Spring 3.X

Spring AOP

plan

- Définition
- Vocabulaire
- Spring AOP
- ProxyFactory
- Advice, pointcut et advisor
- AspectJ
- Advice avec AspectJ
- Ajout de l'aspect
- Expression du pointcut
- Join points
- Annotations

Définition

- La Programmation Orientée Aspect (POA ou AOP) permet d'intervenir comme son nom l'indique sur un ou plusieurs 'Aspects' d'une application :
  - Gestion transactionnelle
  - Sécurité
  - etc...
- Elle ne doit pas être opposée à la Programmation Orientée Objet, les deux sont complémentaires

Vocabulaire

- Weaving : processus qui va effectuer la mise en relation des objets avec les aspects. (le comment)

On dit que les objets sont alors 'avisés' (target objects)

- Join point ou point de jointure : point où va se situer la jointure entre le code et les aspects. Ces points peuvent se situer (le où) :
  - A l'exécution d'une méthode
  - Au traitement d'une exception
  - A l'utilisation d'un constructeur
  - etc...
- Pointcut : Condition d'intervention de l'aspect (le quand)
- Advice : Opération effectuée par l'aspect (le quoi)
- Advisor : Regroupement du pointcut et de l'advice

Spring AOP

- Spring n'offre pas toute la palette des possibilités offertes par l'AOP
- En revanche Spring autorise l'intégration du framework tiers AspectJ pour les fonctionnalités qu'il n'est pas en mesure de gérer
- Spring AOP permet uniquement les points de jointure au niveau des méthodes
- L'intégration des target objects se fait à base de proxies. Les proxies étant fournis par une classe `ProxyFactory`

ProxyFactory

- Le `ProxyFactory` va permettre d'obtenir un objet de classe « proxy » sur le « Target object »

```
ProxyFactory factory = new ProxyFactory(unComposant);
factory.addInterface(IComposant.class);
factory.addAdvice(new MyAdvice());
IComposant proxy = (IComposant) factory.getProxy();

// cette méthode est appelée à partir du proxy!
proxy.method();
```

Target object

Interface

Advice

- Au passage on aura fourni l'advice que l'on souhaite mettre en oeuvre. Celui-ci intervient dans l'exemple de manière globale. En effet, nous ne précisons aucun pointcut, l'advice sera donc appliqué quelle que soit la méthode appelée. Mais quand intervient-il et que contient-il ?

Advice

- L'advice peut être exécuté à différents instants
  - Before : l'opération sera effectuée avant la méthode (join point) et ne pourra interrompre l'exécution de la méthode
  - After returning : l'opération s'effectue à la fin normale de la méthode
  - After throwing : l'opération s'effectue à la levée d'une exception de type `RuntimeException` par la méthode
    - Dans le cas de Spring AOP, il n'y en a pas d'autres

Advice

- A quoi ressemble alors le code de l'advice ?

```
class MyAdvice implements MethodBeforeAdvice{
      public void before(Method method, Object[] args, Object target) throws
      Throwable {
            System.out.println("Execute avant la methode : " + method.getName());
            }
      }
```

- Dans le code ci-dessus, l'advice est déclenché avant l'exécution de la méthode car la classe implémente `MethodBeforeAdvice`.
  - On aurait pu également implémenter :
    - `AfterReturningAdvice`
    - `ThrowsAdvice`

TP n°1

- Voir document dédié.

Pointcut et advisor

- Si nous n'allons pas étudier plus avant Spring AOP, ceux qui souhaiterons utiliser Spring AOP devront avoir à connaître 2 fonctionnalités supplémentaires importantes.
- Tout d'abord, il aurait bien-sûr été possible d'appliquer l'advice sur un pointcut spécifique. Pointcut et Advice seront alors regroupés au sein d'un advisor.
  - Nous aurions alors utilisé la méthode `addAdvisor()` du `ProxyFactory`
  - L'advisor le plus générique est probablement représenté par la classe `RegExpMethodPointcutAdvisor`

```
ProxyFactory factory = new ProxyFactory(unComposant);
factory.addInterface(IComposant.class);
factory.addAdvisor(new MyAdvisorOnGetMethods());IComposant
proxy = (IComposant) factory.getProxy();

// cette méthode est appelée à partir du proxy!
proxy.method();
```

Advisor

ProxyFactoryBean

- Ensuite, instancier manuellement dans le code un target object à chaque fois que l'on a besoin d'effectuer une opération susceptible de bénéficier d'aspects est somme toute plutôt contraignant.
- Il est possible d'automatiser l'instanciation de proxy par le biais de Spring via un `ProxyFactoryBean` déclaré dans la configuration.
  - Un `ProxyFactoryBean` par Objet sera nécessaire
- Attention, les proxys sont alors instanciés sur la base du constructeur sans paramètre, il faudra donc prendre garde à bien les conserver.

ProxyFactoryBean

Exemple :

```
<bean id="myinterfaceproxybean"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="proxyInterfaces">
      <value>com.jnesis.IComposant</value>
  </property>
  <property name="target">
      <ref local="myimpl" />
  </property>
  <property name="interceptorNames">
      <list>
          <value>myAdvisorOnGetMethods</value>
      </list>
  </property>
</bean>

<!-- Bean Classe -->
<bean id="myimpl" class="com.jnesis.Composant" />

<!-- Advisor -->
<bean id="myAdvisorOnGetMethods"
      class="com.jnesis.MyAdvisorOnGetMethods">
</bean>
```

ProxyFactoryBean

Le target object

L'advisor

AspectJ

- Certes Spring AOP offre des possibilités que nous n'avons pas vues, mais comme exposé en introduction Spring AOP ne peut pas tout faire
- Si Spring AOP ne fournit pas la fonctionnalité AOP désirée, on utilisera alors AspectJ
- Tout comme Spring AOP, nous utiliserons AspectJ comme “tisseur d'aspect” s'effectuant à l'exécution (par opposition aux tisseurs d'apects s'effectuant à la compilation).
- AspectJ est plus complet que Spring AOP

AspectJ

- Pour utiliser AspectJ il faudra ajouter 1 librairie :
  - aspectjweaver.jar
- De plus, dans la configuration Spring il faudra rajouter l'espace de nommage aop :
  - `xmlns:aop="http://www.springframework.org/schema/aop"`
  - Et ajouter au xsi:schemaLocation :
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-3.0.xsd

Advice avec AspectJ

- Contrairement à Spring AOP, les Advices ne sont pas « typés », pas d'interfaçage nécessaire.

```
class MyAdvice {
      public void doSomething() {
            System.out.println("On ne sait pas quand la methode sera executee...");
            }
      }
```

Le moment où sera effectué l'appel de `doSomething` est encore à ce niveau indéterminé.

Ajout de l'aspect

- L'aspect sera ajouté à la configuration et on y spécifiera :
  - La méthode cible à exécuter (méthode de l'advice)
  - A quel moment déclencher l'advice (join point)
  - Le pointcut à vérifier

```
<bean id="myAdvice" class="formation.MyAdvice"/>

<aop:config>
      <aop:aspect id="myAspect" ref="myAdvice">
            <aop:pointcut id="coupe"
                  expression="execution(* formation.ActeurService.getActeursList())"/>

            <aop:before method="doSomething" pointcut-ref="coupe"/>
      </aop:aspect>
</aop:config>
```

- Il n'y a aucun autre élément de code à rajouter !
  - Le proxy existe toujours mais il est instancié de façon transparente sur la base du constructeur sans paramètre (comme nous l'avions introduit avec la classe ProxyFactoryBean de Spring AOP)

Ajout de l'aspect

- On remarquera que le type de déclenchement de l'advice est précisé dans l'exemple par `aop:before`. On pourra utiliser d'autres types, même plus étendus que ceux disponible avec Spring AOP
  - `aop:after-returning` ... après retour de la méthode
  - `aop:after-throwing` ... si exception
  - `aop:after` ... dans les 2 cas précédents (dans un finally en somme)
  - `aop:around` ... c'est le type le plus complet. Il permet notamment de contrôler le traitement de la méthode cible (ou du constructeur cible).
    - Les 2 derniers types en souligné n'étaient pas disponibles avec Spring AOP seul

Join points

- Attention, les méthodes déclenchées suite à un join point de type `aop:before`, `aop:after-returning` ou `aop:after-throwing` devront retourner « void ».
  - Nous le verrons, `aop:around` retournera quant à lui `Object`

Expression du pointcut

- On remarquera de plus que la syntaxe d'écriture de la condition de déclenchement de l'advice est assez riche :

```
<bean id="myAdvice" class="formation.MyAdvice"/>

<aop:config>
      <aop:aspect id="myAspect" ref="myAdvice">
            <aop:pointcut id="coupe"
                  expression="execution(* formation.ActeurService.getActeursList())"/>
            <aop:before method="doSomething" pointcut-ref="coupe"/>
      </aop:aspect>
</aop:config>
```

L'exemple signifie : à l'execution de la méthode `getActeursList()` de tout objet issu de la classe ActeurService

Le prefixe * indiquant ici :quelle que soit la signature de retour de la méthode.

Expression du pointcut

- Le mot clé « execution » est le mot clé le plus utilisé et permet de cibler très précisément la (les) méthodes de Bean qui définissent le pointcut
- Il existe cependant d'autres mots clés tel « within » ou « args » par exemple qui ciblent d'autres mécanismes de reconnaissance de pointcut
- Quel que soit le mot clé utilisé, la condition du pointcut pourra contenir des « and » (ou &&), « or » (ou ||) ou des « not » (ou !) - *signifiant « and not »* - pour combiner les expressions, exemple :
  - within(formation..*) || execution (* toto.Service.get*())
  - within(formation..*) not execution (* toto.Service.get*())
    - Attention : && s'écrit && dans le fichier XML

Expression du pointcut

- La syntaxe de la condition exprimée grâce au mot clé « execution » comporte :
  - Une signature de retour
    - *
    - public * : les méthodes public
    - public void etc...
  - Une méthode à exécuter
    - x.y.MaClass.maMethode
    - *.MaClass.* : toute méthode de toutes classes MaClass
    - x.y.MaClass*.maMethode : méthode maMethode de toute classe commencant par MaClass
    - etc...
  - Une signature de méthode
    - (String) : méthode acceptant un String
    - (..) : toutes signatures
  - Éventuellement les exceptions de la méthode

Ajout de l'aspect

On pourra déclarer dans un aspect plusieurs pointcuts, et lier plusieurs advices à l'un ou l'autre pointcut selon différents join points, par exemple :

```
<bean id="myAdvice" class="formation.MyAdvice"/>

<aop:config>
      <aop:aspect id="myAspect" ref="myAdvice">
            <aop:pointcut id="coupe"
                  expression="execution(* formation.ActeurService.getActeursList())"/>
            <aop:pointcut id="coupe2"
                  expression="execution(* formation.ActeurService.load*(..))"/>
            <aop:before method="doSomething" pointcut-ref="coupe"/>
            <aop:after-returning method="doSomethingElse" pointcut-ref="coupe"/>
            <aop:after-returning method="doSomethingElse" pointcut-ref="coupe2"/>
      </aop:aspect>
</aop:config>
```

2 advices

- Méthode doSomething lancée avant le pointcut « coupe »
- Méthode doSomethingElse après le pointcut « coupe »
- Méthode doSomethingElse après le pointcut « coupe2 »

TP n°2

- Voir document dédié.

aop:around

La balise `<aop:around>` permet de **contrôler** l'exécution au niveau du target object (méthode ou constructeur). Ceci est rendu possible en exécutant la méthode `proceed()` de l'instance de `ProceedingJoinPoint` reçue en paramètre :

```
public Object doSomething(ProceedingJoinPoint joinPoint) throws Throwable {
      if (test) {
            return joinPoint.proceed();
      }
      return null;

}
```

Déclenchement au niveau du target object

La méthode de l'advice retourne désormais Object. Il s'agit traditionnellement du retour de la méthode du target object (s'il s'agit d'une méthode).

`ProceedingJoinPoint` va référencer toutes les données afférente à l'événement source :

```
public Object doSomething(ProceedingJoinPoint joinPoint) throws Throwable {
      String methodName=joinPoint.getSignature().getName();
      Object args[]=joinPoint.getArgs();


      System.out.println("Méthode invoquée "+methodName);
      return joinPoint.proceed();

}
```

Nom de la méthode (ou constructeur) du target object

Arguments de la méthode (ou constructeur) du target object

TP n°3

- Voir document dédié.

Annotations

- Avec Java 5, il est possible grâces aux annotations d'éviter la fastidieuse opération de configuration des aspects AspectJ dans le fichier de configuration Spring
- Si l'on utilise les annotations Java 5, on dit alors que les classes sont auto-proxiées (autoproxy), et non plus à l'inverse proxiées par la configuration Spring.
  - On entend par auto-proxiées le fait que Spring génère automatiquement le proxy sur détection du fait que la classe est soumise à un Aspect (et non plus du fait de la configuration aop:config).
- Spring doit être alerté de l'utilisation de l'auto-proxying ; on ajoutera dans la configuration :
  - <aop:aspectj-autoproxy/>

```
<beans xmlns="http://www.springframework.org/schema/beans"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xmlns:aop="http://www.springframework.org/schema/aop"
      xsi:schemaLocation="http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
      http://www.springframework.org/schema/aop
      http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">

  <aop:aspectj-autoproxy/>
```

Annotations

Pour indiquer que notre Advice va traiter un aspect, on utilise l'annotation `@Aspect` au niveau de la classe comme suit :

```
@Aspect
class MyAdvice {
	public void doSomething {
		System.out.println("On ne sait pas quand la methode sera executee...");
		}
	}
```

Join points et pointcuts seront déclarés via les annotations `@Before`, `@AfterReturning` ....

On pourra encore une fois bien évidemment ajouter plusieurs pointcuts à une méthode

```
@Aspect
class MyAdvice {
	@Before("execution(* *.ActeurService.getActeursList())")
	public void doSomething {
		System.out.println("On ne sait pas quand la methode sera executee...");
		}
	}
```

Annotations

- Attention, le fait d'ajouter `<aop:aspectj-autoproxy/>` et `@Aspect` ne vous libère pas de la nécessité de déclarer l'advice dans la configuration Spring.
  - Si vous voulez vous en passer, il faudra comme indiqué précédemment ajouter une balise `<context:component-scan>` et annoter l'advice avec `@Component`.
    - Attention ! Les controleurs sont détectés par le biais d'un `<context:component-scan>` dans le fichier actions-servlet.xml. Si les services annotés injectés dans ces contrôleurs ont malencontreusement été placés dans le même packages, il seront alors injectés non avisés !

TP n°4

- Voir document dédié.