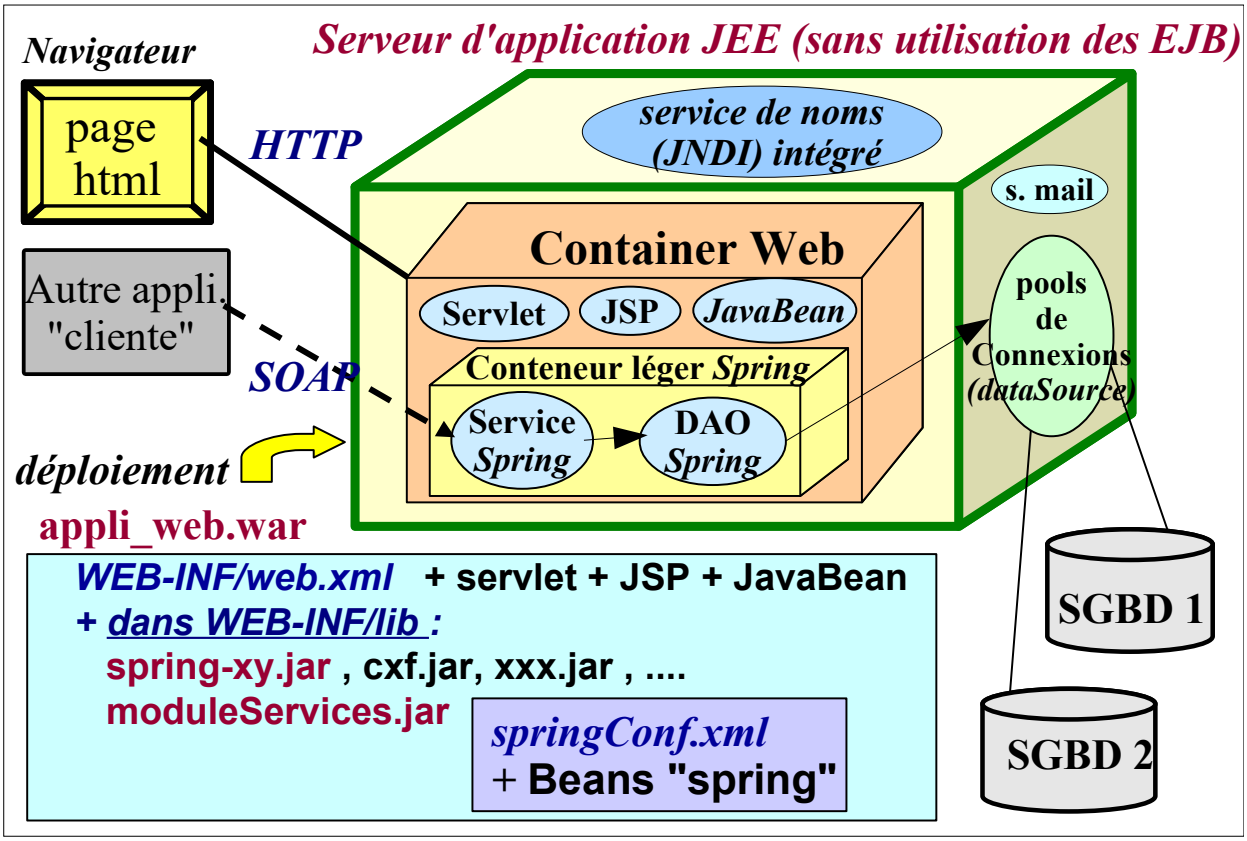
# Spring 4 et 5

## Table des matières

# I - Spring : vue d'ensemble

## 1. Architecture / Ecosystème Spring

Durant la première décennie du XXI siècle , Spring était à essentiellement considérer comme une alternative aux EJB et respectant les spécifications JEE :




Dès les premières versions, le framework open source "Spring" apportait les principales fonctionnalités suivantes :

- **intégration de composants** complémentaires inter-dépendants via le design-pattern "**injection de dépendances / ioc**" . configuration souple et flexible

- prise en charge automatique et "déclarative" (via config xml ou annotations) des **transactions** (commit/rollback)

- **intégration** des principaux autres frameworks java/JEE ( **Hibernate/Jpa** , Struts , **JSF** , JDBC , ...)

- **intercepteurs** (aop)

- **tests unitaires** simples (JUnit + spring-test)

- quelques éléments de sécurité (sécurité JEE simplifiée)

....

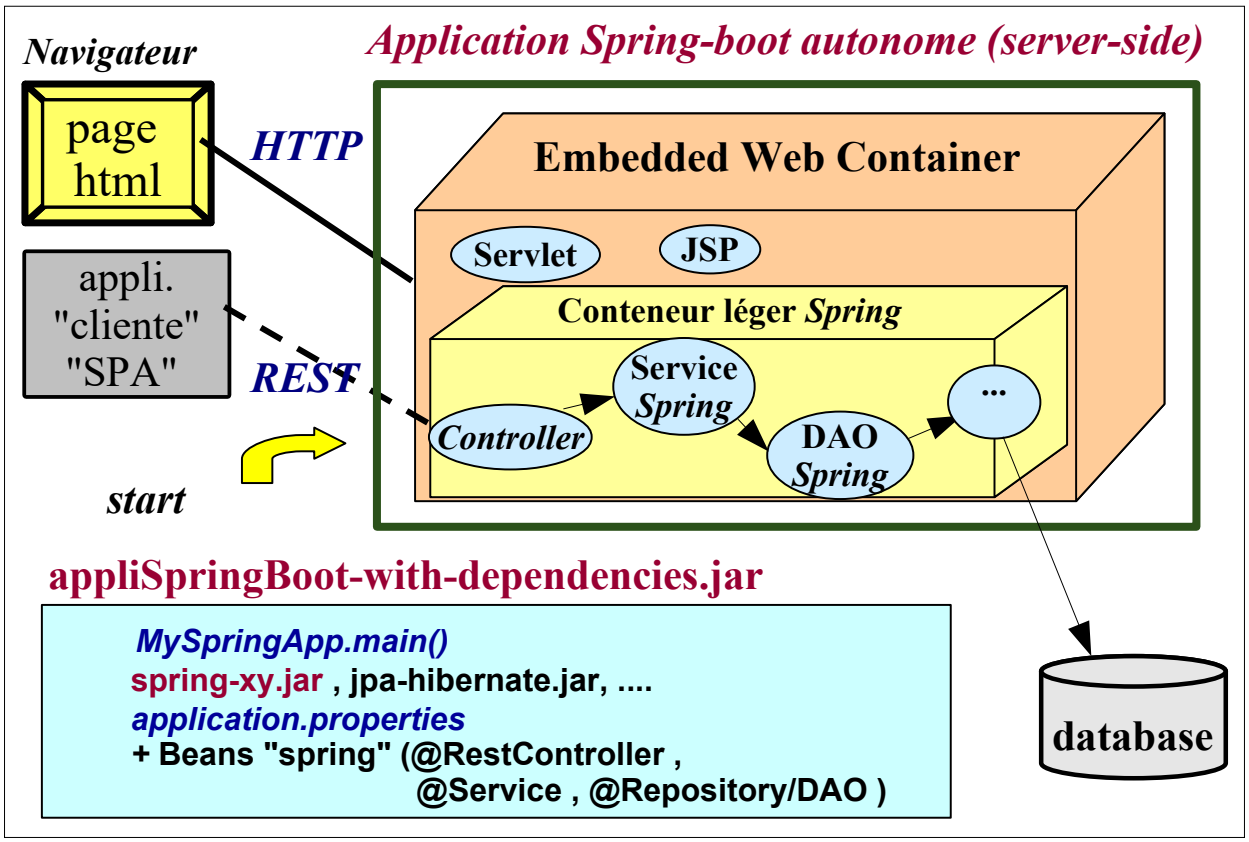
Le framework spring n'est pas associé à un grand éditeur de serveur JEE (tel que IBM , Oracle/BEA , Jboss) . Il a toujours laissé place à une très **grande liberté** dans le choix des technologies utilisées au sein d'une application java/JEE .

A partir de la version 4 , Le framework spring a introduit tout un tas de spécificités très intéressantes qui se démarquent clairement des spécifications JEE officielles .

Principales fonctionnalités supplémentaires apportées par les versions 4 et 5 de Spring :

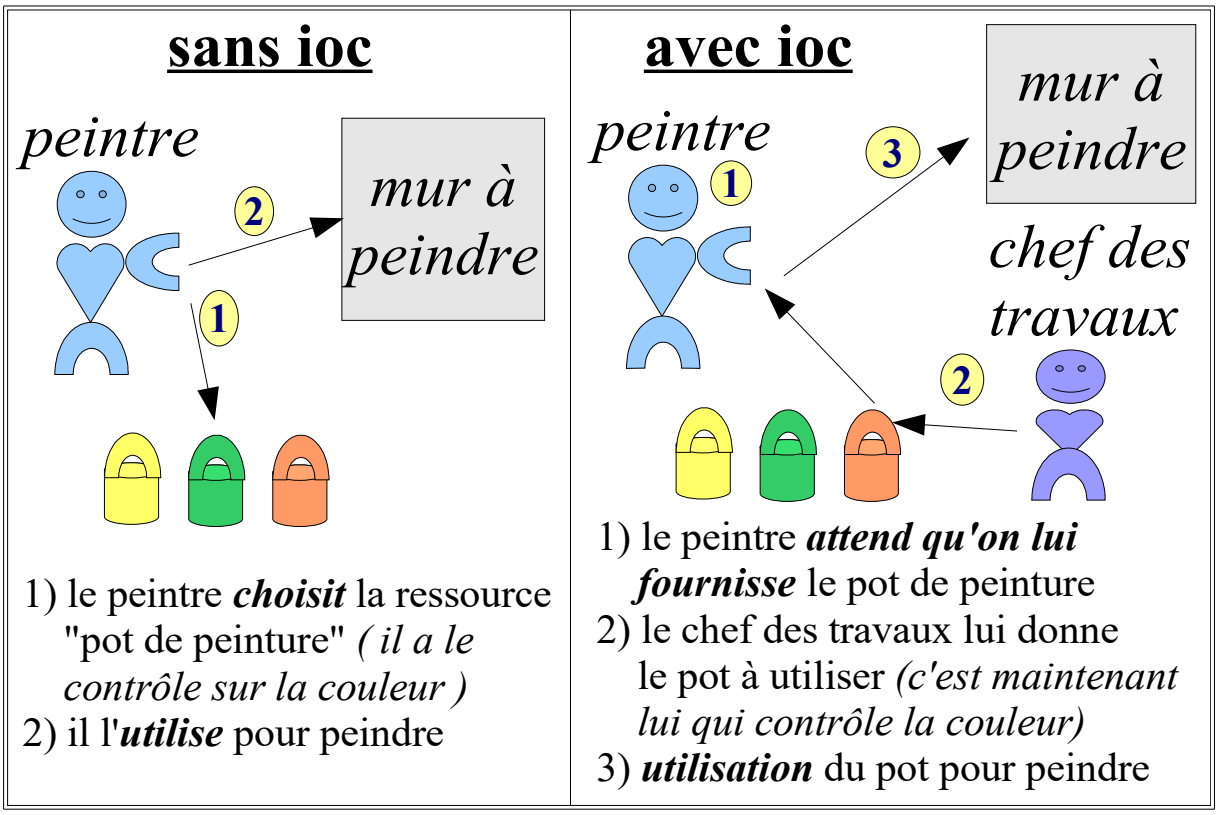
- **Spring boot** (démarrage complètement autonome . l'application incorpore son propre conteneur web (tomcat ou jetty ou netty ou ...)
- simplification de la configuration maven (ou gradle) via héritage de "POM/BOM/parent" .
- **Configuration java** (plus sophistiquée que l'ancienne configuration Xml , auto-complétion, rigueur , héritage , configuration conditionnelle intelligente)
- **AutoConfiguration** et simple fichier **application.properties**
- **Spring Data** (composants "DAO" générés automatiquement à partir des signatures des méthodes d'une interface, implémentation possible via JPA et MongoDB , paramétrages possibles via @NamedQuery ou autres, ...)
- web services REST via **@RestController** de Spring-mvc
- sécurisation flexible via **Spring-security**
- autres fonctionnalités diverses (*actuators* : mesures de perf , ...) , ....

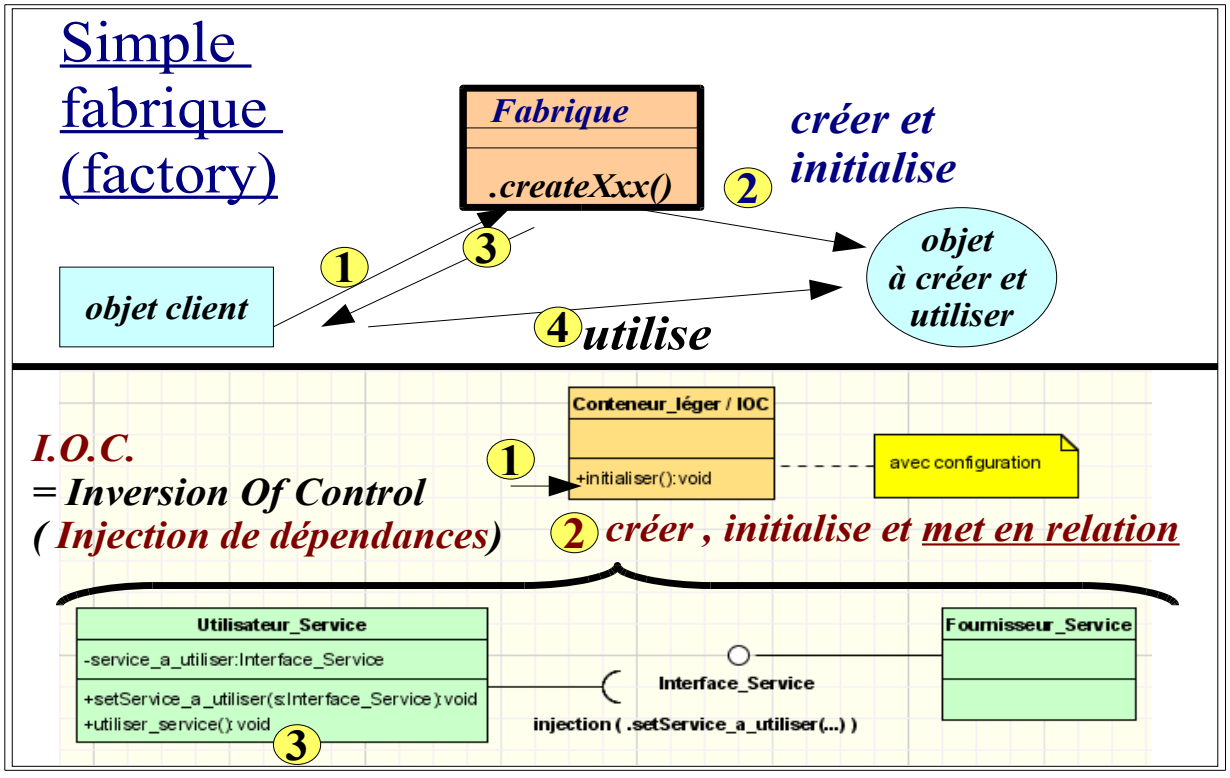
Toutes ces fonctionnalités (bien pratiques) sont "hors spécifications JEE" et l'on peut aujourd'hui considérer que "**Spring**" forme un "**écosystème complet**" pour faire fonctionner des applications professionnelles "java/web" .



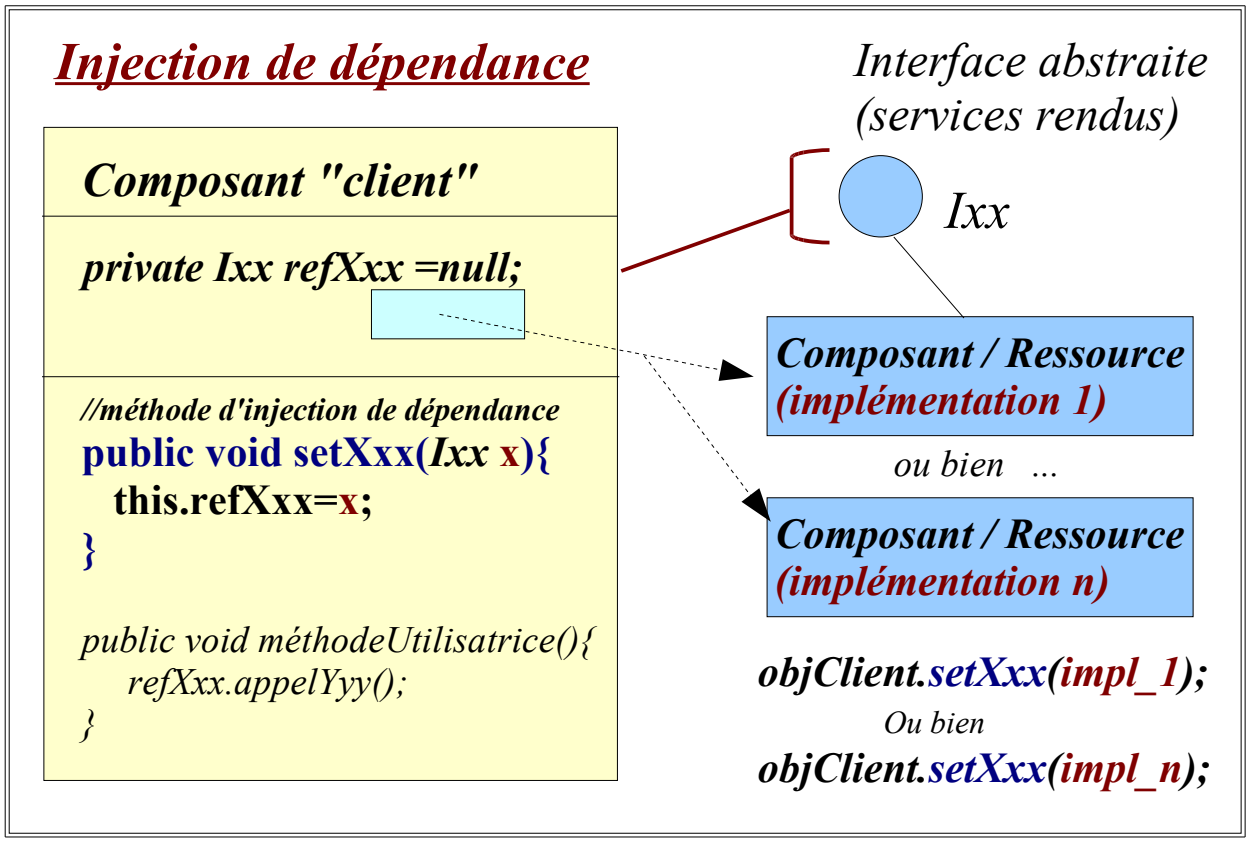
# 2. Design Pattern "I.O.C." / injection de dépendances

## 2.1. IOC = inversion of control



## 2.2. 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)***

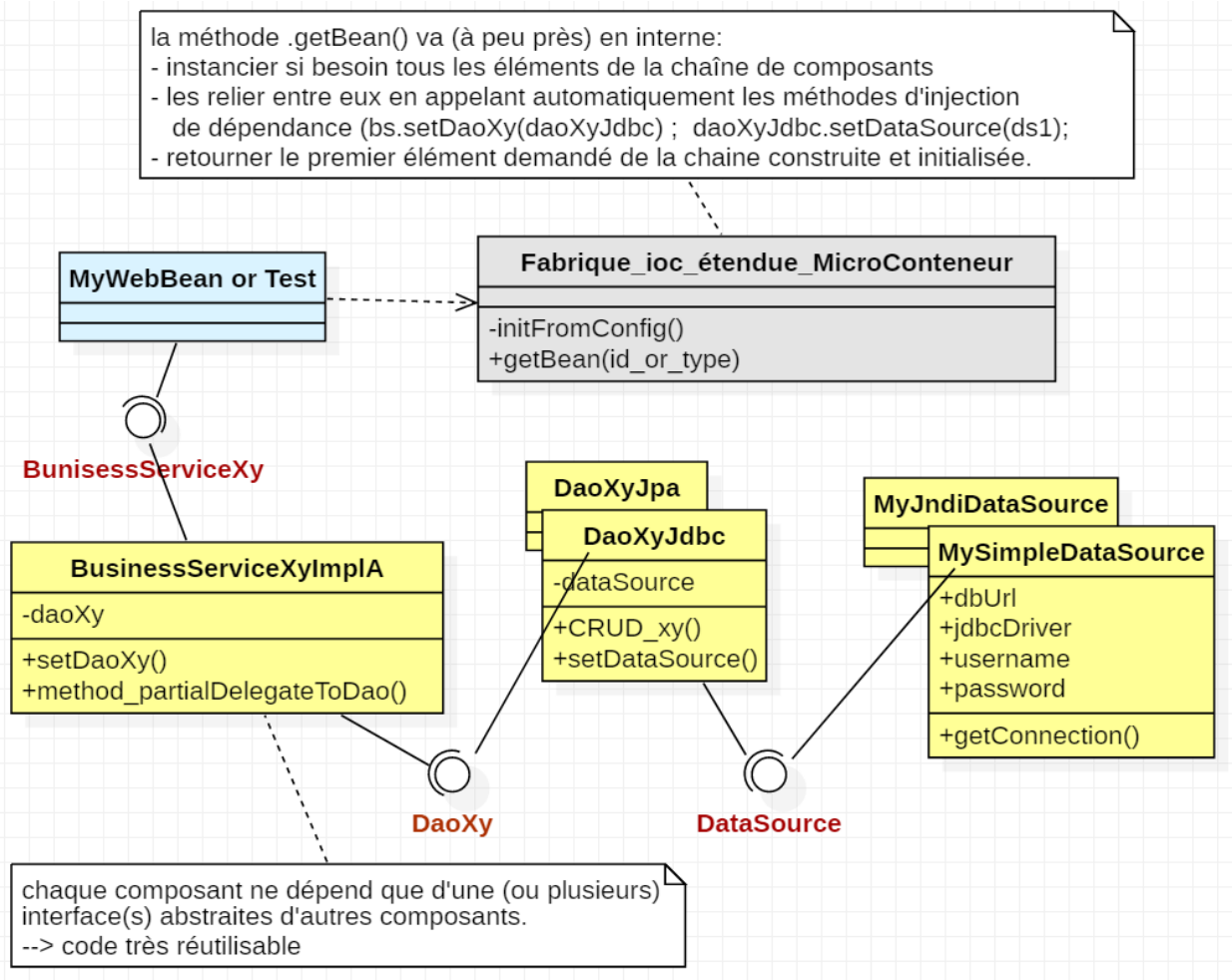
**ou bien un constructeur de type**:

***public CXxx(AbstractRessource res)***

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

Un tel composant est beaucoup plus réutilisable .

## 2.3. avec conteneur I.O.C. (super fabrique globale)




## 2.4. Micro-kernel / conteneur léger

Pour être facilement exploitable , le design pattern "injection de dépendances" 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 ou bien via une ***configuration spécifique (java, implicite ou explicite, ...)***).

# 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, ....)<br>+ 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,<br>points d'intégration pour des frameworks STRUTS , JSF, ... |
| Spring Web **MVC**<br>**(optionnel mais recommandé )** | Version "Spring" pour un framework Web/MVC. Ce framework est "simple/extensible" et "IOC".<br>Vis à vis du concurrent "JSF" , c'est visuellement plus pauvre<br>mais c'est moins exclusif , c'est plus flexible , modulaire et ça peut s'associer à d'autres technolologies complémentaires (ex : thymeleaf).<br>NB : Spring web mvc est très souvent utilisé comme alternative possible à JAX-RS pour développer des services web "REST" |

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

**Modules complémentaires pour Spring (extensions facultatives) :**

***Extensions fondamentales :***

| Extensions Spring | Contenus / spécificités |
|---|---|
| Spring-**security** | Extension très utile pour gérer la sécurité JEE (roles , authentification , …) |
| Spring **Data** | Dao automatiques (pouvant être basés sur JPA ou bien MongoDB ou ... ) . Très bonne extension. Attention aux différences "spring4 , spring5" |

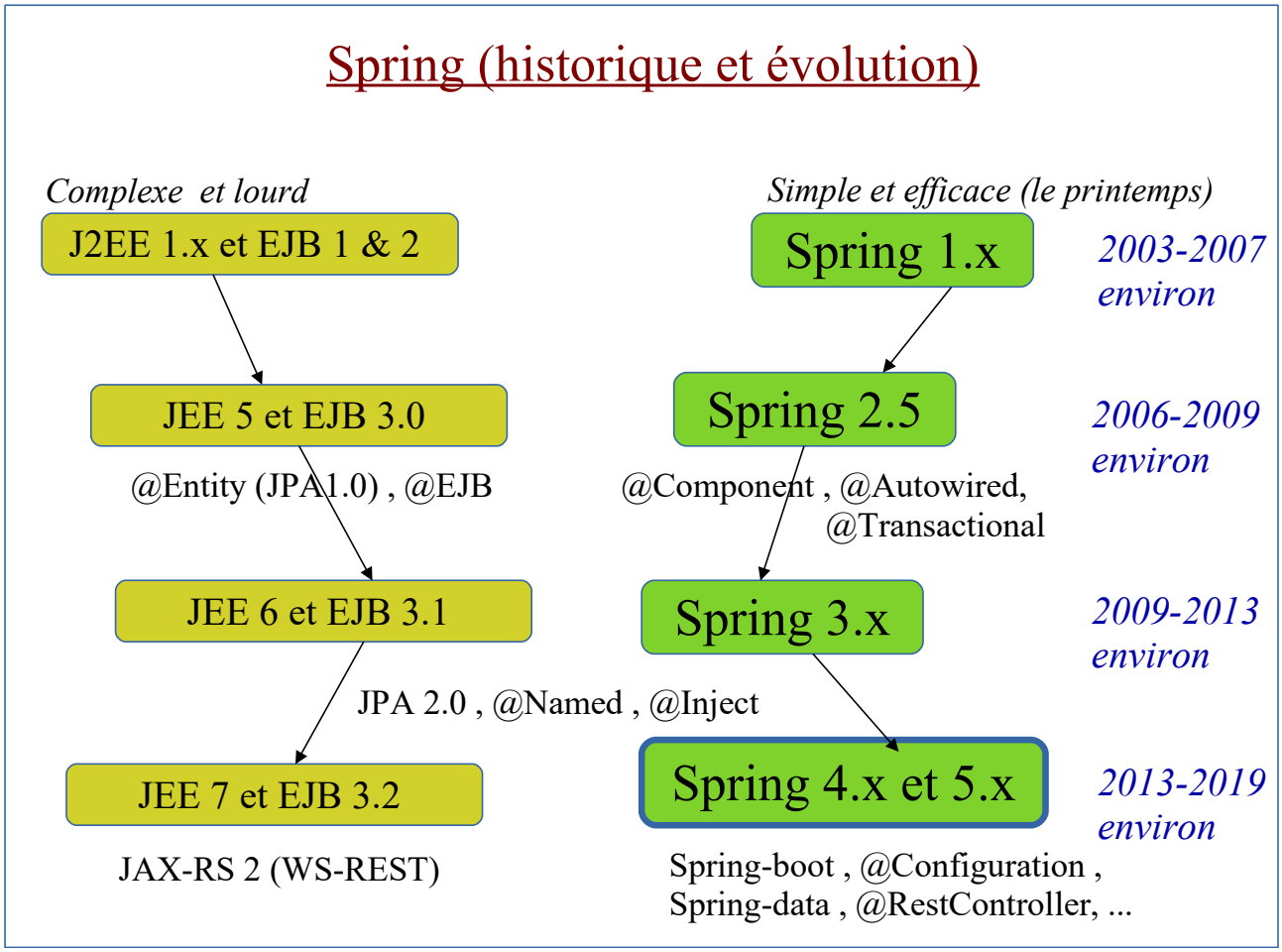
***Extensions secondaires :***

| Extensions Spring | Contenus / spécificités |
|---|---|
| Spring **Web flow** | Extension pour bien contrôler la navigation et rendre abstraite l'IHM<br>(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. Configurations Spring – vue d'ensemble

## 4.1. Historique et évolution

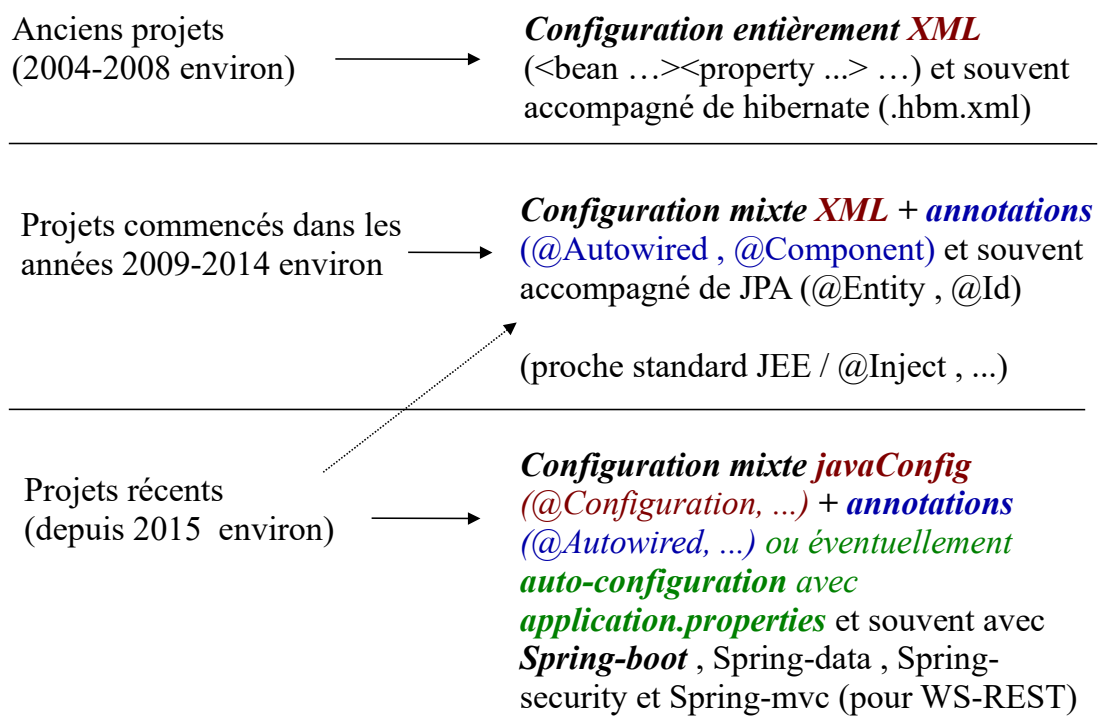
| *Versions de Spring* | *Possibilités au niveau de la configuration* |
|---|---|
| **Depuis Spring 1.x** | Configuration entièrement **XML** (avec entête DTD) **<bean >** |
| Depuis Spring 2.0 | Configuration **XML** (avec entête XSD) + .properties |
| **Depuis Spring 2.5** | **Annotations spécifiques à Spring (@Component , @Autowired, ...)** |
| Depuis Spring 3.0 | Compatibilité avec annotations DI (@Inject , @Named) |
| **Depuis Spring 4.0** | **Java Config (@Configuration , …)** et Spring boot (avec ou sans @EnableAutoConfiguration) |
| **Depuis Spring 5.0** | restructuration interne pour mieux intégrer java 8,9,10 et un début d'architecture asynchrone et réactive (Netty , WebFlux , ....) |



## 4.2. Avantages et inconvénients de chaque mode de configuration

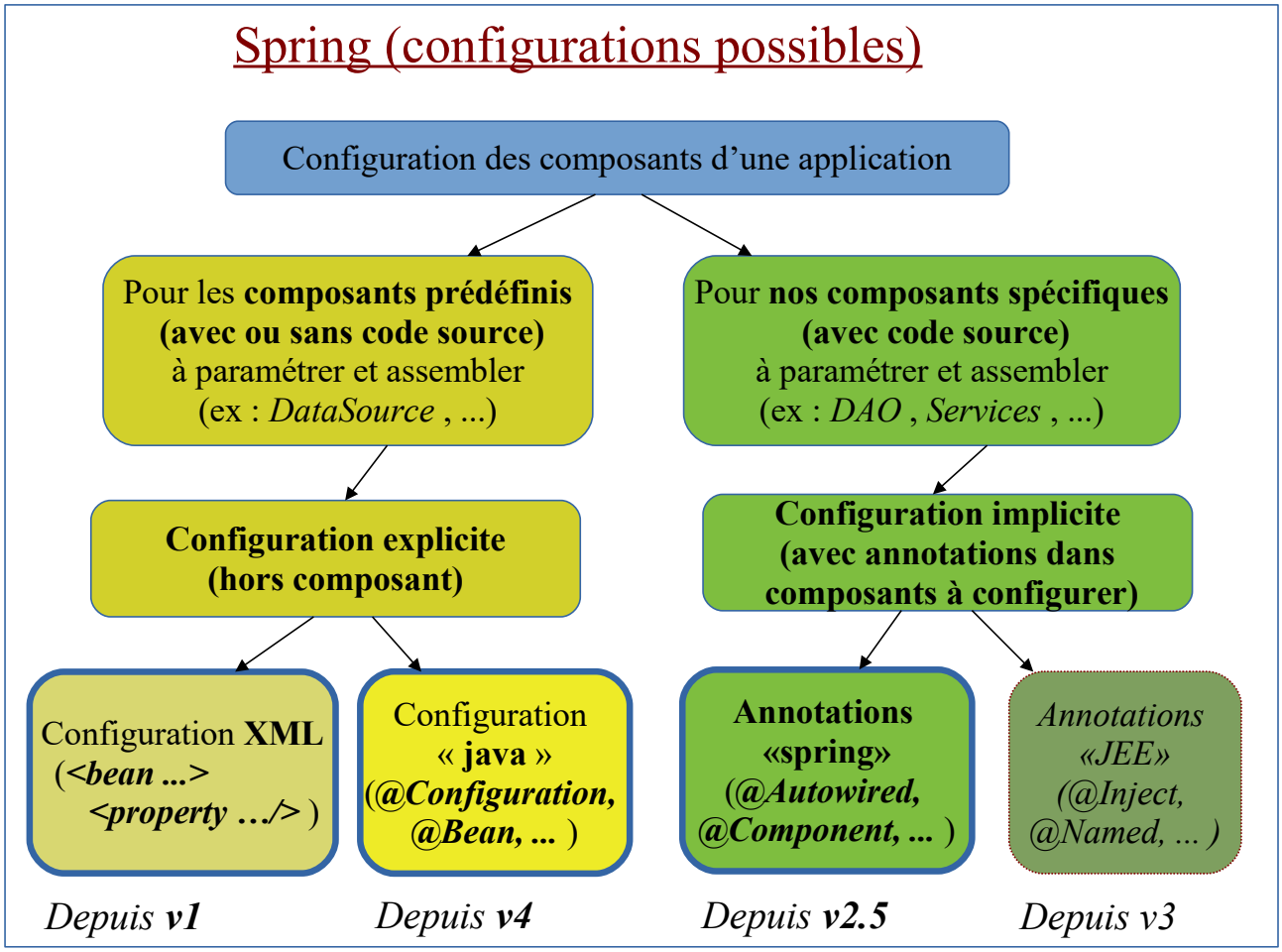
| Mode de config | Avantages | Inconvénients |
|---|---|---|
| **XML** | -Très explicite<br>- Assez centralisé tout en étant flexible (import ) .<br>- utilisation possible de fichiers annexes ".properties | -Verbeux , plus à la mode<br>- à maintenir / ajuster (si refactoring)<br>- délicat (oblige à être très rigoureux "minuscules / majuscules" , noms des packages , namespaces XML , …) |
| **Annotations** au sein des composants (@Autowired, ...) | - très rapide / efficace<br>- suffisament flexible ( component-scan selon packages , @Qualifier , …)<br>- réajustement automatique en cas de refactoring (sauf component-scan) . | - configuration dispersée dans le code de plein de composants<br>- pour nos composants seulement (avec code source) |
| **Classes de configuration "java"** (@Configuration , ...) | -Très explicite<br>- Assez centralisé tout en étant flexible (@Import ) .<br>- Auto complétion java et détection des incompatibilités (types , configurations non prévues, …)<br>- utilisation possible de fichiers annexes ".properties" pour les paramètres amenés à changer<br>- à la mode ("hype" ) | - nécessite une compilation de la configuration java (heureusement souvent automatisée par maven ou autre) |

## Spring (vue d'ensemble sur formats de configuration)




## 4.3. Complémentarité nécessaire / configuration mixte

- **Les annotations @Component , @Autowired , ….** sont très pratiques pour configurer des relations entre composants (injection de dépendances) mais elles **ne peuvent être utilisées qu'au niveau de nos propres composants** (car il faut pouvoir un contrôle total sur le code source).
- Une **configuration XML (classique) ou bien** une **configuration "java config" (moderne) permet de configurer des composants génériques** (ex : DataSource , TransactionManager , ….) **dont on ne dispose pas du code source** .
- Dans tous les cas, il est éventuellement possible de s'appuyer sur des **fichiers annexes** au format **".properties"** pour simplifier l'édition de quelques paramètres clefs succeptibles de changer (ex : url JDBC , username, password , …)


Spring (configurations possibles)
Configuration des composants d'une application
Pour les composants prédéfinis (avec ou sans code source) à paramétrer et assembler (ex : DataSource , ...)
Pour nos composants spécifiques (avec code source) à paramétrer et assembler (ex : DAO , Services , ...)
Configuration explicite (hors composant)
Configuration implicite (avec annotations dans composants à configurer)
Configuration XML (<bean ...> <property …/> )
Configuration « java » (@Configuration, @Bean, ... )
Annotations «spring» (@Autowired, @Component, ... )
Annotations «JEE» (@Inject, @Named, ... )
Depuis v1
Depuis v4
Depuis v2.5
Depuis v3

## 4.4. Démarrages possibles depuis spring 2.5

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

## 4.5. Variantes de démarrages possibles depuis spring 4

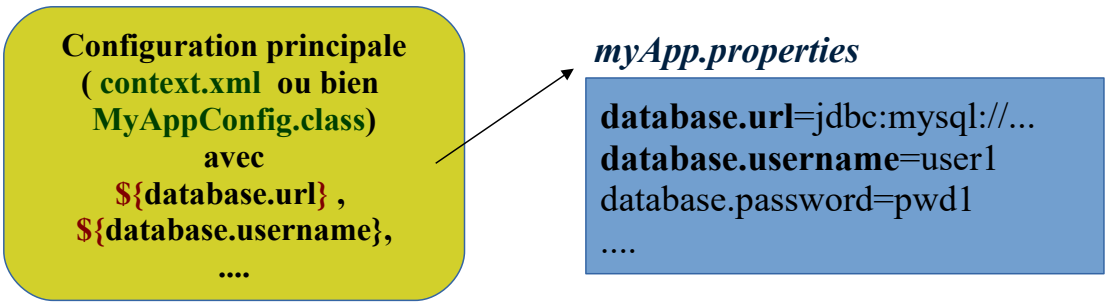
| | |
|---|---|
| Depuis méthode **main**() dans une application « standalone » | ***ApplicationContext*** ***springContext*** = **new AnnotationConfigApplicationContext(MyAppConfig.class, ConfigSupplementaire.class) ;**<br>**Cxy c = (Cxy) *springContext*.getBean("*idBeanXy*");**<br>***//ou bien c = springContext.getBean(Cxy.class);*** |
| Depuis **test unitaire** (Junit + spring-test) | **@RunWith (SpringJUnit4ClassRunner.class )**<br>**@ContextConfiguration(classes={MyAppConfig.class})**<br>**public class TestCxy {**<br>**@Autowired**<br>**private Cxy c ;** *//+ méthodes prefixées par @Test*<br>**}** |
| Depuis « listener web »<br>***(au démarrage d'une application web(.war) dans tomcat ou autre)*** | ***class MyWebApplicationInitializer implements*** **WebApplicationInitializer** {<br>public void **onStartup** (.. ***servletContext*** )...{<br>**WebApplicationContext** *context* = **new AnnotationConfigWebApplicationContext** ();<br>context.**register** (***MyWebAppConfig.class*** );<br>***servletContext*** .**addListener** (new **ContextLoaderListener** (***context*** ));<br>//... }} |

**+ tous les nouveaux démarrages possibles via spring-boot .**

## 4.6. Configuration structurée (properties , import , profiles)

### Spring (paramétrages indirects dans fichiers ".properties")

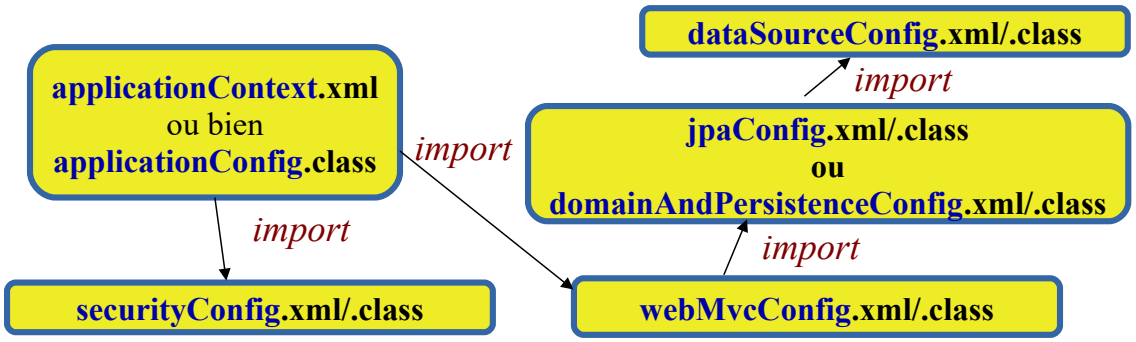
Quelque soit la version de Spring, en partant d'une configuration globale explicite ordinaire (xml/bean ou bien java/@Configuration) , il est possible de récupérer certaines valeurs variables (de paramètres clefs) dans un fichier annexe au format **".properties"**
Ceci s'effectue techniquement via "*PropertySourcesPlaceholderConfigurer*" ou un équivalent .




*configuration technique et stable (quelquefois complexe)*

*sous configuration (indirecte) potentiellement variable et facile à modifier (claire, simple) .*

## Spring (Configuration structurée via "**import**")




## Profiles (Variantes de configurations) depuis Spring4

**@Profile({"!test"})**
ou bien
**@Profile({*"jta","test"*})**
au dessus de **variantes de @Bean** dans **@Configuration**

context.getEnvironment().**setActiveProfiles**(...) ;
*ou bien*
springBootApp.**setAdditionalProfiles**(...);
*ou bien*
**@ActiveProfiles**(**profiles** = *{"test" , "jta"}*)
au dessus d'une classe de test (@RunWith, ...)

## 4.7. Spring boot et auto-configuration (depuis v4)

### Spring >= 4 (éventuelle auto-configuration)

**Configuration d'une application Spring >= 4**

**ou bien**

**Configuration**
*bien contrôlée/maîtrisée*
de manière **explicite**
*ou bien*
*configuration spécifique*
*( jta ou ... , ... )*

**@Configuration**
***précise avec plusieurs @Bean explicites***

*ou équivalent xml*

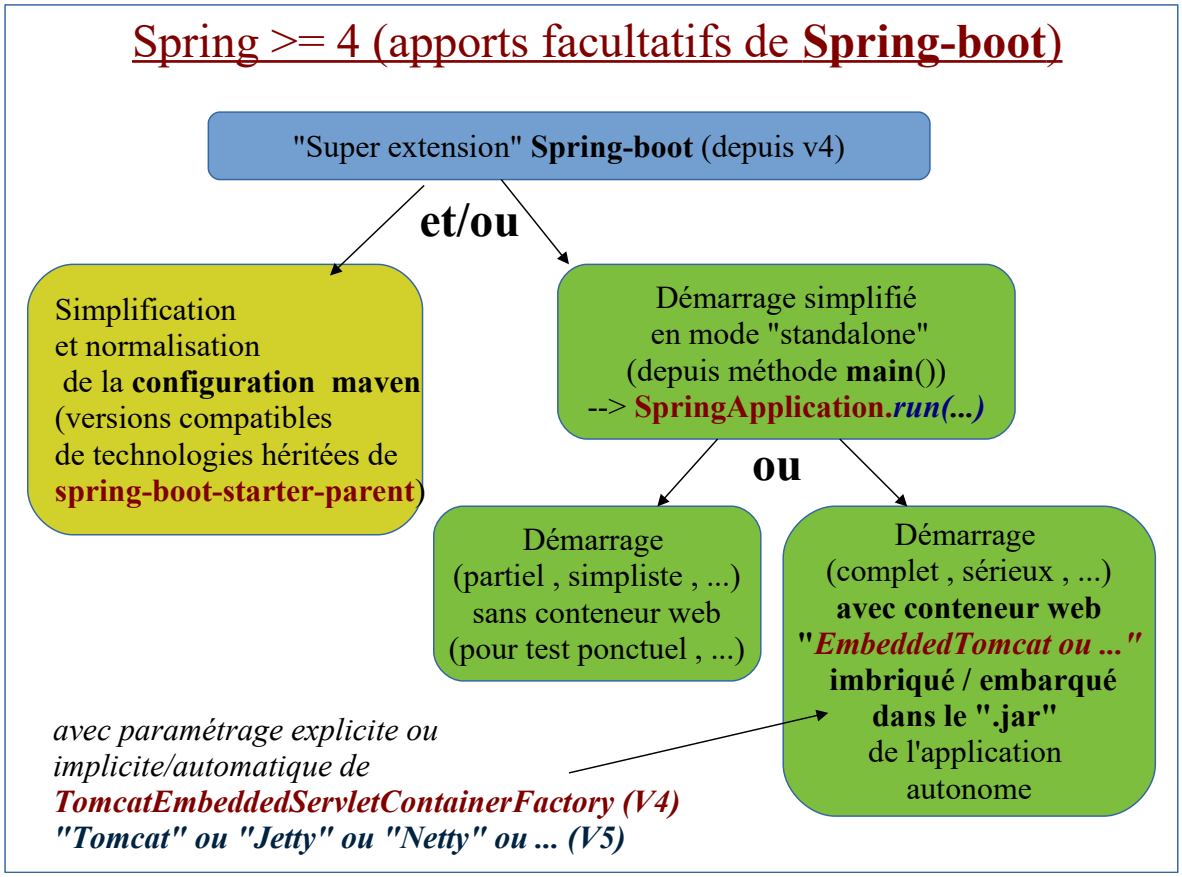
*seule possibilité Spring 2 et 3*

**Configuration automatique**
(tenant compte de
**application.properties**
et des librairies présentes
dans le classpath selon pom.xml)
*pour les applications récentes et ordinaires (pas trop spécifiques)*

**@Configuration**
***avec***
***@EnableAutoConfiguration***

***application.properties*** +

**spring.datasource.url**=*jdbc:mysql://localhost:dbXy*
**spring.datasource.username**=*root*
**spring.datasource.password**=*root*




**Exemple de démarrage avec Spring-Boot**

```
package tp;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

//NB: @SpringBootApplication est un équivalent
//    de @Configuration + @EnableAutoConfiguration + @ComponentScan/current package

@SpringBootApplication
public class MySpringBootApplication extends SpringBootServletInitializer {

   public static void main(String[] args) {
      SpringApplication.run(MySpringBootApplication.class, args);
      System.out.println("http://localhost:8080/myMvcSpringBootApp");
   }
}
```

==> la partie *@EnableAutoConfiguration* de **@SpringBootApplication** fait que le fichier **application.properties** sera automatiquement analysé .

==> **il faut absolument que les classes de tests et de configuration** *(ex : tp.config.WebSecurityConfig extends WebSecurityConfigurerAdapter)* **soient placées dans des sous-packages** car le *@ComponentScan* de **@SpringBootApplication** est par défaut configuré pour n'analyser que le package courant (ici *tp*) et ses sous packages .

```
package tp.test;

import org.junit.Assert;   import org.junit.Test;  import org.junit.runner.RunWith;
import org.slf4j.Logger; import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import tp.MySpringBootApplication;

@RunWith(SpringRunner.class)
@SpringBootTest(classes= {MySpringBootApplication.class})
public class TestServiceXy {
	private static Logger logger = LoggerFactory.getLogger(TestServiceXy.class);

	@Autowired
	private ServiceXy service ; // service métier à tester

	@Test
	 public void testQuiVaBien() {
		logger.debug("testQuiVaBien");
		Assert.assertTrue(1+1==2);
	 }
}
```

# II - Configurations ioc (xml , java , annotations)

## 1. Configuration xml de Spring

### 1.1. Fichier(s) de configuration

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

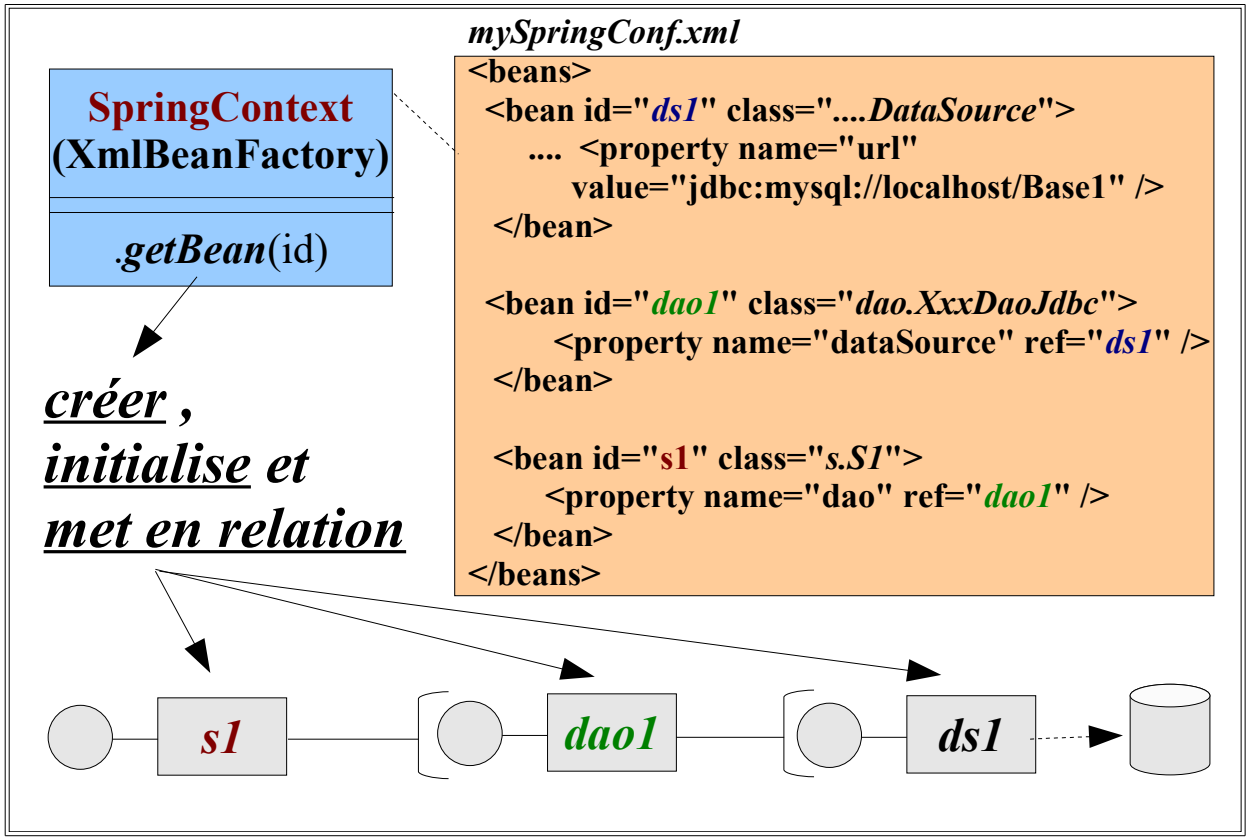
En fonction des réels besoins de l'application, l'entête du fichier de configuration Spring pourra comporter (ou pas) tout un tas d'éléments optionnels (AOP , Transactions , ...).

**Exemple d'entête**:

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

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

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




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

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

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

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

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

## 1.3. Instanciation de composant Spring via une Fabrique

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

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

Ceci pourrait constituer le point de départ d'une petite classe de test élémentaire.
Néanmoins, dans beaucoup de cas on préférera utiliser "ApplicationContext" qui est une version améliorée/sophistiquée de "BeanFactory" .

## 1.4. ApplicationContext et test unitaires

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

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

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

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

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

Attention (pour les performances):

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

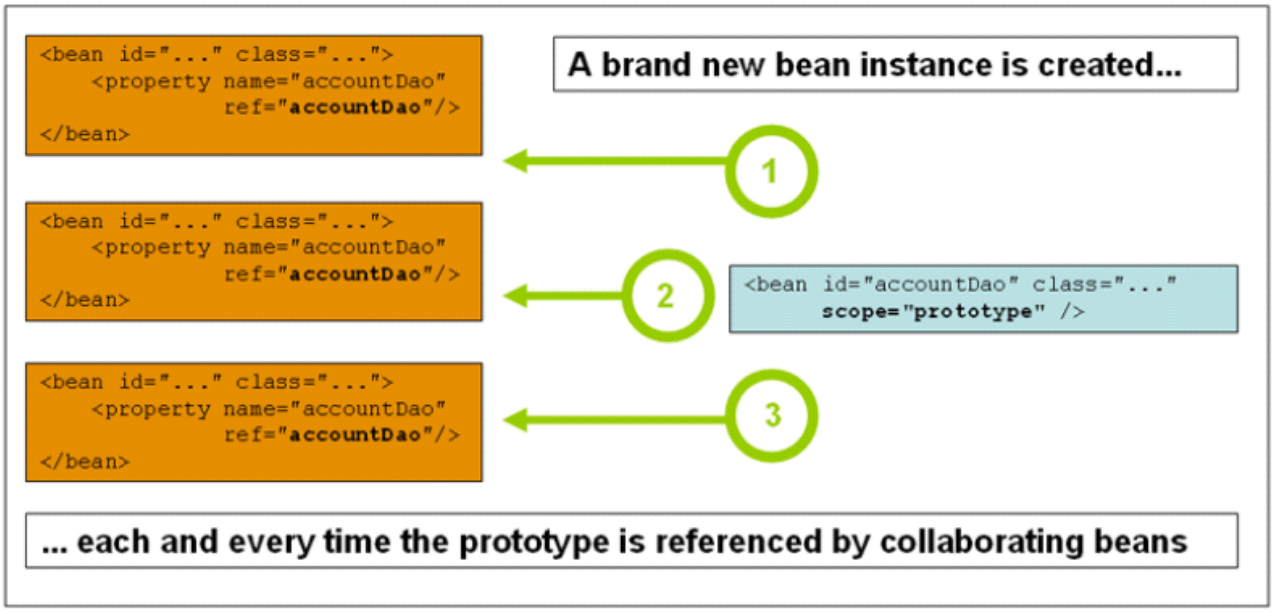
- analyse de toute la configuration Xml
- déclenchement des mécanismes AOP dynamiques
- instanciations des composants
- assemblage par injections de dépendances

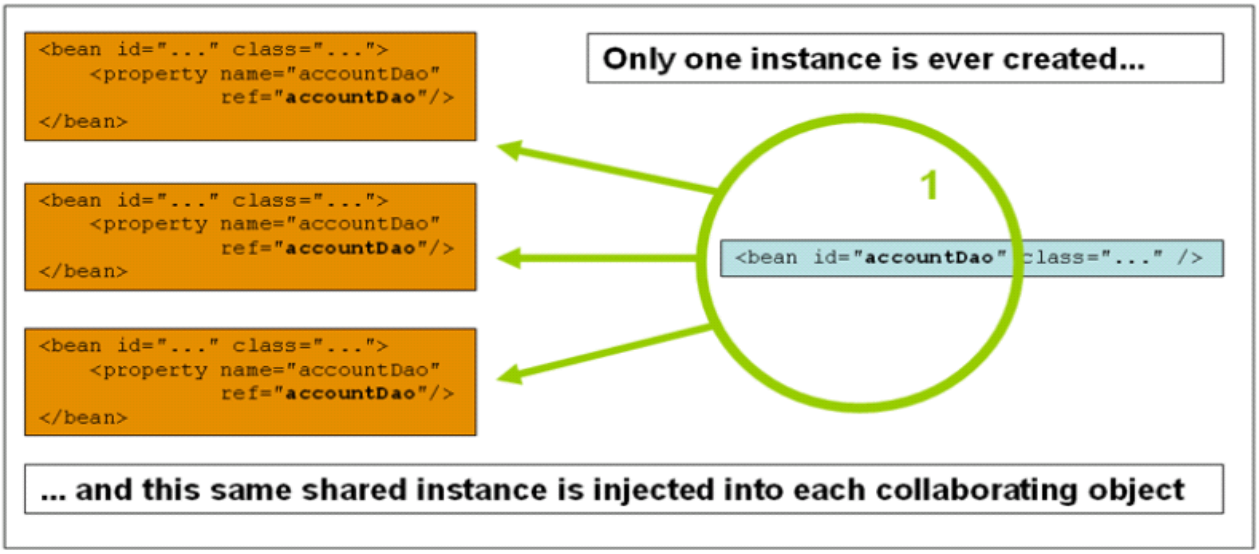
Si plusieurs Tests unitaires (ex: JUnit) doivent être lancés dans la foulée , il faudra veiller à ne pas recréer inutilement un nouveau contexte Spring à chaque fois.

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

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

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



## 1.6. Organisation des fichiers de configurations "Spring"

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

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

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

Exemples :

*applicationContext.xml*

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

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

Rappels :

Spring n'impose pas de nom sur le fichier de configuration principal (celui-ci est simplement référencé par une classe de test ou bien web.xml ).
Ceci dit, les noms les plus classiques sont "***beans.xml***" , "***applicationContext.xml***" , "***context.xml***" .

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

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

**database.properties**

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

**dataSourceSpringConf.xml**

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

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

# 2. Configuration IOC Spring via des annotations

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

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

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

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

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

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).

## 2.1. 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 @Component, .... -->

  <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>
```

## 2.2. 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, ...) |

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

| | |
|---|---|
| **@Required**<br>(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) |

**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);
      }
```

NB :

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

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

**private Payment paiementParCarteDeCredit ;**

**ou bien**

**@Autowired @Qualifier("byCash")**

**private Payment paiementEnLiquide ;**

permettra d'effectuer une injection de la version voulue .

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

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

En organisant bien les packages java de la façon suivante :

**xxx.itf.dao.*DaoXY*** (interface)

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

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

on peut ensuite paramétrer alternativement 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.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">
  <context:annotation-config/> <!-- pour demander à Spring de tenir compte de @ .... -->

  <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

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

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

# 3. Tests "Junit4 + Spring"

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

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

NB :
Les classes de Test annotées via **@RunWith(SpringJUnit4ClassRunner.class)** peuvent utiliser en interne @Autowired ou @Inject même si elles ne sont pas placées dans un package référencé par
`<context:component-scan base-package="...."/>`

**Exemple de classe de Test de Service (avec annotations)**

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

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

@Autowired
private GestionComptes service = null;

 @Test
public void 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
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.

# 4. Paramétrages Spring  quelquefois utiles

## 4.1. Compatibilité avec singleton déjà programmé en java

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

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

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

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

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

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

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

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

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

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

```
}
```

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

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

### 4.3.b. Via configuration 100% xml

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

## 4.4. Autres possibilités de Spring

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

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

# 5. 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.

Premiers avantages d'une configuration explicite java (par rapport à une configuration xml) :

- Auto complétion java et détection des incompatibilités (types , configurations non prévues, …)
- Héritage possible entre classes de configuration (générique, spécifique, ...)

NB : Les exemples de configuration de ce chapitre ne sont à considérer que comme des exemples de configurations possibles (à adapter en fonction du contexte) !!!

## 5.1. 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;
	}
}
```

## 5.2. 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;
	}
}
```

## 5.3. Quelques paramétrages (avancés) possibles :

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

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

## 5.4. 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  @Component , @Controller , ... for @Autowired, @Inject ,...
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= { "entityManagerFactory", "myEmf" , "otherAliasEmf"} )
	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;
        }
}
```

## 5.5. @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* .**

## 5.6. 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
```

## 5.7. Profiles "spring" (variante de configuration)

Variantes :

**@Profile({"!test"})**

ou bien

**@Profile({*"jta","test"*})**

au dessus de **variantes de @Bean** dans **@Configuration**

Sélection du ou des profile(s) à activer

context.getEnvironment().**setActiveProfiles**(...) ;

*ou bien*

springBootApp.**setAdditionalProfiles**(...);

ou bien

**@ActiveProfiles(profiles = *{"test" , "jta"}*)** au dessus d'une classe de test (@RunWith, ...)

## 5.8. Configuration conditionnelle intelligente

*Annotations que l'on peut ajouter à coté de @Bean :*

**@ConditionalOnBean**(name="otherBeanNameThatMustExist")

@ConditionalOnMissingBean
//pour configurer une nouvelle instance de Bean (specific type , specific name via methodName) **que si ce bean n'existe pas encore**

**@ConditionalOnClass**(**name="**com.sample.Dummy**"**)
//pour configurer un Bean **que si une classe est trouvée dans le classpath**

**@ConditionalOnMissingClass**(value={"com.sample.Dummy"})
//pour configurer un Bean **que si une classe n'est pas trouvée dans le classpath**

@ConditionalOnWebApplication et @ConditionalOnNotWebApplication

**@ConditionalOnResource**(**resources**={"**classpath:***application.properties*"})
//pour configurer un Bean **que si une ressource est trouvée dans le classpath**

@ConditionalOnResource(resources={"file:///e:/doc/data.txt"})

**@ConditionalOnJava**(value=JavaVersion.SEVEN,range=Range.OLDER_THAN)
//pour configurer un Bean **que si version de java** < 7

**@ConditionalOnProperty**(name="test.property1", havingValue="A")
//pour configurer un Bean **que si la propriété** *"test.property1"* **existe dans l'environnement et vaut "A"**

**@ConditionalOnProperty**(name="test.property2")
//pour configurer un Bean **que si la propriété** *"test.property2"* **existe dans l'environnement et est différente de "false"**

**@ConditionalOnJndi**(value={"jndiName1" , "jndiName2"})
//pour configurer un Bean **que au moins un nom logique JNDI est trouvé dans InitialContext**

...

C'est ce genre de configuration automatique et intelligente qui est automatiquement activée (de manière implicite) en mode ....

# III - Spring-boot

## 1. 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.

Attention (versions):

- Spring-boot 1.x compatible avec Spring 4.x
- Spring-boot 2.x compatible avec Spring 5.x (et utilisant beaucoup les nouveautés de java >=8) .

--> quelques différences (assez significatives) entre Spring-boot 1 et 2 .

## 1.1. Configuration "maven" pour spring-boot 1.x (et spring 4)

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 !!!!

## 1.2. Configuration maven pour spring-boot 2 (et spring 5)

```
...
    <parent>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-parent</artifactId>
        <version>2.0.5.RELEASE</version>
        <relativePath/> <!-- lookup parent from repository -->
    </parent>
    <properties>
        <packaging.type>jar</packaging.type>
        <java.version>1.8</java.version>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</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-test</artifactId>
                <scope>test</scope>
            </dependency>
            <!-- spring-boot-devtools useful for refresh without restarting -->
            <dependency>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-devtools</artifactId>
                <scope>runtime</scope>
            </dependency>
            <dependency>
                <groupId>org.hibernate</groupId>
                <artifactId>hibernate-core</artifactId>
                  <!-- with hibernate-entitymanager inside -->
                <!-- version conseillee par spring-boot-starter-parent -->
            </dependency>
            <!-- pour vues de type ".jsp" avec eventuellement jstl -->
             <dependency>
                <groupId>org.apache.tomcat.embed</groupId>
                 <artifactId>tomcat-embed-jasper</artifactId>
            </dependency>
            <dependency>
                <groupId>javax.servlet</groupId>
                <artifactId>jstl</artifactId>
            </dependency>
        </dependencies>

        <build>
            <finalName>${project.artifactId}</finalName>
        </build>
...
```

## 1.3. Boot (standalone) sans annotation

```
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);
```

Sans l'annotation @SpringBootApplication sur la classe de démarrage , la configuration (@ComponentScan , ...) doit être explicitée sur la classe de configuration passée en argument du constructeur (ou bien de la méthode run()).

## 1.4. Boot (standalone) avec annotation @SpringBootApplication

**Exemple de démarrage avec @SpringBootApplication**

```
package tp;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.boot.web.servlet.support.SpringBootServletInitializer;

//NB: @SpringBootApplication est un équivalent
//    de @Configuration + @EnableAutoConfiguration + @ComponentScan/current package

@SpringBootApplication
public class MySpringBootApplication extends SpringBootServletInitializer {

   public static void main(String[] args) {
      SpringApplication.run(MySpringBootApplication.class, args);
      System.out.println("http://localhost:8080/myMvcSpringBootApp");
   }
}
```

==> la partie *@EnableAutoConfiguration* de **@SpringBootApplication** fait que le fichier **application.properties** sera automatiquement analysé .

==> **il faut absolument que les classes de tests et de configuration** *(ex : tp.config.WebSecurityConfig extends WebSecurityConfigurerAdapter)* **soient placées dans des sous-packages** car le *@ComponentScan* de **@SpringBootApplication** est par défaut configuré pour n'analyser que le package courant (ici *tp*) et ses sous packages .

NB : avec spring-boot et un packaging "jar" (et pas "war") , le répertoire src/main/webapp n'existe pas et il faut alors placer les ressources web **(.html , .css , ...)** dans le sous répertoire "**static**" (à éventuellement créer) de **src/main/resources** .

## 1.5. Tests unitaires avec Spring-boot

*éventuellement* :

```
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 {
…
}
```

***ou mieux encore*** :

```
package tp.test;

import org.junit.Assert;   import org.junit.Test;  import org.junit.runner.RunWith;
import org.slf4j.Logger; import org.slf4j.LoggerFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.context.junit4.SpringRunner;
import tp.MySpringBootApplication;

@RunWith(SpringRunner.class)
@SpringBootTest(classes= {MySpringBootApplication.class})
public class TestServiceXy {
        private static Logger logger = LoggerFactory.getLogger(TestServiceXy.class);

        @Autowired
        private ServiceXy service ; // service métier à tester

        @Test
         public void testQuiVaBien() {
                logger.debug("testQuiVaBien");
                Assert.assertTrue(1+1==2);
         }
}
```

## 1.6. 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 automatiquement l'application en fonction des bibliothèques trouvées dans son class-path (indirectement défini via le contenu de pom.xml) et en fonction de application.properties .**

**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, … ) Pour les autres bases (mysql, mariadb, postgres, oracle, db2, ...) une configuration complémentaire est nécessaire dans application.properties .
- 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 -->
```

## 1.7. Auto-configuration "spring-boot" avec application.properties

Rappel : l'annotation ***@SpringBootApplication*** *(placée sur la classe de démarrage )*

est un équivalent de

@Configuration + ***@EnableAutoConfiguration*** + @ComponentScan/current package

Dans certains cas (classiques, simples), la configuration de l'application spring-boot peut entièrement être placée dans le fichier **application.properties** (de scr/main/resources) .

Le fichier **application.properties** est implicitement analysé en mode ***@EnableAutoConfiguration*** et peut comporter tous un tas de propriétés (dont les noms sont normalisés dans la documentation de référence de spring) .

Beaucoup de propriétés de **application.properties** peuvent considérées comme une alternative hyper simplifiée d'un énorme paquet de configuration explicite (xml ou java) qui était auparavant placé dans une multitude de fichiers complémentaires (ex : WEB-INF/web.xml , META-INF/persistence.xml , ... ou XyJavaConfig.class ) .

Exemple de fichier **application.properties**

```
server.servlet.context-path=/myMvcSpringBootApp
server.port=8080
logging.level.org=INFO

spring.mvc.view.prefix=/views/
spring.mvc.view.suffix=.jsp

#spring.datasource.driverClassName=com.mysql.jdbc.Driver
#spring.datasource.url=jdbc:mysql://localhost:3306/mydb
#spring.datasource.username=root
#spring.datasource.password=

spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:~/mydb
spring.datasource.username=sa
spring.datasource.password=
spring.datasource.platform=h2
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
```

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

## 1.8. Paramétrages du conteneur web "EmdeddedTomcat" de Spring-boot 1.x et Spring 4

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;
}

}
```

## 1.9. 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
```

## 1.10. Paramétrages "java" explicites 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!";
            }
}
```

## 1.11. Spring-boot 1.x (spring 4.x) + JSF 2.x

```
@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>
*/
}
```

# IV - Spring backend (Services, Dao , Datasource)

...

## 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;
}
```

# 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 "JNDI" d'une source de données JDBC

NB :

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

***mySpringDataSource_JNDI.xml***

```
...<beans ...>
<!-- DataSource nécessitant une config. coté servApp (Pool de connexions avec nom JNDI) -->
        <!-- JNDI=java:/BankDBDataSource pour JBoss -->
        <!-- JNDI=java:comp/env/jdbc/BankDBDataSource pour Tomcat -->
  <bean id="myDataSource"
       class="org.springframework.jndi.JndiObjectFactoryBean"
       destroy-method="close">
         <property name="jndiName" value="java:/BankDBDataSource" />
         <property name="expectedType" value="javax.sql.DataSource" />
  </bean>
</beans>
```

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

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

*Exemple de configuration sous Tomcat >= 5.5*:

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

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

```
        url="jdbc:mysql://localhost/minibankdb"/>
</Context>
```

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

***mySpringDataSource_Simple.xml***

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

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

**Remarques**:

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

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

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

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

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

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

## 2.3. Embedded DataSource with pool

De façon à avoir de meilleurs performances en mode "production" , on pourra utiliser des implémentations plus sophistiquées d'un dataSource jdbc embarqué dans l'application (spring-boot ou autre) :

La classe "***org.apache.commons.dbcp.BasicDataSource***" (de la librairie "**common-dbcp**" de la communauté Apache) correspond à une technologie que l'on peut intégrer facilement un peu partout (dans une application autonome , dans une application web (.war) , ....).

La technologie alternative **c3p0** (souvent utilisée avec hibernate) est également une bonne mise en oeuvre de "embedded jdbc dataSource with pool" .

*...futur exemple ici (en mode java-config) ...*

...

# V - JPA , EntityManager (config. Spring)

# 1. DAO Spring basé sur JPA (Java Persistence Api)

## 1.1. 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;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;

@Entity
//@Table(name="Compte")
public class Compte {
	@Id
	@GeneratedValue(strategy=GenerationType.IDENTITY)
	private long numCpt;

	@Column(length=32)
	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; }
}
```

## 1.2. unité de persistance (persistence.xml facultatif)

**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_2_0.xsd" version="2.0" >
<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>
```

NB : La configuration "Jpa" d'une application spring peut :

- soit être partiellement configurée dans META-INF/persistence.xml et partiellement configurée en mode "spring" (xml ou bien java config)
- soit être entièrement configurée en mode spring (xml , java config) et dans ce cas **le fichier META-INF/persistence.xml peut ne pas exister (il n'est pas absolument nécessaire)**.

## 1.3. Configuration "spring / jpa" classique (en version xml) :

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>
```

## 1.4. TxManager compatible JPA et @PersistenceContext

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

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

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.

## 1.5. Configuration Jpa / Spring (sans xml) en mode java-config

La configuration suivante est équivalente aux configurations xml des paragraphes précédents.

```
package tp.myapp.minibank.impl.config;
import javax.persistence.EntityManagerFactory;
import javax.sql.DataSource; import java.util.Properties ;
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","org.mycontrib.generic"})
  // for interpretation of  @Component , @Controller , ... for @Autowired, @Inject ,...
public class JpaConfig {

	// 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="entityManagerFactory" )
	public EntityManagerFactory entityManagerFactory(
			JpaVendorAdapter jpaVendorAdapter, DataSource dataSource) {
		LocalContainerEntityManagerFactoryBean factory
                                = new LocalContainerEntityManagerFactoryBean();
```

```
                factory.setJpaVendorAdapter(jpaVendorAdapter);
                factory.setPackagesToScan("tp.myapp.minibank.persistence.entity");
                factory.setDataSource(dataSource);

                Properties jpaProperties = new Properties() ; //java.util

                jpaProperties.setProperty("javax.persistence.schema-generation.database.action",
                                                "drop-and-create") ; //à partir de JPA 2.1

                factory.setJpaProperties(jpaProperties) ;
                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;
        }
}
```

NB : La configuration ci dessus n'a pas besoin de META-INF/persistence.xml

## 1.6. Enorme simplification avec "Spring-boot" , "spring-data" et @EnableAutoConfiguration

Toute la **configuration** (xml ou bien "java config explicite") des paragraphes précédents peut éventuellement être **considérée comme "prédéfinie" lorsque l'on utilise "spring-boot" en mode @EnableAutoConfiguration**.

Les seuls petits paramétrages nécessaires (url , packages à scanner, ...)

peuvent être placés dans le fichier **application.properties** ( --> voir chapitre "spring-boot").

D'autre part, l'extension facultative (mais très intéressante) "**spring-data**" permet de simplifier énormément le code des "DAO : Data Access Object" ( --> voir chapitre "spring-data") .

Paramétrages autour de JPA dans **application.properties** (@EnableAutoConiguration) :

```
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create
```

## 1.7. 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
@Component //ou @Repository
public class CompteDaoJpa implements CompteDao {

    @PersistenceContext()
    private 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 - Spring-Data (avec JPA , ...)

## 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 (findByPrimaryKey , 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). |
| `Optional<T> findById(ID primaryKey);` | Recherche par clef primaire (avec jdk >= 1.8) |
| `Iterable<T> findAll();` | Recherche toutes les entités (du type courant/considéré) |
| `Long count();` | Retourne le nombre d'entités existantes |
| `void delete(T entity);`<br><br>`void deleteById(ID primaryKey);` | Supprime une (ou plusieurs) entités |

| | |
|---|---|
| `void deleteAll();` | |
| `boolean exists(ID primaryKey);` | Test l'existence d'une entité |

NB : principal changement entre "spring-data pour spring 4" et "spring-data pour spring 5" :

Le T **findOne**(ID primaryKey); compatible spring-4 retournait auparavant une entité persistante recherchée via sa clef primaire et retournait **null** si rien n'était trouvé .

Depuis la version de Spring-data compatible Spring 5 ,

la sémantique de **Optional**<T> **findOne**(T exempleEntity) consiste à retourner une éventuelle entité ayant les mêmes valeurs non-nulles que l'entité exemple passée en paramètre.

**Optional**<T> **findById**(ID primaryKey) retourne maintenant une éventuelle entité persistante trouvée (nulle ou pas) dans un objet enveloppe **Optional**<T> qui lui n'est jamais nul .

Le service métier appelant pourra appeler la méthode .get() de Optional<T> de manière à récupérer un accès à l'entité persistante remontée :

```
public Compte rechercherCompte(long num) {
                return daoCompte.findById(num).get();     }
```

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 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

***Dépendance maven*** :

```
<dependencies>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
  </dependency>
<dependencies>
```

Exemple de version : **1.12.4.RELEASE** (pour spring 4) , **2.0.10.RELEASE** (pour spring 5)

**Activation en 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:jpa="http://www.springframework.org/schema/data/jpa"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
    http://www.springframework.org/schema/data/jpa
    http://www.springframework.org/schema/data/jpa/spring-jpa.xsd">

  <jpa:repositories   base-package="com.acme.repositories" />

</beans>
```

**Activation en java-config explicite** :

```
import org.springframework.data.jpa.repository.config.EnableJpaRepositories;

@EnableJpaRepositories
...
class Config {}
```

**Activation via application.properties (autoConfiguration)** :

```
spring.data.jpa.repositories.enabled=true
```

Exemple d'interface de DAO/Jpa avec **JpaRepository (héritant lui même de CrudRepository)**:

```
interface UserRepository extends CrudRepository<User, Long> {
  List<User> findByLastname(String lastname);
}
```

**La classe d'implémentation sera générée automatiquement** (si *@EnableJpaRepositories* ou si *<jpa:repositories base-package="..." />* )

il suffit d'une injection via **@Autowired** ou **@Inject** pour accéder au composant DAO généré .

**Conventions de noms sur les méthodes de l'interface** :

`find`…By, `read`…By, `query`…By, `get`…By and `count`…By,

Exemples :

```
List<User> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);

// Enables the distinct flag for the query
List<User> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);
List<User> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);

// Enabling ignoring case for an individual property
List<User> findByLastnameIgnoreCase(String lastname);

// Enabling ignoring case for all suitable properties
List<User> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);

// Enabling static ORDER BY for a query
List<User> findByLastnameOrderByFirstnameAsc(String lastname);
List<User> findByLastnameOrderByFirstnameDesc(String lastname);
```

***methodNameWithKeyWords(?1,$2,...)***

| Keyword | Sample | JPQL snippet |
|---|---|---|
| **And** | findByLastname**And**Firstname | … where x.lastname = ?1 **and** x.firstname = ?2 |
| **Or** | findByLastname**Or**Firstname | … where x.lastname = ?1 **or** x.firstname = ?2 |
| **Is,Equals** | FindByFirstname,<br>findByFirstname**Is**, | … where x.firstname **=** ?1 |

| Keyword | Sample | JPQL snippet |
|---|---|---|
| | findByFirstname**Equals** | |
| **Between** | findByStartDate**Between** | … where x.startDate **between** ?1 **and** ?2 |
| **LessThan** | findByAge**LessThan** | … where x.age **<** ?1 |
| **LessThanEqual** | findByAge**LessThanEqual** | … where x.age **<=** ?1 |
| **GreaterThan** | findByAge**GreaterThan** | … where x.age **>** ?1 |
| **GreaterThanEqual** | findByAge**GreaterThanEqual** | … where x.age **>=** ?1 |
| **After** | findByStartDate**After** | … where x.startDate **>** ?1 |
| **Before** | findByStartDate**Before** | … where x.startDate **<** ?1 |
| **IsNull** | findByAge**IsNull** | … where x.age **is null** |
| **IsNotNull, NotNull** | findByAge(Is)**NotNull** | … where x.age **not null** |
| **Like** | findByFirstname**Like** | … where x.firstname **like** ?1 |
| **NotLike** | findByFirstname**NotLike** | … where x.firstname **not like** ?1 |
| **StartingWith** | findByFirstname**StartingWith** | … where x.firstname like ?1 (parameter bound with appended %) |
| **EndingWith** | findByFirstname**EndingWith** | … where x.firstname like ?1 (parameter bound with prepended %) |
| **Containing** | findByFirstname**Containing** | … where x.firstname like ?1 (parameter bound wrapped in %) |
| **OrderBy** | findByAge**OrderBy**Lastname**Desc** | … where x.age = ?1 **order by** x.lastname **desc** |
| **Not** | findByLastname**Not** | … where x.lastname **<>** ?1 |
| **In** | findByAge**In**(Collection<Age> ages) | … where x.age **in** ?1 |
| **NotIn** | findByAge**NotIn**(Collection<Age> | … where x.age **not** |

| Keyword | Sample | JPQL snippet |
|---|---|---|
| | age) | **in** ?1 |
| **True** | findByActive**True**() | … where x.active **= true** |
| **False** | findByActive**False**() | … where x.active **= false** |
| **IgnoreCase** | findByFirstname**IgnoreCase** | … where UPPER(x.firstame) = UPPER(?1) |

Paramétrage par défaut de JpaRepositories :

**CREATE_IF_NOT_FOUND** (default) combines CREATE and USE_DECLARED_QUERY

→ **on peut donc éventuellement personnaliser l'implémentation des méthodes.**

**Utilisation de @NamedQuery à coté de @Entity (ou <named-query ...> dans orm.xml )** :

Dans orm.xml (référencé par META-INF/persistence.xml ou …) :

```
<named-query name="User.findByLastname">
  <query>select u from User u where u.lastname = ?1</query>
</named-query>
```

et/ou dans la classe d'entité persistante :

```
@Entity
@NamedQuery(name = "User.findByEmailAddress",
  query = "select u from User u where u.emailAddress = ?1")
public class User {

}
```

```
public interface UserRepository extends JpaRepository<User, Long>
{
  List<User> findByLastname(String lastname);

  User findByEmailAddress(String emailAddress);
}
```

**Utilisation (un peu radicale) de @Query (de Spring Data) dans l'interface** :

**Sémantiquement peu être un trop peu radical pour une interface !!!**

Exemple :

```
public interface UserRepository extends JpaRepository<User, Long> {
```

```
  @Query("select u from User u where u.emailAddress = ?1")
  User findByEmailAddress(String emailAddress);
}
```

==> et encore beaucoup d'autres possibilités / options dans la **doc de référence de spring-data** .

## 1.3. Spring-data-mongo

## 1.4. Spring-data-Neo4J

# VII - Essentiel 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 MyPerfLogAspect {

     @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.MyPerfLogAspect">
     </bean> ....
```

ou bien **@Enable...AutoProxy** sur une classe de **@Configuration** en mode java-config .

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

```
...
<bean id="myLogAspectBean"  class="tp...MyPerfLogAspect"></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>
```

# VIII - Transactions "Spring"

## 1. Support des transactions au niveau de Spring

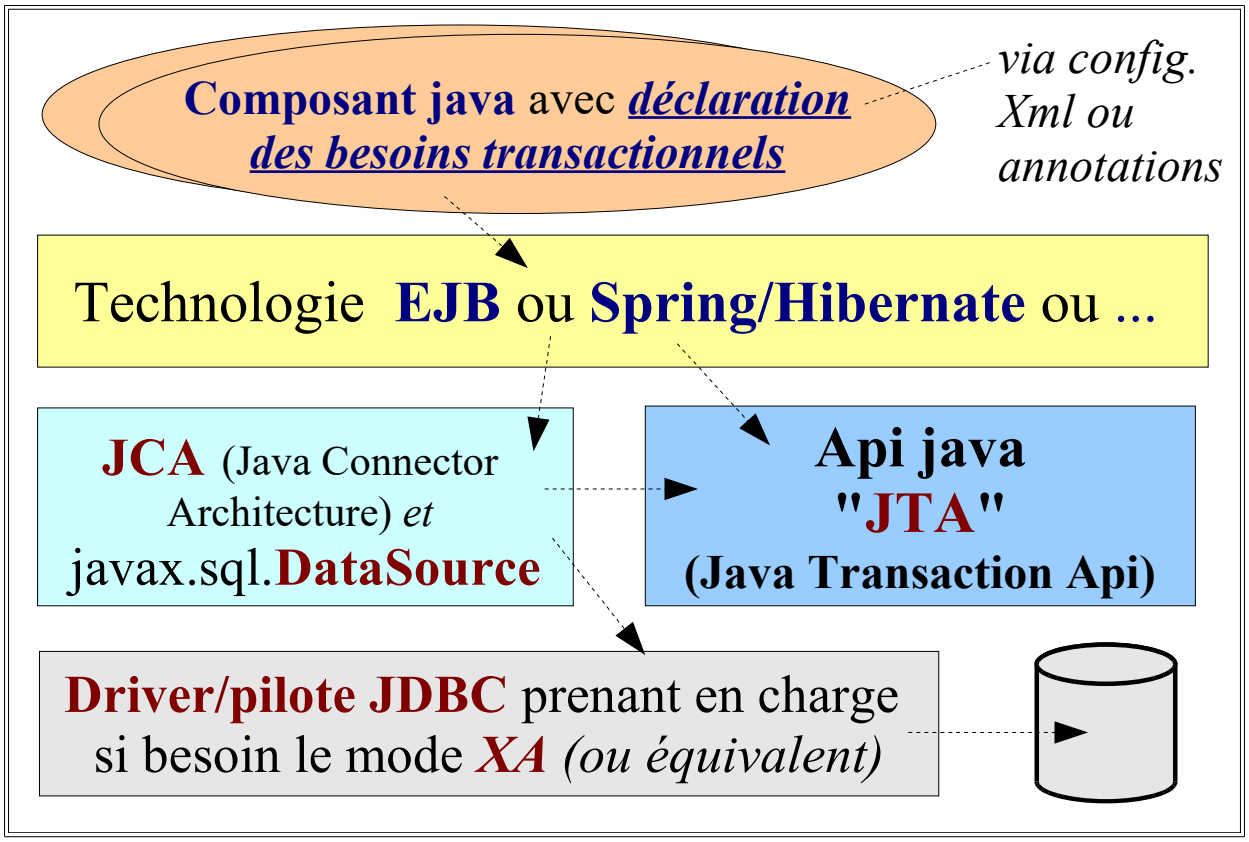
Le framework Spring est capable de gérer (superviser) lui même les transactions devant être menées à bien à partir de certains services applicatifs.

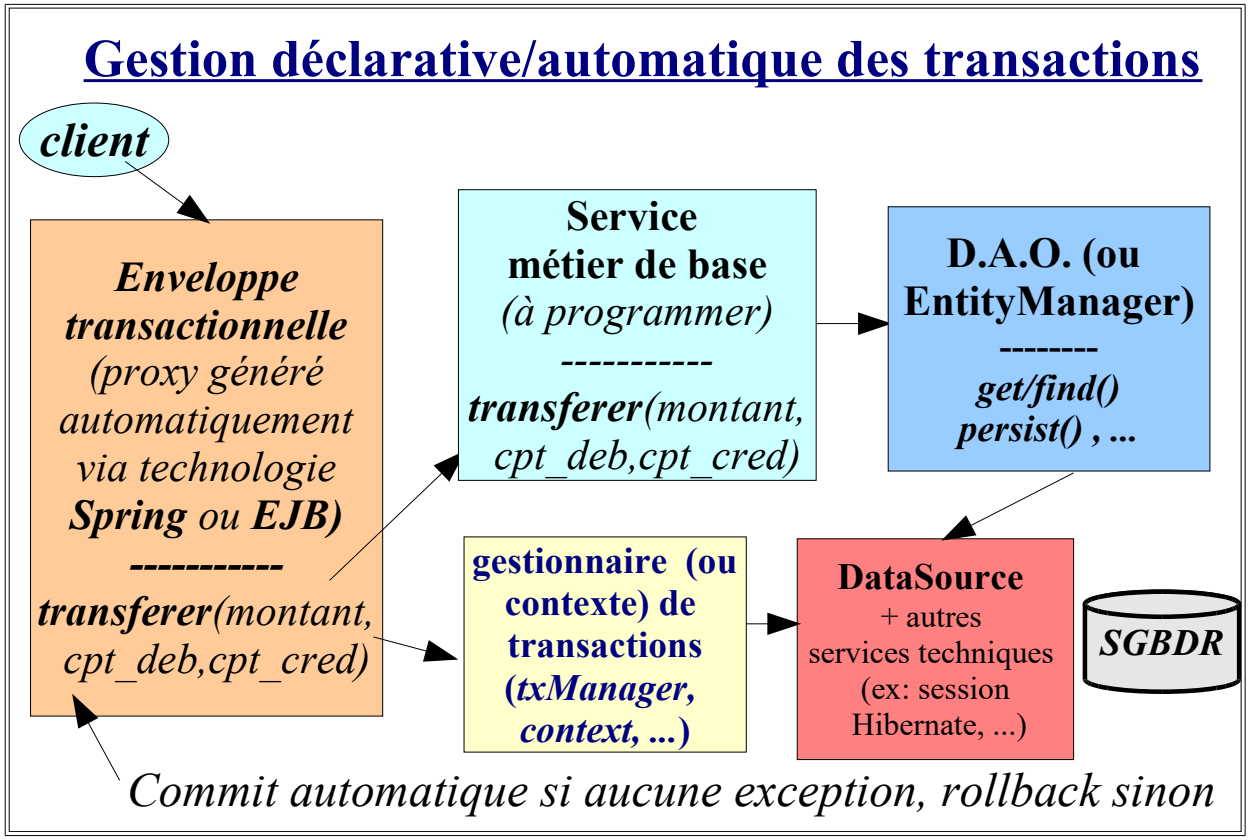
Ceci suppose :

- 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).`





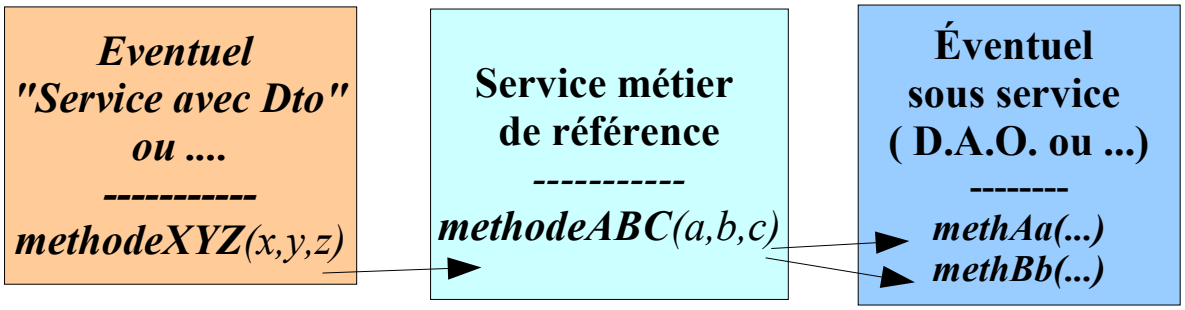

L'enveloppe transactionnelle supervisera automatiquement les "commit" et les "rollback" en fonction d'un paramétrage XML (ou bien en fonction de certaines annotations).

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

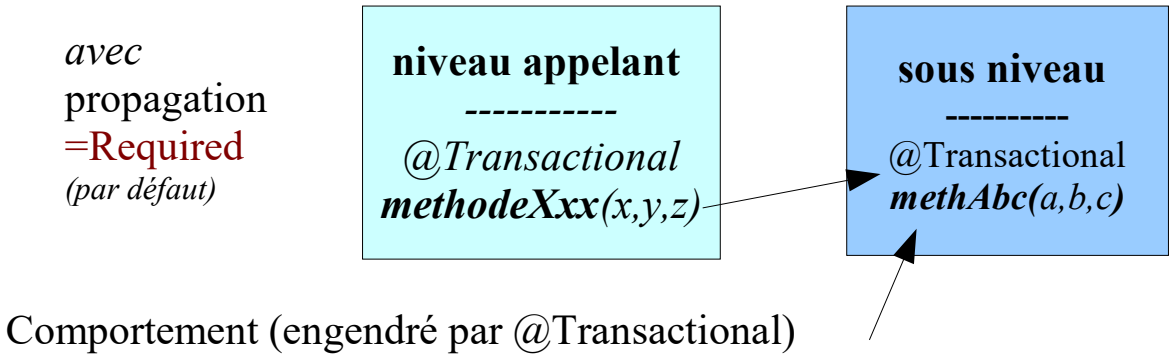
## Propagation du contexte transactionnel




| Propagation | Tx en cours ( appelant) | Tx dans sous service |
|---|---|---|
| Required *(par défaut)* | none<br>tx1 | new_tx<br>tx1 |
| Support | none<br>tx1 | none<br>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 xml 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. Configuration explicite en mode "java-config"

```
@Configuration
@EnableTransactionManagement()
@ComponentScan(basePackages={"tp.myapp.minibank"})
public class ServiceConfig ou JpaConfig{


     ...

     // Transaction Manager for JPA or ...
     @Bean(name="transactionManager")
     public PlatformTransactionManager transactionManager(
               EntityManagerFactory entityManagerFactory) {
          JpaTransactionManager txManager =
                    new JpaTransactionManager();
          txManager.setEntityManagerFactory(entityManagerFactory);
          return txManager;
     }

}
```

## 5. Marquer besoin en transaction avec @Transactional

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

en config XML

ou bien

```
@EnableTransactionManagement()
```

en mode java-config est nécessaire pour bien interpréter `@Transactional` dans le code d'implémentation des **services** et des "DAO" .

--> Exemple :

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

...
//éventuel @Transactional de niveau classe entière
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 exceptions» (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) .

# IX - Spring "web" (intégration avec Servlet, JSF,...)

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

### 1.1. WebApplicationContext (configuration xml)

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

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

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

NB1: le paramètre ***contextConfigLocation*** peut éventuellement comporter une liste de chemin (vers plusieurs fichiers) séparés par des virgules .
*Exemple*: "classpath:/spring/*.xml" ou encore
"classpath:/contextSpring.xml,/classpath:/context2.xml"

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

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

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

### 1.2. WebApplicationContext (configuration java-config)

```
class MyWebApplicationInitializer implements WebApplicationInitializer {
 public void onStartup (.. servletContext )...{
   WebApplicationContext context = new AnnotationConfigWebApplicationContext ();
```

```
context.register (MyWebAppConfig.class );
servletContext .addListener (new    ContextLoaderListener (context ));
//...  }}
```

## 1.3. WebApplicationContext (accès et utilisation)

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

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

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

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

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

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

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

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

**WEB-INF/faces-config.xml**

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

ou bien

```
@ManagedBean
@RequestScoped
public class MyJsfBean{

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

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

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

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

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

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

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

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

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

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

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

avec dans **pom.xml**

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

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

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

# X - Spring-Mvc et 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>
```

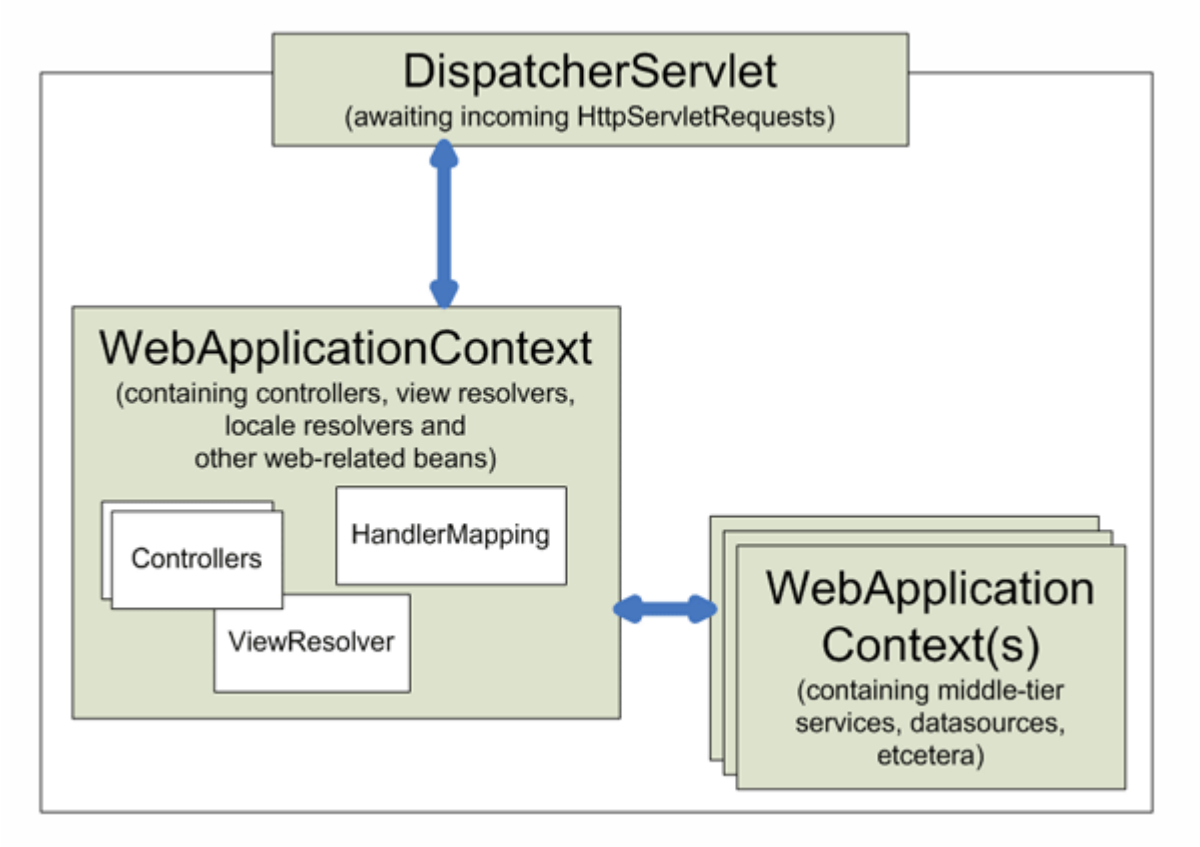
```
                    org.springframework.web.context.ContextLoaderListener
                </listener-class>
        </listener>
```

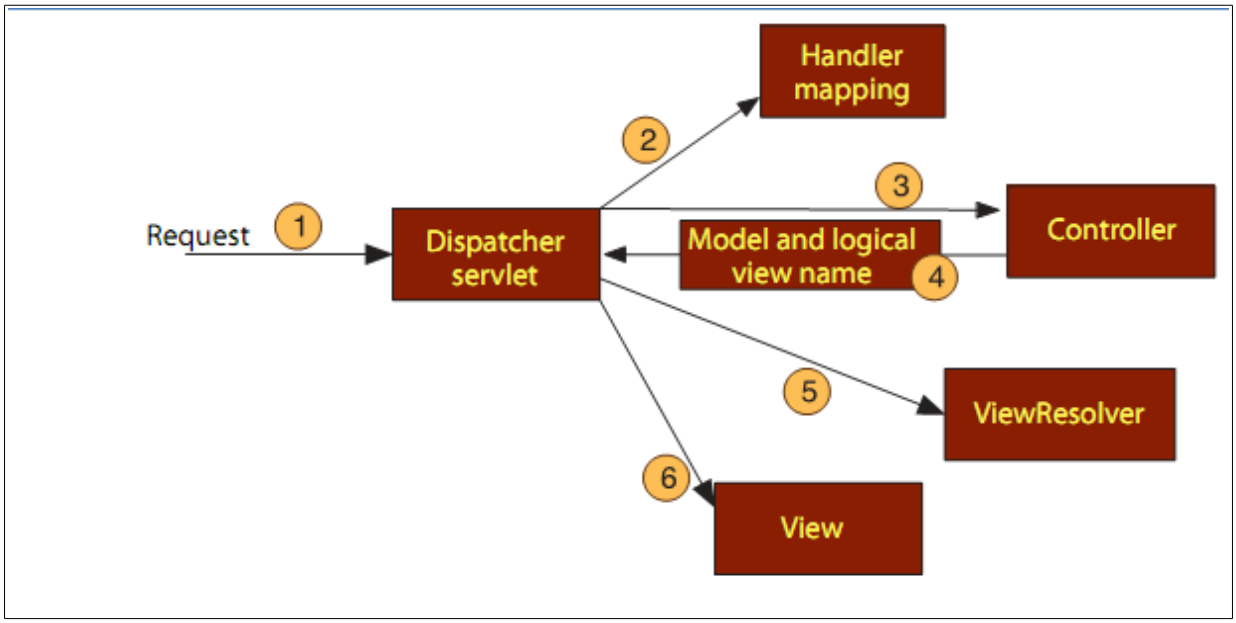
WEB-INF/**mvc-config.xml** (spring mvc)

```
<?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) :

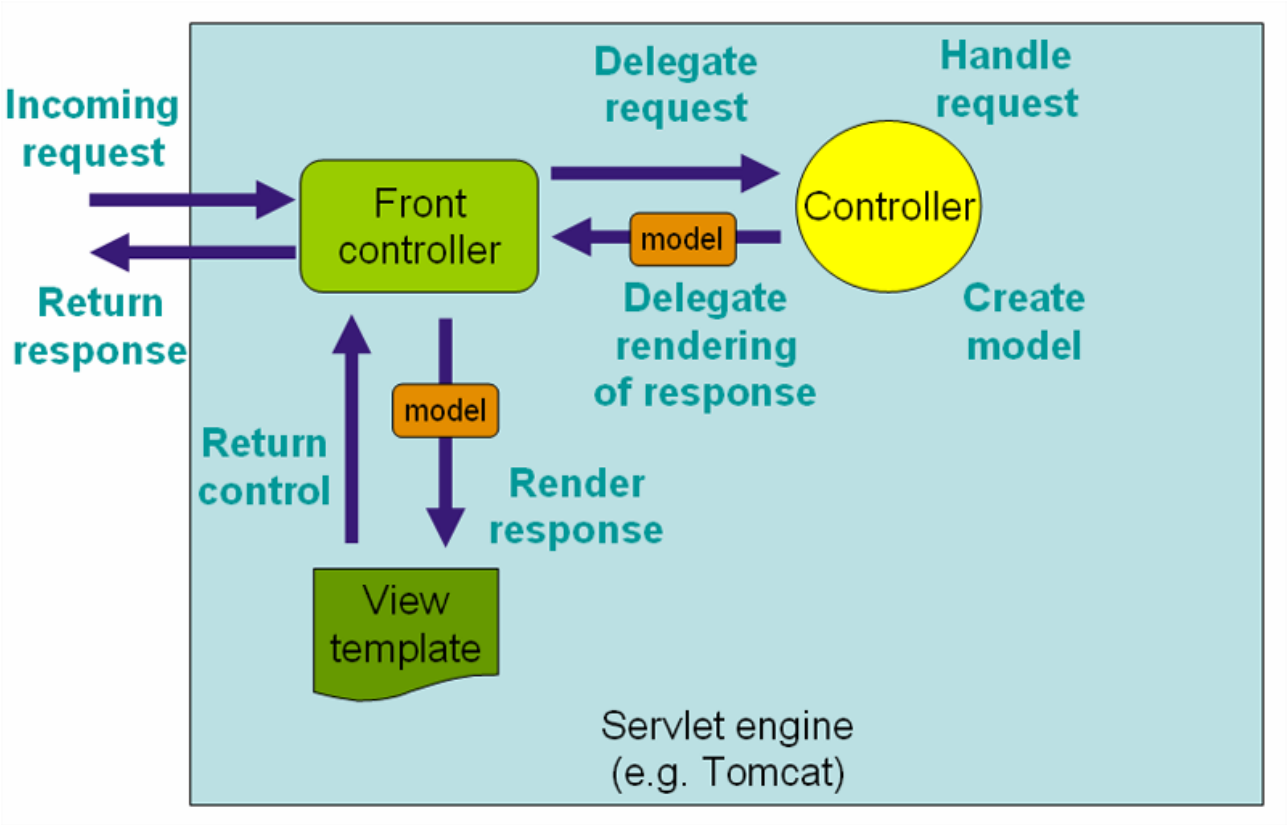
```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@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>
```

**Accès à un attribut pour effectuer une mise à jour:**

```
@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) et s'intègre mieux dans un écosystème spring (spring-security , ....) :

```
        <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

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);
}
}
```

Techniquement possible mais très rare : retour direct d'une simple "String' (text/plain) :

```
 //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"** :v

NB : il est techniquement possible de convertir explicitement une "Json String" en objet java via l'api "jackson" comme le montre l'exemple suivant :

```
...
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;
        }
} ….}
```

Ceci dit , Spring-Mvc est capable d'effectuer de lui même automatiquement cette conversion.

L'écriture suivante (plus simple) assure les mêmes fonctionnalités :

```
@RestController
@RequestMapping(value="/rest/devises" , headers="Accept=application/json")
public class DeviseJsonRestCtrl {
...
 @RequestMapping(value="/" , method=RequestMethod.PUT )
Devise updateDevise(@RequestBody  Devise devise) {
```

```
        System.out.println("devise to update:" + devise);
        gestionDevises.updateDevise(devise);
                return devise;
} ....
}
```

NB : dans tous les cas , il sera souvent nécessaire de contrôler le comportement des "sérialisations/dé-sérialisations java <--> json" en incorporant certaines annotations de "jackson" au sein des classes de données (dto / payload ) à véhiculer.

A ce sujet , l'annotation **@JsonIgnore** (sémantiquement équivalent à @XmlTransient) est assez souvent utile pour limiter la profondeur des données échangées .

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);
}
```

Autre exemple :

```
//url : http://localhost:8080/serverSpringMvc/ws/rest/devise/EUR
@RequestMapping(value="/{codeDev}",method=RequestMethod.DELETE)
public ResponseEntity< ?> deleteDeviseByCode(
                                        @PathVariable("codeDev")String codeDevise){
        try {
                deviseDao.deleteDeviseBycode(codeDevise);
                return new ResponseEntity< ?>(HttpStatus.OK);
```

```
        } catch (Exception e) {
                e.printStackTrace(); //ou logger.error(e) ;
                return new ResponseEntity< ?>(HttpStatus.NOT_FOUND);
                                        //ou HttpStatus.INTERNAL_SERVER_ERROR
                }
}
```

Autres variations :

**@GetMapping**(...) est équivalent à @RequestMapping(... , **method=RequestMethod.GET** )
**@PostMapping**(...) est équivalent à @RequestMapping(... , **method=RequestMethod.POST** )
**@PutMapping**(...) est équivalent à @RequestMapping(... , **method=RequestMethod.PUT** )
**@DeleteMapping**(...) équivalent à @RequestMapping(..., **method=RequestMethod.DELETE** )

## 4.1. Réponse et statut http par défaut en cas d'exception

Si une méthode d'un contrôleur REST remonte une exception java qui n'est pas rattrapée par un try/catch , la technologie Spring-Mvc retourne alors une réponse et un statut HTTP par défaut :

```
{ "timestamp" : 152....56,
"status" : 500 ,
"error" : "Internal Server Error",
"exception" : "java.lang.NullPointerException",
"message" : ".....",
"path" : "/rest/devise/67573567" }
```

Le statut HTTP retourné par défaut dans l'entête de la réponse en cas d'exception est généralement **500** (INTERNAL_SERVER_ERROR) .

Dans le cadre d'un échec de validation de la requête avec **@Valid** sur le paramètre d'entrée d'une méthode d'un contrôleur REST et avec des annotations de javax.validation (@Min , @Max , ...) sur la classe du "DTO" (ex : Devise) , le statut HTTP alors automatiquement remonté dans l'entête de la réponse HTTP est **400** (**Bad Request**) et le le corps de la réponse comporte tous les détails sur les éléments invalides .

```
public ResponseEntity<Void> ajouterDevise(@Valid @RequestBody Devise devise) {
....
}
```

```
public class Devise{
```

```
...
@Length(min=3, max=20, message = "Nom trop long ou trop court")
private String nom;
}
```

Dans le cadre d'une remontée d'exception personnalisée il est possible de préciser le statut HTTP qui sera remonté via l'annotation **`@ResponseStatus()`**

Exemple :

```
@ResponseStatus(HttpStatus.NOT_FOUND)  //404
public class MyEntityNotFoundException extends RuntimeException{
    //... constructeurs avec super()
}
```

## 4.2. Exemple d'appel en js/jquery/ajax

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


```
<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/",
        contentType : "application/json",
        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>
```

## 4.3. Invocation java de service REST via RestTemplate de Spring

Utile pour une **délégation de service** ou bien pour un **test d'intégration** (automatisable via maven et intégration continue).

```
.....
import org.junit.Assert;
import org.junit.BeforeClass;
import org.junit.Test;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.web.client.RestTemplate;

/ * cette classe à un nom qui commence ou se termine par IT (et par par Test)
 * car c'est un Test d'Integration qui ne fonctionne que lorsque toute l'application
 * est entièrement démarrée (avec EmbeddedTomcat ou équivalent) .*/
public class PersonWsRestIT {

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

	private static RestTemplate restTemplate; //objet technique de Spring pour test WS REST

	//pas de @Autowired ni de @RunWith
	//car ce test EXTERNE est censé tester le WebService sans connaître sa structure interne
	// (test BOITE_NOIRE)
	@BeforeClass
	public static void init(){
		restTemplate = new RestTemplate();
	}

	@Test
	public void testGetSpectacleById(){
		final String BASE_URL =
			"http://localhost:8888/spring-boot-spectacle-ws/spectacle-api/public";
		final String uri = BASE_URL + "/spectacle/1";
		String resultAsJsonString = restTemplate.getForObject(uri, String.class);
		logger.info("json string of spectacle 1 via rest: " + resultAsJsonString);
		Spectacle s1 = restTemplate.getForObject(uri, Spectacle.class);
		logger.info("spectacle 1 via rest: " + s1);
		Assert.assertTrue(s1.getId()==1L);
	}
```

```
    @Test
    public void testListeComptesDuClient(){
       final String villeDepart = "Paris";
       final String dateDepart = "2018-09-20";
       final String uri = "http://localhost:8080/flight_web/mvc/rest/vols/byCriteria"
                   +"?villeDepart=" + villeDepart + "&dateDepart=" + dateDepart;
       String resultAsJsonString = restTemplate.getForObject(uri, String.class);
       logger.info("json listeVols via rest: " + resultAsJsonString);
       Vol[] tabVols = restTemplate.getForObject(uri,Vol[].class);
       logger.info("java listeComptes via rest: "  +tabVols.toString());
       Assert.assertNotNull(tabVols);   Assert.assertTrue(tabVols.length>=0);
       for(Vol cpt : tabVols){
            System.out.println("\t" + cpt.toString());
       }
    }

    @Test
    public void testVirement(){
            final String uri =
                    "http://localhost:8080/tpSpringWeb/mvc/rest/compte/virement";
          //post/envoi:
          OrdreVirement ordreVirement = new OrdreVirement();
          ordreVirement.setMontant(50.0);
          ordreVirement.setNumCptDeb(1L);
          ordreVirement.setNumCptCred(2L);
          OrdreVirement savedOrdreVirement =
                restTemplate.postForObject(uri, ordreVirement, OrdreVirement.class);
          logger.info("savedOrdreVirement via rest: " + savedOrdreVirement.toString());
          Assert.assertTrue(savedOrdreVirement.getOk().equals(true));
    }

}
```

Exemple 2 (délégation de service) :

```
...
```

```
import java.nio.charset.Charset;
import java.util.Base64;
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.http.HttpEntity;
import org.springframework.http.HttpHeaders;
import org.springframework.http.HttpMethod;
import org.springframework.http.HttpStatus;
import org.springframework.http.MediaType;
import org.springframework.http.ResponseEntity;
import org.springframework.util.LinkedMultiValueMap;
import org.springframework.util.MultiValueMap;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.client.RestTemplate;

@RestController
@RequestMapping(value="/myapi/auth" ,    headers="Accept=application/json")
public class LoginDelegateCtrl {

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

        private static final String ACCESS_TOKEN_URL =
                                    "http://localhost:8081/basic-oauth-server/oauth/token";

        private static RestTemplate restTemplate = new RestTemplate();

        HttpHeaders createBasicHttpAuthHeaders(String username, String password){
                HttpHeaders headers = new HttpHeaders();
                headers.setContentType(MediaType.APPLICATION_FORM_URLENCODED);
                String auth = username + ":" + password;
                byte[] encodedAuth = Base64.getEncoder().encode(
                                            auth.getBytes(Charset.forName("US-ASCII")) );
                String authHeader = "Basic " + new String( encodedAuth );
                headers.add("Authorization", authHeader);
```

```
            return headers;
            }


        @PostMapping("/login")
         public ResponseEntity<?> authenticateUser(@RequestBody AuthRequest loginRequest) {
        logger.debug("/login , loginRequest:"+loginRequest);
        String authResponse="{}";
        try{
        MultiValueMap<String, String> params= new LinkedMultiValueMap<String,
String>();
        params.add("username", loginRequest.getUsername());
        params.add("password", loginRequest.getPassword());
        params.add("grant_type", "password");
        //ResponseEntity<String> tokenResponse =
        //                restTemplate.postForEntity(ACCESS_TOKEN_URL,params, String.class);
        // si pas besoin de spécifier headers spécifique .


        HttpHeaders headers = createBasicHttpAuthHeaders("fooClientIdPassword","secret");
        HttpEntity<MultiValueMap<String, String>> entityReq =
                    new HttpEntity<MultiValueMap<String, String>>(params, headers);


        ResponseEntity<String> tokenResponse=
                    restTemplate.exchange(ACCESS_TOKEN_URL,
                                          HttpMethod.POST,
                                          entityReq,
                                          String.class);
        authResponse=tokenResponse.getBody();
        logger.debug("/login authResponse:" + authResponse.toString());
        return ResponseEntity.ok(authResponse);
        }
        catch (Exception e) {
          logger.debug("echec authentification:" + e.getMessage()); //for log
         return ResponseEntity.status(HttpStatus.UNAUTHORIZED)
                                        .body(authResponse);
        }
    }
}
```

## 4.4. Test d'un "RestController" via @WebMvcTest et MockMvc

Pour tester le comportement d'un composant "RestController" de Spring-Mvc sans avoir à démarrer un serveur complet tel que Tomcat (ou un équivalent) , on peut utiliser la classe **MockMvc** et l'annotation **@WebMvcTest** qui sont spécialement prévues pour faire fonctionner le code d'un web service rest de spring-mvc en recréant un contexte local ayant à peu près de même comportement que celui d'un conteneur web mais sans accès réseau/http .

```
@RunWith(SpringRunner.class)
@WebMvcTest(DeviseJsonRestCtrl.class)
public class DeviseJsonRestCtrlIntegrationTest {

   @Autowired
   private MockMvc mvc;

   @Test
   public void testXyz(){
   mvc.perform(get("/rest/devises/?name=euro")
                  .contentType(MediaType.APPLICATION_JSON))
                  .andExpect(status().isOk())
                  .andExpect(jsonPath("$", hasSize(1) ))
                  .andExpect(jsonPath("$[0].name", is("euro") ));
   }
}
```

*NB : Spring5 propose une variante @WebFluxTest et WebTestClient pour WebFlux .*

# XI - 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. Ancien mode de configuration (xml)

Les premières versions de spring-security étaient jadis configurées via des fichiers xml .
Dans pom.xml , **spring-security-core , spring-security-web et spring-security-config** avec des versions un peu décalées par rapport à spring-framework :

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

Dans web.xml , il fallait déclarer le filtre à utiliser
***org.springframework.web.filter.DelegatingFilterProxy*** via <filter> et <filter-mapping>

Et la configuration xml/spring de l'application faisait généralement référence à un sous fichier importé ***security-config.xml*** comportant :
- des droits accès selon "rôles utilisateurs" et selon uri/url (<security-http> , <security:intercept-url , permitAll , denyAll , ....)
- des paramétrages de formulaire de login , ...
- un paramétrage de gestionnaire d'authentification (ldap , jdbc , xml) et éventuellement quelques comptes utilisateurs pour effectuer des tests simples en mode développement
( <security:authentication-manager> <security:authentication-provider> <security:user-service> <security:user name="user1" password="pwd1" authorities="ROLE_USER" /> ....)

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

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

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

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

Avec <form> (au lieu de <form:form>) , 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" /> ) .

## 1.3. Configuration moderne de spring-security en java

```
@EnableWebSecurity
public class WebSecurityConfig extends
        WebSecurityConfigurerAdapter {

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

...
```

# XII - Asynchrone (reactor , webFlux , netty, ...)

...
Nouveautés de la version 5 (technologies très récentes , pas encore "classique/mature" ).
...

# ANNEXES

# XIII - Annexe – Spring JMS

## 1. intégration JMS dans Spring

--> étudier la document de référence et les exemples

# XIV - Annexe – Spring et Web Sockets

...

# XV - Annexe – Jta/atomikos  (tx distribuées)

...

# XVI - Annexe – Tests avancés

...

# XVII - Annexe – Aspects divers de Spring

...

## 1. Plugin eclipse Spring-Tools-Suite (STS)

### 1.1. Présentation de STS

**STS** signifie : "**Spring Tool Suite**" et correspond à un **paquet cohérent de plugins eclipse** qui permettent de **travailler confortablement sur des projets "spring" au sein de l'IDE eclipse**.

**Les principales fonctionnalités de STS sont les suivantes** :

- création de nouveaux projets "spring" (basés sur maven ou gradle) avec un début de configuration "maven + spring" et quelques exemples de code (basiques).
- Aide à la mise au point des fichiers de configuration "spring" (bons "namespaces xml" dans entête, …)
- Aide au développement d'applications basées sur "Spring-MVC"
- ….

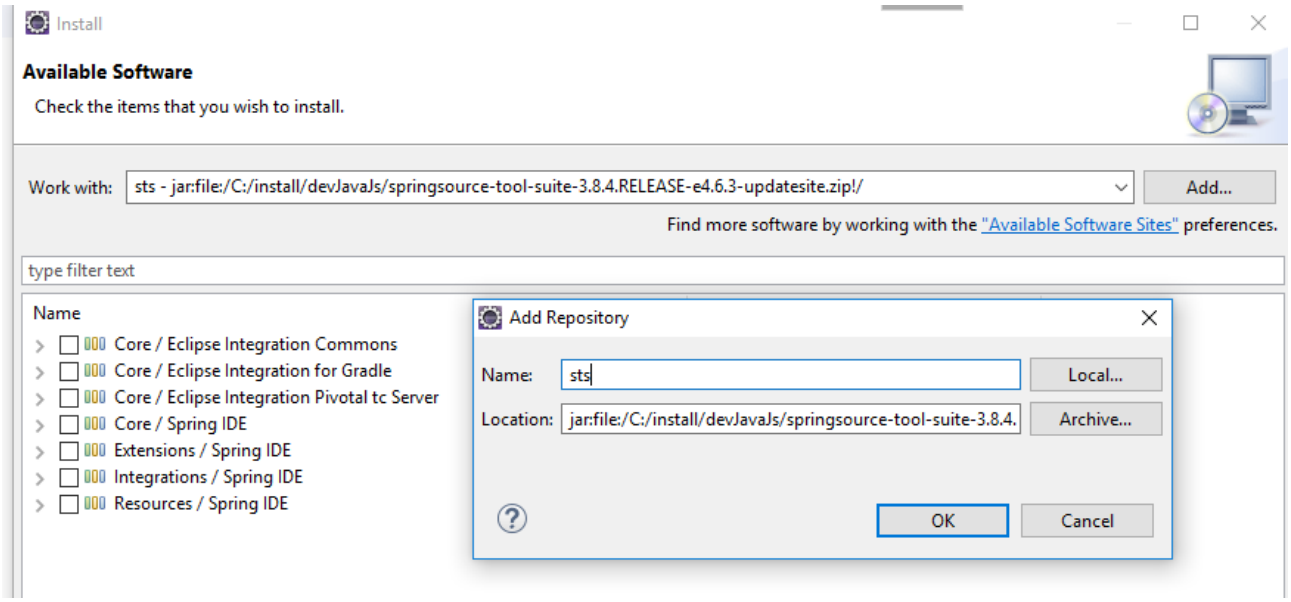
### 1.2. Installation du plugin eclipse STS

*Solution1* : via Eclipse MarketPlace

*Solution 2* : télécharger l'archive "**springsource-tool-suite-3.8.4.RELEASE-e4.6.3-updatesite.zip**" depuis le site "Spring-Tool-Suite" (ici pour eclipse 4.6/neon ou bien pour un autre eclipse) .

Effectuer l'installation via le menu **Help / install new Software** .

Préciser le chemin menant à "sts-….updatesite.zip" via le menu "**add …**" , "**archive …**" :

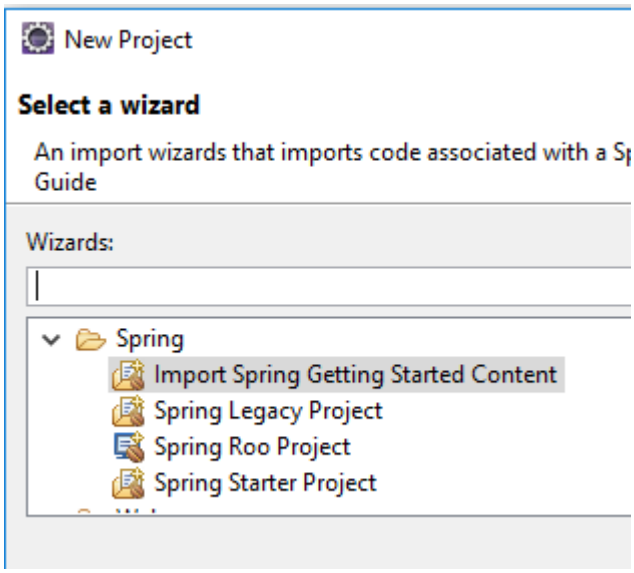



Cocher tout ou bien les parties intéressantes seulement (avec éventuel recul).

Accepter la licence. Redémarrer eclipse

## 1.3. Projets types/exemples de STS

Via le menu "**File / New … / Project … / Spring**" , on peut créer de nouveaux projets basés sur la technologies "spring" :




NB : **La plupart de ces projets sont plutôt à considérer comme des projets exemples dont on peut s'inspirer**. Un vrai projet d'entreprise doit avant tout s'intégrer dans une logique d'entreprise .

| ***Types de projet*** | ***Fonctionnalités*** | ***Caractéristiques*** |
|---|---|---|
| Import Spring Getting Started Content | Projets exemples concordants avec la partie "Getting Started" de la documentation"spring" | Juste pour apprendre "spring" via un tutorial et s'entraîner sur tel ou tel aspect de spring |
| **Spring Legacy Project** | Projet spring ***sans*** *"spring boot"*<br>avec configuration "maven + spring" (".war" déployable par exemple dans tomcat) | Structure classique (assez bien structurée) mais basée sur des versions anciennes (spring3 , java 6, …) → il ne faut pas hésiter à changer les versions (spring 4, java 8, ...) |
| Spring Roo Project | Projet "Spring" basé sur l'extension facultative "Roo" permettant d'ajuster la configuration via des lignes de commande "roo" | Pour ceux qui aiment "roo" |
| **Spring Starter Project** | Projet "Spring 4 moderne" basé sur "**spring boot**" | Bon point de départ pour une application moderne . |

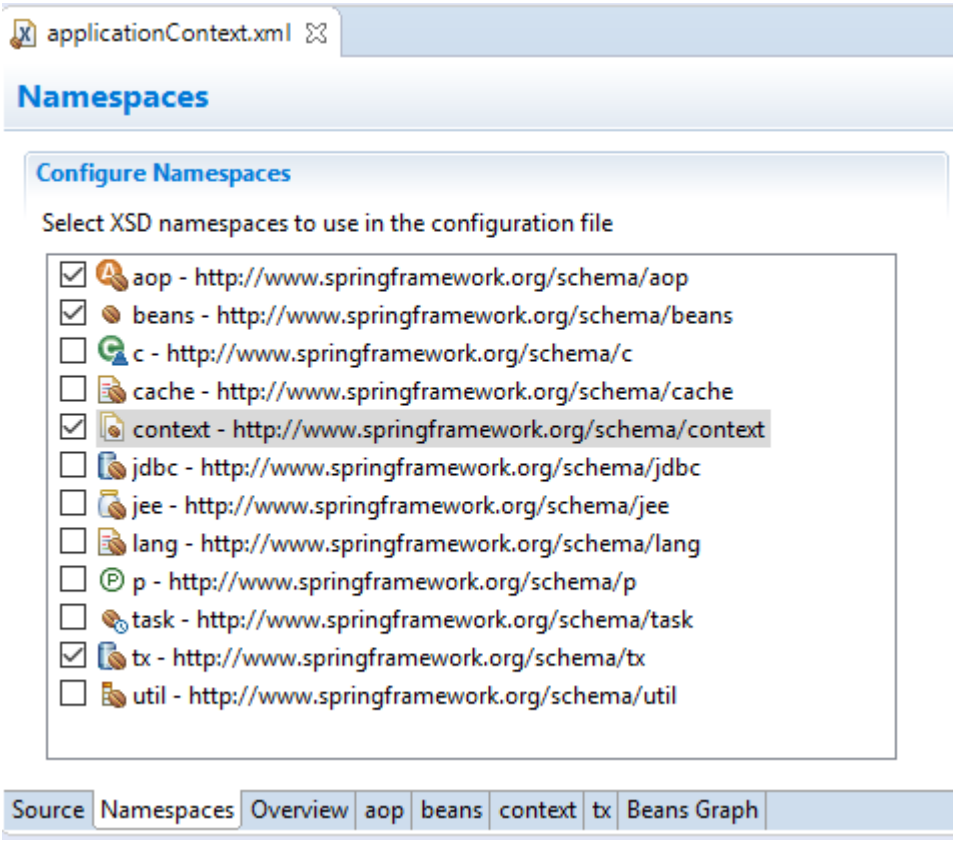
→ Soit "point de départ" à améliorer ,

soit "projet annexe pour isnpiration et copier/coller" .

## 1.4. Assistants STS pour les fichiers de configuration (xml,...)

Un fichier de configuration "spring" au format xml (ex : **applicationContext.xml** ) comporte **une entête complexe basée sur tout un tas de namespaces et de xsd** .

Lorsque le plugin STS est installé, on peut facilement mettre au point cette entête en fonction des besoins en **cochant** ou décochant **les namespaces utiles** au sein de l'onglet "**namespaces**" .




En revenant sur l'onglet "**source**" on peut visualiser l'entête réadaptée :

```
<?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:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
            http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.1.xsd
            http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-4.1.xsd
            http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-4.1.xsd">


</beans>
```

## 1.5. Nature "spring" des projets "eclipse"

Si un projet existant (exemple : "maven + spring") n'a pas été créé via un des assistants "nouveau projet spring" , on peut lui ajouter une "nature spring" via le menu contextuel **"Spring Tools "** / **"Add Spring Project Nature …" .**

Une fois cette configuration effectuée, on poura exploiter à fond les assistants "spring" du plugin STS (configurations , …. )

Il existe également une **perspective "spring"**.

## 1.6. Assistants de STS pour Spring-Mvc

La plupart des **assistants** de STS sont assez **intuitifs** une fois que l'on connaît bien la sutructure de "**spring web mvc**"

...

...

# XVIII - Annexe – Bibliographie, Liens WEB + TP

## 1. Bibliographie et liens vers sites "internet"

| | |
|---|---|
| | |
| | |
| | |

## 2. TP