

---

# Spring 5 et Spring-Boot 2

## Table des matières

I - Spring : vue d'ensemble.....	4
1. Architecture / Ecosystème Spring.....	4
2. Design Pattern "I.O.C." / injection de dépendances.....	6
3. Principaux Modules de Spring.....	9
4. Configurations Spring – vue d'ensemble.....	10
II - Configurations ioc (xml , java , annotations).....	19
1. Ancienne configuration Xml de Spring.....	19
2. Java Config (Spring).....	20
Variantes : .....	26
3. Configuration IOC Spring via des annotations.....	30
4. Tests "JUnit4/5 + Spring" (spring-test).....	35
5. Cycle de vie , @PostConstruct , @PreDestroy.....	37
6. Injection par constructeur (assez conseillé).....	38

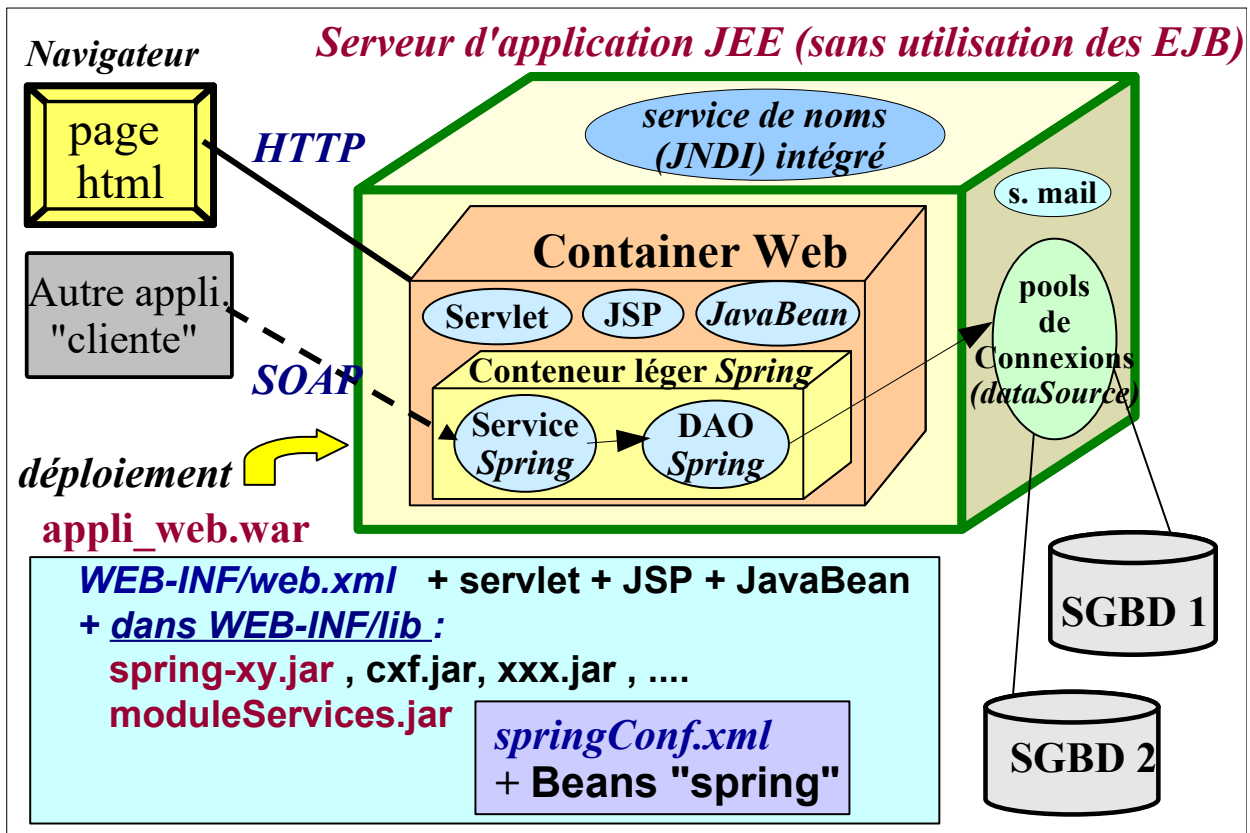
<b>III - Spring-boot.....</b>	<b>39</b>
1. Spring-boot.....	39
<b>IV - Spring backend (Services, Dao , Datasource).....</b>	<b>50</b>
1. Utilisation de Spring au niveau des services métiers.....	50
2. DataSource JDBC (vue Spring).....	52
<b>V - JPA , EntityManager (config. Spring).....</b>	<b>54</b>
1. DAO Spring basé sur JPA (Java Persistence Api).....	54
<b>VI - Spring-Data (avec JPA , ... ).....</b>	<b>60</b>
1. Spring-Data.....	60
<b>VII - Essentiel Spring AOP.....</b>	<b>67</b>
1. Spring AOP (essentiel).....	67
<b>VIII - Transactions "Spring".....</b>	<b>69</b>
1. Support des transactions au niveau de Spring.....	69
2. Propagation du contexte transactionnel et effets.....	71
3. Configuration du gestionnaire de transactions.....	72
4. Marquer besoin en transaction avec @Transactional.....	73
<b>IX - Spring "web" (intégration avec Servlet, JSF,...).....</b>	<b>74</b>
1. Injection de Spring au sein d'un framework WEB.....	74
2. Injection "Spring" au sein du framework JSF.....	76
3. Intégration de JSF 2 au sein de Spring-boot 2.....	78
<b>X - Spring-Mvc et Web Services REST.....</b>	<b>82</b>
1. Présentation du framework "Spring MVC".....	82
2. éléments essentiels de Spring web MVC.....	85
3. Web services "REST" pour application Spring.....	99
4. WS REST via Spring MVC et @RestController.....	100
<b>XI - Spring security.....</b>	<b>117</b>
1. Extension Spring-security.....	117
<b>XII - Asynchrone (reactor , webFlux , netty, ... ).....</b>	<b>128</b>

<b>XIII - Annexe – Spring_INITIALIZER.....</b>	<b>129</b>
1. Spring-Initializer.....	129
<b>XIV - Annexe – Ancienne config. XML / Spring.....</b>	<b>131</b>
1. Configuration xml de Spring.....	131
2. Configuration IOC Spring via des annotations.....	137
3. Tests "JUnit4 + Spring".....	141
4. Paramétrages Spring quelquefois utiles.....	143
<b>XV - Annexe – Ancienne configuration Spring 4.....</b>	<b>145</b>
1. ancienne version Spring-boot 1.x (Spring 4).....	145
<b>XVI - Annexe – Spring Actuator.....</b>	<b>149</b>
1. Spring-Actuator.....	149
<b>XVII - Annexe – Spring JMS.....</b>	<b>160</b>
1. Repères JMS.....	160
2. intégration JMS dans Spring.....	164
<b>XVIII - Annexe – Spring et Web Sockets.....</b>	<b>170</b>
<b>XIX - Annexe – Jta/atomikos (tx distribuées).....</b>	<b>171</b>
1. Transactions distribuées et commit à 2 phases.....	171
2. JTA / Atomikos.....	173
3. JTA/Atomikos intégré dans Spring et Spring-Boot.....	173
<b>XX - Annexe – Tests avancés.....</b>	<b>184</b>
<b>XXI - Annexe – Aspects divers de Spring.....</b>	<b>185</b>
1. Plugin eclipse Spring-Tools-Suite (STS).....	185
<b>XXII - Annexe – Bibliographie, Liens WEB + TP.....</b>	<b>189</b>
1. Bibliographie et liens vers sites "internet".....	189
2. TP.....	189

# I - Spring : vue d'ensemble

## 1. Architecture / Ecosystème Spring

Durant la première décennie du XXI siècle, Spring était à essentiellement considéré 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)

....

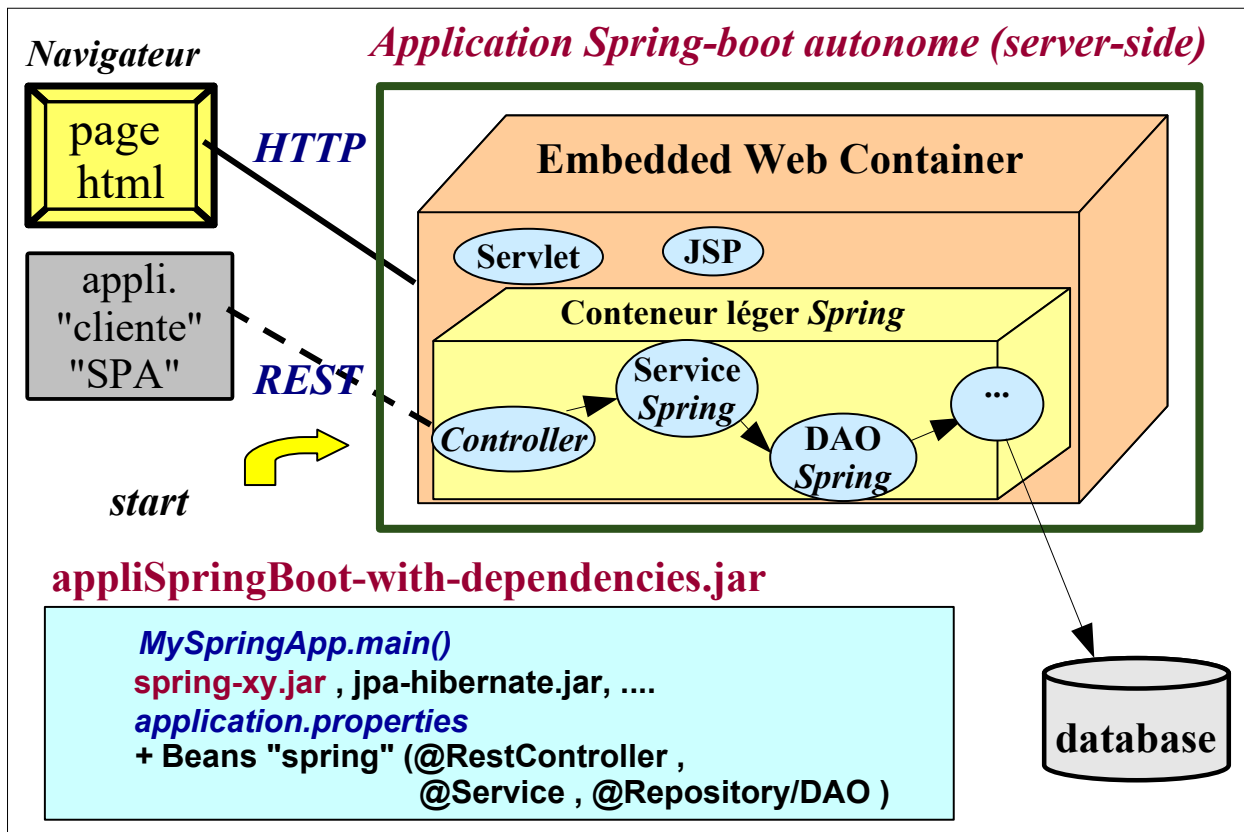
Le framework spring n'est pas associé à un grand éditeur de serveur JEE (tel que IBM , Oracle/BEA , Jboss) . Il a toujours laissé place à une très **grande liberté** dans le choix des technologies utilisées au sein d'une application java/JEE .

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 :

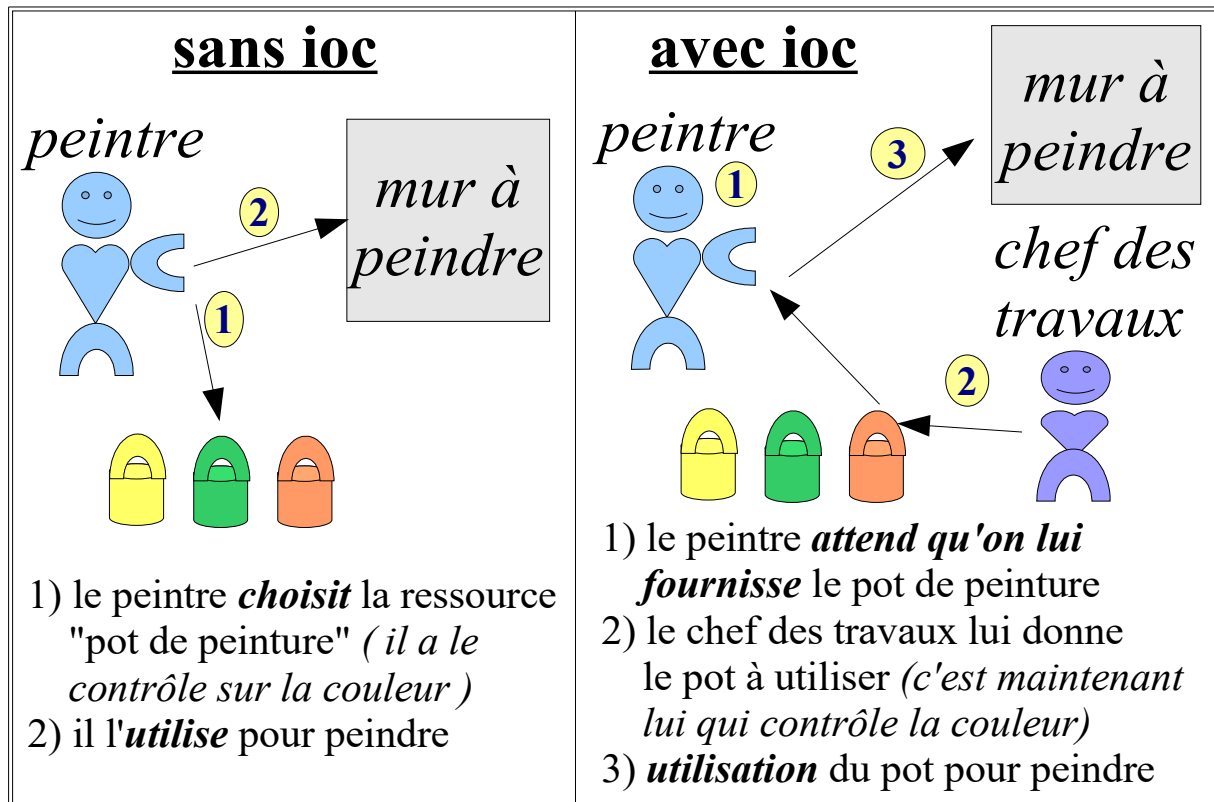
- **Spring boot** (démarrage complètement autonome . l'application incorpore son propre conteneur web (tomcat ou jetty ou netty ou ...)
- simplification de la configuration maven (ou gradle) via héritage de "POM/BOM/parent" .
- **Configuration java** (plus sophistiquée que l'ancienne configuration Xml , auto-complétion, rigueur , héritage , configuration conditionnelle intelligente)
- **AutoConfiguration** et simple fichier **application.properties** ou **.yaml**
- **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" .

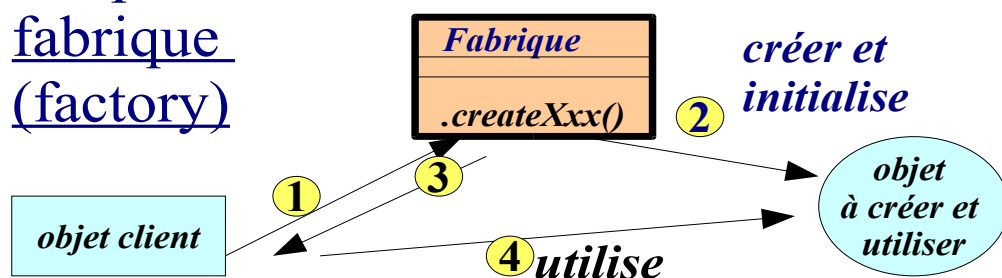


## 2. Design Pattern "I.O.C." / injection de dépendances

### 2.1. IOC = inversion of control



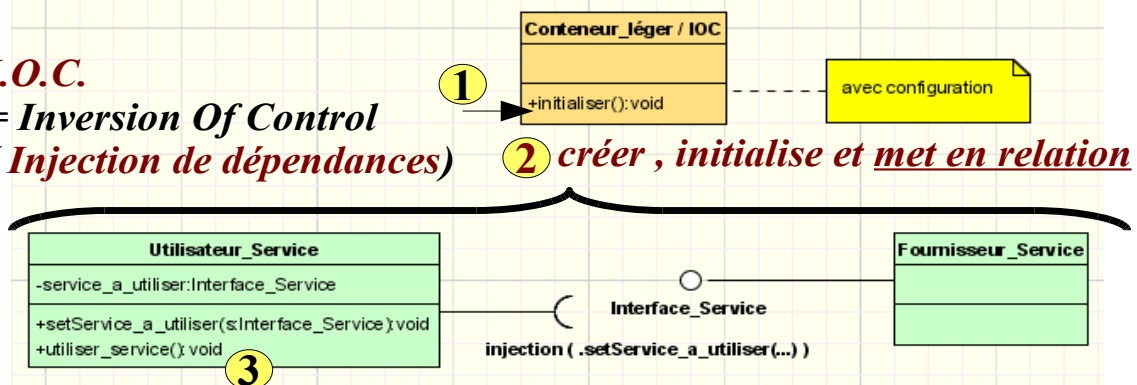
### Simple fabrique (factory)



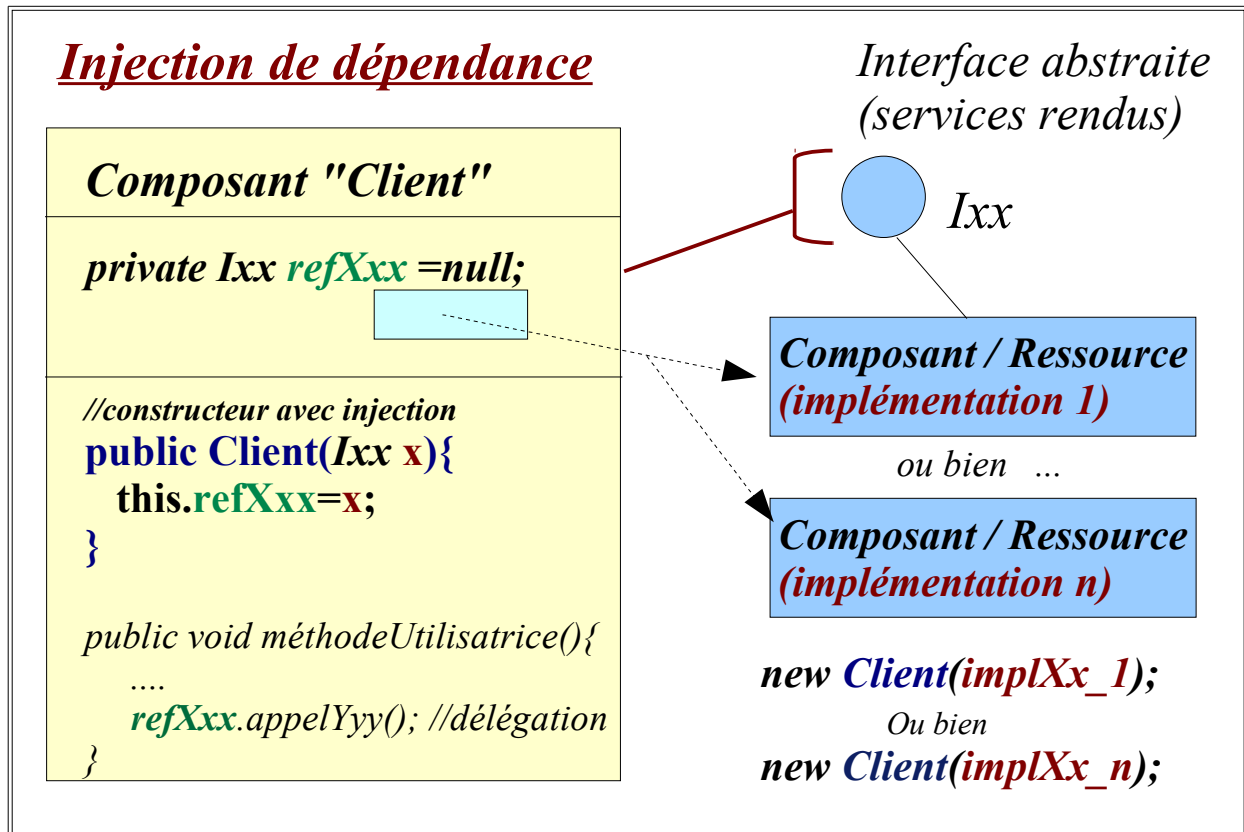
### I.O.C.

= Inversion Of Control

( Injection de dépendances )



## 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 compatible avec l'interface Ixx avant de l'utiliser , cet **objet client exposera une méthode** de type:

```
public void setRefXxx(Ixx res)
```

**ou bien un constructeur** de type:

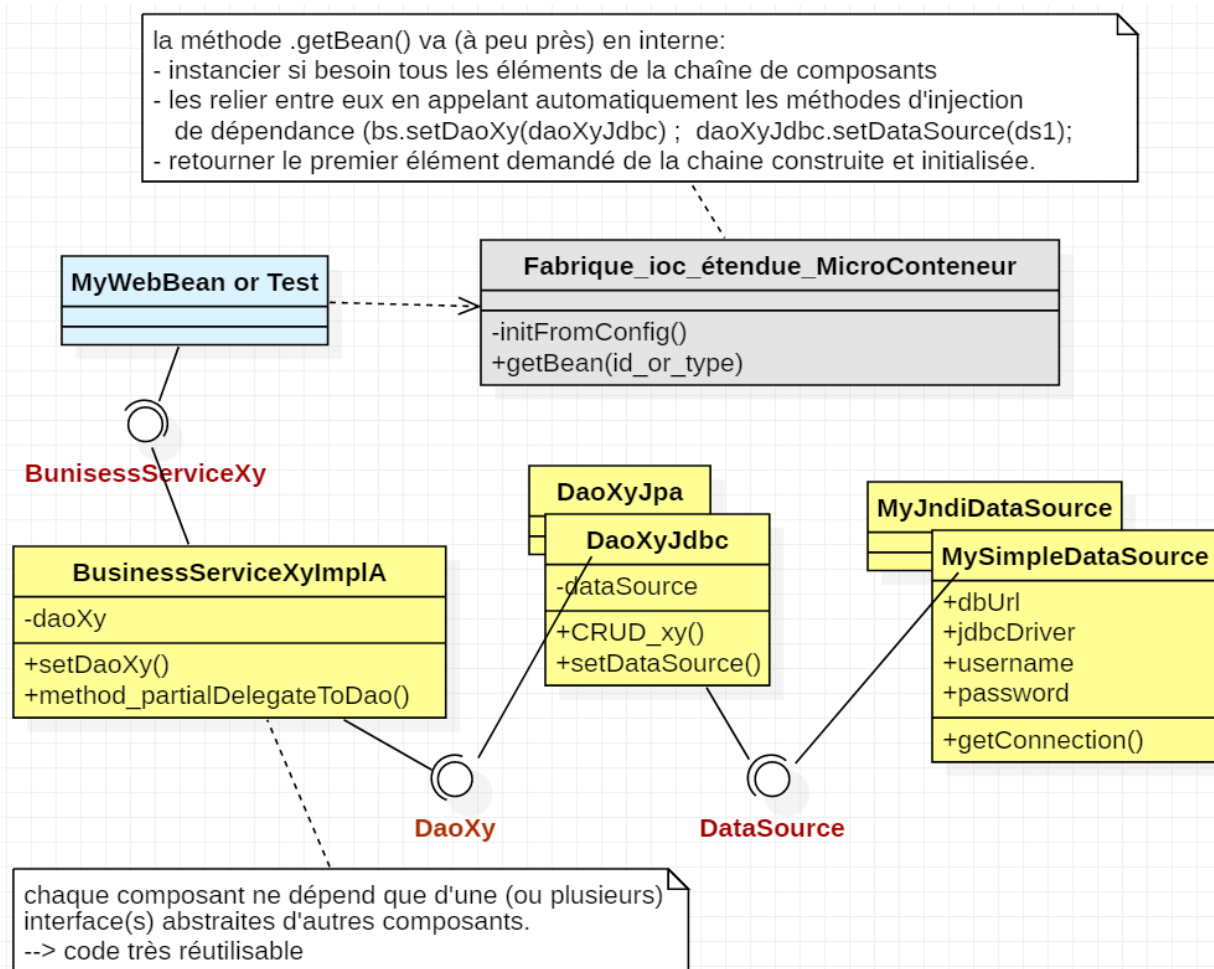
```
public Client(Ixx res)
```

**ou bien une référence annotée** de type :

```
@Autowired ou bien @Inject
private Ixx refXxx;
```

**permettant qu'on lui fournisse la ressource à ultérieurement utiliser**. Un tel composant sera ainsi très 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 <b>Core</b> + Spring <b>Beans</b>	conteneur léger – IOC (base du framework – BeanFactory )
Spring <b>AOP</b>	prise en charge de la programmation orientée aspect
Spring <b>DAO</b>	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 <b>Context</b>	Classes d'implémentation (POJO Wrapper) et de proxy pour les technologies distribuées (EJB, Services Web , RMI , JMS, ....) + Contexte abstrait pour JNDI , ...
Spring <b>ORM</b>	Support abstrait pour les technologies de mapping objet/relationnel (ex: TopLink , <b>Hibernate</b> , iBatis, JDO, <b>JPA</b> ...)
Spring <b>Web</b>	WebApplicationContext , support pour le multipart/UploadFile, points d'intégration pour des frameworks STRUTS , JSF, ...
Spring <b>Web MVC</b> (optionnel mais recommandé )	Version "Spring" pour un framework Web/MVC. Ce framework est "simple/extensible" et "IOC". Vis à vis du concurrent "JSF" , c'est visuellement plus pauvre mais c'est moins exclusif , c'est plus flexible , modulaire et ça peut s'associer à d'autres technologies complémentaires (ex : thymeleaf). <u>NB</u> : 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- <b>security</b>	Extension très utile pour gérer la sécurité JEE (roles , authentification , ...)
Spring <b>Data</b>	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 <b>Web flow</b>	Extension pour bien contrôler la navigation et rendre abstraite l'IHM (paramétrages xml des états , transitions, ...)
Spring <b>Batch</b>	prise en charge efficace des traitements "batch" (job , ...)
Spring <b>Integration</b>	Extensions pour SOA (fonctionnalités d'un mini ESB , EIP, ...)

## 4. Configurations Spring – vue d'ensemble

### 4.1. Historique et évolution

<i>Versions de Spring</i>	<i>Possibilités au niveau de la configuration</i>
<b>Depuis Spring 1.x</b>	Configuration entièrement <b>XML</b> (avec entête DTD) <bean >
Depuis Spring 2.0	Configuration <b>XML</b> (avec entête XSD) + .properties
<b>Depuis Spring 2.5</b>	<b>Annotations spécifiques à Spring (@Component , @Autowired, ...)</b>
Depuis Spring 3.0	Compatibilité avec annotations DI (@Inject , @Named)
<b>Depuis Spring 4.0</b>	<b>Java Config (@Configuration , ...)</b> et Spring boot 1.x (avec ou sans @EnableAutoConfiguration)
<b>Depuis Spring 5.0</b>	restructuration interne pour mieux intégrer java 8,9,10 et un début d'architecture asynchrone et réactive (Netty , WebFlux , ....) <b>Spring Boot 2.x</b> bien au point

## Spring (historique et évolution)

*Complexe et lourd*

J2EE 1.x et EJB 1 & 2

JEE 5 et EJB 3.0

@Entity (JPA1.0) , @EJB

JEE 6 et EJB 3.1

JPA 2.0 , @Named , @Inject

JEE 7 et EJB 3.2

JAX-RS 2 (WS-REST)

*Simple et efficace (le printemps)*

Spring 1.x

2003-2007  
environ

Spring 2.5

@Component , @Autowired,  
@Transactional

2006-2009  
environ

Spring 3.x

2009-2013  
environ

Spring 4.x et 5.x

Spring-boot , @Configuration ,  
Spring-data , @RestController, ...

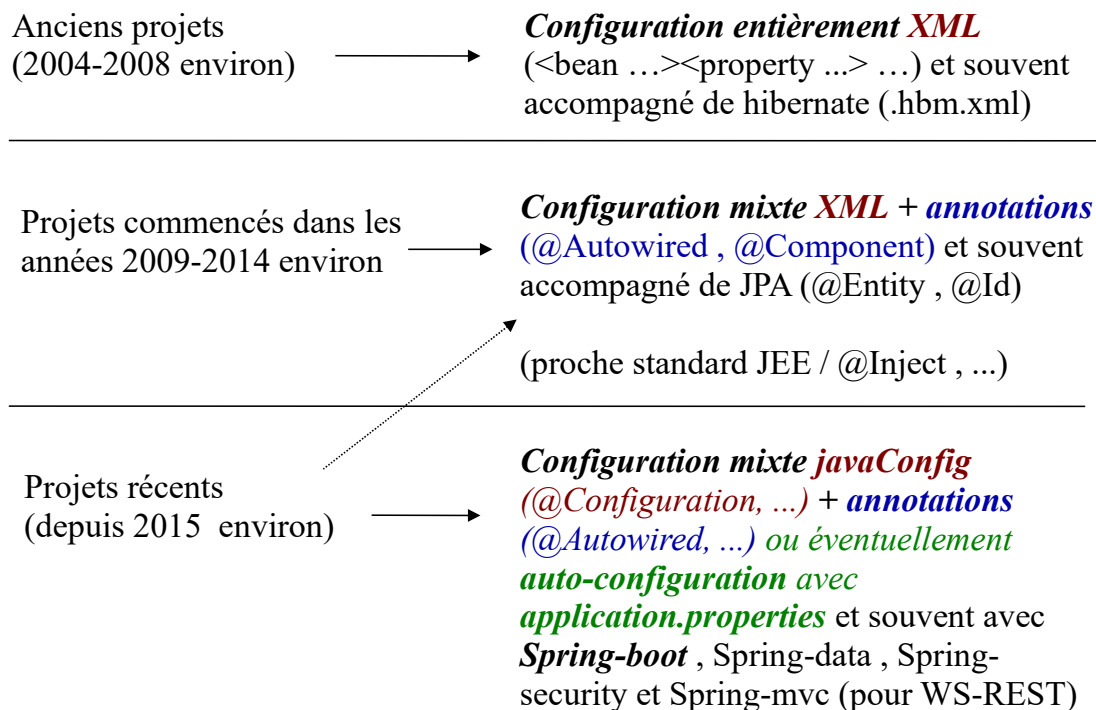
2013-2019  
environ

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

Mode de config	Avantages	Inconvénients
<b>XML</b>	<ul style="list-style-type: none"> <li>-Très explicite</li> <li>- Assez centralisé tout en étant flexible (import) .</li> <li>- utilisation possible de fichiers annexes ".properties"</li> </ul>	<ul style="list-style-type: none"> <li>- Verbeux , plus à la mode</li> <li>- à maintenir / ajuster (si refactoring)</li> <li>- délicat (oblige à être très rigoureux "minuscules / majuscules" , noms des packages , namespaces XML , ...)</li> </ul>
<b>Annotations</b> au sein des composants (@Autowired, ...)	<ul style="list-style-type: none"> <li>- très rapide / efficace</li> <li>- suffisamment flexible ( component-scan selon packages , @Qualifier , ...)</li> <li>- réajustement automatique en cas de refactoring (sauf component-scan) .</li> </ul>	<ul style="list-style-type: none"> <li>- configuration dispersée dans le code de plein de composants</li> <li>- pour nos composants seulement (avec code source)</li> </ul>
<b>Classes de configuration</b>	<ul style="list-style-type: none"> <li>-Très explicite</li> </ul>	<ul style="list-style-type: none"> <li>- nécessite une compilation de la configuration java (heureusement)</li> </ul>

<b>"java"</b> (@Configuration , ...)	- Assez centralisé tout en étant flexible (@Import) . - Auto complétion java et détection des incompatibilités (types , configurations non prévues, ...) - <b>utilisation possible de fichiers annexes ".properties" pour les paramètres amenés à changer</b> - à la mode ("hype" ) - <b>configuration automatique / intelligente possible (selon classpath, env, ...)</b>	souvent automatisée par maven ou autre)
--	--	---

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



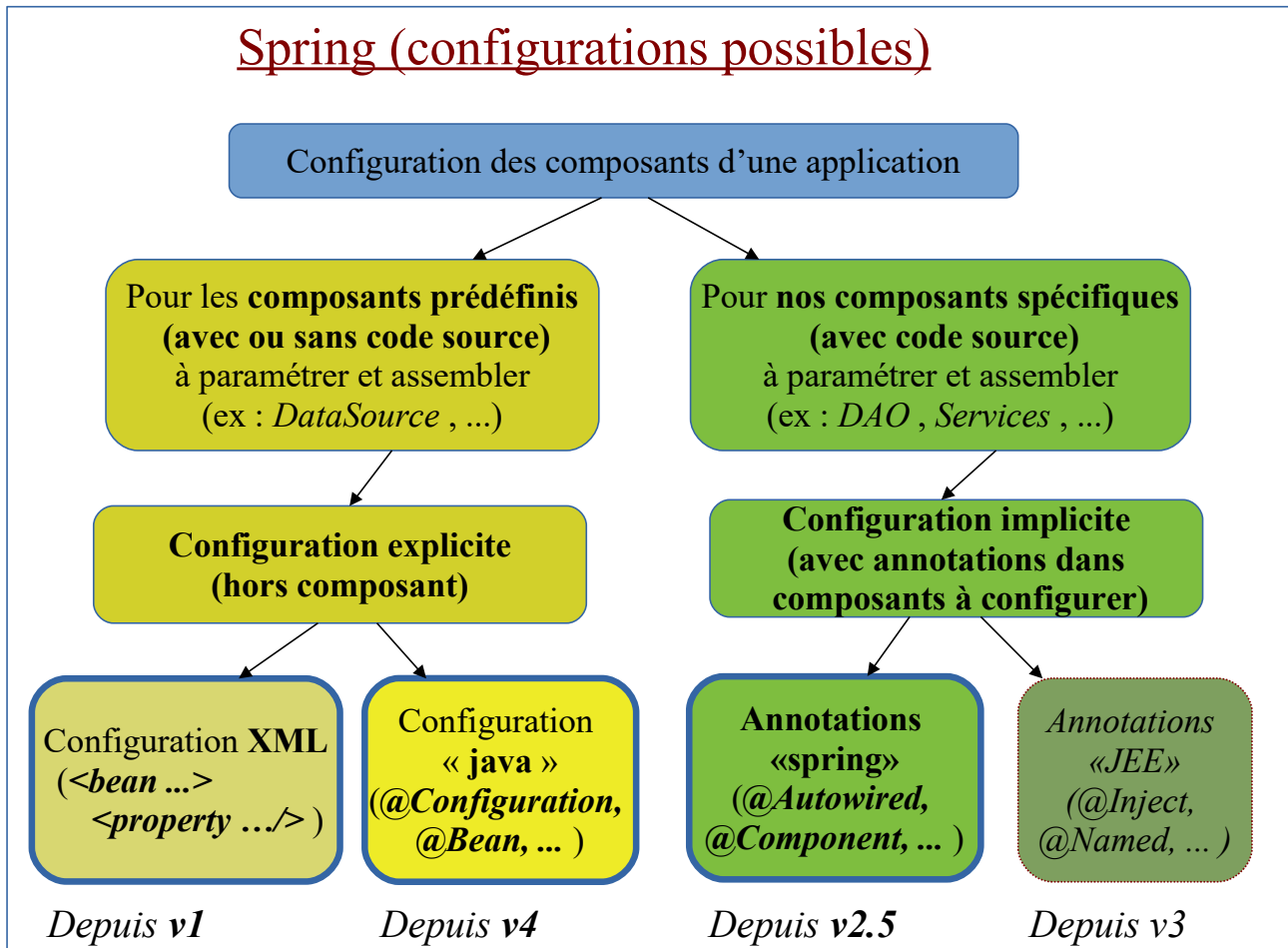
### 4.3. Complémentarité nécessaire / configuration mixte

- 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 avoir un contrôle total sur le code source).
- Une **configuration XML (ancienne)** 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, on s'appuie sur des **fichiers annexes** au format **".properties"** ou bien **".yml"** pour simplifier l'édition de quelques paramètres clefs susceptibles de changer (ex : url JDBC , username, password , ...)

## Spring (configurations possibles)



#### 4.4. Démarrages possibles depuis spring 2.5 (très anciens)

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

#### 4.5. Variantes de démarrages possibles depuis spring 4

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

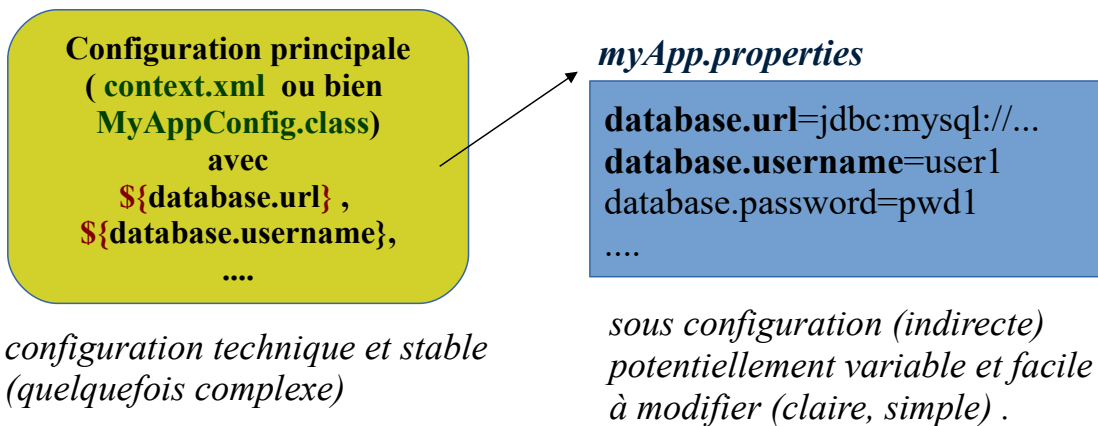
+ tous les nouveaux démarrages possibles via **spring-boot** .

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

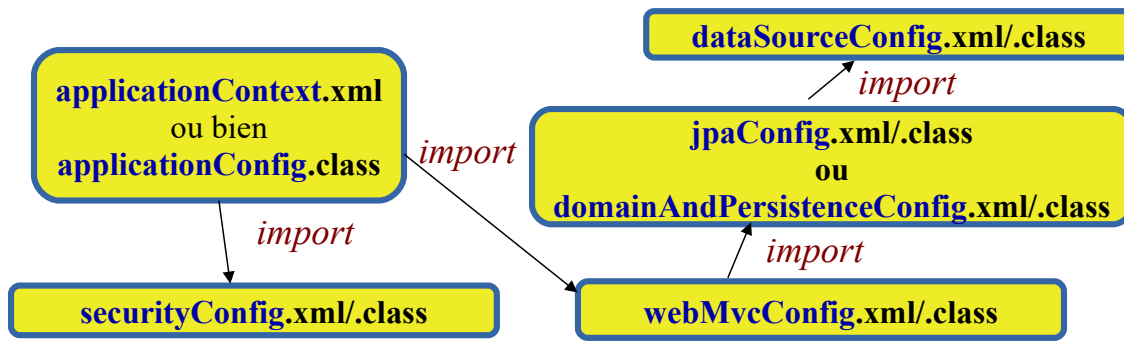
### Spring (paramétrages indirects dans fichiers ".properties")

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 .



## Spring (Configuration structurée via "import")



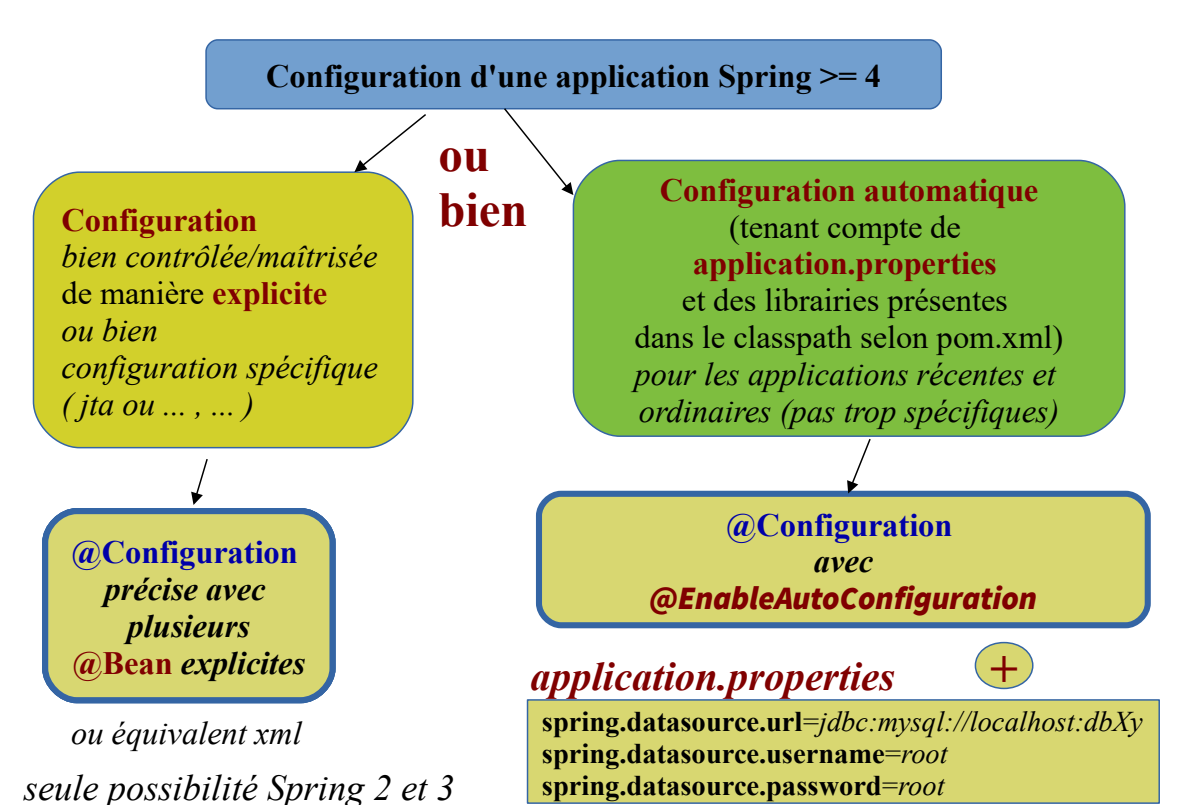
## Profiles (Variantes de configurations) depuis Spring4

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

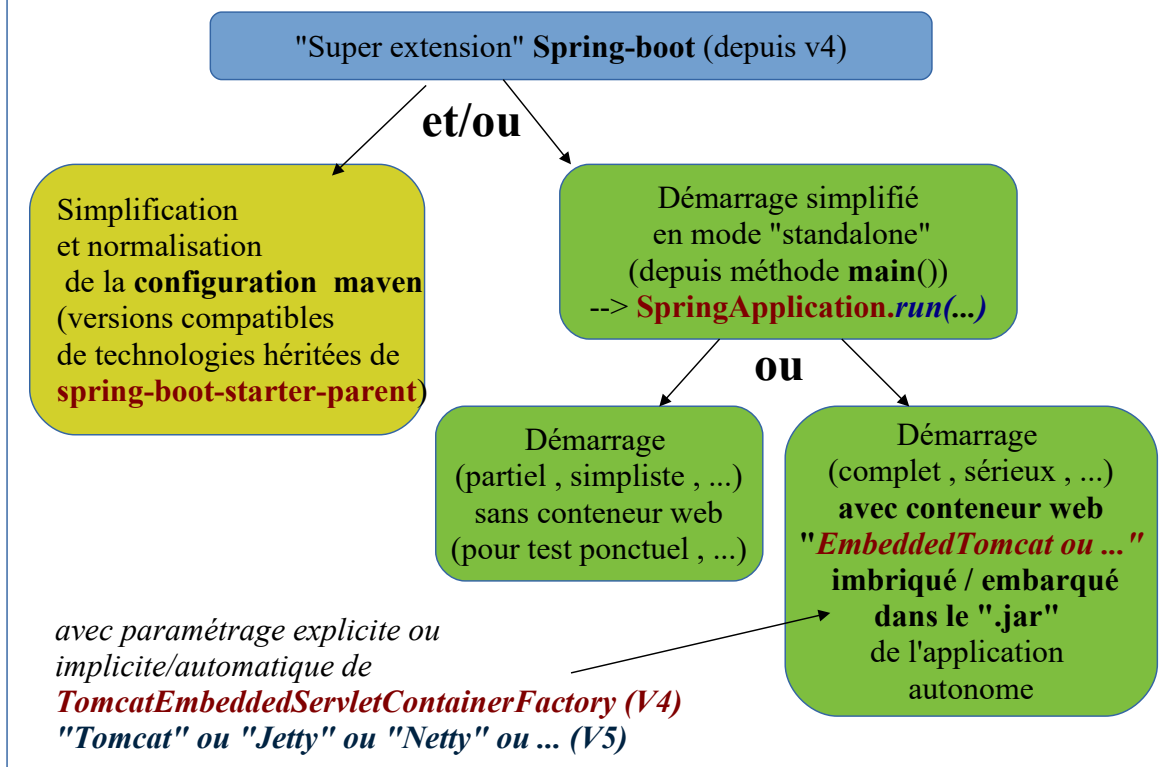
```
context.getEnvironment().setActiveProfiles(...);
ou bien
springBootApplication.setAdditionalProfiles(...);
ou bien
@ActiveProfiles(profiles = {"test", "jta"})
au dessus d'une classe de test (@RunWith, ...)
```

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

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





Spring >= 4 (apports facultatifs de Spring-boot)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/myMvcSpringBootApplication");
    }
}
```

==> 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)

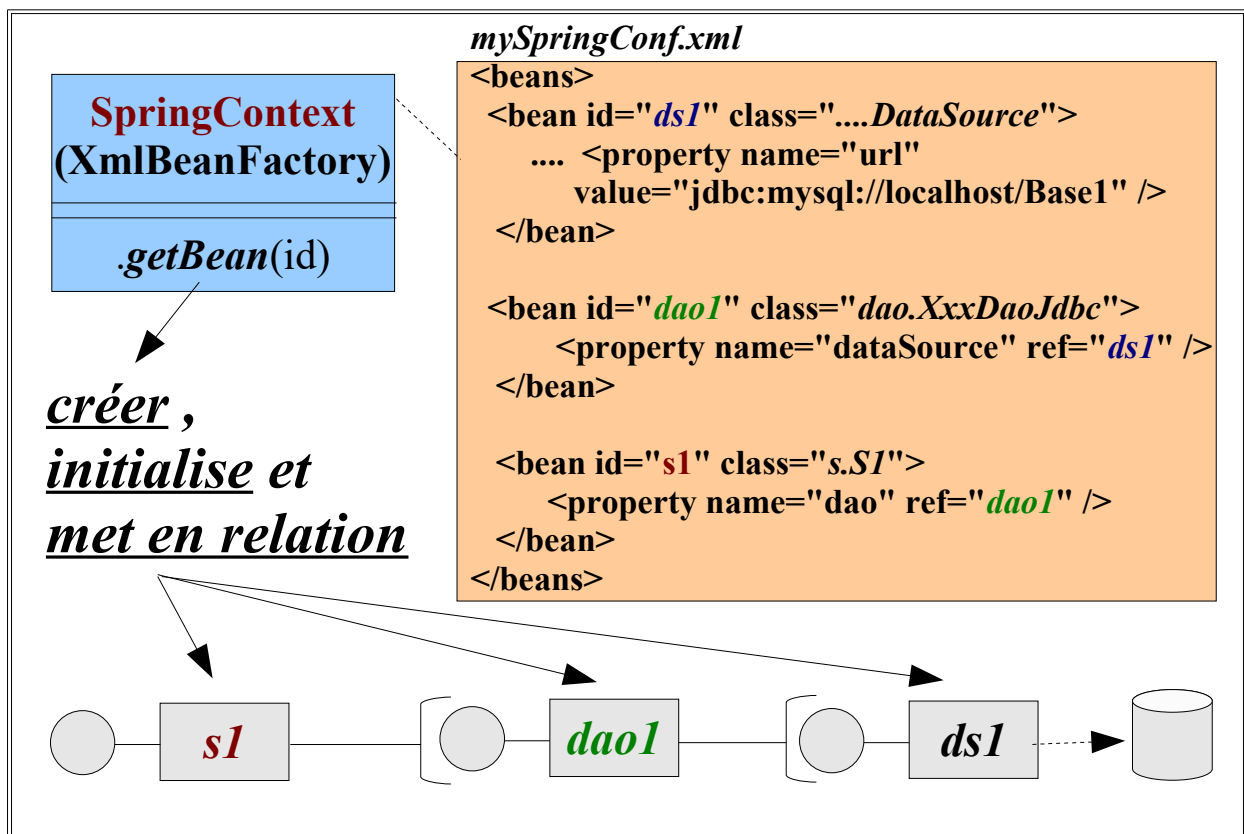
### 1. Ancienne configuration Xml de Spring

Configurer Spring avec des fichiers xml est aujourd'hui un peu obsolète .

La configuration Xml de Spring n'est aujourd'hui qu'à étudier et utiliser que pour maintenir des anciens projets (des années 2005-2015) .

---> le chapitre "configuration Spring XML" a maintenant été déplacé en annexe du cours .

Voici tout de même un micro-aperçu rapidement



## 2. 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 éventuellement possible.

NB : Depuis "Spring 5" et "Spring-Boot" , la configuration "**java-config**" est devenue la **configuration de référence** dans l'écosystème "**spring moderne**" et a complètement éclipsé l'ancienne configuration xml.

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, ...)
- configuration intelligente (selon classpath, selon env, ...)

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) !!!

### 2.1. 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.("root");
        dataSource.("root");//"root" ou "formation" ou "..."
        return dataSource;
    }
}
```

NB : cette classe de configuration "**DataSourceConfig**" sert à configurer un composant spring applicatif basé sur la classe "**DriverManagerDataSource**" prédéfinie dans **spring-jdbc** et implémentant l'interface **DataSource** standard du langage java

## 2.2. Utilisations possibles (ici sans spring-boot):

Dans `main()` :

```
ApplicationContext context =  
new AnnotationConfigApplicationContext(DataSourceConfig.class,  
                                         DomainAndPersistenceConfig.class);  
DataSource ds = context.getBean(DataSource.class);  
...
```

*Possible mais très rare* : dans `springContext.xml` (pour config java intégrée dans config xml):

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

Dans `spring test` (ici sans spring-boot):

```
@RunWith(SpringJUnit4ClassRunner.class) //si JUnit4  
// ou bien @ExtendWith(SpringExtension.class) si JUnit5/jupiter  
  
//@ContextConfiguration(locations="/springContextOfModule.xml") // si xml config  
@ContextConfiguration(classes={tp.myapp.minibank.impl.config.DataSourceConfig.class,  
                                tp.myapp.minibank.impl.config.DomainAndPersistenceConfig.class}) //java config  
// ou bien @SpringBootTest(classes= {MySpringBootApplication.class}) si spring-boot  
public class TestXy {  
    @Autowired  
    private .... ;  
  
    @Test  
    public void testXy(){ .....  
    }  
}
```

## 2.3. 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=
```

NB : cet exemple est "sans spring-boot" et donc sans le nom classique **application.properties**

### **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;
    }
}
```

NB: Dans le cas (très fréquent d'une configuration automatique avec `@EnableAutoConfiguration` ou bien `@SpringBootApplication`) , pas de besoin d'expliquer un objet technique de type `PropertySourcesPlaceholderConfigurer()` car c'est déjà configuré et pas besoin d'expliquer `@PropertySource("classpath:application.properties")` car c'est également déjà configuré automatiquement .

==> **et donc dans la plupart des cas juste besoin de :**

- **placer les propriétés au bon endroit** (dans le fichier **application.properties** ou **application.yml** ou ...)

- référencer à accès à ces propriétés via `@Value("${xx.yy.property-name}")` ou bien

- `@Value("${xx.yy.property-name : default_value_if_no_present_in_application_properties}")`

Syntaxes avec valeurs par défaut :

```
@Value("${some.key:my default value}")
private String stringWithDefaultValue;
```

```
@Value("${some.key:true}")
private boolean booleanWithDefaultValue;
```

```
@Value("${some.key:42}")
private int intWithDefaultValue;
```

```
@Value("${some.key:one,two,three}")
private String[] stringArrayWithDefaults;
```

### 2.4. Quelques paramétrages (avancés) possibles :

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

*//sachant qu'on peut également placer @PostConstruct au dessus de init() et @PreDestroy .*

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

*//sachant que le scope par défaut est DefaultScopes.SINGLETON*

### 2.5. 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.H2);
        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);

        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();
    }

    // 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 explicite ci-dessus est inutile en mode configuration automatique .



## 2.6. éventuel @Import explicite

```
@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* .

---

**NB (en mode configuration explicite):**

**@Import({SousPartieConfig1.class , SousPartie2Config.class})** est essentiellement utile **en mode "configuration explicite , non automatique"** pour importer/imbriquer des classes de configurations annexes/complémentaires rangées dans des packages génériques/utilitaires qui ne sont en règle générale pas directement liés au package principal de l'application courante.

---

**NB (en mode configuration implicite/automatique):**

Dans le cas (très fréquent aujourd'hui) d'une configuration automatique (avec @SpringBootApplication équivalent à peu près à @EnableAutoConfiguration + @ComponentScan/main\_package ), on considérera que le package principal de l'application est celui qui comporte la classe de démarrage de l'application (avec @SpringBootApplication et main()) .

Et dans ce cas toutes les classes de type @Configuration placées dans un des sous-packages du package principal de l'application seront alors automatiquement trouvées et activées (sans besoin de @Import) sauf si elles sont associées à des profils non activés au démarrage.

## 2.7. 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

springBootApplication.**setAdditionalProfiles**(...); au sein de la méthode main()

ou bien

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

## 2.8. Configuration conditionnelle intelligente

Annotations que l'on peut ajouter à côté 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 si 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 "spring-boot-starter-..." .

## 2.9. Chargement automatique d'un paquet de propriétés dans un objet java (avec éventuels sous objets)

Au lieu de charger en mémoire plein de petites propriétés complémentaires avec `@Value("${xx.yy.property-name}")` on peut charger d'un seul coup toute une arborescence de propriétés au sein d'un objet java via l'annotation `@ConfigurationProperties`.

Cette méthode comporte elle même différentes variantes.

La principale variante est la suivante :

```
package org.mygeneric.abc.properties;
@ConfigurationProperties(prefix = "xy")
public class XyProperties {
    private String p1;
    private Boolean p2;
    private ZzProperties zz ;
    //+ get/set et constructeur(s)
}
```

```
package org.mygeneric.abc.properties;
//sub level of properties:
public class ZzProperties {
    private String pa;
    private String pb;
    //+ get/set et constructeur(s)
}
```

application.properties

```
....
xy.p1=valeur1
xy.p2=true
xy.zz.pa=valeurA
xy.zz.pb=valeurB
```

ou bien

application.yml

```
....
xy:
  p1: valeur1
  p2: true
  zz:
    pa: valeurA
    pb: valeurB
```

Exemple d'utilisation d'un bean de "properties" au sein d'une classe de configuration :

```
package org.mygeneric.abc.autoconfigure;

@Configuration
@ConfigurationPropertiesScan("org.mygeneric.abc.properties")
public class MyAbcAutoConfiguration {

    @Autowired(required = false)
    public XyProperties xyProperties;

    @Bean
    public Prefixeur monAbc() {
        if(xyProperties!=null && xyProperties.getZz()!=null) {
            return new Abc(xyProperties.getZz().getPa(),...);
        } else {
            return new Abc(); //par défaut
        }
    }
}
```

L'annotation **@ConfigurationPropertiesScan**("org.mygeneric.abc.properties") permet de préciser les packages à scanner pour trouver des classes avec **@ConfigurationProperties** servant à charger en mémoire des parties de **application.properties** ou **application.yml** sous forme de composant java (injectable via **@Autowired** ou autre) .

### 3. Configuration IOC Spring via des annotations

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

Pour cela , Spring utilise essentiellement les annotations suivantes :

**@Component , @Service , @Repository , @RestController, @Autowired , @Qualifier, ...**

NB :

- **Ces annotations** doivent être placées au bon endroit dans une classe java applicative et **nécessitent donc un accès au code source des composants à paramétrer**
- Ce mode de configuration de l'injection de dépendance(@Component + @Autowired) est le plus simple/rapide à mettre en oeuvre
- La configuration complète d'une application est très souvent un mixte "java-config (@Configuration/@Bean) + annotations (@Component/@Autowired)"

#### 3.1. Annotations (stéréotypées) pour composant applicatif

exemple : XYDaoImplAnot.java

```
package tp.persistance.with_annot;

import org.springframework.stereotype.Repository;

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

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

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

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

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

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

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

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

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

### 3.3. @Autowired (fondamental)

Exemple (assez conseillé) avec @Autowired

```
@Service() //id par défaut = serviceXYAnot
public class ServiceXYAnot implements IServiceXY {
    private XYDao xyDao;

    //injectera automatiquement l'unique composant Spring configuré
    //dont le type est compatible avec l'interface précisée.
    @Autowired
    public void setXyDao(XYDao xyDao) {
        this.xyDao = xyDao;
    }

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

ou bien plus simplement :

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

    @Autowired
    private XYDao xyDao;

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

### 3.4. Paramétrage Java des alternatives au sens "Spring"

En organisant bien les packages java de la façon suivante :

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

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

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

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

**@Configuration**

**@ComponentScan(basePackages={"xxx.impl.dao.v1","org.mycontrib.generic"})**

```
public class XyzConfig {  
    ...  
}
```

ou bien

**@Configuration**

**@ComponentScan(basePackages={"xxx.impl.dao.v2","org.mycontrib.generic"})**

```
public class XyzConfig {  
    ...  
}
```

ceci fait que une seule des deux versions (v1 ou v2) est prise en charge par Spring et donc candidate à une injection paramétrée via **@Autowired** .

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

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

**NB :** **@ComponentScan** comporte plein de variantes syntaxiques (**include , exclude , ...**)

**Autre solution élégante pour choisir entre l'alternative v1 et v2**

**---> utiliser des profiles spring au niveau des composants :**

**@Component**

**@Profile({"profile1"})**

```
class DaoXYImpl1 implements DaoXY {  
    ....  
}
```



```
@Component
@Profile({"profile2"})
class DaoXYImpl2 implements DaoXY {
....
}
```

et

```
SpringApplication app = new SpringApplication(MySpringBootApplication.class);
app.setAdditionalProfiles("profile1", "profileA") ;
ConfigurableApplicationContext context = app.run(args);
//ou autre façon de démarrer (ex : @SpringBootTest et @ActiveProfiles("profile1,pA") )
```

--> selon le profile "profile1" ou bien "profile2" sélectionné au démarrage d'un test ou de l'application spring , une seule des 2 versions *DaoXYImpl1* ou *DaoXYImpl2* sera prise en charge par Spring et donc candidate à une injection paramétrée via @Autowired .

### 3.5. @Qualifier (pour variantes qui coexistent )

```
@Component @Qualifier("byCreditCard")
class PaymentByCreditCard implements Payment {
...
}
```

```
@Component @Qualifier("byCash")
class PaymentByCash implements Payment {
...
}
```

```
@Component
class ServiceXyDelegatingPayment implements ... {
    @Autowired @Qualifier("byCreditCard")
    private Payment paiementParCarteDeCredit ;

    @Autowired @Qualifier("byCash")
    private Payment paiementEnLiquide ;

    public void payer(double montant){...}
}
```

**@Qualifier** est surtout pratique pour injecter différentes variantes pouvant coexister en même temps et non pas des alternatives exclusives .

## 4. Tests "JUnit4/5 + Spring" (spring-test)

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

**Attention:** pour éviter tout problème d'incompatibilité entre versions, il est souhaitable d'utiliser une version très récente de JUnit 4 ou 5 et spring-test .

### Exemple de classe de Test de Service (avec annotations de JUnit4)

```
...
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;

// nécessite spring-test.jar et junit 4 dans le classpath
@RunWith(SpringJUnit4ClassRunner.class)
//@ContextConfiguration(locations={"/mySpringConf.xml"}) //si config xml
@ContextConfiguration(classes={XxConfig.class, YyConfig.class}) //java config
public class TestXy {

    @Autowired
    private IServiceXy service = null;

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

### Adaptations pour JUnit 5 (jupiter)

```
...
import org.junit.jupiter.api.Assertions;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.junit.jupiter.api.extension.ExtendWith;
import org.springframework.test.context.junit.jupiter.SpringExtension;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
...

// nécessite spring-test.jar et junit jupiter dans le classpath
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes={XxConfig.class, YyConfig.class}) //java config
public class TestXy {

    @Autowired
```

```
private IServiceXy service = null;

@Test
public void testXy(){
    Assertions.assertTrue( ... );
}

}
```

### Cas particulier pour certains tests de "DAO":

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 idéalement se comporter comme un service métier et doit normalement gérer les transactions (via les automatismes de Spring).

Via les annotations

```
@TransactionConfiguration(transactionManager="transactionManager",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*").

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

### Cas particulier "Spring-Boot" :

Dans un contexte "Spring + Spring-boot" , il faut idéalement **remplacer**

```
@ContextConfiguration(classes={XxConfig.class, YyConfig.class})
```

par

```
@SpringBootTest(classes= {MySpringBootApplication.class})
```

## 5. Cycle de vie , @PostConstruct , @PreDestroy

### Cycle de vie d'un composant pris en charge par Spring :

- 1) instanciation (appel au constructeur)
- 2) injections de dépendances (selon @Autowired)
- 3) appel à la méthode préfixée par **@PostConstruct** (si elle existe)
- 4) utilisation normale du composant spring
- 5) appel à la méthode préfixée par **@PreDestroy** (si elle existe) lors d'un arrêt (pas brutal) du contexte spring
- 6) éventuel appel à la méthode finalize() (si elle existe) sur l'instance java

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

public class XxxService
{
    @Autowired
    private IZzz zzObj ;

    @Autowired
    private IYyy yyObj ;

    private String v ; //valeur_a_initialiser_au_plus_tôt !

    public XxxService(){
        //NB: Le constructeur est déclenché avant la gestion
        // des @Autowired
        //donc zzObj et yyObj sont à null
        //et ne sont pas encore utilisables
        //dans le ou les constructeur(s)
    }

    @PostConstruct
    public void initBean() {
        //premier endroit où this.zzObj et this.yyObj ne sont normalement plus à null
        this.v = this.zzObj.recupValeur() ; ...
    }

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

```

## 6. Injection par constructeur (assez conseillé)

**@Component**

```
public class Cx {  
    public String ma() { return "abc"; }  
}
```

**@Component**

```
public class Cz {  
    public String mb() { return "def"; }  
}
```

**@Component**

```
public class Cy {  
    private Cx x;  
    private Cz z;  
  
    // @Autowired //explicit or implicit if just one constructor  
    public Cy(Cx x, Cz z) {  
        this.x=x; this.z=z;  
    }  
  
    public String mab() {  
        return x.ma() + "-" + z.mb();  
    }  
}
```

**@Configuration**

**@ComponentScan**(basePackages = {"org.mycontrib.backend.demo" })

```
public class LittleConfig {  
}
```

```
public class LittleDemoApp {  
    public static void main(String[] args) {  
        ApplicationContext context =  
            new AnnotationConfigApplicationContext(LittleConfig.class);  
        Cy y = context.getBean(Cy.class);  
        System.out.println(y.mab());  
    }  
}
```

## 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 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 conseillée 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>

```

...

## 1.2. 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()`).

### 1.3. 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 app = new SpringApplication(MySpringBootApplication.class);
        // app.setAdditionalProfiles("p1","p2","p3");
        ConfigurableApplicationContext context = app.run(args);
        System.out.println("http://localhost:8080/myMvcSpringBootApplication");
    }
}
```

==> 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` .

## 1.4. Tests unitaires avec Spring-boot

éventuellement :

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

@RunWith(SpringJUnit4ClassRunner.class)
//ou @ExtendWith(SpringExtension.class) si junit5/jupiter
@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.junit.jupiter.api.Assertions;
//import org.junit.jupiter.api.Test;
//import org.junit.jupiter.api.extension.ExtendWith;
//import org.springframework.test.context.junit.jupiter.SpringExtension;
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)
//ou bien @ExtendWith(SpringExtension.class) si junit5/jupiter
@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); //ou bien Assertions.assertTrue(1+1==2);
    }
}
```

## 1.5. Eventuelle auto-configuration (facultative)

L'annotation **@EnableAutoConfiguration** (à placer à côté 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;

@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 pour l'instant **"slf4j"+"logback"** par défaut pour générer des lignes de log.

On peut donc configurer les logs de l'application avec **logback.xml** .

#### **logback.xml** (exemple) :

```
<!-- this is a configuration file for LogBack log Api (under SLF4J) . LogBack is faster than old log4J12 but not better than log4j2 and logback is 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"> <i!-- "debug" , "info" , "warn" , "error" , ... -->
    <appender-ref ref="STDOUT" />
  </root>
</configuration>
```

**NB :** Dans l'évolution historique des technologies de log on a vu apparaître successivement :

- log4j 1.2.x
- logback
- log4j 2.x (encore un peu mieux que logback)

Si l'on souhaite utiliser log4j 2.x à la place de logback , on peut éventuellement activer la configuration suivante dans pom.xml :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
  <exclusions>
    <exclusion>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-logging</artifactId>
    </exclusion>
  </exclusions>
</dependency>

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-log4j2</artifactId>
</dependency>
```

## 1.6. 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 src/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=/myMvcSpringBootApplication
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
```

## 1.7. Profiles 'spring' (variantes dans les configurations)

**NB :** Les profiles "spring" (variantes de configurations) peuvent éventuellement être complémentaires . L'annotation **@Profile()** peut être placée sur un composant Spring ordinaire (préfixé par exemple par **@Component**) ou bien sur une classe de configuration (**@Configuration**) .

Exemple :

```
import javax.annotation.PostConstruct;
import org.springframework.context.annotation.Profile;
...
@Component
@Profile("reInit")
public class ReInitDefaultDataSet {

    @Autowired
    private DeviseService deviseService;

    @PostConstruct
    public void initDataSet() {
        deviseService.saveOrUpdate(new Devise("EUR","Euro",1.0));
        deviseService.saveOrUpdate(new Devise("USD","Dollar",1.1243));
    }
}
```

Ce composant (servant ici à initialiser un jeu de données en base) ne sera activé et utilisé au sein de l'application Spring que si le profile "reInit" est activé .

D'autre part, le framework "spring" analyse automatiquement les fichiers **application-profileName.properties** (en complément de application.properties) si le profile "profileName" est activé au démarrage de l'application.

Si un même paramètre a des valeurs différentes dans application.properties et application-profileName.properties , la valeur retenue sera celle du profile activée .

Exemple :

**application-reInit.properties**

```
#database tables will be dropped & re-created at each new restart of the application or tests
# (dev only) ,CREATE TABLE will be generated from @Entity structure
spring.jpa.hibernate.ddl-auto=create
```

**Activation explicite d'un profile "spring" au démarrage d'une application :**

```

...
@SpringBootApplication
public class MySpringBootApplication extends SpringBootServletInitializer {
    public static void main(String[] args) {
        //SpringApplication.run(MySpringBootApplication.class, args);
        SpringApplication app = new SpringApplication(MySpringBootApplication.class);
        app.setAdditionalProfiles("embeddedDb","reInit","appDbSecurity");
        ConfigurableApplicationContext context = app.run(args);
        //sécurité par défaut si la classe WebSecurityConfig n'existe pas dans l'application:
        //System.out.println("default username=user et password précisé au démarrage");
    }
}

```

**Activation automatique d'un profile "spring" via des propriétés d'environnement :**

```

java .... -Dspring.profiles.active=reInit,embeddedDb

```

**Activation d'un profile "spring" au démarrage d'un test unitaire :**

```

@RunWith(SpringRunner.class)
@SpringBootTest(classes= {MySpringBootApplication.class})
@ActiveProfiles("reInit,embeddedDb,permitAllSecurity")
public class TestXy {
    ...
}

```

**1.8. 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

```



## 1.9. 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!";
    }
}
```

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

...

### 1. Utilisation de Spring au niveau des services métiers

#### 1.1. Dépendances classiques



Business **@Service** ---> Data Access Object (**@Repository**) ---> javax.sql.DataSource

Principales variantes au niveau du DAO:

- JDBC seulement
- JPA/Hibernate
- SpringData et base SQL ou NoSQL

#### 1.2. 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 ).

### 1.3. 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 ServiceCompte {

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

## 2. DataSource JDBC (vue Spring)

### 2.1. DataSource élémentaire (sans pool)

#### @Configuration

```
public class DataSourceConfig {

    @Bean(name="datasource") //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;
    }
}
```

#### 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 (directement ou indirectement)

NB : 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
```

## 2.2. 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-dbc**p" 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) , ....).

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-dbcp2</artifactId>
  <version>2.1</version>
</dependency>
```

```
DriverManagerDataSource dataSource = new DriverManagerDataSource();
```

```
BasicDataSource dataSource = org.apache.commons.dbcp2.BasicDataSource() ;
dataSource.setUrl(...) ; ...
```

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

```
<dependency>
  <groupId>com.mchange</groupId>
  <artifactId>c3p0</artifactId>
  <version>0.9.5.5</version>
</dependency>
```

ou bien

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-c3p0</artifactId>
  <version>5.4.10.Final</version>
</dependency>
```

et paramétrages avancés de ce type :

```
...c3p0.min_size=5
...c3p0.max_size=20
...c3p0.acquire_increment=5
...c3p0.timeout=1800
```

...

# 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.persistence.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; }
    //+ autres get/set
}
```

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

#### META-INF/persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.1" xmlns="http://xmlns.jcp.org/xml/ns/persistence"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/persistence
http://xmlns.jcp.org/xml/ns/persistence/persistence_2_1.xsd">
<persistence-unit name="myPersistenceUnit"
transaction-type="RESOURCE_LOCAL">

    <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    <class>entity.persistence.jpa.Compte</class>
    <class>entity.persistence.jpa.XxxYyy</class>
    <properties>
        <!-- <property name="hibernate.dialect"
            value="org.hibernate.dialect.MySQL5InnoDBDialect" /> -->
        <property name="hibernate.dialect"
```

```

        value="org.hibernate.dialect.H2Dialect" />
        <property name="hibernate.hbm2ddl.auto"
            value="create" /> <!-- or "none" -->
    </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 précèdent les méthodes des services métiers soient prises en compte par Spring de façon à générer (via AOP) une enveloppe transactionnelle.

**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, ...).

## 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.H2);
    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();
    }

    // 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. Simplification "Spring-boot" 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**

### Exemple pour H2

```

# JDBC settings for (h2) embedded dataBase
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.url=jdbc:h2:./h2-data/backendApiDb
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

```

### Exemple pour MySQL ou MariaDB

```

spring.datasource.driverClassName=com.mysql.jdbc.Driver
spring.datasource.url=jdbc:mysql://localhost:3306/backendApiDb
                        ?createDatabaseIfNotExist=true&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.database-platform=org.hibernate.dialect.MySQL5InnoDBDialect
spring.jpa.hibernate.ddl-auto=none
#spring.jpa.hibernate.ddl-auto=create

```

### Exemple pour PostgreSQL

```
#NB: avec postgresql , la base doit exister (même vide) avec username/password
spring.datasource.driverClassName=org.postgresql.Driver
spring.datasource.url=jdbc:postgresql://localhost:5432/backendApiDb
spring.datasource.username=postgres
spring.datasource.password=root
spring.jpa.database-platform=org.hibernate.dialect.PostgreSQLDialect
spring.jpa.hibernate.ddl-auto=none
#spring.jpa.hibernate.ddl-auto=create
```

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") .

## 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 , Cassandra, 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>** :

<code>&lt;S extends T&gt; S <b>save</b>(S entity);</code>	Sauvegarde l'entité (au sens saveOrUpdate) et retourne l'entité (éventuellement ajustée/modifiée dans le cas d'une auto-incrémentation ou autre).
<code><b>Optional</b>&lt;T&gt; <b>findById</b>(ID primaryKey);</code>	Recherche par clef primaire (avec jdk >= 1.8)
<code>Iterable&lt;T&gt; <b>findAll</b>();</code>	Recherche toutes les entités (du type courant/considéré)
<code>Long <b>count</b>();</code>	Retourne le nombre d'entités existantes
<code>void <b>delete</b>(T entity);</code>	Supprime une (ou plusieurs) entités
<code>void <b>deleteById</b>(ID primaryKey);</code>	

<code>void deleteAll();</code>	
<code>boolean exists(ID primaryKey);</code>	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 renvoyait auparavant une entité persistante recherchée via sa clef primaire et renvoyait **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()** ou bien **.orElse()** de **Optional<T>** de manière à récupérer un accès à l'entité persistante remontée :

```
public Compte rechercherCompte(long num) {
    return daoCompte.findById(num).get();    //retourne exception si null interne
    //return daoCompte.findById(num).orElse(null); //retourne null si null interne
}
```

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

<code>&lt;S extends T&gt; <b>Iterable</b>&lt;S&gt; save(<b>Iterable</b>&lt;S&gt; entities);</code>	Sauvegarde une liste d'entités
<code><b>Iterable</b>&lt;T&gt; <b>findAll</b>(<b>Iterable</b>&lt;<b>ID</b>&gt; ids );</code>	Recherche toutes les entités (du type considéré) ayant les Ids demandés
<code>void <b>delete</b>(<b>Iterable</b>&lt; ? Extends T&gt; entities)</code>	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 =
    // Ire 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 directe (sans spring-boot):

```

<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)

Dépendance maven indirecte (avec spring-boot):

```

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-data-jpa</artifactId>
</dependency>

```

Activation en (rare) configuration 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	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname =

Keyword	Sample	JPQL snippet
<b>Or</b>	findByLastname <b>Or</b> Firstname  FindByFirstname,	?2 ... where x.lastname = ?1 <b>or</b> x.firstname = ?2
<b>Is, Equals</b>	findByFirstname <b>Is</b> ,  findByFirstname <b>Equals</b>	... where x.firstname = ?1
<b>Between</b>	findByStartDate <b>Between</b>	... where x.startDate <b>between</b> ?1 <b>and</b> ?2
<b>LessThan</b>	findByAge <b>LessThan</b>	... where x.age < ? 1
<b>LessThanEqual</b>	findByAge <b>LessThanEqual</b>	... where x.age <b>&lt;=</b> ?1
<b>GreaterThan</b>	findByAge <b>GreaterThan</b>	... where x.age > ? 1
<b>GreaterThanEqual</b>	findByAge <b>GreaterThanEqual</b>	... where x.age <b>&gt;=</b> ?1
<b>After</b>	findByStartDate <b>After</b>	... where x.startDate > ?1
<b>Before</b>	findByStartDate <b>Before</b>	... where x.startDate < ?1
<b>IsNull</b>	findByAge <b>IsNull</b>	... where x.age <b>is</b> <b>null</b>
<b>IsNotNull,</b> <b>NotNull</b>	findByAge(Is) <b>NotNull</b>	... where x.age <b>not</b> <b>null</b>
<b>Like</b>	findByFirstname <b>Like</b>	... where x.firstname <b>like</b> ?1
<b>NotLike</b>	findByFirstname <b>NotLike</b>	... where x.firstname <b>not</b> <b>like</b> ?1
<b>StartingWith</b>	findByFirstname <b>StartingWith</b>	... where x.firstname like ?1 (parameter bound with appended %)
<b>EndingWith</b>	findByFirstname <b>EndingWith</b>	... where x.firstname like ?1 (parameter bound with prepended %)
<b>Containing</b>	findByFirstname <b>Containing</b>	... where x.firstname like ?1 (parameter



Keyword	Sample	JPQL snippet
<b>OrderBy</b>	<code>findByAge<b>OrderBy</b>LastName<b>Desc</b></code>	bound wrapped in %) ... where x.age = ? <b>1 order by</b> x.lastname <b>desc</b>
<b>Not</b>	<code>findByLastName<b>Not</b></code>	... where x.lastname <> ?1
<b>In</b>	<code>findByAge<b>In</b>(Collection&lt;Age&gt; ages)</code>	... where x.age <b>in</b> ?1
<b>NotIn</b>	<code>findByAge<b>NotIn</b>(Collection&lt;Age&gt; age)</code>	... where x.age <b>not</b> <b>in</b> ?1
<b>True</b>	<code>findByActive<b>True</b>()</code>	... where x.active <b>= true</b>
<b>False</b>	<code>findByActive<b>False</b>()</code>	... where x.active <b>= false</b>
<b>IgnoreCase</b>	<code>findByFirstname<b>IgnoreCase</b></code>	... where UPPER(x.firstname) <b>= UPPER(?1)</b>

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 côté de **@Entity** (ou **<named-query ...>** dans orm.xml) :

Dans orm.xml (référéncé 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-Cassandra

...

## 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  $\geq 2.x$ ) 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
@Component
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 `<aop:aspectj-autoproxy/>` dans une configuration spring xml  
ou bien `@EnableAspectJAutoProxy` sur une classe de `@Configuration` en mode java-config .

#### Ancienne 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

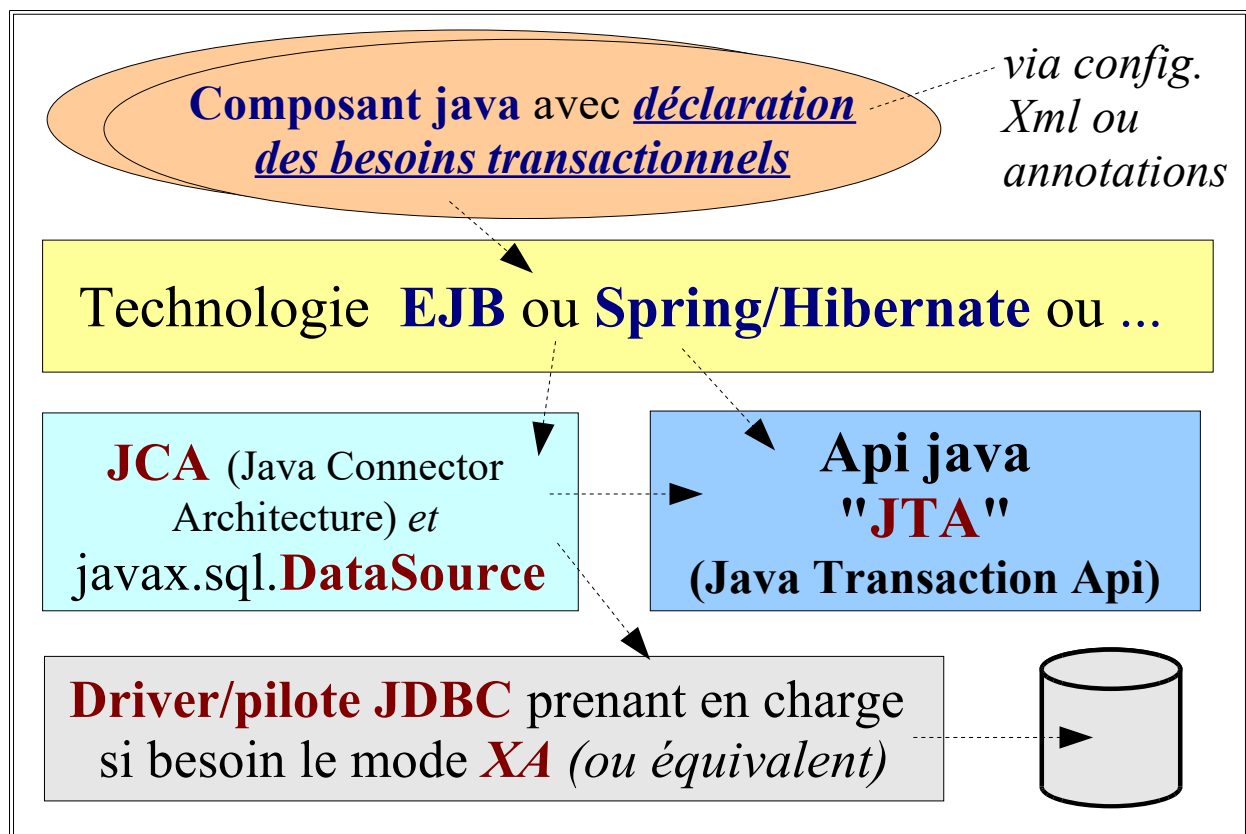
Le framework Spring est capable de gérer (superviser) lui même les transactions devant être menées à bien à partir de certains services applicatifs.

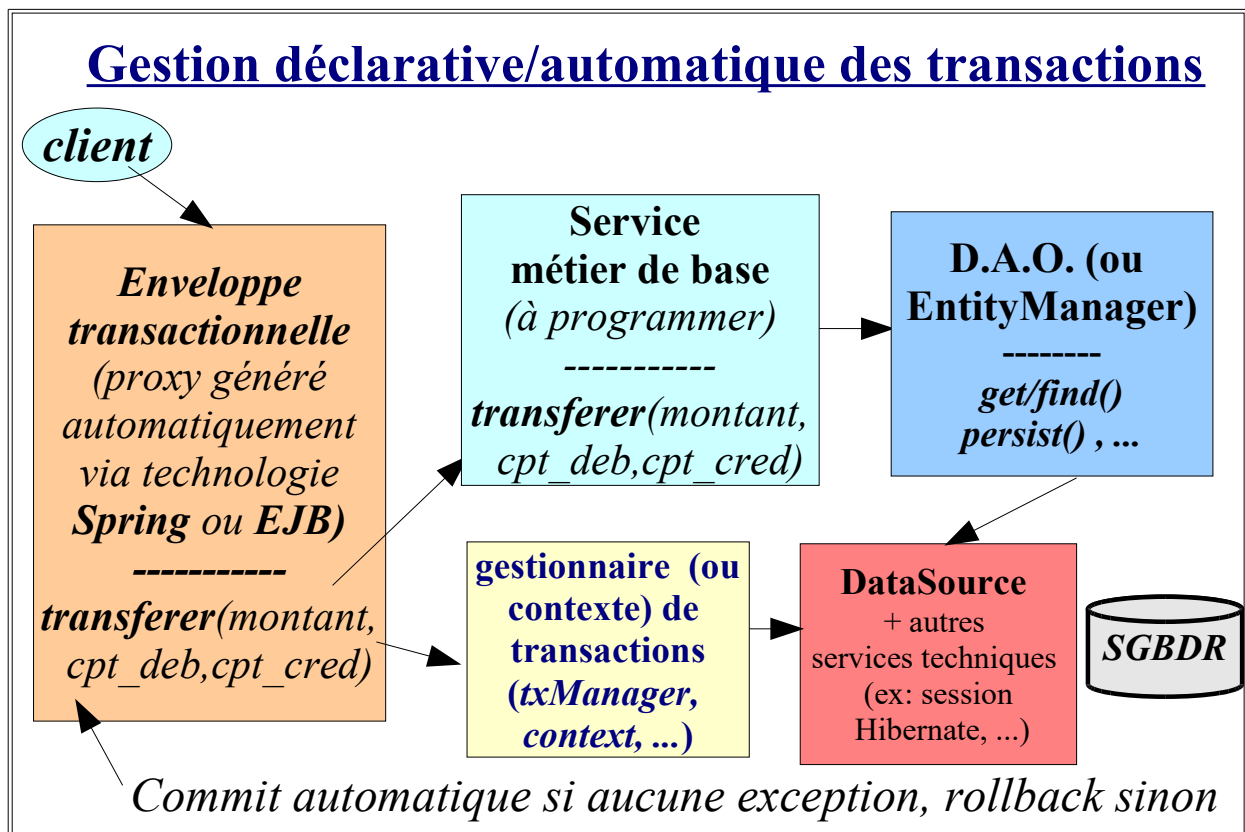
Ceci suppose :

- 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 à peu près de cette teneur:

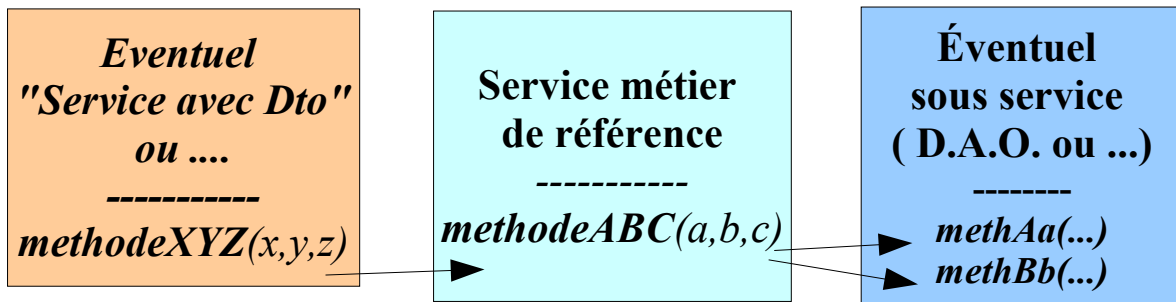
```

public void transférer(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.transférer(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 session Hibernate ou EntityManager JPA
    // (si ouvert en début de cette méthode)
}
}

```

## 2. Propagation du contexte transactionnel et effets

### Propagation du contexte transactionnel

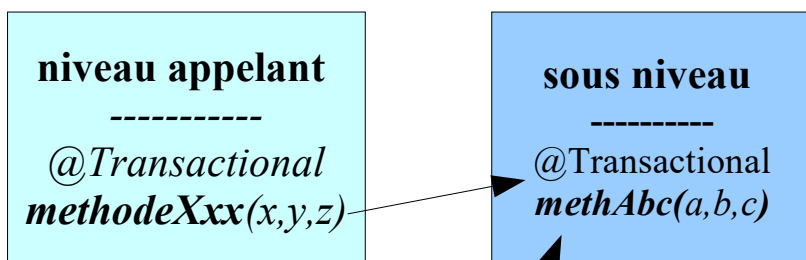


Propagation	Tx en cours ( appelant)	Tx dans sous service
<b>Required</b> (par défaut)	none tx1	new_tx tx1
<b>Support</b>	none tx1	none tx1
<b>Nested</b>	tx1	sub_tx (in tx1)
...		

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

### Effets de `@Transactional` (de Spring)

avec  
propagation  
=**Required**  
(par défaut)



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 ce que j'ai moi même ouvert/initialisé (tx et/ou ...) ou bien sinon: simple `tx.setRollbackOnly()` en cas d'exception locale.

### 3. Configuration du gestionnaire de transactions

#### 3.1. Différentes implémentations de *PlatformTransactionManager*

Principale technologie utilisée au niveau d'un DAO	implémentation de l'interface <i>PlatformTransactionManager</i>
<b>JDBC</b> (jdbcTemplate)	<code>org.springframework.jdbc.datasource.DataSourceTransactionManager</code> avec injection <code>dataSource</code>
<b>JTA</b>	<code>org.springframework.transaction.jta.JtaTransactionManager</code> avec injection <code>dataSource</code>
<b>Hibernate</b> ( <i>en direct</i> )	<code>org.springframework.orm.hibernate3.HibernateTransactionManager</code> ou hibernate4/5 avec injection <code>sessionFactory</code>
<b>JPA</b> (over-hibernate)	<code>org.springframework.orm.jpa.JpaTransactionManager</code> avec injection <code>entityManagerFactory</code>

#### 3.2. Exemple de 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;
    }
}

```



## 4. 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 transférer(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>myJsBean</managed-bean-name>
  <managed-bean-class>myjsf.MyJsBean</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 MyJsBean {

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

### Effets:

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

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

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

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

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

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

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

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

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

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

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

avec dans **pom.xml**

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

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

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

### 3. Intégration de JSF 2 au sein de Spring-boot 2

pom.xml

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

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

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

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

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

```

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

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

```

Dans **application.properties**

```

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

```

### WelcomePageRedirect.java

```

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

@Configuration
public class WelcomePageRedirect implements WebMvcConfigurer {

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

```

**XyMBean.java**

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

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

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

    //@Autowired
    @Inject
    private XyService xyService;

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

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



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

**src/main/resources/META-INF/resources**

### p1.xhtml

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

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

### index.html

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

# 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".

A l'origine (vers les années 2005-2012), "Spring MVC" était à voir comme un petit framework java/web (pour le côté serveur) qui se posait comme une alternative à Struts2 ou JSF2.

Plus récemment, "Spring MVC" (souvent intégré dans SpringBoot) est énormément utilisé pour développer des Web-Services REST et est quelquefois encore un peu utilisé pour générer des pages HTML (via des vues ".jsp" ou bien des vues ".html" de Thymeleaf).

La suite de ce chapitre montrera comment utiliser "Spring MVC" pour générer des pages HTML côté serveur. Les "Web Services REST/ Spring MVC" seront étudiés dans un autre chapitre.

Dépendances maven nécessaires (en intégration moderne "spring-boot"):

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

<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

et (si vues de type ".jsp")

```
<dependency>
  <groupId>org.apache.tomcat.embed</groupId>
  <artifactId>tomcat-embed-jasper</artifactId>
  <scope>provided</scope>
</dependency>

<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
</dependency>
```

ou bien (avec Thymeleaf)

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>

<dependency>
  <groupId>nz.net.ultraq.thymeleaf</groupId>
```

```
<artifactId>thymeleaf-layout-dialect</artifactId>
</dependency>
```

Configuration en version ".jsp":

src/main/resources/application.properties

```
server.servlet.context-path=/myMvcSpringBootApplication
server.port=8080
#spring.mvc.view.prefix=/WEB-INF/view/
spring.mvc.view.prefix=/jsp/
spring.mvc.view.suffix=.jsp
```

Avec cette configuration , un **return "xy"** d'un contrôleur déclenchera l'affichage de la page **/jsp/xy.jsp** et selon la structure du projet , le répertoire **/jsp** sera placé dans **src/main/resources/META-INF/resources** ou ailleurs .

### Exemple élémentaire :

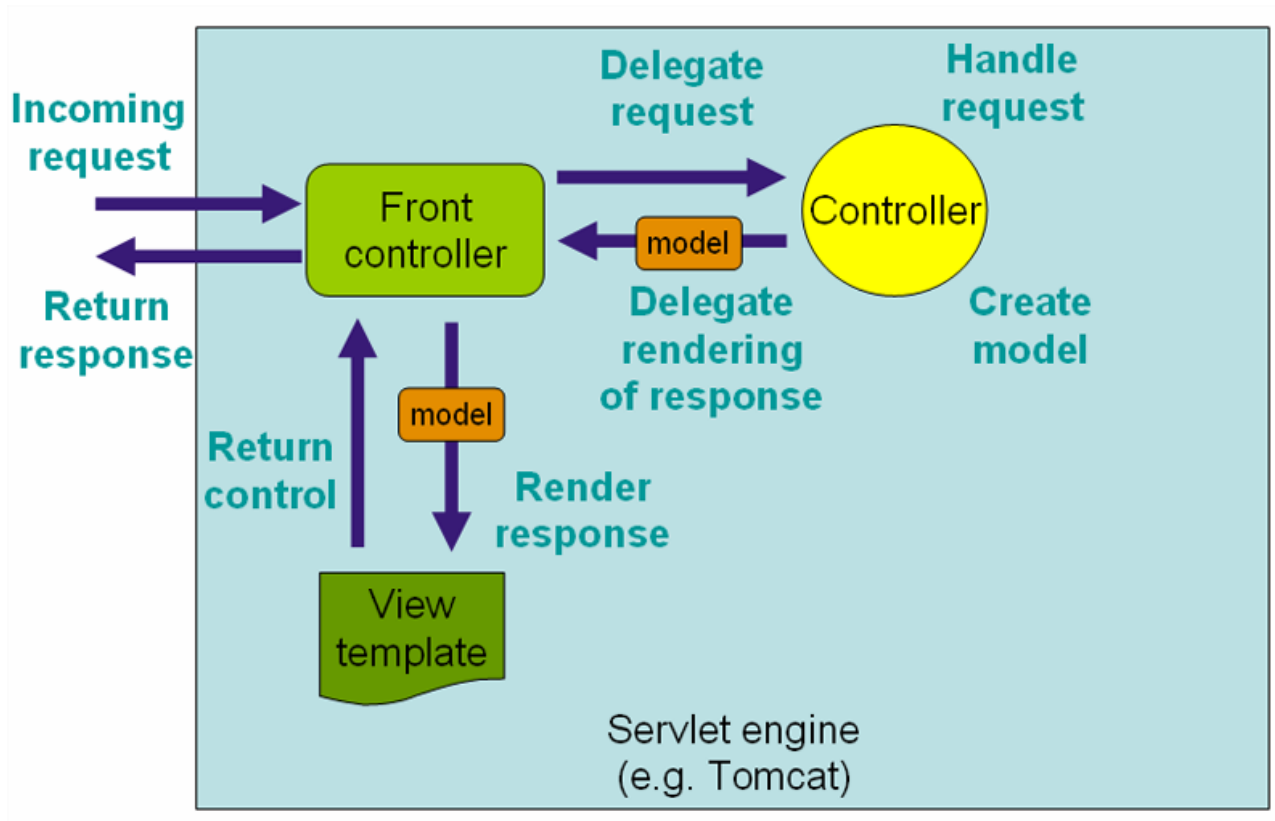
```
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class HelloWorldController {

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

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

```
<html>
<head><title>showMessage</title></head>
<body>
    <p>message=<b>${message}</b></p>
</body>
</html>
```

**Principe de fonctionnement de SpringMvc :****NB.:**

- Le **contrôleur** est une instance d'une classe java préfixée par **@Controller** (composant spring de type contrôleur web) et de **@Scope("singleton")** par défaut .  
Ce contrôleur a la responsabilité de préparer des données (souvent récupérées en base et quelquefois à partir de critères de recherches)
- Le **model** est une table d'association (nomAttribut, valeurAttribut) (par défaut en scope=request) permettant de passer des objets de valeurs à afficher au niveau de la vue.
- La **vue** est responsable d'effectuer un rendu (souvent "html + css + js") à partir des valeurs du modèle.  
La **vue** est souvent une **page JSP** ou bien un **template "Thymeleaf"** .

## 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 (.jsp ou .html thymeleaf)
    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 JSP (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-edeb7b00779a2" />` . Ce champ caché correspond au "Synchronizer Token Pattern" (que l'on retrouve dans les frameworks web concurrents "Stuts" ou "JSF" ) : le côté 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:  ▼  
 cible:  ▼  
 montant:   
  
 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:    
 cible:    
 montant:  **doit être plus grand que 0**   
   
 sommeConvertie=   


---

[retour page accueil](#)

numero: 0   
 nom:    
 prenom:    
 adresse:  **Your address must between 4 and 128 characters**   
 telephone:    
 email:  **Adresse email mal formée**

## 2.7. Spring-mvc avec Thymeleaf

La technologie "Thymeleaf" est une alternative intéressante vis à vis des pages JSP et qui offre les avantages suivants :

- **syntaxe plus développée** (plus concise , plus expressive , plus sophistiquée)
- **meilleures possibilités/fonctionnalités pour la mise en page** (héritage de layout , ...)
- technologie assez souvent utilisée avec SpringMvc et SpringBoot

**Rappel des dépendances maven nécessaires :**

```
<dependency>
```

```

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

    <dependency> <!-- pour @Max , ... @Valid -->
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-validation</artifactId>
    </dependency>
    <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-thymeleaf</artifactId>
    </dependency>

    <dependency>
        <groupId>nz.net.ultraq.thymeleaf</groupId>
        <artifactId>thymeleaf-layout-dialect</artifactId>
    </dependency>

    <!--
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-security</artifactId>
        </dependency>

        <dependency>
            <groupId>org.thymeleaf.extras</groupId>
            <artifactId>thymeleaf-extras-springsecurity5</artifactId>
        </dependency>
    -->

```

Sans configuration spécifique dans application.properties le répertoire prévu pour accueillir les templates de **thymeleaf** est **src/main/resources/templates**.

Sachant que les fichiers annexes ".css", ".js", ... sont à ranger dans **src/main/resources/static**.

```

▼ [Icon] > springSecurityThymeleafApp (in springSecurityThymeleafApp)
  > [Icon] src/main/java
  ▼ [Icon] > src/main/resources
    ▼ [Icon] > static
      > [Icon] > css
      > [Icon] images
      > [Icon] js
    ▼ [Icon] > templates
      [Icon] _footer.html
      [Icon] _header.html
      [Icon] _layout.html
      [Icon] carousel_slideshow.html
      > [Icon] commande.html
      [Icon] comptes.html
      [Icon] error.html

```

Il n'y a pas de différence notable dans l'écriture des contrôleurs (JSP ou Thymeleaf : peu importe).

**Voici quelques exemples de "vues/templates" basés sur la technologie "Thymeleaf" :**

\_header.html

```
<header xmlns:th="http://www.thymeleaf.org" th:fragment="_header">
  <div style="width:100%;font-size:36px;line-height:48px;background-color:navy;color:white">
    My SpringMVC Thymeleaf Application</div> </header>
```

\_footer.html

```
<footer xmlns:th="http://www.thymeleaf.org" th:fragment="_footer">
  <div style="background-color:navy;width:100%;color:white">
    Mon pied de page ... <a th:href="@{/to-welcome}" style="color:yellow" >welcome</a>
  </div> </footer>
```

NB : les *sous fichiers* \_header.html et \_footer.html seront **inclus** dans \_layout.html via **th:replace="..."**

Le fichier *\_layout.html* suivant correspond à un template/**modèle commun/générique de mise en page** . La plupart des pages ordinaires de l'application reprendront (par héritage) la structure de \_layout.html .

Le contenu des zones identifiées par **layout:fragment="nomLogiqueFragment"** pourront si besoin est redéfinies/remplacées au sein des futures pages basées sur ce template :

\_layout.html

```
<html xmlns:th="http://www.thymeleaf.org"
      xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout" >
  <head>
    <meta charset="UTF-8" />
    <title layout:fragment="title" th:utext="${title}"></title>
    <link rel="stylesheet" type="text/css" th:href="@{/css/bootstrap.min.css}" />
    <link rel="stylesheet" type="text/css" th:href="@{/css/styles.css}" />
  </head>
  <body>
    <div class="container-fluid">
      <div th:replace="_header"></div>
      <div layout:fragment="content">
        default content from _layout.html (to override)
      </div>
      <div th:replace="_footer"></div>
    </div><!-- end of bootstrap css container-fluid -->
```

```
</body>
</html>
```

Le fichier *welcome.html* suivant est basé sur le modèle générique *\_layout.html* via le lien d'héritage / de composition **layout:decorate**="**~{\_layout}**".

Au sein de *welcome.html*, tout le contenu imbriqué entre début et fin de la balise marquée via **layout:fragment**="**content**" va automatiquement remplacer le texte *default content from \_layout.html (to override)* qui était encadré par la même nom logique de fragment au sein de *\_layout.html*.

Le rendu globalement fabriqué par Thymeleaf sera ainsi une page HTML complète ayant comme structure celle de *\_layout.html* (et donc avec *\_header* et *\_footer* par défaut) et dont le fragment "content" aura été redéfini avec un contenu spécifique à *welcome.html*.

#### welcome.html

```
<div xmlns:th="http://www.thymeleaf.org"
    xmlns:sec="http://www.thymeleaf.org/extras/spring-security"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{_layout}" layout:fragment="content">
  <h1>Welcome Thymeleaf (public part)</h1>
  <p>message=<b><span th:utext="${message}"></span></b></p>
  <hr/> ... </div>
```

## My SpringMVC Thymeleaf Application

### Welcome Thymeleaf (public part)

message=**bienvenu(e)**

[nouveau client](#)  
[welcome-authenticated with loginSpringSecurity.html automatic hook \(client or admin\)](#)  
[update commande](#)  
[exemple ajax](#)  
[exemple carousel](#)

[fin de session / deconnexion](#)  
 num session http/jee= F4386246590C8C7FD57CC251B4AB40C0

valid accounts (dev): customer(1,pwd1), customer(2,pwd2), admin(superAdmin,007)

Mon pied de page ... **welcome**

Exemple de formulaire simple avec thymeleaf :

```
<!DOCTYPE HTML>
```

```
<div xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{_layout}"
    layout:fragment="content">
<h3>nouveau (client banque)</h3>
    <hr/>
    <form th:action="@{/nouveauClient}"
        th:object="${client}" method="POST">
        numero : <input th:field="*{numero}" type="text" /> <br/>
        password : <input th:field="*{password}" type="text" /> <br/>
        nom : <input th:field="*{nom}" type="text" /> <br/>
        prenom : <input th:field="*{prenom}" type="text" /> <br/>
        <input type="submit" value="enregistrer nouveau client" /> <br/>
    </form>
</div>
```

pour déclencher :

```
@RequestMapping(value="/nouveauClient" )
public String nouveauClient(Model model,
    @ModelAttribute("client") Client client) {
    //client avec propriétés .getNom() , .getPrenom() , ...
}
```



## Exemple partiel de formulaire complexe avec thymeleaf :

```

<div xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{_layout}" layout:fragment="content">
<script>    function onDelete(idToDelete,bToDelete){ /* .... */ } </script>
<h3>commande</h3>
<hr/>
<form th:action="@{/update-commande}" th:object="${cmdeF}" method="POST">
    numero : <label th:text="${cmde.numero}" ></label>
        <input th:field="${cmde.numero}" type="hidden" /><br/>
    sDate : <input th:field="${cmde.sDate}" type="text" /> <br/>
    id (client) : <label th:text="${cmde.client.id}" ></label>
        <input th:field="${cmde.client.id}" type="hidden" /><br/>
    nom (client) : <input th:field="${cmde.client.nom}" type="text"
        th:class="${cmde.client.nom == 'Bon' ? 'enEvidence' : ''} " /> <br/>
    prenom (client): <input th:field="${cmde.client.prenom}" type="text" /> <br/>
    <div th:each="p, rowStat : ${cmde.produits}">
<hr/>
        ref (produit) : <label th:text="${cmde.produits[__${rowStat.index}__].ref}" ></label>
            <input th:field="${cmde.produits[__${rowStat.index}__].ref}" type="hidden" /><br/>
        label (produit) : <input type="text" th:field="${cmde.produits[__${rowStat.index}__].label}" />
        prix (produit) : <input type="text" th:field="${cmde.produits[__${rowStat.index}__].prix}" />
        : <input type="checkbox" value="delete"
            th:onclick="'onDelete(' + ${cmde.produits[__${rowStat.index}__].ref} + ',this.checked)' " /> delete

    </div>
<hr/>
    nb new product to add : <input th:field="${prodActions.nbNew}" type="text" /> <br/>
    id of product(s) to delete : <input th:field="${prodActions.idsToDelete}"
        class="RedCssClass" type="text" /> <br/>
    <input type="submit" value="update commande" /> <br/>
</form>
</div>

```

numero : 1

sDate :

id (client) : 1

nom (client) :

prenom (client):

---

ref (produit) : 1

label (produit) :  prix (produit) :  : ☒ delete

---

ref (produit) : 2

label (produit) :  prix (produit) :  : ☒ delete

---

ref (produit) : 3

label (produit) :  prix (produit) :  : ☐ delete

---

nb new product to add :

id of product(s) to delete :

Si thymeleaf est utilisé conjointement avec **spring-security** alors les éléments suivants peuvent être utiles :

#### @ControllerAdvice

```
public class ErrorCtrlAdvice {
    private static Logger logger = LoggerFactory.getLogger(ErrorController.class);

    @ExceptionHandler(Throwable.class)
    @ResponseStatus(HttpStatus.INTERNAL_SERVER_ERROR)
    public String exception(final Throwable throwable, final Model model) {
        logger.error("Exception during execution of SpringSecurity application", throwable);
        String errorMessage = (throwable != null ? throwable.getMessage() : "Unknown error");
        model.addAttribute("errorMessage", errorMessage);
        return "error";
    }
}
```

#### error.html

```
<!DOCTYPE HTML>
<div xmlns:th="http://www.thymeleaf.org"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{_layout}"    layout:fragment="content">

    <div th:with="httpStatus=${
        {T(org.springframework.http.HttpStatus).valueOf(#response.status)}">
        <h3 th:text="|${httpStatus} - ${httpStatus.reasonPhrase}|">404</h3>
        <p th:utext="${errorMessage}">Error java.lang.NullPointerException</p>
        <a href="welcome.html" th:href="@{/to-welcome}">Back to Home Page (welcome)</a>
    </div>
</div>
```

## 500 INTERNAL\_SERVER\_ERROR - Internal Server Error

Accès refusé

[Back to Home Page \(welcome\)](#)

```

<!-- necessite thymeleaf-extras-springsecurity5 dans pom.xml -->
<div xmlns:th="http://www.thymeleaf.org"
    xmlns:sec="http://www.thymeleaf.org/extras/spring-security"
    xmlns:layout="http://www.ultraq.net.nz/thymeleaf/layout"
    layout:decorate="~{_layout}" layout:fragment="content">
<h1>Welcome Thymeleaf (for Authenticated user)</h1>
<p>message=<b><span th:utext="{message}"></span></b></p>
<hr/>
<div sec:authorize="isAuthenticated()">
    <h3>authenticated user </h3>
    Logged user: <span sec:authentication="name">Unknown</span> <br/>
    Roles: <span sec:authentication="principal.authorities">[]</span> <br/>
</div>
<hr/>
....
</div>

```

## authenticated user

Logged user: superAdmin

Roles: [ROLE\_ADMIN]

Quelques liens hypertextes pour approfondir "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 , ....) .

#### Dépendances "maven" sans spring-boot :

```
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>5.2.3.RELEASE</version>
    <scope>compile</scope>
</dependency>

<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.10.2</version> <!-- to produces json -->
</dependency>
```

...

#### Dépendances "maven" indirecte (avec spring-boot) :

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

Dans *application.properties* :

```
server.servlet.context-path=/webappXy ou ...
server.port=8181 ou 8080 ou ...
```

## 4. WS REST via Spring MVC et @RestController

L'annotation fondamentale **@RestController** (héritant de @Controller et de @Component) déclare que la classe ....RestCtrl correspond à l'implémentation "spring-mvc" d'un composant de l'application de type "Contrôleur de Web Service REST" .

On a par défaut @ResponseBody avec @RestController et cela signifie que la valeur de retour d'une des méthodes publiques du contrôleur sera quasi directement renvoyée au client http (sans passer par une page JSP ni un autre type de vue) .

Cependant , Lorsque la valeur de retour sera un *objet java* , *celui ci sera automatiquement transformé en JSON* (ou autre) avant d'être retourné au client http (ex : code js / appel ajax )

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/devise" , headers="Accept=application/json")
public class DeviseJsonRestCtrl {

    @Autowired //ou @Inject
    private GestionDevises gestionDevises; //internal business service or DAO

    //RECHERCHE UNIQUE selon RESOURCE-ID:
    //URL de déclenchement: .../webappXy/rest/devise/EUR
    @RequestMapping(value="/{codeDevise}" , method=RequestMethod.GET)
    public Devise getDeviseByName(@PathVariable("codeDevise") String codeDevise) {
        return gestionDevises.getDeviseByPk(codeDevise);
    }

    //RECHERCHE MULTIPLE :
    //URL de déclenchement: webappXy/rest/devise
    //ou webappXy/rest/devise?name=euro
    @RequestMapping(value="", method=RequestMethod.GET)
    public 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;
        }
    }
}
```

NB :

**@RequestParam** avec required=false si paramètre facultatif en fin d'URL

Si l'ensemble de la classe java préfixée par @RestController comporte

**@RequestMapping**(value="..." , headers="Accept=application/json")

alors par défaut les valeurs en retour des méthodes publiques préfixées par **@RequestMapping** seront automatiquement converties au format **JSON** (en s'appuyant en interne sur la technologie *jackson-databind*) .

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

```
//URL : webappXy/rest/devise/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);
}
```

==> L'exemple ci-dessus est très déconseillé sur une api REST .

Un format de retour homogène (XML ou très souvent JSON) est en général attendu à la place .

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

NB : il est techniquement possible de convertir explicitement une "Json String" en objet java via l'api "jackson" comme le montre l'exemple inutilement long suivant (à ne pas reproduire , juste pour montrer certains mécanismes internes):

```
...
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 DeviseJsonRestController {
    ...
    @RequestMapping(value="" , method=RequestMethod.PUT )
    Devise updateDevise(@RequestBody String deviseAsString) {
        Devise devise=null;
        try {
            ObjectMapper jacksonMapper = new ObjectMapper();
            jacksonMapper.configure(
                DeserializationFeature.FAIL_ON_UNKNOWN_PROPERTIES, false);
            devise = jacksonMapper.readValue(deviseAsString,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, à reproduire) assure les mêmes fonctionnalités :

```
@RestController
```

```

@RequestMapping(value="/rest/devise" , headers="Accept=application/json")
public class DeviseJsonRestController {
...
    @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 peut éventuellement 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);
    if(dev!=null)
        return new ResponseEntity<Devise>(dev, HttpStatus.OK);
    else
        return new ResponseEntity<Devise>(HttpStatus.NOT_FOUND); //404
}

```

ou bien

```

ResponseEntity< ?> getDeviseByName(...){
....
    else
        return new ResponseEntity<String> (" { \"err\" : \"devise not found\" } ",
            HttpStatus.NOT_FOUND ); //404
}

```

Autre exemple (ici en mode **DELETE**) :

```
//url : http://localhost:8181/webappXy/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
    }
}
```

NB : Bien que très finement paramétrable , un **return new ResponseEntity<?>** sera **généralement moins bien** qu'un un simple **throw new ...ClasseExceptionPréfixéePar\_@ResponseStatus** plus simple et plus efficace (vu dans le paragraphe ci-après)

Eventuelles variations (simplifications):

**@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) .

## 4.2. @ResponseStatus

Dans le cadre d'une remontée d'exception personnalisée il est possible de préciser le statut HTTP (pas systématiquement 500) qui sera remonté via l'annotation **@ResponseStatus ( )**

Exemple :

```
@ResponseStatus(HttpStatus.NOT_FOUND) //404
public class MyEntityNotFoundException extends RuntimeException {
    public MyEntityNotFoundException() {
    }
    public MyEntityNotFoundException(String message) {
        super(message);
    }
    public MyEntityNotFoundException(Throwable cause) {
        super(cause);
    }
    public MyEntityNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    ...
}
```

.../...

```
@RequestMapping(value="/{codeDevise}" , method=RequestMethod.DELETE)
public void deleteDeviseByCode(@PathVariable("codeDevise")
                               String codeDevise) throws MyEntityNotFoundException {
    try {
        deviseService.deleteByCode(codeDevise);
    } catch (Exception e) {
```

```

        logger.error(e.getMessage());
        throw new MyEntityNotFoundException(
            "echec suppression devise pour codeDevise="+codeDevise ,e);
    }
}

```

Un appel HTTP avec une URL finissant (avec une erreur ici volontaire) par `"/devise/EURy"`

---> renvoie **404** et un message d'erreur au format JSON/spring-Web-MCV HOMOGÈNE :

```

{
  "timestamp": "2020-02-03T17:23:45.888+0000",
  "status": 404,
  "error": "Not Found",
  "message": "echec suppression devise pour codeDevise=EURy",
  "trace": "org.mycontrib.backend.exception.MyEntityNotFoundException:.....",
  "path": "/spring-boot-backend/rest/devise-api/private/role_admin/devise/EURy"
}

```

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 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;
}

```

### 4.3. Exemples d'appels en js/ajax

**js/ajax-util.js**

```

//fonction utilitaire pour preparer xhr en vu d'effectuer juste apres un appel ajax en mode Get ou post ou ...
function initXhrWithCallback(callback,errCallback){

```

```

var xhr = new XMLHttpRequest();
xhr.onreadystatechange = function() {
    if (xhr.readyState == 4){
        if (xhr.status == 200 || xhr.status == 0) {
            callback(xhr.responseText,xhr);
        }
        else {
            errCallback(xhr);
        }
    }
};
return xhr;
}

function xhrStatusToErrorMessage(xhr){
    var errMsg = "ajax error";//by default
    var detailsMsg=""; //by default
    console.log("xhr.status="+xhr.status);
    if(xhr.responseText!=null)
        detailsMsg = xhr.responseText;
    switch(xhr.status){
        case 400 :
            errMsg = "Server understood the request, but request content was invalid."; break;
        case 401 :
            errMsg = "Unauthorized access (401)"; break;
        case 403 :
            errMsg = "Forbidden resource can't be accessed (403)"; break;
        case 404 :
            errMsg = "resource not found (404)"; break;
        case 500 :
            errMsg = "Internal server error (500)"; break;
        case 503 :
            errMsg = "Service unavailable (503)"; break;
    }
    return errMsg+" "+detailsMsg;
}

function makeAjaxGetRequest(xhr,url) {
    xhr.open("GET", url, true);
    xhr.send(null);
}

```

```
function makeAjaxPostRequest(xhr,url,jsonData) {
    xhr.open("POST", url, true);
    xhr.setRequestHeader("Content-Type", "application/json");
    //pour re-vehiculer (si necessaire) un jeton d'authentification (jwt ou pas):
    var authToken = sessionStorage.getItem("authToken");
    if(authToken != null ){
        xhr.setRequestHeader('Authorization','Bearer '+ authToken);
    }
    xhr.send(jsonData);
}
```

username :

password :

roles :

login successful with roles=admin

### login.html

```
<html>
<head> <title>login</title><script src="js/ajax-util.js"></script>  <script src="js/login.js"></script>
</head>
<body>
    <h3> login (ws security) </h3>
    username : <input id="txtUsername" type='text' value="admin1"/><br/>
    password : <input id="txtPassword" type='text' value="pwdadmin1"/><br/>
    roles : <input id="txtRoles" type='text' value="admin"/><br/>
    <input type='button' value="login" id="btnLogin"/> <br/>
    <span id="spanMsg"></span> <br/>
    <hr/> <a href="index.html">retour vers index.html</a>
</body>
</html>
```

### js/login.js

```
window.onload=function(){
    var spanMsg = document.querySelector('#spanMsg');
    var btnLogin=document.querySelector('#btnLogin');
    btnLogin.addEventListener("click" , function (){
        var auth = { username : null, password : null , roles : null } ;
        auth.username = document.querySelector('#txtUsername').value;
```

```

auth.password = document.querySelector('#txtPassword').value;
auth.roles = document.querySelector('#txtRoles').value;

var cbLogin = function(data,xhr){
    console.log(data); //data as json string;
    var authResponse = JSON.parse(data);
    if(authResponse.status){
        spanMsg.innerHTML=authResponse.message + " with roles=" + authResponse.roles;
        //localStorage.setItem("authToken",authResponse.token);
        sessionStorage.setItem("authToken",authResponse.token);
    } else{
        spanMsg.innerHTML=authResponse.message ;
    }
} //end of cbLogin

var cbError = function(xhr){
    spanMsg.innerHTML= xhrStatusToErrorMessage(xhr) ;
}

var xhr = initXhrWithCallback(cbLogin,cbError);
makeAjaxPostRequest(xhr,"./rest/login-api/public/auth" , JSON.stringify(auth));

}); //end of btnLogin.addEventListener/click
} //end of window.onload

```

## recherche devises selon taux mini (public)

changeMini :

- Euro , 1
- Dollar , 1.1243
- Yen , 121.6477

## ajout de monnaie (after logging as ADMIN)

codeMonnaie:  (ex: EUR,USD,...)

nommonnaie:  (ex: euro,dollar,...)

tauxChange:  (ex: 1, 0.85 , 1.5, ... )

{"code":"ms","name":"monnaieSinge","change":1.23456}

**appel\_ajax.html**

```

<html>
<head>
    <script src="js/ajax-util.js"></script>    <script src="js/appelAjax.js"></script>
    <meta charset="UTF-8"> <title>appel_ajax</title>
</head>
<body>
    <h3>recherche devises selon taux mini (public)</h3>
    changeMini : <input type="text" id="txtChangeMini" value="1"/> <br/>
                <input type="button" value="getDevises" id="btnGetDevises" /> <br/>
    <div id="divRes"></div>

    <h3> ajout de monnaie (after logging as ADMIN)</h3>
    codeMonnaie: <input type="text" id="txtCode" value="ms" /> (ex: EUR,USD,...)<br/>
    nommonnaie: <input type="text" id="txtName" value="monnaieSinge" /> (ex: euro,dollar,...)<br/>
    tauxChange: <input type="text" id="txtChange" value="1.23456" /> (ex: 1, 0.85 , 1.5, ... )<br/>
    <input type="button" id="btnPostDevise" value="sauvegarder devise" /> <br/>
    <div id="divMessage"></div>
    <hr/>
    <a href="index.html">retour index.html</a>
</body>
</html>

```

**js/appelAjax.js**

```

window.onload=function(){
    var inputChangeMini = document.querySelector("#txtChangeMini");
    var btnGetDevises = document.querySelector("#btnGetDevises");
    var btnPostDevise = document.querySelector("#btnPostDevise");
    var divRes = document.querySelector("#divRes");
    var divMessage = document.querySelector("#divMessage");
    var cbError = function(xhr){
        divMessage.innerHTML= xhrStatusToErrorMessage(xhr) ;
    }
    btnGetDevises.addEventListener("click" , function (){
        var changeMini = inputChangeMini.value;
        var cbAffDevises=function(texteReponse,xhr){
            //divRes.innerHTML = texteReponse;
            var listeDeviseJs = JSON.parse(texteReponse /* au format json string */)
            var htmlListeDevises = "<ul>" ;
            for(i=0; i<listeDeviseJs.length ; i++){

```

```

        htmlListeDevises = htmlListeDevises + "<li>" + listeDeviseJs[i].name + " , "
                                + listeDeviseJs[i].change + "</li>";
    }
    htmlListeDevises = htmlListeDevises + "</ul>";
    divRes.innerHTML= htmlListeDevises;
}
var xhr = initXhrWithCallback(cbAffDevises , cbError);
makeAjaxGetRequest(xhr,"./rest/devise-api/public/devise?changeMini="+changeMini );
});//end of btnGetDevises.addEventListener/"click"

btnPostDevise.addEventListener("click" , function (){
    var nouvelleDevise = {   code : null,   name : null, change : null   };
    nouvelleDevise.code = document.querySelector("#txtCode").value;
    nouvelleDevise.name = document.querySelector("#txtName").value;
    nouvelleDevise.change = document.querySelector("#txtChange").value;
    var cbGererResultatPostDevise = function (texteReponse,xhr){
        divMessage.innerHTML= texteReponse;
    }
    var xhr = initXhrWithCallback(cbGererResultatPostDevise, cbError);
    makeAjaxPostRequest(xhr,"./rest/devise-api/private/role_admin/devise" ,
                        JSON.stringify(nouvelleDevise));
});//end of btnGetDevises.addEventListener/"click"
} //end of window.onload

```

## 4.4. 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 resultAsString = restTemplate.getForObject(uri, String.class);
        logger.info("json string of spectacle 1 via rest: " + resultAsString);
        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 resultAsString = restTemplate.getForObject(uri, String.class);
        logger.info("json listeVols via rest: " + resultAsString);
        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);
}
}
}

```

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

### 1.1. Ancien mode de configuration (xml)

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

Dans pom.xml , **spring-security-core** , **spring-security-web** et **spring-security-config** avec des versions un peu décalées par rapport à spring-framework :

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

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

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

Et la configuration xml/spring de l'application faisait généralement référence à un sous fichier importé **security-config.xml** comportant :

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

Le préfixe (par défaut) attendu pour les rôles est "ROLE\_" .

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

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

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

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

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

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

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

```
....
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
@Configuration
public class MySecurity {
    @Bean
    public BCryptPasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder(); //By default since spring 5
    }
}
```

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

```
package .....;

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

### @Configuration

**public class WebSecurityConfig extends WebSecurityConfigurerAdapter {**

**@Autowired**

private BCryptPasswordEncoder passwordEncoder;

**@Autowired**

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

**@Override**

**@Bean**

public AuthenticationManager **authenticationManagerBean**() throws Exception {  
 return super.authenticationManagerBean();  
}

**@Override**

protected void **configure**(final **HttpSecurity** http) throws Exception {  
 /\*

*// config pour Spring-mvc avec pages jsp :*

*http.authorizeRequests()*  
*.antMatchers("/",*  
*"/favicon.ico",*  
*"/\*\*/\*.png",*  
*"/\*\*/\*.gif",*  
*"/\*\*/\*.svg",*  
*"/\*\*/\*.jpg",*  
*"/\*\*/\*.html",*  
*"/\*\*/\*.css",*  
*"/\*\*/\*.js").permitAll()*  
*.anyRequest().authenticated()*  
*.and()*  
*.formLogin().permitAll()*  
*.and().csrf();*

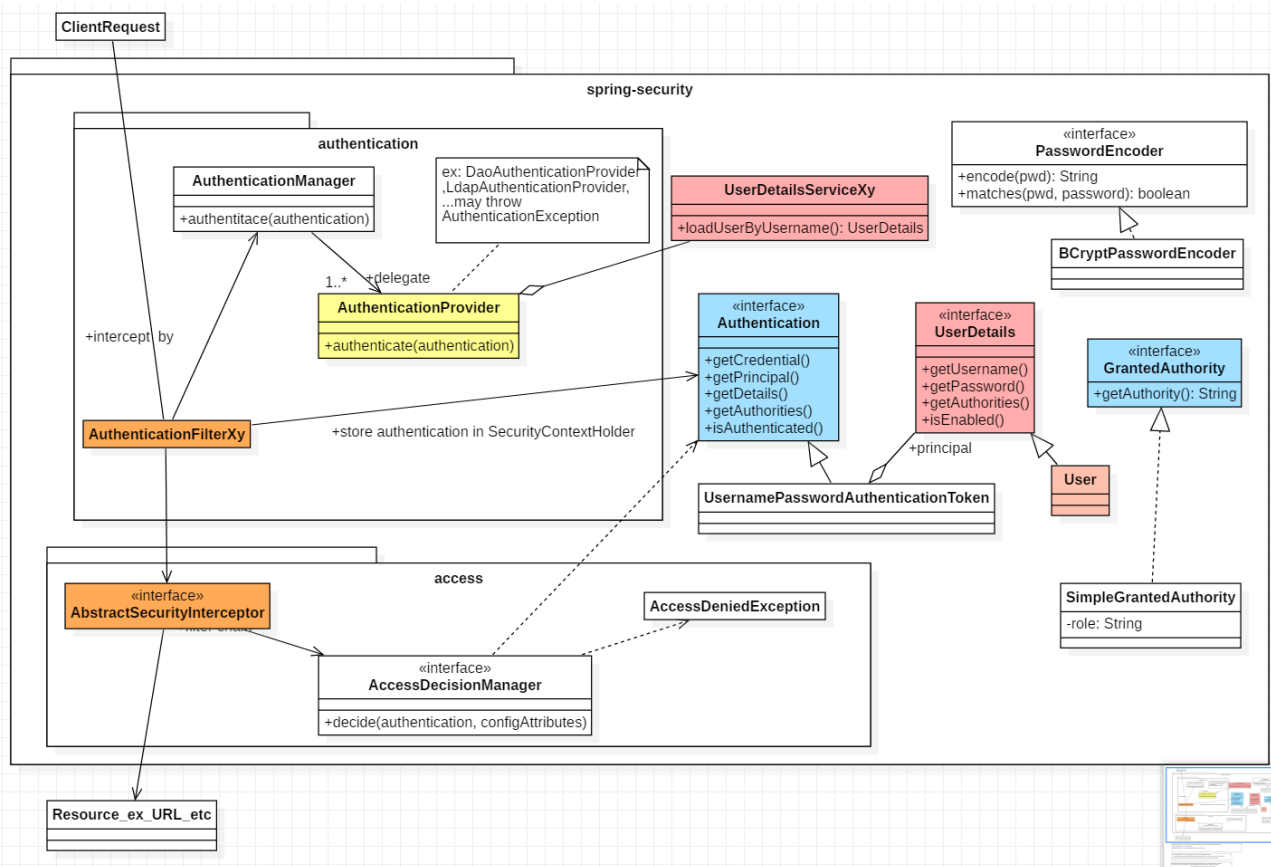
*\*/*

*// config pour Spring-mvc avec WS-REST et tokens (ex : jwt) :*

**http.authorizeRequests()**  
*//.antMatchers("/rest/\*\*").permitAll()*  
**.anyRequest().permitAll()**  
*//.anyRequest().authenticated()*  
*//.anyRequest().hasRole("ADMIN")*  
**.and().csrf().disable();**  
*//.and().httpBasic()*

**}**  
**}**

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



*NB:* [https://en.wikipedia.org/wiki/Spring\\_Security](https://en.wikipedia.org/wiki/Spring_Security) comporte un assez bon schéma montrant les mécanismes fondamentaux de spring-security.

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

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

**Authentication *authenticate*(Authentication authentication) throws AuthenticationException;**

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

`getCredential()` retournant password ou autre.

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

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

**UserDetails *loadUserByUsername*(String username) throws UsernameNotFoundException;**



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

Ces infos "utilisateur" doivent être une implémentation de l'**interface "UserDetails"** (classe "User" par exemple). L'objet "User" (ou un équivalent implémentant "UserDetails") est censée comporter le bon mot de passe.

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

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

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

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

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

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

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

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

"SimpleGrantedAuthority" comporte un nom de rôle (ex "ROLE\_ADMIN" ou "ROLE\_USER" , ...)

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

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

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

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

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

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

L'instruction

```
JdbcUserDetailsManagerConfigurer jdbcUserDetailsManagerConfigurer =  
    auth.jdbcAuthentication().dataSource(realmDataSource);
```

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

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

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

*JdbcAppDbGlobalUserDetailsConfig.java* à adapter au contexte

```
package org.mycontrib.generic.security.config;  
import java.sql.Connection;  
import java.sql.DatabaseMetaData;  
import java.sql.ResultSet;  
import javax.sql.DataSource;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.context.annotation.Configuration;  
import org.springframework.context.annotation.Profile;  
import org.springframework.jdbc.datasource.DriverManagerDataSource;  
import org.springframework.security.config.annotation.authentication.builders.AuthenticationManagerBuilder;  
import org.springframework.security.config.annotation.authentication.configurers.provisioning.JdbcUserDetailsManagerConfigurer;  
import org.springframework.security.config.annotation.authentication.configurers.provisioning.UserDetailsManagerConfigurer;  
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;  
  
@Configuration  
//@Profile("appDbSecurity") //with jdbc  
public class JdbcAppDbGlobalUserDetailsConfig {  
    @Autowired  
    private BCryptPasswordEncoder passwordEncoder;  
  
    private static DataSource realmDataSource;  
  
    private static void initRealmDataSource() {  
        DriverManagerDataSource driverManagerDataSource = new DriverManagerDataSource();  
        driverManagerDataSource.setDriverClassName("org.h2.Driver");  
        driverManagerDataSource.setUrl("jdbc:h2:~/realmdb");  
        driverManagerDataSource.setUsername("sa");  
        driverManagerDataSource.setPassword("");  
        realmDataSource = driverManagerDataSource;  
    }  
  
    private boolean isRealmSchemaInitialized() {  
        int nbExistingTablesOfRealmSchema = 0;  
        try {  
            Connection cn = realmDataSource.getConnection();  
            DatabaseMetaData meta = cn.getMetaData();  
            String tabOfType[] = {"TABLE"};  
            ResultSet rs = meta.getTables(null,null,"%",tabOfType);  
            while(rs.next()){
```

```

        String existingTableName = rs.getString(3);
        if(existingTableName.equalsIgnoreCase("users")
            || existingTableName.equalsIgnoreCase("authorities")) {
            nbExistingTablesOfRealmSchema++;
        }
    }
    rs.close();
    cn.close();
} catch (Exception e) {
    e.printStackTrace();
}
return (nbExistingTablesOfRealmSchema>=2);
}

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

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

```

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

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

Exemple :

```
package .....;

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

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

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

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

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

```

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

```

Enregistrement de cette classe au sein d'un WebSecurityConfigurerAdapter :

```

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

    @Autowired
    private MyUserDetailsService myUserDetailsService;

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

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

```

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

```

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

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

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

```

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

```

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

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

```

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

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

# ANNEXES



# XIII - Annexe – Spring Initializer

## 1. Spring-Initializer



Spring Initializr  
Bootstrap your application

"Spring Initializer" ( <https://start.spring.io/> ) est une application web en ligne disponible publiquement sur internet et qui permet de construire un point de départ d'une nouvelle application basé sur spring-boot .

The screenshot shows the Spring Initializr web application interface. The left sidebar contains the following sections: Project, Language, Spring Boot, Project Metadata, and Dependencies. The main content area has the following fields and options:

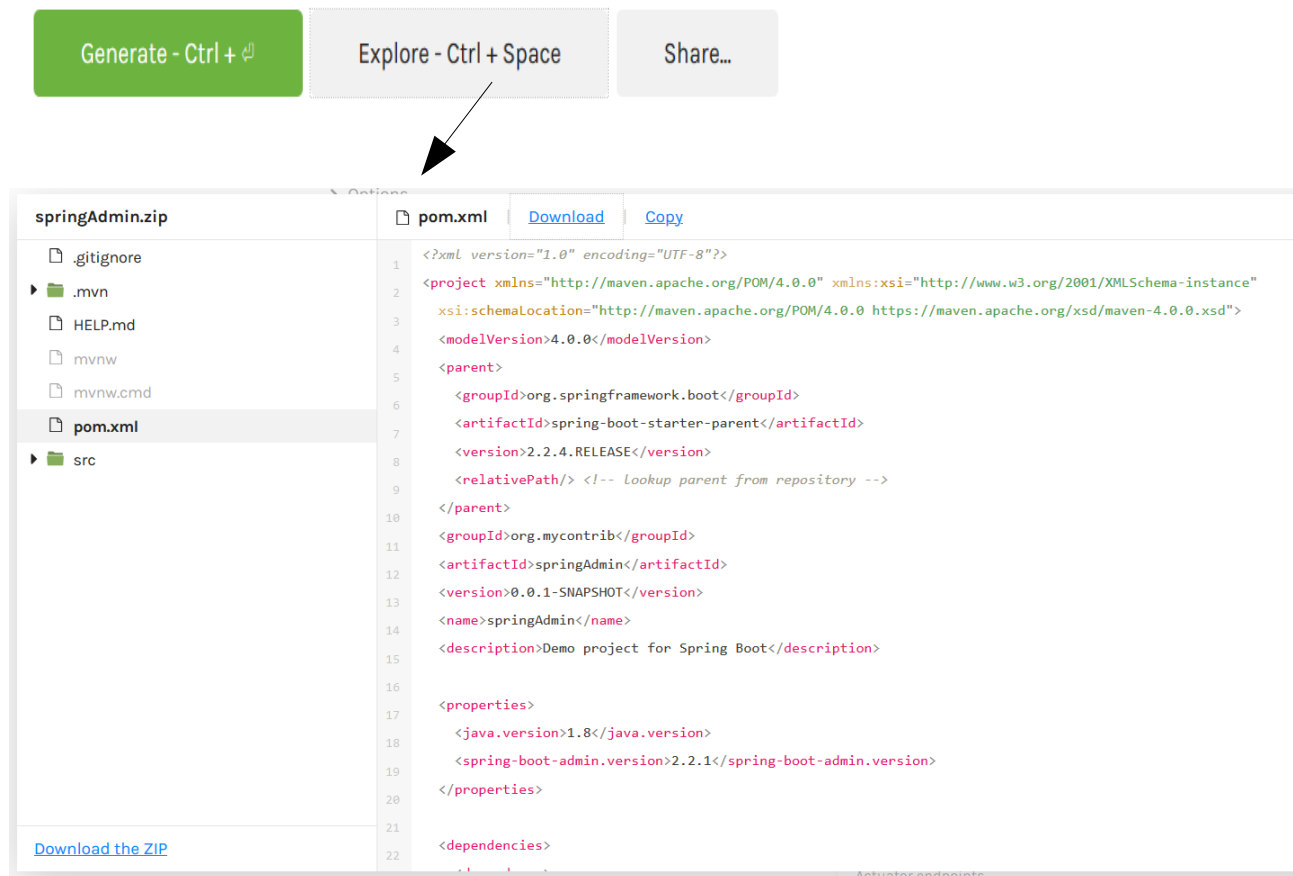
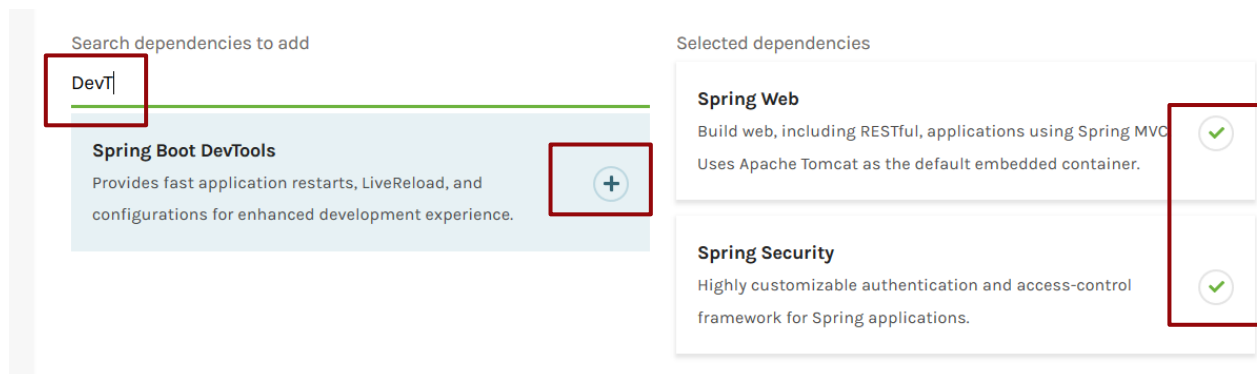
- Project:** Maven Project (highlighted), Gradle Project
- Language:** Java (highlighted), Kotlin, Groovy
- Spring Boot:** 2.3.0 M1, 2.3.0 (SNAPSHOT), 2.2.5 (SNAPSHOT), 2.2.4 (highlighted), 2.1.13 (SNAPSHOT), 2.1.12
- Project Metadata:**
  - Group: com.example
  - Artifact: demo
  - > Options
- Dependencies:**
  - Search dependencies to add: Web, Security, JPA, Actuator, Devtools... (highlighted)
  - Selected dependencies: No dependency selected

At the bottom, there are three buttons: Generate - Ctrl + G (highlighted with an arrow), Explore - Ctrl + Space, and Share... A red box highlights the text "voir selections page suivante" below the dependencies section.

génère **demo.zip** avec dans le sous répertoire "demo" ou autre (selon choix "artifactId") :

- **pom.xml** (avec les "starters" sélectionnés)
- **src/main/resources/application.properties**
- **src/main/java/.../.../DemoApplication** (avec main())
- **src/test/java/.../.../DemoApplicationTest** (avec JUnit)

**NB : le contenu initial de pom.xml dépendra essentiellement de la sélection des technologies/starters :**



# XIV - Annexe – Ancienne config. XML / Spring

## 1. Configuration xml de Spring

### 1.1. Fichier(s) de configuration

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

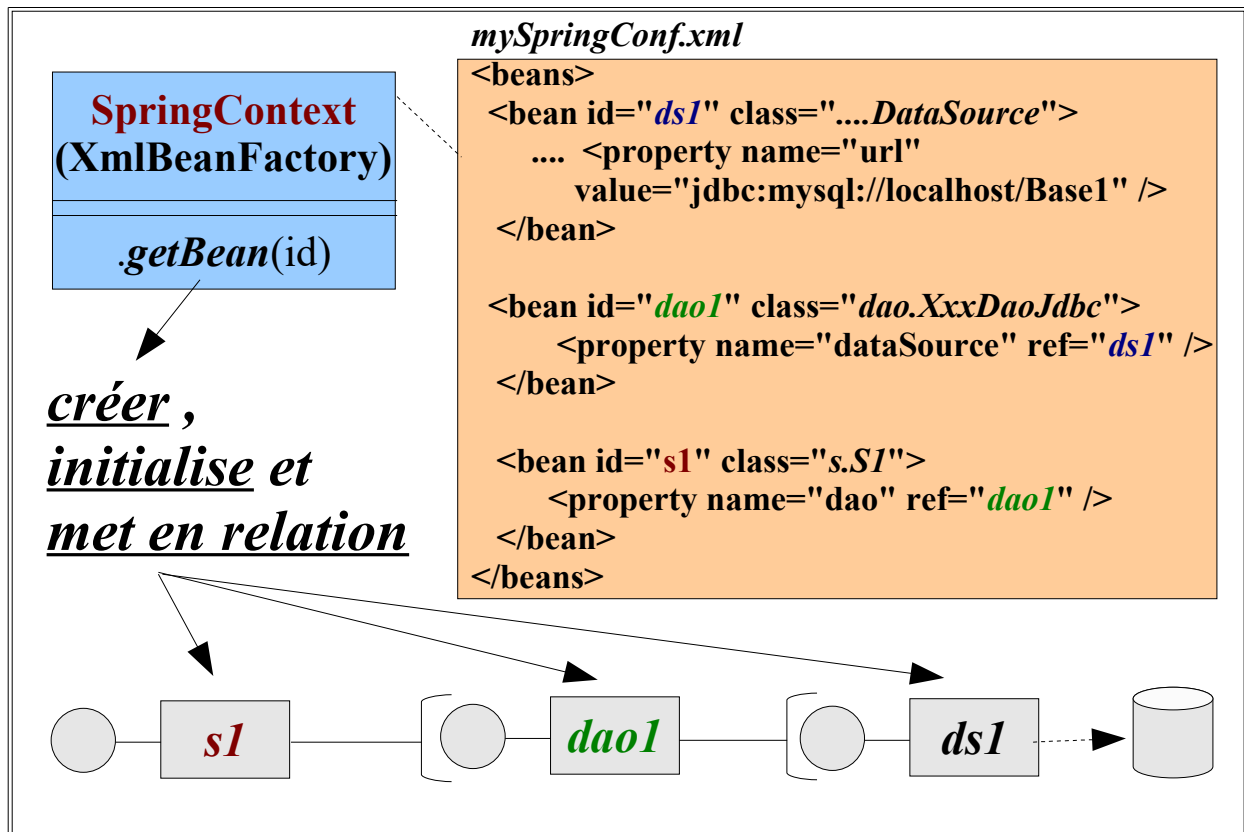
En fonction des réels besoins de l'application, l'entête du fichier de configuration Spring pourra comporter (ou pas) tout un tas d'éléments optionnels (AOP , Transactions , ...).

#### Exemple d'entête:

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

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

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



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

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

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

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

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

## 1.3. Instanciation de composant Spring via une Fabrique

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

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

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

## 1.4. ApplicationContext et test unitaires

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

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

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

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

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

Attention (pour les performances):

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

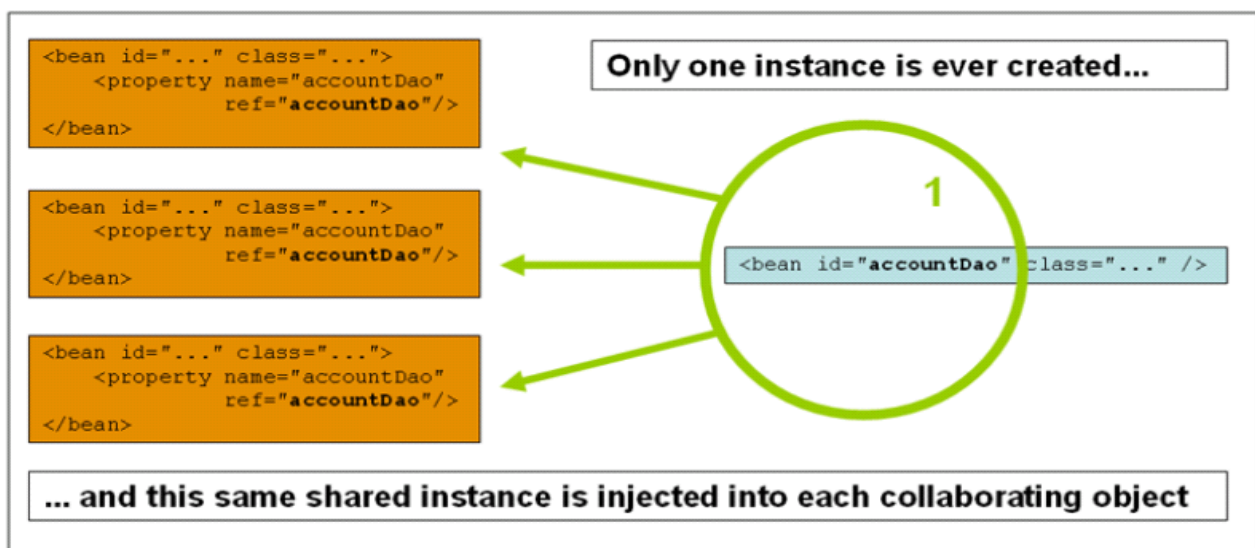
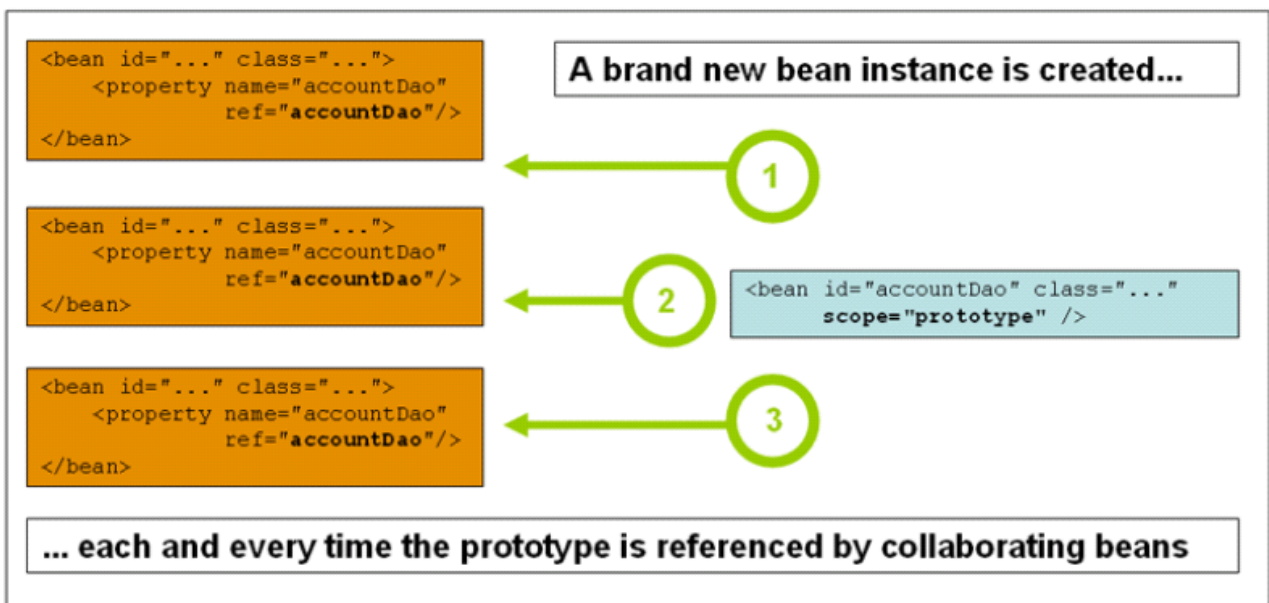
- analyse de toute la configuration Xml
- déclenchement des mécanismes AOP dynamiques
- instanciations des composants
- assemblage par injections de dépendances

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

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

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

<i>portée (scope) d'un composant Spring</i>	<i>comportement / cycle de vie</i>
<b>singleton (par défaut)</b>	un seul composant instancié et partagé au niveau de l'ensemble du conteneur léger Spring. (sémantique "Stateless / sans état" )
<b>prototype</b>	une instance par utilisation (sémantique "Stateful / à état" )
<b>session</b>	une instance rattachée à chaque session Http (valable uniquement au sein d'un "web-aware ApplicationContext")
<b>request</b>	une instance rattachée à une requête Http (valable uniquement au sein d'un "web-aware ApplicationContext")
<b>global session</b>	(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 où la valeur de l'attribut **"resource"** commence par **"classpath:"** le chemin indiqué sera alors recherché en relatif par rapport à l'intégralité de tout le classpath (tous les ".jar")

Exemples :

*applicationContext.xml*

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

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

Rappels :

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

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

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

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

**database.properties**

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

**dataSourceSpringConf.xml**

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

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

</bean>
```

...



## 2. Configuration IOC Spring via des annotations

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

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

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

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

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

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

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

### 2.1. Configuration xml pour "xml + annotations"

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

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

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

</beans>
```

### 2.2. Annotations (stéréotypées) pour composant applicatif

exemple : XYDaoImplAnot.java

```
package tp.persistance.with_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 .

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

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

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

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

### Exemple (assez conseillé) avec @Autowired

```

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

    private XYDao xyDao;
}

```

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

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

ou bien plus simplement :

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

    @Autowired
    private XYDao xyDao;

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

NB :

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

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

**private Payment paiementParCarteDeCredit ;**

**ou bien**

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

**private Payment paiementEnLiquide ;**

permettra d'effectuer une injection de la version voulue .

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

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

En organisant bien les packages java de la façon suivante :

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

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

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

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

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

ou bien

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

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

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

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

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

### 3. Tests "JUnit4 + Spring"

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

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

NB :

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

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

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

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

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

    @Autowired
    private GestionComptes service = null;

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

}
```

NB :

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

Via les annotations

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

et

**@Transactional()**

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

### Exemple de classe de Test de Dao (avec annotations)

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

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

    // injection du doa à tester 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")



# XV - Annexe – Ancienne configuration Spring 4

## 1. ancienne version Spring-boot 1.x (Spring 4)

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 entre 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. 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=/deviseSpringBootTest in application.properties :
    factory.setContextPath("/deviseSpringBootTest");

    //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;
}
}

```

### 1.3. 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>
    */
}
```

# XVI - Annexe – Spring Actuator

## 1. Spring-Actuator

### 1.1. Présentation de "spring-actuator"

L'extension standard "spring-actuator" permet de récupérer automatiquement des indications (métriques) sur le fonctionnement interne d'une application spring-boot .

**Spring-actuator** permet très concrètement d'intégrer dans l'application développée tout un tas de micros-services REST techniques/annexes permettant de *récupérer certaines métriques (variables d'environnement , consommation mémoire , ...)* .

Exemple :

http://localhost:8181\_ou\_autre/my-spring-boot-app/**actuator/health**

retourne **{"status":"UP"}** quand l'application fonctionne bien  
ou bien **{"status":"DOWN"}** quand l'application ne fonctionne pas bien

En production/exploitation, Ceci peut être très utile pour effectuer un suivi/pilotage de l'application et surveiller son bon fonctionnement .

En développement/debug , spring-actuator peut grandement aider à trouver certains réglages permettant d'optimiser les performances de l'application .

### 1.2. Mise en oeuvre de spring-actuator

dans pom.xml :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
```

dans **application.properties**

```
#actuators:
management.endpoints.web.exposure.include=*
#management.endpoints.web.exposure.include=health,info,env,metrics
```

**Important :** \* en dev seulement ,

**En production , trop de mesures = application ralentie !!!**

**Et si besoin dans `WebSecurityConfig` ou ... :**

```
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    ....
    protected void configure(HttpSecurity http) throws Exception {
        http
            .authorizeRequests()
                .antMatchers("/actuator/**").permitAll()
                .anyRequest().authenticated();
    }
    ...}
}
```

### 1.3. URLs des "endpoints" de spring actuator :

`http://localhost:8181_ou_autre/my-spring-boot-app/actuator`

renvoie au format JSON la liste des urls précises des "endpoints" de spring-actuator :

```
{
  "_links": {
    "self": {
      "href": "http://localhost:8181/spring-boot-backend/actuator",
      "templated": false
    },
    "beans": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/beans",
      "templated": false
    },
    "caches-cache": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/caches/{cache}",
      "templated": true
    },
    "caches": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/caches",
      "templated": false
    },
    "health": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/health",
      "templated": false
    },
    "health-path": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/health/{*path}",
      "templated": true
    },
    "info": {
      "href": "http://localhost:8181/spring-boot-backend/actuator/info",
```

```

    "templated": false
  },
  "conditions": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/conditions",
    "templated": false
  },
  "configprops": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/configprops",
    "templated": false
  },
  "env": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/env",
    "templated": false
  },
  "env-toMatch": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/env/{toMatch}",
    "templated": true
  },
  "loggers": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/loggers",
    "templated": false
  },
  "loggers-name": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/loggers/{name}",
    "templated": true
  },
  "heapdump": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/heapdump",
    "templated": false
  },
  "threaddump": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/threaddump",
    "templated": false
  },
  "metrics-requiredMetricName": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/metrics/{requiredMetricName}",
    "templated": true
  },
  "metrics": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/metrics",
    "templated": false
  },
  "scheduledtasks": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/scheduledtasks",
    "templated": false
  },
  "mappings": {
    "href": "http://localhost:8181/spring-boot-backend/actuator/mappings",
    "templated": false
  }
}

```

http://localhost:8181/my-spring-boot-app/**actuator/metrics/http.server.requests**

retourne

```

{
  "name": "http.server.requests",
  "description": null,
  "baseUnit": "seconds",
  "measurements": [

```

```

{
  "statistic": "COUNT",
  "value": 95
},
{
  "statistic": "TOTAL_TIME",
  "value": 4.6482036000000001
},
{
  "statistic": "MAX",
  "value": 0.7233778
}
],
....
}

```

<http://localhost:8181/my-spring-boot-app/actuator/env>

retourne

```

{
  "activeProfiles": [
    "embeddedDb",
    "reInit",
    "appDbSecurity"
  ],
  "propertySources": [
    {
      "name": "server.ports",
      "properties": {
        "local.server.port": {
          "value": 8181
        }
      }
    },
    {
      "name": "servletContextInitParams",
      "properties": {}
    },
    {
      "name": "systemProperties",
      "properties": {
        "sun.desktop": {
          "value": "windows"
        },
        "awt.toolkit": {
          "value": "sun.awt.windows.WToolkit"
        },
        "java.specification.version": {
          "value": "11"
        },
        "sun.cpu.isalist": {
          "value": "amd64"
        },
        "sun.jnu.encoding": {
          "value": "Cp1252"
        },
        "java.class.path": {
          "value": "D:\\tp\\local-git-mycontrib-repositories\\env-ic-my-java-rest-app\\target\\classes;...."
        }
      }
    },
    ....
  ],
  ....
}

```



```

"java.vendor.url": {
  "value": "http://java.oracle.com/"
},
"catalina.useNaming": {
  "value": "false"
},
"user.timezone": {
  "value": "Europe/Paris"
},
"os.name": {
  "value": "Windows 10"
},
"java.vm.specification.version": {
  "value": "11"
},
"sun.java.launcher": {
  "value": "SUN_STANDARD"
},
"user.country": {
  "value": "FR"
}, ...

"JAVA_HOME": {
  "value": "C:\\Program Files\\Java\\jdk-11.0.4",
  "origin": "System Environment Property \\\"JAVA_HOME\\\""
}, ...

```

et `http://localhost:8181/my-spring-boot-app/actuator/env/user.timezone`

retourne

```

{
  "property": {
    "source": "systemProperties",
    "value": "Europe/Paris"
  }, ...
}

```

...

## 1.4. Paramétrages (application.properties) de `actuator/info`

En ajoutant dans `app.properties`

```

#app infos for actuator/info
#info.app.MY_APP_PROP_NAME=ValeurQuiVaBien
info.app.name=spring-boot-backend
info.app.description=appli spring-boot , backend with rest-api (micro services)

```

alors l'actuator prédéfini `actuator/info` renvoi

```

{
  "app": {
    "name": "spring-boot-backend",

```

```
"description": "appli spring-boot , backend with rest-api (micro services)"  
}  
}
```

## 1.5. Codage d'un "health indicator" spécifique

```
@Component  
public class MyHealthIndicator implements HealthIndicator {  
  
    @Override  
    public Health health() {  
        long result = checkSomething();  
        if (result <= 0) {  
            return Health.down().withDetail("Something Result", result).build();  
        }  
        return Health.up().build();  
    }  
}
```

Autre variante possible : ... implements **ReactiveHealthIndicator**

## 1.6. SpringBootAdmin (extension de de.codecentric)

**SpringBootAdmin** est une extension spring de "de.codecentric" qui permet de mettre assez facilement en oeuvre un serveur de surveillance/administration de certaines applications springBoot.

- Dans la terminologie "spring-boot-admin", l'application spring-admin qui va surveiller les autres sera vu comme le coté **serveur**.
- Les applications ordinaires (avec WS REST et actuators) seront considérées comme le coté "**client**".
- **Dans le mode de fonctionnement le plus simple , une instance d'une application ordinaire s'enregistre au démarrage auprès du serveur de surveillance "spring-admin" en précisant si besoin url,username,password .**
- Dans un mode de fonctionnement plus élaboré , l'application spring-admin ("serveur") peut quelquefois découvrir automatiquement certaines applications micro-services à surveiller via certains services techniques additionnels (ex : *Eureka* ou ...)

Spring Boot Admin

Tableau de bordApplicationsJournalÀ proposadminfr

APPLICATIONS

1

INSTANCES

1

STATUT

tout est disponible

OK

✓ spring-boot-application

16mhttp://LAPTOP-DDC:8181/spring-boot-backend

spring-boot-application

Id: e17564762c69

http://LAPTOP-DDC:8181/spring-boot-backend

http://LAPTOP-DDC:8181/spring-boot-backend/actuator

http://LAPTOP-DDC:8181/spring-boot-backend/actuator/health

Info

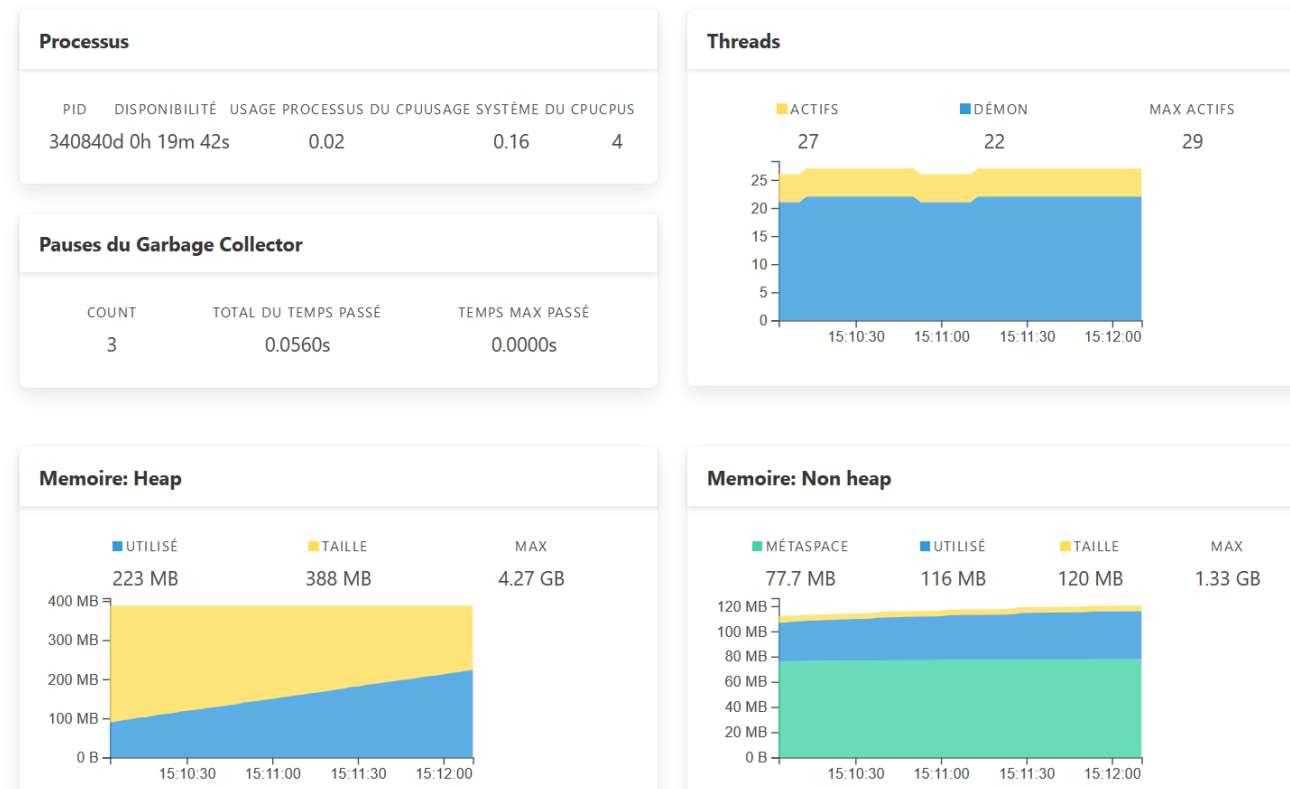
app name: spring-boot-backend  
description: appli spring-boot , backend

État

InstanceUP

Métadonnées

startup2020-02-03T14:52:36.2960337+01:00



## Code minimaliste de l'application "spring-admin"

### pom.xml

```
....
<properties>
  <java.version>1.8</java.version>
  <spring-boot-admin.version>2.2.1</spring-boot-admin.version>
</properties>
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>de.codecentric</groupId>
```

```

        <artifactId>spring-boot-admin-starter-server</artifactId>
    </dependency>
...
</dependencies>

<dependencyManagement>
    <dependencies>
        <dependency>
            <groupId>de.codecentric</groupId>
            <artifactId>spring-boot-admin-dependencies</artifactId>
            <version>${spring-boot-admin.version}</version>
            <type>pom</type>
            <scope>import</scope>
        </dependency>
    </dependencies>
</dependencyManagement>
...
</project>

```

### application.properties

```

server.servlet.context-path=/spring-admin
server.port=8787
logging.level.org=INFO

#this "spring-boot-admin server" app can monitor
#several "spring-boot-admin client" ordinary app with actuators

#spring-boot-admin SERVER" properties:
spring.security.user.name=admin
spring.security.user.password=admin-pwd

```

### SecurityConfig.java

```

@Configuration
public class SecurityConfig extends WebSecurityConfigurerAdapter {

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

```

```

SavedRequestAwareAuthenticationSuccessHandler successHandler
    = new SavedRequestAwareAuthenticationSuccessHandler();
successHandler.setTargetUrlParameter("redirectTo");
successHandler.setDefaultTargetUrl("/");

http.authorizeRequests()
    .antMatchers("/assets/**").permitAll()
    .antMatchers("/login").permitAll()
    .anyRequest().authenticated().and()
    .formLogin().loginPage("/login")
    .successHandler(successHandler).and()
    .logout().logoutUrl("/logout").and()
    .httpBasic().and()
    .csrf()
    .csrfTokenRepository(CookieCsrfTokenRepository.withHttpOnlyFalse())
    .ignoringAntMatchers(
        "/instances",
        "/actuator/**"
    );
}
}

```

### SpringAdminApplication.java

```

...
@SpringBootApplication
@EnableAdminServer //de.codecentric.boot.admin.server.config.EnableAdminServer
public class SpringAdminApplication {
    public static void main(String[] args) {
        SpringApplication.run(SpringAdminApplication.class, args);
        System.out.println("http://localhost:8787/spring-admin");
    }
}

```

.../...

Partie spring-boot-admin-starter-client à ajouter dans application ordinaire :

dans *pom.xml* :

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-actuator</artifactId>
</dependency>
<dependency>
  <groupId>de.codecentric</groupId>
  <artifactId>spring-boot-admin-starter-client</artifactId>
  <version>2.2.1</version>
</dependency>
```

dans *application.properties*

```
management.endpoints.web.exposure.include=*
#management.security.enabled=false or .antMatchers("/actuator/**").permitAll() or ...

#spring.boot.admin.client params to register in spring.boot.admin.server "spring-admin" :
spring.boot.admin.client.url=http://localhost:8787/spring-admin
spring.boot.admin.client.username=admin
spring.boot.admin.client.password=admin-pwd
```

+ tous les paramétrages "spring-actuator" habituels .

# XVII - Annexe – Spring JMS

## 1. Repères JMS

### JMS (Java Message Service)

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

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

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

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

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

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

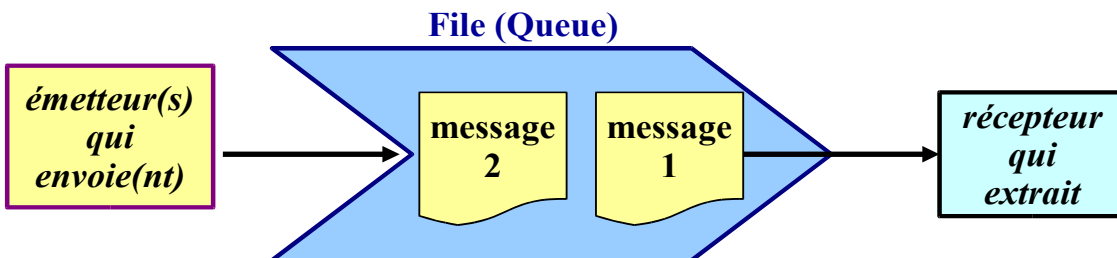
L'objet **ConnectionFactory** sera utilisé pour établir une connexion avec une file.

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

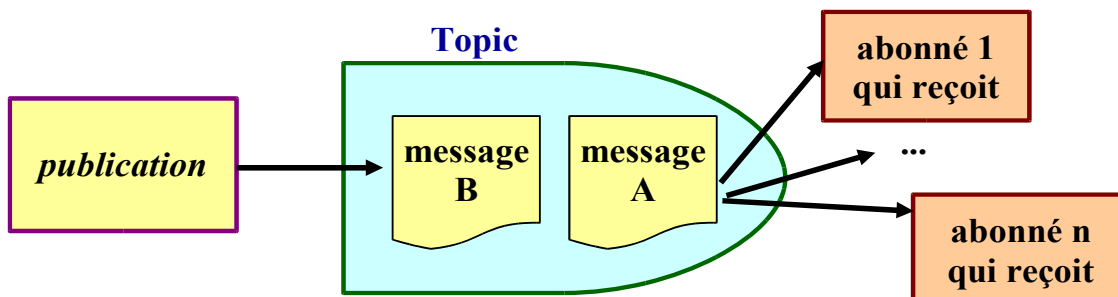
**JMS** permet de mettre en oeuvre les 2 modèles suivants:

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

#### JMS Queue : Point To Point



#### JMS Topic : Publish / Subscribe

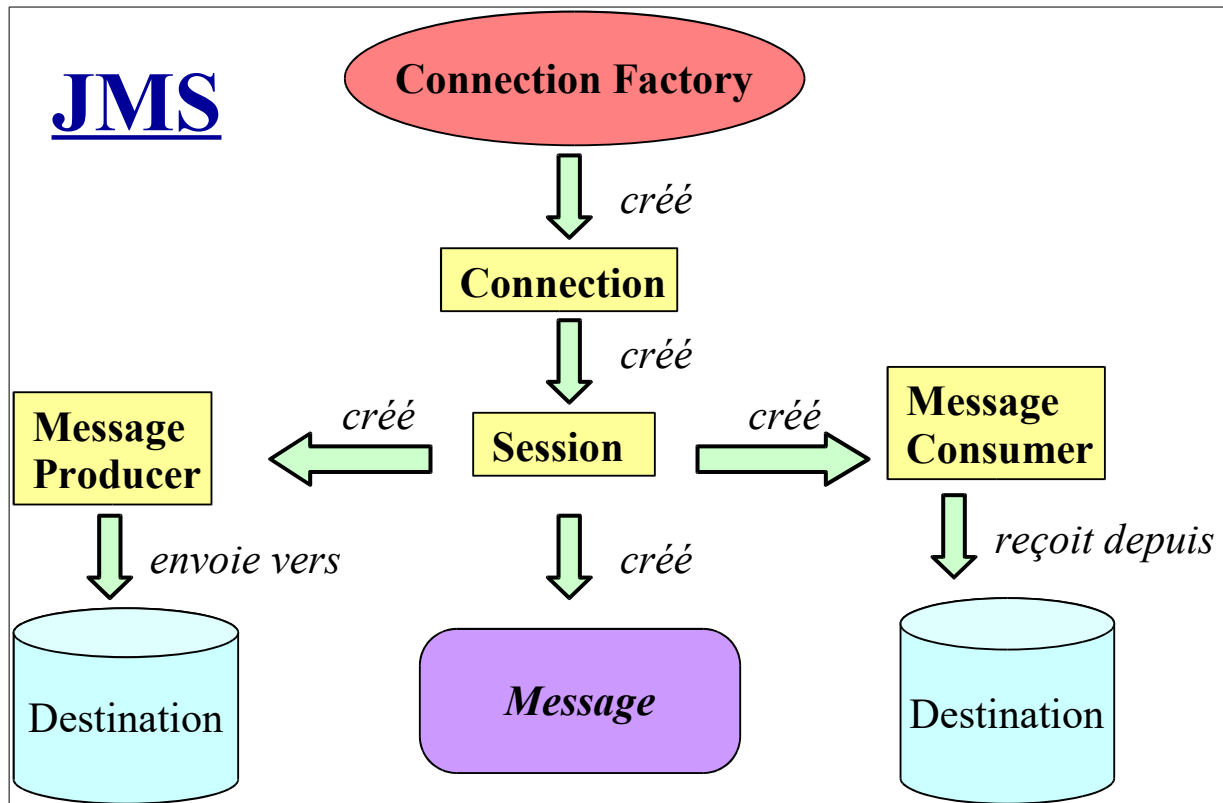




Le tableau ci-dessous résume les différentes **interfaces** utilisées au niveau de l'api JMS:

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

(mt) : multi-threading support.



#### Champs des entêtes de message :

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

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

## 1.1. ActiveMq

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

default username/password : admin/admin

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

*start\_activeMq.bat*

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

activemq start

*stop\_activeMq.bat*

activemq stop

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

### Browse MyDataQueue

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

[View Consumers](#)

## 1.2. Artemis (nouvelle génération de ActiveMq)

dans **artemis/bin**

écrire et lancer createArtemisBroker.bat

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

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

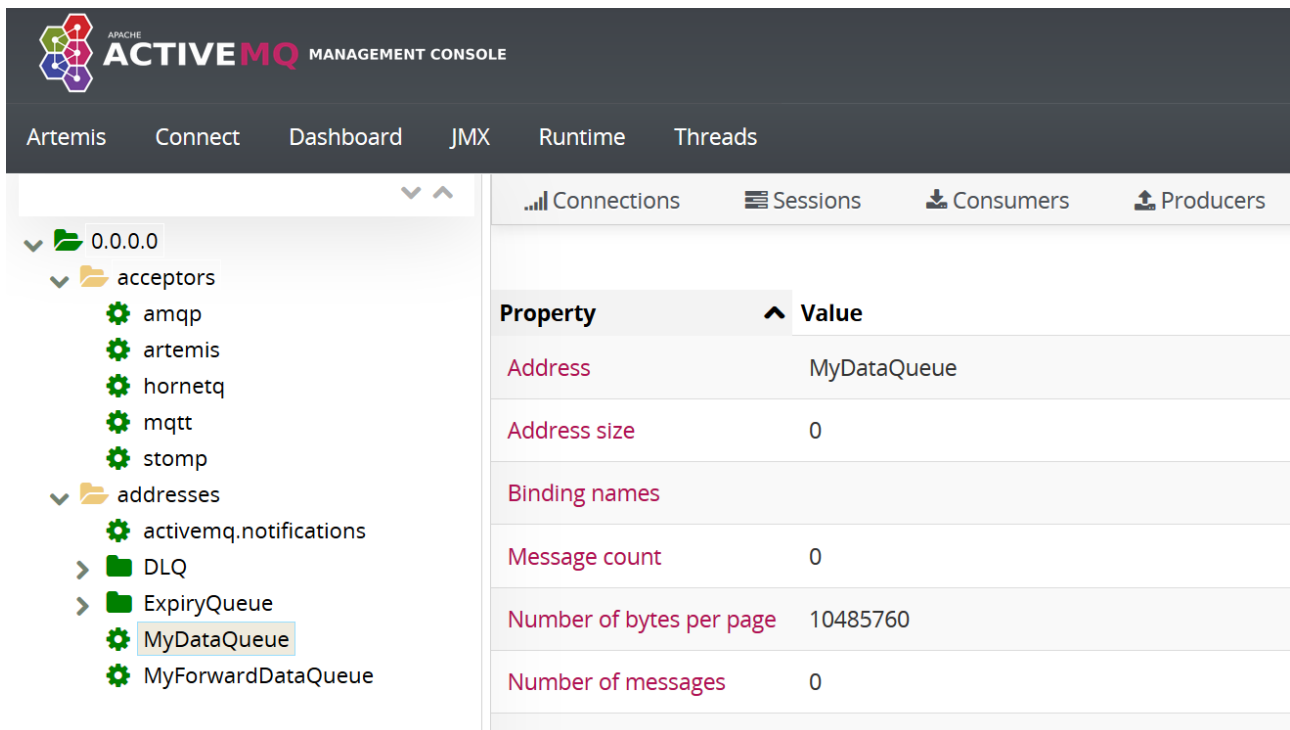
**artemis run**

**artemis stop**

**Console admin (artemis)**

***http://localhost:8161/console***

username/password : **admin/admin**



The screenshot shows the Apache ActiveMQ Management Console interface. The left sidebar displays a tree view of the system components, including '0.0.0.0', 'acceptors', 'addresses', and 'MyDataQueue'. The main panel shows the configuration for 'MyDataQueue' with a table of properties and values.

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

## 2. intégration JMS dans Spring

### 2.1. Configuration "JMS avec Spring-boot"

pom.xml

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

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

application.properties

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

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

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

MySpringBootApplication --> comme d'habitude avec @SpringBootApplication

**JmsConfig.java**

```

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

@Configuration
@EnableJms
public class JmsConfig {

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

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

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

```

**org.mycontrib.xyz.dto.MyData.java**

```

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

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

```

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

```

package org.mycontrib.xyz.jms;

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

@Component
public class MyDataJmsReceiver {

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

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

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

```

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

```

package org.mycontrib.xyz.rest;

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

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

    @Autowired
    private JmsTemplate jmsTemplate;

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

```

```

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

```

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

```

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

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

```

```

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

```

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

ref:  (as string)  
 value:  (as number)  
  
 savedData={"ref":"r1","value":123456.0}

### Dans console java :

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

## 2.2. Application java externe qui envoie des messages

### MyOtherJmsAppSendingMessage.java

```

package org.mycontrib.xyz;

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

```



```

import com.fasterxml.jackson.databind.ObjectMapper;

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

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

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

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

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

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

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

```

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

```

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

```

## **XVIII - Annexe – Spring et Web Sockets**

...

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

### 1. Transactions distribuées et commit à 2 phases

#### 1.1. Qualités (A.C.I.D.) d'une transaction distribuée basique

### Transactions distribuées

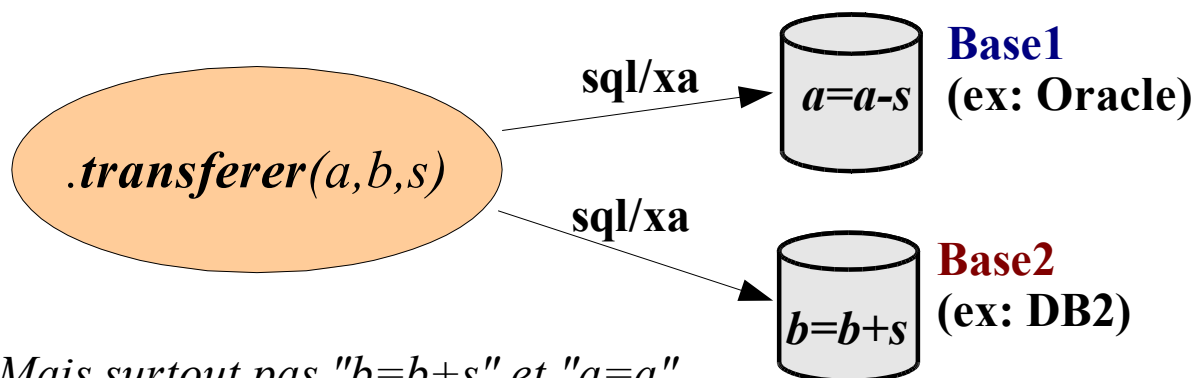
- **A.C.I.D.** ==> **A**tomicity , **C**onsistency , **I**solation , **D**urability
  - L' **Atomicité** désigne le comportement "**tout ou rien**" (Le tout vu en tant qu'élément unique et atomique doit soit réussir , soit échouer). Il n'y a pas de demi-mesure.
  - La **Consistance** d'une transaction désigne le fait que **les différentes opérations doivent laisser le système dans un état stable et cohérent.**
  - Le concept d' **Isolation** signifie ici que **2 transactions concurrentes n'interfèrent pas entre elles** (Points critiques: résultats intermédiaires et opérations annulées).
  - La **Durabilité** indique que **les résultats d'une transaction doivent absolument être mémorisés de façon durable** (sur un support physique) de façon à survivre suite à une éventuelle défaillance(un fichier de Log peut également être très utile).

## 1.2. Protocole XA pour le commit à 2 phases

### *Protocole XA et commit à 2 phases (1)*

Une **transaction distribuée** peut faire intervenir de **multiples ressources** telles que celles-ci par exemple:

- Base 1 (ex: Oracle via JDBC) , Base 2 (ex: DB2 via JDBC)
- Moniteur transactionnel (ex: Tuxedo ou CICS via connecteurs).
- Système de message asynchrone (ex: MQSeries via JMS), ...



### *Protocole XA et commit à 2 phases (2)*

De façon à ce que toutes les opérations à tous les niveaux (chacune des bases de données, ...) soient globalement annulées ou validées, on a recours à la technique suivante:

- 1 - Chaque ressource mise en jeu dans la transaction effectue des opérations dans une zone mémoire à part (ex: opérations SQL que l'on pourra éventuellement annuler) puis envoie un signal pour indiquer qu'à son niveau tout va bien.
- 2 - Un élément "pilote de la transaction" centralise ces acquittements.
- 3 - Si chaque protagoniste de la transaction distribuée a réussi sa tâche, le pilote envoie à chacun d'eux l'ordre d'entériner la mise à jour (commit final). Si un seul protagoniste de la transaction distribuée a échoué dans sa tâche, le pilote envoie à tout le monde l'ordre d'annuler la mise à jour (rollback final).
- Cette technique standard du **commit à deux phases** est formalisée au niveau d'un **protocole** normalisé dénommé **XA**.

## 2. JTA / Atomikos

### 2.1. Cadre général des transactions distribuées

Dans le monde "Java EE" :

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

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

### 2.2. JTA

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

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

## 3. JTA/Atomikos intégré dans Spring et Spring-Boot

### 3.1. Configuration explicite "java-config-jta"

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

```
package org.mycontrib.ext;

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

/**
 * pour properties.put("hibernate.transaction.jta.platform",
```

```

        MyAtomikosJtaPlatform.class.getName());
dans paramétrage des "entityManagerFactory.jpa"
*/

public class MyAtomikosJtaPlatform extends AbstractJtaPlatform {

    private static final long serialVersionUID = 1L;

    static protected TransactionManager transactionManager;
    static protected UserTransaction transaction;

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

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

```

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

```

package org.mycontrib.ext;

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

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

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

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

    @Bean(name = "atomikosTransactionManager", initMethod = "init", destroyMethod = "close")
    public TransactionManager atomikosTransactionManager() throws Throwable {
        UserTransactionManager userTransactionManager = new UserTransactionManager();
    }
}

```

```

        userTransactionManager.setForceShutdown(false);
        MyAtomikosJtaPlatform.transactionManager = userTransactionManager;
        return userTransactionManager;
    }

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

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

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

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

    public static JpaVendorAdapter jpaVendorAdapterFromXaDataSourceClassName(
        String xaDataSourceClassName) {

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

```

```

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

```

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

```

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

```

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

```

package org.mycontrib.ext.customers;

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

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

    @Bean
    @ConfigurationProperties(prefix = "spring.jta.atomikos.datasource.customers")
    public DataSource customersDataSource() {
        logger.trace("init customersDataSource in CustomersConfig");
        return new AtomikosDataSourceBean();
    }
}

```



```

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

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

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

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

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

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

```

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

### MySpringBootApplication.java (avec profiles)

```

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

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

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

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

```

Exemple de configuration globale *application.properties*

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

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

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

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

## 3.2. Configuration en mode test/H2

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

**application-reInit.properties**

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

**application-embeddedDb.properties**

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

spring.jta.atomikos.datasource.orders.unique-resource-name=ordersDataSource
...
```

```
spring.jta.atomikos.datasource.orders.xa-properties.URL=
    jdbc:h2:~/orders;DB_CLOSE_ON_EXIT=FALSE

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

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

### 3.3. Configuration en mode prod/Mysql & postgres

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

**application-noReInit.properties**

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

**application-remoteDb.properties**

```
# JDBC settings for (h2) embedded dataBases
# ici pour 3 bases de données "customers" avec mysql, "orders" et "purchases" avec postgres:
spring.jta.atomikos.datasource.customers.unique-resource-name=customersDataSource
spring.jta.atomikos.datasource.customers.max-pool-size=5
spring.jta.atomikos.datasource.customers.min-pool-size=1
spring.jta.atomikos.datasource.customers.max-life-time=25000
spring.jta.atomikos.datasource.customers.borrow-connection-timeout=10000
spring.jta.atomikos.datasource.customers.xa-properties.pinGlobalTxToPhysicalConnection=true
spring.jta.atomikos.datasource.customers.xa-data-source-class-name=
    com.mysql.cj.jdbc.MySQLXADataSource
spring.jta.atomikos.datasource.customers.xa-properties.user=root
spring.jta.atomikos.datasource.customers.xa-properties.password=root
spring.jta.atomikos.datasource.customers.xa-properties.URL=
jdbc:mysql://127.0.0.1:3306/customers?createDatabaseIfNotExist=true&serverTimezone=UTC

spring.jta.atomikos.datasource.orders.unique-resource-name=ordersDataSource
```

```

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

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

```

NB :

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

### 3.4. Exemple de service transactionnel avec JTA

Organisation possibles des packages java :

```

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

```

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

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

```

...
public interface CustomerRepository extends JpaRepository<Customer,Long>{
    Customer findByEmail(String email);
}

```

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

```
package org.mycontrib.ext.global.service;

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

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

    @Autowired
    private PurchaseRepository purchaseRepository;

    @Autowired
    private OrderRepository orderRepository;

    @Autowired
    private ProductRefRepository productRefRepository;

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

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

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

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

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

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

**TestOrderAndPurchaseService.java**

```

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

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

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

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

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

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

        long nbOrdersBeforePurchaseOrder = orderRepository.count();
        long nbPurchasesBeforePurchaseOrder = purchaseRepository.count();

        try {
            //a tester avec customerId=999L (non existant) with reInit profile
            Long newOrderId = orderAndPurchaseService.purchaseOrder(999L, listOfProductRef);

```

```

        Assertions.fail("une exception aurait du remonter");
    } catch (RuntimeException e) {
        logger.info("exception attendue:" + e);
    }
    //tester le bon rollback :
    long nbOrdersAfterPurchaseOrder = orderRepository.count();
    Assertions.assertTrue(nbOrdersAfterPurchaseOrder==nbOrdersBeforePurchaseOrder);

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

```

**ReInitCustomersOrdersPurchasesDefaultDataSet.java** (avec profile "**reInit**") :

```

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

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

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

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

```

## **XX - Annexe – Tests avancés**

...



# XXI - 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"
- ....

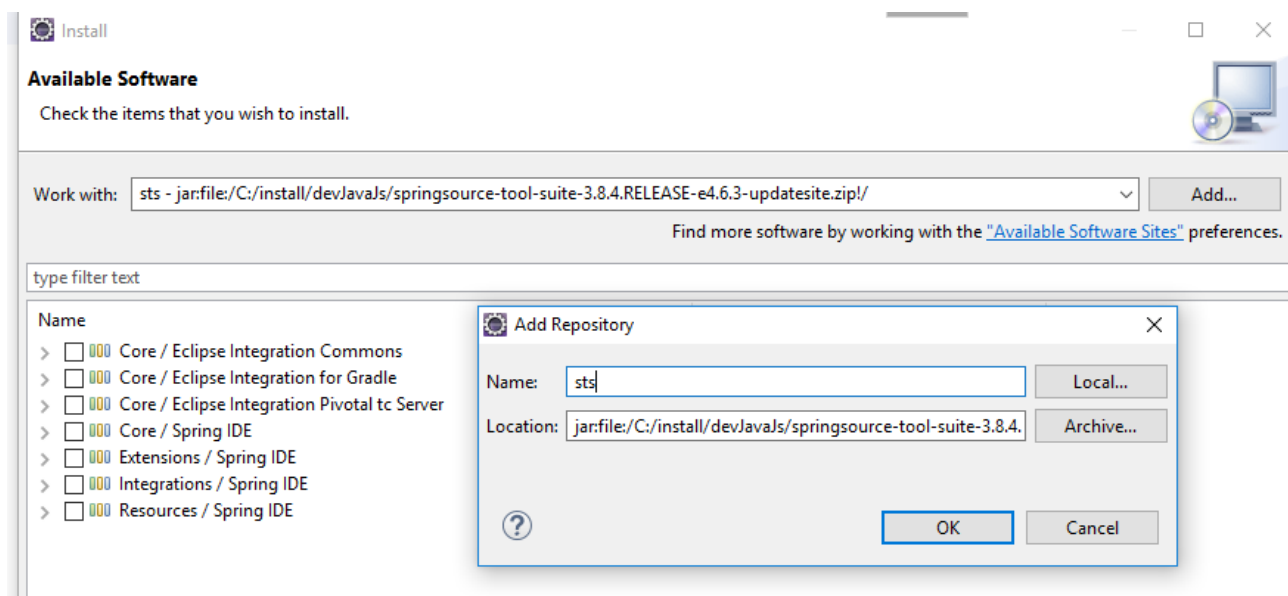
### 1.2. Installation du plugin eclipse STS

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 ..." :

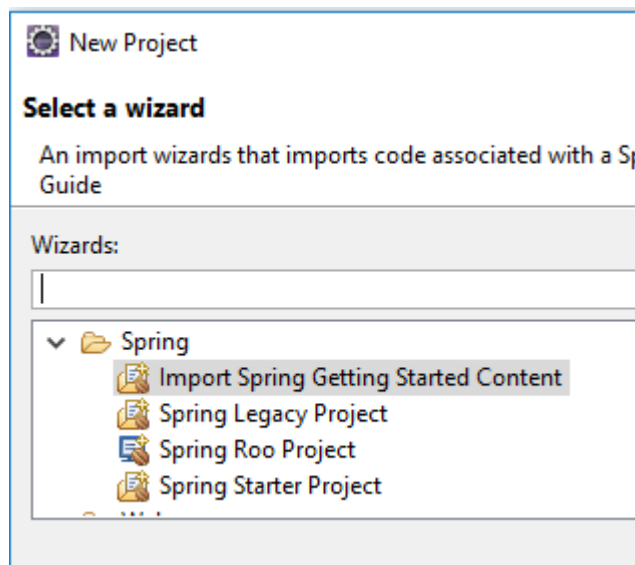


Cocher tout ou bien les parties intéressantes seulement (avec éventuel recul).

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 .

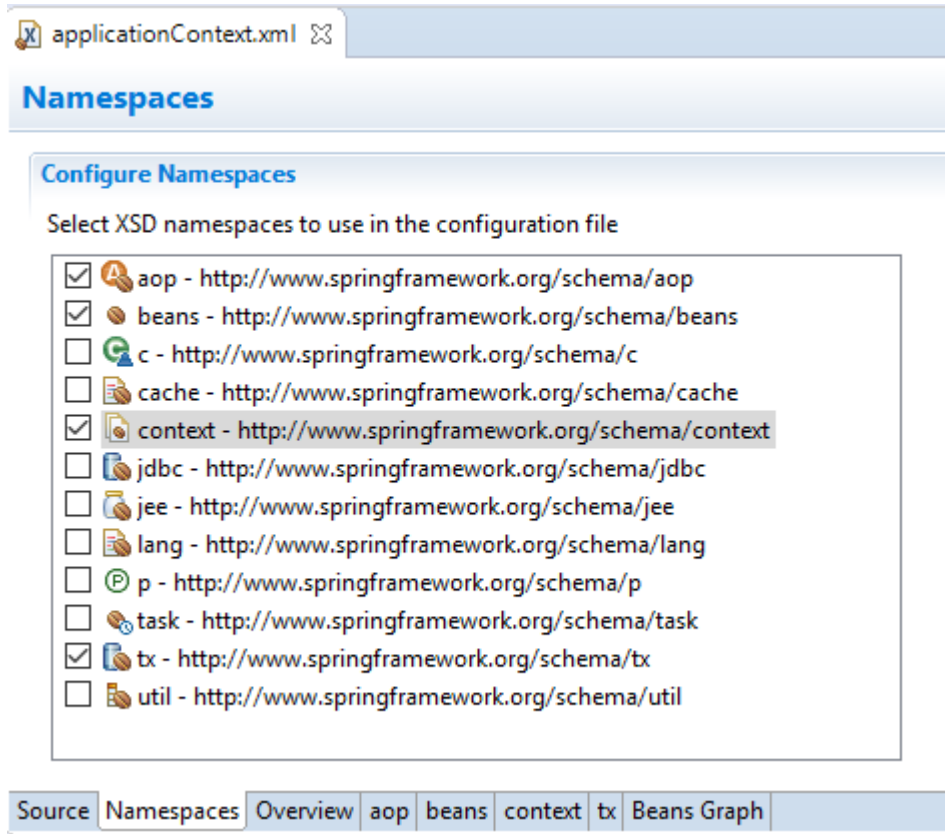
Types de projet	Fonctionnalités	Caractéristiques
Import Spring Getting Started Content	Projets exemples concordants avec la partie "Getting Started" de la documentation "spring"	Juste pour apprendre "spring" via un tutorial et s'entraîner sur tel ou tel aspect de spring
<b>Spring Legacy Project</b>	Projet spring <u>sans</u> "spring boot" 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"
<b>Spring Starter Project</b>	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 inspiration 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/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 pourra 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 structure de **"spring web mvc"**

...

...

## XXII - Annexe – Bibliographie, Liens WEB + TP

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


### 2. TP