

SPRING

Présentation de Spring

Problématiques des développements Java EE

- Malgré une simplification progressive, la plate-forme Java reste complexe, demandant des connaissances techniques approfondies.
- Développer des applications avec une approche 100 % Java EE révèle cependant quatre faiblesses récurrentes :

Présentation de Spring

- **Mauvaise séparation des préoccupations** : des problématiques d'ordres différents (technique et métier) sont mal isolées.
- **Complexité** : Java EE reste complexe, notamment à cause de la pléthore de spécifications disponibles.
- **Mauvaise interopérabilité** : malgré la notion de standard, les technologies ne sont pas toujours interoperables et portables.
- **Mauvaise testabilité** : les applications dépendant fortement de l'infrastructure d'exécution, elles sont plus difficilement testables.

Présentation de Spring

L'état de l'art des bonnes pratiques de conception d'une application.

- Dans le **développement logiciel**, la division en couches est une technique répandue pour décomposer logiquement un système complexe.
- **La décomposition en couches** revient à modéliser un système en un arrangement vertical de couches.
- **Chaque couche constitue l' une des parties du système** : en tant que telle, elle a des responsabilités et s' appuie sur la couche immédiatement inférieure.

Présentation de Spring

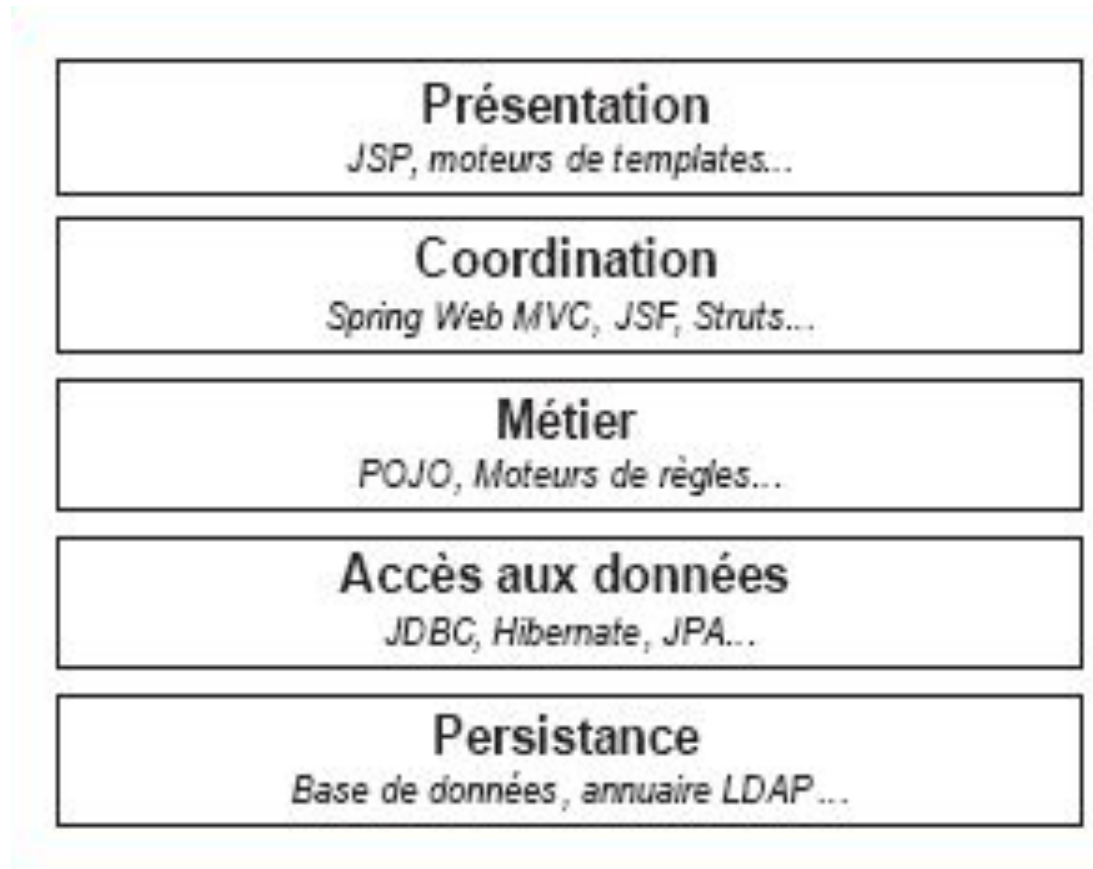


Illustration (Modèle à cinq couches et technologies associées)

Présentation de Spring

- **La communication entre couches suit des règles très strictes :**
 - ❑ une couche ne connaît que la couche inférieure ;
 - ❑ ne doit jamais faire référence à la couche supérieure.
- En Java (ou en termes de langage objet), la communication entre couches s'effectue en établissant des contrats **via des interfaces**.
- La difficulté d'une décomposition en couches est **d'identifier** les couches du système, c'est-à-dire **d'établir leurs responsabilités** respectives et leur façon de communiquer.

Présentation de Spring

Avantages:

- **possibilité d'isoler une couche**, afin de faciliter sa compréhension et son développement ;
- très bonne **gestion des dépendances** ;
- possibilité de **substituer** les implémentations de couches ;
- **réutilisation** facilitée ;
- **testabilité** favorisée (grâce à l'isolement et à la possibilité de substituer les implémentations).

Présentation de Spring

Inconvénients:

- Un système trop décomposé peut s'avérer difficile à appréhender conceptuellement.
- Une succession de couches peut avoir un effet négatif sur les performances, par rapport à un traitement direct.

Présentation de Spring

La programmation par interface

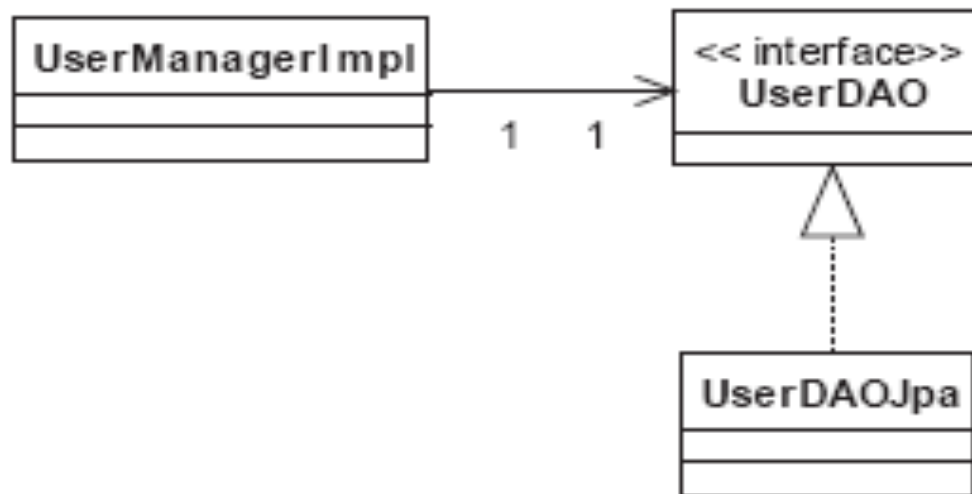
- Dans le modèle à cinq couches que nous avons présenté, les services métier se fondent sur des DAO pour communiquer avec l'entrepôt de données, généralement une base de données.
- Chaque service métier contient des références vers des objets d'accès aux données.
- Exemple :



Le service métier **dépend fortement** de l'implémentation du DAO, d'où un couplage fort.

Présentation de Spring

- Pour réduire ce couplage, il est non seulement nécessaire de définir une interface pour le DAO, mais que le service se repose sur celle-ci.

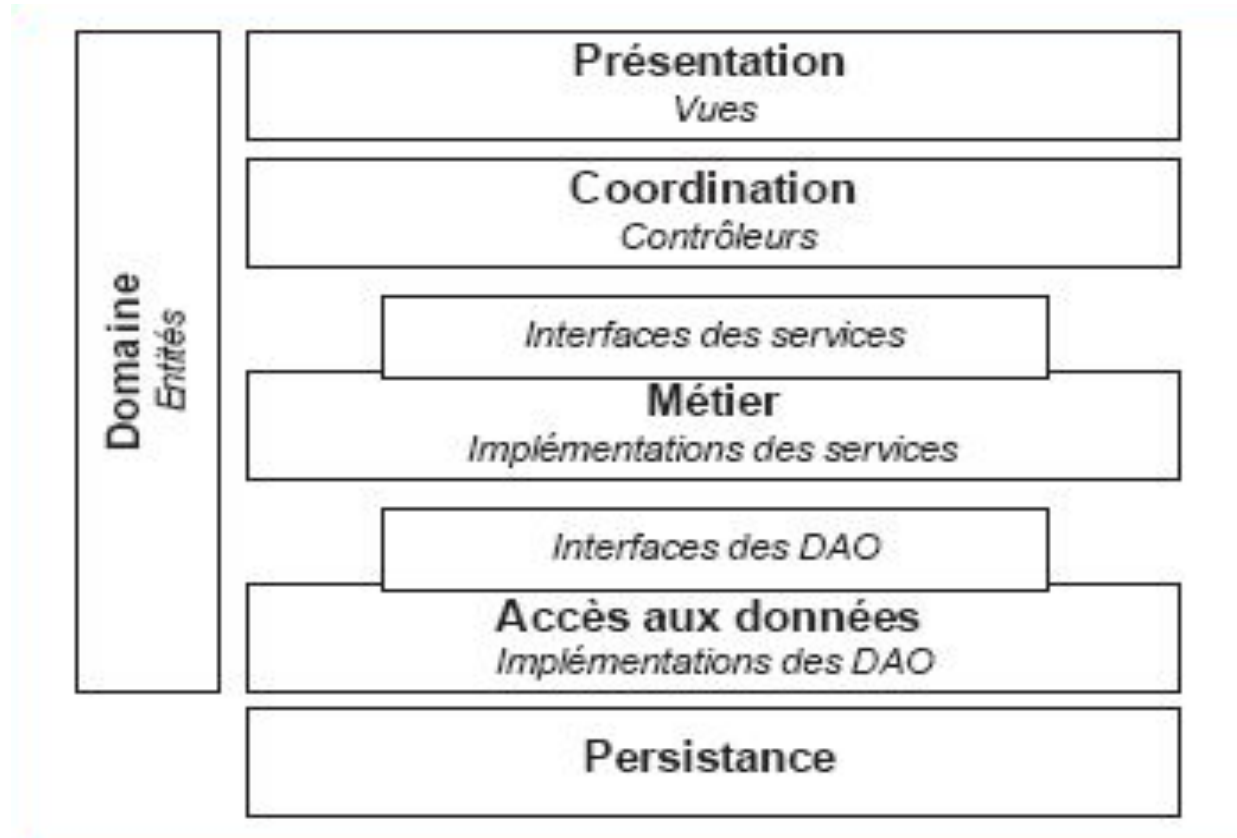


Présentation de Spring

L' utilité de Spring dans un tel modèle.

- Spring est en réalité capable d' intervenir au niveau de chacune des couches.
- Dans un langage orienté objet tel que Java, le contrat de communication entre les couches s' effectue grâce à la notion d' interface.
- Les couches métier et d' accès aux données ne sont connues que par les interfaces qu' elles exposent, comme le montre la figure suivante :

Présentation de Spring



Communication inter couche par le biais des interfaces

Présentation de Spring

- Dans une application Spring, l'ensemble des composants (contrôleurs, services et DAO) sont gérés par **le conteneur léger**.
- Celui-ci se charge de leur **cycle de vie**, ainsi que de leurs **interdépendances**.
- Grâce à la **POA**, ces composants peuvent aussi être décorés, c'est-à-dire que des opérations telles que **la gestion des transactions** peuvent leur être ajoutées, et ce de façon transparente.

Présentation de Spring

- Spring est un framework de développement Java basé sur le concept de « **conteneur léger** »

Spring en bref :

- début du projet en 2002
- créé par **Rod Johnson (et Juergen Holler)**
- projet **Open Source**

❑ site : <http://www.springsource.org/>

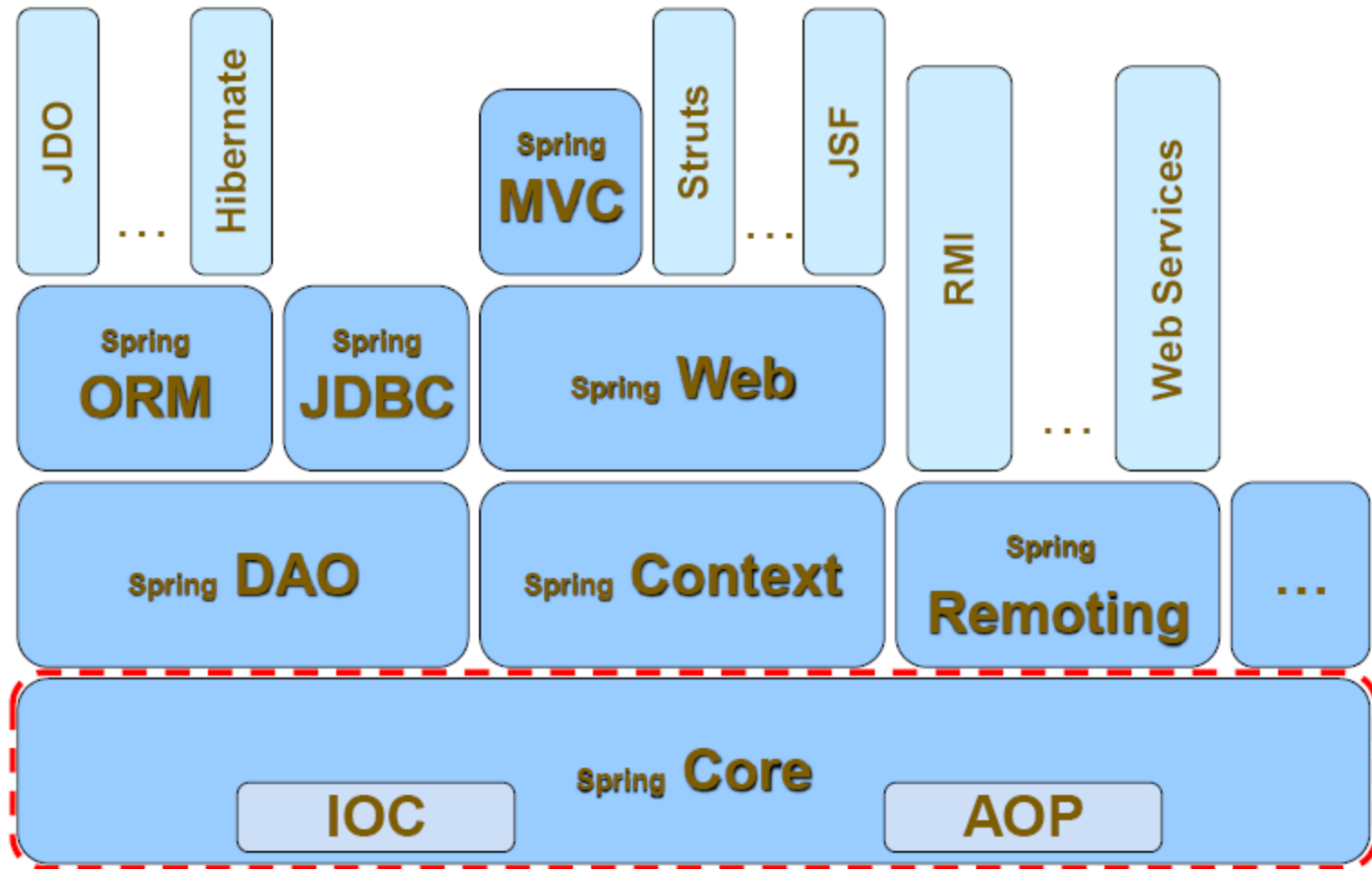
- «SpringSource» appartient maintenant à «VMware»

Présentation de Spring

Les principales versions :

- Spring 1.0 (Mars 2004), Spring 2.0 (Oct 2006),
- Spring 2.5 (Nov 2007, annotations, ...)
- Spring 3.0(Dec 2009)
 - ❑ Spring 3.0.4 (2010)
 - ❑ Spring 3.0.6 (2011)
 - ❑ Spring 3.1.2 (2012-07-09)
- **Spring 3.0 version majeure, révisée pour tirer profit de Java 5(Generics, Varargs, ...)**

Présentation de Spring



Spring est composé de nombreuses "briques" qui peuvent être utilisées indépendamment les unes des autres ...

Présentation de Spring

- **Core**, le noyau, qui contient à la fois un ensemble de classes utilisées par toutes les briques du framework et le conteneur léger.
- **AOP**, le module de programmation orientée aspect.
- **DAO**, qui constitue le socle de l'accès aux dépôts de données, avec notamment une implémentation pour JDBC. La solution de gestion des transactions de Spring fait aussi partie de ce module.
- **ORM**, qui propose une intégration avec des outils populaires de mapping objet-relationnel, tels que Hibernate, JPA, EclipseLink ou iBatis.

Présentation de Spring

- **Java EE**, un module d'intégration d'un ensemble de solutions populaires dans le monde de l'entreprise.
- **Web**, le module comprenant le support de Spring pour les applications Web.

Il contient notamment Spring Web MVC, la solution de Spring pour les applications Web, et propose une intégration avec de nombreux frameworks Web et des technologies de vue.

- **Test**, qui permet d'appliquer certaines techniques du conteneur léger Spring aux tests unitaires *via* une intégration avec JUnit et TestNG.

Présentation de Spring

- Un ensemble de projets gravitent autour de Spring, utilisant le conteneur léger et ses modules comme bases techniques et conceptuelles :
 - **Spring Web Flow**
 - **Spring Web Services**
 - **Spring Security**
 - **Spring Batch**
 - **Spring LDAP**
 - ...

Présentation de Spring

Notion de "**conteneur léger**"

- Se définit essentiellement par opposition aux conteneurs dits "lourds", par exemple les conteneurs d'EJB (en particulier avant les EJB 3) :
 - Environnement souvent inadapté pour gérer des composants simples
 - Règles de développement et de déploiement contraignantes
- Avantages d'un conteneur léger :
 - Pas de "couplage" entre le conteneur et les composants qu'il gère (non intrusif)
 - N'importe quel objet peut être géré par le conteneur
 - Rien à installer (code Java + .jar : c'est tout !)

Présentation de Spring

Les concepts du "conteneur léger" Spring

- "Séparation des préoccupations" (**Separation of Concerns**)
- "Inversion de contrôle" (**IoC**) / "Injection de dépendances" (**DI**)
- Gestion du cycle de vie des objets(Factory, Singleton, ...)
- "Programmation Orientée Aspect" (**AOP: Aspect Oriented Programming**)

Présentation de Spring

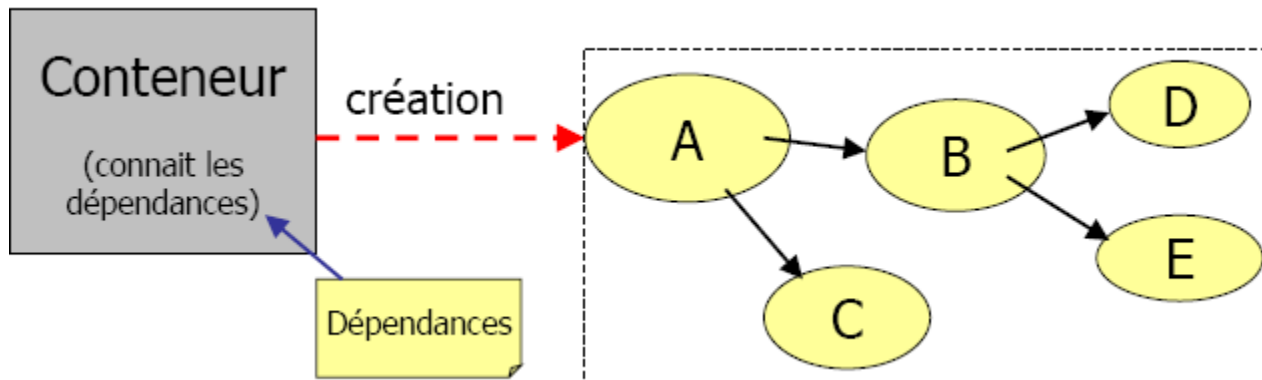
Separation Of Concerns

- Principe qui consiste à isoler les différentes problématiques à traiter, par exemple :
 - ❑ Présentation (IHM, ...)
 - ❑ Services Métiers
 - ❑ Persistance
 - ❑ etc...
- => notion de **composants spécialisés**

Présentation de Spring

Injection de dépendance/Inversion Of Control

- Pour pouvoir fonctionner un composant a généralement besoin d'autres composants ...
- L'injection de dépendance permet de "fabriquer" les composants et de leur "injecter" automatiquement les composants nécessaires



Présentation de Spring

Approche "naïve"

Fabrication directe des composants nécessaires

"**A**" a besoin de "**B**" et de "**C**"

=> il les fabrique lui-même !

```
public A () {  
    this.b = new B();  
    this.c = new C();  
}
```

- * Si "B" et "C" peuvent être "mono-instance"
→ multiplication inutile des instances !
- * "A" doit connaître la classe concrète
(choix figé de l'implémentation, pas d'interface)
→ "couplage fort" !
- * Si les constructeurs évoluent => impact



Présentation de Spring

Approche par "factory" ou "provider"

Délégation de la fabrication des instances
à une "factory"
(un objet "fournisseur" d'instances)

```
public A () {  
    this.b = factory.getB();  
    this.c = factory.getC();  
}
```

```
public A () {  
    this.b = p.lookup("bb");  
    this.c = p.lookup("cc");  
}
```

```
// Exemple : JNDI  
ejbHome = initialContext.lookup("java:comp/env/abc/AccountEJB");
```

C'est mieux ...
· pas de "new"
· interfaces utilisables



Mais il reste des
contraintes ...

* Dépendance à une méthode
ou à un nom symbolique ("lookup")



* Les composants sont dépendants de la "factory"
(ou du "provider")

Présentation de Spring

Approche "idéale"

Objectifs :

- "A" ne crée pas les composants dont il a besoin
- "A" ne récupère pas lui-même ces composants par un tiers
- "A" est totalement autonome

Comment ?

- Un objet externe va lui "injecter" les composants dont il a besoin
- Le cycle vie de "A" est donc géré par cet objet "invisible" : le conteneur de composants

Le **conteneur** crée les composants
et leur injecte les dépendances nécessaires

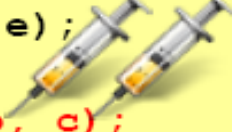


Présentation de Spring

Comment faire de l'injection de dépendance en Java ?

- Par le **constructeur**

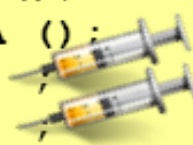
```
B b = new B(d,e);  
C c = new C();  
A a = new A(b, c);  
// Composant "a" prêt ...
```



Plus sécurisant
(objet correctement
initialisé dès
sa création)

- Par les **"setters"**

```
B b = new B(d, e);  
C c = new C();  
A a = new A();  
a.setB(b);  
a.setC(c);  
// Composant "a" prêt ...
```



Plus pratique
(plus simple en
cas d'héritage
de configuration
dans Spring)

Présentation de Spring

Avantages des conteneurs légers

Construction d'applications par **assemblage de composants**

Le conteneur permet d'obtenir un "**couplage faible**" (ou "*couplage lâche*" = "*loosely coupled*") :

→ peu de dépendance entre les composants

Pas d'adhérence avec le conteneur

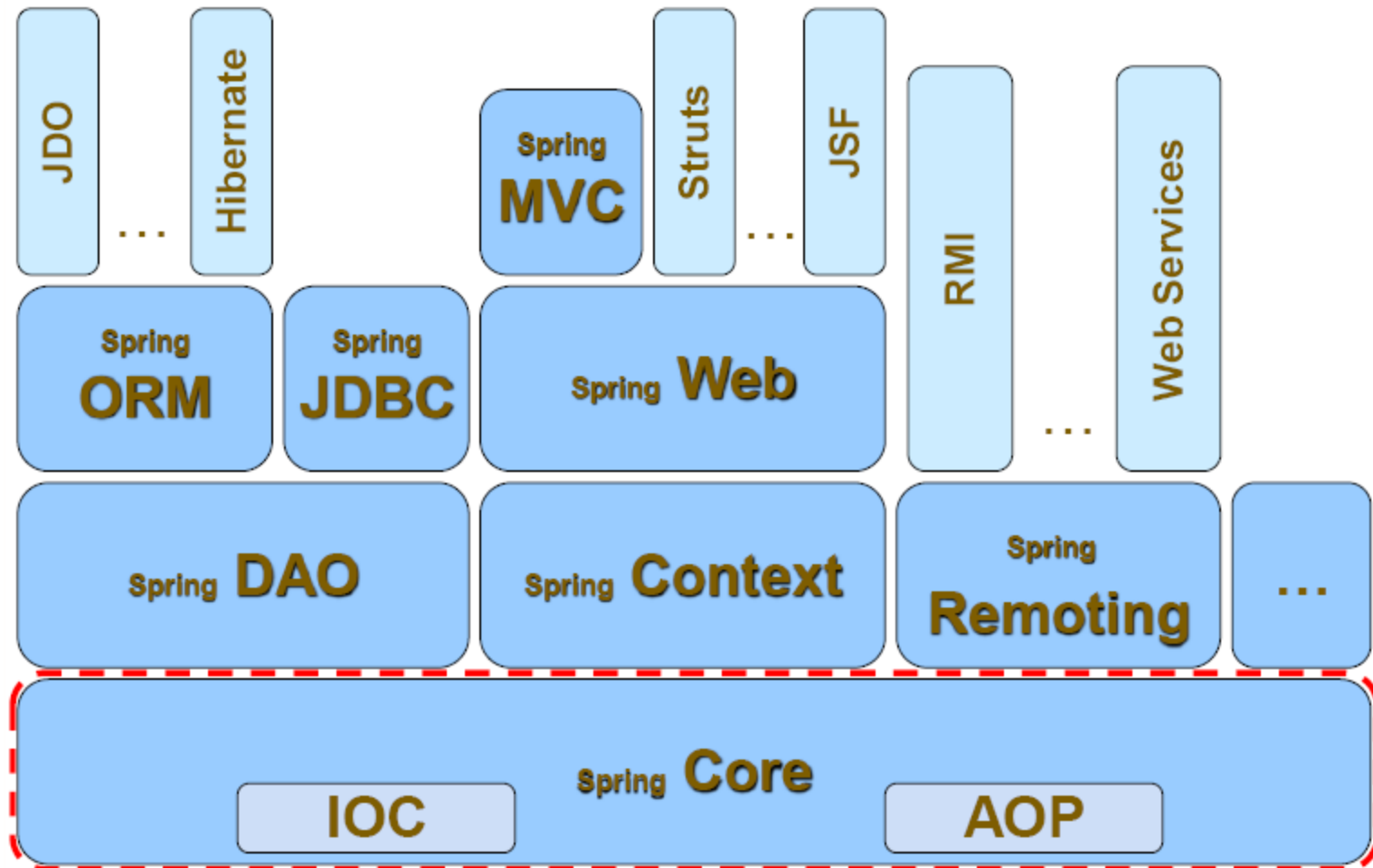
- Chaque composant est un simple "POJO" (Plain Old Java Object), pas d'héritage



Travaux pratiques

- **Installation de Java (Lab 1)**
- **Installation de Eclipse (Lab 2)**
- **Installation de SpringTools (plugin eclipse) (Lab 3)**

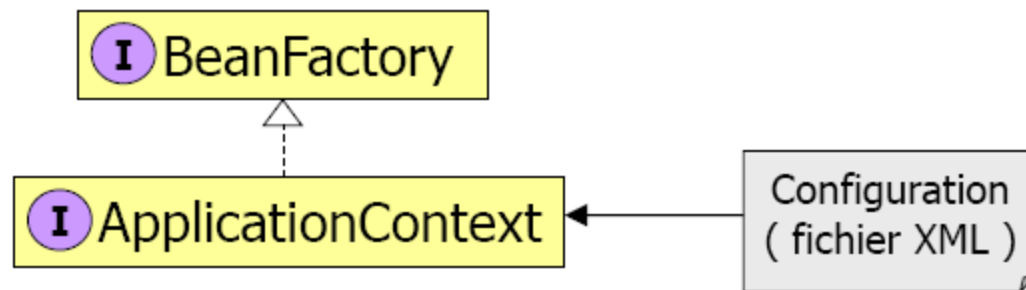
Spring Core



Spring est composé de nombreuses "briques" qui peuvent être utilisées indépendamment les unes des autres ...

Spring Core

- **"Spring core"** est le noyau du framework
- Principaux éléments du "conteneur léger" :
 - Les **"beans"** : objets gérés par le conteneur léger
 - La **"fabrique de bean"** ("BeanFactory") : Mécanisme de configuration qui permet de gérer des "beans" de différentes natures (singleton/prototype)
 - Le **"contexte d'application"** ("ApplicationContext") : extension de la BeanFactory, concrètement un contexte est initialisé à partir d'un fichier de configuration (XML)

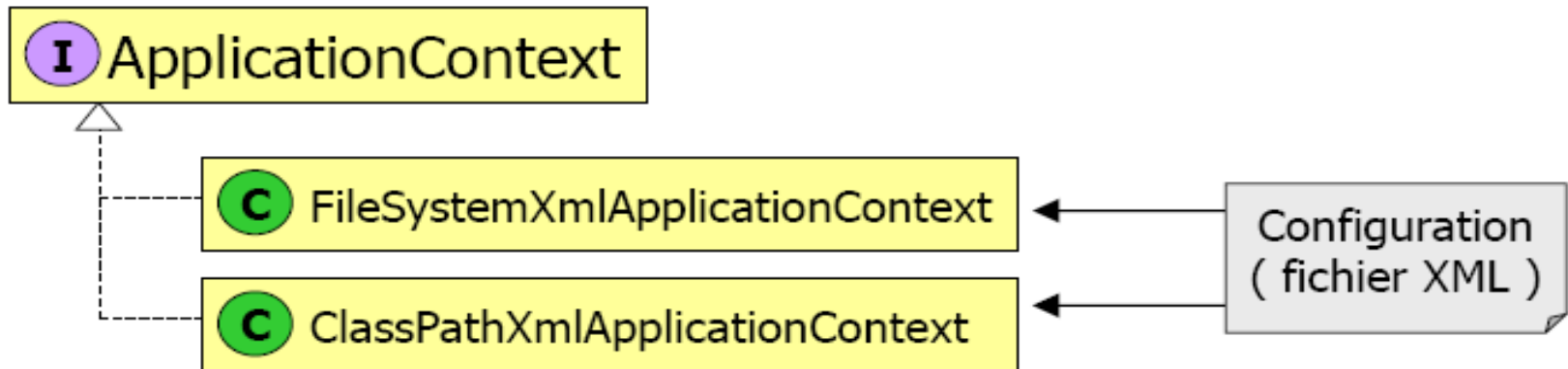


Spring Core

L'application utilise Spring à travers l'interface **"ApplicationContext"**

"ApplicationContext" dispose de plusieurs implémentations, dont ...

- **FileSystemXmlApplicationContext**
(charge la configuration xml à partir du « file system »)
- **ClassPathXmlApplicationContext**
(charge la configuration xml à partir du « ClassPath »)



Spring Core

Application Context

Exemple:

- Chargement via le « filesystem path » du fichier

```
ApplicationContext ac =  
    new FileSystemXmlApplicationContext("C:/dir/myContext.xml");
```

- Chargement via le « class path »

```
ApplicationContext ac =  
    new ClassPathXmlApplicationContext("myContext.xml");
```

- Dans une Servlet (chargé au démarrage dans le ServletContext)

```
public void doGet(HttpServletRequest request, HttpServletResponse  
response)throws ServletException, IOException {  
    ...  
    ServletContext context = getServletContext();  
    WebApplicationContext ac =  
        WebApplicationContextUtils.getWebApplicationContext(context);  
}
```

Spring Core

EXEMPLE « HELLOWORLD » avec SPRING

Le bean "**HelloWorld**"

```
package com.formation.hello;  
  
public class HelloWorld {  
    public void hello() {  
        System.out.println("Bonjour !");  
    }  
}
```

Utilisation classique (sans Spring)

```
HelloWorld hello = new HelloWorld();  
hello.hello();
```

Spring Core

Utilisation avec Spring

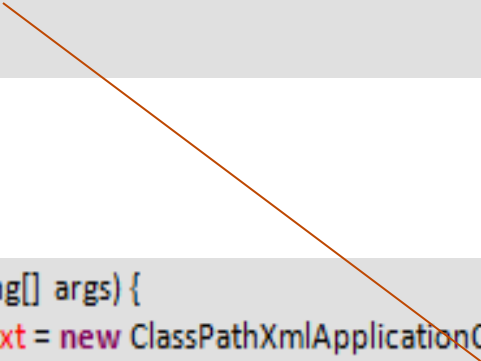
beans.xml

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

    <bean id="helloWorld" class="com.formation.hello.HelloWorld" />

</beans>
```

```
public static void main(String[] args) {
    ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
    HelloWorld helloWorld = (HelloWorld) context.getBean("helloWorld");
    helloWorld.hello();
}
```



Spring Core

Evolution : ajout d'une propriété

Le message « Bonjour...! » devient modifiable.

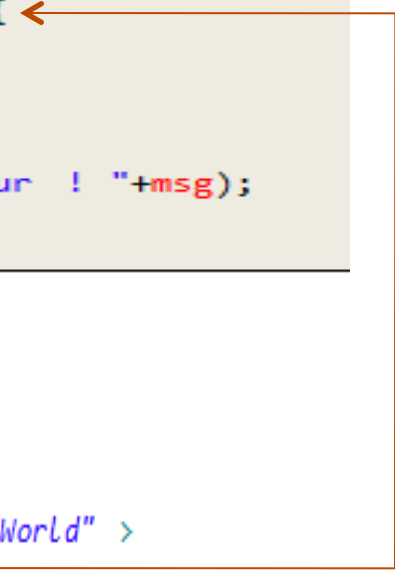
```
package com.formation.hello;

public class HelloWorld {

    private String msg = "Marc";

    public void setMsg(String msg) {
        this.msg = msg;
    }

    public void hello() {
        System.out.println("Bonjour ! "+msg);
    }
}
```



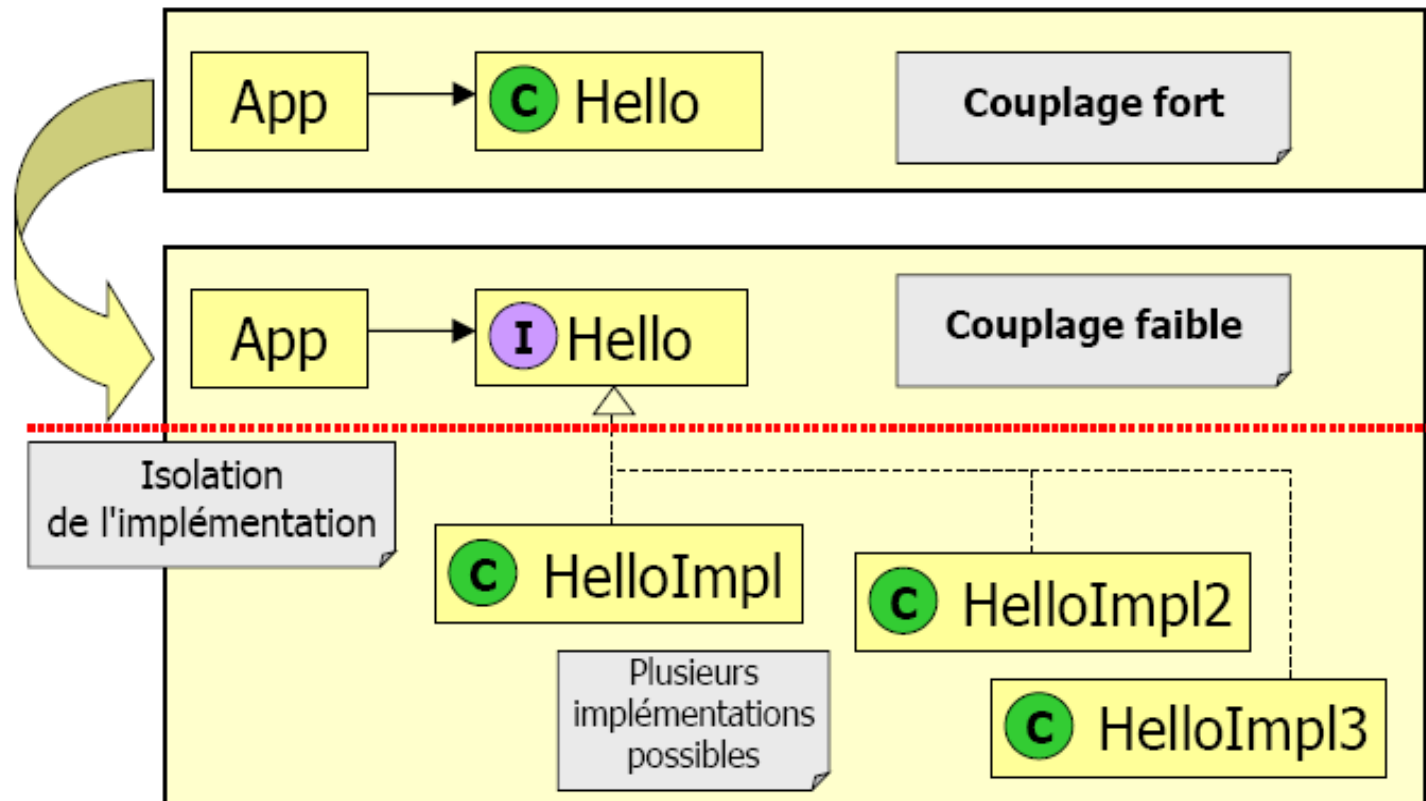
Affectation de la propriété dans le .xml :

```
<bean id="helloWorld" class="com.formation.hello.HelloWorld" >
    <property name="msg" value="Marc" />
</bean>
```

Spring Core

Evolution : ajout d'une interface

- L'application ne crée plus les objets elle-même, mais elle reste liée à la classe qui fournit le service
- => il est préférable de passer par une **interface**



Spring Core

Evolution : ajout d'une interface

■ Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  <!--
  <bean id="hello" class="my.package.HelloImplEN" />
  -->
  <bean id="hello" class="my.package.HelloImplFR" />
</beans>
```

Choix de l'implémentation
dans la configuration

Anglais

Français

■ Au niveau application ... aucun impact

```
Hello serv = (Hello) ac.getBean("hello");
serv.hello("Homer");
serv.hello("Bart");
```

Travaux pratiques

- **Exercice: 4_SpringCore_1**

Spring Core

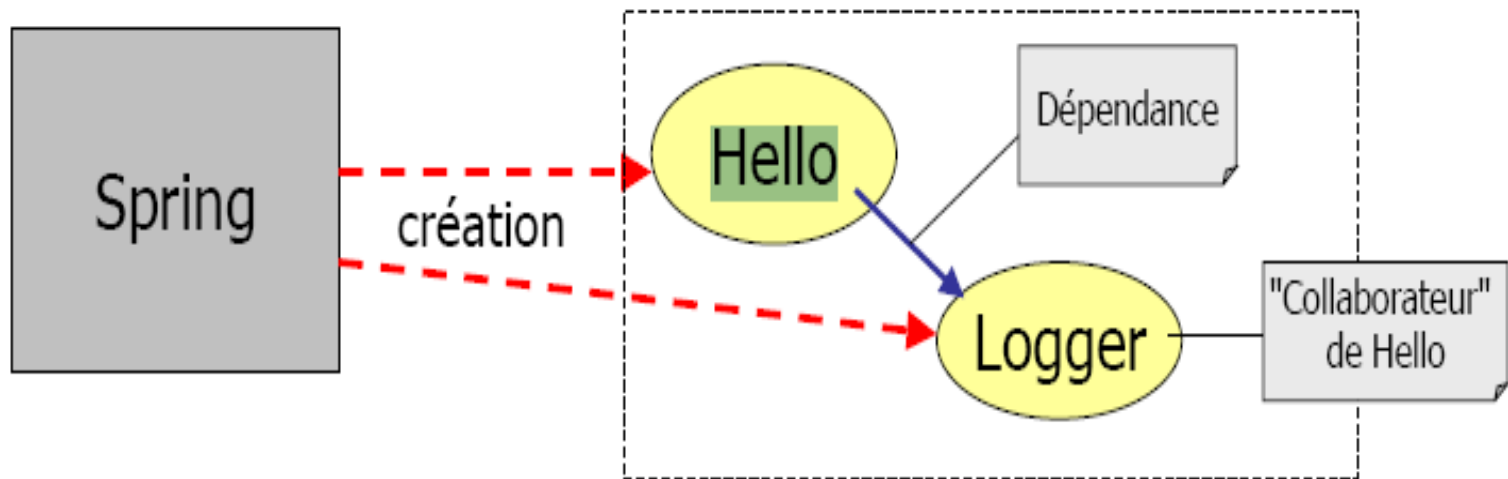
Injection de dépendances

- Spring permet de gérer les dépendances en toute transparence
 - Lorsque qu'un bean est fourni à l'application, toutes les dépendances déclarées ont été résolues par Spring
 - Le bean est "prêt à l'emploi"
- Les dépendances sont de deux types :
 - **Propriété(valeur) => "injection de propriété"**
 - **Collaborateur(objet) => "injection de collaborateur"**

Spring Core

Evolution : ajout d'un « collaborateur »

- Le service "Hello" doit utiliser un service de trace, pour cela il doit avoir une référence sur une instance de la classe "Logger"
- Spring désigne par "**collaborateur**" une propriété d'un bean qui est elle-même un bean géré par le conteneur



Spring Core

Evolution : ajout d'un "collaborateur"

```
public class HelloImpl implements Hello {  
    private Logger logger = null ;  
    ... ..  
    public void hello(String s) {  
        if ( logger != null ) logger.log("call hello");  
        ... ..  
    }  
    public void setLogger(Logger l) {  
        logger = l ;  
    }  
}
```

Modification
de
l'implémentation

```
<?xml version="1.0" encoding="UTF-8"?>  
<beans ...>  
    <bean id="hello" class="my.package.HelloImpl" >  
        <property name="msg" value="Bonjour" />  
        <property name="logger" ref="logger" />  
    </bean>  
    <bean id="logger" class="my.package.Logger" />  
</beans>
```

Configuration

Dépendance

Nouveau bean

Spring Core

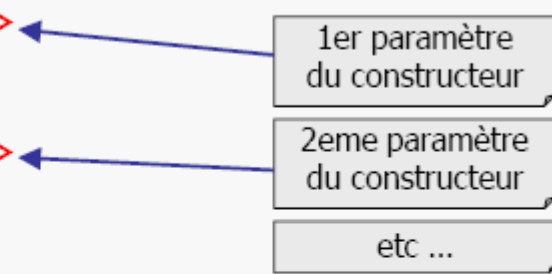
Injection de dépendances

- L'injection de dépendances peut être réalisée
 - par **modificateur**(en utilisant un "setter")
 - par **constructeur**

Spring Core

- Injection de dépendances **par constructeur**:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  <bean id="hello" class="my.package.HelloImpl" >
    <constructor-arg index="0" >
      <value>Bonjour</value>
    </constructor-arg>
    <constructor-arg index="1" >
      <ref bean="logger" />
    </constructor-arg>
  </bean>
  <bean id="logger" class="my.package.Logger" />
</beans>
```



1er paramètre
du constructeur

2eme paramètre
du constructeur

etc ...

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  <bean id="hello" class="my.package.HelloImpl" >
    <constructor-arg index="0" value="Bonjour" />
    <constructor-arg index="1" ref="logger" />
  </bean>
  <bean id="logger" class="my.package.Logger" />
</beans>
```

Forme plus
concise

Travaux pratiques

- Exercice: 4_SpringCore_2

Spring Core

Singleton ou prototype

- C'est Spring qui a fabriqué et renvoyé une instance de la classe Hello
- Pattern «**Factory**»
- Si on demande plusieurs fois le même bean ...

```
Hello serv = (Hello) ac.getBean("hello");  
serv.hello("Homer");  
Hello serv2 = (Hello) ac.getBean("hello");  
serv2.hello("Bart");
```

... il n'est créé qu'une seule fois

- Par défaut Spring considère que les beans sont des «**singletons**»

Spring Core

Exemple / singleton ou prototype

- Configuration du nombre d'instances des beans
- Dans le fichier de configuration XML :
 - Attribut « **scope** »

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  <bean id="hello" class="my.package.Hello" scope = |"singleton" />
  |"prototype"
</beans>
```

- Dans les anciennes versions de Spring
 - Attribut « **singleton** »

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  <bean id="hello" class="my.package.Hello" singleton = |"true" />
  |"false"
</beans>
```

Travaux pratiques

- Exercice: 4_SpringCore_3

Spring Core

Créer des beans en invoquant une méthode statique de fabrique

- Spring est capable de créer un bean en invoquant la méthode statique de fabrique précisée dans l'attribut **factory-method**.
- Nous pouvons par exemple encapsuler la procédure de création d'un objet dans une methode statique **createProduct** d'un autre objet **fabrique**.
- Le client qui demande un objet appelle simplement cette méthode sans avoir besoin de connaître les détails de la création.

Spring Core

Exemple :

- Nous pouvons par exemple écrire la méthode statique de fabrique **createProduct()** pour créer un produit à partir d'un identifiant.
- En fonction de cet identifiant, la méthode choisit la classe concrète à instancier.
- Si aucun produit ne correspond à l'identifiant, elle lance une exception **IllegalArgumentException**.

```
package com.formation.shop;

public class ProductCreator {
    public static Product createProduct(String productId) {

        if ("aaa".equals(productId)) {
            return new Battery("AAA", 2.5);
        } else if ("cdrw".equals(productId)) {
            return new Disc("CD-RW", 1.5);
        }
        throw new IllegalArgumentException("Produit inconnu");
    }
}
```

Spring Core

Exemple: Suite...

- Pour déclarer un bean créé à l'aide d'une méthode statique de fabrication, nous indiquons la classe qui contient la méthode dans l'attribut **class** et le nom de la méthode dans l'attribut **factory-method**.
- Les arguments de la méthode sont passés en utilisant des éléments **<constructor-arg>**.

```
<bean id="aaa" class="com.formation.shop.ProductCreator" factory-method="createProduct">  
    <constructor-arg value="aaa" />  
</bean>  
<bean id="cdrw" class="com.formation.shop.ProductCreator" factory-method="createProduct">  
    <constructor-arg value="cdrw" />  
</bean>
```

- Si la méthode de fabrication lance une exception, Spring l'enveloppe dans une exception **BeanCreationException**.

Travaux pratiques

- Exercice: 4_SpringCore_4

Spring Core

Cycle de vie d' un bean

- Outre l' enregistrement des beans, le conteneur Spring IoC est également responsable de la gestion de leur cycle de vie.
- Voici les étapes du cycle de vie d' un bean tel qu' établi par le conteneur Spring IoC.
 1. Créer l' instance du bean, en invoquant un constructeur ou une méthode de fabrique.
 2. Affecter des valeurs et des références de beans aux propriétés du bean.
 3. **Invoquer les méthodes de rappel pour l' initialisation.**
 4. Le bean est prêt à l' emploi.
 5. **Lorsque le conteneur est arrêté, invoquer les méthodes de rappel pour la destruction.**

Spring Core

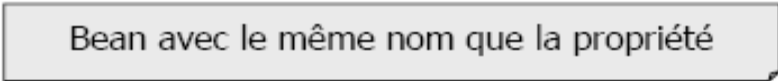
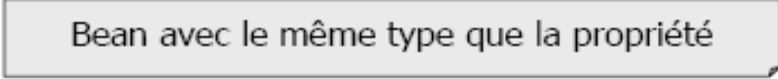
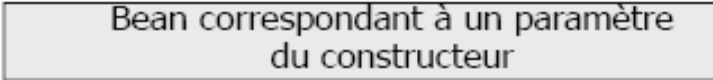
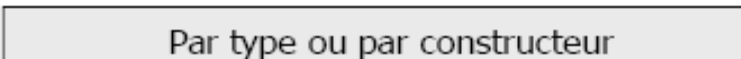
- Il existe trois façons de faire en sorte que Spring identifie les méthodes de rappel pour l'initialisation et la destruction.
- Par Interface : **InitializingBean** et **DisposableBean** du cycle de vie ; les méthodes **after-PropertiesSet()** et **destroy()** de ces interfaces correspondent à l'initialisation et à la destruction.
- Deuxièmement dans le fichier de configuration , définir les attributs **init-method** et **destroy-method** dans la déclaration du bean pour préciser les noms des méthodes de rappel.
- Troisièmement, marquer les méthodes de rappel avec les annotations du cycle de vie **@PostConstruct** et **@PreDestroy**, il suffit d'enregistrer une instance de **CommonAnnotationBeanPostProcessor** dans le conteneur IoC pour invoquer ces méthodes de rappel.

Travaux pratiques

- Exercice: 4_SpringCore_5

Spring Core

Injection automatique : « autowiring »

- La déclaration des dépendances nécessite plusieurs lignes de configuration (parfois lourd)
- Spring propose un **mécanisme d'injection automatique** ("autowiring") qui permet de "cabler" les composants sans déclarer explicitement les dépendances.
- Il suffit d'utiliser le paramètre "autowire" en indiquant l'algorithme de décision à utiliser
 - `autowire="byName"` 
 - `autowire="byType"` 
 - `autowire="constructor"` 
 - `autowire="autodetect"` 

Spring Core

- Bien que la fonctionnalité de liaison automatique soit très puissante, elle a pour inconvénient de diminuer la lisibilité de la configuration des beans.
- **En pratique**, il est conseillé de mettre en place la liaison automatique uniquement dans les applications dont les dépendances entre composants restent simples.

Spring Core

Liaison automatique par type

- Spring tente de lier un bean dont le type est compatible.
- Dans ce cas, le bean sera lié automatiquement.
- Problème majeur : plusieurs beans du conteneur IoC peuvent être compatibles avec le type cible.
- Dans ce cas, Spring n'est pas en mesure de choisir le bean adapté à la propriété et ne peut donc pas effectuer de liaison automatique.
- Spring lance une exception **UnsatisfiedDependencyException** lorsque plusieurs beans peuvent servir à la liaison automatique.

Spring Core

Liaison automatique par type: Exemple

```
<bean id="sequenceGenerator" class="com.formation.sequence.SequenceGenerator"
    autowire="byType">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
<bean id="datePrefixGenerator"
    class="com.formation.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
</bean>
```

Spring Core

Liaison automatique par nom

- Il fonctionne de manière comparable à la liaison par type, mais, dans ce cas, Spring tente de lier un bean de même nom à la place d'un bean de type compatible.
- Puisque les noms des beans sont uniques au sein d'un conteneur, la liaison automatique par nom n'est pas ambiguë.

```
<bean id="sequenceGenerator" class="com.formation.sequence.SequenceGenerator"
      autowire="byName">
  <property name="initial" value="100000" />
  <property name="suffix" value="A" />
</bean>
<bean id="prefixGenerator" class="com.formation.sequence.DatePrefixGenerator">
  <property name="pattern" value="yyyyMMdd" />
</bean>
```

Spring Core

Liaison automatique par constructeur

- La liaison automatique par constructeur fonctionne de manière semblable au mode byType.

Spring Core

Liaison automatique par « autowire »

- Le mode autodetect demande à Spring de choisir lui-même entre les modes de liaison automatique byType et constructor.
- Nous pouvons lier automatiquement une propriété en plaçant une annotation `@Autowired` sur un mutateur, un constructeur, un champ.
- Deux solutions :
 - Pour demander à Spring de lier automatiquement les propriétés de bean marquées par **@Autowired**, nous devons enregistrer une instance de **AutowiredAnnotationBeanPostProcessor** dans le conteneur IoC.

```
<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor" />
```

Spring Core

- Ou bien inclure l'élément **<context:annotation-config>** dans le fichier de configuration des beans.

Exemple:

```
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:annotation-config />
...
</beans>
```

Travaux pratiques

- Exercice: 4_SpringCore_5

AOP avec Spring

- Dans le monde de la programmation orientée objet, les applications sont organisées en classes et interfaces.
- Ces **concepts** sont **bien adaptés** à l'implémentation des **exigences métier** centrales, **non** à celle des **préoccupations transversales**.
- Les préoccupations transversales sont très fréquentes dans les applications d'entreprise, notamment :
 - ☐ la journalisation,
 - ☐ la validation
 - ☐ la gestion des transactions.
 - ☐ ...

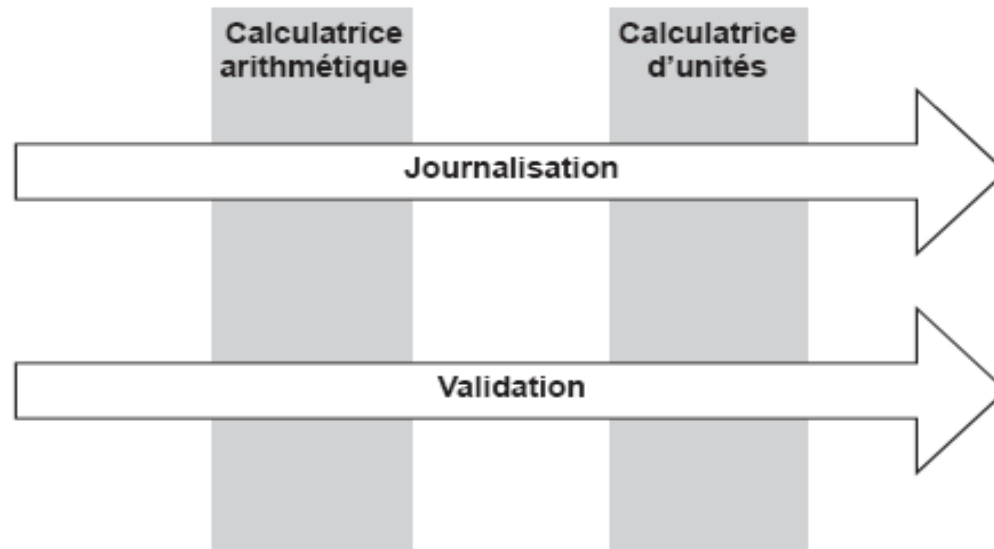
AOP avec Spring

- Il existe aujourd'hui sur le marché de nombreux frameworks POA :
 - ❑ AspectJ
 - ❑ JBoss AOP
 - ❑ Spring AOP
- **Spring AOP** s'occupe des préoccupations transversales **uniquement** pour les **beans déclarés** dans son **conteneur IoC**.

AOP avec Spring

Préoccupations transversales

- Par définition, une préoccupation transversale est une fonctionnalité qui concerne plusieurs modules d'une application.



AOP avec Spring

Problèmes associés aux préoccupations transversales

- Le premier se nomme *mélange de code*.
- Le second se nomme *dispersion de code*

Modulariser les préoccupations transversales avec des greffons Spring

- L'opération transversale réalisée à un point précis de l'exécution est encapsulée dans un *greffon (advice)*.

AOP avec Spring

- Spring AOP classique prend en charge **quatre types de greffons**, chacun intervenant à différents points de l'exécution.
 - **Greffon Before.** Intervient avant une exécution de méthode.
 - **Greffon After returning.** Intervient après que la méthode a retourné un résultat.
 - **Greffon After throwing.** Intervient après que la méthode a lancé une exception.
 - **Greffon Around.** Intervient autour d'une exécution de méthode.

AOP avec Spring

Greffon Before

- Un greffon *Before* intervient avant l'exécution d'une méthode. Il est créé en implémentant l'interface **MethodBeforeAdvice**.
- Depuis la méthode `before()`, nous avons accès aux détails de la méthode ainsi qu'à ses arguments.

Exemple:

```
package com.formation.calculator;  
.....  
  
public class LoggingBeforeAdvice implements MethodBeforeAdvice {  
    ...  
  
    public void before(Method method, Object[] args, Object target) throws Throwable {  
        log.info("Méthode " + method.getName() + "() invoquée avec " + Arrays.toString(args));  
    }  
}
```

AOP avec Spring

- Le greffon étant prêt, l'étape suivante consiste à l'appliquer à nos beans.

```
<bean id="loggingBeforeAdvice" class="com.formation.calculator.LoggingBeforeAdvice" />
```

- Puis l'étape la plus importante est de créer un proxy pour chacun de nos beans de calculatrice afin d'appliquer le greffon.
- Dans Spring AOP, la création d'un proxy se fait *via* un bean de fabrique nommé **ProxyFactoryBean**.

```
<bean id="arithmeticCalculatorProxy"
class="org.springframework.aop.framework.ProxyFactoryBean">

    <property name="proxyInterfaces">
        <list>
            <value> com.formation.calculator.ArithmeticCalculator </value>
        </list>
    </property>
    <property name="target" ref="arithmeticCalculator" /> ← Bean Cible
    <property name="interceptorNames">
        <list>
            <value>loggingBeforeAdvice</value>
        </list>
    </property>
</bean>
```

AOP avec Spring

- Dans le code :

```
ArithmeticCalculator arithmeticCalculator =  
(ArithmeticCalculator)context.getBean("arithmeticCalculatorProxy");
```


AOP avec Spring

Greffon After returning

- Nous pouvons également écrire un greffon ***After returning*** pour journaliser la fin d'une méthode et sa valeur de retour.

Exemple:

```
public void afterReturning(Object returnValue, Method method,  
    Object[] args, Object target) throws Throwable {  
    log.info("Méthode " + method.getName() + "() terminée par "  
        + returnValue);  
}
```

AOP avec Spring

Greffon After throwing

- Pour ce greffon, nous devons implémenter l'interface **ThrowsAdvice**, qui ne déclare aucune méthode.
- Ainsi, nous pouvons traiter différents types d'exceptions dans différentes méthodes.
- Cependant, le nom de chaque méthode doit être **afterThrowing** pour traiter un type particulier d'exception, qui est indiqué par le type de l'argument de la méthode.

AOP avec Spring

Greffon After throwing

Exemple:

- Par exemple, pour traiter une exception `IllegalArgumentException`, nous écrivons la méthode suivante.

```
****  
public class LoggingThrowsAdvice implements ThrowsAdvice {  
    ****  
    public void afterThrowing(IllegalArgumentException e) throws Throwable {  
        log.error("Argument invalide");  
    }  
}
```

- À l'exécution, toute exception compatible avec le type indiqué (c'est-à-dire le type et ses sous-types) est prise en charge par cette méthode.

EXERCICE_7 Suite

AOP avec Spring

Greffon Around

- Il est le plus puissant de tous.
- Il a un contrôle total sur l'exécution de la méthode et permet de combiner toutes les actions des greffons précédents en un seul.
- Un greffon **Around** doit implémenter l'interface **Method-Interceptor**.
- Lorsqu'on écrit un greffon Around, il ne faut surtout pas oublier d'invoquer **methodInvocation.proceed()** pour exécuter la méthode d'origine.

AOP avec Spring

Greffon Around

- Exemple:

```
package com.formation.calculator;

....

public class LoggingAroundAdvice implements MethodInterceptor {
    private Logger log = Logger.getLogger(this.getClass());

    public Object invoke(MethodInvocation methodInvocation) throws Throwable {
        log.info("Méthode " + methodInvocation.getMethod().getName()
            + "() invoquée avec "
            + Arrays.toString(methodInvocation.getArguments()));

        try{
            Object result = methodInvocation.proceed();
            log.info("Méthode " + methodInvocation.getMethod().getName()
                + "() terminée par " + result);

            return result;
        } catch (IllegalArgumentException e) {
            log.error("Argument invalide " + Arrays.toString(methodInvocation.getArguments())
                + " pour la méthode "
                + methodInvocation.getMethod().getName() + "()");

            throw e;
        }
    }
}
```

AOP avec Spring

- Lorsque nous précisons un greffon pour un **proxy AOP**, toutes les méthodes déclarées dans la classe cible ou les interfaces du proxy sont interceptées.
- Dans la plupart des cas, nous souhaitons uniquement intercepter certaines d'entre elles.
- Un ***point d'action (pointcut)*** permet de désigner certains points d'exécution du programme **pour appliquer un greffon**.
- Dans Spring AOP, les points d'action sont déclarés comme des **beans Spring** en utilisant **des classes de point d'action**.

AOP avec Spring

- Spring propose plusieurs classes de point d'action pour désigner les points d'exécution d'un programme.
 - *Point d'action avec nom de méthode*
 - *Point d'action avec expression régulière*
 - *Point d'action avec expression AspectJ*

AOP avec Spring

L'AOP est un des mécanismes importants utilisés par Spring : il l'utilise lui-même pour mettre en oeuvre certaines fonctionnalités notamment les transactions, l'annotation `@Configurable`, ROO, ...

Ainsi, l'AOP peut être utilisée :

- Indirectement, lors de l'utilisation de ces fonctionnalités

- Directement, pour mettre en oeuvre ses propres Aspects : Spring facilite alors cette mise en oeuvre

Spring met en oeuvre l'AOP de deux façons :

- Spring AOP : solution de Spring reposant sur des proxys créés dynamiquement

- AspectJ : solution open source du projet Eclipse qui permet un tissage des aspects au runtime ou à la compilation par enrichissement du bytecode.

Depuis la version 2.0, Spring propose un support AspectJ pour la mise en oeuvre de l'AOP en complément de sa propre solution reposant sur les proxys.

AOP avec Spring

L'AOP peut être mise en oeuvre via Spring AOP ou AspectJ de plusieurs manières :

- Avec AspectJ avec un tissage au chargement (Load Time Weaving) ou à la compilation

- Avec ou sans les annotations AspectJ avec Spring AOP

- Avec un mixte de Spring AOP et AspectJ

AOP avec Spring

Point d'action avec nom de méthode

- Pour intercepter une seule méthode, nous pouvons employer **NameMatchMethodPointcut** de manière à définir statiquement la méthode par son nom.
- Dans la propriété **mappedName**, nous donnons:
 - un nom de méthode précis
 - Ou une expression avec des caractères génériques.

```
<bean id="methodNamePointcut"  
class="org.springframework.aop.support.NameMatchMethodPointcut">  
  
    <property name="mappedName" value="add" />  
  
</bean>
```

AOP avec Spring

Point d'action avec nom de méthode

- Un point d'action doit être associé à un greffon pour indiquer où celui-ci doit s'appliquer.
- Cette association se nomme **conseiller (advisor)** dans SpringAOP.
- La classe **DefaultPointcutAdvisor** sert à associer un point d'action et un greffon.
- Pour appliquer un conseiller à un proxy, nous procédons comme pour un greffon.

AOP avec Spring

Point d'action avec nom de méthode

```
<beans ...>
...
<bean id="methodNameAdvisor"
      class="org.springframework.aop.support.DefaultPointcutAdvisor">
  <property name="pointcut" ref="methodNamePointcut" />
  <property name="advice" ref="loggingAroundAdvice" />
</bean>

<bean id="arithmeticCalculatorProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="arithmeticCalculator" />
  <property name="interceptorNames">
    <list>
      <value>methodNameAdvisor</value>
    </list>
  </property>
</bean>
</beans>
```

AOP avec Spring

Point d'action avec nom de méthode

- Si nous souhaitons désigner plusieurs méthodes en utilisant un point d'action avec nom de méthode, nous devons les indiquer dans la propriété **mappedNames** de type **java.util.List**.

```
<bean id="methodNamePointcut"  
class="org.springframework.aop.support.NameMatchMethodPointcut">  
  <property name="mappedNames">  
    <list>  
      <value>add</value>  
      <value>sub</value>  
    </list>  
  </property>  
</bean>
```

AOP avec Spring

Point d'action avec expression régulière

- La classe **RegexMethodPointcutAdvisor** nous permet d'indiquer une ou plusieurs expressions régulières.
- Par exemple, l'expression suivante correspond aux méthodes dont le nom contient le terme **mul** ou **div** :

AOP avec Spring

```
<beans ...>
...
<bean id="regexAdvisor"
      class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
  <property name="patterns">
    <list>
      <value>.*mul.*</value>
      <value>.*div.*</value>
    </list>
  </property>
  <property name="advice" ref="loggingAroundAdvice" />
</bean>
|
<bean id="arithmeticCalculatorProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
  <property name="target" ref="arithmeticCalculator" />
  <property name="interceptorNames">
    <list>
      <value>methodNameAdvisor</value>
      <value>regexAdvisor</value>
    </list>
  </property>
</bean>
</beans>
```

Spring ASPECT-J

- Ce chapitre explique comment bénéficier du framework **AspectJ** dans les applications Spring.
- AspectJ est un **framework AOP** complet et répandu.

Activer la prise en charge des annotations AspectJ dans Spring

Pour activer la prise en charge des annotations AspectJ dans le conteneur Spring IoC, il suffit d'ajouter un élément

```
<bean class="org.springframework.aop.aspectj.annotation.AnnotationAwareAspectJAutoProxyCreator"/>
```

dans le fichier de configuration des beans.

Ensuite, Spring crée automatiquement des proxies pour les beans qui correspondent aux aspects AspectJ.

Spring ASPECT-J

Déclarer des aspects avec des annotations AspectJ

- AspectJ accepte que les aspects soient écrits sous forme de POJO marqués avec un ensemble **d'annotations AspectJ**.
- Avec les annotations AspectJ, un aspect n'est rien d'autre qu'une classe Java marquée par **@Aspect**.
- Un greffon est une simple méthode Java marquée par l'une des annotations de greffons :
 - ❑ **@Before,**
 - ❑ **@After,**
 - ❑ **@AfterReturning,**
 - ❑ **@AfterThrowing**
 - ❑ **@Around.**

Spring ASPECT-J

- ***Greffon Before***
- *Traite les préoccupations transversales avant certains points d' exécution d' un programme.*
- Utilisez l' annotation @Before et inclure l' expression de point d' action dans la valeur de l' annotation.

EXEMPLE

Spring ASPECT-J

Exécution de la méthode **add()** de
l'interface **ArithmeticCalculator**

un nombre quelconque
d'arguments

```
....  
  
@Aspect  
public class CalculatorLoggingAspect {  
    private Log log = LogFactory.getLog(this.getClass());  
  
    @Before("execution(*com.formation.calculator.ArithmeticCalculator.add(..))")  
    public void logBefore() {  
        log.info("Début de la méthode add()");  
    }  
}
```

n'importe quel modificateur
(public, protected et private)

EXERCICE SpringAspectJ_1 Action1

Spring ASPECT-J

- Pour que notre greffon accède aux détails du point de jonction courant,
 - comme le nom de la méthode
 - la valeur des arguments,
 - ...

nous déclarons un argument de type **JoinPoint** dans la méthode de greffon.

```
public void logBefore(JoinPoint joinPoint) {  
    log.info("Méthode " + joinPoint.getSignature().getName()  
        + "() invoquée avec " + Arrays.toString(joinPoint.getArgs()));  
}
```

Nous pouvons écrire un point d'action qui désigne toutes les méthodes, simplement en remplaçant les noms de classe et de méthode par des caractères génériques.

```
@Before("execution(* *.*(..))")
```

EXERCICE SpringAspectJ_1 Action2

Spring ASPECT-J

Greffon After

- Un greffon **After** *est exécuté après la fin d'un point de jonction, dès que celui-ci retourne un résultat ou lance une exception.*
- Le greffon **After** suivant consigne la fin d'une méthode de la calculatrice.
- Un aspect peut inclure **un ou plusieurs greffons**.

Exemple:

```
@After("execution(* *.*(..))")
public void logAfter(JoinPoint joinPoint) {
    log.info("Méthode " + joinPoint.getSignature().getName()
            + "() terminée");
}
```

Spring ASPECT-J

Greffon After returning

- Un greffon **After** est exécuté que le point de jonction retourne de manière normale ou lance une exception.
- Pour que la journalisation se fasse **uniquement lors d'un retour normal**, nous remplaçons le greffon **After** par un greffon *After returning*.

Exemple:

```
@AfterReturning("execution(* *.*(..))")
public void logAfter(JoinPoint joinPoint) {
    log.info("Méthode " + joinPoint.getSignature().getName()
            + "() terminée");
}
```

Spring ASPECT-J

Greffon After returning

- Nous pouvons accéder à la valeur de retour d' un point de jonction en ajoutant un attribut **returning** à l' annotation **@AfterReturning**.
- La valeur de cet attribut = le nom de l' argument de la méthode de greffon (valeur de retour placée par Spring)
- Ajouter un argument de ce nom à la méthode de greffon.
- L' expression du point d' action doit alors être donnée dans l' attribut pointcut.

Spring ASPECT-J

Exemple:

```
@AfterReturning(pointcut = "execution(* *.*(..))", returning = "result")
public void logAfterReturning(JoinPoint joinPoint, Object result) {
    log.info("Méthode " + joinPoint.getSignature().getName()
            + "() terminée par " + result);
}
```


Spring ASPECT-J

Greffon After throwing

- Un greffon ***After throwing*** est exécuté uniquement lorsqu'une exception est lancée par un point de jonction.
- Exemple:

```
@AfterThrowing("execution(* *.*(..))")  
public void logAfterThrowing(JoinPoint joinPoint) {  
    log.error("Exception lancée dans " + joinPoint.getSignature().getName() + "()");  
}
```

Spring ASPECT-J

Greffon After throwing

- Il est possible d'accéder à l'exception lancée par le point de jonction en ajoutant un attribut throwing à l'annotation `@AfterThrowing`.
- Dans le langage Java, **Throwable** est la superclasse de toutes les erreurs et exceptions.
- Par conséquent, le greffon suivant intercepte toutes les erreurs et exceptions générées par les points de jonction :

Spring ASPECT-J

```
@AfterThrowing(pointcut = "execution(* *.*(..))", throwing = "e")
public void logAfterThrowing(JoinPoint joinPoint, Throwable e) {
    log.error("Exception " + e + " lancée dans "
        + joinPoint.getSignature().getName() + "()");
}
```

Spring ASPECT-J

Greffon Around

- Il a un contrôle total sur l'exécution de la méthode et permet de combiner toutes les actions des greffons précédents en un seul.
- Pour ce type de greffon, l'argument du point de jonction doit être de type **ProceedingJoinPoint**.
- Il s'agit d'une sous-interface de **JoinPoint** qui nous permet de contrôler la poursuite de l'exécution dans le point de jonction d'origine.

Spring ASPECT-J

Greffon Around

```
@Around("execution(* *.*(..))")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    log.info("Méthode " + joinPoint.getSignature().getName()
        + "() invoquée avec " + Arrays.toString(joinPoint.getArgs()));
    try {
        Object result = joinPoint.proceed();
        log.info("Méthode " + joinPoint.getSignature().getName()
            + "() terminée par " + result);
        return result;
    } catch (IllegalArgumentException e) {
        log.error("Argument invalide "
            + Arrays.toString(joinPoint.getArgs()) + " dans "
            + joinPoint.getSignature().getName() + "()");
        throw e;
    }
}
```

Spring ASPECT-J

Préciser la précedence des aspects

- ***Problème***

Lorsque plusieurs aspects sont appliqués au même point de jonction, leur précedence est indéfinie, sauf si elle a été fixée de manière explicite.

- ***Solution***

La précedence des aspects peut être définie, soit en implémentant l'interface **Ordered**, soit en utilisant l'annotation **@Order**.

Spring ASPECT-J

Exemple:

- Soient deux aspect définis dans les fichiers de configuration de beans Spring.
- Les deux aspects doivent implémenter l'interface.

Ordered

- **Plus** la valeur retournée par la méthode **getOrder()** est **faible**, plus la priorité est élevée.

Exemple:

```
@Aspect
public class CalculatorValidationAspect implements Ordered {
    ...
    public int getOrder() {
        return 0;
    }
}
```

Spring ASPECT-J

- Pour préciser la précédence, nous pouvons également utiliser l'annotation `@Order`.
- Le numéro d'ordre doit être indiqué dans la valeur de l'annotation.

Exemple:

```
package com.formation.calculator;  
  
...  
import org.springframework.core.annotation.Order;  
  
@Aspect  
@Order(0)  
public class CalculatorValidationAspect {  
    ....  
}
```


SPRING ORM

- Spring propose un support pour de nombreuses technologies d'accès aux données.
- Malgré la diversité de ces technologies, ce support reste homogène dans son utilisation par le développeur.
- Le support d'accès aux données de Spring s'intègre parfaitement avec le motif de conception DAO.
- L'idée du support d'accès aux données est d'affranchir le développeur des tâches répétitives et propices à des erreurs :
 - ❑ ouverture/fermeture des connexions,
 - ❑ manipulation des API spécifiques.

SPRING ORM

- Les classes du support implémentant le comportement décrit ci-dessus s'appellent des *templates*.
- Il existe des classes de templates pour les différentes technologies supportées par Spring.
 - JDBC
 - Hibernate,
 - JDO,
 - TopLink,
 - iBATIS ,
 - JPA.

SPRING ORM

Utiliser un template JDBC pour la mise à jour

- La première interface de rappel examinée se nomme **PreparedStatementCreator**.
- Lorsqu' on implémente l' interface **PreparedStatementCreator**, la connexion à la base de données est passée en argument de la méthode **createPreparedStatement()**.
- Dans cette méthode, il suffit ensuite de créer un objet **PreparedStatement** sur la connexion fournie et de lier nos paramètres à cet objet.
- Pour finir, nous devons retourner l' objet **PreparedStatement** dans la valeur de retour de la méthode.

SPRING ORM

Exemple:

```
public class InsertVehicleStatementCreator implements PreparedStatementCreator {
    private Vehicle vehicle;

    public InsertVehicleStatementCreator(Vehicle vehicle) {
        this.vehicle = vehicle;
    }

    public PreparedStatement createPreparedStatement(Connection conn)
        throws SQLException {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, vehicle.getVehicleNo());
        ps.setString(2, vehicle.getColor());
        ps.setInt(3, vehicle.getWheel());
        ps.setInt(4, vehicle.getSeat());
        return ps;
    }
}
```

Grâce à ce créateur de requête, nous pouvons simplifier l'opération d'ajout d'un véhicule.

SPRING ORM

- Tout d'abord, nous créons une instance de la classe JdbcTemplate et lui passons la source de données de manière à obtenir une connexion.
- Ensuite, nous invoquons la méthode update() en lui fournissant notre créateur de requête.
- EXEMPLE:

```
public void insert(Vehicle vehicle) {  
    JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);  
    jdbcTemplate.update(new InsertVehicleStatementCreator(vehicle));  
}
```

EXERCICE_10

SPRING ORM

- **Utiliser un template JDBC pour interroger une base de données**
- **RowCallbackHandler** est une interface permettant de traiter la ligne courante de l'ensemble de résultat.
- La méthode **query()** parcourt le contenu de l'ensemble de résultat à notre place et invoque **RowCallbackHandler** pour chaque ligne.
- La méthode **processRow()** est ainsi invoquée une fois pour chaque ligne de l'ensemble de résultat.

SPRING ORM

EXAMPLE:

```
public Vehicle findByVehicleNo(String vehicleNo) {  
    String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";  
    final Vehicle vehicle = new Vehicle();  
    jdbcTemplate.query(sql, new Object[] { vehicleNo },  
        new RowCallbackHandler() {  
            public void processRow(ResultSet rs) throws SQLException {  
                vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));  
                vehicle.setColor(rs.getString("COLOR"));  
                vehicle.setWheel(rs.getInt("WHEEL"));  
                vehicle.setSeat(rs.getInt("SEAT"));  
            }  
        });  
    return vehicle;  
}
```

SPRING ORM

- Pour faciliter la réutilisation, il est préférable d'implémenter l'interface **RowMapper** sous forme d'une classe normale plutôt que sous forme d'une classe interne.
- Dans la méthode **mapRow()** de cette interface, nous construisons l'objet qui représente une ligne et l'utilisons comme valeur de retour.

SPRING ORM

EXEMPLE:

```
public class VehicleRowMapper implements RowMapper<Vehicle> {  
    public Vehicle mapRow(ResultSet rs, int rowNum) throws SQLException {  
        Vehicle vehicle = new Vehicle();  
        vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));  
        vehicle.setColor(rs.getString("COLOR"));  
        vehicle.setWheel(rs.getInt("WHEEL"));  
        vehicle.setSeat(rs.getInt("SEAT"));  
        return vehicle;  
    }  
}
```

Dans le code

```
public Vehicle findByVehicleNo(String vehicleNo) {  
    String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";  
  
    Vehicle vehicle = (Vehicle) jdbcTemplate.queryForObject(sql,  
        new Object[] { vehicleNo }, new VehicleRowMapper());  
    return vehicle;  
}
```

SPRING ORM

- Spring fournit une implémentation pratique de **RowMapper**, **BeanPropertyRow- Mapper**, qui peut associer automatiquement une ligne à une nouvelle instance de la classe indiquée.
- Elle commence par instancier cette classe, puis elle associe chaque valeur de colonne à une propriété en mettant en correspondance leur nom.

EXEMPLE

```
@Override
public List<Vehicle> findAll() {
    String sql = "SELECT * FROM VEHICLE";

    List<Vehicle> vehicles = jdbcTemplate.query(sql,
        BeanPropertyRowMapper.newInstance(Vehicle.class));
    return vehicles;
}
```

EXERCICE_11

SPRING ORM

- **Configurer des fabriques de ressources ORM dans Spring**
- Lorsqu' on utilise un framework ORM seul, nous devons configurer sa fabrique de ressources au travers de son API.
- Spring fournit plusieurs beans de fabrique pour que nous puissions créer une fabrique de sessions Hibernate ou une fabrique de gestionnaires d' entités JPA sous forme d' un bean unique dans le conteneur IoC.
- Ces fabriques peuvent être partagées par plusieurs beans grâce à l' injection de dépendance.

SPRING ORM

Configurer une fabrique de sessions Hibernate dans Spring

Exemple:

```
public class HibernateCourseDao implements CourseDao {  
  
    private SessionFactory sessionFactory;  
  
    public void setSessionFactory(SessionFactory sessionFactory) {  
        this.sessionFactory = sessionFactory;  
    }  
}
```

La classe HibernateCourseDao accepte une fabrique de sessions par injection de dépendance

SPRING ORM

- Pour la correspondance des classes d'entité avec le format XML d'Hibernate, on crée un fichier **.hbm.xml** par classe.
- Nous le plaçons dans le même paquetage que la classe d'entité.

EX:

```
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.lab3.course">
  <class name="Course" table="COURSE">
    <id name="id" type="long" column="ID">
      <generator class="identity" />
    </id>
    <property name="title" type="string">
      <column name="TITLE" length="100" not-null="true" />
    </property>
    <property name="beginDate" type="date" column="BEGIN_DATE" />
    <property name="endDate" type="date" column="END_DATE" />
    <property name="fee" type="int" column="FEE" />
  </class>
</hibernate-mapping>
```

SPRING ORM

- Chaque application qui utilise Hibernate a besoin d' un fichier global pour configurer certaines propriétés, comme les paramètres de la base de données, le dialecte de la base de données,...
- Par défaut, Hibernate examine le **fichier hibernate.cfg.xml** à la racine du chemin d' accès aux classes.

EXEMPLE:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
<hibernate-configuration>
    <session-factory>
        <property name="connection.driver_class">
            com.mysql.jdbc.Driver
        </property>
        <property name="connection.url">
            jdbc:mysql://127.0.0.1:3306/spring
        </property>
        <property name="connection.username">root</property>
        <property name="connection.password"></property>
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
        <property name="show_sql">true</property>
        <property name="hbm2ddl.auto">update</property>
        <mapping resource="com/lab3/course/Course.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

SPRING ORM

- Nous créons ensuite un fichier de configuration des beans pour utiliser Hibernate comme framework ORM (Ex: beans4.xml).
- Nous déclarons une fabrique de sessions qui utilise le fichier hibernate.cfg.xml dans le bean de fabrique **LocalSessionFactoryBean**.
- La propriété **configLocation** nous permet d'indiquer au bean de fabrique qu'il doit charger le fichier de configuration d'Hibernate.

SPRING ORM

Exemple:

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
        <property name="configLocation" value="classpath:hibernate.cfg.xml" />
    </bean>
    <bean id="courseDao"
        class="com.lab3.course.hibernate.HibernateCourseDao">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>
```

EXERCICE_12 Action1

SPRING ORM

- Nous pouvons même ignorer le fichier de configuration d' Hibernate en regroupant tous les éléments de configuration dans **LocalSessionFactoryBean**.
- Exemple:

SPRING ORM

```
<beans ... >

  <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url"
        value="jdbc:mysql://127.0.0.1:3306/spring" />
    <property name="username" value="root" />
    <property name="password" value="" />
  </bean>

  <bean id="sessionFactory"
        class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
      <list>
        <value>com/lab3/course/Course.hbm.xml</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
        <prop key="hibernate.show_sql">true</prop>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
      </props>
    </property>
  </bean>
  <bean id="courseDao"
        class="com.lab3.course.hibernate.HibernateCourseDao">
    <property name="sessionFactory" ref="sessionFactory" />
  </bean>
</beans>
```

SPRING ORM

- *Configurer une fabrique de gestionnaires d'entités JPA dans Spring*
- *On crée un DAO pour qu'elle accepte une fabrique de gestionnaires d'entités via l'injection de dépendance.*

EXEMPLE:

```
public class JpaCourseDao implements CourseDao {  
  
    private EntityManagerFactory entityManagerFactory;  
  
    public void setEntityManagerFactory(  
        EntityManagerFactory entityManagerFactory) {  
        this.entityManagerFactory = entityManagerFactory;  
    }  
}
```

SPRING ORM

- Au lieu de faire référence au fichier de configuration d' Hibernate, nous mettons tous les éléments de configuration dans **persistence.xml**.

Exemple:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="course" transaction-type="RESOURCE_LOCAL">
    <properties>
      <property name="javax.persistence.jdbc.url"
value="jdbc:mysql://127.0.0.1:3306/spring"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.show_sql" value="true" />
      <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect"/>
    </properties>
  </persistence-unit>
</persistence>
```

SPRING ORM

JPA

- Spring fournit un bean de fabrique, **LocalEntityManagerFactoryBean**, pour que nous puissions créer une fabrique de gestionnaires d'entités dans le conteneur IoC.
- La configuration de JPA se fait au travers du fichier XML central persistence.xml, qui se trouve dans le répertoire **META-INF** à la racine du chemin d'accès aux classes.
- Chaque fichier de configuration contient un ou plusieurs éléments **<persistence-unit>**.

SPRING ORM

- EXAMPLE:

```
<persistence version="2.0" xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.xml.org/2001/XMLSchema-instance" xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/persistence/persistence_2_0.xsd">
  <persistence-unit name="course" transaction-type="RESOURCE_LOCAL">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <properties>
      <property name="javax.persistence.jdbc.url" value="jdbc:mysql://127.0.0.1:3306/spring"/>
      <property name="javax.persistence.jdbc.password" value=""/>
      <property name="javax.persistence.jdbc.driver" value="com.mysql.jdbc.Driver"/>
      <property name="javax.persistence.jdbc.user" value="root"/>
      <property name="hibernate.hbm2ddl.auto" value="update" />
      <property name="hibernate.show_sql" value="false" />
    </properties>
  </persistence-unit>
</persistence>
```

SPRING ORM

- Le module **Hibernate EntityManager** détecte automatiquement que les annotations JPA sont des métadonnées de correspondance, il est inutile de les préciser explicitement.
- Reste à initialiser une fabrique de gestionnaires d'entités:

Exemple:

```
public class JpaCourseDao implements CourseDao {  
  
    private EntityManagerFactory entityManagerFactory;  
  
    public void setEntityManagerFactory(  
        EntityManagerFactory entityManagerFactory) {  
        this.entityManagerFactory = entityManagerFactory;  
    }  
}
```

SPRING ORM

Dans le code:

```
public void store(Course course) {  
    EntityManager manager = entityManagerFactory.createEntityManager();  
    EntityTransaction tx = manager.getTransaction();  
    try{  
        tx.begin();  
        manager.merge(course);  
        tx.commit();  
    } catch (RuntimeException e) {  
        tx.rollback();  
        throw e;  
    } finally {  
        manager.close();  
    }  
}
```


SPRING TRANSACTION

- La gestion des transactions permet:
 - de garantir l'intégrité
 - et la cohérence des données.
- Spring, définit une couche abstraite au-dessus des différentes API de gestion des transactions.
- ***transactions par programmation***: code incorporé dans les méthodes métier de manière à contrôler la validation (***commit***) et l'annulation (***rollback***) des transactions.
- ***gestion des transactions par déclaration***: code de gestion des transactions est séparé des méthodes métier *via des déclarations*.

SPRING TRANSACTION

Le concept de transactions peut être décrit par les quatre propriétés ACID :

- **Atomicité.** Une transaction est une opération atomique constituée d'une suite d'opérations. L'atomicité d'une transaction garantit que toutes les actions sont entièrement exécutées ou qu'elles n'ont aucun effet.
- **Cohérence.** Dès lors que toutes les actions d'une transaction se sont exécutées, la transaction est validée. Les données et les ressources sont alors dans un état cohérent qui respecte les règles métier.
- **Isolation.** Puisque plusieurs transactions peuvent manipuler le même jeu de données au même moment, chaque transaction doit être isolée des autres afin d'éviter la corruption des données.
- **Durabilité.** Dès lors qu'une transaction est terminée, les résultats doivent survivre à toute panne du système. En général, les résultats d'une transaction sont écrits dans une zone de stockage persistant.

SPRING TRANSACTION

Tp: 1

SPRING TRANSACTION

Choisir une implémentation de gestionnaire de transactions

- L'abstraction au coeur de la gestion des transactions dans Spring se nomme **PlatformTransactionManager**.
- Cette interface encapsule un ensemble de méthodes indépendantes de la technologie pour la gestion des transactions.
- Spring propose plusieurs implémentations de cette interface qui sont utilisées avec différentes API de gestion des transactions :
 - **DataSourceTransactionManager(JDBC)**
 - **JtaTransactionManager(JTA)**
 - **HibernateTransactionManager**
 - **JpaTransactionManager**

SPRING TRANSACTION

- Pour déclarer un gestionnaire de transactions dans le conteneur Spring IoC, nous procédons comme pour un bean normal.

EXEMPLE:

```
<bean id="transactionManager"  
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">  
    <property name="dataSource" ref="dataSource" />  
</bean>
```

SPRING TRANSACTION

- Pour débiter une nouvelle transaction on invoque la méthode **getTransaction()** et on gère cette transaction par l'intermédiaire des méthodes **commit()** et **rollback()**.
- Puisqu'elle doit travailler avec un gestionnaire de transactions, nous ajoutons une propriété de type **PlatformTransactionManager**, dont l'injection peut se faire par un mutateur.

EXEMPLE:

```
public class TransactionalJdbcBookShop extends JdbcDaoSupport implements
    BookShop {
    private PlatformTransactionManager transactionManager;

    public void setTransactionManager(
        PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }
}
```

SPRING TRANSACTION

- Avant de débiter une nouvelle transaction, nous devons préciser les attributs transactionnels dans un objet de définition de transaction de type **TransactionDefinition**.
- Pour faire simple utiliser **DefaultTransactionDefinition** pour utiliser les attributs transactionnels par défaut.
- Après avoir défini une transaction, nous demandons au gestionnaire de transactions de démarrer une nouvelle transaction correspondant à cette définition en invoquant la méthode **getTransaction()**.
- Elle retourne un objet **TransactionStatus** qui nous permet de suivre l'état de la transaction.

SPRING TRANSACTION

EXEMPLE:

```
public void purchase(String isbn, String username) {  
    TransactionDefinition def = new DefaultTransactionDefinition();  
    TransactionStatus status = transactionManager.getTransaction(def);  
    try {  
        // votre code transactionnel  
  
        transactionManager.commit(status);  
    } catch (DataAccessException e) {  
        transactionManager.rollback(status);  
        throw e;  
    }  
}
```

EXERCICE_Transaction1

EXERCICE_Transaction2

SPRING TRANSACTION

Gérer les transactions par déclaration avec l'annotation *@Transactional*

- Spring nous permet de déclarer des transactions en marquant simplement les méthodes transactionnelles avec l'annotation **@Transactional** et en activant l'élément **<tx:annotation-driven>**.
- Pour indiquer le caractère transactionnel d'une méthode, nous la marquons simplement avec l'annotation @Transactional.
- **! Seules les méthodes publiques doivent être annotées.**

EXEMPLE:

```
@Transactional
public void purchase(String isbn, String username) {
    ...
}
```

SPRING TRANSACTION

- L'annotation **@Transactional** peut être appliquée au niveau de la méthode ou de la classe.
- Lorsqu'elle concerne une classe, toutes ses méthodes publiques sont considérées comme transactionnelles.
- Dans le fichier de configuration des beans, nous ajoutons simplement un élément **<tx:annotation-driven>** en lui indiquant **un gestionnaire de transactions**.

EXEMPLE:

```
<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost:3306/spring" />
  <property name="username" value="root" />
  <property name="password" value="" />
</bean>

<tx:annotation-driven transaction-manager="transactionManager" />

<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```

- **EXERCICE Spring_Transaction_4**

SPRING TRANSACTION

Fixer l'attribut transactionnel de propagation

- Lorsqu'une méthode transactionnelle est invoquée par une autre méthode, il est nécessaire de préciser la manière dont la transaction est propagée.
- Par exemple, la méthode peut continuer à s'exécuter au sein de la transaction existante ou elle peut démarrer une nouvelle transaction et s'exécuter à l'intérieur de celle-ci.
- L'attribut transactionnel de ***propagation nous permet de préciser la manière dont se fait*** la propagation des transactions.
- Spring définit sept comportements de propagation (voir Tableau).

SPRING TRANSACTION

<i>Propagation</i>	<i>Description</i>
REQUIRED	S'il existe une transaction en cours, la méthode actuelle doit s'exécuter au sein de cette transaction. Sinon elle doit démarrer une nouvelle transaction et s'exécuter dans celle-ci.
REQUIRES_NEW	La méthode actuelle doit démarrer une nouvelle transaction et s'exécuter dans celle-ci. S'il existe une transaction en cours, elle doit être suspendue.
SUPPORTS	S'il existe une transaction en cours, la méthode actuelle peut s'exécuter au sein de cette transaction. Sinon elle n'est pas obligée de s'exécuter dans une transaction.
NOT_SUPPORTED	La méthode actuelle ne doit pas s'exécuter au sein d'une transaction. S'il existe une transaction en cours, elle doit être suspendue.
MANDATORY	La méthode actuelle doit s'exécuter au sein d'une transaction. S'il n'existe aucune transaction en cours, une exception est lancée.
NEVER	La méthode actuelle ne doit pas s'exécuter au sein d'une transaction. S'il existe une transaction en cours, une exception est lancée.
NESTED	S'il existe une transaction en cours, la méthode actuelle doit s'exécuter au sein d'une transaction imbriquée (prise en charge par la fonctionnalité de point de sauvegarde de JDBC 3.0) dans cette transaction. Sinon elle doit démarrer une nouvelle transaction et s'exécuter dans celle-ci.

SPRING TRANSACTION

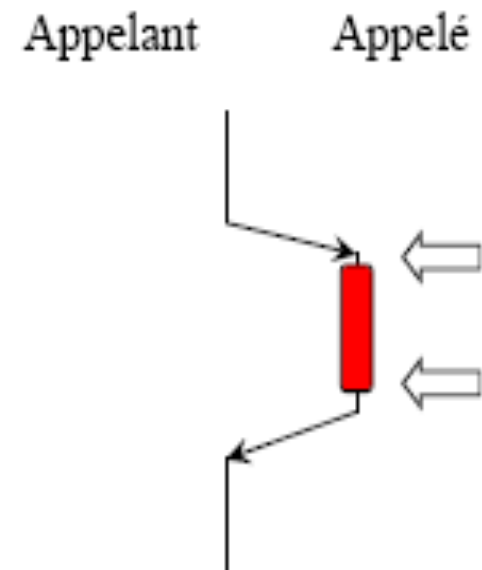
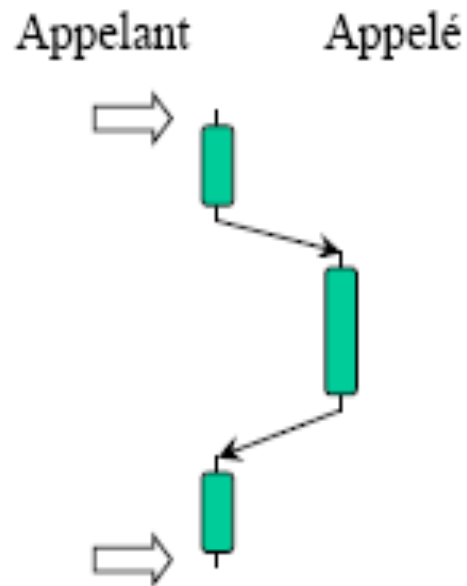
2 Cas pour le bean appelant

- Soit il s'exécute dans une transaction
- Soit il s'exécute en dehors de tout contexte transactionnel

SPRING TRANSACTION

Granularité des transactions

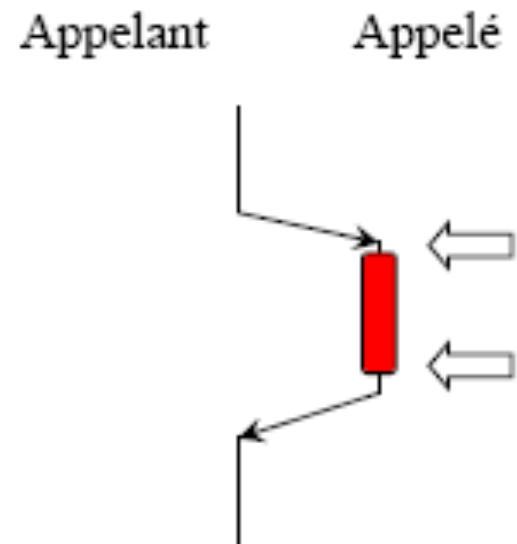
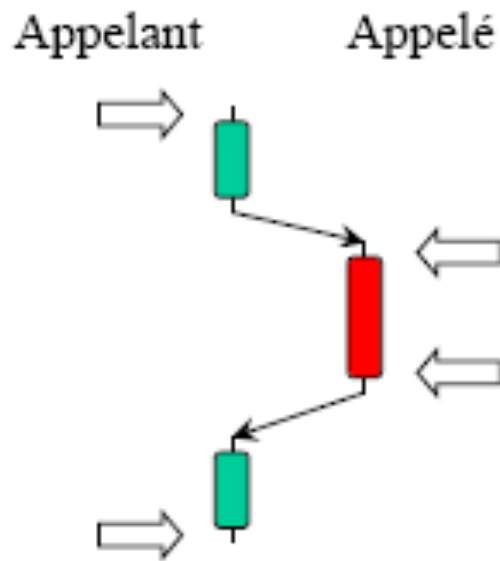
REQUIRED



SPRING TRANSACTION

Granularité des transactions

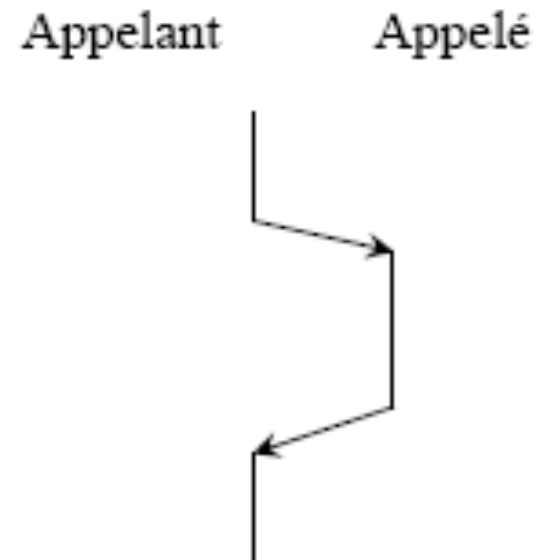
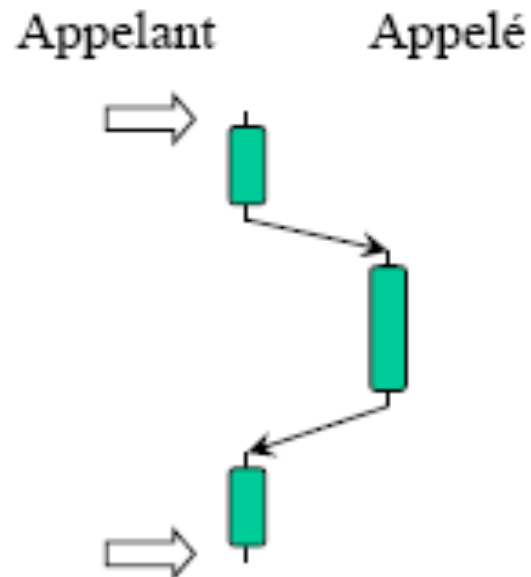
REQUIRES_NEW



SPRING TRANSACTION

Granularité des transactions

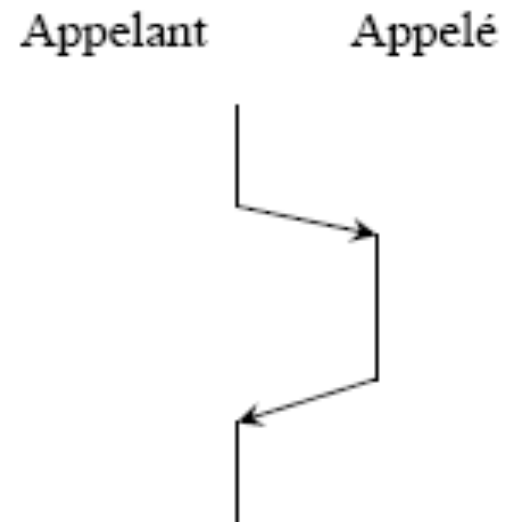
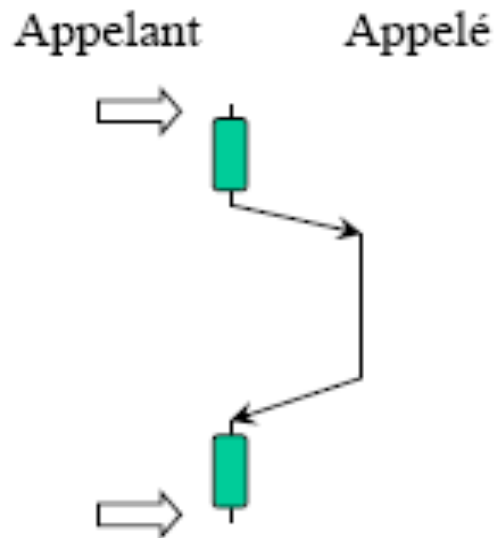
SUPPORTS



SPRING TRANSACTION

Granularité des transactions

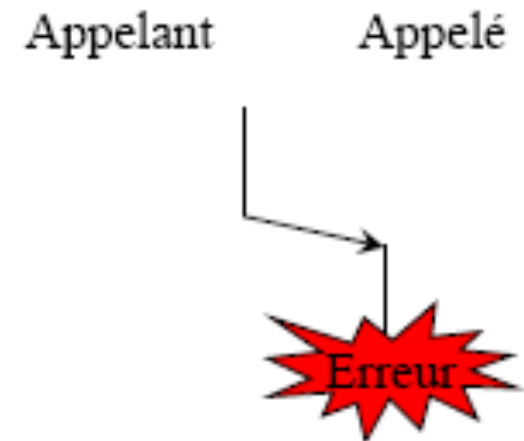
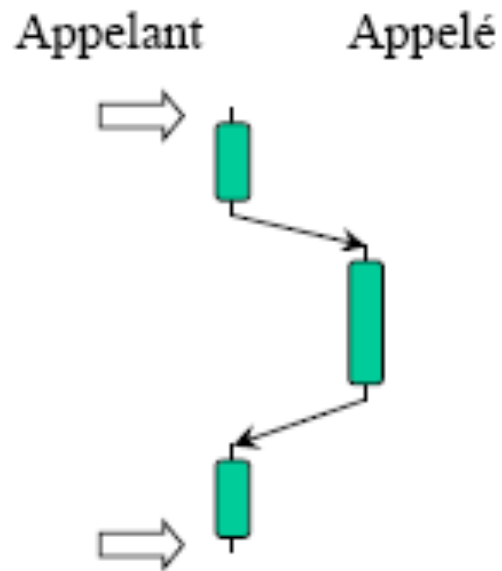
NOT_SUPPORTED



SPRING TRANSACTION

Granularité des transactions

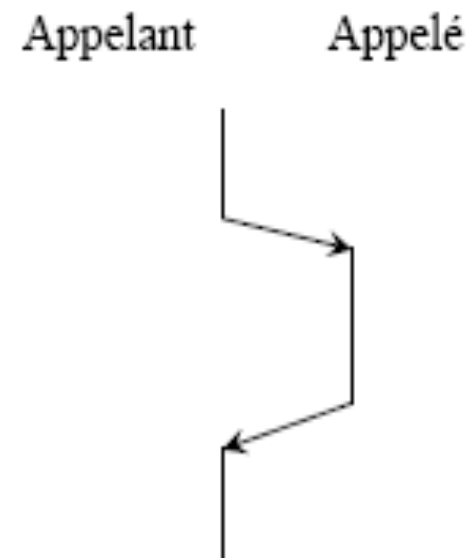
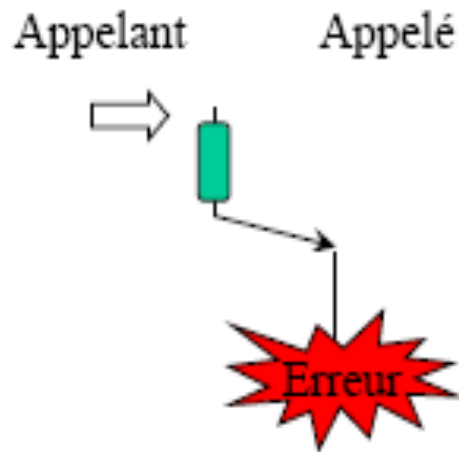
MANDATORY



SPRING TRANSACTION

Granularité des transactions

NEVER



SPRING TRANSACTION

Codage

```
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(propagation = Propagation.REQUIRED)
    public void purchase(String isbn, String username) {
        ...
    }
}

---
```

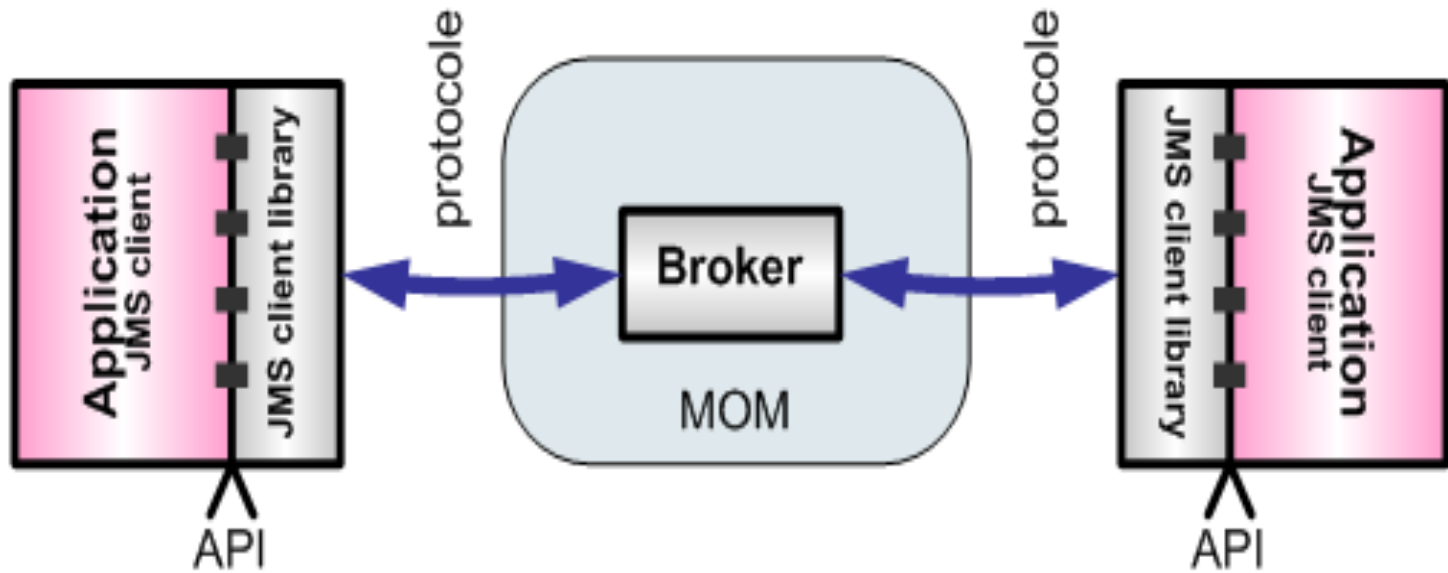
Spring JMS

Qu'est-ce que JMS ?

- **JMS** (Java Messaging System) est l'implémentation Java de ce qui est appelé **MOM**(*Message-Oriented Middleware*).
- Un MOM est une plateforme logicielle fournissant un moyen de communication entre divers applications, sans que celles-ci soient consciente de l'existence de l'autres.
- Le principe est qu'une application ne communique pas directement avec l'autre, mais dépose son "**message**" dans le MOM.
- L'autre application de son coté, viendra simplement vérifier l'arrivée de message sur le MOM.

Spring JMS

Exemple:



Spring JMS

Avantages de JMS

- Faible Couplage
- Asynchrone
- Persistant
- Transactionnel

Spring JMS

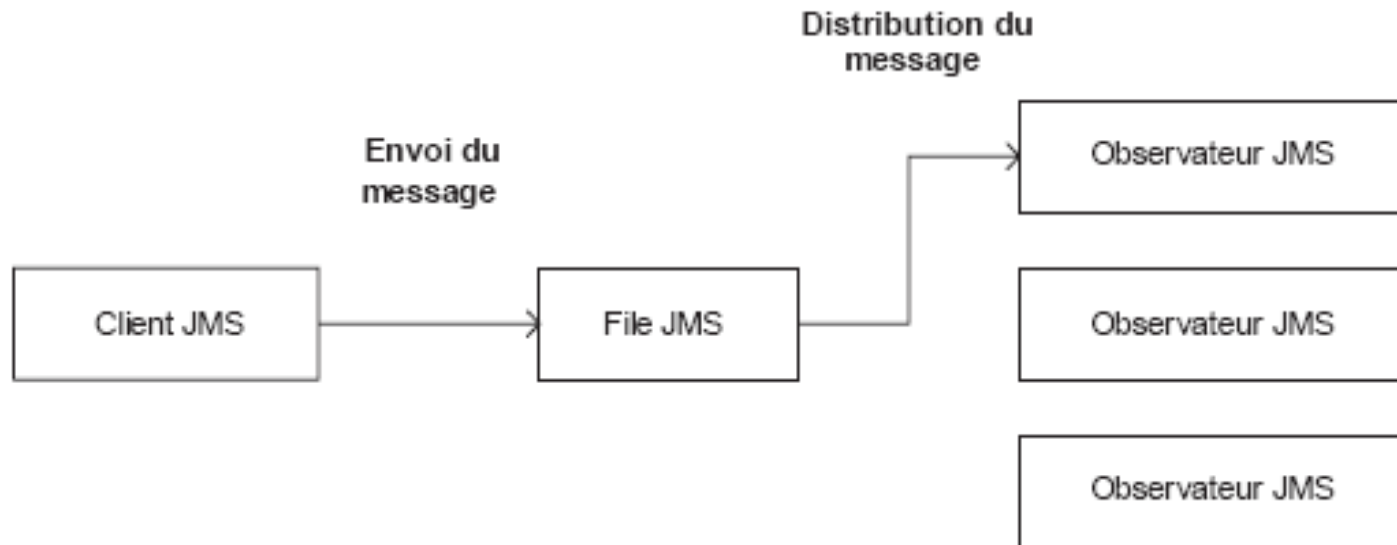
Le provider JMS

- Apache ActiveMQ(OpenSource Apache)
- OpenJMS (OpenSource)
- JBoss Messaging(OpenSource)
- ...
- WebShere MQ – IBM(Commercial)
- Oracle Advanced Queueing(Commercial)
- TIBCO Enterprise Message Service(Commercial)
- Sun Java System Message Queue(Commercial)
- ...

Spring JMS

Mécanisme de distribution des messages

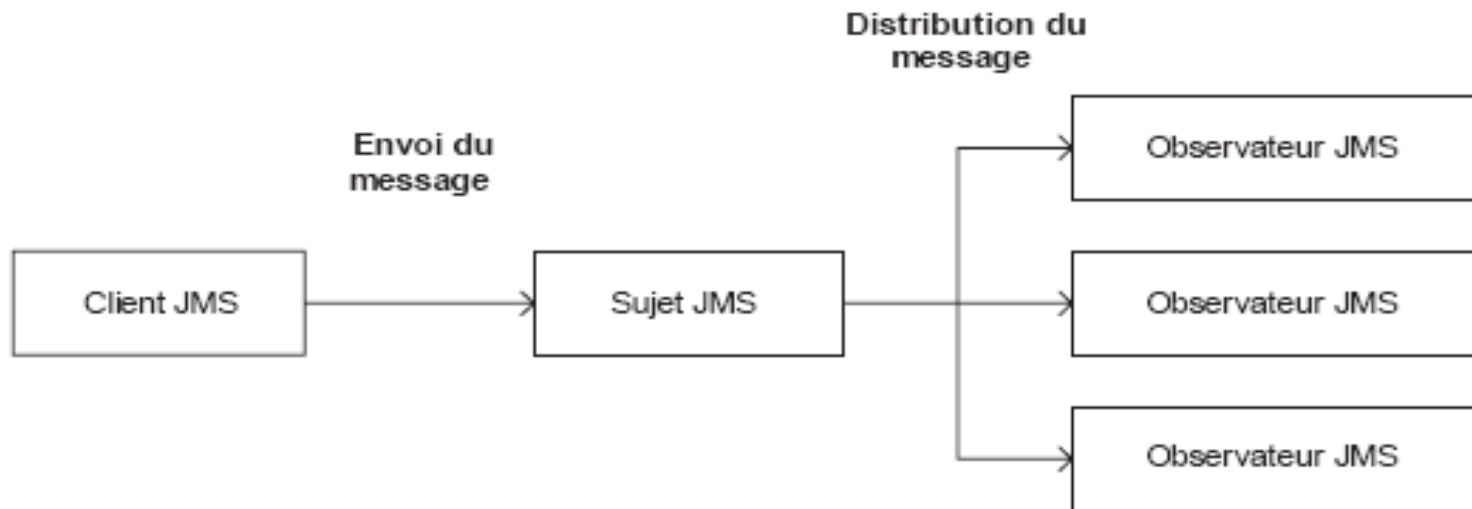
- Le premier, appelé file, ou **queue**, correspond à une distribution **point-à-point**.
- Un message envoyé sur un domaine de ce type est distribué une seule fois et à un seul observateur enregistré



Spring JMS

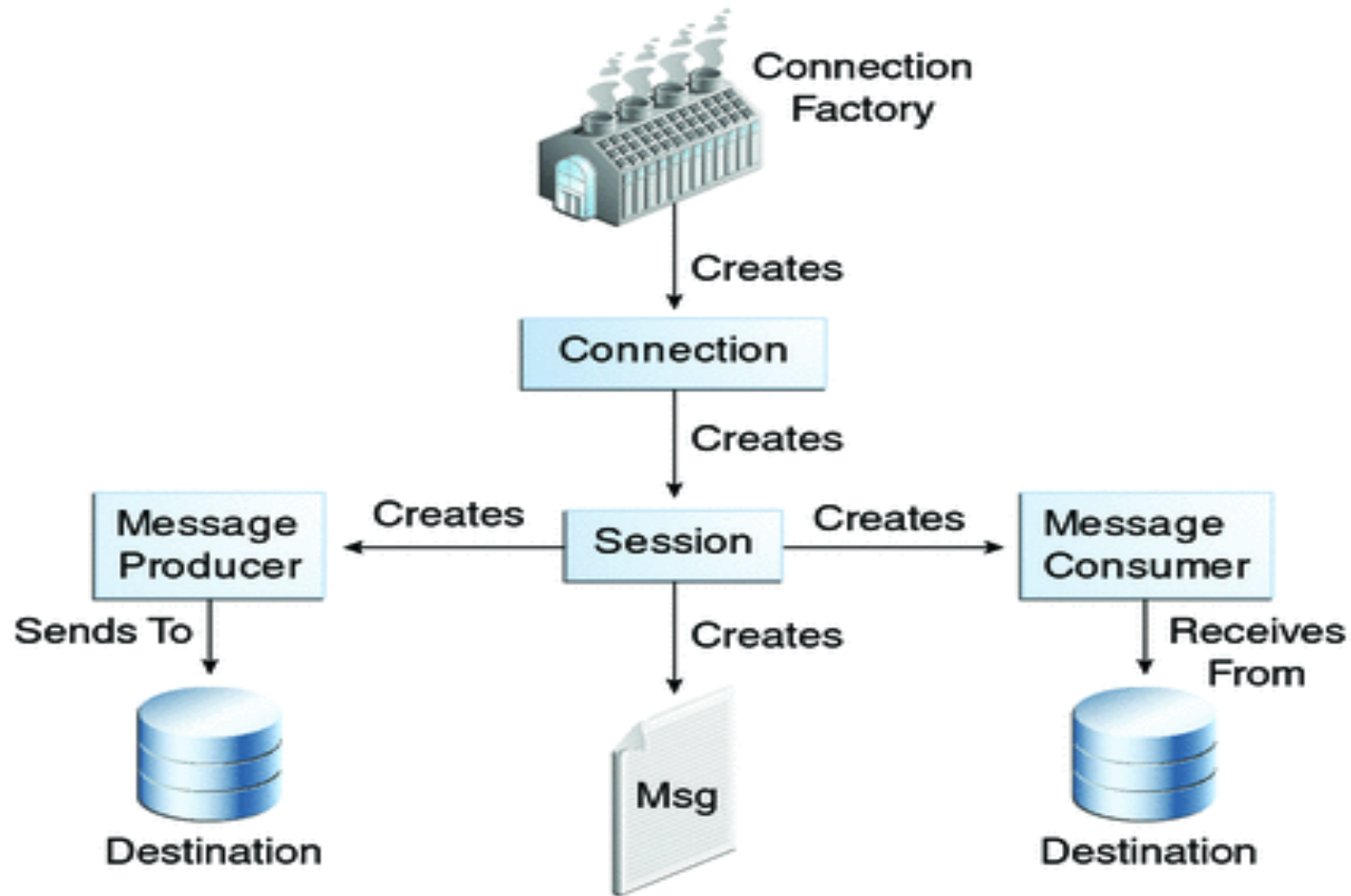
Mécanisme de distribution des messages

- Le second domaine, appelé sujet, ou ***topic***, fonctionne sur le principe des ***listes de diffusion***.
- Tous les observateurs enregistrés sur le domaine reçoivent le message envoyé.



Spring JMS

Classes Principales de l'api JMS



Spring JMS

Classes Principales de JMS

- Connection factorie
- Connections
- Sessions
- Messages
- Message producers
- Message consumers
- Destinations

Spring JMS

La fabrique de connexion

- Il existe deux types de factory : **QueueConnectionFactory** et **TopicConnectionFactory** selon le type d'échanges que l'on fait.
- Ce sont des interfaces que le broker de message doit implémenter pour fournir des objets.
- Pour obtenir un objet de ce type, il faut soit instancier directement un tel objet soit faire appel à JNDI pour l'obtenir.
- Chaque provider fournit sa propre solution pour gérer les objets contenus dans l'annuaire JNDI.
- La fabrique de type **ConnectionFactory** permet d'obtenir une instance de l'interface **Connection**.

Spring JMS

L'interface Connection

- Cette interface définit des méthodes pour la connexion au broker de messages.
- Cette connexion doit être établie en fonction du mode utilisé :
 - l'interface **QueueConnection** pour le mode point à point
 - l'interface **TopicConnection** pour le mode publication/abonnement
- La méthode **start()** permet de démarrer la connexion.

```
1. | connection.start();
```
- La méthode **stop()** permet de suspendre temporairement la connexion.
- La méthode **close()** permet de fermer la connexion.

Spring JMS

L'interface Session

- Comme pour la connexion, la création d'un objet de type Session dépend du mode de fonctionnement.
 - l'interface **QueueSession**
 - l'interface **TopicSession**
- Pour obtenir l'un ou l'autre, il faut utiliser un objet **Connection** correspondant de type **QueueConnection** ou **TopicConnection** avec la méthode correspondante : **createQueueSession()** ou **createTopicSession()**.

Spring JMS

Les messages

- Un message est constitué de trois parties :
 - L'en-tête (**header**) : contient des données techniques
 - Les propriétés (**properties**) : contient des données fonctionnelles
 - Le corps du message (**body**) : contient les données du message
- L'interface **Session** propose plusieurs méthodes **createXXXMessage()** pour créer des messages contenant des données au format XXX.
- Il existe aussi pour chaque format des interfaces filles de l'interface **Message** :
 - **BytesMessage** : message composé d'octets
 - **MapMessage** : message composé de paire clé/valeur
 - **ObjectMessage** : message contenant un objet sérialisé
 - **StreamMessage** : message issu d'un flux
 - **TextMessage** : message contenant du texte

Spring JMS

L'envoi de messages

- L'interface **MessageProducer** est la super interface des interfaces (**QueueSender** et **TopicPublisher**.) qui définissent des méthodes pour l'envoi de messages.
- Ces objets sont créés à partir d'un objet représentant la session :
 - la méthode **createSender()** pour obtenir un objet de type **QueueSender**
 - la méthode **createPublisher()** pour obtenir un objet de type **TopicPublisher**

Spring JMS

La réception de messages

- L'interface **MessageConsumer** est la super interface des interfaces(**QueueReceiver** et **TopicSubscriber**) qui définissent des méthodes pour la réception de messages.
- La réception d'un message peut se faire avec **deux modes** :
 - **synchrone** : dans ce cas, l'attente d'un message bloque l'exécution du reste de code
 - **asynchrone** : dans ce cas, un thread est lancé qui attend le message et appelle une méthode (callback) à son arrivée. L'exécution de l'application n'est pas bloquée.

Spring JMS

- La classe **JmsTemplate** : le template JMS de Spring
- La classe **JmsTemplate** est la classe de base pour faciliter l'envoi et la réception de message JMS de façon synchrone.
- Cette classe se charge de gérer la création et la libération des ressources utiles pour l'utilisation de JMS.
- La classe **JmsTemplate** propose des méthodes permettant :
 - d'envoyer un message,
 - de consommer un message de manière synchrone,
 - de permettre un accès à la session JMS et au message producer.

Spring JMS

ActiveMQ

Spring JMS

Configuration de ActiveMQ 5.4.2

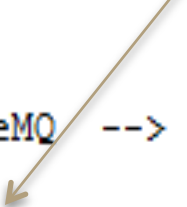
- Le bean **amqConnectionFactory** est une instance de type **ActiveMQConnectionFactory** : ce bean est une fabrique de connexions à un ActiveMQ installé en local et qui utilise le port **61616**.

Exemple :

Assurer la connexion avec le broker **ActiveMQ**.

```
<beans ....>

<!-- Fabrique de connexions à ActiveMQ -->
<bean id="amqConnectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>
.....
</beans>
```



Spring JMS

Configuration de ActiveMQ 5.4.2

- Déclaration de la queue qui sera utilisée pour nos échanges de messages :

```
<!-- Destination dans ActiveMQ -->  
<bean id="destination"  
      class="org.apache.activemq.command.ActiveMQQueue">  
    <constructor-arg value="local.maqueue" />  
</bean>
```

Le bean destination est une instance de type **ActiveMQQueue** : ce bean encapsule une queue nommée « **local.maqueue** ».

Spring JMS

L'envoi d'un message avec JmsTemplate

- L'envoi d'un message se fait en utilisant la méthode **send()** de la classe JmsTemplate qui attend en paramètre dans ces surcharges un objet de type **MessageCreator**.

```
public class JmsProducer {  
  
    private JmsTemplate jmsTemplate;  
    // Get+Set  
  
    public void envoyerMessage() {  
        jmsTemplate.send(new MessageCreator() {  
            public Message createMessage(final Session session)  
                throws JMSEException {  
                return session.createTextMessage("Message " + new Date());  
            }  
        });  
    }  
}
```

La classe JmsTemplate prend en charge la fermeture de la session JMS.

Spring JMS

Réception d'un message avec JmsTemplate

- Une instance de JmsTemplate peut être utilisée pour recevoir les messages de manière synchrone.
- La méthode **receive()** attend un nouveau message sur la destination par défaut de l'instance de JmsTemplate.
- Une autre surcharge de la méthode **receive()** attend un nouveau message sur la destination fournie en paramètre.

Spring JMS

Réception d'un message avec JmsTemplate

```
public class JmsConsumer {  
    private JmsTemplate jmsTemplate;  
  
    //Get+Set  
  
    public void recevoirMessage() {  
        Message msg = jmsTemplate.receive();  
        try {  
            TextMessage textMessage = (TextMessage) msg;  
            if (msg != null) {  
                System.out.println("Message = " + textMessage.getText());  
            }  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

La propriété **receiveTimeout** de la classe **JmsTemplate** permet de préciser un timeout d'attente puisque la réception est synchrone.

Remarque : La classe **JmsTemplate** peut être utilisé pour envoyer des messages mais elle n'est pas recommandée pour la réception de messages.

Pour la réception d'un message, il est préférable d'utiliser une solution asynchrone reposant sur un **MessageListenerContainer** de Spring.

Spring JMS

Configuration dans Spring

```
<!-- Instance de JmsTemplate qui utilise ConnectionFactory et la Destination -->
<bean id="producerTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="amqConnectionFactory" />
    <property name="defaultDestination" ref="destination" />
</bean>

<bean id="consumerTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="amqConnectionFactory" />
    <property name="defaultDestination" ref="destination" />
</bean>
```

Spring JMS

Configuration dans Spring

- Les beans **producerTemplate** et **consumerTemplate** sont des instances de type `JmsTemplate`.
- La propriété **connectionFactory** est initialisée avec le bean **amqConnectionFactory** et la propriété est initialisée avec le bean **destination**.

```
<bean id="jmsProducer" class="com.formation.jms.JmsProducer">
    <property name="jmsTemplate" ref="producerTemplate" />
</bean>

<bean id="jmsConsumer" class="com.formation.jms.JmsConsumer">
    <property name="jmsTemplate" ref="consumerTemplate" />
</bean>
```

Une instance de **JmsProducer** et de **JmsConsumer** sont déclarées avec en dépendante l'instance de `JmsTemplate` correspondante.

Spring JMS

Amélioration des performances

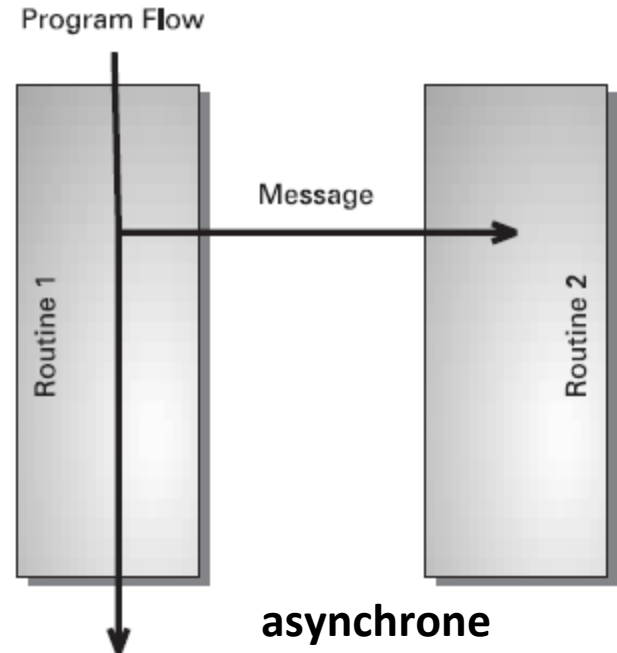
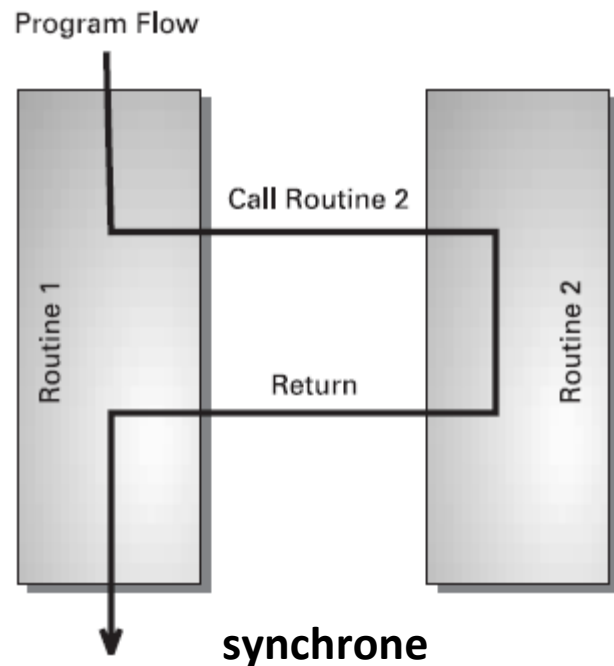
- La classe **CachingConnectionFactory** est un wrapper qui encapsule une connexion à un MOM en **proposant** une **reconnexion** au besoin **et** une **mise en cache** de certaines ressources (**connections, sessions**).
- Par défaut, la classe **CachingConnectionFactory** utilise une seule session pour créer les connections.
- Il est possible d'utiliser plusieurs sessions pour améliorer la montée en charge en utilisant la propriété **sessionCacheSize**.

```
<!-- Fabrique de connexions à ActiveMQ -->
<bean id="amqConnectionFactory"
      class="org.apache.activemq.ActiveMQConnectionFactory">
  <property name="brokerURL" value="tcp://localhost:61616" />
</bean>

<!-- Cache des connexions à ActiveMQ -->
<bean id="cachedConnectionFactory"
      class="org.springframework.jms.connection.CachingConnectionFactory">
  <property name="targetConnectionFactory" ref="amqConnectionFactory" />
  <property name="sessionCacheSize" value="3" />
</bean>
```

Spring JMS

La réception asynchrone de messages



•Spring propose plusieurs solutions pour permettre de recevoir des messages de manière asynchrone en utilisant des **MessageListenerContainer**.

- SimpleMessageListenerContainer**
- DefaultMessageListenerContainer**

Spring JMS

La réception asynchrone de messages

- La classe **SimpleMessageListenerContainer** est l'implémentation la plus simple : elle offre donc des fonctionnalités limitées(pas de support pour les transactions).
- L'utilisation de la classe **DefaultMessageListenerContainer** possède plusieurs avantages :
 - de bonnes **performances** grâce à la mise en cache des ressources JMS (connexions, sessions, consumers)
 - le nombre de consumers peut être modifié dynamiquement (méthodes **setConcurrentConsumers()** et **setMaxConcurrentConsumers()**) ce qui permet de traiter plus de messages de manière concurrente
 - le **support de plusieurs modes d'acquittement** des messages (acknowledgement)

Spring JMS

Exemple Configuration d'un MessageListenerContainer

```
<beans ...>

<bean id="amqConnectionFactory" .../>

<!-- Destination dans ActiveMQ -->
<bean id="destination" .../>

<!-- Message Driven POJO (MDP) -->
<bean id="messageListener"
      class="com.formation.jms.MonSimpleMessageListener" />

<!-- and this is the message listener container -->
<bean id="jmsContainer"
      class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="amqConnectionFactory" />
  <property name="destination" ref="destination" />
  <property name="messageListener" ref="messageListener" />
</bean>
....
</beans>
```

assurer la connexion avec le broker ActiveMQ

le message listener JMS

listener-container

Spring JMS

La réception asynchrone de messages

- Le **message listener** est un bean qui être implémenté de plusieurs manières :
 - en implémentant l'interface `javax.jms.MessageListener`
 - en implémentant l'interface **`SessionAwareMessageListener`** qui permet un accès l'objet Session JMS. La gestion des exceptions est à gérer par la classe par exemple en redéfinissant la méthode **`handleListenerException()`**
 - en implémentant l'interface **`MessageListenerAdapter`** qui permet de gérer les messages en masquant l'API JMS

Spring JMS

La réception asynchrone de messages

- L'utilisation du message listener container de Spring possède plusieurs **avantages** :
 - il peut être utilisé dans différents contextes : conteneur web, conteneur Java EE, application standalone
 - il peut utiliser n'importe quel MOM qui respecte les spécifications JMS. Il faut définir un bean de type connection factory et éventuellement définir quelques propriétés dans le listener-container

Spring JMS

Exemple :

```
...  
public class MonSimpleMessageListener implements MessageListener {  
  
    public void onMessage(final Message message) {  
        try {  
            TextMessage msg = (TextMessage) message;  
            System.out.println(" Message reçu : " + msg.getText());  
        } catch (JMSEException e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Spring MVC

- **SpringMVC** est un framework de présentation, pour application WEB, suivant le modèle MVC, et fondé sur le conteneur léger de SPRING.
- Dans le cas de **SpringMVC** le conteneur va servir à créer:
 - Le contexte de l'application Web
 - Les objets traitant les requêtes (Controller)
 - Les objets créant les pages HTML (View)
 - Les objets données des formulaires (Command)
 - Les liens avec les couches métiers et BD
 - Et pleins d'autres
 - Le mapping des URL vers les controleurs
 - Le mapping des vues , etc.
- L'inversion du contrôle permet ensuite de changer le comportement de l'application, en modifiant la description xml du conteneur, sans changer les éléments programmés!

Spring MVC : les concepts

ARCHITECTURE

Handler Mapping

Controller

View Resolver

Form Handling

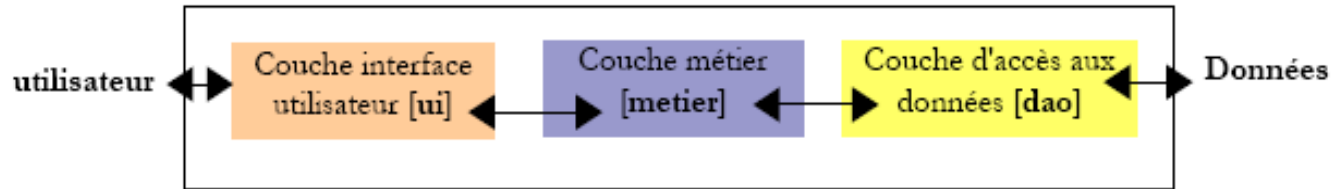
Spring's Form Tag Library

Integrating Spring MVC with other frameworks

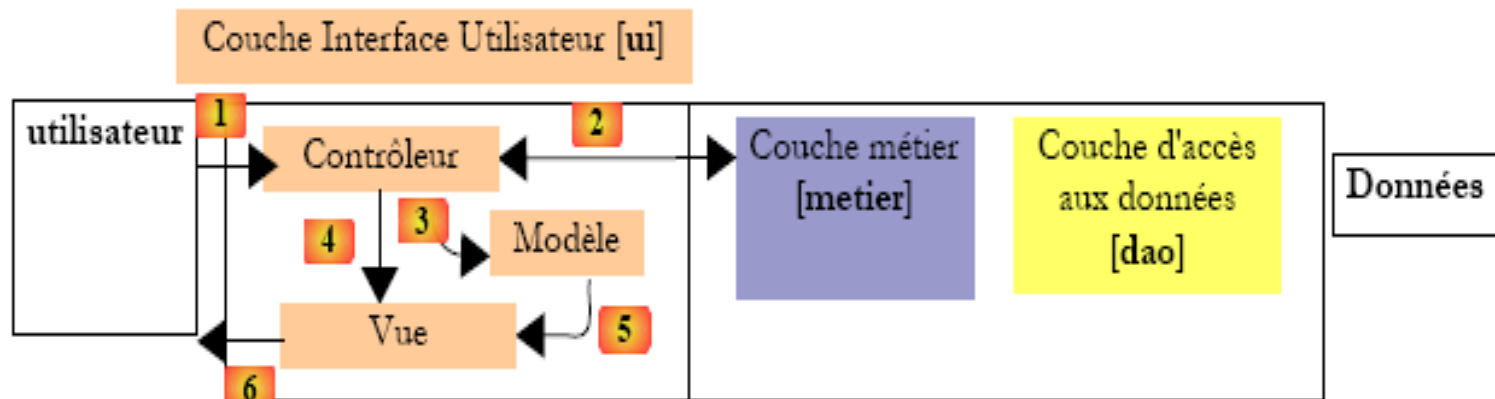
Spring MVC

Retour sur le modèle MVC

Une application 3tier classique:

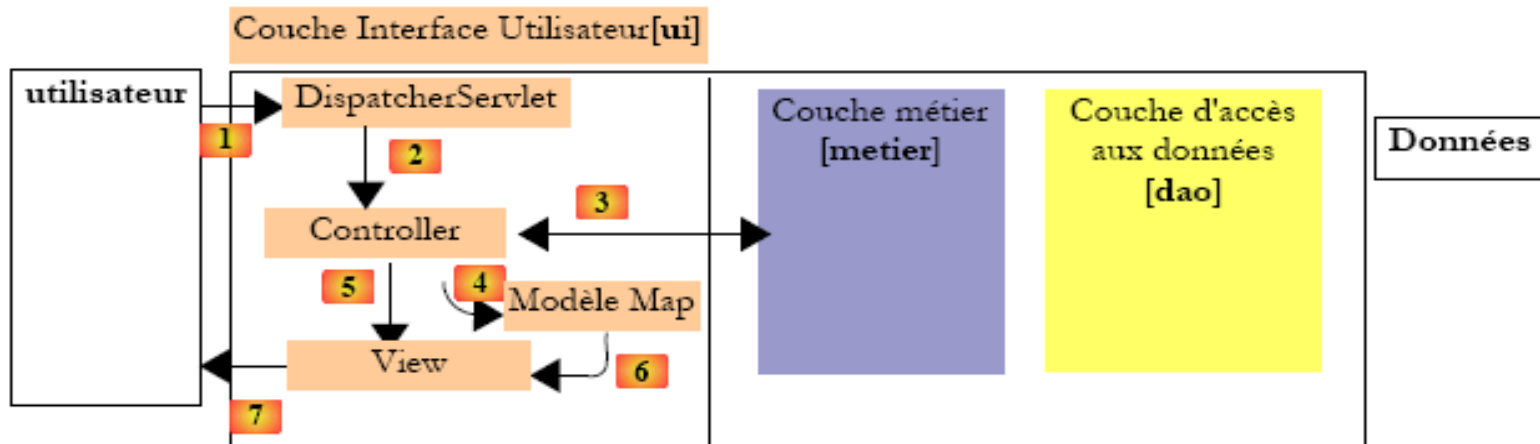


Une application 3tier avec MVC:



Spring MVC

La vision de SpringMVC



La **org.springframework.web.servlet.DispatcherServlet** est le point d'entrée générique qui délègue les requêtes à des **Controller**

Un **org.springframework.web.servlet.mvc.Controller** prend en charge une requête, et utilise la couche métier pour y répondre.

Un **Controller** fabrique un modèle sous la forme d'une **java.util.Map** contenant les éléments de la réponse.

Un **Controller** choisit une **org.springframework.web.servlet.View** qui sera paramétrée par la **Map** pour donner la page qui sera affichée.

Spring MVC

- L'interface **Controller la plus simple (sans formulaire)** n'a qu'une méthode ***handleRequest***.

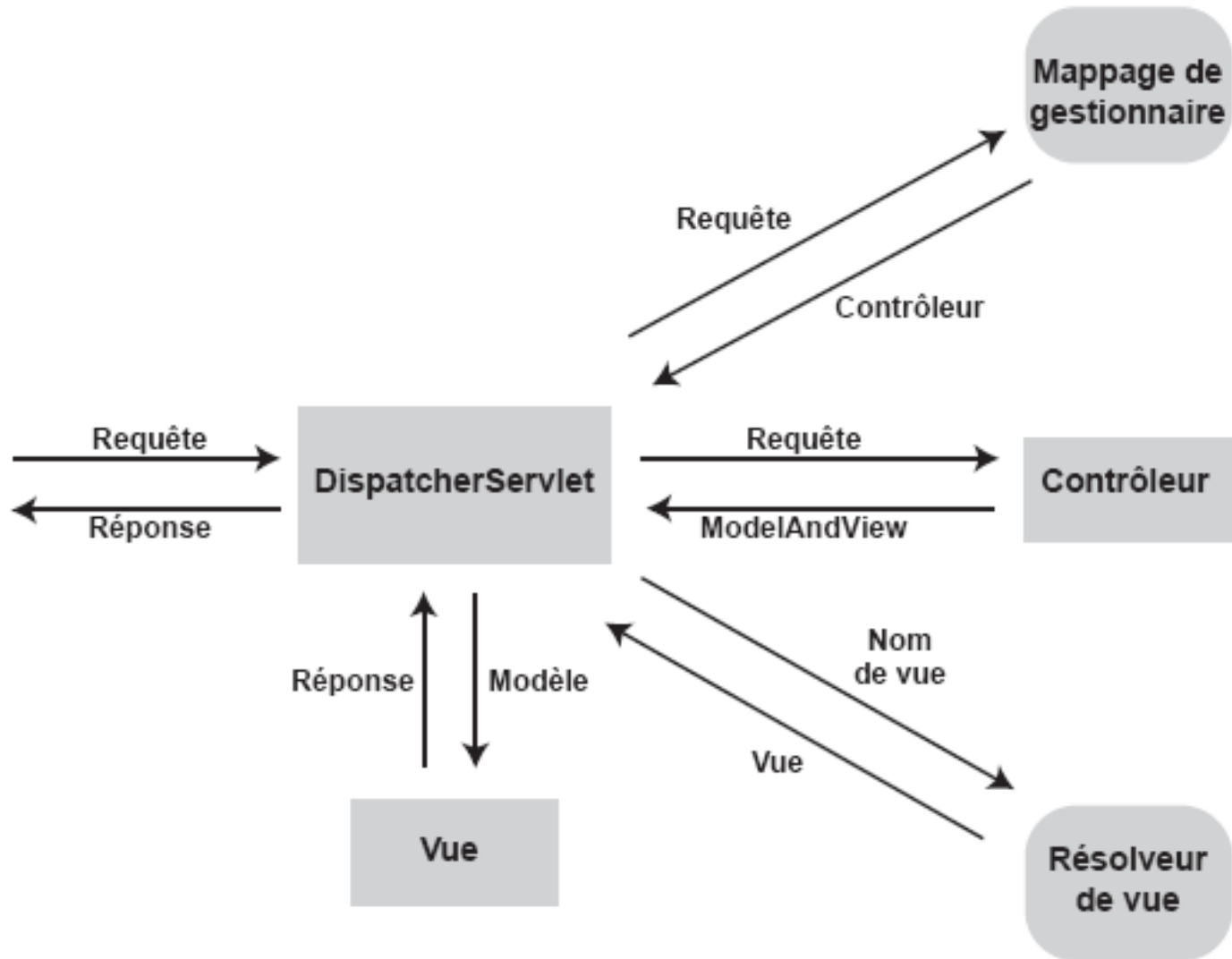
ModelAndView handleRequest(HttpServletRequest requete, HttpServletResponse response)

- Cette méthode reçoit la requête, doit la traiter (c'est à dire fabriquer les données de réponse grâce à la couche métier) et retourner un objet **ModelAndView**
- Le principal constructeur de ModelAndView utilisé est celui-ci:

ModelAndView (String ViewName, Map model)

- Il faut fournir le nom de la vue à utiliser pour créer la page de réponse et une HashMap contenant les données de réponse à y insérer.

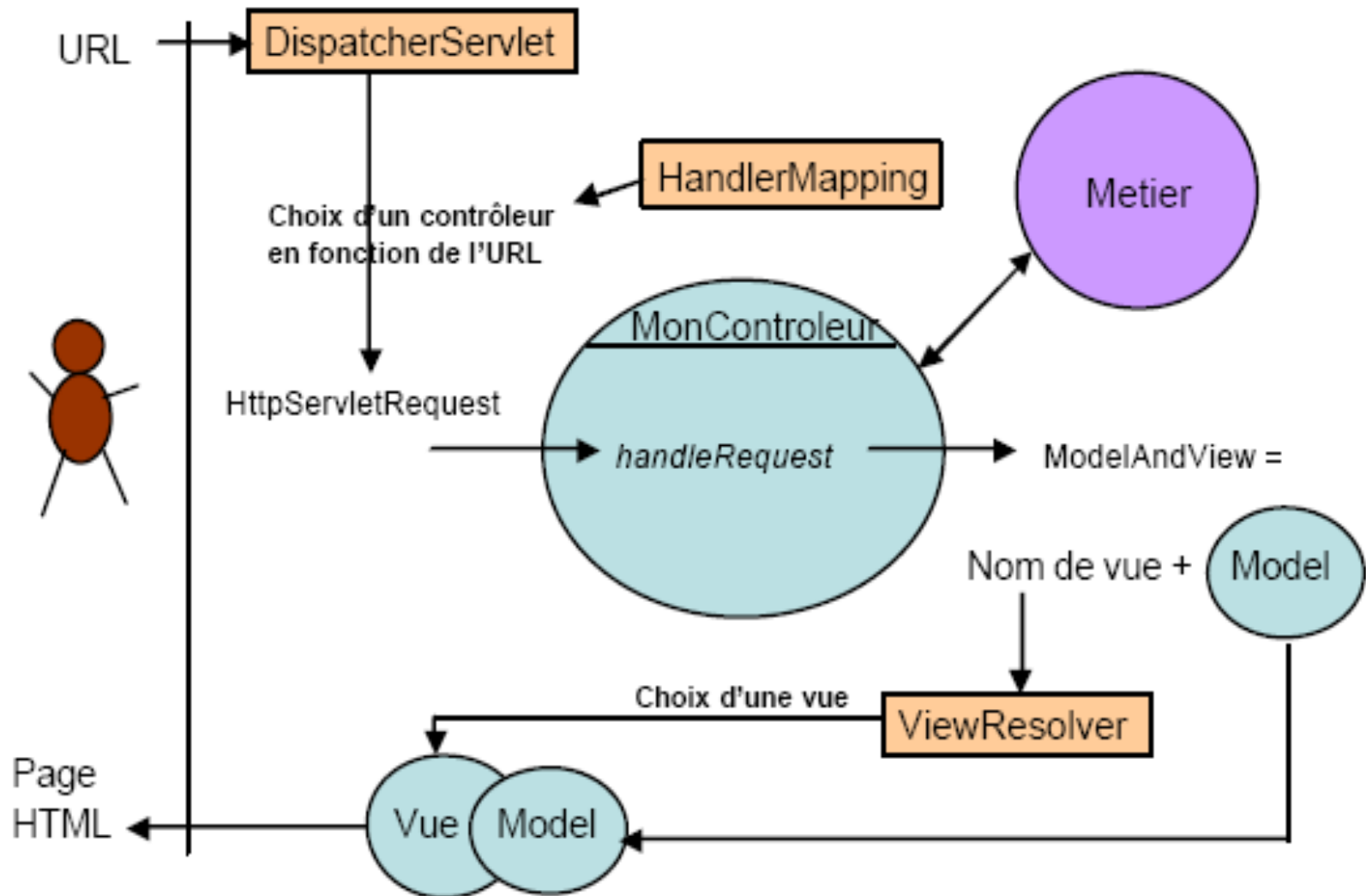
Spring MVC



Spring MVC

- **DispatcherServlet** reçoit une requête.
- Le DispatcherServlet consulte le **HandlerMapping** et invoque le **contrôleur** associé à la requête.
- Le processus de contrôleur de la requête appelle les méthodes appropriées de service et retourne un objet **ModeAndView** à DispatcherServlet. L'objet ModelAndView contient les données du modèle et le nom de la vue.
- Le DispatcherServlet envoie le nom de la vue à un **ViewResolver** afin de trouver la vue réelle à invoquer.
- Le DispatcherServlet passe l'objet du modèle à la **vue**.
- La vue à l'aide des données du modèle rend le résultat à l'utilisateur

Spring MVC



Spring MVC

Exemple: 1 – La couche métier

```
package com.formation.court.domain;

public class Player {

    private String name;
    private String phone;

    // Get+set+constructeur+toString
    ...
}
```

```
package com.formation.court.domain;

public class SportType {

    private int id;
    private String name;

    // Get+set+constructeur+toString
    ...
}
```

```
package com.formation.court.domain;
...
public class Reservation {

    private String courtName;
    private Date date;
    private int hour;
    private Player player;
    private SportType sportType;

    // Get+set+constructeur+toString
    ...
}
```

Spring MVC

Exemple: 2 – La couche service

```
package com.formation.court.service;
...
public interface ReservationService {
    public List<Reservation> query(String courtName);
}
```

```
package com.formation.court.service;
...
public class ReservationServiceImpl implements ReservationService {

    public static final SportType TENNIS = new SportType(1, "Tennis");
    public static final SportType SOCCER = new SportType(2, "Football");

    private List<Reservation> reservations;

    public ReservationServiceImpl() {
        reservations = new ArrayList<Reservation>();
        reservations.add(new Reservation("Tennis #1", new GregorianCalendar(2008,0,
14).getTime(), 16, new Player("Roger", "N/A"), TENNIS));
        ....
    }

    public List<Reservation> query(String courtName) {
        List<Reservation> result = new ArrayList<Reservation>();
        for (Reservation reservation : reservations) {
            if (reservation.getCourtName().equals(courtName)) {
                result.add(reservation);
            }
        }
        return result;
    }
}
```

Spring MVC

Exemple: 3 – La couche web (contrôleur I)

```
package com.formation.court.web;
....
public class WelcomeController extends AbstractController {

    public ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        Date today = new Date();
        return new ModelAndView("welcome", "today", today);
    }
}
```

Dans la méthode **handleRequest()**, le traitement d'une requête web peut se faire comme dans une servlet.

Nous devons retourner un objet de type **ModelAndView** qui contient un nom de vue ou un objet de vue, ainsi que les attributs du modèle.

Spring MVC

Exemple: 3 – La couche web (contrôleur II)

```
package com.formation.court.web;
....
public class ReservationQueryController extends AbstractController {

    private ReservationService reservationService;

    public void setReservationService(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    public ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        String courtName = ServletRequestUtils.getStringParameter(request,
            "courtName");

        List<Reservation> reservations = null;
        if (courtName != null) {
            reservations = reservationService.query(courtName);
        }
        return new ModelAndView("reservationQuery", "courtName", courtName)
            .addObject("reservations", reservations);
    }
}
```

ServletRequestUtils :

fournit des méthodes statiques :

- évite les conversions
- évite les vérifications.
- ...

Spring MVC

Exemple: 4 – La couche web (Création des vues)

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<html>
<head>
<title>Bienvenue</title>
</head>
<body>
    <h2>Bienvenue sur le système de réservation des terrains</h2>
    Nous sommes le
    <fmt:formatDate value="${today}" pattern="dd-MM-yyyy" />.
</body>
</html>
```

Fichier /WEB-INF/jsp/welcome.jsp

Spring MVC

Exemple: 5 – La couche web (Création des vues)

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt"%>
<html>
<head>
<title>Demande de réservation</title>
</head>
<body>
    <form method="POST">
        Nom du terrain <input type="text" name="courtName"
            value="${courtName}" /> <input type="submit" value="Query" />
    </form>
    <table border="1">
        <tr>
            <th>Nom du terrain</th>
            <th>Date</th>
            <th>Horaire</th>
            <th>Joueur</th>
        </tr>
        <c:forEach items="${reservations}" var="reservation">
            <tr>
                <td>${reservation.courtName}</td>
                <td><fmt:formatDate value="${reservation.date}"
                    pattern="dd-MM-yyyy" /></td>
                <td>${reservation.hour}</td>
                <td>${reservation.player.name}</td>
            </tr>
        </c:forEach>
    </table>
</body>
</html>
```

Fichier /WEB-INF/jsp/reservationQuery.jsp

Spring MVC

Exemple: 6 – *Créer les fichiers de configuration*

```
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="http://java.sun.com/xml/ns/javaee" xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
  id="WebApp_ID" version="2.5">
  <display-name>Court Reservation System</display-name>

  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/court-service.xml</param-value>
  </context-param>

  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>

  <servlet>
    <servlet-name>court</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>court</servlet-name>
    <url-pattern>*.htm</url-pattern>
  </servlet-mapping>

</web-app>
```



charge les
fichiers de
configuration
des beans

Le descripteur de déploiement web web.xml

Spring MVC

Exemple: 6 – *Créer les fichiers de configuration*

Par défaut, le nom de ce fichier est celui de la servlet auquel est adjoint **-servlet.xml**.

Par défaut, **DispatcherServlet** utilise un **BeanNameUrlHandlerMapping** pour le mappage des gestionnaires

```
<beans ....>

<bean name="/welcome.htm" class="com.formation.court.web.WelcomeController" />

<bean name="/reservationQuery.htm"
      class="com.formation.court.web.ReservationQueryController">
    <property name="reservationService" ref="reservationService" />
</bean>

<bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>

</beans>
```

court-servlet.xml

InternalResourceViewResolver convertit les noms de vues en fichiers JSP stockés dans le répertoire **/WEB-INF/jsp/**

Spring MVC

court-service.xml

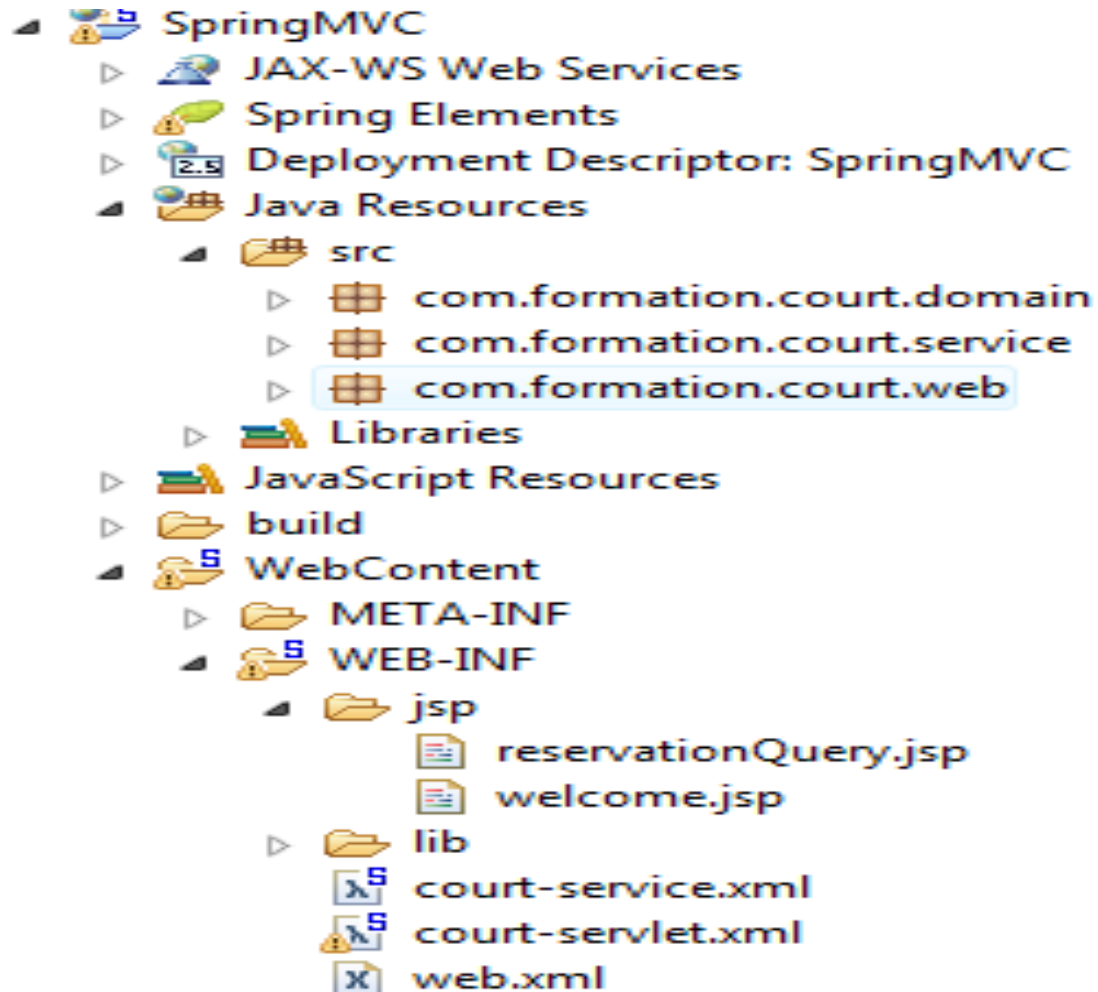
```
<?xml version="1.0" encoding="UTF-8"?>

<beans ...>
    <bean id="reservationService"
          class="com.formation.court.service.ReservationServiceImpl" />
</beans>
```

Il est préférable de déclarer un fichier de configuration des beans pour chaque couche, comme **court-persistence.xml** pour la couche de persistance et **court-service.xml** pour la couche de service.

Spring MVC

Exemple: 7 – *Arborescence du projet Web (Eclipse)*



Spring MVC

Associer des requêtes à des gestionnaires

- Par défaut, DispatcherServlet utilise **BeanNameUrlHandlerMapping**, qui associe les requêtes aux gestionnaires en fonction des motifs d'URL indiqués dans les noms des beans.
- Pour la mettre en œuvre, nous devons déclarer le nom de bean de chaque gestionnaire sous forme d'un motif d'URL.

```
<bean name="/welcome.htm" class="com.formation.court.web.WelcomeController" />
```

Spring MVC

Associer des requêtes d' après des définitions de mappage personnalisées

- La stratégie la plus directe et la plus souple pour associer les requêtes à des gestionnaires consiste à définir explicitement les mappages entre des motifs d' URL et des gestionnaires.
- Pour cela, nous utilisons **SimpleUrlHandlerMapping**.

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  <property name="mappings">
    <props>
      <prop key="/welcome.htm">welcomeController</prop>
      <prop key="/reservationQuery.htm">
        reservationQueryController
      </prop>
    </props>
  </property>
</bean>

<bean id="welcomeController" class="com.formation.court.web.WelcomeController" />

<bean id="reservationQueryController" class="com.formation.court.web.ReservationQueryController">
  <property name="reservationService" ref="reservationService" />
</bean>
```

Spring Remoting

- Le **remoting** est un mécanisme qui permet de mettre en place des applications distribuées en permettant à des composants d'invoquer des traitements sur des autres composants distants.
- La mise en place d'un mécanisme de remoting implique plusieurs autres mécanismes qui sont :
 - la sérialisation/désérialisation des objets pour pouvoir les transférer,
 - un mécanisme d'exposition de services
 - et un mécanisme de localisation et d'invocation de ces services.
- On dénombre plusieurs solutions de remoting, comme par exemple *Corba*, *Java RMI*, les *Web Services*, etc.

Spring Remoting

- Spring remoting facilite l' exposition de services et l' appel distants à ces services par des clients.
- Il fournit des « **Exporter** » qui facilitent l' exposition des services en fonction de la technologie et des « **proxy** » .
- Les **protocoles** gérés par le framework sont les suivants :
 - **RMI**
 - **HTTP avec Hessian**, Hessian2 (protocole pour les web services avec un envoi de données binaires)
 - **HTTP avec Burlap** (similaire à Hessian, mais le format des données est du XML).
 - **SOAP avec JAX-WS et JAX-RPC** (ancêtre de JAX-WS)
 - **JMS**

Spring Remoting

Spring RMI

La mise en place « standard » de service par RMI:

1. créer son service qui étend **java.rmi.Remote**
2. gérer les **java.rmi.RemoteException** pour chaque méthode
3. générer le **stub** et **skeleton** avec **rmic** (optionel depuis java 5)
4. **démarrer le service de registre RMI**
5. **enregistrer le service**
6. **implémenter le client** dépendant du stub
7. faire des **lookup** et **try-catch** etc...

Spring Remoting

Spring RMI

Maintenant voyons ce que Spring propose, pour exposer un service .

Prenons l' exemple ci-dessous :

```
public interface CustomerService {  
    List<Client> getCustomers(int numberOfCustomers);  
}
```

Interface du service

```
public class CustomerServiceImpl implements CustomerService {  
    @Override  
    public List<Client> getCustomers(int numberOfCustomers) {  
        List<Client> clients = new ArrayList<Client>();  
        // code  
        return clients;  
    }  
}
```

Implémentation

Spring Remoting

Spring RMI

La configuration pour l'exposition du service de notre exemple est la suivante :

```
<!-- Service -->  
<bean id="customerService" class="com.formation.service.CustomerServiceImpl" />
```

Déclarer un exporter pour chaque service

```
<!-- RMI -->  
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">  
  <property name="serviceName" value="ourCustomerService" />  
  <property name="service" ref="customerService" />  
  <property name="serviceInterface"  
    value="com.formation.service.CustomerService" />  
  <property name="registryPort" value="1299" />  
</bean>
```

Nommer le service

Référencer le bean service

port utilisé par le serveur

Référencer de l'interface

Spring Remoting

Spring RMI

Pour le serveur:

- Un démarrage du contexte Spring permet d'exposer le service.

Pour le client:

```
<!-- RMI proxy client -->  
<bean id="customerServiceProxy" class="org.springframework.remoting.rmi.RmiProxyFactoryBean">  
    <property name="serviceUrl" value="rmi://localhost:1299/ourCustomerService" />  
    <property name="serviceInterface"  
        value="com.formation.service.CustomerService" />  
</bean>
```

Spring Remoting

Spring Hessian

- Hessian est un protocole binaire, basé sur http pour exporter un service.
- Il est normalement indépendant du langage, mais actuellement principalement utilisé pour des applications Java à Java.
- Afin d'exporter un service avec Hessian, il est nécessaire que celui-ci soit composé d'une interface et d'une implémentation.

```
public interface CustomerService {  
    public List<Client> getCustomers(int numberOfCustomers);}  
  
public class CustomerServiceImpl implements CustomerService {  
    @Override  
    public List<Client> getCustomers(int numberOfCustomers) {  
        List<Client> clients = new ArrayList<Client>();  
        // code  
        return clients;    }  
}
```

Spring Remoting

Spring Hessian

Exporter un service avec Hessian.

```
<servlet>
  <servlet-name>clientService</servlet-name>
  <servlet-class>
    com.caucho.hessian.server.HessianServlet
  </servlet-class>
  <init-param>
    <param-name>home-class</param-name>
    <param-value>com.formation.service.CustomerServiceImpl</param-value>
  </init-param>
  <init-param>
    <param-name>home-api</param-name>
    <param-value>com.formation.service.CustomerService</param-value>
  </init-param>
  <load-on-startup>0</load-on-startup>
</servlet>

<servlet-mapping>
  <servlet-name>clientService</servlet-name>
  <url-pattern>/client</url-pattern>
</servlet-mapping>
```

Fournie par *Hessian*

Pointe sur l'implémentation

Pointe sur l'interface du service

Associe cette *Servlet* avec un *url*

Spring Remoting

Spring Hessian

Pour le serveur:

- Télécharger Hessian et l'inclure dans WEB-INF/lib
- <http://hessian.caucho.com/>

Pour le client : il se connecter au service comme suit:

```
<bean id="hessianServiceProxy"
  class="org.springframework.remoting.caucho.HessianProxyFactoryBean">
  <!-- Configuration de l'url d'accès -->
  <property name="serviceUrl">
    <value>http://localhost:8080/remoting-server/client</value>
  </property>

  <!-- Configuration de l'interface du service que le proxy doit implémenté -->
  <property name="serviceInterface">
    <value>com.formation.service.CustomerService</value>
  </property>
</bean>
```


Spring Remoting

Spring Burlap

- **Burlap** est un protocole basé sur XML et sur http pour exporter un service.
- Malheureusement, L'implémentation de **Burlap** impose que les implémentations des service héritent d'une **Servlet** particulière (*com.caucho.burlap.server.BurlapServlet*).

Exemple:

```
public class CustomerBurlapServiceImpl extends BurlapServlet implements CustomerService{  
    @Override  
    public List<Client> getCustomers(int numberOfCustomers) {  
        //code  
        return clients;  
    }  
}
```

Spring Remoting

Spring Burlap

Configuration Spring (idem Hersian)

```
<bean id="burlapServiceProxy"
      class="org.springframework.remoting.caucho.BurlapProxyFactoryBean">
  <!-- Configuration de l'url d'accès -->
  <property name="serviceUrl">
    <value>http://localhost:8080/remoting-server/burlap</value>
  </property>
  <!-- Configuration de l'interface du service que le proxy doit implémenté -->
  <property name="serviceInterface">
    <value>com.formation.service.CustomerService</value>
  </property>
</bean>
```

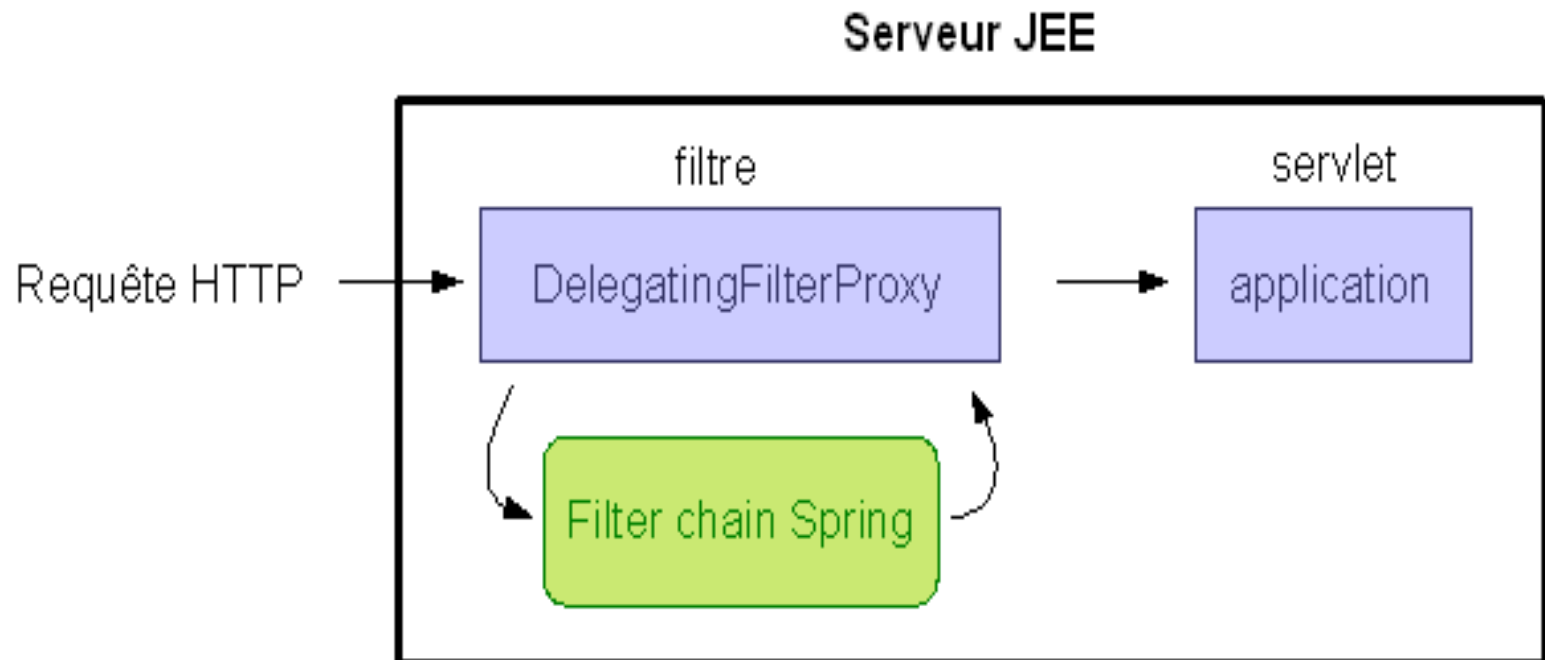
Spring Security

- Spring Security permet **de gérer l'accès** aux ressources d'une application Java.
- Ces ressources peuvent être des pages web, mais aussi des objets de **services métier**.
- Toute **ressource** sollicitée par un appelant est rendue **accessible** si :
 - d'une part, **l'appelant s'est identifié**,
 - et si d'autre part, il possède les **droits nécessaires (rôles)**.
- Dans ce chapitre article, nous traiterons le cas de la sécurisation des pages web.

Spring Security

Principe de Spring Security

- Les requêtes HTTP sont interceptées par un filtre de servlet qui délègue à un bean Spring les traitements de vérification d'accès aux pages web.



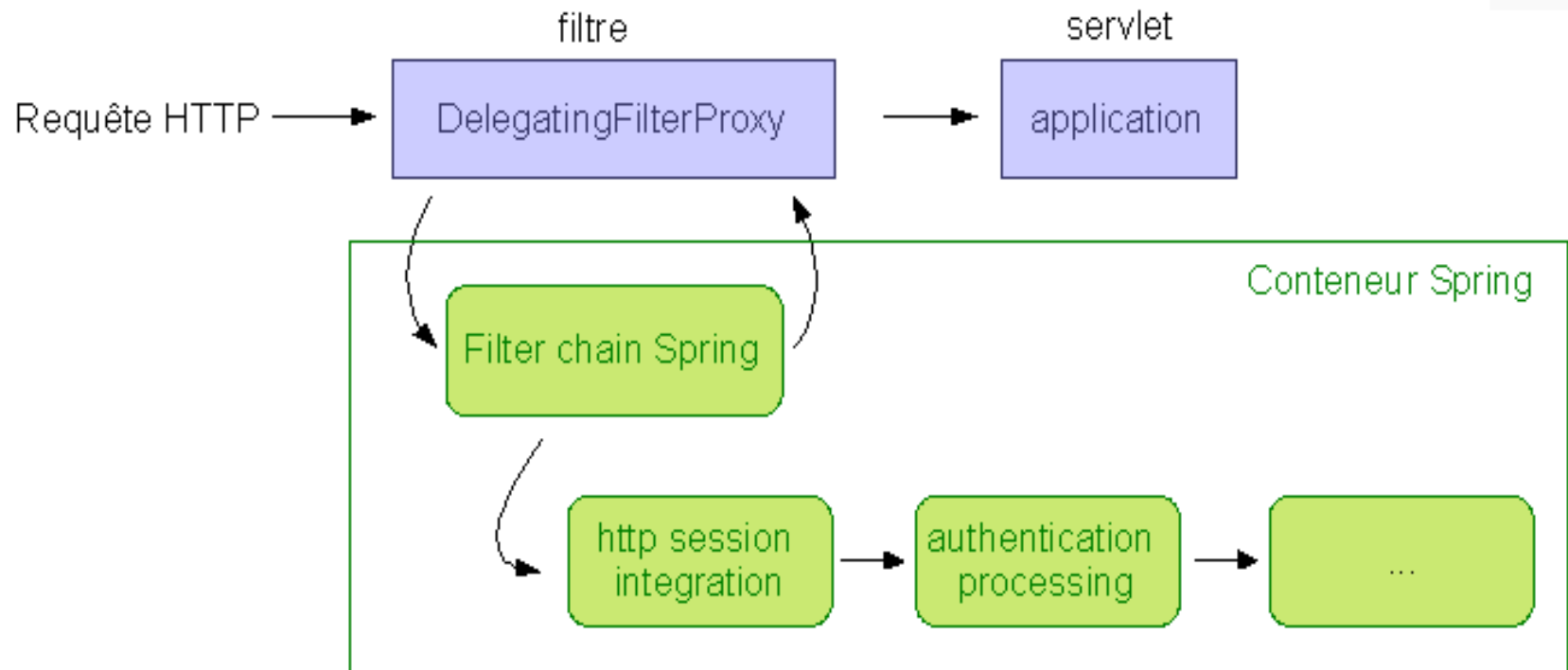
Spring Security

Principe de Spring Security

- Ce bean met en œuvre une chaîne de filtres.
- Chacun des filtres est un bean auquel est attribué une tâche précise :
 - **Intégration dans la session** HTTP des informations de sécurité contenues dans la requête
 - **Vérification de l'identité** de l'appelant et affichage d'une invite de connexion si nécessaire
 - **Vérification des droits d'accès** à la ressource sollicitée
 - ...

Spring Security

Principe de Spring Security



Spring Security

Principe de Spring Security

- Certains filtres sont obligatoires, d'autres optionnels.
- La chaîne de filtres est configurable.
- Spring Security offre ainsi les fonctionnalités suivantes :
 - Authentification anonyme
 - Fonction *Remember Me*
 - Gestion NTLM
 - Intégration avec un serveur LDAP ou un serveur CAS
 - Gestion des certificats X509

Spring Security

Installer Spring Security

- Les fichiers jar

<http://www.springsource.org/download>

- Configurer le fichier web.xml

```
<filter>
  <filter-name>springSecurityFilterChain</filter-name>
  <filter-class>
    org.springframework.web.filter.DelegatingFilterProxy
  </filter-class>
</filter>

<filter-mapping>
  <filter-name>springSecurityFilterChain</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```


Spring Security

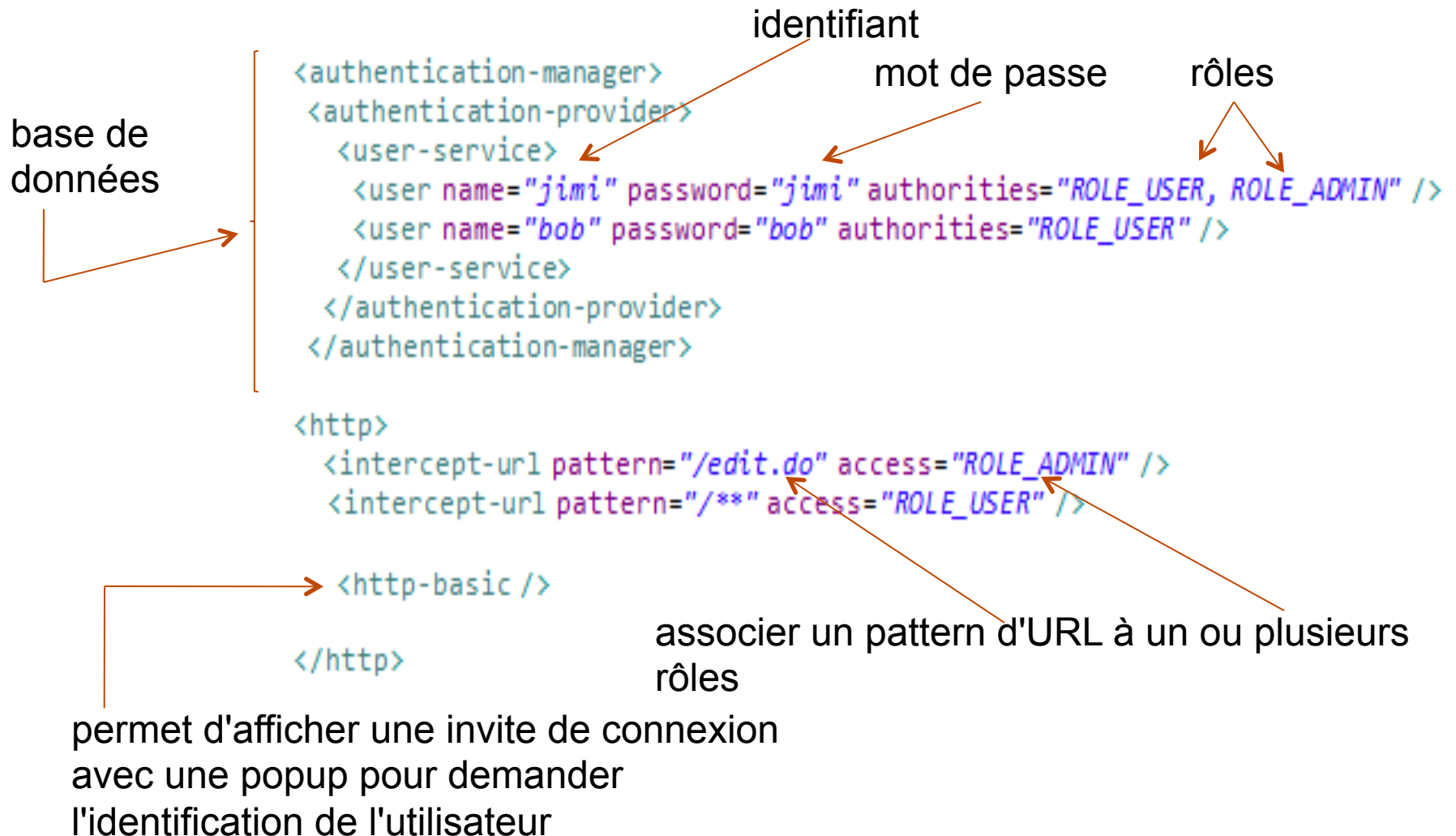
Installer Spring Security

- Schéma XML

```
<beans:beans xmlns="http://www.springframework.org/schema/security"
  xmlns:beans="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-
3.0.xsd
    http://www.springframework.org/schema/security
    http://www.springframework.org/schema/security/spring-
security-3.1.xsd">
  ...
</beans:beans>
```

Spring Security

Configuration de base



Spring Security

EXERCICE SpringSecurity_1

Spring Security

Une configuration de base

Dans le fichier de spring-security , y ajouter les balises suivantes :

```
27
28
29
30 <security:authentication-provider>
31   <security:user-service>
32     <security:user name="admin" password="admin"
33       authorities="ROLE_ADMIN,ROLE_USER" />
34     <security:user name="guest" password="guest"
35       authorities="ROLE_USER" />
36   </security:user-service>
37 </security:authentication-provider>
38
39 <security:http>
40   <security:intercept-url pattern="/editerproduit.do"
41     access="ROLE_ADMIN" />
42   <security:intercept-url pattern="/**" access="ROLE_USER" />
43
44   <security:http-basic />
45 </security:http>
46
```

Spring Security

Cet exemple est complet et tout à fait fonctionnel.

La base de données utilisateur est définie en dur dans le fichier de configuration. Pour chaque utilisateur, on spécifie l'identifiant, le mot de passe et les rôles qui lui sont attribués avec l'élément `user`.

On définit aussi les règles d'accès aux pages de l'application à l'aide de l'élément `intercept-url`.

Il s'agit simplement d'associer un pattern d'URL à un ou plusieurs rôles (séparation avec une virgule).

A noter que les patterns d'URL sont traités dans l'ordre de déclaration : si `/**` était déclaré en premier, les autres patterns ne seraient pas vérifiés.

Le dernier élément `http-basic` permet d'afficher une invite de connexion avec une popup pour demander l'identification de l'utilisateur :

Spring Security

Connexion à localhost



Le serveur localhost à l'adresse Spring Security Application requiert un nom d'utilisateur et un mot de passe.

Avertissement : ce serveur requiert que votre nom d'utilisateur et votre mot de passe soient envoyés de façon non sécurisée (authentification de base sans connexion sécurisée).

Nom d'utilisateur :

Mot de passe :

☐ Mémoriser mon mot de passe

Spring Security

Dans notre exemple, toutes les pages sont filtrées.

La demande d'identification se produit donc quelle que soit la page sollicitée. Une fois l'utilisateur identifié, les informations de sécurité le concernant sont stockées en session HTTP dans le security context.

L'identification n'est par conséquent pas demandée une nouvelle fois lorsque l'on navigue vers d'autres pages.

Si l'utilisateur tente de naviguer vers une page non autorisée (par exemple `editerproduit.do` pour l'utilisateur `guest`), Spring Security renvoie une erreur 403 au navigateur web :

Spring Security

Etat HTTP 403 - Access is denied

type Rapport d'état

message Access is denied

description L'accès à la ressource demandée (Access is denied) a été interdit.

Si l'identification de l'utilisateur est incorrecte (identifiant ou mot de passe), Spring Security renvoie une erreur 401 au navigateur web :

Etat HTTP 401 - Bad credentials

type Rapport d'état

message Bad credentials

description La requête nécessite une authentification HTTP (Bad credentials).

Spring Security

Authentification par formulaire

- **Déclarer l'authentification par formulaire**

Pour activer l'authentification par formulaire, il suffit de remplacer l'élément **<http-basic>** par l'élément **<form-login>** :

```
<http>
  <intercept-url pattern="/edit.do" access="ROLE_ADMIN" />
  <intercept-url pattern="/**" access="ROLE_USER" />

  <del>http-basic /> <form-login/>
</http>
```

Login with Username and Password

User:

Password:

Soumettre la requête

Réinitialiser

Quand **<form-login>** n'est pas paramétré, Spring Security génère une page de formulaire par défaut :

Spring Security

Authentification par formulaire

- Page de login personnalisée

la page login ne doit pas être filtrée par Spring Security.

```
<http pattern="/login.jsp*" security="none"/>
```

```
<http>
```

```
<intercept-url pattern="/edit.do" access="ROLE_ADMIN" />
```

```
<intercept-url pattern="/**" access="ROLE_USER" />
```

```
<form-login login-page="/login.jsp"/>  
</http>
```

spécifier la page de login Personnalisée

- Page de login personnalisée

Règle pour la page de login:

- **action** =>

j_spring_security_check

- **identifiant** => **j_username**

- **mot de passe** => **j_password**

```
<html>
```

```
<head>
```

```
<title>Login Page</title>
```

```
</head>
```

```
<body>
```

```
<form method="post" action="j_spring_security_check">
```

```
Identifiant :<input name="j_username" value="" type="text" /> <br>
```

```
Mot de passe :<input name="j_password" type="password" /> <br>
```

```
<input value="Valider" type="submit" />
```

```
</form>
```

```
</body>
```

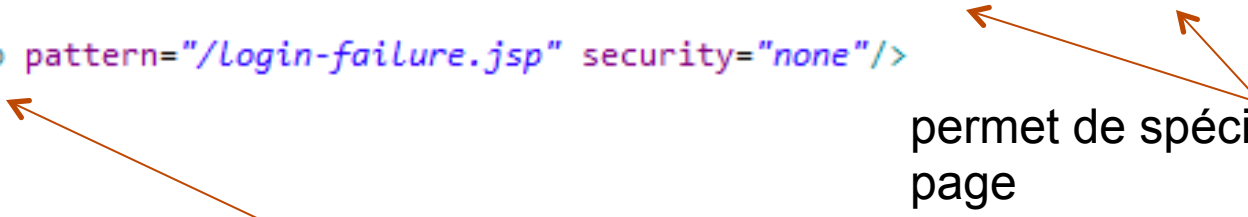
```
</html>
```

Spring Security

Authentification par formulaire

Page d'échec d'authentification

```
<form-login login-page="/login.jsp" authentication-failure-url="/login-failure.jsp"/>  
<http pattern="/login-failure.jsp" security="none"/>
```



déclarer une règle d'interception
afin de ne pas filtrer la page
d'erreur

permet de spécifier une
page
indiquant un échec de
l'authentification

Page d'échec d'authentification

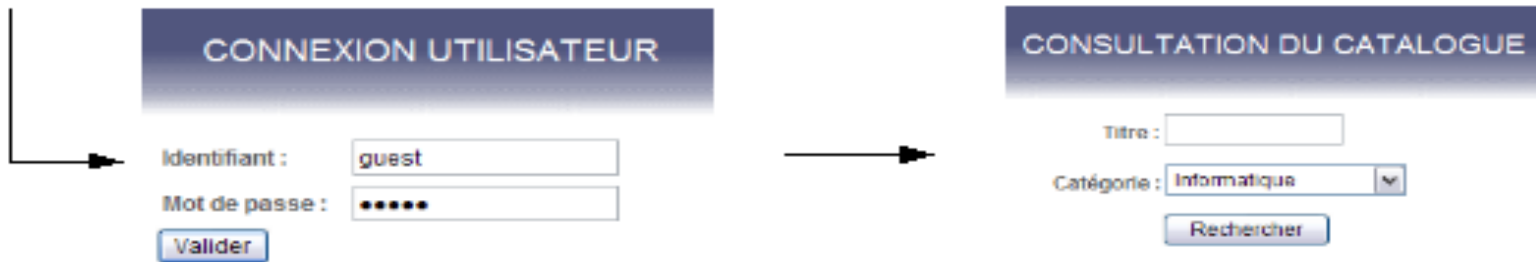
- Quand l'utilisateur saisit un identifiant ou un mot de passe incorrect, la page login-failure.jsp est alors affichée .

Spring Security

URL post login

- Quand une URL est sollicitée la page de login est affichée .
- Une fois l'authentification effectuée, Spring Security renvoie la page initialement demandée :

<http://server/app/catalogue.do>



- Si la première page demandée par l'utilisateur est la page de login elle-même, il faut la page post login à l'aide de l'attribut default-target-url :

```
<form-login login-page="/login.jsp"  
  authentication-failure-url="/login-failure.jsp"  
  default-target-url="/accueil.jsp"/>  
...
```

Spring Security

Page indiquant un accès non-autorisé

- Quand un utilisateur authentifié tente d'accéder à une ressource non-autorisée, Spring Security renvoie par défaut une erreur HTTP 403.
- A la place, on peut demander l'affichage d'une page d'erreur personnalisée à l'aide de l'attribut access-denied-page :

```
<http access-denied-page="/denied.jsp">  
  ...  
</http>
```

<http://server/app/editerproduit.do>

ROLE_USER



ACCÈS REFUSÉ

Vous n'êtes pas autorisé à consulter cette page

Spring Security

Gestion du logout

- Le logout est piloté par l'invocation de l'URL `/j_spring_security_logout`.
- Il provoque l'**invalidation de la session HTTP** et la redirection **vers la page racine de l'application**.
- Pour l'activer, il suffit d'ajouter l'élément **<logout>** dans la configuration Spring :

```
<http access-denied-page="/denied.jsp">
    ...
    <logout/>
</http>
```

- Une JSP fournit le lien de déconnexion :

```
<a href="/j_spring_security_logout">Déconnexion</a>
```

Spring Security

Authentification anonyme

- L'authentification anonyme permet de créer un utilisateur sans identification explicite.
- Cette authentification automatique se produit au premier accès HTTP sur l'application.
- Les caractéristiques de cet utilisateur sont les suivantes :
 - Identifiant : *anonymousUser*
 - Rôle : *ROLE_ANONYMOUS*
- On peut ainsi donner l'accès à certaines ressources à tout utilisateur non authentifié :
 - Page d'accueil
 - Page de login
 - Images
 - Fichiers CSS et JavaScript
 - ...

Spring Security

- Pour activer ce mode d'authentification, on utilise l'élément **<anonymous/>** :

```
<http access-denied-page="/denied.jsp">  
    ...  
    <anonymous/>  
</http>
```

- Une fois l'utilisateur authentifié explicitement, *anonymousUser* est remplacé dans le *security context* par ce nouvel utilisateur.
- Il convient donc d'affecter le rôle anonyme à tous les utilisateurs pour ne pas filtrer l'accès aux ressources accessibles au rôle anonyme :

```
<authentication-manager>  
  <authentication-provider>  
    <user-service>  
      <user name="jimi" password="jimi" authorities="ROLE_USER, ROLE_ADMIN, ROLE_ANONYMOUS" />  
      <user name="bob" password="bob" authorities="ROLE_USER, ROLE_ANONYMOUS" />  
    </user-service>  
  </authentication-provider>  
</authentication-manager>
```


Spring Security

Provider JDBC

- Les informations d'authentification et d'habilitation peuvent être stockées en base de données :

```
<security:authentication-provider>  
  <security:jdbc-user-service data-source-ref="dataSource" />  
</security:authentication-provider>
```

- L'attribut **data-source-ref** permettant de référencer un bean Spring de type **DataSource**.
- Quand on déclare un service de type **jdbc-user-service**, Spring utilise une classe DAO qui respecte le schéma SQL suivant :

Spring Security

Provider JDBC

```
CREATE TABLE users (  
  username VARCHAR(50) NOT NULL PRIMARY KEY,  
  password VARCHAR(50) NOT NULL,  
  enabled BIT NOT NULL  
);
```

```
CREATE TABLE authorities (  
  username VARCHAR(50) NOT NULL,  
  authority VARCHAR(50) NOT NULL  
);
```

```
ALTER TABLE authorities ADD CONSTRAINT fk_authorities_users foreign key (username) REFERENCES users(username);
```

Spring Security

- Il est possible de personnaliser les requêtes SQL de ce service JDBC si les noms des tables et colonnes de la base de données "utilisateurs" sont différents du schéma ci-dessus :

```
<security:authentication-provider>
  <security:jdbc-user-service data-source-ref="dataSource"
    users-by-username-query="SELECT identifiant,motdepasse,actif FROM
                                utilisateurs WHERE identifiant = ?"
    authorities-by-username-query="SELECT identifiant,role FROM roles WHERE
                                    identifiant = ?"/>
</security:authentication-provider>
```

Ref:

http://www.jtips.info/index.php?title=Spring_Security

Spring Web Services RestFul

Les applications clientes utilisent des méthodes HTTP comme:

- GET,
- POST,
- PUT
- DELETE

pour manipuler la ressource ou la collecte des ressources

En règle générale une méthode:

GET est utilisée pour obtenir la liste ou la ressource ou la collecte des ressources,

POST est utilisée pour créer,

PUT est utilisé pour mettre à jour ou remplacer, et

DELETE est utilisé pour supprimer la ressource.

Spring Web Services RestFul

Par exemple, GET `http: // hôte /contexte/employees /12345` devient la représentation de l'employé avec l'ID 12345.

La représentation de réponse pourrait être un XML ou JSON qui contient les informations des employés détaillée, ou il pourrait être un JSP / HTML page qui donne une meilleure interface.

Dont la représentation, vous verrez dépend de la mise en œuvre côté serveur et le type MIME votre demande de clients.

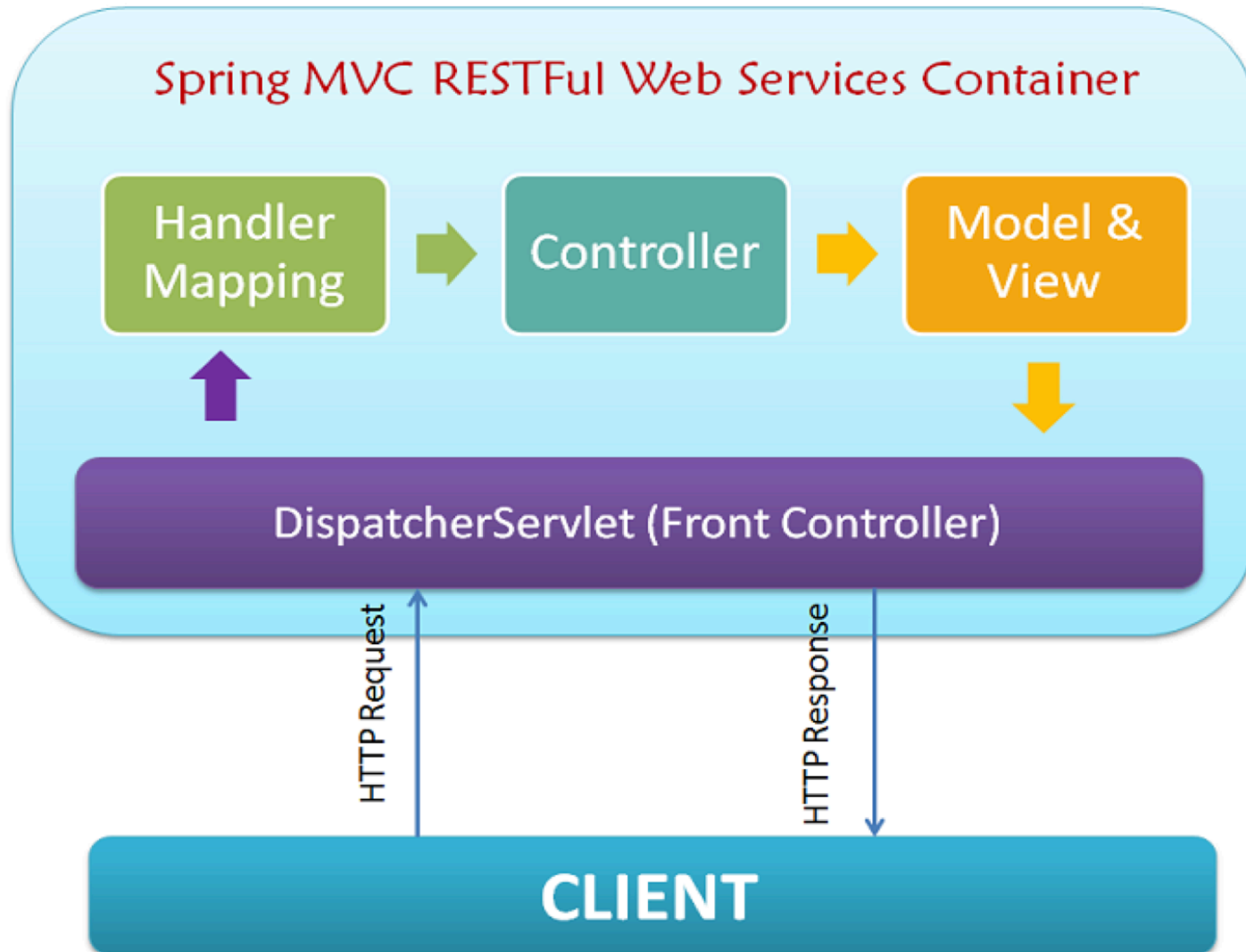
Spring Web Services RestFul

Un service Web RESTful est un service web implémentant le protocole HTTP et les principes de REST.

En règle générale, un service Web RESTful définira la ressource de base URI, les types MIME représentation / réponse qu'elle obtient, et les opérations qu'il peut réaliser

Spring Web Services RestFul

ARCHITECTURE



Spring Web Services RestFul

1-Définir le contrôleur par défaut à savoir le dispatcherServlet

Listing 11 Enable Spring Web application context in web.xml

```
<context-param>
<param-name>contextConfigLocation</param-name>
<param-value>
    /WEB-INF/rest-context.xml
</param-value>
</context-param>

<!-- This listener will load other application context file in addition to
    rest-servlet.xml -->
<listener>
<listener-class>
    org.springframework.web.context.ContextLoaderListener
</listener-class>
</listener>

<servlet>
<servlet-name>rest</servlet-name>
<servlet-class>
    org.springframework.web.servlet.DispatcherServlet
</servlet-class>
<load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
<servlet-name>rest</servlet-name>
<url-pattern>/service/*</url-pattern>
</servlet-mapping>
```

Spring Web Services RestFul

3-Configurer sa méthode pour implémenter le service REST dans sa méthode

```
@Controller
public class EmployeeController {
    @RequestMapping(method=RequestMethod.GET, value="/employee/{id}")
    public ModelAndView getEmployee(@PathVariable String id) {
        Employee e = employeeDS.get(Long.parseLong(id));
        return new ModelAndView(XML_VIEW_NAME, "object", e);
    }
}
```

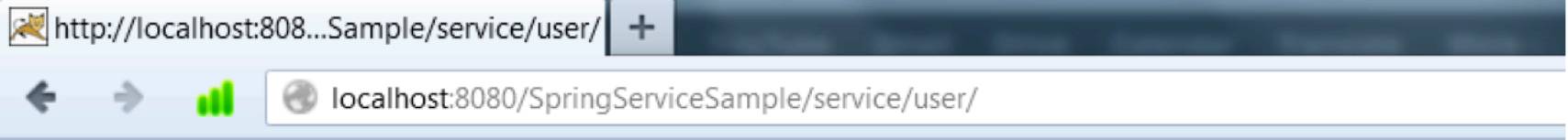
Nous avons ici une méthode avec une partie variable de l'ID, pour rechercher le bon employé.

Spring Web Services RestFul

Les différentes façons d'implémenter le protocole

```
@RequestMapping(method=RequestMethod.POST, value="/employee")  
public ModelAndView addEmployee(@RequestBody String body) {  
    Source source = new StreamSource(new StringReader(body));  
    Employee e = (Employee) jaxb2Marshaller.unmarshal(source);  
    employeeDS.add(e);  
    returnnew ModelAndView(XML_VIEW_NAME, "object", e);  
}  
  
@RequestMapping(method=RequestMethod.PUT, value="/employee/{id}")  
public ModelAndView updateEmployee(@RequestBody String body) {  
    Source source = new StreamSource(new StringReader(body));  
    Employee e = (Employee) jaxb2Marshaller.unmarshal(source);  
    employeeDS.update(e);  
    returnnew ModelAndView(XML_VIEW_NAME, "object", e);  
}  
  
@RequestMapping(method=RequestMethod.DELETE, value="/employee/{id}")  
public ModelAndView removeEmployee(@PathVariable String id) {  
    employeeDS.remove(Long.parseLong(id));  
    List<Employee> employees = employeeDS.getAll();  
    EmployeeList list = new EmployeeList(employees);  
    returnnew ModelAndView(XML_VIEW_NAME, "employees", list);  
}
```

Spring Web Services RestFul



```
[{"userid":2,"firstName":"Priya","lastName":"Balachandran","email":"abc@abc.com"},
{"userid":3,"firstName":"Anita","lastName":"Lidiya","email":"sfjsd@sdvk.com"},
{"userid":4,"firstName":"Jude","lastName":"Rao","email":"sifijfsdif@sdfjdgf.com"},
{"userid":5,"firstName":"John","lastName":"Edwin","email":"sdjf@sdfd.com"},
{"userid":6,"firstName":"Mariya","lastName":"Joseph","email":"abc@abc.com"},
{"userid":7,"firstName":"Sarah","lastName":"Abraham","email":"abc@def.com"},
{"userid":8,"firstName":"Carolyn","lastName":"Stoner","email":"asds@aedr.com"},
{"userid":9,"firstName":"Angeline","lastName":"Chelladurai","email":"asds@aedr.com"},
{"userid":10,"firstName":"Meena","lastName":"Muthu","email":"asds@aedr.com"},
{"userid":11,"firstName":"Mariya","lastName":"Magdaline","email":"asds@aedr.com"},
{"userid":12,"firstName":"Isaac","lastName":"Newton","email":"asds@aedr.com"},
{"userid":13,"firstName":"Charles","lastName":"Babbage","email":"asds@aedr.com"},
{"userid":14,"firstName":"Indhira","lastName":"Priyadarshini","email":"asds@aedr.com"},
{"userid":15,"firstName":"Goutham","lastName":"Kumar","email":"asds@aedr.com"},
{"userid":16,"firstName":"Sruthi","lastName":"Rao","email":"asds@aedr.com"}]
```