ECOLE NORMALE SUPÉRIEURE DE L'ENSEIGNEMENT
TECHNIQUE DE MOHAMMEDIA
UNIVERSITÉ HASSAN II DE CASABLANCA

Département Mathématique informatique

Rapport

filière :

“ Ingénierie Informatique - Big Data & Cloud Computing ”

II-BDCC

Framework pour l’injection des dépendances

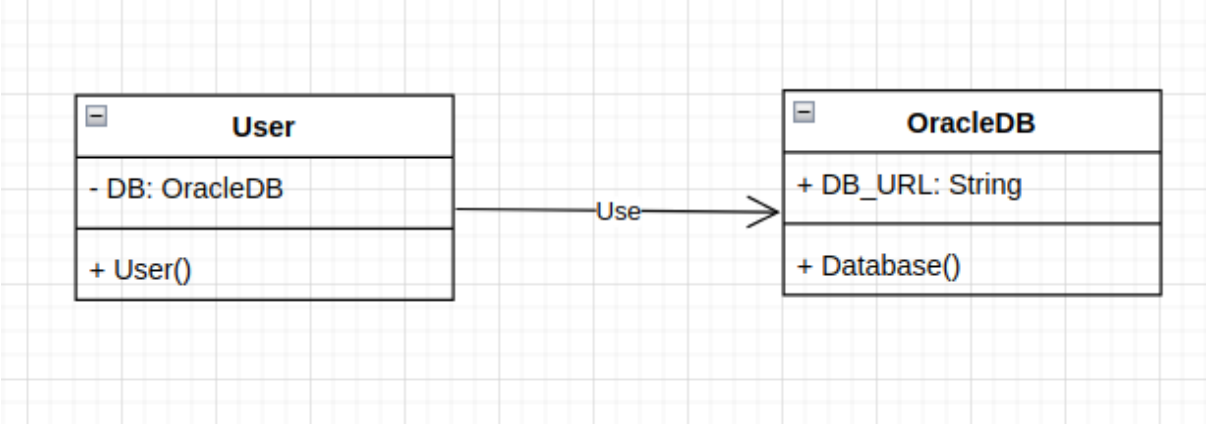
Réalisation :

Zakaria Mansouri

Année Universitaire : 2022 - 2023

Problématique:

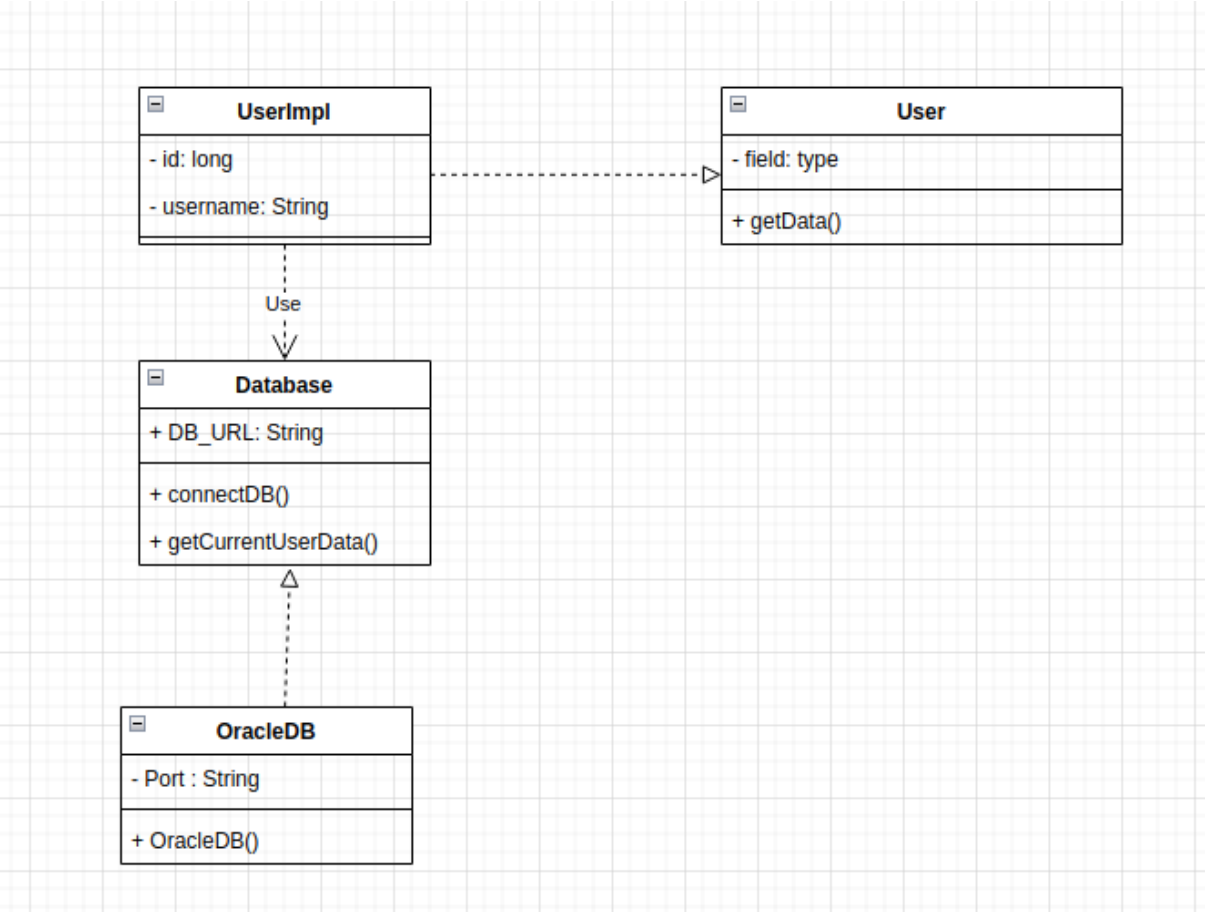
L'idée c'est de créer un framework qui nous permet de créer des applications faciles à maintenir . par l'utilisation du concept de l'inversion de contrôle et l'injection des dépendances .



dans l'exemple ci dessus on a utilisé le couplage forte , si on veut migrer vers une autre database comme postgresql on doit changer complètement le code c'est pour cela on doit utiliser le couplage faible qui se base sur la règle suivante :

"Dépendez les interfaces et ne dépendez pas les classes"

donc on va généraliser la classe OracleDB :



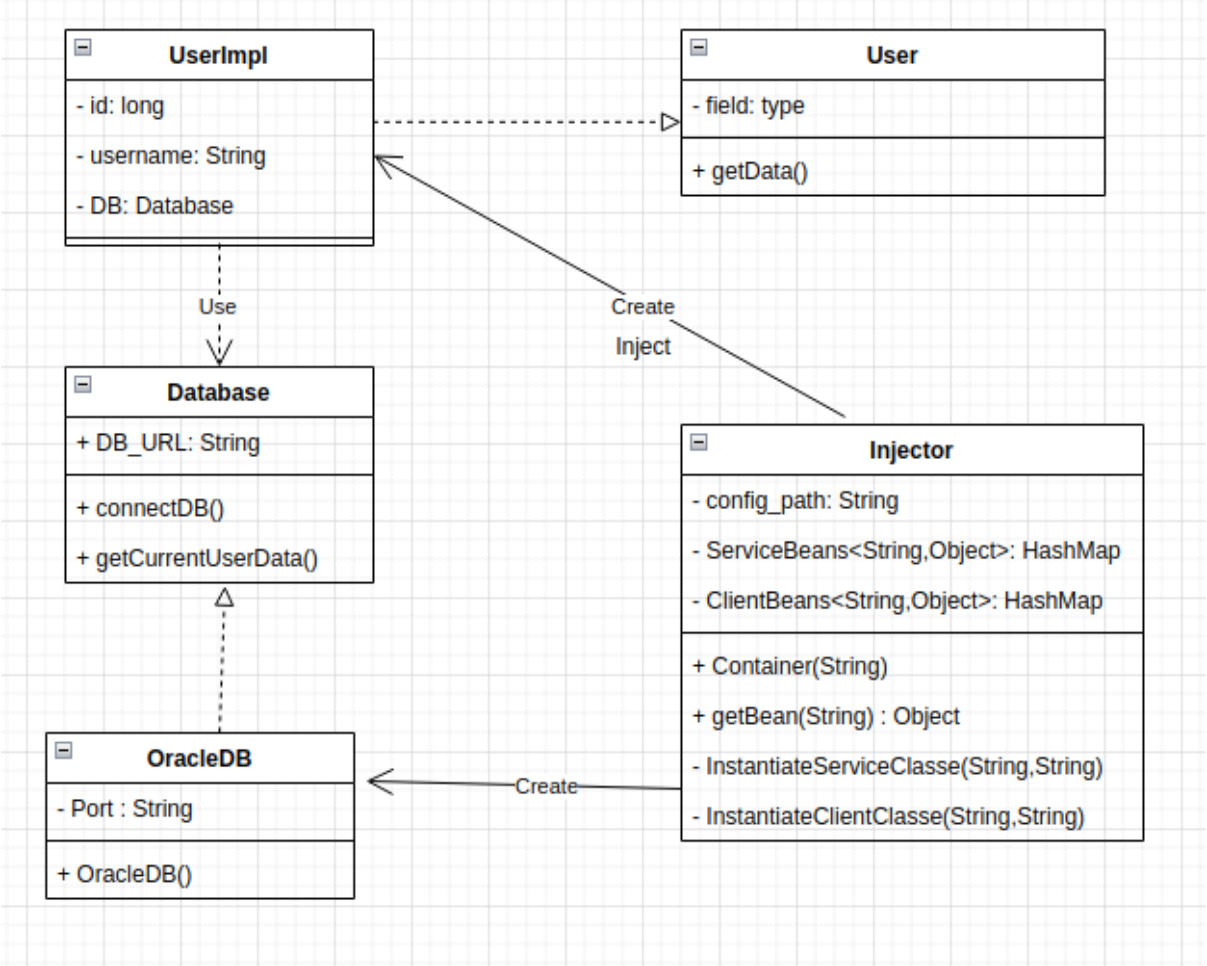
Partie 1 :

Maintenant on va créer notre framework pour injecter les dépendances via un fichier de configuration xml :

➔ fichier config.xml :

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans [
        <!ELEMENT beans (bean)*>
        <!ELEMENT bean (property)*>
        <!ATTLIST bean
                class CDATA #REQUIRED
                name CDATA #IMPLIED>
        <!ELEMENT property (#PCDATA)>
        <!ATTLIST property
                name CDATA #REQUIRED
                ref CDATA #REQUIRED>
        ]>
<beans>
    <bean class="dao.OracleDB" name="db" />
    <bean class="metier.UserImpl" name="user">
        <property name="Database" ref="db"/>
    </bean>
</beans>
```

➔ **l'architecture de notre framework :**



➔ Classe Injector :

d'après le fichier de configuration fourni en paramètre dans le constructeur , la classe va créer et instancier les classes services et clients , puis elle va injecter les classes services nécessaires pour que les classes clients puissent fonctionner

```
public class Injector {
    private String config_path;
    private HashMap<String, Object> ServiceBeans = new HashMap<String, Object>();
    private HashMap<String, Object> ClientBeans = new HashMap<String, Object>();

    public Injector(String config_path) {
        List<String> clientsclasses = new ArrayList<String>();
        List<String> servicesclasses= new ArrayList<String>();
        this.config_path = config_path;


        try {
            DocumentBuilder db =  DocumentBuilderFactory.newInstance().newDocumentBuilder();
            org.w3c.dom.Document doc = db.parse(new File(config_path));
            XPath xPath = XPathFactory.newInstance().newXPath();
            XPathExpression expression_beans = xPath.compile("//bean");
            final NodeList beans = (NodeList) expression_beans.evaluate(doc, XPathConstants.NODESET);
            Element bean;
            for (int i = 0; i < beans.getLength(); i++) {
                bean = (Element) beans.item(i);

                if (!bean.hasChildNodes()) {
                    String beanname = bean.getAttribute("name").equals("") ? bean.getAttribute("class")
: bean.getAttribute("name");
                    String classname = bean.getAttribute("class");
                    InstantiateServiceClasse(classname,beanname);
                }
                else
                {
                    String beanname = bean.getAttribute("name").equals("") ? bean.getAttribute("class")
: bean.getAttribute("name");
                    String classname = bean.getAttribute("class");
                    String ref,name;
                    Node node = bean.getElementsByTagName("property").item(0);
                    Element property = (Element) node;
                    ref = property.getAttribute("ref"); //ref of object to be injected
                    name = property.getAttribute("name");
                    InstantiateClientClasse(classname,beanname);
                    Object serviceObject = getBean(ref);
                    Object clientobject = getBean(beanname);
                    String methodname = "set" + StringUtils.capitalize(name);
                    Method methods[] = clientobject.getClass().getMethods();
                    Method method=null;
                    for (Method method_ : methods) {
                        if (methodname.equals(method_.getName())) {
                            method = method_;
                        }
                    }
                    method.invoke(clientobject,serviceObject);

                }
            }
        } catch (Exception e) {
            e.printStackTrace();

        }
```

➔ Fonction getBean(String beanname)

permet de retourner une instanciation d’un objet par leur nom et cela à partir des HashMap ClientsBeans et ServicesBeans :

```java
public Object getBean(String beaname) {
        Object object=null;
        for (HashMap.Entry<String, Object> entry : ServiceBeans.entrySet()) {
            if (entry.getKey().equals(beaname)) {
                object= entry.getValue();
                return object;
            }
        }
        for (HashMap.Entry<String, Object> entry : ClientBeans.entrySet()) {
            if (entry.getKey().equals(beaname)) {
                object = entry.getValue();
                return object;
            }
        }
        return object;
    }
```

➔ Fonction InstantiateServiceClasse(String classname,String BeanName)

permet d'instancier une classe service :

```java
    private void InstantiateServiceClasse(String classname,String BeanName) {
        try {
            Class beanclasse = Class.forName(classname);
            Object beanobj = beanclasse.newInstance();
            ServiceBeans.put(BeanName, beanobj);
        } catch (Exception e) {
            e.printStackTrace();
        }

    }
```

➔ Fonction InstantiateClientClasse(String classname,String BeanName)

```
private void InstantiateClientClasse(String classname, String BeanName) {
        try {
            Class beanclasse = Class.forName(classname);
            Object beanobj = beanclasse.newInstance();
            ClientBeans.put(BeanName, beanobj);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

Test du framework :

➔ Création de l'interface Database :

```
1 package dao;
2
3 public interface Database {
4     String getUserData(String username);
5 }
6
```

➔ Création du classe OracleDB qui implémente la classe Database:

```
1 package dao;
2
3 public class OracleDB implements Database {
4
5     public String getUserData(String username) {
6         System.out.println("getting your info from oracle
  database");
7         System.out.println("wait...");
8         System.out.println("success");
9         System.out.println("Username is : "+username);
10        return username;
11    }
12 }
```

➔ Création de l’interface User :

```java
package metier;

public interface User {

    void getCurrentUserData();
}

```

➔ Création du classe UserImpl qui implémente l’interface User :

pour faire l’injection des dépendances on va utiliser le setter donc il faut juste déclarer dans notre cas l’objet database et ajouter le setter .

Attention ! : le nom doit être comme dans l’attribut name de l'Élément property dans le fichier xml

```java
package metier;

import dao.Database;

public class UserImpl implements User {
    private long id ;
    private String name = "zakaria";
    private String password = "mansouri";
    private Database Database;

    public UserImpl() {
    }
    public void setDatabase(dao.Database database) {
        Database = database;
    }
    public void getCurrentUserData() {
        Database.getUserData("zakaria");
    }
}

```

➔ Création du classe Application pour tester notre Framework :

```java
import metier.User;
import xml.Injector;

public class Application {
    public static void main(String args[]) {
        Injector injector = new Injector("config.xml");
        User user = (User) injector.getBean("user");
        user.getCurrentUserData();
    }
}

```

➔ Exécution :

```
/usr/lib/jvm/java-8-openjdk/bin/java ...
getting your info from oracle database
wait...
success
Username is : zakaria

Process finished with exit code 0
```

Si on change le fichier config.xml :

```xml
<beans>
    <bean class="dao.PostgreDB" name="db" />
    <bean class="metier.UserImpl" name="user">
        <property name="Database" ref="db"/>
    </bean>
</beans>
```

➔ Exécution :

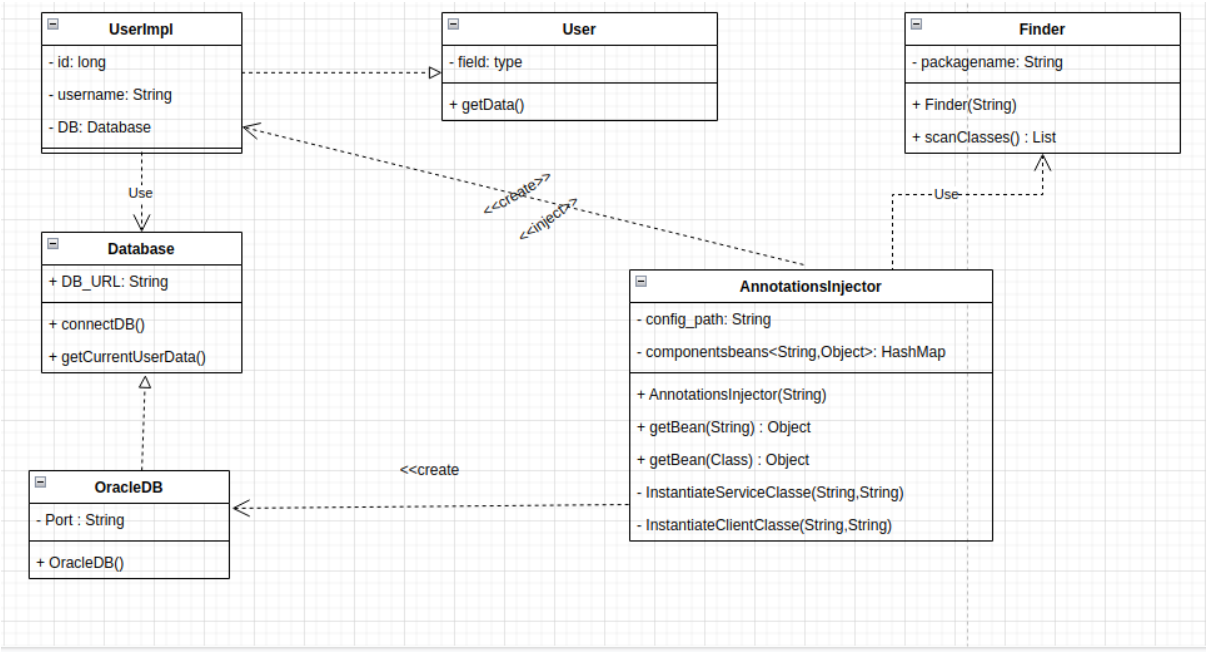
```
/usr/lib/jvm/java-8-openjdk/bin/java ...
getting your info from Postgre database
wait...
success
Username is : zakaria

Process finished with exit code 0
```

Partie 2 :

Création du framework qui fait l'injection des dépendances en utilisant les annotations :

Dans cette partie on va se focaliser sur la création et la manipulation des annotations qui nous permettra par la suite d'injecter les dépendances .

on va travailler sur le diagramme suivant :



la première étape consiste à créer les annotations qu'on va utiliser , qui sont Component (marquer une classe comme un composant qu'on doit instancier) et Autowired (pour chercher un objet déjà instancié et l'injecter dans l'attribut qui est marqué par cette annotation), j'étais inspiré par le framework Spring pour les noms 😆.

➔ Annotation Component :

```java
@Retention(value = RetentionPolicy.RUNTIME)
@Target(value = ElementType.TYPE)
public @interface Component {
    String name() default "";
}
```

➔ Annotation Autowired

```java
@Retention(RetentionPolicy.RUNTIME)
@Target(value = {ElementType.CONSTRUCTOR,ElementType.FIELD})
public @interface Autowired {
    String name() default "";
}

```

➔ Maintenant on doit créer la classe AnnotationsInjector qui nous permet d'injecter les dépendances en 3 étapes :

1- Scanner tous les classes appartient à un package donné en paramètres

2- parcourir la liste des classes trouvés et si la classe est annoté par l'annotation @Component l'instancier puis mettre l'objet instancié dans un Hashmap.

3- parcourir la liste des classes trouvés et pour chaque classe parcourir les listes des attributs et si un attribut est annoté par l'annotation @Autowired chercher et injecter un objet existant dans cet attribut.

➔ la classe AnnotationsInjector :

```java
public class AnnotationsInjector {
    private HashMap<String, Object> componentsbeans = new HashMap<>();

    public AnnotationsInjector(String packagename) throws IllegalAccessException {
        Finder finder = new Finder(packagename);
        Collection<Class<?>> classCollection = finder.scanClasses();
        //Instantiate all components
        for (Class<?> class_ : classCollection) {
            if (class_.isAnnotationPresent(Component.class)) {
                String name = class_.getAnnotation(Component.class).name().equals("") ? class_.getCanonicalName() : class_.getAnnotation(Component.class).name();
                InstantiateComponentClasse(class_.getName(), name);
            }
        }
        for (Class<?> class_ : classCollection) {
            Field fields[] = class_.getDeclaredFields();
            for (Field field : fields) {
                if (field.isAnnotationPresent(Autowired.class)) {
                    Object servicebean,clientbean = null;
                    if (!field.getAnnotation(Autowired.class).name().equals("")) {
                        servicebean = getBean(field.getAnnotation(Autowired.class).name());
                    } else
                        servicebean = getBean(field.getType());
                    if (field.getDeclaringClass().isAnnotationPresent(Component.class)) {
                        if (!field.getDeclaringClass().getAnnotation(Component.class).name().equals("")) {
                            clientbean = getBean(field.getDeclaringClass().getAnnotation(Component.class).name());
                        } else clientbean = getBean(field.getDeclaringClass().getName());
                    }
                    field.setAccessible(true);
                    field.set(clientbean, servicebean);
                }
            }
        }
    }
}
```

➔ La classe Finder :

pour scanner tous les classes dans un package spécifique

j’ai utilisé un bibliothèque qui s'appelle **burningwave**

```java
public class Finder {
    private String packagename;
    public Finder(String packagename) {
        this.packagename = packagename;
    }
    public Collection<Class<?>> scanClasses() {
        ComponentSupplier componentSupplier = ComponentContainer.getInstance();
        ClassHunter classHunter = componentSupplier.getClassHunter();
        try (SearchResult result = classHunter.findBy(
                //Highly optimized scanning by filtering resources before loading from ClassLoader
                SearchConfig.forResources(
                        this.packagename
                )
        )) {
            return result.getClasses();
        }
    }
}
```

➔ la fonction InstantiateComponentClasse

qui permet d'instancier un objet en lui donnant le nom complet de la classe et le nom de l’objet qui sera dans le hashmap

```java
private void InstantiateComponentClasse(String classname, String BeanName) {
        try {
            Class beanclasse = Class.forName(classname);
            Object beanobj = beanclasse.newInstance();
            componentsbeans.put(BeanName, beanobj);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
```

➔ la fonction getBean(String beanname) :

permet de retourner un objet depuis le hashmap des objets instanciés par son nom.

```java
public Object getBean(String beaname) {
        Object object;
        for (HashMap.Entry<String, Object> entry : componentsbeans.entrySet()) {
            if (entry.getKey().contains(beaname)) {
                object= entry.getValue();
                return object;
            }
        }
        return null;
    }
```

➔ la fonction getBean(Class classz)

permet de retourner un objet depuis le hashmap des objets instanciés par une interface qui implémente.

```java
   public Object getBean(Class classz) {
        for (HashMap.Entry<String, Object> entry : componentsbeans.entrySet()) {
            System.out.println();
            if (Arrays.toString(entry.getValue().getClass().getInterfaces()).contains(classz.getName()) ||
entry.getValue().getClass().getName().equals(classz.getName())) {
                return entry.getValue();
            }
        }
        return null;
    }
```

Test du framework :

on ajoute l'annotation @Component sur la classe UserImpl et l'annotation @Autowired sur l'objet qu'on veut injecter avec le nom du component:

```java
public class UserImpl implements User {
    private long id;
    private String name = "zakaria";

    @Autowired
    private Database database;

    public UserImpl() {
    }
    public void getCurrentUserData() {
        System.out.println("-------------------------------------------------");
        System.out.println(this.getClass().getName() + " constructor executed");
        database.getUserData("zakaria");
        System.out.println("-------------------------------------------------");
    }
}
```

➔ la classe OracleDB :

```java
@Component(name = "oracledb")
public class OracleDB implements Database {

    public OracleDB() {
    }

    public String getUserData(String username) {
        System.out.println("getting your info from oracle database");
        System.out.println("wait...");
        System.out.println("success");
        System.out.println("Username is : "+username);
        return username;
    }
}
```

Maintenant on va créer une classe pour tester le fonctionnement du framework :

```java
public class ApplicationVersionAnnotations {
    public static void main(String args[]) throws IllegalAccessException {
        AnnotationsInjector injector = new AnnotationsInjector(".");
        User user= (User) injector.getBean(User.class);
        user.getCurrentUserData();
    }
}
```

➔ Exécution :

```
-------------------------------------------------
metier.UserImpl constructor executed
getting your info from oracle database
wait...
success
Username is : zakaria
-------------------------------------------------
```

création d'une nouvelle classe qui implémente l'interface User

```java
@Component
public class UserImplv2 implements User {

    @Autowired
    private Database database;

    public UserImplv2() {
    }

    @Override
    public void getCurrentUserData() {
        System.out.println(this.getClass().getName() + " constructor executed");
        database.getUserData("zakaria");
        System.out.println("-------------------------------------------------");
    }
}

```

➔ Exécution :

```
1  ------------------------------------------------
2  metier.UserImpl constructor executed
3  getting your info from oracle database
4  wait...
5  success
6  Username is : zakaria
7  ------------------------------------------------
8  metier.UsetImplv2 constructor executed
9  getting your info from oracle database
10 wait...
11 success
12 Username is : zakaria
13 ------------------------------------------------
```

Code Complet sur Github : Dependency-Injection-Framework