
Anciens éléments de Spring

*(anciennes syntaxes ,
anciennes technologies)*

Table des matières

I - Config Spring en XML (avant 2015).....	6
1. Configuration xml de Spring.....	6
1.1. Fichier(s) de configuration.....	6
1.2. Configuration des composants "spring" et des injections de dépendances.....	7
1.3. Instanciation de composant Spring via une Fabrique.....	7
1.4. ApplicationContext et test unitaires.....	8
1.5. scope (singleton/prototype/...) pour Stateless/Stateful.....	9
1.6. Organisation des fichiers de configurations "Spring"	10

1.7. Utilisation d'un fichier ".properties" annexe.....	10
2. Configuration IOC Spring via des annotations.....	12
2.1. Configuration xml pour "xml + annotations".....	12
2.2. Annotations (stéréotypées) pour composant applicatif.....	12
2.3. Autres annotations ioc (@Required , @Autowired , @Qualifier).....	13
2.4. Paramétrage XML ou Java de ce qui existe au sens "Spring".....	14
3. Tests "JUnit4 + Spring".....	16
4. Paramétrages Spring quelquefois utiles.....	18
4.1. Compatibilité avec singleton déjà programmé en java.....	18
4.2. Réutilisation (rare) d'une petite fabrique existante:.....	18
4.3. méthodes associées au cycle de vie d'un "bean" spring.....	18
4.3.a. Via annotations @PostConstruct et @PreDestroy.....	18
4.3.b. Via configuration 100% xml.....	19
4.4. Autres possibilités de Spring.....	19

II - Anciennes configurations Spring..... 19

1. DataSource JDBC (vue Spring).....	19
1.1. Mise en oeuvre "JNDI" d'une source de données JDBC.....	19
1.2. Mise en oeuvre basique d'une source de donnée autonome.....	20
2. DAO Spring basé sur Hibernate 3 ou 4.....	21
2.1. Anciennes versions (de Spring et Hibernate).....	21
2.2. Dao utilisant "sessionFactory et session" en mode @Transactional plutôt que HibernateTemplate.....	22
3. Mise en oeuvre des transactions avec Spring 1.2.....	23
3.1. Enveloppe transactionnelle.....	23
4. Démarrages possibles depuis spring 2.5.....	25
5. Injection de Spring au sein d'un framework WEB.....	25
5.1. WebApplicationContext (configuration xml).....	25
5.2. WebApplicationContext (accès et utilisation).....	26
6. Injection "Spring" au sein du framework JSF.....	27
7. Intégration de JSF 2 au sein de Spring-boot 2.....	29
8. Injection "Spring" au sein du framework STRUTS.....	33
9. Injection "Spring" au sein du framework STRUTS2.....	35
10. Spring + CXF (pour Web Services SOAP).....	37
10.1. Présentation de CXF (apache).....	37
10.2. implémentation avec CXF (et Spring) au sein d'une application Web (pour Tomcat 5.5, 6 ou 7).....	38
10.3. Client JAX-WS sans wsimport avec CXF et Spring.....	39
11. Spring (proxy / business_delegate).....	41
11.1. implémentation d'un service RMI basé sur un POJO.....	42
11.2. proxy automatique "Spring" pour objet distant RMI.....	42

III - Anciennes configurations / Spring security.....43

1. Extension Spring-security (old versions).....	43
1.1. Ancien mode de configuration (xml).....	43
1.1.a. Filtre web pour spring-security à déclarer dans web.xml.....	44
1.1.b. Exemple de configuration spring pour spring-security:.....	44
1.2. Champ caché "_csrf" de spring-mvc utile pour pages/vues "java/jsp" mais inutile pour Api-REST avec tokens	47
1.3. Configuration un peu plus moderne de spring-security en java.....	48
1.4. Vue d'ensemble sur "Spring-security".....	49
1.5. Authentification jdbc ("realm" en base de données).....	51
1.6. Authentification "personnalisée" en implémentant l'interface UserDetailsService...54	
1.7. Configuration type pour un projet de type Thymeleaf ou JSP.....	56
1.8. Configuration type pour un projet de type "Api REST".....	57

IV - Jta/Atomikos et spring JMS.....58

1. JTA / Atomikos.....	58
1.1. Cadre général des transactions distribuées.....	58
1.2. JTA.....	58
2. JTA/Atomikos intégré dans Spring et Spring-Boot.....	58
2.1. Configuration explicite "java-config-jta".....	58
2.2. Configuration en mode test/H2.....	63
2.3. Configuration en mode prod/Mysql & postgres.....	64
2.4. Exemple de service transactionnel avec JTA.....	65
3. Repères JMS.....	69
3.1. ActiveMq.....	71
3.2. Artemis (nouvelle génération de ActiveMq).....	72
4. intégration JMS dans Spring.....	73
4.1. Configuration "JMS avec Spring-boot".....	73
4.2. Application java externe qui envoie des messages.....	77

V - Divers, Spring web flow, Spring intégration.....79

1. Divers aspects secondaires ou avancés de Spring.....	79
1.1. Plug eclipse "Spring IDE" et STS (Spring Tools Suite).....	79
1.2. Détails sur Autowired.....	79
1.3. Bean (config spring) abstrait et héritage.....	79
1.4. Internationalisation gérée par Spring (MessageSource).....	80
1.5. Profiles "spring".....	82
1.6. Divers aspects avancés (cas pointus).....	82
2. Spring – diverses extensions.....	83
3. OpenSessionInViewFilter pour lazy="true"	84
3.1. Version Spring/Hibernate.....	86
3.2. Version Spring/JPA.....	86

4. Dozer (copy properties with xml mapping).....	87
4.1. Installation de "dozer".....	87
4.2. Initialisation/utilisation java.....	87
4.3. Copies avec paramétrage xml.....	87
4.4. Eventuelle intégration au sein de Spring.....	89
5. Conversion générique Dto/Entity via Dozer.....	90
6. Configuration maven pour Spring 3.....	93
6.1. illustration concrète des modules spring via maven.....	93
7. Les concepts de AOP (vocabulaire).....	96
8. Les grands axes de la mise en oeuvre d' A.O.P.....	97
8.1. avantages et inconvénients selon les choix technologiques.....	97
8.2. solutions basées sur des pré-compilations (pré-processeur).....	98
8.3. solutions basées sur des mécanismes dynamiques lors de l'exécution.....	98
9. L'essentiel de Spring AOP.....	99
9.1. PointCut de "Spring AOP":.....	99
9.2. Advices / intercepteurs de "Spring AOP":.....	100
9.3. ProxyFactoryBean de Spring AOP :.....	102
9.4. "Auto proxy" de Spring AOP :.....	103
10. Présentation du framework "Spring Web Flow".....	104
11. Définition d'un Spring Web Flow.....	104
11.1. Eléments fondamentaux (Flow, viewState , transitions ,).....	105
11.2. Liste des portées (scope).....	105
11.3. Actions (evaluate , actionState , ...).....	106
11.4. SubFlow (avec input/output) et "onRender".....	106
11.5. Validations et messages.....	108
12. Intégration JSF dans Spring Web Flow.....	108
12.1. Configuration & arborescence nécessaire.....	108
12.2. Liens entre pages ".xhtml" et "flow + flowBean".....	113
12.3. Utilisation du DataModel JSF au sein de Spring WebFlow.....	113
12.4. Autres spécificités "JSF + Spring Web Flow".....	114
13. Spring-Integration (présentation).....	115
14. "channels".....	115
14.1. Interfaces java pour "Channels".....	116
14.2. Implémentations "channels".....	116
14.3. ChannelInterceptor.....	116
14.4. Configuration "spring" d'un "channel".....	117
15. "Channel Adapter" et "Bridge".....	118
15.1. InBound Channel Adapter (configuration).....	118
15.2. OutBound Channel Adapter (configuration).....	118
15.3. Messaging Bridge.....	119
16. messages (structure et construction).....	119
16.1. Structure d'un message (spring-integration).....	119

16.2. Implémentations et constructions des messages.....	120
17. Routage des messages.....	120
17.1. "Router".....	120
17.2. "Filter".....	120
17.3. "Aggregator".....	121
17.4. "Resequencer".....	121
17.5. "Message Handler Chain".....	121
18. Transformations des messages.....	121
18.1. "Transformer".....	121
18.2. Content-enricher.....	121
18.3. Claim-check.....	121
19. (Message) Endpoint.....	121
19.1. Message endpoints.....	121
19.2. InBound messaging gateways.....	121
19.3. ServiceActivator.....	122
19.4. Delayer.....	122
20. Gestion système (administration , supervision).....	122
20.1. JMX.....	122
20.2. Historique des messages.....	122
20.3. Control Bus.....	122
21. Chanel Adpater (avec principaux détails).....	122
21.1. File Adapter.....	122
21.2. FTP/FTPs Adapter.....	122
21.3. Http (Inbound & Outbound) Gateway.....	122
21.4. Mail Adapter (sending , receiveing).....	122
21.5. TCP Adapters.....	122
21.6. JDBC Adapters (inbound & outbounds).....	122
21.7. JMS Adapters.....	123
21.8. RMI Gateway.....	123
21.9. (java.io) Stream Adapters.....	123
21.10. Web Services Gateway.....	123
21.11. Xml, Xpath & Xslt Support (transformers).....	123
22. Aspects divers et avancés.....	123
22.1. Intercepteurs et AOP.....	124
22.2. Transactions.....	124
22.3. Sécurité.....	124
23. Exemple(s) complet(s).....	124

I - Config Spring en XML (avant 2015)

1. Configuration xml de Spring

1.1. Fichier(s) de configuration

La configuration de Spring est basée sur un (ou plusieurs) fichier(s) de configuration XML que l'on peut nommer comme on veut.. Depuis la version 2.0 de Spring il faut utiliser des entêtes xml basées sur des schémas "xsd" de façon à bénéficier de toutes les possibilités du framework.

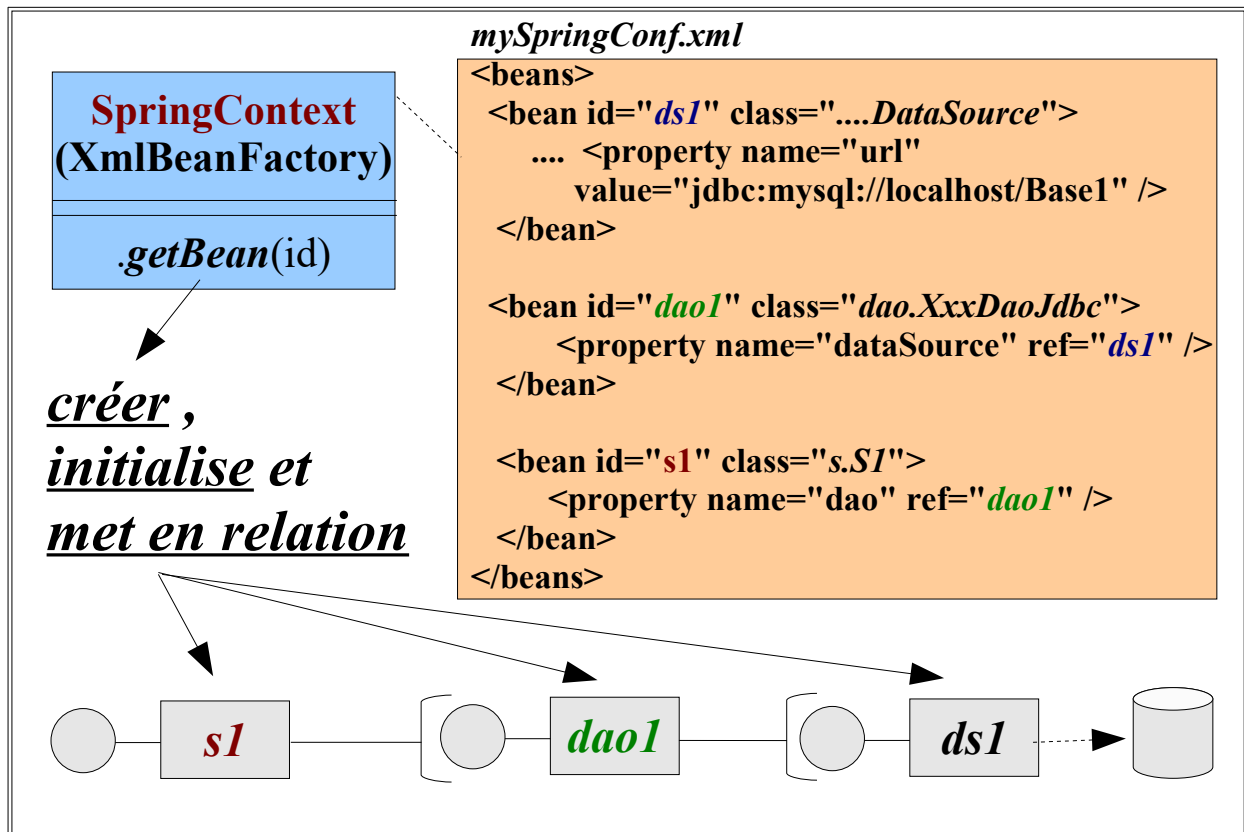
En fonction des réels besoins de l'application, l'entête du fichier de configuration Spring pourra comporter (ou pas) tout un tas d'éléments optionnels (AOP , Transactions , ...).

Exemple d'entête:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:aop="http://www.springframework.org/schema/aop"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xmlns:context="http://www.springframework.org/schema/context"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx.xsd
    http://www.springframework.org/schema/context
    http://www.springframework.org/schema/context/spring-context.xsd" >

  <bean ...../> <bean ...../>
  <tx:annotation-driven transaction-manager="txManager" />
  <context:annotation-config/>
  ...
</beans>
```

1.2. Configuration des composants "spring" et des injections de dépendances



NB1: bien que l'id d'un composant Spring puisse être *une chaîne de caractères quelconque (complètement libre)*, un plan d'ensemble sur les noms logiques (avec des conventions) est souvent indispensable pour s'y retrouver sur un gros projet.

Une solution élégante consiste à utiliser des identifiants proches des noms des classes des objets (ex : nom de classe en remplaçant la majuscule initiale par une minuscule)

NB2: la valeur de `class=""` doit correspondre au nom complet de la classe d'implémentation (avec le package en préfixe)

NB3: `<property name="xy" value="valeurPropriete" />` permet de fixer la valeur d'une propriété `xy` existante (appel automatique à `setXy()`).

NB4: `<property name="xy" ref="idBeanAinjecter" />` permet de paramétrer une injection de dépendance (appel automatique à `setXy()` ou l'argument en entrée correspondra à la référence mémoire vers le bean "spring" dont l'id vaut celui précisé par `ref="..."`).

1.3. Instanciation de composant Spring via une Fabrique

Le *paramètre d'entrée* de la méthode `getBean()` est l'*id du composant Spring* que l'on souhaite récupérer.


```
XmlBeanFactory bf = new XmlBeanFactory( new ClassPathResource("mySpringConf.xml"));
MyService s1 = (MyService) bf.getBean("myService");
```

Ceci pourrait constituer le point de départ d'une petite classe de test élémentaire.

Néanmoins, dans beaucoup de cas on préférera utiliser "ApplicationContext" qui est une version améliorée/sophistiquée de "BeanFactory" .

1.4. ApplicationContext et test unitaires

Un objet "**ApplicationContext**" est une sorte de "BeanFactory" évoluée apportant tout un tas de fonctionnalités supplémentaires:

- gestion des ressources (avec internationalisation) : (ex: MessageRessources , ...).
- gestion de AOP et des transactions.
- Instanciation de tous les composants nécessaires dès le démarrage et rangement de ceux-ci dans un contexte (plutôt qu'une instanciation tardive au fur et à mesure des besoins).

```
ApplicationContext contextSpring =
    new ClassPathXmlApplicationContext("mySpringConf.xml");
//BeanFactory bf = (BeanFactory) context;
MyService s1 = (MyService) contextSpring.getBean("idService");
//ou bien MyService s1 = contextSpring.getBean(MyService.class);
...
```

NB1: L'instanciation de l'objet "**ApplicationContext**" peut *si besoin* s'effectuer en précisant **plusieurs fichiers de configuration xml complémentaires**.
(Ex: myServiceSpringConf.xml + myDataSourceSpringConf.xml + myCxfWebServiceConf.xml).

NB2: Une instance de **ClassPathXmlApplicationContext** devrait idéalement être fermée (via un appel à **.close()**)

Attention (pour les performances):

L'initialisation du contexte Spring (effectuée généralement une fois pour toute au démarrage de l'application) est une opération longue:

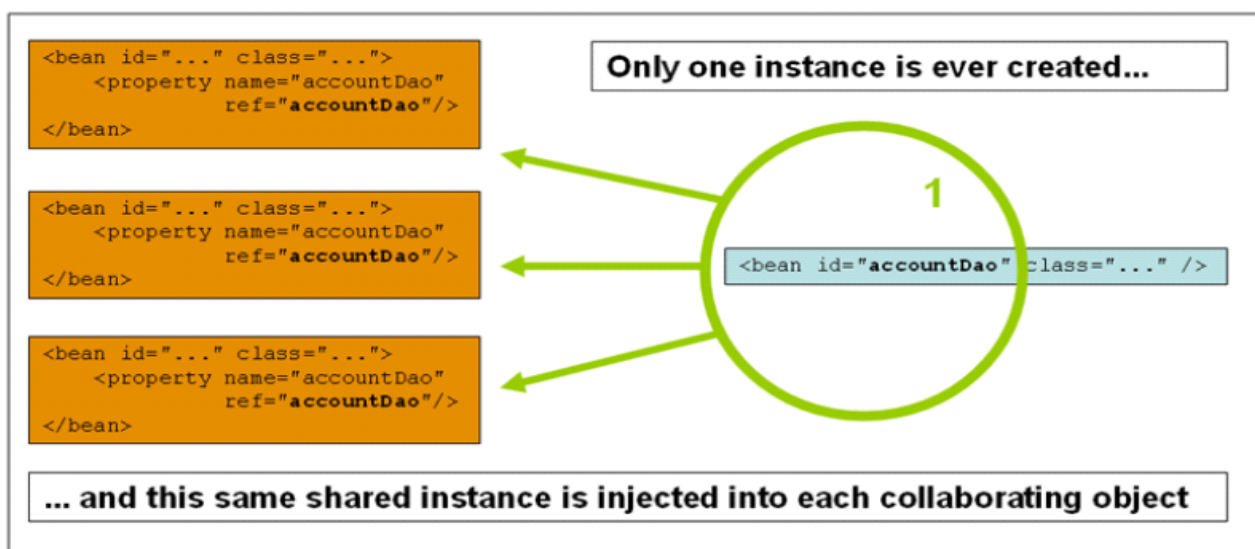
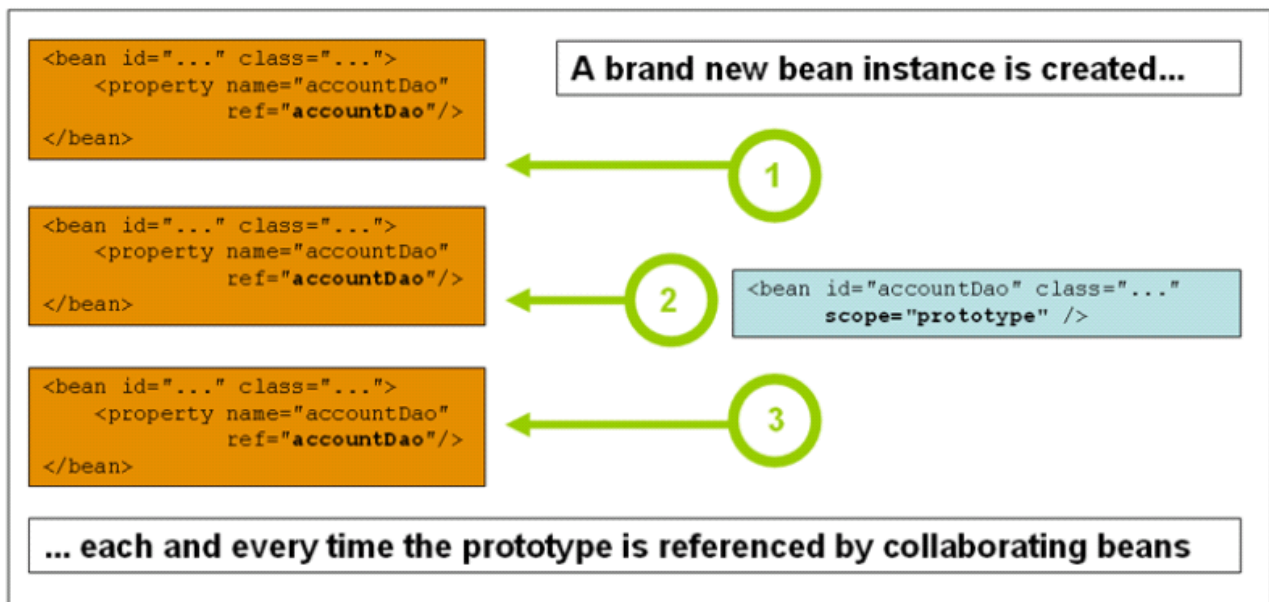
- analyse de toute la configuration Xml
- déclenchement des mécanismes AOP dynamiques
- instanciations des composants
- assemblage par injections de dépendances

Si plusieurs Tests unitaires (ex: JUnit) doivent être lancés dans la foulée , il faudra veiller à ne pas recréer inutilement un nouveau contexte Spring à chaque fois.

NB : en s'appuyant sur "spring-test" (**@RunWith(SpringJUnit4ClassRunner.class)** et **@ContextConfiguration(locations={"/mySpringConf.xml"})**) , il y aura une réutilisation automatique du contexte spring dans le cas où plein de tests unitaires sont basés sur le même fichier de configuration principal .

1.5. scope (singleton/prototype/...) pour Stateless/Stateful

portée (scope) d'un composant Spring	comportement / cycle de vie
singleton (par défaut)	un seul composant instancié et partagé au niveau de l'ensemble du conteneur léger Spring. (sémantique "Stateless / sans état")
prototype	une instance par utilisation (sémantique "Stateful / à état")
session	une instance rattachée à chaque session Http (valable uniquement au sein d'un "web-aware ApplicationContext")
request	une instance rattachée à une requête Http (valable uniquement au sein d'un "web-aware ApplicationContext")
global session	(global session) pour "portlet" par exemple [web uniquement]



1.6. Organisation des fichiers de configurations "Spring"

Un fichier de configuration Spring peut inclure des sous fichiers via la balise xml **"import"**.

La valeur de l'attribut **"resource"** de la balise import doit correspondre à un chemin relatif menant au sous fichier de configuration.

Dans le cas particulier ou la valeur de l'attribut **"resource"** commence par **"classpath:"** le chemin indiqué sera alors recherché en relatif par rapport à l'intégralité de tout le classpath (tous les ".jar")

Exemples :

applicationContext.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans .... >
  <import resource="dataSourceSpringConf.xml" />
  <import resource="serviceSpringConf.xml" />
  <import resource="webServiceEndPointSpringConf.xml" />
</beans>
```

```
<import resource="classpath:META-INF/cxf/cxf.xml"/>
```

Rappels :

Spring n'impose pas de nom sur le fichier de configuration principal (celui-ci est simplement référencé par une classe de test ou bien web.xml).

Ceci dit, les noms les plus classiques sont **"beans.xml"** , **"applicationContext.xml"** , **"context.xml"** .

Etant par défaut recherchés à la racine du "classpath" , les fichiers de configuration "spring" doivent généralement être placés dans **"src"** ou bien **"src/main/resources"** dans le cas d'un projet **"maven"** .

1.7. Utilisation d'un fichier ".properties" annexe

database.properties

```
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/mydatabase
jdbc.username=root
jdbc.password=password
```

dataSourceSpringConf.xml

```
...
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
  <property name="location" value="database.properties" />
</bean>
<bean id="dataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
```

```
<property name="driverClassName" value="${jdbc.driverClassName}" />
<property name="url" value="${jdbc.url}" />
<property name="username" value="${jdbc.username}" />
<property name="password" value="${jdbc.password}" />

</bean>
```

...

2. Configuration IOC Spring via des annotations

Depuis la version 2.5 de Spring, il est possible d'utiliser une configuration IOC paramétrée par des annotations directement insérées dans le code java à la place d'une configuration entièrement XML.

Pour cela , Spring peut utiliser des annotations dans un ou plusieurs des groupes suivants :

* standard Java EE>=5 (**@Resource** , ...)

* spécifiques Spring (**@Component** , **@Service** , **@Repository** , **@Autowired** , ...)

* IOC JEE6 [depuis Spring 3 seulement] (**@Named** , **@Inject** , ...)

NB : en interne Spring ne fait qu'interpréter @Inject comme un équivalent de @Autowired et @Named comme un équivalent de @Component

Une configuration mixte (XML + annotations) ou bien (JavaConfig + annotations) est tout à fait possible et il est également possible d'utiliser le mode "autowire" dans tous les cas de figures (annotations , XML , javaConfig, mixte).

2.1. Configuration xml pour "xml + annotations"

```
<?xml version="1.0" encoding="UTF-8"?>
<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.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

  <context:annotation-config/> <!-- pour demander à Spring de tenir compte de @Component, .... -->

  <context:component-scan base-package="tp"/>
    <!-- pour indiquer à Spring quelles sont les classes à scanner pour trouver des annotations
    telles que @Component , @Service , @Named , @Autowired , @Inject ou .... -->

</beans>
```

2.2. Annotations (stéréotypées) pour composant applicatif

exemple : XYDaoImplAnot.java

```
package tp.persistance.with_annot;

import org.springframework.stereotype.Repository;

import tp.domain.XY;
import tp.persistance.XYDao;
```

```

@Component("myXyDao")
public class XYDaoImplAnot implements XYDao {

    public XY getXYByNum(long num) {
        XY xy = new XY();
        xy.setNum(num);
        xy.setLabel("?? simu ??");
        return xy;
    }
}

```

dans cet exemple , l'annotation `@Component()` marque (ou stéréotype) la classe Java comme étant celle d'un **composant pris en charge par Spring** . D'autre part, la valeur facultative "myXyDao" correspond à l'ID qui lui est affecté. (*l'id par défaut est le nom de la classe avec une minuscule sur la première lettre*).

NB: Les stéréotypes `@Repository` , `@Service` et `@Controller` (qui héritent tous les 3 de `@Component`) sont avant tout destinés à marquer le type des composants dans une architecture n-tiers. Ceci permet alors d'automatiser certains traitements en tenant compte de ces stéréotypes que l'on peut découvrir/filtrer par introspection .

On peut éventuellement utiliser ces annotations pour **renseigner l'id précis** d'un composant Spring.

@Component	Composant spring quelconque
@Repository	Composant d'accès aux données (DAO)
@Service	Service métier (alias business service) avec transactions
@Controller	Composant de contrôle IHM (coordinateur, ...)
@RestController	Composant de contrôleur de Web Service REST

2.3. Autres annotations ioc (@Required , @Autowired , @Qualifier)

@Required (à placer au dessus d'une méthode d'injection ou d'une propriété privée)	Pour vérifier dès le début (initialisation du contexte Spring et ses composants) qu'une injection a bien été effectuée . Si la valeur de la référence est restée à null --> exception dès l'initialisation plutôt qu'en cours d'exécution du programme.
@Autowired	Pour demander une auto-liaison par type (injections de dépendances automatiques et implicites en fonction des correspondances de type).
@Qualifier	Permet de marquer une injection Spring avec un qualificatif / nom de variante (ex: "test" ou "prod" ou ...) dans le but de paramétrer plus finement les auto-liaisons (éventuel filtrage selon le qualificatif attendu)

Exemple (assez conseillé) avec @Autowired

```

@Service() //id par défaut = serviceXYAnot
public class ServiceXYAnot implements IServiceXY {

    private XYDao xyDao;
}

```

```
//injectera automatiquement l'unique composant Spring
//dont le type est compatible avec l'interface précisée.
@Autowired //ici ou bien au dessus du "private ..."
public void setXyDao(XYDao xyDao) {
    this.xyDao = xyDao;
}

public XY getXyByNum(long num) {
    return xyDao.getXyByNum(num);
}
```

ou bien plus simplement :

```
@Service //id par défaut = serviceXYAnot
public class ServiceXYAnot implements IServiceXY {

    @Autowired
    private XYDao xyDao;

    public XY getXyByNum(long num) {
        return xyDao.getXyByNum(num);
    }
}
```

NB :

Si plusieurs classes d'implémentation de l'interface "Payment" existent avec des **@Qualifier("byCreditCard")** et **@Qualifier("byCash")** en plus de **@Component()**

alors une syntaxe de type **@Autowired @Qualifier("byCreditCard")**

private Payment paiementParCarteDeCredit ;

ou bien

@Autowired @Qualifier("byCash")

private Payment paiementEnLiquide ;

permettra d'effectuer une injection de la version voulue .

2.4. Paramétrage XML ou Java de ce qui existe au sens "Spring"

Ceci permettra de contrôler astucieusement ce qui sera injecté via **@Autowired** (ou **@Inject**)

En organisant bien les packages java de la façon suivante :

xxx.itf.dao.DaoXY (interface)

xxx.impl.dao.v1.DaoXYImpl1 (classe d'implémentation du Dao en version 1 avec **@Component**)

xxx.impl.dao.v2.DaoXYImpl2 (classe d'implémentation du Dao en version 2 avec **@Component**)

on peut ensuite paramétrer alternativement une configuration XML Spring de l'une des 2 façons suivantes :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ....>
  <context:annotation-config/>
  <context:component-scan base-package="xxx.yyy"/>
  <context:component-scan base-package="xxx.impl.dao.v1"/>
</beans>
```

ou bien

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
  ....
  <context:component-scan base-package="xxx.yyy"/>
  <context:component-scan base-package="xxx.impl.dao.v2"/>
</beans>
```

ceci fait que une seule des deux versions (v1 ou v2) est prise en charge par Spring .

Il n'y a alors plus d'ambiguïté au niveau de

```
@Autowired //ou @Inject
private DaoXY xyDao ;
```

NB : **component-scan** (en xml) comporte plein de variantes syntaxiques (**include** , **exclude** , ...)

3. Tests "JUnit4 + Spring"

Depuis la version 2.5 de Spring, existent de nouvelles annotations permettant d'initialiser simplement et efficacement une classe de Test JUnit 4 avec un contexte (configuration) Spring.

Attention: pour éviter tout problème d'incompatibilité entre versions, il est souhaitable d'utiliser une version très récente de "jUnit4.x.jar" de JUnit4 (ex: 4.x) et Spring.

NB :

Les classes de Test annotées via **@RunWith(SpringJUnit4ClassRunner.class)** peuvent utiliser en interne **@Autowired** ou **@Inject** même si elles ne sont pas placées dans un package référencé par

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

Exemple de classe de Test de Service (avec annotations)

```
...
import org.junit.Assert;    import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.transaction.TransactionConfiguration;
import org.springframework.transaction.annotation.Transactional;

// nécessite spring-test.jar et junit>=4.8.jar dans le classpath
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from "/mySpringConf.xml" in the root of the classpath
@ContextConfiguration(locations={"/mySpringConf.xml"})
public class TestGestionComptes {

    @Autowired
    private GestionComptes service = null;

    @Test
    public void testTransférer(){
        Assert.assertTrue( ... );
    }

}
```

NB :

Un **Dao** est normalement utilisé par un service métier dont les méthodes sont transactionnelles. Pour qu'une classe de **Test de dao** soit au plus près de la réalité, elle doit se comporter comme un service métier et doit gérer les transactions (via les automatismes de Spring).

Via les annotations

@TransactionConfiguration(transactionManager="txManager",defaultRollback=false)

et

@Transactional()

la *classe de test de dao* peut gérer convenablement les transactions Spring (et indirectement résoudre les problèmes de "lazy initialisation exception").

Exemple de classe de Test de Dao (avec annotations)

```
...
import org.junit.Assert;    import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.test.context.transaction.TransactionConfiguration;
import org.springframework.transaction.annotation.Transactional;

// nécessite spring-test.jar et junit4.8.1.jar dans le classpath
@RunWith(SpringJUnit4ClassRunner.class)
// ApplicationContext will be loaded from "/mySpringConf.xml" in the root of the classpath
@ContextConfiguration(locations={"/mySpringConf.xml"})
@Transactional(transactionManager="txManager",defaultRollback=false)
public class TestDaoXY {

    // injection du doa à tester contrôlée par @Autowired (par type)
    @Autowired
    private DaoXY xyDao = null;

    @Test
    @Transactional(readOnly=true)
    public void testGetComptesOfClient(){
        ...
        Assert.assertTrue( ... );
    }

    ...
}
```

Attention : il ne vaut mieux pas placer de **@TransactionalConfiguration** ni de **@Transactional** sur une classe testant un service métier car cela pourrait fausser les comportements des tests.

4. Paramétrages Spring quelquefois utiles

4.1. Compatibilité avec singleton déjà programmé en java

Eventuelle instanciation d'un composant Spring via une méthode de fabrique "static":

```
....
<bean id="exampleBean"
      class="examples.ExampleBean2"
      factory-method="createInstance"/>
...
```

4.2. Réutilisation (rare) d'une petite fabrique existante:

```
<!-- the factory bean, which contains a method called createInstance() -->
<bean id="myFactoryBean" class="...">
...
</bean>
<!-- the bean to be created via the factory bean -->
<bean id="exampleBean"
      factory-bean="myFactoryBean"
      factory-method="createInstance"/>
```

4.3. méthodes associées au cycle de vie d'un "bean" spring

4.3.a. Via annotations `@PostConstruct` et `@PreDestroy`

```
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class XxxService
{
    String message; //+get/setMessage()

    @PostConstruct
    public void initBean() {
        System.out.println("Init method after properties are set : "
                           + message);
    }

    @PreDestroy
    public void cleanUp() {
        System.out.println("cleanUp before end of Spring");
    }
}
```

NB: Spring ne prend en compte les annotations `@PostConstruct` et `@PreDestroy` que si le pré-processeur `'CommonAnnotationBeanPostProcessor'` a été enregistré dans le fichier de configuration spring ou bien si `'<context:annotation-config />'` a été configuré pour prendre en charge plein d'annotations.

```
<bean class="org.springframework.context.annotation.CommonAnnotationBeanPostProcessor" />
```

4.3.b. Via configuration 100% xml

```
...
<bean id="xxxService" class="ppp.XxxService"
      init-method="initBean" destroy-method="cleanUp">
  <property name="message" value="message in the bottle" />
</bean>
```

4.4. Autres possibilités de Spring

- injection via constructeur
- lazy instanciation (initialisation retardée à l'utilisation)

==> voir documentation de référence (chapitre "The IOC Container")

II - Anciennes configurations Spring

1. DataSource JDBC (vue Spring)

Certaines parties d'une application JEE ont souvent besoin d'être testées au dehors du serveur d'application . Un pool de connexion associé à un accès JNDI n'est cependant généralement accessible qu'au sein d'un serveur d'application J2EE (ex: JBoss, WebSphere , Tomcat , ...).

Spring offre heureusement la possibilité d'injecter une source de données via le mécanisme IOC . L'utilisation de la source de donnée est toujours la même (vision abstraite , nom logique). La mise en oeuvre peut être très variable et se (re)configure très rapidement (switch rapide).

1.1. Mise en oeuvre "JNDI" d'une source de données JDBC

NB :

- Ceci ne fonctionne qu'avec un certain serveur d'application JEE (tomcat ou jboss ou)
- Cette mise en oeuvre est de plus en plus **rare** avec les versions récentes de spring . La philosophie "spring moderne" consiste de plus en plus à ne plus s'appuyer sur les spécificités d'un serveur d'application JEE .

mySpringDataSource_JNDI.xml

```
...<beans ...>
<!-- DataSource nécessitant une config. coté servApp (Pool de connexions avec nom JNDI) -->
<!-- JNDI=java:/BankDBDataSource pour JBoss -->
```

```

<!-- JNDI=java:comp/env/jdbc/BankDBDataSource pour Tomcat -->
<bean id="myDataSource"
  class="org.springframework.jndi.JndiObjectFactoryBean"
  destroy-method="close">
  <property name="jndiName" value="java:/BankDBDataSource" />
  <property name="expectedType" value="javax.sql.DataSource" />
</bean>
</beans>

```

Cette configuration permet de configurer une source de données "Spring" injectable dont l'implémentation repose sur un véritable pool de connexions géré par un serveur d'application JEE (ex: WebSphere, Tomcat, JBoss, ...)

L'élément fondamental de la configuration "Spring" est la propriété *jndiName* de la classe *org.springframework.jndi.JndiObjectFactoryBean* ==> la valeur doit correspondre au nom JNDI d'une ressource à mettre en place au niveau du serveur d'application.

Exemple de configuration sous Tomcat >= 5.5:

conf/Catalina/localhost/bankWeb.xml ou bien dans **conf/server.xml**

```

<?xml version='1.0' encoding='utf-8'?>
<Context className="org.apache.catalina.core.StandardContext"
  docBase="bankWeb" path="/bankWeb" privileged="false" reloadable="true" >
  <Resource name="jdbc/BankDBDataSource" auth="Container"
    type="javax.sql.DataSource"
    username="mydbuser" password="mypwd"
    driverClassName="com.mysql.jdbc.Driver"
    url="jdbc:mysql://localhost/minibankdb"/>
</Context>

```

1.2. Mise en oeuvre basique d'une source de donnée autonome

mySpringDataSource_Simple.xml

```

...
<beans ...>
<!-- DataSource en version embarquée (sans JNDI ni aucune config. coté servApp) -->

<!-- <bean id="myDataSource" class="org.apache.commons.dbcp.BasicDataSource"
  destroy-method="close"> -->
<bean id="myDataSource"
  class="org.springframework.jdbc.datasource.DriverManagerDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver" />
  <property name="url" value="jdbc:mysql://localhost/minibankdb" />
  <property name="username" value="mydbuser" />
  <property name="password" value="mypwd" />
</bean>
</beans>

```

Remarques:

La classe "***org.springframework.jdbc.datasource.DriverManagerDataSource***" est une version basique (sans pool de connexions recyclables , juste pour les tests) et qui a l'avantage de ne pas nécessiter de ".jar" supplémentaire.

Seules choses à bien mettre en place (dans le ClassPath) :

- le ".jar" contenant le code du **driver JDBC** pour "MySQL" ou "Oracle" ou "..." (ex: *mysql-connector-java-.....jar*)
- spring-jdbc

NB1 : Une version "**java config**" (pas xml) **équivalente** est située (en exemple) dès le début du paragraphe "java-config" du chapitre "config ioc (xml, annotations, java-config)" .

NB2 : Dans un contexte "spring-boot" + "@EnableAutoconfiguration" , il suffit de paramétrer le fichier de configuration principal **application.properties** :

```
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:~/mydb
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.platform=h2
```

2. DAO Spring basé sur Hibernate 3 ou 4

2.1. Anciennes versions (de Spring et Hibernate)

A l'époque de Spring 1.x et Hibernate 3.x , les classes *HibernateDaoSupport* et *HibernateTemplate* avaient été créées au niveau du framework Spring pour intégrer la technologie hibernate.

Depuis, les 2 frameworks ont beaucoup évolués :

- Hibernate existe maintenant en version 3.5 et 4
- Spring gère plus subtilement les transactions (via @Transactional) depuis la version 2 ou 2.5

Les classes *HibernateDaoSupport* et *HibernateTemplate* sont maintenant (en 2013) à considérées comme anciennes et un peu obsolètes (et sont maintenant déplacées dans une annexe du support de cours).

Remarque importante : **HibernateTemplate** fonctionne encore très bien avec Hibernate 3.x mais **ne fonctionne pas avec Hibernate 4** .

L'intégration actuelle conseillée de Hibernate dans Spring est celle qui va être présentée dans le paragraphe suivant et qui fonctionne aussi bien avec Hibernate 3 que Hibernate 4.

2.2. Dao utilisant "sessionFactory et session" en mode @Transactional plutôt que HibernateTemplate

```
public class XYDaoHibernate implements XYDao{

    protected SessionFactory sessionFactory=null;

    public void setSessionFactory(SessionFactory sf) { //injection de dépendance
        sessionFactory = sf;
    }

    public Session getCurrentSession(){ return sessionFactory.getCurrentSession();
    }

    @Transactional
    public void removeEntity(Object e) {
        getCurrentSession().delete(e);
    }

    @Transactional
    public XY updateEntity(XY e) {
        getCurrentSession().update(e);    return e;
    }

    @Transactional
    public XY getEntityById(ID pk) {
        return (XY) getCurrentSession().get(XY.class, pk);
    }

    @Transactional
    public XY persistNewEntity(XY e) {
        getCurrentSession().persist(e);    //getCurrentSession().save(e);
        return e;
    }
}
```

Schéma d'injections IOC:

```
... <bean id="mySessionFactory"
class="org.springframework.orm.hibernate4.LocalSessionFactoryBean"> <!-- ou bien
org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean -->
    <property name="dataSource" ref="myDataSource" />
    <!-- pour ancien org.springframework.orm.hibernate3.LocalSessionFactoryBean :
    <property name="mappingResources">
        <list> <value>Compte_with_details.hbm.xml</value>
            <value>Client_with_details.hbm.xml</value>
        </list>
    </property> -->
    <property name="annotatedClasses">
        <list><value>tp.entity.Compte</value>
            <value>tp.entity.Client</value>
        </list>
    </property>
```



```

    </property>
    <property name="hibernateProperties">
    <props><prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
    </props></property>
</bean>
...
<bean id="xxxxDAO" class="pppp.XYDaoHibernate">
    <property name="sessionFactory" ref="mySessionFactory" />
</bean> ...

```

Remarque :

Dans les anciennes versions de Spring (1.x et 2.0 / 2.5 par inertie) , les transactions au niveau des DAO étaient automatiquement prises en charge par la classe utilitaire "*HibernateTemplate*".

Dans les versions récentes de Spring (2.5 , 3, ...) , il est plus habituel de gérer les transactions via une configuration Xml de Spring-AOP ou bien via **@Transactional** .

NB: **@Transactional** est bien interprété par Spring si les configurations suivantes n'ont pas été oubliées:

```

<bean id="txManager"
      class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <!-- ou bien class="org.springframework.orm.hibernate4.HibernateTransactionManager" -->
    <property name="sessionFactory" ref="mySessionFactory" />
</bean>

```

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

3. Mise en oeuvre des transactions avec Spring 1.2

3.1. Enveloppe transactionnelle

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">

<beans>

....

<!-- version de base sans gestion des transactions :      -->
<bean id="basic.as.gestionComptes" class="as.pojo_impl.GestionComptes">
    <property name="compteDao" ref="ds.compteDAO" />
    <property name="clientDao" ref="ds.clientDAO" />
</bean>

<!-- version améliorée - avec gestion des transactions (Jdbc , Hibernate) -->

<bean id="as.gestionComptes" class=
"org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="txManager" />
    <property name="target" ref="basic.as.gestionComptes" />
    <property name="transactionAttributes">
        <props>
            <prop key="get*">PROPAGATION_REQUIRED,readOnly</prop>
            <prop key="transferer">PROPAGATION_REQUIRED</prop>

```

```
        <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
      </props>
    </property>
  </bean>

<!-- version non améliorée - sans gestion des transactions (Simu , Tests) -->
<!--
<alias name="basic.as.gestionComptes" alias="as.gestionComptes" />
-->

</beans>
```

Remarques:

L'enveloppe transactionnelle reprend toutes les méthodes publiques du composant de base et ajoute automatiquement une gestion des transactions.

En cas d'échec au niveau d'une transaction , l'enveloppe annule bien évidemment toutes les sous opérations déclenchées (mises à jour temporaires ,) et remonte une exception de type ***TransactionException*** (héritant de ***RuntimeException*** et ne nécessitant donc pas absolument de try/catch au niveau du client externe)

4. Démarrages possibles depuis spring 2.5

Depuis méthode main() dans une application « standalone »	<pre> ApplicationContext springContext = new ClassPathXmlApplicationContext("context.xml") ; Cxy c = (Cxy) springContext.getBean("idBeanXy"); //ou bien c = springContext.getBean(Cxy.class); </pre>
Depuis test unitaire (JUnit + spring-test)	<pre> @RunWith(SpringJUnit4ClassRunner.class) @ContextConfiguration(locations={"/context.xml"}) public class TestCxy { @Autowired private Cxy c ; //+ méthodes prefixées par @Test } </pre>
Depuis « listener web » (au démarrage d'une application web(.war) dans tomcat ou autre)	<pre> <context-param> <!-- dans WEB_INF/web.xml --> <param-name>contextConfigLocation</param-name> <param-value>classpath:/context.xml</param-value> </context-param> <listener><listener-class> org.springframework.web.context.ContextLoaderListener </listener-class></listener> ----- ... ctx = WebApplicationContextUtils .getWebApplicationContext(ap plication ou servletContext) ; ... ctx.getBean(...) ; //dans servlet ou jsp </pre>

5. Injection de Spring au sein d'un framework WEB

5.1. WebApplicationContext (configuration xml)

A intégrer au sein de *WEB-INF/web.xml*

<pre> <context-param> <param-name>contextConfigLocation</param-name> <param-value>classpath:/mySpringConf.xml</param-value> </context-param> <listener> <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class> </listener> </pre>
--

Ceci permet de charger automatiquement en mémoire la configuration "Spring" (ici le fichier "mySpringConf.xml" d'une partie du classpath (répertoire /WEB-INF/classes et/ou autre(s))) dès le démarrage de l'application WEB.

NB1: le paramètre *contextConfigLocation* peut éventuellement comporter une liste de chemin (vers plusieurs fichiers) séparés par des virgules .

Exemple: "classpath:/spring/*.xml" ou encore
"classpath:/contextSpring.xml,/classpath:/context2.xml"

NB2: les fichiers de configurations "xxx.xml" placé (en mode source) dans "src" (ou bien dans les ressources de maven) se retrouvent normalement dans /WEB-INF/classes en fin de "build" .

NB3: via le **préfixe "classpath*:/"** on peut préciser des chemins qui seront recherchés dans tous les éléments du classpath (c'est à dire dans tous les ".jar" du projet : par exemple tous les ".jar" présents dans WEB-INF/lib)

exemple:

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>
    classpath*/:serviceSpringConf.xml,classpath*/:dataSourceForTestSpringConf.xml
  </param-value>
</context-param>
```

5.2. WebApplicationContext (accès et utilisation)

Au sein d'un servlet ou bien d'un élément annexe on peut instancier des Beans via Spring :

```
application = .... getServletContext(); // application prédéfini au sein d'une page JSP
WebApplicationContext ctx =
    WebApplicationContextUtils.getWebApplicationContext(application);
IXxx bean = (IXxx) ctx.getBean(...);
....
request.setAttribute("nomBean",bean); // on stocke le bean au sein d'un scope (session,request,...)
rd.forward(request,response); // redirection vers page JSP
```

NB: Spring-web propose en plus des configurations complémentaires spécifiques pour bien intégrer la plupart des frameworks java-web (Struts, JSF , ...)

6. Injection "Spring" au sein du framework JSF

Rappel:

Intégrer au sein de *WEB-INF/web.xml*

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>classpath:/springConf1.xml, ...</param-value>
</context-param>
....
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
```

En plus de la configuration évoquée plus haut au niveau de *WEB-INF/web.xml*, il faut :

Modifier le fichier *WEB-INF/faces-config.xml* en y ajoutant le bloc "<application> ...</application>" précisant l'utilisation de *SpringBeanFacesELResolver*.

```
<faces-config>
  <application>
    <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
  </application>
```

Ceci permettra d'injecter des "beans Spring" (ex: services métiers) au sein des "managed-bean" de JSF de la façon suivante:

WEB-INF/faces-config.xml

```
<managed-bean>
  <managed-bean-name>myJsfBean</managed-bean-name>
  <managed-bean-class>myjsf.MyJsfBean</managed-bean-class>
  <managed-bean-scope>request</managed-bean-scope>
  <managed-property>
    <property-name>myService</property-name>
    <value>#{mySpringService}</value>
  </managed-property>
</managed-bean>
...
```

ou bien

```
@ManagedBean
@RequestScoped
public class MyJsfBean {

  @ManagedProperty("#{mySpringService}")
  private (I)ServiceSpring serviceSpring ; //+get/set
}
```

Effets:

Les noms #{xxx} utilisés par JSF seront résolus:

- par les mécanismes standards de JSF
- par le **SpringBeanFacesELResolver** de Spring puisant à son tour des "beans" instanciés via une fabrique de Spring (dans un second temps).

La résolution s'effectue sur les valeurs des ID ou Noms des composants "Spring".

En d'autres termes, les mécanismes JSF, déjà en partie basés sur des principes IOC, peuvent ainsi être ajustés pour injecter des composants Spring au sein des "Managed Bean" (ici *setMyService()* de la classe *myjsf.MyJsfBean*).

NB : Le lien automatique entre JSF et Spring peut se faire de 2 façons :

- Beans JSF utilisant des services métiers "Spring" (exemple précédent avec annotations JSF)
 - ManagedBean "JSF" d'abord instanciés par "Spring" et réutilisés par JSF
- Il faut pour cela bien régler le component-scan de spring pour qu'il englobe le package des mbeans et remplacer toutes les annotations JSF par des annotations équivalentes "Spring" (avantage : *@Autowired* plus simple que *@ManagedProperty* , inconvénient : moins d'auto-complétion dans .xhtml sous eclipse , idéal : *@Named* à la place de *@Component*)
exemple :

```
@Component
@Scope("request")
public class MyJsfBean {

    @Autowired
    private (I)ServiceSpring serviceSpring ; //pas besoin de get/set
}
```

ou encore (version à priori idéale avec *java.inject.**) :

```
@Named
@Scope("request")
public class MyJsfBean {

    @Inject // ou bien @Autowired
    private (I)ServiceSpring serviceSpring ; //pas besoin de get/set
}
```

avec dans **pom.xml**

```
...
<dependency>
    <groupId>org.apache.myfaces.core</groupId>
    <artifactId>myfaces-impl</artifactId> <!-- apache jsf impl -->
    <version>2.3.0</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>${springframework.version}</version>
</dependency>

<dependency>
    <groupId>javax.inject</groupId>
    <artifactId>javax.inject</artifactId> <!-- @Named et @Inject compatible spring -->
    <version>1</version>
</dependency>
```

7. Intégration de JSF 2 au sein de Spring-boot 2

pom.xml

```
...
<properties>
  <joinfaces.version>4.0.8</joinfaces.version>
  <java.version>1.8</java.version>
</properties>

<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.joinfaces</groupId>
      <artifactId>joinfaces-dependencies</artifactId>
      <version>${joinfaces.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <!-- utile pour class WelcomePageRedirect implements WebMvcConfigurer
    et pour event WS REST -->
  </dependency>

  <dependency>
    <groupId>org.joinfaces</groupId>
    <artifactId>primefaces-spring-boot-starter</artifactId>
    <!-- et indirectement jsf-spring-boot-starter -->
  </dependency>
...
</dependencies>

<build>
  <finalName>${project.artifactId}</finalName>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.joinfaces</groupId>
```



```

        <artifactId>joinfaces-maven-plugin</artifactId> <!-- ?????? -->
        <version>${joinfaces.version}</version>
    </plugin>
</plugins>
</pluginManagement>

<plugins>
    <plugin>
        <groupId>org.joinfaces</groupId>
        <artifactId>joinfaces-maven-plugin</artifactId>
    </plugin>
</plugins>
</build>

```

Dans **application.properties**

```

...
server.servlet.context-parameters.javafx.faces.PROJECT_STAGE=Development

```

WelcomePageRedirect.java

```

package tp.web;
import org.springframework.context.annotation.Configuration;
import org.springframework.core.Ordered;
import org.springframework.web.servlet.config.annotation.ViewControllerRegistry;
import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;

@Configuration
public class WelcomePageRedirect implements WebMvcConfigurer {

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/")
            .setViewName("forward:/index.html");
            //.setViewName("forward:/welcome.xhtml");
        registry.setOrder(Ordered.HIGHEST_PRECEDENCE);
    }
}

```

XyMBean.java

```
package tp.web;
import java.util.Date;
import javax.annotation.ManagedBean;
import javax.annotation.PostConstruct;
import javax.enterprise.context.RequestScoped;
import javax.inject.Inject;
import tp.service.XyService;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

//@Component
//@Named
@ManagedBean
//@Scope("request")
@RequestScoped
@Getter @Setter
@NoArgsConstructor
public class XyMBean {

    private String data;
    private String s;
    private Date date;

    //@Autowired
    @Inject
    private XyService xyService;

    @PostConstruct
    public void init() {
        //data="blabla";
        data=xyService.getData();
    }

    public String doDo() {
        System.out.println("doDo() , s="+s + " date="+date.toString());
        return null;
    }
}
```

Pages **.xhtml** fonctionnant bien si placées dans le répertoire

src/main/resources/META-INF/resources

p1.xhtml

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:ui="http://xmlns.jcp.org/jsf/facelets" xmlns:h="http://xmlns.jcp.org/jsf/html"
      xmlns:f="http://xmlns.jcp.org/jsf/core"      xmlns:p="http://primefaces.org/ui">

<h:head>
  <title>p1</title>
</h:head>
<h:body>
  <p3>p1.xhtml (jsf)</p3>
  <h:form>
    s:<h:inputText value="#{xyMBean.s}"/> <br/>
    date:<p:calendar value="#{xyMBean.date}"/> <br/>
    <h:commandButton value="submit" action="#{xyMBean.doDo}" />
  </h:form>
  data = <h:outputText value="#{xyMBean.data}" />
</h:body>
</html>
```

index.html

```
<html>
<head> <meta charset="ISO-8859-1">
  <title>Index majeur</title>
</head>
<body>
  <h1>ok (index.html)</h1>
  <a href="p1.jsf">p1.jsf</a><br/>
  <a href="p1.faces">p1.faces</a><br/>
  <a href="p1.xhtml">p1.xhtml</a><br/>
  <!-- les 3 formulations d'url .jsf ou .faces ou .xhtml fonctionnent bien , en choisir une -->
</body>
</html>
```

8. Injection "Spring" au sein du framework STRUTS

Rappel:

Intégrer au sein de *WEB-INF/web.xml*

```
....
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/classes/springConf1.xml, ...</param-value>
</context-param>
....
<listener>
  <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>
....
```

Pour injecter des composants "Spring" au sein du framework STRUTS on peut, au choix, utiliser l'une des deux approches suivantes:

- Utiliser le **ContextLoaderPlugin** pour que les mécanismes de STRUTS demandent (délèguent) à Spring d'instancier les beans d'actions (avec des injections de services "métier" paramétrés au sein d'un fichier de configuration de Spring [exemple: *WEB-INF/applicationContext.xml*]).

OU BIEN

- Programmer un bean d'action "STRUTS" en héritant de la classe "**ActionSupport**" fournie par Spring pour aider à récupérer des services "Spring" selon leur id en utilisant la méthode **getWebApplicationContext()** comme point d'entrée.

ContextLoaderPlugin et instanciation/injection "Spring" au niveau des "Action Beans"

Ajouter le plugin suivant à la fin de *struts-config.xml*:

```
... <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
  <set-property property="contextConfigLocation"
    value="/WEB-INF/action-servlet.xml,/WEB-INF/applicationContext.xml"/>
</plug-in> ...
```

Les déclaration/définition des beans d'actions (avec les injections de dépendances) se font au sein du fichier *action-servlet.xml*.

Les **correspondances de noms** entre les fichiers *struts-config.xml* et *action-servlet.xml* sont effectuées sur les attributs "path" (coté STRUTS) et "name" (coté Spring) de la façon suivante:

```
<action path="/myAction" .../>
dans struts-config.xml
```

```
<bean name="/myAction" .../>
dans action-servlet.xml
```

De façon à ce que STRUTS puisse tenir compte de certains noms provenant de action-servlet.xml , il faut en outre mettre en place l'un des deux mécanismes suivants:

- Remplacer le "RequestProcessor" de STRUTS par le "**DelegatingRequestProcessor**" fourni par Spring

OU BIEN

- Utiliser le "**DelegatingActionProxy**" au sein de l'attribut *type* de `<action-mapping .../>` .

La solution consistant à utiliser le " **DelegatingRequestProcessor** " se met en oeuvre en redéfinissant la valeur de la propriété *processorClass* au sein de **strut-config.xml**:

```
...<controller>
<set-property property="processorClass"
               value="org.springframework.web.struts.DelegatingRequestProcessor"/>
</controller>...
```

Ceci permet d'utiliser un "Action Bean" contrôlé par Spring quelque soit son type (l'attribut type n'est pas obligatoire):

```
<action path="/myAction" type="com.whatever.struts.MyAction"/>
```

ou bien

```
<action path="/myAction"/>
```

La solution alternative (lorsqu'un "RequestProcessor" personnalisé est déjà utilisé et ne peut pas être remplacé ou bien pour expliciter le fait que le bean d'action est pris en charge par Spring) consiste à utiliser le "**DelegatingActionProxy**" au niveau de l'attribut *type* :

```
<action path="/myAction" type="org.springframework.web.struts.DelegatingActionProxy"
      name="myForm" scope="request" validate="false" >
  <forward ..../> <forward ..../>
</action>
```

Ces éléments suffisent dans la plupart des cas .

Quelques considérations avancées (modules STRUTS , Tiles ,) figurent au sein de la documentation de référence.

Eventuelle récupération explicite d'un service "Spring" depuis un "Action bean" non directement contrôlé (instancié) par Spring:

```
public class MySpecifAction extends ActionSupport {
    public ActionForward execute(ActionMapping mapping,ActionForm form,
        HttpServletRequest request,HttpServletResponse response) throws Exception {
        WebApplicationContext ctx = getWebApplicationContext();
        myService service = (MyService) ctx.getBean("myService");
        ....
        return mapping.findForward("success");
    }
}
```

9. Injection "Spring" au sein du framework STRUTS2

Dépendances maven (dans pom.xml) :

```
...
    <dependency>
        <groupId>org.apache.struts</groupId>
        <artifactId>struts2-core</artifactId>
        <version>2.3.15.3</version>
    </dependency>

    <dependency>
        <groupId>org.apache.struts</groupId>
        <artifactId>struts2-spring-plugin</artifactId>
        <version>2.3.15.1</version>
    </dependency>
....
```

Dans WEB-INF/web.xml :

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/classes/springActions.xml,...</param-value>
</context-param>
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

...
<filter>
    <filter-name>struts2</filter-name>
    <filter-class>org.apache.struts2.dispatcher.ng.filter.StrutsPrepareAndExecuteFilter</filter-
class>
</filter>

<filter-mapping>
    <filter-name>struts2</filter-name>
    <url-pattern>*.action</url-pattern>
</filter-mapping>
```

Les beans d'action de Struts2 peuvent accéder aux services "spring" via une façade , via des appels explicites à `webApplicationContext.getBean()` ou via des injections de dépendances directes si les "bean d'action de Struts2" sont d'abord pris en charge par spring et ensuite ré-utilisés par Struts2.

.../...

Dans **springActions.xml** ouxml :

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

    <!-- NB les id des springActions doivent correspondent aux pseudo-classes des actions de
struts.xml
         et le struts2-spring-plugin-x-x-x.jar doit etre dans le classpath -->

    <bean id="clientComptesSpAction"
class="tp.myapp.web.spring_action.ClientComptesSpAction" scope="session"
        init-method="initClientContextInSession" >
        <property name="serviceGestionClients" ref="gestionClientsImpl"/>
        <property name="serviceGestionComptes" ref="gestionComptesImpl"/>
    </bean>
...
</beans>
```

Dans **struts.xml** du classpath (pas dans WEB-INF) :

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE struts PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 2.0//EN"
"http://struts.apache.org/dtds/struts-2.0.dtd">

<struts>
<constant name="struts.enable.DynamicMethodInvocation" value="false" />
<constant name="struts.objectFactory" value="spring" />
<constant name="struts.devMode" value="false" />

    <!-- NB les id des springActions doivent correspondent aux pseudo-classes des actions de
struts.xml et le struts2-spring-plugin-x-x-x.jar doit etre dans le classpath -->

    <package name="tp.myapp.web.action" namespace="/" extends="struts-default">

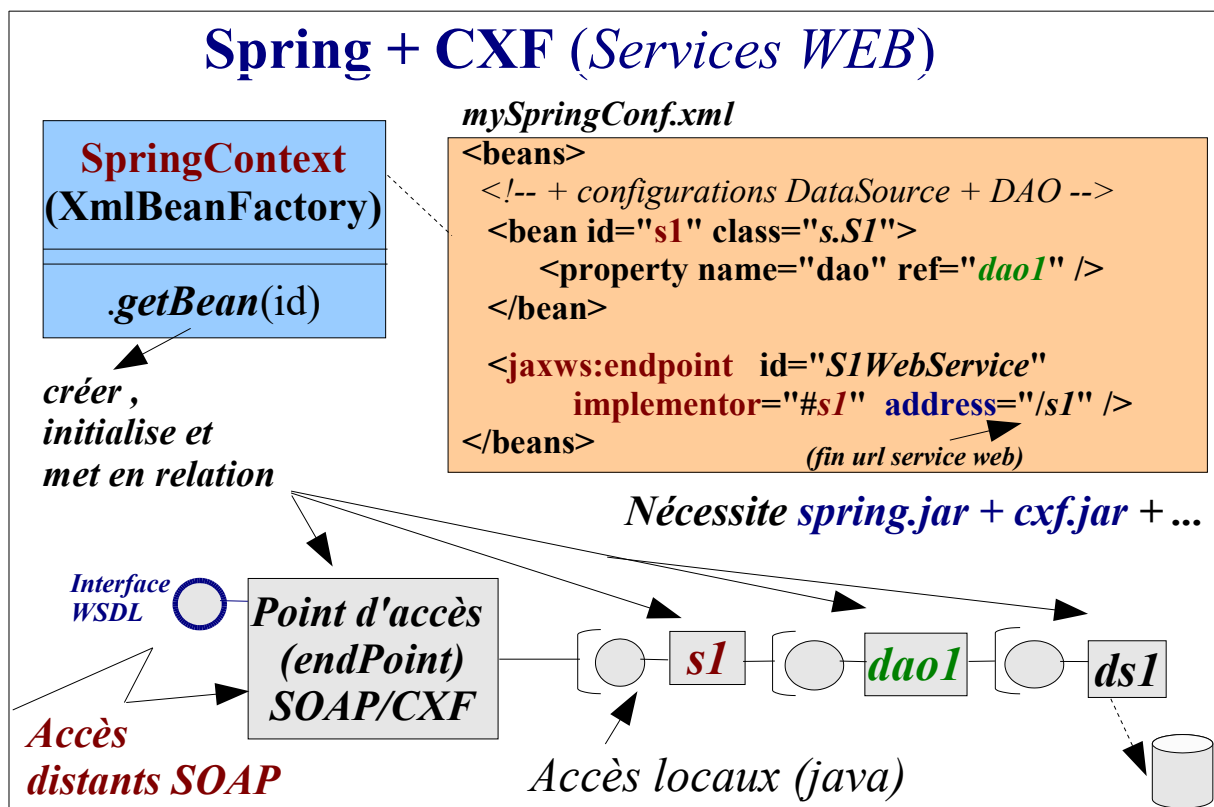
        <action name="identifierClient" class="clientComptesSpAction"
            method="identifierClient">
            <result name="listeComptes" >/pages/listeComptes.jsp</result>
            <result name="input">/pages/identificationClient.jsp</result>
        </action>
    ...
</package>
```


10. Spring + CXF (pour Web Services SOAP)

10.1. Présentation de CXF (apache)

- **CXF** est une technologie facilitant le développement et la construction de **Web Services** en se basant sur l'API **JAX-WS**.
- Le framework **CXF** (de la fondation Apache) permet de mettre en place des **points d'accès (SOAP)** vers les **services métiers** d'une application.
- Les points d'accès (endpoints) sont pris en charge par le servlet prédéfini "**CxfServlet**" qu'il suffit de paramétrer et d'intégrer dans une application java web (*sans nécessiter d'EJB*).

L'intégration de CXF avec Spring est simple :



- Une description WSDL du service web sera alors automatiquement générée et le service métier Spring sera non seulement accessible localement en java mais sera accessible à distance en étant vu comme un service Web que l'on peut invoquer par une URL http .
- Il faudra penser à déclarer le servlet de CXF dans WEB-INF/web.xml et placer tous les ".jar" de CXF dans WEB-INF/lib .
- Le paramétrage fin du service Web s'effectue en ajoutant les annotations standards de JAX-WS (@WebService , @WebParam , ...) dans l'interface et la classe d'implémentation du service métier .

Remarque importante: CXF (et CxfServlet) peut à la fois prendre en charge *SOAP* (via JAX-WS) et aussi *REST* (via JAX-RS) . CXF peut également s'utiliser sans Spring.

10.2. implémentation avec CXF (et Spring) au sein d'une application Web (pour Tomcat 5.5, 6 ou 7)

- Rapatrier les ".jar" de CXF au sein du répertoire **WEB-INF/lib** d'un projet Web. (ou bien configurer convenablement les dépendances "maven").
- Ajouter la configuration suivante au sein de **WEB-INF/web.xml**:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>WEB-INF/classes/beans.xml</param-value> <!-- ou .... -->
</context-param>

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

<servlet>
    <servlet-name>CXFServlet</servlet-name>
    <servlet-class>org.apache.cxf.transport.servlet.CXFServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>CXFServlet</servlet-name>
    <url-pattern>/services/*</url-pattern> <!-- ou autre que services/* -->
</servlet-mapping>
```

- Ajouter la configuration Spring suivante dans WEB-INF ou bien dans [src ou src/main/resources ==> WEB-INF/classes] :

beans.xml

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

    <import resource="classpath:META-INF/cxf/cxf.xml" />
    <import resource="classpath:META-INF/cxf/cxf-extension-soap.xml" />
    <import resource="classpath:META-INF/cxf/cxf-servlet.xml" />

    <jaxws:endpoint id="calculateurService"
        implementor="service.CalculateurService" address="/Calculateur" />

</beans>
```

variante plus pratique si plusieurs niveaux d'injections (ex: DAO , DataSource, ...):

```
<bean id="calculateurService_impl" class="service.CalculateurService" >
  <property name="...." ref="...." />
</bean>

<jaxws:endpoint
  id="calculateurService"
  implementor="#calculateurService_impl"
  address="/Calculateur" />
```

NB:

- La syntaxe est "#idComposantSpring"
- Dans l'exemple ci dessus "service.CalculateurService" correspond à la classe d'implémentation du service Web (avec @WebService) et "/Calculateur" est la partie finale de l'url menant au service web pris en charge par Spring+CXF .
- Après avoir déployé l'application et démarré le serveur d'application (ex: run as/ run on serveur),il suffit de préciser une URL du type <http://localhost:8080/myWebApp/services> pour obtenir la liste des services Web pris en charge par CXF. En suivant ensuite les liens hypertextes on arrive alors à la description WSDL .

==> c'est tout simple , modulaire via Spring et ça fonctionne avec un simple Tomcat !!!

Remarque :

En ajoutant le paramétrage bindingUri="<http://www.w3.org/2003/05/soap/bindings/HTTP/>" on peut générer un deuxième "endpoint" en version "soap 1.2" plutôt que soap 1.1 :

```
<jaxws:endpoint id="serviceCalculateur_soap12_endPoint"
  bindingUri="http://www.w3.org/2003/05/soap/bindings/HTTP/"
  implementor="#calculateurService_impl" address="/calculateur_soap12">
  <!-- version SOAP 1.2 -->
</jaxws:endpoint>
```

10.3. Client JAX-WS sans wsimport avec CXF et Spring

Rappel: Dans le cas (plus ou moins rares) où les cotés "client" et "serveur" sont tous les deux en Java , il est possible de se passer de la description WSDL et l'on peut générer un client d'appel (proxy/business delegate) en se basant directement sur l'interface java du service web (SEI : Service EndPoint Interface)

La configuration Spring suivante permet de générer automatiquement un "business_delegate" (dont l'id est ici "client" et qui est basé sur l'interface "service.Calculateur" du service Web).

Ce "business delegate" délèguera automatiquement les appels au service Web distant dont l'url est précisé par la propriété "address".

src/bean.xml

```
<beans xmlns="http://www.springframework.org/schema/beans"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jaxws="http://cxf.apache.org/jaxws"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
http://cxf.apache.org/jaxws http://cxf.apache.org/schema/jaxws.xsd">
<bean id="clientFactory" class="org.apache.cxf.jaxws.JaxWsProxyFactoryBean">
  <property name="serviceClass" value="service.Calculateur" /> <!-- interface_SEI -->
  <property name="address" value="http://localhost:8080/serveur_cxf/Calculateur"/>
</bean>
<bean id="client" class="service.Calculateur" factory-bean="clientFactory"
  factory-method="create"/>
</beans>

```

autre possibilité (syntaxe plus directe) :

```

<jaxws:client id="client"
  serviceClass="service.Calculateur"
  xmlns:tp="http://service.tp/
  serviceName="tp:CalculateurImplService"
  endpointName="tp:CalculateurServiceEndpoint"
  address="http://localhost:8080/serveur\_cxf/Calculateur" />
  <!-- avec serviceClass="package.Interface_SEI" et la possibilité d'ajouter des
intercepteurs -->

```

il ne reste alors plus qu'à utiliser ce "business_delegate" au sein du code client basé sur Spring:

```

import org.springframework.context.support.ClassPathXmlApplicationContext;
...
public class WsTestApp {

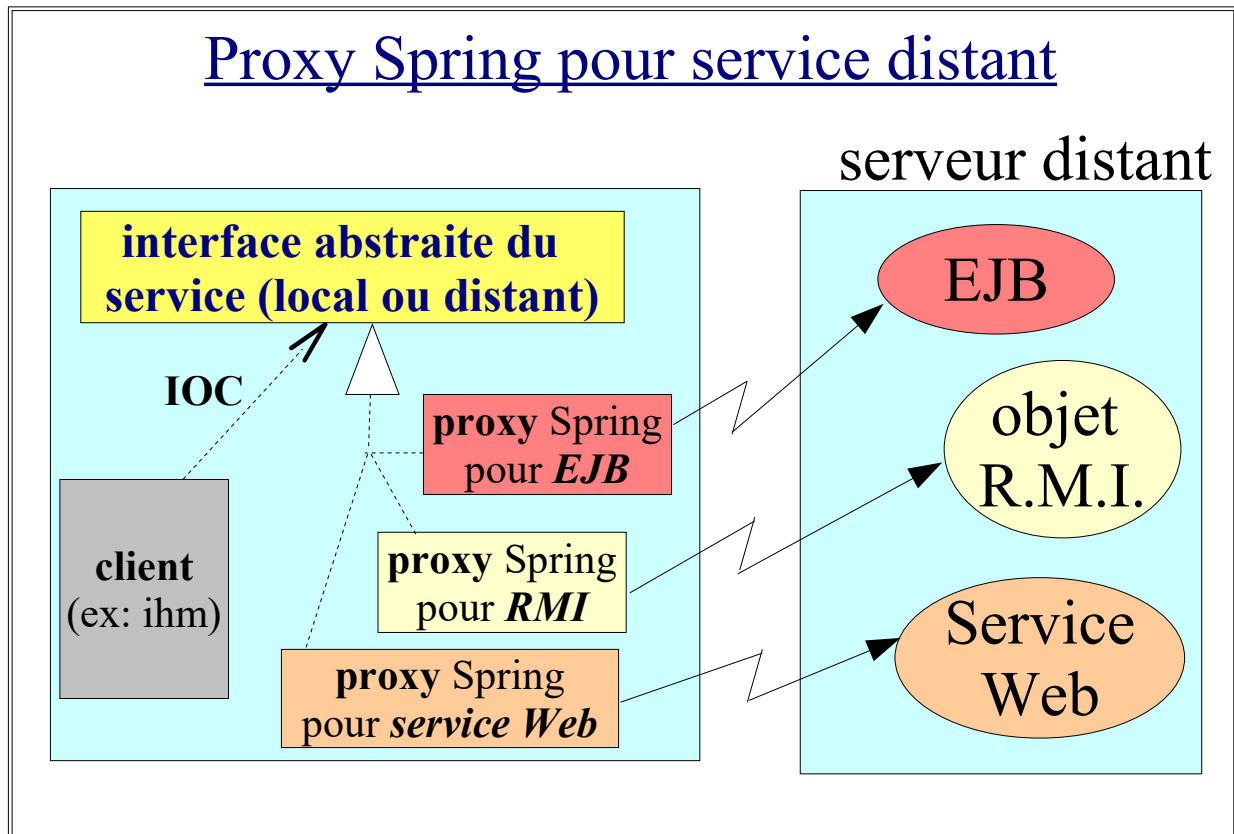
    public static void main(String[] args) {
        ClassPathXmlApplicationContext context = new
            ClassPathXmlApplicationContext(new String[] { "beans.xml" });

        Calculateur calculateur = (Calculateur) context.getBean("client");
        System.out.println(calculateur.getTva(200, 19.6));
    }
}

```

NB: il faut penser à rapatrier les ".jar" de cxf et le code source de l'interface SEI (ici service.Calculateur) .

11. Spring (proxy / business delegate)



NB: les proxys "Spring" (coté client) sont décorélés et indépendants des éventuelles implémentations "Spring" coté serveur (un proxy à base de Spring n'est pas obligatoirement associé à une implémentation serveur à base de Spring et vice-versa).

11.1. implémentation d'un service RMI basé sur un POJO

```
<bean id="accountService" class="example.AccountServiceImpl">
<!-- any additional properties, maybe a DAO? -->
</bean>
```

```
<bean class="org.springframework.remoting.rmi.RmiServiceExporter">
<!-- does not necessarily have to be the same name as the bean to be exported -->
<property name="serviceName" value="MyAccountService"/>
<property name="service" ref="accountService"/>
<property name="serviceInterface" value="example.AccountService"/> <!-- sans Remote -->
<property name="registryPort" value="1099"/> <!-- defaults to 1099 -->
</bean>
```

==> L'enveloppe RMI ainsi générée sera accessible à distance via l'URL suivante:

rmi://SERVERHOST:1099/MyAccountService

11.2. proxy automatique "Spring" pour objet distant RMI

```
<bean id="accountService"
      class="org.springframework.remoting.rmi.RmiProxyFactoryBean">
<property name="serviceUrl" value="rmi://HOST:1099/MyAccountService"/>
<property name="serviceInterface" value="example.AccountService"/>
</bean>
```

```
<bean class="example.MySimpleClientObject">
  <property name="accountService" ref="accountService"/>
</bean>
```

```
public class MySimpleClientObject {
    private AccountService accountService; // interface ordinaire (sans Remote)
    public void setAccountService(AccountService accountService) {
        this.accountService = accountService;
    }
    ...
}
```

==> Si `java.rmi.RemoteException` ==> transformée en `java.lang.RuntimeException` .

III - Anciennes configurations / Spring security

1. Extension Spring-security (old versions)

L'extension **Spring-security** permet de simplifier le paramétrage de la **sécurité JEE** dans le cadre d'une application JEE/Web basée sur Spring.

Les principaux apports de spring-security sont les suivants :

- syntaxe xml ou java simplifiée (plus compacte et plus lisible que le standard "web.xml")
- possibilité de contrôler entièrement par configuration Spring le "realm" (domaine d'utilisateurs) qui servira à gérer les authentifications. La sécurisation de l'application devient ainsi plus indépendante du serveur d'application hôte.
- possibilité de switcher facilement de configuration (liste memory ou xml , database , ldap)
- possibilité de configurer via l'annotation `@PreAuthorize("hasRole('role1') or hasRole('role2')")` les méthodes des composants "spring" qui seront ou pas accessibles selon le rôle de l'utilisateur authentifié.
- autres fonctionnalités utiles (cryptage des mots de passe via bcrypt, ...)
- possibilité d'interfacer spring-security et oauth2

1.1. Ancien mode de configuration (xml)

Les premières versions de spring-security étaient jadis configurées via des fichiers xml .

Dans pom.xml , **spring-security-core** , **spring-security-web** et **spring-security-config**

```
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-core</artifactId>
  <version>${org.springframework.security.version}</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-web</artifactId>
  <version>${org.springframework.security.version}</version>
</dependency>
<dependency>
  <groupId>org.springframework.security</groupId>
  <artifactId>spring-security-config</artifactId>
  <version>${org.springframework.security.version}</version>
</dependency>
```

avec des versions un peu décalées par rapport à spring-framework :

```
<properties>
  <org.springframework.version>4.3.2.RELEASE</org.springframework.version>
  <org.springframework.security.version>4.1.3.RELEASE</org.springframework.security.version>
</properties>
```

Dans web.xml , il fallait déclarer le filtre à utiliser

org.springframework.web.filter.DelegatingFilterProxy via `<filter>` et `<filter-mapping>`

Et la configuration xml/spring de l'application faisait généralement référence à un sous fichier

importé **security-config.xml** comportant :

- des droits accès selon "rôles utilisateurs" et selon uri/url (<security-http> , <security:intercept-url , permitAll , denyAll ,)
- des paramétrages de formulaire de login , ...
- un paramétrage de gestionnaire d'authentification (ldap , jdbc , xml) et éventuellement quelques comptes utilisateurs pour effectuer des tests simples en mode développement
(<security:authentication-manager> <security:authentication-provider> <security:user-service> <security:user name="user1" password="pwd1" authorities="ROLE_USER" />)

1.1.a. Filtre web pour spring-security à déclarer dans 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>
```

1.1.b. Exemple de configuration spring pour spring-security:

<import resource="security-config.xml" />

puis **security-config.xml** :

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:security="http://www.springframework.org/schema/security"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-3.1.xsd
        http://www.springframework.org/schema/security
        http://www.springframework.org/schema/security/spring-security-3.1.xsd">

    <!-- parametrage spring utile si filtre web DelegatingFilterProxy actif (dans web.xml) -->
    <security:http use-expressions="true">
        <security:intercept-url pattern="/index.html" access="permitAll" />
        <security:intercept-url pattern="/libre.jsp" access="permitAll" />
        <security:intercept-url pattern="/spring/*" access="permitAll" />
        <!-- <security:intercept-url pattern="/*" access="permitAll" /> -->
        <security:intercept-url pattern="/member.jsp"
            access="hasAnyRole('ROLE_SUPERVISOR','ROLE_TELLER')"/>
```



```

<!-- <security:intercept-url pattern="/listAccounts.html" access="isAuthenticated()" /> -->
<security:intercept-url pattern="/**" access="denyAll" />
<security:form-login />
<security:logout />
<!-- attention form et WS REST ( en mode POST , PUT, ...) par défaut bloqués
      si csrf non désactivé ou bien token _csrf non géré (hidden , ...) -->
< !-- <security:csrf disabled="true"/> -->

</security:http>

<!-- pour ne pas securiser les .css, ... -->
<!-- <security:http pattern="/static/**" security="none" /> -->

<!-- pour encoder/decoder des mots de passe cryptés (encoder.encode("..."))
password="9992e040d32b6a688ff45b6e53fd0f5f1689c754ecf638cce5f09aa57a68be3c6dae699091e58324"
-->
<!-- <bean id="encoder"
      class="org.springframework.security.crypto.password.StandardPasswordEncoder"/> -->

<security:authentication-manager>
  <security:authentication-provider>
    <!-- <security:password-encoder ref="encoder" /> -->
    <!-- en prod , via jdbc , ldap ou ... -->
    <security:user-service> <!-- petite liste explicite pour tests (mode dev) -->
      <security:user name="admin" password="adminpwd"
        authorities="ROLE_SUPERVISOR,ROLE_TELLER,ROLE_USER" />
      <security:user name="user1" password="pwd1" authorities="ROLE_USER" />
      <security:user name="didier" password="didierpwd"
        authorities="ROLE_TELLER,ROLE_USER" />
      <security:user name="user2" password="pwd2" authorities="ROLE_USER" />
    </security:user-service>
  </security:authentication-provider>
</security:authentication-manager>

<!-- <security:global-method-security pre-post-annotations="enabled" />
pour tenir compte de @PreAuthorize("hasRole('supervisor') or hasRole('teller')")
au dessus des méthodes des services Spring -->

</beans>

```

Le préfixe (par défaut) attendu pour les rôles est "ROLE_" .

1.2. Champ caché "_csrf" de spring-mvc utile pour pages/vues "java/jsp" mais inutile pour Api-REST avec tokens .

NB: Ce champ caché correspond au "Synchronizer Token Pattern" (que l'on retrouve dans les frameworks web concurrents "Stuts" ou "JSF") : le coté serveur compare la valeur d'un jeton aléatoire stockée en session http avec celle stockée dans un champ caché et refuse de gérer la requête "re-postée" si la comparaison n'est pas réussie.

D'autre part , le terme **CSRF** (signifiant "Cross Site Request Forgery" correspond à un éventuel problème de sécurité : un site "malveillant" (utilisé en parallèle au sein d'un navigateur) déclenche automatiquement (via javascript ou autre) des requêtes non voulues (ex : virement monétaire) en utilisant le contexte d'un site à priori de confiance (mais pas assez protégé) .

Avec <form> (au lieu de <form:form> de SpringMvc / jsp) , il faut insérer nous même le champ suivant au sein du formulaire d'une page ".jsp" :

```
<input type="hidden" name="${_csrf.parameterName}" value="${_csrf.token}"/>
```

<form:form ...> de SpringMvc / jsp ou bien l'équivalent thymeleaf gère (génère) automatiquement le champ caché _csrf attendu par spring-security . *Exemple* : <input type="hidden" name="_csrf" value="8df91b84-74c1-4013-bd44-ed7b00779a2" />) .

Lorsque qu'il est nécessaire de faire transiter le champ "_csrf" via **JSON / Ajax (par exemple depuis une page HTML/jQuery ou angularJs invoquant un WS REST)**, on utilise plutôt un cookie et un champ de l'entête HTTP :

Au sein de security-config.xml :

```
<bean id="tokenRepository"
class="org.springframework.security.web.csrf.CookieCsrfTokenRepository">
    <property name="cookieHttpOnly" value="false"/>
</bean> <!-- pour stocker _csrf dans un cookie XSRF-
TOKEN dont la valeur est a renvoyer dans le champ X-
XSRF-TOKEN de l'entete HTTP par jQuery ou AngularJs -->

<security:http use-expressions="true">
    ...
    <security:csrf
        token-repository-ref="tokenRepository"/>
</security:http>
```

Dans page html/jquery :

```
...
<script src="jquery-2.2.1.js"></script>
<script src="jquery.cookie.js"></script>
```

et

```
$ (function() {

//...

var xsrfToken = $.cookie('XSRF-TOKEN');
//necessite plugin jquery-cookie
if(xsrfToken) {
    $(document).ajaxSend(function(e, xhr, options) {
        xhr.setRequestHeader('X-XSRF-TOKEN', xsrfToken);
    });
}
//...
})
```

1.3. Configuration un peu plus moderne de spring-security en java

```
....
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
@Configuration
public class MySecurity {

    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(); //By default since spring 5
    }
}
```

L'exemple élémentaire ci-dessous permet de configurer une liste d'utilisateurs en mémoire (pratique pour effectuer quelques tests rapides en phase de développement).

```
package .....;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import org.springframework.security.config.annotation.web.configuration.WebSecurityConfigurerAdapter;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
```

```

private BCryptPasswordEncoder passwordEncoder;

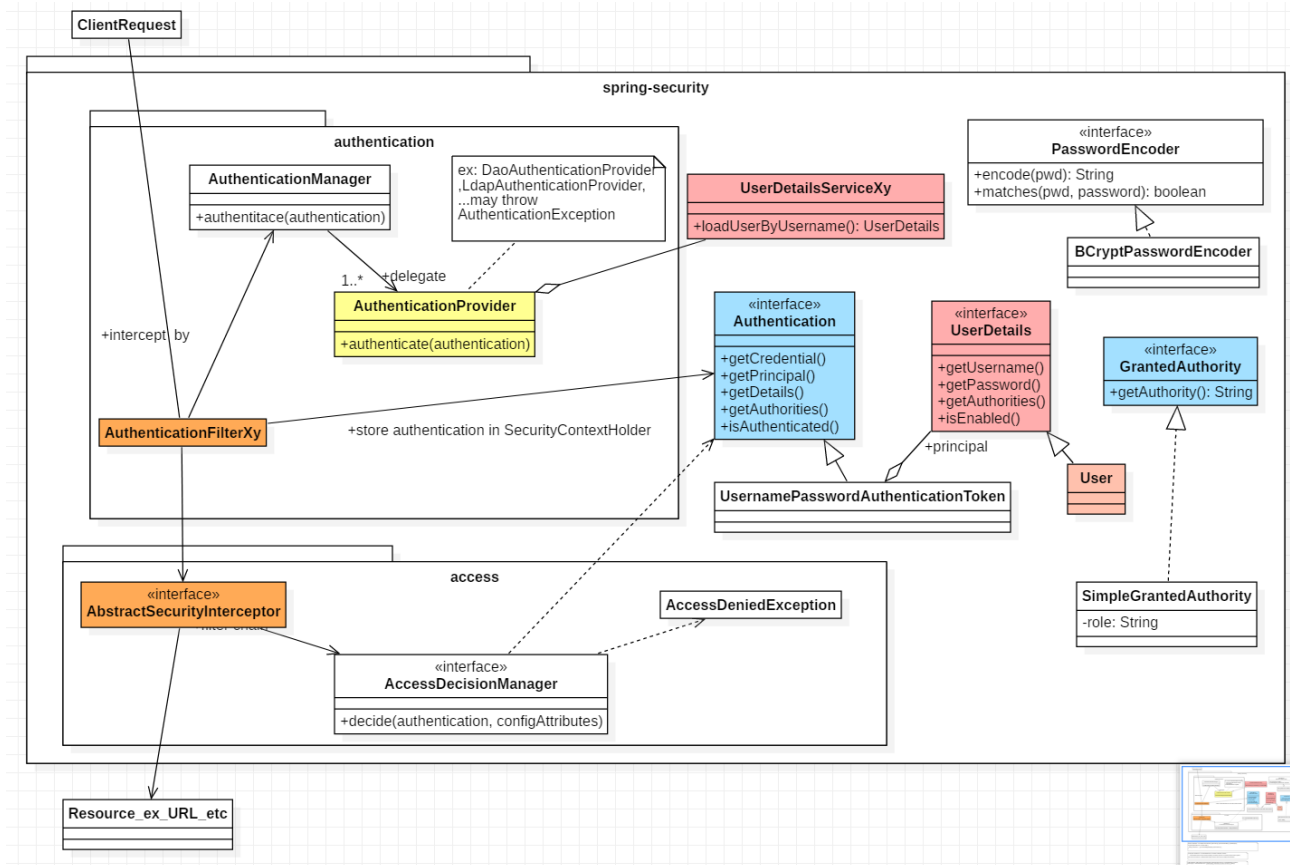
@Autowired
public void globalUserDetails(final AuthenticationManagerBuilder auth) throws Exception {
    auth.inMemoryAuthentication()
        .withUser("user1").password(passwordEncoder.encode("pwd1")).roles("USER").and()
        .withUser("admin1").password(passwordEncoder.encode("pwd1")).roles("ADMIN").and()
        .withUser("user2").password(passwordEncoder.encode("pwd2")).roles("USER").and()
        .withUser("admin2").password(passwordEncoder.encode("pwd2")).roles("ADMIN");
}

@Override
@Bean
public AuthenticationManager authenticationManagerBean() throws Exception {
    return super.authenticationManagerBean();
}

@Override
protected void configure(final HttpSecurity http) throws Exception {
    /*
    // config pour Spring-mvc avec pages jsp :
    http.authorizeRequests()
        .antMatchers("/",
            "/favicon.ico",
            "/*/*.*png",
            "/*/*.*gif",
            "/*/*.*svg",
            "/*/*.*jpg",
            "/*/*.*html",
            "/*/*.*css",
            "/*/*.*js").permitAll()
        .anyRequest().authenticated()
        .and()
        .formLogin().permitAll()
        .and().csrf();
    */
    // config pour Spring-mvc avec WS-REST et tokens (ex : jwt) :
    http.authorizeRequests()
        // .antMatchers("/rest/*").permitAll()
        .anyRequest().permitAll()
        // .anyRequest().authenticated()
        // .anyRequest().hasRole("ADMIN")
        .and().csrf().disable();
        // .and().httpBasic()
    }
}

```

1.4. Vue d'ensemble sur "Spring-security"



NB: https://en.wikipedia.org/wiki/Spring_Security comporte un assez bon schéma montrant les mécanismes fondamentaux de spring-security .

Lorsqu'une requête HTTP (de "login" ou autre) arrive , celle-ci est interceptée par un filtre web (prédéfini ou bien personnalisé) .

Ce filtre web va alors appeler la méthode `authenticate()` sur un objet de type "AuthenticationProvider" géré par un "AuthenticationManager" .

Authentication *authenticate*(Authentication authentication) throws AuthenticationException;

avant appel : authentication avec `getPrincipal()` retournant souvent username (String)

`getCredential()` retournant password ou autre .

après appel : authentication avec `getPrincipal()` retournant UserDetails si ok ou bien AuthenticationException sinon

En interne l'objet "AuthenticationProvider" s'appuie sur une implémentation de l'interface **UserDetailsService** avec cette unique méthode :

UserDetails *loadUserByUsername*(String username) throws UsernameNotFoundException;

Cette méthode est censée remonter les données d'un compte utilisateur depuis un certain endroit (base de données , LDAP ,) .

Ces infos "utilisateur" doivent être une implémentation de l'interface "UserDetails" (classe "User"

par exemple). L'objet "User" (ou un équivalent implémentant "UserDetails") est censée comporter le bon mot de passe.

Les mécanismes internes de Spring-security ("AuthenticationProvider" , ...) vont alors pouvoir comparer le bon mot de passe avec celui renseigné par l'utilisateur qui souhaite s'authentifier.

Dans certains cas la comparaison passe par une implémentation de "PasswordEncoder" (ex : "BCryptPasswordEncoder") lorsque les mots de passe sont cryptés dans la base de données.

Si l'authentification échoue --> AuthenticationException --> fin (pas de bras , pas de chocolat)

Si l'authentification est réussie --> la méthode authenticate() retourne un objet (implémentant l'interface "Authentication") bien complet (comportant "Roles utilisateurs" , ...).

L'objet "Authentication" est alors automatiquement stocké dans le "SecurityContextHolder" par spring-security .

Une fois l'authentification effectuée et stockée dans le contexte , on peut alors très facilement accéder aux infos "utilisateur" vérifiées via des instructions de ce type :

```
Object principal = SecurityContextHolder.getContext().getAuthentication().getPrincipal();
if (principal instanceof UserDetails) {
    String username = ((UserDetails)principal).getUsername();
}
```

L'objet "Authentication" comporte une méthodes **getAuthorities()** retournant un paquet d'éléments de type "GrantedAuthority" dont "SimpleGrantedAuthority" est l'implémentation la plus classique.

"SimpleGrantedAuthority" comporte un nom de rôle (ex "ROLE_ADMIN" ou "ROLE_USER" , ...)

Lorsqu'un peu plus tard , un accès à une partie de l'application sera tenté (page jsp , méthode appelée sur un contrôleur , ...) les mécanismes de la partie "contrôle d'accès" de spring-security pour alors assez facilement autoriser ou refuser les actions en comparant les rôles mémorisés dans l'objet "Authentication" du contexte avec certaines configurations du genre :

```
@PreAuthorize("hasRole('ADMIN')")
```

1.5. Authentification jdbc ("realm" en base de données)

La configuration ci-après permet de configurer **spring-security** pour qu'il accède à une **liste de comptes "utilisateurs" dans une base de données relationnelle** (ex : H2 ou Mysql ou ...).

Cette base de données sera éventuellement différente de celle utilisée par l'aspect fonctionnel de l'application .

Au sein de l'exemple suivant , la méthode **initRealmDataSource()** paramètre un objet DataSource vers une base h2 spécifique à l'authentification (**jdbc:h2:~/realmdb**).

L'instruction

```
JdbcUserDetailsManagerConfigurer jdbcUserDetailsManagerConfigurer =
```

```
auth.jdbcAuthentication().dataSource(realDataSource);
```

permet d'initialiser AuthenticationManagerBuilder en mode jdbc en précisant le DataSource et donc la base de données à utiliser .

L'instruction jdbcUserDetailsManagerConfigurer.**withDefaultSchema()**; (à ne lancer que si les tables **"users"** et **"authorities"** n'existent pas encore dans la base de données) permet de créer les tables nécessaires (avec noms et structures par défaut) dans la base de données.

Par défaut , la table **users(username, password)** comporte les mots de passe (souvent cryptés) et la table **authorities(username, authority)** comporte la liste des rôles de chaque utilisateur

JdbcAppDbGlobalUserDetailsConfig.java à adapter au contexte

```
package org.mycontrib.generic.security.config;
import java.sql.Connection;
import java.sql.DatabaseMetaData;
import java.sql.ResultSet;
import javax.sql.DataSource;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Profile;
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;
import org.springframework.security.config.annotation.authentication.configurers.provisioning.JdbcUserDetailsManagerConfigurer;
import org.springframework.security.config.annotation.authentication.configurers.provisioning.UserDetailsManagerConfigurer;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

@Configuration
//@Profile("appDbSecurity") //with jdbc
public class JdbcAppDbGlobalUserDetailsConfig {
    @Autowired
    private BCryptPasswordEncoder passwordEncoder;

    private static DataSource realmDataSource;

    private static void initRealmDataSource() {
        DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();
        driverManagerDataSource.setDriverClassName("org.h2.Driver");
        driverManagerDataSource.setUrl("jdbc:h2:~/realmdb");
        driverManagerDataSource.setUsername("sa");
        driverManagerDataSource.setPassword("");
        realmDataSource = driverManagerDataSource;
    }

    private boolean isRealmSchemaInitialized() {
        int nbExistingTablesOfRealmSchema = 0;
        try {
            Connection cn = realmDataSource.getConnection();
            DatabaseMetaData meta = cn.getMetaData();
            String tabOfTableType[] = {"TABLE"};
            ResultSet rs = meta.getTables(null,null,"%",tabOfTableType);
            while(rs.next()){
                String existingTableName = rs.getString(3);
                if(existingTableName.equalsIgnoreCase("users")
                    || existingTableName.equalsIgnoreCase("authorities")) {
```



```

        nbExistingTablesOfRealmSchema++;
    }
    }
    rs.close();
    cn.close();
} catch (Exception e) {
    e.printStackTrace();
}
return (nbExistingTablesOfRealmSchema>=2);
}

@Autowired
public void globalUserDetails(final AuthenticationManagerBuilder auth) throws Exception {
    initRealmDataSource();
    JdbcUserDetailsManagerConfigurer jdbcUserDetailsManagerConfigurer =
        auth.jdbcAuthentication().dataSource(realmDataSource);
    if(isRealmSchemaInitialized()) {
        /*
        jdbcUserDetailsManagerConfigurer
        .usersByUsernameQuery("select username,password,enabled from users where username=?")
        .authoritiesByUsernameQuery("select username, authority from authorities where username=?");
        //by default
        */
        // or .authoritiesByUsernameQuery("select username, role from user_roles where username=?")
        //if custom schema
    }else {
        //creating default schema and default tables "users" , "authorities"
        jdbcUserDetailsManagerConfigurer.withDefaultSchema();
        //insert default users:
        configureDefaultUsers(jdbcUserDetailsManagerConfigurer);
    }
}

void configureDefaultUsers(UserDetailsManagerConfigurer udmc){
    udmc
        .withUser("user1").password(passwordEncoder.encode("pwduser1")).roles("USER").and()
        .withUser("admin1").password(passwordEncoder.encode("pwdadmin1")).roles("ADMIN","USER").and()
        .withUser("publisher1").password(passwordEncoder.encode("pwdpublisher1")).roles("PUBLISHER","USER").and()
        .withUser("user2").password(passwordEncoder.encode("pwduser2")).roles("USER").and()
        .withUser("admin2").password(passwordEncoder.encode("pwdadmin2")).roles("ADMIN").and()
        .withUser("publisher2").password(passwordEncoder.encode("pwdpublisher2")).roles("PUBLISHER");
}
}

```

1.6. Authentication "personnalisée" en implémentant l'interface UserDetailsService

Si l'on souhaite coder un accès spécifique à la liste des comptes utilisateurs (ex : via JPA ou autres), on peut implémenter l'interface **UserDetailsService** .

Exemple :

```
package .....;

import java.util.ArrayList;  import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;
import org.springframework.security.core.userdetails.User;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.stereotype.Service;

import .....tp.entity.Customer;
import .....tp.service.CustomerService;

@Service
public class MyUserDetailsService implements UserDetailsService {
    @Autowired
    private PasswordEncoder passwordEncoder;

    @Autowired
    private CustomerService customerService; //ex with JpaRepository or ...

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
        String password=null;
        if(username.equals("superAdmin")) {
            password=passwordEncoder.encode("007");
            authorities.add(new SimpleGrantedAuthority("ROLE_ADMIN"));
        }
        else {
            try {
                String email = username; //dans cet exemple l'email fait office de username
                Customer c = customerService.findCustomerByEmail(email);
                authorities.add(new SimpleGrantedAuthority("ROLE_CUSTOMER"));
                password=c.getPassword();
            } catch (NumberFormatException e) {
                e.printStackTrace();
            }
        }
    }
}
```

```

    }
    //NB : User est une classe prédéfinie de SpringSecurity
    // qui implémente l'interface UserDetails .
    return new User(username, password, authorities);
    //On retourne ici comme information une association entre usernameRecherché et
    //(bonMotDePasseCrypté + liste des rôles)
    //Le bonMotDePasseCrypté servira simplement à effectuer une comparaison avec le mot
    //de passe qui sera saisi ultérieurement par l'utilisateur.
}
}

```

Enregistrement de cette classe au sein d'un WebSecurityConfigurerAdapter :

```

package .....;
....
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
//necessary for @PreAuthorize("hasRole('ADMIN or ...')")
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    private PasswordEncoder passwordEncoder;

    @Autowired
    private MyUserDetailsService myUserDetailsService;

    @Autowired
    public void globalUserDetails(final AuthenticationManagerBuilder auth) throws Exception
    {
        auth.userDetailsService(myUserDetailsService)
            .passwordEncoder(passwordEncoder);
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        ...
    }
}

```

1.7. Configuration type pour un projet de type Thymeleaf ou JSP

```

package .....;
....
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
//necessary for @PreAuthorize("hasRole('ADMIN or ...')")
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    ...

    @Override
    protected void configure(HttpSecurity http) throws Exception {

        http.authorizeRequests()
            .antMatchers("/",
                "/favicon.ico",
                "/*/*/*.png",
                "/*/*/*.gif",
                "/*/*/*.svg",
                "/*/*/*.jpg",
                "/*/*/*.css",
                "/*/*/*.map",
                "/*/*/*.js").permitAll()
            .antMatchers("/to-welcome").permitAll()
            .antMatchers("/session-end").permitAll()
            .antMatchers("/xyz").permitAll()
            .anyRequest().authenticated()
            .and().formLogin().permitAll()
            /*.and().formLogin()
                .loginPage("/login")
                .failureUrl("/login-error")
                .permitAll() */
            .and().csrf();
    }
}

```

1.8. Configuration type pour un projet de type "Api REST"

```

package .....;
....
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
//necessary for @PreAuthorize("hasRole('ADMIN or ...')")
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    ...

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http.authorizeRequests()
            .antMatchers("/", "/favicon.ico", "/*/*.*png", "/*/*.*gif", "/*/*.*svg",
                "/*/*.*jpg", "/*/*.*html", "/*/*.*css", "/*/*.*js").permitAll()
            .antMatchers(HttpMethod.POST, "/auth/**").permitAll()
            .antMatchers("/xyz-api/public/**").permitAll()
            .antMatchers("/xyz-api/private/**").authenticated()
            .and().cors() //enable CORS (avec @CrossOrigin sur class @RestController)
            .and().csrf().disable()
            // If the user is not authenticated, returns 401
            .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
            // This is a stateless application, disable sessions
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            .and()
            // Custom filter for authenticating users using tokens
            .addFilterBefore(jwtAuthenticationFilter,
                UsernamePasswordAuthenticationFilter.class);
    }
}
...

```

IV - Jta/Atomikos et spring JMS

1. JTA / Atomikos

1.1. Cadre général des transactions distribuées

Dans le monde "Java EE" :

- les EJB et les serveurs d'applications associées (ex : WebSphere AS, JBoss AS, ...) gèrent par défaut les transactions de façon sophistiquées (JTA/XA) avec potentiellement plusieurs bases de données .
Les transactions sont prises en charge par le serveur d'application (et ses connecteurs "JCA"). Certains paramétrages fondamentaux (DataSources selon bases de données , driver jdbc, ...) s'effectuent d'une manière très spécifique au type de serveur (ex : standalone.xml de jboss).
- Spring et SpringBoot gèrent par défaut les transactions de façon simple (sur une seule base de données en mode "RESOURCE_LOCAL").
Spring peut toutefois gérer des transactions distribuées (sur plusieurs bases différentes) via des extensions open-source compatibles JTA/Xa : "*Bitronix*" ou "*Atomikos*" .
Les paramétrages (pointus et peu classiques) sont à effectuer de manière explicite (en mode java-config) au sein de l'application spring-boot .

Dans les 2 contextes (JEE/JTA/EJB ou bien Spring/JTA/Bitronix_ou_Atomikos) , les transactions distribuées peuvent éventuellement faire intervenir des sous-traitements basés sur des technologies autres que les bases de données relationnelles , ex : file d'attente JMS) .

1.2. JTA

JTA = Java Transaction Api est une api standard du monde javaEE .

Cette api permet de contrôler des transactions distribuées (avec des sources de données en mode "xa" pour le commit à 2 phases) .

2. JTA/Atomikos intégré dans Spring et Spring-Boot

2.1. Configuration explicite "java-config-jta"

MyAtomikosJtaPlatform.java (classe utilitaire pour mécanismes JPA/Hibernate) :

```
package org.mycontrib.ext;

import javax.transaction.TransactionManager;
import javax.transaction.UserTransaction;
import org.hibernate.engine.transaction.jta.platform.internal.AbstractJtaPlatform;
```

```

/**
pour properties.put("hibernate.transaction.jta.platform",
    MyAtomikosJtaPlatform.class.getName());
dans paramétrage des "entityManagerFactory jpa"
*/

public class MyAtomikosJtaPlatform extends AbstractJtaPlatform {

    private static final long serialVersionUID = 1L;

    static protected TransactionManager transactionManager;
    static protected UserTransaction transaction;

    @Override
    public TransactionManager locateTransactionManager() {
        return transactionManager;
    }

    @Override
    public UserTransaction locateUserTransaction() {
        return transaction;
    }
}

```

JtaConfig.java (exemple de configuration explicite en mode "java-config")

```

package org.mycontrib.ext;

import javax.transaction.TransactionManager;
import javax.transaction.UserTransaction;
import org.slf4j.Logger; import org.slf4j.LoggerFactory; import java.util.HashMap;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.DependsOn;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.vendor.Database;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.transaction.transaction.PlatformTransactionManager;
import org.springframework.transaction.jta.JtaTransactionManager;
import com.atomikos.icatch.jta.UserTransactionImp;
import com.atomikos.icatch.jta.UserTransactionManager;

@Configuration
@ComponentScan
//sub-config in sub-packages : orders.OrdersConfig , customers.CustomersConfig , purchases.PurchasesConfig
public class JtaConfig {

    private static Logger logger = LoggerFactory.getLogger(JtaConfig.class);

    @Bean(name = "userTransaction")
    public UserTransaction userTransaction() throws Throwable {
        UserTransactionImp userTransactionImp = new UserTransactionImp();
        userTransactionImp.setTimeout(10000);
        return userTransactionImp;
    }
}

```

```

}

@Bean(name = "atomikosTransactionManager", initMethod = "init", destroyMethod = "close")
public TransactionManager atomikosTransactionManager() throws Throwable {
    UserTransactionManager userTransactionManager = new UserTransactionManager();
    userTransactionManager.setForceShutdown(false);
    MyAtomikosJtaPlatform.transactionManager = userTransactionManager;
    return userTransactionManager;
}

@Bean(name = "transactionManager")
@DependsOn({ "userTransaction", "atomikosTransactionManager" })
public PlatformTransactionManager jtaTransactionManager() throws Throwable {
    UserTransaction userTransaction = this.userTransaction();
    MyAtomikosJtaPlatform.transaction = userTransaction;
    TransactionManager atomikosTransactionManager = atomikosTransactionManager();
    return new JtaTransactionManager(userTransaction, atomikosTransactionManager);
}

//fonctions utilitaires (utilisée dans plusieurs autres classes):

public static Database vendorDataBaseFromXaDataSourceClassName(String xaDataSourceClassName) {
    Database db=null;
    switch(xaDataSourceClassName) {
        case "oracle.jdbc.xa.client.OracleXADataSource":
            db=Database.ORACLE; break;
        case "com.mysql.jdbc.jdbc2.optional.MysqlXADataSource":
        case "com.mysql.cj.jdbc.MysqlXADataSource":
            db=Database.MYSQL; break;
        case "org.postgresql.xa.PGXDataSource":
            db=Database.POSTGRESQL; break;
        case "org.h2.jdbcx.JdbcDataSource":
        default:
            db=Database.H2;
    }
    return db;
}

public static String hibernateDialectFromXaDataSourceClassName(String xaDataSourceClassName) {
    String hbDialect=null;
    switch(xaDataSourceClassName) {
        case "oracle.jdbc.xa.client.OracleXADataSource":
            hbDialect="org.hibernate.dialect.OracleDialect"; break;
        case "com.mysql.jdbc.jdbc2.optional.MysqlXADataSource":
        case "com.mysql.cj.jdbc.MysqlXADataSource":
            /* important : InnoDB engine for transaction, not MyISAM */
            hbDialect="org.hibernate.dialect.MySQL5InnoDBDialect"; break;
        case "org.postgresql.xa.PGXDataSource":
            hbDialect="org.hibernate.dialect.PostgreSQLDialect";break;
        case "org.h2.jdbcx.JdbcDataSource":
        default:
            hbDialect="org.hibernate.dialect.H2Dialect";
    }
    return hbDialect;
}

public static JpaVendorAdapter jpaVendorAdapterFromXaDataSourceClassName(
    String xaDataSourceClassName) {

    HibernateJpaVendorAdapter hibernateJpaVendorAdapter = new HibernateJpaVendorAdapter();
    //hibernateJpaVendorAdapter.setShowSql(true);
    Database db = JtaConfig.vendorDataBaseFromXaDataSourceClassName(xaDataSourceClassName);
    hibernateJpaVendorAdapter.setDatabase(db);//Database.H2 or .MYSQL or ...
}

```



```

        return hibernateJpaVendorAdapter;
    }

    public static HashMap<String, Object> jpaPropertiesFromXaDataSourceClassNameAndHibernateDdlAuto(
        String xaDataSourceClassName, String hibernateDdlAuto) {
        HashMap<String, Object> properties = new HashMap<String, Object>();
        properties.put("hibernate.transaction.jta.platform", MyAtomikosJtaPlatform.class.getName());
        properties.put("javax.persistence.transactionType", "JTA");
        properties.put("hibernate.hbm2ddl.auto", hibernateDdlAuto);
        String hbDialect =
            JtaConfig.hibernateDialectFromXaDataSourceClassName(xaDataSourceClassName);
        properties.put("hibernate.dialect", hbDialect);
        return properties;
    }
}

```

//selon contexte , si nécessaire (à priori non) , désactiver certaines configuration par défaut via
*/**

```

@EnableAutoConfiguration(exclude = {
    DataSourceAutoConfiguration.class,
    HibernateJpaAutoConfiguration.class, //if you are using Hibernate
    DataSourceTransactionManagerAutoConfiguration.class
})*/

```

customers/**CustomersConfig**.java (ou ici "customers" est une des bases de données)

```

package org.mycontrib.ext.customers;

import java.util.HashMap; import javax.sql.DataSource;
import org.mycontrib.ext.JtaConfig;
import org.slf4j.Logger; import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Value;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.boot.context.properties.EnableConfigurationProperties;
import org.springframework.boot.jta.atomikos.AtomikosDataSourceBean;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.DependsOn;
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;

@Configuration
@EnableConfigurationProperties
@DependsOn("transactionManager")
@EnableJpaRepositories(basePackages = "org.mycontrib.ext.customers.dao",
    entityManagerFactoryRef = "customersEntityManagerFactory",
    transactionManagerRef = "transactionManager")
public class CustomersConfig {
    private static Logger logger = LoggerFactory.getLogger(CustomersConfig.class);

    @Bean
    @ConfigurationProperties(prefix = "spring.jta.atomikos.datasource.customers")

```

```

public DataSource customersDataSource() {
    logger.trace("init customersDataSource in CustomersConfig");
    return new AtomikosDataSourceBean();
}

@Value("${spring.jpa.hibernate.ddl-auto}")
private String hibernateDdlAuto; // "none or "create" or ...

@Value("${spring.jta.atomikos.datasource.customers.xa-data-source-class-name}")
private String xaDataSourceClassName;

@Bean
public JpaVendorAdapter customersJpaVendorAdapter() {
    return JtaConfig.jpaVendorAdapterFromXaDataSourceClassName(xaDataSourceClassName);
}

@Bean(name = "customersEntityManagerFactory")
public LocalContainerEntityManagerFactoryBean customersEntityManagerFactory() throws Throwable {

    HashMap<String, Object> properties =
        JtaConfig.jpaPropertiesFromXaDataSourceClassNameAndHibernateDdlAuto(
            xaDataSourceClassName,hibernateDdlAuto);

    LocalContainerEntityManagerFactoryBean entityManagerFactory = new
        LocalContainerEntityManagerFactoryBean();
    entityManagerFactory.setJtaDataSource(customersDataSource());
    entityManagerFactory.setJpaVendorAdapter(customersJpaVendorAdapter());
    entityManagerFactory.setPackagesToScan("org.mycontrib.ext.customers.entity");
    entityManagerFactory.setPersistenceUnitName("customersPersistenceUnit");
    entityManagerFactory.setJpaPropertyMap(properties);
    return entityManagerFactory;
}
}

```

==> et autres classes identiques "orders/OrdersConfig.java" , "purchases/PurchasesConfig.java" pour les autres bases de données utilisées par l'application.

MySpringBootApplication.java (avec profiles)

```

...
@SpringBootApplication
public class MySpringBootApplication extends SpringBootServletInitializer {
    public static void main(String[] args) {
        SpringApplication app = new SpringApplication(MySpringBootApplication.class);
        app.setAdditionalProfiles("reInit","embeddedDb");//ok with H2

        //app.setAdditionalProfiles("noReInit","remoteDb");
        //ok with prepared mysql or postgres database in docker

        //one of "reInit" or "noReInit" profile is required
        //one of "embeddedDb" or "remoteDb" profile is required

        ConfigurableApplicationContext context = app.run(args);
        System.out.println("http://localhost:8181/spring-boot-backend");
    }
}

```

}

Exemple de configuration globale *application.properties*

```
server.servlet.context-path=/spring-boot-backend
server.port=8181
logging.level.org=INFO

spring.jta.enabled=true
spring.jta.service=com.atomikos.icatch.standalone.UserTransactionServiceFactory
spring.jta.max-actives=200
spring.jta.enable-logging=false

# ==> others JTA properties xa/datasource in application-embeddedDb.properties
#                                     or application-remoteDb.properties

#enable spring-data (generated dao implementation classes)
spring.data.jpa.repositories.enabled=true
```

2.2. Configuration en mode test/H2

```
app.setAdditionalProfiles("reInit","embeddedDb");//ok with H2
```

application-reInit.properties

```
spring.jpa.hibernate.ddl-auto=create
```

application-embeddedDb.properties

```
#JDBC settings for (h2) embedded dataBases
# ici pour 3 bases de données "customers", "orders" et "purchases" :
spring.jta.atomikos.datasource.customers.unique-resource-name=customersDataSource
spring.jta.atomikos.datasource.customers.max-pool-size=5
spring.jta.atomikos.datasource.customers.min-pool-size=1
spring.jta.atomikos.datasource.customers.max-life-time=25000
spring.jta.atomikos.datasource.customers.borrow-connection-timeout=10000
spring.jta.atomikos.datasource.customers.xa-data-source-class-name=org.h2.jdbcx.JdbcDataSource
spring.jta.atomikos.datasource.customers.xa-properties.user=sa
spring.jta.atomikos.datasource.customers.xa-properties.password=
spring.jta.atomikos.datasource.customers.xa-properties.URL=
jdbc:h2:~/customers;DB_CLOSE_ON_EXIT=FALSE
```

```

spring.jta.atomikos.datasource.orders.unique-resource-name=ordersDataSource
...
spring.jta.atomikos.datasource.orders.xa-properties.URL=
    jdbc:h2:~/orders;DB_CLOSE_ON_EXIT=FALSE

spring.jta.atomikos.datasource.purchases.unique-resource-name=purchasesDataSource
...
spring.jta.atomikos.datasource.purchases.xa-properties.URL=
    jdbc:h2:~/purchases;DB_CLOSE_ON_EXIT=FALSE

```

```

@ExtendWith(SpringExtension.class)
@SpringBootTest(classes= {MySpringBootApplication.class})
@ActiveProfiles("reInit,embeddedDb")
public class TestOrderAndPurchaseService {
...
}

```

2.3. Configuration en mode prod/Mysql & postgres

```
app.setAdditionalProfiles("noReInit","remoteDb");
```

application-noReInit.properties

```
spring.jpa.hibernate.ddl-auto=none
```

application-remoteDb.properties

```

# JDBC settings for (h2) embedded dataBases
# ici pour 3 bases de données "customers" avec mysql, "orders" et "purchases" avec postgres:
spring.jta.atomikos.datasource.customers.unique-resource-name=customersDataSource
spring.jta.atomikos.datasource.customers.max-pool-size=5
spring.jta.atomikos.datasource.customers.min-pool-size=1
spring.jta.atomikos.datasource.customers.max-life-time=25000
spring.jta.atomikos.datasource.customers.borrow-connection-timeout=10000
spring.jta.atomikos.datasource.customers.xa-properties.pinGlobalTxToPhysicalConnection=true
spring.jta.atomikos.datasource.customers.xa-data-source-class-name=
    com.mysql.cj.jdbc.MySQLXADataSource
spring.jta.atomikos.datasource.customers.xa-properties.user=root
spring.jta.atomikos.datasource.customers.xa-properties.password=root
spring.jta.atomikos.datasource.customers.xa-properties.URL=

```

```
jdbc:mysql://127.0.0.1:3306/customers?createDatabaseIfNotExist=true&serverTimezone=UTC
```

```
spring.jta.atomikos.datasource.orders.unique-resource-name=ordersDataSource
spring.jta.atomikos.datasource.orders.max-pool-size=5
spring.jta.atomikos.datasource.orders.min-pool-size=1
spring.jta.atomikos.datasource.orders.max-life-time=25000
spring.jta.atomikos.datasource.orders.borrow-connection-timeout=10000
spring.jta.atomikos.datasource.orders.xa-data-source-class-name=org.postgresql.xa.PGXADDataSource
spring.jta.atomikos.datasource.orders.xa-properties.user=postgres
spring.jta.atomikos.datasource.orders.xa-properties.password=root
spring.jta.atomikos.datasource.orders.xa-properties.URL=jdbc:postgresql://localhost:5432/orders

spring.jta.atomikos.datasource.purchases.unique-resource-name=purchasesDataSource
...
spring.jta.atomikos.datasource.purchases.xa-properties.URL=jdbc:postgresql://localhost:5432/purchases
```

NB :

- **xa-properties.pinGlobalTxToPhysicalConnection=true** for *MysqlXADataSource* only, not H2 , not PGXADDataSource
- le serveur **Postgres** doit être démarré avec l'option ***max_prepared_transactions=64*** (pas =0 par défaut)

2.4. Exemple de service transactionnel avec JTA

Organisation possibles des packages java :

```

v [icon] > src/main/java
  v [icon] > org.mycontrib.ext
    > [icon] JtaConfig.java
    > [icon] MyAtomikosJtaPlatform.java
    > [icon] MySpringBootApplication.java
    > [icon] WebSecurityConfig.java
    [icon] NoJtaConfig.java.notUsed.txt
  v [icon] > org.mycontrib.ext.customers
    > [icon] CustomersConfig.java
  v [icon] org.mycontrib.ext.customers.dao
    > [icon] AddressRepository.java
    > [icon] CustomerRepository.java
  v [icon] org.mycontrib.ext.customers.entity
    > [icon] Address.java
    > [icon] Customer.java

```

et idem pour autres bases "orders" et "purchases"

Dao classique (rien de spécial) avec Spring-Data :

```
...
public interface CustomerRepository extends JpaRepository<Customer,Long>{
```

```

    Customer findByEmail(String email);
}

```

org.mycontrib.ext.global.service.**OrderAndPurchaseServiceImpl.java**

```

package org.mycontrib.ext.global.service;

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

@Transactional
//@Transactional("transactionManager") by default ("transactionManager" = JTA in this app )
@Service
public class OrderAndPurchaseServiceImpl implements OrderAndPurchaseService {
    @Autowired
    private CustomerRepository customerRepository;

    @Autowired
    private PurchaseRepository purchaseRepository;

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private ProductRefRepository productRefRepository;

    @Override
    public Long purchaseOrder(Long customerId, List<ProductRef> listOfProducts) {
        Long orderId=null;
        Order newOrder = orderRepository.save(new Order(null,new Date(), customerId ));
        orderId= newOrder.getOrderId();

        Map<Integer,OrderLine> mapOrderLines = new HashMap<Integer,OrderLine>();
        int i=0; double prixTotal = 0;
        for(ProductRef prod : listOfProducts){ i++;
            productRefRepository.save(prod);
            OrderLine orderLine =new OrderLine(null,prod,1 /*quantity*/);
            orderLine.setOrderId(orderId);
            orderLine.setLineNumber(i);
            mapOrderLines.put(i, orderLine);
            prixTotal+=prod.getPrice();
        }

        newOrder.setOrderLines(mapOrderLines); newOrder.setTotalPrice(prixTotal);

        purchaseRepository.save(new Purchase(null,new Date(),customerId , prixTotal));

        //vérification de l'existence du client:
        Customer customer = customerRepository.findById(customerId).orElse(null);
        if(customer==null){
            throw new RuntimeException("customer not exists with id="+customerId);
            //transaction will be rollback (all in jta mode)
        }

        orderRepository.save(newOrder) ; System.out.println("savedOrder: "+newOrder.toString());
    }
}

```

```

        return orderId;
    }
}

```

TestOrderAndPurchaseService.java

```

package org.mycontrib.api.test;
...
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.junit.jupiter.SpringExtension;

@ExtendWith(SpringExtension.class)
@SpringBootTest(classes= {MySpringBootApplication.class})
@ActiveProfiles("reInit,embeddedDb")
public class TestOrderAndPurchaseService {
    private static Logger logger = LoggerFactory.getLogger(TestOrderAndPurchaseService.class);
    @Autowired
    private OrderAndPurchaseService orderAndPurchaseService ;
    @Autowired
    private OrderRepository orderRepository ;
    @Autowired
    private ProductRefRepository productRefRepository;
    @Autowired
    private PurchaseRepository purchaseRepository;

    @Test
    public void testPurchaseOrderForExistingCustomer() throws Exception{
        List<ProductRef> listOfProductRef = new ArrayList<ProductRef>();
        ProductRef prA = productRefRepository.findById(5L)
            .orElse(new ProductRef(5L,"stylo bille noir " , 1.5));
        listOfProductRef.add(prA);
        ProductRef prB = productRefRepository.findById(6L)
            .orElse(new ProductRef(6L,"cahier 48 pages " , 2.5));
        listOfProductRef.add(prB);

        //a tester avec customerId=1L (existant) with reInit profile , avec bases h2 ou mysql & postgres
        Long newOrderId = orderAndPurchaseService.purchaseOrder(1L, listOfProductRef);
        Assertions.assertNotNull(newOrderId);

        Order newOrder = orderRepository.findById(newOrderId).orElse(null);
        Assertions.assertTrue(newOrder.getOrderLines().keySet().size()==2);
        logger.info("new order : " + newOrder.toString());
        for(Integer numLine : newOrder.getOrderLines().keySet() ){
            logger.info("\t" + numLine + " : " + newOrder.getOrderLines().get(numLine));
        }
    }

    @Test
    public void testPurchaseOrderForNotExistingCustomer() throws Exception{
        List<ProductRef> listOfProductRef = new ArrayList<ProductRef>();
        ProductRef prA = productRefRepository.findById(7L)
            .orElse(new ProductRef(7L,"stylo bille rouge " , 1.6));
        listOfProductRef.add(prA);
        ProductRef prB = productRefRepository.findById(8L)
            .orElse(new ProductRef(8L,"cahier96 pages " , 2.9));
        listOfProductRef.add(prB);

        long nbOrdersBeforePurchaseOrder = orderRepository.count();
    }
}

```

```

    long nbPurchasesBeforePurchaseOrder = purchaseRepository.count();

    try {
        //a tester avec customerId=999L (non existant) with reInit profile
        Long newOrderId = orderAndPurchaseService.purchaseOrder(999L, listOfProductRef);
        Assertions.fail("une exception aurait du remonter");
    } catch (RuntimeException e) {
        logger.info("exception attendue:" + e);
    }
    //tester le bon rollback :
    long nbOrdersAfterPurchaseOrder = orderRepository.count();
    Assertions.assertTrue(nbOrdersAfterPurchaseOrder==nbOrdersBeforePurchaseOrder);

    long nbPurchasesAfterPurchaseOrder = purchaseRepository.count();
    Assertions.assertTrue(nbPurchasesAfterPurchaseOrder==nbPurchasesBeforePurchaseOrder);
}
}

```

ReInitCustomersOrdersPurchasesDefaultDataSet.java (avec profile "reInit") :

```

...
@Component
@Profile("reInit")
public class ReInitCustomersOrdersPurchasesDefaultDataSet {
    @Autowired
    private CustomerRepository customerRepository;
    ....
    @Autowired
    private PurchaseRepository purchaseRepository;

    @PostConstruct
    public void initDataSet() {
        //new Address(Long id, String numberAndStreet, String zip, String town, String country)
        Address a1 = new Address(null,"8 rue elle" , "75000" , "Paris" , "France");
        addressRepository.save(a1);
        //new Customer(Long id, String firstName, String lastName, String email, String phoneNumber)
        Customer c1 = new Customer(null,"alex" , "Therieur" , "alex-therieur@iciOula.fr" , "0102030405");
        c1.setAddress(a1);
        customerRepository.save(c1);

        //new ProductRef(Long productId, String label, double price)
        ProductRef pr1 = new ProductRef(1L,"smartPhone xy" , 120.5);
        productRefRepository.save(pr1);
        ProductRef pr2 = new ProductRef(2L,"micro SD memory card" , 8.0);
        productRefRepository.save(pr2);
        //new Order(Long orderId, Date orderDate, Long cutomerId)
        Order o1 = new Order(null,new Date() , c1.getId());
        orderRepository.save(o1); //first call to save() for initialize auto_incr orderId
        o1.addOrderLine(pr1,1);//(productRef,quantity)
        o1.addOrderLine(pr2,3);//(productRef,quantity)
        orderRepository.save(o1); //second call to save() for saving orderLines

        //new Purchase(Long purchaseId, Date purchaseDateTime, Long cutomerId, double amount)
        Purchase p1 = new Purchase(null,new Date() , c1.getId() , o1.getTotalPrice());
        purchaseRepository.save(p1);
    }
}

```


3. Repères JMS

JMS (Java Message Service)

JMS est une API permettant de faire **dialoguer des applications** de façon **asynchrone**.

Architecture associée: **MOM** (Message Oriented MiddleWare).

NB: JMS n'est qu'une API qui sert à accéder à un véritable fournisseur de Files de messages (ex: MQSeries/Websphere_MQ d'IBM, ActiveMQ d'apache, ...)

Dans la terminologie JMS, les Clients JMS sont des programmes Java qui envoient et reçoivent des messages dans/depuis une file (**message queue**).

Une file de message sera gérée par un "Provider JMS".

Les clients utiliseront **JNDI** pour accéder à une file.

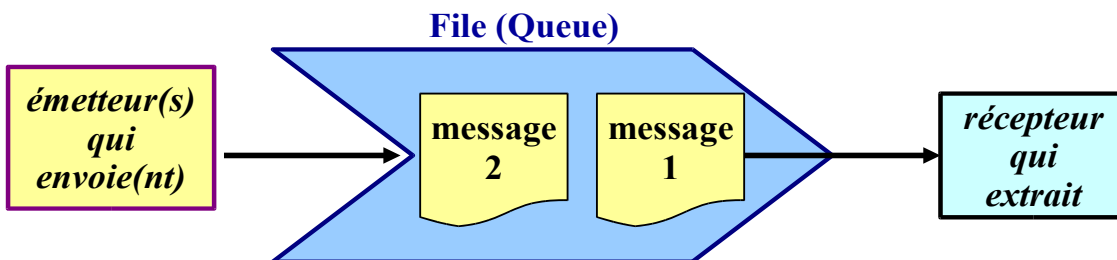
L'objet **ConnectionFactory** sera utilisé pour établir une connexion avec une file.

L'objet **Destination** (*File* ou *Topic*) sert à préciser la destination d'un message que l'on envoie ou bien la source d'un message que l'on souhaite récupérer.

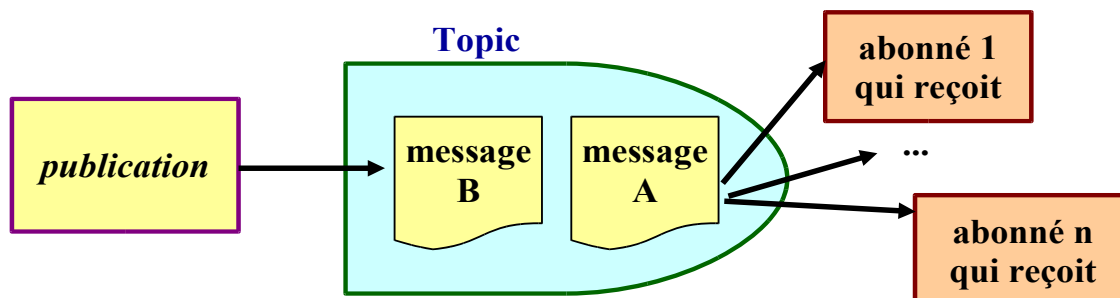
JMS permet de mettre en oeuvre les 2 modèles suivants:

- **PTP** (Point To Point)
- **Pub/Sub** (Published & Subscribe) .../...

JMS Queue : Point To Point



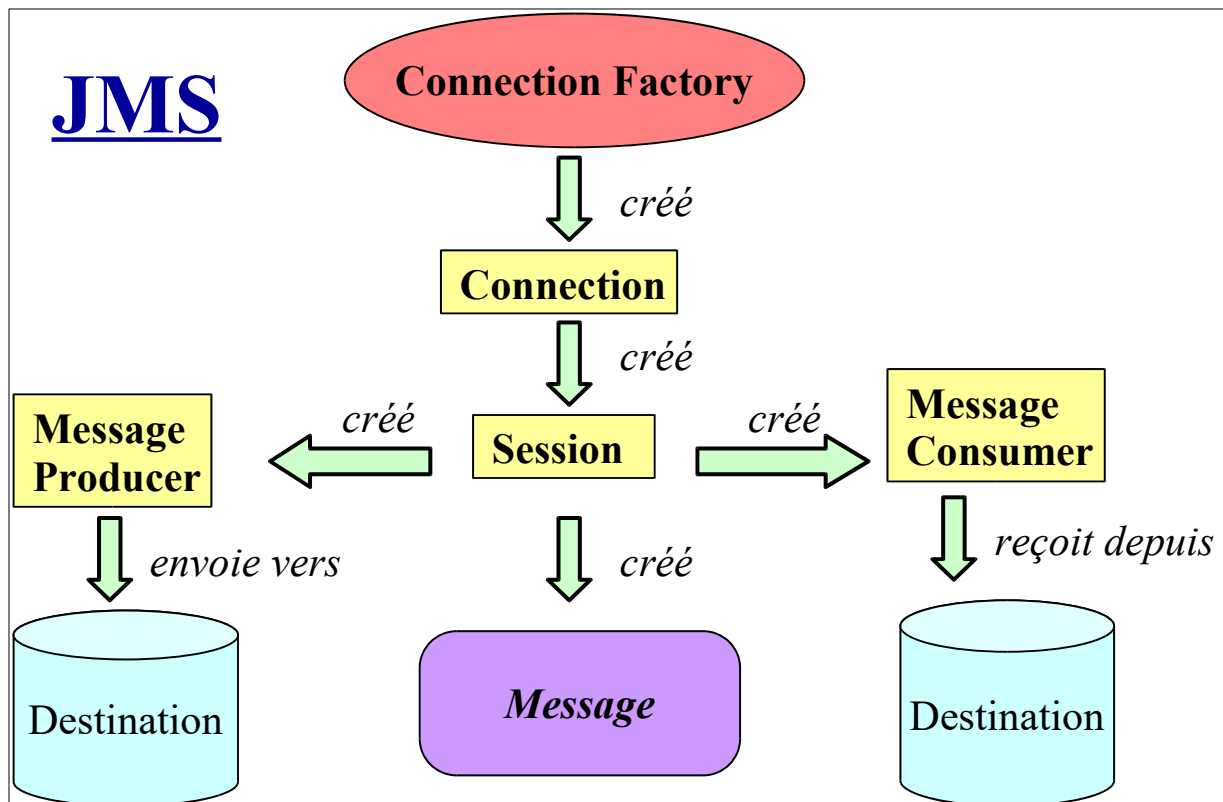
JMS Topic : Publish / Subscribe



Le tableau ci-dessous résume les différentes **interfaces** utilisées au niveau de l'api JMS:

JSM (Interface générique)	PTP Domain	Pub/Sub Domain
ConnectionFactory (mt)	QueueConnectionFactory	TopicConnectionFactory
Connection (mt)	QueueConnection	TopicConnection
Destination (mt)	Queue	Topic
Session	QueueSession	TopicSession
MessageProducer	QueueSender	TopicPublisher
MessageConsumer	QueueReceiver, QueueBrowser	TopicSubscriber

(mt) : multi-threading support.



Champs des entêtes de message :

Champ de l'entête	signification	fixé (affecté) par
JMSDestination	File de destination	méthode send()
JMSDeliveryMode	PERSISTENT ou NON PERSISTENT	méthode send()
JMSExpiration	0 : pas d'expiration. sinon <i>n ms</i> à vivre.	méthode send()
JMSPriority	priorité de 0 à 9 (0-4: normal) (5-9: high)	méthode send()
JMSMessageID	ID:xxx identifiant du message	méthode send()
JMSTimestamp	estampillage de temps	méthode send()
JMSCorrelationID	identifiant de la requête associée à la réponse	Client
JMSReplyTo	File où il faut placer la réponse.	Client
JMSType	selon le contexte , catégorie , ...	Client
JMSRedelivered	si réception multiple d'un même message	Provider

Le champ **JMSReplyTo** peut comporter le nom d'une file (éventuellement temporaire) que l'émetteur de la requête a préalablement créé pour récupérer la réponse..

3.1. ActiveMq

Url de la console web : <http://localhost:8161/admin/>

default username/password : admin/admin

Scripts à écrire et lancer dans D:\...\JMS\apache-activemq-5.15.11\bin

start_activeMq.bat

echo console will be available at <http://localhost:8161/admin>

activemq start

stop_activeMq.bat

activemq stop

et menu "Queues" pour observer (et éventuellement ajuster) les files de messages et leurs contenus (messages)

ActiveMQ

Home | Queues | Topics | Subscribers | Connections | Network | Scheduled | Send

Queue Name Queue Name Filter

Queues:

Name ↑	Number Of Pending Messages	Number Of Consumers	Messages Enqueued	Messages Dequeued	Views	Operations
MyDataQueue	2	0	0	0	Browse Active Consumers Active Producers 	Send To Purge Delete
MyForwardDataQueue	0	0	0	0	Browse Active Consumers Active Producers 	Send To Purge Delete

Browse MyDataQueue

Message ID ↑	Correlation ID	Persistence	Priority	Redelivered	Reply To	Timestamp	Type	Operations
ID:LAPTOP-DDC-56968-1579078028066-1:1:1:1:1		Persistent	4	false		2020-01-15 09:47:08:562 CET		Delete
ID:LAPTOP-DDC-62094-1579081908336-1:1:1:1:1		Persistent	4	false		2020-01-15 10:51:48:744 CET		Delete

[View Consumers](#)

3.2. Artemis (nouvelle génération de ActiveMq)

dans **artemis/bin**

écrire et lancer createArtemisBroker.bat

```
./artemis create ../brokers/my-broker --user=admin --password=admin --allow-anonymous
pause
```

dans **artemis/brokers/my-broker/bin**

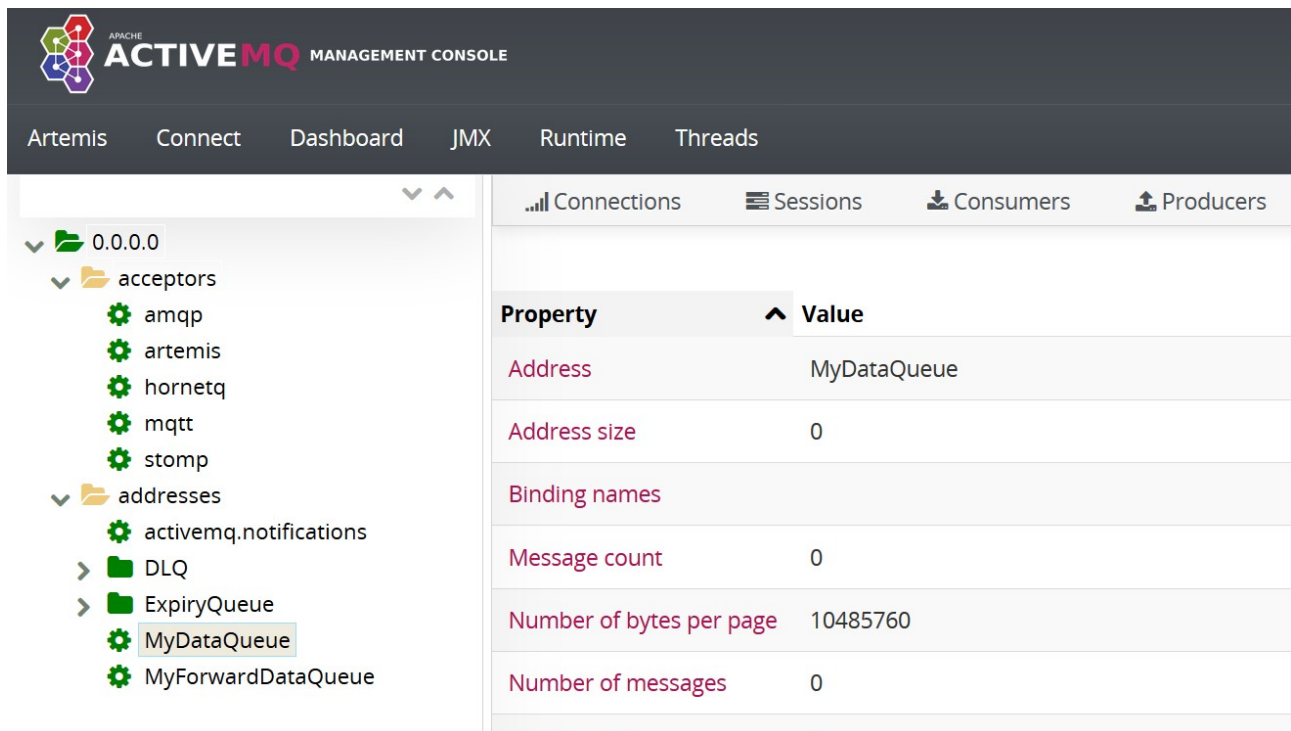
artemis run

artemis stop

Console admin (artemis)

http://localhost:8161/console

username/password : **admin/admin**



The screenshot shows the Apache ActiveMQ Management Console interface. The left sidebar displays a tree view of the broker's configuration, with 'MyDataQueue' selected under the 'addresses' folder. The main panel shows the configuration for 'MyDataQueue' with the following properties:

Property	Value
Address	MyDataQueue
Address size	0
Binding names	
Message count	0
Number of bytes per page	10485760
Number of messages	0

4. intégration JMS dans Spring

4.1. Configuration "JMS avec Spring-boot"

pom.xml

```
...
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-artemis</artifactId>
    </dependency>
<!-- <dependency> -->
<!-- <groupId>org.springframework.boot</groupId> -->
<!-- <artifactId>spring-boot-starter-activemq</artifactId> -->
<!-- </dependency> -->
<!-- et indirectement spring-jms -->

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
</dependency>    <!-- si besoin de JSON dans message JMS -->
```

application.properties

```
server.servlet.context-path=/springBootJms
server.port=8484
logging.level.org=INFO

spring.artemis.mode=native
spring.artemis.host=localhost
spring.artemis.port=61616
spring.artemis.user=admin
spring.artemis.password=admin

#spring.activemq.user=admin
#spring.activemq.password=admin
#spring.activemq.broker-url=tcp://localhost:61616?jms.redeliveryPolicy.maximumRedeliveries=1
```

MySpringBootApplication --> comme d'habitude avec @SpringBootApplication

JmsConfig.java

```

package org.mycontrib.xyz;
import javax.jms.ConnectionFactory;
import org.springframework.boot.autoconfigure.jms.DefaultJmsListenerContainerFactoryConfigurer;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jms.annotation.EnableJms;
import org.springframework.jms.config.DefaultJmsListenerContainerFactory;
import org.springframework.jms.config.JmsListenerContainerFactory;
import org.springframework.jms.support.converter.MappingJackson2MessageConverter;
import org.springframework.jms.support.converter.MessageConverter;
import org.springframework.jms.support.converter.MessageType;

@Configuration
@EnableJms
public class JmsConfig {

    //NB: spring.activemq.... properties in application.properties

    // Only required due to defining myFactory in the receiver
    @Bean
    public JmsListenerContainerFactory<?> myFactory(
        ConnectionFactory connectionFactory,
        DefaultJmsListenerContainerFactoryConfigurer configurer) {
        DefaultJmsListenerContainerFactory factory = new DefaultJmsListenerContainerFactory();
        factory.setErrorHandler(t -> System.err.println("An error has occurred (jms/activemq)"));
        configurer.configure(factory, connectionFactory);
        return factory;
    }

    // Serialize message content to json using TextMessage
    @Bean
    public MessageConverter jacksonJmsMessageConverter() {
        MappingJackson2MessageConverter converter = new MappingJackson2MessageConverter();
        converter.setTargetType(MessageType.TEXT);
        converter.setTypeIdPropertyName("_type");
        return converter;
    }
}

```

org.mycontrib.xyz.dto.MyData.java

```

@Getter @Setter @NoArgsConstructor @ToString
public class MyData {
    private String ref;
    private Double value;

    public MyData(String ref, Double value) {
        super();
        this.ref = ref;
        this.value = value;
    }
}

```

MyDataJmsReceiver.java (***pour réception des messages***)

```

package org.mycontrib.xyz.jms;

import javax.jms.Message;
import org.mycontrib.xyz.dto.MyData;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.annotation.JmsListener;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.stereotype.Component;

@Component
public class MyDataJmsReceiver {

    @JmsListener(destination = "MyDataQueue", containerFactory = "myFactory")
    public void receiveMessage(MyData data, Message msg) {
        System.out.println("JMS Message received: "+msg);
        System.out.println("Received <" + data + ">");
        //...
        forwardData(data);
    }

    @Autowired
    private JmsTemplate jmsTemplate; //for re-sending / forwarding message in other queue

    private void forwardData(MyData data){
        jmsTemplate.convertAndSend("MyForwardDataQueue", data);
    }
}

```

MyDataRestCtrl.java (WS REST qui envoie des messages dans une file JMS)

```

package org.mycontrib.xyz.rest;

import org.mycontrib.xyz.dto.MyData;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping(value="my-api/data" , headers="Accept=application/json")
public class MyDataRestCtrl {

    @Autowired
    private JmsTemplate jmsTemplate;

    //POST http://localhost:8484/springBootJms/my-api/data
    // { "ref" : "EUR" , "value" : 0.923 }
    @PostMapping("")

```

```

public MyData send(@RequestBody MyData data) {
    System.out.println("Sending data in queue as jms message");
    // send message to the message queue named "MyDataQueue"
    jmsTemplate.convertAndSend("MyDataQueue", data);
    return data;
}
}

```

page src/main/resources/static/**index.html** pour envoyer des données au WS REST

```

<html>
<head><meta charset="ISO-8859-1"><title>Index majeur</title></head>
<body>
    <h1>welcome to springBootJsF</h1>
    <pre>
cette mini application ne fonctionne bien que si le serveur/agent/broker
"activemq" ou "artemis" est préalablement démarré .
Les messages envoyés en mode POST vers l'api REST seront automatiquement envoyés
vers une file JMS dénommée "MyDataQueue"
Le composant spring "MyDataJmsReceiver" de cette application récupère/extrait
les messages de la file "MyDataQueue" et les stocke dans une autre file
nommée "MyForwardDataQueue" .
Via la console de activemq
(url=http://localhost:8161/admin,username=admin,password=admin)
on pourra visualiser les messages accumulés dans la file "MyForwardDataQueue".
La console artemis (url=http://localhost:8161/console) est moins complète.
Sachant que ces messages seraient récupérables par n'importe quelle autre application
connectée à l'agent "activemq" et à la file "MyForwardDataQueue" .
    </pre>
    <hr/>
    ref: <input type="text" id="txtRef" /> (as string) <br/>
    value: <input type="text" id="txtValue" /> (as number) <br/>
    <input type="button" id="btnPostData" value="post data" /> <br/>
    <span id="spanMsg"></span>
</body>
<script>
function makePostAjaxRequest(url,obj,callback) {
    var xhr = new XMLHttpRequest();
    xhr.onreadystatechange = function() {
        if (xhr.readyState == 4 && (xhr.status == 200 || xhr.status == 0)) {
            callback(xhr.responseText);
        }
    };
    xhr.open("POST", url, true);
    xhr.setRequestHeader("Content-Type", "application/json");
    xhr.send(JSON.stringify(obj));
}

var inputRef=document.getElementById("txtRef");
var inputValue=document.getElementById("txtValue");
var btnPostData=document.getElementById("btnPostData");
var spanMsg=document.getElementById("spanMsg");

```



```

btnPostData.addEventListener("click", function(){
    var dataObj = { ref : null , value : 0 };
    dataObj.ref = inputRef.value;
    dataObj.value = Number(inputValue.value);
    var url="/my-api/data";
    makePostAjaxRequest(url,dataObj,function(savedData){
        spanMsg.innerHTML="savedData="+savedData;
    });
});
</script>
</html>

```

<http://localhost:8484/springBootJms/>

ref: (as string)
 value: (as number)

 savedData={"ref":"r1","value":123456.0}

Dans console java :

Sending data in queue as jms message
JMS Message received: ActiveMQMessage[ID:78b6dc98-4843-11ea-af7b-0a0027000002]:PERSISTENT/ClientMessageImpl[messageID=8589934697, durable=true, address=MyDataQueue,userID=78b6dc98-4843-11ea-af7b-0a0027000002,properties=TypedProperties[__AMQ_CID=5ffa28b4-4843-11ea-af7b-0a0027000002,_type=org.mycontrib.xyz.dto.MyData,_AMQ_ROUTING_TYPE=1]]
 Received <MyData(ref=r1, value=123456.0)>

4.2. Application java externe qui envoie des messages

MyOtherJmsAppSendingMessage.java

```

package org.mycontrib.xyz;

import javax.jms.Connection;
import javax.jms.Destination;
import javax.jms.MessageProducer;
//import javax.jms.Queue;
import javax.jms.Session;
import javax.jms.TextMessage;
import org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory;
//import org.apache.activemq.artemis.jms.client.ActiveMQConnectionFactory;
//import org.apache.activemq.ActiveMQConnectionFactory;
import org.mycontrib.xyz.dto.MyData;

```

```

import com.fasterxml.jackson.databind.ObjectMapper;

public class MyOtherJmsAppSendingMessage {
    public static void main(String[] args) {
        try {
            ActiveMQConnectionFactory amqConnectionFactory =
                new ActiveMQConnectionFactory("tcp://localhost:61616"/*"vm://localhost"*/);

            //Connection : QueueConnection or TopicConnection
            Connection jmsCn = amqConnectionFactory.createConnection("admin","admin");

            Session jmsSession = jmsCn.createSession(false,
                Session.AUTO_ACKNOWLEDGE);

            //Destination : Queue or Topic
            /*Queue*/ Destination myDataQueue =
                jmsSession.createQueue("MyDataQueue"); //open existing queue or create new one

            TextMessage msg = jmsSession.createTextMessage();
            ObjectMapper jacksonObjectMapper = new ObjectMapper();
            MyData data = new MyData("ref1",123.456);
            msg.setText(jacksonObjectMapper.writeValueAsString(data));
            msg.setStringProperty("_type", data.getClass().getName());

            //queueSender = queueSession.createSender(queue); queueSender.send(msg);
            //topicPublisher topicSession.createPublisher(topic); ....
            //MessageProducer msgProducer = jmsSession.createProducer() for queue or topic
            MessageProducer msgProducer = jmsSession.createProducer(myDataQueue);
            msgProducer.send(msg);
            System.out.println("Message sent successfully to remote queue.");

            jmsSession.close(); jmsCn.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

```

---> message reçu dans la console du serveur spring/jms :

```

JMS Message received:ActiveMQMessage[ID:03e55ad3-4844-11ea-9241-
0a0027000002]:PERSISTENT/ClientMessageImpl[messageID=8589935173, durable=true,
address=MyDataQueue,userID=03e55ad3-4844-11ea-9241-
0a0027000002,properties=TypedProperties[__AMQ_CID=03c305c0-4844-11ea-9241-
0a0027000002,_type=org.mycontrib.xyz.dto.MyData,_AMQ_ROUTING_TYPE=1]]
Received <MyData(ref=ref1, value=123.456)>

```

V - Divers, Spring web flow, Spring intégration

1. Divers aspects secondaires ou avancés de Spring

1.1. Plug eclipse "Spring IDE" et STS (Spring Tools Suite)

En ajoutant le plugin eclipse "Spring-IDE" ou bien en téléchargeant STS (eclipse avec tous les plug-ins pour spring), on bénéficie de quelques assistants pratiques :

- création de nouveaux projets "Spring maven" avec les bonnes dépendances pour Spring
- auto-complétion des balises "spring" (avec aide contextuelle)
- gestion assistées des namespaces spring supplémentaires (aop, tx, ...)
- ...

1.2. Détails sur Autowired

`<bean autowire-candidate="false" .../>` pour qu'un bean ne soit pas utilisé par `@Autowired` ou ...

`@Autowired(required="false")` // `required="true"` by default.

```
....
<context:annotation-config/>
<context:component-scan base-package="tp.service">
  <context:exclude-filter type="regex"
    expression="tp\.service\..*V1" />
</context:component-scan>
```

1.3. Bean (config spring) abstrait et héritage

On peut définir un `<bean id="..." ...>` de spring avec l'attribut `abstract="true"`

Dans ce cas la précision `class="..."` n'est pas obligatoire

Un bean "spring" abstrait correspond simplement à un paquet de paramétrages (au niveau des propriétés) qui pourra être réutilisé via un héritage au niveau d'un futur bean concret :

```
<bean id="beanConcret" parent="inheritedBeanId" class="..." >
  <!-- héritage automatique de toutes les propriétés héritées
```

```

    avec redéfinitions possibles -->
    <property name="..." value="..." /> <!-- autre(s) propriétés -->
</bean>

```

Exemple :

```

<bean abstract="true" id="produitOrdinaireAbstract">
    <property name="tauxTva" value="20" />
</bean>

<bean id="produitPrototype" scope="prototype"
parent="produitOrdinaireAbstract"
class="tp.data.Produit">
    <property name="numero" value="0" />
    <property name="prix" value="0.0" />
</bean>

```

1.4. Internationalisation gérée par Spring (MessageSource)

```

<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basenames">
        <list>
            <value>locale/messages/customer</value>
            <value>locale/messages/errors</value>
        </list>
    </property>
</bean>

```

NB : il faut absolument fixer id="messageSource" , plusieurs implémentations possibles pour la classe d'implémentation (ici "ResourceBundleMessageSource" pour fichier ".properties")

Dans *src* ou *src/main/resources* : locale/messages/**customer_xx.properties** et **errors_xx.properties**
customer_**fr**.properties

```
customer.description={0} a {1} ans et habite à {2}
```

customer_en.properties

```
customer.description={0} is {1} years old and lives in {2}
```

Accès aux messages via le context spring :

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations={"/beans.xml"})
public class TestMessageSource {
    @Autowired
    private ApplicationContext context;

    @Test
    public void testMessageSource(){
        String frenchDescription = context.getMessage("customer.description",
            new Object[] { "didier" , 28,"vernon" }, Locale.FRENCH);
        System.out.println("Customer description (french) : " + frenchDescription);

        String englishDescription = context.getMessage("customer.description",
            new Object[] { "didier" , 28,"vernon" }, Locale.ENGLISH);
        System.out.println("Customer description (english) : " + englishDescription);
    }
}
```

Injection éventuelle de "messageSource" au sein d'un service :

//@Component

```
public class ServiceProduitV1 implements ServiceProduit , MessageSourceAware{
    private MessageSource messageSource;

    @Override //méthode appelée automatiquement par Spring pour injecter bean
    //dont id="messagesource" car implements MessageSourceAware
    public void setMessageSource(MessageSource messageSource) {
        this.messageSource= messageSource;
    }

    public Produit getProduitByNum(Long num) {
        ... messageSource.getMessage("message.suffix", null, Locale.FRENCH));
        ....
    }
}
```

```
}
```

1.5. Profiles "spring"

```
<beans profile="dev" > ... </beans>
```

```
<beans profile="dev" > ... </beans>
```

Choix du profile par défaut :

```
<context-param>
  <param-name>spring.profiles.default</param-name>
  <param-value>production</param-value>
</context-param>
```

```
-Dspring.profiles.default=production
```

Choix du ou des profiles à utiliser au niveau d'une classe de test :

```
@ActiveProfile ("dev")
```

ou bien

```
@ActiveProfile ("dev,profile2")
```

1.6. Divers aspects avancés (cas pointus)

`@DirtiesContext()` au dessus d'une méthode d'un test unitaire permet de demander à ce que le context "spring" soit rechargé à la fin de l'exécution de la méthode de test (et avant l'exécution des autres méthodes de test) → attention aux performances !!!!

Une classe java (correspondant à un composant "spring") peut éventuellement implémenter certaines interfaces de Spring :

- org.springframework.beans.factory.**BeanNameAware**
- org.springframework.beans.factory.**BeanFactoryAware**
- org.springframework.context.**ApplicationContextAware**
- org.springframework.beans.factory.**InitializingBean**
- org.springframework.beans.factory.**DisposableBean**

Ceci permettra à spring d'appeler automatiquement des "callback" sur le composant java (si détection de l'implémentation d'une des interfaces via `if instanceof`)

Par exemple, une classe java "MyCustomFactory" qui implémente "**BeanFactoryAware**" aura automatiquement accès au "BeanFactory/ApplicationContext" car celui-ci sera automatiquement injecté par spring via la méthode **setBeanFactory()** imposée par l'interface.

Et finalement cette fabrique spécialisée pourra construire des "beans" d'une façon ou d'une autre (en se basant sur certains paramétrages et en s'appuyant sur la méga-fabrique de spring).

A vérifier : la balise **@Bean** placée au dessus d'une méthode de fabrication de bean semble faire comprendre à spring que cette méthode est capable de générer des "beans" dont de type correspond au type de retour de la méthode de fabrication (sorte d'équivalent de **@Produces** de CDI/JEE6) . Les beans ainsi fabriqués seront des candidats pour de l'autowiring .

Il est possible d'enrichir le comportement du framework Spring via des implémentations (à enregistrer) de l'interface **BeanPostProcessor** . Ceci permet d'avoir la main sur un bean au moment de son initialisation pour si besoin l'enrichir ou le construire de façon très particulière.

Dans un même ordre d'idée, il est possible d'enregistrer (via `<context:component-scan base-package="tp.myapp.web.mbean" scope-resolver=".....MyScopeMetadataResolver"/>`) une sous classe de **AnnotationScopeMetadataResolver** qui pourra par exemple servir à réinterpréter des annotations "non spring" comme des annotations "spring" équivalentes.

2. Spring – diverses extensions

Parties intégrées dans la partie "standard" du framework mais à considérer comme des fonctionnalités facultatives (non indispensables)

Extensions	Fonctionnalités
Spring RMI <i>et autres "spring-remote"</i>	Implémentation ou appels d'objets distants (via RMI). Quelques éléments pour invoquer des EJB ou des services soap
Spring JMS <i>(très utile)</i>	Equivalent "spring" des EJB "MDB" + gestion des "files d'attentes" de l'api JMS (queue / topic / destination) . Facile à intégrer avec ActiveMQ (ou un équivalent).
Spring Web MVC	Mini framework "MVC" (coté java/serveur) basé sur Spring . Offre des fonctionnalités assez proches de celles de "struts2" Graphiquement beaucoup moins évolué que "JSF2 +extensions "primefaces" ou "...") Spring Web MVC peut être utilisé pour fabriquer et retourner des choses très diverses (pages html , pdf , xml , json) en décomposant à souhait la logique des traitements sur un assemblage de composants reliés par de l'injection de dépendances et peut éventuellement servir à implémenter des services web de type "REST" sachant que "jersey" est plus simple/classique pour mettre en œuvre des services REST.

...	
-----	--

Extensions "spring" officielles (packagées en dehors du framework standard)

Extensions	Fonctionnalités
Spring Batch	Gestion efficace des traitements "batch" (par paquets d'enregistrements à traiter) avec supervision des éléments bien traités ou traités avec erreurs.
Spring Security (très utile)	<p>Api pour simplifier le paramétrage de la sécurité jee (au niveau des servlet/jsp par exemples)</p> <p>---> simplification de la syntaxe (en comparant au standard "security-constraint" de WEB-INF/web.xml).</p> <p>---> permet éventuellement de gérer de façon "pur spring" (indépendant du serveur d'application) des "realms" (liste d'utilisateurs avec username/password et rôles) avec une logique flexible (injection d'une implémentation xml ou jdbc ou ldap,)</p> <p>→ intégration simple de la sécurité à tous les niveaux de l'application "spring" (pages web , services métiers , services web , ...)</p>
Spring Intégration (un peu comme MuleESB)	<p>Complément pour JMS ou ... , permet d'implémenter les fonctionnalités classiques d'un mini-ESB en s'appuyant sur spring .</p> <p>(routage conditionné , médiation de services , quelques patterns EIP , transformation de formats et/ou protocoles , ...)</p>
Spring WebFlow	<p>Extension spring permettant de mieux contrôler la navigation entre les pages (jsp ou xhtml) d'une application java/web/spring .</p> <p>L'extension "SpringWebFlow" peut être (entre autres) couplée avec le framework "JSF2" .</p> <p>Cette extension peut aussi servir à simplifier certains tests de la partie web (en simulant des valeurs saisies via une logique proche des "mock-objects").</p>
...	

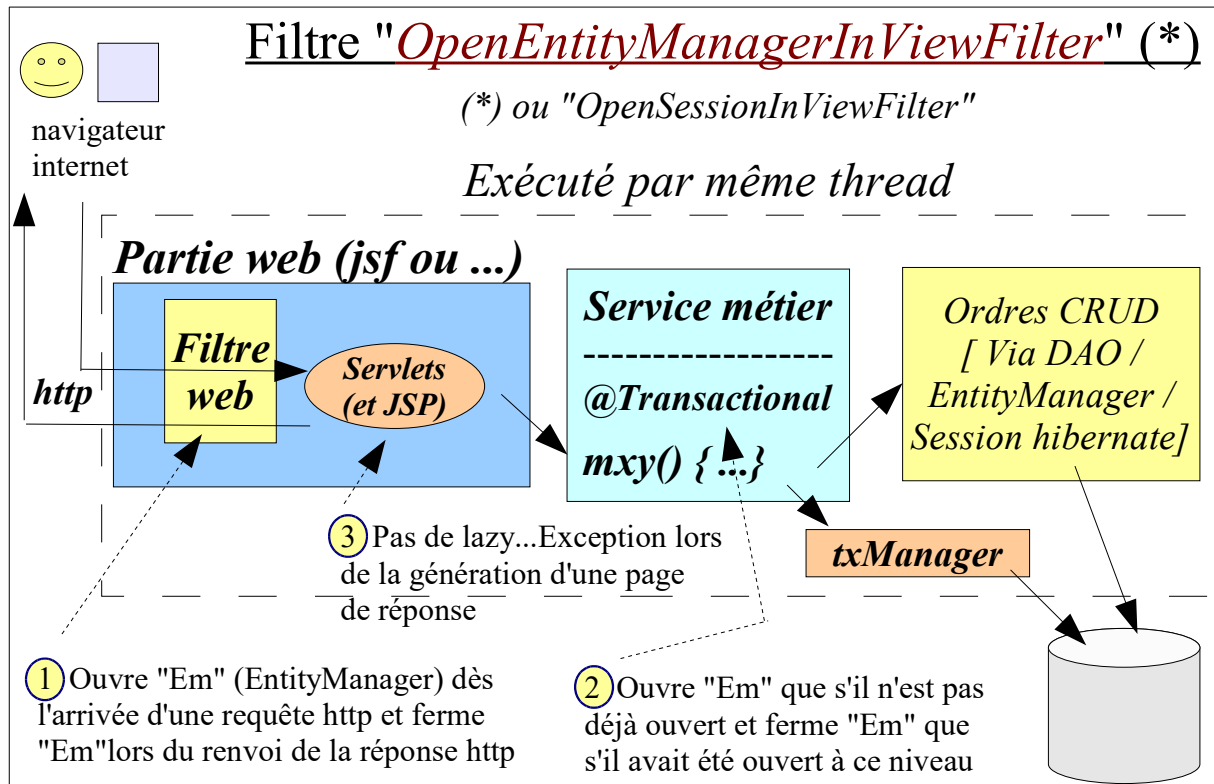
3. OpenSessionInViewFilter pour lazy="true"

De façon à ce que le *code IHM/Web d'une application JEE/Spring puisse manipuler directement les objets persistants remontés par Spring/Hibernate sans rencontrer le "Lazy ... Exception"* , on peut éventuellement mettre en place le filtre "**OpenSessionInViewFilter**" (ou bien **OpenEntityManagerInViewFilter**) au sein de WEB-INF/web.xml .

Ce cas de figure n'est utile que si les services métiers remontent directement des références sur les objets de la couche persistance sans effectuer des copies/conversions au sein d'objets de type "VO/DTO/...." . Autrement dit , ce filtre fait exploser la notion de couche logicielle stricte.

Cependant, ceci permet d'obtenir un gain substantiel sur les différents points suivants:

- performance (évite des copies)
- rapidité de développement (plus de DTO/VO à systématiquement programmé)
- traçabilité modèle UML <--> code java



3.1. Version Spring/Hibernate

Bien que le mode "flush in database" soit par défaut désactivé au moment du rendu des données (génération HTML via Struts ou JSF) , il faut néanmoins faire **attention** aux **effets de bords** liés au **contexte élargi** . ==> à tester consciencieusement .

```
<filter>
  <filter-name>hibernateFilter</filter-name>
  <filter-class>
    org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
  </filter-class>
  <init-param>
    <param-name>singleSession</param-name>
    <param-value>true</param-value>
  </init-param>
  <init-param>
    <param-name>flushMode</param-name>
    <param-value>AUTO</param-value>
  </init-param>
  <init-param>
    <param-name>sessionFactoryBeanName</param-name>
    <param-value>sessionFactory</param-value>
  </init-param>
</filter>

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

3.2. Version Spring/JPA

```
<filter>
  <filter-name>JPAFilter</filter-name>
  <filter-class>
    org.springframework.orm.jpa.support.OpenEntityManagerInViewFilter
  </filter-class>
  <init-param>
    <param-name>entityManagerFactoryBeanName</param-name>
    <param-value>myEmf</param-value>
  </init-param>
</filter>

<!-- avec impact sur servlet CXF aussi -->
<filter-mapping>
  <filter-name>JPAFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

4. Dozer (copy properties with xml mapping)

Pour recopier les propriétés d'un objet persistant vers un DTO ou vice versa , on peut éventuellement utiliser l'api open source "DOZER" .

Mieux que plein de `o2.setXxx(o1.getXxx())` ,
mieux que `BeanUtils.copyProperties(ox,oy)`,
"dozer" est capable de recopier d'un coup un objet java (ainsi que tous les sous objets référencés dans d'éventuelles sous collections) dans un autre objet java (d'une structure éventuellement différente). Un mapping Xml (facultatif) permet de paramétrer les copies (et sous copies) à effectuer.

4.1. Installation de "dozer"

Télécharger "**dozer-5.3.1.jar**" sur le site <http://dozer.sourceforge.net/>

NB: "dozer" a besoin de quelques sous dépendances:

- commons-beanutils
- commons-lang
- slf4j-api
- stax , jaxb , xmlbeans selon contexte (api pour parsing xml , peut être déjà présente si jdk 1.6)
- ...

NB: plus simplement , on peut ajouter cette dépendance dans un fichier de configuration (.pom) d'un projet basé sur "maven" :

```
...
<dependency>
  <groupId>net.sf.dozer</groupId>
  <artifactId>dozer</artifactId>
  <version>5.3.1</version>
</dependency>
```

4.2. Initialisation/utilisation java

```
import org.dozer.DozerBeanMapper;
import org.dozer.Mapper;
```

```
Mapper mapper = new DozerBeanMapper(); // avec un idéal singleton
// car initialisation potentiellement assez longue
```

et

```
DestinationObject destObject = mapper.map(sourceObject, DestinationObject.class);
```

Dans le cas d'un besoin de copies automatiques (sans différence de structure) --> Rien à paramétrer .
Par défaut , "dozer" se comporte comme `BeanUtils.copyProperties()` et il recopie d'un objet vers un autre toutes les propriétés de mêmes noms en effectuant si besoin des conversions élémentaires (int/double/... <--> String).

4.3. Copies avec paramétrage xml

dozerBeanMapping.xml (à placer dans le classpath)

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

```

<mappings xmlns="http://dozer.sourceforge.net"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://dozer.sourceforge.net
    http://dozer.sourceforge.net/schema/beanmapping.xsd">
  <mapping>
    <class-a>entity.Compte</class-a>    <class-b>dto.CompteDto</class-b>
    <field>    <a>numCpt</a>                <b>numero</b>    </field>
  </mapping>

  <mapping> <!-- with wildcard="true" by default -->
    <class-a>entity.Operation</class-a>    <class-b>dto.OperationDto</class-b>
    <field>    <a>date</a>                    <b>dateOp</b>    </field>
  </mapping>
</mappings>

```

TestApp.java

```

package tests;

import java.util.ArrayList; import java.util.List;
import org.dozer.DozerBeanMapper; import org.dozer.Mapper;
import dto.CompteDto; import dto.OperationDto;
import entity.Compte; import entity.Operation;

public class TestApp {
    private Mapper mapper = null;

    public static void main(String[] args) {
        TestApp testApp= new TestApp();
        testApp.initDozer();        testApp.test_dozer();
    }
    public void initDozer(){
        mapper = new DozerBeanMapper();
        List<String> myMappingFiles = new ArrayList<String>();
        myMappingFiles.add("dozerBeanMapping.xml");
        ((DozerBeanMapper)mapper).setMappingFiles(myMappingFiles);
    }
    public void test_dozer(){
        Compte cpt = new Compte (1,"compte 1",150.50);
        cpt.addOperation(new Operation(1,"achat 1",-45.0));
        cpt.addOperation(new Operation(2,"achat 2",-5.0));
        CompteDto cptDto = mapper.map(cpt, CompteDto.class);
        System.out.println(cptDto);
        for(OperationDto opDto : cptDto.getOperations()){
            System.out.println("\t"+opDto.toString());
        }
    }
}

```

NB:

- La basile <mapping ...> a un attribut "**wildcard**" dont la valeur par défaut est "**true**" et dans ce cas "**dozer**" copie en plus toutes les propriétés de mêmes noms.
Une valeur wildcard="false" demande à "dozer" de ne recopier que les propriétés

explicitement renseignées dans le fichier xml de mapping.

- Si l'objet à recopier comporte des sous objets ou bien des sous collections, ils/elles seront alors automatiquement recopié(e)s également (si des correspondances de types sont paramétrées sur les sous éléments).
- La documentation de référence sur "Dozer" (<http://dozer.sourceforge.net/documentation>) montre également tout un tas d'options sophistiquées (<custom-converters> , ...).

4.4. Eventuelle intégration au sein de Spring

Intégration la plus simple (suffisante dans la plupart des cas):

```
...
<!-- il faut absolument utiliser le scope="singleton" par défaut -->
<bean id="myDozerMapper" class="org.dozer.DozerBeanMapper">
  <property name="mappingFiles">
    <list>
      <value>dozer-global-configuration.xml</value>
      <value>dozer-bean-mappings.xml</value>
      <value>more-dozer-bean-mappings.xml</value>
    </list>
  </property>
</bean>
```

--> à utiliser via

```
Mapper mapper = springContext.getBean("myDozerMapper");
```

ou

```
private Mapper mapper;
```

```
@Autowired /*injection de dépendance */
public void setMapper(Mapper mapper){
    this.mapper=mapper;
}
```

et

```
DestinationObject destObject = mapper.map(sourceObject, DestinationObject.class);
```

Intégration plus sophistiquée possible

--> basée sur DozerBeanMapperFactoryBean avec propriétés facultative "customConverters" , "eventListeners" , ... [voir documentation de référence pour approfondir le sujet]

5. Conversion générique Dto/Entity via Dozer

La technologie open source "Dozer" (présentée en annexe) permet assez facilement de mettre en oeuvre un convertisseur générique (entity <--> dto) dont voici un code possible en exemple:

interface **GenericBeanConverter.java**

```
package generic.util;

import java.util.Collection;

/**
 * @author Didier Defrance
 *
 * GenericBeanConverter = interface abstraite d'un convertisseur générique de JavaBean
 * (ex: Entity_persistante <--> DTO )
 * Comportement des copies (identique à celui de BeanUtils.copyProperties()):
 * les propriétés de mêmes noms seront automatiquement recopiées d'un bean à l'autre
 * en effectuant si besoin des conversions (ex: String <--> Integer, ...)
 * NB: si les propriétés à recopier n'ont pas les mêmes noms --> config xml (dozer)
 * implémentation recommandée: MyDozerBeanConverter
 */
public interface GenericBeanConverter {

    /**
     * convert() permet de convertir d'un seul coup
     * un JavaBean ainsi que toutes ses sous parties (sous collection, ...)
     *
     * @param o = objet source à convertir
     * @param destC = type/classe destination (ex: p.XxxDto.class)
     * @return nouvel objet (de type destC) = résultat de la conversion
     */
    public abstract <T> T convert(Object o, Class<T> destC);

    /**
     * convertCollection() permet de convertir d'un seul coup une collection
     * de JavaBean de type <T1> en une autre collection de JavaBean de type
     * destC/T2 .
     *
     * @param col1 = collection source à convertir
     * @param destC = type/classe destination (ex: p.XxxDto.class)
     * des elements de la collection cible à fabriquer
     * @return nouvelle collection (d'objets de type destC) = résultat de la conversion
     */
    public abstract <T1,T2> Collection<T2>
        convertCollection(Collection<T1> col1,Class<T2> destC);
}
```

classe d'implémentation **MyDozerBeanConverter.java**

```
package generic.util;
import java.util.ArrayList;
```

```

import java.util.Collection;
import java.util.List;

import org.dozer.DozerBeanMapper;
import org.dozer.Mapper;
import org.springframework.stereotype.Component;

/* exemple de fichier src/dozerBeanMapping.xml
<?xml version="1.0" encoding="UTF-8"?>
<mappings xmlns="http://dozer.sourceforge.net" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://dozer.sourceforge.net
    http://dozer.sourceforge.net/schema/beanmapping.xsd">
  <mapping>
    <class-a>tp.entity.Compte</class-a>  <class-b>tp.dto.CompteDto</class-b>
    <field>
      <a>numCpt</a>    <b>numero</b>
    </field>
  </mapping>  <mapping>.... </mapping>
</mappings>
*/

/**
 * MyDozerBeanConverter = classe d'implementation de GenericBeanConverter
 * basée sur la technologie "Dozer" (nécessite "dozer...jar" + ...)
 * ---
 * NB: cette classe peut s'utiliser avec ou sans Spring:
 *
 * Sans Spring (enlever eventuellement l'inutile annotation @Component) et:
 *   GenericBeanConverter beanConverter = new MyDozerBeanConverter();
 *   avec "dozerBeanMapping.xml" comme nom par défaut de fichier de config "dozer"
 *   ou bien
 *   GenericBeanConverter beanConverter = new MyDozerBeanConverter({"dozer1.xml","dozer2.xml"});
 *   puis:   XxxDto xDto = beanConverter.convert(objX,XxxDto.class);
 *
 * Avec Spring 2.5 ou 3.0 (laisser @Component) et:
 *   @Autowired // ou @Inject
 *   private GenericBeanConverter beanConverter;
 *   puis directement:   XxxDto xDto = beanConverter.convert(objX,XxxDto.class);
 */

@Component
public class MyDozerBeanConverter implements GenericBeanConverter {

    private Mapper mapper = null;
    private String[] myMappingFiles = { "dozerBeanMapping.xml" } ; //par default

    /**
     * constructeur par défaut (avec "dozerBeanMapping.xml" par default)
     */
    public MyDozerBeanConverter() {

```

```

        initDozer();
    }
    /**
     * constructeur avec fichier(s) de mapping "dozer" paramétrable
     * @param myMappingFiles = tableau des noms de fichiers '.xml' pour "dozer"
     */
    public MyDozerBeanConverter(String[] myMappingFiles){
        this.myMappingFiles=myMappingFiles;
        initDozer();
    }

    /**
     * tableau des fichiers xml configurant le mapping pour dozer
     * valeur par défaut = { "dozerBeanMapping.xml" }
     * @return tableau de noms de fichiers xml
     */
    public String[] getMyMappingFiles() {
        return myMappingFiles;
    }

    /**
     * @param myMappingFiles = nouveau tableau de fichiers de config xml (dozer)
     * NB: appeler ensuite initDozer() pour (re)-initialiser Dozer.
     */
    public void setMyMappingFiles(String[] myMappingFiles) {
        this.myMappingFiles = myMappingFiles;
    }

    public void initDozer() {
        mapper = new DozerBeanMapper();
        List<String> myMappingFilesList = new ArrayList<String>();
        for(String mf: this.myMappingFiles){
            myMappingFilesList.add(mf);
        }
        ((DozerBeanMapper)mapper).setMappingFiles(myMappingFilesList);
    }

    public <T> T convert(Object o,Class<T> destC){
        return mapper.map(o, destC);
    }

    public <T1,T2> Collection<T2> convertCollection(Collection<T1> col1,Class<T2> destC){
        java.util.ArrayList<T2> col2= new java.util.ArrayList<T2>();
        for(T1 o1: col1){
            col2.add(mapper.map(o1,destC));
        }
        return col2;
    }
}

```


6. Configuration maven pour Spring 3

Cette configuration "type" est évidemment à adapter en fonction du contexte

6.1. illustration concrète des modules spring via maven

Le fichier de configuration "maven" suivant montre une collection de technologies complémentaires (Servlet/JSP, JSF/richFaces/facelets, JPA/Hibernate, CXF, ...) compatibles entre elles (au niveau des versions) et intégrées dans Spring.

Ce fichier "pom.xml" est très important car il sert à construire la liste des librairies (de WEB-INF/lib) qui seront à déployées avec le code de l'application dans tomcat6.

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>tp</groupId> <artifactId>bibliotheque</artifactId> <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version> <name>bibliotheque</name>
  <description>tp bibliotheque</description>
  <repositories>
    <!-- specific repository needed for richfaces -->
    <repository><id>jboss.org</id> <url>http://repository.jboss.org/maven2/</url> </repository>
  </repositories><properties>
    <org.springframework.version>3.0.5.RELEASE</org.springframework.version>
    <org.apache.cxf.version>2.3.0</org.apache.cxf.version>
    <org.apache.myfaces.version>2.0.3</org.apache.myfaces.version>
  </properties>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId> <artifactId>spring-core</artifactId>
      <version>${org.springframework.version}</version> <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId> <artifactId>spring-context</artifactId>
      <version>${org.springframework.version}</version> <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId> <artifactId>spring-beans</artifactId>
      <version>${org.springframework.version}</version> <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId> <artifactId>spring-aop</artifactId>
      <version>${org.springframework.version}</version> <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId> <artifactId>spring-jdbc</artifactId>
      <version>${org.springframework.version}</version> <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId> <artifactId>spring-orm</artifactId>
      <version>${org.springframework.version}</version> <scope>compile</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId> <artifactId>spring-tx</artifactId>
      <version>${org.springframework.version}</version> <scope>compile</scope>
    </dependency>
  </dependencies>
</project>
```

```

</dependency> <dependency>
    <groupId>org.springframework</groupId> <artifactId>spring-web</artifactId>
    <version>${org.springframework.version}</version> <scope>compile</scope>
</dependency> <dependency>
    <groupId>org.springframework</groupId> <artifactId>spring-test</artifactId>
    <version>${org.springframework.version}</version> <scope>compile</scope>
</dependency> <dependency>
    <groupId>junit</groupId> <artifactId>junit</artifactId>
    <version>4.8.1</version> <scope>compile</scope>
</dependency>
<dependency>
    <groupId>log4j</groupId> <artifactId>log4j</artifactId>
    <version>1.2.15</version> <scope>compile</scope> <exclusions>
    <exclusion> <groupId>com.sun.jmx</groupId> <artifactId>jmxri</artifactId> </exclusion>
    <exclusion> <groupId>javax.jms</groupId> <artifactId>jms</artifactId> </exclusion>
    <exclusion><groupId>com.sun.jdmk</groupId><artifactId>jmxtools</artifactId> </exclusion>
    </exclusions>
</dependency> <dependency>
    <groupId>org.slf4j</groupId> <artifactId>slf4j-api</artifactId>
    <version>1.5.6</version> <scope>compile</scope>
</dependency> <dependency>
    <groupId>org.slf4j</groupId> <artifactId>slf4j-log4j12</artifactId>
    <version>1.5.6</version> <scope>compile</scope>
</dependency> <dependency>
    <groupId>net.sf.dozer</groupId> <artifactId>dozer</artifactId>
    <version>5.3.1</version> <scope>runtime</scope>
</dependency><dependency>
    <groupId>mysql</groupId><artifactId>mysql-connector-java</artifactId>
    <version>5.1.6</version>
</dependency> <dependency>
    <groupId>org.hibernate</groupId> <artifactId>hibernate-core</artifactId>
    <version>3.5.1-Final</version> <scope>compile</scope> <exclusions>
    <exclusion> <groupId>javax.transaction</groupId><artifactId>jta</artifactId></exclusion>
    <exclusion><groupId>asm</groupId><artifactId>asm</artifactId></exclusion>
    <exclusion><groupId>asm</groupId><artifactId>asm-attrs</artifactId></exclusion>
    <exclusion> <groupId>cglib</groupId><artifactId>cglib</artifactId></exclusion> </exclusions>
</dependency> <dependency>
    <groupId>javax.inject</groupId> <artifactId>javax.inject</artifactId> <version>1</version>
</dependency> <dependency>
    <groupId>javax.transaction</groupId><artifactId>jta</artifactId><version>1.1</version>
</dependency> <dependency>
    <groupId>org.hibernate</groupId><artifactId>hibernate-commons-annotations</artifactId>
    <version>3.2.0.Final</version>
    <exclusions>
    <exclusion><groupId>org.hibernate</groupId> <artifactId>hibernate-core</artifactId></exclusion>
    <exclusion> <groupId>org.hibernate</groupId><artifactId>hibernate</artifactId></exclusion>
    </exclusions>
</dependency> <dependency>
    <groupId>org.hibernate</groupId><artifactId>hibernate-entitymanager</artifactId>
    <version>3.5.1-Final</version>
</dependency> <dependency>

```

```

        <groupId>org.hibernate</groupId> <artifactId>hibernate-validator</artifactId>
        <version>4.0.2.GA</version>
    </dependency><dependency>
        <groupId>javax.servlet</groupId> <artifactId>servlet-api</artifactId>
        <version>2.5</version> <scope>provided</scope>
    </dependency> <dependency>
        <groupId>javax.servlet.jsp</groupId> <artifactId>jsp-api</artifactId>
        <version>2.1</version> <scope>provided</scope>
    </dependency> <dependency>
        <groupId>javax.servlet</groupId> <artifactId>jstl</artifactId>
        <version>1.2</version> <scope>compile</scope>
    </dependency> <dependency>
        <groupId>org.apache.myfaces.core</groupId> <artifactId>myfaces-api</artifactId>
        <version>${org.apache.myfaces.version}</version><scope>compile</scope>
    </dependency> <dependency>
        <groupId>org.apache.myfaces.core</groupId> <artifactId>myfaces-impl</artifactId>
        <version>${org.apache.myfaces.version}</version> <scope>compile</scope>
    </dependency> <dependency>
        <groupId>com.sun.facelets</groupId><artifactId>jsf-facelets</artifactId>
        <version>1.1.15</version>
    </dependency> <dependency>
        <groupId>org.richfaces.ui</groupId> <artifactId>richfaces-ui</artifactId>
        <version>3.3.3.Final</version>
    </dependency> <dependency>
        <groupId>org.richfaces.framework</groupId> <artifactId>richfaces-impl-jsf2</artifactId>
        <version>3.3.3.Final</version>
    </dependency><dependency>
        <groupId>org.apache.cxf</groupId><artifactId>cxf-api</artifactId>
        <version>${org.apache.cxf.version}</version><scope>compile</scope>
    </dependency><dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-frontend-jaxws</artifactId><version>${org.apache.cxf.version}</version>
        <exclusions><exclusion> <groupId>asm</groupId> <artifactId>asm</artifactId>
            </exclusion><exclusion> <groupId>org.apache.geronimo.specs</groupId>
                <artifactId>geronimo-javamail_1.4_spec</artifactId>
            </exclusion></exclusions>
    </dependency><dependency>
        <groupId>org.apache.cxf</groupId>
        <artifactId>cxf-rt-transport-http</artifactId><version>${org.apache.cxf.version}</version>
    </dependency>
</dependencies>
<build>
    <plugins> <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId> <version>2.0.2</version>
        <configuration>
            <source>1.6</source> <!-- jdk -->
            <target>1.6</target>
        </configuration>
    </plugin><plugin>
        <groupId>org.apache.maven.plugins</groupId>

```

```
<artifactId>maven-war-plugin</artifactId> <version>2.1-alpha-1</version>
</plugin></plugins>
<finalName>bibliotheque</finalName>
</build>
</project>
```

7. Les concepts de AOP (vocabulaire)

Aspect:

Un **aspect** correspond à une **fonctionnalité transverse** qui **concerne une multitude d'objets** (ex: gestion des logs , gestion des transactions , ...).

JoinPoint:

Un **point de jonction** (*JoinPoint*) correspond à un **point précis de l'exécution d'un programme** (ex: *appel d'une méthode , remontée d'une exception*) . C'est à ce genre d'endroit que pourront être insérés de nouvelles fonctionnalités (traitements transverses/génériques).

Advice:

Un "advice" correspond à une **action (ajout de traitements supplémentaires)** qui sera déclenchée au niveau de certain(s) point(s) de jonction.

Il en existe plusieurs catégories : "around" , "before" , "throws" ,

PointCut:

Un "*PointCut*" correspond à un ensemble de points de jonction qui doivent être associés à un certain "advice" (ajout de traitements supplémentaires) . Autrement dit un "*pointcut*" correspond au paramétrage global du tissage des aspects: [*PointCut* = liste des points d'intersection entre le code principal et un certain aspect]

Target (advised) object:

Objet contenant un (ou plusieurs) point(s) de jonction.

Introduction:

Action particulière consistant à ajouter un nouveau membre (méthode ou champ) à une classe d'objet cible (advised object).

AOP Proxy:

Nouvel objet créé par les mécanismes d'AOP et comportant des ajouts ("advice").

Weaving:

Assembler (tisser les aspects) pour former de nouveaux objets plus complets.

Ceci peut être effectué lors de la compilation ou bien à l'exécution du programme (selon la

technologie employée).

8. Les grands axes de la mise en oeuvre d' A.O.P.

Il existe actuellement tout un tas d'implémentation de AOP qui diffèrent essentiellement selon les grands axes suivants:

- le **langage de programmation** utilisé (Java , C++ , C# , ...).
- l'**encodage des points de jonctions** (fichiers xml , annotations , composants spécifiques , ...)
- la **manière utilisée et le moment pour tisser les aspects** (pré-compilation , dynamiquement à l'exécution ,)

Chaque solution a ses avantages et ses inconvénients qui sont à évaluer au cas par cas (en fonction du contexte) avant de choisir une technologie précise.

8.1. avantages et inconvénients selon les choix technologiques

Technologies utilisées	avantages (points forts)	inconvénients (points faibles)
pré-compilation (pré-processeur)	solution efficace et robuste (bonnes performances, ...) . beaucoup de possibilités (ajout d'attributs , ...). méthode assez statique (vis à vis du code compilé)	nécessite un environnement de développement particulier (IDE avec pré-processeur) ==> petite dépendance vis à vis de la plateforme de développement (avant qu'apparaisse UN standard).
dynamique – à l'exécution	solution plus simple et plus souple (éventuellement re-paramétrable sans recompilation). pas de dépendances vis à vis du compilateur (pas de pré-processeur)	performances et possibilités moins évoluées.
mécanisme double (pré-compilation + complément dynamique)	compromis intéressant	complexité. ensemble peu homogène (si détails non masqués)
encodage xml des "pointCut"	rien à ajouter dans le code de base. possibilités évoluées (ex: dans toutes les classes du package , ...).	fichier de paramétrage (.xml) basé sur une syntaxe spécifique à la technologie employée (en attendant une éventuelle standardisation).
"pointCut" sous forme d'annotations (java 5,)	très pratique pour fournir des paramétrages fins . les informations sont très proches du code concerné.	Si les annotations utilisées ne sont pas liées à un standard , on introduit alors une dépendance assez forte vis à vis de la technologie employée.

Technologies utilisées	avantages (points forts)	inconvénients (points faibles)
"pointCut" sous forme de composant (bean + ioc , ...)	très pratique pour injecter de façon modulaire des mécanismes (AOP) au sein d'un framework IOC (conteneur léger de type HiveMind / Spring)	Le paramétrage de cette variante utilise généralement des fichiers xml ou des annotations ou un mixte des deux.
encodage mixte (xml + annotations) des "pointCut"	souple – compromis. Elements génériques ==> xml Paramétrages fins ==> annotations	double dépendance (syntaxe des annotations + syntaxe du fichier xml)

==> Après une phase de recherche en grande partie déjà effectuée, A.O.P. est aujourd'hui entré dans une phase d'ingénierie (beaucoup de projets concurrents).

==> Une future phase de standardisation est très attendue .

==> Les valeurs ajoutées de AOP sont suffisamment importantes pour se lancer dès aujourd'hui dans une technologie pilote (quitte à restructurer le code lorsqu' apparaîtra une standardisation).

8.2. solutions basées sur des pré-compilations (pré-processeur)

AspectJ

==> Cette technologie "AOP" dédiée au langage Java est assez avancée et est adoptée par une grande communauté de développeur.

==> AspectJ est une technologie basée sur une pré-compilation .

==> Le projet "Eclipse-AspectJ" permet d'intégrer la technologie "AspectJ" dans l'I.D.E. "Eclipse".

8.3. solutions basées sur des mécanismes dynamiques lors de l'exécution

Spring AOP v1.2.x

Les mécanismes de Spring AOP (en version 1.2.x) sont entièrement dynamiques. ils sont déclenchés lors de l'exécution du programme (et n'influent en rien la compilation).

Spring AOP v2.x

Les mécanismes de Spring AOP (en version 2.x) sont toujours dynamiques (déclenchés lors de l'exécution du programme) . Spring AOP 2 utilise néanmoins des syntaxes de paramétrage (annotations) volontairement proches de AspectJ-weaver .

9. L'essentiel de Spring AOP

Spring AOP utilise le terme "**Advisor**" pour désigner une classe dont le contenu correspond à l'ensemble d'un aspect, c'est à dire (à la fois) à :

- un advice (nouvelle action / traitement supplémentaire)
- et au paramétrage d'un "PointCut" (ensemble de points de jonction)

Spring AOP est basé sur le projet open source "AOP Alliance" (org.aopalliance...)

Les mécanismes internes de Spring AOP reposent essentiellement sur des proxys dynamiques de Java/J2SE et utilise quelquefois des proxys de types "CGLIB" .

9.1. PointCut de "Spring AOP":

Les "PointCut" de Spring AOP sont des implémentations de l'interface suivante (org.springframework.aop.Pointcut) :

```
public interface Pointcut {
    ClassFilter getClassFilter();
    MethodMatcher getMethodMatcher();
}
```

```
public interface ClassFilter {
    boolean matches(Class clazz);
}
```

```
public interface MethodMatcher {
    boolean matches(Method m, Class targetClass);
    boolean isRuntime();
    boolean matches(Method m, Class targetClass, Object[] args);
}
```

En pratique, on utilise essentiellement des implémentations basées sur des expressions régulières (telles que `RegexpMethodPointcutAdvisor`) pour filtrer les méthodes devant faire l'objet d'un "advice" (action/ajout).

9.2. Advices / intercepteurs de "Spring AOP":

Around advice / method interceptor (cas le plus fréquent):

```
public interface MethodInterceptor extends Interceptor {
    Object invoke(MethodInvocation invocation) throws Throwable;
}
```

exemple:

```
public class DebugInterceptor implements MethodInterceptor
{
    public Object invoke(MethodInvocation invocation) throws Throwable
    {
        System.out.println("Before: invocation=[" + invocation + "]");
        Object rval = invocation.proceed();
        System.out.println("Invocation returned");
        return rval;
    }
}
```

Before advice :

```
public interface MethodBeforeAdvice extends BeforeAdvice {
    void before(Method m, Object[] args, Object target) throws Throwable;
}
```

exemple:

```
public class CountingBeforeAdvice implements MethodBeforeAdvice
{
    private int count;
    public void before(Method m, Object[] args, Object target) throws Throwable
    {
        ++count;
    }
    public int getCount() { return count; }
}
```

Throws advices:

L'interface `org.springframework.aop.ThrowsAdvice` ne comporte aucune méthode (il ne s'agit que d'une interface de marquage). Néanmoins, les implémentations doivent être basées sur une (ou plusieurs) méthode(s) ayant un prototype compatible avec le modèle suivant:

afterThrowing([Method], [args], [target], subclassOfThrowable)

exemple:

```
public class ExempleThrowsAdvice implements ThrowsAdvice
{
    public void afterThrowing(RemoteException ex) throws Throwable {
        // Do something with remote exception
    }
    public void afterThrowing(Method m, Object[] args, Object target, ServletException ex) {
        // Do something with all arguments
    }
}
```

After returning advices:

```
public interface AfterReturningAdvice extends Advice {
    void afterReturning(Object returnValue, Method m, Object[] args, Object target)
        throws Throwable;
}
```

exemple:

```
public class CountingAfterReturningAdvice implements AfterReturningAdvice {
    private int count;
    public void afterReturning(Object returnValue, Method m, Object[] args, Object target)
        throws Throwable { ++count; }
    public int getCount() { return count; }
}
```

Introduction advice :

==> cas très particulier ==> consulter la documentation de référence

9.3. ProxyFactoryBean de Spring AOP :

exemple:

```
<bean id="personTarget" class="com.mycompany.PersonImpl">
    <property name="name"><value>Tony</value></property>
    <property name="age"><value>51</value></property>
</bean>

<bean id="myAdvisor" class="com.mycompany.MyAdvisor">
    <property name="someProperty"><value>Custom string property
value</value></property>
</bean>

<bean id="debugInterceptor" class="org.springframework.aop.interceptor.DebugInterceptor">
</bean>

<bean id="person"
class="org.springframework.aop.framework.ProxyFactoryBeanmyAdvisor</value>
            <value>debugInterceptor</value>
        </list>
    </property>
</bean>
```

==> Effets: le composant d'id "person" sera construit par Spring AOP en:

- partant du composant de base "personTarget"
- ajoutant les fonctionnalités des advices "myAdvisor" et "debugInterceptor"

--> consulter la documentation de référence pour approfondir "ProxyFactoryBean".

9.4. "Auto proxy" de Spring AOP :

Application automatique d'une interposition de "proxy AOP"

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
<property name="beanNames"><value>jdk*,onlyJdk</value></property>
<property name="interceptorNames">
  <list>
    <value>myInterceptor</value>
  </list>
</property>
</bean>
```

DefaultAdvisorAutoProxyCreator

```
<bean class=
"org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator">

<bean class="org.springframework.transaction.interceptor.TransactionAttributeSourceAdvisor">
<property name="transactionInterceptor" ref="transactionInterceptor"/>
</bean>

<bean id="customAdvisor" class="com.mycompany.MyAdvisor"/>

<bean id="businessObject1" class="com.mycompany.BusinessObject1">
<!-- Properties omitted -->
</bean>
<bean id="businessObject2" class="com.mycompany.BusinessObject2"/>
```

==> Application automatique de tous les "advisors" éligibles du contexte sur tout les "beans" du contexte.

NB: Seront appliqués des "advisors" (avec pointcut) et non pas des seuls "interceptors" / "advices"

Un advisor personnalisé peut facilement se configurer de la façon suivante:

```
<bean id="myAdvisor"
class="org.springframework.aop.support.RegexMethodPointcutAdvisor">
<property name="advice">
  <ref local="beanNameOfAopAllianceInterceptor"/>
</property>
<property name="patterns">
  <list>
    <value>.*set.*</value>
  </list>
</property>
</bean>
```

10. Présentation du framework "Spring Web Flow"

Le framework Spring propose un petit framework Web/MVC appelée "*Spring Web MVC*" ainsi d'un framework de plus haut niveau "*Spring Web Flow*".

Spring MVC peut être utilisé pour des I.H.M. Web simples (sans composants graphiques évolués) ou bien en complément des frameworks web habituels "Struts ou JSF".

Spring Web Flow est plutôt à considérer comme un framework complémentaire qui prend rigoureusement en charge les navigations entre les vues et les actions.

Spring Web Flow peut très bien être intégré avec JSF/JSF2 ou d'autres frameworks web.

Spring Web Flow et JSF se complètent assez bien dans le sens où :

- JSF2 (et ses extensions "richfaces", "primefaces") gère très bien l'affichage mais gère les navigations que de manière simpliste.
- Spring Web Flow gère très bien les navigations (c'est son rôle).

NB : Lorsque JSF est utilisé avec "Spring Web Flow", les pages "jsp/xhtml" gardent à peu près la même structure mais les managed bean en arrière plan sont pas mal chamboulés : ils deviennent des "flowBean" presque toujours sans action car les actions sont reportées dans les définitions ".xml" des flots d'exécutions.

Principales Abstractions et Fonctionnalités :

Spring Web Flow utilise un certain nombre d'abstractions (Flow, SubFlow, State, ViewState, ActionState, Event, ...) qui se veulent "utilisables avec différents frameworks web (JSF, Spring MVC,)" et ceci permet même d'effectuer des tests de logique navigationnelle d'IHM sans serveur avec JUnit (via des "saisies simulées").

Les éléments de **Spring Web Flow** (FlowBean, Flow, ...) s'interposent entre les pages (jsp/xhtml) et les services métiers pour **prendre en charge la logique applicative (successions contrôlées d'interactions utilisateurs et d'actions déléguées au back office)**.

La définition d'un "Spring Web Flow" découle très naturellement d'un diagramme d'état UML comportant des états de type "vue" et "action" et des transitions liés à des événements. Les éléments avancés des diagrammes d'état UML (Entry, Exit, condition/gardien) se retrouvent aussi dans Spring Web Flow (*onRender*, *onEntry*, *onExit*,).

Spring Web Flow gère des contextes d'exécutions pour chaque utilisateur de l'application et ceci lui permet de prendre en charge de nouveaux scopes "**flowScope**" et "**viewScope**" (en plus des habituels et standards "requestScope", "sessionScope", "...").

Attention : il faut veiller à utiliser une version de Spring web flow qui soit bien en adéquation avec la version de "spring framework (core)" et la version de "JSF". Par exemple, la version 2.3.1 de "Spring web Flow" nécessite absolument une version ≥ 3.1 de Spring Framework.

11. Définition d'un Spring Web Flow

11.1. Éléments fondamentaux (Flow, viewState, transitions,)

Emplacement possible : `src\main\webapp\WEB-INF\flows\main\main-flow.xml`

main-flow.xml (exemple)

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <var name="identification" class="tp.myapp.web.wfbean.Identification" />
    <!-- in flowScope -->
    <var name="clientComptes" class="tp.myapp.web.wfbean.ClientComptes" />
    <!-- in flowScope -->

    <!-- welcome.xhtml (of this main webflow) -->
    <view-state id="welcome">
        <transition on="identification-client" to="identificationClient"> </transition>
        <transition on="quitter" to="finish"> </transition>
    </view-state>

    <view-state id="identificationClient"> <!-- identificationClient.xhtml ou .... -->
        <transition on="identifierClient" to="verifIdentification"> </transition>
        <transition on="annuler" to="welcome"> </transition>
    </view-state>

    ...

    <end-state id="finish" />

</flow>
```

11.2. Liste des portées (scope)

Scope	Délimitation de la portée	Usage
<i>request, session, application</i>	<i>standard habituel</i>	<i>standard habituel</i>
flowScope	Utilisable du début à la fin d'un flot (ou d'un sous flot).	Très utile (très bon intermédiaire entre "request" et "session")
viewScope	Utilisable qu'au sein du "viewState" courant	Un peu plus long que "request" (état conservé si "refresh")
conversationScope	Durée de vie = "top-level flow"	très proche de "session" mais avec "finish" automatisé
flashScope	Très éphémère, contenu réinitialisé lors de chaque "onRender"	pour message à n'afficher qu'une seule fois

11.3. Actions (evaluate , actionState , ...)

main-flow.xml (exemple)

```
<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <var name="identification" class="tp.myapp.web.wfbean.Identification" />
    <!-- in flowScope -->

    ...

    <view-state id="identificationClient"> <!-- identificationClient.xhtml -->
        <transition on="identifierClient" to="verifIdentification"> </transition>
        <transition on="annuler" to="welcome"> </transition>
    </view-state>

    <action-state id="verifIdentification">
        <evaluate expression="gestionClientsImpl.isGoodPasswordOfClient(
            identification.numClient,identification.password)"/>
        <transition on="yes" to="sessionClientIdentifie" /> <!-- yes for true -->
        <transition on="no" to="identificationClient" /> <!-- no for false -->
    </action-state>

    <subflow-state id="sessionClientIdentifie" subflow="clientIdentifie">
        ....
    </subflow-state>

    <end-state id="finish" />

</flow>
```

11.4. SubFlow (avec input/output) et "onRender"

Référencement/Appel au sein de main-flow.xml :

```
....
<subflow-state id="sessionClientIdentifie" subflow="clientIdentifie">
    <input name="numClient" value="identification.numClient" />
```

```

        <transition on="quit" to="welcome" />
        <!-- <transition on="other_return_value_of_this_sub_flow"
              to="other_state_id" /> -->
    </subflow-state>
    ....

```

Définition du "subflow" :

clientIdentifie-flow.xml (exemple dans src/main/webapp/WEB-INF/flows/clientIdentifie)

```

<?xml version="1.0" encoding="UTF-8"?>
<flow xmlns="http://www.springframework.org/schema/webflow"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.springframework.org/schema/webflow
      http://www.springframework.org/schema/webflow/spring-webflow-2.0.xsd">

    <var name="clientComptes" class="tp.myapp.web.wfbean.ClientComptes" />
    <!-- in flowScope -->
    <var name="virement" class="tp.myapp.web.wfbean.Virement" />
    <!-- in flowScope -->

    <input name="numClient" required="true" />

    <on-start>
        <evaluate expression="gestionClientsImpl.getClientByNum(numClient)"
              result="flowScope.clientComptes.client" />
    </on-start>

    <view-state id="listeComptes"><!-- listeComptes.xhtml -->
        <on-render>
            <evaluate expression="gestionComptesImpl.getComptesOfClient(numClient)"
                  result="flowScope.clientComptes.listeComptes" />
            <evaluate expression="flowScope.clientComptes.listeComptes"
                  result="viewScope.listeComptesDataModel"
                  result-type="dataModel" />
        </on-render>
        <transition on="quitter" to="quit"> </transition>
        <transition on="dernieresOperations" to="operations">
            <evaluate expression="listeComptesDataModel.selectedRow.numero"
                  result="flowScope.clientComptes.selectedCptNum"/>
        </transition>
        <transition on="paramVirement" to="paramVirement"> </transition>
    </view-state>

    <view-state id="paramVirement">
        <!-- paramVirement.xhtml (numCptDeb, numCptCred, montant)-->
        <transition on="effectuerVirement" to="transferer">
            </transition>
    </view-state>

    <action-state id="transferer">

```

```

    <evaluate expression="gestionComptesImpl.transférer(
        virement.montant,virement.numCptDeb,virement.numCptCred)"/>
    <transition to="listeComptes" />
</action-state>

<view-state id="operations"> <!-- operations.xhtml -->
    <on-render>
        <evaluate expression="gestionComptesImpl.getOperationsOfCompte(
            clientComptes.selectedCptNum)"
            result="viewScope.listeOperations" />
        <!-- result-type="dataModel" uniquement nécessaire
            si besoin de récupérer .selectedRow ultérieurement -->
    </on-render>
    <transition on="retour" to="listeComptes"> </transition>
    <transition on="quitter" to="quit"> </transition>
</view-state>

<end-state id="quit" /> <!-- quitter la session "clientIdentifie" / deconnexion -->
<!-- <end-state id="other_return_value_of_this_sub_flow" /> -->

</flow>

```

11.5. Validations et messages

Voir documentation de référence.

12. Intégration JSF dans Spring Web Flow

12.1. Configuration & arborescence nécessaire

WEB-INF/web.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<web-app .... >
    <display-name>minibank-sp-web-flow</display-name>
    <welcome-file-list>
        <welcome-file>index.html</welcome-file>
        <welcome-file>index.jsp</welcome-file>
    </welcome-file-list>

    <servlet>
        <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
        <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
        <init-param>
            <param-name>contextConfigLocation</param-name>
            <!-- <param-value>/WEB-INF/classes/web-application-config.xml
                </param-value> --> <param-value></param-value>
            <!-- utiliser plutot la config globale en context-param / listener -->
        </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

```



```

</servlet>
<servlet-mapping>
    <servlet-name>Spring MVC Dispatcher Servlet</servlet-name>
    <url-pattern>/spring/*</url-pattern>
</servlet-mapping>

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/classes/web-application-config.xml</param-value>
</context-param>

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

<servlet>
    <servlet-name>Faces Servlet</servlet-name>
    <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>Faces Servlet</servlet-name>
    <url-pattern>*.jsf</url-pattern>
</servlet-mapping>

<!-- Use JSF view templates saved as *.xhtml, for use with Facelets -->
<context-param>
    <param-name>javax.faces.DEFAULT_SUFFIX</param-name>
    <param-value>.xhtml</param-value>
</context-param>

<!-- PARTIAL_STATE_SAVING not yet enabled with MyFaces2 ,
doit absolument être à false avec spring webflow-->
<context-param>
    <param-name>javax.faces.PARTIAL_STATE_SAVING</param-name>
    <param-value>false</param-value>
</context-param>

<!-- Enables special Facelets debug output during development -->
<context-param>
<param-name>facelets.DEVELOPMENT</param-name>
    <param-value>true</param-value>
</context-param>

<!-- Causes Facelets to refresh templates during development -->
<context-param>
    <param-name>facelets.REFRESH_PERIOD</param-name>
    <param-value>1</param-value>
</context-param>

<!-- Serves static resource content from .jar files such as spring-faces.jar -->
<servlet>

```

```

        <servlet-name>Resources Servlet</servlet-name>
        <servlet-class>org.springframework.js.resource.ResourceServlet</servlet-class>
        <load-on-startup>0</load-on-startup>
    </servlet>

    <!-- Map all /resources requests to the Resource Servlet for handling -->
    <servlet-mapping>
        <servlet-name>Resources Servlet</servlet-name>
        <url-pattern>/resources/*</url-pattern>
    </servlet-mapping>
</web-app>

```

Navigation possible :

dans index.html (de src/main/webapp) :

```

...
<a href="spring/static-welcome"/> welcome (minibank) / spring web flow</a> <br/>
....

```

dans WEB-INF/static-welcome.xhtml (de src/main/webapp) :

```

...
<a href="main"> main flow</a> <br/>
....

```

Rien de spécial dans **WEB-INF/faces-config.xml** .

Les fichiers de configurations Spring peuvent être placés dans *src/main/resources* de maven.

Configuration Spring ensembliste "**web-application-config.xml**" :

```

<?xml version="1.0" encoding="UTF-8"?>
<beans .... >
    <import resource="webmvc-config.xml" />
    <import resource="webflow-config.xml" />

    <import resource="dataSourceSpringConf.xml" />
    <import resource="serviceSpringConf.xml" />

    <!-- <import resource="security-config.xml" /> -->
</beans>

```

webmvc-config.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:faces="http://www.springframework.org/schema/faces"
    xsi:schemaLocation="
        http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
        http://www.springframework.org/schema/faces
http://www.springframework.org/schema/faces/spring-faces-2.2.xsd">

    <!-- <faces:resources /> -->

```

```

<!-- Maps request paths to flows in the flowRegistry; e.g. a path of /hotels/booking looks
for a flow with id "hotels/booking" -->
<bean class="org.springframework.webflow.mvc.servlet.FlowHandlerMapping">
  <property name="order" value="1"/>
  <property name="flowRegistry" ref="flowRegistry" />
  <property name="defaultHandler">
    <!-- If no flow match, map path to a view to render; e.g. the "/intro" path
    would map to the view named "intro" -->
    <bean class="org.springframework.web.servlet.mvc.UrlFilenameViewController" />
  </property>
</bean>

<!-- Maps logical view names to Facelet templates in /WEB-INF
(e.g. 'search' to '/WEB-INF/search.xhtml' -->
<bean id="faceletsViewResolver"
  class="org.springframework.web.servlet.view.UrlBasedViewResolver">
  <property name="viewClass" value="org.springframework.faces.mvc.JsfView"/>
  <property name="prefix" value="/WEB-INF/" />
  <property name="suffix" value=".xhtml" />
</bean>

<!-- Dispatches requests mapped to
org.springframework.web.servlet.mvc.Controller implementations -->
<bean class="org.springframework.web.servlet.mvc.SimpleControllerHandlerAdapter" />

<!-- Dispatches requests mapped to flows to FlowHandler implementations -->
<bean class="org.springframework.faces.webflow.JsfFlowHandlerAdapter">
  <property name="flowExecutor" ref="flowExecutor" />
</bean>

</beans>

```

webflow-config.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:webflow="http://www.springframework.org/schema/webflow-config"
  xmlns:faces="http://www.springframework.org/schema/faces"
  xsi:schemaLocation="
http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/webflow-config
http://www.springframework.org/schema/webflow-config/spring-webflow-config-2.3.xsd
http://www.springframework.org/schema/faces
http://www.springframework.org/schema/faces/spring-faces-2.2.xsd">

  <bean class="org.springframework.faces.webflow.JsfFlowHandlerAdapter">
    <property name="flowExecutor" ref="flowExecutor" />
  </bean>

  <!-- Executes flows: the central entry point into the Spring Web Flow system -->
  <webflow:flow-executor id="flowExecutor">
    <webflow:flow-execution-listeners>

```

```

        <webflow:listener ref="facesContextListener" />
    </webflow:flow-execution-listeners>
</webflow:flow-executor>

<!-- The registry of executable flow definitions -->
<webflow:flow-registry id="flowRegistry" base-path="/WEB-INF/flows"
    flow-builder-services="facesflowBuilderServices" >
    <webflow:flow-location-pattern value="/**/*-flow.xml" />
</webflow:flow-registry>

<!-- Configures the Spring Web Flow JSF integration -->
<faces:flow-builder-services id="facesflowBuilderServices" development="true" />

<!-- A listener maintain one FacesContext instance per Web Flow request. -->
<bean id="facesContextListener"
    class="org.springframework.faces.webflow.FlowFacesContextLifecycleListener"

</beans>

```

Configuration **maven** (pour Spring Web Flow 2.3.1 intégré dans Spring 3.1) :

pom.xml

```

....
    <!-- si pas avec JSF
    <dependency>
        <groupId>org.springframework.webflow</groupId>
        <artifactId>spring-webflow</artifactId>
        <version>2.3.1.RELEASE</version>
    </dependency>
    -->

    <!-- si avec JSF -->
    <dependency>
        <groupId>org.springframework.webflow</groupId>
        <artifactId>spring-faces</artifactId>
        <version>2.3.1.RELEASE</version>
    </dependency>

....

```

12.2. Liens entre pages ".xhtml" et "flow + flowBean"

Lorsque Spring-web-flow est utilisé avec JSF, les id des **"view-state"** coïncident avec les noms des fichiers *pageXy.xhtml* de JSF (ex : `<view-state id="pageXy" >`)

Les noms des actions de JSF (ex : `<h:commandButton value="identification" action="identifierClient" />`) correspondent aux noms des événements des transitions (ex : `<transition on="identifierClient" to="verifIdentification"> </transition>`).

Les variables objets des "flow" (souvent en `flowScope`) sont accessibles au sein des fichiers *pageXy.xhtml* de JSF via la syntaxe habituelle `{objectName.propertyName}` .

12.3. Utilisation du DataModel JSF au sein de Spring WebFlow

```
<!-- result-type="dataModel" uniquement nécessaire
    si besoin de récupérer .selectedRow ultérieurement
NB : le "DataModel" de JSF est utilisé (de façon cachée par JSF) au sein des "h:dataTable" et
..... -->
```

```
<view-state id="listeComptes"><!-- listeComptes.xhtml -->
    <on-render>
        <evaluate expression="gestionComptesImpl.getComptesOfClient(numClient)"
            result="flowScope.clientComptes.listeComptes" />
        <evaluate expression="flowScope.clientComptes.listeComptes"
            result="viewScope.listeComptesDataModel"
            result-type="dataModel" />
    </on-render>
    <transition on="quitter" to="quit"> </transition>
    <transition on="dernieresOperations" to="operations">
        <evaluate expression="listeComptesDataModel.selectedRow.numero"
            result="flowScope.clientComptes.selectedCptNum"/>
    </transition>
    <transition on="paramVirement" to="paramVirement"> </transition>
</view-state>
```

.../...

listeComptes.xhtml

```

...
<h:form id="comptes">
    <h:dataTable var="cpt" border="2" value="#{listeComptesDataModel}" >
        <!-- anciennement value=...clientComptes.listeComptes... -->

        <h:column>
            <f:facet name="header"><f:verbatim>numero</f:verbatim></f:facet>
            <h:outputText value="#{cpt.numero}"/>
        </h:column>

        <h:column>
            <f:facet name="header"><f:verbatim>label</f:verbatim></f:facet>
            <h:outputText value="#{cpt.label}"/>
        </h:column>

        <h:column>
            <f:facet name="header"><f:verbatim>solde</f:verbatim></f:facet>
            <h:outputText value="#{cpt.solde}"/>
        </h:column>

        <h:column>
            <f:facet name="header"><f:verbatim>détails</f:verbatim></f:facet>
            <h:commandButton action="dernieresOperations" value="operations" />
            <!-- pas besoin passer de paramètre (.selectedRow automatiquement pris
                en charge -->
        </h:column>

    </h:dataTable>

    <h:commandButton action="quitter" value="se déconnecter" />
    <hr/>
    <h:commandButton action="paramVirement" value="effectuer un virement interne" />
</h:form>
...

```

12.4. Autres spécificités "JSF + Spring Web Flow"

Voir la documentation de référence.

13. Spring-Integration (présentation)

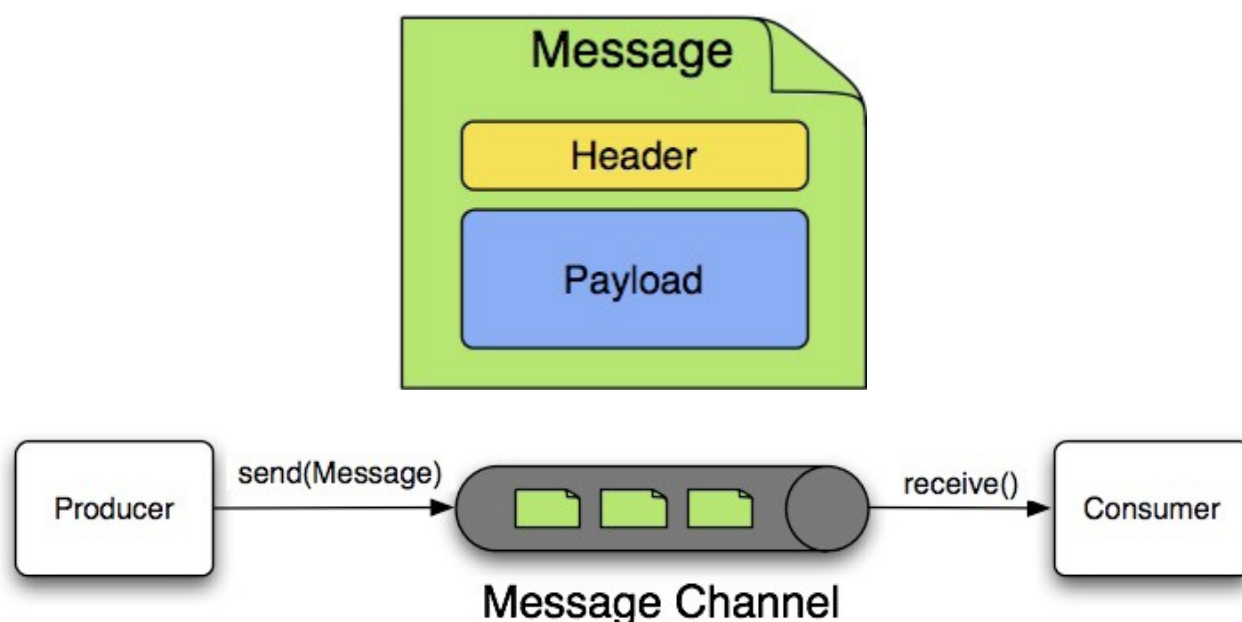
L'extension "**Spring-integration**" permet de programmer en **java/spring** des fonctionnalités de type "**MOM**" (*MiddleWare Orienté Message*) et/ou *mini "ESB"* avec:

- des communications extérieures selon divers API / Protocoles (HTTP , RMI , JMS, ...)
- des routages de messages
- des transformations de messages , déclencher des traitements ad-hoc fonctionnels.
- tout un tas d'options asynchrones (décomposition / recomposition , ...).

"**Spring-integration**" est basé sur le *paradigme "pipe and filtering"* c'est à dire "acheminer des messages dans des files associées à certaines destinations". Chaque destination (endpoint) correspond à un point de traitement fonctionnel ou à un point de communication .

Au sein du framework java "Spring Integration" :

- les **messages** ("**java**") comportent une entête imposée et un contenu libre
- les messages sont véhiculés via des pipes appelés "**Channel**" [*ceux-ci existent en version "Point-to-Point" ou en version "Publication/Souscription"*].
- Au niveau des routages internes gérés automatiquement par le framework "Spring integration", chaque destination ou départ est associé à un "**message endpoint**" pouvant prendre l'une des formes suivantes:
 - **Transformateur** (ex : "Xml" to java String)
 - **Filtre** (laissant passer que certains messages)
 - **Routeur** (vers tel ou tel "channel" selon contenu ou entête du message)
 - **Splitter et Aggregator** (pour découper un message en petits messages et inversement pour recomposer un gros message depuis plusieurs parties).
 - **Service Activator** (pour déclencher une méthode de traitement)
 - "**Channel adapter**" : *connecteurs* externes (ex: HTTP , JMS ,)



Analogie : fleuve(JMS) et canaux(spring-integration) ?

14. "channels"

14.1. Interfaces java pour "Channels"

```
public interface MessageChannel {
    boolean send(Message message);
    boolean send(Message message, long timeout);
}
```

```
public interface PollableChannel extends MessageChannel {
    Message<?> receive();
    Message<?> receive(long timeout);
}
```

```
public interface SubscribableChannel extends MessageChannel {
    boolean subscribe(MessageHandler handler);
    boolean unsubscribe(MessageHandler handler);
}
```

14.2. Implémentations "channels"

<i>Classe d'implémentation</i>	<i>Principale Interface implémentée</i>	<i>caractéristiques</i>
PublishSubscribeChannel	SubscribableChannel	Mode publication/souscription
QueueChannel	PollableChannel	Mode point à point , "FIFO" avec capacité maxi .
PriorityChannel	PollableChannel	Comme QueueChannel mais avec niveau de priorité en plus.
RendezvousChannel	PollableChannel	Sorte de "QueueChannel" à capacité vide : send() bloquant , tant qu'aucun receive() depuis autre thread.
DirectChannel	SubscribableChannel	Sorte de "PublishSubscribeChannel" limité à un seul receveur (sémantiquement proche du point à point) mais avec "handler" coté réception.
ExecutorChannel	SubscribableChannel	Sorte de DirectChannel mais avec un "TaskExecutor" géré par un autre thread coté réception.

14.3. ChannelInterceptor

```
public interface ChannelInterceptor {
```



```

Message<?> preSend(Message<?> message, MessageChannel channel);
void postSend(Message<?> message, MessageChannel channel, boolean sent);
boolean preReceive(MessageChannel channel);
Message<?> postReceive(Message<?> message, MessageChannel channel);
}

```

enregistrement "en java":

```
channel.addInterceptor(someChannelInterceptor);
```

enregistrement "spring":

```

<channel id="exampleChannel">
<interceptors>
  <ref bean="trafficMonitoringInterceptor"/>
</interceptors>
</channel>

```

WireTap (intercepteur prédéfini pour envoyer quelque-part une copie des messages)

```

<channel id="in">
<interceptors>
  <wire-tap channel="logger"/>
</interceptors>
</channel>

<logging-channel-adapter id="logger" level="DEBUG"/>

```

14.4. Configuration "spring" d'un "channel"

```
<channel id="exampleChannel"/> <!-- en mode point à point (DirectChannel par défaut) -->
```

```

<channel id="queueChannel">
  <queue capacity="25"/> <!-- QueueChannel avec capacité choisie -->
</channel>

```

```

<channel id="priorityChannel">
  <priority-queue capacity="20"/> <!-- PriorityChannel avec capacité choisie -->
</channel>

```

```

<channel id="rendezvousChannel">
  <rendezvous-queue>
</channel>

```

```
<publish-subscribe-channel id="exampleChannel"/>
```

....

Canaux spéciaux:

nullChannel	
errorChannel	

15. "Channel Adapter" et "Bridge"

<i>ChannelAdapter</i>	<i>Caractéristiques</i>
Adapter for Spring Application Event	
Feed Adapter (ex:RSS , ATOM, ...)	
File Adapter	
FTP/FTP s Adapter	
Http (Inbound & Outbound) Gateway	
Mail Adapter (sending , receiveing)	
TCP (and UDP) Adapters	
JDBC Adapters (inbound & outbounds)	
JMS Adapters	
RMI Gateway	
(java.io) Stream Adapters	
Twitter Adapter	
Web Services Gateway	
Xml, Xpath & Xslt Support (<i>transformers</i>)	
Xmpp Adapter	

15.1. InBound Channel Adapter (configuration)

```
<inbound-channel-adapter ref="source1" method="method1" channel="channel1">
  <poller fixed-rate="5000"/>
</inbound-channel-adapter>
```

```
<inbound-channel-adapter ref="source2" method="method2" channel="channel2">
  <poller cron="30 * 9-17 * * MON-FRI"/>
</inbound-channel-adapter>
```

15.2. OutBound Channel Adapter (configuration)

```
<outbound-channel-adapter channel="channel1" ref="target" method="handle"/>
```

```
<beans:bean id="target" class="org.Foo"/> <!-- pojo consumer -->
```

15.3. Messaging Bridge

```
<bridge input-channel="input" output-channel="output"/>
```

```
<bridge input-channel="pollable" output-channel="subscribable">
  <poller max-messages-per-poll="10" fixed-rate="5000"/>
</bridge>
```

16. messages (structure et construction)

16.1. Structure d'un message (spring-integration)

```
public interface Message<T> {
  T getPayload();
  MessageHeaders getHeaders();
}
```

```
public final class MessageHeaders implements Map<String, Object>, Serializable {
  ...
}
```

Exemples d'accès aux propriétés de l'entête :

```
Object someValue = message.getHeaders().get("someKey");
CustomerId customerId = message.getHeaders().get("customerId", CustomerId.class);
Long timestamp = message.getHeaders().getTimestamp();
```

Propriétés prédéfinies de l'entête:

Header Name	Header Type
ID	java.util.UUID
TIMESTAMP	java.lang.Long
CORRELATION_ID	java.lang.Object
REPLY_CHANNEL	java.lang.Object (can be a String or MessageChannel)
ERROR_CHANNEL	java.lang.Object (can be a String or MessageChannel)
SEQUENCE_NUMBER	java.lang.Integer
SEQUENCE_SIZE	java.lang.Integer
EXPIRATION_DATE	java.lang.Long

PRIORITY	MessagePriority (an <i>enum</i>)
-----------------	-----------------------------------

```
public enum MessagePriority {
HIGHEST,
HIGH,
NORMAL,
LOW,
LOWEST
}
```

16.2. Implémentations et constructions des messages

```
new GenericMessage<T>(T payload);
new GenericMessage<T>(T payload, Map<String, Object> headers)
```

```
ErrorMessage message = new ErrorMessage(someThrowable);
Throwable t = message.getPayload();
```

via classe utilitaire "*MessageBuilder*":

```
Message<String> message1 = MessageBuilder.withPayload("test")
.setHeader("foo", "bar")
.build();
Message<String> message2 = MessageBuilder.fromMessage(message1).build();
assertEquals("test", message2.getPayload());
assertEquals("bar", message2.getHeaders().get("foo"));
```

```
Message<String> message3 = MessageBuilder.withPayload("test3")
.copyHeaders(message1.getHeaders())
.build();
Message<String> message4 = MessageBuilder.withPayload("test4")
.setHeader("foo", 123)
.copyHeadersIfAbsent(message1.getHeaders())
.build();
```

```
Message<Integer> importantMessage = MessageBuilder.withPayload(99)
.setPriority(MessagePriority.HIGHEST)
.build();
```

17. Routage des messages

17.1. "Router"

17.2. "Filter"

17.3. "Aggregator"

17.4. "Resequencer"

17.5. "Message Handler Chain"

18. Transformations des messages

18.1. "Transformer"

18.2. Content-enricher

18.3. Claim-check

19. (Message) Endpoint

19.1. Message endpoints

Message Handler

Event Driven Consumer

Polling consumer

Asynchronous Polling

19.2. InBound messaging gateways

19.3. ServiceActivator

19.4. Delayer

20. Gestion système (administration , supervision)

20.1. JMX

20.2. Historique des messages

20.3. Control Bus

21. Chanel Adpater (avec principaux détails)

21.1. File Adapter

21.2. FTP/FTP's Adapter

21.3. Http (Inbound & Outbound) Gateway

21.4. Mail Adapter (sending , receiveing)

21.5. TCP Adapters

21.6. JDBC Adapters (inbound & outbounds)

21.7. JMS Adapters

Lien avec JMS :

destination="inQueue" (spring JmsTemplate)

ou bien destination(Name) + connectionFactory

```
<jms:inbound-channel-adapter id="jmsIn" destination="inQueue" channel="exampleChannel">
  <integration:poller fixed-rate="30000"/>
</jms:inbound-channel-adapter>
```

```
<jms:outbound-channel-adapter id="jmsOut" destination="outQueue"
  channel="exampleChannel"/>
```

Si extract-payload est à false on récupère le message JMS "brut" ,
sinon (si extract-payload est à "true" explicitement ou par bien défaut), on récupère un payload selon le type de message JMS (ex: String si JMS texte , Map si JMS Map, ...) .

```
<jms:inbound-channel-adapter id="jmsIn"
  destination="inQueue"
  channel="exampleChannel"
  extract-payload="false"/>
  <integration:poller fixed-rate="30000"/>
</jms:inbound-channel-adapter>
```

Lien avec "messageListener" de JMS:

```
<jms:message-driven-channel-adapter id="jmsIn" .... channel="exampleChannel"/>
```

21.8. RMI Gateway

21.9. (java.io) Stream Adapters

21.10. Web Services Gateway

21.11. Xml, Xpath & Xslt Support (transformers)

22. Aspects divers et avancés

22.1. Intercepteurs et AOP

..

22.2. Transactions

...

22.3. Sécurité

...

23. Exemple(s) complet(s)

...