

# Tutoriel sur la réalisation d'application Web simple avec Spring



Par Régis **POUILLER**

Date de publication : 19 mai 2014

Cet article présente la création d'une application Web avec le framework **Spring**.  
Il présente diverses techniques : utilisation de ressources JNDI, Spring MVC (contrôleur, formulaire avec validation, mapping, tiles), internationalisation, service et DAO avec JPA.

Une discussion a été ouverte pour les commentaires sur la publication de cet article :  
[\[Commentez\]](#)

I - Introduction.....	3
I-A - Objectif.....	3
I-B - Prérequis.....	3
I-C - Ressources.....	4
I-D - Versions des logiciels.....	4
II - Première application : Bonjour le monde !.....	4
II-A - Création de l'application.....	4
II-B - Déploiement de l'application.....	6
II-C - Création d'une JSP « bonjour.jsp ».....	11
II-D - Utilisation de Spring avec la JSP « bonjour.jsp ».....	11
II-E - Passage de donnée à la JSP « bonjour.jsp ».....	13
II-F - Récupération d'un paramètre de la requête HTTP.....	15
III - Affichage de données en base.....	17
III-A - Création de la base.....	17
III-B - Paramétrage du serveur Tomcat.....	19
III-C - Modification du projet afin d'inclure l'affichage des données.....	20
IV - Création de données en base.....	25
V - Suppression de données en base.....	29
VI - Modification de données en base.....	32
VII - Unification de l'application par un menu.....	37
VIII - Conclusion.....	42
IX - Remerciements.....	42

## I - Introduction

### I-A - Objectif

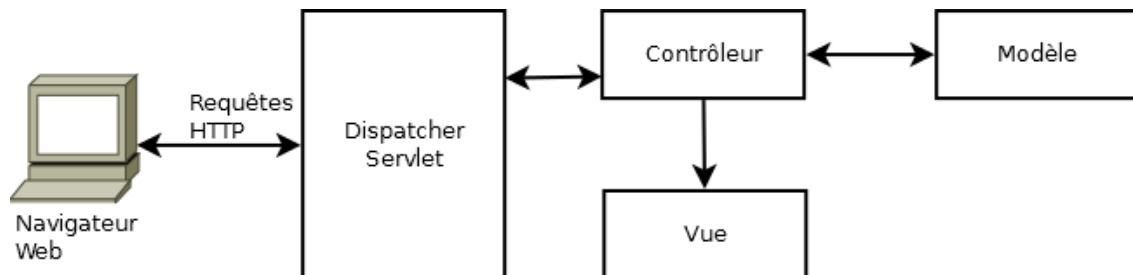
Tout au long de cet article, nous allons voir différents aspects de la réalisation d'une application Web avec le framework **Spring**. Les différentes étapes sont :

- développer une page Web affichant « Bonjour le monde » : création d'une JSP simple affichant le message et modification de la JSP simple avec Spring (avec et sans paramètre de requête) ;
- développer une page Web affichant les valeurs d'une base de données ;
- développer une page Web permettant de créer des valeurs en base de données ;
- développer une page Web permettant de supprimer des valeurs en base de données ;
- développer une page Web permettant de modifier des valeurs en base de données ;
- unifier ces pages afin d'en faire une application Web regroupant ces fonctionnalités.



Ces étapes sont l'occasion d'approcher diverses techniques : utilisation de ressources JNDI, Spring MVC (contrôleur, formulaire avec validation, mapping, tiles), internationalisation, service et DAO avec JPA.

En plus de **Spring**, la solution  **ORM (Object Relational Mapping ou Mapping objet-relationnel) Hibernate** sera utilisée pour l'accès aux données en **JPA (Java Persistence API)** et **Apache Tiles** sera utilisé pour la mise en place d'un menu unifiant l'application.

 Cet article est dans le même esprit qu'un article sur Struts : **Application Struts pas à pas avec Eclipse (Web Tools Platform) et Tomcat**



Spring (tout comme Struts) permet d'organiser une application Web selon le patron de conception MVC. L'image ci-dessus présente simplement ce patron de conception. Les différents éléments sont :

- Dispatcher Servlet : cette partie est fournie par Spring. La  **Servlet** reçoit les requêtes HTTP et dirige le traitement vers le contrôleur correspondant à l'URL de la requête (mapping) ;
- Modèle : il contient la partie « métier ». Dans notre cas, il sera implémenté par des **DAO (Data Access Object ou Objet d'accès aux données)** ;
- Vue : c'est la partie d'interface avec l'utilisateur. Dans notre cas, il s'agit des  **JSP (JavaServer Pages)** ;
- Contrôleur : il a la charge du choix du traitement à déclencher et des informations à afficher en fonction des entrées.

### I-B - Prérequis

Il est important d'avoir des notions sur Java et les applications Web. Par ailleurs, l'article n'expliquera pas le fonctionnement ou la configuration d'Eclipse, Tomcat, Maven ou Hibernate.

## I-C - Ressources

En complément à cet article, vous pouvez trouver des informations dans la **documentation officielle de Spring** ou dans la série d'articles « **Spring MVC par l'exemple** » de Serge Tahé.

## I-D - Versions des logiciels

Les versions des principaux logiciels et bibliothèques utilisés dans cet article sont :

- **Eclipse** : Kepler (4.3.2) SR2 for Java EE Developers ;
- **Java** : JDK 7 ;
- **Tomcat** : 7.0 ;
- **Spring** : 4.0 ;
- **Hibernate** : 4.3 ;
- **HSQLDB (HyperSQL DataBase)** : 2.3 ;
- **Tiles** : 3.0.

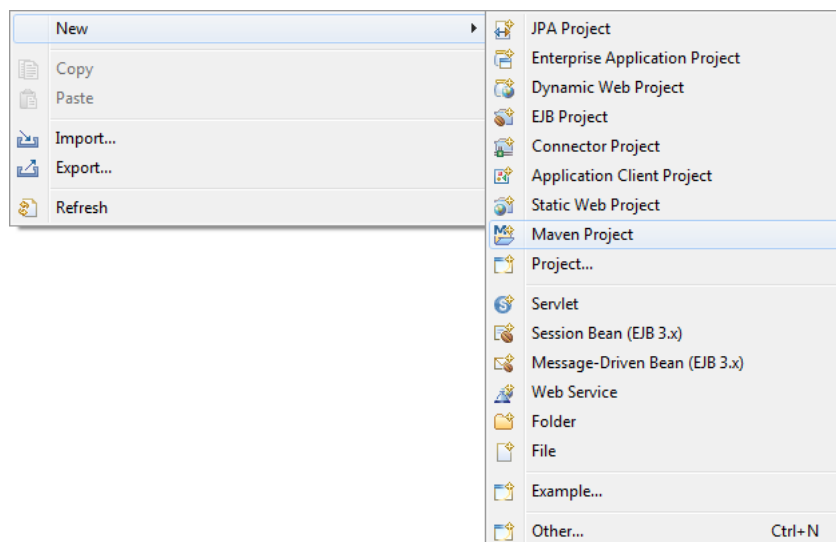
## II - Première application : Bonjour le monde !

Dans ce chapitre, nous allons créer l'application, la déployer, créer une première page Web (le classique « Bonjour le monde ») puis nous modifierons cette page afin de rendre le message personnalisé en fonction de valeurs dans l'URL.

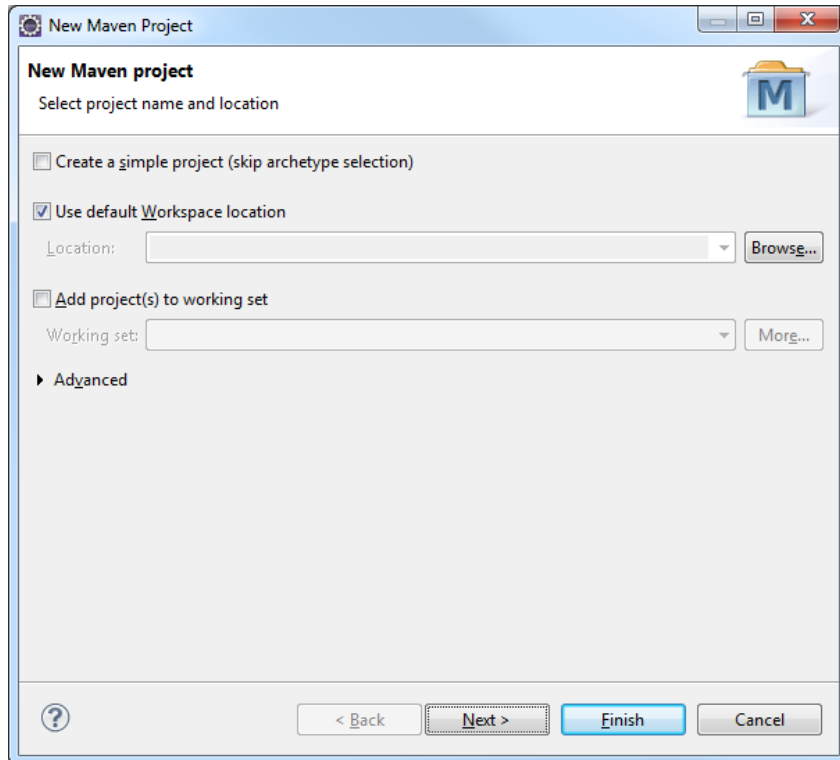
### II-A - Création de l'application

Pour le moment, nous allons créer une application Web avec Maven.

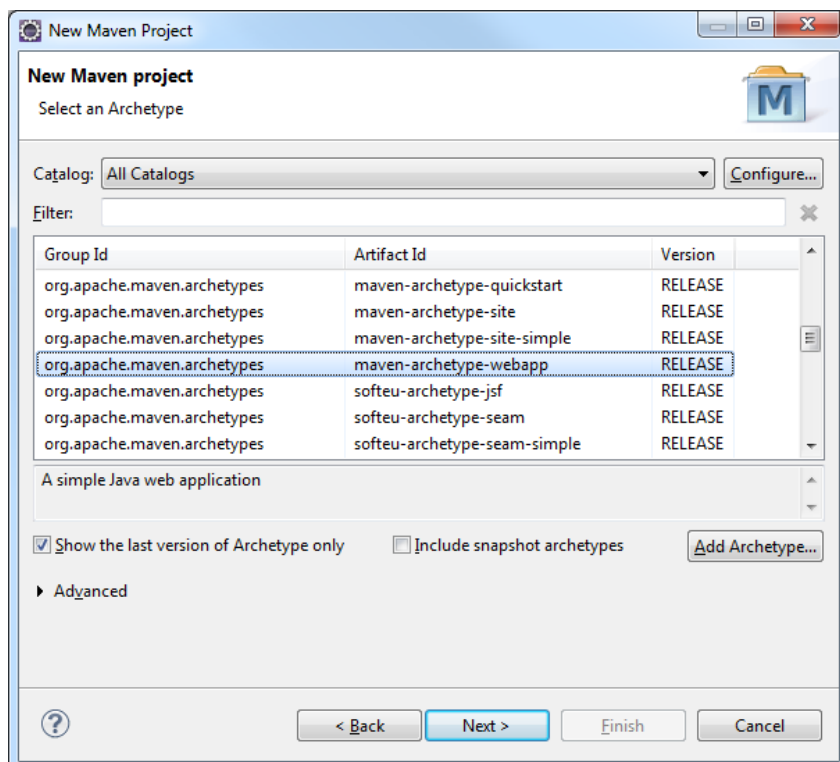
Choisir « Maven Project » dans le menu de création.



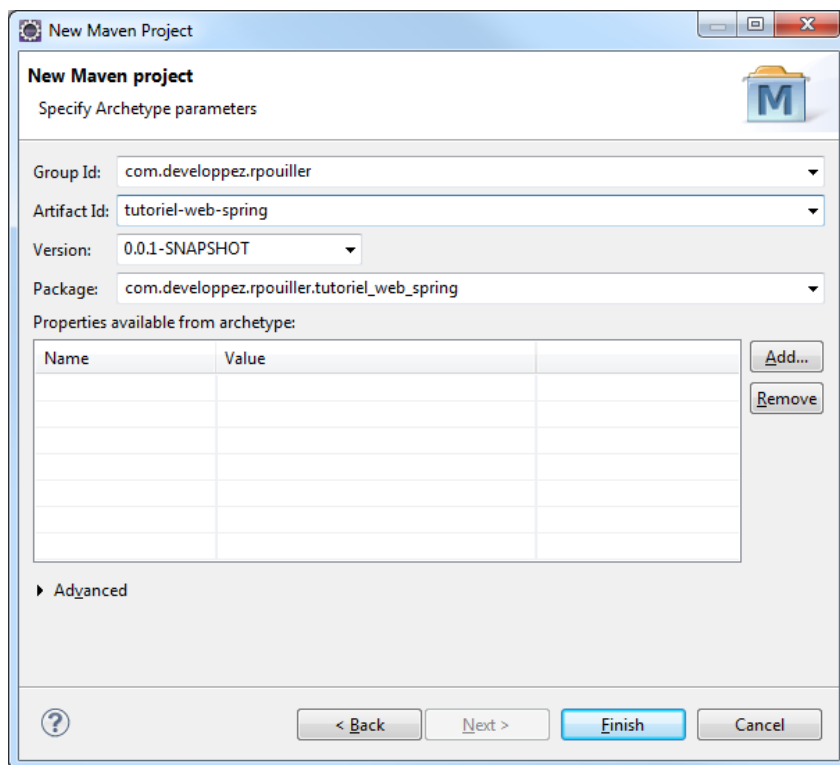
Laisser les valeurs par défaut et cliquer que le bouton « **Next >** ».



Choisir l'archetype « `maven-archetype-webapp` » et cliquer sur le bouton « `Next >` ».



Entrer les valeurs « `Group Id` » et « `Artifact Id` » ; puis cliquer sur le bouton « `Finish` ».



Modifier le fichier « pom.xml » afin qu'il contienne la dépendance vers « servlet-api » :

```

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.developpez.rpouiller</groupId>
  <artifactId>tutoriel-web-spring</artifactId>
  <packaging>war</packaging>
  <version>0.0.1-SNAPSHOT</version>

  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.5</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</project>

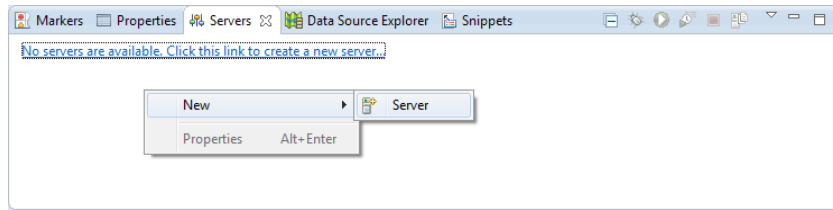
```

Nous avons maintenant une application Web (pour le moment un peu vide) qui compile.

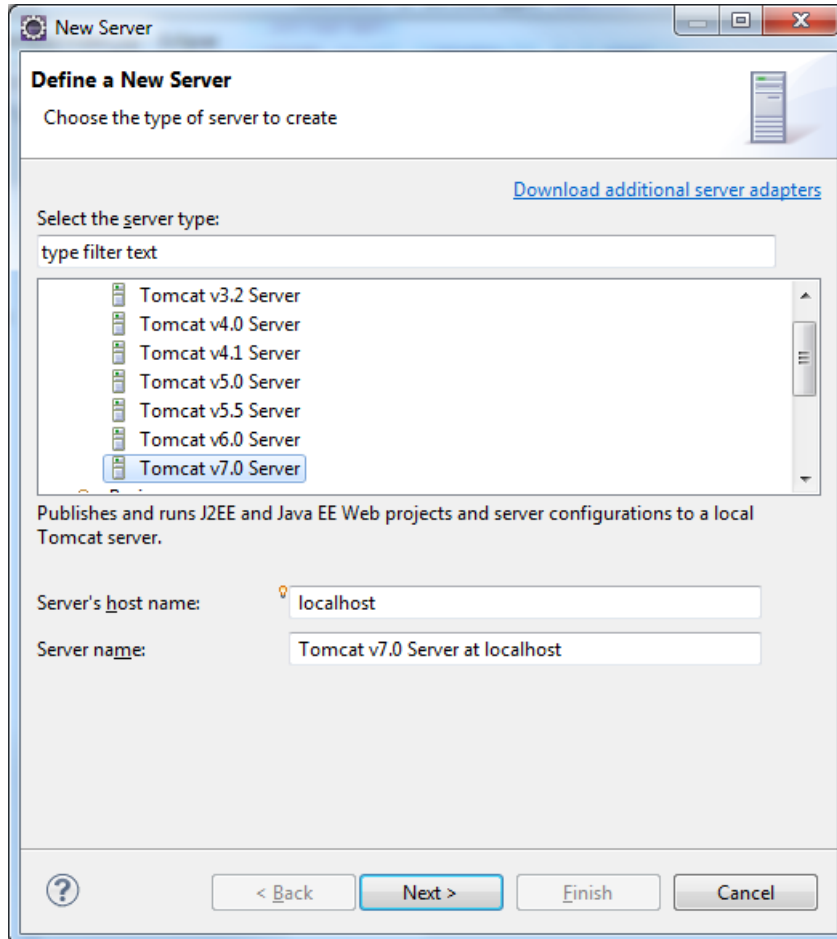
## II-B - Déploiement de l'application

Nous allons maintenant voir comment déployer cette application dans un environnement de développement.

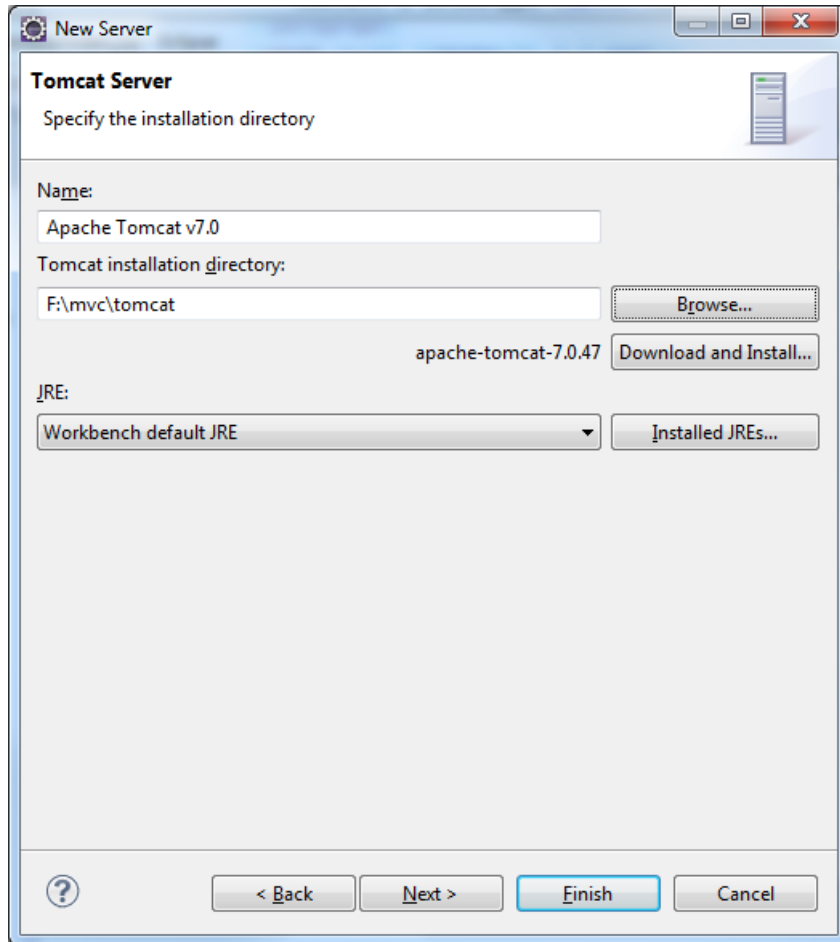
Faire un clic droit dans la vue « Servers » et choisir « New » / « Server ».



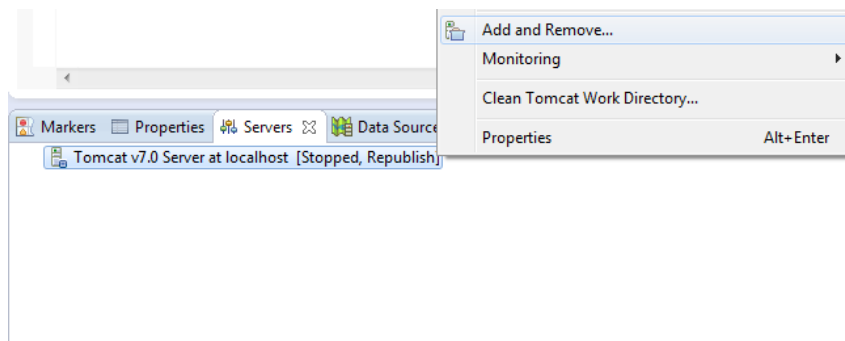
Choisir « Tomcat v7.0 Server » et cliquer sur « Next > ».



Saisir un dossier où Tomcat est installé et cliquer sur « Finish ».

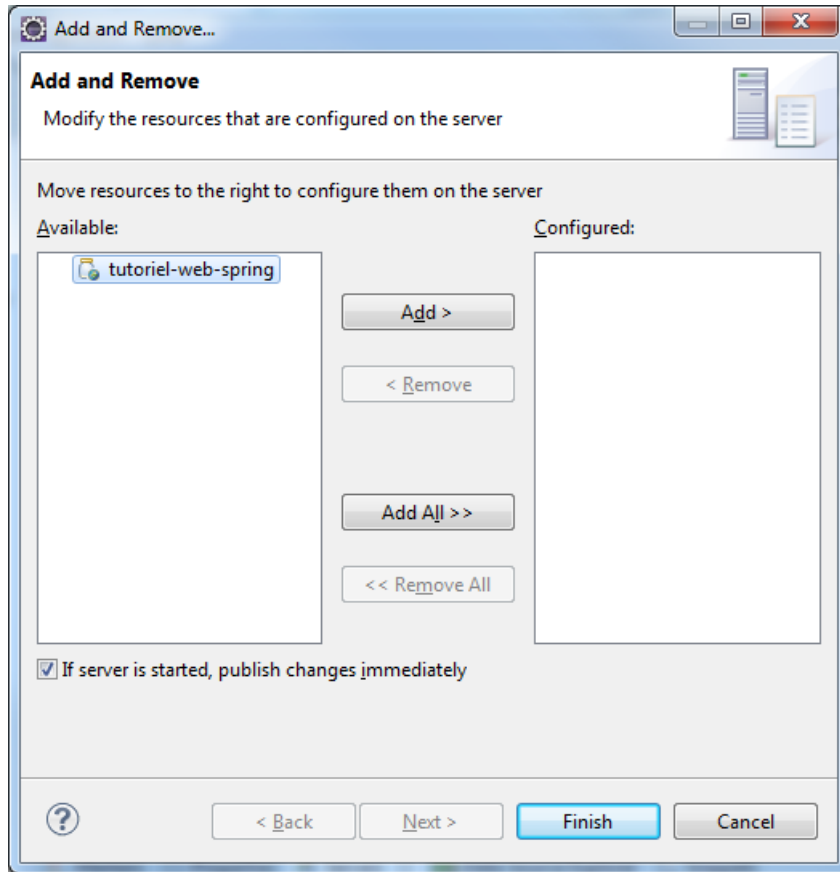


Une fois le serveur créé, il faut faire un clic droit sur le serveur afin de choisir « Add and Remove... ».

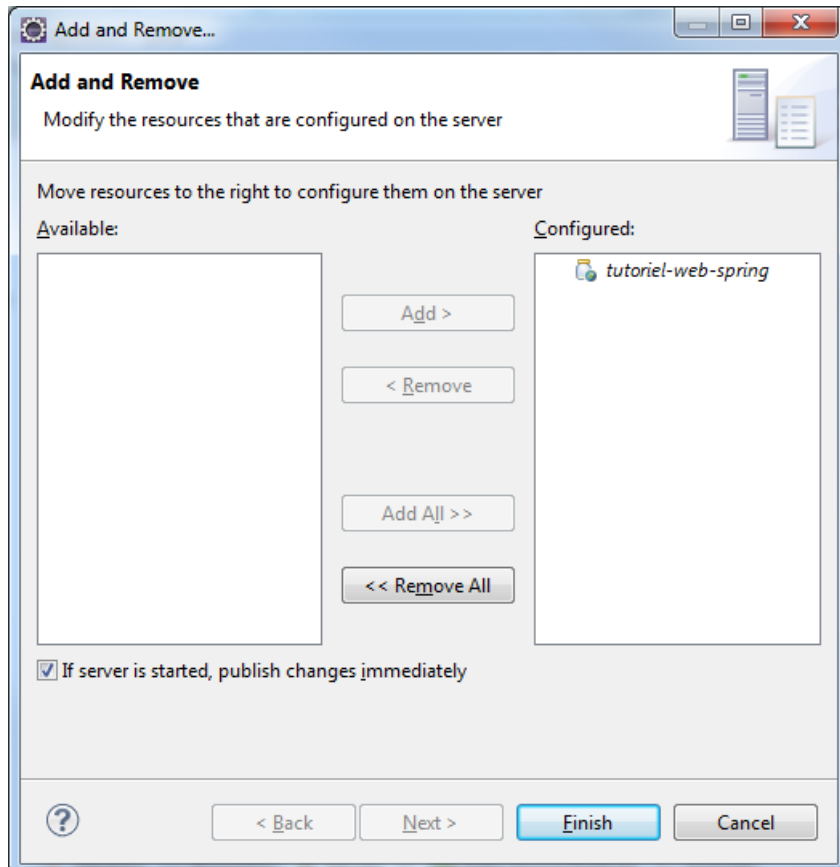


Sélectionner le projet et cliquer sur « Add > » afin de l'ajouter dans le colonne de droite.

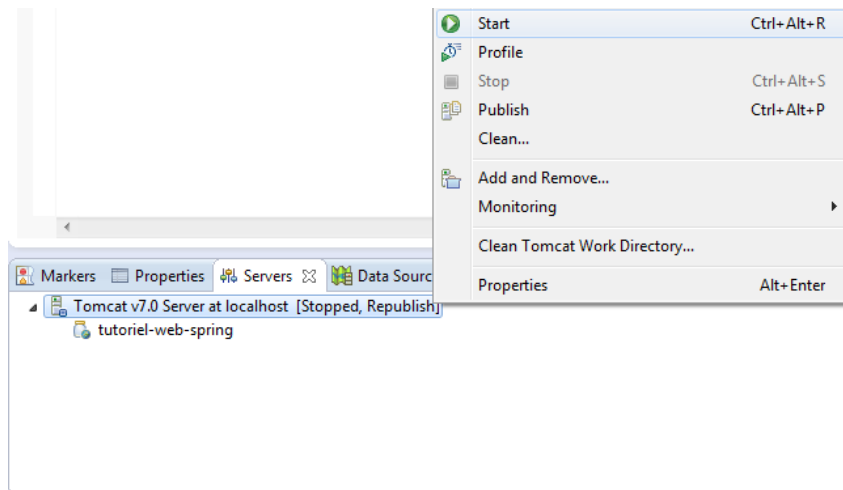




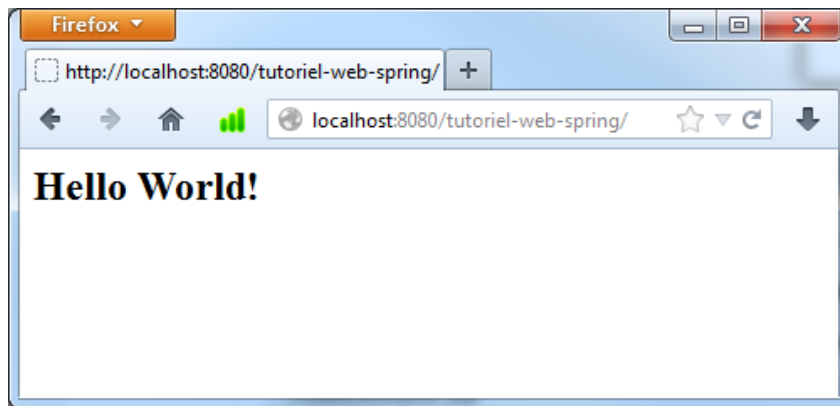
Cliquer sur le bouton « Finish ».



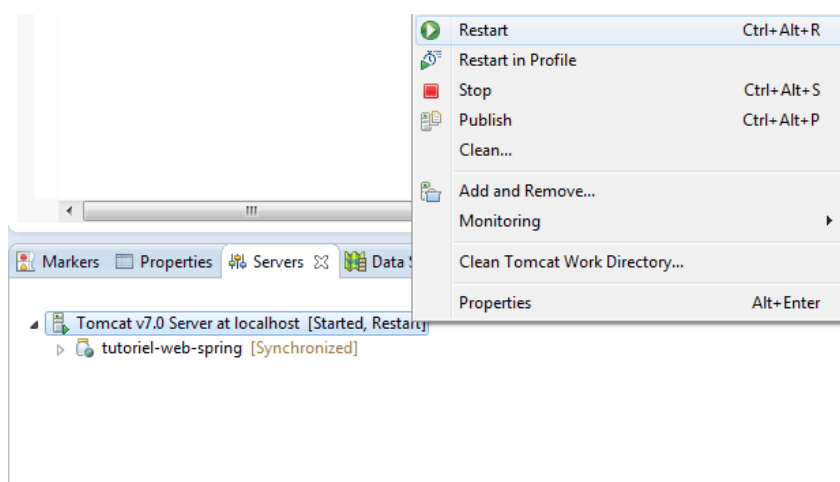
Faire un clic droit sur le serveur et choisir « Start ».



Après avoir démarré, ouvrir un navigateur Web à l'adresse : <http://localhost:8080/tutoriel-web-spring/>.



Pour relancer le déploiement, faire un clic droit sur le serveur et choisir « Restart ».



À partir de maintenant nous savons déployer et redéployer notre application sur le serveur déclaré dans Eclipse.

## II-C - Création d'une JSP « bonjour.jsp »

Nous allons créer une JSP simple sans utilisation de Spring.

Supprimer le fichier « `index.jsp` » contenu dans le dossier « `src/main/webapp` ».

Cette étape consiste simplement à ajouter une JSP dans le projet Web. Pour le moment, il n'y a aucun lien avec Spring.

Créer un dossier « `vues` » dans le dossier « `src/main/webapp` ».

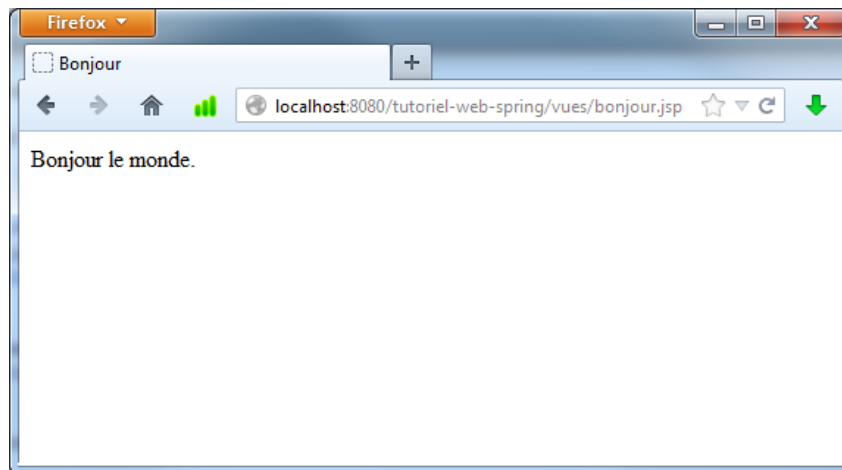
Créer un fichier « `bonjour.jsp` » (avec le contenu ci-dessous) dans ce nouveau dossier.

```

/vues/bonjour.jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
  <head>
    <title>Bonjour</title>
  </head>
  <body>
    Bonjour le monde.
  </body>
</html>

```

Après avoir déployé le projet dans le serveur Tomcat, ouvrir un navigateur Web à l'adresse : <http://localhost:8080/tutoriel-web-spring/vues/bonjour.jsp>.



Nous avons su créer une JSP (correspondant à la partie vue du patron de conception MVC).

## II-D - Utilisation de Spring avec la JSP « bonjour.jsp »

Dans ce paragraphe, nous allons modifier l'application afin d'intégrer Spring. Cela reste une utilisation très simple de Spring car elle se limite à l'utilisation des fichiers d'internationalisation. Toutefois, cela valide le déploiement correct des dépendances Maven ainsi que le chargement du fichier de configuration Spring.

Ajouter la dépendance vers « `spring-webmvc` » dans le fichier « `pom.xml` ».

**pom.xml**

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-webmvc</artifactId>
  <version>4.0.2.RELEASE</version>
</dependency>
```

Modifier le fichier « web.xml » comme ci-dessous. Le listener « ContextLoaderListener » charge la configuration Spring à partir de la variable de contexte « contextConfigLocation ».

**/WEB-INF/web.xml**

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/dispatcher-servlet.xml</param-value>
  </context-param>

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

Créer le fichier « dispatcher-servlet.xml » dans « WEB-INF » comme ci-dessous. Le bean « messageSource » de classe « **ReloadableResourceBundleMessageSource** » (depuis Spring 1.0) est déclaré. Il permettra de charger les messages internationalisés dans des fichiers « messages\_xx.properties » contenus à la racine du classpath.

**/WEB-INF/dispatcher-servlet.xml**

```
<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"
  xmlns:mvc="http://www.springframework.org/schema/mvc"
  xmlns:tx="http://www.springframework.org/schema/tx"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.0.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd
http://www.springframework.org/schema/mvc
http://www.springframework.org/schema/mvc/spring-mvc.xsd
http://www.springframework.org/schema/tx http://www.springframework.org/schema/tx/spring-
tx-3.0.xsd">

  <bean id="messageSource"
    class="org.springframework.context.support.ReloadableResourceBundleMessageSource">
    <property name="basename" value="classpath:messages" />
    <property name="defaultEncoding" value="ISO-8859-1" />
  </bean>
</beans>
```

Créer le fichier « messages\_fr.properties » dans « src/main/resources » comme ci-dessous. Le dossier peut également contenir, par exemple, des fichiers « messages\_en.properties » pour la langue anglaise ou « messages.properties » pour les messages par défaut (quand il n'y a pas de fichier correspondant à la langue).

**messages\_fr.properties**

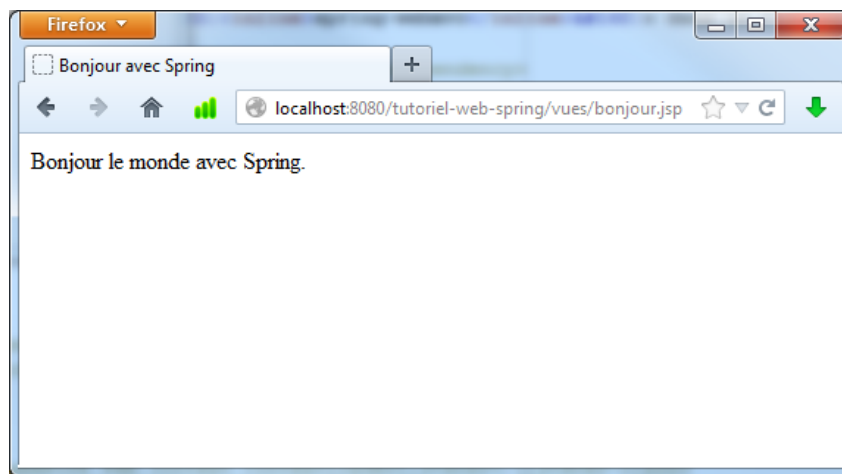
```
titre.bonjour=Bonjour avec Spring
libelle.bonjour.lemonde=Bonjour le monde avec Spring.
```

Modifier le fichier « bonjour.jsp » comme ci-dessous. On a ajouté la déclaration de la taglib « spring » et les utilisations du tag « spring:message ».

```
/vues/bonjour.jsp
```

```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
  <head>
    <title><spring:message code="titre.bonjour"/></title>
  </head>
  <body>
    <spring:message code="libelle.bonjour.lemonde"/>
  </body>
</html>
```

Après déploiement, on obtient le résultat ci-dessous à l'adresse : <http://localhost:8080/tutoriel-web-spring/vues/bonjour.jsp>



Nous avons utilisé le système d'internationalisation de Spring et vérifié ainsi que les dépendances s'étaient déployées correctement.



## II-E - Passage de donnée à la JSP « `bonjour.jsp` »

Nous allons maintenant modifier cela afin que la JSP affiche le texte en fonction d'une donnée. Pour cela, nous ajoutons un contrôleur qui transmettra une donnée à la JSP.

Pour avoir une présentation de Spring MVC, vous pouvez lire l'article :



- « *Spring MVC par l'exemple* » de *Serge Tahé*.

Ajouter la déclaration de la  **servlet** « `DispatcherServlet` » dans le fichier « `web.xml` » comme ci-dessous. C'est la  **servlet** qui redirigera les requêtes HTTP vers les contrôleurs qui conviennent (elle est équivalente à la « `ActionServlet` » de Struts).

```
/WEB-INF/web.xml
```

```
<!-- Declaration de la servlet de Spring et de son mapping -->
<servlet>
  <servlet-name>servlet-dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
  <init-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/dispatcher-servlet.xml</param-value>
```

```
/WEB-INF/web.xml
```

```

    </init-param>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>servlet-dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
    
```

Voici la classe « BonjourController ». Elle est déclarée en tant que contrôleur grâce à l'annotation « **@Controller** » (elle existe depuis Spring 2.5). L'annotation « **@RequestMapping** » (elle existe depuis Spring 2.5) indique que le contrôleur traite les requêtes GET dont l'URI est « /bonjour ». On constate que la valeur « Regis » est associée à l'attribut « personne » grâce à la méthode « **addAttribute** » de « **ModelMap** ». Ensuite, le contrôleur redirige vers la ressource « bonjour ».

```
BonjourController.java
```

```

package com.developpez.rpouiller.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

@Controller
@RequestMapping("/bonjour")
public class BonjourController {

    @RequestMapping(method = RequestMethod.GET)
    public String afficherBonjour(final ModelMap pModel) {
        pModel.addAttribute("personne", "Regis");
        return "bonjour";
    }
}
    
```

Dans le fichier « dispatcher-servlet.xml » il faut ajouter les lignes ci-dessous. « **component-scan** » active la configuration par annotations. La déclaration du bean « **InternalResourceViewResolver** » (qui existe depuis Spring 1.0) permet d'indiquer où chercher les ressources (ici la ressource « bonjour » indiquée dans le contrôleur sera cherchée avec l'extension « .jsp » dans le dossier « /vues/ ».

```
/WEB-INF/dispatcher-servlet.xml
```

```

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

<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">
        <value>/vues/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>
    
```

Modifier la valeur ci-dessous dans le fichier « messages\_fr.properties » afin de prendre en compte un paramètre dans le texte.

```
messages_fr.properties
```

```
libelle.bonjour.lemonde=Bonjour {0} avec Spring.
```

Modifier le fichier « bonjour.jsp » comme ci-dessous. On a ajouté des Expressions Languages « **\${personne}** » afin de restituer la donnée.

```
/vues/bonjour.jsp
```

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false"
    pageEncoding="ISO-8859-1"%>
    
```

/vues/bonjour.jsp

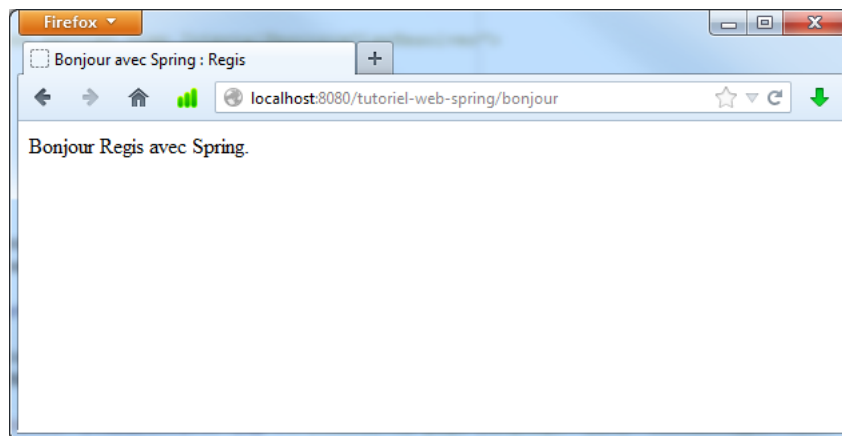
```
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
  <head>
    <title><spring:message code="titre.bonjour"/> : ${personne}</title>
  </head>
  <body>
    <spring:message code="libelle.bonjour.lemonde" arguments="${personne}"/>
  </body>
</html>
```

Pour avoir une présentation des Expressions Languages (EL), vous pouvez lire l'article :



- « **Présentation des Expressions Languages** » de Fred Martini.

Après déploiement, on obtient le résultat ci-dessous à l'adresse : <http://localhost:8080/tutoriel-web-spring/bonjour>.



Nous avons passé une valeur dans un attribut depuis le contrôleur que nous avons affiché dans le JSP grâce aux Expressions Languages (EL).

## II-F - Récupération d'un paramètre de la requête HTTP

Dans ce chapitre, nous allons récupérer un paramètre de la requête pour le passer à la place de la valeur de l'attribut.

Dans la classe « `BonjourController` », modifier la méthode « `afficherBonjour` » comme ci-dessous. Le paramètre « `personne` » est récupéré de la requête grâce à l'annotation « `@RequestParam` » (l'annotation existe depuis Spring 2.5 mais l'élément « `defaultValue` » a été rajouté avec Spring 3.0)

BonjourController.java

```
package com.developpez.rpouiller.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
@RequestMapping("/bonjour")
public class BonjourController {
```

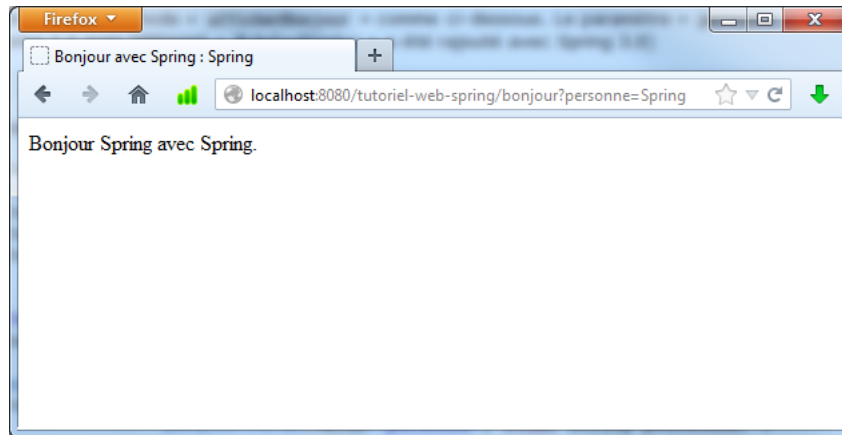
### BonjourController.java

```

@RequestMapping (method = RequestMethod.GET)
public String afficherBonjour (final ModelMap pModel,
    @RequestParam (value="personne") final String pPersonne) {

    pModel.addAttribute ("personne", pPersonne);
    return "bonjour";
}
    
```

Après déploiement, on obtient le résultat ci-dessous à l'adresse : <http://localhost:8080/tutoriel-web-spring/bonjour?personne=Spring>.



Une alternative possible est d'extraire le paramètre depuis l'URI comme dans l'exemple ci-dessous. Cela est possible en indiquant dans l'annotation « **@RequestMapping** » qu'il y a une variable (indiqué avec les accolades). Ensuite, l'annotation « **@PathVariable** » (existant depuis Spring 3.0) permet d'indiquer l'utilisation de cette variable extraite de l'URI.

### BonjourController.java

```

package com.developpez.rpouiller.controller;

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

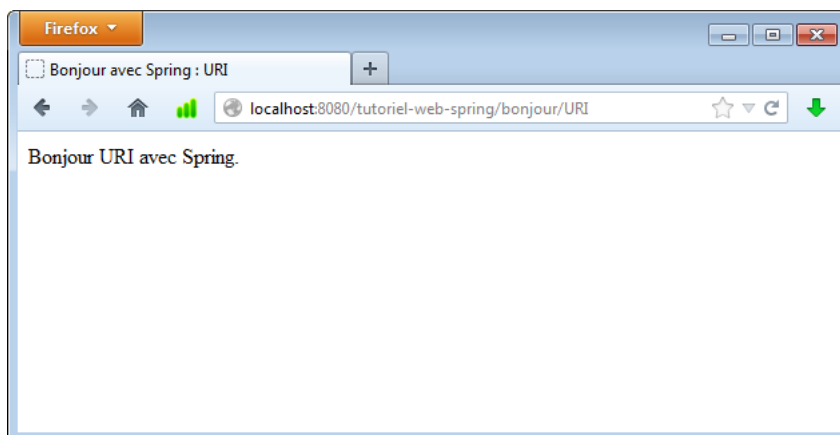
@Controller
@RequestMapping ("/bonjour/{personne}")
public class BonjourController {

    @RequestMapping (method = RequestMethod.GET)
    public String afficherBonjour (final ModelMap pModel,
        @PathVariable (value="personne") final String pPersonne) {

        pModel.addAttribute ("personne", pPersonne);
        return "bonjour";
    }
}
    
```

Après déploiement, on obtient le résultat ci-dessous à l'adresse : <http://localhost:8080/tutoriel-web-spring/bonjour/URI>.





Nous avons vu comment récupérer un paramètre depuis la requête (dans cet exemple, dans une requête GET, donc dans l'URL, mais le principe est le même pour une requête POST) et comment parser une valeur dans l'URI.

### III - Affichage de données en base

Dans ce chapitre, nous allons voir comment récupérer et afficher des valeurs contenues dans une base données.

#### III-A - Création de la base

Pour les besoins de ce tutoriel, nous utiliserons la base de données **HSQLDB (HyperSQL DataBase)**. Dans ce chapitre, nous allons créer la base de données, une table et remplir cette table avec des valeurs de test.

Pour avoir une présentation de HSQLDB, vous pouvez lire l'article :



- « **Présentation et utilisation d'HSQLDB** » de **Baptiste Wicht**.

Il faut donc rajouter la dépendance Maven suivante :

```
Extrait du fichier « pom.xml »
<dependency>
  <groupId>org.hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>2.3.1</version>
</dependency>
```

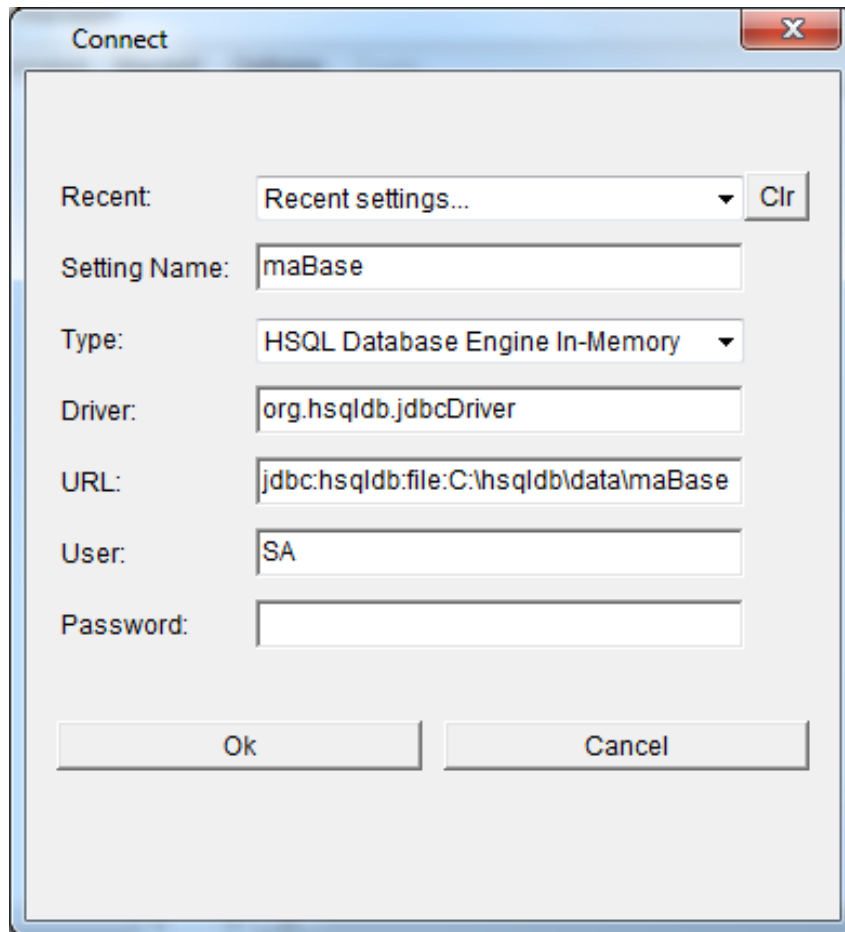
Voici la table qui sera utilisée en exemple dans ce tutoriel :

LISTECOURSES	
•IDOBJET	INTEGER
°LIBELLE	VARCHAR
°QUANTITE	INTEGER

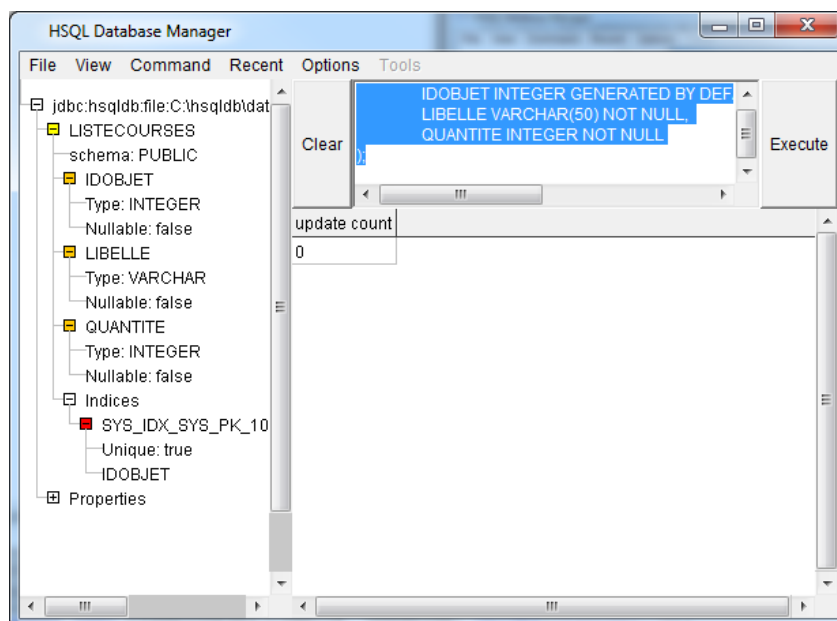
Voici son script SQL de création :

```
Script SQL de création de la table « LISTECOURSES »
CREATE TABLE LISTECOURSES (
  IDOBJET INTEGER GENERATED BY DEFAULT AS IDENTITY(START WITH 1) PRIMARY KEY,
  LIBELLE VARCHAR(50) NOT NULL,
  QUANTITE INTEGER NOT NULL
);
```

Pour lancer le gestionnaire de base HSQLDB, exécuter la classe « org.hsqldb.util.DatabaseManager ». Voici la connexion à la base de données d'exemple (ici avec l'URL « jdbc:hsqldb:file:C:\hsqldb\data\maBase ») :



Voici le résultat de la création de la table :



Script SQL d'insertion de données dans la table « LISTECOURSES »

```
INSERT INTO LISTECOURSES (LIBELLE, QUANTITE) VALUES ('Banane', 3);
```

### Script SQL d'insertion de données dans la table « LISTECOURSES »

```
INSERT INTO LISTECOURSES (LIBELLE, QUANTITE) VALUES ('Sucre blanc', 75);
INSERT INTO LISTECOURSES (LIBELLE, QUANTITE) VALUES ('Oeuf', 1);
INSERT INTO LISTECOURSES (LIBELLE, QUANTITE) VALUES ('Levure', 1);
INSERT INTO LISTECOURSES (LIBELLE, QUANTITE) VALUES ('Sel', 1);
INSERT INTO LISTECOURSES (LIBELLE, QUANTITE) VALUES ('Farine', 150);
INSERT INTO LISTECOURSES (LIBELLE, QUANTITE) VALUES ('Beurre', 70);
```

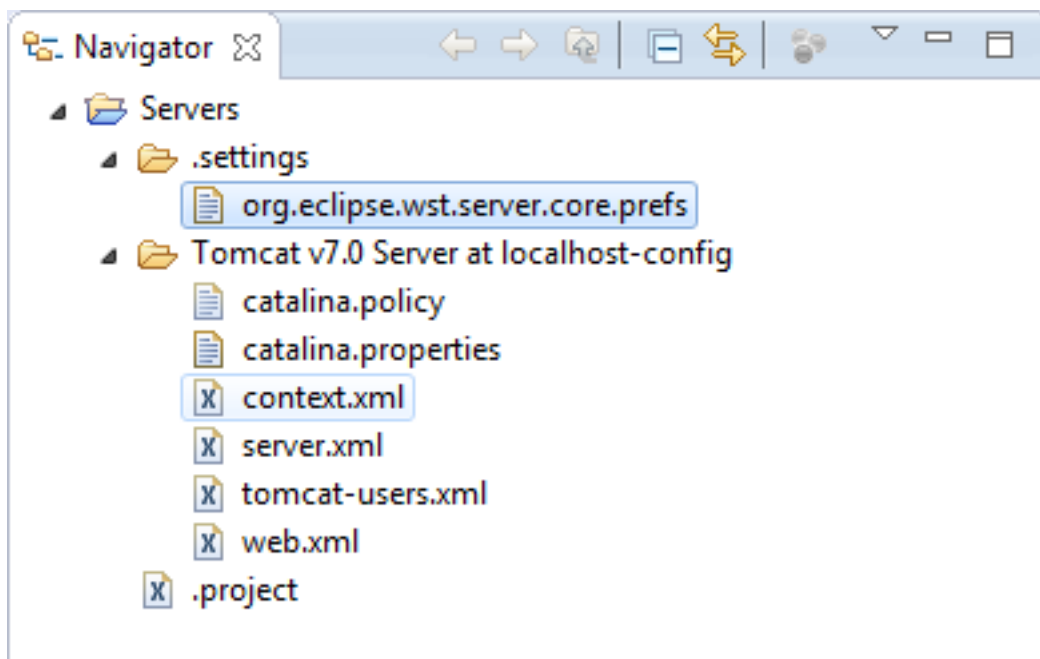
Voici le résultat de l'insertion de données dans la table :

The screenshot shows the HSQL Database Manager interface. The 'LISTECOURSES' table is selected in the left pane. The main window displays the following data:

IDOBJET	LIBELLE	QUANTITE
1	Banane	3
2	Sucre blanc	75
3	Oeuf	1
4	Levure	1
5	Sel	1
6	Farine	150
7	Beurre	70

### III-B - Paramétrage du serveur Tomcat

Nous allons paramétrer la ressource JDBC dans le serveur Tomcat (depuis Eclipse). C'est cette ressource qui est reliée à la base HSQLDB et qui sera utilisée dans l'application.



Il faut ajouter la ressource ci-dessous dans la partie « GlobalNamingResources ».

**server.xml**

```

<GlobalNamingResources>
  ...
  <Resource auth="Container" driverClassName="org.hsqldb.jdbcDriver"
            maxActive="100" maxIdle="30" maxWait="10000" name="jdbc/dsMaBase"
            password="" type="javax.sql.DataSource" url="jdbc:hsqldb:file:C:\hsqldb\data
\maBase"
            username="sa" />
  ...
</GlobalNamingResources>
    
```

Dans le fichier « context.xml », il faut associer la ressource que l'on vient d'indiquer dans le fichier « server.xml » avec le nom « jdbc/dsMonApplication » que l'on utilisera dans l'application.

**context.xml**

```

<ResourceLink name="jdbc/dsMonApplication" global="jdbc/dsMaBase"
              type="javax.sql.DataSource" />
    
```

### III-C - Modification du projet afin d'inclure l'affichage des données

Nous allons modifier le projet afin de réaliser l'affichage des données que l'on vient d'ajouter en base de données.

Il faut rajouter la déclaration de la ressource JDBC de l'application dans le fichier « web.xml ».

**/WEB-INF/web.xml**

```

<!-- Declaration de l'utilisation de la ressource JDBC -->
<resource-ref>
  <description>Ressource JDBC de l'application</description>
  <res-ref-name>jdbc/dsMonApplication</res-ref-name>
  <res-type>javax.sql.DataSource</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
</resource-ref>
    
```

Il faut donc rajouter les dépendances Maven suivantes. Les dépendances « spring-orm » et « hibernate-entitymanager » sont pour l'accès aux données. Tandis que la dépendance « jstl » est pour l'utilisation de la JSTL (Java Standard Tag Library) dans la JSP d'affichage.

**Extrait du fichier « pom.xml »**

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-orm</artifactId>
  <version>4.0.2.RELEASE</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-entitymanager</artifactId>
  <version>4.3.4.Final</version>
</dependency>
<dependency>
  <groupId>javax.servlet</groupId>
  <artifactId>jstl</artifactId>
  <version>1.2</version>
</dependency>
    
```

Le fichier « persistence.xml » ci-dessous permet d'indiquer que la persistance est réalisée grâce à Hibernate.

**src/main/resources/META-INF/persistence.xml**

```

<?xml version="1.0" encoding="UTF-8"?>
<persistence version="2.0"
             xmlns="http://java.sun.com/xml/ns/persistence" xmlns:xsi="http://www.w3.org/2001/XMLSchema-
instance"
    
```

**src/main/resources/META-INF/persistence.xml**

```

xsi:schemaLocation="http://java.sun.com/xml/ns/persistence http://java.sun.com/xml/ns/
persistence/persistence_2_0.xsd">

    <persistence-unit name="unit">
        <provider>org.hibernate.jpa.HibernatePersistenceProvider</provider>
    </persistence-unit>
</persistence>
    
```

Il faut rajouter les lignes ci-dessous dans le fichier « dispatcher-servlet.xml ». Le bean « **JndiObjectFactoryBean** » permet de déclarer l'utilisation de la ressource JDBC. Le bean « **LocalContainerEntityManagerFactoryBean** » utilise la ressource JDBC et le fichier « persistence.xml » pour aboutir à la création du « **EntityManager** » qui est utilisé dans la DAO. Le bean « **JpaTransactionManager** » permet d'instancier le gestionnaire de transaction et lui associer la fabrique de « **EntityManager** ».

**/WEB-INF/dispatcher-servlet.xml**

```

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

<bean id="dataSource" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="java:comp/env/jdbc/dsMonApplication" />
</bean>

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="dataSource" ref="dataSource" />
</bean>

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

La classe « **Course** » est une entité correspondant à la table « LISTECOURSES ».

**Course.java**

```

package com.developpez.rpouiller.bean;

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

@Entity
@Table(name="LISTECOURSES")
public class Course {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name="IDOBJET")
    private Integer id;
    private String libelle;
    private Integer quantite;

    public Integer getId() {
        return id;
    }

    public void setId(final Integer pId) {
        id = pId;
    }

    public String getLibelle() {
        return libelle;
    }

    public void setLibelle(final String pLibelle) {
        libelle = pLibelle;
    }
}
    
```

**Course.java**

```

    }

    public Integer getQuantite() {
        return quantite;
    }

    public void setQuantite(final Integer pQuantite) {
        quantite = pQuantite;
    }
}

```

Voici les interfaces de la DAO et du service : « `IListeCoursesDAO` » et « `IServiceListeCourses` ».

**IListeCoursesDAO.java**

```

package com.developpez.rpouiller.dao;

import java.util.List;

import com.developpez.rpouiller.bean.Course;

public interface IListeCoursesDAO {
    List<Course> rechercherCourses();
}

```

**IServiceListeCourses.java**

```

package com.developpez.rpouiller.services;

import java.util.List;

import com.developpez.rpouiller.bean.Course;

public interface IServiceListeCourses {
    List<Course> rechercherCourses();
}

```

La DAO « `ListeCoursesDAO` » utilise le « `EntityManager` » pour lister les entités « `Course` » contenues dans la base de données.

**ListeCoursesDAO.java**

```

package com.developpez.rpouiller.dao;

import java.util.List;

import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.TypedQuery;
import javax.persistence.criteria.CriteriaBuilder;
import javax.persistence.criteria.CriteriaQuery;
import javax.persistence.criteria.Root;

import org.springframework.stereotype.Repository;

import com.developpez.rpouiller.bean.Course;

@Repository
public class ListeCoursesDAO implements IListeCoursesDAO {

    @PersistenceContext
    private EntityManager entityManager;

    public List<Course> rechercherCourses() {
        final CriteriaBuilder lCriteriaBuilder = entityManager.getCriteriaBuilder();

        final CriteriaQuery<Course> lCriteriaQuery = lCriteriaBuilder.createQuery(Course.class);
        final Root<Course> lRoot = lCriteriaQuery.from(Course.class);
        lCriteriaQuery.select(lRoot);
        final TypedQuery<Course> lTypedQuery = entityManager.createQuery(lCriteriaQuery);
    }
}

```

### ListeCoursesDAO.java

```

        return lTypedQuery.getResultList();
    }
}

```

Pour cette méthode, le service « ServiceListeCourses » sert surtout de passe-plat. Il faut noter toutefois que la transaction est indiquée en lecture seule.

### ServiceListeCourses.java

```

package com.developpez.rpouiller.services;

import java.util.List;

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

import com.developpez.rpouiller.bean.Course;
import com.developpez.rpouiller.dao.IListeCoursesDAO;

@Service
public class ServiceListeCourses implements IServiceListeCourses {

    @Autowired
    private IListeCoursesDAO dao;

    @Transactional(readOnly=true)
    public List<Course> rechercherCourses() {
        return dao.rechercherCourses();
    }
}

```

Le contrôleur « AfficherListeCoursesController » appelle le service et place le résultat dans l'attribut « listeCourses ».

### AfficherListeCoursesController.java

```

package com.developpez.rpouiller.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.developpez.rpouiller.bean.Course;
import com.developpez.rpouiller.services.IServiceListeCourses;

@Controller
@RequestMapping(value="/afficherListeCourses")
public class AfficherListeCoursesController {

    @Autowired
    private IServiceListeCourses service;

    @RequestMapping(method = RequestMethod.GET)
    public String afficher(ModelMap pModel) {
        final List<Course> lListeCourses = service.rechercherCourses();
        pModel.addAttribute("listeCourses", lListeCourses);
        return "listeCourses";
    }
}

```

Il faut ajouter de nouveaux libellés dans le fichier « messages\_fr.properties ».

**messages\_fr.properties**

```
titre.listecourses=Liste de courses
colonne.identifiant=IDOBJET
colonne.libelle=LIBELLE
colonne.quantite=QUANTITE
```

La JSP utilise la Java Standard Tag Library (JSTL) pour afficher le résultat.

**/vues/listeCourses.jsp**

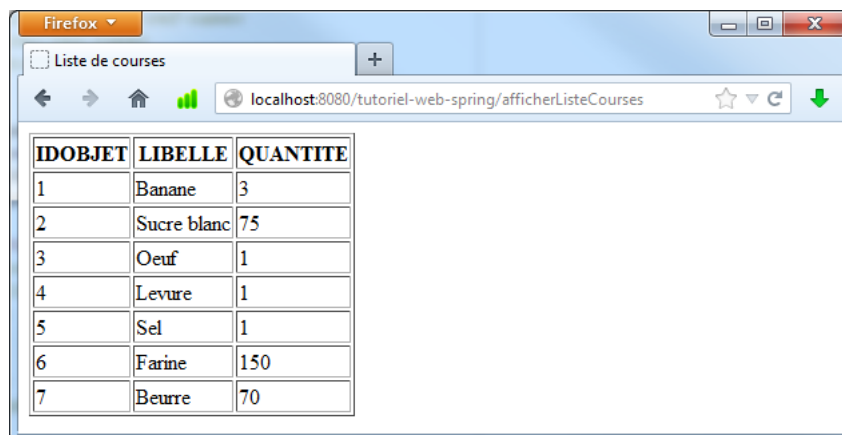
```
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
<title><spring:message code="titre.listecourses"/></title>
</head>
<body>
<table border="1">
<thead>
<tr>
<th><spring:message code="colonne.identifiant"/></th>
<th><spring:message code="colonne.libelle"/></th>
<th><spring:message code="colonne.quantite"/></th>
</tr>
</thead>
<tbody>
<c:forEach items="${listeCourses}" var="course">
<tr>
<td><c:out value="${course.id}"/></td>
<td><c:out value="${course.libelle}"/></td>
<td><c:out value="${course.quantite}"/></td>
</tr>
</c:forEach>
</tbody>
</table>
</body>
</html>
```

Pour avoir une présentation de la Java Standard Tag Library (JSTL), vous pouvez lire l'article :



- « **Présentation de la Java Standard Tag Library (JSTL)** » de **Fred Martini**.

Après déploiement, on obtient le résultat ci-dessous à l'adresse : <http://localhost:8080/tutoriel-web-spring/afficherListeCourses>.



IDOBJET	LIBELLE	QUANTITE
1	Banane	3
2	Sucre blanc	75
3	Oeuf	1
4	Levure	1
5	Sel	1
6	Farine	150
7	Beurre	70



## IV - Création de données en base

Dans ce paragraphe, nous allons créer de nouvelles données dans la base de données. Pour cela, nous aurons besoin d'un formulaire que sera validé pour vérifier les données le constituant.

Il faut donc rajouter les dépendances Maven suivantes. Les dépendances « `validation-api` » et « `hibernate-validator` » permettent la validation du formulaire.

Extrait du fichier « `pom.xml` »

```
<dependency>
  <groupId>javax.validation</groupId>
  <artifactId>validation-api</artifactId>
  <version>1.1.0.Final</version>
</dependency>
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-validator</artifactId>
  <version>5.0.1.Final</version>
</dependency>
```

Il faut rajouter la ligne ci-dessous dans le fichier « `dispatcher-servlet.xml` ». Elle permet d'activer les annotations de validation de formulaire.

`/WEB-INF/dispatcher-servlet.xml`

```
<mvc:annotation-driven />
```

Il faut ajouter de nouveaux libellés dans le fichier « `messages_fr.properties` ».

`messages_fr.properties`

```
titre.creation.elementcourses=Création d'élément de la liste de courses
creation.elementcourses.libelle.libelle=Libellé
creation.elementcourses.libelle.quantite=Quantité

NotEmpty.creation.libelle=Le libellé est nécessaire.
NotEmpty.creation.quantite=La quantité est nécessaire.
Pattern.creation.quantite=La quantité doit être numérique et positive.
```

Le formulaire « `CreationForm` » utilise les annotations « `NotEmpty` » et « `Pattern` » pour indiquer les contraintes de validation.

`CreationForm.java`

```
package com.developpez.rpouiller.controller;

import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.NotEmpty;

public class CreationForm {

    @NotEmpty
    private String libelle;
    @NotEmpty
    @Pattern(regexp="\\d*")
    private String quantite;

    public String getLibelle() {
        return libelle;
    }

    public void setLibelle(final String pLibelle) {
        libelle = pLibelle;
    }
}
```

**CreationForm.java**

```

public String getQuantite() {
    return quantite;
}

public void setQuantite(final String pQuantite) {
    quantite = pQuantite;
}
}
    
```

Dans l'interface « `IListeCoursesDAO` » de la DAO, nous ajoutons la méthode recevant une entité en paramètre.

**IListeCoursesDAO.java**

```

void creerCourse(final Course pCourse);
    
```

Tandis que dans l'interface « `IServiceListeCourses` » du service, nous ajoutons la méthode recevant en paramètres les valeurs de l'entité.

**IServiceListeCourses.java**

```

void creerCourse(final String pLibelle, final Integer pQuantite);
    
```

L'implémentation de la méthode de la DAO « `ListeCoursesDAO` » sauve la nouvelle entité en base.

**ListeCoursesDAO.java**

```

public void creerCourse(final Course pCourse) {
    entityManager.persist(pCourse);
}
    
```

L'implémentation de la méthode du service « `ServiceListeCourses` » constitue l'entité et l'envoi en paramètre à la DAO. Cette fois-ci la transaction n'est pas en lecture seule.

**ServiceListeCourses.java**

```

@Transactional
public void creerCourse(final String pLibelle, final Integer pQuantite) {
    final Course lCourse = new Course();
    lCourse.setLibelle(pLibelle);
    lCourse.setQuantite(pQuantite);

    dao.creerCourse(lCourse);
}
    
```

Le contrôleur « `CreerListeCoursesController` » comporte deux méthodes « `afficher` » et « `creer` ». La méthode « `afficher` » place la liste des courses dans l'attribut « `listeCourses` » et initialise le formulaire « `creation` » s'il n'est pas déjà présent dans l'attribut « `creation` ». L'annotation « `@ModelAttribute` » (existant depuis Spring 2.5) de la méthode « `creer` » indique que le paramètre « `pCreation` » est constitué à partir de l'attribut « `creation` ». L'annotation « `@Valid` » indique que le formulaire doit être validé grâce aux annotations contenues dans la classe de formulaire « `CreationForm` ». Ensuite, la méthode « `creer` » appelle la méthode de création en base de données s'il n'y a pas d'erreurs dans la validation, puis appelle simplement la méthode « `afficher` » pour l'affichage de la page.

Les messages d'erreur sont présents dans le fichier « `messages_fr.properties` ». On peut remarquer que dans ce cas les clés des messages sont de la forme « `contrainteValidation.nomAttribut.nomChamp` » (par exemple « `NotEmpty.creation.libelle` »).

**CreerListeCoursesController.java**

```

package com.developpez.rpouiller.controller;

import java.util.List;

import javax.validation.Valid;

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

### CreerListeCoursesController.java

```

import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.developpez.rpouiller.bean.Course;
import com.developpez.rpouiller.services.IServiceListeCourses;

@Controller
public class CreerListeCoursesController {

    @Autowired
    private IServiceListeCourses service;

    @RequestMapping(value="/afficherCreationListeCourses", method = RequestMethod.GET)
    public String afficher(final ModelMap pModel) {
        final List<Course> lListeCourses = service.rechercherCourses();
        pModel.addAttribute("listeCourses", lListeCourses);
        if (pModel.get("creation") == null) {
            pModel.addAttribute("creation", new CreationForm());
        }
        return "creation";
    }

    @RequestMapping(value="/creerCreationListeCourses", method = RequestMethod.POST)
    public String creer(@Valid @ModelAttribute(value="creation") final CreationForm pCreation,
        final BindingResult pBindingResult, final ModelMap pModel) {

        if (!pBindingResult.hasErrors()) {
            final Integer lIntQuantite = Integer.valueOf(pCreation.getQuantite());
            service.creerCourse(pCreation.getLibelle(), lIntQuantite);
        }
        return afficher(pModel);
    }
}

```

La JSP comporte le formulaire de création d'une nouvelle « course ». Ce formulaire contient également l'affichage des messages d'erreur.

### /vues/creation.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
    loose.dtd">
<html>
<head>
    <title><spring:message code="titre.creation.elementcourses"/></title>
</head>
<body>
    <form:form method="post" modelAttribute="creation" action="creerCreationListeCourses">
        <spring:message code="creation.elementcourses.libelle.libelle" />
        <form:input path="libelle"/>
        <b><i><form:errors path="libelle" cssclass="error"/></i></b><br>
        <spring:message code="creation.elementcourses.libelle.quantite"/>
        <form:input path="quantite"/>
        <b><i><form:errors path="quantite" cssclass="error"/></i></b><br>
        <input type="submit"/>
    </form:form>
    <table border="1">
        <thead>
            <tr>
                <th><spring:message code="colonne.identifiant"/></th>
                <th><spring:message code="colonne.libelle"/></th>
                <th><spring:message code="colonne.quantite"/></th>
            </tr>
        </thead>
    </table>

```

/vues/creation.jsp

```

        </tr>
    </thead>
    <tbody>
        <c:forEach items="${listeCourses}" var="course">
            <tr>
                <td><c:out value="${course.id}"/></td>
                <td><c:out value="${course.libelle}"/></td>
                <td><c:out value="${course.quantite}"/></td>
            </tr>
        </c:forEach>
    </tbody>
</table>
</body>
</html>

```

Après déploiement, on obtient le résultat ci-dessous à l'adresse : <http://localhost:8080/tutoriel-web-spring/afficherCreationListeCourses>.

Création d'élément de la liste de courses

localhost:8080/tutoriel-web-spring/afficherCreationListeCourses

Libellé

Quantité

Envoyer

IDOBJET	LIBELLE	QUANTITE
1	Banane	3
2	Sucre blanc	75
3	Oeuf	1
4	Levure	1
5	Sel	1
6	Farine	150
7	Beurre	70

Voici le résultat de la validation lorsque les champs ne sont pas renseignés.

Création d'élément de la liste de courses

localhost:8080/tutoriel-web-spring/creerCreationListeCourses

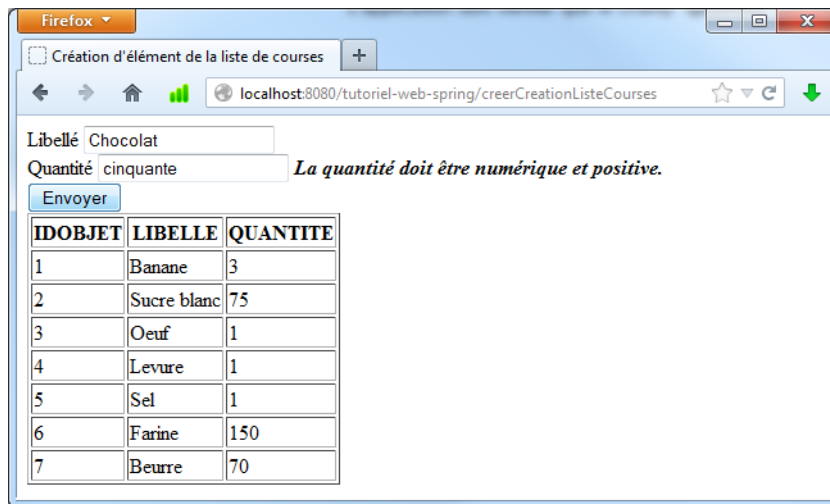
Libellé  *Le libellé est nécessaire.*

Quantité  *La quantité est nécessaire.*

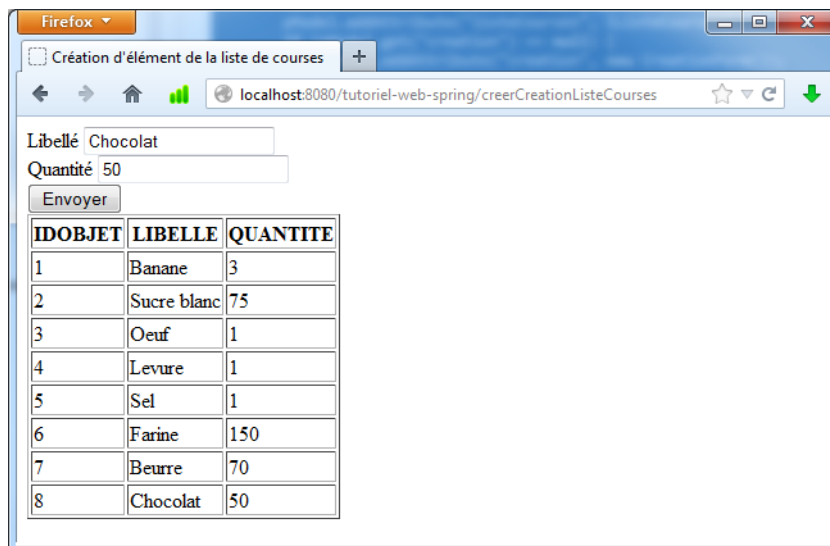
Envoyer

IDOBJET	LIBELLE	QUANTITE
1	Banane	3
2	Sucre blanc	75
3	Oeuf	1
4	Levure	1
5	Sel	1
6	Farine	150
7	Beurre	70

Et le résultat de la validation lorsque le champ « quantité » n'est pas numérique.



Lorsque le formulaire respecte les contraintes de la validation, l'occurrence est créée en base puis est visible au rafraîchissement de la page.



## V - Suppression de données en base

Dans ce paragraphe, nous allons supprimer des données dans la base de données. Pour cela, nous allons créer des liens avec paramètre qui permettront de déterminer l'occurrence à supprimer.

Il faut ajouter de nouveaux libellés dans le fichier « messages\_fr.properties ».

messages\_fr.properties

```
titre.suppression.elementcourses=Suppression d'élément de la liste de courses
suppression.supprimer.libelle=Supprimer
```

Dans l'interface « IListeCoursesDAO » de la DAO, nous ajoutons la méthode de suppression recevant une entité en paramètre.

IListeCoursesDAO.java

```
void supprimerCourse(final Course pCourse);
```

Tandis que dans l'interface « `IServiceListeCourses` » du service, nous ajoutons la méthode recevant en paramètre l'identifiant de l'entité.

```
IServiceListeCourses.java
void supprimerCourse(final Integer pIdCourse);
```

L'implémentation de la méthode de la DAO « `ListeCoursesDAO` » supprime l'entité en base.

```
ListeCoursesDAO.java
public void supprimerCourse(final Course pCourse) {
    final Course lCourse = entityManager.getReference(Course.class, pCourse.getId());
    entityManager.remove(lCourse);
}
```

L'implémentation de la méthode du service « `ServiceListeCourses` » instancie une entité avec l'identifiant et l'envoi en paramètre à la DAO.

```
ServiceListeCourses.java
@Transactional
public void supprimerCourse(final Integer pIdCourse) {
    final Course lCourse = new Course();
    lCourse.setId(pIdCourse);

    dao.supprimerCourse(lCourse);
}
```

Le contrôleur « `SupprimerListeCoursesController` » comporte deux méthodes « `afficher` » et « `supprimer` ». La méthode « `afficher` » place la liste des courses dans l'attribut « `listeCourses` ». Ensuite, la méthode « `supprimer` » utilise le paramètre « `idCourse` » de la requête pour appeler la méthode « `supprimerCourse` » du service, puis elle relance l'affichage de la page.

```
SupprimerListeCoursesController.java
package com.developpez.rpouiller.controller;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;

import com.developpez.rpouiller.bean.Course;
import com.developpez.rpouiller.services.IServiceListeCourses;

@Controller
public class SupprimerListeCoursesController {

    @Autowired
    private IServiceListeCourses service;

    @RequestMapping(value="/afficherSuppressionListeCourses", method = RequestMethod.GET)
    public String afficher(final ModelMap pModel) {
        final List<Course> lListeCourses = service.rechercherCourses();
        pModel.addAttribute("listeCourses", lListeCourses);
        return "suppression";
    }

    @RequestMapping(value="/supprimerSuppressionListeCourses", method = RequestMethod.GET)
    public String supprimer(@RequestParam(value="idCourse") final Integer pIdCourse, final
    ModelMap pModel) {

        service.supprimerCourse(pIdCourse);
        return afficher(pModel);
    }
}
```

**SupprimerListeCoursesController.java**

```

    }
}

```

La JSP affiche un tableau des différents éléments de la liste de courses. Ce tableau comporte une colonne avec un lien pour supprimer.

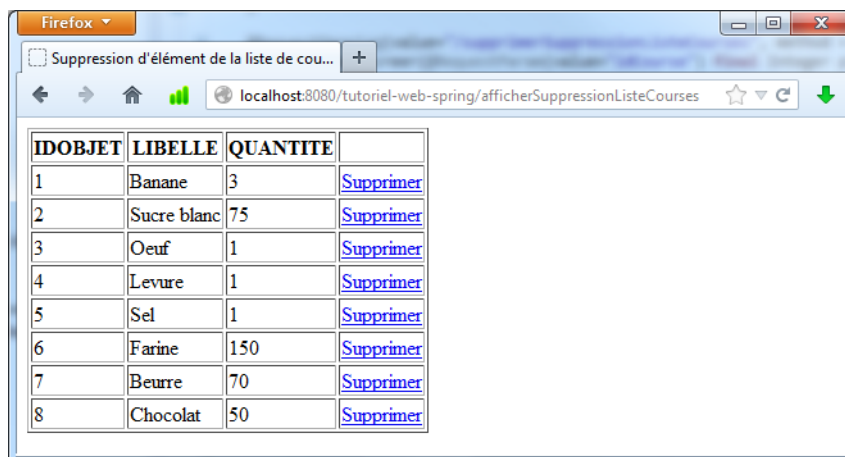
**/vues/suppression.jsp**

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
<title><spring:message code="titre.suppression.elementcourses"/></title>
</head>
<body>
<table border="1">
<thead>
<tr>
<th><spring:message code="colonne.identifiant"/></th>
<th><spring:message code="colonne.libelle"/></th>
<th><spring:message code="colonne.quantite"/></th>
<th>&nbsp;</th>
</tr>
</thead>
<tbody>
<c:forEach items="${listeCourses}" var="course">
<tr>
<td><c:out value="${course.id}"/></td>
<td><c:out value="${course.libelle}"/></td>
<td><c:out value="${course.quantite}"/></td>
<td>
<c:url value="/supprimerSuppressionListeCourses" var="url">
<c:param name="idCourse" value="${course.id}"/>
</c:url>
<a href="${url}">
<spring:message code="suppression.supprimer.libelle" />
</a>
</td>
</tr>
</c:forEach>
</tbody>
</table>
</body>
</html>

```

Après déploiement, on obtient le résultat ci-dessous à l'adresse : <http://localhost:8080/tutoriel-web-spring/afficherSuppressionListeCourses>.



Voici le résultat après avoir cliqué sur le dernier lien de suppression.



## VI - Modification de données en base

Dans ce paragraphe, nous allons modifier des données dans la base de données. Pour cela, nous allons créer un formulaire avec une liste de valeurs qui seront mises à jour dans la base de données.

Il faut ajouter de nouveaux libellés dans le fichier « messages\_fr.properties ».

messages\_fr.properties

```
titre.modification.elementcourses=Modification d'élément de la liste de courses
```

Il faut créer un nouveau fichier d'internationalisation « ValidationMessages\_fr.properties » (également dans le dossier « src/main/resources »).

ValidationMessages\_fr.properties

```
modification.course.quantite.notempty=La quantité est nécessaire.  
modification.course.quantite.numerique=La quantité doit être numérique et positive.
```

Dans l'interface « IListeCoursesDAO » de la DAO, nous ajoutons la méthode de modification recevant une entité en paramètre.

IListeCoursesDAO.java

```
void modifierCourse(final Course pCourse);
```

Tandis que dans l'interface « IServiceListeCourses » du service, nous ajoutons la méthode recevant en paramètres une liste d'entités.



**IServiceListeCourses.java**

```
void modifierCourses (final List<Course> pListeCourses);
```

L'implémentation de la méthode de la DAO « ListeCoursesDAO » supprime l'entité en base. Elle lève une exception uniquement si la requête modifie un nombre d'occurrences différent de 1 (ce qui aura pour conséquence de provoquer un rollback de transaction au niveau du service). Le libellé de l'exception contient le texte de la requête SQL.

*Pour avoir plus d'information sur la manière de récupérer la requête SQL, vous pouvez lire l'article :*



- « [How to get the JPQL/SQL String From a CriteriaQuery in JPA ?](#) » de Antonio Goncalves.

**ListeCoursesDAO.java**

```
public void modifierCourse (final Course pCourse) {
    final CriteriaBuilder lCriteriaBuilder = entityManager.getCriteriaBuilder();

    final CriteriaUpdate<Course> lCriteriaUpdate =
lCriteriaBuilder.createCriteriaUpdate (Course.class);
    final Root<Course> lRoot = lCriteriaUpdate.from (Course.class);
    final Path<Course> lPath = lRoot.get ("id");
    final Expression<Boolean> lExpression = lCriteriaBuilder.equal (lPath, pCourse.getId());
    lCriteriaUpdate.where (lExpression);
    lCriteriaUpdate.set ("quantite", pCourse.getQuantite());
    final Query lQuery = entityManager.createQuery (lCriteriaUpdate);
    final int lRowCount = lQuery.executeUpdate();

    if (lRowCount != 1) {
        final org.hibernate.Query lHQuery = lQuery.unwrap (org.hibernate.Query.class);
        final String lSql = lHQuery.getQueryString();
        throw new RuntimeException ("Nombre d'occurrences (" + lRowCount +
            ") modifiés différent de 1 pour " + lSql);
    }
}
```

L'implémentation de la méthode du service « ServiceListeCourses » parcourt les entités de la liste pour les passer en paramètre l'une après l'autre à la DAO.

**ServiceListeCourses.java**

```
@Transactional
public void modifierCourses (final List<Course> pListeCourses) {
    for (final Course lCourse : pListeCourses) {
        dao.modifierCourse (lCourse);
    }
}
```

Le formulaire « ModificationForm » contient une liste qui comporte l'annotation « **@Valid** ». Cela provoquera la validation de chaque élément de la liste lorsque le formulaire sera validé en entrée de la méthode « modifier » du contrôleur.

**ModificationForm.java**

```
package com.developpez.rpouiller.controller;

import java.util.List;

import javax.validation.Valid;

public class ModificationForm {

    @Valid
    private List<ModificationCourse> listeCourses;
```

### ModificationForm.java

```
public void setListeCourses(final List<ModificationCourse> pListeCourses) {
    listeCourses = pListeCourses;
}

public List<ModificationCourse> getListeCourses() {
    return listeCourses;
}
}
```

La classe « `ModificationCourse` » correspond à un élément de la liste contenue dans le formulaire « `ModificationForm` ». Nous pouvons remarquer que les annotations « `NotEmpty` » et « `Pattern` » comportent des valeurs « `message` » qui correspondent aux messages internationalisés contenus dans le fichier « `ValidationMessages_fr.properties` ».

### ModificationCourse.java

```
package com.developpez.rpouiller.controller;

import javax.validation.constraints.Pattern;
import org.hibernate.validator.constraints.NotEmpty;

public class ModificationCourse {
    private Integer id;
    private String libelle;
    @NotEmpty(message="{modification.course.quantite.notempty}")
    @Pattern(regexp="\\d*", message="{modification.course.quantite.numerique}")
    private String quantite;

    public Integer getId() {
        return id;
    }

    public void setId(final Integer pId) {
        id = pId;
    }

    public String getLibelle() {
        return libelle;
    }

    public void setLibelle(final String pLibelle) {
        libelle = pLibelle;
    }

    public String getQuantite() {
        return quantite;
    }

    public void setQuantite(final String pQuantite) {
        quantite = pQuantite;
    }
}
```

Le contrôleur « `ModifierListeCoursesController` » comporte les méthodes « `afficher` » et « `modifier` ». La méthode « `afficher` » place la liste des courses dans le formulaire et la méthode « `modifier` » récupère la liste des courses du formulaire pour appeler la méthode de modification du service.

### ModifierListeCoursesController.java

```
package com.developpez.rpouiller.controller;

import java.util.LinkedList;
import java.util.List;

import javax.validation.Valid;
```

### ModifierListeCoursesController.java

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;

import com.developpez.rpouiller.bean.Course;
import com.developpez.rpouiller.services.IServiceListeCourses;

@Controller
public class ModifierListeCoursesController {

    @Autowired
    private IServiceListeCourses service;

    @RequestMapping(value="/afficherModificationListeCourses", method = RequestMethod.GET)
    public String afficher(final ModelMap pModel) {
        if (pModel.get("modification") == null) {
            final List<Course> lListeCourses = service.rechercherCourses();
            final ModificationForm lModificationForm = new ModificationForm();
            final List<ModificationCourse> lListe = new LinkedList<ModificationCourse>();
            for (final Course lCourse : lListeCourses) {
                final ModificationCourse lModificationCourse = new ModificationCourse();
                lModificationCourse.setId(lCourse.getId());
                lModificationCourse.setLibelle(lCourse.getLibelle());
                lModificationCourse.setQuantite(lCourse.getQuantite().toString());
                lListe.add(lModificationCourse);
            }
            lModificationForm.setListeCourses(lListe);
            pModel.addAttribute("modification", lModificationForm);
        }
        return "modification";
    }

    @RequestMapping(value="/modifierModificationListeCourses", method = RequestMethod.POST)
    public String modifier(@Valid @ModelAttribute(value="modification") final ModificationForm
pModification,
        final BindingResult pBindingResult, final ModelMap pModel) {

        if (!pBindingResult.hasErrors()) {
            final List<Course> lListeCourses = new LinkedList<Course>();
            for (final ModificationCourse lModificationCourse :
pModification.getListeCourses()) {
                final Course lCourse = new Course();
                lCourse.setId(lModificationCourse.getId());
                final Integer lQuantite = Integer.valueOf(lModificationCourse.getQuantite());
                lCourse.setQuantite(lQuantite);
                lListeCourses.add(lCourse);
            }
            service.modifierCourses(lListeCourses);
        }

        return afficher(pModel);
    }
}

```

La JSP affiche un tableau des différents éléments de la liste de courses. La colonne des quantités comporte des champs de saisie permettant de modifier les différentes quantités. Nous remarquons l'utilisation de la variable « status » qui permet de nommer les champs du formulaire et de filtrer les messages d'erreur selon l'occurrence de la liste des courses.

### /vues/modification.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>

```

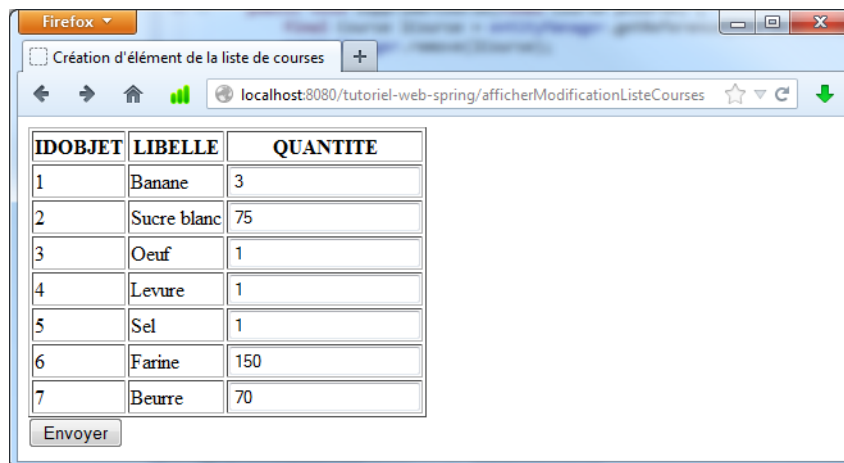
/vues/modification.jsp

```
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
  <head>
    <title><spring:message code="titre.creation.elementcourses"/></title>
  </head>
  <body>

    <form:form method="post" modelAttribute="modification" action="modifierModificationListeCourses">
      <table border="1">
        <thead>
          <tr>
            <th><spring:message code="colonne.identifiant"/></th>
            <th><spring:message code="colonne.libelle"/></th>
            <th><spring:message code="colonne.quantite"/></th>
          </tr>
        </thead>
        <tbody>

          <c:forEach items="${modification.listeCourses}" var="course" varStatus="status">
            <tr>
              <td>
                <c:out value="${course.id}"/>
                <input type="hidden" name="listeCourses[${status.index}].id" value="${course.id}"/>
              </td>
              <td>
                <c:out value="${course.libelle}"/>
                <input type="hidden" name="listeCourses[${status.index}].libelle" value="${course.libelle}"/>
              </td>
              <td>
                <input type="text" name="listeCourses[${status.index}].quantite" value="${course.quantite}"/><br/>
                <b><i><form:errors path="listeCourses[${status.index}].quantite" /></i></b>
              </td>
            </tr>
          </c:forEach>
        </tbody>
      </table>
      <input type="submit"/>
    </form:form>
  </body>
</html>
```

Après déploiement, on obtient le résultat ci-dessous à l'adresse : <http://localhost:8080/tutoriel-web-spring/afficherModificationListeCourses>.



Voici le résultat d'une validation en erreur.

IDOBJET	LIBELLE	QUANTITE
1	Banane	3
2	Sucre blanc	3 <i>La quantité est nécessaire.</i>
3	Oeuf	1
4	Levure	1
5	Sel	un <i>La quantité doit être numérique et positive.</i>
6	Farine	150
7	Beurre	70

Envoyer

Et celui d'une modification réussie.

IDOBJET	LIBELLE	QUANTITE
1	Banane	30
2	Sucre blanc	75
3	Oeuf	1
4	Levure	1
5	Sel	1
6	Farine	150
7	Beurre	70

Envoyer

## VII - Unification de l'application par un menu

Dans ce paragraphe, nous allons ajouter aux pages Web précédemment créées un menu qui permettra de naviguer dans l'application. Pour cela, nous allons utiliser les **Tiles**.

Il faut donc rajouter les dépendances Maven suivantes. Les dépendances « `tiles-jsp` » et « `slf4j-log4j12` » permettent l'utilisation de Tiles.

Extrait du fichier « pom.xml »

```
<dependency>
  <groupId>org.apache.tiles</groupId>
  <artifactId>tiles-jsp</artifactId>
  <version>3.0.3</version>
</dependency>
<dependency>
  <groupId>org.slf4j</groupId>
  <artifactId>slf4j-log4j12</artifactId>
  <version>1.5.11</version>
</dependency>
```

Il faut ajouter un nouveau libellé dans le fichier « messages\_fr.properties ».

**messages\_fr.properties**

```
titre.application=Application de liste de courses
```

Il faut modifier le fichier « dispatcher-servlet.xml » comme ci-dessous. Le bean « **InternalResourceViewResolver** » (déjà présent dans le fichier) doit se trouver après les nouvelles déclarations (simplement parce que Spring teste les « **ViewResolver** » dans l'ordre de déclaration, ce qui permet de résoudre une ressource avec « **UriBasedViewResolver** » puis d'essayer avec « **InternalResourceViewResolver** » en deuxième). Le bean « **UriBasedViewResolver** » (depuis Spring 1.0) utilise « **TilesView** » pour traiter les vues. Le bean « **TilesConfigurer** » charge la configuration dans le fichier « /WEB-INF/tiles.xml ».

**/WEB-INF/dispatcher-servlet.xml**

```
<bean
    class="org.springframework.web.servlet.view.UriBasedViewResolver">

<property name="viewClass" value="org.springframework.web.servlet.view.tiles3.TilesView" />
</bean>

<bean id="tilesConfigurer"
    class="org.springframework.web.servlet.view.tiles3.TilesConfigurer">
    <property name="definitions">
        <list>
            <value>/WEB-INF/tiles.xml</value>
        </list>
    </property>
</bean>

<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix">
        <value>/vues/</value>
    </property>
    <property name="suffix">
        <value>.jsp</value>
    </property>
</bean>
```

Le fichier « tiles.xml » contient la définition des différents tiles. Nous reprenons les noms des ressources utilisées dans les contrôleurs, ce qui nous évite de modifier les contrôleurs.

**/WEB-INF/tiles.xml**

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE tiles-definitions PUBLIC
    "-//Apache Software Foundation//DTD Tiles Configuration 3.0//EN"
    "http://tiles.apache.org/dtds/tiles-config_3_0.dtd">

<tiles-definitions>
    <definition name="listeCourses" template="/vues/page.jsp">
        <put-attribute name="principal" value="/vues/listeCourses.jsp" />
    </definition>

    <definition name="creation" template="/vues/page.jsp">
        <put-attribute name="principal" value="/vues/creation.jsp" />
    </definition>

    <definition name="suppression" template="/vues/page.jsp">
        <put-attribute name="principal" value="/vues/suppression.jsp" />
    </definition>

    <definition name="modification" template="/vues/page.jsp">
        <put-attribute name="principal" value="/vues/modification.jsp" />
    </definition>
</tiles-definitions>
```

La JSP « page.jsp » contient le squelette des pages (le « template » dans le fichier « tiles.xml »). Le tag « **tiles:insertAttribute** » permet d'inclure les JSP contenant le corps de la page et définis dans le fichier « tiles.xml »

**/vues/page.jsp**

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<%@taglib uri="http://tiles.apache.org/tags-tiles" prefix="tiles"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title><spring:message code="titre.application"/></title>
</head>
<body>
<table>
<tbody>
<tr>
<td valign="top">
<table>
<tbody>
<tr><td>
<c:url value="/afficherListeCourses" var="url" />
<a href="{url}">
<spring:message code="titre.listecourses"/>
</a>
</td></tr>
<tr><td>
<c:url value="/afficherCreationListeCourses" var="url" />
<a href="{url}">
<spring:message code="titre.creation.elementcourses"/>
</a>
</td></tr>
<tr><td>
<c:url value="/afficherSuppressionListeCourses" var="url" />
<a href="{url}">
<spring:message code="titre.suppression.elementcourses"/>
</a>
</td></tr>
<tr><td>
<c:url value="/afficherModificationListeCourses" var="url" />
<a href="{url}">
<spring:message code="titre.modification.elementcourses"/>
</a>
</td></tr>
</tbody>
</table>
</td>
<td valign="top">
<tiles:insertAttribute name="principal" />
</td>
</tr>
</tbody>
</table>
</body>
</html>
    
```

Il faut modifier les JSP « listeCourses.jsp », « creation.jsp », « suppression.jsp » et « modification.jsp » pour ne garder que les parties dans la balise « body ».

**/vues/listeCourses.jsp**

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
loose.dtd">

<table border="1">
<thead>
    
```

## /vues/listeCourses.jsp

```

        <tr>
            <th><spring:message code="colonne.identifiant"/></th>
            <th><spring:message code="colonne.libelle"/></th>
            <th><spring:message code="colonne.quantite"/></th>
        </tr>
    </thead>
    <tbody>
        <c:forEach items="${listeCourses}" var="course">
            <tr>
                <td><c:out value="${course.id}"/></td>
                <td><c:out value="${course.libelle}"/></td>
                <td><c:out value="${course.quantite}"/></td>
            </tr>
        </c:forEach>
    </tbody>
</table>
    
```

## /vues/creation.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
    loose.dtd">

<form:form method="post" modelAttribute="creation" action="creerCreationListeCourses">
    <spring:message code="creation.elementcourses.libelle.libelle" />
    <form:input path="libelle"/>
    <b><i><form:errors path="libelle" cssclass="error"/></i></b><br>
    <spring:message code="creation.elementcourses.libelle.quantite"/>
    <form:input path="quantite"/>
    <b><i><form:errors path="quantite" cssclass="error"/></i></b><br>
    <input type="submit"/>
</form:form>
<table border="1">
    <thead>
        <tr>
            <th><spring:message code="colonne.identifiant"/></th>
            <th><spring:message code="colonne.libelle"/></th>
            <th><spring:message code="colonne.quantite"/></th>
        </tr>
    </thead>
    <tbody>
        <c:forEach items="${listeCourses}" var="course">
            <tr>
                <td><c:out value="${course.id}"/></td>
                <td><c:out value="${course.libelle}"/></td>
                <td><c:out value="${course.quantite}"/></td>
            </tr>
        </c:forEach>
    </tbody>
</table>
    
```

## /vues/suppression.jsp

```

<%@ page language="java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
    loose.dtd">

<table border="1">
    <thead>
        <tr>
            <th><spring:message code="colonne.identifiant"/></th>
            <th><spring:message code="colonne.libelle"/></th>
            <th><spring:message code="colonne.quantite"/></th>
            <th>&nbsp;</th>
        </tr>
    </thead>
    <tbody>
        <c:forEach items="${listeCourses}" var="course">
            <tr>
                <td><c:out value="${course.id}"/></td>
                <td><c:out value="${course.libelle}"/></td>
                <td><c:out value="${course.quantite}"/></td>
                <td><input type="button" value="Supprimer" /></td>
            </tr>
        </c:forEach>
    </tbody>
</table>
    
```



**/vues/suppression.jsp**

```

        </tr>
    </thead>
    <tbody>
        <c:forEach items="${listeCourses}" var="course">
            <tr>
                <td><c:out value="${course.id}"/></td>
                <td><c:out value="${course.libelle}"/></td>
                <td><c:out value="${course.quantite}"/></td>
                <td>
                    <c:url value="/supprimerSuppressionListeCourses" var="url">
                        <c:param name="idCourse" value="${course.id}"/>
                    </c:url>
                    <a href="${url}">
                        <spring:message code="suppression.supprimer.libelle" />
                    </a>
                </td>
            </tr>
        </c:forEach>
    </tbody>
</table>
    
```

**/vues/modification.jsp**

```

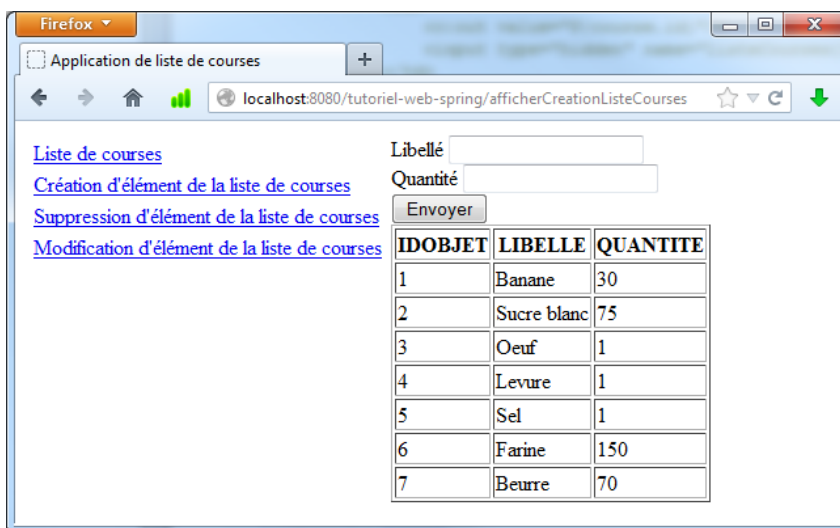
<%@ page language="java" contentType="text/html; charset=ISO-8859-1" isELIgnored="false"
    pageEncoding="ISO-8859-1"%>
<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c"%>
<%@taglib uri="http://www.springframework.org/tags/form" prefix="form"%>
<%@taglib uri="http://www.springframework.org/tags" prefix="spring"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.w3.org/TR/html4/
    loose.dtd">

<form:form method="post" modelAttribute="modification" action="modifierModificationListeCourses">
    <table border="1">
        <thead>
            <tr>
                <th><spring:message code="colonne.identifiant"/></th>
                <th><spring:message code="colonne.libelle"/></th>
                <th><spring:message code="colonne.quantite"/></th>
            </tr>
        </thead>
        <tbody>
            <c:forEach items="${modification.listeCourses}" var="course" varStatus="status">
                <tr>
                    <td>
                        <c:out value="${course.id}"/>
                    </td>
                    <input type="hidden" name="listeCourses[${status.index}].id" value="${course.id}"/>
                    <td>
                        <c:out value="${course.libelle}"/>
                    </td>
                    <input type="hidden" name="listeCourses[${status.index}].libelle" value="${course.libelle}"/>
                    <td>
                        <input type="text" name="listeCourses[${status.index}].quantite" value="${course.quantite}"/><br/>
                    </td>
                    <b><i><form:errors path="listeCourses[${status.index}].quantite" /></i></b>
                    </td>
                </tr>
            </c:forEach>
        </tbody>
    </table>
    <input type="submit"/>
</form:form>
    
```

Après déploiement, on obtient le résultat ci-dessous à l'adresse : <http://localhost:8080/tutoriel-web-spring/afficherListeCourses>.



Les liens dans le menu à gauche permettent de naviguer entre les pages.



## VIII - Conclusion

Attention, cet article n'a pour but que de présenter différentes techniques. Il n'est absolument pas rigoureux dans son code : par exemple, il n'y a aucune gestion des exceptions qui peuvent être levées dans les différentes couches.

Mais, cet article aura permis de voir les techniques principales liées aux applications Web avec Spring (contrôleurs, formulaires, internationalisation).

Nous avons également utilisé la gestion de la transaction avec l'annotation « **@Transactional** », l'injection de dépendances et le support des ressources JDBC et d'Hibernate.

Pour aller plus loin, je vous invite par exemple à vous renseigner sur les différentes « **View** » possibles. Parmi ces « View », on trouve « **JasperReportsPdfView** » qui permet de générer des rapports au format PDF.

Les sources de cet article sont présentes sur GitHub : <https://github.com/regis1512/tutoriel-web-spring>.

## IX - Remerciements

Je remercie très sincèrement :

- **www.developpez.com** qui me permet de publier cet article ;
- **Nono40** et **djibril** pour leurs outils ;
- **Yann Caron** pour sa relecture technique ;
- **Philippe DUVAL** pour sa relecture orthographique rigoureuse.