

# Langages et Environnements Évolués

## TP2: Spring

5 décembre 2017

*Exercice 1* (Injection de dépendance (DI)).

1. Lancez IntelliJ et créez un nouveau projet.
2. Sélectionnez **Spring** dans la colonne de gauche.
3. Nommez votre projet et continuez. Dans la suite, nous allons l'appeler<sup>1</sup> `$Module`, car ceci est en fait le nom du module principal (on peut par la suite ajouter des modules à un même projet).
4. Créez un nouveau package : `$Module` → `src` → double clic → `New` → `Package`, que vous nommez comme vous voulez (ici `$package`).
5. Dans `src/$package`,<sup>2</sup> créez les classes `Main`, `Client`, `Voiture` et `Camion`, ainsi que l'interface `Vehicule`.
6. L'interface `Vehicule` ne demande pour l'instant que la méthode `String louer()`.
7. Implémentez un compteur dans les classes `Client`, `Voiture` et `Camion` pour que chaque entité créée ai un numéro différent.<sup>3</sup>
8. Implémenter `Vehicule` dans `Camion` et dans `Voiture` comme rendant une `String` respectant le schéma "loue la voiture n.5".
9. Rajoutez `@Configuration` avant la déclaration de classe de `Client` et `Voiture` :

```
@Configuration
public class Voiture implements Vehicule {
    ...
}
```

Cela indique que l'on peut utiliser leur constructeur pour injecter des clients et et voitures.

10. Dans la fenêtre Spring (`View` → `Tool Windows` → `Spring`), faites

`$Module` → double clic → `Spring` → `Spring Application Context` → `editer`

puis sélectionnez `Client` et `Voiture`. Cela permet à `Spring` de savoir qui peut être lié à qui et d'indiquer les erreurs.

11. Dans la classe `client`, ajoutez un attribut `Vehicule vehicule` et générez un setter (clique droit → `Generate...` → `Setter`), puis annotez le par `@Autowired` :

```
@Autowired
public void setVehicule(Vehicule vehicule) {
    this.vehicule = vehicule;
}
```

12. Ajoutez la méthode suivante :

```
public void louer_vehicule() {
    System.out.println(String.format("Le client n.%1$d %2$s", number ,vehicule.louer()));
}
```

13. Ajoutez l'annotation `@ComponentScan` dans le `main` :

---

1. `$Module` désigne une variable dans ce pdf, ne l'appellez pas vraiment comme ça...  
2. `$package` désigne une variable dans ce pdf, ne l'appellez pas vraiment comme ça...  
3. Faire deux champs, un compteur (statique) et un numéro (normal).

```

@ComponentScan
public Class Main{
    ...
}

```

Cela oblige Spring à scanner les annotations visibles.

14. Dans la classe Main, ajoutez la méthode suivante :

```

public static void main(String[] args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(Main.class);
    Client p = context.getBean(Client.class);
    Client q = context.getBean(Client.class);
    p.louer_vehicule();
    q.louer_vehicule();
}

```

Exécutez et interprétez.

15. Ajoutez l'annotation @Scope("prototype") à la classe Client. Exécutez. Ajoutez-le à la classe Voiture. Exécutez. Interprétez

Le scope représente l'endroit où Spring cherche l'existence d'une instance déjà créé. Le scope de base cherche sur toute l'exécution du programme, lorsque le scope "prototype" (le nom est mal choisi) ne cherche nul part, il crée donc toujours une nouvelle instance. Lorsque l'on travaillera sur des applis web, on aura accès à d'autres scopes comme "session" ou "request".

16. Maintenant, ajoutez le @Configuration à la classe Camion. Que se passe-t-il ?

17. Pour éviter ce soucis, on peut ajouter @Primary à la classe Voiture :

```

@Configuration
@Primary
public class Voiture implements Vehicule {

```

18. De cette façon, on a toujours accès à camion si on le souhaite; en particulier on peut écrire les lignes suivantes dans le main :

```

Client r = new Client();
r.setVehicule(context.getBean(Camion.class));
r.louer_vehicule();
}

```

19. On peut aussi créer une factory; par exemple, on peut ajouter dans la classe Main :<sup>4</sup>

```

@Bean
@Primary
@Scope("prototype")
@Autowired
public static Vehicule vehiculeFactory (Voiture v,Camion c) {
    if (v.number<3) {
        return v;
    } else {
        return c;
    }
}

```

20. Enlevez le @Primary de la classe Voiture. (Il ne peut y avoir qu'un @Primary pour des véhicules)

21. Exécutez et interprétez.

22. Enfin, il est possible d'utiliser une toute autre configuration pour le contexte d'application. Pour ça créez fichier de configuration Spring :

\$Module → src → \$package → clic droit → New → XML Configuration File → Spring Config  
 que vous nommerez MyApContext.xml

---

4. Pour des raisons historiques, Spring utilise le mot-clef "bean" dans ce cas, mais cela n'a pas grand chose à voir avec les EJBs...

23. que vous remplirez avec la simple configuration :

```
<bean id="camion" class="$package.Camion"></bean>

<bean id="client" class="$package.Client">
  <property name="vehicule" ref="camion"/>
</bean>
```

24. Dans le main, écrire les lignes suivantes :

```
ApplicationContext context2 = new FileSystemXmlApplicationContext("src/MyApContext.xml");
Client r = context2.getBean(Client.class);
r.louer_vehicule();
```

Exécutez et interprétez.

25. Créez un nouveau répertoire dans /src/package (\$Module → src → package → cloc droit → New → Package) et déplacez y Client et Camion.

26. Vérifiez que le refactoring c'est bien fait et exécutez le main. Interprétez.

27. Ajoutez un nouveau main dans le dossier nouvellement créé avec simplement :

```
@ComponentScan
public Class Main2{
  public static void main(String[] args) {
    ApplicationContext2 context = new AnnotationConfigApplicationContext(Main2.class);
    Client q = context.getBean(Client.class);
    q.louer_vehicule();
    ApplicationContext context = new AnnotationConfigApplicationContext(Main.class);
    Client p = context.getBean(Client.class);
    p.louer_vehicule();
  }
}
```

executez-le et interprétez.

En fait, on peut créer un context applicatif à partir de n'importe quel classe C contenant un @ComponentScan, y sera alors visible toutes les classes annotées @Configuration et toutes les méthodes annotées @Bean présentes dans la classe C, dans une classe du dossier de C ou dans un sous dossier.

*Exercice 2* (Spring (M)VC). Malheureusement, l'ajout à la volée de frameworks Spring n'est pas bien implémenté dans IntelliJ, on va donc devoir commencer un nouveau projet à chaque fois...

Dans ce premier exo sur MVC, on va plus ou moins bypasser le model. L'application étant très simple, c'est naturelle, on réintroduira un vrais modèle plus tard.

1. Lancez IntelliJ et créez un nouveau projet.
2. Sélectionnez **Spring** dans la colonne de gauche ainsi que **Spring** et **Spring MVC** dans la colonne de droite.
3. Nommez votre projet et continuez. Dans la suite, nous allons l'appeler<sup>5</sup> **\$Module**.
4. Allez dans **File** → **Project Structure...** → **Project Settings** → **Artifacts** → **\$Module:War Exploded** → **Output Layout** → **WEB-INF** → clique droit → **Add Copy of** → **File** et ajoutez un fichier **lib** puis, dessus, faites clique droit → **Add Copy of** → **Library files** deux fois en ajoutant les deux bibliothèques proposées. Acceptez.
5. Utilisez Glassfish : en haut à droite, dans l'onglet d'exécution, allez dans **Edit Configurations...** puis **+** → **Glassfish Server** → **local** ; précisez le domaine (probablement **Domain1**) et acceptez. Exécutez le code. **Troubleshooting** : En cas d'erreur allez dans la fenêtre de projet, allez dans **\$Module** → **web** → **WEB-INF** → **dispatcher-servlet.xml** et ajoutez la ligne suivante (avant **</beans>**) :

```
<bean class = "org.springframework.web.servlet.view.InternalResourceViewResolver"></bean>
```

6. Créez deux sous-package dans **src** : un package **controllers** et un package **entities**.
7. Dans **entities**, vous pouvez ajouter **Client**, **Vehicule** et **Voiture** (ainsi que **Camion** si vous voulez) de la section précédente.
8. Pour que IntelliJ voit les lien ID de Spring, on ajoute la ligne suivante à **applicationContext.xml** :

```
<context:component-scan base-package="entities"/>
```

Cela fait la même chose que le **@Component-scan** de l'exercice précédent, mais sous forme XML...

9. De même, on va ajouter le scan des contrôleurs dans **dispatcher-servlet.xml** :

```
<context:component-scan base-package="controllers"/>
```

10. Dans **controllers**, vous pouvez ajouter une classe **HomeController** qui servira à contrôler notre page d'accueil.
11. **HomeController** doit implémenter **Serializable**, il n'y a pas de méthode à définir, juste à préciser l'interface<sup>6</sup>
12. Y ajoutez l'annotation **@Controller** à la classe **HomeController**.
13. Y ajoutez un paramètre **public Client p** qui est autowired.
14. On veut mettre nos vues dans le même dossier : créez un nouveau sous dossier dans **web** et y créer un **.jsp** appelé **home.jsp**.
15. On veut que l'application se lance directement sur le **home** ; pour ça on ajoute la ligne suivante à **web.xml** :

```
<welcome-file-list>
  <welcome-file>home.form</welcome-file>
</welcome-file-list>
```

Remarquez que l'on appelle **home.form** et non pas **home.jsp**. En fait, **.form** est une extension fictive qui va être rattrapé par le dispatcher grâce au **<servlet-mapping>** défini dans **web.xml**. Il faut maintenant dire au dispatcher que faire à ce moment là.

16. Ajouter à **HomeController** la méthode suivante :

```
@RequestMapping("home")
public String home (Model model) {
    model.addAttribute("loc", p.louer_vehicule());
    return "/views/home.jsp";
}
```

Décortiquons cette méthode :

- Le **@RequestMapping("home")** permet au dispatcher de rediriger toutes les requêtes à **home.form** vers cette méthode.

---

5. **\$Module** désigne une variable dans ce pdf, ne l'appellez pas vraiment comme ça...

6. Ceci n'est nécessaire que si on utilise Glassfish, pas avec Tomcat par exemple. La raison est que Glassfish compile avec les norme JEE, et en particulier va en faire une version remote...


- Le `Model model` est une version abstraite du modèle que l'on bypass en l'appellant comme argument (et en utilisant implicitement l'autowire de Spring).
- La ligne `model.addAttribute("loc", p.louer_vehicule());` permet d'associer la string `p.louer_vehicule()` à la variable `loc` dans le model.
- On retourne la string `"/views/home.jsp"` qui va créer une vue appelant le `.jsp` voulu avec le modèle instancié.

17. Il ne nous reste plus qu'à écrire le contenu de `home.jsp`, par exemple :

```
<h2>${loc}</h2>
```

18. Exécutez...

### Exercice 3 (JPA).

19. Allez dans `File` → `Project Structure...` puis dans `Modules` → `$Module` → cliquer droit → `Add` → `JPA`.
20. En bas, tant que l'icône  s'affiche, résolvez les problèmes comme proposés.
21. En haut à droite, cliquez sur `+` → `persistance`.
22. Dans `Default JPA Provider`, choisissez `hibernate`, puis validez.
23. Ouvrez un BDD (Derby embeded ou autre si vous avez).

On voudrait faire une structure avec `Voiture` et `Camion` en sous classe de `Vehicule`, mais tout de même autoriser des vehicules qui ne sont ni des voitures ni des camions. De plus, on voudrait parler de client avec un lien 1 :1 entre les clients et les vehicules. Enfin, il faudrait ajouter, pour les camions, un champ `Volume` qui n'ai pas de sens pour les voitures. Le paradigme relationnel ne permet pas de faire ca efficacement ; on va donc faire une unique classe avec tout et utiliser l'ORM pour séparer ca en classes.

24. Ajoutez une table `VEHICULE` avec comme champs : `ID` (int, clef primaire), `TYPE` (varchar), `CLIENT_ID` (int, unique), `CLIENT_NOM` (varchar) et `VOLUME` (int).
25. Extraire les entités de la base de donnée (dans la suite on n'utilise pas le suffix `Entity` pour plus de concision).
26. Dans la classe `Vehicule` nouvellement créé, ajoutez les annotations :

```
@Entity
@Inheritance(strategy=InheritanceType.SINGLE_TABLE)
@DiscriminatorColumn(name="TYPE", discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("Autre")
public class Vehicule {
```

En plus de `@Entity` que l'on connaît déjà, il y a

- `@Inheritance(strategy=InheritanceType.SINGLE_TABLE)` qui dit qu'il s'agit d'une classe dont d'autres entités peuvent hériter,
  - `@DiscriminatorColumn(name="TYPE", discriminatorType=DiscriminatorType.STRING)` qui indique que pour savoir à quelle classe appartient une ligne, on regarde la colonne `TYPE`,
  - `@DiscriminatorValue("Autre")` qui indique que si `TYPE` vaut "Autre", alors la ligne n'est ni une voiture ni un camion, mais juste un vehicule.
27. Depuis la fenêtre de persistance, créez une entité `Voiture` et ajoutez l'annotation :

```
@Entity
@DiscriminatorValue("Voiture")
public class Voiture extends Vehicule {
    ...
```

idem pour `Camion`.

28. Déplacez le champ, le getter et le setter de `volume` depuis la classe `Vehicule` vers la classe `Camion`.
29. Créez une classe `Client` en embedded avec l'id et le nom (utiliser le TP1).
30. Dans `Vehicule`, `Voiture` et `Camion`, ajoutez une action `louer` qui renvoie une phrase de location.
31. Créez un nouveau packages appelé `repositories` avec quatres classes : `ClientDAO`, `VeheculeDAO`, `VoitureDAO` et `CamionDAO`. Un DAO (ou un repository, c'est quasiment la même chose) est un objet qui fabrique des entités.
32. Précédez chacune de ces classes par l'annotation `@Repository` et ajoutez un attribut `EntityManager` :

```
@Repository
public Class ClientDAO {
    @PersistenceUnit
    EntityManager em
    ...
```

33. Dans `vehicule`, ajoutez une méthode `List<Vehicule> getAll()` qui récupère tous les véhicules.
34. Ajoutez une méthode `List<Vehicule> getAllFree()` qui récupère tous les véhicule qui sont libre (aucun client n'est associé).
35. Faites de même dans `VoitureDAO` et `CamionDAO`.

36. (difficile, optionel) Factorisez votre code.
37. Ajoutez un `getAll` dans `Client`.
38. Ajoutez, dans `ClientDAO`, un `@Bean` nommé `default` qui fournira un client construit à partir du contexte. Pour l'instant on met des valeurs bidons (faites tout de même un compteur pour l'identifiant afin de ne pas avoir deux même identifiants).
39. Ajoutez, dans `VehiculeDAO`, un `@Bean` nommé `default` qui fournit un une voiture libre s'il y en a une, ou un autre vehicule (on suppose qu'il y a assez de vehicule au total).
40. La méthode `default` doit alors indexer le client courant (récupéré du context) comme client ayant loué la voiture.
41. (difficile, optionel) Faites en sorte d'éviter la location concurrentielle du même vehicule.
42. Dans le `HomeController`, appelez la location d'un vehicule récupéré du contexte.