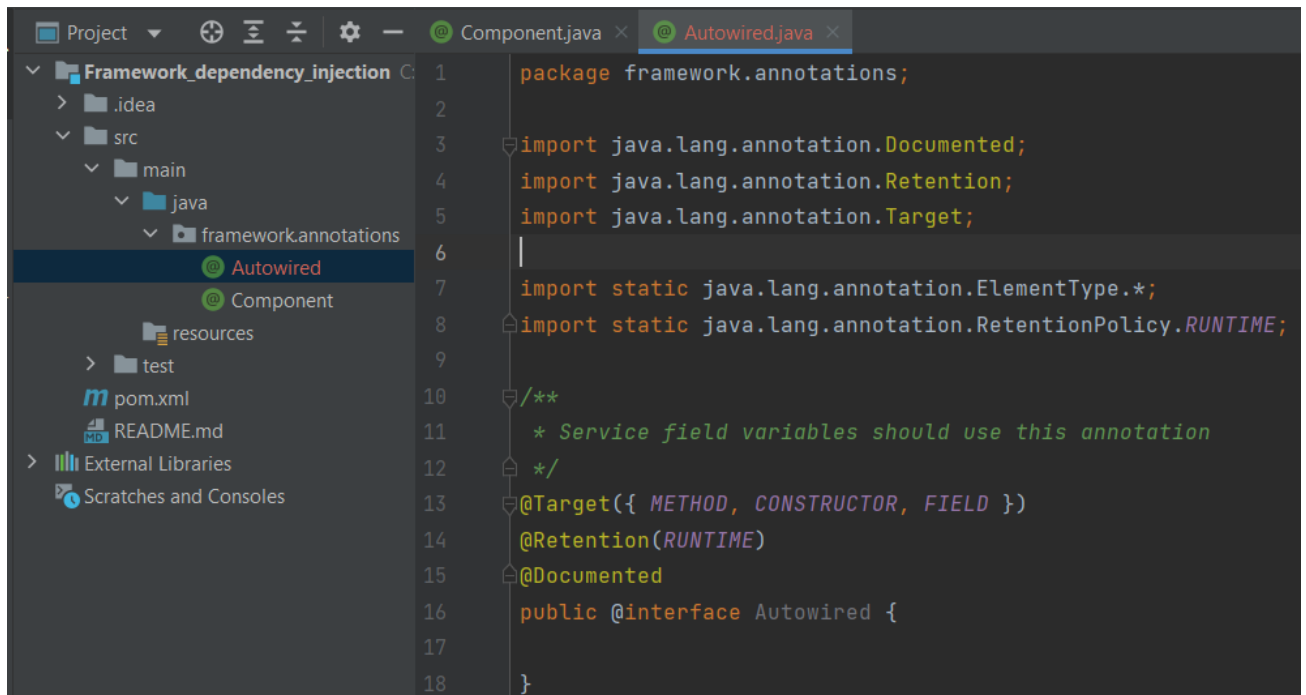
Ajoutons des annotations de base, qui peuvent être utilisées par les classes client et service:

Component



```
1 package framework.annotations;
2
3 import ...
4
5 /**
6  * Client class should use this annotation
7  */
8 @Retention(RetentionPolicy.RUNTIME)
9 @Target(ElementType.TYPE)
10 public @interface Component {
11 }
12
```

Autowired



```
1 package framework.annotations;
2
3 import java.lang.annotation.Documented;
4 import java.lang.annotation.Retention;
5 import java.lang.annotation.Target;
6
7 import static java.lang.annotation.ElementType.*;
8 import static java.lang.annotation.RetentionPolicy.RUNTIME;
9
10 /**
11  * Service field variables should use this annotation
12  */
13 @Target({ METHOD, CONSTRUCTOR, FIELD })
14 @Retention(RUNTIME)
15 @Documented
16 public @interface Autowired {
17 }
18
```

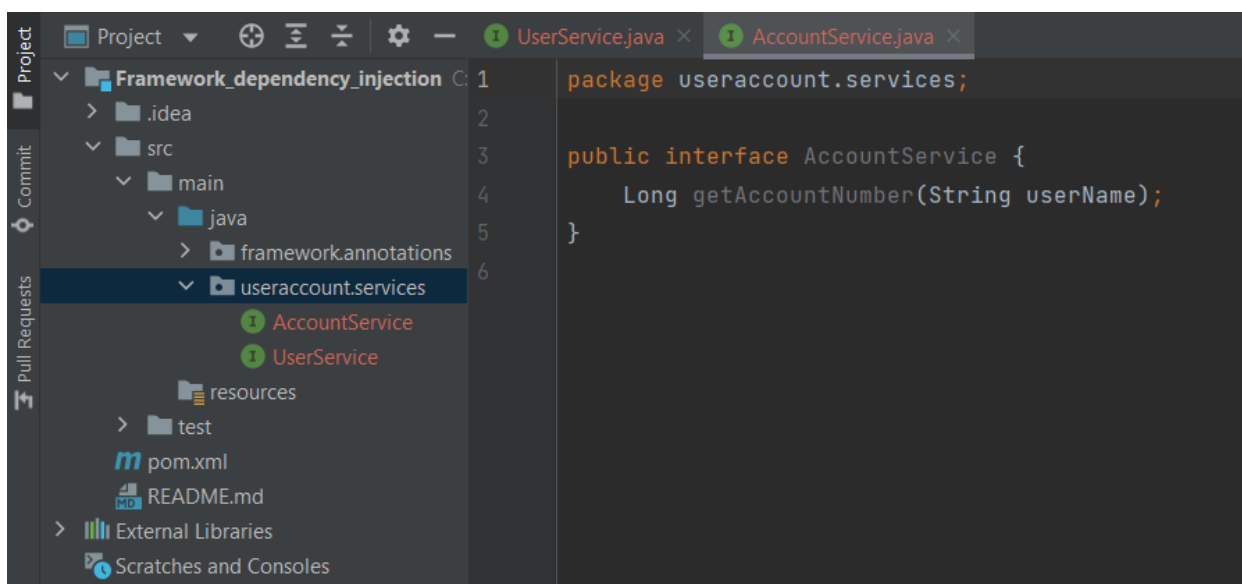
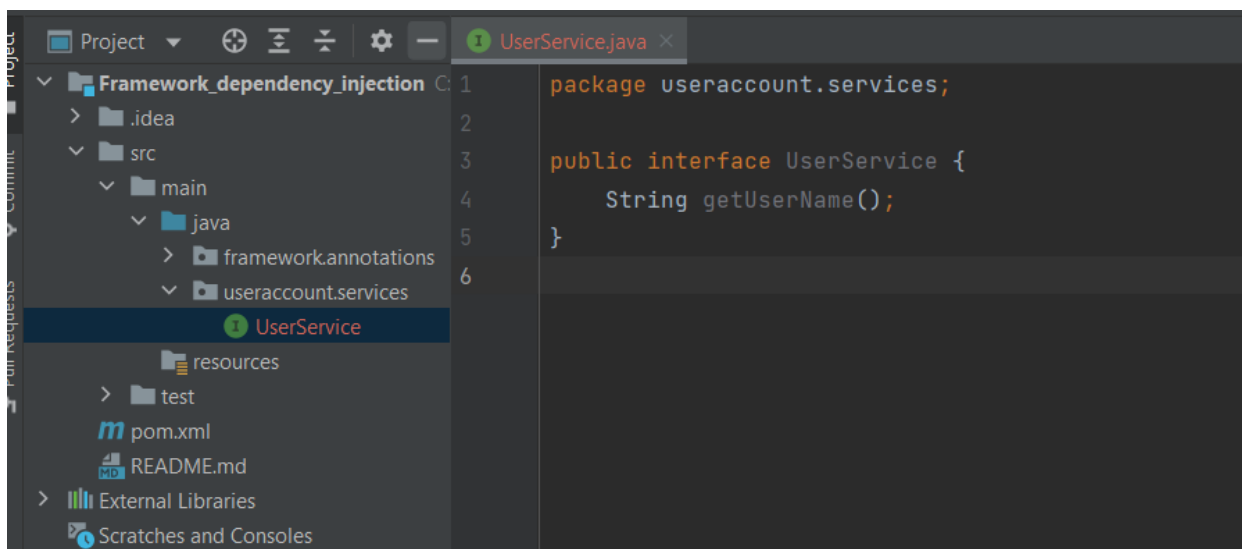
Qualifier

```
package framework.annotations;

import java.lang.annotation.*;

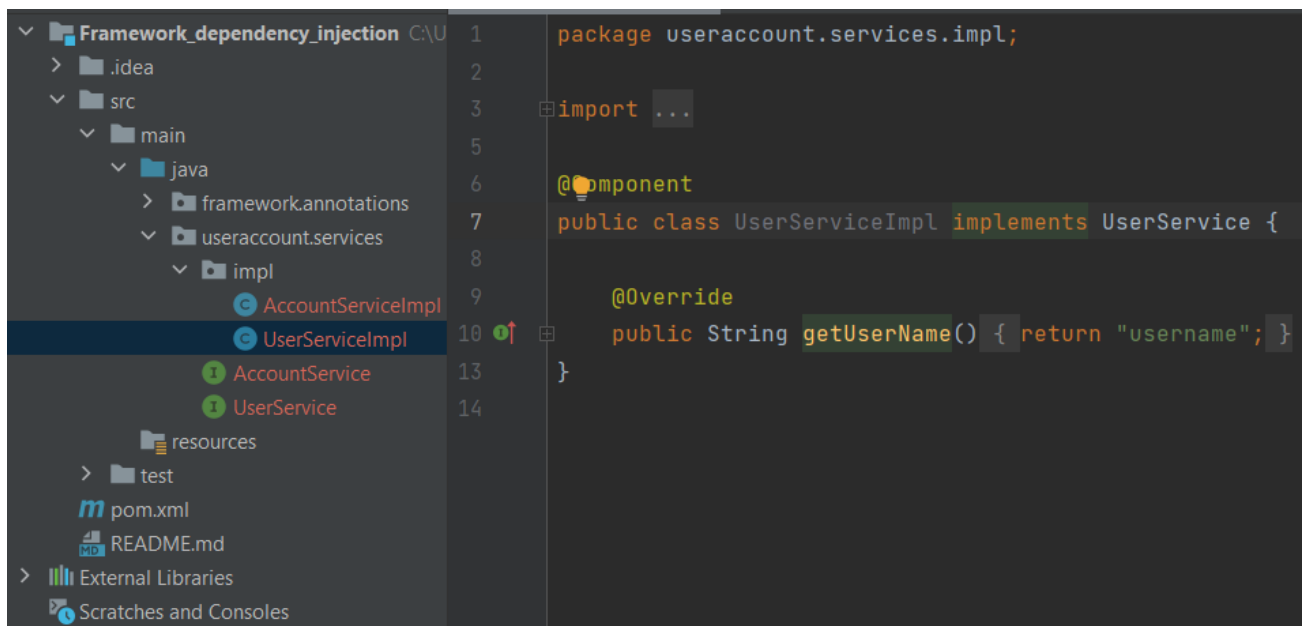
/**
 * Service field variables should use this annotation
 * This annotation Can be used to avoid conflict if there are multiple implementations of the same interface
 */
@Target({ ElementType.FIELD, ElementType.METHOD, ElementType.PARAMETER, ElementType.TYPE, ElementType.ANNOTATION_TYPE })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
@Documented
public @interface Qualifier {
    String value() default "";
}
```

Interfaces de service

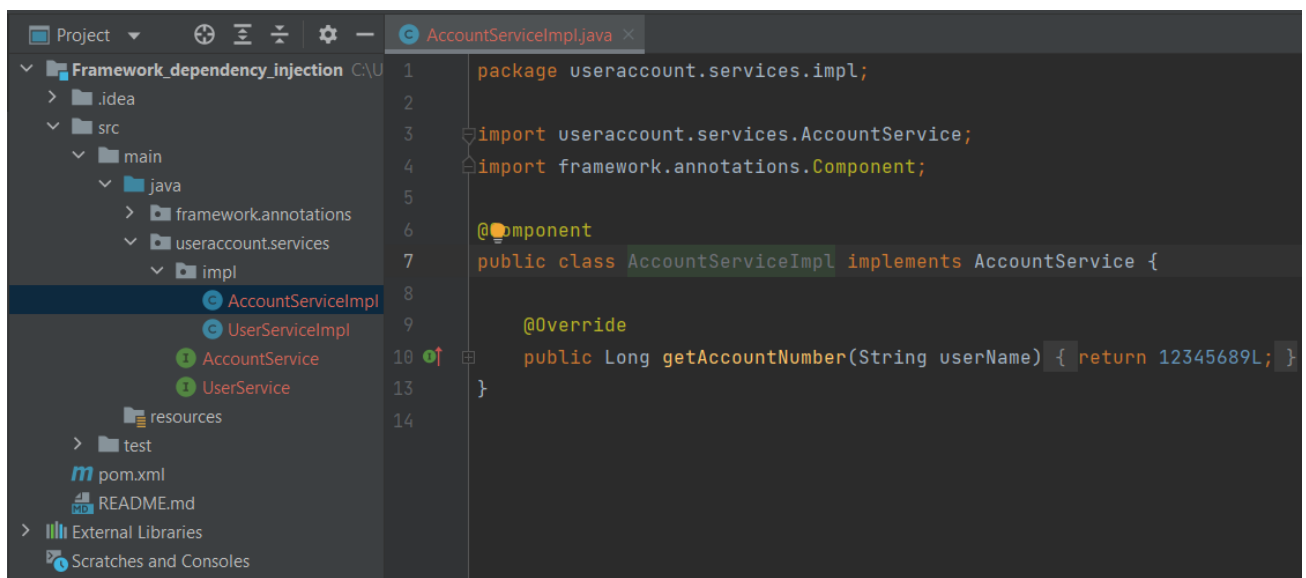


Classes de Service

Ces classes implémentent des interfaces de service et utilisent des annotations DI.



```
1 package useraccount.services.impl;
2
3 import ...
4
5
6 @Component
7 public class UserServiceImpl implements UserService {
8
9     @Override
10    public String getUsername() { return "username"; }
11
12 }
13
14
```



```
1 package useraccount.services.impl;
2
3 import useraccount.services.AccountService;
4 import framework.annotations.Component;
5
6 @Component
7 public class AccountServiceImpl implements AccountService {
8
9     @Override
10    public Long getAccountNumber(String userName) { return 12345689L; }
11
12 }
13
14
```

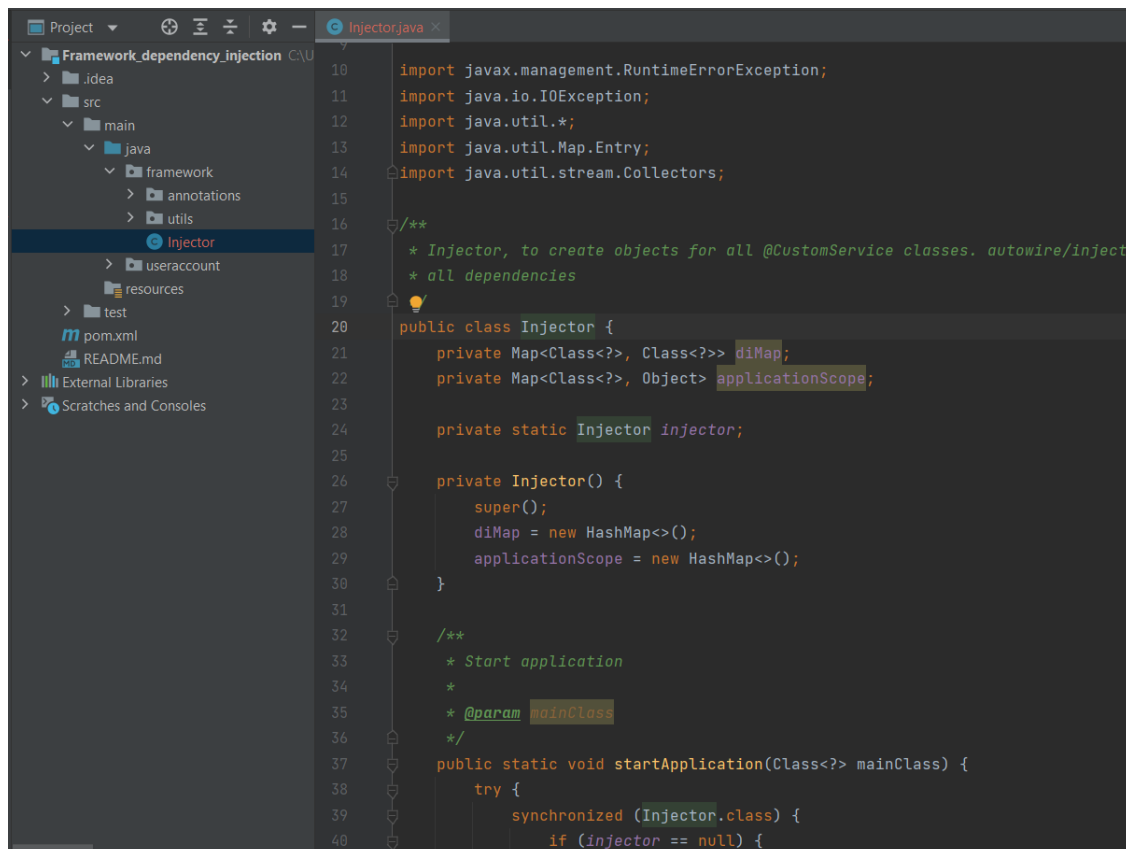
Classe Injector

La classe d'injecteur 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 connexion automatique pour chaque service dans les classes de clients.

Pas :

- Analyser 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
- Exposer 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.

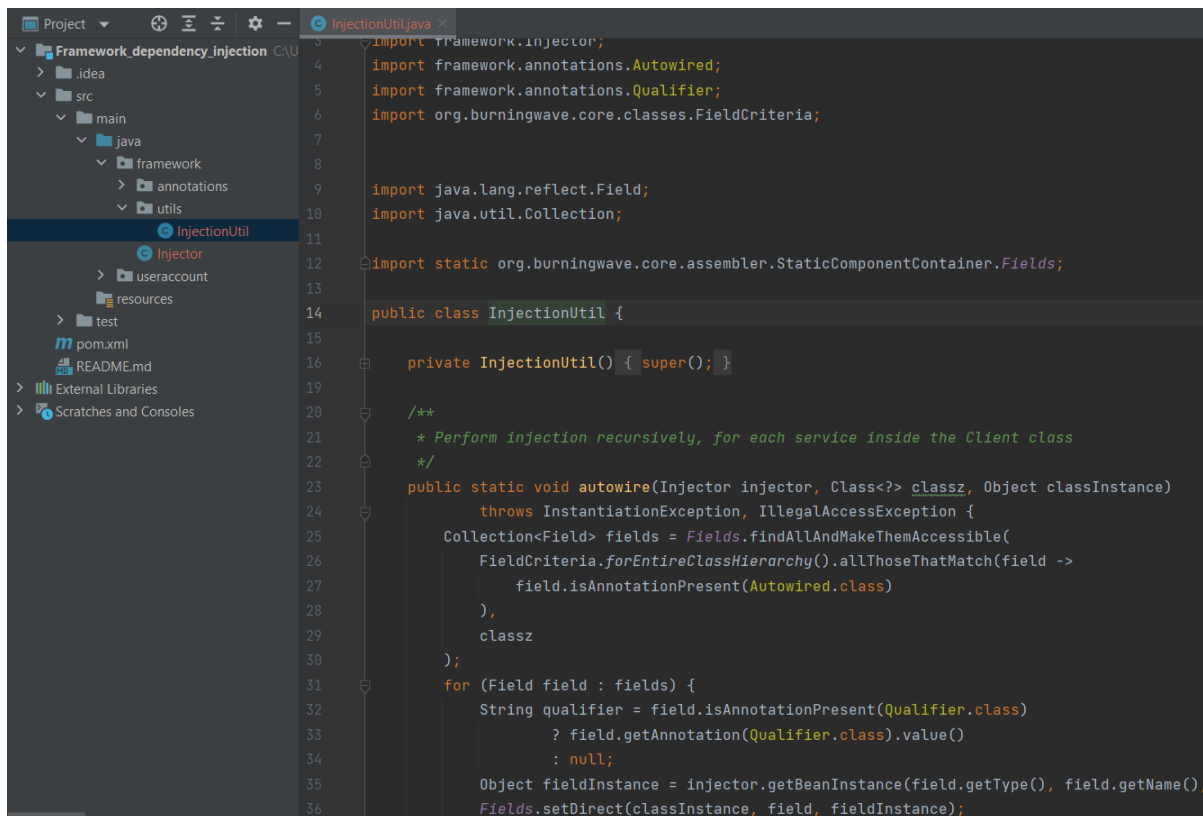
Cette classe utilise fortement la méthode de base fournie par `java.lang.Class` et `org.burningwave.classes.ClassHunter`



```
10 import javax.management.RuntimeErrorException;
11 import java.io.IOException;
12 import java.util.*;
13 import java.util.Map.Entry;
14 import java.util.stream.Collectors;
15
16 /**
17  * Injector, to create objects for all @CustomService classes. autowire/inject
18  * all dependencies
19  */
20 public class Injector {
21     private Map<Class<?>, Class<?>> diMap;
22     private Map<Class<?>, Object> applicationScope;
23
24     private static Injector injector;
25
26     private Injector() {
27         super();
28         diMap = new HashMap<>();
29         applicationScope = new HashMap<>();
30     }
31
32     /**
33      * Start application
34      *
35      * @param mainClass
36      */
37     public static void startApplication(Class<?> mainClass) {
38         try {
39             synchronized (Injector.class) {
40                 if (injector == null) {
```

Cette classe utilise fortement la méthode de base fournie par le `java.lang.reflect.Field`.

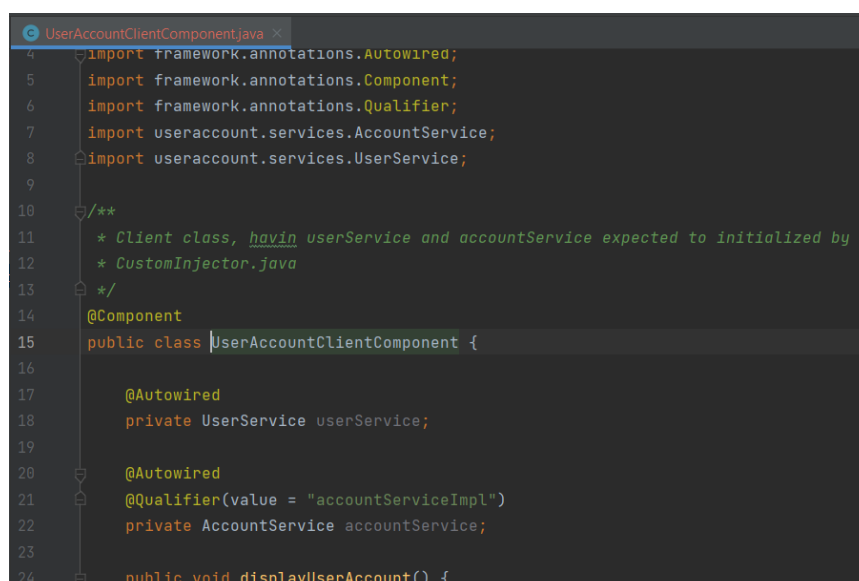
La méthode `autowire()` de cette classe est une méthode récursive car elle se charge d'injecter les dépendances déclarées à l'intérieur des classes de service (c'est-à-dire les dépendances imbriquées) :



```
1 import framework.Injector;
2 import framework.annotations.Autowired;
3 import framework.annotations.Qualifier;
4 import org.burningwave.core.classes.FieldCriteria;
5
6 import java.lang.reflect.Field;
7 import java.util.Collection;
8
9 import static org.burningwave.core.assembler.StaticComponentContainer.Fields;
10
11 public class InjectionUtil {
12     private InjectionUtil() { super(); }
13
14     /**
15      * Perform injection recursively, for each service inside the Client class
16      */
17     public static void autowire(Injector injector, Class<?> classz, Object classInstance)
18         throws InstantiationException, IllegalAccessException {
19         Collection<Field> fields = Fields.findAllAndMakeThemAccessible(
20             FieldCriteria.forEntireClassHierarchy().allThoseThatMatch(field ->
21                 field.isAnnotationPresent(Autowired.class)
22             ),
23             classz
24         );
25         for (Field field : fields) {
26             String qualifier = field.isAnnotationPresent(Qualifier.class)
27                 ? field.getAnnotation(Qualifier.class).value()
28                 : null;
29             Object fieldInstance = injector.getBeanInstance(field.getType(), field.getName(),
30                 Fields.setDirect(classInstance, field, fieldInstance));
31         }
32     }
33 }
```

Classe Client

Pour utiliser les fonctionnalités DI, la classe client doit utiliser des annotations prédéfinies fournies par le Framework DI pour le client et la classe de service.



```
1 import framework.annotations.Autowired;
2 import framework.annotations.Component;
3 import framework.annotations.Qualifier;
4 import useraccount.services.AccountService;
5 import useraccount.services.UserService;
6
7 /**
8  * Client class, having userService and accountService expected to initialized by
9  * CustomInjector.java
10 */
11 @Component
12 public class UserAccountClientComponent {
13
14     @Autowired
15     private UserService userService;
16
17     @Autowired
18     @Qualifier(value = "accountServiceImpl")
19     private AccountService accountService;
20
21     public void displayUserAccount() {
22     }
23 }
```

Application Classe main

```
UserAccountApplication.java
1 package useraccount;
2
3 import framework.Injector;
4
5 public class UserAccountApplication {
6
7     public static void main(String[] args) {
8         long startTime = System.currentTimeMillis();
9         Injector.startApplication(UserAccountApplication.class);
10        Injector.getService(UserAccountClientComponent.class).displayUserAccount();
11        long endTime = System.currentTimeMillis();
12        System.out.println("\tElapsed time: " + getFormattedDifferenceOfMillis(endTime, startTime) + " seconds\n");
13    }
14
15    @ static String getFormattedDifferenceOfMillis(long value1, long value2) {
16        String valueFormatted = String.format("%04d", (value1 - value2));
17        return valueFormatted.substring(0, valueFormatted.length() - 3) + "," + valueFormatted.substring(valueFormatted.length() - 3);
18    }
19 }
20
```

Conclusion

Ce projet nous a donné une compréhension claire du fonctionnement des dépendances DI ou Autowired.

Avec l'implémentation de notre propre Framework DI, nous n'avons pas besoin de Frameworks lourds, si on n'utilise vraiment pas la plupart de leurs fonctionnalités dans notre application, comme les exécutions de méthodes de gestion du cycle de vie des Bean et bien plus de choses lourdes.

On peut faire beaucoup de choses comme :

- Ajouter plus d'annotations définies par l'utilisateur à des fins diverses (par exemple, comme les portées de Bean singleton, prototype, demande, session, session globale)
- Configuration de la configurabilité avec des espaces réservés
- Définition de la configurabilité avec les propriétés codées Java
- Génération de classes à l'exécution

Et bien d'autres fonctionnalités.

Le lien du projet sur GitHub :

<https://github.com/FatimaZahraHASBI/Framework-Dependency-Injection.git>