

COURS SPRING

Tutoriel Spring 3 : installation SpringSource ToolSuite (STS)



Maîtrisez l'installation de SpringSource ToolSuite, environnement de développement recommandé pour tout développement Spring. Améliorez votre productivité à travers des assistants et des vues spécifiques Spring, ainsi que la possibilité d'être au courant des informations de la communauté Spring.

Prérequis

Les outils et logiciels utilisés :

- ▶ [JDK 6](#)

Objectifs

- ▶ Maîtriser l'installation de SpringSource ToolSuite
- ▶ Identifier le lien entre STS et Spring IDE
- ▶ Comprendre la valeur ajoutée de STS

Programme

- ▶ Partie 1 : téléchargement
- ▶ Partie 2 : Installation
- ▶ Partie 3 : Analyse

Durée

20 min.

Partie 1 : téléchargement

La page officielle du projet STS est : <http://www.springsource.com/developer/sts>

The screenshot shows the official website for SpringSource Tool Suite. The URL in the address bar is www.springsource.com/developer/sts. The main content area features a large orange header: "SpringSource Tool Suite — The Best Development Tool for Enterprise Java". Below this, there's a brief description of what STS offers: "SpringSource Tool Suite™ (STS) provides the best Eclipse-powered development environment for building Spring-powered enterprise applications. STS supplies tools for all of the latest enterprise Java, Spring, Groovy and Grails based technologies as well as the most advanced tooling available for enterprise OSGi development." To the right of the text is the "springsource TOOLSUITE" logo, which consists of two crossed hammers forming a shield-like shape with a green and yellow circular emblem in the center. On the far right, there's a testimonial from Charlie Babcock of InformationWeek: "The SpringSource Tool Suite lets Java developers quickly focus on a problem segment of code and use the Tool Suite's 'consultant in a box' to help fix the problem". At the bottom right of the main content area is a "Download STS" button.

installation-springsource-toolsuite-1

A partir de cette page, vous pouvez arriver sur la [page de téléchargement](#) (ici bouton 'download') de STS.

▶

The screenshot shows the "SPRINGSOURCE TOOL SUITE DOWNLOAD" page. The URL in the address bar is www.springsource.org/springsource-tool-suite-download. The page contains a form with fields for First Name, Last Name, Company, Role (a dropdown menu with "select..."), Email, Phone, and Zip/Postal code. Below the form is a checkbox with the text: "I agree and accept the [license terms and export restrictions](#) associated with this product. (I'd rather not fill in the form. Just take me to the [download page](#))". At the bottom left is a "Submit" button.

installation-springsource-toolsuite-2

Un formulaire vous est présenté. Si vous ne souhaitez pas fournir vos coordonnées, vous pouvez simplement cocher la case à cocher , puis le **lien 'download'**

SpringSource Tool Suite Download

CURRENT VERSION - 2.8.1.RELEASE

Description	Link	Size
Eclipse 3.7.1		
Windows	springsource-tool-suite-2.8.1.RELEASE-e3.7.1-win32-installer.exe	358MB
Windows	springsource-tool-suite-2.8.1.RELEASE-e3.7.1-win32.zip	357MB
Windows (64bit)	springsource-tool-suite-2.8.1.RELEASE-e3.7.1-win32-x86_64-installer.exe	358MB
Windows (64bit)	springsource-tool-suite-2.8.1.RELEASE-e3.7.1-win32-x86_64.zip	357MB
Eclipse 3.7.1.x		
Update Site	springsource-tool-suite-2.8.1.RELEASE-e3.7.1-updatesite.zip	183MB
Eclipse 3.6.x		
Update Site	springsource-tool-suite-2.8.1.RELEASE-e3.6-updatesite.zip	205MB

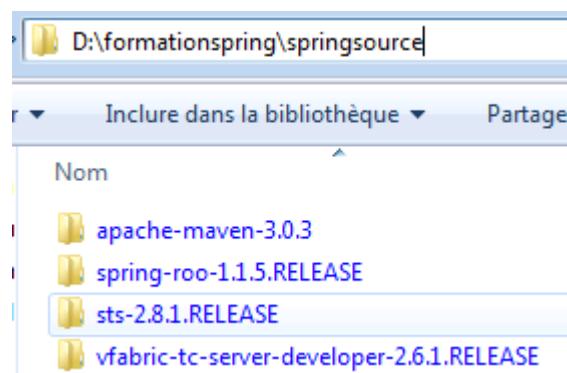
installation-sprinsource-toolsuite-2bis



installation-sprinsource-toolsuite-2ter

Partie 2 : installation & lancement

- Dézippez le fichier téléchargé . Analysez le contenu du répertoire 'springsource' créé :

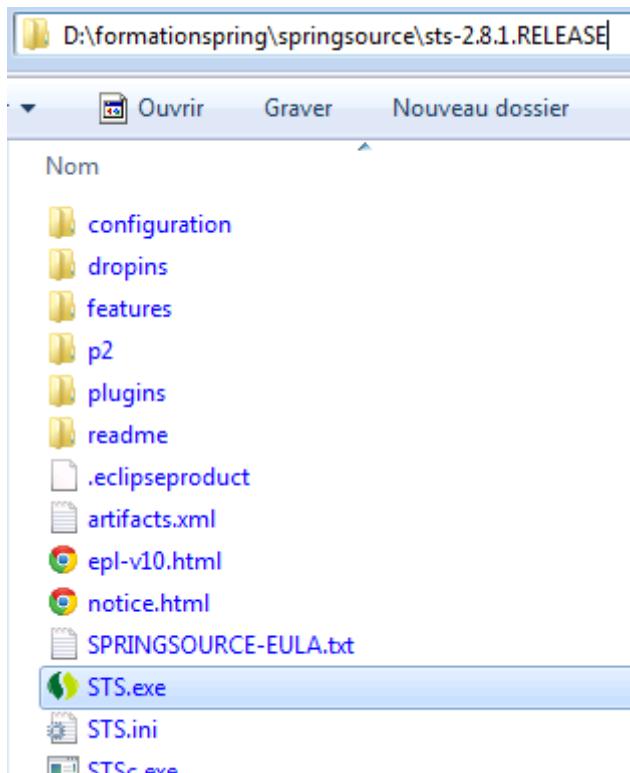


installation-sprinsource-toolsuite-3

Notez la présence des répertoires suivants :

- **Apache Maven** : outil de build MAven, version 3
- **Spring Roo** : outil permettant de créer des squelettes d'application Spring
- **STS** : le répertoire dans lequel se trouve une version packagée d'eclipse, optimisée pour le développement Spring
- **Tc-server** : une variante du serveur Tomcat

- ▶ Entrez dans le répertoire de STS



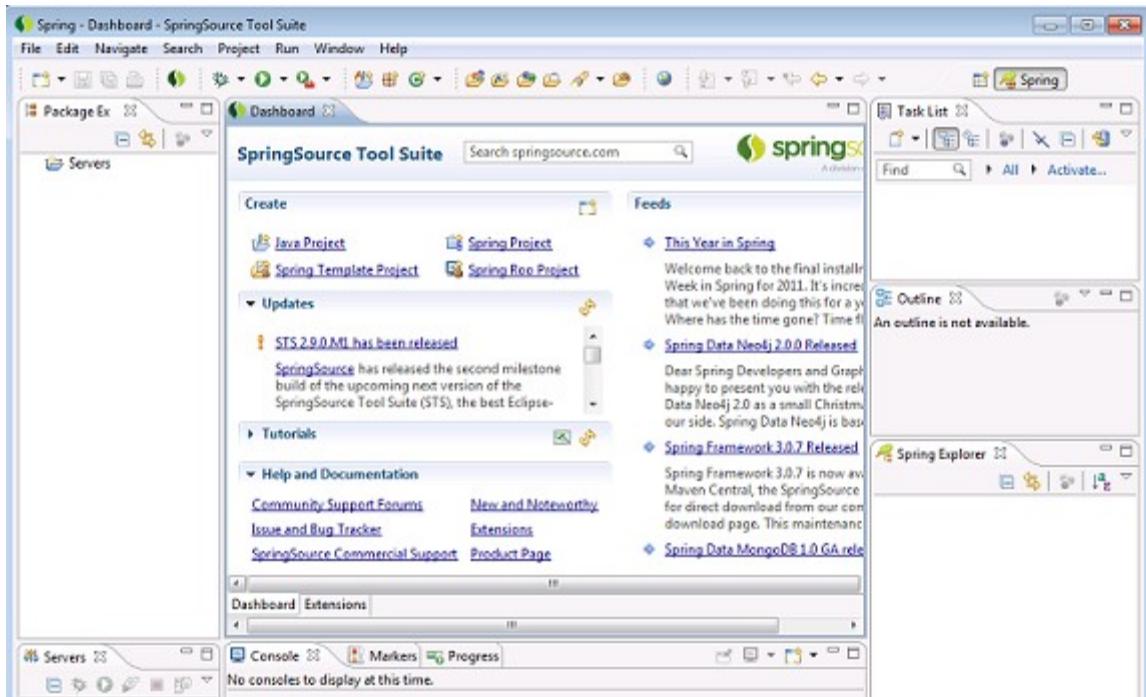
installation-springsource-toolsuite-4

- ▶ Double-cliquer sur STS.exe



installation-springsource-toolsuite-5

Après quelques secondes, l'environnement Eclipse personnalisé apparaît, avec en particulier le 'dashboard' (tableau de bord) Spring.



installation-springsource-toolsuite-6

Partie 3 : analyse

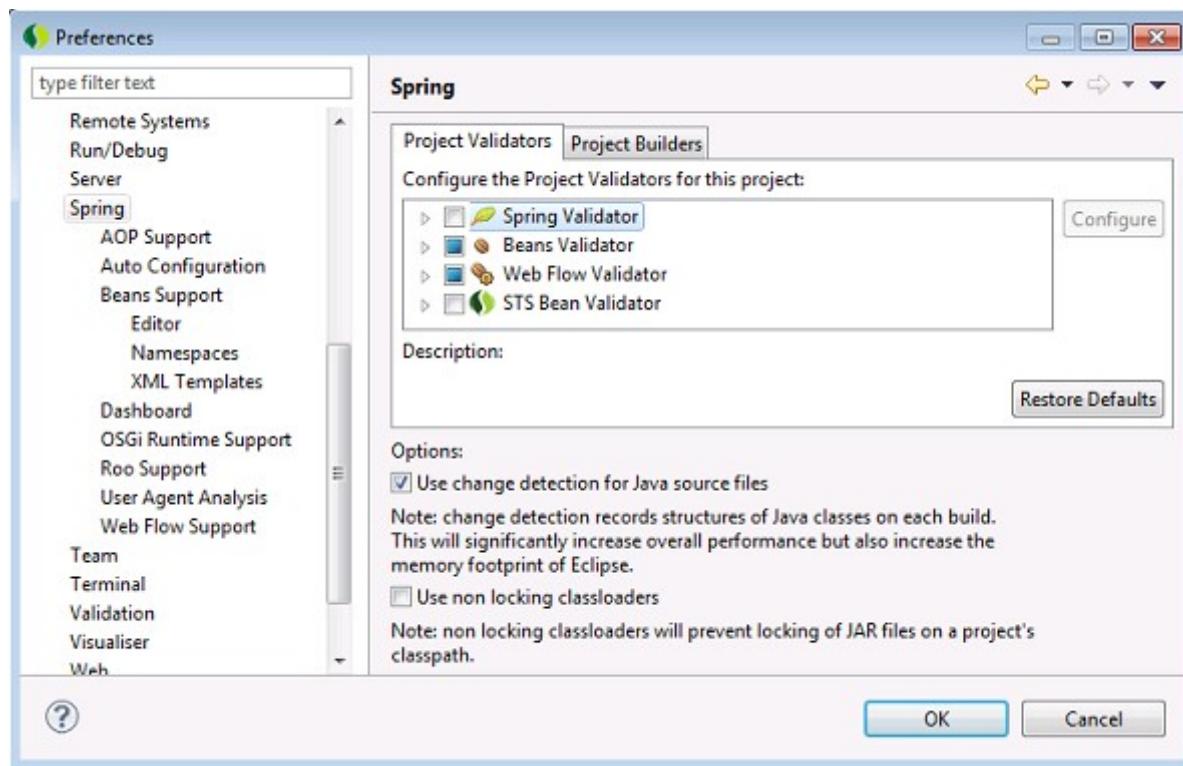
Dashboard

A partir du tableau de bord STS, vous pouvez :

- ▶ lancer des **assistants** de création d'application Spring
- ▶ être **informé** des dernières versions de produits et modules Spring
- ▶ vous **auto-former** à travers des tutoriaux Spring
- ▶ suivre **l'actualité** de la communauté Spring (blogs, sites...)

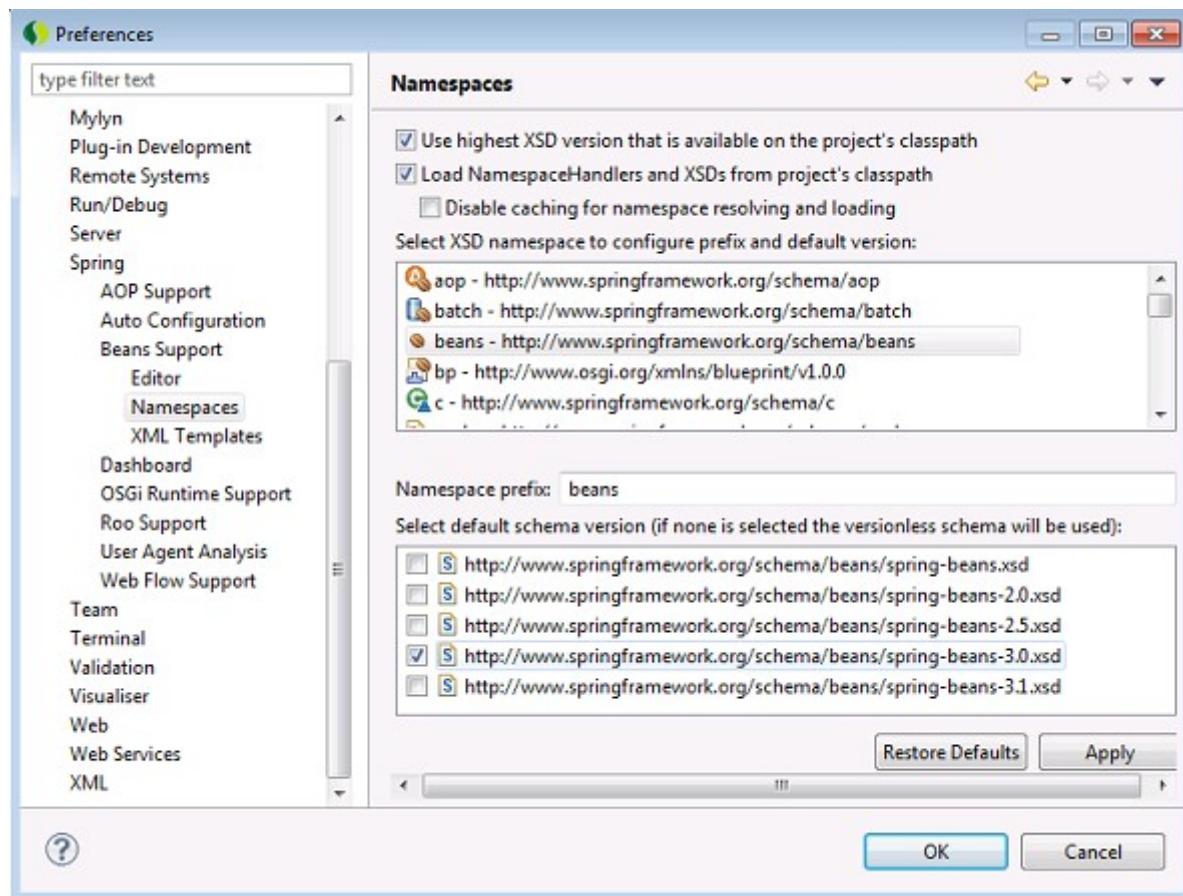
Préférences

- ▶ Menu windows/preferences/spring



installation-springsource-toolsuite-9

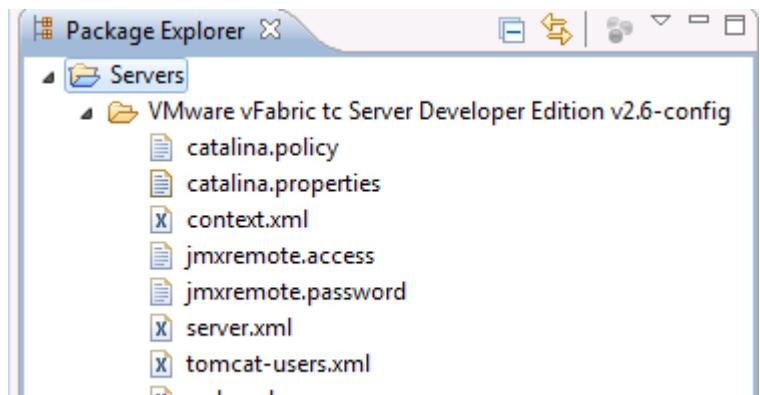
Vous pouvez ici , par exemple, spécifier la version du XSD à utiliser par défaut pour vos fichiers Spring. Ici c'est Spring 3.0



installation-springsource-toolsuite-10

Server

En cas d'utilisation de Spring dans un projet web, il vous sera possible d'utiliser le serveur Tc Serveur déjà intégré.



installation-springsource-toolsuite-7

Bien sur vous pourrez toujours ajouter une configuration pour un autre serveur comme le serveur Tomcat.

Conclusion

Vous venez d'installer SpringSource ToolSuite, une suite d'outils contribuant à améliorer votre productivité de développeur Spring.

Tutoriel Spring N°1 : Installation Spring IDE



Maîtrisez l'installation du plugin eclipse Spring IDE, outil indispensable pour un développement efficace avec le framework Spring. Installez les extensions Web Flow, Aspect AJDT, et Security.

INFO : depuis mars 2009, Spring IDE fait désormais partie de la suite **Spring Source Tool Suite (STS)**, que nous vous recommandons de privilégier désormais pour vos développements SPRING.

INSTALLATION STS

Partie 1 : Stratégies d'installation et fonctionnalités

Remarque : il ya 2 façons d'installer Spring IDE :

- ▶ **A partir d'Eclipse Update Site** : Menu Help/Software Updates/Find and Install/New remote Site /Bouton New remote Site . (Un peu Long)
- ▶ **A partir de l'archive fournie par SpringIDE** : Exemple pour la version 2.1.0 : spring-ide_updatesite_2.1.0_v200808011800.zip

Installation via archive Zip

C'est l'installation la plus rapide. [**L'archive est fournie sur le site de Spring IDE.**](#)

Contenu de Spring IDE core : L'essentiel du plugin

Le plugin Spring IDE permet de :

- ▶ créer un projet Spring
- ▶ créer un fichier de définition de beans.
- ▶ voir les beans dans une vue 'explorateur' Spring,
- ▶ bénéficier de la complétion de code, coloration syntaxique Spring
- ▶ Voir graphe de dépendances entre les beans

Extensions Spring IDE

Des extensions du plugin Spring IDE. Par exemple

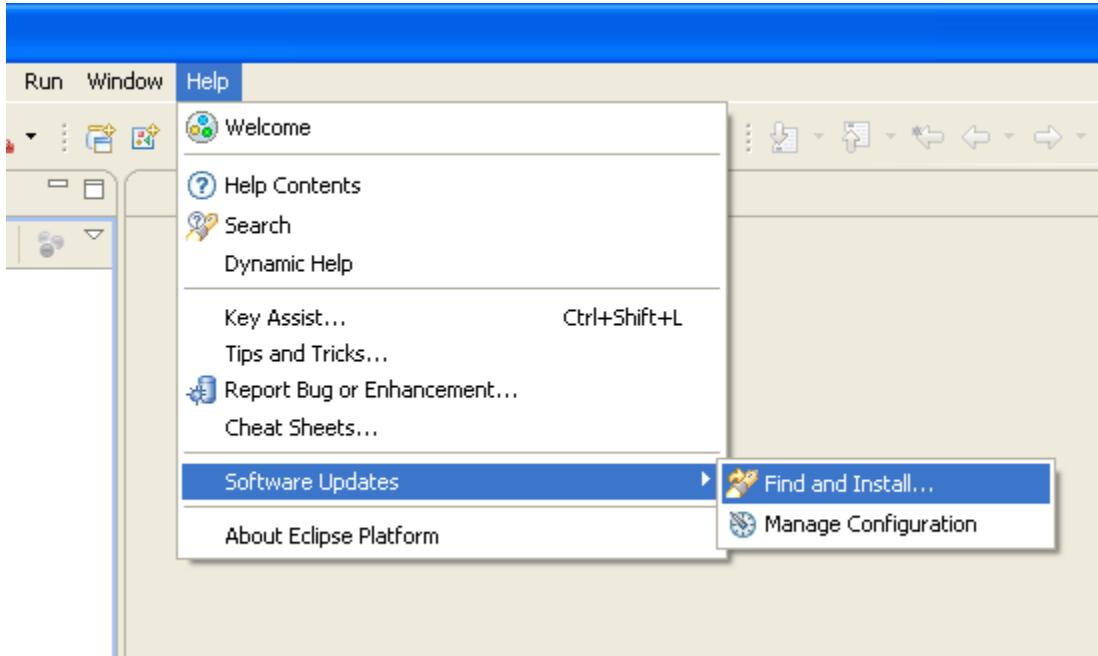
- ▶ Spring IDE AOP Extension (gère les vues Cross References et AOP Event Trace)
- ▶ Spring Web Flow Extension (Mise en oeuvre Spring Web Flow)
- ▶ Plugin AJDT (AspectJ Development Tools) : Développement d'Aspects AspectsJ

Partie 2 : Installation pas à pas

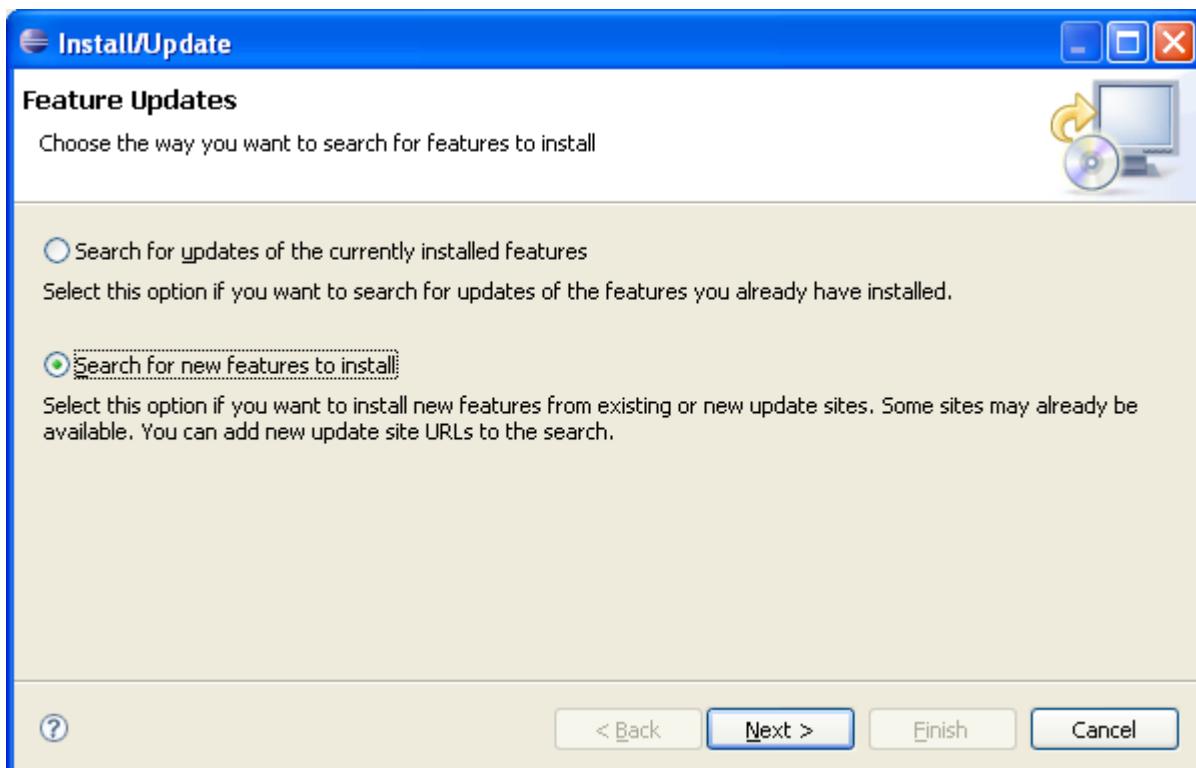
- ▶ Lancez Eclipse . Vérifiez que la version d'Eclipse est 3.3 (Menu Help/About Eclipse Platform)



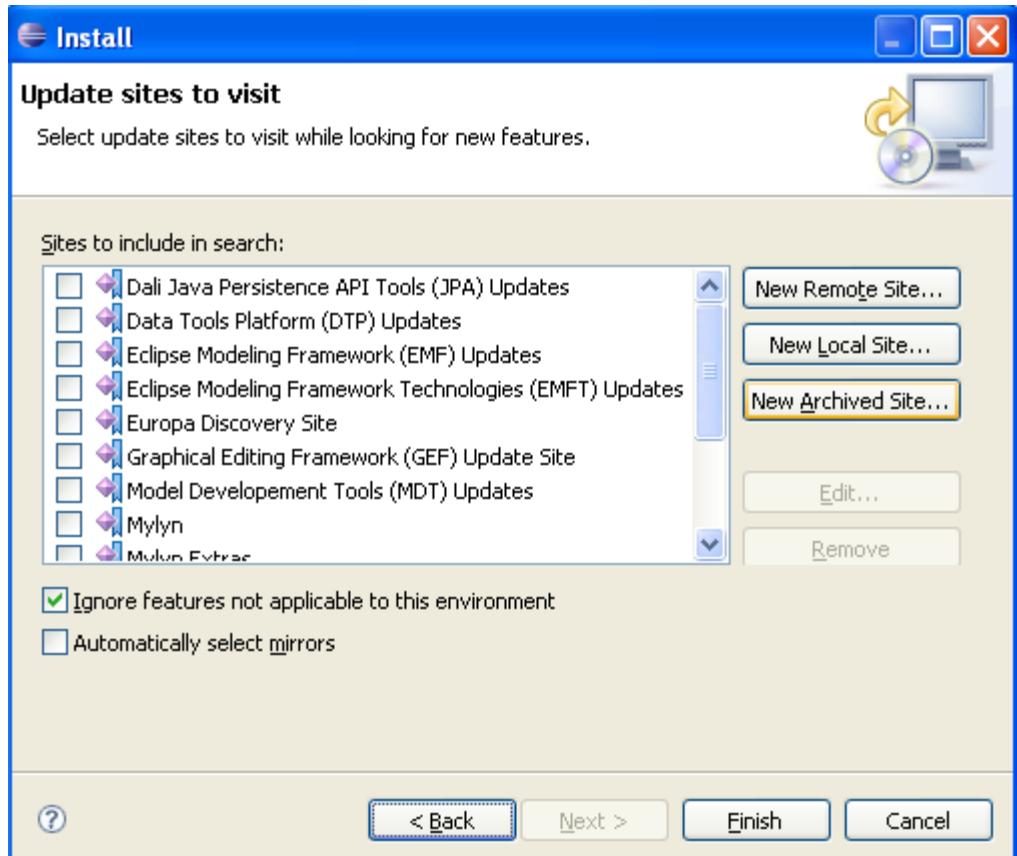
- ▶ Menu Help/Software Updates/Find and Install



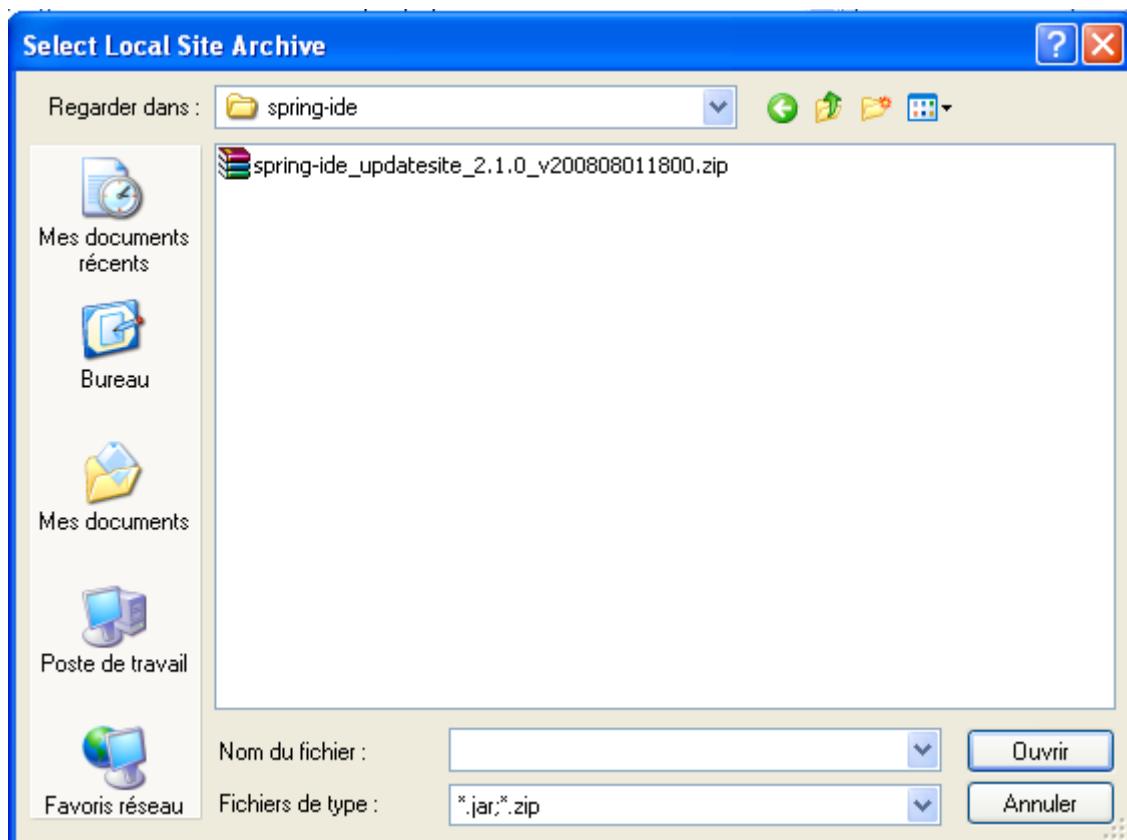
- ▶ Cliquez sur 'Search for new features to install'

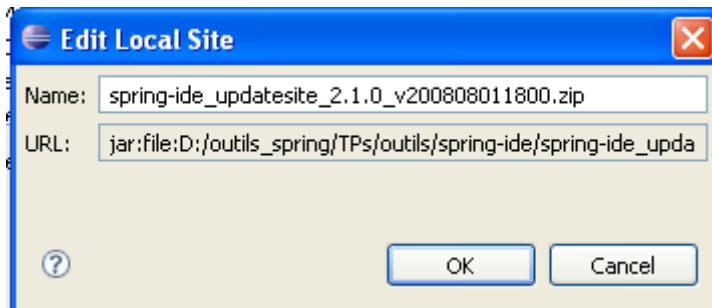


- ▶ Choisir bouton 'New Archived Site' pour retrouver l'archive fournie



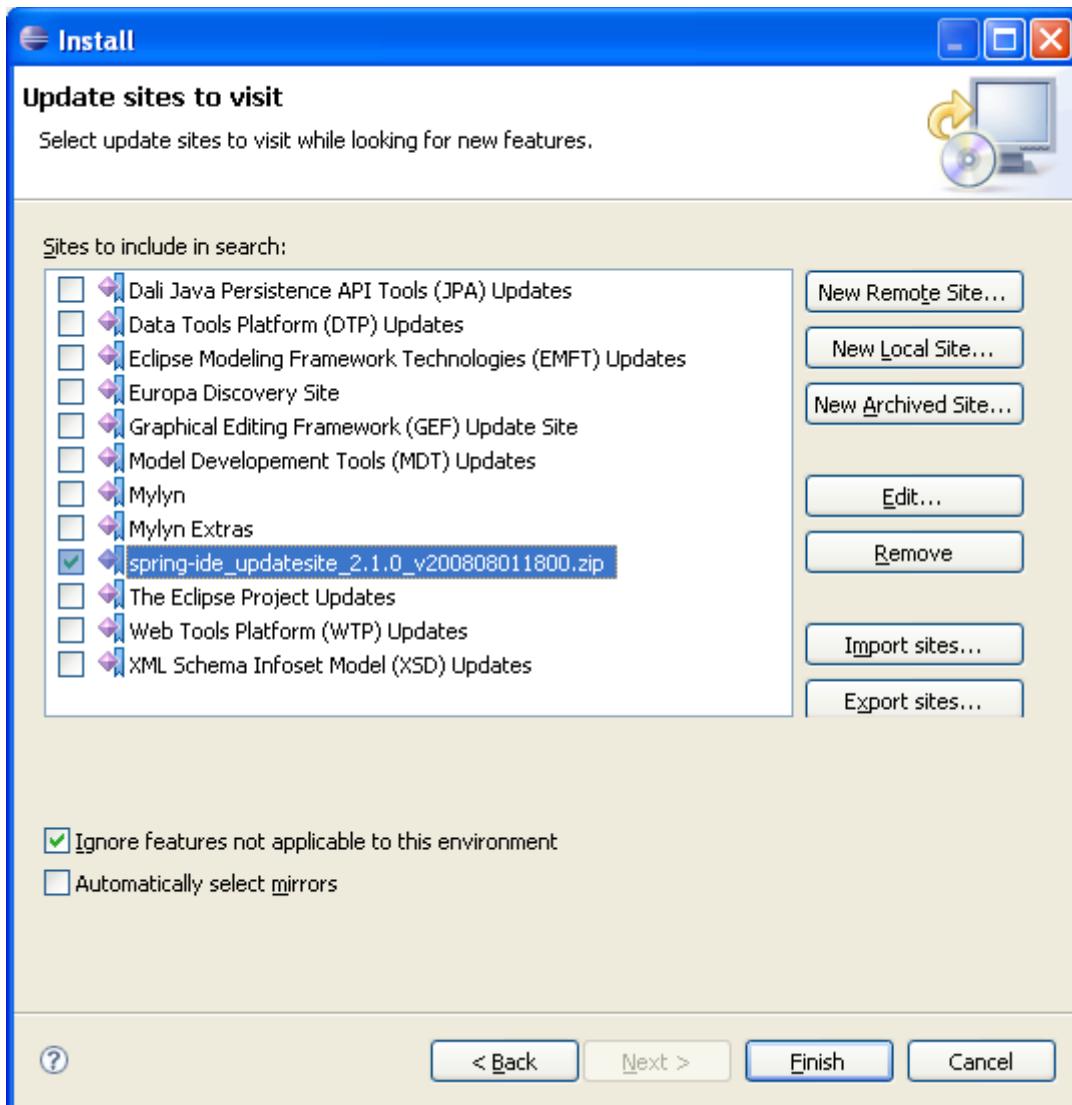
- Choisir l'archive de Spring IDE que vous avez



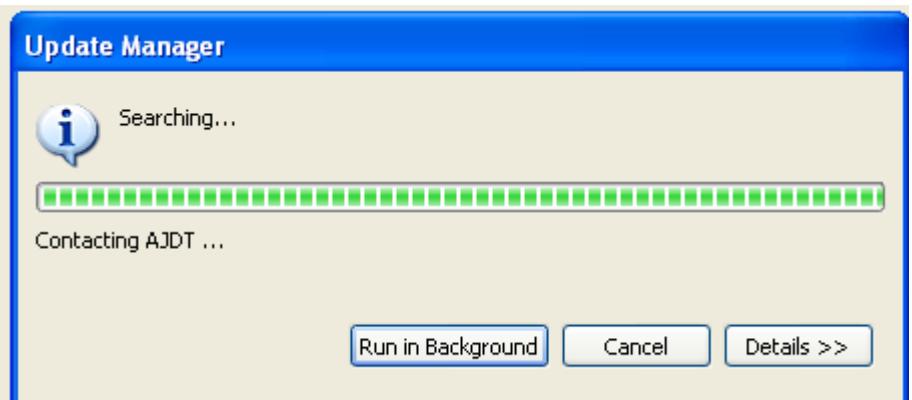


- Valider (Bouton OK)

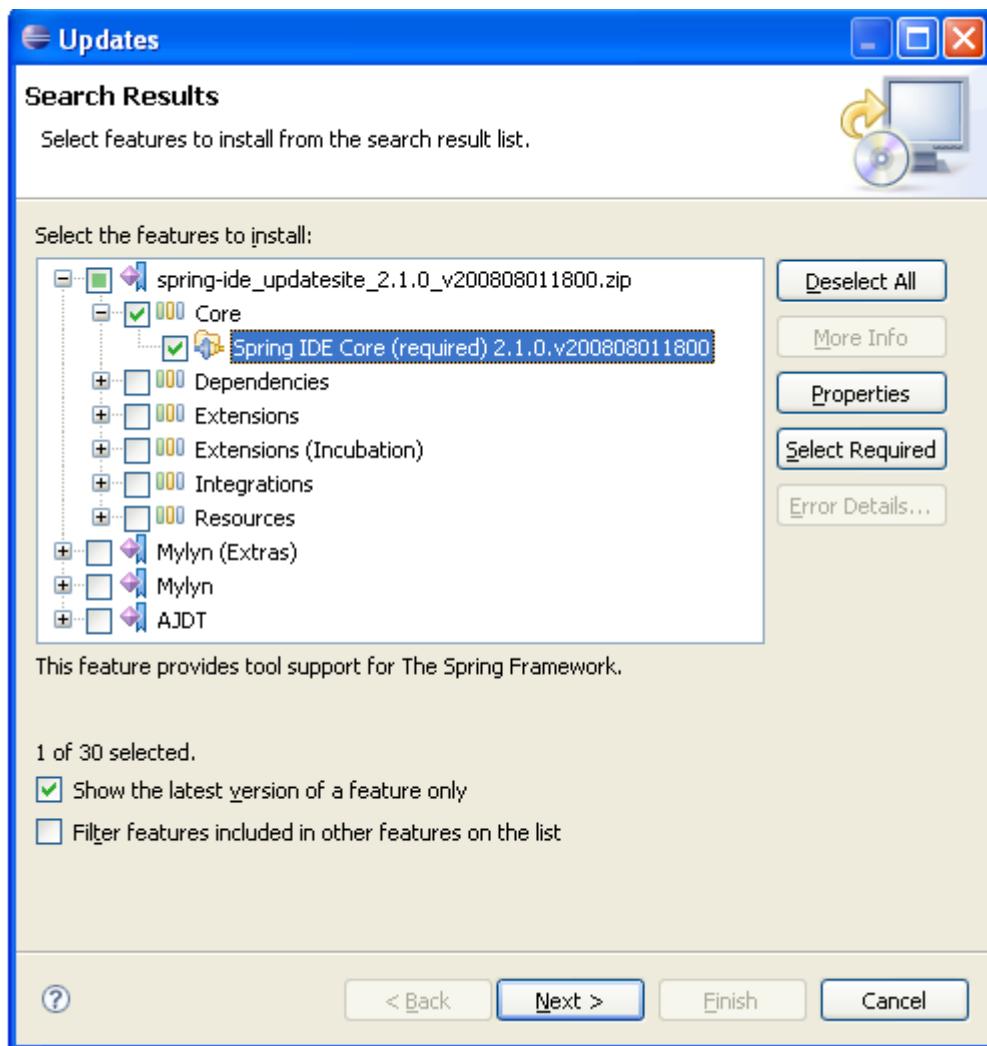
L'archive apparaît alors dans la liste des sites à rechercher



- Cliquez sur bouton 'Finish'



La liste des modules disponibles apparaît .Choisissez module 'Core'. Ne choisissez rien d'autre pour l'instant (Vous installerez des extensions plus tard)



- (Lisez et) Acceptez la licence. Puis Next.

Install

Feature License

Some of the features have license agreements that you need to accept before proceeding with the installation.



Spring IDE Core (required)

SPRING IDE PROJECT SOFTWARE USER AGREEMENT March 26, 2007

Usage Of Content

THE SPRING IDE PROJECT MAKES AVAILABLE SOFTWARE, DOCUMENTATION, INFORMATION AND/OR OTHER MATERIALS FOR OPEN SOURCE PROJECTS (COLLECTIVELY "CONTENT"). USE OF THE CONTENT IS GOVERNED BY THE TERMS AND CONDITIONS OF THIS AGREEMENT AND/OR THE TERMS AND CONDITIONS OF LICENSE AGREEMENTS OR NOTICES INDICATED OR REFERENCED BELOW. BY USING THE CONTENT, YOU AGREE THAT YOUR USE OF THE CONTENT IS GOVERNED BY THIS AGREEMENT AND/OR THE TERMS AND CONDITIONS OF ANY APPLICABLE LICENSE AGREEMENTS

I accept the terms in the license agreement

I do not accept the terms in the license agreement



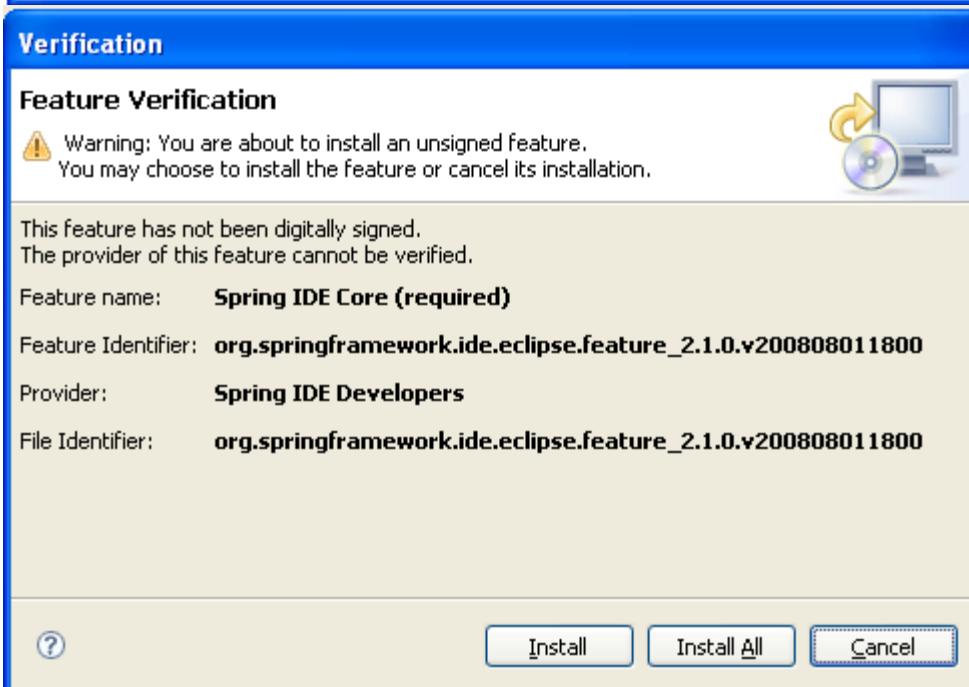
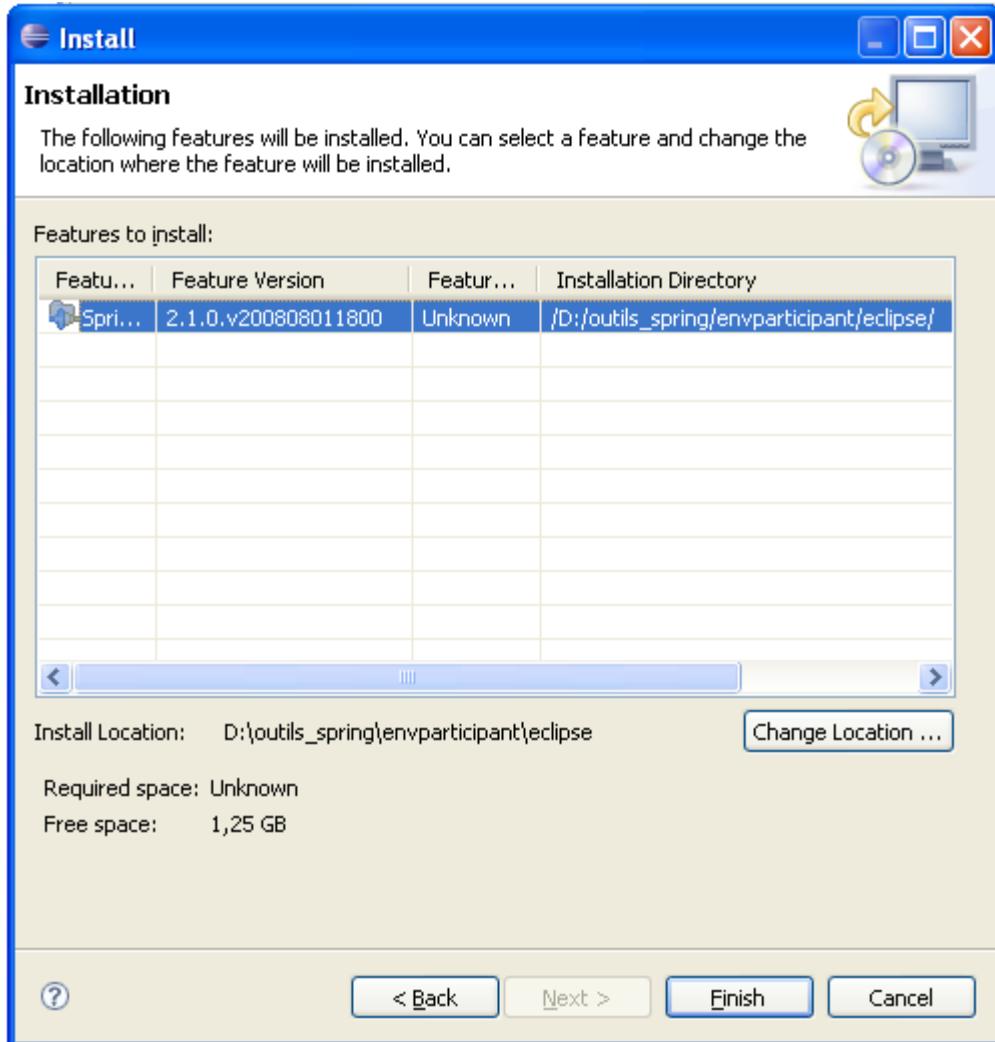
< Back

Next >

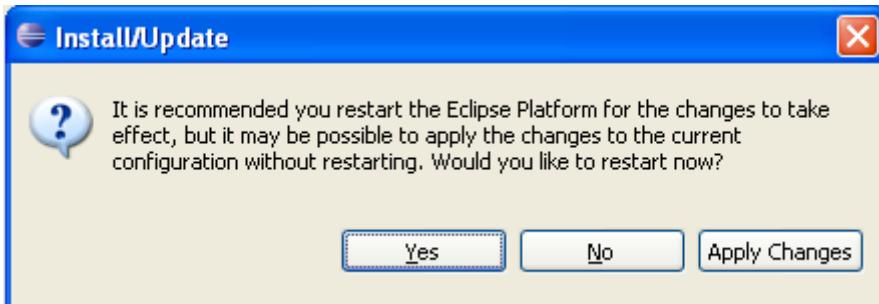
Finish

Cancel

- ▶ Cliquez sur 'Finish', ce qui lance l'installation. Validez messages suivants de vérification



- ▶ L'assistant vous propose de relancer eclipse. Acceptez (YES)



SPRING IDE est désormais installé. Pour mettre en oeuvre Spring IDE, nous vous recommandons notre [**tutoriel Spring N°2 : injection de dépendances.**](#)

Partie 3 : Installation d'extensions (Optionnels)

Vous pourrez en cas de besoin installer d'autres modules, comme l'extension AOP de Spring.

Installer le module Spring AOP

Reprenez la procédure d'installation de Spring IDE. A l'étape du choix de modules à installer, cochez la case du module désiré : Spring AOP. Puis valider.

- ▶ Vérifiez que le module Spring AOP a bien été installé à travers l'ajout de 2 nouvelles vues :
 - Beans Cross References
 - Spring AOP Event Trace

Pour cela , faire Menu window>Show view/Other/Spring

Installer l'extension Spring Web Flow

- ▶ Reprenez la procédure d'installation de Spring IDE. A l'étape du choix de modules à installer, cochez la case du module désiré : Spring Web Flow. Puis valider.

Installer l'extension Spring Security

- ▶ Reprenez la procédure d'installation de Spring IDE. A l'étape du choix de modules à installer, cochez la case du module désiré : Spring Security. Puis valider.

Installer le PlugIn AJDT : AspectJ Development Tool

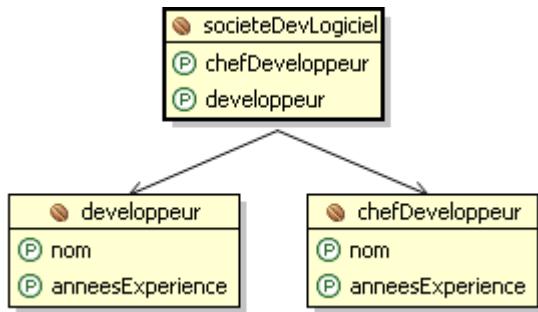
- ▶ Reprenez la procédure d'installation de Spring IDE. A l'étape du choix de modules à installer, cochez la case du module désiré : AJDT. Puis valider.

Conclusion

Dans ce tutoriel, vous vous êtes préparé au développement SPRING de façon productive en installant SPring IDE, un plugin eclipse permettant de mieux développer des applications SPRING.

Vous avez également identifié l'existence de **Spring Source Tool Suite**, un environnement de développement basé sur eclipse, gratuit et disponible (après inscription formulaire)

Tutoriel Spring 3 : injection dépendances



Maîtrisez par la pratique l'injection de dépendances avec la version 3 de Spring, et à travers l'environnement de développement Spring SpringSource Tools Suite (STS)

Objectifs

- ▶ Déclarer dans le conteneur Spring des javabean indépendants
- ▶ Comprendre le chargement du conteneur Spring
- ▶ Mettre en œuvre l'injection de dépendances
- ▶ Utiliser les vues STS, dont la vue graphe.

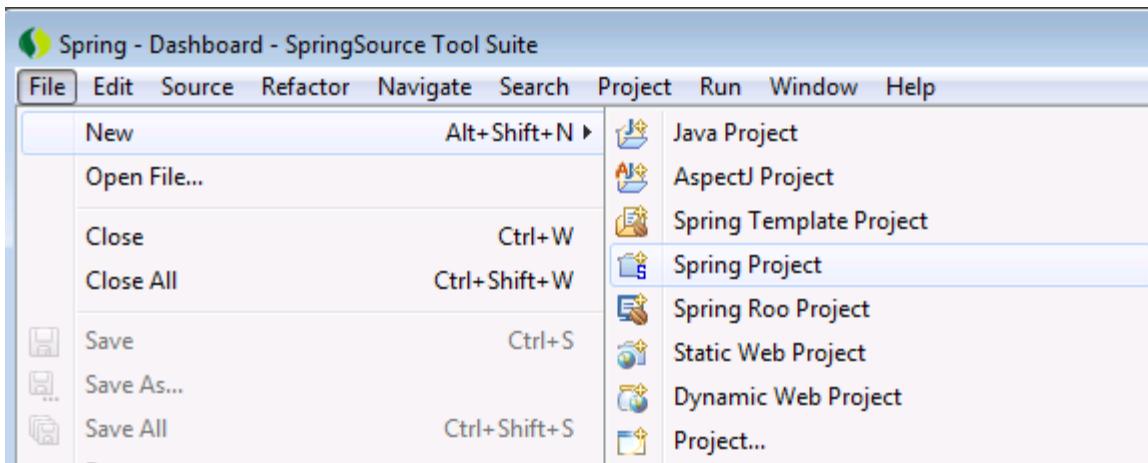
Durée

2h.

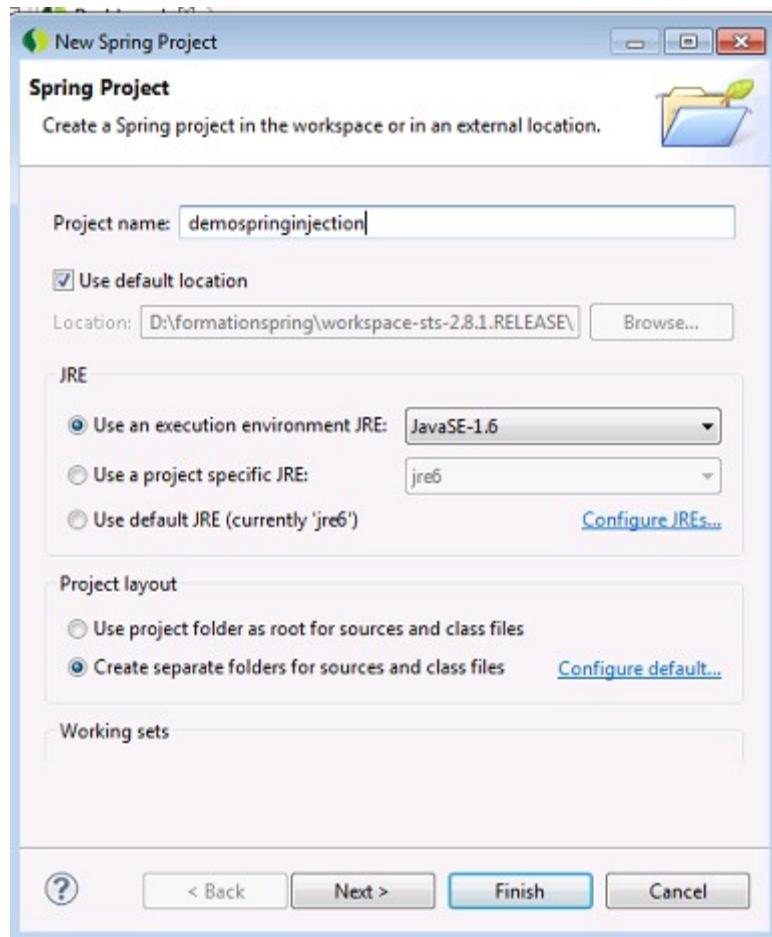
Partie 1 : création projet Spring

Dans cette partie, vous créez un 'projet Spring' à l'aide de l'environnement de développement SpringSource ToolSuite (STS).

- ▶ Dans STS, File/New/Spring Project.

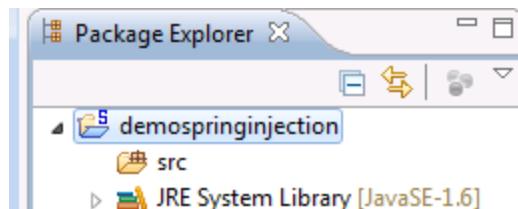


L'écran suivant apparaît :



tutoriel-spring-3-objis-injection-dependances-2bis

- ▶ Entrez comme nom du projet : 'demospringinjection'. Laissez les autres options.
- ▶ Appuyez sur le bouton Finish. Voici ce qui apparaît dans Eclipse



tutoriel-spring-3-sts-objis-injection-dependances-5

- ▶ **REMARQUE** : le 'S' en bleu à gauche du projet signifie que c'est un projet Spring. C'est la marque de projet Spring ou de projets ayant acquis des 'Capacités Spring'.
- ▶ **REMARQUE** : lorsque vous souhaitez donner à un projet non Spring des 'capacités Spring', cliquez droit sur le projet puis 'Spring Tools/Add Spring Project Nature'.

Partie 2 : codage des beans

Dans cette partie, vous créez deux Javabeans indépendants l'un de l'autre.

- ▶ Dans le projet 'demospring', créez un package com.objis.spring.demo

Création du bean Developpeur

- ▶ Créez la classe Developpeur, comme ci-dessous

```

package com.objis.spring.demo;

/*
 * @author Douglas MBIANDOU
 */
public class Developpeur {

    private String nom;
    private int anneesExperience;

    public String getNom() {
        return nom;
    }

    public void setNom(String nom) {
        this.nom = nom;
    }

    public int getAnneesExperience() {
        return anneesExperience;
    }

    public void setAnneesExperience(int anneesExperience) {
        this.anneesExperience = anneesExperience;
    }
}

```

tutoriel-spring-3-sts-objis-injection-dependances-6

- CONSEIL : faîtes générer les Getters / Setters par STS : après la déclaration des attributs 'nom' et 'anneesExperience', cliquez droit sur le source puis '/Source/generate Getters and Setters' pour terminer la création de la classe.

Création du bean SocieteDevLogiciel

- Créez la classe SocieteDevLogiciel, comme ci-dessous

```

package com.objis.spring.demo;

public class SocieteDevLogiciel {

    private Developpeur developpeur;
    private Developpeur chefDeveloppeur;

    public Developpeur getDeveloppeur() {
        return developpeur;
    }

    public void setDeveloppeur(Developpeur developpeur) {
        this.developpeur = developpeur;
    }

    public Developpeur getChefDeveloppeur() {
        return chefDeveloppeur;
    }

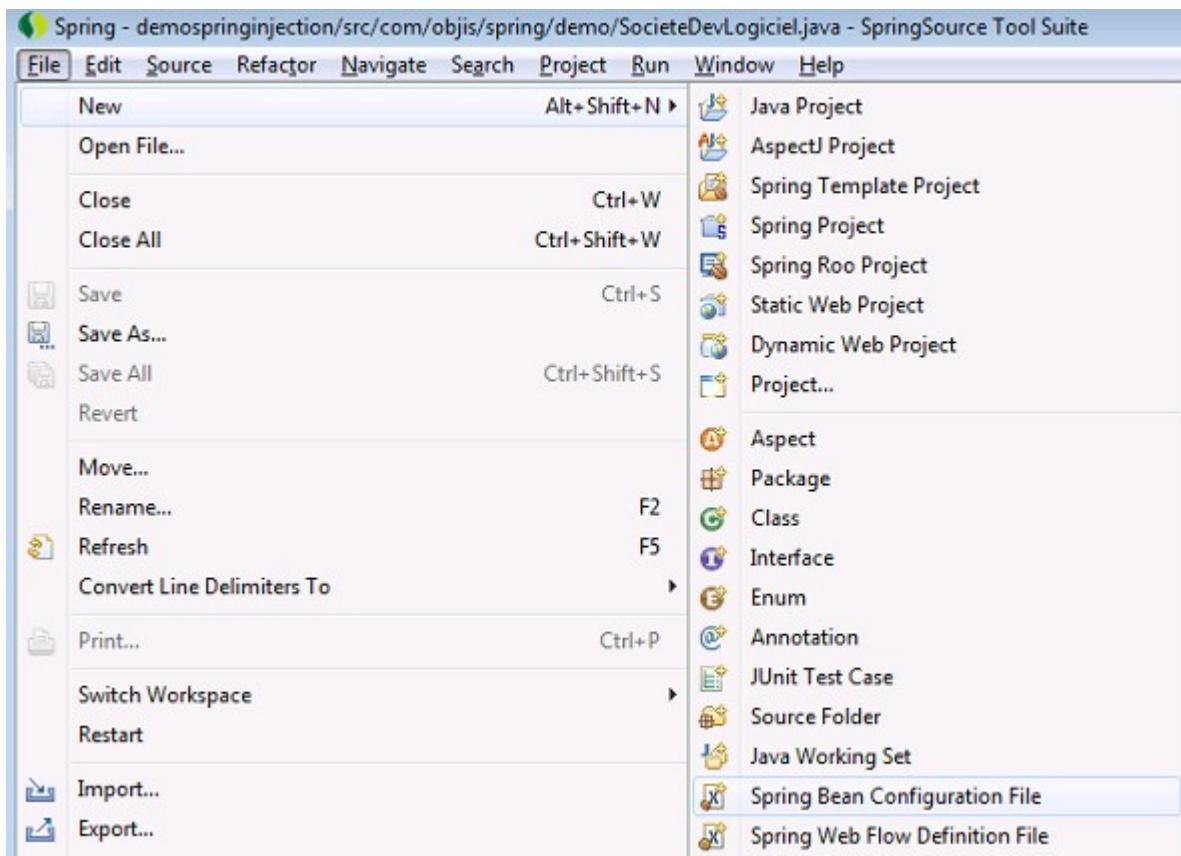
    public void setChefDeveloppeur(Developpeur chefDeveloppeur) {
        this.chefDeveloppeur = chefDeveloppeur;
    }
}

```

tutoriel-spring-3-sts-objis-injection-dependances-7

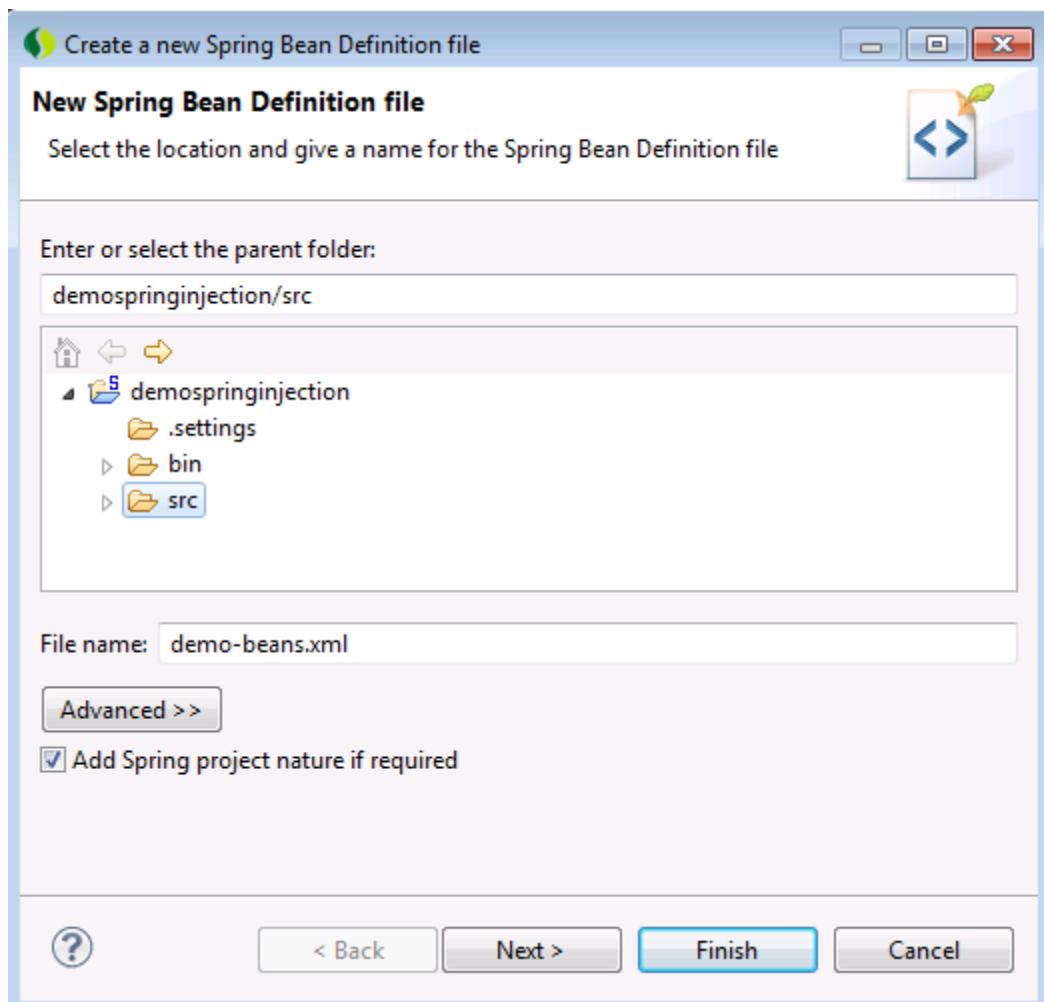
Partie 3 : Création fichier définitions de beans

- Menu : File/New/Spring Bean Definition



tutoriel-spring-3-sts-objis-injection-dependances-8

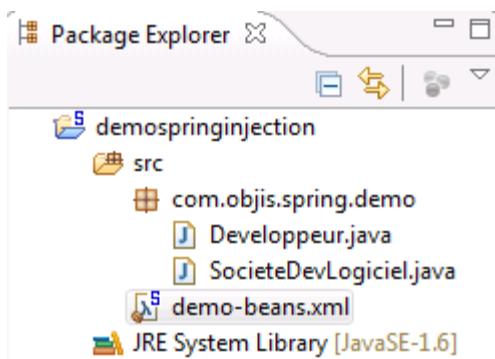
L'écran suivant apparaît :



tutoriel-spring-3-sts-objis-injection-dependances-9

- ▶ Donnez le nom 'demo-beans.xml' au fichier de définition de beans. Puis cliquez sur Finish.

Eclipse génère alors le fichier...



...avec comme contenu les lignes suivantes :

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

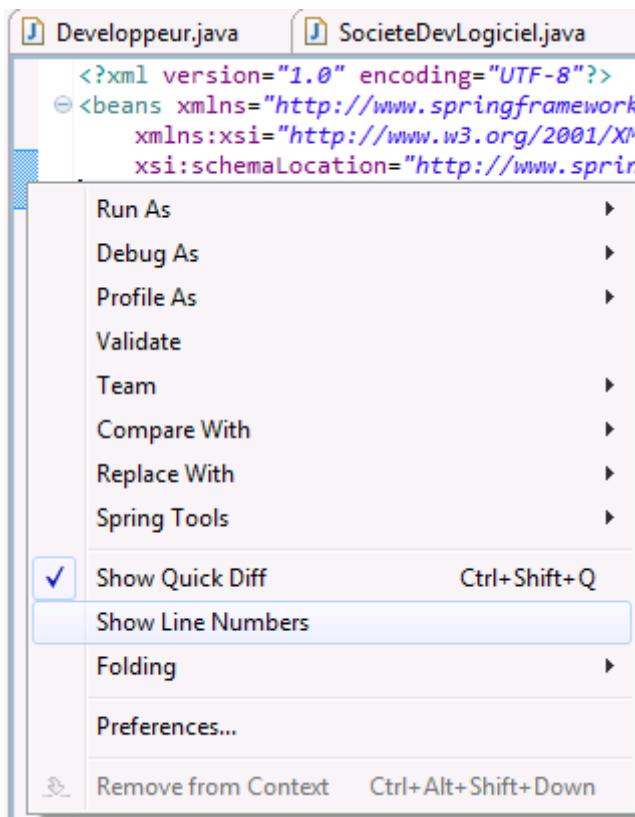
tutoriel-spring-3-sts-objis-injection-dependances-13

- ▶ Expliquez

QUESTION : montrez, en utilisant les préférences Spring de STS, qu'il est possible de spécifier un format spécifique de XSD pour la validation du fichier, en fonction de la version de Spring (Spring 2, Spring 3.0, Spring 3.1...)

En déduire un contenu de demo-spring.xml spécifique à Spring 3.0

- ▶ Ajoutez des lignes au fichier :



tutoriel-spring-3-sts-objis-injection-dependances-14

ce qui donne ceci :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/sch...
```

tutoriel-spring-3-sts-objis-injection-dependances-15

Reste désormais à ajouter la déclaration des deux beans codés plus haut : **Développeur** et **SocieteDevLogiciel**

Partie 4 : Déclaration des beans dans spring

Editeur XML et complétion de code

- Déclarez un Développeur Olivier possédant 7 ans d'expériences.

Pour cela , suite à un CTRL + Espace dans la balise <beans> ,sélectionnez la balise <bean>

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/s
5
6   <> alias
7   <> bean
8   <> beans
9   <> description
10  <> import
11  <> Bean - Inserts a bean tag
12  <> # comment - xml comment
13  <> Web Flow Executor - Inserts a Web Flow Executor tag
14  <> Web Flow Registry - Inserts a Web Flow Registry tag
15  <> # XSL processing instruction - XSL processing instruction
```

Element : alias
Defines an alias for a bean (which can reside in a different definition resource).

tutoriel-spring-3-sts-objis-injection-dependances-16

- Idem pour sélectionner l'attribut 'id'...

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/s
5   <bean></bean>
6 </beans>
```

Attribute : id
The unique identifier for a bean. A bean id may not be used more than once within the same <beans> element.

Data Type : string

tutoriel-spring-3-sts-objis-injection-dependances-17

et l'attribut 'class' de la balise 'bean'

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.springframework.org/schema/beans h
5   <bean id="developpeur" class="Developpeur - com.objis.spring.demo"></bean>
6 </beans>
7
```

Developpeur - com.objis.spring.demo

tutoriel-spring-3-sts-objis-injection-dependances-18

- Injectez les propriétés 'nom' et 'anneesExpereince' de façon statique :

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.springframework.org/schema/beans ht
5 <bean id="developpeur" class="com.objis.spring.demo.Developpeur">
6   <property name="nom">Olivier</property>
7 </bean>
8 </beans>
9
```

anneesExperience - Developpeur.setAnneesExperience(int an
nom - Developpeur.setNom(String nom)

tutoriel-spring-3-sts-objis-injection-dependances-19

Remarquez les 2 façons (attribut ou balise) de déclarer une propriété du bean.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.springframework.org/schema/beans ht
5 <bean id="developpeur" class="com.objis.spring.demo.Developpeur">
6   <property name="nom" value="Olivier"></property>
7   <property name="anneesExperience">
8     <value>7</value>
9   </property>
10 </bean>
11 </beans>
```

tutoriel-spring-3-sts-objis-injection-dependances-20

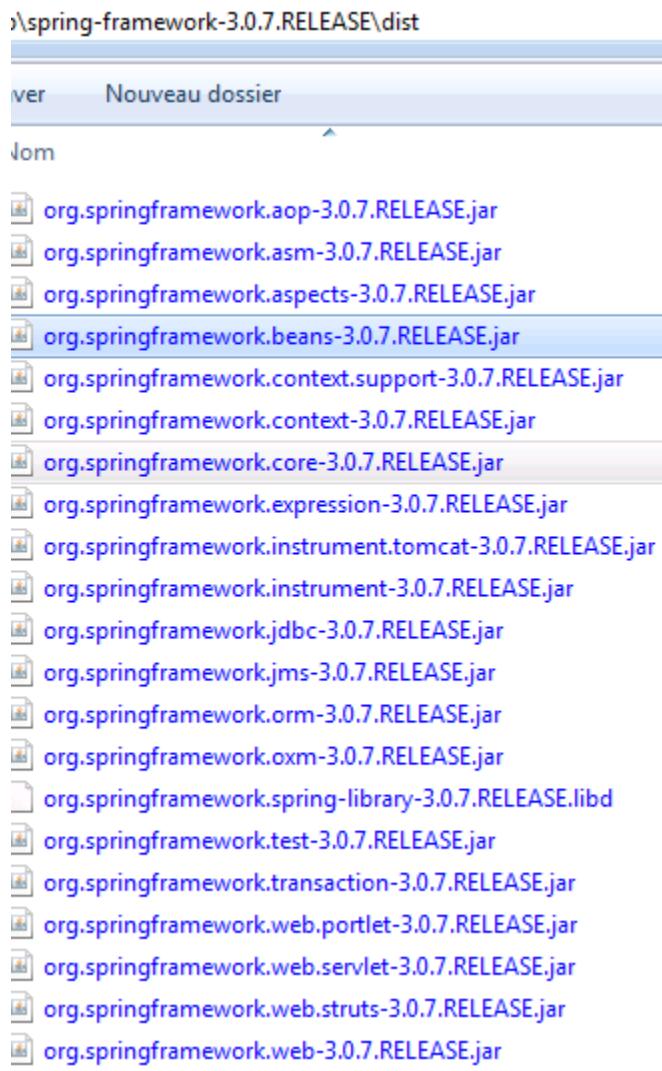
ça y est, la déclaration du bean Developpeur est terminée.

- ▶ **A VOUS DE JOUER :** Déclarez dans le même fichier de configuration demo-beans.xml un deuxième développeur nommé Franck, plus expérimenté qu'Olivier (10 ans d'expérience) et possédant dans le conteneur Spring un id='chefDeveloppeur'.
- ▶ **A VOUS DE JOUER :** Déclarez dans le même fichier de configuration demo-beans.xml une société de développement logicielle possédant les 2 développeurs déjà déclarés, et possédant un id='societeDevLogiciel'. Utilisez l'attribut '**ref**' de la balise bean en pointant sur l'id de chacun des beans.

Partie 5 : librairies du projet spring

- ▶ Pour démarrer un projet spring 3 minimal, 4 dépendances sont à ajouter.
 - 2 dépendances de la distribution Spring : core + beans

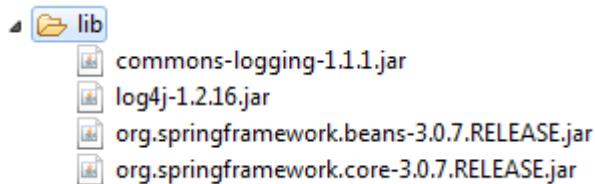
Ces dépendances se trouvent dans le **répertoire 'dist'** du répertoire Spring comme ci-dessous (après avoir dézippé par exemple le zip spring-framework-3.0.7.RELEASE-with-docs.zip téléchargé à partir du site de SpringSource)



tutoriel-spring-3-sts-objis-injection-dependances-20bis

— 2 dépendances hors distribution Spring : log4j ([téléchargez](#)) + commons-logging ([téléchargez](#))

- ▶ créez un répertoire 'lib' à la racine du projet
- ▶ Copiez-collez dans lib les 4 dépendances

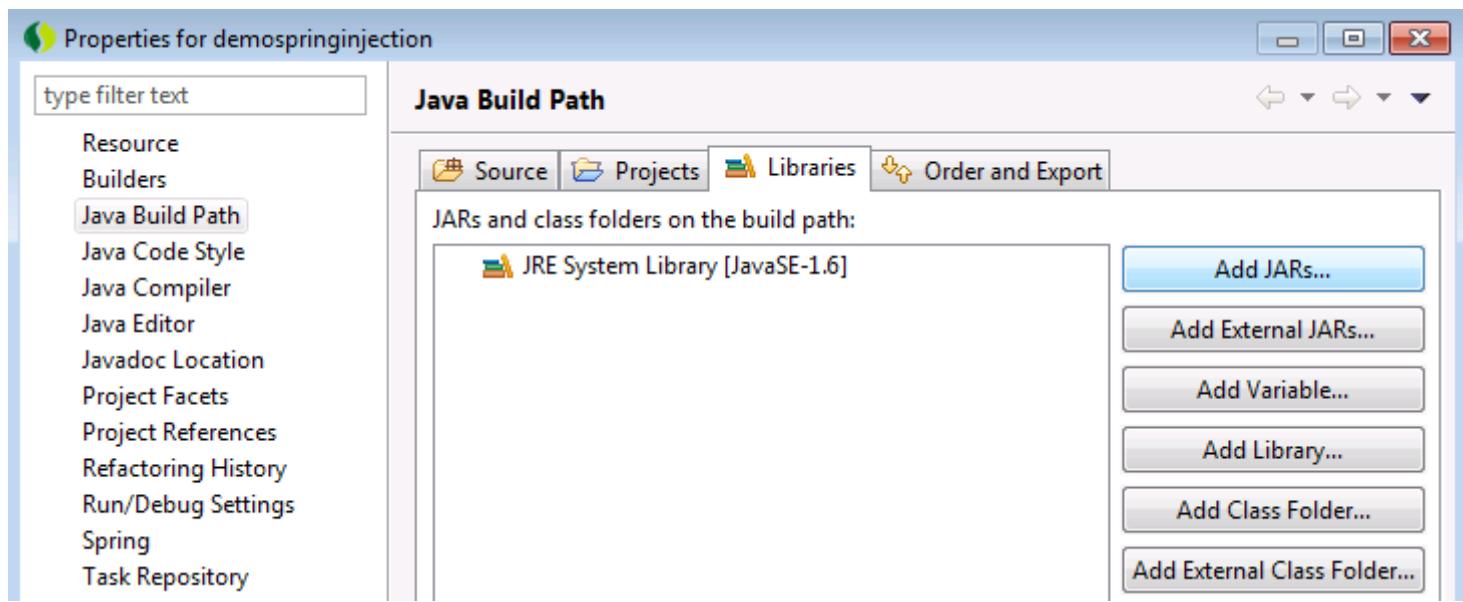


tutoriel-spring-3-sts-objis-injection-dependances-25-0

Attention : actuellement ces librairies ne sont pas prises en compte par Eclipse...nous allons y remédier.

Configuration du build Path dans Eclipse

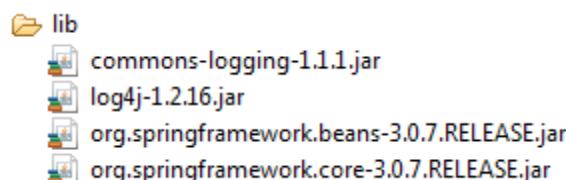
- ▶ Cliquez-droit sur le projet —> configure build path
- ▶ Dans l'écran qui apparaît, sélectionnez l'onglet 'Libraries' et cliquez sur le boutton 'Add jars' qui permet d'ajouter des jars relatifs au projet, comme ceux que nous venons d'ajouter dans 'lib'.



tutoriel-spring-3-sts-objis-injection-dependances-25

- Naviguez jusqu'au répertoire 'lib' du projet et sélectionnez tous les 4 jars. Validez.

L'écran suivant met en évidence que les jars sont bien inclus dans le classpath.



tutoriel-spring-3-sts-objis-injection-dependances-25bis

La configuration des librairies du projet est terminée.

configuration log4j

- Créez à la racine 'src' le fichier **log4j.xml**
- Ajoutez dans le fichier le contenu suivant.

```

1 <?xml version="1.0" encoding="UTF-8" ?>
2 <!DOCTYPE log4j:configuration PUBLIC
3   "http://logging.apache.org/log4j/docs/api/org/apache/log4j/xml/log4j.dtd"
4   "http://logging.apache.org/log4j/docs/api/org/apache/log4j/xml/log4j.dtd">
5
6<log4j:configuration xmlns:log4j="http://jakarta.apache.org/Log4j/">
7
8<appender name="console" class="org.apache.log4j.ConsoleAppender">
9   <layout class="org.apache.log4j.SimpleLayout" />
10</appender>
11
12<root>
13   <level value="info" />
14   <appender-ref ref="console" />
15</root>
16
17</log4j:configuration>

```

tutoriel-spring-3-sts-objis-injection-dependances-26bis

- Expliquez.

Partie 6 : client

- ▶ Codez et analysez le contenu de la classe Principale 'DemoApp' suivante :

The screenshot shows the Eclipse IDE interface. On the left, the Package Explorer view displays a project structure with a 'src' folder containing 'com.objis.spring.demo' and 'com.objis.spring.demo.test' packages, each with a 'DemoApp.java' file. It also shows 'demo-beans.xml', 'log4j.xml', and JRE System Library [JavaSE-1.6]. On the right, the editor view shows the 'DemoApp.java' code:

```
1 package com.objis.spring.demo.test;
2
3 import org.apache.log4j.Logger;
4 import org.springframework.beans.factory.BeanFactory;
5 import org.springframework.beans.factory.xml.XmlBeanFactory;
6 import org.springframework.core.io.ClassPathResource;
7
8 import com.objis.spring.demo.SocieteDevLogiciel;
9
10 public class DemoApp {
11
12     private static final Logger logger = Logger.getLogger(DemoApp.class);
13
14     /**
15      * @param args
16     */
17     public static void main(String[] args) {
18         BeanFactory beanFactory = new XmlBeanFactory(new ClassPathResource("demo-beans.xml"));
19
20         SocieteDevLogiciel societe = (SocieteDevLogiciel) beanFactory.getBean("societeDevLogiciel");
21
22         logger.info("Chef Developpeur: " + societe.getChefDeveloppeur().getNom());
23         logger.info("Developpeur : " + societe.getDeveloppeur().getNom());
24
25     }
26
27 }
28
```

tutoriel-spring-3-sts-objis-injection-dependances-26

- ▶ Expliquez les lignes de code de la méthode main(). En particulier, mettre en évidence les 3 étapes suivantes :

- 1.Chargement du conteneur Spring
- 2.Récupération d'un bean du conteneur
- 3.Utilisation du bean avec injection de dépendances

Résultat

- ▶ Après clic droit/Run as/ Java Application, vous obtenez sur la console ceci

The screenshot shows the Eclipse Console view with the following output:

```
<terminated> DemoApp [Java Application] C:\Program Files (x86)\Java\jre6\bin\javaw.exe ... 10:17:19
INFO - Loading XML bean definitions from class path resource [demo-beans.xml]
INFO - Chef Developpeur: Franck
INFO - Developpeur : Olivier
```

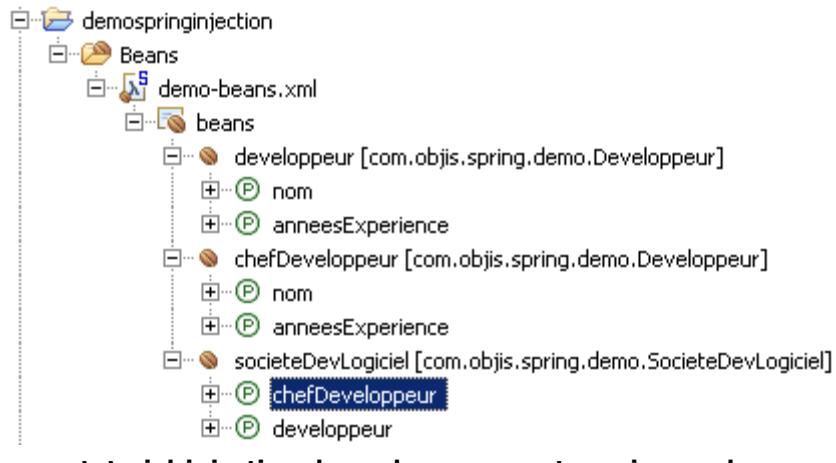
tutoriel-spring-3-sts-objis-injection-dependances-28

- ▶ Expliquez

Vues Spring

Mettez en évidence les beans Spring dans les vues suivantes :

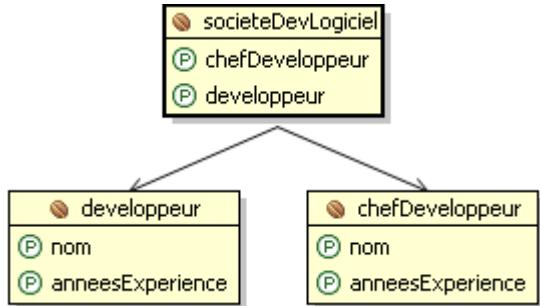
- Vue Spring Explorer



tutoriel-injection-dependances-vue-sts-spring-explorer

Quel avantage pour cette vue ?

— Vue Dependencies Graph



Conclusion

Dans ce tutoriel, vous avez pratiqué la mise en œuvre du concept clé N°1 de Spring : l'injection de dépendances.

L'étape suivante de votre apprentissage Spring : mettre en œuvre le concept clé N°2 de Spring : la programmation orientée aspect.

Constructor Injection Type Ambiguities In Spring

Eclipse Project name: 02_springConstructorInjection

In Spring framework, when your class contains multiple constructors with same number of arguments, it will always cause the **constructor injection argument type ambiguities** issue.

Problem

Let's see this customer bean example. It contains two constructor methods, both accept 3 arguments with different data type.

```
package com.mkyong.common;
public class Customer
{
    private String name;
    private String address;
    private int age;
    public Customer(String name, String address, int age) {
        this.name = name;
        this.address = address;
        this.age = age;
    }
    public Customer(String name, int age, String address) {
        this.name = name;
        this.age = age;
        this.address = address;
    }
    //getter and setter methods
    public String toString(){
        return " name : " +name + "\n address : "
        + address + "\n age : " + age;
    }
}
```

In Spring bean configuration file, pass a 'mkyong' for name, '188' for address and '28' for age.

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

    <bean id="CustomerBean" class="com.mkyong.common.Customer">

        <constructor-arg>
            <value>mkyong</value>
        </constructor-arg>

        <constructor-arg>
            <value>188</value>
        </constructor-arg>

        <constructor-arg>
            <value>28</value>
        </constructor-arg>
    </bean>
</beans>
```

Run it, what's your expected result?

```
package com.mkyong.common;
```

```

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
            new ClassPathXmlApplicationContext(new String[] {"Spring-Customer.xml"]);

        Customer cust = (Customer)context.getBean("CustomerBean");
        System.out.println(cust);
    }
}

```

Output

```

name : mkyong
address : 28
age : 188

```

The result is not what we expected, the second constructor is run, instead of the first constructor. In Spring, the argument type '188' is capable convert to int, so Spring just convert it and take the second constructor, even you assume it should be a String.

In addition, if Spring can't resolve which constructor to use, it will prompt following error message

```

constructor arguments specified but no matching constructor
found in bean 'CustomerBean' (hint: specify index and/or
type arguments for simple parameters to avoid type ambiguities)

```

Solution

To fix it, you should always specify the exact data type for constructor, via type attribute like this :

```

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

    <bean id="CustomerBean" class="com.mkyong.common.Customer">

        <constructor-arg type="java.lang.String">
            <value>mkyong</value>
        </constructor-arg>

        <constructor-arg type="java.lang.String">
            <value>188</value>
        </constructor-arg>

        <constructor-arg type="int">
            <value>28</value>
        </constructor-arg>

    </bean>

```

</beans>

Run it again, now you get what you expected.

Output

```
name : mkyong
address : 188
age : 28
```

Note

It's always a good practice to explicitly declared the data type for each constructor argument, to avoid constructor injection type ambiguities issue above.

[ne pas faire l'exo, juste expliquer le concept]

How To Load Multiple Spring Bean Configuration File

Problem

In a large project structure, the Spring's bean configuration files are located in different folders for easy maintainability and modular. For example, `Spring-Common.xml` in common folder, `Spring-Connection.xml` in connection folder, `Spring-ModuleA.xml` in ModuleA folder...and etc.

You may load multiple Spring bean configuration files in the code :

```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"Spring-Common.xml",
    "Spring-Connection.xml","Spring-ModuleA.xml"});
```

Put all spring xml files under project classpath.

```
project-classpath/Spring-Common.xml
project-classpath/Spring-Connection.xml
project-classpath/Spring-ModuleA.xml
```

Solution

The above ways are lack of organizing and error prone, the better way should be organized all your Spring bean configuration files into a single XML file. For example, create a `Spring-All-Module.xml` file, and import the entire Spring bean files like this :

File : Spring-All-Module.xml

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

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

<import resource="common/Spring-Common.xml"/>
<import resource="connection/Spring-Connection.xml"/>
<import resource="moduleA/Spring-ModuleA.xml"/>

</beans>
```

Now you can load a single xml file like this :

```
ApplicationContext context =
new ClassPathXmlApplicationContext(Spring-All-Module.xml);
```

Put this file under project classpath.

```
project-classpath/Spring-All-Module.xml
```

Spring Inner Bean Examples

Eclipse project name : 03_springInnerBean

In Spring framework, whenever a bean is used for only one particular property, it's advise to declare it as an inner bean. And the inner bean is supported both in setter injection '`property`' and constructor injection '`constructor-arg`'.

See a detail example to demonstrate the use of Spring inner bean.

```
package com.mkyong.common;

public class Customer
{
    private Person person;

    public Customer(Person person) {
        this.person = person;
    }

    public void setPerson(Person person) {
        this.person = person;
    }

    @Override
    public String toString() {
        return "Customer [person=" + person + "]";
    }
}
package com.mkyong.common;

public class Person
{
    private String name;
    private String address;
    private int age;

    //getter and setter methods

    @Override
    public String toString() {
        return "Person [address=" + address + ", "
               + "age=" + age + ", name=" + name + "]";
    }
}
```

Often times, you may use '`ref`' attribute to reference the "Person" bean into "Customer" bean, person property as following :

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

    <bean id="CustomerBean" class="com.mkyong.common.Customer">
        <property name="person" ref="PersonBean" />
    </bean>
```

```

<bean id="PersonBean" class="com.mkyong.common.Person">
    <property name="name" value="mkyong" />
    <property name="address" value="address1" />
    <property name="age" value="28" />
</bean>

</beans>

```

In general, it's fine to reference like this, but since the 'mkyong' person bean is only used for Customer bean only, it's better to declare this 'mkyong' person as an inner bean as following :

```

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

    <bean id="CustomerBean" class="com.mkyong.common.Customer">
        <property name="person">
            <bean class="com.mkyong.common.Person">
                <property name="name" value="mkyong" />
                <property name="address" value="address1" />
                <property name="age" value="28" />
            </bean>
        </property>
    </bean>
</beans>

```

This inner bean also supported in constructor injection as following :

```

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

    <bean id="CustomerBean" class="com.mkyong.common.Customer">
        <constructor-arg>
            <bean class="com.mkyong.common.Person">
                <property name="name" value="mkyong" />
                <property name="address" value="address1" />
                <property name="age" value="28" />
            </bean>
        </constructor-arg>
    </bean>
</beans>

```

Note

The id or name value in bean class is not necessary in an inner bean, it will simply ignored by the Spring container.

Run it

```

package com.mkyong.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
            new ClassPathXmlApplicationContext(new String[] {"Spring-Customer.xml"});
    }
}

```

```

Customer cust = (Customer) context.getBean("CustomerBean");
System.out.println(cust);

}

```

Output

```
Customer [person=Person [address=address1, age=28, name=mkyong]]
```

Spring Bean Scopes Example

In Spring, bean scope is used to decide which type of bean instance should be return from Spring container back to the caller.

5 types of bean scopes supported :

1. singleton – Return a single bean instance per Spring IoC container
2. prototype – Return a new bean instance each time when requested
3. request – Return a single bean instance per HTTP request. *
4. session – Return a single bean instance per HTTP session. *
5. globalSession – Return a single bean instance per global HTTP session. *

In most cases, you may only deal with the Spring's core scope – singleton and prototype, and the default scope is singleton.

*P.S * means only valid in the context of a web-aware Spring ApplicationContext*

Singleton vs Prototype

Here's an example to show you what's the different between bean scope : **singleton** and **prototype**.

```

package com.mkyong.customer.services;

public class CustomerService
{
    String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }
}

```

1. Singleton example

If no bean scope is specified in bean configuration file, default to singleton.

```

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

    <bean id="customerService"
        class="com.mkyong.customer.services.CustomerService" />

</beans>

```

Run it

```

package com.mkyong.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.services.CustomerService;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
            new ClassPathXmlApplicationContext(new String[] { "Spring-Customer.xml" });

        CustomerService custA = (CustomerService)context.getBean("customerService");
        custA.setMessage("Message by custA");
        System.out.println("Message : " + custA.getMessage());

        //retrieve it again
        CustomerService custB = (CustomerService)context.getBean("customerService");
        System.out.println("Message : " + custB.getMessage());
    }
}

```

Output

```

Message : Message by custA
Message : Message by custA

```

Since the bean 'customerService' is in singleton scope, the second retrieval by 'custB' will display the message set by 'custA' also, even it's retrieve by a new getBean() method. In singleton, only a single instance per Spring IoC container, no matter how many time you retrieve it with getBean(), it will always return the same instance.

2. Prototype example

If you want a new 'customerService' bean instance, every time you call it, use prototype instead.

```

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

    <bean id="customerService" class="com.mkyong.customer.services.CustomerService"

```

```
    scope="prototype"/>  
</beans>
```

Run it again

```
Message : Message by custA  
Message : null
```

In prototype scope, you will have a new instance for each `getBean()` method called.

3. Bean scopes annotation

You can also use annotation to define your bean scope.

```
package com.mkyong.customer.services;  
  
import org.springframework.context.annotation.Scope;  
import org.springframework.stereotype.Service;  
  
@Service  
@Scope ("prototype")  
public class CustomerService  
{  
    String message;  
  
    public String getMessage () {  
        return message;  
    }  
  
    public void setMessage (String message) {  
        this.message = message;  
    }  
}
```

Enable auto component scanning

```
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xmlns:context="http://www.springframework.org/schema/context"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans  
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd  
                           http://www.springframework.org/schema/context  
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">  
    <context:component-scan base-package="com.mkyong.customer" />  
</beans>
```

Spring Collections (List, Set, Map, And Properties) Example

Eclipse project name : 04_springCollections

(le code source est dans le dossier :
Presentation cours\Code cours\Spring-Collection-Example)

Spring examples to show you how to inject values into collections type (List, Set, Map, and Properties). 4 major collection types are supported :

- List – <list/>
- Set – <set/>
- Map – <map/>
- Properties – <props/>

Spring beans

A Customer object, with four collection properties.

```
package com.mkyong.common;

import java.util.List;
import java.util.Map;
import java.util.Properties;
import java.util.Set;

public class Customer
{
    private List<Object> lists;
    private Set<Object> sets;
    private Map<Object, Object> maps;
    private Properties pros;

    //...
}
```

See different code snippets to declare collection in bean configuration file.

1. List example

```
<property name="lists">
    <list>
        <value>1</value>
        <ref bean="PersonBean" />
        <bean class="com.mkyong.common.Person">
            <property name="name" value="mkyongList" />
            <property name="address" value="address" />
            <property name="age" value="28" />
        </bean>
    </list>
</property>
```

2. Set example

```
<property name="sets">
    <set>
        <value>1</value>
        <ref bean="PersonBean" />
        <bean class="com.mkyong.common.Person">
            <property name="name" value="mkyongSet" />
            <property name="address" value="address" />
            <property name="age" value="28" />
        </bean>
    </set>
</property>
```

3. Map example

```
<property name="maps">
    <map>
        <entry key="Key 1" value="1" />
        <entry key="Key 2" value-ref="PersonBean" />
        <entry key="Key 3">
            <bean class="com.mkyong.common.Person">
                <property name="name" value="mkyongMap" />
                <property name="address" value="address" />
                <property name="age" value="28" />
            </bean>
        </entry>
    </map>
</property>
```

4. Properties example

```
<property name="pros">
    <props>
        <prop key="admin">admin@nosspam.com</prop>
        <prop key="support">support@nosspam.com</prop>
    </props>
</property>
```

Full Spring's bean configuration file.

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

    <bean id="CustomerBean" class="com.mkyong.common.Customer">
        <!-- java.util.List -->
        <property name="lists">
            <list>
```

```

<value>1</value>
<ref bean="PersonBean" />
<bean class="com.mkyong.common.Person">
    <property name="name" value="mkyongList" />
    <property name="address" value="address" />
    <property name="age" value="28" />
</bean>
</list>
</property>

<!-- java.util.Set -->
<property name="sets">
    <set>
        <value>1</value>
        <ref bean="PersonBean" />
        <bean class="com.mkyong.common.Person">
            <property name="name" value="mkyongSet" />
            <property name="address" value="address" />
            <property name="age" value="28" />
        </bean>
    </set>
</property>

<!-- java.util.Map -->
<property name="maps">
    <map>
        <entry key="Key 1" value="1" />
        <entry key="Key 2" value-ref="PersonBean" />
        <entry key="Key 3">
            <bean class="com.mkyong.common.Person">
                <property name="name" value="mkyongMap" />
                <property name="address" value="address" />
                <property name="age" value="28" />
            </bean>
        </entry>
    </map>
</property>

<!-- java.util.Properties -->
<property name="pros">
    <props>
        <prop key="admin">admin@nospam.com</prop>
        <prop key="support">support@nospam.com</prop>
    </props>
</property>

</bean>

<bean id="PersonBean" class="com.mkyong.common.Person">
    <property name="name" value="mkyong1" />
    <property name="address" value="address 1" />
    <property name="age" value="28" />
</bean>

</beans>

```

Run it...

```

package com.mkyong.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context = new
ClassPathXmlApplicationContext("SpringBeans.xml");

        Customer cust = (Customer)context.getBean("CustomerBean");
        System.out.println(cust);

    }
}

```

Output

```

Customer [
lists=[
1,
Person [address=address 1, age=28, name=mkyong1],
Person [address=address, age=28, name=mkyongList]
],

maps={
key 1=1,
key 2=Person [address=address 1, age=28, name=mkyong1],
key 3=Person [address=address, age=28, name=mkyongMap]
},

pros={admin=admin@nospam.com, support=support@nospam.com},
sets=[
1,
Person [address=address 1, age=28, name=mkyong1],
Person [address=address, age=28, name=mkyongSet]]
]

```

Spring Bean Configuration Inheritance

Eclipse Project name: 04_springBeanHeritage

(le code source est dans le dossier :
Presentation cours\Code cours\Spring-Bean-Inheritance-Example)

In Spring, the inheritance is supported in bean configuration for a bean to share common values, properties or configurations.

A child bean or inherited bean can inherit its parent bean configurations, properties and some attributes. In addition, the child beans are allow to override the inherited value.
See following full example to show you how bean configuration inheritance works in Spring.

```
package com.mkyong.common;

public class Customer {

    private int type;
    private String action;
    private String Country;

    //...
}
```

Bean configuration file

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

    <bean id="BaseCustomerMalaysia" class="com.mkyong.common.Customer">
        <property name="country" value="Malaysia" />
    </bean>

    <bean id="CustomerBean" parent="BaseCustomerMalaysia">
        <property name="action" value="buy" />
        <property name="type" value="1" />
    </bean>

</beans>
```

Above is a 'BaseCustomerMalaysia' bean contains a 'Malaysia' value for country property, and the 'CustomerBean' bean inherited this value from its parent ('BaseCustomerMalaysia').

Run it

```
package com.mkyong.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("SpringBeans.xml");

        Customer cust = (Customer)context.getBean("CustomerBean");
        System.out.println(cust);

    }
}
```

Output

```
Customer [type=1, action=buy, Country=Malaysia]
```

The 'CustomerBean' bean just inherited the country property from its parent ('BaseCustomerMalaysia').

Inheritance with abstract

In above example, the 'BaseCustomerMalaysia' is still able to instantiate, for example,

```
Customer cust = (Customer)context.getBean("BaseCustomerMalaysia");
```

If you want to make this base bean as a template and not allow others to instantiate it, you can add an '**abstract**' attribute in the <bean> element. For example

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

    <bean id="BaseCustomerMalaysia" class="com.mkyong.common.Customer"
          abstract="true">
        <property name="country" value="Malaysia" />
    </bean>

    <bean id="CustomerBean" parent="BaseCustomerMalaysia">
        <property name="action" value="buy" />
        <property name="type" value="1" />
    </bean>
</beans>
```

Now, the 'BaseCustomerMalaysia' bean is a pure template, for bean to inherit it only, if you try to instantiate it, you will encounter the following error message.

```
Customer cust = (Customer) context.getBean("BaseCustomerMalaysia");
org.springframework.beans.factory.BeanIsAbstractException:
    Error creating bean with name 'BaseCustomerMalaysia':
        Bean definition is abstract
```

Pure Inheritance Template

Actually, parent bean is not necessary to define class attribute, often times, you may just need a common property for sharing. Here's is an example

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

    <bean id="BaseCustomerMalaysia" abstract="true">
        <property name="country" value="Malaysia" />
    </bean>

    <bean id="CustomerBean" parent="BaseCustomerMalaysia"
          class="com.mkyong.common.Customer">

        <property name="action" value="buy" />
        <property name="type" value="1" />
    </bean>

</beans>
```

In this case, the 'BaseCustomerMalaysia' bean is a pure template, to share its 'country' property only.

Override it

However, you are still allow to override the inherited value by specify the new value in the child bean. Let's see this example

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

    <bean id="BaseCustomerMalaysia" class="com.mkyong.common.Customer"
          abstract="true">
        <property name="country" value="Malaysia" />
    </bean>

    <bean id="CustomerBean" parent="BaseCustomerMalaysia">
        <property name="country" value="Japan" />
        <property name="action" value="buy" />
        <property name="type" value="1" />
    </bean>

</beans>
```

The 'CustomerBean' bean is just override the parent ('BaseCustomerMalaysia') country property, from 'Malaysia' to 'Japan'.

```
Customer [Country=Japan, action=buy, type=1]
```

Conclusion

The Spring bean configuration inheritance is very useful to avoid the repeated common value or configurations for multiple beans.

Spring Init-Method And Destroy-Method Example

Eclipse Project name: 05_springInitDestroyMethod

(*le code source est dans le dossier :*

Presentation cours\Code cours\Spring-init-method-destroy-method-Example)

Explanations:

Cycle de vie des Beans

Le cycle de vie de chaque Bean comporte une naissance et une mort. Dans le cadre du conteneur léger de Spring, la naissance de l'ensemble des Beans singletons s'effectue au démarrage de celui-ci par défaut. Cela induit un temps de chargement plus long de l'application, mais présente l'avantage de s'assurer dès le démarrage que la création des Beans ne posera pas de problème.

Une fois le Bean créé et ses propriétés (ou collaborateurs) configurées, Spring peut appeler une méthode d'initialisation paramétrée par le développeur. Cette méthode sera particulièrement utile pour vérifier la cohérence de l'état du Bean (s'il lui manque des dépendances, par exemple) ou effectuer une initialisation complexe.

La mort des Beans dépend de leur nature. S'il s'agit de prototypes, ceux-ci disparaissent dès lors que plus aucun objet ne les référence et que le ramasse-miettes a fait son oeuvre. Spring ne conservant pas de référence en interne pour les prototypes, il n'a pas « conscience » de leur mort.

Par contre, il conserve une référence pour chaque singleton dont il a la charge. Il a donc « conscience » de leur mort, ce qui permet de réaliser des traitements lorsque celle-ci survient, par exemple libérer des ressources.

Le lancement (par Spring) d'une méthode avant la destruction d'un Bean n'est donc garanti que pour les singletons.

L'exemple caractéristique d'un Bean devant réagir à son cycle de vie est un pool de connexions, qui va créer les connexions à son démarrage, une fois qu'il connaîtra les paramètres, et fermer ces connexions lors de sa destruction.

Spring propose trois moyens pour appeler des méthodes à la création et à la destruction d'un Bean :

- paramétrage des méthodes dans la configuration XML ;
- implémentation d'interfaces ;
- utilisation d'annotations.

In Spring, you can use **init-method** and **destroy-method** as attribute in bean configuration file for bean to perform certain actions upon initialization and destruction. Alternative to [InitializingBean and DisposableBean interface](#).

Example

Here's an example to show you how to use **init-method** and **destroy-method**.

```

package com.mkyong.customer.services;

public class CustomerService
{
    String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public void initIt() throws Exception {
        System.out.println("Init method after properties are set : " + message);
    }

    public void cleanUp() throws Exception {
        System.out.println("Spring Container is destroy! Customer clean up");
    }
}

```

File : Spring-Customer.xml, define **init-method** and **destroy-method** attribute in your bean.

```

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

    <bean id="customerService" class="com.mkyong.customer.services.CustomerService"
          init-method="initIt" destroy-method="cleanUp">

        <property name="message" value="I'm property message" />
    </bean>
</beans>

```

Run it

```

package com.mkyong.common;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.services.CustomerService;

public class App
{
    public static void main( String[] args )
    {
        ConfigurableApplicationContext context =
            new ClassPathXmlApplicationContext(new String[] {"Spring-
Customer.xml"});

        CustomerService cust = (CustomerService)context.getBean("customerService");

        System.out.println(cust);

        context.close();
    }
}

```

The ConfigurableApplicationContext.close will close the application context, releasing all resources and destroying all cached singleton beans.

Output

```

Init method after properties are set : i'm property message
com.mkyong.customer.services.CustomerService@47393f
...
INFO: Destroying singletons in org.springframework.beans.factory.
support.DefaultListableBeanFactory@77158a:
defining beans [customerService]; root of factory hierarchy
Spring Container is destroy! Customer clean up

```

The `initIt()` method is called, after the message property is set, and the `cleanUp()` method is called after the `context.close()`;

Thoughts...

It's always recommended to use `init-method` and `destroy-method` in bean configuration file, instead of implement the `InitializingBean` and `DisposableBean` interface to cause unnecessarily coupled your code to Spring.

Spring @PostConstruct And @PreDestroy Example

Eclipse Project name: *06_springInitDestroyMethodAnnotations*

(le code source est dans le dossier :

Presentation cours\Code cours\Spring-PostConstruct-PreDestroy-Example)

In Spring, you can either implements `InitializingBean` and `DisposableBean` interface or specify the `init-method` and `destroy-method` in bean configuration file for the initialization and destruction callback function. In this article, we show you how to use annotation `@PostConstruct` and `@PreDestroy` to do the same thing.

Note

The `@PostConstruct` and `@PreDestroy` annotation are not belong to Spring, it's located in the J2ee library – common-annotations.jar.

@PostConstruct and @PreDestroy

A CustomerService bean with `@PostConstruct` and `@PreDestroy` annotation

```

package com.mkyong.customer.services;

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

public class CustomerService
{
    String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

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

    @PreDestroy

```

```

    public void cleanUp() throws Exception {
        System.out.println("Spring Container is destroy! Customer clean up");
    }
}

```

By default, Spring will not aware of the @PostConstruct and @PreDestroy annotation. To enable it, you have to either register 'CommonAnnotationBeanPostProcessor' or specify the '<context:annotation-config />' in bean configuration file,

1. CommonAnnotationBeanPostProcessor

```

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

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

    <bean id="customerService"
        class="com.mkyong.customer.services.CustomerService">
        <property name="message" value="i'm property message" />
    </bean>

</beans>

```

2. <context:annotation-config />

```

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

    <context:annotation-config />

    <bean id="customerService"
        class="com.mkyong.customer.services.CustomerService">
        <property name="message" value="i'm property message" />
    </bean>

</beans>

```

Run it

```

package com.mkyong.common;

import org.springframework.context.ConfigurableApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.services.CustomerService;

public class App
{
    public static void main( String[] args )
    {
        ConfigurableApplicationContext context =
            new ClassPathXmlApplicationContext( new String[] { "Spring-Customer.xml" });

        CustomerService cust = (CustomerService) context.getBean("customerService");

        System.out.println(cust);

        context.close();
    }
}

```

```
}
```

Output

```
Init method after properties are set : im property message
com.mkyong.customer.services.CustomerService@47393f
...
INFO: Destroying singletons in org.springframework.beans.factory.
support.DefaultListableBeanFactory@77158a:
defining beans [customerService]; root of factory hierarchy
Spring Container is destroy! Customer clean up
```

The **initIt()** method (**@PostConstruct**) is called, after the message property is set, and the **cleanUp()** method (**@PreDestroy**) is call after the `context.close()`;

Spring EL Hello World Example

Eclipse Project name: 07_springELHelloWorld

(*le code source est dans le dossier :*
Presentation cours\Code cours\Spring3-EL-Hello-Worldr-Example)

The Spring EL is similar with OGNL and JSF EL, and evaluated or executed during the bean creation time. In addition, all Spring expressions are available via XML or annotation.

In this tutorial, we show you how to use **Spring Expression Language(SpEL)**, to inject String, integer and bean into property, both in XML and annotation.

1. Spring EL Dependency

Declares the core Spring jars in Maven `pom.xml` file, it will download the Spring EL dependencies automatically.
File : `pom.xml`

```
<properties>
    <spring.version>3.0.5.RELEASE</spring.version>
</properties>

<dependencies>

    <!-- Spring 3 dependencies -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>

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

</dependencies>
```

2. Spring Beans

Two simple beans, later use SpEL to inject values into property, in XML and annotation.

```
package com.mkyong.core;

public class Customer {

    private Item item;

    private String itemName;

}

package com.mkyong.core;

public class Item {

    private String name;

    private int qty;

}
```

3. Spring EL in XML

The SpEL are enclosed with `#{ SpEL expression }`, see following example in XML bean definition file.

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

    <bean id="itemBean" class="com.mkyong.core.Item">
        <property name="name" value="itemA" />
        <property name="qty" value="10" />
    </bean>

    <bean id="customerBean" class="com.mkyong.core.Customer">
        <property name="item" value="#{itemBean}" />
        <property name="itemName" value="#{itemBean.name}" />
    </bean>

</beans>
```

1. `#{itemBean}` – inject “itemBean” into “customerBean” bean’s “item” property.
2. `#{itemBean.name}` – inject “itemBean”’s “name” property into “customerBean” bean’s “itemName” property.

4. Spring EL in Annotation

See equivalent version in annotation mode.

Note

To use SpEL in annotation, you must register your component via annotation. If you register your bean in XML and define `@Value` in Java class, the `@Value` will failed to execute.

```
package com.mkyong.core;
```

```

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("customerBean")
public class Customer {

    @Value("#{itemBean}")
    private Item item;

    @Value("#{itemBean.name}")
    private String itemName;

    //...

}

package com.mkyong.core;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("itemBean")
public class Item {

    @Value("itemA") //inject String directly
    private String name;

    @Value("10") //inject interger directly
    private int qty;

    public String getName() {
        return name;
    }

    //...
}

```

Enable auto component scanning.

```

<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-3.0.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-3.0.xsd">

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

</beans>

```

In annotation mode, you use `@Value` to define Spring EL. In this case, you inject a String and Integer value directly into the "itemBean", and later inject the "itemBean" into "customerBean" property.

5. Output

Run it, both SpEL in XML and annotation are display the same result :

```

package com.mkyong.core;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

```

```

public class App {
    public static void main(String[] args) {
        ApplicationContext context = new
ClassPathXmlApplicationContext("SpringBeans.xml");

        Customer obj = (Customer) context.getBean("customerBean");
        System.out.println(obj);
    }
}

```

Output

```
Customer [item=Item [name=itemA, qty=10], itemName=itemA]
```

Spring EL Method Invocation Example

Eclipse Project name: 08_springELMethodInvocation

(le code source est dans le dossier :

Presentation cours\Code cours\Spring3-EL-Method-Invocation-Example)

Spring expression language (SpEL) allow developer uses expression to execute method and inject the method returned value into property, or so called “SpEL method invocation”.

Spring EL in Annotation

See how to do Spring EL method invocation with `@Value` annotation.

```

package com.mkyong.core;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("customerBean")
public class Customer {

    @Value("#{ 'mkyong'.toUpperCase() }")
    private String name;

    @Value("#{ priceBean.getSpecialPrice() }")
    private double amount;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public double getAmount() {
        return amount;
    }

    public void setAmount(double amount) {
        this.amount = amount;
    }

    @Override
}

```

```

public String toString() {
    return "Customer [name=" + name + ", amount=" + amount + "]";
}

}

package com.mkyong.core;

import org.springframework.stereotype.Component;

@Component("priceBean")
public class Price {

    public double getSpecialPrice() {
        return new Double(99.99);
    }
}

```

Output

Customer [name=MKYONG, amount=99.99]

Explanation

Call the 'toUpperCase()' method on the string literal.

```

@Value("#{ 'mkyong'.toUpperCase() }")
private String name;

```

Call the 'getSpecialPrice()' method on bean 'priceBean'.

```

@Value("#{priceBean.getSpecialPrice() }")
private double amount;

```

Spring EL in XML

This is the equivalent version in bean definition XML file.

```

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

    <bean id="customerBean" class="com.mkyong.core.Customer">
        <property name="name" value="#{'mkyong'.toUpperCase()}" />
        <property name="amount" value="#{priceBean.getSpecialPrice()}" />
    </bean>

    <bean id="priceBean" class="com.mkyong.core.Price" />

```

Output

Customer [name=MKYONG, amount=99.99]

Spring EL Ternary Operator (If-Then-Else) Example

Eclipse Project name: 09_springOperateurTernaire

(le code source est dans le dossier :

Presentation cours\Code cours\Spring3-EL-Ternary-Operator-Example)

Spring EL supports ternary operator , perform “if then else” conditional checking. For example,

```
condition ? true : false
```

Spring EL in Annotation

Spring EL ternary operator with @Value annotation. In this example, if “itemBean.qtyOnHand” is less than 100, then set “customerBean.warning” to true, else set it to false.

```
package com.mkyong.core;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("customerBean")
public class Customer {

    @Value("#{itemBean.qtyOnHand < 100 ? true : false}")
    private boolean warning;

    public boolean isWarning() {
        return warning;
    }

    public void setWarning(boolean warning) {
        this.warning = warning;
    }

    @Override
    public String toString() {
        return "Customer [warning=" + warning + "]";
    }
}

package com.mkyong.core;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("itemBean")
public class Item {

    @Value("99")
    private int qtyOnHand;

    public int getQtyOnHand() {
        return qtyOnHand;
    }

    public void setQtyOnHand(int qtyOnHand) {
        this.qtyOnHand = qtyOnHand;
    }
}
```

Output

```
Customer [warning=true]
```

Spring EL in XML

See equivalent version in bean definition XML file.

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

```

xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

<bean id="customerBean" class="com.mkyong.core.Customer">
    <property name="warning"
              value="#{itemBean.qtyOnHand < 100 ? true : false}" />
</bean>

<bean id="itemBean" class="com.mkyong.core.Item">
    <property name="qtyOnHand" value="99" />
</bean>

</beans>

```

Output

```
Customer [warning=true]
```

In XML, you need to replace less than operator “<” with “<“.

Spring EL Lists, Maps Example

Eclipse Project name: 10_springELCollections

(le code source est dans le dossier :
Presentation cours\Code cours\Spring3-EL-Map-List-Example)

In this article, we show you how to use Spring EL to get value from Map and List. Actually, the way of SpEL works with Map and List is exactly same with Java. See example :

```

//get map where key = 'MapA'
@Value("#{testBean.map['MapA']}")
private String mapA;

//get first value from list, list is 0-based.
@Value("#{testBean.list[0]}")
private String list;

```

Spring EL in Annotation

Here, created a `HashMap` and `ArrayList`, with some initial data for testing.

```

package com.mkyong.core;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("customerBean")
public class Customer {

    @Value("#{testBean.map['MapA']}")
    private String mapA;

    @Value("#{testBean.list[0]}")
    private String list;

    public String getMapA() {
        return mapA;
    }
}

```

```

public void setMapA(String mapA) {
    this.mapA = mapA;
}

public String getList() {
    return list;
}

public void setList(String list) {
    this.list = list;
}

@Override
public String toString() {
    return "Customer [mapA=" + mapA + ", list=" + list + "]";
}

}

package com.mkyong.core;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import org.springframework.stereotype.Component;

@Component("testBean")
public class Test {

    private Map<String, String> map;
    private List<String> list;

    public Test() {
        map = new HashMap<String, String>();
        map.put("MapA", "This is A");
        map.put("MapB", "This is B");
        map.put("MapC", "This is C");

        list = new ArrayList<String>();
        list.add("List0");
        list.add("List1");
        list.add("List2");
    }

    public Map<String, String> getMap() {
        return map;
    }

    public void setMap(Map<String, String> map) {
        this.map = map;
    }

    public List<String> getList() {
        return list;
    }

    public void setList(List<String> list) {
        this.list = list;
    }

}

```

Run it

```

Customer obj = (Customer) context.getBean("customerBean");
System.out.println(obj);

```

Output

```
Customer [mapA=This is A, list=List0]
```

Spring EL in XML

See equivalent version in bean definition XML file.

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

    <bean id="customerBean" class="com.mkyong.core.Customer">
        <property name="mapA" value="#{testBean.map['MapA']}" />
        <property name="list" value="#{testBean.list[0]}" />
    </bean>

    <bean id="testBean" class="com.mkyong.core.Test" />

</beans>
```

Spring EL Regular Expression Example

Eclipse Project name: 11_springRegExp

(*le code source est dans le dossier :*

Presentation cours\Code cours\Spring3-EL-Regular-Expression-Example)

Spring EL supports regular expression using a simple keyword “matches”, which is really awesome! For examples,

```
@Value("#{'100' matches '\\d+' }")
private boolean isDigit;
```

It test whether ‘100’ is a valid digit via regular expression ‘\\d+’.

Explanation :

\\d is called a character class and will match digits. It is equal to [0-9].

+ matches 1 or more occurrences of the character before.

So \\d+ means match 1 or more digits.

Spring EL in Annotation

See following Spring EL regular expression examples, some mixed with ternary operator, which makes Spring EL pretty flexible and powerful.

Below example should be self-explanatory.

```
package com.mkyong.core;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("customerBean")
public class Customer {
```

```

// email regular expression
String emailRegEx = "^[_A-Za-z0-9-]+(\.\[_A-Za-z0-9-\]+)" +
    "*@[A-Za-z0-9]+(\.\[A-Za-z0-9\]+)*(\.\[A-Za-z\]{2,\})$";

// if this is a digit?
@Value("#{100 matches '\\d+' }")
private boolean validDigit;

// if this is a digit + ternary operator
@Value("#{('100' matches '\\d+') == true ? " +
        "'yes this is digit' : 'No this is not a digit' }")
private String msg;

// if this emailBean.emailAddress contains a valid email address?
@Value("#{emailBean.emailAddress matches customerBean.emailRegEx}")
private boolean validEmail;

//getter and setter methods, and constructor
}

package com.mkyong.core;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.stereotype.Component;

@Component("emailBean")
public class Email {

    @Value("nospam@abc.com")
    String emailAddress;

    //...
}

```

Output

```
Customer [isValidEmail=true, msg=yes this is digit, isDigit=true]
```

Spring EL in XML

See equivalent version in bean definition XML file.

```

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

    <bean id="customerBean" class="com.mkyong.core.Customer">
        <property name="validDigit" value="#{'100' matches '\\d+' }" />
        <property name="msg"
                  value="#{('100' matches '\\d+') == true ? 'yes this is digit' : 'No this
is not a digit' }" />
        <property name="validEmail"
                  value="#{emailBean.emailAddress matches '^[_A-Za-z0-9-]+\.\[_A-Za-z0-9-]
+)*@[A-Za-z0-9]+(\.\[A-Za-z0-9\]+)*(\.\[A-Za-z\]{2,\})$' }" />
    </bean>

    <bean id="emailBean" class="com.mkyong.core.Email">
        <property name="emailAddress" value="nospam@abc.com" />
    </bean>

```

Spring Auto Scanning Components

Eclipse Project name: 12_springAutoScanningComponents

(le code source est dans le dossier :

Presentation cours\Code cours\Spring-Auto-Scan-Component-Example)

Normally you declare all the beans or components in XML bean configuration file, so that Spring container can detect and register your beans or components. Actually, Spring is able to auto scan, detect and instantiate your beans from pre-defined project package, no more tedious beans declaration in XML file.

Following is a simple Spring project, including a customer service and dao layer. Let's explore the different between declare components manually and auto components scanning in Spring.

1. Declares Components Manually

See a normal way to declare a bean in Spring.

Normal bean.

```
package com.mkyong.customer.dao;

public class CustomerDAO
{
    @Override
    public String toString() {
        return "Hello , This is CustomerDAO";
    }
}
```

DAO layer.

```
package com.mkyong.customer.services;

import com.mkyong.customer.dao.CustomerDAO;

public class CustomerService
{
    CustomerDAO customerDAO;

    public void setCustomerDAO(CustomerDAO customerDAO) {
        this.customerDAO = customerDAO;
    }

    @Override
    public String toString() {
        return "CustomerService [customerDAO=" + customerDAO + "]";
    }
}
```

Bean configuration file (Spring-Customer.xml), a normal bean configuration in Spring.

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

    <bean id="customerService"
          class="com.mkyong.customer.services.CustomerService">
        <property name="customerDAO" ref="customerDAO" />
    </bean>

    <bean id="customerDAO" class="com.mkyong.customer.dao.CustomerDAO" />

```

```
</beans>
```

Run it

```
package com.mkyong.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.services.CustomerService;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
            new ClassPathXmlApplicationContext(new String[] {"Spring-Customer.xml"});

        CustomerService cust = (CustomerService)context.getBean("customerService");
        System.out.println(cust);

    }
}
```

output

```
CustomerService [customerDAO=Hello , This is CustomerDAO]
```

2. Auto Components Scanning

Now, enable Spring auto component scanning features.

Annotate with **@Component** to indicate this is a scan component.

```
package com.mkyong.customer.dao;

import org.springframework.stereotype.Component;

@Component
public class CustomerDAO
{
    @Override
    public String toString() {
        return "Hello , This is CustomerDAO";
    }
}
```

DAO layer, add **@Component** to indicate this is an auto scan component also.

```
package com.mkyong.customer.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

import com.mkyong.customer.dao.CustomerDAO;

@Component
public class CustomerService
{
    @Autowired
    CustomerDAO customerDAO;

    @Override
    public String toString() {
        return "CustomerService [customerDAO=" + customerDAO + "]";
    }
}
```

Put this “`context:component`” in bean configuration file, it means, enable auto scanning feature in Spring.

The **base-package** is indicate where are your components stored, Spring will scan this folder and find out the bean (annotated with `@Component`) and register it in Spring container.

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

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

</beans>
```

Run it

```
package com.mkyong.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.customer.services.CustomerService;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
            new ClassPathXmlApplicationContext( new String[] { "Spring-AutoScan.xml" } );

        CustomerService cust = (CustomerService) context.getBean("customerService");
        System.out.println(cust);

    }
}
```

output

```
CustomerService [customerDAO=Hello , This is CustomerDAO]
```

This is how auto components scanning works in Spring.

Custom auto scan component name

By default, Spring will lower case the first character of the component – from ‘CustomerService’ to ‘customerService’. And you can retrieve this component with name ‘customerService’.

```
CustomerService cust = (CustomerService) context.getBean("customerService");
```

To create a custom name for component, you can put custom name like this :

```
@Service ("AAA")
public class CustomerService
...
```

Now, you can retrieve it with this name ‘AAA’.

```
CustomerService cust = (CustomerService) context.getBean("AAA");
```

Auto Components Scan Annotation Types

In Spring 2.5, there are 4 types of auto components scan annotation types

- `@Component` – Indicates a auto scan component.
- `@Repository` – Indicates DAO component in the persistence layer.
- `@Service` – Indicates a Service component in the business layer.
- `@Controller` – Indicates a controller component in the presentation layer.

So, which one to use? It’s really doesn’t matter. Let see the source code

of `@Repository`, `@Service` or `@Controller`.

```

@Target({ElementType.TYPE})
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Component
public interface Repository {

    String value() default "";
}

```

You will noticed that all `@Repository`, `@Service` or `@Controller` are annotated with `@Component`. So, can we use just `@Component` for all the components for auto scanning? Yes, you can, and Spring will auto scan all your components with `@Component` annotated.

It's working fine, but not a good practice, for readability, you should always declare `@Repository`, `@Service` or `@Controller` for a specified layer to make your code more easier to read, as following :

DAO layer

```

package com.mkyong.customer.dao;

import org.springframework.stereotype.Repository;

@Repository
public class CustomerDAO
{
    @Override
    public String toString() {
        return "Hello , This is CustomerDAO";
    }
}

```

Service layer

```

package com.mkyong.customer.services;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.mkyong.customer.dao.CustomerDAO;

@Service
public class CustomerService
{
    @Autowired
    CustomerDAO customerDAO;

    @Override
    public String toString() {
        return "CustomerService [customerDAO=" + customerDAO + "]";
    }
}

```

Spring Autowiring By Name

Eclipse Project name: 13_springAutoWiring

[Expliquer les autres types d'autowiring après l'exo]

(le code source est dans le dossier :

Presentation cours\Code cours\Spring-AutoWiring-by-Name-Example)

In Spring, “Autowiring by Name” means, if the name of a bean is same as the name of other bean property, auto wire it.

For example, if a “customer” bean exposes an “address” property, Spring will find the “address” bean in current container and wire it automatically. And if no matching found, just do nothing.

You can enable this feature via `autowire="byName"` like below :

```
<!-- customer has a property name "address" -->
<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />

<bean id="address" class="com.mkyong.common.Address" >
    <property name="fulladdress" value="Block A 888, CA" />
</bean>
```

See a full example of Spring auto wiring by name.

1. Beans

Two beans, customer and address.

```
package com.mkyong.common;

public class Customer
{
    private Address address;
    //...
}

package com.mkyong.common;

public class Address
{
    private String fulladdress;
    //...
}
```

2. Spring Wiring

Normally, you wire the bean explicitly, via ref attribute like this :

```
<bean id="customer" class="com.mkyong.common.Customer" >
    <property name="address" ref="address" />
</bean>

<bean id="address" class="com.mkyong.common.Address" >
    <property name="fulladdress" value="Block A 888, CA" />
</bean>
```

Output

```
Customer [address=Address [fulladdress=Block A 888, CA]]
```

With autowire by name enabled, you do not need to declares the property tag anymore. As long as the “address” bean is same name as the property of “customer” bean, which is “address”, Spring will wire it automatically.

```
<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />

<bean id="address" class="com.mkyong.common.Address" >
    <property name="fulladdress" value="Block A 888, CA" />
</bean>
```

Output

```
Customer [address=Address [fulladdress=Block A 888, CA]]
```

See another example, this time, the wiring will failed, caused the bean “addressABC” is not match the property name of bean “customer”.

```
<bean id="customer" class="com.mkyong.common.Customer" autowire="byName" />

<bean id="addressABC" class="com.mkyong.common.Address" >
    <property name="fulladdress" value="Block A 888, CA" />
</bean>
```

Output

```
Customer [address=null]
```

Spring Auto-Wiring Beans With @Autowired Annotation

Eclipse Project name: 14_springAutoWiringAnnotation

(le code source est dans le dossier :
Presentation cours\Code cours\Spring-Autowired-Annotation-Example)

In last [Spring auto-wiring in XML](#) example, it will autowired the matched property of any bean in current Spring container. In most cases, you may need autowired property in a particular bean only.
In Spring, you can use **@Autowired** annotation to auto wire bean on the setter method, constructor or a field.
Moreover, it can autowired property in a particular bean.

Note

The **@Autowired** annotation is auto wire the bean by matching data type.

See following full example to demonstrate the use of **@Autowired**.

1. Beans

A customer bean, and declared in bean configuration file. Later, you will use “**@Autowired**” to auto wire a person bean.

```
package com.mkyong.common;

public class Customer
{
    //you want autowired this field.
    private Person person;

    private int type;
    private String action;

    //getter and setter method
}

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

    <bean id="CustomerBean" class="com.mkyong.common.Customer">
        <property name="action" value="buy" />
        <property name="type" value="1" />
    </bean>

    <bean id="PersonBean" class="com.mkyong.common.Person">
        <property name="name" value="mkyong" />
        <property name="address" value="address 123" />
        <property name="age" value="28" />
    </bean>
</beans>
```

2. Register AutowiredAnnotationBeanPostProcessor

To enable **@Autowired**, you have to register ‘**AutowiredAnnotationBeanPostProcessor**’, and you can do it in two ways :

1. Include `<context:annotation-config />`

Add Spring context and `<context:annotation-config />` in bean configuration file.

```
<beans>
```

```

//...
xmlns:context="http://www.springframework.org/schema/context"
//...
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">
//...

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

```

Full example,

```

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

    <context:annotation-config />

    <bean id="CustomerBean" class="com.mkyong.common.Customer">
        <property name="action" value="buy" />
        <property name="type" value="1" />
    </bean>

    <bean id="PersonBean" class="com.mkyong.common.Person">
        <property name="name" value="mkyong" />
        <property name="address" value="address ABC" />
        <property name="age" value="29" />
    </bean>

```

</beans>

2. Include AutowiredAnnotationBeanPostProcessor

Include 'AutowiredAnnotationBeanPostProcessor' directly in bean configuration file.

```

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

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

    <bean id="CustomerBean" class="com.mkyong.common.Customer">
        <property name="action" value="buy" />
        <property name="type" value="1" />
    </bean>

    <bean id="PersonBean" class="com.mkyong.common.Person">
        <property name="name" value="mkyong" />
        <property name="address" value="address ABC" />
        <property name="age" value="29" />
    </bean>

```

</beans>

3. @Autowired Examples

Now, you can autowire bean via `@Autowired`, and it can be applied on setter method, constructor or a field.

1. @Autowired setter method

```

package com.mkyong.common;

import org.springframework.beans.factory.annotation.Autowired;

```

```

public class Customer
{
    private Person person;
    private int type;
    private String action;
    //getter and setter methods

    @Autowired
    public void setPerson(Person person) {
        this.person = person;
    }
}

```

2. @Autowired constructor

```

package com.mkyong.common;

import org.springframework.beans.factory.annotation.Autowired;

public class Customer
{
    private Person person;
    private int type;
    private String action;
    //getter and setter methods

    @Autowired
    public Customer(Person person) {
        this.person = person;
    }
}

```

3. @Autowired field

```

package com.mkyong.common;

import org.springframework.beans.factory.annotation.Autowired;

public class Customer
{
    @Autowired
    private Person person;
    private int type;
    private String action;
    //getter and setter methods
}

```

The above example will autowire ‘PersonBean’ into Customer’s person property.
Run it

```

package com.mkyong.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

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

        Customer cust = (Customer)context.getBean("CustomerBean");
        System.out.println(cust);
    }
}

```

```
}
```

Output

```
Customer [action=buy, type=1,  
person=Person [address=address 123, age=28, name=mkyong]]
```

Dependency checking

By default, the `@Autowired` will perform the dependency checking to make sure the property has been wired properly. When Spring can't find a matching bean to wire, it will throw an exception. To fix it, you can disable this checking feature by setting the “`required`” attribute of `@Autowired` to false.

```
package com.mkyong.common;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class Customer  
{  
    @Autowired(required=false)  
    private Person person;  
    private int type;  
    private String action;  
    //getter and setter methods  
}
```

In the above example, if the Spring can't find a matching bean, it will leave the `person` property unset.

@Qualifier

The `@Qualifier` annotation us used to control which bean should be autowire on a field. For example, bean configuration file with two similar person beans.

```
<beans xmlns="http://www.springframework.org/schema/beans"  
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
       xmlns:context="http://www.springframework.org/schema/context"  
       xsi:schemaLocation="http://www.springframework.org/schema/beans  
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd  
                           http://www.springframework.org/schema/context  
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">  
  
    <context:annotation-config />  
  
    <bean id="CustomerBean" class="com.mkyong.common.Customer">  
        <property name="action" value="buy" />  
        <property name="type" value="1" />  
    </bean>  
  
    <bean id="PersonBean1" class="com.mkyong.common.Person">  
        <property name="name" value="mkyong1" />  
        <property name="address" value="address 1" />  
        <property name="age" value="28" />  
    </bean>  
  
    <bean id="PersonBean2" class="com.mkyong.common.Person">  
        <property name="name" value="mkyong2" />  
        <property name="address" value="address 2" />  
        <property name="age" value="28" />  
    </bean>  
  
</beans>
```

Will Spring know which bean should wire?

To fix it, you can use `@Qualifier` to auto wire a particular bean, for example,

```
package com.mkyong.common;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Qualifier;
```

```
public class Customer
{
    @Autowired
    @Qualifier("PersonBean1")
    private Person person;
    private int type;
    private String action;
    //getter and setter methods
}
```

It means, bean “PersonBean1” is autowired into the Customer’s person property. Read this full example – [Spring Autowiring @Qualifier example](#)

Conclusion

This **@Autowired** annotation is highly flexible and powerful, and definitely better than “**autowire**” attribute in bean configuration file.

Programmation AOP

Eclipse Project name: *15_DemoAP_Calculator*

(le code source est sur eclipse : 03_DemoAOP_calculator

Dossier : E:\Eclipse\workspace

Exercice du livre : Spring par l'exemple – Gary Mark – page: 141)

Spring Object/XML Mapping Example (spring 3.0)

Eclipse Project name: 15_springObjectXMLMapper

*(le code source est dans le dossier :
Presentation cours\Code cours\ Spring3-Object-XML-Mapping-Example)*

The Spring's Object/XML Mapping, is converting Object to XML or vice versa. This process is also known as

1. XML Marshalling – Convert Object to XML.
2. XML UnMarshalling – Convert XML to Object.

In this tutorial, we show you how to use Spring's oxm to do the conversion, **Object <--- Spring oxm ---> XML**.

Note

No nonsense, for why and what benefits of using Spring's oxm, read this [official Spring Object/XML mapping article](#).

1. Project Dependency

Dependencies in this example.

Note

Spring's oxm itself doesn't handle the XML marshalling or UnMarshalling, it depends developer to inject their prefer XML binding framework. In this case, you will use Castor binding framework.

```
<properties>
    <spring.version>3.0.5.RELEASE</spring.version>
</properties>

<dependencies>

    <!-- Spring 3 dependencies -->
    <dependency>
        <groupId>org.springframework</groupId>
        <artifactId>spring-core</artifactId>
        <version>${spring.version}</version>
    </dependency>

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

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

    <!-- Uses Castor for XML -->
    <dependency>
        <groupId>org.codehaus.castor</groupId>
        <artifactId>castor</artifactId>
    </dependency>

```

```

</dependency>

<!-- Castor need this -->
<dependency>
    <groupId>xerces</groupId>
    <artifactId>xercesImpl</artifactId>
    <version>2.8.1</version>
</dependency>

</dependencies>

```

2. Simple Object

A simple object, later convert it into XML file.

```

package com.mkyong.core.model;

public class Customer {

    String name;
    int age;
    boolean flag;
    String address;

    //standard getter, setter and toString() methods.
}

```

3. Marshaller and Unmarshaller

This class will handle the conversion via Spring's oxm interfaces : [Marshaller](#) and [Unmarshaller](#).

```

package com.mkyong.core;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
import org.springframework.oxm.Marshaller;
import org.springframework.oxm.Unmarshaller;

public class XMLConverter {

    private Marshaller marshaller;
    private Unmarshaller unmarshaller;

    public Marshaller getMarshaller() {
        return marshaller;
    }

    public void setMarshaller(Marshaller marshaller) {
        this.marshaller = marshaller;
    }

    public Unmarshaller getUnmarshaller() {
        return unmarshaller;
    }

    public void setUnmarshaller(Unmarshaller unmarshaller) {
        this.unmarshaller = unmarshaller;
    }
}

```

```

}

public void convertFromObjectToXML(Object object, String filepath)
throws IOException {

    FileOutputStream os = null;
    try {
        os = new FileOutputStream(filepath);
        getMarshaller().marshal(object, new StreamResult(os));
    } finally {
        if (os != null) {
            os.close();
        }
    }
}

public Object convertFromXMLToObject(String xmlfile) throws IOException {

    FileInputStream is = null;
    try {
        is = new FileInputStream(xmlfile);
        return getUnmarshaller().unmarshal(new StreamSource(is));
    } finally {
        if (is != null) {
            is.close();
        }
    }
}
}

```

4. Spring Configuration

In Spring's bean configuration file, inject `CastorMarshaller` as the XML binding framework.

```

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

    <bean id="XMLConverter" class="com.mkyong.core.XMLConverter">
        <property name="marshaller" ref="castorMarshaller" />
        <property name="unmarshaller" ref="castorMarshaller" />
    </bean>
    <bean id="castorMarshaller"
          class="org.springframework.oxm.castor.CastorMarshaller" />
</beans>

```

5. Test

Run it.

```

package com.mkyong.core;

import java.io.IOException;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

```

```

import com.mkyong.core.model.Customer;

public class App {
    private static final String XML_FILE_NAME = "customer.xml";

    public static void main(String[] args) throws IOException {
        ApplicationContext appContext = new
ClassPathXmlApplicationContext("App.xml");
        XMLConverter converter = (XMLConverter)
appContext.getBean("XMLConverter");

        Customer customer = new Customer();
        customer.setName("mkyong");
        customer.setAge(30);
        customer.setFlag(true);
        customer.setAddress("This is address");

        System.out.println("Convert Object to XML!");
        //from object to XML file
        converter.convertFromObjectToXML(customer, XML_FILE_NAME);
        System.out.println("Done \n");

        System.out.println("Convert XML back to Object!");
        //from XML to object
        Customer customer2 =
(Customer) converter.convertFromXMLToObject(XML_FILE_NAME);
        System.out.println(customer2);
        System.out.println("Done");

    }
}

```

Output

```

Convert Object to XML!
Done

Convert XML back to Object!
Customer [name=mkyong, age=30, flag=true, address=This is address]
Done

```

The following XML file “customer.xml” is generated in your project root folder.

File : customer.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<customer flag="true" age="30">
    <address>This is address</address>
    <name>mkyong</name>
</customer>

```

Castor XML Mapping

Wait, why flag and age are converted as attribute? Is that a way to control which field should use as attribute or element? Of course, you can use [Castor XML mapping](#) to define the relationship between Object and XML. Create following mapping file, and put it into your project classpath.

File : mapping.xml

```

<mapping>
    <class name="com.mkyong.core.model.Customer">

```

```

<map-to xml="customer" />

<field name="age" type="integer">
    <bind-xml name="age" node="attribute" />
</field>

<field name="flag" type="boolean">
    <bind-xml name="flag" node="element" />
</field>

<field name="name" type="string">
    <bind-xml name="name" node="element" />
</field>

<field name="address" type="string">
    <bind-xml name="address" node="element" />
</field>
</class>
</mapping>

```

In Spring bean configuration file, inject above **mapping.xml** into CastorMarshaller via “**mappingLocation**”.

```

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

    <bean id="XMLConverter" class="com.mkyong.core.XMLConverter">
        <property name="marshaller" ref="castorMarshaller" />
        <property name="unmarshaller" ref="castorMarshaller" />
    </bean>
    <bean id="castorMarshaller"
          class="org.springframework.oxm.castor.CastorMarshaller" >
        <property name="mappingLocation" value="classpath:mapping.xml" />
    </bean>
</beans>

```

Test it again, the XML file “**customer.xml**” will be updated.

File : customer.xml

```

<?xml version="1.0" encoding="UTF-8"?>
<customer age="30">
    <flag>true</flag>
    <name>mkyong</name>
    <address>This is address</address>
</customer>

```

Spring + JDBC Example

Eclipse Project name: 15_SpringJDBCTemplateDAO

(*le code source est dans le dossier :
Presentation cours\Code cours\ Spring-JDBC-Example*)

In this tutorial, we will extend last [Maven + Spring hello world example](#) by adding JDBC support, to use Spring + JDBC to insert a record into a customer table.

1. Customer table

In this example, we are using MySQL database.

```
CREATE TABLE `customer` (
  `CUST_ID` INT(10) UNSIGNED NOT NULL AUTO_INCREMENT,
  `NAME` VARCHAR(100) NOT NULL,
  `AGE` INT(10) UNSIGNED NOT NULL,
  PRIMARY KEY (`CUST_ID`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8;
```

2. Project Dependency

Add Spring and MySQL dependencies in Maven `pom.xml` file.

File : pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mkyong.common</groupId>
  <artifactId>SpringExample</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>SpringExample</name>
  <url>http://maven.apache.org</url>

  <dependencies>

    <!-- Spring framework -->
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.5.6</version>
    </dependency>

    <!-- MySQL database driver -->
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>5.1.9</version>
    </dependency>

  </dependencies>
</project>
```

3. Customer model

Add a customer model to store customer's data.

```
package com.mkyong.customer.model;
```

```

import java.sql.Timestamp;

public class Customer
{
    int custId;
    String name;
    int age;
    //getter and setter methods

}

```

4. Data Access Object (DAO) pattern

Customer Dao interface.

```

package com.mkyong.customer.dao;

import com.mkyong.customer.model.Customer;

public interface CustomerDAO
{
    public void insert(Customer customer);
    public Customer findByCustomerId(int custId);
}

```

Customer Dao implementation, use JDBC to issue a simple insert and select statement.

```

package com.mkyong.customer.dao.impl;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;
import com.mkyong.customer.dao.CustomerDAO;
import com.mkyong.customer.model.Customer;

public class JdbcCustomerDAO implements CustomerDAO
{
    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void insert(Customer customer) {

        String sql = "INSERT INTO CUSTOMER " +
                    "(CUST_ID, NAME, AGE) VALUES (?, ?, ?)";
        Connection conn = null;

        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setInt(1, customer.getCustId());
            ps.setString(2, customer.getName());
            ps.setInt(3, customer.getAge());
            ps.executeUpdate();
            ps.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {

```

```

        } catch (SQLException e) {}
    }

}

public Customer findCustomerId(int custId) {
    String sql = "SELECT * FROM CUSTOMER WHERE CUST_ID = ?";
    Connection conn = null;

    try {
        conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setInt(1, custId);
        Customer customer = null;
        ResultSet rs = ps.executeQuery();
        if (rs.next()) {
            customer = new Customer(
                rs.getInt("CUST_ID"),
                rs.getString("NAME"),
                rs.getInt("Age")
            );
        }
        rs.close();
        ps.close();
        return customer;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {}
        }
    }
}
}

```

5. Spring bean configuration

Create the Spring bean configuration file for customerDAO and datasource.

File : Spring-Customer.xml

```

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

    <bean id="customerDAO" class="com.mkyong.customer.dao.impl.JdbcCustomerDAO">
        <property name="dataSource" ref="dataSource" />
    </bean>

</beans>

```

File : Spring-Datasource.xml

Configurer une source de données dans Spring

L'interface standard javax.sql.DataSource est définie par les spécifications de JDBC.

Il existe de nombreuses implémentations d'une source de données, proposées par différents fournisseurs et projets.

Il est très facile de passer d'une implémentation à l'autre, car elles implémentent toute l'interface commune `DataSource`.

En tant que framework applicatif Java, Spring fournit également plusieurs implémentations pratiques, quoique moins puissantes, d'une source de données.

La plus simple, nommée **`DriverManager-DataSource`**, ouvre une nouvelle connexion à chaque demande.

`DriverManagerDataSource` n'est pas une implémentation très efficace car une nouvelle connexion est ouverte pour le client à chaque requête.

Spring propose une autre implémentation d'une source de données, **`SingleConnectionDataSource`**.

Comme son nom l'indique, elle maintient une seule connexion réutilisée à chaque requête et jamais fermée. Elle n'est donc pas adaptée à un environnement multithread.

Les implémentations fournies par Spring sont principalement destinées aux tests.

En revanche, de nombreuses implémentations professionnelles d'une source de données prennent en charge la mutualisation des connexions.

Par exemple, le module **`DBCP (Database Connection Pooling Services)`** de la bibliothèque Apache Commons dispose de plusieurs implémentations d'une source de données qui prennent en charge les pools de connexions.

En particulier, **`BasicDataSource`** accepte les mêmes propriétés de connexion que **`DriverManagerDataSource`**, mais nous permet de préciser le nombre initial de connexions du pool et le nombre maximal de connexions actives.

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

    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">

        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/mkyongjava" />
        <property name="username" value="root" />
        <property name="password" value="password" />
    </bean>
</beans>
```

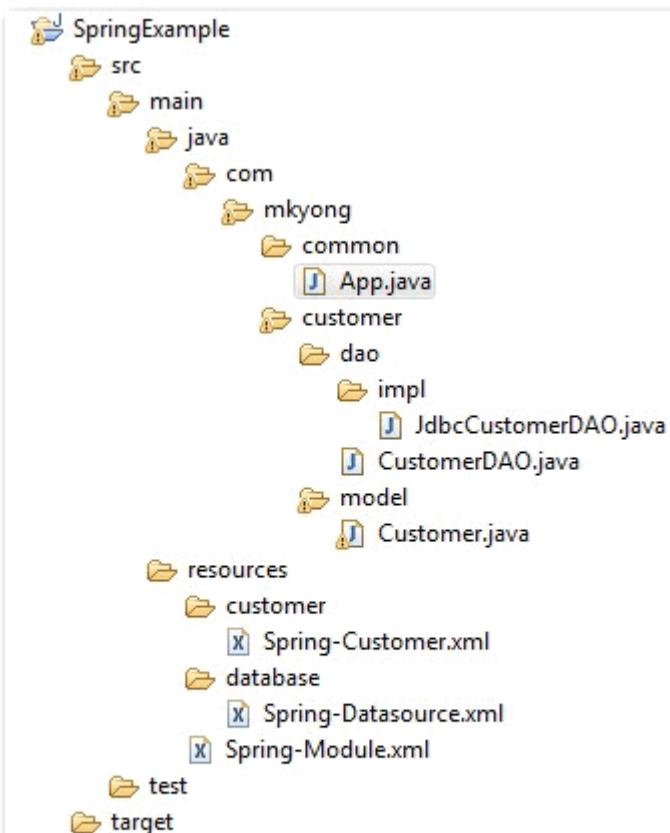
File : `Spring-Module.xml`

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

    <import resource="database/Spring-Datasource.xml" />
    <import resource="customer/Spring-Customer.xml" />
</beans>
```

6. Review project structure

Full directory structure of this example.



7. Run it

```
package com.mkyong.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.mkyong.customer.dao.CustomerDAO;
import com.mkyong.customer.model.Customer;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("Spring-Module.xml");

        CustomerDAO customerDAO = (CustomerDAO) context.getBean("customerDAO");
        Customer customer = new Customer(1, "mkyong", 28);
        customerDAO.insert(customer);

        Customer customer1 = customerDAO.findByCustomerId(1);
        System.out.println(customer1);
    }
}

output
Customer [age=28, custId=1, name=mkyong]
```

Spring + JdbcTemplate + JdbcDaoSupport Examples

⇒ Continuer sur le même projet

Eclipse Project name: 15_SpringJDBCTemplateDAO

(le code source est dans le dossier :
Presentation cours\Code cours\Spring-JDBC-Example)

In Spring JDBC development, you can use `JdbcTemplate` and `JdbcDaoSupport` classes to simplify the overall database operation processes.

In this tutorial, we will reuse the last [Spring + JDBC example](#), to see the different between a before (No `JdbcTemplate` support) and after (With `JdbcTemplate` support) example.

1. Example Without `JdbcTemplate`

Without `JdbcTemplate`, you have to create many redundant codes (create connection , close connection , handle exception) in all the DAO database operation methods – insert, update and delete. It just not efficient, ugly, error prone and tedious.

```
private DataSource dataSource;

public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}

public void insert(Customer customer) {

    String sql = "INSERT INTO CUSTOMER " +
                 "(CUST_ID, NAME, AGE) VALUES (?, ?, ?)";
    Connection conn = null;

    try {
        conn = dataSource.getConnection();
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setInt(1, customer.getCustId());
        ps.setString(2, customer.getName());
        ps.setInt(3, customer.getAge());
        ps.executeUpdate();
        ps.close();
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {}
        }
    }
}
```

2. Example With JdbcTemplate

With JdbcTemplate, you save a lot of typing on the redundant codes, because JdbcTemplate will handle it automatically.

Explication de jdbcTemplate et sa méthode update(): (livre page 221 : spring par l'exemple)

La classe **JdbcTemplate** déclare plusieurs méthodes **update()** surchargées qui contrôlent le déroulement global d'une mise à jour. Les différentes versions de cette méthode nous permettent de remplacer des sous-tâches du processus par défaut. Le framework Spring JDBC définit plusieurs interfaces de rappel pour l'encapsulation des différents sous-ensembles de tâches. Nous pouvons implémenter l'une de ces interfaces de rappel et passer son instance à la méthode update() correspondante afin d'intervenir sur le processus.

dans sa variante la plus simple, la méthode update() accepte une requête SQL et un tableau d'objets définissant ses paramètres. Elle se charge de créer un objet PreparedStatement à partir de la requête SQL et de lier les paramètres. Par conséquent, il devient inutile de redéfinir les tâches de la mise à jour.

On ajoute linstanciation de jdbcTemplate et on appelle update() dans insert :
(ajouter le jar transactions)

```
private DataSource dataSource;
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}

public void insert(Customer customer) {

    String sql = "INSERT INTO CUSTOMER " +
        "(CUST_ID, NAME, AGE) VALUES (?, ?, ?)";

    jdbcTemplate = new JdbcTemplate(dataSource);

    jdbcTemplate.update(sql, new Object[] { customer.getCustId(),
        customer.getName(), customer.getAge()
    });
}
```

See the different?

On passe à `findById()`:

Quand on veut extraire des données par correspondance de ligne, on a l'interface **RowMapper**.

(il y a aussi une autre interface nommée **RowCallbackHandler**, **RowMapper** est plus générale).

Le rôle de **RowMapper** est de créer une correspondance entre une seule ligne de l'ensemble de résultat et un objet personnalisé.

Elle peut donc être appliquée à un ensemble de résultat contenant une ou plusieurs lignes. Pour faciliter la réutilisation, il est préférable d'implémenter l'interface RowMapper sous forme d'une classe normale plutôt que sous forme d'une classe interne. Dans la méthode `mapRow()` de cette interface, nous construisons l'objet qui représente une ligne et l'utilisons comme valeur de retour.

On va ajouter une classe dans le package model qu'on va appeler : **ClientRowMapper**. Cette classe va implémenter **RowMapper** avec la méthode `mapRow()`.

```
package fr.adaming.model;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class CustomerRowMapper implements RowMapper
{
    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Customer customer = new Customer();
        customer.setCustomerId(rs.getInt("CUST_ID"));
        customer.setName(rs.getString("NAME"));
        customer.setAge(rs.getInt("AGE"));
        return customer;
    }
}
```

Maintenant on revient sur `findById()`:

Nous l'avons mentionné, **RowMapper** peut être utilisée avec un ensemble de résultat contenant une ou plusieurs lignes. Lorsque l'interrogation de la base de données retourne un seul objet, comme c'est le cas pour `findById()`, nous devons invoquer la méthode `queryForObject()` de JdbcTemplate.

```
private DataSource dataSource;
private JdbcTemplate jdbcTemplate;

public void setDataSource(DataSource dataSource) {
    this.dataSource = dataSource;
}

//query single row with RowMapper
public Customer findById(int custId) {

    String sql = "SELECT * FROM CUSTOMER WHERE CUST_ID = ?";
    jdbcTemplate = new JdbcTemplate(dataSource);

    Customer customer = (Customer)jdbcTemplate().queryForObject(
        sql, new Object[] { custId }, new CustomerRowMapper());

    return customer;
}
```

```
}
```

3. Example With JdbcDaoSupport

Simplifier la création d'un template JDBC

Problème

La création d'une nouvelle instance de JdbcTemplate à chaque utilisation est peu efficace, car nous devons répéter l'instruction de création et payer le prix de création d'un nouvel objet.

Solution

La classe JdbcTemplate est conçue pour être sûre vis-à-vis des threads. Nous pouvons donc en déclarer une seule instance dans le conteneur IoC et l'injecter dans toutes nos instances d'objets d'accès aux données. Par ailleurs, Spring JDBC offre une classe pratique, **JdbcDaoSupport**, qui simplifie l'implémentation d'un DAO. Elle déclare une propriété jdbcTemplate qui peut être injectée depuis le conteneur IoC ou créée automatiquement à partir d'une source de données. Notre DAO peut étendre cette classe pour hériter de cette propriété.

By extended the JdbcDaoSupport, set the datasource and JdbcTemplate in your class is no longer required, you just need to inject the correct datasource into JdbcCustomerDAO. And you can get the JdbcTemplate by using a getJdbcTemplate() method.

On a plus besoin de declarer le dataSource

```
public class JdbcCustomerDAO extends JdbcDaoSupport implements CustomerDAO
{
    //no need to set datasource here
    public void insert(Customer customer) {

        String sql = "INSERT INTO CUSTOMER " +
                     "(CUST_ID, NAME, AGE) VALUES (?, ?, ?)";

        getJdbcTemplate().update(sql, new Object[] { customer.getCustId(),
                                                    customer.getName(), customer.getAge()
                                                });
    }

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

    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">

        <property name="driverClassName" value="com.mysql.jdbc.Driver" />
        <property name="url" value="jdbc:mysql://localhost:3306/mkyongjava" />
        <property name="username" value="root" />
        <property name="password" value="password" />
    </bean>

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

```

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

<bean id="customerDAO" class="com.mkyong.customer.dao.impl.JdbcCustomerDAO">
    <property name="dataSource" ref="dataSource" />
</bean>

</beans>

```

Note

In Spring JDBC development, it's always recommended to use `JdbcTemplate` and `JdbcDaoSupport`, instead of coding JDBC code yourself.

Hibernate Mysql

Eclipse Project name: `16_hibernateMysql`

(*le code source est dans le dossier :*
Presentation cours\Code cours\Spring-JDBC-Example)

1- Introduire le framework Hibernate.(voir le cahier cours java JEE et le tuto hibernate_tutorial - tutorialspoint).

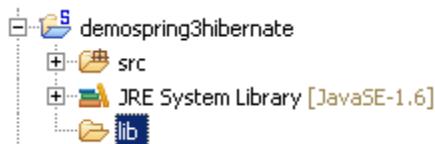
2- Projet :

a- Mettre en place la DB -> dbClient
 ->table : clients : id-nom-prenom-salaire

b- Créer le projet sur eclipse :

➔ Ajouter un dossier 'lib' : les jars de spring + hibernate + ...

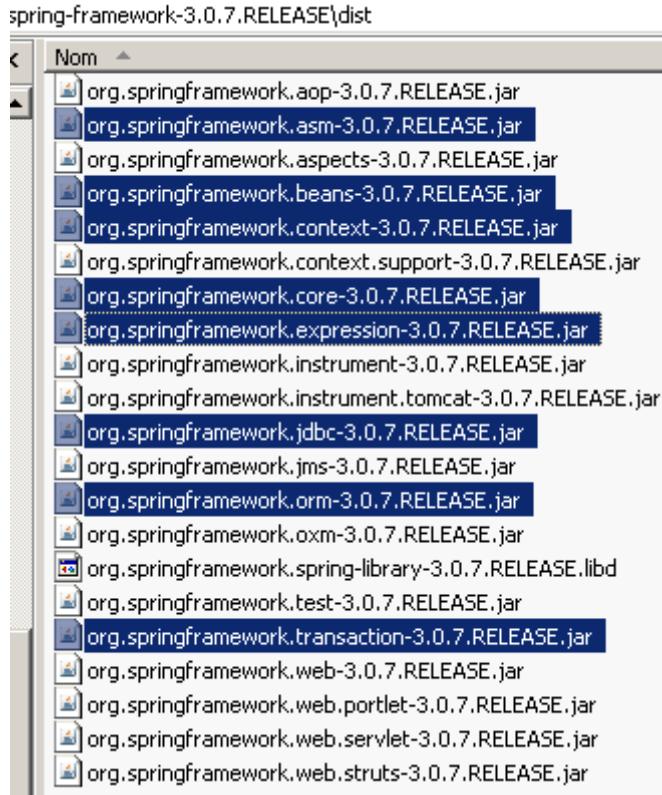
► Créez un répertoire 'lib'



tutoriel-integration-spring-3-hibernate-8

Vous allez par la suite ajouter dans ce répertoire les jars Spring et Hibernate nécessaire pour la première architecture d'intégration Spring / Hibernate (basée sur `HibernateTemplate`) :

Librairies Spring



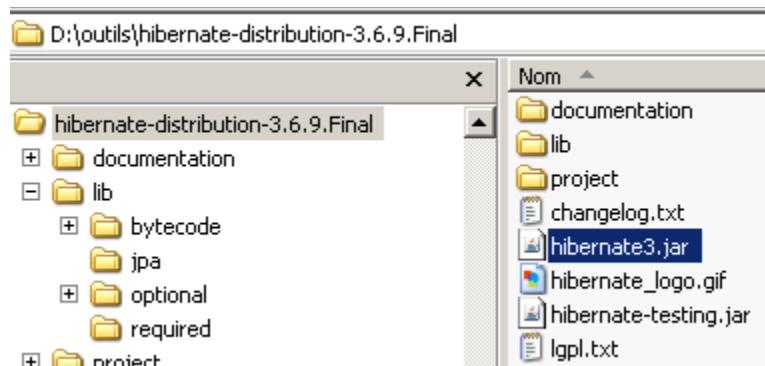
tutoriel-integration-spring-3-hibernate-4

- ▶ Notez , en plus des librairies utilisées lors du TP Spring JDBC, l'utilisation de la librairie ORM (ici org.springframework.orm-3.0.7.RELEASE.jar) qui contient les classes d'intégration de Spring avec frameworks de mapping du marché.

Librairies Hibernate

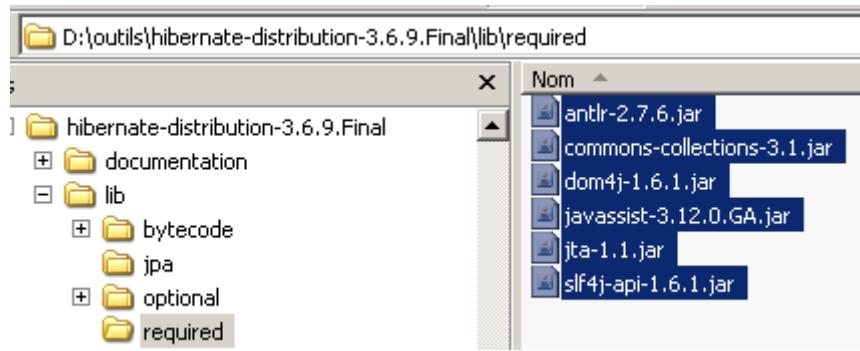
- ▶ Avec Hibernate 3.6 :

— > Le jar hibernate



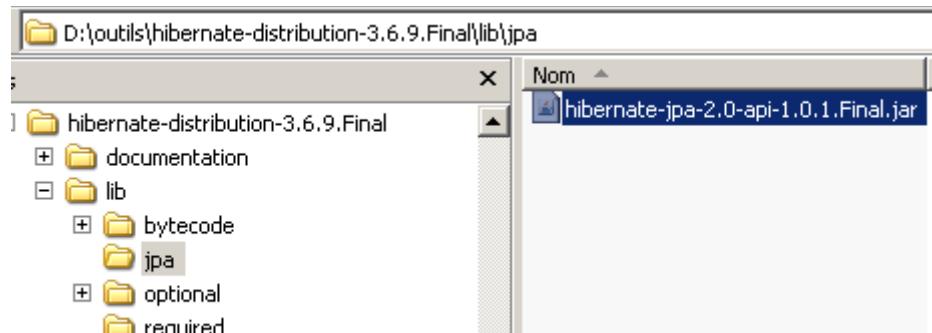
tutoriel-integration-spring-3-hibernate-5

— > Le dépendances 'obligatoires' (répertoire 'required')



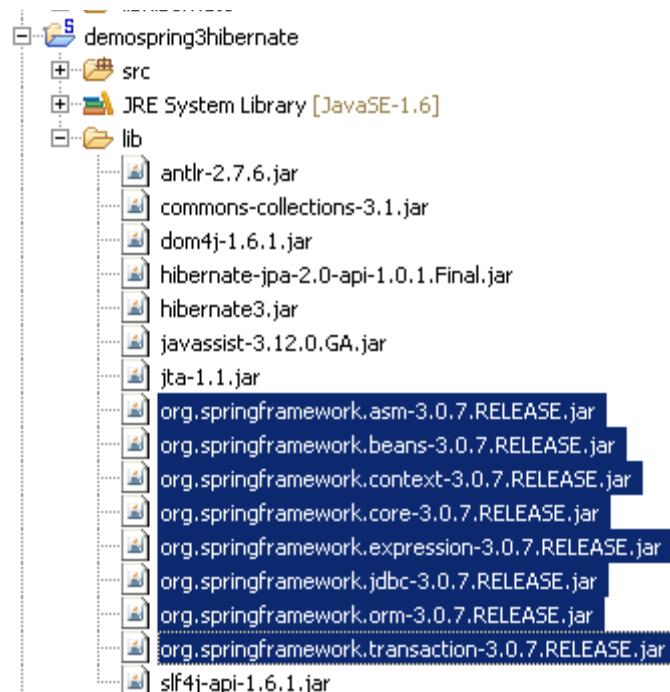
tutoriel-integration-spring-3-hibernate-6

- > La dépendance 'jpa' (répertoire 'jpa')



tutoriel-integration-spring-3-hibernate-7

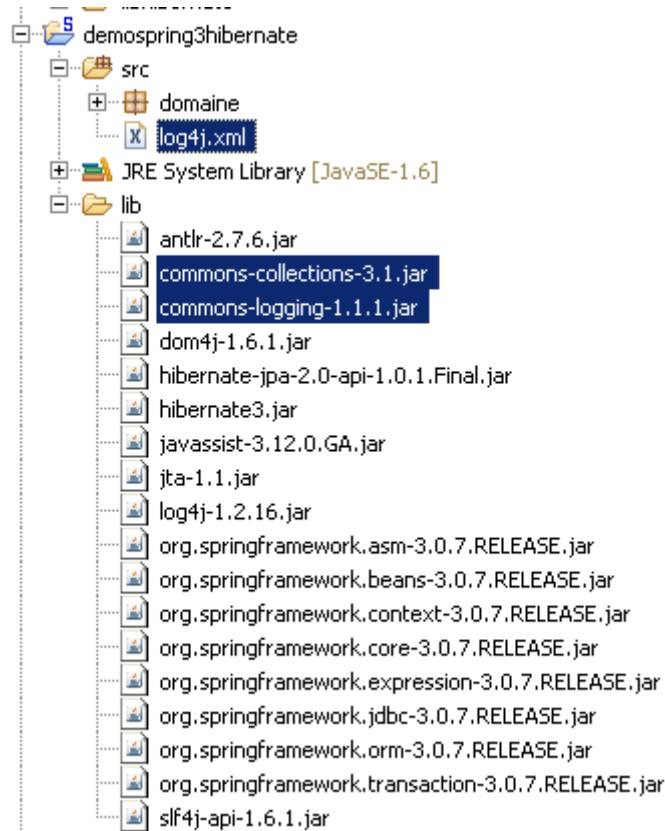
Après l'ajout des librairies Spring et Hibernate, le projet devient :



tutoriel-integration-spring-3-hibernate-9

Log4j

- > jar Commons-Logging (à partir du site téléchargement commons-logging)
- > jar Log4j
- > fichier log4j.xml



tutoriel-integration-spring-3-hibernate-10

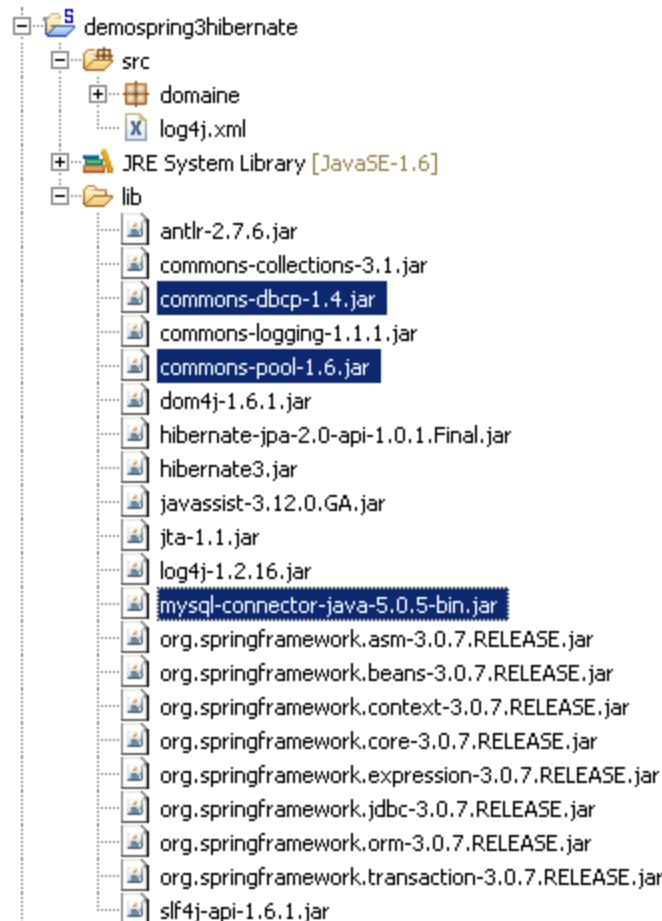
INFO : vous avez aussi besoin d'ajouter la librairie **slf4j-log4j12-1.6.4.jar** d'adaptation SL4J pour log4j (jar à récupérer dans zip téléchargé du [site produit SL4J](#))



tutoriel-integration-spring-3-hibernate-20

Pool de connexion

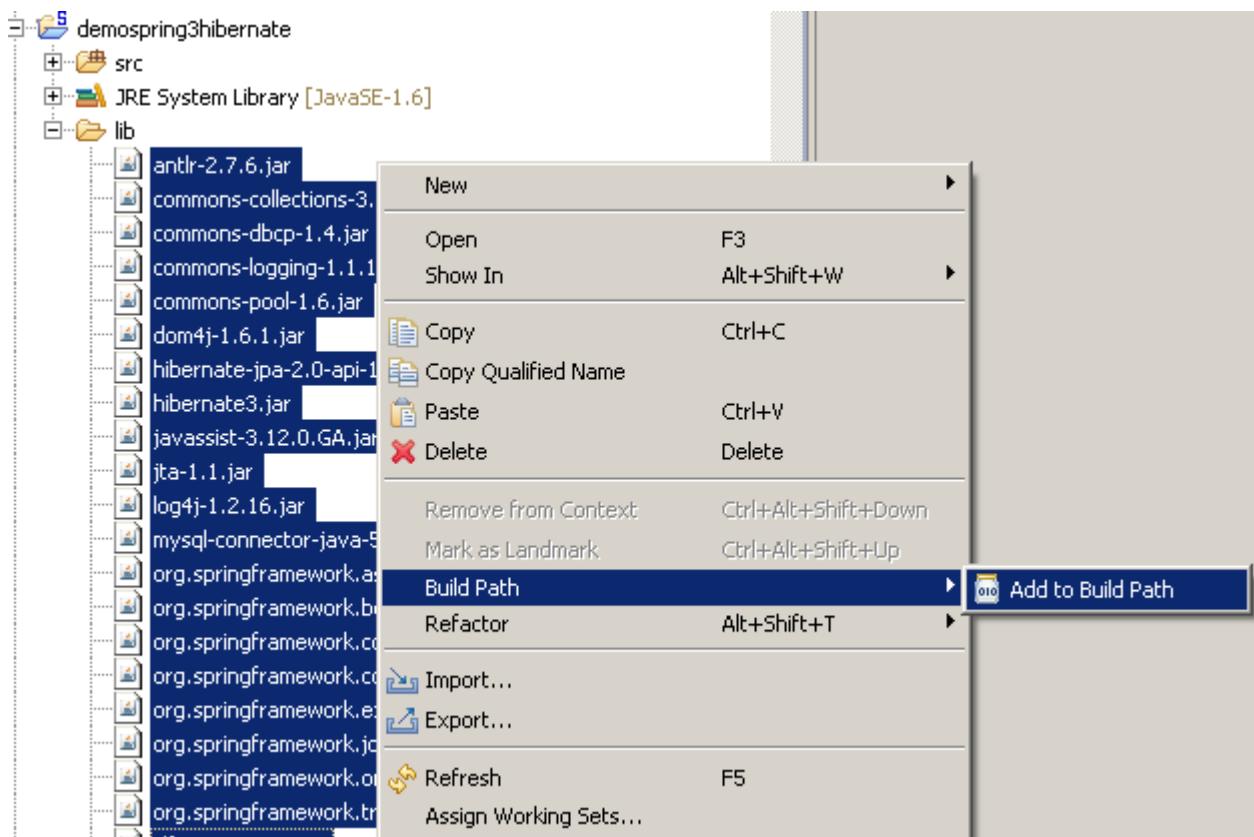
- > jar commons-dbcp (à partir du site commons-dbcp)
- > jar commons-pool (à partir du site commons-pool)
- ▶ > jar mysql



tutoriel-integration-spring-3-hibernate-11

Ajout librairies dans Classpath Eclipse

- ▶ Sélectionner toutes les librairies puis click-droit Add build Path



tutoriel-integration-spring-3-hibernate-9bjis

→ Ajouter le fichier : **log4j.properties**

Dans src->new->other->general->File->name :log4j.properties

```
1 # Direct log messages to a log file
2 log4j.appenders.file=org.apache.log4j.RollingFileAppender
3 log4j.appenders.file.File=E:/Eclipse/spring.log
4 log4j.appenders.file.MaxFileSize=1MB
5 log4j.appenders.file.MaxBackupIndex=1
6 log4j.appenders.file.layout=org.apache.log4j.PatternLayout
7 log4j.appenders.file.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
8
9 # Direct log messages to stdout
10 log4j.appenders.stdout=org.apache.log4j.ConsoleAppender
11 log4j.appenders.stdout.Target=System.out
12 log4j.appenders.stdout.layout=org.apache.log4j.PatternLayout
13 log4j.appenders.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %c{1}:%L - %m%n
14
15 # Root logger option
16 log4j.rootLogger=INFO, file, stdout
17
18 # Log everything. Good for troubleshooting
19 log4j.logger.org.hibernate=INFO
20
21 # Log all JDBC parameters
22 log4j.logger.org.hibernate.type=ALL
```

La journalisation sera sauvegardé dans le fichier :
E:/Eclipse/spring.log

- ➔ Ajout du package : fr.adaming.model
->classe : Client : id-nom-prenom-salaire (ctor sans id + getters/setters)
- ➔ Ajout du fichier du Mpping hibernate : **Client.hbm.xml**
Dans le packge : model : new->hibernate->Hibernate xml Mapping File

```
1 <?xml version="1.0"?>
2 <!DOCTYPE hibernate-mapping PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
3 "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">
4 <!-- Generated 25 janv. 2015 10:25:17 by Hibernate Tools 3.4.0.CR1 -->
5<hibernate-mapping>
6
7<class name="fr.adaming.model.Client" table="clients">
8<id name="id" type="int">
9<column name="id_client" />
10<generator class="native" />
11</id>
12<property name="nom" type="java.lang.String">
13<column name="nom" />
14</property>
15<property name="age" type="int">
16<column name="age" />
17</property>
18</class>
19
20</hibernate-mapping>
21
```

You should save the mapping document in a file with the format <classname>.hbm.xml. We saved our mapping document in the file Employee.hbm.xml. Let us see little detail about the mapping document:

- The mapping document is an XML document having <hibernate-mapping> as the root element which contains all the <class> elements.
- The <class> elements are used to define specific mappings from a Java classes to the database tables. The Java class name is specified using the **name** attribute of the class element and the database table name is specified using the **table** attribute.
- The <meta> element is optional element and can be used to create the class description.
- The <id> element maps the unique ID attribute in class to the primary key of the database table. The **name** attribute of the id element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

- The **<generator>** element within the **id** element is used to automatically generate the primary key values. Set the **class** attribute of the generator element is set to **native** to let hibernate pick up either **identity**, **sequence** or **hilo** algorithm to create primary key depending upon the capabilities of the underlying database.
- The **<property>** element is used to map a Java class property to a column in the database table. The **name** attribute of the element refers to the property in the class and the **column** attribute refers to the column in the database table. The **type** attribute holds the hibernate mapping type, this mapping types will convert from Java to SQL data type.

→ Ajout du fichier : **Hibernate.cfg.xml**

Sur : src->new->hibernate->Hibernate configuration file->...etc

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <!DOCTYPE hibernate-configuration PUBLIC
3   "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
4   "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">
5<hibernate-configuration>
6<session-factory>
7
8   <!-- Database connection settings -->
9     <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
10    <property name="connection.url">jdbc:mysql://localhost:3306/clientsspring</property>
11    <property name="connection.username">root</property>
12    <property name="connection.password"></property>
13
14    <!-- Pool de connection (interne) -->
15    <property name="connection.pool_size">1</property>
16
17    <!-- SQL dialect -->
18    <property name="dialect">org.hibernate.dialect.MySQLDialect</property>
19
20<!-- Supprimer et re-crée le schéma de base de données au démarrage
21<property name="hbm2ddl.auto">create</property>
22-->
23
24<!-- Montrer toutes les requêtes générées -->
25<property name="show_sql">true</property>
26
27  <mapping resource="fr/adaming/model/Client.hbm.xml"/>
28
29</session-factory>
30</hibernate-configuration>

```

Following is the list of important properties you would require to configure for a databases in a standalone situation:

S.N.	Properties and Description
1	hibernate.dialect This property makes Hibernate generate the appropriate SQL for the chosen database.
2	hibernate.connection.driver_class The JDBC driver class.
3	hibernate.connection.url The JDBC URL to the database instance.
4	hibernate.connection.username The database username.
5	hibernate.connection.password The database password.
6	hibernate.connection.pool_size Limits the number of connections waiting in the Hibernate database connection pool.
7	hibernate.connection.autocommit Allows autocommit mode to be used for the JDBC connection.

If you are using a database along with an application server and JNDI then you would have to configure the following properties:

S.N.	Properties and Description
1	hibernate.connection.datasource The JNDI name defined in the application server context you are using for the application.

2	hibernate.jndi.class The InitialContext class for JNDI.
3	hibernate.jndi.<JNDIpropertyname> Passes any JNDI property you like to the JNDI <i>InitialContext</i> .
4	hibernate.jndi.url Provides the URL for JNDI.
5	hibernate.connection.username The database username.
6	hibernate.connection.password The database password.

➔ Ajout du package : fr.adaming.utils

->classe : HibernateUtil.java

Cette classe va permettre de recuperer un sessionFactory

```
// settings
// from
// hibernate.cfg.xml

StandardServiceRegistryBuilder serviceRegistryBuilder = new StandardServiceRegistryBuilder();

serviceRegistryBuilder.applySettings(configuration.getProperties());

ServiceRegistry serviceRegistry = serviceRegistryBuilder.build();

return configuration.buildSessionFactory(serviceRegistry);
*/
```

```
private static final SessionFactory sessionFactory;

static {
    try {
        // Crée une SessionFactory à partir de hibernate.cfg.xml
        sessionFactory = new Configuration().configure().buildSessionFactory();
    } catch (Throwable ex) {
        // Gestion exception
        System.err.println("Echec création SessionFactory" + ex);
        throw new ExceptionInInitializerError(ex);
    }
}

public static SessionFactory getSessionFactory() {
    return sessionFactory;
}
```

➔ Ajout du package : fr.adaming.test

->classe : AppTest.java

```
public class AppTest {

    /**
     * @param args
     */
    public static void main(String[] args) {

        // 1 : Ouverture unité de travail hibernate
        Session session = HibernateUtil.getSessionFactory().openSession();

        // 2 : Ouverture transaction
        Transaction tx = session.beginTransaction();

        // 3 : Instanciation Objet métier
        Client formation = new Client("michel", 25);

        // 4 : Persistance Objet/Relationnel : création d'un enregistrement en
        // base
        Integer formationId = (Integer) session.save(formation);
        System.out.println("Clé primaire :" + formationId);

        // 5 : Fermeture transaction
        tx.commit();

        // 6 : Fermeture unité de travail hibernate
        session.close();
    }
}
```

Session Interface Methods:

There are number of methods provided by the **Session** interface but I'm going to list down few important methods only, which we will use in this tutorial. You can check Hibernate documentation for a complete list of methods associated with **Session** and **SessionFactory**.

S.N.	Session Methods and Description
1	Transaction beginTransaction() Begin a unit of work and return the associated Transaction object.
2	void cancelQuery() Cancel the execution of the current query.
3	void clear() Completely clear the session.
4	Connection close() End the session by releasing the JDBC connection and cleaning up.
5	Criteria createCriteria(Class persistentClass) Create a new Criteria instance, for the given entity class, or a superclass of an entity class.
6	Criteria createCriteria(String entityName) Create a new Criteria instance, for the given entity name.
7	Serializable getIdentifier(Object object) Return the identifier value of the given entity as associated with this session.
8	Query createFilter(Object collection, String queryString) Create a new instance of Query for the given collection and filter string.
9	Query createQuery(String queryString) Create a new instance of Query for the given HQL query string.
10	SQLQuery createSQLQuery(String queryString) Create a new instance of SQLQuery for the given SQL query string.
11	void delete(Object object) Remove a persistent instance from the datastore.
12	void delete(String entityName, Object object) Remove a persistent instance from the datastore.
13	Session get(String entityName, Serializable id) Return the persistent instance of the given named entity with the given identifier, or null if there is no such persistent instance.
14	SessionFactory getSessionFactory() Get the session factory which created this session.
15	void refresh(Object object) Re-read the state of the given instance from the underlying database.
16	Transaction getTransaction() Get the Transaction instance associated with this session.
17	boolean isConnected() Check if the session is currently connected.
18	boolean isDirty() Does this session contain any changes which must be synchronized with the database?
19	boolean isOpen() Check if the session is still open.
20	Serializable save(Object object) Persist the given transient instance, first assigning a generated identifier.
21	void saveOrUpdate(Object object) Either save(Object) or update(Object) the given instance.
22	void update(Object object) Update the persistent instance with the identifier of the given detached instance.
23	void update(String entityName, Object object)

Update the persistent instance with the identifier of the given detached instance.

S'inspirer du code suivant pour développer la classe AppTest

```
import java.util.List;
import java.util.Date;
import java.util.Iterator;

import org.hibernate.HibernateException;
import org.hibernate.Session;
import org.hibernate.Transaction;
import org.hibernate.SessionFactory;
import org.hibernate.cfg.Configuration;

public class ManageEmployee {
    private static SessionFactory factory;
    public static void main(String[] args) {
        try{
            factory = new Configuration().configure().buildSessionFactory();
        }catch (Throwable ex) {
            System.err.println("Failed to create sessionFactory object." + ex);
            throw new ExceptionInInitializerError(ex);
        }
        ManageEmployee ME = new ManageEmployee();

        /* Add few employee records in database */
        Integer empID1 = ME.addEmployee("Zara", "Ali", 1000);
        Integer empID2 = ME.addEmployee("Daisy", "Das", 5000);
        Integer empID3 = ME.addEmployee("John", "Paul", 10000);

        /* List down all the employees */
        ME.listEmployees();

        /* Update employee's records */
        ME.updateEmployee(empID1, 5000);

        /* Delete an employee from the database */
        ME.deleteEmployee(empID2);

        /* List down new list of the employees */
        ME.listEmployees();
    }

    /* Method to CREATE an employee in the database */
    public Integer addEmployee(String fname, String lname, int salary){
        Session session = factory.openSession();
        Transaction tx = null;
        Integer employeeID = null;
        try{
            tx = session.beginTransaction();
            Employee employee = new Employee(fname, lname, salary);
            employeeID = (Integer) session.save(employee);
            tx.commit();
        }catch (HibernateException e) {
            if (tx!=null) tx.rollback();
            e.printStackTrace();
        }finally {
            session.close();
        }
        return employeeID;
    }

    /* Method to READ all the employees */
    public void listEmployees(){
        Session session = factory.openSession();
        Transaction tx = null;
        try{
            tx = session.beginTransaction();
            List employees = session.createQuery("FROM Employee").list();
            for (Iterator iterator =
                  employees.iterator(); iterator.hasNext();){
                Employee employee = (Employee) iterator.next();

```

```
        System.out.print("First Name: " + employee.getFirstName());
        System.out.print(" Last Name: " + employee.getLastName());
        System.out.println(" Salary: " + employee.getSalary());
    }
    tx.commit();
}catch (HibernateException e) {
    if (tx!=null) tx.rollback();
    e.printStackTrace();
}finally {
    session.close();
}
}

/* Method to UPDATE salary for an employee */
public void updateEmployee(Integer EmployeeID, int salary ) {
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Employee employee =
            (Employee)session.get(Employee.class, EmployeeID);
        employee.setSalary( salary );
        session.update(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}

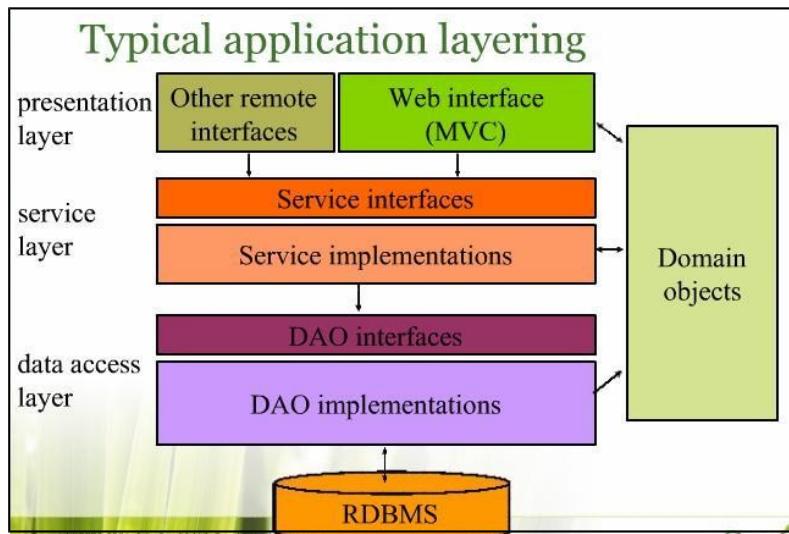
/* Method to DELETE an employee from the records */
public void deleteEmployee(Integer EmployeeID){
    Session session = factory.openSession();
    Transaction tx = null;
    try{
        tx = session.beginTransaction();
        Employee employee =
            (Employee)session.get(Employee.class, EmployeeID);
        session.delete(employee);
        tx.commit();
    }catch (HibernateException e) {
        if (tx!=null) tx.rollback();
        e.printStackTrace();
    }finally {
        session.close();
    }
}
}
```

Maven + Spring + Hibernate + MySql Example

Eclipse Project name: 17_SpringHibernateMysql

(le code source est dans le dossier :
Presentation cours\Code cours\Spring-JDBC-Example)

Explication de l'architecture Model-service(BO)-DAO à utiliser dans le projet :



- Les objets du package BO (BusinessObject) représentent les données du modèle (en l'occurrence celles de la base de données). Ces objets ne contiennent que des informations (des attributs) mais pas de 'comportements'. Ils sont accessibles dans toutes les couches de l'application.
- Le package DAO contient les classes qui ont la charge de la persistance des données (utilisation du framework Hibernate). Seules les interfaces doivent être accédées afin de permettre tout changement d'implémentation de manière transparente. En effet, le framework Spring permet de spécifier (aux classes services) la classe d'implémentation via le mécanisme d'inversion de contrôle (IoC) par injection par accesseur : utilisation d'un fichier de configuration et accès aux setter méthodes. Spring prend également en charge la gestion des sessions/transactions pour Hibernate : les classes DAO doivent étendre la classe HibernateDaoSupport qui prend en charge les accès JDBC.
- Le package Service contient les interfaces qui doivent être utilisées par toutes les autres applications (ou couches) afin de manipuler (créer, modifier, supprimer) les objets de données. Les classes d'implémentation des services doivent définir des setter méthodes pour les classes DAO afin d'être configurées par Spring (voir paragraphe précédent).

En résumé, la couche model rend visible d'une part les objets représentant les données (package BO) et, d'autre part, les interfaces Services qui permettent de manipuler ces données.

4. Model & BO & DAO

The **Model**, **Business Object** (BO) and **Data Access Object** (DAO) pattern is useful to identify the layer clearly to avoid mess up the project structure.

Stock Model

A Stock model class to store the stock data later.

```
package com.mkyong.stock.model;

import java.io.Serializable;

public class Stock implements Serializable {

    private static final long serialVersionUID = 1L;

    private Long stockId;
    private String stockCode;
    private String stockName;

    //getter and setter methods...
}
```

Stock Business Object (BO)

Stock business object (BO) interface and implementation, it's used to store the project's business function, the real database operations (CRUD) works should not involved in this class, instead it has a DAO (StockDao) class to do it.

```
package com.mkyong.stock.bo;

import com.mkyong.stock.model.Stock;

public interface StockBo {

    void save(Stock stock);
    void update(Stock stock);
    void delete(Stock stock);
    Stock findByStockCode(String stockCode);
}

package com.mkyong.stock.bo.impl;

import com.mkyong.stock.bo.StockBo;
import com.mkyong.stock.dao.StockDao;
import com.mkyong.stock.model.Stock;

public class StockBoImpl implements StockBo{

    StockDao stockDao;

    public void setStockDao(StockDao stockDao) {
        this.stockDao = stockDao;
    }

    public void save(Stock stock) {
        stockDao.save(stock);
    }

    public void update(Stock stock) {
        stockDao.update(stock);
    }

    public void delete(Stock stock) {
```

```

        stockDao.delete(stock);
    }

    public Stock findByStockCode(String stockCode) {
        return stockDao.findByStockCode(stockCode);
    }
}

```

Stock Data Access Object

A Stock DAO interface and implementation, the dao implementation class extends the Spring's "**HibernateDaoSupport**" to make Hibernate support in Spring framework. Now, you can execute the Hibernate function via **getHibernateTemplate()**.

```

package com.mkyong.stock.dao;

import com.mkyong.stock.model.Stock;

public interface StockDao {

    void save(Stock stock);
    void update(Stock stock);
    void delete(Stock stock);
    Stock findByStockCode(String stockCode);

}

package com.mkyong.stock.dao.impl;

import java.util.List;

import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

import com.mkyong.stock.dao.StockDao;
import com.mkyong.stock.model.Stock;

public class StockDaoImpl extends HibernateDaoSupport implements StockDao{

    public void save(Stock stock) {
        getHibernateTemplate().save(stock);
    }

    public void update(Stock stock) {
        getHibernateTemplate().update(stock);
    }

    public void delete(Stock stock) {
        getHibernateTemplate().delete(stock);
    }

    public Stock findByStockCode(String stockCode) {
        List list = getHibernateTemplate().find(
            "from Stock where stockCode=?",
            stockCode
        );
        return (Stock)list.get(0);
    }

}

```

5. Resource Configuration

Create a '**resources**' folder under '**project_name/main/java/**', Maven will treat all files under this folder as resources file. It will used to store the Spring, Hibernate and others configuration file.

Hibernate Configuration

Create a Hibernate mapping file (**Stock.hbm.xml**) for Stock table, put it under "**resources/hibernate/**" folder.

```
<?xml version="1.0"?>
```

```

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

<hibernate-mapping>
    <class name="com.mkyong.stock.model.Stock" table="stock" catalog="mkyong">
        <id name="stockId" type="java.lang.Long">
            <column name="STOCK_ID" />
            <generator class="identity" />
        </id>
        <property name="stockCode" type="string">
            <column name="STOCK_CODE" length="10" not-null="true" unique="true" />
        </property>
        <property name="stockName" type="string">
            <column name="STOCK_NAME" length="20" not-null="true" unique="true" />
        </property>
    </class>
</hibernate-mapping>

```

Spring Configuration

Database related....

Create a properties file (**database.properties**) for the database details, put it into the “**resources/properties**” folder. It’s good practice disparate the database details and Spring bean configuration into different files.

database.properties

```

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

```

Create a “dataSource” bean configuration file (**DataSource.xml**) for your database, and import the properties from database.properties, put it into the “**resources/database**” folder.

DataSource.xml

```

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

    <bean
        class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="location">
            <value>properties/database.properties</value>
        </property>
    </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>

</beans>

```

Hibernate related....

Create a session factory bean configuration file (**Hibernate.xml**), put it into the “**resources/database**” folder. This**LocalSessionFactoryBean** class will set up a shared Hibernate SessionFactory in a Spring application context.

Hibernate.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"

```

```

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

<!-- Hibernate session factory -->
<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

    <property name="dataSource">
        <ref bean="dataSource"/>
    </property>

    <property name="hibernateProperties">
        <props>
            <prop key="hibernate.dialect">org.hibernate.dialect.MySQLDialect</prop>
            <prop key="hibernate.show_sql">true</prop>
        </props>
    </property>

    <property name="mappingResources">
        <list>
            <value>/hibernate/Stock.hbm.xml</value>
        </list>
    </property>
</bean>
</beans>

```

Spring beans related....

Create a bean configuration file (**Stock.xml**) for BO and DAO classes, put it into the “**resources/spring**” folder. Dependency inject the dao (stockDao) bean into the bo (stockBo) bean; sessionFactory bean into the stockDao. **Stock.xml**

```

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

    <!-- Stock business object -->
    <bean id="stockBo" class="com.mkyong.stock.bo.impl.StockBoImpl" >
        <property name="stockDao" ref="stockDao" />
    </bean>

    <!-- Stock Data Access Object -->
    <bean id="stockDao" class="com.mkyong.stock.dao.impl.StockDaoImpl" >
        <property name="sessionFactory" ref="sessionFactory"></property>
    </bean>
</beans>

```

Import all the Spring's beans configuration files into a single file (BeanLocations.xml), put it into the “**resources/config**” folder.

BeanLocations.xml

```

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

    <!-- Database Configuration -->
    <import resource="../database/DataSource.xml"/>
    <import resource="../database/Hibernate.xml"/>

    <!-- Beans Declaration -->
    <import resource="../beans/Stock.xml"/>

```

```
</beans>
```

6. Run it

You have all the files and configurations , run it.

```
package com.mkyong.common;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.mkyong.stock.bo.StockBo;
import com.mkyong.stock.model.Stock;

public class App
{
    public static void main( String[] args )
    {
        ApplicationContext appContext =
            new ClassPathXmlApplicationContext("spring/config/BeanLocations.xml");

        StockBo stockBo = (StockBo)appContext.getBean("stockBo");

        /** insert */
        Stock stock = new Stock();
        stock.setStockCode("7668");
        stock.setStockName("HAIO");
        stockBo.save(stock);

        /** select */
        Stock stock2 = stockBo.findByStockCode("7668");
        System.out.println(stock2);

        /** update */
        stock2.setStockName("HAIO-1");
        stockBo.update(stock2);

        /** delete */
        stockBo.delete(stock2);

        System.out.println("Done");
    }
}
```

output

```
Hibernate: insert into mkyong.stock (STOCK_CODE, STOCK_NAME) values (?, ?)
Hibernate: select stock0_.STOCK_ID as STOCK1_0_,
stock0_.STOCK_CODE as STOCK2_0_, stock0_.STOCK_NAME as STOCK3_0_
from mkyong.stock stock0_
where stock0_.STOCK_CODE=?
Stock [stockCode=7668, stockId=11, stockName=HAIO]
Hibernate: update mkyong.stock set STOCK_CODE=?, STOCK_NAME=? where STOCK_ID=?
Hibernate: delete from mkyong.stock where STOCK_ID=?
Done
```

Problemes :

```
Exception in thread "main"
org.springframework.beans.factory.parsing.BeanDefinitionParsingException: Configuration
problem: Failed to import bean definitions from relative location [../DataSource.xml]
```

Slt:

install the "Spring IDE OSGi Extension"
->eclipse->help->check for updates ->spring IDE OSGI extension

(voir: <http://www.eclipse.org/gemini/blueprint/documentation/reference/2.0.0.M02/html/appendix-pde-integration.html>)

Probleme2 :

flushMode → add into DAO implementation- method save:
getHibernateTemplate.setCheckWriteOperations(false)

introduction Spring MVC

Eclipse Project name: 18_SpringMVC

(le code source est dans le dossier :

*Presentation cours\Code cours\
correction_tutoriel_objis_spring_introduction_springMVC*

Partie avec annotations :

Presentation cours\Code cours\ correction-tutoriel8-springg-mvc-partie10-annotations-controleur-spring25

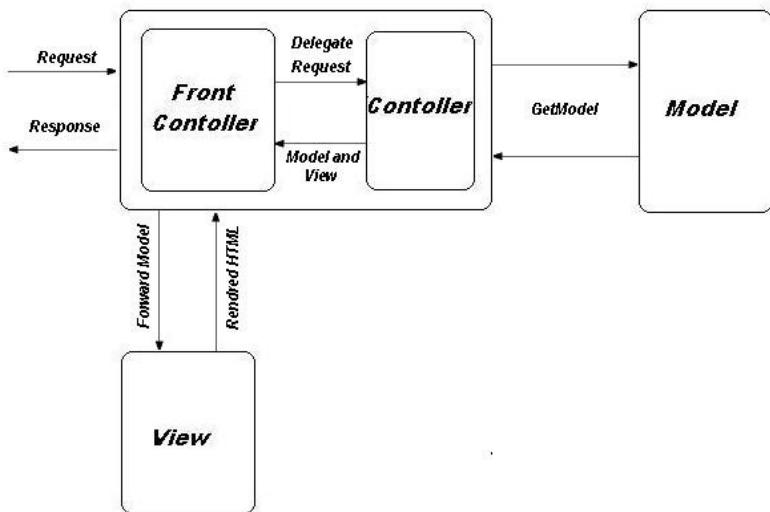
Spring MVC – Présentation

In : <http://jeefacile.unblog.fr/2009/05/13/spring-mvc-presentation/#more-61>

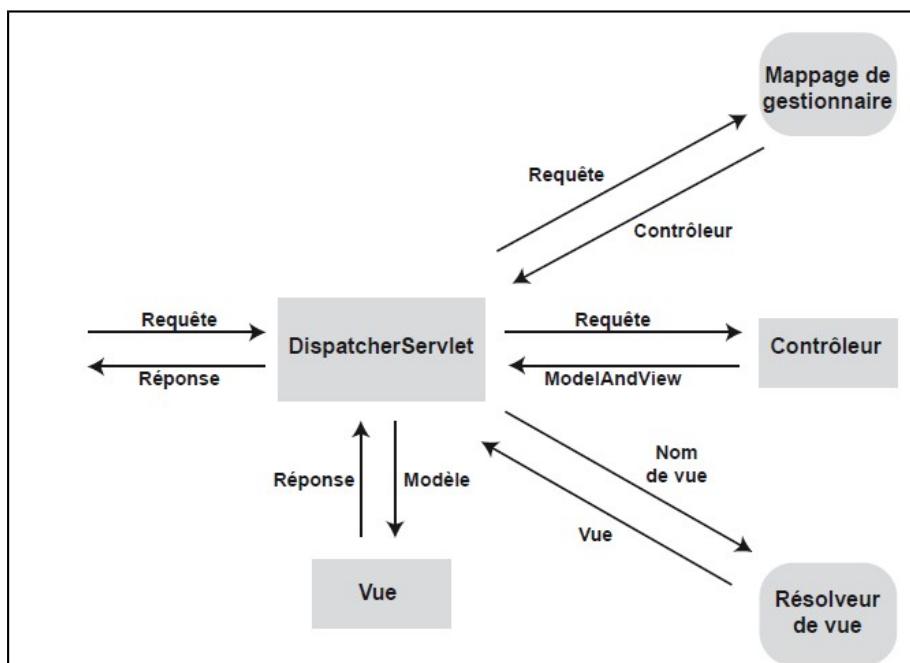
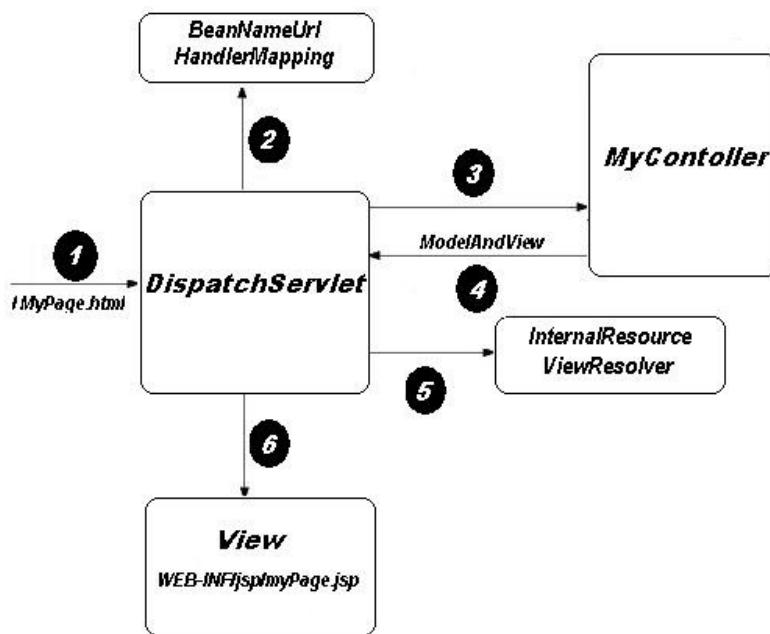
L'objectif de cet article est de présenter le principe de fonctionnement de spring MVC ainsi que les bases de ce framework.

Principe de fonctionnement

Spring MVC est un framwork qui permet d'implémenter des applications selon le design pattern MVC. Donc, comme tous autre MVC framework, Spring MVC se base sur le principe décrit par le schéma ci-dessous :



Passant maintenant aux détails : ci-dessous la cinématique de la gestion d'une requête par Spring MVC.



Les étapes sont :

1. Le DispatcherServlet reçoit une requête dont l'URI-pattern est '/myPage.html'.
2. Le DispatcherServlet consulte son Handler Mapping (Exemple : BeanNameUrlHandlerMapping) pour connaître le contrôleur dont le nom de bean est '/myPage.html'.
3. Le DispatcherServlet dispatche la requête au contrôleur identifié (Exemple : MyController)
4. Le contrôleur retourne au DispatcherServlet un objet de type ModelandView possédant comme paramètre au minimum le nom logique de la vue à renvoyer.
5. Le DispatcherServlet consulte son View Resolver lui permettant de trouver la vue dont le nom logique est 'myPage'. Ici le type de View Resolver choisi est InternalResourceViewResolver.
6. Le DispatcherServlet forward la requête à la vue associée . Ici la page /WEB-INF/jsp/myPage.jsp

A NOTER : Pour les fans de struts, la servlet DispatcherServlet de Spring est l'équivalent du contrôleur ActionServlet de Struts.

DispatcherServlet

C'est le point d'entrée de l'application, le DispatcherServlet effectue le premier mapping de l'application et distribue les requêtes aux servlets correspondantes.

Exemple : Admettant qu'on a cette configuration dans le web.xml :

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

Avec cette configuration, toutes les urls de notre application commençant par myApplication seront « dispatchées » vers cette servlet.

Controller

Le contrôleur (le C dans le modèle MVC) permet d'intercepter la requête et retourner la vue appropriée.

Spring MVC offre plusieurs types de contrôleurs : AbstractController, AbstractController, AbstractCommandController, AbstractFormController, SimpleFormController, AbstractWizardFormController...

L'interface 'Controller' définit le comportement basique d'un contrôleur : intercepter la requête et retourner un 'ModelVue' qui représente le modèle et la vue.

MultiActionController : un contrôleur dont les méthodes seront lancées en fonction des urls d'entrées.

Handler Mappings

Permet de mapper les requêtes HTTP avec le contrôleur correspondant.

View Resolvers

Permet de déterminer le nom de la vue.

Exemple : Revenons à notre exemple ci-dessous décrivant la cinématique de la gestion d'une requête par Spring MVC, pour déterminer le nom la page jsp qui correspond au url /myPage nous devons ajouter dans le fichier la configuration ci-dessous :

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

Le handler de notre contrôleur doit retourner un objet de type ModelAndView, celui-ci est déclaré de la manière suivante :

```
return new ModelAndView("myPage") ;
```

À partir du nom de vue retourné, myPage par exemple, le viewResolver trouvera la page jsp correspondante qui sera dans le dossier "/WEB-INF/jsp/" avec le suffixe « .jsp »

Conclusion

Dans cet article nous avons vu le principe de fonctionnement de spring MVC ainsi que les principaux composants. Dans l'article prochain nous allons implémenter notre première application avec spring MVC.

Partie 1 : présentation architecture SpringMVC

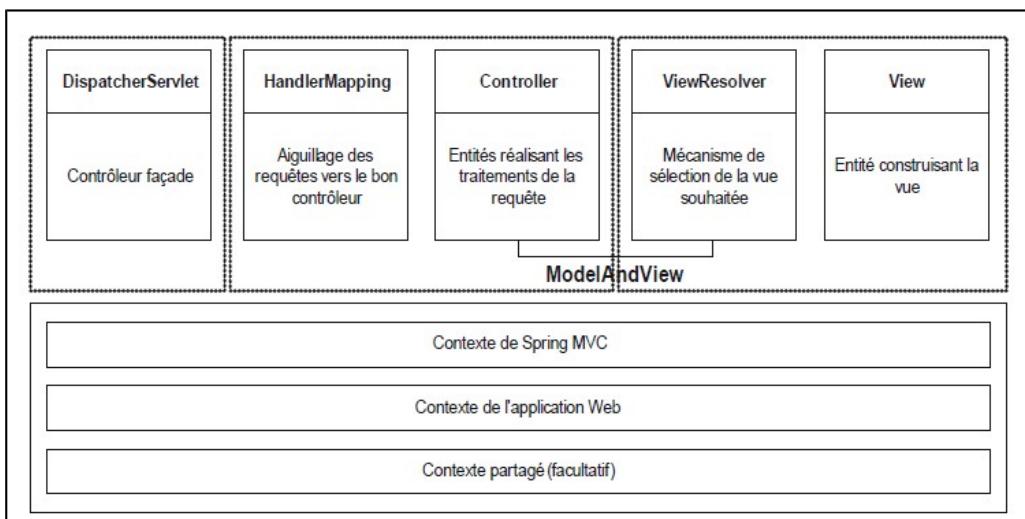


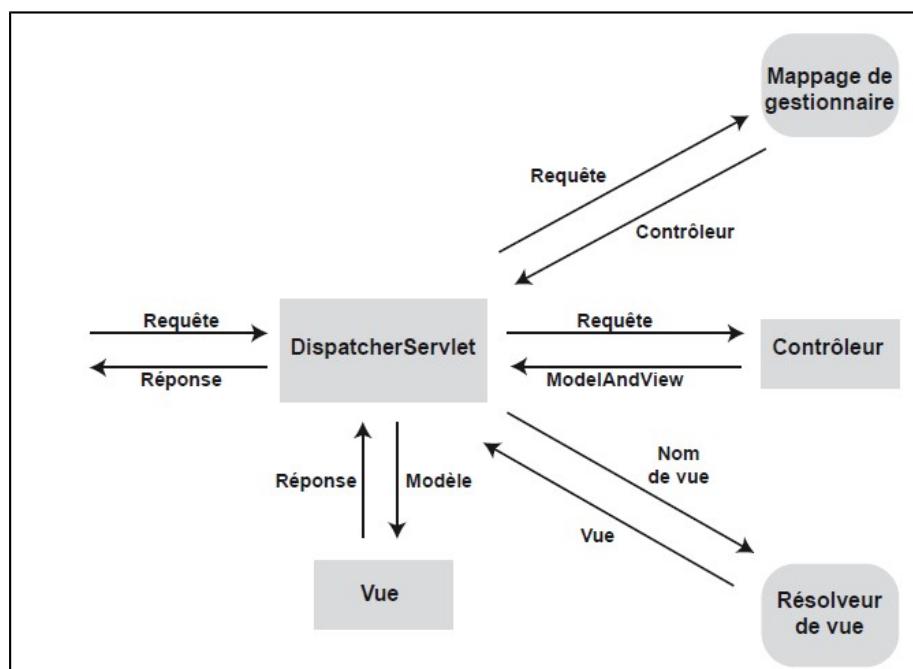
Figure : Entités de traitement des requêtes de Spring MVC

Les principaux composants de Spring MVC peuvent être répartis en trois groupes, selon leur fonction :

- Gestion du contrôleur façade et des contextes applicatifs. Permet de spécifier les fichiers des différents contextes ainsi que leurs chargements. Le contrôleur façade doit être configuré de façon à spécifier l'accès à l'application.

- Gestion des contrôleurs. Consiste à configurer la stratégie d'accès aux contrôleurs, ainsi que leurs différentes classes d'implémentation et leurs propriétés. L'aiguillage se configure désormais directement dans les classes mettant en œuvre des contrôleurs en se fondant sur des annotations. Ces dernières permettent également de mettre facilement à disposition les données présentes dans la requête, la session et le modèle.

- Gestion des vues. Consiste à configurer la ou les stratégies de résolution des vues ainsi que les frameworks ou technologies de vue mis en œuvre.



DispatcherServlet est le composant central de Spring MVC. Comme son nom l'indique, il distribue les requêtes vers les gestionnaires appropriés pour qu'ils prennent en charge ces requêtes. La seule servlet à configurer dans le descripteur de déploiement web, web.xml.

Les étapes sont :

- 1. Le DispatcherServlet reçoit une requête dont l'URI-pattern est '/home.htm'

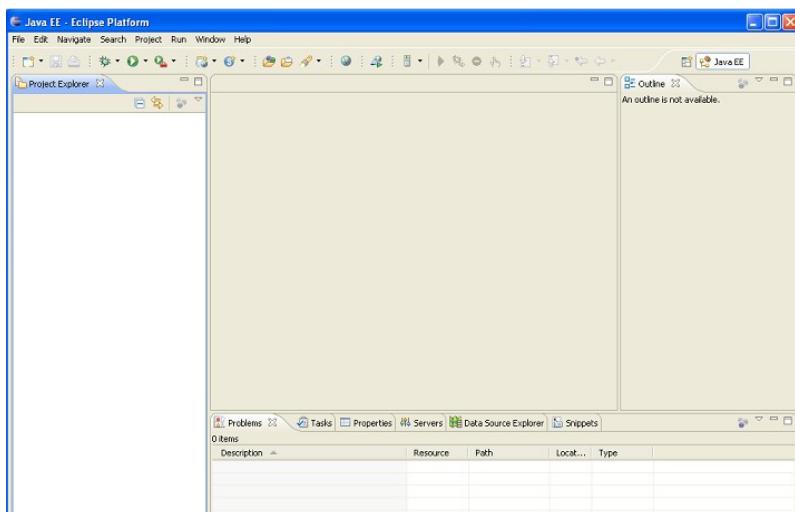
- ▶ 2.Le DispatcherServlet consulte son Handler Mapping (Ex : BeanNameUrlHandlerMapping) pour connaitre le contrôleur dont le nom de bean est '/home.htm'. En effet avec Spring MVC, vous créez vos contrôleurs.
- ▶ 3.Le DispatcherServlet dispatche la requête au contrôleur identifié (Ex :HomePageController)
- ▶ 4.Le contrôleur retourne au DispatcherServlet un objet de type ModelAndView possédant comme paramètre au minimum le nom logique de la vue à renvoyer.
- ▶ 5.Le DispatcherServlet consulte son View Resolver lui permettant de trouver la vue dont le nom logique est 'home'. Ici le type de View Resolver choisi est InternalResourceViewResolver.
- ▶ 6.Le DispatcherServlet forward la requête à la vue associé . Ici la page /WEB-INF/jsp/home.jsp

Partie 2 : création projet eclipse

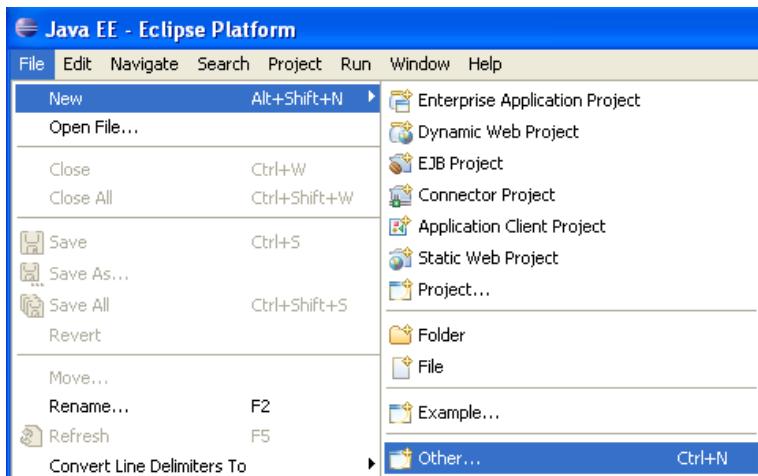
Création projet et intégration Tomcat

intégration tomcat

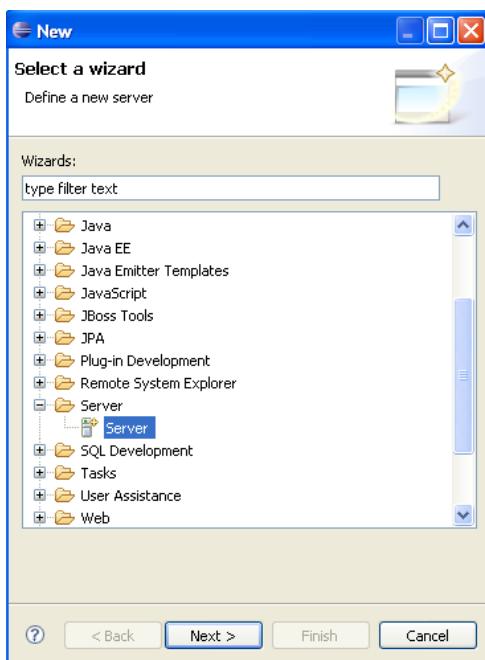
- ▶ Lancez votre version d'Eclipse Entreprise, qui contient un module WTP (Web Tool Platform) qui va nous servir pour intégrer tomcat dans Eclipse.



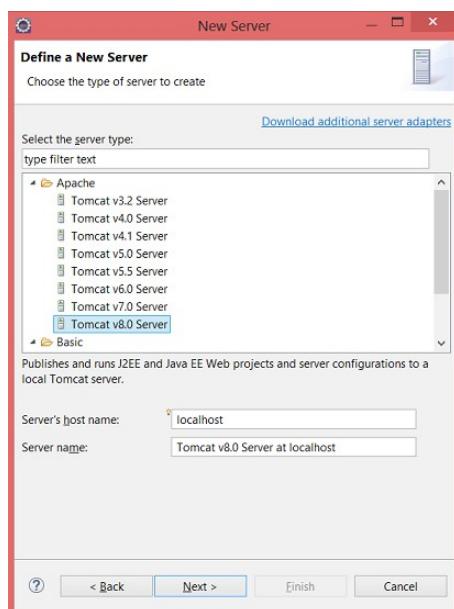
- ▶ Menu New->Other



L'écran suivant apparaît

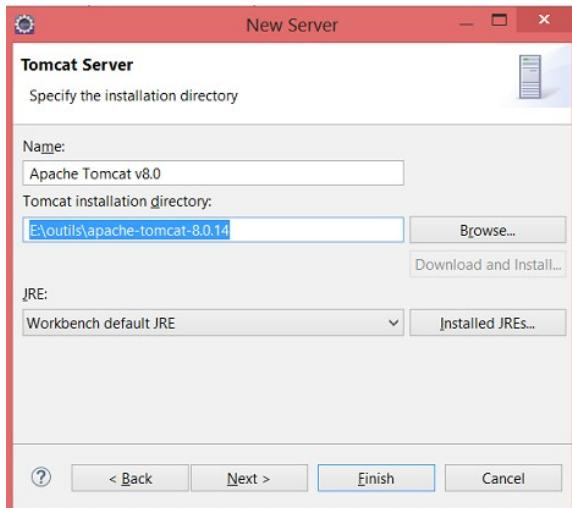


► Choisir **Server/Server** puis bouton **Next**

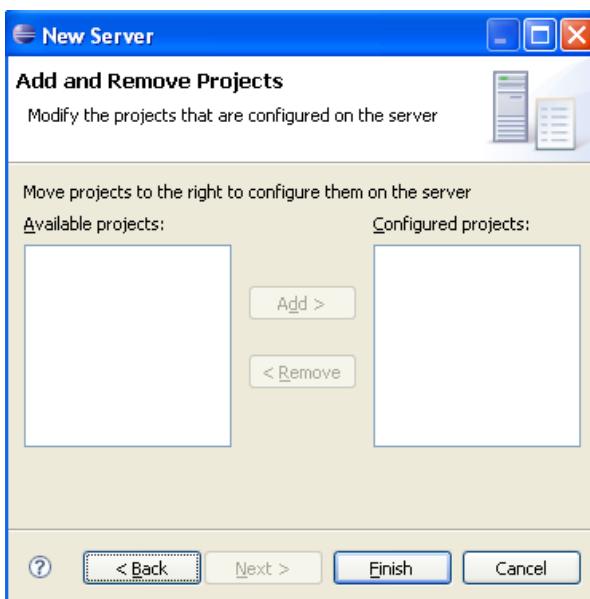


► Choisissez **Apache/Tomcat v8.0 Server** puis bouton **Next**

L'écran suivant apparaît

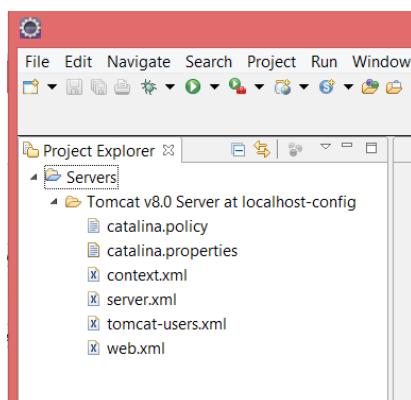


- ▶ Cliquez sur 'Browse' puis naviguez jusqu'au répertoire d'installation de tomcat.
- ▶ Cliquez sur le bouton **Next**



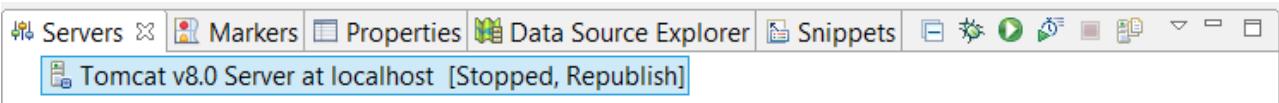
- ▶ Comme il n'y a aucun projet web à déployer, Cliquez sur le bouton **Finish**

Tomcat génère les informations suivantes dans la vue Explorer.

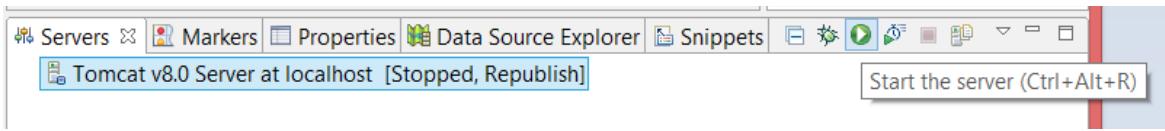


REMARQUE : Eclipse a fait **une copie** des fichiers de configuration du tomcat installé dans le workspace.

- ▶ Remarquez que qu'une nouvelle vue 'Servers' est apparue, vous informant que Tomcat est bien intégré à Eclipse, avec un statut Arrété (Stopped).

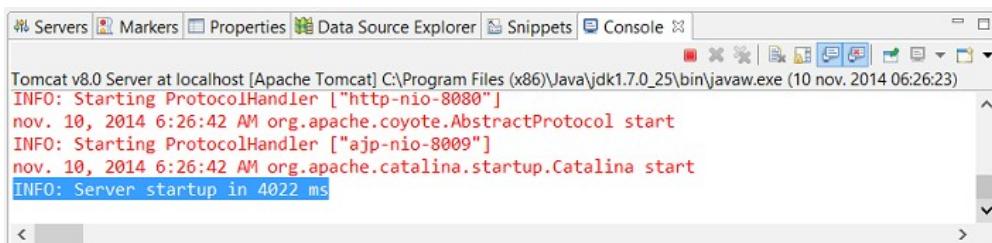


Partie 2 : Démarrage tomcat dans eclipse



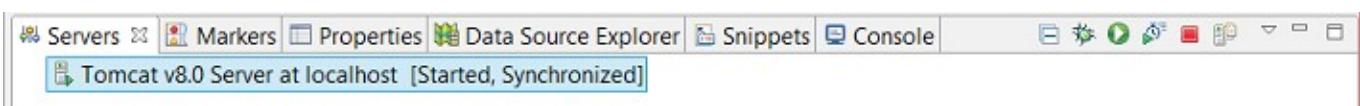
tutoriel-integration-tomcat-eclipse-5

Eclipse affiche dans la vue console les infos de logs Tomcat habituellement présent dans la console Invite de commande (ms dos).



tutoriel-integration-tomcat-eclipse-6

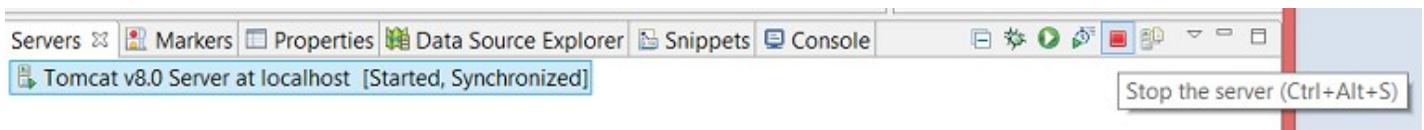
- Remarquez que dans la nouvelle vue 'Servers', Eclipse affiche un statut démarré (Started).



tutoriel-integration-tomcat-eclipse-7

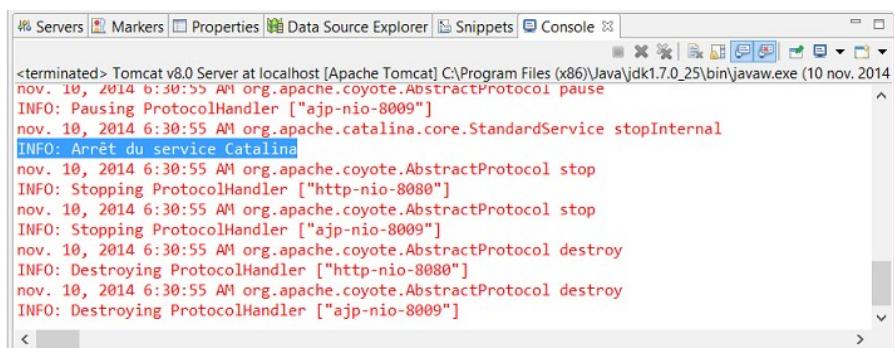
Partie 3 : Arrêt tomcat dans eclipse

- Bouton rouge .



tutoriel-integration-tomcat-eclipse-8

Tomcat est désormais arrêté.



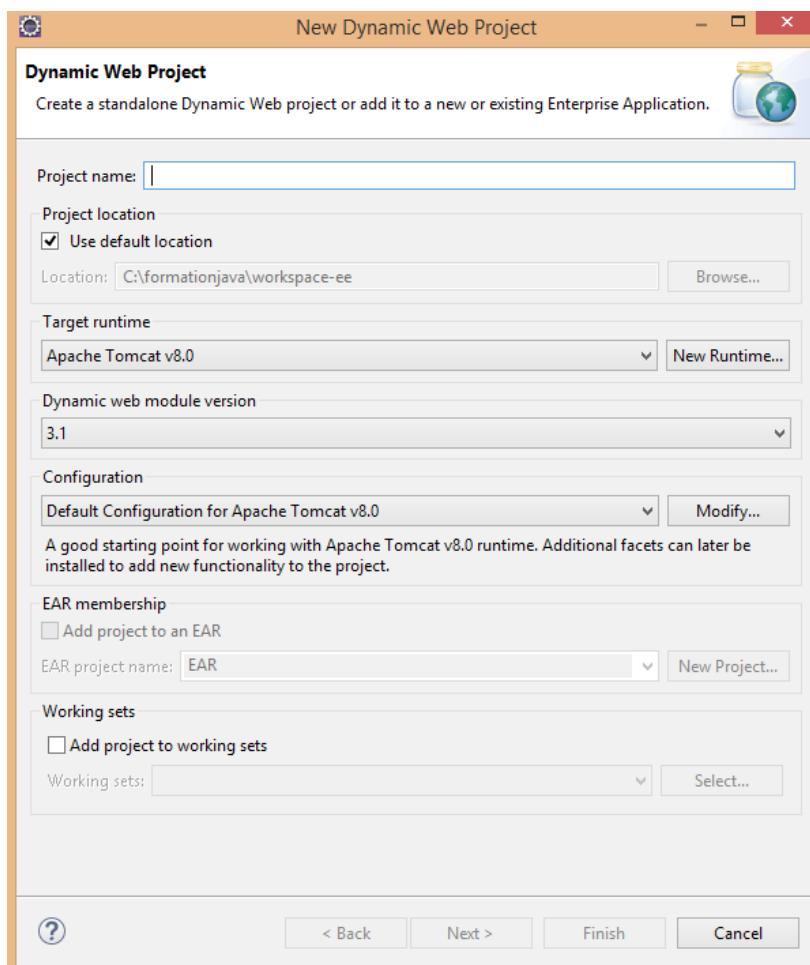
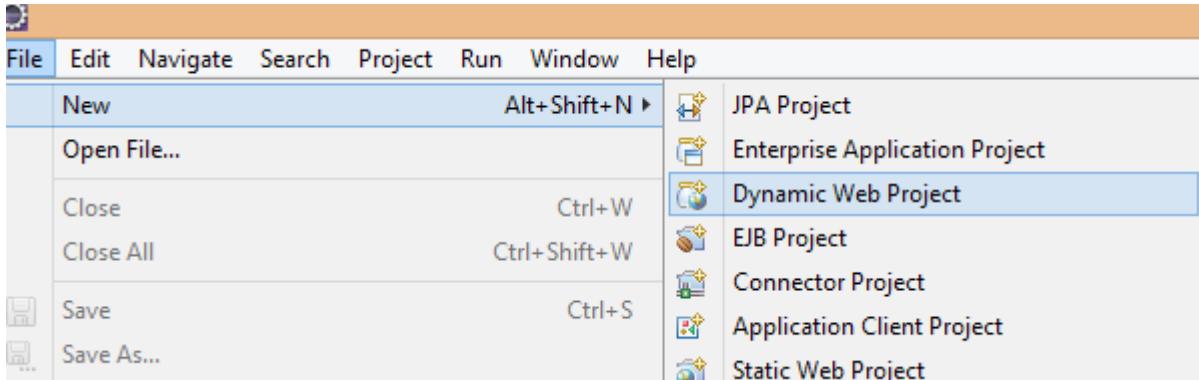
tutoriel-integration-tomcat-eclipse-9

- Créez un projet 'web dynamique' **I8_SpringMVC**. Menu File/new/Web/project puis web dynamique. (dynamic web project)

- ▶ Ajoutez un serveur Tomcat à votre environnement Eclipse.
- ▶ Testez que le serveur se lance bien à partir d'Eclipse.

Si besoin , voir 'notre 'tutoriel Eclipse : création projet web dynamique avec Tomcat 6 '

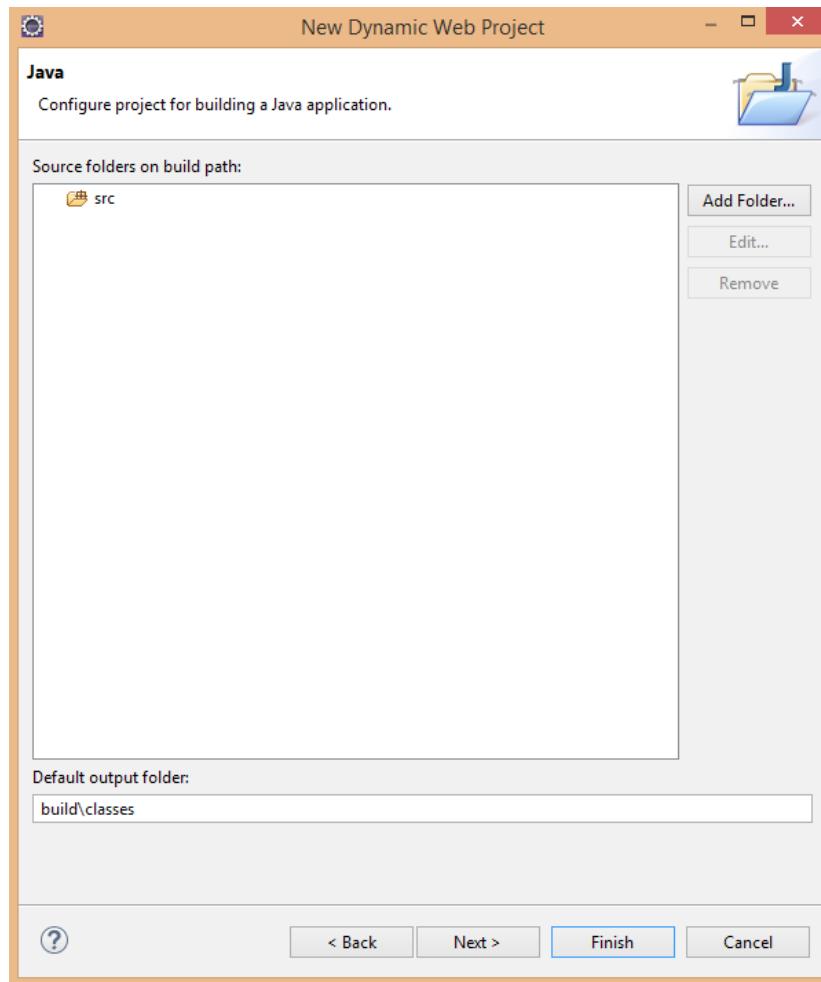
A partir d'Eclipse, faire menu File → New → Dynamic Web Project



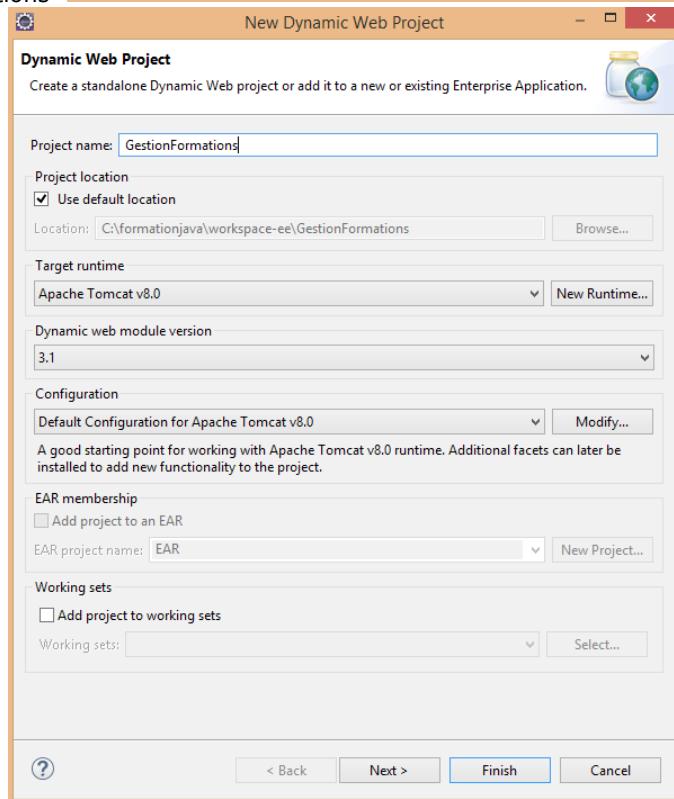
► L'écran suivant apparaît

REMARQUE

notez que le projet est associé à un serveur d'application, ici Tomcat 8. Cela est nécessaire. C'est une bonne pratique d'intégrer un serveur d'application Java dans Eclipse avant de créer un projet web dynamique. ▶ donnez un nom au projet,

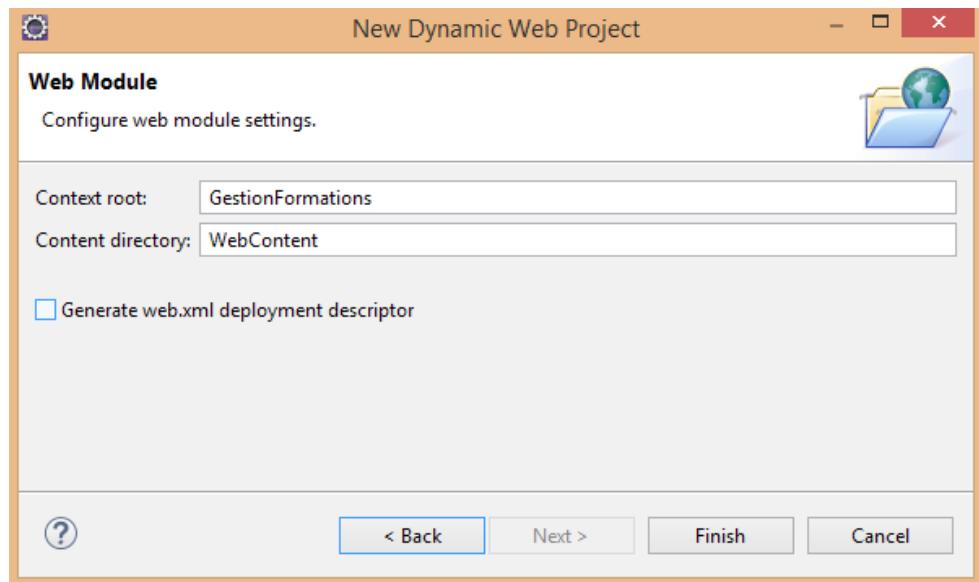


ici 'GestionFormations'



- ▶ Puis cliquez sur le bouton Next

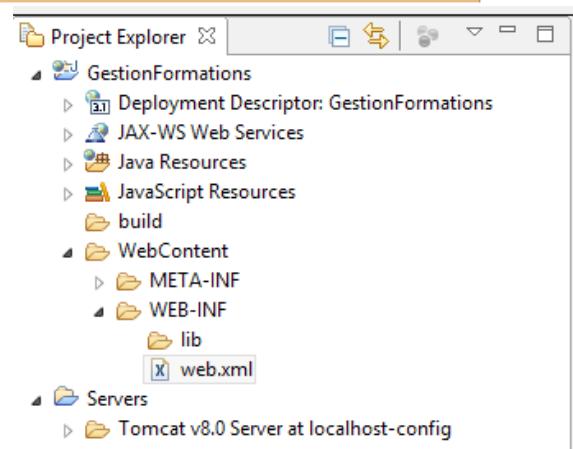
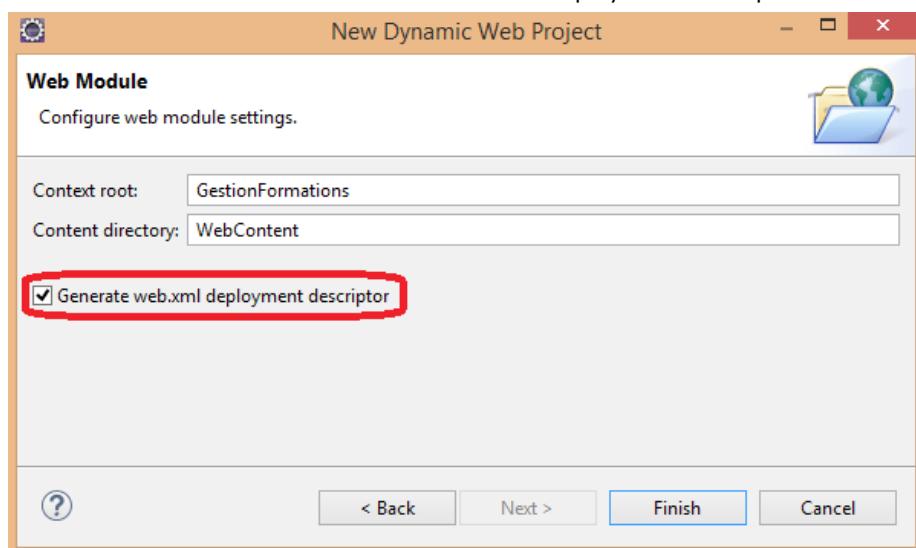
► Cliquez encore sur le bouton Next , l'écran suivant apparaît



Remarque

Qu'est ce que le fichier web.xml pour une application web Java ? Est-t-il obligatoire ?

► Cochez la case 'Generate web.xml deployment descriptor'



► Cliquez sur le bouton Finish et le projet suivant apparaît :

- Analysez le contenu du fichier web.xml

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/:
3   <display-name>GestionFormations</display-name>
4   <welcome-file-list>
5     <welcome-file>index.html</welcome-file>
6     <welcome-file>index.htm</welcome-file>
7     <welcome-file>index.jsp</welcome-file>
8     <welcome-file>default.html</welcome-file>
9     <welcome-file>default.htm</welcome-file>
10    <welcome-file>default.jsp</welcome-file>
11  </welcome-file-list>
12 </web-app>

```

Expliquez !!!

- Supprimez les pages d'accueil autre que index.html

```

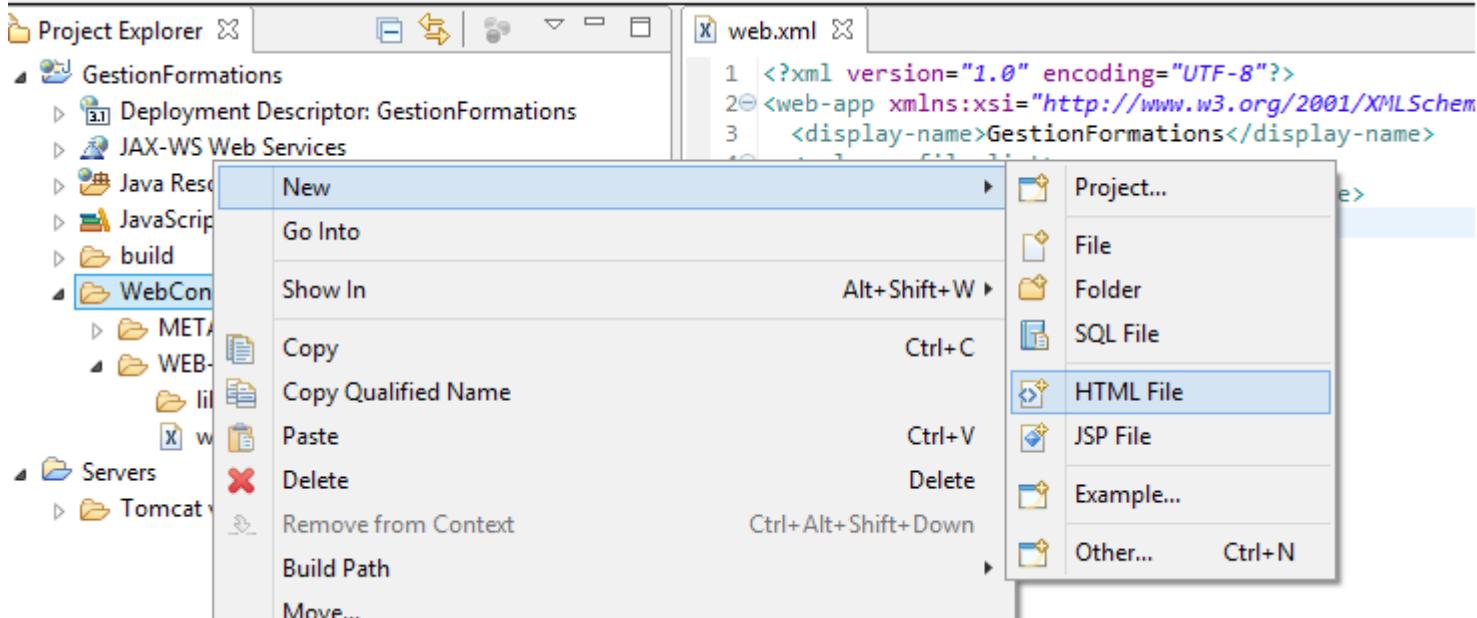
1 <?xml version="1.0" encoding="UTF-8"?>
2 <web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/:
3   <display-name>GestionFormations</display-name>
4   <welcome-file-list>
5     <welcome-file>index.html</welcome-file>
6   </welcome-file-list>
7 </web-app>

```

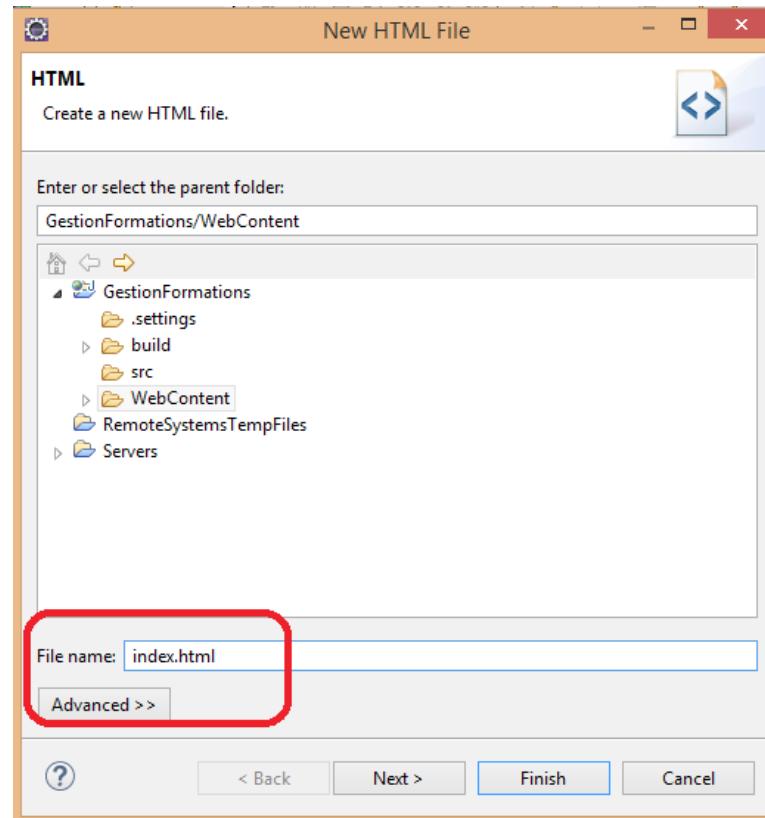
- Expliquez la grammaire du fichier. Quelles autres balises sont possibles ?

Partie 2 : Crédation d'une page d'accueil

Click droit sur le dossier WebContent -> New -> Html File



► L'écran suivant apparaît , vous remplacez le nom du fichier par index.html



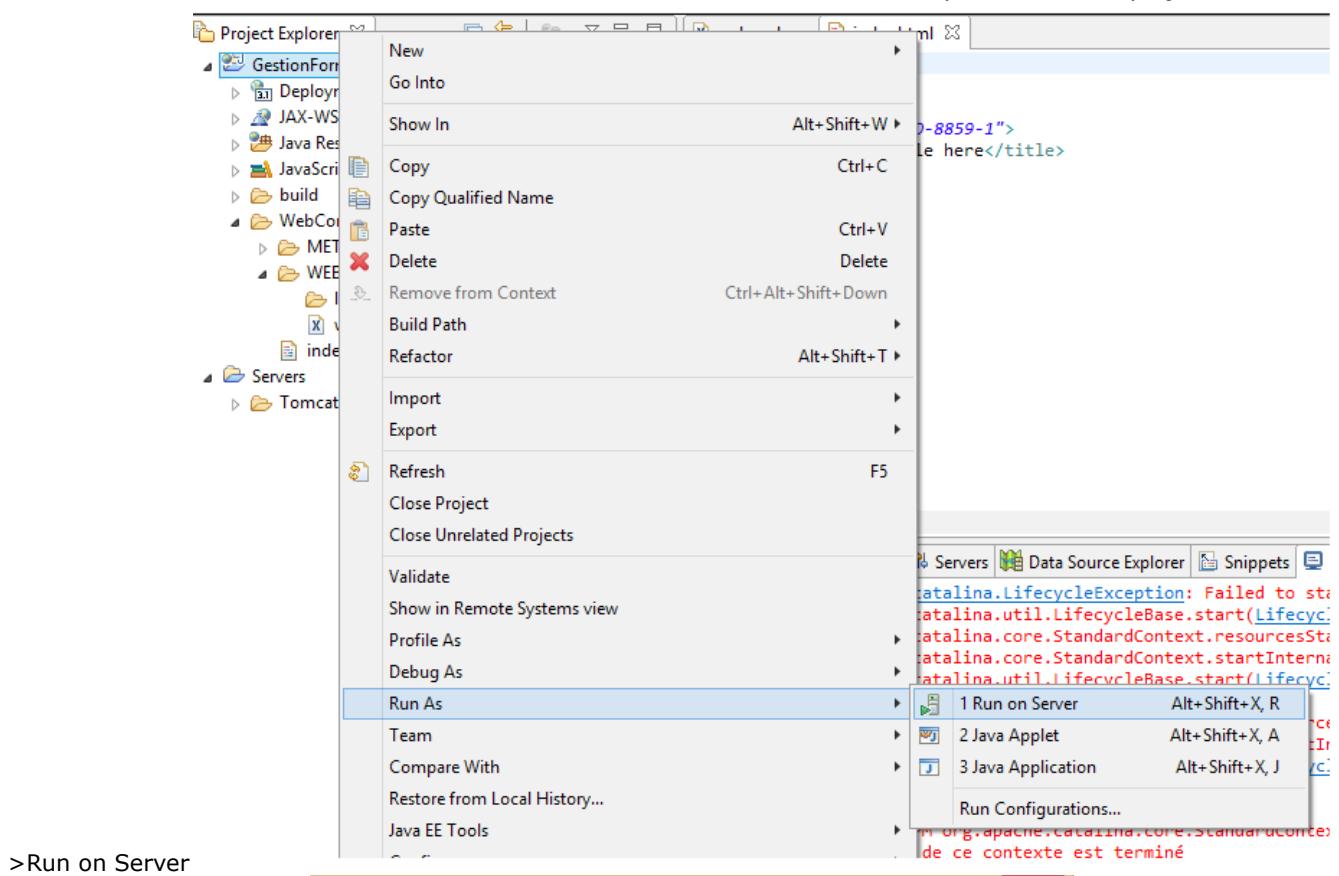
Cliquez sur le bouton Finish Et veuillez modifier le contenu du fichier index.html comme suit :

The screenshot shows the Eclipse editor with the 'index.html' tab selected. The code in the editor is:

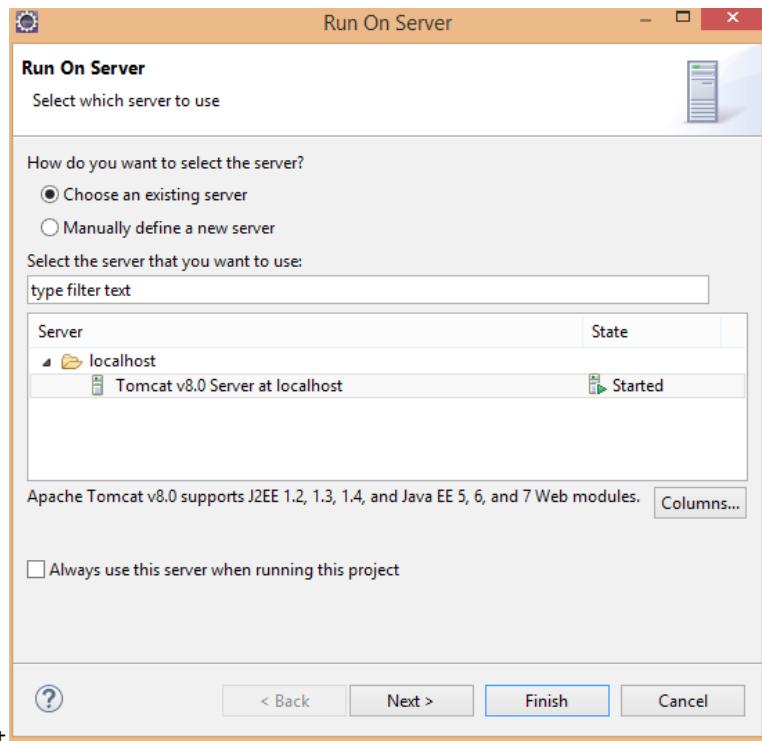
```
1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset="ISO-8859-1">
5 <title>Insert title here</title>
6 </head>
7 <body>
8 Bienvenue sur cette application web .
9 </body>
10 </html>
```

Partie 3 : Déploiement à partir d'Eclipse

► Cliquez droit sur le projet, Puis Run As

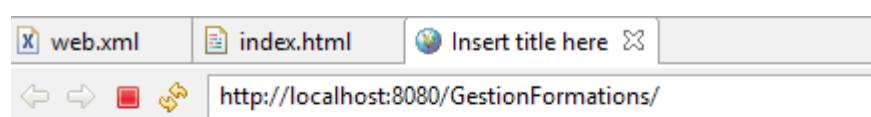


>Run on Server



► L'écran suivant apparaît

Cliquez sur le bouton Finish et l'écran suivant apparaît



To change eclipse's Internal Browser :

Preferences -> General

-> Web Browser.

Ajout des librairies

Dans le répertoire WEB-INF\lib du projet eclipse, importez (cliquez droit/import) les librairies du projet, composées de librairie de base ainsi que de librairies Spécifiques Spring MVC

Librairies de base

- ▶ **spring.jar** (le framework spring) : se trouve dans \spring-framework-2.5.4\dist\spring.jar
- ▶ **commons-logging.jar** (Interface pour gestion logs) se trouve dans \spring-framework-2.5.4\lib\jakarta-commons\commons-logging.jar
- ▶ **log4j.jar** (Implémentation pour les logs) : \spring-framework-2.5.4\lib\log4j\log4j-1.2.15.jar

Librairies spécifiques Spring MVC

- ▶ **spring-web.jar** (conteneur spécifique web) se trouve dans \spring-framework-2.5.4\dist\modules\spring-web.jar
- ▶ **spring-webmvc.jar** (Spring MVC) : se trouve dans \spring-framework-2.5.4\dist\modules\spring-webmvc.jar

Rendre le projet eclipse 'Spring Oriented'

- ▶ Cliquez-droit sur le projet puis Spring tools / Add Spring project Nature Notez le 'S' qui apparaît à gauche du nom du projet. C'est la marque d'un projet Spring dans Eclipse.

Mettre en oeuvre Log4j

En complément des jars commons-logging.jar et log4j.jar , un fichier de configuration de log est nécessaire pour terminer notre infrastructure de logs.

- ▶ Dans le répertoire WEB-INF, ajoutez **le fichier log4j.xml fournit**.
- ▶ Expliquez le contenu de ce fichier.

Partie 3 : Architecture en couche

Il s'agit d'une application en couche avec :

Couche accès données (DAO)

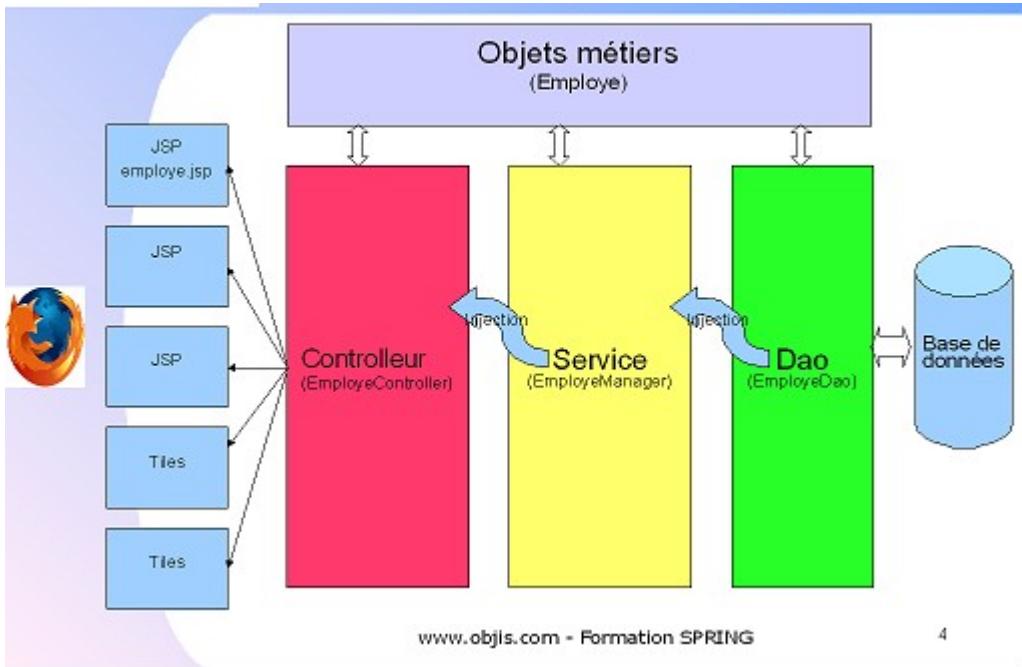
On y trouve des beans implémentant l'interface IEmployeDao permettant de communiquer avec la base à travers des opérations CRUD (Create, Retrieve, Update, Delete).

- >Ajout du package : **fr.adaming.springmvc.dao**
 - >Ajout interface : **IEmployeDao.java**
 - >Ajout classe : **JdbcEmployeDaoSupport.java**

Couche des services métiers dans package

On y trouvera des beans implémentant l'interface IEmployeManager permettant de gérer les services métiers.

- >Ajout du package : **fr.adaming.springmvc.service**
 - >Ajout interface : **IEmployeManager.java**
 - >Ajout classe : **EmployeManager.java**



Couche domaine métier

On y trouvera des objets métiers JavaBeans (comme Employe)

Couche contrôleur

Avec Spring MVC, vous devez créer vous-même vos Contrôleurs (contrairement à Struts, où il n'y qu'un contrôleur : l'`ActionServlet`). Vos contrôleurs doivent impérativement retourner un objet de type Spring **ModelAndView**

Bonne pratique spring MVC : 1 fichier de définitions par couche

Les concepteurs de l'application ont choisi de ne pas déclarer TOUS les beans dans un seul fichier, par souci de maintenance et de modularité de l'application.

La configuration du chargeur de contexte est nécessaire afin que le ou les fichiers de déclaration de bean (à créer) suivant soient pris en charge. Bonne pratique : un fichier xml par couche.

- ▶ A partir de l'assistant création de fichiers de définitions du **plugin Spring IDE**, créez les fichiers suivants :
 - \WEB-INF\springapp-data.xml : beans de la couche persistence
 - \WEB-INF\springpp-service.xml : beans de la couche services métiers
 - \WEB-INF\springpp-servlet.xml : beans couche Contrôleurs, Mapping URL et Vues(viewResolver)

Les fichiers, créés par exemple, ressembleront à ceci :

```

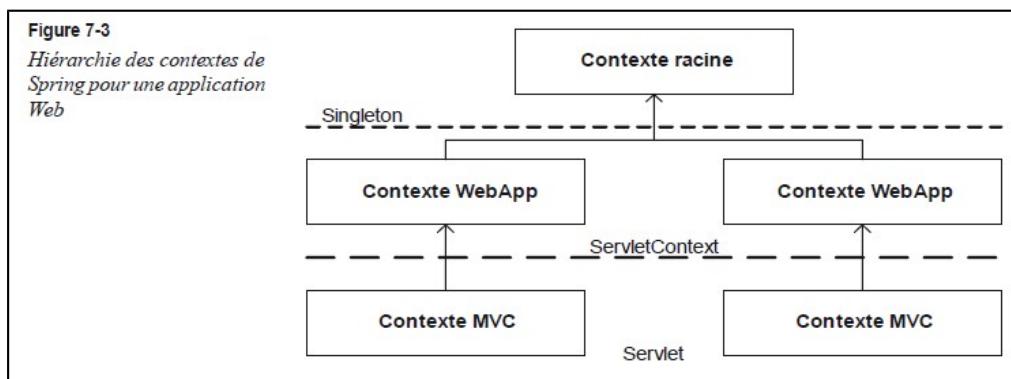
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans.xsd">
</beans>

```

Aucun bean n'y est déclaré pour le moment.

IMPORTANT : le nom de la servlet du Dispatcher (springapp) a un impact direct sur le nom des fichiers de déclaration des beans Contrôleurs, et doivent suivre le modèle 'nomservlet'-servlet.xml . Ici : springapp-servlet.xml

Partie 4 : Configuration web.xml en 2 étapes



(livre : spring par la pratique – page 188)

La mise en oeuvre du contexte de l'application Web est obligatoire, tandis que celle du contexte racine est optionnelle et n'est donc pas détaillée ici. Si ce dernier est omis, Spring positionne de manière transparente celui de l'application Web en tant que contexte racine. Le contexte utilise les mécanismes du conteneur de servlets

Chargement du contexte de l'application Web

Le framework Spring fournit une implémentation de la classe ServletContextListener de la spécification servlet permettant de configurer et d'initialiser ce contexte au démarrage et de le finaliser à l'arrêt de l'application Web.

Cette fonctionnalité est utilisable avec un conteneur de servlets supportant au moins la version 2.3 de la spécification. Cet observateur paramètre les fichiers de configuration XML du contexte en ajoutant les lignes suivantes dans le fichier **web.xml** de l'application

Chargement du contexte de Spring MVC

La configuration tout comme le chargement de ce contexte sont liés à ceux de la servlet du contrôleur façade de Spring MVC. Précisons que, par défaut, cette dernière initialise un contexte applicatif fondé sur un fichier <nom-servlet>-**servlet.xml**, lequel utilise le nom de la servlet précédente pour <nom-servlet>. Ce fichier se situe par défaut dans le répertoire **WEB-INF** ; la valeur de <nom-servlet> est spécifiée grâce à la balise **servlet-name**.

Dans notre étude de cas, la servlet de Spring MVC s'appelle tudu. Le fichier **tudu-servlet.xml** contient les différents composants utilisés par ce framework, comme les contrôleurs, les vues et les entités de résolution des requêtes et des vues.

Nous pouvons personnaliser le nom de ce fichier à l'aide du paramètre d'initialisation **contextConfigLocation** de la servlet. Le code suivant montre comment spécifier un fichier **mvc-context.xml** pour le contexte de Spring MVC par l'intermédiaire du paramètre précédemment cité.

- ▶ Etape 1 : Configurez le **DispatcherServlet** dans le fichier web.xml de l'application. Associez en particulier le DispatcherServlet aux URL-pattren '*.htm' . Nommez la servlet adamingapp.

```
<!-- CONFIG DispatcherServlet -->

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

<servlet-mapping>
    <servlet-name>objjisapp</servlet-name>
    <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

- ▶ Etape 2 : Configuration le chargeur de contexte

Afin d'exploiter au mieux toutes les informations présentes dans les différents fichiers de définition, le chargeur de contexte a besoin des 2 paramètres suivants, que vous devez préciser : La classe écouteur (Listener) de contextes Spring Le lieu du ou des fichiers à charger : **contextConfigLocation**

Ajoutez les lignes suivantes au fichier web.xml

```
<!-- CONFIG Chargeur de contexte (ContextLoader) -->

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

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/objjisapp-data.xml
        /WEB-INF/objjisapp-service.xml
        /WEB-INF/objjisapp-servlet.xml
    </param-value>
</context-param>
```

Remarque : autre choix possible pour le listener : **ContextLoaderServlet** . A utiliser uniquement si Serveur d'application supportant spécification servlet 2.2 et inférieur Serveur d'application n'initialisant pas les écouteurs (listeners) avant les servlets

Partie 5 : mise en œuvre Couche Domaine métier

->**dans dossier 'Java Resources'/src**

->Ajout du package : **fr.adaming.springmvc.domaine**

->Ajout classe : **Employe.java**

```
public class Employe {

    private int id;
    private String login;
    private String password;
    private String prenom;
    private String nom;
    private String email;
    private String role;

    public Employe() {
        super();
    }
    public Employe(int id, String login, String password, String prenom,
                  String nom, String email, String role) {
        super();
        this.id = id;
        this.login = login;
        this.password = password;
        this.prenom = prenom;
        this.nom = nom;
        this.email = email;
        this.role = role;
    }

    public int getId() {
        return id;
    }
}
```

Partie 5 : mise en oeuvre Couche DAO

- ▶ Ajoutez dans les librairies de l'application les jars relatifs au pool de connexion DBCP. Ces sont fournis dans Spring (\lib\jakarta-commons\commons-dbcp.jar et \lib\jakarta-commons\commons-pool.jar)
- ▶ Ajoutez le jar de MySQL : **mysql-connector-java-bin.jar** , fourni dans le zip de **mysql-connector**
- ▶ En utilisant les techniques mises en oeuvre dans le TP '[Accès aux données avec Spring JDBC](#)' , mettez en oeuvre dans le fichier **objjisapp-data.xml** une couche d'accès aux données MySQL impliquant :
 - Définition d'un bean '**dataSource**' : configuration via classe org.apache.commons.dbcp.BasicDataSource, (de Apache DBCP DataBase Connexion Pool) avec informations d'accès à la base récupérées d'un fichier de propriété : db.properties du package com.objjis.springmvcdemo.dao . Rappel : cette technique implique la déclaration d'un bean PropertyPlaceholderConfigurer
 - Définition d'un bean '**employeJdbcDaoSupport**' correspondant à l'utilisation d'un DAO Support de Spring pour accéder aux données. Le bean implementera l'interface IEmployeDAO ci-dessous.

```

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

    <!-- Déclaration du PropertyPlaceholderConfigurer -->
    <bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
        <property name="locations">
            <list>
                <value>classpath:/com/objis/springmvcdemo/dao/db.properties</value>
            </list>
        </property>
    </bean>

    <!-- Déclaration de la DATASOURCES -->

    <bean id="dataSource" destroy-method="close"
        class="org.apache.commons.dbcp.BasicDataSource">
        <property name="driverClassName" value="${db.driver}" />
        <property name="url" value="${db.url}" />
        <property name="username" value="${db.username}" />
        <property name="password" value="${db.password}" />
    </bean>

    <!-- Déclaration des DAO JDBC-->

    <bean id="jdbcEmployeDaoSupport"
        class="com.objis.springmvcdemo.dao.JdbcEmployeDaoSupport">
        <property name="dataSource" ref="dataSource" />
    </bean>

</beans>

```

Remarque : la datasource est injectée dans le DAO à l'exécution.

Exemple de configuration de spring-data.xml

```

<bean
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="locations">
        <list>
            <value>
                classpath:/com/objis/springmvcdemo/dao/db.properties
            </value>
        </list>
    </property>
</bean>

<bean id="dataSource" destroy-method="close"
    class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${db.driver}" />
    <property name="url" value="${db.url}" />
    <property name="username" value="${db.username}" />
    <property name="password" value="${db.password}" />
</bean>

<bean id="employeJdbcDaoSupport"
    class="com.objis.springmvcdemo.dao.EmployeJdbcDaoSupport">
    <property name="dataSource" ref="dataSource" />
</bean>

```

Avec un contenu de db.properties (dans com.objis.springmvcdemo.dao) :

```

db.driver=com.mysql.jdbc.Driver
db.url=jdbc:mysql://localhost/formation
db.username=root
db.password=

```

Interface à implémenter pour la couche DAO

```

package com.objis.springmvcdemo.dao;

import java.util.List;
import com.objis.springmvcdemo.domaine.Employe;

public interface IEmployeDao {

    public void saveEmploye(Employe employe);
    public Employe getEmployeById(int id);
    public Employe getEmployeByLogin(String login);
    public int getEmployesCount();
    public List<Employe> getAllEmployes();

}

```

Classe à implémenter pour la couche DAO : JdbcEmployeDaoSupport

```

public class JdbcEmployeDaoSupport extends JdbcDaoSupport implements IEmployeDao{

    public void saveEmploye(Employe employe) {

        final String EMPLOYE_INSERT = "insert into employe (id,login, password, prenom, nom, email, role) "
                + "values (?, ?, ?, ?, ?, ?, ?)";

        /*
         * On récupère et on utilisera directement le jdbcTemplate
         */
        getJdbcTemplate().update(
            EMPLOYE_INSERT,
            new Object[] { employe.getId(), employe.getLogin(), employe.getPassword(),
                employe.getPrenom(), employe.getNom(), employe.getEmail(), employe.getRole() });
    }

    public List<Employe> getAllEmployes() {
        final String ALL_EMPLOYE = "select * from employe";
        // TODO récupération de tous les employés

        return (List<Employe>)getJdbcTemplate().queryForList(ALL_EMPLOYE);
    }

    public Employe getEmployeById(int id) {
        // TODO récupération d'un employé en fonction de son Id

        String sql = "select id, nom, prenom from employe where id = ?";

        // Mapping d'un enregistrement vers un ResultSet
        RowMapper mapper = new RowMapper() {
            public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
                Employe employe = new Employe();

```

Partie 6 : mise en oeuvre couche Service

Codage du Service

- ▶ Proposez un code pour la classe **EmployeManager** , dans package com.objis.springmvcdemo.service

```

package com.objis.springmvcdemo.service;

import java.util.ArrayList;
import java.util.List;

import com.objis.springmvcdemo.dao.IEmployeDao;
import com.objis.springmvcdemo.domaine.*;

public class EmployeManager implements IEmployeManager {

    private IEmployeDao employeDAO;

    public void setEmployeDao(IEmployeDao employeDAO) {
        this.employeDAO = employeDAO;
    }

    public Employe getEmploye(String login) {
        Employe employe = employeDAO.getEmployeByLogin(login);
        return employe ;
    }

    public Employe getEmploye(int id) {
        Employe employe = employeDAO.getEmployeById(id);
        return employe ;
    }

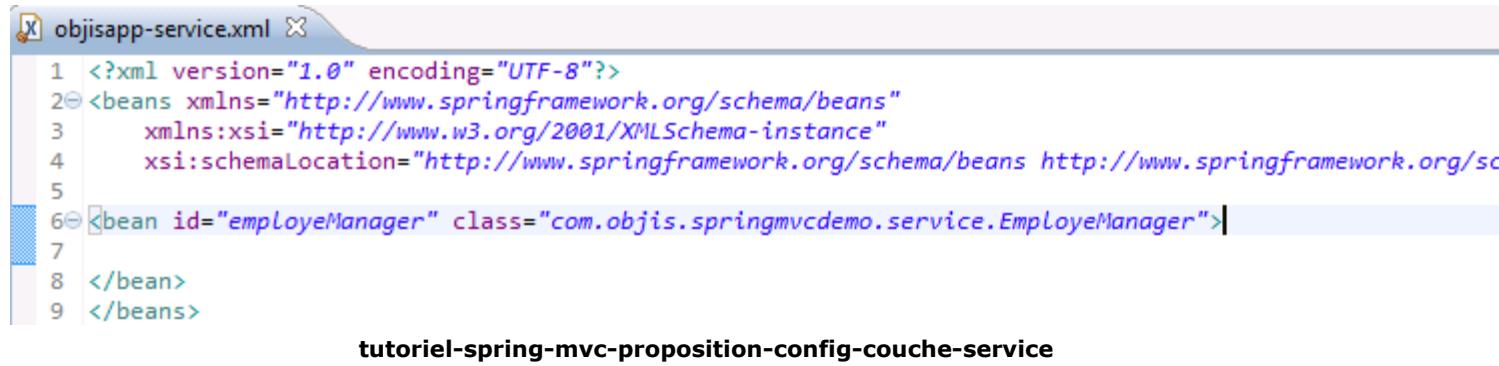
    public List<Employe> getAllEmployees() {

        List<Employe> listAllEmployees = new ArrayList<Employe>();
        listAllEmployees = (List<Employe>) employeDAO.getAllEmployees();
        return listAllEmployees;
    }
}

```

Analyse

Un développeur propose la déclaration suivante du Manager Employe, dans le fichier adamingapp-service.xml.



```

objisapp-service.xml

1 <?xml version="1.0" encoding="UTF-8"?>
2<beans xmlns="http://www.springframework.org/schema/beans"
3   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4   xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/s
5
6<bean id="employeManager" class="com.objis.springmvcdemo.service.EmployeManager">
7
8</bean>
9</beans>

```

tutoriel-spring-mvc-proposition-config-couche-service

Validez-vous sa proposition ? Que manque t'il ?

Injection du DAO

- ▶ Injectez un le DAO dans le Manager.

```

<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-beans.xsd">
    ...
    <bean id="employeManager" class="com.objis.springmvcdemo.service.EmployeManager">
        <property name="employeDAO" ref="jdbcEmployeDaoSupport"></property>
    </bean>
</beans>

```

Partie 7 : mise en oeuvre couche controller.

Cours sur les controllers :

Les types de contrôleurs :

Spring MVC Controllers

In: <http://jeefacile.unblog.fr/2009/06/09/spring-mvc-controllers/>

Objectifs:

- Déterminer les cas d'utilisation des contrôleurs implémentés par Spring MVC
- Déterminer le fonctionnement des contrôleurs : AbstractController et AbstractCommandController

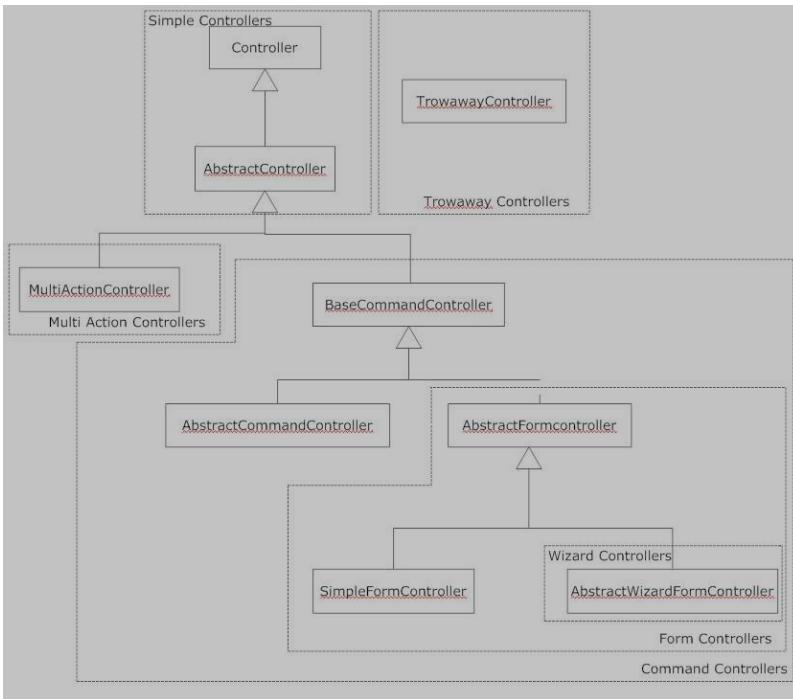
Spring MVC controllers

Un grand plus que marque Spring MVC par rapport aux autres framework telle que Struts ou WebWork, c'est qu'il offre une panoplies de contrôleurs dont chaque un est spécifique à un traitement bien particulier. Mais pour profiter de cet avantage il faut bien comprendre le fonctionnement de chaque contrôleur à fin de faire le bon choix.

Les contrôleurs de Spring MVC peuvent être classés en six catégories, nous décrivons dans le tableau ci-dessous en quel cas on peut choisir chaque type.

Type de Contrôleur	Classes	Cas d'utilisation
Simple	Controller (interface) AbstractController	Simple contrôleur permettant de faire le traitement basique d'une servlet (intercepter la requête, la traiter et retourner le résultat)
Throwaway	ThrowawayController	Considéré la requête comme une commande. Vous avez besoin d'un simple contrôleur sans avoir besoin d'accéder aux classes HttpServletRequest et HttpServletResponse.
Multi Action	MultiActionController	Vous avez plusieurs traitements similaires
Command	BaseCommandController AbstractCommandController	Récupérer les données envoyées en paramètre de la requête
Form	AbstractFormController SimpleFormController	Vous avez besoin d'un formulaire avec tout le processus de récupération des données de validation...
Wizard	AbstractWizardFormController	Vous avez besoin d'un formulaire avec un workflow assez complexe.

Le diagramme de classe ci-dessous décrit les contrôleurs définis par Spring MVC.



Interface Controller

L'interface Controller définit le comportement minimal d'un contrôleur qui consiste à intercepter une requête et retourner le modèle et le view adéquats. L'interfaceController définit une seule méthode handleRequest().

AbstractController

AbstractController implémente le comportement minimal d'un contrôleur. Il implémente la méthode handleRequest() et définit une méthode abstracthandleRequestInternal() qui doit être implémentée par les contrôleurs spécifiques qui héritent de AbstractController.

La méthode handleRequest() implémente le scénario suivant :

1. Vérifier si la méthode utilisée dans la requête HTTP (GET, POST, ...) est supportée par le contrôleur, si non une exception sera envoyée.
2. Vérifier si la session existe (en cas où le contrôleur est configuré pour que la session soit obligatoire). Exemple : dans le cas d'un workflow où la présence de la session est obligatoire pour le traitement, dans ce cas le contrôleur peut être configuré pour vérifier l'existence de la session si non une exception sera envoyée.
3. Vérifier si le Header Cache doit être utilisé, dans ce cas le Header sera configuré pour ceci.
4. Synchroniser la session (si le contrôleur est configuré pour ceci).
5. Exécuter la méthode handleRequestInternal() qui sera implémentée par les subClass.

Certes que AbstractController permet d'implémenter le comportement minimal d'un contrôleur, mais il permet aussi de vérifier la présence de la session, utiliser le Headercache et la synchronisation de la session. L'utilisation de ces fonctionnalités est configurable soit dans le fichier de configuration XML soit en utilisant des setters dans le constructeur du contrôleur.

AbstractCommandController

Tout d'abord il faut savoir que le mot « commande » n'a rien avoir avec le design pattern « commande » de GoF ; en fait dans notre cas le mot « commande » désigne le bean créé à partir des paramètres du requête HTTP.

Donc, comme vous avez compris, le AbstractCommandController est utilisé dans le cas où on a besoin de récupérer données envoyées en paramètre de la requête. Pour ceci il faut

1. Préciser au contrôleur le nom de la commande (ou le bean) qui va encapsuler les paramètres récupérés de la requête, ceci en [appelant](#) la méthode setCommandClass().
2. Implémenter la méthode handle() qui sera appelée par le dispatcher, cette méthode recevra en paramètre la « commande » créé à partir des données récupérées de la requête.

Exemple : soit l'exemple ci-dessous qui permet de récupérer les informations sur un utilisateur à partir d'un formulaire.

```
import javax.servlet.http.HttpServletRequest;

public class MyCommandController extends AbstractCommandController{
    public void MySimpleController(){
        setCommandClass(User.class);
    }

    @Override
    protected ModelAndView handle(HttpServletRequest request,
        HttpServletResponse response, Object command, BindException errors)
        throws Exception {
        User user = (User)command;
        if ( user.getUserName().equals("admin")){
            return new ModelAndView("success");
        }else{
            return new ModelAndView("failure");
        }
    }
}
```

Conclusion

Dans cet article nous avons vu quelques cas d'utilisation des contrôleurs définis par Spring MVC, ainsi que le mécanisme de fonctionnement des contrôleurs AbstractController et AbstractCommandController. Dans l'article prochain nous verrons, en détail, le contrôleur SimpleFormController.

Références

Expert Spring MVC and Web Flow by Seth Ladd, Darren Davison, Steven Devijver, and Colin Yates.

1- Contrôleurs de base :

Abstract controller

provides the basic support which can be used when you want to implement your own controller from scratch. It should be used for simple use like returning a resource (For example, any jsp page) to the client without checking request parameters.

Abstract controller needs you to override the **handleRequestInternal(HttpServletRequest, HttpServletResponse)** method and write your logic inside it. This method should return **ModelAndView** object.

```
public class HomeController extends AbstractController{  
    public void handleRequestInternal(HttpServletRequest request, HttpServletResponse response)  
    {  
        return new ModelAndView("home");  
    }  
}
```

If you want to **process request parameters** in the controller, then **instead of using servlet classes to get request parameters** in Abstract controller based controller, you can use [AbstractCommandController](#).

2- Contrôleur de gestion de formulaire : support pour l'affichage des données des formulaires et leur soumission

Le SimpleFormController

Le SimpleFormController est le controller standard fourni par Spring MVC. Il implémente comme tous les controllers l'interface Controller et met à disposition de son utilisateur toute une gamme de méthodes pour effectuer le traitement des formulaires basiques.

Le MultiActionController

Le MultiActionController est un controller spring mvc qui permet le traitement de plusieurs requêtes par un seul et unique controller. Il permet de mutualiser les pages qui nécessitent un traitement qui pourrait être plus ou moins similaire et de réduire la multiplication des controllers Spring.

AbstractWizardFormController

Ce controller permet de construire des formulaires formés de plusieurs pages.

Configuration du contrôleur de la page d'accueil

Rappel : avec Spring MVC, vous devez créer vous-même vos Contrôleurs (contrairement à Struts, où il n'y a qu'un contrôleur : l'ActionServlet).

Vos contrôleurs doivent impérativement retourner un objet de type Spring **ModelAndView**

Cinématique type du contrôleur – Relation avec DispatcherServlet et le Service métier

Pour la page d'accueil, les développeurs ont choisi un contrôleur (HomePageController) héritant de **org.springframework.web.servlet.mvc.AbstractController**. A partir de la documentation officielle de Spring, expliquez : Quelle est la particularité de ce type de contrôleur ?

Déclaration du contrôleur

```
<bean id="homePageController" class="com.objis.springmvcdemo.HomePageController">
    <property name="employeManager" ref="employeManager"></property>
</bean>
```

Remarque : le bean service employeManager (défini dans adamingapp-service.xml) est injecté à l'exécution dans le contrôleur. Le couplage entre les 2 est faible. C'est là un atout qui contribue à la modularité et une meilleure maintenance de l'application

Code source du contrôleur de la page d'accueil : HomePageController

-> ajout package : fr.adaming.springmvc.controllers

-> ajout classe : HomePageController

--EXEMPLE CONTROLLER SPRING MVC

```
package com.objis.springmvcdemo.controleur;

import java.util.List;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;
import com.objis.springmvcdemo.domaine.Employe;
import com.objis.springmvcdemo.service.EmployeManager;

public class HomePageController extends AbstractController {
    /*
     * INJECTION Spring d'un bean Service dans le contrôleur.
     * Le bean Service est configuré dans objissapp-service.xml
     */
    private EmployeManager employeManager;

    public void setEmployeManager(EmployeManager employeManager) {
        this.employeManager = employeManager;
    }

    protected ModelAndView handleRequestInternal(HttpServletRequest request,
                                                HttpServletResponse response) throws Exception {
        /*
         * Lancement du Service et récupération données en base
         */
        List<Employe> listeEmployes = employeManager.getAllEmployes();

        /*
         * Envoi Vue + Modèle MVC pour Affichage données vue
         */
        return new ModelAndView("home", "employes", listeEmployes);
    }
}
```

2

3

1

4

EXPLICATION

1) le bean Service est injecté à l'exécution dans le bean Controleur

2) Un objet de type ModelAndView objet est retournée via méthode handleRequestInternal du contrôleur. Vous créez vos contrôleurs en héritant un des contrôleurs de Spring, ici de type AbstractController.

Les différents contrôleurs ont un savoir-faire spécifique. Par exemple AbstractController sait uniquement renvoyer des infos , alors que SimpleForm sait récupérer infos d'un formulaire et aussi renvoyer à l'utilisateur (Aller/Retour)

3) Utilisation du service pour traitement métier.

4) En plus du nom logique de la vue à afficher (View), le contrôleur peut renvoyer un objet métier (model) .Cet objet métier sera utile pour les données à afficher dans la vue à afficher (jsp, tiles...). L'objet métier aura alors été crée et chargé via un Service métier en relation par exemple avec une base de données.

Exemple : Afficher en page d'accueil les employés de la société. return new ModelAndView("home","employes", listeEmployes) ;

- "**home**" : nom logique de la vue à afficher. Utilisé par le view-resolver pour rechercher la vue
- "**employes**" : Nom de l'objet du modèle (MVC) à transmettre à la vue
- "**listeEmployes**" : valeur de l'objet du modèle à transmettre à la vue

Partie 7 : Handler Mapping page d'accueil.

Pour la page d'accueil (url '/home.htm'), les développeurs ont choisi un handler de type org.springframework.web.servlet.handler.BeanNameUrlHandlerMapping. Pourquoi ?

Déclaration du Handler Mapping dans \WEB-INF\adamingapp-servlet.xml

Dans le fichier objisapp-servlet.xml, configurez un tel handler en ajoutant les lignes suivantes :

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

Cette configuration permet de déclarer ceci (à ajouter) :

```
<bean name="/home.htm" class="com.objis.springmvcdemo.HomePageController">
    <property name="employeManager" ref="employeManager" />
</bean>
```

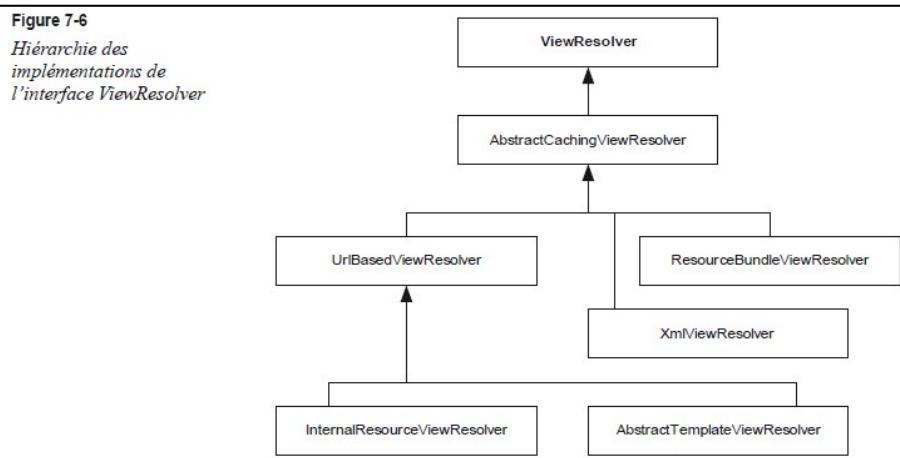
Ainsi c'est le contrôleur HomePageController qui va gérer le traitement de la requête. Le bean Service lui est injecté.

Configuration du View resolver pour la page d'accueil : InternalResourceViewResolver

Configuration de la vue (livre page 207)

La sélection des vues dans Spring MVC est effectuée par le biais d'une implémentation de l'interface ViewResolver dans le package org.springframework.web.servlet, comme le montre le code suivant :

```
public interface ViewResolver {
    View resolveViewName(String viewName, Locale locale);
}
```



ResourceBundleViewResolver

La première implémentation, ResourceBundleViewResolver, correspond à une configuration au cas par cas des vues utilisées. Cette approche est particulièrement intéressante pour une utilisation des vues fondées sur différentes technologies de présentation.

XmlViewResolver

Les vues sont définies par l'intermédiaire de cette implémentation au cas par cas, comme précédemment, mais dans un sous-contexte de Spring. L'utilisation de toutes les fonctionnalités et mécanismes du framework est donc envisageable, de même que l'injection de dépendances sur la classe d'implémentation des vues.

InternalResourceViewResolver

L'implémentation InternalResourceViewResolver utilise les URI dans le but de résoudre les vues fondées, par exemple, sur les technologies JSP/JSTL. Ce mécanisme construit l'URI à partir de l'identifiant de la vue puis dirige les traitements vers d'autres ressources gérées par le conteneur de servlets, telles que des servlets ou des JSP, comme dans l'exemple ci-dessous :

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

Cette implémentation générale s'applique à toutes les vues, excepté celles qui sont résolues précédemment par une autre implémentation dans une chaîne de ViewResolver.

Le choix de la stratégie de Vue a été porté par les développeurs sur le **InternalResourceViewResolver**.

Les lignes suivantes à recopier dans le fichier objsiapp-servlet.xml, permettent de configurer un viewResolver par défaut. C'est le choix recommandé et le plus simple si vos vues sont des JSP.

```
<!-- View Resolver : Toutes les vues sont des JSP-->  
  
<bean id="viewResolver" class="org.springframework.web.servlet.view.InternalResourceViewResolver">  
    <property name="prefix">  
        <value>/WEB-INF/jsp/</value>  
    </property>  
    <property name="suffix">  
        <value>.jsp</value>  
    </property>  
</bean>
```

À l'URL '**home**.htm' sera renvoyé une page JSP suivante : \WEB-INF\jsp\home.jsp

Partie 8 : codage de la vue : fichier JSP

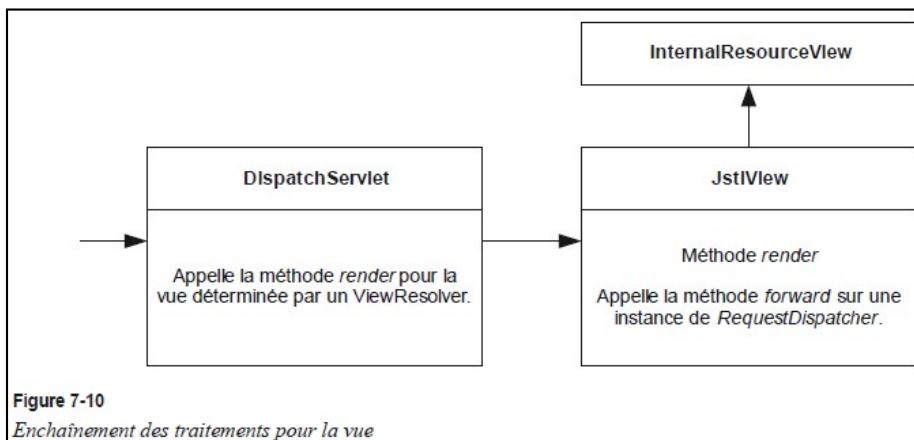
Les technologies de présentation

Dans Spring MVC, une vue correspond à une implémentation de l'interface View du package **org.springframework.web.servlet**

Vue fondée sur JSP/JSTL

Spring MVC fournit une vue fondée sur JSP/JSTL, dirigeant les traitements de la requête vers une page JSP dont l'URI est construit à partir de l'identifiant de la vue.

La figure 7-10 illustre l'enchaînement des traitements afin d'utiliser une vue de type JstlView.



Le développeur prend uniquement en charge le développement de la page JSP, tandis que Spring MVC a la responsabilité de mettre à disposition dans la requête tout le contenu du modèle ainsi qu'éventuellement des informations concernant le formulaire.

Les balises et expressions JSTL peuvent être utilisées d'une manière classique en utilisant les données du modèle, ces dernières étant mises à disposition par Spring MVC pour les pages JSP. Pour une entrée ayant pour clé maVariable dans le modèle, la page JSP récupère la valeur de sa propriété maPropriete correspondante de la manière suivante.

Exemple d'implémentation d'une vue récupérant le modèle identifié par 'emploies' et envoyé par le contrôleur à travers objet ModelAndView.

Ici la page est une page JSP .

Des balises JSTL et langage EL sont utilisés pour parcourir liste d'employés et les afficher leur propriétés.

Vous aurez besoin d'importer dans vos librairies une implémentation de JSTL, regroupant les jars suivants :

- **jstl.jar** , à récupérer dans \lib\j2ee\jstl.jar
- **standard.jar** , à récupérer dans \lib\jakarta-taglibs\standard.jar

```

<%@ page contentType="text/html"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<title>Equipe société Objis </title>
</head>
<body>
<h2>Ils vont vous faire aimer Java :</h2>
<ul>
    <c:forEach items="${employes}" var="employe">
        <li>
            <c:out value="${employe.nom}" />
            <c:out value="${employe.prenom}" />
            <c:out value="${employe.email}" />
        </li>
    </c:forEach>
</ul>

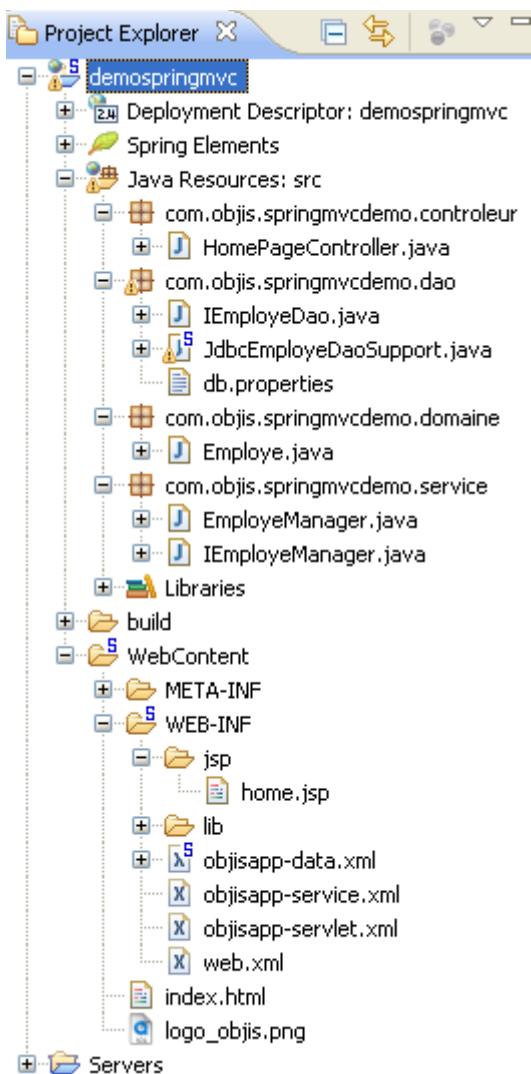
<a href="http://www.objis.com"></a>

</html>

```

Arborescence Eclipse

Voici l'arborescence juste avant de faire un premier test :



Partie 9 : Déploiement

Cliquez-droit sur le projet puis /Run as/ Run on Server. Vous obtenez un écran similaire à celui-ci :



Partie 10 : mapper les requêtes avec un autre contrôleur Spring

Dans la page d'accueil, nous avons utilisé le Handler **BeanNameUrlHandlerMapping**.

Spring 2.0 rend disponible d'autres handlers parmi lesquels le plus utilisé est **SimpleUrlHandlerMapping**. Ajoutez dans objisapp-servlet.xml la déclaration suivante, dans laquelle un autre contrôleur est configuré :

```
<bean id="simpleUrlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/home.htm">homePageController</prop>
        </props>
    </property>
</bean>
```

et le bean

```
<bean id="homePageController" class="com.objis.springmvcdemo.HomePageController">
    <property name="employeManager" ref="employeManager"></property>
</bean>
```

Redéployez l'application. Que se passe t'il ?

QUESTION : Quelle différence entre SimpleUrlHandlerMapping et BeanNameUrlHandlerMapping ?
Comment gérer le risque de conflit dans le traitement de '/home.htm'

Modifiez la déclaration du SimpleUrlHandlerMapping comme ci-dessous :

```

<bean id="simpleUrlMapping" class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/home.htm">homePageController</prop>
        </props>
    </property>
    <property name="order" value="0"/>
</bean>

```

Quelle différence ?

- ▶ Redéployez l'application. Quel impact ?

Partie 11 : Annotations Spring 2.5

Avec Spring 2.5 les amateurs d'annotations peuvent se réjouir ! En particulier en ce qui concerne le codage du contrôleur.

Avantages

- pas besoin d'une interface à implémenter (Ex : interface Controller) ou d'une classe à étendre (Ex : SimpleFormController)
- Permet plusieurs méthodes pour gérer requêtes (POST , GET)
- Permet grande flexibilité sur signature des méthodes

Nouvelle configuration Spring

Analysez le nouveau contenu du fichier objsapp-servlet.xml

The screenshot shows the XML configuration file for the application. The XML code is as follows:

```

<?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-2.5.xsd">

    <!-- Tenir compte de configuration par annotations (@Autowired, @Required, @Qualifier...) -->
    <context:annotation-config/>

    <!-- Location des composants de type @Component, @Controller, @Service, @Repository... -->
    <context:component-scan base-package="com.objis"></context:component-scan>

    <!-- Handler Mapping pour annotations -->
    <bean class="org.springframework.web.servlet.mvc.annotation.DefaultAnnotationHandlerMapping"></bean>

    <!-- View Resolver : Toutes les vues sont des JSP-->

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

```

Annotations:

- 1**: Lines 1-7: XML declaration and imports for beans, context, and their schema locations.
- 2**: Line 11: Configuration for annotations using the `context:annotation-config` element.
- 3**: Line 15: Configuration for component scanning with the `context:component-scan` element.
- 4**: Line 19: Configuration for annotation-based handler mapping using the `DefaultAnnotationHandlerMapping` bean.
- 5**: Lines 23-31: Configuration for the view resolver, specifying the prefix and suffix for JSP files.

tutoriel-springmvc-config-controleur-annotations-spring-2-5-etapes

- ▶ Expliquez les 5 points de configuration identifiés

Nous allons désormais mettre à profit cette nouvelle configuration.

Implémentation N°1 contrôleur annoté /home

- ▶ Mettez en œuvre l'implémentation suivante

```
@Controller
public class HomePageController {
    /*
     * INJECTION Spring d'un bean Service dans le contrôleur.
     * Le bean Service est configuré dans objisapp-service.xml
     */
    private IEmployeManager employeManager;

    @Autowired
    public void setEmployeManager(IEmployeManager employeManager) {
        this.employeManager = employeManager;
    }

    @RequestMapping("/home")
    protected ModelAndView retourControleur(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        /*
         * Lancement du Service et récupération données en base
         */
        List<Employe> listeEmployes = employeManager.getAllEmployes();

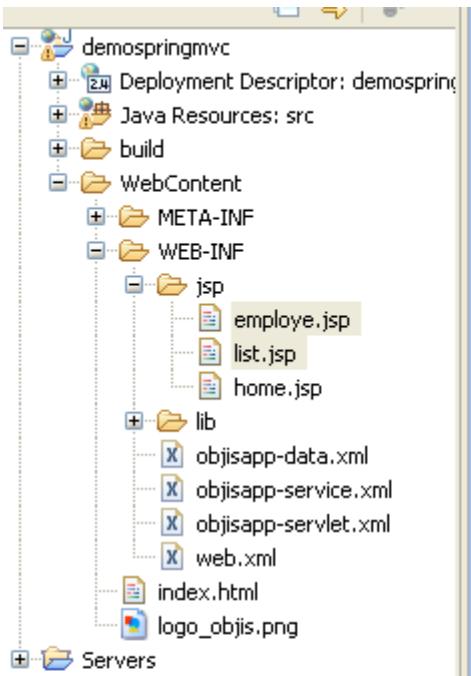
        /*
         * Envoi Vue + Modèle MVC pour Affichage données vue
         */
        return new ModelAndView("home", "employes", listeEmployes);
    }
}
```

tutoriel-springmvc-config-controleur-annotations-spring-2-5-code-controleur

Implémentation N°2 contrôleur annoté

Spring 2.5 vous permet d'aller plus loin dans la simplification du code du contrôleur.

- Nous allons supprimer l'objet Response des paramètres d'entrée.
- Au lieu de retourner un objet type ModelAndView, nous allons ajouter une Map comme paramètre d'entrée représentant le model.



```

@Controller
public class HomePageController {

    private IEmployeManager employeManager;

    @Autowired
    public void setEmployeManager(IEmployeManager employeManager) {
        this.employeManager = employeManager;
    }

    // Flexibilité du contrôleur avec Spring 2.5 : le contrôleur gère directement les requêtes
    @RequestMapping("/home")
    public void home(Model model) {
        // Lancement du Service et récupération données en
        List<Employe> listeEmployes = employeManager.getAll();
        model.addAttribute("employes", listeEmployes);
    }

    @RequestMapping("/list")
    public void list(Model model) {
        List<Employe> listeEmployes = employeManager.getAll();
        model.addAttribute("employes", listeEmployes);
    }

    @RequestMapping("/employe")
    public void detailEmploye(Model model) {
        // Lancement du Service et récupération données en
        Employe employe = employeManager.getEmployeById(1);
        model.addAttribute("employe", employe);
    }
}

```

tutoriel-springmvc-config-controleur-annotations-spring-2-5-code-controleur-v2

INFO : Pour tout objet du modèle ajouté dans la Map, Spring crée automatiquement une clé qui permettra de manipuler le modèle dans la vue renvoyée.

- ▶ Expliquez

Gestion des paramètres avec @RequestParam

- ▶ Récupérez un paramètre de l'url

```

@RequestMapping("/employe")
public void detailEmploye(@RequestParam("id") Integer id, Model model) {
    Employe employe = employeManager.getEmployeById(id);
    model.addAttribute("employe", employe);
}

```

tutoriel-springmvc-config-controleur-annotations-spring-2-5-page-employe-url-RequestParam

- ▶ Exemple de page Vue dédiée



```
<%@ page contentType="text/html"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<html>
<head>
<title>Equipe société Objis </title>
</head>
<body>
<h2>Employé du mois : ${employe.nom} ${employe.prenom}</h2>

<a href="http://www.objis.com"></a>

</html>
```

tutoriel-springmvc-config-controleur-annotations-spring-2-5-page-employe-url-employe

- ▶ Testez le rendu final



tutoriel-springmvc-config-controleur-annotations-spring-2-5-page-employe-url-employe-demo

Partie 12 : Sortir un PDF ou XLS

Sortie PDF

- ▶ A partir de la documentation officielle de Spring, créez une vue héritant de **AbstractPdfView** et permettant de ranger les informations de la vue dans un fichier PDF.
- ▶ Modifiez la configuration de Spring afin de bénéficier de cette nouvelle vue

Sortie XLS

- ▶ A partir de la documentation officielle de Spring, créez une vue héritant de **AbstractXlsView** et permettant de ranger les informations de la vue dans un fichier PDF.
- ▶ Modifiez la configuration de Spring afin de bénéficier de cette nouvelle vue

Partie 13 : Velocity au lieu de JSP

- ▶ Expliquez ce qu'est le moteur de template **Apache Velocity**. Quelle valeur ajoutée par rapport à JSP ? Quelle extension de fichier ?
- ▶ A partir de la documentation officielle de Spring, configurer le moteur velocity puis créez une vue associée
- ▶ Créez enfin un template velocity pour mettre velocity en action.

Conclusion

Dans ce tutoriel, nous avons introduit Spring MVC, l'offre de Spring pour la création d'écrans / vues web.

