Une introduction à Spring

1 Mise en place d'un projet Eclipse

⏩ Préparer Eclipse version JEE

- **Sur les machines gérées par la DOSI** : Choisissez l'application `Eclipse-Multi` dans le menu catégorie *Programmation*.
- **Sur votre machine personnelle** :
  - ▷ Téléchargez Eclipse pour JEE pour Linux ou windows,
  - ▷ **puis** installez le plugin pour Lombok.

⏩ Préparer un projet dans Eclipse

- Téléchargez le projet Maven déjà préparé.
- Décompressez cette archive.
- Importez, dans Eclipse, un projet Maven et choisissez le répertoire précédent.
- Lancez les tests de ce projet.
- Exécutez ce projet (**Run as .. Java application** classe `myapp.MyApp`).
- Faites un **Run as ... Maven build... goal : package** dans Eclipse et vérifiez la création d'un fichier `.jar` dans le répertoire `target` (après avoir fait F5 pour rafraîchir votre projet).

⏩ Exécution hors-Eclipse

- Construire votre projet :

Build du projet

```
cd repertoire_de_votre_projet
mvn package
```

- Cette opération a généré une fichier `.jar` dans le répertoire `target` .
- Lancez votre application en ligne de commande :

Exécution du projet

```
cd repertoire_de_votre_projet
java -jar target/jeeApp-0.0.1-SNAPSHOT.jar
```

ℹ Explications

- Nous utilisons Maven comme outil de gestion des dépendances et de construction de notre projet.
- Les messages visibles lors de l'exécution sont générés par Spring boot. Spring boot est un outil d'intégration qui permet très simplement de gérer les librairies dont nous avons besoin et d'assembler une application complexe. Spring boot est donc un facilitateur de mise en oeuvre de Spring.

Explorer ce projet. Les projets **Maven** ont une structure particulière :

- `pom.xml` : configuration de Maven
- `src/main/java` : le code source Java
- `src/main/resources` : les ressources (fichiers XML, images, propriétés, etc.)
- `src/main/resources/application.properties` : le fichier de paramétrage de Spring boot
- `src/test/java` : le code source Java de test
- `src/test/resources` : les ressources pour le test

2 Introduction

Le framework Spring est une boite à outils très riche permettant de structurer, d'améliorer et de simplifier l'écriture d'application JEE. Spring est organisé en module

- Gestion des instances de classes (JavaBean et/ou métier),
- Programmation orientée Aspect,
- Modèle MVC et outils pour les applications WEB,
- Outils pour la DAO (JDBC),
- Outils pour les ORM (Hibernate, iBatis, ...),
- Outils pour les applications JEE (JMX, JMA, JCA, EJB, ...),

Préparez, dans un navigateur, un onglet vers la documentation Spring.

3 Programmation par contrat

3.1 Explorer le code fourni

- `myapp.IHello` : Interface de définition d'un service
- `myapp.HelloService` : implantation de ce service. Vous remarquerez
  - ▷ `@Service("helloService")` la déclaration Spring et le nommage du service. Ce service sera automatiquement découvert et exploiter.
  - ▷ `@Value("$helloMessage")` l'accès à un paramètre de configuration Spring Boot que vous trouverez dans le fichier `src/main/resources/application.properties`. Nous pouvons donc changer le message sans étape de modification du code.
- `myapp.MyApp` : code `main` de démarrage. Vous remarquerez
  - ▷ `@SpringBootApplication()`
    C'est une application Spring Boot et nous souhaitons que le package `myapp` qui contient la classe `MyApp` soit exploré afin de découvrir les configurations et les services.
  - ▷ `@Autowired IHello helloService;`
    Nous demandons à Spring d'injecter un servive `IHello` pour pouvoir l'utiliser.
  - ▷ `@Override public void run(String... args)`
    Cette méthode est le point de démarrage de l'application. Elle provient de l'interface `CommandLineRunner` car nous avons une application ligne de commande.

Comment l'application fonctionne-t-elle ?

Nous exécutons `myapp.MyApp.main` qui va provoquer l'appel de `SpringApplication.run`. Six étapes vont s'enchaîner :

1. **Exploration et découverte**
   - Explorer les annotations de `MyApp`.
   - Explorer le package de `MyApp` c'est-à-dire `myapp`. Ce qui provoque
     - ▷ la découverte de la classe de configuration `myapp.SpringConfiguration`. Cette configuration provoque
       - `@Bean public String bye()` La création d'un service associé à `String.class` (le résultat de la méthode).
     - ▷ la découverte du service `myapp.HelloService`
2. **Instanciation**
   - Spring va créer une instance A de `MyApp`.
   - Spring va créer une instance B de `HelloService`.
3. **Injection des dépendances**
   - Spring va injecter le paramètre `helloMessage` dans B.
   - Spring va injecter B dans A.
4. **Initialisation**
   - Spring va initialiser B (appel de `start`).
   - Spring va initialiser A (rien à faire).
5. **Exécution**
   - Spring va appeler la méthode `run` sur A.
6. **Fermeture et fin de l'application**
   - Spring va fermer le service A (rien à faire).
   - Spring va fermer le service B (appel de `close`).
   - Fin de l'application.

⏩ **Travail à faire :** Explorer la classe de test et comprendre son fonctionnement.

3.2 Principe

La **programmation par contrat** consiste à séparer la spécification d'une couche logicielle (aussi appelée service) de sa réalisation. La spécification donne lieu à la création d'une **interface** et la réalisation fournit une **classe qui implante cette interface**. Ce ne sont pas nécessairement les mêmes personnes qui développent l'interface et son implantation. On peut également remarquer que la phase de réalisation peut produire plusieurs implantations différentes d'une même interface. Le choix entre ces implantations est réalisé à l'intégration entre les couches. Les objectifs de cette approche :

- **Réduire les dépendances.** Les classes d'implantation ne se connaissent pas. Elles dialoguent au moyen des interfaces. De ce fait, on peut facilement changer un implantation contre une autre sans avoir à mettre à jour la totalité du logiciel.
- **Faciliter les tests.** Chaque couche logicielle ayant une spécification claire, il est facile de lui associer un

jeu de tests utilisable quelque soit la nature de l'implantation.

- **Simplifier le code.** Dans certains cas de figure, le code d'une méthode est une suite de considérations sans liaison directe entre-elles. La programmation par contrat va faciliter la construction d'implantations façade qui se chargent chacune d'une partie du travail.
- **Organiser le développement.**

3.3 Spécifier un service logiciel

Prenons un exemple pour éclairer ces principes. Nous avons besoin dans nos applications de pouvoir tracer un certain nombre d'évènements. Nous allons donc créer un service de trace (un *logger* en anglais). Ce service est spécifié par l'interface ci-dessous (**déjà présente**) :

Interface `ILogger.java`

```
package myapp;

public interface ILogger {

    default void log(String message) { };

}
```

3.4 Une première implantation

Pour utiliser ce service, nous avons besoin d'une classe qui implante ce service. Il existe plusieurs manières de faire. Nous allons, dans une premier temps, envoyer les messages de trace sur la console de sortie d'erreur standard :

Un service pour faire une trace sur la sortie standard

```java
package myapp;

import java.util.Date;

import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.context.annotation.Primary;
import org.springframework.stereotype.Service;

@Service("stderrLogger")
@Primary
@Qualifier("stderr")
public class StderrLogger implements ILogger {

    @PostConstruct
    public void start() {
        System.err.printf("Start␣%s\n", this);
    }

    @PreDestroy
    public void stop() {
        System.err.printf("Stop␣%s\n", this);
    }

    @Override
    public void log(String message) {
        System.err.printf("%tF␣%1$tR␣|␣%s\n", new Date(), message);
    }

}
```

Explications

- L'annotation `@Service` permet de déclarer cette implantation et de lui associer un nom.
- L'annotation `@Primary` permet de déclarer cette classe comme implantation par défaut (utile si nous avons plusieurs versions du logger).
- L'annotation `@Qualifier` permet de qualifier cette implantation. Nous pourrons ainsi la distinguer des autres implantations (nécessaire si nous voulons choisir cette implantation). Découvrez d'autres possibilités de `@Qualifier`.
- Les méthodes `start` et `stop` correspondent à la phase de démarrage et de terminaison du service. Nous retrouverons ces méthodes dans toutes les implantations (sauf si elles sont vides).

A propos des packages. Par soucis de simplification, l'interface est dans le meme package que la classe d'implantation. Dans la réalité nous pourrions avoir deux packages. La spécification d'un service peut être composé de plusieurs interfaces accompagnées de javaBeans ou de classes d'exception. L'implantation de ce service peut également contenir plusieurs classes ce qui justifie clairement l'utilisation de plusieurs packages.

Travail à faire :

- Prévoir un test unitaire dans lequel vous aller demander l'injection d'un `ILogger` et d'un `StderrLogger` afin de vérifier le bon fonctionnement.
- Prévoir un test unitaire afin d'utiliser `StderrLogger`.

3.5 Une deuxième implantation

Nous pouvons aussi donner une deuxième implantation qui stocke les traces dans un fichier :

Une version pour tracer dans un fichier

```
package myapp;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.PrintWriter;
import java.util.Date;

import jakarta.annotation.PreDestroy;

public class FileLogger implements ILogger {

    // parameter: the writer
    private final PrintWriter writer;

    public FileLogger(String fileName) {
        try {
            this.writer = new PrintWriter(new FileOutputStream(fileName, true));
        } catch (FileNotFoundException e) {
            throw new IllegalArgumentException("bad␣fileName");
        }
    }

    @PreDestroy
    public void stop() {
        writer.close();
    }

    @Override
    public void log(String message) {
        writer.printf("%tF␣%1$tR␣|␣%s\n", new Date(), message);
    }

}
```

Cette nouvelle implantation a absolument besoin d'un paramètre (le nom du fichier) pour être fonctionnelle. La solution retenue est la plus simple : ajouter un argument au constructeur. Nous remarquons que de ce fait, la méthode `start` n'a plus vraiment d'intérêt. Cette version n'est pas JavaBean (pas de constructeur sans argument). Il ne peut donc pas être instancié simplement par Spring.

Pour l'utiliser néanmoins, nous allons enrichir la classe de configuration (`SpringConfiguration`) afin de lui demander de créer un service qui va instancier cette classe :

```
...

@Bean
@Qualifier("fileLoggerWithConstructor")
public ILogger fileLoggerWithConstructor(@Value("${logfile}") String logFile) {
    return new FileLogger(logFile);
}

...
```

⏩ **Travail à faire :** Prévoir le paramètre `logfile` dans le fichier de paramétrage (`application.properties`) et préparez un test unitaire afin de vérifier le bon fonctionnement. **Attention** : respecter la forme c-dessous :

```
...

    @Autowired
    @Qualifier("fileLoggerWithConstructor") // pour choisir l'implantation
    ILogger fileLoggerWithConstructor;

    @Test
    public void testFileLoggerWithConstructor() {
        ...
    }

...
```

3.6 Une troisième implantation

La plupart des classes d'implantation ont besoin de paramètres pour assurer leur service. Le choix de placer ces paramètres en argument du constructeur pose plusieurs problèmes :

- La classe obtenue **n'est pas un javaBean** (pas de constructeur vide). C'est particulièrement gênant car l'intérêt de ces composants élémentaires est très important.
- Les paramètres du service sont fixés à sa création (par le constructeur). Il n'est donc **pas possible de les changer en cours de route**, voir même d'envisager un recyclage du service (changement des paramètres et nouvelle initialisation).
- Si nous avons **beaucoup de paramètres**, le constructeur est difficile à utiliser.
- Il est difficile de prévoir des **valeurs par défaut** pour certains paramètres.

Nous allons donc introduire une nouvelle solution au problème des paramètres : les paramètres vont être codés comme des propriétés de la classe d'implantation et la méthode `start` devra les utiliser pour initialiser le service. Nous obtenons donc cette nouvelle version :

```java
package myapp;

import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.util.Date;

import jakarta.annotation.PostConstruct;
import jakarta.annotation.PreDestroy;

import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.stereotype.Service;
import org.springframework.util.Assert;

import lombok.Data;

@Service
@Qualifier("beanFileLogger")
@Data
public class BeanFileLogger implements ILogger {

    // parameter: writer name
    private String fileName = "myapp.log";

    // property: writer
    private PrintWriter writer;

    // start service
    @PostConstruct
    public void start() {
        Assert.notNull(fileName, "no fileName");
        try {
            OutputStream os = new FileOutputStream(fileName, true);
            this.writer = new PrintWriter(os);
        } catch (FileNotFoundException e) {
            throw new IllegalStateException("bad fileName");
        }
    }

    // stop service
    @PreDestroy
    public void stop() {
        writer.close();
    }

    @Override
    public void log(String message) {
        writer.printf("%tF %1$tR | %s\n", new Date(), message);
    }

}
```

Le code d'intégration a maintenant la responsabilité de fixer les paramètres du service avant d'appeler la méthode d'initialisation. Cette solution est plus simple et plus systématique quand le nombre de paramètres est important.

⏩ **Travail à faire :** Prévoir un test unitaire pour tester l'utilisation de la valeur par défaut.

4 Injection des dépendances

L'**injection des dépendances** traite le délicat problème de la communication et de la dépendance entre services logiciels. Prenons l'exemple d'une classe métier :

```
package myapp;

public interface ICalculator {

    int add(int a, int b);

}
```

Construisons maintenant une implantation de ce service qui génère une trace après chaque appel d'une méthode métier. Cette implantation a donc besoin d'une couche *logger* pour s'exécuter correctement. Nous pourrions envisager de placer dans cette implantation la propriétés suivante :

Liaison directe entre deux implantations

```
package myapp;

public class SimpleCalculator implements ICalculator {

    private ILogger logger = new myapp.StderrLogger();

    ...
}
```

Cette solution pose deux problèmes :

1. Une dépendance directe vient d'être introduite entre cette implantation de la calculatrice et une implantation particulière de la couche *logger*. Cette dépendance est regrettable car inutile. La calculatrice doit utiliser l'interface `ILogger` et pas une implantation.

2. Si nous avions choisi une couche de trace ayant besoin d'un paramètre (comme celle vue précédemment), nous aurions sans doute dû inclure ce paramètre (le fichier de sortie) comme un paramètre de la calculatrice. En d'autres termes, les paramètres d'une couche A doivent inclure tous les paramètres des couches utilisées par A.

Pour éviter ces problèmes, nous allons simplement introduire dans l'implantation de la calculatrice un paramètre faisant référence à une implantation de la couche *logger*. De ce fait, les deux implantations resteront indépendantes l'une de l'autre. Le seul point de contact sera l'interface `ILogger` :

Une calculatrice qui trace

```java
package myapp;

import jakarta.annotation.PostConstruct;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.util.Assert;

import lombok.Data;

@Service("calculator")
@Data
public class CalculatorWithLog implements ICalculator {

    @Autowired
    private ILogger logger;

    @PostConstruct
    public void start() {
        Assert.notNull(logger, "no logger");
    }

    @Override
    public int add(int a, int b) {
        logger.log(String.format("add(%d,%d)", a, b));
        return (a + b);
    }

}
```

Travail à faire : Prévoir un test unitaire pour utiliser la calculatrice :

```java
...

    @Autowired
    ICalculator calculator;

    @Test
    public void testCalculator() {
        var res = calculator.add(10, 20);
        assertEquals(30, res);
        assertTrue(calculator instanceof CalculatorWithLog);
    }

...
```

Remarque : Nous pouvons très facilement et **sans modifier la couche métier de la calculatrice** changer la politique de trace en utilisant un fichier. Il suffit de changer l'implantation par défaut des loggers (modifier la classe équipée de `@Primary`). Faites un test de cette possibilité.

Nous venons de mettre en oeuvre le principe de **l'injection de dépendances**. C'est la partie intégration qui se charge d'injecter dans la couche métier la référence vers la couche *logger*. Initialiser une application revient à créer les couches logicielles, injecter les dépendances et appeler les méthodes d'initialisation.

⏩ **Travail à faire :** Malheureusement, nous ne testons pas que la calculatrice et nous testons également le logger associé. Afin d'éviter ce désagrément, prévoyez un logger qui ne fait rien (`NullLogger`) et enrichissez la classe de configuration afin de fournir une calculatrice qui utilise ce logger. **Moralité** : Nous pouvons créer plusieurs instances d'un même service dont le fonctionnement est paramétré en fonction des besoins.

⏩ **Travail à faire :** Vous pouvez parcourir avec profit les trois premières sections de ce chapitre.

5 Nouvelles implantations

Notre première version de la calculatrice mélange du code métier (addition) et du code de trace. Ce n'est pas une très bonne idée.

⏩ **Travail à faire :** Proposez une nouvelle implantation de décoration de la calculatrice qui est construite sur deux paramètres :

- une référence vers une autre implantation de la calculatrice (qui ne fait aucune trace),
- une référence vers une implantation de la couche *logger*,
- Ce décorateur va retransmettre les requêtes et y associer une trace.

6 Configuration XML

Spring offre aussi la possibilité de configurer la création des instances via un fichier XML. Cela permet d'éviter d'utiliser les annotations.

⏩ **Travail à faire :**

- Placez vous dans le répertoire de votre projet et tapez les commandes suivantes :

```
cd repertoire_du_projet
wget https://jean-luc-massat.pedaweb.univ-amu.fr/ens/jee/xml-config.zip
unzip xml-config.zip
rm -f xml-config.zip
```

- Après avoir rafraichi votre projet (touche F5), vous trouverez de nouveaux fichiers :

```
src/main/java/myapp/xml/Counter.java           Un compteur
src/main/java/myapp/xml/MessageFactory.java    Un service pour les msg
src/main/resources/myapp/xml/message-config.xml Fichier de config XML
src/test/java/myapp/xml/TestXmlConfigMessage.java Classe de test
```

- Vous pouvez maintenant explorer le fichier de configuration XML, la classe de test et comprendre cette nouvelle solution (qui, historiquement, était la première).

7 Les profils

Terminons cette séance avec la notion de profil. Nous pouvons, depuis Spring 5, associer nos beans à un profil (par exemple `devel` pour le mode développement ou `prod` pour le mode production). Nous pouvons ainsi faire varier la configuration de nos logiciels en fonction du profil choisi.

⏩ Travail à faire :

- Créez une implantation `StdoutLogger` :

```
@Service
@Primary
@Profile("!devel") // classe pour tous les profils sauf devel
public class StdoutLogger implements ILogger {
    ...
}
```

- Ajoutez l'annotation `@Profile("devel")` à la classe `StderrLogger`. Vous venez de définir deux versions : une pour le développement, et l'autre pour les autres profils.
- Vos tests unitaires ne doivent plus fonctionner. Ajoutez l'annotation `@ActiveProfiles("devel")` à vos tests unitaires pour qu'ils soient associés au profil développement.
- Préparez une nouvelle classe de test unitaire avec le profil `prod` et vérifiez que vous obtenez bien `StdoutLogger`.
- Explorez les autres possibilités.