Framework

SPRING 4

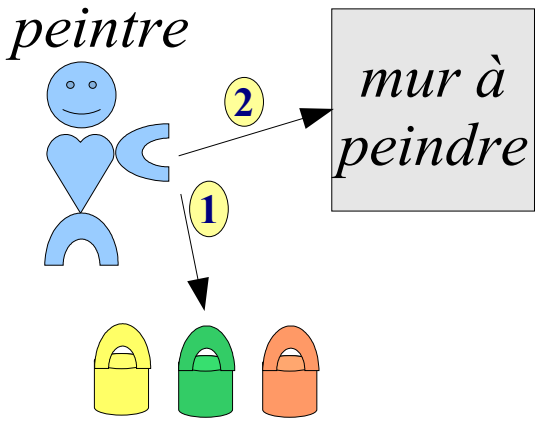
Table des matières

I - Présentation (Spring , IOC , architectures)

1. Design Pattern "I.O.C." / injection de dépendances

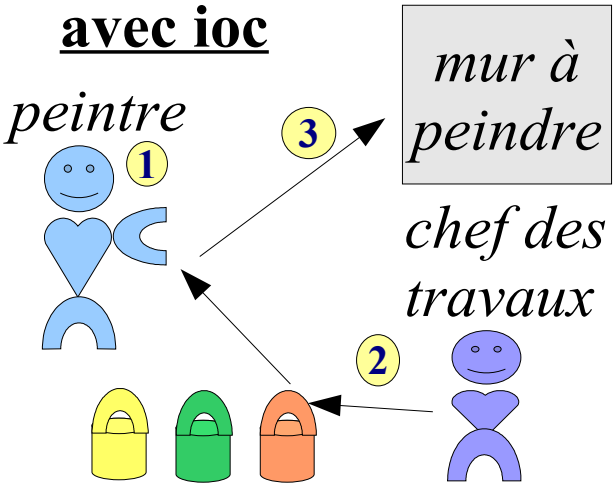
1.1. IOC = inversion of control

sans ioc



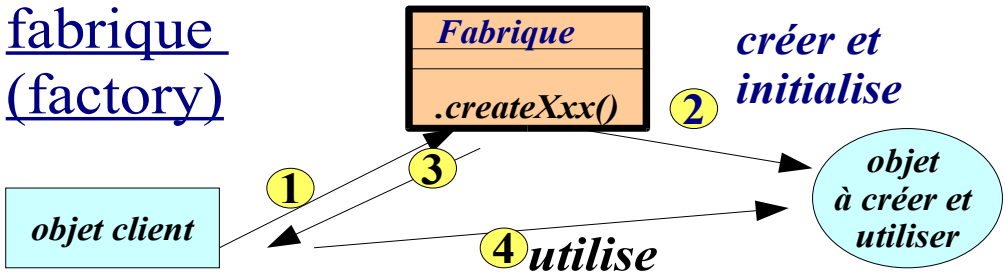
1) le peintre ***choisit*** la ressource "pot de peinture" *(il a le contrôle sur la couleur)*
2) il l'***utilise*** pour peindre

avec ioc

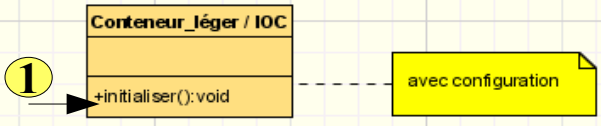


1) le peintre ***attend qu'on lui fournisse*** le pot de peinture
2) le chef des travaux lui donne le pot à utiliser *(c'est maintenant lui qui contrôle la couleur)*
3) ***utilisation*** du pot pour peindre

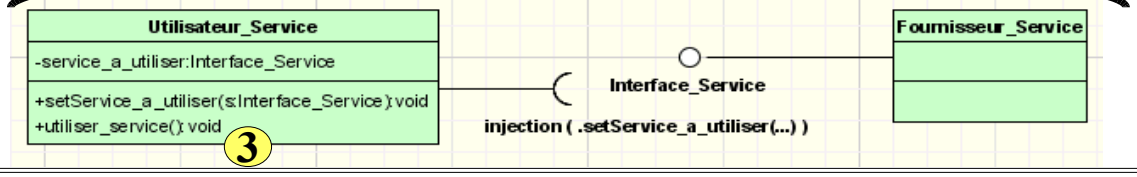
Simple fabrique (factory)



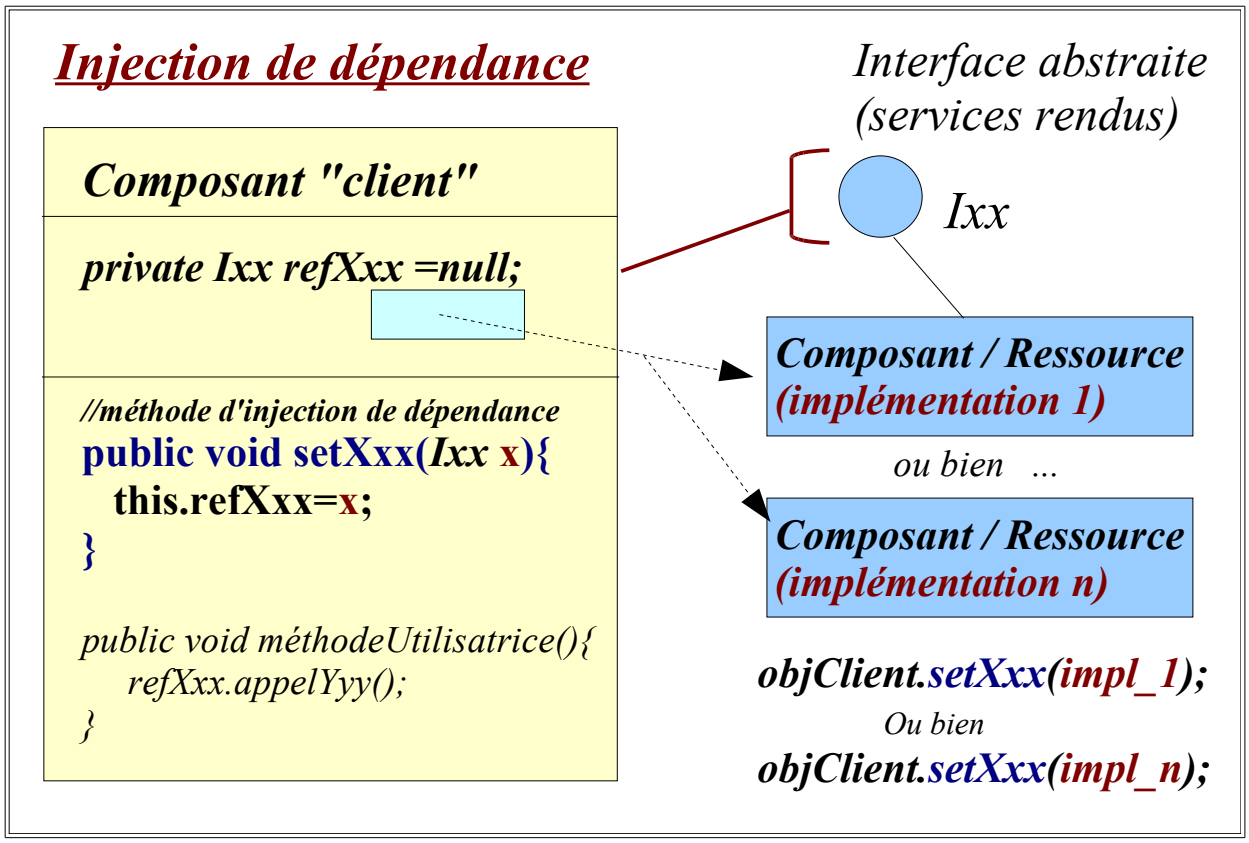
I.O.C.
= Inversion Of Control
(Injection de dépendances)



2 ***créer , initialise et met en relation***



1.2. injection de dépendance



Le ***design pattern "IOC"*** (*Inversion of control*) correspond à la notion d'**injection de dépendances abstraites**.

Concrètement au lieu qu'un composant "client" trouve (ou choisisse) lui même une ressource avant de l'utiliser , cet **objet client exposera une méthode de type:**

public void setRessources(AbstractRessource res)

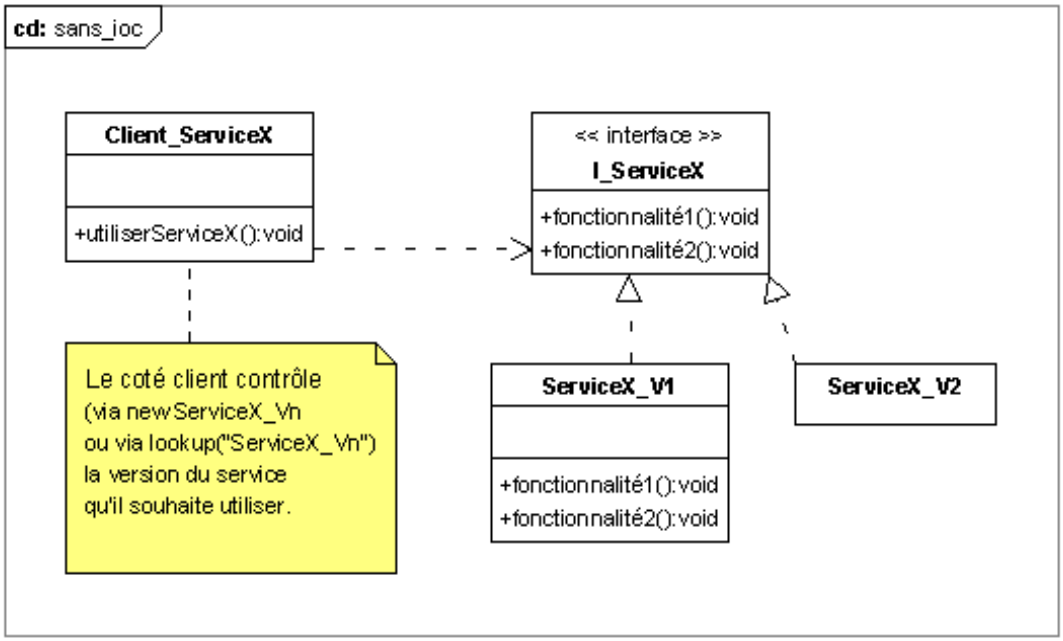
ou bien un constructeur de type:

public CXxx(AbstractRessource res)

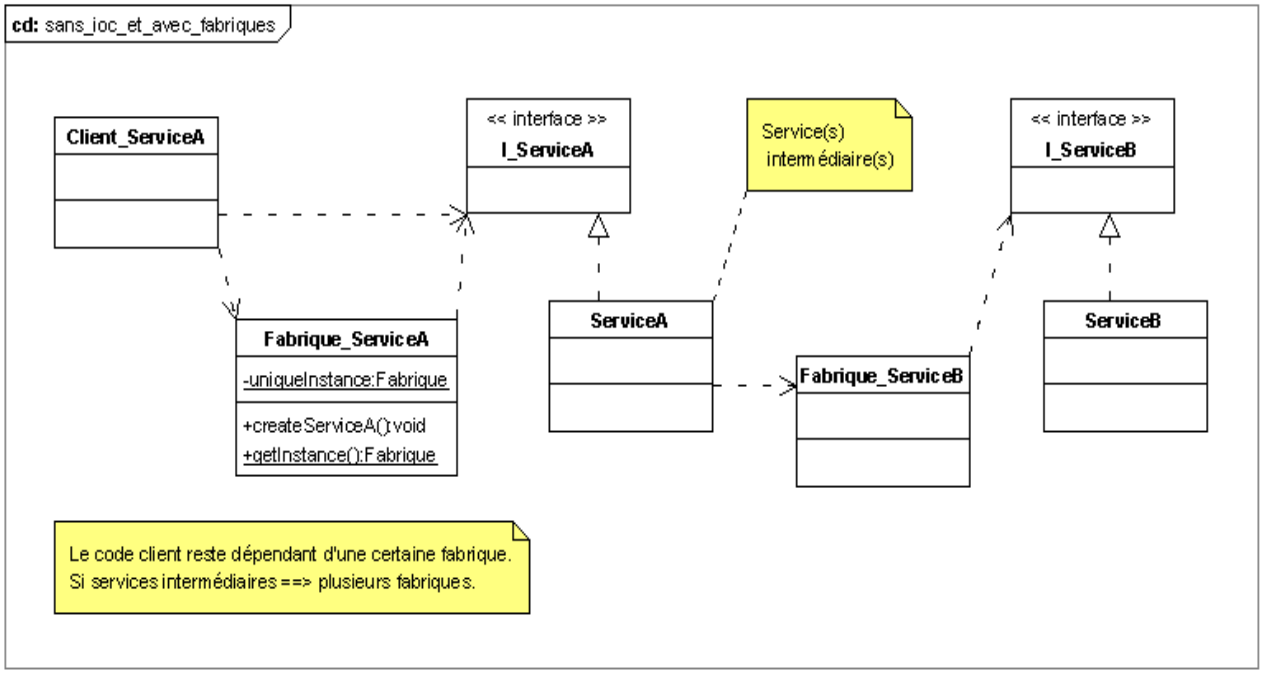
permettant qu'on lui fournisse la ressource à ultérieurement utiliser.

Un tel composant est beaucoup plus réutilisable .

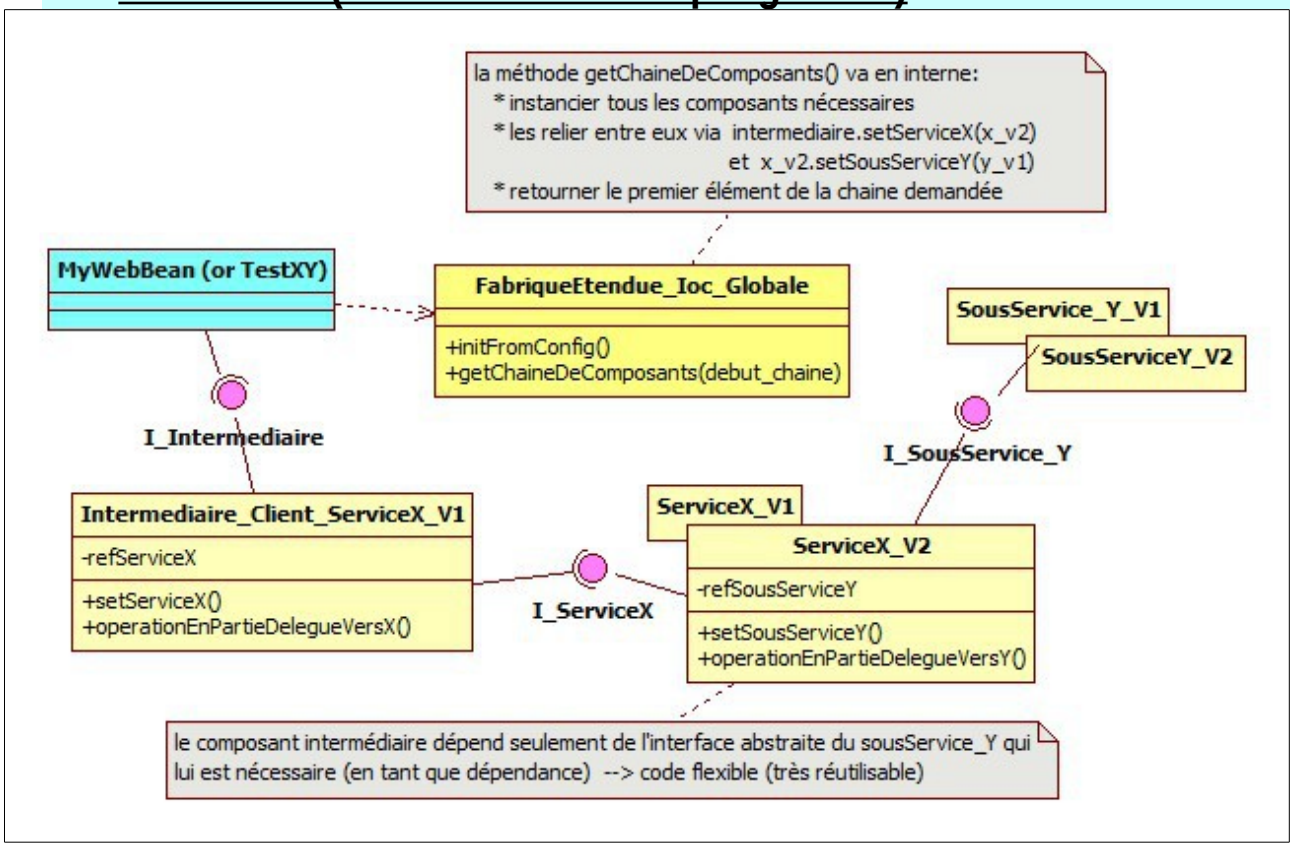
1.3. sans I.O.C. ni fabrique



1.4. sans I.O.C. et avec de multiples petites fabriques



1.5. avec I.O.C. (et donc avec fabrique globale)



1.6. Micro-kernel / conteneur léger

Pour être facilement exploitable , le design pattern IOC nécessite un **petit framework** généralement appelé "**micro-kernel**" ou "**conteneur léger**" prenant à sa charge les fonctionnalités suivantes:

Enregistrement des "ressources" (composants concrets basés sur interfaces abstraites) avec des **identifiants** *(noms logiques)* associés.

Instanciation et/ou initialisation des composants en tenant compte des dépendances à injecter (==> liaisons automatiques avec composants "ressources" nécessaires)

Ceci nécessite **quelques paramétrages** (***fichier de configuration XML*** ou bien ***annotations*** au sein du code).

2. Présentation de Spring

Spring fut l'un des tous premiers frameworks IOC qui a été mis au point.

Un peu comme STRUTS à son époque, le framework Spring fut à son origine très novateur.

Il est aujourd'hui adopté par une assez grande communauté de développeurs JEE.

Spring peut être considéré comme un standard de fait (à l'époque de J2EE 1.4).

Néanmoins, Spring n'est pas un standard officiel de Sun/JavaSoft et quelques bonnes idées de Spring ont été reprises dans les nouvelles versions officielles de JEE5 (ex: EJB3) et JEE6.

Spring est un excellent framework pour obtenir rapidement une architecture logicielle à la fois simple et très modulaire.

Principales fonctionnalités de Spring

- **Framework IOC perfectionné (pour *assembler les parties de façon modulaire*)**
- ***Configuration xml* et/ou par *annotations* .**
- **Bon support des transactions (avec ou sans EJB , locales / JTA , ...)**
- **Facilités appréciables pour la mise en oeuvre des DAO (Data Access Object)**
- **Bon support/intégration des principales technologies ORM (*Hibernate, JPA, ...*) et de JDBC**
- **L'intégration de composants "Spring" au niveaux des frameworks WEB (STRUTS , JSF , ...) est prête et opérationnelle.**
- **Proxy automatiques pour les technologies "RPC Objets" (RMI, EJB , Services WEB,....) --> développement rapide de "business delegate"**
- **Programmation par aspects (Spring-AOP) , ...**
- *Tests unitaires simple à mettre en oeuvre (via Junit ou autre)*

Qualités récurrentes de la plupart des composants "Spring":

- **Code Java basé sur des interfaces simples ou bien des POJO/JavaBean ==> Assez peu de chose(s) imposée(s) ==> Grande souplesse d'utilisation (héritage possible/libre , ...) .**
- **La vision abstraite des composants applicatifs** (DAO , Services métiers) est **100% fonctionnelle** (aucun élément technique imposé par Spring) ==> très grande souplesse/modularité (réutilisation possible des interfaces abstraites même au dehors de Spring) .
- **Seules certaines implémentations internes sont fortement liées à Spring** (implémentation à base d'héritage, ...)

En bref , Spring incarne très bien les concepts de modularité et de liberté au sein des développements J2EE/JEE5/JEE6.

URL du site web de référence: http://projects.spring.io/**spring-framework**

Spring est **avant tout** un **framework d'intégration** :

--> il intègre et assemble différentes technologies complémentaires (JSF , JPA/Hibernate , CXF/WebServices, ...) .

Le seul autre "framework d'intégration" java concurrent de Spring (et sérieux) est aujourd'hui le

framework "Seam" de Jboss .

Les frameworks "Seam" et "Spring" se ressemblent beaucoup. Ils apportent à peu près les mêmes fonctionnalités. La principale différence entre ces deux frameworks est l'utilisation omniprésente des "EJB3" au sein de Seam qui est n'est par contre pas présente dans Spring (car "*partie de Spring" proposée comme alternative aux EJB*) .

3. Principaux Modules de Spring

Modules de Spring	Contenus / spécificités
Spring **Core** + **Spring Beans**	conteneur léger – IOC (base du framework – BeanFactory)
Spring **AOP**	prise en charge de la programmation orientée aspect
Spring **DAO**	Classes d'exceptions pour DAO (Data Access Object), Classes abstraites facilitant l'implémentation d'un DAO basé sur Hibernate ou JDBC. Infrastructure/support pour les transactions
Spring **Context**	Classes d'implémentation (POJO Wrapper) et de proxy pour les technologies distribuées (EJB, Services Web , RMI , JMS, ...) + Contexte abstrait pour JNDI , ...
Spring **ORM**	Support abstrait pour les technologies de mapping objet/relationnel (ex: TopLink , **Hibernate** , iBatis, JDO, **JPA** ...)
Spring **Web**	WebApplicationContext , support pour le multipart/UploadFile, points d'intégration pour des frameworks STRUTS , JSF, ...
Spring Web **MVC** **(optionnel)**	Version "Spring" pour un framework Web/MVC. Ce framework est "simple/extensible" et "IOC". --> *Yet Another Framework !!!* (***très peu utilisé*** car bien sur "modularité/ioc" mais pauvre graphiquement et assez complexe à paramétrer)

==> soit un seul grand "***spring.jar***" ou bien

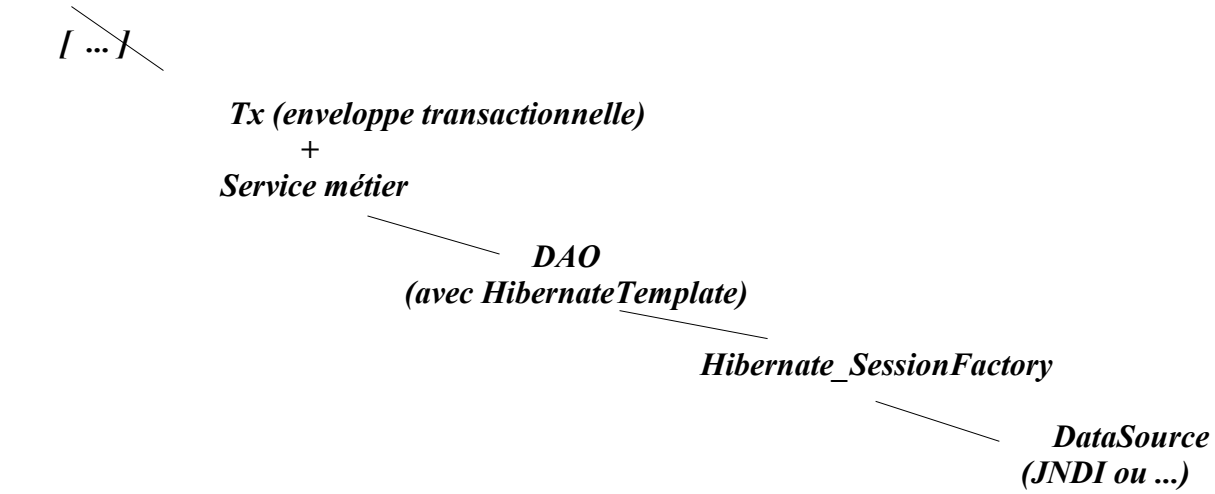
plusieurs petits "***spring-moduleXY.jar***" complémentaires (souvent précisés via "maven").

Modules complémentaires pour Spring (extensions facultatives) :

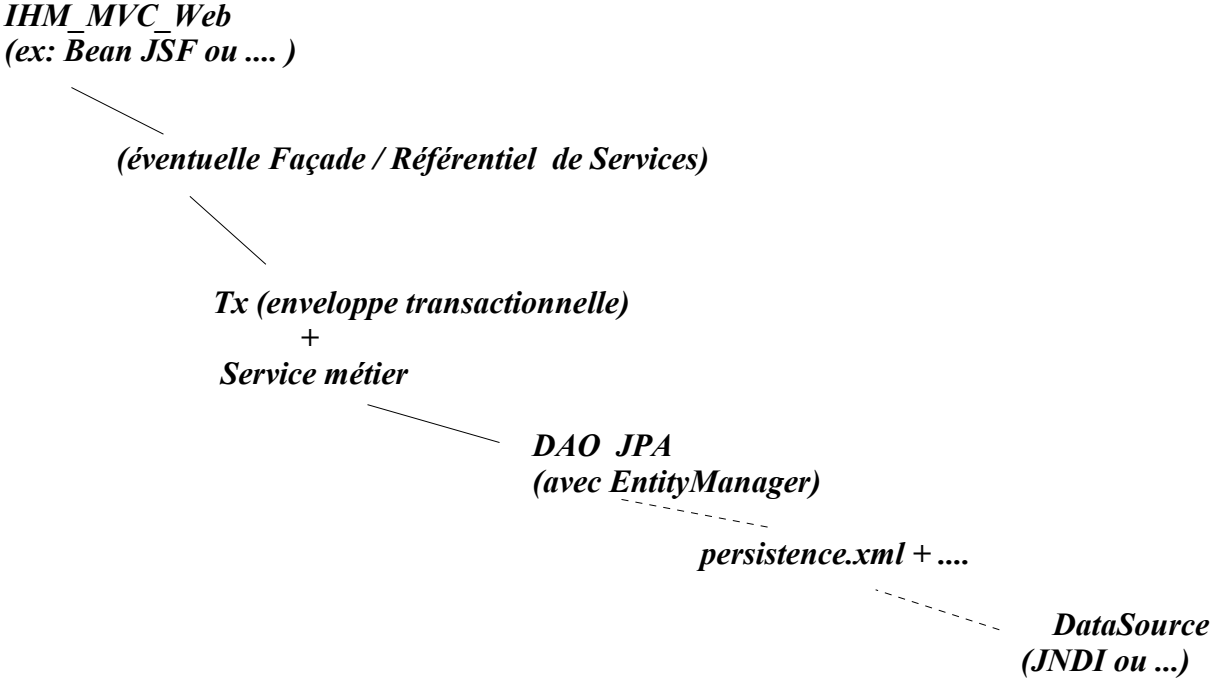
Extensions Spring	Contenus / spécificités
Spring **Web flow**	Extension pour bien contrôler la navigation et rendre abstraite l'IHM (paramétrages xml des états , transitions, ...)
Spring **Batch**	prise en charge efficace des traitements "batch" (job , ...)
Spring **Integration**	Extensions pour SOA (fonctionnalités d'un mini ESB , EIP, ...)

4. injections classiques / applications n-tiers:

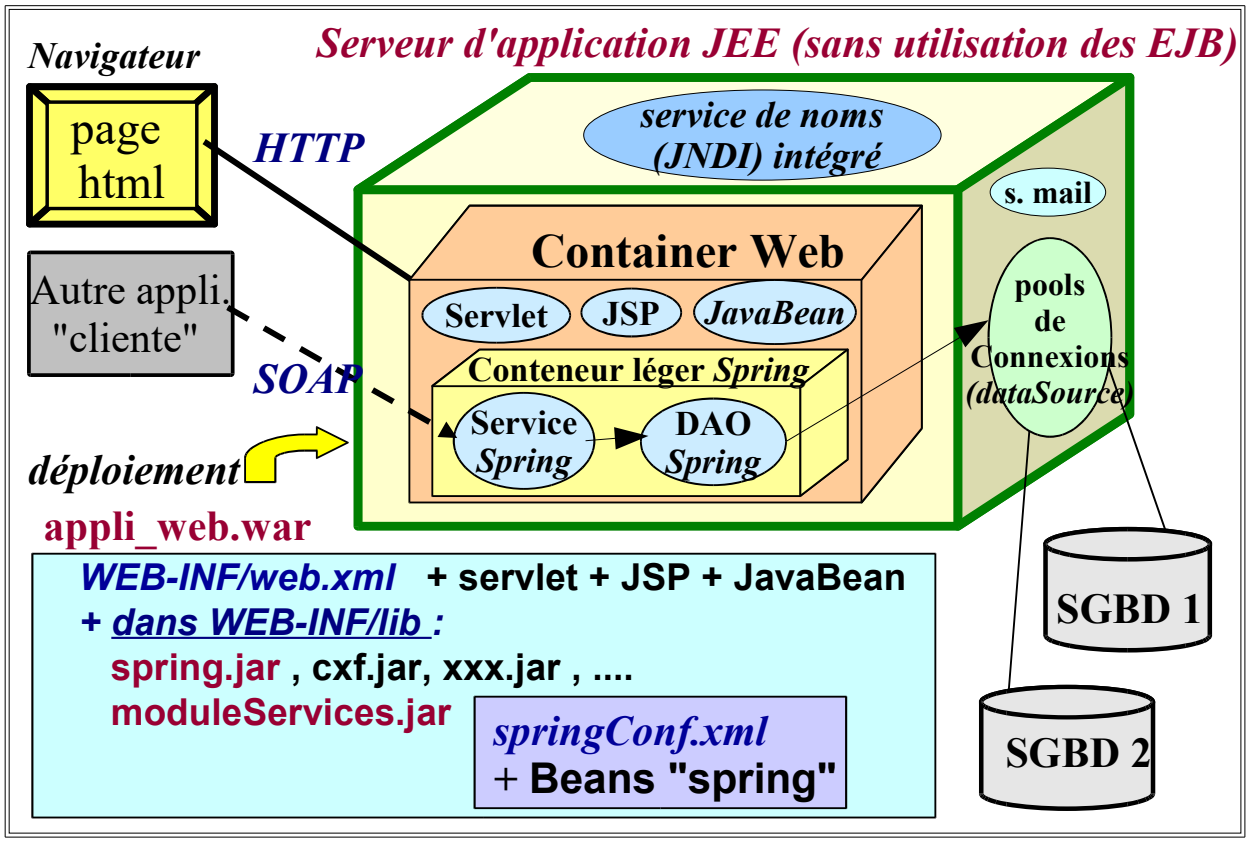
Chaîne d'injection classique (Spring/Hibernate) des années 2005/2008:



Chaîne d'injection classique (Spring/JPA) des années 2009/2012:



5. Intégration Spring dans serveur d'application JEE



Spring peut être vu comme un "**conteneur léger**" utilisé à la place du conteneur d'EJB .

Attention, le diagramme ci-dessus ne montre qu'une possibilité d'utiliser Spring parmi plein d'autres !!! Les "beans" de la partie Web peuvent éventuellement être pris en charge par Spring également .

II - Configuration(s) Spring & Tests

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.

Les noms choisis des fichiers de configuration Spring seront précisés au lancement des tests unitaires (ex: Junit) ou bien au démarrage de l'application "web" dans le serveur d'application (paramétrage <context-param> lié à un "listener web" dans WEB-INF/web.xml) .

L'entête des fichiers "Spring" était basée sur des "DTD" dans les anciennes versions "1.x" de Spring:

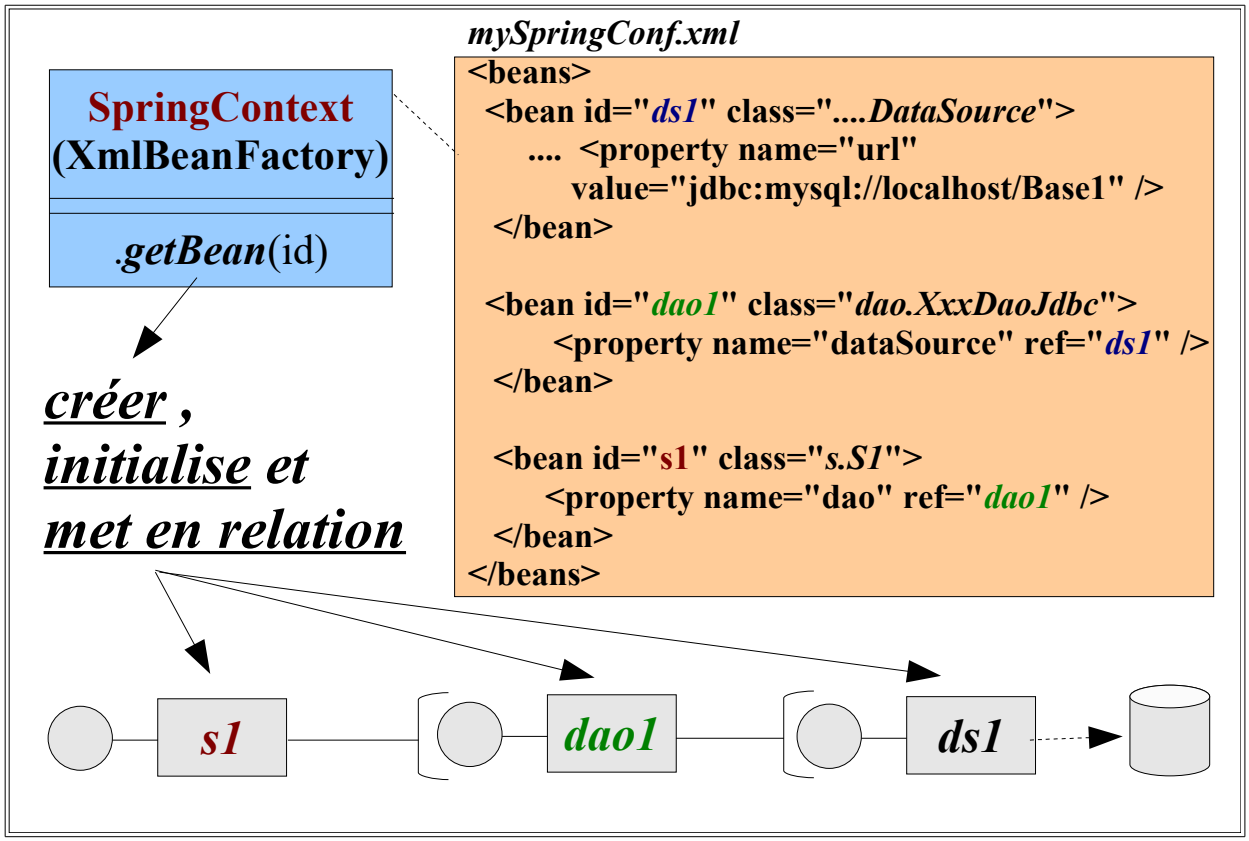
```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
   ....
</beans>
```

Dans les versions récentes de Spring (2.0 , 2.5 , 3.0) , il faut utiliser des entêtes basées sur des schémas "xsd" de façon à bénéficier de toutes les possibilités du framework.

Cependant, 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-2.5.xsd
       http://www.springframework.org/schema/aop
            http://www.springframework.org/schema/aop/spring-aop-2.5.xsd
       http://www.springframework.org/schema/tx
            http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
       http://www.springframework.org/schema/context
            http://www.springframework.org/schema/context/spring-context-2.5.xsd" >

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



Exemple de fichier de configuration Spring:

mySpringConf.xml

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

  <bean id="ds1"
        class="org.apache.commons.dbcp.BasicDataSource"
        destroy-method="close">
    <property name="driverClassName" value="com.mysql.jdbc.Driver" />
    <property name="url" value="jdbc:mysql://localhost:3306/mydb" />
    <property name="username" value="someone" />
  </bean>


   <bean id="dao1"
                class="xxx.dao.jdbc_impl.ExampleDataAccessObject">
    <property name="dataSource" ref="ds1" />
  </bean>


   <bean id="s1"
                class="xxx.services.MyJavaService">
    <property name="dataAccessObject" ref="dao1"/>
    <property name="exampleParam" value="10" />
  </bean>

</beans>
```

NB: 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 un utiliser des identifiants proches des noms des classes des objets (ex : nom de classe en remplaçant la majuscule initiale par une minuscule)

1.2. 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"));
MyJavaService s1 = (MyJavaService) 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" .

1.3. 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 context =
      new ClassPathXmlApplicationContext(
                new String[] {"mySpringConf.xml",
                              "config2.xml"});
//BeanFactory bf = (BeanFactory) context;
MyJavaService s1 = (MyJavaService) context.getBean("myService");
...
```

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

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.

Une classe utilitaire basée sur un singleton peut par exemple être utilisée pour configurer

efficacement la récupération du contexte Spring.

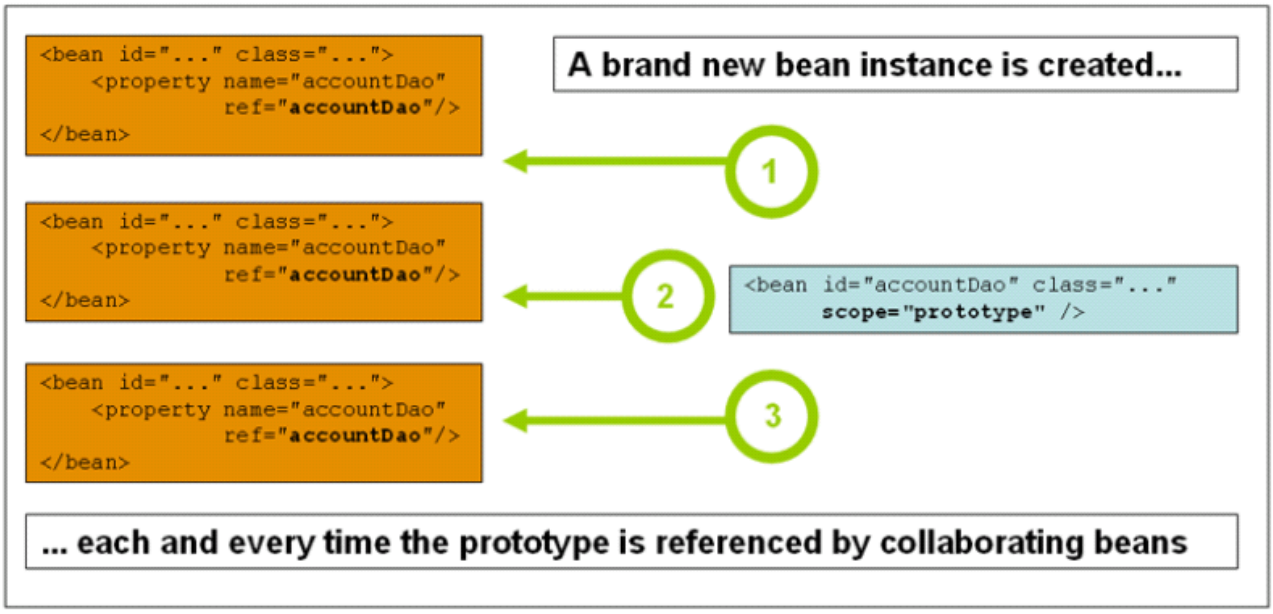
```
package tp.test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MySpringContextUtil {
        private static   ApplicationContext   mySpringContext =null;  // singleton

        public static ApplicationContext   getMySpringContext(){
                if(mySpringContext==null){
                        mySpringContext =new ClassPathXmlApplicationContext(new String[]
                                                                        {"mySpringConf.xml"});
                }
                return mySpringContext;
        }
}
```

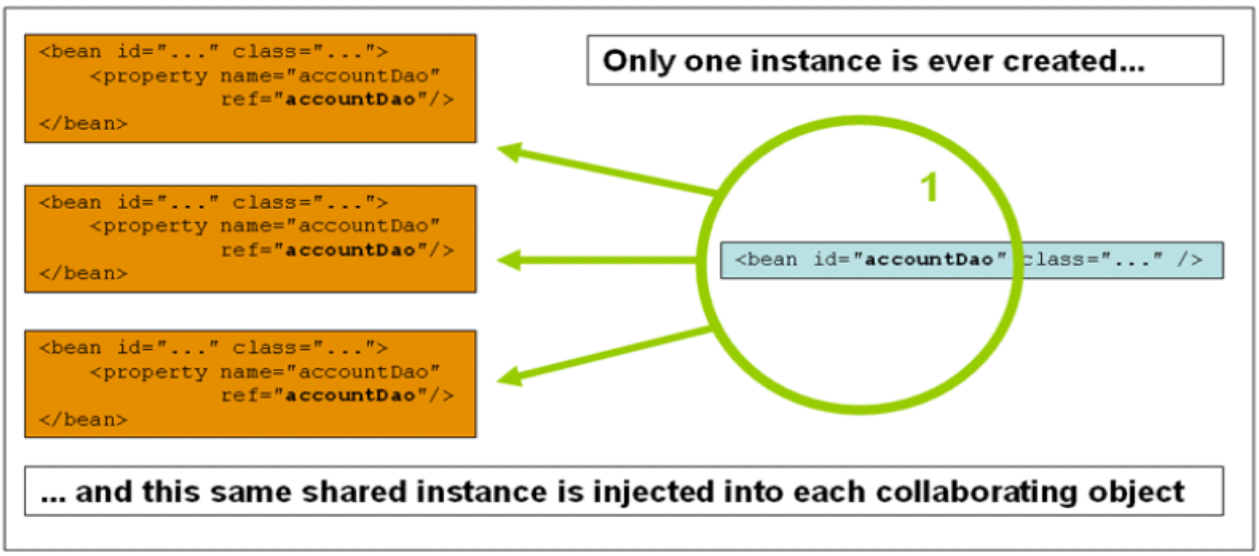
```
package tp.test;
import ....;

public class TestXY extends TestCase{
         private MyJavaService s1 =null;

        protected void setUp() throws Exception {
                ApplicationContext  ctx =   MySpringContextUtil. getMySpringContext();
                 s1 = (MyJavaService) ctx.getBean("myService");
        }

        public void testXxx() {
                Xx x = s1.getXxxById(1);
                 TestCase.assertTrue(  x.getId()==1  );
         }

        public void testYyy() { }   // @Test avec Junit4

        /* public static void main(String[] args) {
                TestXY testObj = new TestXY();
                try {    testObj.setUp();
                        testObj.testXxxx();
                        testObj.testYyyy();
                        //testObj.tearDown();
                } catch (Exception e) {e.printStackTrace();
             }
        } */
}
```

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

portée (scope) d'un	*comportement / cycle de vie*

composant Spring	
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 ratachée à une requête Http (valable uniquement au sein d'un "web-aware ApplicationContext")
global session	(global session) pour "portlet" par exemple [web uniquement]





1.5. Alias

Via la syntaxe *<alias name="nom existant" alias="nouveau nom" />* il est possible de donner d'autres nouveaux noms à un composant Spring existant.

Ceci peut être utile pour paramétrer certaines injections avec des noms attendus particuliers.

exemples:

```
<!-- éventuelle switch de version via une reconfiguration d'alias -->
<-- <alias  name="compteDaoJdbc"     alias="compteDao" /> -->
    <alias  name="compteDaoHibernate" alias="compteDao" />
```

```
<!-- alias sans "." mais "_" pour compatibilité des noms avec JSF -->

<alias  name="as.gestionComptes" alias="as_gestionComptes" />
```

1.6. Syntaxe abrégée alternative via "p-namespace" de Spring2

Depuis la version 2 de Spring , il est éventuellement possible de paramétrer les propriétés des beans via une syntaxe abrégée à base d'attributs (au lieu d'utiliser des sous balises <property ..../>).

```
<?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:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

       <bean id="myXyDao" class="tp.persistance.with_xml.XYDaoImpl" />

       <bean id="serviceXY" class="tp.domain.with_xml.ServiceXY"
             p:xyDao-ref="myXyDao"
             p:paramXy="valeurXY">
       </bean>
</beans>
```

est équivalent à

```
<?xml version="1.0" encoding="UTF-8"?>
<beans  ....>
       <bean id="myXyDao" class="tp.persistance.with_xml.XYDaoImpl" />

       <bean id="serviceXY" class="tp.domain.with_xml.ServiceXY" >
             <property  name="xyDao"     ref="myXyDao" />
             <property  name="paramXy"   value="valeurXY" />
       </bean>
</beans>
```

et le suffixe "-ref" est réservé au propriétés qui concernent des injections de dépendances.

--> en gros "syntaxe plus compacte" mais "moins claire" !!!

1.7. Eventuel mode "auto-wire" (auto-liaisons)

L'attribut optionnel **autowire** peut avoir deux valeurs importantes "***byType***" et "***byName***" .

Une **auto-liaison par nom** est effectuée en **injectant un composant Spring existant dont le nom (ou l'id) est le même que le nom de la propriété à fixer** par injection de dépendance.

Exemple:

```
<bean id="myDao" class="xxx.dao.jdbc_impl.ExampleDataAccessObject">...  </bean>

<bean id="myService"  class="xxx.services.MyJavaService">
   <property name="myDao"   ref="myDao"/>
</bean>
```

peut éventuellement être paramétré comme suit:

```
<bean id="myDao" class="xxx.dao.jdbc_impl.ExampleDataAccessObject">...  </bean>

<bean id="myService" autowire="byName" class="xxx.services.MyJavaService">
</bean>
```

Une **auto-liaison par type** est effectuée en **injectant un composant Spring existant dont le type est le même que celui du paramètre d'entrée de la propriété à fixer** par injection de dépendance.

Exemple:

```
<bean id="myDao" class="xxx.dao.jdbc_impl.ExampleDataAccessObject">...  </bean>

<bean id="myService" autowire="byType" class="xxx.services.MyJavaService">
</bean>
```

Permet d'injecter automatiquement le composant "myDao" dans la propriété myDao du service métier si le **type** est **compatible** .

Ce qui est le cas si la classe du DAO (ici "*xxx.dao.jdbc_impl.ExampleDataAccessObject*") implémente une interface **"*IDaoXXX*"** et que la méthode d'injection au niveau du service est une signature du type "**public void setMyDao(*IDaoXXX dao*)**" .

Le mode **"autowire"** correspond à de la **configuration implicite (non explicite)** !!!

Ce mode "auto-wire" est de moins en moins utilisé en Xml mais de plus en plus au sein de la configuration par annotations (exposée dans le chapitre suivant).

1.8. 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***" .

2. Paramétrages Spring quelquefois utiles

2.1. 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"/>
...
```

2.2. 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"/>
```

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

2.3.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" />
```

2.3.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>
```

2.4. Autres possibilités de Spring

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

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

3. 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 EE5 (@Resource , ...)**

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

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

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

3.1. Configuration xml indispensable pour 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-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

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

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

</beans>
```

3.2. Annotations (stéréotypées) pouvant préciser l'id

exemple : XYDaoImplAnot.java

```
package tp.persistance.with_anot;

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 .

Cependant, on peut également utiliser ces annotations pour **renseigner l'id** d'un composant Spring.

@Component	Composant quelconque
@Repository	Composant d'accès aux données (DAO)
@Service	Service métier
@Controller	Composant de contrôle IHM (coordinateur, ...)

3.3. Rare Injection paramétrée (en dur) via @Resource

exemple: ***`ServiceXYAnot.java`***

```
package tp.domain.with_anot;

import javax.annotation.Resource;

import org.springframework.stereotype.Service;
import tp.domain.IServiceXY;
import tp.domain.XY;
import tp.persistance.XYDao;

//@Component("serviceXY")  --- NB: @Service herite de @Component
@Service("serviceXY")
public class ServiceXYAnot implements IServiceXY {

      private XYDao xyDao;

      //@Resource(name="myXyDao")
      @Resource(name="myXyDaoV2")
      public void setXyDao(XYDao xyDao) {
            this.xyDao = xyDao;
      }

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

NB: l'annotation **`@Resource(name="myXyDaoV2")`**

peut éventuellement être placée directement au dessus l'attribut privée *xyDao* (et dans ce cas la méthode public "setXyDao" n'est plus nécessaire) .

NB: Si l'annotation **`@Resource()`** est utilisée sans le paramètre *name* , le nom (id) par défaut du composant à injecter correspondra alors au nom de la propriété (nom de l'attribut privé *xyDao* ou bien nom dérivé de la méthode publique d'injection *setXyDao*())

NB: Dans beaucoup de cas où il n'existe qu'un seul composant Spring implémentant une interface spécifique, **@Autowired()** peut être utilisé à la place de @Resource() . Ceci permet de demander l'injection de la seule ressource dont le type est compatible.

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

@Required (à placer au dessus d'une méthode d'injection)	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 (ex: "test" ou "prod" ou ...) dans le but de paramétrer plus finement les auto-liaisons (éventuel filtrage selon le qualificatif attendu)

NB: pour demander une (rare) auto-liaison par nom, il faut utiliser @Resource() sans argument .

Exemple (assez conseillé) avec @Autowired

```
@Service("serviceXY")
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);
	}
```

3.5. Eventuelle extension "java-config"

pour plus de souplesse (et peut être aussi plus de complexité) dans la configuration

==> extension facultative "java-config"

--> voir documentation de référence de Spring3

4. Nouvelles annotations CDI (Spring 3 , JEE 6)

JEE6 est maintenant décliné en une multitude d'API plus ou moins optionnelles.

Parmi elles , l'API **DI** *(Depedency Injection)* apporte enfin des annotations **@Named** et **@Inject** (censées être "standard" , c'est à dire avec des packages en javax.inject...) pour paramétrer les injections de dépendances.

Depuis la version 3 de Spring, il est possible d'utiliser ces annotations au niveau des composants gérés par le framework Spring .

Concrètement, ceci peut pragmatiquement se résumer en:

Annotation javax.inject.**Named**() a peu près équivalent à @**Component**() et servant donc à :

- donner un identifiant (éventuellement par défaut)
- déclarer que ce composant doit être pris en charge par le framework IOC (Spring ou ...)

Annotation javax.inject.**Inject**() a peu près équivalent à @**Autowired**() et servant à paramétrer une injection d'un composant de type compatible.

NB:

- Ces annotations nécessitent l'intégration du fichier **javax-inject-1.jar** (ou équivalent) dans le classpath.
- **@Inject** et **@Named** sont pour l'instant supportés par les frameworks ***Spring 3*** et ***Seam ...***

Exemple :

```
@Named("myXyDao")   //ou bien @Named()
public class XYDaoImplAnot implements XYDao {
…
}
```

et

```
@Named("serviceXY")
public class ServiceXYAnot implements IServiceXY {

	private XYDao xyDao;

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

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

Gros avantage de @Autowired ou @Inject
==> pas besoin de changer les noms en cas de refactoring !!!

4.1. Paramétrage XML de ce qui existe au sens "Spring"

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

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 @Named)

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

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

```
<?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-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
  <context:annotation-config/> <!-- pour demander à Spring de tenir compte de @ .... -->

  <context:component-scan base-package="xxx.yyy"/>
  <context:component-scan base-package="xxx.impl.dao.v1"/>
            <!-- pour indiquer à Spring quelles sont les classes à scaner pour trouver des annotations
                 telles que @Component , @Service , @Named , @Autowired , @Inject ou .... -->
</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

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

5. 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: 4.8.1) et Spring 3 .

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 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"})
public class TestGestionComptes {

private GestionComptes service = null;

 // injection du service à tester controlée par @Autowired (par type)
@Autowired //ou bien @Inject
public void setService(GestionComptes service) {
                this.service = service;
        }

// avec JUnit 4 , les méthodes de test doivent être  préfixées par @Test
 @Test
public void testTransferer(){
        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"})
@TransactionConfiguration(transactionManager="txManager",defaultRollback=false)
public class TestDaoXY {

// injection du doa à tester controlée par @Autowired (par type)
@Autowired //ou bien @Inject
private DaoXY xyDao = null;


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

...
}
```

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

III - Essentiel de Spring AOP

1. Spring AOP (essentiel)

1.1. Technologies AOP et "Spring AOP"

- AOP (Aspect Oriented Programming) est un complément à la programmation orientée objet.

- *AOP* consiste à programmer une bonne fois pour toute certains aspects techniques (logs , sécurité , transaction, ...) au sein de classes spéciales.
- Une configuration (xml ou ...) permettra ensuite à un framework AOP (ex: AspectJ ou Spring-AOP) d'appliquer (par ajout automatique de code) ces aspects à certaines méthodes de certaines classes "fonctionnelles" du code de l'application.

- *Vocabulaire AOP*:
  PointCut : endroit du code (fonctionnel) où seront ajoutés des aspects
  Advice : ajout de code/aspect (avant, après ou bien autour de l'exécution d'une méthode)
- On parle de tissage ("weaver") du code :
  Le code complet est obtenu en tissant les fils/aspects techniques avec les fils/méthodes fonctionnel(le)s .

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 du standard de fait java "AspectJ-weaver" .

1.2. Mise en oeuvre rapide de Spring aop via des annotations

```
package util;
//Nécessite quelquefois aspectjrt.jar , aspectjweaver.jar
//(de spring.../lib/aspectj)
import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;
```

```
@Aspect
public class MyLogAspect {

     @Around("execution(* xxx.services.*.*(..))")
     public Object doXxxLog(ProceedingJoinPoint pjp)
     throws Throwable {
          System.out.println("<< trace == debut == "
                    + pjp.getSignature().toLongString() + " <<");
          long td=System.nanoTime();
          Object objRes =  pjp.proceed();
          long tf=System.nanoTime();
          System.out.println(">> trace == fin == "
                    + pjp.getSignature().toShortString() +
                       " [" + (tf-td)/1000000.0 + " ms] >>");
          return objRes;
     }
}
```

avec `@Around("execution(typeRetour package.Classes.methode(..))")`

et dans ***dans myspringConf.xml***

```
....
 <aop:aspectj-autoproxy/>
     <bean id="myLogAspect"
          class="util.MyLogAspect">
     </bean>
....
```

Configuration aop en xml (sans annotations dans la classe java de l'aspect)

```
...
<bean id="myLogAspectBean"  class="tp.util.MyLogAspect"></bean>
  <aop:config>
     <aop:pointcut id="execution_methodes_package_livre"
     expression="execution(* tp.bibliotheque.livres.*.*.*(..))" />

     <aop:pointcut id="execution_methodes_package_ab_emp"
     expression="execution(* tp.bibliotheque.ab_emp.*.*.*(..))" />

     <aop:aspect id="myLogAspect" ref="myLogAspectBean" >
         <aop:around  method="doXxxLog"
               pointcut-ref="execution_methodes_package_livre" />
         <aop:around  method="doXxxLog"
              pointcut-ref="execution_methodes_package_ab_emp" />
     </aop:aspect>

 </aop:config>
```

IV - Services métiers et persistance

1. Utilisation de Spring au niveau des services métiers

1.1. Principales fonctionnalités d'un service métier

- Contrôler / superviser une séquence de traitements élémentaires sur quelques entités.
- Offrir des méthodes «créerXx rechercherXx , majXx , supprimerXx» (C.R.U.D.) dont le code interne consistera essentiellement à déléguer ces opérations de persistance aux D.A.O. (génériques ou spécifiques).
- Comporter des règles de gestions (méthodes vérifierXxx() , vérifierYyy()).
- Offrir des méthodes spécifiques à l'objet métier considéré (ex: transferer() , ...)
- Gérer/superviser des transactions (commit / rollback).

1.2. Vision abstraite d'un service métier

Interface abstraite avec méthodes *métiers* ayant:
- des POJOs de données en paramètres d'entrée et/ou en sortie (valeur de retour)
- des remontées d'exceptions métiers uniformes (héritant de *Exception* ou bien *RuntimeException*) quelque soit la technologie utilisée en arrière plan.

exemple:

```
//public class MyApplicationException extends RuntimeException {
public class MyApplicationException extends Exception {

        private static final long serialVersionUID = 1L;

        public MyApplicationException() {  super();}
        public MyApplicationException(String msg) {super(msg); }
        public MyApplicationException(String msg,Throwable cause) {super(msg,cause); }
}
```

et

```
public interface GestionComptes {

   public ae.Compte getCompteByNum(long numCpt) throws  MyApplicationException;
   ...
   public void transferer(long numCompteADebiter,
                          long numCompteACrediter,
                          double montant)      throws  MyApplicationException;
}
```

1.3. implémentations sous forme de POJO local

Simple classe java concrète implémentant l'interface abstraite du service.
C'est élémentaire mais efficace.
D'autant plus efficace que l'on peut configurer une gestion quasi automatique des transactions (via Spring & Spring-AOP). Ce dernier point (les transactions) sera étudié au sein d'un chapitre ultérieur.

```
public class GestionComptesImpl implements GestionComptes {

        private CompteDAO compteDao = null;
        ....
        public void setCompteDao(CompteDAO compteDao) { this.compteDao = compteDao; }
        ...
}
```

En règle générale, l'implémentation d'un service applicatif est souvent dépendante d'un (ou plusieurs) DAO (Data. Access. Object) :

```
<bean id="service_gestionComptes" class="services.GestionComptesImpl">
   ...
    <property name="compteDao" ref="compteDAO" />
  ...
</bean>
```

Néanmoins, lorsque la technologie **JPA** est utilisée, "***EntityManager***" peut éventuellement être directement utilisé par le service en tant que **DAO générique**.

1.4. implémentations sous forme de service distant et proxy

==> voir le chapitre ultérieur «Business Delegate»

2. DataSource (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).

2.1. Mise en oeuvre d'une source de donnée autonome (sans serveur J2EE)

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.apache.commons.dbcp.BasicDataSource***" (livrée avec Tomcat/Apache) correspond à une technologie que l'on peut intégrer facilement un peu partout (dans une application autonome , dans une application web (.war) , ...).
La classe "***org.springframework.jdbc.datasource.DriverManagerDataSource***" est une version simplifiée (juste pour les tests) qui a l'avantage de ne pas nécessiter de ".jar" supplémentaire.

Le fichier de configuration ci dessus montre que l'ensemble des paramètres nécessaires à la connexion JDBC vers la base de données sont intégrés dans la configuration "Spring".
Il n'est donc pas nécessaire de s'appuyer sur une ressource JNDI hébergée par un serveur.

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-3.0.16-ga-bin.jar*)
- les ".jar" nécessaires à "*org.apache.commons.dbcp.BasicDataSource*" (***commons-dbcp-1.2.1.jar*** + dépendances *commons-pool-1.2.jar , commons-xxx.jar*)

2.2. Mise en oeuvre classique d'une source de données JDBC (avec nom JNDI)

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 ou 6:

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>
```

Exemple de configuration sous JBoss 4 ou 5:

JBoss_Home/server/default/deploy/bankdb_mysql-ds.xml

```
<datasources>
  <local-tx-datasource>
    <jndi-name>BankDBDataSource</jndi-name> <!-- sera préfixé par java: -->
    <connection-url>jdbc:mysql://localhost/bankdb</connection-url>
    <driver-class>com.mysql.jdbc.Driver</driver-class>
    <user-name>mydbuser</user-name>    <password>mypwd</password>
  </local-tx-datasource>
</datasources>
```

3. Utilisation de Spring au niveau d'un DAO

3.1. Vision abstraite du DAO "classique" (Data Access Object)

Interface abstraite avec méthodes *CRUD (Create , Retreive , Update , Delete)* ayant:
- des POJOs de données en paramètres d'entrée et/ou en sortie (valeur de retour)
- des remontées d'exceptions uniformes (quelque soit la technologie utilisée).

exemple:

soit de.***Compte*** une classe (JavaBean / POJO) représentant une entité "Compte"
avec getNum() / setNum(...) , getSolde() / setSolde(...) , ...

la vision abstraite du DAO sera alors ressemblant à:

```
public interface CompteDAO
{
public java.util.Collection getCompteOfClient(int num_client) throws DataAccessException ;
public void updateCompte(de.Compte cpt) throws DataAccessException;
public void deleteCompte(de.Compte cpt) throws DataAccessException;
… }
```

Sur ce point , tout est classique; les interfaces sont complètement libres.
Spring offre simplement tout une hiérarchie de classes abstraites pour les exceptions (classes héritant de ***DataAccessException***) .

```
==> consulter la documentation de référence pour lister les sous classes de
DataAccessException.
```

3.2. Variantes et alternatives sur les D.A.O.

Si l'on ne souhaite pas être directement dépendant de ***DataAccessException*** (spécifique Spring) , on peut éventuellement mentionner la surclasse ***java.lang.RuntimeException*** (moins précise mais de type compatible) .

==> public void ***updateCompte***(de.Compte cpt) ***throws RuntimeException***;

ou bien (encore):

public void ***updateCompte***(de.Compte cpt); *//avec throws RuntimeException; implicite*

D'autre part , l'utilisation d'un DAO "classique/spécifique à un type d'entité" est facultative.

Un service métier peut très bien choisir d'utiliser directement "EntityManager" de l'api JPA en tant que DAO générique/universel .

NB: avec du temps (et une bonne organisation), on peut mettre en place une hiérarchie astucieuse concernant les "DAO" (Dao spécifiques héritant d'un Dao générique).

--> Etudier si besoin l'annexe sur les DAO génériques.

4. DAO Spring basé directement sur JDBC

De façon à coder rapidement une classe d'implémentation concrète d'un DAO basée directement sur la technologie JDBC on pourra avantageusement s'appuyer sur la classe abstraite ***JdbcDaoSupport***.

exemple:

```
public class JdbcCompteDAO extends JdbcDaoSupport implements CompteDAO
{
...
}
```

Schéma d'injections IOC:

```
...
<bean id="compteDAO" class="dao.JdbcCompteDAO">
    <property name="dataSource" ref="myDataSource" />
</bean>
...
```

Principales fonctionnalités héritées de JdbcDaoSupport:

==> ***getDataSource***() permet de récupérer au niveau du code la source de données JDBC qui a été obligatoirement injectée.
==> ***setDataSource***(DataSource ds) permet d'injecter la source de données JDBC .

==> ***getConnection***() , ***releaseConnection***(cn) permet d'obtenir et libérer une connexion JDBC .
NB: Prises en charge par Spring , les méthodes JdbcDaoSupport.***getConnection***() et JdbcDaoSupport.***releaseConnection***() seront automatiquement synchronisées avec le contexte transactionnel du thread courant (cn.close() différé après la fin de la transaction, ...).

==> **getJdbcTemplate**() permet de récupérer un objet de plus haut niveau (de type **JdbcTemplate**) qui libère automatiquement la connexion en fin d'opération et qui permet de simplifier un peu la syntaxe.

.../...

Exemple de code (classique JEE) n'utilisant pas de "JdbcTemplate":

```
public class JdbcInfosAccesDAO extends JdbcDaoSupport implements InfosAccesDAO {
        ...
   public InfosAcces getVerifiedInfosAccesV1(String userName, String password)
           throws   DataAccessException {
                InfosAcces infos=null;
                Connection cn = null;
                try
                {
                cn = this.getConnection();
                PreparedStatement prst = cn.prepareStatement("select NUM_CLIENT FROM
                                        INFOSACCES WHERE username=? and password=?");
                prst.setString(1,userName);
                prst.setString(2,password);
                ResultSet rs = prst.executeQuery();
                if(rs.next())
                  {
                        infos=new InfosAcces();
                        infos.setUserName(userName);  infos.setPassword(password);
                        infos.setClient_id(rs.getLong("NUM_CLIENT"));
                  }
                rs.close();
                prst.close();
                }
                catch(SQLException se)
                {
                        se.printStackTrace();
                        throw new DataRetrievalFailureException(se.getMessage());
                }
                finally
                {
                        this.releaseConnection(cn);
                }
                return infos;
        }
....
}
```

V - Intégration Spring ORM (Hibernate, JPA)

1. DAO Spring basé sur Hibernate 3 ou 4

1.1. 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.

1.2. 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 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 étés 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" />
```

2. DAO Spring basé sur JPA (Java Persistence Api)

2 grands «styles» de programmation sont possibles pour coder des «DAO Spring» basés sur JPA:

- **JpaTemplate** *(spécifique Spring , assez déconseillé (style obsolète))*
- **pure JPA** ***(style EJB3 , plus standard/portable)***

2.1. Rappel: Entité prise en charge par JPA

```
package entity.persistance.jpa;

import javax.persistence.Column; import javax.persistence.Entity;
import javax.persistence.Id; import javax.persistence.Table;

@Entity
@Table(name="Compte")
public class Compte {
      @Id
      @Column(name="NUM_CPT")
      private long numCpt;
      private String label;
      private double solde;
      public String getLabel() { return this.label; }
      public void setLabel(String label) { this.label=label; }
      public double getSolde() { return this.solde; }
      public void setSolde(double solde) {  this.solde= solde; }
      public long getNumCpt() { return numCpt; }
       public void setNumCpt(long numCpt){ this.numCpt= numCpt; }
}
```

2.2. unité de persistance (persistence.xml + config. Spring)

META-INF/persistence.xml

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
   http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd" version="1.0" >
<persistence-unit name="myPersistenceUnit"
                  transaction-type="RESOURCE_LOCAL">

 <!-- <provider>org.hibernate.ejb.HibernatePersistence</provider> -->
 <class>entity.persistance.jpa.Compte</class>
 <class>entity.persistance.jpa.XxxYyy</class>
 <properties>
    <property name="hibernate.dialect"
       value="org.hibernate.dialect.MySQLDialect" />
     <!--  <property name="hibernate.hbm2ddl.auto" value="update" /> -->
 </properties>
</persistence-unit>
</persistence>
```

src/**mySpringConf.xml**

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

<bean id="myEmf"
       class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
        <property name="dataSource" ref="myDataSource"/>
     <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter" />
   </property>
</bean>
```

2.3. TxManager compatible JPA et @PersistenceContext

```
<bean id="txManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="myEmf"/>
</bean>

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

Cette configuration est indispensable pour que les annotations **@Transactional(readOnly=true)** et **@Transactional(rollbackFor=Exception.class)** qui précédent les méthodes des services métiers soient prises en compte par Spring de façon à générer (via AOP) une enveloppe transactionnelle.

```
<bean id="compteDao" class="dao.jpa.CompteDaoJpa"/>


<!-- Annotation indispensable pour la prise en compte de @PersistenceContext()  par Spring />
<bean class=org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor"  />
```

NB : L'annotation @PersistenceContext() d'origine EJB3 permet d'initialiser automatiquement une instance de "entityManager" en fonction de la configuration JPA (META-INF/persistence.xml + entityManagerFactory, …).

Pour que cette annotation soit prise en compte (interprétée) par Spring , il faut que le bean "***PersistenceAnnotationBeanPostProcessor***" soit présent dans la configuration xml de Spring.

2.4. DAO «JPA» style «pure JPA,Ejb3» pris en charge par Spring

```
package dao.jpa;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query; ...

@Transactional
public class CompteDaoJpa implements CompteDao {
    private EntityManager entityManager;

    @PersistenceContext()
     public void setEntityManager(EntityManager entityManager) {
     this.entityManager = entityManager;
      }

     public List<Compte> getAllComptes() {
     return entityManager.createQuery(
       "Select c from Compte as c",Compte.class)
        .getResultList();
     }
     public Compte getCompteByNum(long num_cpt) {
          return entityManager.find(Compte.class, num_cpt);
     }
     public void updateCompte(Compte cpt) {
          entityManager.merge(cpt);
     }
     public Long createCompte(Compte cpt) {
          entityManager.persist(cpt);
           return cpt.getNumCpt() ; //return auto_incr pk
     }
     public void deleteCompte(long numCpt){
       Compte cpt = entityManager.find(Compte.class, numCpt) ;
       entityManager.remove(cpt);
     }
}
```

VI - Transactions Spring et architecture(s)

1. Support des transactions au niveau de Spring

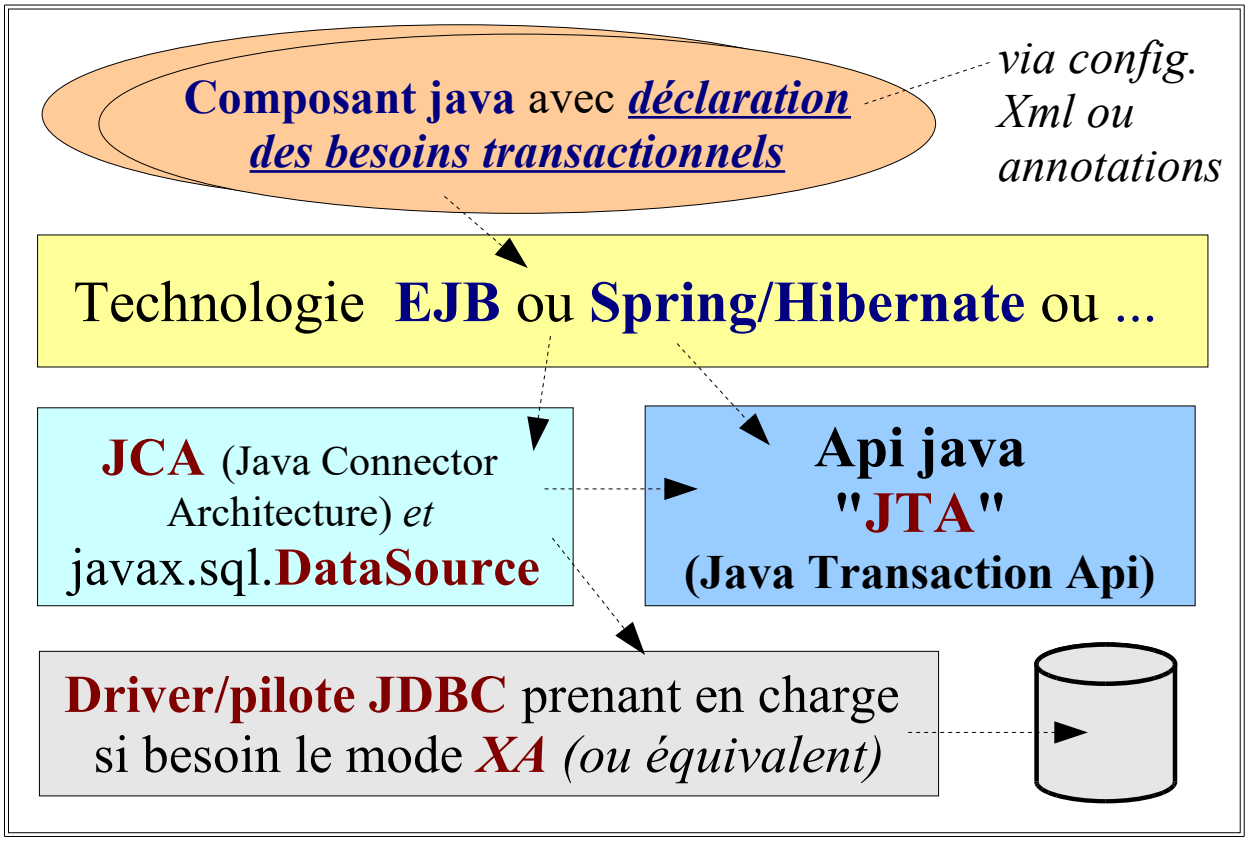
Le framework Spring est capable de gérer (superviser) lui même les transactions devant être menées à bien à partir de certains services applicatifs.

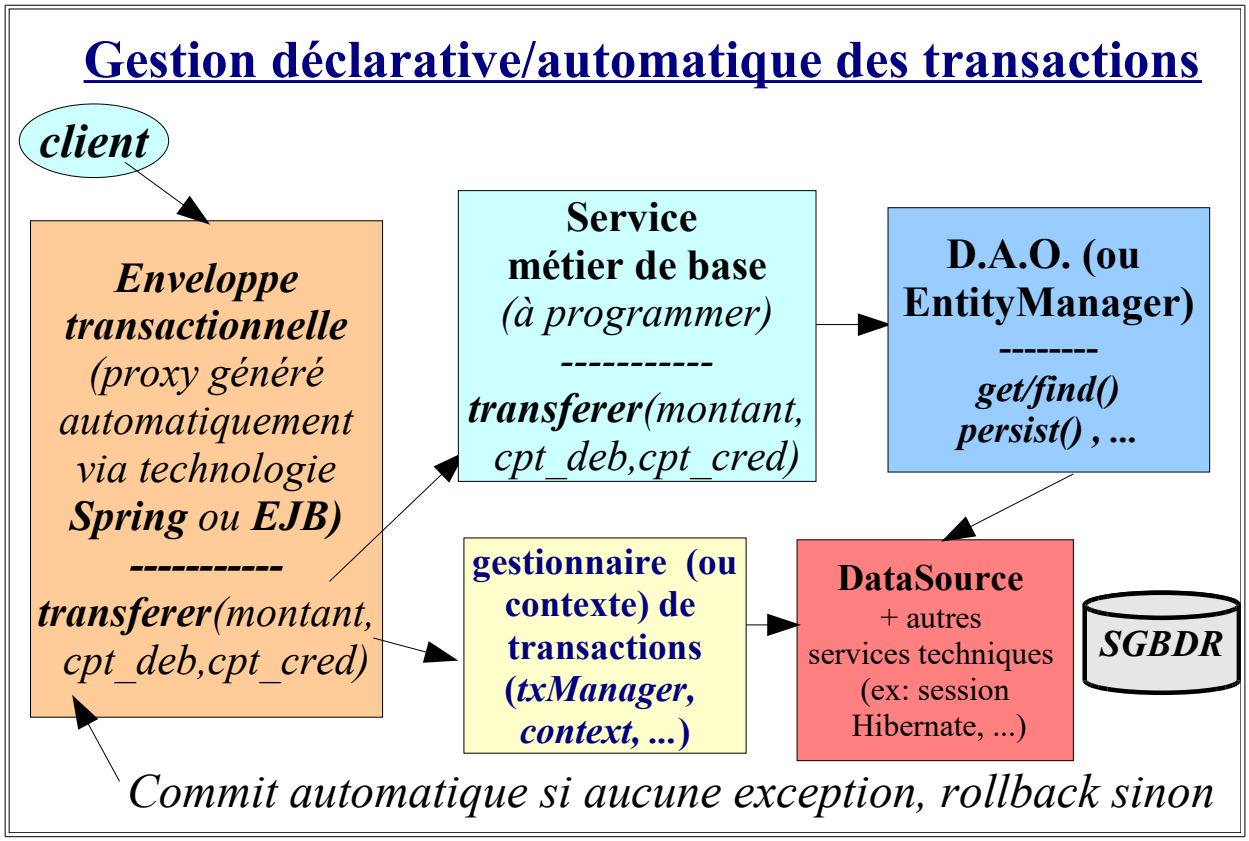
Ceci suppose :

- un paramétrage simple des besoins transactionnels (via xml ou annotations)
- une propagation des ordres transactionnels vers les couches basses (services techniques JDBC , XA , JTA ....).
-

Etant donné la grande étendue des configurations possibles (JTA ?, Hibernate ? , serveur J2EE ? , EJB ? , ...) les mécanismes transactionnels de Spring doivent être relativement flexibles de façon à pouvoir s'adapter à des situations très variables.

Le composant technique "***txManager***" `servira à relayer les ordres de «commit» ou «rollback» vers la source de données (SGBDR).`





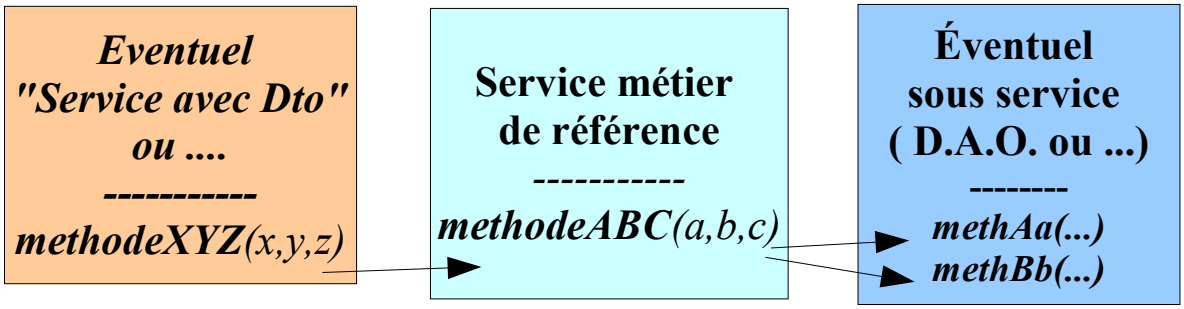
L'enveloppe transactionnelle supervisera automatiquement les "commit" et les "rollback" en fonction d'un paramétrage XML (ou bien en fonction de certaines annotations).

Le code généré dans l'enveloppe transactionnelle est a peu près de cette teneur:

```
public void transferer(double montant, long num_cpt_deb, long num_cpt_cred){
// initialisation (si nécessaire) de la session Hibernate ou de l'entityManager de JPA
// selon existence dans le thread courant
tx = ...beginTransaction(); // sauf si transaction (englobante) déjà en cours
try{
      serviceDeBase.transferer(montant,num_cpt_deb,num_cpt_cred);
      tx.commit(); // ou ... si transaction (englobante) déjà en cours
}
catch(RuntimeException ex){      tx.rollback();   /* ou setRollbackOnly(); */      ... }
catch(Exception e){    e.printStackTrace(); }
finally{    // fermer si nécessaire  ession Hibenate ou EntityManager JPA
              //  (si ouvert en début de cette méthode)
 }
}
```

2. Propagation du contexte transactionnel et effets

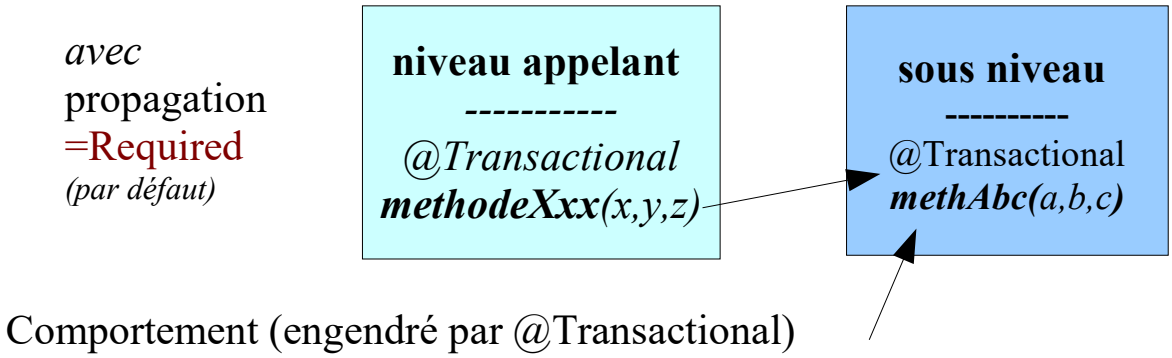
Propagation du contexte transactionnel



Propagation	Tx en cours (appelant)	Tx dans sous service
Required *(par défaut)*	none tx1	new_tx tx1
Support	none tx1	none tx1
Nested	tx1	sub_tx (in tx1)
...		

NB: Le choix de la propagation peut se faire via @Transactional(propagation=....)

Effets de @Transactional (de Spring)



Comportement (engendré par @Transactional)

Au début:
Si aucun "entityManager/..." était ouvert au début j'ai dû en ouvrir un.
Si aucune transaction existait auparavant j'ai alors dû en créer une nouvelle .

A la fin:
Je ferme ou finalise que ce que j'ai moi même ouvert/initialisé (tx et/ou ...)
ou bien sinon: simple tx.setRollbackOnly() en cas d'exception locale.

3. Configuration du gestionnaire de transactions

3.1. Transactions locales directement gérées par JDBC

```
<bean id="myDataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSource">
      <property name="driverClassName" value="com.mysql.jdbc.Driver"  />
      <property name="url" value="jdbc:mysql://localhost/bankdb"       />
      <property name="username" value="mydbuser"      />
      <property name="password" value="mypwd"         />
</bean>
```

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

3.2. Transactions distribuées (JTA) gérées par un serveur J2EE

```
<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>
```

```
<bean id="txManager"
class="org.springframework.transaction.jta.JtaTransactionManager"/>
     <property name="dataSource" ref="myDataSource"/>
</bean>
```

3.3. Transactions gérées par Hibernate

```
...
<bean id="mySessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
     <property name="dataSource" ref="myDataSource" />
     <property name="mappingResources">
          <list>
               <value>Compte.hbm.xml</value>
               <value>Client.hbm.xml</value>
          </list>
     </property>
     <property name="hibernateProperties">
          <props>
               <prop key="hibernate.dialect">
                    org.hibernate.dialect.MySQLDialect</prop>
          </props>
     </property>
</bean>
```

```
<bean id="txManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="mySessionFactory"/>
</bean>
```

3.4. Transactions gérées par JPA

```
<bean id="myEmf"
```

```
  class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
      <property name="persistenceUnitName" value="myPersistenceUnit"/>
</bean>
```

```
<bean id="txManager"
      class="org.springframework.orm.jpa.JpaTransactionManager">
      <property name="entityManagerFactory" ref="myEmf"/>
</bean>
```

4. Mise en oeuvre des transactions avec Spring 2 et 3

4.1. Paramétrage AOP/Xml de l'enveloppe transactionnelle (ajoutée automatiquement)

```
<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"
           xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
           http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-2.0.xsd
           http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-2.0.xsd">

<!--   + config DataSource , SessionFactory , txManager , DAO , Services
métiers et injection directe du service métier dans le reste (l'enveloppe
transactionnelle sera appliquée/insérée directement via auto-proxy aop).  -->

   <tx:advice id="txAdvice" transaction-manager="txManager">
      <tx:attributes>
            <tx:method name="get*" propagation="REQUIRED" read-only="true" />
            <tx:method name="*" propagation="REQUIRED"/>
      </tx:attributes>
   </tx:advice>

      <aop:config>
            <aop:advisor advice-ref="txAdvice"
                  pointcut="execution(* xxx.yyy.service.*.*(..))" />
      </aop:config>
</beans>
```

Remarque: cette manière de configurer les transactions (en XML) est assez complexe (mais peut être générique et appliquer de façon assez universelle/systématique) .

Le paramétrage par annotations (présenté dans le paragraphe suivant) est syntaxiquement plus simple , explicite mieux le code mais doit être répété à tous les endroits utiles.

4.2. Paramétrage via annotations "java 5"

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

à la place de

<tx:advice ...> + <aop:config > `<aop:advisor advice-ref="txAdvice">...`

et utilisation de **`@Transactional`** dans le code d'implémentation.

--> détails dans l'exemple suivant

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

...

public class GestionComptesImpl implements GestionComptes {
...

      @Transactional(readOnly=true)
      public Compte getCompteByNum(long numCpt) throws MyApplicationException {
        ... }

      @Transactional
      public void transferer(long numCompteADebiter, long numCompteACrediter,
                  double montant) throws MyApplicationException {
      ...}
...
}
```

Important:

L'enveloppe transactionnelle générée automatiquement par Spring_AOP ne déclenche par défaut des **rollbacks** que suite à des «unchecked execptions» (exceptions héritant de ***RuntimeException***).

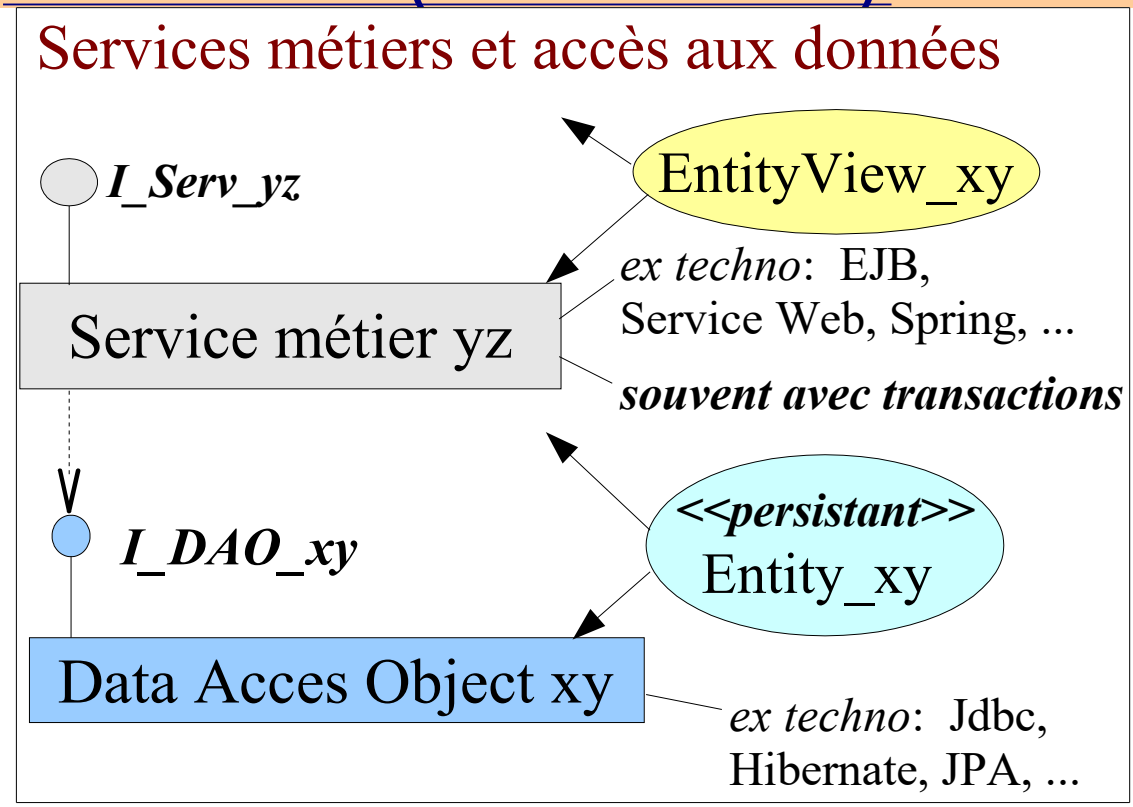
Si l'on souhaite que Spring déclenche des rollback suite à d'autres types d'exceptions, il faut le préciser via le paramètre optionnel **rollbackFor** de l'annotation **@Transactional** (ou de la balise xml <tx:method ..../>).

syntaxe générale: rollbackFor="Exception1,Exception2,Exception3" .

Exemple: **@Transactional(rollbackFor=Exception.class)**

On peut également choisir le mode de **propagation** du contexte transactionnel via l'attribut propagation de l'annotation @Transactional (sachant que la valeur par défaut "**Required**" convient parfaitement dans la majorité des cas) .

5. Vues facultatives (alias DTO alias VO)



Vues "métier" (alias DTO / VO)

Une "***vue métier***" est un ***objet sérialisable*** et qui servira à faire communiquer un service métier avec un client (Web ou ...) potentiellement distant. [*synonymes classiques*: ***V**alue **O**bject ou **D**ata **T**ransfert **O**bject*].
Outre son aspect technique lié aux appels éventuellement distants, une vue métier est utile pour que:

* La couche N (Appli-Web,...) ne voit pas directement les entités persistantes remontées par la couche N-2 (D.A.O.).
* L'application puisse manipuler des vues souvent simplifiées (sans tous les détails des tables relationnelles ou des objets persistants).

Un DTO (Data Transfert Object) [alias VO=Value Object , alias "vue orienté objet"]

est un objet sérialisable qui est une copie (souvent simplifiée) d'une entité persistante.

NB: Les classes des "DTO" ne devraient idéalement pas être reliées aux classes d'entité par un héritage (ni dans un sens , ni dans l'autre) pour garantir une bonne indépendance entre deux formats de données qui seront utilisés dans deux couches logicielles bien distinctes .

Une classe de "DTO" peut être une version "partielle" d'une classe d'entité (ne reprenant que les principales propriétés qui ont besoin d'être affichées ou saisies) ou peut être une structure "ad hoc" résultant d'un assemblage de valeurs puisées dans différentes entités.

Analogie : "Vue relationnelle basée sur une ou plusieurs tables d'une base de données).

structure du code (si avec DTO/VO):

entity.Compte , entity.Operation , ... (les entités persistantes)

dto.CompteVo , dto.OperationVo , ... (les vues / dto / vo)

service.GestionXxx travaillant en interne sur les entités persistantes (entity.Xxxx)

mais retournant des dto.XXXVo à la couche IHM/présentation appelante.

Exemple de code :

```
@Transactional(readOnly=true)
public CompteVo getCompteByNum(long num) {
Compte cpt=null; CompteVo cptVo=new CompteVo();
   try { cpt = cptDao.getCompteByNum(num);
    //avec lazy=true , les dernières opérations ne sont pas remontées
    // car on appelle pas cpt.getDernieresOperations()
  BeanUtils.copyProperties(cpt, cptVo);// version spring (src --> dest)
  } catch (Exception e) {    e.printStackTrace();  }
return cptVo;
}

@Transactional(readOnly=true)
public Collection<OperationVo> getDernieresOperations(long num_cpt){
Collection<OperationVo> listeOpVo = new ArrayList<OperationVo>();
         try {
            Compte  cpt = cptDao.getCompteByNum(num_cpt);
            for(Operation op: cpt.getDernieres_operations()){
                  OperationVo opVo = new OperationVo();
                  BeanUtils.copyProperties(op, opVo);
                  listeOpVo.add(opVo);
                }
        } catch (Exception e) {     e.printStackTrace();  }
return listeOpVo;
}
```

Astuce technique:

Lorsque l'on a besoin de recopier beaucoup de valeurs d'un objet vers un autre (qui ont des propriétés communes) on peut s'appuyer sur la méthode statique utilitaire **BeanUtils.copyProperties(obj1,obj2);** de Spring (ou de jakarta commons).

Ceci est un équivalent plus compact pour:

obj2.setXxx(obj1.getXxx()); obj2.setYyy(obj1.getYyy()); obj2.setZzz(obj1.getZzz());

NB: copyProperties recopie toutes les propriétés qui ont le même nom (en effectuant si besoin des conversions élémentaires String ---> int , ... et en ignorant les propriétés sans correspondance de nom).

Attention: BeanUtils.copyProperties(source,destination) en version Spring et inversement BeanUtils.copyProperties(destination, source) en version "jakarta-commons" .

6. Services avec ou sans DTO

6.1. Avec DTO

Avantages : couches logicielles bien indépendantes , plus de problème "lazyException" , on ne remonte à la couche présentation que ce qui est nécessaire (et dans un format bien ajusté/contrôlé).

Inconvénients: long à programmer (assez lourd) et un peu pénalisant sur les performances (charges mémoire & CPU supplémentaires).

6.2. Sans DTO

Avantages : architecture simple et légère , performances optimisées , rapide à développer.

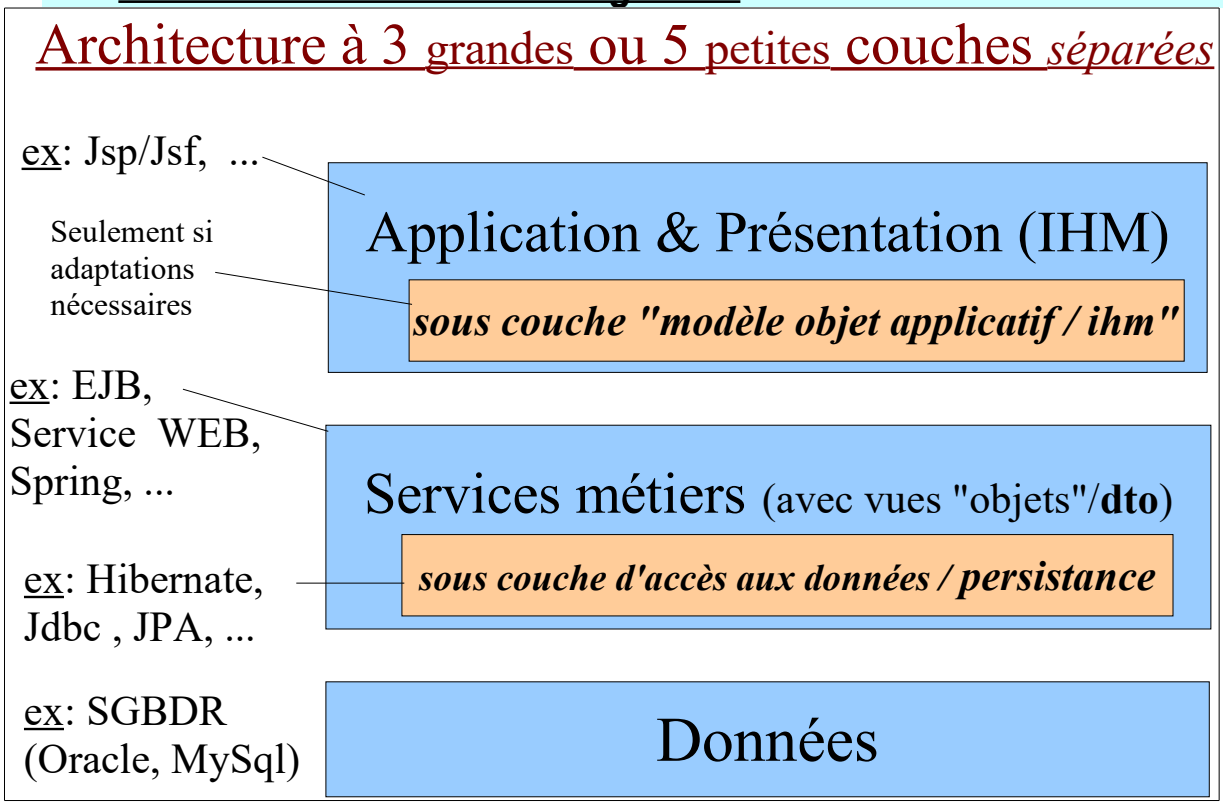
Inconvénients: Filtre "Open....InViewFilter" nécessaire pour ne pas trop avoir de "LazyException" (avec tout de même quelques petits effets de bord) , couches logicielles fortement couplées .

Astuce technique: via @XmlTransient (de JAXB2 utilisé en interne par JAX-WS) et via le filtre "Open....InViewFilter" il est éventuellement possible de remonter directement les entités persistantes via des "Services WEB" (avec CXF ou autre) sans pour autant avoir des boucles dans le suivi des relations bidirectionnelles .

NB: l'architecture sans DTO est quelquefois appelée "DRY" (Don't Repeat Yourself) .

"DRY" n'est cependant qu'un "slogan" auquel on est pas obligé d'adhérer !

6.3. Eléments d'architecture logicielle



Si on tient absolument à bien séparer les couches logicielles , les DTO sont quasi indispensables.

Ceci explique pourquoi environ 80% des projets "java" utilisent des "DTO" en interne.

7. Architecture conseillée (DTO , Dozer , ...)

En pratique, l'architecture "DRY" (sans DTO et avec filtre "Open...InViewFilter" montre rapidement ses limites sur des projets de taille respectable (gros projet , moyen projet ou petit projet

qui évoluera en moyen projet).
Il est en effet très difficile de faire cohabiter au sein d'une même structure de données :

- des annotations paramétrant la persistance (@Table , @Id, @OneToMany, ...)
- des annotations paramétrant la sérialisation XML (@XmlTransient , ...)

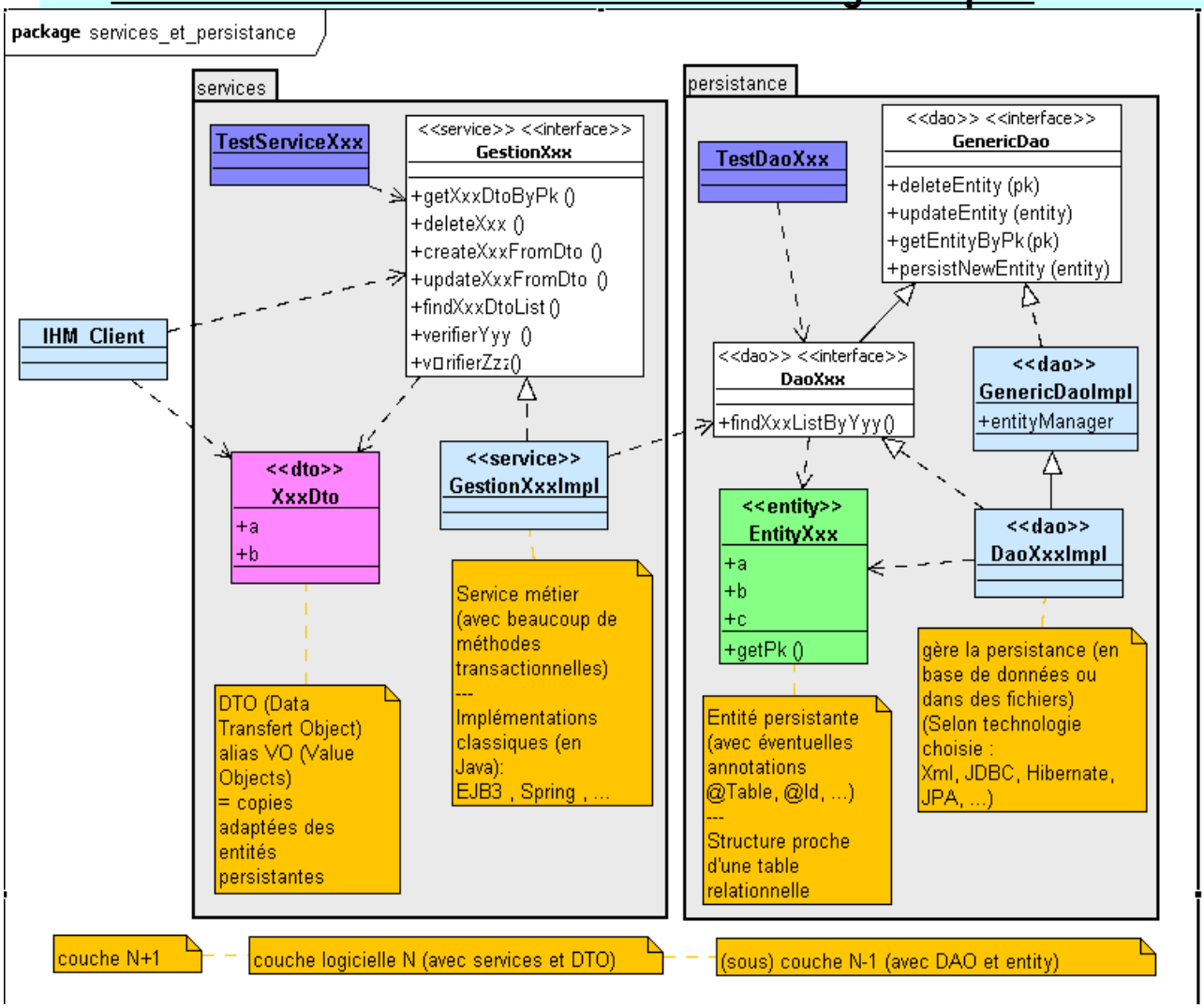
De plus, on a souvent besoin d'un format de données assez spécialisé selon les couches logicielles:

- format s'adaptant à la structure d'une base de données
  (et/ou aux limitations de JPA 1.0 ou 2.0)
- format s'adaptant aux besoins spécifiques de l'application (attributs à montrer ou cacher, DTO comme assemblage d'entités persistantes , ...)
- format s'adaptant aux contraintes des API "Web Services" (JAXWS, ...)

Cependant , pour ne pas introduire trop de lourdeur au niveau du code des services métiers et des DAO, on aura tout intérêt à utiliser quelques unes des astuces suivantes:

- Copie en partie automatisée des "Entity" persistantes en "Dto" et vice-versa .
  (La technologie open source "Dozer" est très bien adaptée pour cela).
- DAO héritant si possible d'un DAO générique (basé sur Hibernate ou JPA)

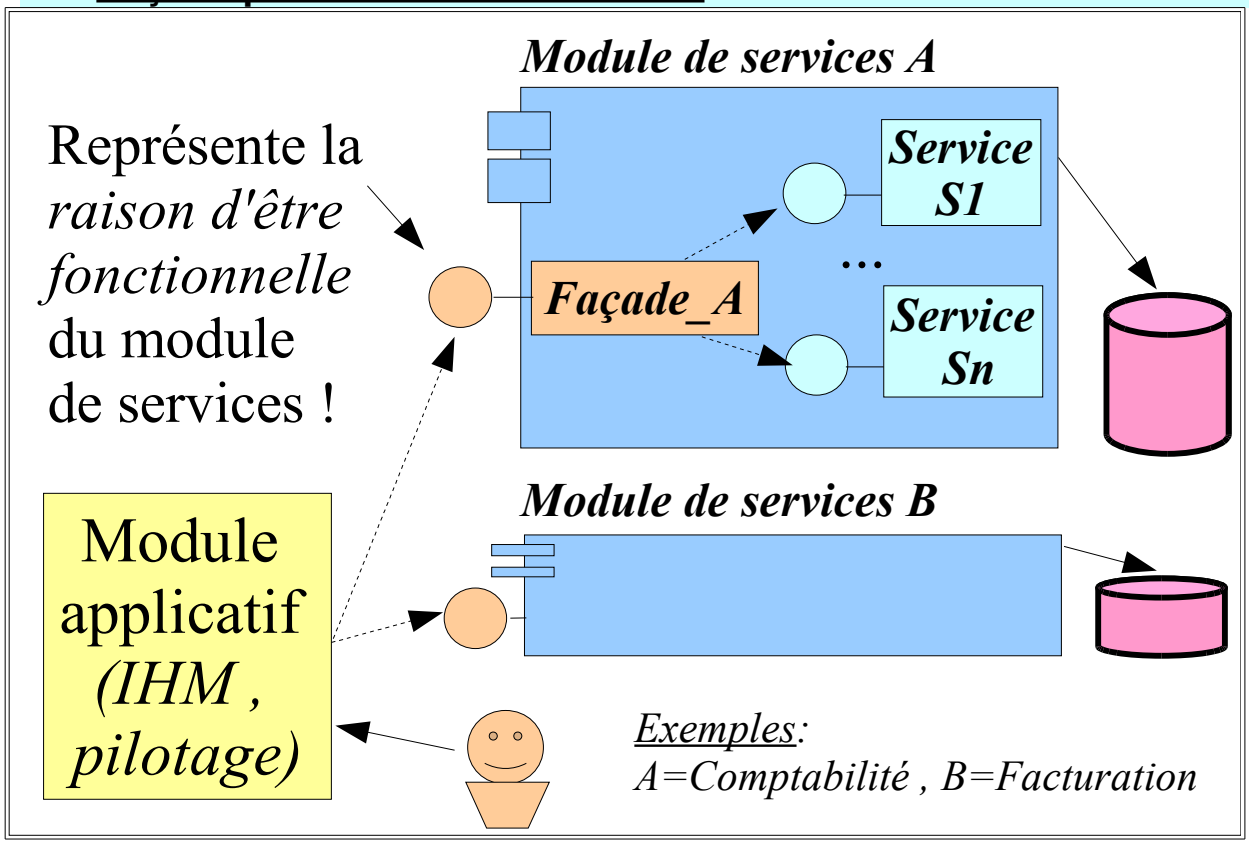
7.1. Architecture "dto-service-dao" avec "dao générique"



VII - Façade , business delegate, ...

1. Façade et "business_delegate"

1.1. Façade pour module de services



D'un point de vue fonctionnel , une façade donne du sens à un module de service. Son nom doit donc être bien choisi (pour être évocateur).

D'un point de vue technique, une façade n'est qu'un nouvel élément intermédiaire de type "Façade d'accueil" qui servira à orienter les clients vers les différents services existants.

Exemple:

Façade_Comptabilité
. getServicePostesComptables()
.getServiceBilan()
.getServiceJournal()
.getServiceGrandLivre()
...
.getServiceN()

utilisation:

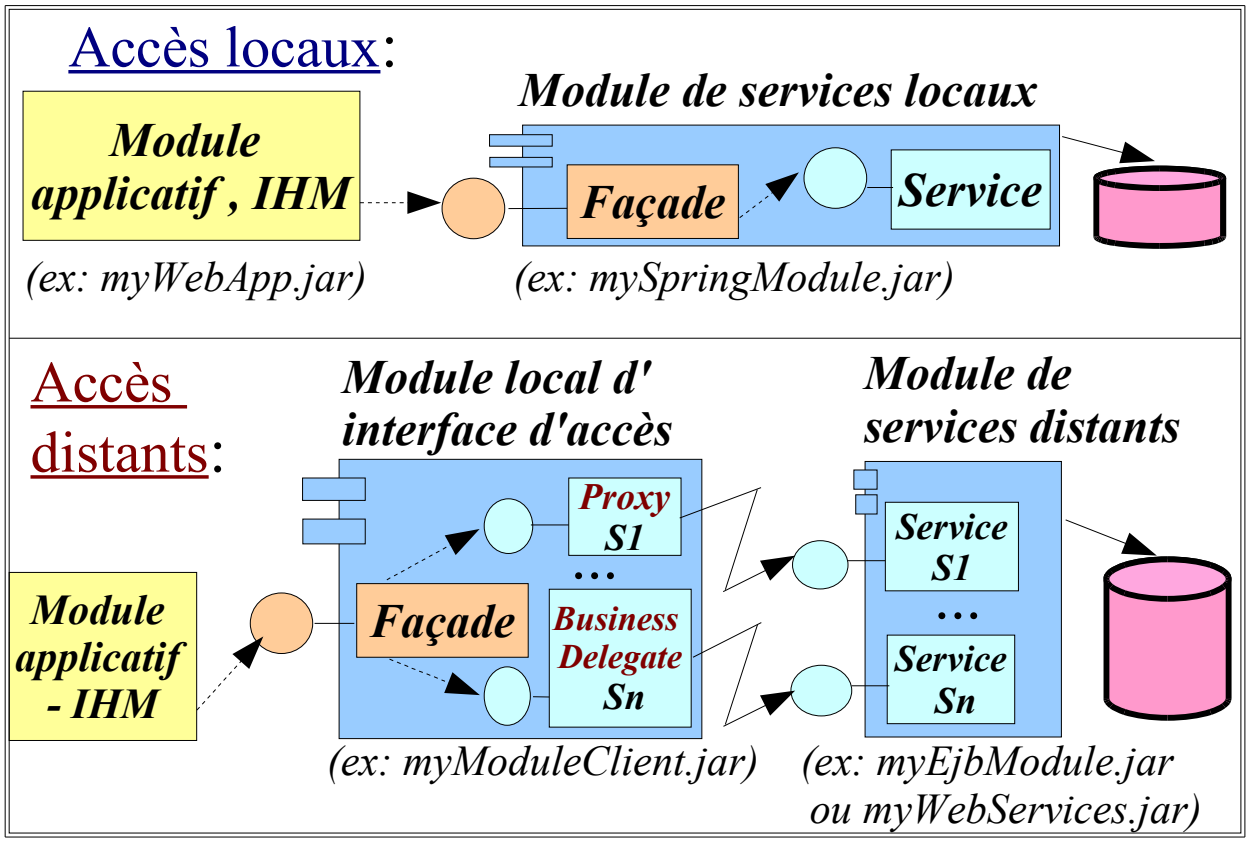
facadeCompta.getServiceBilan().xxx()

facadeCompta.getServiceGrandLivre().yyy();

Une facade simple (sans accès distants) est techniquement un simple niveau intermédiaire avec :

- injection des services (setXxxService())
- méthode d'accès aux services (getServices())
- initialisation via framework IOC (spring ou Jsf ou ....)

1.2. "Business delegate" pour accès distants transparents



Derrière une façade (très souvent locale) on peut éventuellement trouver des services distants.

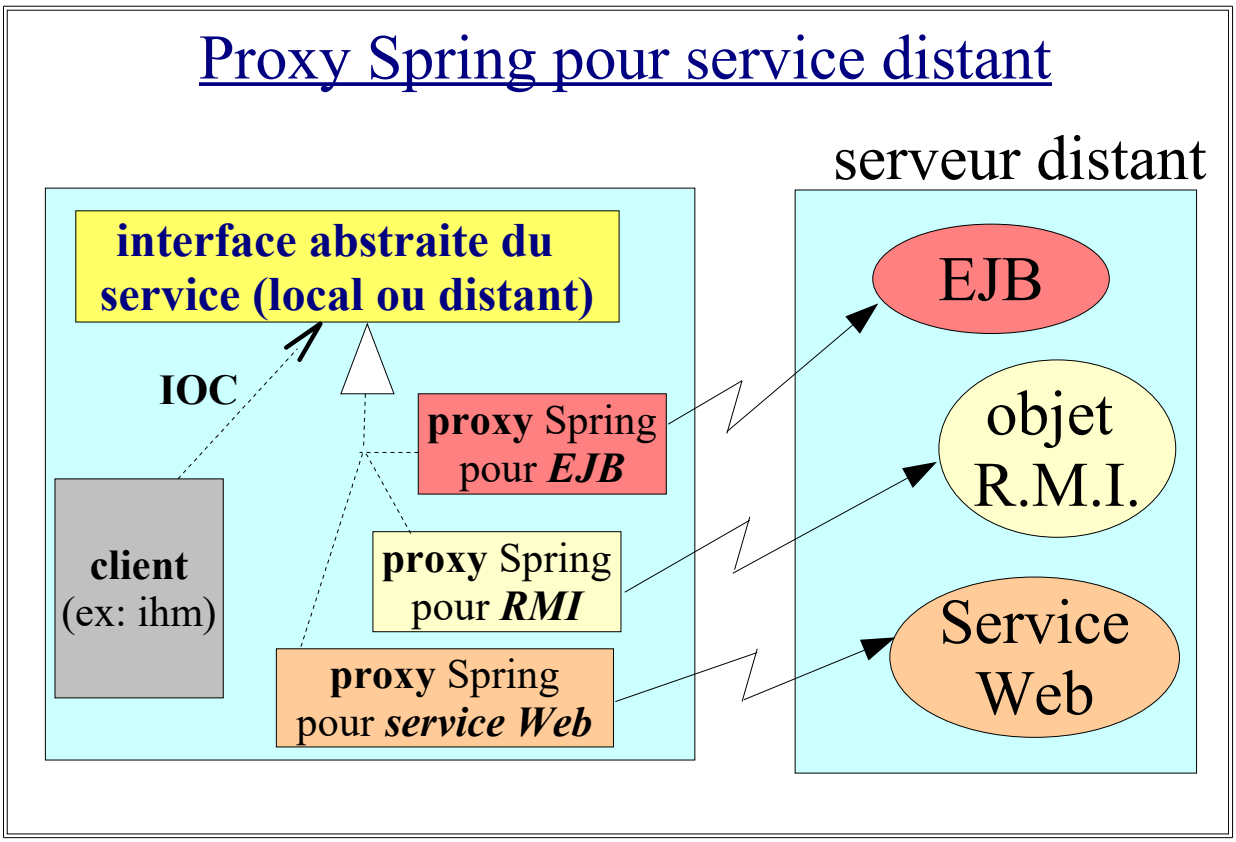
Un "**proxy**" est un représentant local d'un service distant .

Ce terme (plutôt technique) désigne assez souvent du code généré automatiquement (à partir d'un fichier WSDL par exemple).

Pour bien contrôler l'interface locale d'un service distant (de façon à n'introduire aucune dépendance vis à vis d'une technologie particulière), on met parfois en oeuvre des objets de type "**business delegate**" qui cache dans le code privé d'implémentation tous les aspects techniques liés aux communications réseaux:

- localisation du service , connexion (lookup(EJB) ou ...)
- préparations/interprétations des messages/paramètres .
- ...

2. 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).

2.1. 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*

2.2. 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* .

VIII - Injections Spring dans frameworks WEB

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

1.1. WebApplicationContext

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

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

Ceci permet de charger automatiquement en mémoire la configuration "Spring" (ici le fichier "mySpringConf.xml" du répertoire /WEB-INF/classes) 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: "/WEB-INF/classes/spring/*.xml" ou encore
"/WEB-INF/classes/contextSpring.xml,/WEB-INF/classes/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>
```

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

```
servletContext = .... getServletContext(); // vue comme objet application au sein d'une page JSP
WebApplicationContext ctx =
                    WebApplicationContextUtils.getWebApplicationContext(servletContext);
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
```

2. 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>/WEB-INF/classes/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>
       <!-- <variable-resolver>org.springframework.web.jsf.DelegatingVariableResolver
            </variable-resolver> pour jsf 1.1 -->
       <el-resolver>org.springframework.web.jsf.el.SpringBeanFacesELResolver</el-resolver>
        <!-- a partir de jsf 1.2 --> ...
   </application>
```

Ceci permettra d'injecter des "beans Spring" (ex: services métiers) au sein des "back-bean" ou "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>
...
```

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

- par les mécanismes standards de JSF
- par le *DelegatingVariableResolver* de Spring puisant à son tour des "beans" instanciés via une fabrique de Spring selon la configuration de WEB-INF/applicationcontext.xml ou autre (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*).

3. Injection via les nouvelles annotations de JSF2

Depuis la version 2 de JSF , de nouvelles annotations sont apparues pour simplifier la configuration.

Ces annotations (@ManagedBean , @ManagedProperty, ...) sont spécifiques au framework JSF2.

Elles constituent néanmoins une partie des spécifications du nouveau standard officiel "JEE6" .

Si l'on souhaite que la partie Web soit assez indépendante de Spring , il vaut mieux utiliser les annotations JSF2 (@ManagedBean , @ManagedProperty, ...) plutôt que celles de Spring (@Component , @Autowired) car les annotations de JSF2 sont également compatibles avec les EJB3.

NB: Pour l'instant (début 2011) la plupart des implémentations de JSF2 (myFaces 2 , ...) n'interprètent pas encore bien @Inject au sein des "managed bean".

Si les futures versions de JSF2 interprètent mieux @Inject dans l'avenir , on pourra sans hésiter utiliser @Inject à la place de @ManagedProperty parce-que :

- @Inject effectue une auto-injection par type compatible et n'a pas besoin d'être paramétré par le nom logique (id) du composant à injecter.
- @Inject est supposé (à terme) être interprété par tous les frameworks (Spring >=3.0.1 , EJB >= 3.1 , JSF2 ??? , ...)

En attendant , ce type de code est opérationnel (avec JSF2 et Spring 3 ou ...):

```
@ManagedBean("xxxManagedBean")
@SessionScoped()  // ou  @RequestScoped
public class XxxManagedBean {

   @ManagedProperty(value="#{xyzServiceOuFacade}" )   //plus tard @Inject ???
   private IserviceOuFacadeXyz  refXyz;

   // setter  à priori indispensable
   public vois setRefXyz( IserviceOuFacadeXyz refXyz){
         this.refXyz =  refXyz;
   }
    private String xxx; //+get/set
    private Long yyy; //+get/set
   ...
}
```

4. 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 ***strust-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 avancés (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");
                 }
}
```

ANNEXES

IX - Java config et Spring boot

1. Java Config (Spring)

Depuis "Spring 4" , l'extension "java config" est maintenant intégrée dans le cœur du framework et il est maintenant possible de **configurer une application spring par des classes java** spéciales (dites de configuration").
NB : une configuration mixte "xml + java-config" est possible.

1.1. Exemple1: DataSourceConfig :

```
package tp.myapp.minibank.impl.config;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.datasource.DriverManagerDataSource;

@Configuration
public class DataSourceConfig {

	@Bean(name="myDataSource") //by default beanName is same of method name
	public DataSource dataSource() {
		DriverManagerDataSource dataSource = new DriverManagerDataSource();
		dataSource.setDriverClassName("com.mysql.jdbc.Driver");
		dataSource.setUrl("jdbc:mysql://localhost:3306/minibank_db_ex1");
		dataSource.setUsername("root");
		dataSource.setPassword("root");//"root" ou "formation" ou "..."
		return dataSource;
	}
}
```

1.2. Avec placeHolder et fichier ".properties"

src/main/resources/**datasource.properties** (exemple) :

```
jdbc.driver=org.hsqldb.jdbc.JDBCDriver
db.url=jdbc:hsqldb:mem:mymemdb
db.username=SA
db.password=
```

DataSourceConfig.java

```
...
import org.springframework.beans.factory.annotation.Value;
import org.springframework.context.annotation.PropertySource;
import org.springframework.context.support.PropertySourcesPlaceholderConfigurer;

@Configuration

//equivalent de <context:property-placeholder location="classpath:datasource.properties" /> :
@PropertySource("classpath:datasource.properties")
public class DataSourceConfig {

	@Value("${jdbc.driver}")
	private String jdbcDriver;

	@Value("${db.url}")
	private String dbUrl;

	@Value("${db.username}")
	private String dbUsername;

	@Value("${db.password}")
	private String dbPassword;

	@Bean
	public static PropertySourcesPlaceholderConfigurer
				propertySourcesPlaceholderConfigurer(){
		return new PropertySourcesPlaceholderConfigurer();

		//pour pouvoir interpréter ${} in @Value()
	}

	@Bean(name="myDataSource")
	public DataSource dataSource() {
		DriverManagerDataSource dataSource = new DriverManagerDataSource();
		dataSource.setDriverClassName(jdbcDriver);
		dataSource.setUrl(dbUrl);
		dataSource.setUsername(dbUsername);
		dataSource.setPassword(dbPassword);
		return dataSource;
	}
}
```

1.3. Quelques paramétrages (avancés) possibles :

@Bean(**initMethodName**="init") , @Bean(**destroyMethodName**="cleanup")

@Bean(**scope**=DefaultScopes.**PROTOTYPE**) , @Bean(**scope** = DefaultScopes.**SESSION**)

1.4. Exemple2: *DomainAndPersistenceConfig*:

```
package tp.myapp.minibank.impl.config;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.orm.jpa.JpaTransactionManager;
import org.springframework.orm.jpa.JpaVendorAdapter;
import org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean;
import org.springframework.orm.jpa.support.PersistenceAnnotationBeanPostProcessor;
import org.springframework.orm.jpa.vendor.Database;
import org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@Configuration
@EnableTransactionManagement() //"transactionManager" (not "txManager") is expected !!!
@ComponentScan(basePackages={"tp.myapp.minibank.impl","org.mycontrib.generic"})
  // for interpretation of  @Autowired, @Inject , @Component , ...
public class DomainAndPersistenceConfig {

	// JpaVendorAdapter (Hibernate ou OpenJPA ou ...)
	@Bean
	public JpaVendorAdapter jpaVendorAdapter() {
		HibernateJpaVendorAdapter hibernateJpaVendorAdapter
                                                = new HibernateJpaVendorAdapter();
		hibernateJpaVendorAdapter.setShowSql(false);
		hibernateJpaVendorAdapter.setGenerateDdl(false);
		hibernateJpaVendorAdapter.setDatabase(Database.MYSQL);
		//hibernateJpaVendorAdapter.setDatabase(Database.HSQL);
		return hibernateJpaVendorAdapter;
	}


	// EntityManagerFactory
	@Bean(name="myEmf")
	public EntityManagerFactory entityManagerFactory(
                JpaVendorAdapter jpaVendorAdapter, DataSource dataSource) {
		LocalContainerEntityManagerFactoryBean factory
                        = new LocalContainerEntityManagerFactoryBean();
		factory.setJpaVendorAdapter(jpaVendorAdapter);
		factory.setPackagesToScan("tp.myapp.minibank.impl.persistence.entity");
```

```
                factory.setDataSource(dataSource);
                factory.afterPropertiesSet();
                return factory.getObject();
        }


        // pour activer la prise en charge de @PersistentContext dans le code
        @Bean
        public PersistenceAnnotationBeanPostProcessor enablePersistentContextAnnotation(){
                return new PersistenceAnnotationBeanPostProcessor();
        }

        // Transaction Manager for JPA or ...
        @Bean(name="transactionManager") //("transactionManager" but not "txManager")
        public PlatformTransactionManager transactionManager(
                        EntityManagerFactory entityManagerFactory) {
                JpaTransactionManager txManager = new JpaTransactionManager();
                txManager.setEntityManagerFactory(entityManagerFactory);
                return txManager;
        }
}
```

1.5. @Import

```
@Configuration
@Import(DomainAndPersistenceConfig.class)
//@ImportResource("classpath:/xy.xml")
@ComponentScan(basePackages={"tp.app.zz.web"})
@EnableWebMvc //un peu comme <mvc:annotation-driven  />
public class WebMvcConfig {

  @Bean
  public ViewResolver mcvViewResolver(){
        InternalResourceViewResolver viewResolver =new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/view/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
        }
}
```

Ou bien ***ApplicationConfig*** incluant (via 2 **@Import**)

DomainAndPersistenceConfig.class

et *WebMvcConfig.class* .

1.6. Différentes utilisations (classiques):

Dans main() :

```
ApplicationContext context =
new AnnotationConfigApplicationContext(DataSourceConfig.class,
                                       DomainAndPersistenceConfig .class);
Service service = context.getBean(Service.class);
```

Dans springContext.xml :

```
<context:annotation-config /> <!-- pour interprétation de @Configuration , @Bean -->
<bean class="tp.myapp.minibank.impl.config.DomainAndPersistenceConfig"/>
```

Dans spring test:

```
@RunWith(SpringJUnit4ClassRunner.class)
//@ContextConfiguration(locations="/springContextOfModule.xml") // xml config
@ContextConfiguration(classes={tp.myapp.minibank.impl.config.DataSourceConfig.class,
      tp.myapp.minibank.impl.config.DomainAndPersistenceConfig.class}) //java config
```

2. Spring-boot

L'extension "**spring-boot**" permet (entre autre) de :

- **démarrer une application java/web depuis un simple "main()"** (sans avoir besoin d'effectuer un déploiement au sein d'un serveur de type de tomcat)
- simplifier la déclaration de certaines dépendances ("maven") via des héritages de configuration type (bonnes combinaisons de versions)
- (éventuellement) ***auto-configurer une partie de l'application selon les librairies trouvées dans le classpath*** .
- **Spring-boot** est assez souvent utilisé en coordination avec **Spring-MVC** (bien que ce ne soit pas obligatoire).

Quelques avantages d'une configuration "spring-boot" :

- **tests d'intégrations facilités** dès la phase de développement (l'application démarre toute seule depuis un main() ou un test JUnit sans serveur et l'on peut alors simplement tester le comportement web de l'application via selenium ou un équivalent).
- **déploiements simplifiés** (plus absolument besoin de préparer un serveur d'application JEE , de le paramétrer pour ensuite déployer l'application dedans).
- **Possibilité de générer un fichier ".war"** si l'on souhaite déployer l'application de façon standard dans un véritable serveur d'applications .
- **Configuration et démarrage très simples** (pas plus compliqué que node-js si l'on connaît bien java) .
- **Application java pouvant** (dans des cas simples) **être totalement autonome** si l'on s'appuie sur une base de données "embedded" (de type "H2" ou bien "HSQLDB").

Quelques traits particuliers (souvent perçus de façons subjectives) :

- Spring-boot (et Spring-mvc) sont des technologies propriétaires "Spring" qui s'écartent volontairement du standard officiel "JEE 6/7" pour se démarquer de la technologie concurrente EJB/CDI .
- Un web-service REST "java" codé avec Spring-boot + Spring-mvc comporte ainsi des annotations assez éloignées de la technologie concurrente CDI/Jax-RS bien qu'au final, les fonctionnalités apportées soient très semblables.

2.1. Configuration "maven" pour spring-boot

Le lien de parenté suivant

```
<parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>1.2.5.RELEASE</version>
</parent>
```

est à placer dans le haut du pom.xml et permet (entre autres) de récupérer (par héritage) une configuration en grande partie pré-définie (avec des valeurs par défaut que l'on peut ré-définir).

En interne **spring-boot-starter-parent** hérite lui-même de **spring-boot-dependencies** qui sert à définir tout un tas de versions de technologies compatibles entres elles .

On hérite ainsi d'un tas de propriétés de ce type :

```
<commons-beanutils.version>1.9.1</commons-beanutils.version>
<commons-collections.version>3.2.1</commons-collections.version>
<hibernate.version>4.3.10.Final</hibernate.version>
….
<dependencyManagement>
        <dependencies>
                <dependency>
                        <groupId>commons-beanutils</groupId>
                        <artifactId>commons-beanutils</artifactId>
                        <version>${commons-beanutils.version}</version>
```

Ce qui permet d'exprimer des dépendances sans avoir à absolument préciser les versions dans le pom.xml de notre projet :

```
<dependencies>
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <!--
        <dependency>
```

```
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-security</artifactId>
        </dependency>
         -->
        <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-tomcat</artifactId>
                <scope>provided</scope>
        </dependency>
```

Attention : si "spring-security" (ou spring-boot-starter-security) est présent dans le classpath , l'application sera automatiquement sécurisée en mode @EnableAutoConfiguration !!!!

2.2. Boot (standalone)

```
ConfigurableApplicationContext context =
                                SpringApplication.run(MyApplicationConfig.class);
ServiceXy serviceXy = context.getBean(ServiceXy.class);
….
context.close() ;
```

ou bien (en plusieurs phases mieux contrôlées) :

```
SpringApplication app = new SpringApplication(DomainAndPersistenceConfig.class);
app.setLogStartupInfo(false);
ConfigurableApplicationContext context = app.run(args);
```

2.3. Tests unitaires avec Spring-boot

```
import org.springframework.boot.test.SpringApplicationConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@SpringApplicationConfiguration(classes = MyApplicationConfig.class)
//au lieu du classique @ContextConfiguration(....)
public class MyApplicationTest {
…
```

```
}
```

2.4. Eventuelle auto-configuration (facultative)

L' annotation @EnableAutoConfiguration *(à placer à coté du classique @Configuration de java-config)* demande à **Spring Boot** via la classe ***SpringApplication*** de **configurer l'application en fonction des bibliothèques trouvées dans son class-path**.

Par exemple:

- Parce que les bibliothèques Hibernate sont dans le Classpath, le bean *EntityManagerFactory* de JPA sera implémenté avec Hibernate.
- Parce que la bibliothèque du SGBD H2 est dans le Classpath, le bean "dataSource" sera implémenté avec H2 (avec administrateur par défaut "sa" et sans mot de passe) .

  Le "dialecte" hibernate sera également auto-configuré pour "H2" .

  Cette auto-configuration ne fonctionne qu'avec des bases "embedded" (H2 , hsqldb, …)
- Parce que la bibliothèque [spring-tx] est dans le Classpath, c'est le gestionnaire de transactions de Spring qui sera utilisé.
- Parce que une bibliothèque "spring...security" sera trouvée dans le classpath , l'application java/web sera automatiquement sécurisée (en mode basic-http) avec un username "…" et un mot de passe qui s'affichera au démarrage de l'application dans la console .
- …

Exemple (*DomainAndPersistenceAutoConfig.java*) :

```
package tp.app.zz.config.auto;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.boot.orm.jpa.EntityScan;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;
import org.springframework.transaction.annotation.EnableTransactionManagement;

/*  different classic uses:

  in main() of XyBoot :
        SpringApplication app = new SpringApplication(DomainAndPersistenceAutoConfig.class);
        app.setWebEnvironment(false);
        ConfigurableApplicationContext context = app.run(args);

  in spring test:
        @RunWith(SpringJUnit4ClassRunner.class)
        @SpringApplicationConfiguration(classes={
                              tp.app.zz.config.auto.DomainAndPersistenceAutoConfig.class})
  ...
 */

@Configuration
@EnableAutoConfiguration  //auto configuration en tenant compte des librairies du "classpath"
```

```
    //pour découvrir et configurer automatiquement datasource en version H2 ou hsqldb ,
    //jpaVendor en version Hibernate ou ...
@EnableTransactionManagement() //"transactionManager" (not "txManager") is expected !!!
@ComponentScan(basePackages={"tp.app.zz.impl","org.mycontrib.generic"})
                //to find and interpret @Component , @Named, ...
@EntityScan(basePackages={"tp.app.zz.impl.persistence.entity"})
                //to find and interpret @Entity, ...
public class DomainAndPersistenceAutoConfig {

   /* Via @EnableAutoConfiguration, les éléments suivants seront automatiquement configurés:
            - JpaVendorAdapter (par exemple en version HibernateJpaVendorAdapter)
            -  EntityManagerFactory ou ....
            -  PlatformTransactionManager (par exemple en version JPA)   */
}
```

NB : par défaut, **spring-boot** utilise **"slf4j"+"logback"** pour générer des lignes de log.

Il vaut mieux donc configurer les logs de l'application avec **logback.xml** plutôt qu'avec *log4j.properties* .

logback.xml (exemple) :

```
<!-- this is a configuration file for LogBack log Api (under SLF4J)
    LogBack is faster than log4J and used by default in Spring-boot -->
<configuration>

 <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
  <encoder>
   <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logger{36} - %msg%n</pattern>
  </encoder>
 </appender>

 <root level="info"> <!-- "debug" , "info" , "warn" , "error" , ... -->
  <appender-ref ref="STDOUT" />
 </root>
</configuration>
```

avec dans **pom.xml**

```
        <!-- logback-classic is now better than log4J .
          it's the defaut (native)  SLF4J implementation
          (and used by default with spring-boot)

 Configuration file is src/main/resources/logback.xml (used if logback-test.xml not found)
                and src/test/resources/logback-test.xml (not stored in built artifact)
        -->

            <dependency>
                <groupId>ch.qos.logback</groupId>
```

```
                    <artifactId>logback-classic</artifactId>
                </dependency>
            <!-- plus dépendance slf4j-api directe ou indirecte -->
```

2.5. Paramétrages du conteneur web "EmdeddedTomcat"

NB: le **paramétrage explicite** de "***EmbeddedServletContainerFactory***" **n'est nécessaire qu'en l'absence de @EnableAutoConfiguration** .

```
@Configuration
@ComponentScan(basePackages={"tp.app.zz.web"})
@Import({DomainAndPersistenceConfig.class,XyConfig.class})
public class WebAppConfig {

@Bean
public EmbeddedServletContainerFactory servletContainer() {
TomcatEmbeddedServletContainerFactory factory = new
                                TomcatEmbeddedServletContainerFactory();
factory.setPort(8080);
factory.setSessionTimeout(5, TimeUnit.MINUTES);
//equivalent of server.context-path=/deviseSpringBootWeb in application.properties :
factory.setContextPath("/deviseSpringBootWeb");

//factory.addErrorPages(new ErrorPage(HttpStatus.404, "/notfound.html");
TomcatContextCustomizer contextCustomizer = new TomcatContextCustomizer() {
            @Override
            public void customize(org.apache.catalina.Context context) {
               context.addWelcomeFile("/index.html");
            }
        };
factory.addContextCustomizers(contextCustomizer);
return factory;
}

}
```

2.6. Boot "web" en mode @EnableAutoConfiguration

src/main/resources/**application.properties**

```
# this file (application.properties) is used by Spring-boot (en mode @EnabledAutoConfiguration)

# server.context-path is equivalent of "root-context" of web app (same as project name)
server.context-path=/deviseSpringBootWeb
```

2.7. Paramétrages d'une application "spring-boot" + "spring-mvc"

Exemple (***CtrlSimpleConfig.java***) :

```
package tp.app.zz.web.mvc.simple.boot;

import org.springframework.boot.autoconfigure.EnableAutoConfiguration;
import org.springframework.context.annotation.ComponentScan;
import org.springframework.context.annotation.Configuration;

@Configuration
@EnableAutoConfiguration
@ComponentScan(basePackages={"tp.app.zz.web.mvc"}) //to find and interpret @Controller
public  class CtrlSimpleConfig{
}
```

CtrlSimpleBoot.java

```
package tp.app.zz.web.mvc.simple.boot;

import org.springframework.boot.SpringApplication;

public class CtrlSimpleBoot {

	public static void main(String[] args) {

		SpringApplication.run(CtrlSimpleConfig.class, args);

	}

}
```

avec éventuel mixage de ces 2 classes en une seule.

MySimpleCtrl.java

```
package tp.app.zz.web.mvc;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller //but not "@Component" for spring web controller
@RequestMapping("/simple")
public class DeviseListCtrl {
    //complete path/url is "http://localhost:8080"
    //      + "/deviseSpringBootWeb" (value of server.context-path in application.properties)
    //      + "/simple" + "/hello"
          @RequestMapping("/hello")
            @ResponseBody
            String say_hello() {
               return "Hello World!";
            }
}
```

2.8. Spring-boot + JSF

```
@Configuration
@Import({DomainAndPersistenceConfig.class})
@ComponentScan(basePackages={"tp.app.zz.web.mbean"})
//@EnableAutoConfiguration
public class JsfConfig extends SpringBootServletInitializer {

@Bean
   public ServletRegistrationBean servletRegistrationBean() {
      FacesServlet facesServlet = new FacesServlet();
      ServletRegistrationBean facesServletRegistrationBean
         = new ServletRegistrationBean(facesServlet, "*.jsf");
      //l'enregistrement du FacesServlet de jsf est nécessaire pour le bon fonctionnement
      // de Spring-boot .
      //bizarrement , le fichier WEB-INF/web.xml doit obligatoirement être également présent
      //(avec la declaration du FacesServlet et son url-mapping)
   return facesServletRegistrationBean;
   }

// partie indispensable que si l'annotation @EnableAutoConfiguration n'est pas présente :
        @Bean
        public EmbeddedServletContainerFactory servletContainer() {
           TomcatEmbeddedServletContainerFactory factory =
                                        new TomcatEmbeddedServletContainerFactory();
           factory.setPort(8080);
           factory.setContextPath("/deviseSpringBootWeb");
           factory.setSessionTimeout(5, TimeUnit.MINUTES);

           TomcatContextCustomizer contextCustomizer = new TomcatContextCustomizer() {
              @Override
              public void customize(org.apache.catalina.Context context) {
                 context.addWelcomeFile("/index.html");
              }
           };
           factory.addContextCustomizers(contextCustomizer);

           return factory;
        }


        @Override //de SpringBootServletInitializer
        //utile que si fonctionnement dans .war deploye dans servApp
        protected SpringApplicationBuilder configure( SpringApplicationBuilder application) {
                        return application.sources(JsfConfig.class);
         }

/* la dépendance suivante est nécessaire pour JSF pour éviter une erreur de type missing
factory/ServletContextListener (aussi bien avec MyFaces que com.sun.faces):
            <dependency>
                <groupId>org.apache.tomcat.embed</groupId>
                <artifactId>tomcat-embed-jasper</artifactId>
        </dependency>
*/
```

```
}
```

X - Extensions diverses pour Spring

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 plugins 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 ***et autres "spring-remote"***	Implémentation ou appels d'objets distants (via RMI). Quelques éléments pour invoquer des EJB ou des services soap
Spring JMS ***(très utile)***	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)***	Api pour simplifier le paramétrage de la sécurité jee (au niveau des servlet/jsp par exemples) ---> simplification de la syntaxe (en comparant au standard "security-constraint" de WEB-INF/web.xml). ---> 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, ….) → intégration simple de la sécurité à tous les niveaux de l'application "spring" (pages web , services métiers , services web , ...)
Spring Intégration *(un peu comme MuleESB)*	Complément pour JMS ou … , permet d'implémenter les fonctionnalités classiques d'un mini-ESB en s'appuyant sur spring . (routage conditionné , médiation de services , quelques patterns EIP , transformation de formats et/ou protocoles , ...)
Spring WebFlow	Extension spring permettant de mieux contrôler la navigation entre les pages (jsp ou xhtml) d'une application java/web/spring . L'extension "SpringWebFlow" peut être (entre autres) couplée avec le framework "JSF2" . 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").
...	

XI - Spring MVC avec ou sans web-services REST

1. Présentation du framework "Spring MVC"

"Spring Web MVC" est une partie optionnelle du framework spring servant à gérer la logique du design pattern "MVC" dans le cadre d'une intégration "spring" .

"Spring MVC" est à voir comme un petit framework java/web (pour le coté serveur) qui peut être soit vu comme une alternative à Struts2 ou JSF2 soit être vu comme un petit framework web complémentaire à Struts2 ou JSF2.

Dépendances maven nécessaires :

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

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

Configuration :

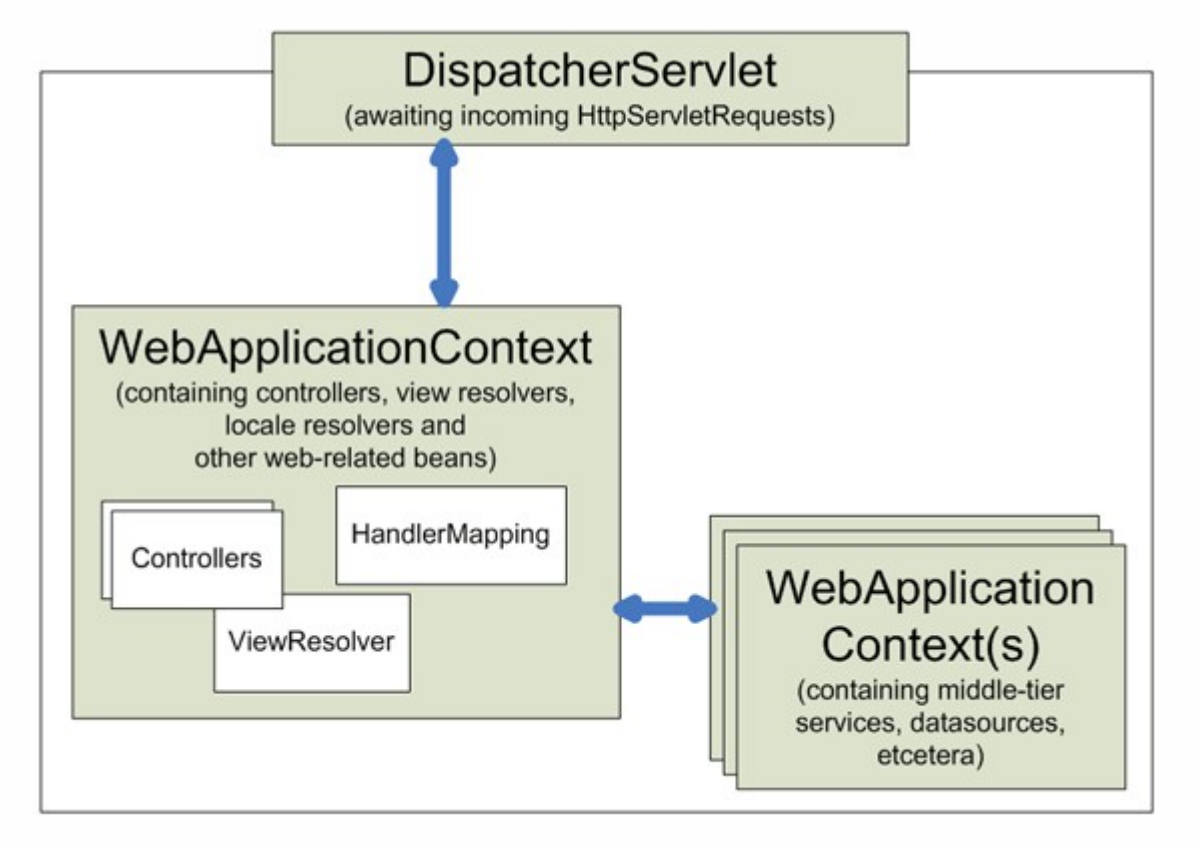
WEB-INF/**web.xml**

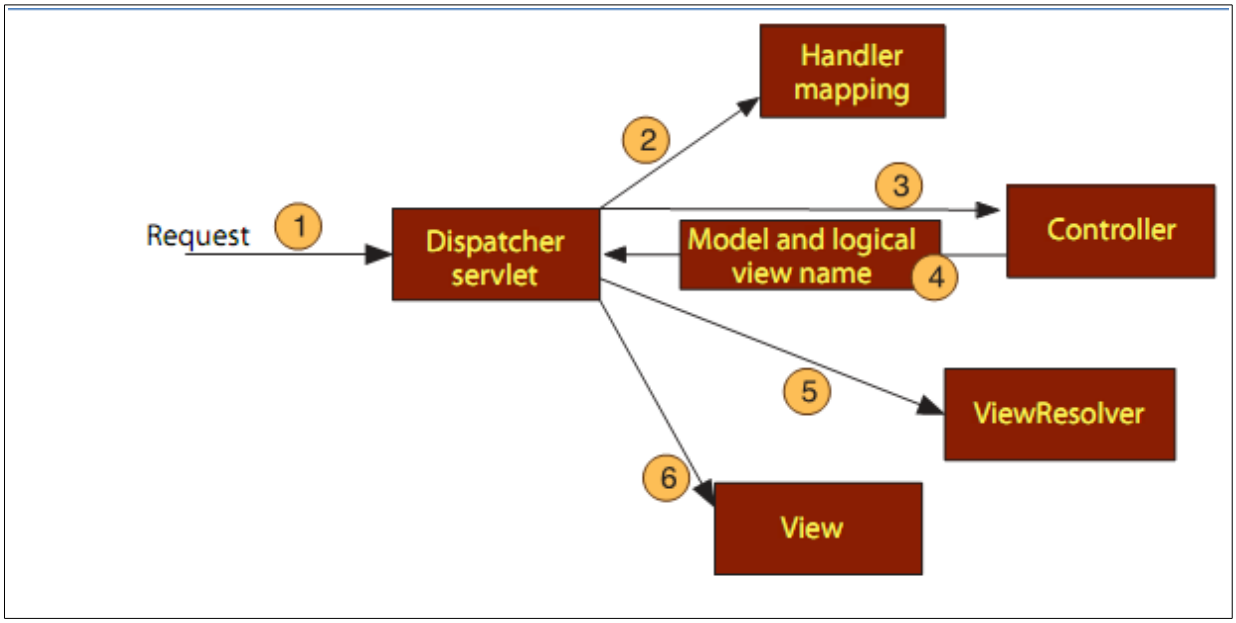
```
...
    <servlet>
        <servlet-name>mvc-dispatcher</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
      <init-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/mvc-config.xml</param-value>
      </init-param>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>mvc-dispatcher</servlet-name>
        <url-pattern>/mvc/*</url-pattern>
    </servlet-mapping>

    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/classes/spring-mvc.xml,...</param-value>
    </context-param>

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

WEB-INF/**mvc-config.xml** (spring mvc)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
      <context:component-scan     base-package="tp.web.mvc.controller"/>
      <mvc:annotation-driven />

       <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
            <!-- Example: a logical view name of 'showMessage' is mapped to
                          '/WEB-INF/view/showMessage.jsp' -->
            <property name="prefix" value="/WEB-INF/view/"/>
            <property name="suffix" value=".jsp"/>
       </bean>
</beans>
```

Fonctionnement





les étapes fondamentales sont :
1.Le DispatcherServlet reçoit une requête dont l'URI-pattern est '/xy.htm'
2.Le DispatcherServlet consulte son ***Handler Mapping*** (Ex : *BeanNameUrlHandlerMapping*) ***pour connaître le contrôleur dont le nom de bean est '/xy.htm'*** .
3.Le DispatcherServlet dispatche la requête au contrôleur identifié (Ex : XyPageController)
4.**Le contrôleur retourne au DispatcherServlet un objet de type ModelAndView possédant comme paramètre au minimum le nom logique de la vue à renvoyer (ou bien un objet Model plus le nom logique de la vue sous forme de String depuis la version 3)**
5.Le DispatcherServlet consulte son ***View Resolver*** lui permettant de trouver la vue dont le nom logique est 'xy' ou 'zzz'. Ici le type de View Resolver choisit est *InternalResourceViewResolver*.
6.Le DispatcherServlet "forward" ensuite la requête à la **vue associé (page jsp ou ...)**

Exemple simple de contrôleur :

```
package tp.web.mvc.controller;
...
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

@Controller
public class HelloWorldController extends AbstractController{

        @Override
        @RequestMapping("/helloWorld")
        protected ModelAndView handleRequestInternal(HttpServletRequest request,
                        HttpServletResponse response) throws Exception {
                return new ModelAndView("showMessage","message","helloWorld");
                                        //viewName      , nameData  , valueData
```

```
        }
}
```

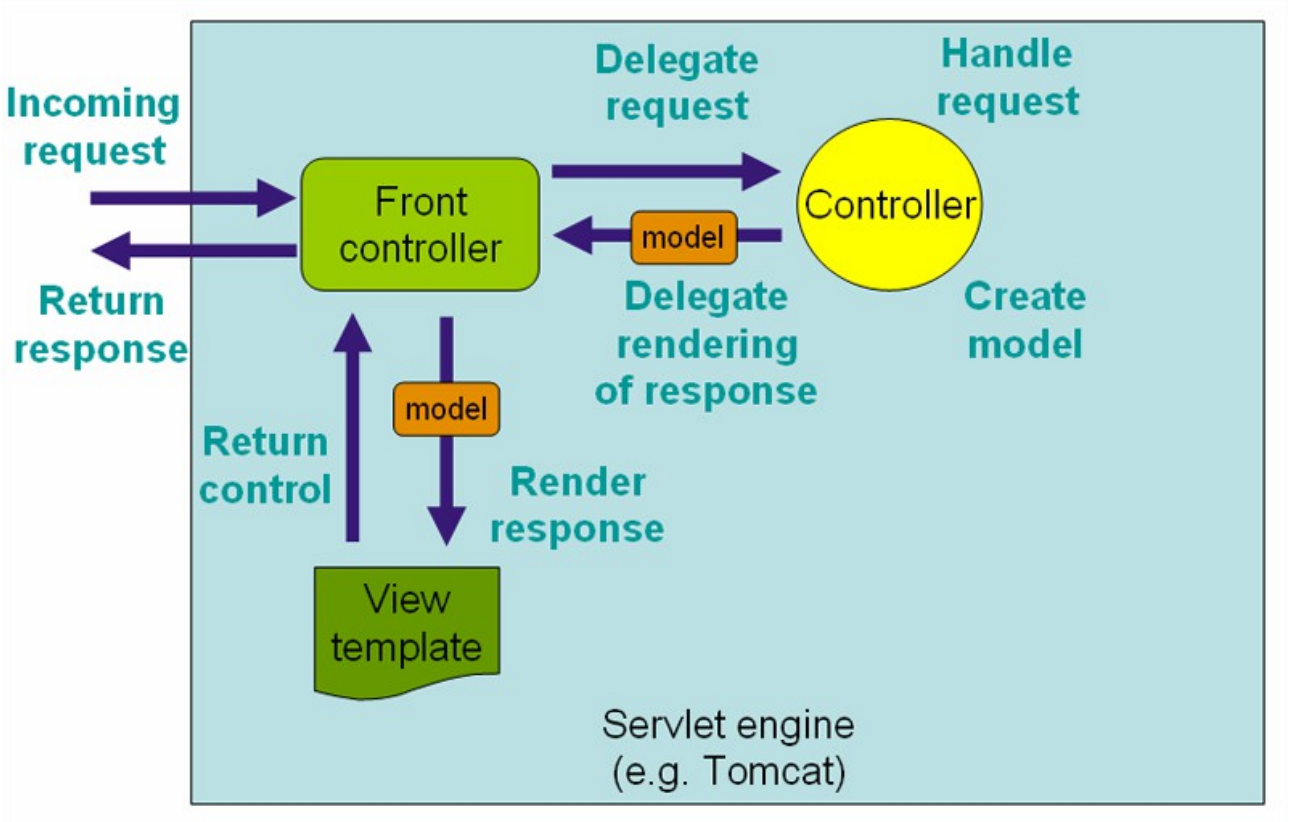
et éventuellement avec cette configuration xml (si pas d'annotation "@Controller" ni "@RequestMapping") :

```
<bean  class="org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping"/>

  <bean name="/helloWorld"    class="tp.web.mvc.controller.HelloWorldController" />
  <bean name="/url2"    class="tp.web.mvc.controller.ForUrl2Controller" />
```

ou bien (plus simplement sans héritage depuis la v3) :

```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HelloWorldController {

   @RequestMapping("/helloWorld")
   public String helloWorld(Model model) {
      model.addAttribute("message", "Hello World!");
      return "showMessage";
   }

}
```

Au niveau de showMessage.jsp, l'affichage de message pourra être effectué via ${message}.



2. éléments essentiels de Spring web MVC

2.1. éventuelle génération directe de la réponse HTTP

```
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.ResponseBody;

@Controller //but not "@Component" for spring web controller
@RequestMapping("/app")
public class WelcomeCtrl {

        @RequestMapping("/hello")
          @ResponseBody //si @ResponseBody , génération directe de la réponse ,
                        // sinon viewResolver (mvc-config.xml)
          String say_hello() {
             return "Hello World!";
          }
}
```

2.2. @RequestParam (accès aux paramètres HTTP)

conversion.jsp

```
… <form action="doConversion" method="GET_ou_POST">
          source: <select name="source" >
                <c:forEach var="d" items="${allDevises}">
                     <option  value="${d.monnaie}">${d.monnaie}</option>
                </c:forEach>
               </select>  <br/>
          cible: <select name="cible" > ... </select> <br/>
          montant: <input name="montant"  value="${montant}"/> <br/>
          <input  type="submit" value="convertir" /> <br/>
        </form>
        sommeConvertie=<b>${sommeConvertie}</b>   ...
```

```
@RequestMapping("/doConversion")
public String doConversion(Model model,@RequestParam(name="montant")double montant,
                                       @RequestParam(name="source")String monnaieSrc,
                                       @RequestParam(name="cible")String monnaieDest) {
….
   model.addAttribute("sommeConvertie",
                      gestionDevises.convertir(montant, monnaieSrc, monnaieDest));
   return "conversion";
}
```

2.3. @ModelAttribute

Pour spécifier un attribut du modèle on peut appeler ***model.addAttribute("attrName", attrVal);*** au sein d'une méthode préfixée par @RequestMapping .

Une autre solution consiste à coder une méthode addXyModelAttribute() préfixée par @ModelAttribute("attrName").

Exemple :

```
@ModelAttribute("conv")
        public ConversionForm addConvAttributeInModel() {
                return new ConversionForm();
        }
```

Le framework "spring mvc" va alors appeler automatiquement (*) toutes les méthodes préfixées par

@ModelAttribute pour initialiser certains attributs du modèle avant de déclencher les méthodes préfixées par *@RequestMapping*.

L'appel n'est effectué que pour initialiser la valeur d'un attribut n'existant pas encore (pas d'écrasement des valeurs en session ni des valeurs saisies via <form:form …./>)

Une méthode préfixée par ***@ModelAttribute*** peut éventuellement avoir un paramètre préfixé par ***@RequestParam(name="numCli",required=true_or_false)*** mais elle n'a pas le droit de retourner une valeur "null" pour un attribut du modèle .

Variante syntaxique (en void et avec model) pour de multiples initialisations :

```
@ModelAttribute
        public void addAttributesInModel(Model model){
  model.addAttribute("xx", new Cxx());
  model.addAttribute("yy", new Cyy());
}
```

Autre Exemple :

```
@Controller //but not "@Component" for spring web controller
//@Scope(value="singleton")//by default
@RequestMapping("/devises")
public class DeviseListCtrl {

        @Autowired //ou @Inject
        private GestionDevises gestionDevises;
```

```
        private List<Devise> listeDevises = null; //cache


        @PostConstruct
        private void loadListeDevises(){
                if(listeDevises==null)
                        listeDevises=gestionDevises.getListeDevises();
        }

    @ModelAttribute("allDevises")
        public List<Devise> addAllDevisesAttributeInModel() {
             return listeDevises;
        }

         @RequestMapping("/liste")
           public String toDeviseList(Model model) {
             //model.addAttribute("allDevises", listeDevises);
             return "deviseList";
           }
}
```

deviseList.jsp

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
"http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>liste des devises</title>
</head>
<body>
      <h3>liste des devises (spring web mvc)</h3>
      <table border="1" >
      <tr><th>code</th><th>devise</th><th>change</th></tr>
            <c:forEach var="d" items="${allDevises}">
                  <tr><td>${d.codeDevise}</td><td>${d.monnaie}</td>
                        <td>${d.DChange}</td></tr>
            </c:forEach>
      </table>
      <hr/>
      <a href="../app/to_welcome">retour page accueil</a> <br/>
</body>
</html>
```

<u>Accès à un attribut pour effectuer une mise à jour</u>:

```
@RequestMapping("/info")
   public String toInfosClient(Model model) {
      //mise à jour du telephone du client 0L (pour le fun / la syntaxe):
                Client cli = (Client) model.asMap().get("customer");
                if(cli!=null && cli.getNumero()==0L)
                        cli.setTelephone("0102030405");
```

```
    return "infosClient";
  }
```

2.4. @SessionAttributes

```
@Controller
//@Scope(value="singleton")//by default
@RequestMapping("/client")
@SessionAttributes( value={"customer"} )
   //noms des "modelAttributes" qui sont EN PLUS récupérés/stockés
   // en SESSION HTTP au niveau de la page de rendu
   // --> visibles en requestScope ET en sessionScope
public class ClientCtrl {


   //NB: @SessionAttributes et @ModelAttribute sont gérés avant @RequestMapping


        @ModelAttribute("customer") //NB: cette méthode n'est pas appelée/déclenchée
                        //si "customer" est déjà présent en session (et par copie) dans le modèle
        public Client addCustomerAttributeInModel() {
                        return new Client(0L,null,null) ;
        }
}
```

Mettre fin à une session http:

```
@RequestMapping("/endSession")
   public String endSession(Model model,HttpSession session) {
      if(model.containsAttribute("customer"))
                model.asMap().remove("customer");
      session.invalidate();
      return "infosClient";
   }
```

2.5. tags pour formulaires (form:form , form:input , ...)

Spring-mvc offre une bibliothèque de tags permettant de simplifier la structuration d'une page JSP comportant un formulaire (à saisir , à valider , ….).

```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"%>
```

Ces nouvelles balises préfixées par *form:* s'utilisent quasiment de la même façon que les balises

standards HTML (path="nomPropJava" à la place de name="nomParamHttp").

La principale valeur ajoutée des balises préfixées par *form:* consiste dans les liaisons automatiques entre certaines propriétés d'un objet java et les champs d'un formulaire.

Les balises <form:input ...> , <form:select ….> doivent être imbriquées dans <form:form >.

La balise principale d'un formulaire <***form:form*** *action="actionXY"* ***modelAttribute****="beanName" method="POST" > ... <form:form>* … comporte un attribut clef **modelAttribute** qui doit correspondre à un nom de "modelAttribute" lui même associé à un **objet java comportant toutes les données du formulaire à soumettre**.

Autrement dit , form:form ne fonctionne correctement que si la classe du sous-contrôleur est structurée avec au moins un "@ModelAttribute" (existant dès le départ , pas "null") dont le type correspond à une classe souvent spécifique au formulaire (ex : "UserForm" , "OrderForm" , ….) .

Exemple:

```
public class ConversionForm {
	private Double montant;
	private String monnaieSrc;
	private String monnaieDest;

	public ConversionForm(){
		monnaieSrc="dollar";
		monnaieDest="dollar"; //par défaut (dans formulaire avant saisies)
	}
	 //+ get/set
}
```

```
@Controller
//@Scope(value="singleton")//by default
@RequestMapping("/devises")
public class DeviseListCtrlV2 {
…
   //pour modelAttribute="conv" de  form:form
   @ModelAttribute("conv")
	public ConversionForm addConvAttributeInModel() {
		return new ConversionForm();
	}
…
}
```

L'attribut path="..." des sous balises <form:input ...> , <form:select ….> font alors référence aux propriétés de l'objet java (en lecture/écriture , get/set) .

NB: <form:form …> gère (génère) automatiquement le champ caché **_csrf** attendu par **spring-**

security . *Exemple* : <input type="hidden" name="_csrf" value="8df91b84-74c1-4013-bd44-ede7b00779a2" />) . 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>) , 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}"/>

conversionV2.jsp

```
<form:form action="doConversion"  modelAttribute="conv" method="POST">
  source: <form:select path="monnaieSrc" >
                <form:options items="${allDevises}" itemLabel="monnaie" itemValue="monnaie"/>
          </form:select>  <br/>
  cible: <form:select path="monnaieDest" >
                <form:options items="${allDevises}" itemLabel="monnaie" itemValue="monnaie"/>
        </form:select> <br/>
   montant: <form:input path="montant" />
              <form:errors path="montant" cssClass="error"/><br/>
   <input  type="submit" value="convertir" /> <br/>
</form:form>
sommeConvertie=<b>${sommeConvertie}</b>
```

conversion de devises

source: dollar
cible: euro
montant: 45.0
convertir
sommeConvertie=37.5

Finalement , au sein du contrôleur , la méthode déclenchée par le formulaire peut s'écrire de la façon suivante:

```
@RequestMapping("/doConversion")
public String doConversion(Model model,@ModelAttribute("conv")  ConversionForm conv ) {

model.addAttribute("sommeConvertie",
                        gestionDevises.convertir(conv.getMontant(),
```

```
                                                    conv.getMonnaieSrc(), conv.getMonnaieDest()));

  return "conversionV2";
}
```

2.6. validation lors de la soumission d'un formulaire

Rappel: la classe de l'objet utilisé en tant que "modelAttribute" au niveau d'un formulaire peut comporter des annotations @Min , @Max , @Size , @NotEmpty , … de l'api normalisée javax.validation .

Exemples :

```
import javax.validation.constraints.Max;
import javax.validation.constraints.Min;

public class ConversionForm {

        @Min(value=0)
        @Max(value=999999)
        private Double montant;
    …
}
```

```
import javax.validation.constraints.Size;
import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotEmpty;

public class Client {
        private Long numero;    private String nom;       private String prenom;

        @NotEmpty(message = "Please enter your address.")
        @Size(min = 4, max = 128, message = "Your address must between 4 and 128 characters")
        private String adresse;
        private String telephone;

        @NotEmpty
        @Email
        private String email;
…
}
```

Il suffit en suite d'ajouter **@Valid** au niveau du paramètre de la méthode associée à la soumission du formulaire pour que spring-mvc tienne compte des contraintes de validation.

D'autre part, le paramètre (facultatif mais conseillé) de type "**BindingResult**" permet de gérer finement les cas d'erreur de validation :

```
@RequestMapping("/doConversion")
public String doConversion(Model model,
```

```
                                @ModelAttribute("conv") @Valid ConversionForm conv ,
                                BindingResult bindingResult) {
        if (bindingResult.hasErrors()) {
                // form validation error
                System.out.println("form validation error: " + bindingResult.toString());
        } else {
                 // form input is ok*/
        model.addAttribute("sommeConvertie", gestionDevises.convertir(conv.getMontant(),
                                                conv.getMonnaieSrc(), conv.getMonnaieDest()));
        }
return "conversionV2";
}
```

conversion de devises

source: dollar
cible: livre
montant: -5.0 **doit être plus grand que 0**
convertir
sommeConvertie=

---

retour page accueil

numero: 0
nom: Therieur
prenom: alex
adresse: ici **Your address must between 4 and 128 characters**
telephone:
email: alex-therieur **Adresse email mal formée**
update

2.7. Compléments pour mise en page

Pour obtenir de belles mises en pages , on pourra coupler "spring-mvc" avec **bootstrap-css** et/ou "tiles" ou "**thymeleaf**" .

3. Web services "REST" pour application Spring

Pour développer des Web Services "REST" au sein d'une application Spring , il y a deux possibilités distinctes (à choisir) :

- s'appuyer sur l'API standard **JAX-RS** et choisir une de ses implémentations (**CXF3** ou **Jersey** ou …)
- s'appuyer sur le framework "**Spring web mvc**" et utiliser **@RestController** .

La version "JAX-RS standard" nécessite pas mal de librairies (jax-rs, jersey ou cxf , jackson et tout un tas de dépendances indirectes) .

La version spécifique spring nécessite un peu moins de librairies (spring-web , spring-mvc , jackson) :

```
        <dependency>
                        <groupId>org.springframework</groupId>
                        <artifactId>spring-webmvc</artifactId>
                        <version>4.3.2.RELEASE</version>
                        <scope>compile</scope>
                </dependency>

                <dependency>
                        <groupId>com.fasterxml.jackson.core</groupId>
                        <artifactId>jackson-databind</artifactId>
                        <version>2.5.4</version> <!-- to produces json -->
                </dependency>
...
```

4. WS REST via Spring MVC et @RestController

NB : La compréhension de cet exemple nécessite de maîtriser les bases du framework **spring-mvc** (**@RequestMapping** , **mvc-config.xml** ou équivalent , **DispatcherServlet , ...)**

Exemple :

DeviseJsonRestCtrl.java

```
package tp.app.zz.web.rest;

import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.ResponseBody;
import org.springframework.web.bind.annotation.RestController;
...
```

```
@RestController
@RequestMapping(value="/rest/devises" , headers="Accept=application/json")
public class DeviseJsonRestCtrl {

@Autowired //ou @Inject
private GestionDevises gestionDevises;

//URL de déclenchement: webappXy/mvc/rest/devises/
                        //ou webappXy/mvc/rest/devises/?name=euro
@RequestMapping(value="/" , method=RequestMethod.GET)
//@ResponseBody  par défaut avec @RestController
List<Devise> getDevisesByCriteria(@RequestParam(value="name",required=false)
                                        String nomMonnaie) {
        if(nomMonnaie==null)
                return gestionDevises.getListeDevises();
        else{
                List<Devise> listeDev= new ArrayList<Devise>();
                Devise devise = gestionDevises.getDeviseByName(nomMonnaie);
                if(devise!=null)
                        listeDev.add(devise);
                return listeDev;
           }
}


//URL de déclenchement: webappXy/mvc/rest/devises/EUR
@RequestMapping(value="/{codeDevise}" , method=RequestMethod.GET)
//@ResponseBody  par défaut avec @RestController
 Devise getDeviseByName(@PathVariable("codeDevise") String  codeDevise) {
        return gestionDevises.getDeviseByPk(codeDevise);
}

 //URL : webappXy/mvc/rest/devises/convert?amount=50&src=EUR&target=USD
@RequestMapping(value="/convert" , method=RequestMethod.GET ,
                        headers="Accept=text/plain")
//@ResponseBody  par défaut avec @RestController
 String convert(@RequestParam("amount") double  amount,
                @RequestParam("src") String  src ,
                @RequestParam("target") String  target) {
        double sommeConvertie=gestionDevises.convertir(amount, src, target);
        System.out.println("sommeConvertie="+sommeConvertie);
        return String.valueOf(sommeConvertie);
        //ou bien résultat au format ClasseSpecifiqueJava convertie au format json
}

}
```

Prise en charge des modes "PUT" , "POST" , "DELETE" :

```
...
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMethod;
import com.fasterxml.jackson.databind.DeserializationFeature;
import com.fasterxml.jackson.databind.ObjectMapper;

@RestController
@RequestMapping(value="/rest/devises" , headers="Accept=application/json")
public class DeviseJsonRestCtrl {
...
 @RequestMapping(value="/" , method=RequestMethod.PUT )
 @ResponseBody  // ou @ResponseStatus(value = HttpStatus.OK)
Devise updateDevise(@RequestBody String deviseAsJsonString) {
        Devise devise=null;
        try {
                ObjectMapper jacksonMapper = new ObjectMapper();
                jacksonMapper.configure(
                        DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
                devise = jacksonMapper.readValue(deviseAsJsonString,Devise.class);
                System.out.println("devise to update:" + devise);
                gestionDevises.updateDevise(devise);
                return devise;
        } catch (Exception e) {
                e.printStackTrace();
                return null;
        }
} ….}
```

Apport important de la version 4 : **ResponseEntity<T>**

Depuis "Spring4" , une méthode d'un web-service REST a plutôt intérêt à retourner une réponse de Type **ResponseEntity<T>** ce qui permet de **retourner d'un seul coup**:

- **un statut** (OK , NOT_FOUND , ...)
- **le corps de la réponse** : objet (ou liste) T convertie en json
- **un éventuel "header"** (ex: url avec id si auto_incr lors d'un POST)

Exemple:

```
@RequestMapping(value="/{codeDev}" , method=RequestMethod.GET)
 ResponseEntity<Devise> getDeviseByName(@PathVariable("codeDev") String  codeDevise) {
        Devise dev = gestionDevises.getDeviseByPk(codeDevise);
        return new ResponseEntity<Devise>(dev, HttpStatus.OK);
}
```

Exemple de page HTML/jquery pour le déclenchement des WS "REST" :

JSON tests for devise app (REST/JSON via spring)

devises avec code=EUR
devises avec name=dollar
toutes les devises
50 euros en dollar

---

devise (to update) : EUR (euro)

---

code : EUR
monnaie : euro
change : 1.1152
update devise
updated data (server side):
{"monnaie":"euro","codeDevise":"EUR","dchange":1.1152,"pk":"EUR"}

```
<html >
  <head>
       <script src="jquery-2.2.1.js"></script>
  <script>
        var deviseList;
        var deviseIdSelected;//id=.codeDevise
        var deviseSelected;

        function display_selected_devise(){
              $("#spanMsg").html( "selected devise:" + deviseIdSelected) ;
              $('#spanCode').html(deviseSelected.codeDevise);
              $('#txtName').val(deviseSelected.monnaie);
              $('#txtExchangeRate').val(deviseSelected.dchange);
        }

        function local_update_selected_devise(){
              deviseSelected.monnaie = $('#txtName').val();
              deviseSelected.dchange= $('#txtExchangeRate').val();
        }



$(function() {
    $.ajax({
        type: "GET",
        url: "mvc/rest/devises/",
       success: function (data) {
```

```
            if (data) {
                  //alert(JSON.stringify(data));
                  deviseList = data;
                  for(deviseIndex in deviseList){
                        var devise=deviseList[deviseIndex];
                         if(deviseIndex==0)
                              { deviseSelected = devise; deviseIdSelected = devise.codeDevise; }
                                 //alert(JSON.stringify(devise));
                        $('#selDevise').append('<option  value="'+ devise.codeDevise +'">'+
                                          devise.codeDevise + ' (' + devise.monnaie + ')</option>');
                  }
                   display_selected_devise();
            } else {
               $("#spanMsg").html("Cannot GET devises !");
            }
         }
 });









$('#btnUpdate').on('click',function(){
// $("#spanMsg").html( "message in the bottle" ) ;
 local_update_selected_devise();
$.ajax({
         type: "PUT",
         url: "mvc/rest/devises/",
         dataType: "json",
         data: JSON.stringify(deviseSelected),
         success: function (updatedData) {
            if (updatedData) {
              $("#spanMsg").html("updated data (server side):" + JSON.stringify(updatedData));
            } else {
               $("#spanMsg").html("Cannot PUT updated data");
            }
         }
         });
});

$('#selDevise').on('change',function(evt){
         deviseIdSelected = $(evt.target).val();
         for(deviseIndex in deviseList){
                  var devise=deviseList[deviseIndex];
                   if(devise.codeDevise == deviseIdSelected)
                                    deviseSelected  = devise;
         }
         display_selected_devise();
         });
```

```
});
 </script>
 </head>
  <body>

   <h3> JSON tests for devise app (REST/JSON via spring) <h3>
        <a href="mvc/rest/devises/EUR"> devises avec code=EUR </a> <br/>
        <a href="mvc/rest/devises/?name=dollar"> devises avec name=dollar </a> <br/>
        <a href="mvc/rest/devises/"> toutes les devises</a> <br/>
        <a href="mvc/rest/devises/convert?amount=50&src=euro&target=dollar">
            50 euros en dollar</a> <br/>
  <hr/>
       devise (to update) : <select id="selDevise"> </select>
       <hr/>
       code : <span id="spanCode" ></span><br/>
       monnaie : <input id="txtName" type='text' /><br/>
       change : <input id="txtExchangeRate" type='text' /><br/>
       <input type='button' value="update devise" id="btnUpdate"/> <br/>
       <span id="spanMsg"></span> <br/>
       </body>
  </html>
```

XII - Spring security

1. Extension Spring-security

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 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 xml , database , ldap)
- possibilité de configurer via l'annotation *@PreAuthorize("hasRole('role1') or hasRole('role2')")* les méthodes des services "spring" qui seront ou pas accessibles selon le rôle de l'utilisateur authentifié.
- autres fonctionnalités avancés (cryptage des mots de passe , ...)

1.1. Exemple de configuration maven pour spring-security :

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

avec par exemple :

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

1.2. 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>
```

1.3. 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_" .

Exemple (en version "java-config") :

```
@EnableWebSecurity
public class WebSecurityConfig extends
        WebSecurityConfigurerAdapter {

      @Override
       protected void configure(HttpSecurity http) throws Exception {
            // http.csrf().disable();
         }
}
```

1.4. Champ caché "_csrf "

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>) , 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 …> gère (génère) automatiquement le champ caché **_csrf** attendu par **spring-security** . *Exemple* : <input type="hidden" name="_csrf" value="8df91b84-74c1-4013-bd44-ede7b00779a2" />) .

../..

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);
        });
 }
//...
}
```

XIII - Spring Data

1. Spring-Data

L'extension "**Spring-Data**" permet (entre autre) de :

- **générer automatiquement des composants "DAO / Repository" modernes** (utilisables avec des technologies SQL , NO-SQL ou orientées graphes telles que JPA , MongoDB , Neo4J)
- accélérer le temps de développement (l'interface suffit souvent, la classe d'implémentation sera générée dynamiquement par introspection et selon certaines conventions).
- standardiser le format des composants "DAO/Repository" : mêmes méthodes fondamentales. On parle alors en termes de "composants DAO consistants" → des automatismes sont possibles (tests en partie automatique , ….) .

1.1. Spring-data-commons

"**Spring-data-commons**" est la partie centrale de Spring-data sur laquelle pourra se greffer certaines extensions (pour jpa , pour mongo , …) .

"Spring-data-commons" est essentiellement constituée de **3 interfaces** : **Repository** , **CrudRepository** et **PagingAndSortingRepository** .

- **Repository<T,ID>** n'est qu'une interface de marquage dont toutes les autres héritent.
- **CrudRepository<T,ID>** standardise les méthodes fondamentales (findOne , findAll , save , delete, …)
- **PagingAndSortingRepository<T,ID>** étend CrudRepository en ajoutant des méthodes supportant le tri et la pagination.

Méthodes fondamentales de **CrudRepository**<T ,ID extends Serializable> :

`<S extends T> S save(S entity);`	Sauvegarde l'entité (au sens saveOrUpdate) et retourne l'entité (éventuellement ajustée/modifiée dans le cas d'une auto-incrémentation ou autre).
`T findOne(ID primaryKey);`	Recherche par clef primaire
`Iterable<T> findAll();`	Recherche toutes les entités (du type courant/considéré)
`Long count();`	Retourne le nombre d'entités existantes

`void delete(T entity);`
 	
`void delete(ID primaryKey);`
 	
`void deleteAll();`
	Supprime une (ou plusieurs) entités

`boolean exists(ID primaryKey);`	Test l'existence d'une entité

Variantes de quelques méthodes (surchargées) au sein de CrudRepository :

`<S extends T> Iterable<S> save(Iterable<S> entities);`	Sauvegarde une liste d'entités
`Iterable<T> findAll(Iterable<ID> ids );`	Recherche toutes les entités (du type courant/considéré) ayant les Ids demandés
`void delete(Iterable< ? Extends T> entities)`	Supprime une liste d'entités

Rappel : java.util.**Collection**<E> et java.util.**List**<E> héritent de **Iterable**<E>

Fonctionnalité "**tri**" apportée en plus par l'interface **PagingAndSortingRepository** :

```
...
   Iterable<Personne> personnesTrouvees =
   personnePaginationRep.findAll(new Sort(Sort.Direction.DESC, "nom"));
   …
```

où **org.springframework.data.domain.Sort** est spécifique à Spring-data .

Fonctionnalité "**pagination**" apportée en plus par l'interface **PagingAndSortingRepository** :

```
public void testPagination() {
   assertEquals(10, personnePaginationRep.count());
   Page<Personne> pageDePersonnes =
   // 1re page de résultats et 3 résultats max.
   personnePaginationRep.findAll(new PageRequest(1, 3));
   assertEquals(1, pageDePersonnes.getNumber());
   assertEquals(3, pageDePersonnes.getSize());// la taille d'une page
   assertEquals(10, pageDePersonnes.getTotalElements());
   assertEquals(4, pageDePersonnes.getTotalPages());
   assertTrue(pageDePersonnes .hasContent());
   ...
  }
```

Avec comme types précis :

org.springframework.data.domain.Page<T>

et **org.springframework.data.domain.PageRequest** implémentant l'interface **org.springframework.data.domain.Pageable**

1.2. Spring-data-jpa

1.3. Spring-data-mongo

1.4. Spring-data-Neo4J

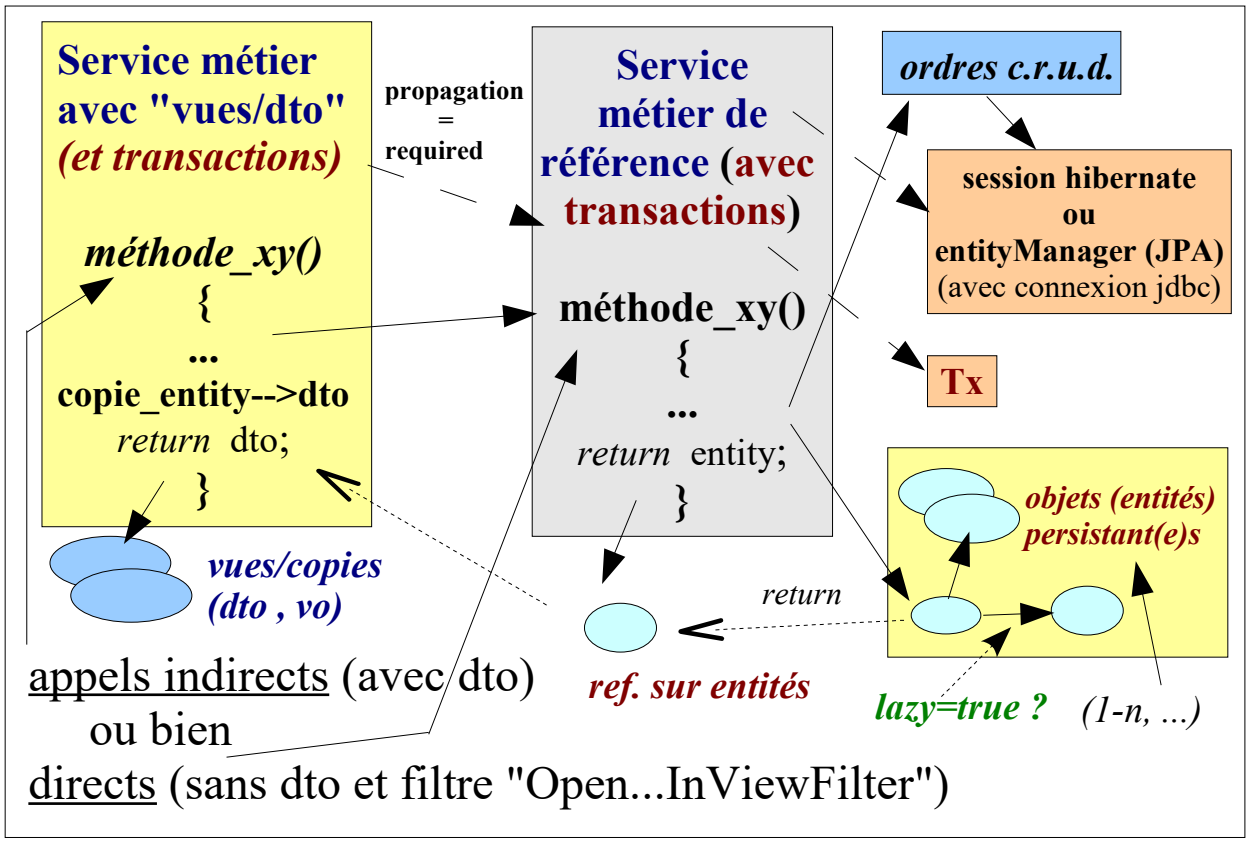
XIV - OpenInViewFilter et mode "D.R.Y."

D.R.Y. (Don't Repeat Yourself)

1.1. Eventuelle solution mixte/combinée (avec ou sans DTO) :

Deux versions du service peuvent éventuellement cohabiter :

- une version (de référence) sans DTO que l'on peut utiliser en direct avec certaines précautions (Open...InViewFilter , ...).
- une version avec DTO qui s'appuie en interne sur la version de référence (injectée) et qui effectue en plus des conversions de format de données ([Dto-->Entity] sur les paramètres d'entrée et [Entity-->Dto] sur les valeurs de retour).
  NB: avec @Transactional placé sur les méthodes du service avec DTO , on a le temps de parcourir les collections sur les entités persistantes même en mode "lazy" car l' entityManager de JPA (ou la session hibernate) ne sera fermé(e) qu'en fin d'exécution des méthodes.



Il est cependant clair que l'on "s'embête généralement à programmer certaines choses (ici avec DTO)" que si l'on pense qu'elles seront utilisées un jour !!!!!

---

en général :

sur petit ou moyen projet (à réaliser "vite, vite, vite") --> pas de DTO

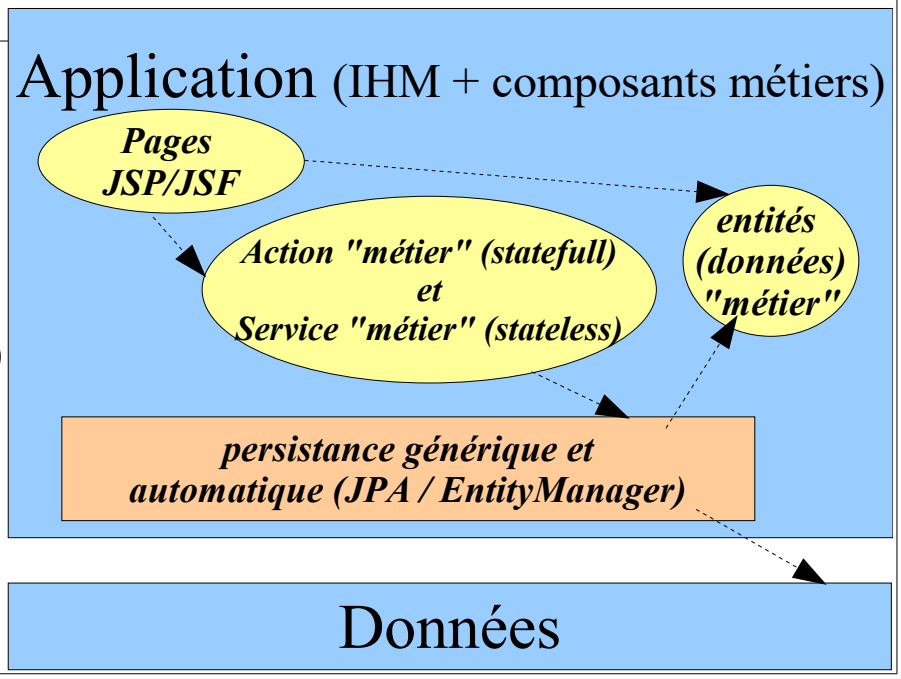
sur gros projet (où il y a plusieurs équipes séparées "back-office" , "front-office" , ...)

--> avec DTO

2. Eléments d'architecture logicielle (DRY)

Architecture à couches *ré-assemblées/fusionnées*

Ex:
Framework Seam
(Jsp/Jsf + EJB3/JPA)
ou bien ***Spring*** *(+Jsf +Jpa+Cxf)*



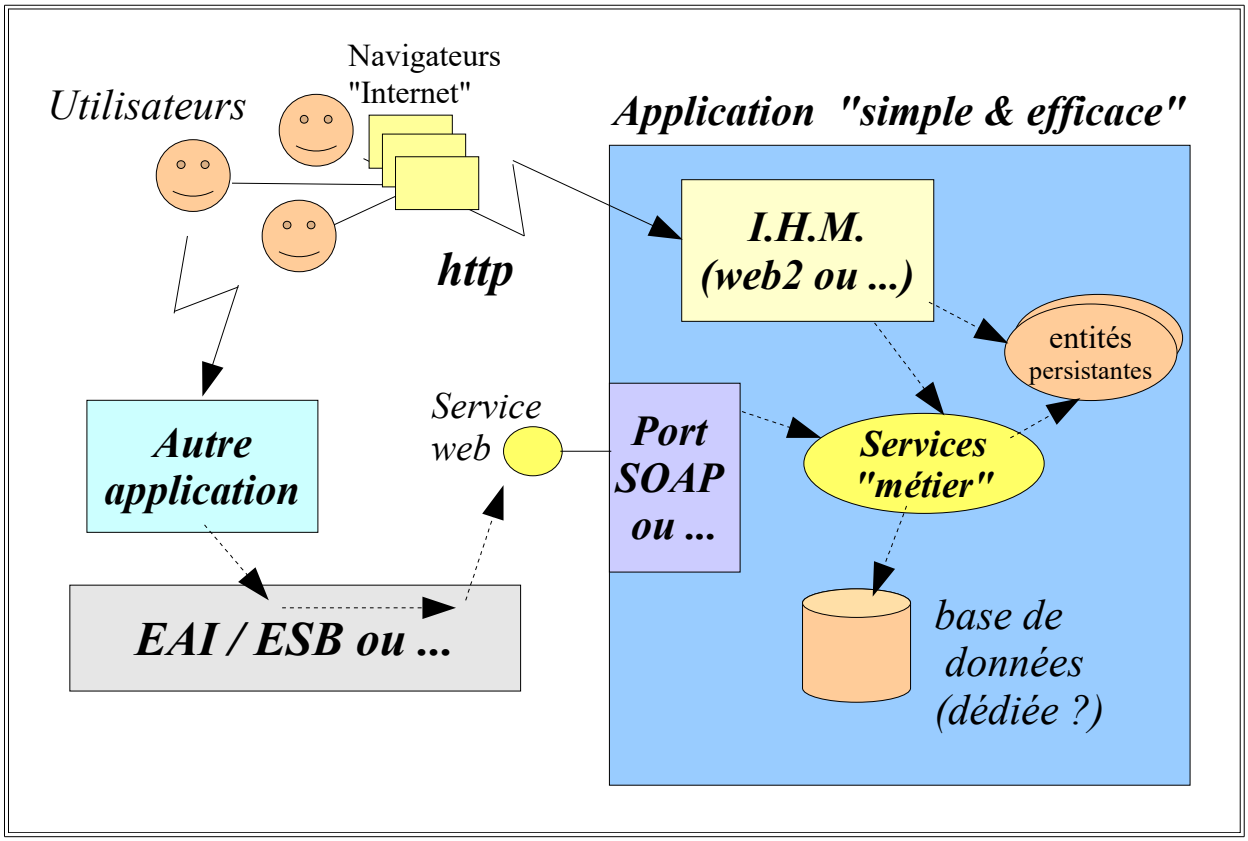
ex: SGBDR (Oracle, MySql)

DRY : Don't Repeat Yourself !!!

Si plein de couches logicielles, alors:
- éventuelles vérifications (non nul , >0 , ...) à plusieurs niveaux (saisies , traitements métiers , persistance , ...)
- plein de remontées d'exceptions à gérer
- éventuelles reprises structurelles (partielles) : "Dto/Vo" proches des entités , ...
- si évolution fonctionnelle de type nouveau champ --> tous les niveaux sont alors impactés (saisies , ..., persistance).

Si peu de couches logicielles, alors :
- simples @NotNull , @Min(...) , @Max(..) utilisés automatiquement par tous les niveaux (ihm web2/ajax , persistance , ...) --> Esprit "D.R.Y." .
- collaboration plus forte entre les niveaux, plus d'automatismes.

NB: Les annotations @NotNull , ... (pour JavaBean de HibernateValidator ou autre) peuvent directement être utilisées par l'extension "richFaces" pour JSF .



Pour garantir une bonne "maintenance & évolution applicative", ce qui compte le plus c'est l'échelle de ce qui doit être re-déployé (appliWeb.war ou .....) et les contrats avec l'extérieur.

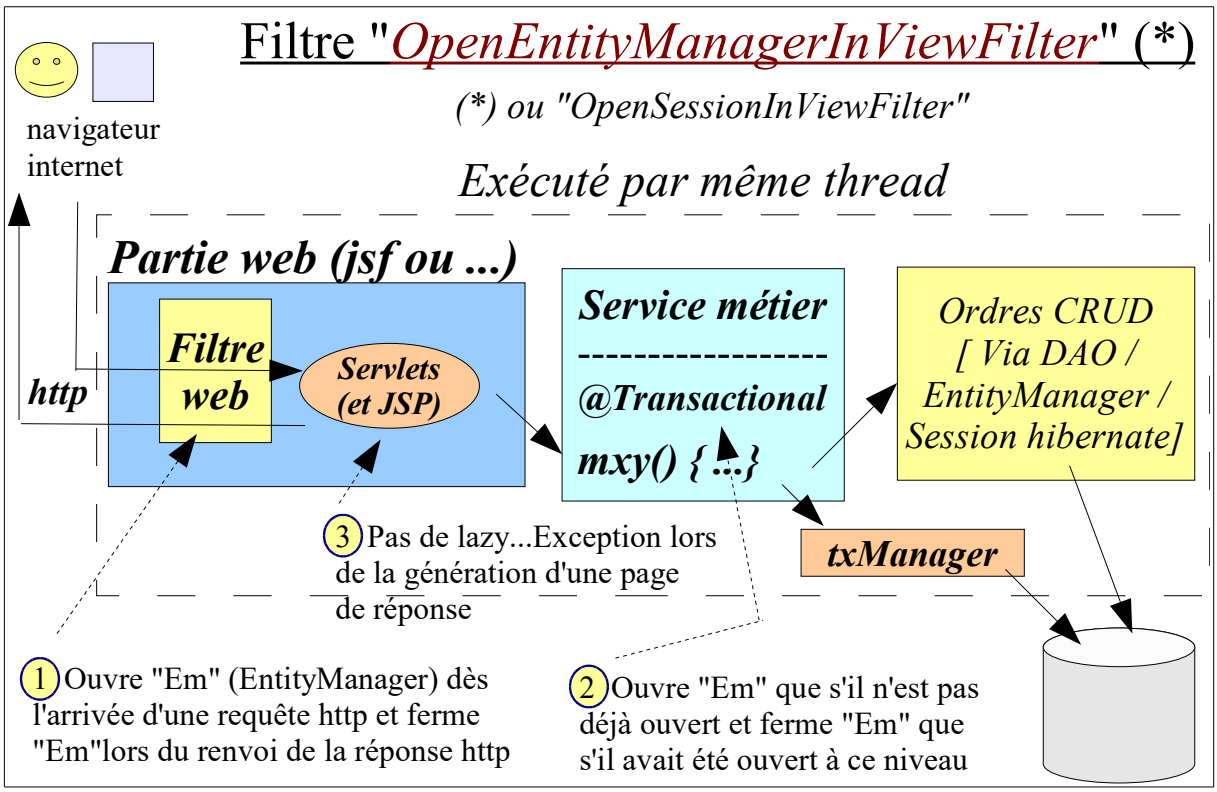
3. OpenSessionInViewFilter pour lazy="true"

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>
```

XV - Dao Générique , Convertisseur et DOZER

1. 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.

1.1. 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>
```

1.2. 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).

1.3. 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> , ...).

1.4. 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]

2. 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 defaut

        /**
         * constructeur par défaut (avec "dozerBeanMapping.xml" par defaut)
         */
        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 defaut =     { "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;
        }
}
```