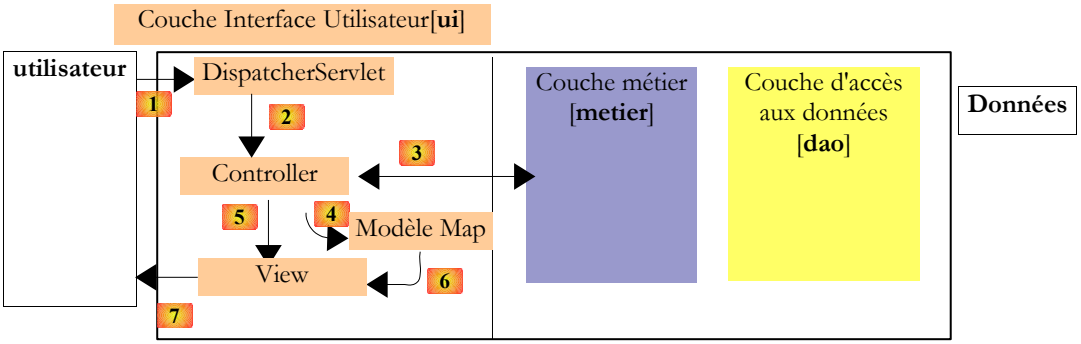
Spring MVC par l'exemple
- Partie 2 -

serge.tahe@istia.univ-angers.fr, mars 2006

1 Rappels

Nous poursuivons dans cet article le travail fait dans le précédent article [http://tahe.developpez.com/java/springmvc-part1]. Rappelons, en quelques lignes, où nous en étions. L'architecture d'une application Spring MVC est la suivante :



1. le client fait une demande au contrôleur. Celui-ci voit passer toutes les demandes des clients. C'est la porte d'entrée de l'application. C'est le **C** de MVC. Ici le contrôleur est assuré par une servlet générique :
   org.springframework.web.servlet.DispatcherServlet
2. le contrôleur principal [DispatcherServlet] fait exécuter l'action demandée par l'utilisateur par une classe implémentant l'interface :
   org.springframework.web.servlet.mvc.Controller
   A cause du nom de l'interface, nous appellerons une telle classe un contrôleur secondaire pour le distinguer du contrôleur principal [**DispatcherServlet**] ou simplement contrôleur lorsqu'il n'y a pas d'ambiguïté. Le schéma ci-dessus s'est contenté de représenter un contrôleur particulier. Il y a en général plusieurs contrôleurs, un par action.
3. le contrôleur **[Controller]** traite une demande particulière de l'utilisateur. Pour ce faire, il peut avoir besoin de l'aide de la couche métier. Une fois la demande du client traitée, celle-ci peut appeler diverses réponses. Un exemple classique est :
   - une page d'erreurs si la demande n'a pu être traitée correctement
   - une page de confirmation sinon
4. le contrôleur choisit la réponse (= vue) à envoyer au client. Choisir la réponse à envoyer au client nécessite plusieurs étapes :
   - choisir l'objet qui va générer la réponse. C'est ce qu'on appelle la vue **V**, le **V** de MVC. Ce choix dépend en général du résultat de l'exécution de l'action demandée par l'utilisateur.
   - lui fournir les données dont il a besoin pour générer cette réponse. En effet, celle-ci contient le plus souvent des informations calculées par la couche métier ou le contrôleur lui-même. Ces informations forment ce qu'on appelle le modèle **M** de la vue, le **M** de MVC. Spring MVC fournit ce modèle sous la forme d'un dictionnaire de type **java.util.Map**.

   L'étape 4 consiste donc en le choix d'une vue **V** et la construction du modèle **M** nécessaire à celle-ci.
5. le contrôleur **DispatcherServlet** demande à la vue choisie de s'afficher. Il s'agit d'une classe implémentant l'interface
   org.springframework.web.servlet.View
   Spring MVC propose différentes implémentations de cette interface pour générer des flux HTML, Excel, PDF, ... Le schéma ci-dessus s'est contenté de représenter une vue particulière. Il y a en général plusieurs vues.
6. le générateur de vue **View** utilise le modèle **Map** préparé par le contrôleur **Controller** pour initialiser les parties dynamiques de la réponse qu'il doit envoyer au client.
7. la réponse est envoyée au client. La forme exacte de celle-ci dépend du générateur de vue. Ce peut être un flux HTML, PDF, Excel, ...

Nous avons vu dans la partie 1 de l'article les points suivants :

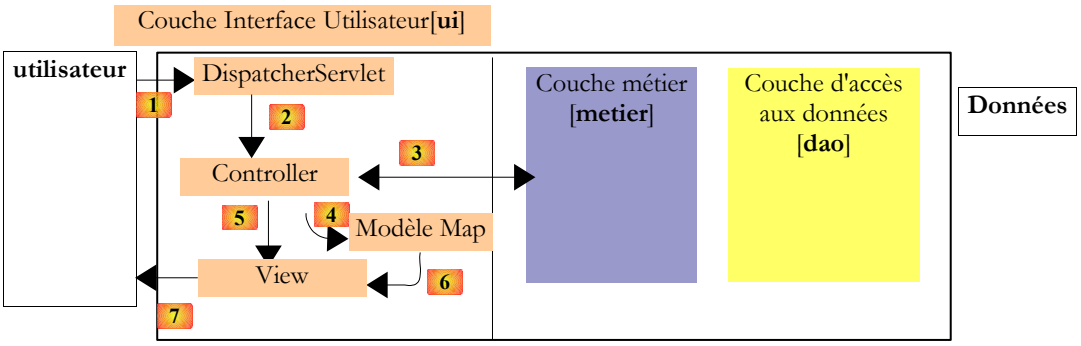
- les différentes stratégies de résolutions d'URL qui associent à une URL demandée par le client, un contrôleur implémentant l'interface [Controller] qui va traiter la demande du client
- les différentes façons qu'avait un contrôleur [Controller] d'accéder au contexte de l'application, par exemple pour accéder aux instances des couches [métier] et [dao]
- les différentes stratégies de résolutions de noms de vue qui, à un nom de vue rendu par le contrôleur [Controller] associe une classe implémentant l'interface [View] chargée d'afficher le modèle [Map] construit par le contrôleur [Controller]

Dans le présent article, nous allons nous intéresser plus particulièrement à la chaîne de traitement de la requête du client..

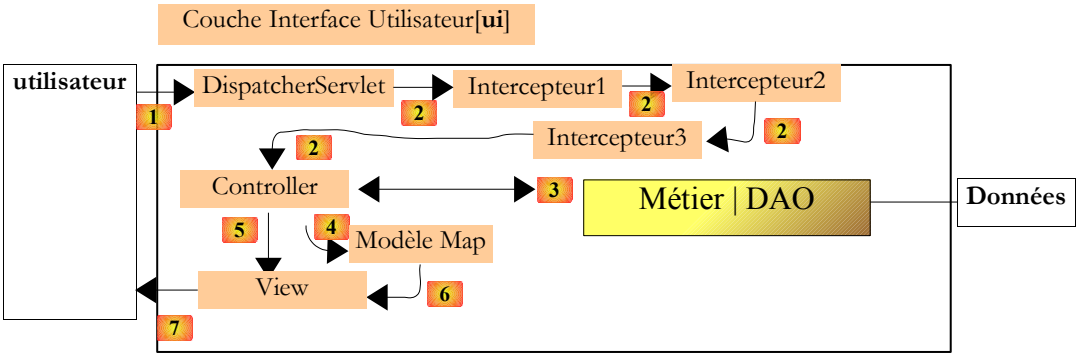
2 Les classes de type "intercepteur"

2.1 L'interface HandlerInterceptor

Revenons sur le schéma de base d'une architecture Spring MVC :



Lors de l'étape 2 ci-dessus, le contrôleur général [DispatcherServlet] transmet la requête du client à l'objet [Controller] associé à l'URL demandée. On peut placer entre [DispatcherServlet] et [Controller] une série d'objets appelés Intercepteurs. Le schéma précédent devient alors le suivant :



Au cours de l'étape 2, la requête [HttpRequest] du client va être traitée par les différents modules placés entre le contrôleur général [DispatcherServlet] et le contrôleur particulier [Controller] qui doit traiter la demande du client. Les intercepteurs ont pour vocation de factoriser un comportement commun à plusieurs objets [Controller] d'une application. Parmi les exemples généralement cités, on trouve l'authentification des utilisateurs et la production de logs sur l'activité de l'application, mais il en existe potentiellement bien d'autres.

Dans une application avec authentification, chaque [Controller] devrait vérifier que la requête qu'il doit traiter provient bien d'un utilisateur authentifié. Plutôt que de dupliquer le code d'authentification dans chacun des contrôleurs [Controller], on préfèrera placer ce code dans une classe *Intercepteur* et indiquer par configuration, que toute requête doit passer par cet intercepteur avant d'être relayée au [Controller] qui doit la traiter. L'intercepteur peut rendre lui-même la réponse au client, court-circuitant ainsi le [Controller]. C'est ce qui serait fait si l'intercepteur découvrait que l'utilisateur n'était pas authentifié. Il interromprait le traitement de la requête pour rediriger le client vers une page d'authentification.

Les modules d'interception doivent implémenter l'interface [HandlerInterceptor] suivante :

org.springframework.web.servlet

Interface HandlerInterceptor

All Known Implementing Classes:

HandlerInterceptorAdapter, LocaleChangeInterceptor, OpenPersistenceManagerInViewInterceptor, OpenSessionInViewInterceptor, OpenSessionInViewInterceptor, ThemeChangeInterceptor, UserRoleAuthorizationInterceptor, WebContentInterceptor

Les méthodes de l'interface [HandlerInterceptor] sont les suivantes :

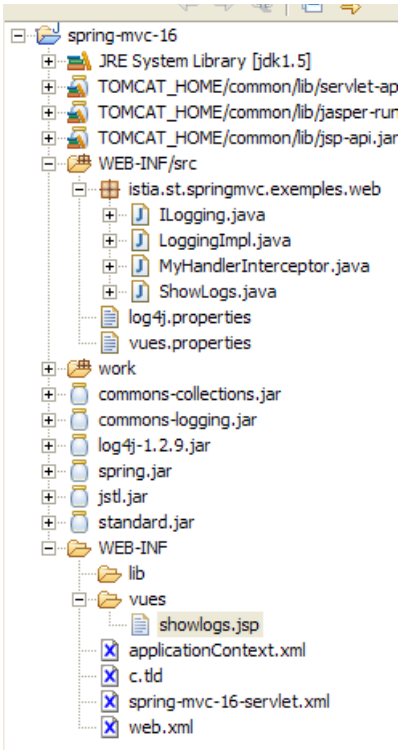
Method Summary	
void	**afterCompletion**(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) Callback after completion of request processing, that is, after rendering the view.
void	**postHandle**(HttpServletRequest request, HttpServletResponse response, Object handler, ModelAndView modelAndView) Intercept the execution of a handler.
boolean	**preHandle**(HttpServletRequest request, HttpServletResponse response, Object handler) Intercept the execution of a handler.

Si on a une chaîne de plusieurs intercepteurs, les règles précédentes s'appliquent à chacun d'eux. Un paramètre de configuration des intercepteurs permet de fixer leur ordre d'exécution dans la chaîne.

Nous allons tout d'abord décrire un intercepteur "maison". Puis nous parlerons des intercepteurs offerts par Spring.

2.2 Implémentation de l'interface HandlerInterceptor

Le projet Eclipse sera le suivant :



Nous voulons écrire un intercepteur pour enregistrer des logs. Il y a plusieurs façons d'implémenter un système de logs. Nous ferons simple. Néanmoins pour garder de la souplesse, nous implémenterons une interface. Ceci nous permettra de travailler avec cette interface et de garder ainsi de l'indépendance vis à vis des classes d'implémentation de celle-ci.

L'interface sera la suivante :

```
package istia.st.springmvc.exemples.web;

import java.util.List;

public interface ILogging {
  public void log(String info);
  public List showLogs();
}
```

`void log(String info)`	mémorise une chaîne [info] dans les logs
`List showLogs()`	demande les logs sous la forme d'un type [List]

Pour implémenter cette interface, nous nous conterons d'une classe enregistrant les logs dans un objet [ArrayList] :

```
1.  package istia.st.springmvc.exemples.web;
```

```
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  public class LoggingImpl implements ILogging {
7.
8.    // liste de logs
9.    private ArrayList logs=new ArrayList();
10.
11.   // ajout d'un log
12.   public synchronized void log(String info) {
13.     logs.add(info);
14.   }
15.   // liste des logs
16.   public synchronized List showLogs(){
17.     return logs;
18.   }
19.
20. }
```

- ligne 6 : on implémente l'interface ILogging
- ligne 9 : le conteneur des logs est un objet [ArrayList]
- lignes 12-14 : la méthode **log**
- lignes 16-18 : la méthode **showLogs**

Comme la classe va être instanciée en un seul exemplaire (singleton) et qu'elle sera utilisée par plusieurs threads, nous avons synchronisé (synchronized) les deux méthodes (lignes 12 et 16). La méthode [log], n'ayant qu'une unique instruction, il est difficile de prévoir les conséquences d'une interruption de la méthode avant son terme. Il faudrait connaître le détail de la séquence d'instructions "processeur" exécutées par la méthode. Je ne sais pas dire si la synchronisation est nécessaire ou non. J'ai suivi ici un principe de précaution en synchronisant les deux méthodes.

Le singleton chargé de mémoriser les logs sera sera instancié par Spring et placé dans le contexte de l'application web à son démarrage. Il est défini dans [applicationContext.xml] :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- service de logging -->
5.    <bean id="logger" class="istia.st.springmvc.exemples.web.LoggingImpl"/>
6.  </beans>
```

- ligne 6 : le bean a l'id "logger" et est une instance de la classe [LoggingImpl]

La configuration de la servlet [spring-mvc-16] faite dans [spring-mvc-16-servlet.xml] est la suivante :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les mappings de l'application-->
5.    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
6.      <property name="interceptors">
7.        <list>
8.          <bean class="istia.st.springmvc.exemples.web.MyHandlerInterceptor">
9.            <property name="logger">
10.             <ref bean="logger"/>
11.           </property>
12.         </bean>
13.       </list>
14.     </property>
15.     <property name="mappings">
16.       <props>
17.         <prop key="/showlogs.html">ShowLogsController</prop>
18.       </props>
19.     </property>
20.   </bean>
21.   <!-- les contrôleurs de l'application-->
22.   <bean id="ShowLogsController"
23.     class="istia.st.springmvc.exemples.web.ShowLogs">
24.     <property name="logger">
25.       <ref bean="logger"/>
26.     </property>
27.   </bean>
28.   <!-- le résolveur de vues -->
29.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
30.     <property name="basename">
31.       <value>vues</value>
32.     </property>
33.   </bean>
34. </beans>
```

Commençons par décrire ce qui est analogue avec les applications précédentes :

- lignes 28-33 : le résolveur de noms de vues est [ResourceBundleViewResolver].
- ligne 31 : le fichier de définition des vues est [vues.properties] qui devra être dans le *ClassPath* de l'application
- lignes 5-20 : les liaisons URL <-> Controller de l'application.
- ligne 17 : **/showlogs.html** sera l'unique Url acceptée. Elle est associée au contrôleur [ShowLogsController]
- lignes 22-27 : définissent le contrôleur d'id [ShowLogsController]. On voit que ce contrôleur aura une référence sur le singleton [logger] défini dans [applicationContext.xml].

La nouveauté vient de la présence d'intercepteurs dans le fichier de configuration :

- lignes 6-14 : définissent les classes intercepteurs de l'application. Il n'y en a qu'une.
- lignes 8-11 : définissent l'unique classe intercepteur, une instance de type [MyHandlerInterceptor]. Cette classe propriétaire implémente l'interface [HandlerInterceptor]. Nous y reviendrons bientôt. On voit (lignes 9-11) que cette classe détient également une référence sur le singleton "logger" de l'application.

Qu'avons-nous appris ?

- l'application n'accepte que l'url **/showlogs.html**
- que la requête liée à cette URL va être traitée d'abord par l'intercepteur [MyHandlerInterceptor] puis par le contrôleur [ShowLogsController].
- ce dernier va demander l'affichage d'une des vues définies dans [vues.properties]

L'intercepteur [MyHandlerInterceptor] sera le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.text.SimpleDateFormat;
4.  import java.util.Date;
5.  import java.util.Iterator;
6.  import java.util.Map;
7.  import javax.servlet.http.HttpServletRequest;
8.  import javax.servlet.http.HttpServletResponse;
9.
10. import org.springframework.web.servlet.HandlerInterceptor;
11. import org.springframework.web.servlet.ModelAndView;
12.
13. public class MyHandlerInterceptor implements HandlerInterceptor {
14.
15.  // le logueur
16.  private ILogging logger;
17.
18.  public ILogging getLogger() {
19.    return logger;
20.  }
21.
22.  public void setLogger(ILogging logger) {
23.    this.logger = logger;
24.  }
25.
26.  // traitement avant controller
27.  public boolean preHandle(HttpServletRequest request,
28.      HttpServletResponse response, Object controller) throws Exception {
29.    // on rassemble qqs informations sur la requête
30.    String client = request.getRemoteAddr() + ":" + request.getRemotePort();
31.    String heure = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss:SSS")
32.        .format(new Date());
33.    String controllerClassName = controller.getClass().getName();
34.    String requête = request.getQueryString();
35.    // on logue ces informations
36.    logger.log("(" + this.getClass().getName() + ") preHandle : heure=" + heure + ", client=" +
   client
37.        + ", requête=" + requête + ", contrôleur="
38.        + controllerClassName);
39.    // on passe la main à l'intecepteur suivant
40.    return true;
41.  }
42.
43.  // traitement après controller
44.  public void postHandle(HttpServletRequest request,
45.      HttpServletResponse response, Object controller,
46.      ModelAndView modelAndView) throws Exception {
47.    // on rassemble qqs informations sur la requête
48.    String client = request.getRemoteAddr() + ":" + request.getRemotePort();
49.    String heure = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss:SSS")
50.        .format(new Date());
51.    String controllerClassName = controller.getClass().getName();
52.    String requête = request.getQueryString();
```

```
53.    // des informations sur le ModelAndView
54.    String vue = modelAndView.getViewName();
55.    Map modèle = modelAndView.getModel();
56.    Iterator iter = modèle.keySet().iterator();
57.    String clés = "[";
58.    while (iter.hasNext()) {
59.      clés += "(" + iter.next() + ")";
60.    }
61.    clés += "]";
62.    // on logue ces informations
63.    logger.log("(" + this.getClass().getName() + ") postHandle : heure"
64.        + heure + ", client=" + client + ", requête=" + requête
65.        + ", contrôleur=" + controllerClassName + ", vue=" + vue
66.        + ", clés du modèle=" + clés);
67.  }
68.
69.  public void afterCompletion(HttpServletRequest request,
70.      HttpServletResponse response, Object controller, Exception ex)
71.      throws Exception {
72.    // on rassemble qqs informations sur la requête
73.    String client = request.getRemoteAddr() + ":" + request.getRemotePort();
74.    String heure = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss:SSS")
75.        .format(new Date());
76.    String controllerClassName = controller.getClass().getName();
77.    String requête = request.getQueryString();
78.    // on logue ces informations
79.    logger.log("(" + this.getClass().getName() + ") afterCompletion : heure=" + heure + ", client="
    + client
80.        + ", requête=" + requête + ", contrôleur="
81.        + controllerClassName);
82.  }
83.
84. }
```

- ligne 13 : la classe [MyHandlerInterceptor] implémente l'interface [HandlerInterceptor]. De ce fait, elle doit implémenter les méthodes [preHandle] (lignes 27-41), [postHandle] (lignes 44-67), [afterCompletion] (lignes 69-82).
- ligne 16 : le champ privé [logger] qui sera une référence au singleton d'id "logger" créé par Spring au démarrage de l'application. Lorsque l'une des trois méthodes précédentes est exécutée, ce champ est déjà initialisé. Il est accompagné de son *getter* et de son *setter*. Seul ce dernier est obligatoire pour permettre l'initialisation du champ par Spring IoC. On notera qu'en ligne 16, c'est l'interface [ILogging] qui est utilisée et non une implémentation particulière de celle-ci.

Avant d'expliquer le code des trois méthodes, rappelons le moment où [DispatcherServlet] va leur passer le flux d'exécution :

- **preHandle** : s'exécute avant l'objet [Controller] qui va traiter la requête
- **postHandle** : s'exécute après l'objet [Controller] qui va traiter la requête
- **afterCompletion** : s'exécute après que la réponse ait été envoyée au client

Dans les trois méthodes, nous loguons des informations grâce à l'objet "logger" de l'application web. Ces informations vont nous permettrent de suivre le flux d'exécution de la requête.

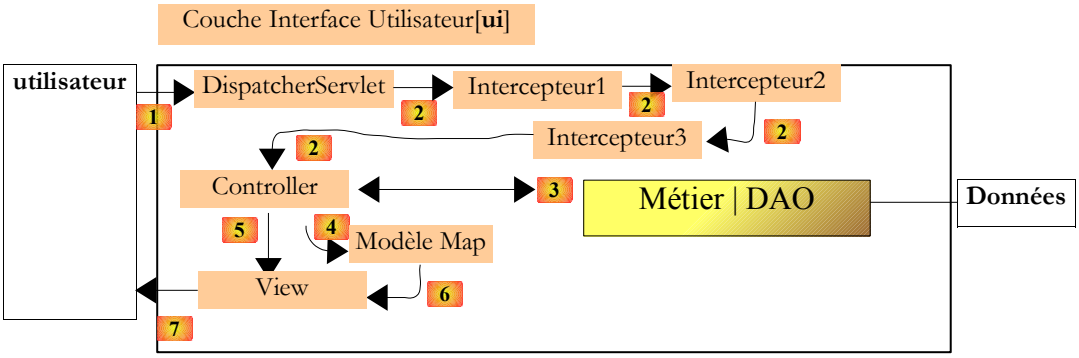
- ligne 27 : **preHandle** logue les informations suivantes :
- ligne 30 : les caractéristiques TCP-IP du client : adresse IP, port
- ligne 31 : l'heure du moment
- ligne 33 : le nom de la classe du contrôleur. Celui-ci est le 3ième paramètre de la méthode. On veut simplement montrer ici que l'intercepteur a bien connaissance du contrôleur qui, en bout de chaîne, va traiter la requête. Ici, le contrôleur sera [ShowLogs] (cf spring-mvc-16-servlet.xml).
- ligne 34 : la chaîne de paramètres associée à la requête courante. Ici, on veut montrer que l'intercepteur a bien accès à la requête courante.
- ligne 36 : on demande au "logger" de mémoriser les informations précédentes
- ligne 40 : on rend à [DispatcherServlet] le booléen *true* pour indiquer qu'on peut passer le flux d'exécution à l'intercepteur suivant ou au contrôleur si la chaîne des intercepteurs est terminée. Un intercepteur peut en effet décider d'arrêter le traitement de la requête courante. Grâce à l'objet [HttpResponse response] qu'il a obtenu en 2ième paramètre, il peut envoyer lui-même la réponse au client et dire à [DispatcherServlet] que la réponse a été envoyée en renvoyant le booléen *false*. [DispatcherServlet] arrêtera alors la chaîne de traitement de la requête. Un exemple de ce cas d'usage est le défaut d'authentification. Un intercepteur d'authentification découvrant qu'une requête n'est pas accompagnée des éléments d'authentification nécessaires, redirigera probablement le client vers une page d'authentification et rendra *false* à [DispatcherServlet].

- ligne 44 : **postHandle** logue les informations suivantes :
- ligne 48-52 : on enregistre les mêmes informations que dans [preHandle]. On veut pouvoir relier les différents logs entre-eux. C'est le client qui servira de lien. L'heure nous permet de voir quel log vient avant tel autre. La classe du contrôleur et la chaîne de paramètres nous permettent également de vérifier que ces paramètres sont identiques à ceux de la méthode [preHandle].

- lignes 54-61 : on enregistre des informations sur le [ModelAndView] créé par le contrôleur [Controller]. Lorsque [postHandle] s'exécute, la méthode [handleRequest] du contrôleur [Controller] a été exécutée et a retourné un objet [ModelAndView] à [DispatcherServlet]. Celui-ci appelle alors la méthode [postHandle] de tous les intercepteurs en leur passant l'objet [ModelAndView] reçu (4ième paramètre de postHandle).
- ligne 54 : le nom de la vue
- ligne 55 : le modèle (dictionnaire) de la vue
- lignes 56-61 : on parcourt le modèle, pour enregistrer ses clés et ses valeurs

- ligne 69 : **afterCompletion** logue les informations suivantes :
- lignes 73-82 : on logue les mêmes informations que [preHandle]. On notera qu'en 4ième paramètre, la méthode reçoit une exception, celle qui s'est éventuellement produite au cours du traitement de la requête. Ce peut être là l'occasion de la mettre dans des logs.

Revenons au schéma général du traitement d'une requête :



Dans notre application, il n'y a qu'un intercepteur. [MyHandlerInterceptor] logue des informations à trois moments différents du traitement d'une requête :

- avant que la requête ne soit traitée par son [Controller] (preHandle)
- après que la requête ait été traitée par son [Controller] (postHandle)
- lorsque la réponse à la requête ait été envoyée.

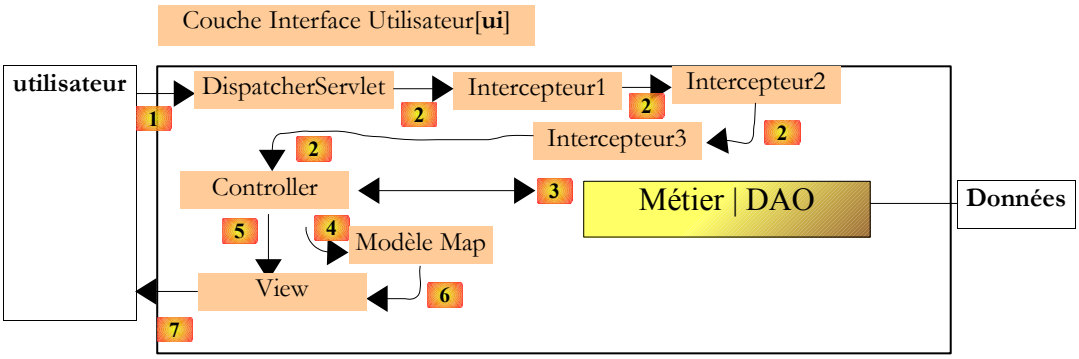
Le schéma ci-dessus montre qu'après l'intercepteur, le contrôleur [Controller] va traiter la requête. Notre contrôleur sera le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.text.SimpleDateFormat;
4.  import java.util.Date;
5.  import java.util.HashMap;
6.
7.  import javax.servlet.http.HttpServletRequest;
8.  import javax.servlet.http.HttpServletResponse;
9.
10. import org.springframework.web.servlet.ModelAndView;
11. import org.springframework.web.servlet.mvc.Controller;
12.
13. public class ShowLogs implements Controller {
14.
15.  // le logueur de l'application
16.  ILogging logger;
17.
18.  public ILogging getLogger() {
19.    return logger;
20.  }
21.
22.  public void setLogger(ILogging logger) {
23.    this.logger = logger;
24.  }
25.
26.  // traitement de la requête
27.  public ModelAndView handleRequest(HttpServletRequest request, HttpServletResponse response)
    throws Exception {
28.    // on rassemble qqs informations sur la requête
29.    String client = request.getRemoteAddr() + ":" + request.getRemotePort();
30.    String heure = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss:SSS")
31.        .format(new Date());
32.    String requête = request.getQueryString();
33.    // on logue ces informations
34.    logger.log("(" + this.getClass().getName() + ") handleRequest : heure=" + heure + ", client=" +
    client
35.        + ", requête=" + requête);
36.    // on construit le modèle
```

```
37.     HashMap modèle=new HashMap();
38.     // on met les logs dedans
39.     modèle.put("logs",logger.showLogs());
40.     // on ajoute un élément arbitraire dans le modèle
41.     modèle.put("something",new Object());
42.     // on rend la vue à afficher avec son modèle
43.     return new ModelAndView("showlogs",modèle);
44.   }
45.
46. }
```

- ligne 13 : [ShowLogs] est un contrôleur, donc il implémente l'interface [Controller] et donc la méthode [handleRequest] (ligne 27). Le contrôleur [ShowLogs] a pour objet d'afficher les informations mémorisées par le "logger" de l'application. Il fait ce travail dans sa méthode [handleRequest].
- lignes 16-24 : [ShowLogs] a besoin d'avoir accès au "logger" de l'application. On met dans le champ [logger] de la ligne 16, une référence au "logger" de portée application défini dans [applicationContext.xml] et créé par Spring au démarrage de l'application. On notera qu'en ligne 16, c'est l'interface [ILogging] qui est utilisée et non une implémentation particulière de celle-ci.
- lignes 29-32 : on mémorise les mêmes informations que dans l'intercepteur
- lignes 37-41 : on construit le modèle [Map] qui va être rendu à [DispatcherServlet].
- ligne 39 : on y met les logs mémorisés par le "logger" de l'application
- ligne 41 : on ajoute une clé arbitraire pour l'exemple
- ligne 43 : on rend un [ModelAndView] ayant pour nom de vue "showlogs" et pour modèle, celui qui vient d'être construit.

Revenons au schéma général du traitement d'une requête :



Nous sommes aux étapes 4-5 : le contrôleur vient de rendre à [DispatcherServlet] :

- le nom de la vue à afficher
- le modèle que doit afficher cette vue

Dans [spring-mvc-16-servlet.xml], la stratégie de résolution des vues est la suivante :

```
1.   <!-- le résolveur de vues -->
2.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
3.     <property name="basename">
4.       <value>vues</value>
5.     </property>
6.   </bean>
```

Le fichier [vues.properties] va être exploité. son contenu est le suivant :

```
1.  #showlogs
2.  showlogs.class=org.springframework.web.servlet.view.JstlView
3.  showlogs.url=/WEB-INF/vues/showlogs.jsp
```

Les lignes 2 et 3 caractérisent la vue nommée "showlogs". La classe d'affichage est [JstlView] (ligne 2), la classe adéquate pour les pages JSP. La ligne 3 donne l'url de la page JSP chargée d'afficher le modèle.

La page [showlogs.jsp] est la suivante :

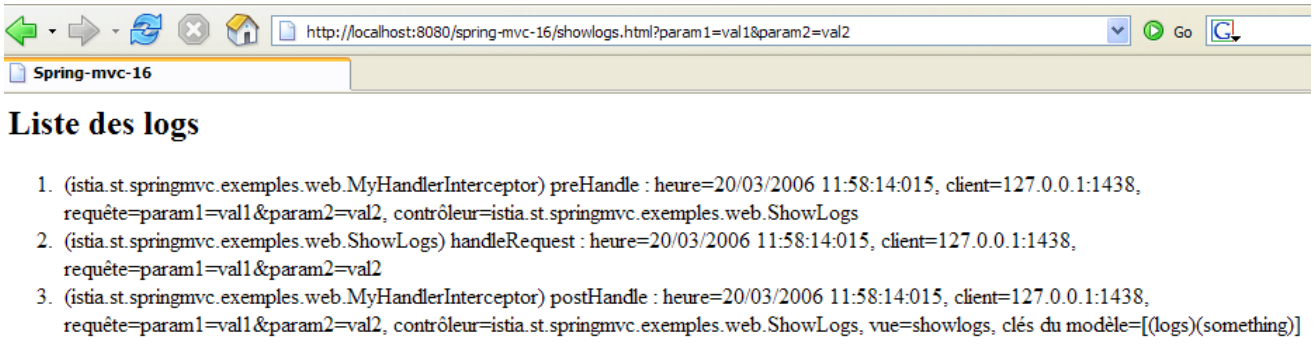
```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-16</title>
8.    </head>
9.    <body>
```

```
10.     <h2>Liste des logs</h2>
11.     <ol>
12.       <c:forEach var="log" items="${logs}">
13.         <li>${log}</li>
14.       </c:forEach>
15.     </ol>
16.   </body>
17. </html>
```

On se rappelle que le contrôleur [ShowLogs] a mis deux clés dans le modèle :

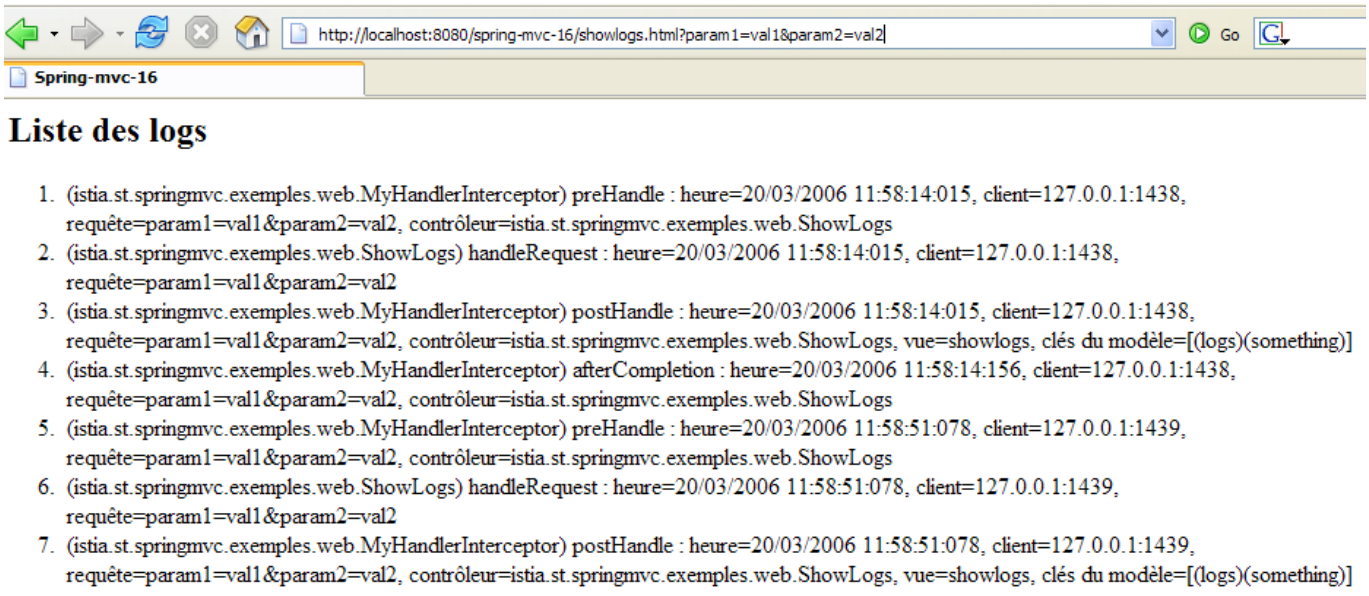
- "logs" associée à un objet [ArrayList] de chaîne de caractères que nous affichons lignes 12-14
- "something" associée à un objet [Object] que nous n'affichons pas ici

Maintenant que nous avons une bonne idée de toute la chaîne de traitement de la requête, nous pouvons passer aux tests. Nous demandons l'url [http://localhost:8080/spring-mvc-16/showlogs.html?param1=val1&param2=val2] :



- **log 1** : on est dans la méthode [preHandle] de l'intercepteur [MyHandlerInterceptor]. Le client est sur localhost (127.0.0.1) et travaille depuis le port 1437. La chaîne de paramètres de la requête en cours de traitement est [param1=val1&param2=val2]. La classe du contrôleur est [ShowLogs].
- **log 2** : on est dans la méthode [handleRequest] du contrôleur [ShowLogs]. Le log n'apporte pas de précisions supplémentaires vis à vis du log précédent : même client, même chaîne de paramètres.
- **log 3** : on est dans la méthode [postHandle] de l'intercepteur [MyHandlerInterceptor]. On voit ici que [postHandle] connaît le nom de la vue "showlogs" demandée par le contrôleur [ShowLogs] et son modèle dont il est capable de récupérer les clés "logs" et "something".

On notera qu'on n'a pas de logs de la méthode [afterCompletion] de l'intercepteur. Ceci est normal, car lorsque celle-ci s'exécute, la réponse a déjà été envoyée au client. Rechargeons la page :



On retrouve les trois précédents logs 1 à 3. Le log 4 lui montre l'exécution de la méthode [afterCompletion] de l'intercepteur. Ensuite les logs 5 à 7 sont analogues aux logs 1 à 3, à l'heure eu port près (1439 au lieu de 1438). Au passage, ceci montre que le client a ouvert des connexions séparées pour chacune des requêtes comme le veut le protocole sans état HTTP.

Nous présentons maintenant un intercepteur particulier appelé [LocaleChangeInterceptor]qui permet de gérer l'internationalisation des vues.

3 Gestion de la localisation des vues d'une application

3.1 La localisation par [AcceptHeaderLocaleResolver]

Pour décider de la langue à utiliser pour les vues envoyées à l'utilisateur, Spring MVC peut utiliser différentes stratégies. Le choix de l'une d'elles en particulier, est fait par configuration, en choisissant la classe qui va implémenter l'interface [LocaleResolver] :

org.springframework.web.servlet

Interface LocaleResolver

All Known Implementing Classes:

AcceptHeaderLocaleResolver, CookieLocaleResolver, FixedLocaleResolver, SessionLocaleResolver

L'interface a deux méthodes :

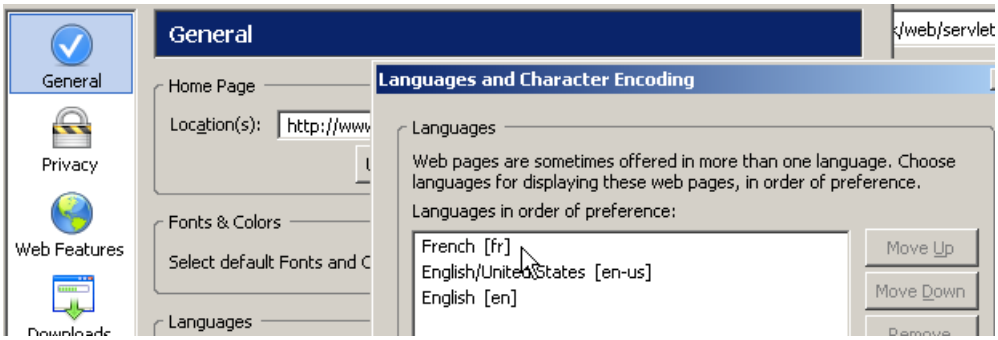
Method Summary

Locale	**resolveLocale**(HttpServletRequest request) Resolve the current locale via the given request.
void	**setLocale**(HttpServletRequest request, HttpServletResponse response, Locale locale) Set the current locale to the given one.

`resolveLocale`	permet de connaître la " localisation " de la requête courante
`setLocale`	permet de changer cette localisation

Par défaut, c'est la classe [AcceptHeaderLocaleResolver] qui est utilisée comme stratégie de résolution de la localisation. Celle-ci est alors faite d'après l'entête HTTP " accept-language " envoyé par le navigateur client. La méthode [setLocale] de la classe [AcceptHeaderLocaleResolver] renvoie une exception. Il n'est en effet pas possible de changer de localisation par programme.

Nous avons rencontré dans la partie 1, une méthode de résolution des noms de vues appelée [ResourceBundleViewResolver] qui permettait l'internationalisation des vues. La langue utilisée dans les vues était fixée par le navigateur client dans l'entête HTTP " accept-language ". Si l'utilisateur voulait en changer, il devait modifier la configuration de son navigateur. Par exemple avec la navigateur Firefox (Tools/ Options / Languages) :



L'inconvénient de cette méthode est son manque de souplesse. Pour changer de langue, l'utilisateur doit changer la configuration de son navigateur. Avec le même navigateur, il ne peut pas travailler en même temps avec deux applications dans deux langues différentes, l'une en français, l'autre en anglais par exemple. La stratégie [SessionLocaleResolver] apporte une solution plus souple.

3.2 La localisation par [SessionLocaleResolver]

Revenons sur l'interface [LocaleResolver] :

org.springframework.web.servlet

Interface LocaleResolver

All Known Implementing Classes:

AcceptHeaderLocaleResolver, CookieLocaleResolver, FixedLocaleResolver, SessionLocaleResolver

Parmi les classes d'implémentation de l'interface [LocaleResolver], il y a la classe [SessionLocaleResolver] :

```
public class SessionLocaleResolver
extends Object
implements LocaleResolver
```

Implementation of LocaleResolver that uses a locale attribute in the user's session in case of a custom setting, with a fallback to the accept header locale. This is most appropriate if the application needs user sessions anyway.

Custom controllers can override the user's locale by calling `setLocale`, e.g. responding to a locale change request.

La classe [SessionLocaleResolver] gère en session un attribut associé à la localisation. La valeur de cet attribut peut être changée. Ainsi un utilisateur peut :

- décider en début d'application de travailler avec une langue L1 qu'il aura choisie (un combo sur une page par exemple)
- tant qu'il ne la changera pas, il recevra les vues dans cette langue (mécanisme de la session) et ce quelque soit la configuration de son navigateur
- en cours d'utilisation de l'application, il peut décider de changer de langue

Pour permettre la détection du changement de langue, Spring MVC propose l'intercepteur [LocaleChangeInterceptor] :

org.springframework.web.servlet.i18n

Class LocaleChangeInterceptor

java.lang.Object
  └ org.springframework.web.servlet.handler.HandlerInterceptorAdapter
    └ **org.springframework.web.servlet.i18n.LocaleChangeInterceptor**

All Implemented Interfaces:

HandlerInterceptor

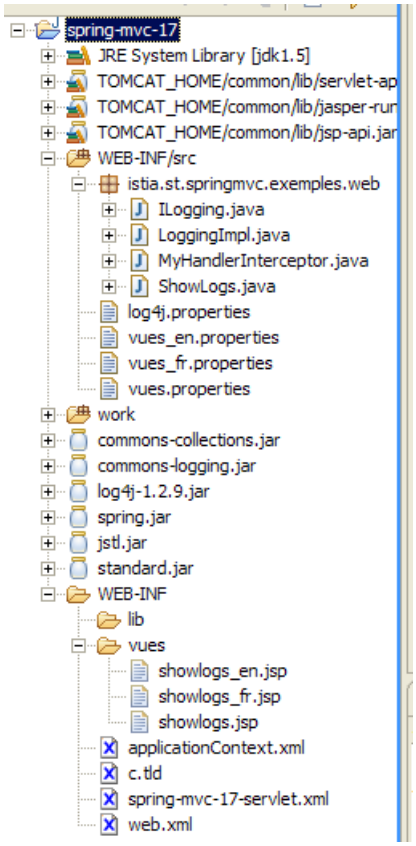
Cet intercepteur permet à l'utilisateur de changer la langue utilisée dans les vues, au moyen d'un paramètre présent dans l'URL de la requête.

En plus d'implémenter l'interface [HandlerInterceptor], la classe [LocaleChangeInterceptor] possède la méthode suivante :

void	**setParamName**(String paramName) Set the name of the parameter that contains a locale specification in a locale change request.

Cette méthode permet de préciser quel paramètre de l'URL d'une requête client définit la langue. Par défaut ce paramètre est **language**. Ainsi l'url [/URL?language=de_AT] demande une localisation des vues en allemand d'Autriche. Le nom de ce paramètre peut donc être changé par la méthode [setParamName] ci-dessus. Tant qu'une première requête n'a pas précisé ce paramètre, c'est la localisation demandée par l'entête HTTP " accept-language " qui est utilisée.

Pour illustrer cette stratégie de localisation, nous allons internationaliser l'application de logs précédente. Le projet Eclipse sera le suivant :



Ce projet est très proche du projet [spring-mvc-16] précédent. Nous ne décrivons que les nouveautés.

Le fichier [spring-mvc-17-servlet.xml] décrivant la servlet est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les résolveurs de localisation -->
5.    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.SessionLocaleResolver"/>
6.    <!-- les mappings de l'application-->
7.    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
8.      <property name="interceptors">
9.        <list>
10.         <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
11.           <property name="paramName">
12.             <value>langue</value>
13.           </property>
14.         </bean>
15.         <bean class="istia.st.springmvc.exemples.web.MyHandlerInterceptor">
16.           <property name="logger">
17.             <ref bean="logger"/>
18.           </property>
19.         </bean>
20.       </list>
21.     </property>
22.     <property name="mappings">
23.       <props>
24.         <prop key="/showlogs.html">ShowLogsController</prop>
25.       </props>
26.     </property>
27.   </bean>
28.   <!-- les contrôleurs de l'application-->
29.   <bean id="ShowLogsController"
30.     class="istia.st.springmvc.exemples.web.ShowLogs">
31.     <property name="logger">
32.       <ref bean="logger"/>
33.     </property>
34.   </bean>
35.   <!-- le résolveur de vues -->
36.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
37.     <property name="basename">
38.       <value>vues</value>
39.     </property>
40.   </bean>
41. </beans>
```

La première nouveauté est en lignes 5-6 : la stratégie [SessionLocaleResolver] est utilisée pour internationaliser les pages. Elle va permettre à l'utilisateur de préciser la langue désirée sans changer la configuration de son navigateur. Pour cela, il faut ajouter à la liste des intercepteurs de l'application, l'intercepteur [LocaleChangeInterceptor] défini lignes 10-14. On indique, ligne 12, que le paramètre de l'URL qui sert à définir la langue est le paramètre " **langue** ".

Pour le reste, le projet est en tout point identique au projet [spring-mvc-16], si ce n'est que du fait de l'internationalisation des vues, on a créé différentes versions de pages pour un même nom de vue. Les fichiers de définition de vues se trouvent dans [WEB-INF/src] :

[vues_fr.properties] : définit les vues en français

```
#showlogs
showlogs.class=org.springframework.web.servlet.view.JstlView
showlogs.url=/WEB-INF/vues/showlogs_fr.jsp
```

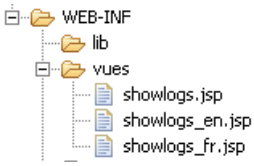
[vues_en.properties] : définit les vues en anglais

```
#showlogs
showlogs.class=org.springframework.web.servlet.view.JstlView
showlogs.url=/WEB-INF/vues/showlogs_en.jsp
```

[vues.properties] : définit les autres vues

```
#showlogs
showlogs.class=org.springframework.web.servlet.view.JstlView
showlogs.url=/WEB-INF/vues/showlogs.jsp
```

On a trois versions de la page JSP [showlogs] :



[**showlogs_fr.jsp**] : version française

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-17</title>
8.    </head>
9.    <body>
10.     <h2>Liste des logs (fr)</h2>
11.     <ul>
12.       <c:forEach var="log" items="${logs}">
13.         <li>${log}</li>
14.       </c:forEach>
15.     </ul>
16.   </body>
17. </html>
```

[**showlogs_en.jsp**] : version anglaise

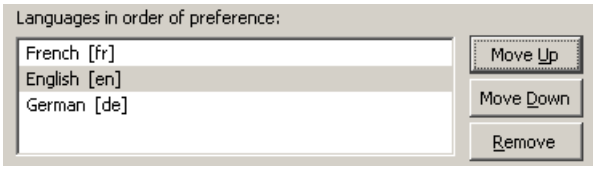
```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-17</title>
8.    </head>
9.    <body>
10.     <h2>List of logs (en)</h2>
11.     <ul>
12.       <c:forEach var="log" items="${logs}">
13.         <li>${log}</li>
14.       </c:forEach>
15.     </ul>
```

```
16.   </body>
17. </html>
18.
```

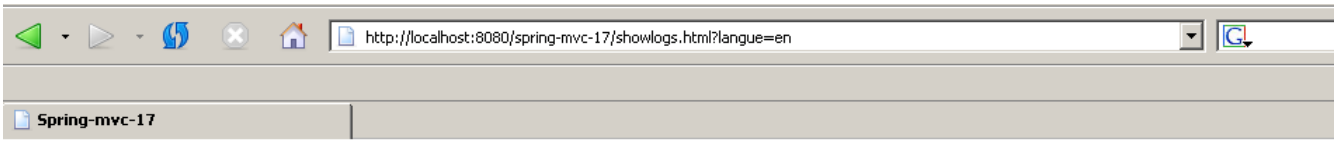
[**showlogs.jsp**] : autres langues

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-17</title>
8.    </head>
9.    <body>
10.     <h2>Liste des logs (default)</h2>
11.     <ul>
12.       <c:forEach var="log" items="${logs}">
13.         <li>${log}</li>
14.       </c:forEach>
15.     </ul>
16.   </body>
17. </html>
```

La ligne 10 de chacun des trois fichiers devrait nous permettre de distinguer les trois vues. Nous sommes prêts pour les tests. Nous travaillons avec un navigateur Firefox dont les langues sont configurées comme suit :



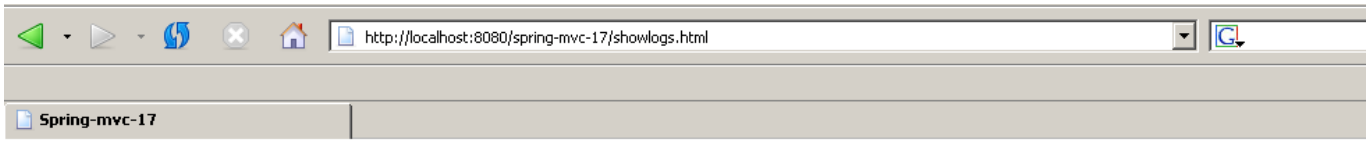
Nous demandons l'url [http://localhost:8080/spring-mvc-17/showlogs.html?langue=en] :



List of logs (en)

1. (istia.st.springmvc.exemples.web.MyHandlerInterceptor) preHandle : heure=20/03/2006 15:35:32:012, client=127.0.0.1:1211, requête=langue=en, contrôleur=istia.st.springmvc.exemples.web.ShowLogs
2. (istia.st.springmvc.exemples.web.ShowLogs) handleRequest : heure=20/03/2006 15:35:32:012, client=127.0.0.1:1211, requête=langue=en
3. (istia.st.springmvc.exemples.web.MyHandlerInterceptor) postHandle : heure20/03/2006 15:35:32:022, client=127.0.0.1:1211, requête=langue=en, contrôleur=istia.st.springmvc.exemples.web.ShowLogs, vue=showlogs, clés du modèle=[(logs)(something)]

Nous obtenons la version anglaise des logs alors même que notre navigateur a le français (fr) comme langue préférée. Redemandons la même url sans préciser la langue maintenant :

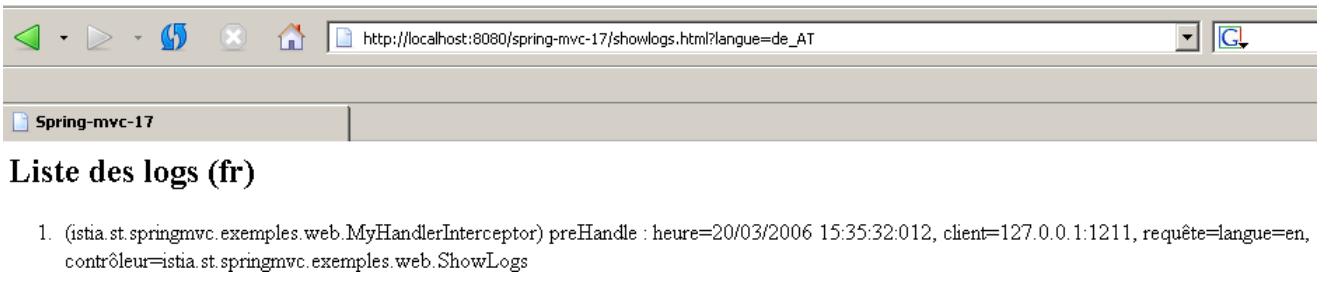


List of logs (en)

1. (istia.st.springmvc.exemples.web.MyHandlerInterceptor) preHandle : heure=20/03/2006 15:35:32:012, client=127.0.0.1:1211, requête=langue=en, contrôleur=istia.st.springmvc.exemples.web.ShowLogs
2. (istia.st.springmvc.exemples.web.ShowLogs) handleRequest : heure=20/03/2006 15:35:32:012, client=127.0.0.1:1211, requête=langue=en
3. (istia.st.springmvc.exemples.web.MyHandlerInterceptor) postHandle : heure20/03/2006 15:35:32:022, client=127.0.0.1:1211, requête=langue=en, contrôleur=istia.st.springmvc.exemples.web.ShowLogs, vue=showlogs, clés du modèle=[(logs)(something)]
4. (istia.st.springmvc.exemples.web.MyHandlerInterceptor) afterCompletion : heure=20/03/2006 15:35:39:333, client=127.0.0.1:1211, requête=langue=en, contrôleur=istia.st.springmvc.exemples.web.ShowLogs
5. (istia.st.springmvc.exemples.web.MyHandlerInterceptor) preHandle : heure=20/03/2006 15:36:45:899, client=127.0.0.1:1212, requête=null, contrôleur=istia.st.springmvc.exemples.web.ShowLogs
6. (istia.st.springmvc.exemples.web.ShowLogs) handleRequest : heure=20/03/2006 15:36:45:899, client=127.0.0.1:1212, requête=null
7. (istia.st.springmvc.exemples.web.MyHandlerInterceptor) postHandle : heure20/03/2006 15:36:45:899, client=127.0.0.1:1212, requête=null, contrôleur=istia.st.springmvc.exemples.web.ShowLogs, vue=showlogs, clés du modèle=[(logs)(something)]

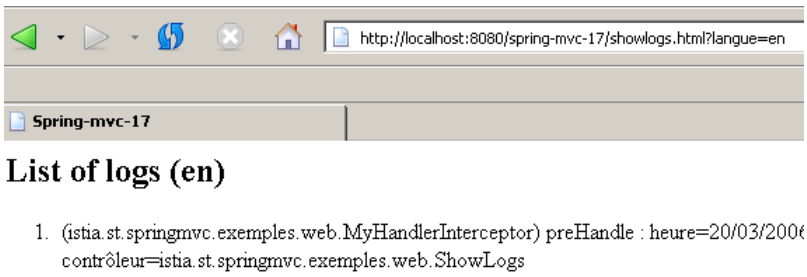
Nous continuons à obtenir la version anglaise des logs. Le choix précédent de la langue a été mémorisé en session. C'est la stratégie [SessionLocaleResolver] qui est ici à l'oeuvre.

Demandons la version allemande d'Autriche (de_AT) des logs :



On retrouve une anomalie déjà signalée dans la partie 1 de l'article. Le titre [Liste des logs (fr)] montre que c'est la page JSP [showlogs_fr.jsp] qui a été affichée et non la page [showlogs.jsp] qu'on attendait. Celle-ci a pour titre [Liste des logs (defaut)]. Comme dans la partie 1 de l'article, nous n'avons pas d'explication à proposer pour cette anomalie.

Maintenant changeons la langue en anglais :

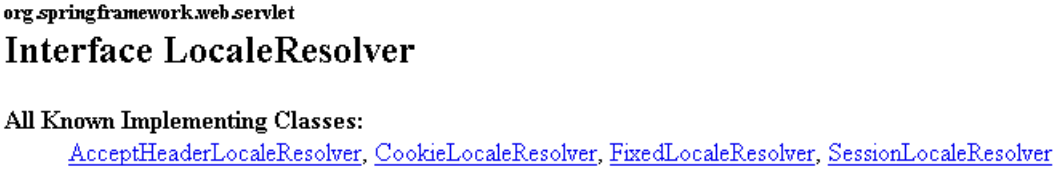


Fermons le navigateur, rouvrons-le et demandons la même Url mais sans préciser la langue :

Nous sommes revenus en français. La session en anglais a été perdue lorsqu'on a fermé le navigateur. Comme nous ne précisons pas ci-dessus le paramètre [langue], c'est l'entête HTTP " accept-language " qui a servi à fixer la langue, le français dans notre cas. La stratégie [CookieLocaleResolver] permet de garder le souvenir du choix de la langue.

3.3 La stratégie de localisation [CookieLocaleResolver]

Revenons sur l'interface [LocaleResolver] :



Parmi les classes d'implémentation de l'interface [**LocaleResolver**], il y a la classe [**CookieLocaleResolver**] :

org.springframework.web.servlet.i18n

Class CookieLocaleResolver

java.lang.Object
  └ org.springframework.web.util.CookieGenerator
    └ **org.springframework.web.servlet.i18n.CookieLocaleResolver**

All Implemented Interfaces:

LocaleResolver

Cette classe permet de mémoriser la langue choisie par l'utilisateur dans un cookie qui sera stocké sur l'ordinateur client (s'il autorise les cookies). Cela permet de garder le souvenir de la langue choisie par l'utilisateur et de l'utiliser dès la première page de sa prochaine utilisation de l'application.

La classe [CookieLocaleResolver] dérive de la classe [CookieGenerator] qui sert à gérer les cookies envoyés au client. L'une des méthodes de la classe [CookieGenerator] est [setCookieMaxAge] qui permet de fixer la durée de vie du cookie :

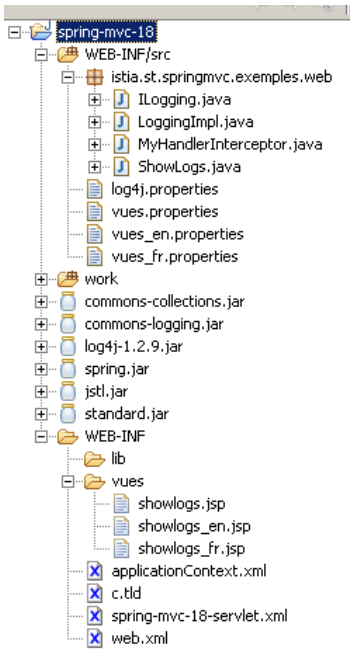
setCookieMaxAge

```
public void setCookieMaxAge(int cookieMaxAge)
```

Use the given maximum age (in seconds) for cookies created by this generator.

On voit que cette méthode est un *setter*, ce qui va nous permettre de fixer la durée de vie du cookie par configuration Spring. Cette durée est exprimée en secondes. Lorsque l'utilisateur quitte l'application à la date / heure T1, le cookie est stocké localement sur l'ordinateur du client. S'il réutilise l'application à la date / heure T2, le navigateur renverra le cookie à l'application si T2-T1<[durée de vie du cookie]. Alors l'application reprendra la langue mémorisée dans le cookie. Si T2-T1>[durée de vie du cookie], le navigateur ne renverra pas le cookie à l'application qui utilisera alors la langue de l'entête HTTP " accept-language " pour fixer la langue de la première page envoyée au client.

Le projet Eclipse utilisant la stratégie de localisation [CookieLocaleResolver] est le suivant :



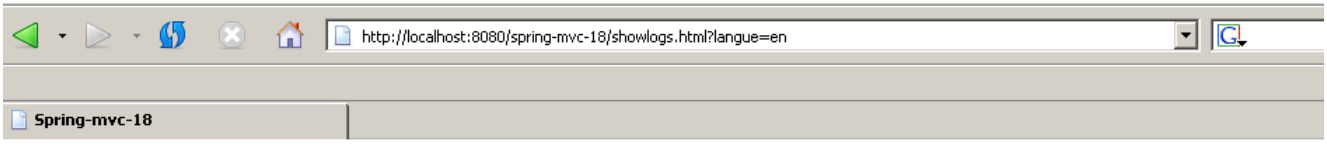
Ce projet est identique au précédent, sauf en ce qui concerne la stratégie de localisation. La configuration [spring-mvc-18-servlet.xml] est la suivante :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les résolveurs de localisation -->
5.    <bean id="localeResolver" class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
6.      <property name="cookieMaxAge">
7.        <value>1000</value>
8.      </property>
9.    </bean>
```

```
10.  <!-- les mappings de l'application-->
11.  <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
12.    <property name="interceptors">
13.      <list>
14.        <bean class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
15.          <property name="paramName">
16.            <value>langue</value>
17.          </property>
18.        </bean>
19.        <bean class="istia.st.springmvc.exemples.web.MyHandlerInterceptor">
20.          <property name="logger">
21.            <ref bean="logger"/>
22.          </property>
23.        </bean>
24.      </list>
25.    </property>
26.    <property name="mappings">
27.      <props>
28.        <prop key="/showlogs.html">ShowLogsController</prop>
29.      </props>
30.    </property>
31.  </bean>
32.  <!-- les contrôleurs de l'application-->
33.  <bean id="ShowLogsController"
34.    class="istia.st.springmvc.exemples.web.ShowLogs">
35.    <property name="logger">
36.      <ref bean="logger"/>
37.    </property>
38.  </bean>
39.  <!-- le résolveur de vues -->
40.  <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
41.    <property name="basename">
42.      <value>vues</value>
43.    </property>
44.  </bean>
45. </beans>
```

La stratégie de localisation définie lignes 5-9 est désormais [CookieLocaleResolver] avec une durée de vie du cookie de 1000 secondes.

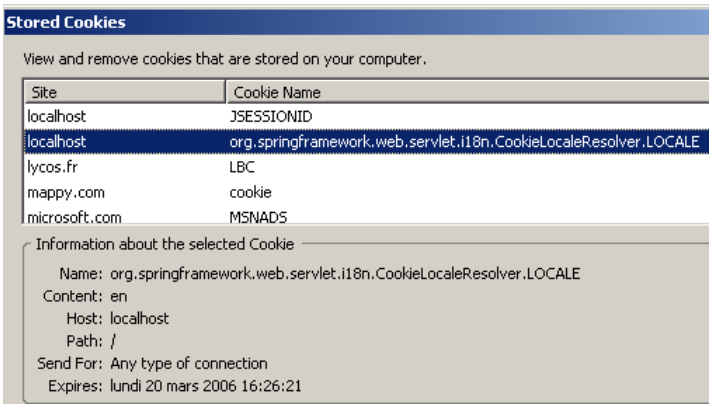
Nous pouvons faire les tests. Nous demandons l'url [http://localhost:8080/spring-mvc-18/showlogs.html?langue=en] :



List of logs (en)

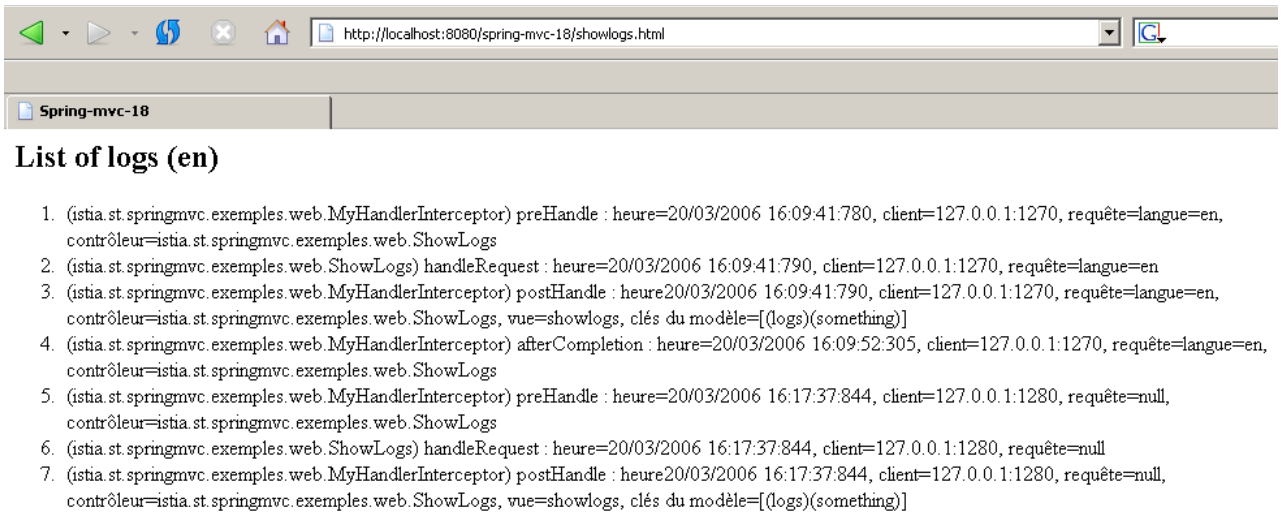
1. (istia.st.springmvc.exemples.web.MyHandlerInterceptor) preHandle : heure=20/03/2006 16:09:41:780, client=127.0.0.1:1270, requête=langue=en, contrôleur=istia.st.springmvc.exemples.web.ShowLogs
2. (istia.st.springmvc.exemples.web.ShowLogs) handleRequest : heure=20/03/2006 16:09:41:790, client=127.0.0.1:1270, requête=langue=en
3. (istia.st.springmvc.exemples.web.MyHandlerInterceptor) postHandle : heure20/03/2006 16:09:41:790, client=127.0.0.1:1270, requête=langue=en, contrôleur=istia.st.springmvc.exemples.web.ShowLogs, vue=showlogs, clés du modèle=[(logs)(something)]

Nous avons obtenu les logs en anglais. Avec le navigateur FireFox, il est possible d'examiner les cookies reçus (Tools / Options / Privacy / Strored Cookies). Celui envoyé par l'application est le suivant :



Les logs montrent que la requête a eu lieu à 16 h 09 mn 41 s. Ci-dessus, on voit que le cookie est valide jusqu'à 16 h 26 mn 21 s. Le lecteur pourra vérifier que 1000 secondes séparent ces deux moments.

Maintenant fermons le navigateur et rouvrons-le pour demander la même url mais sans le paramètre langue :



Nous obtenons bien la page en anglais grâce au cookie qui a été renvoyé par le navigateur.

4 Les gestionnaires d'exceptions

Lors du traitement d'une requête, il peut se produire des exceptions. Il y a plusieurs façons de les gérer :

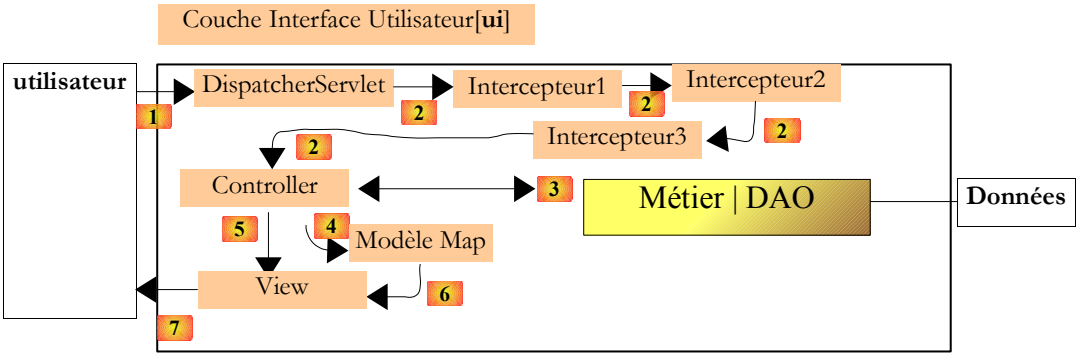
- on les gère explicitement dans le code. Ce sera obligatoire si ces exceptions sont contrôlées c.a.d. qu'elles ne sont pas dérivées de la classe [RuntimeException].
- on les laisse remonter jusqu'à [DispatcherServlet] sans s'en préoccuper. Ce ne sera possible qu'avec des execptions non contrôlées dérivées de la classe [RuntimeException]. Par configuration, on indique à [DispatcherServlet] les vues associées aux différentes exceptions possibles. Cette seconde méthode a plusieurs mérites :
  - elle décharge les intercepteurs et les contrôleurs de la gestion de certaines exceptions
  - on peut changer les vues associées aux exceptions sans toucher au code Java

Les classes implémentant le lien entre les exceptions et les vues implémentent l'interface [HandlerExceptionResolver] :

L'unique méthode de l'interface est [**resolveException**] :

Method Summary	
ModelAndView	**resolveException**(HttpServletRequest request, HttpServletResponse response, Object handler, Exception ex) Try to resolve the given exception that got thrown during on handler execution, returning a ModelAndView that represents a specific error page if appropriate.

Il faut resituer le contexte de gestion de l'exception. Revenons sur le parcours de traitement d'une requête :



[DispatcherServlet] a reçu une requête qu'il fait traiter par une chaîne d'intercepteurs et un contrôleur. Au cours de ce traitement se produit une exception qui remonte jusqu'à [DispatcherServlet], soit par volonté délibérée des développeurs, soit parce que l'exception est non gérée. Dans ce cas, [DispatcherServlet] va regarder si l'application a défini un résolveur d'exceptions, c.a.d une classe capable de générer l'objet [ModelAndView] dont a besoin [DispatcherServlet] pour envoyer la réponse au client.

Le résolveur d'exceptions doit implémenter l'interface [HandlerExceptionResolver] ci-dessus et son unique méthode [resolveException]. Celle-ci reçoit en paramètres :

- la requête [request] en cours de traitement
- la réponse [response] qui devra être utilisée pour envoyer la réponse
- l'objet [handler] qui était chargé de traiter la requête, une fois que les intercepteurs ont fait leur travail
- l'exception [exception] qui s'est produite.

Avec ces informations, la classe d'implémentation doit renvoyer un objet [ModelAndView] à [DispatcherServlet] qui va l'afficher.

Spring propose une unique classe d'implémentation de l'interface [HandlerExceptionResolver], la classe [**SimpleMappingExceptionResolver**] :

org.springframework.web.servlet.handler

Class SimpleMappingExceptionResolver

java.lang.Object
  └ **org.springframework.web.servlet.handler.SimpleMappingExceptionResolver**

All Implemented Interfaces:

Ordered, HandlerExceptionResolver

Ci-dessus, on voit qu'outre l'interface [HandlerExceptionResolver], la classe implémente l'interface [Ordered]. Cette interface permet d'avoir une chaîne ordonnée de gestionnaires d'exceptions, c.a.d. que les gestionnaires de la chaîne sont exécutés selon un ordre qui peut être défini par configuration.

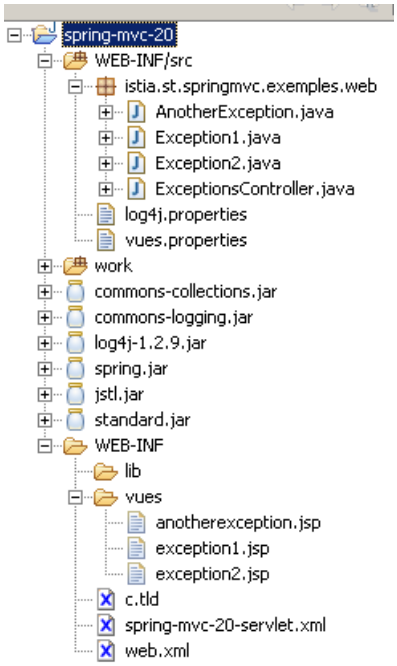
Outre les méthodes des interfaces [Ordered] et [HandlerExceptionResolver], la classe présente quelques méthodes de type *setter* qui permettent sa configuration par Spring :

void	**setDefaultErrorView**(String defaultErrorView) Set the name of the default error view.
void	**setDefaultStatusCode**(int defaultStatusCode) Set the default HTTP status code that this exception resolver will apply if it resolves an error view.
void	**setExceptionAttribute**(String exceptionAttribute) Set the name of the model attribute as which the exception should be exposed.
void	**setExceptionMappings**(Properties mappings) Set the mappings between exception class names and error view names.

`setExceptionMappings`	permet de lier un type d'exception à une vue spécifiée par son nom
`setDefaultErrorView`	permet de lier une exception non définie dans [exceptionMappings] à une vue spécifiée par son nom. Si ce paramètre n'est pas défini et que [DispatcherServlet] ne peut associer une vue à l'exception qu'elle a reçue alors l'exception remontera jusqu'au serveur web qui généralement envoie une page d'erreurs au client avec le détail de l'exception.

`setDefaultStatusCode`	permet de définir le code HTTP qui sera renvoyé au client au cas où le résolveur d'exceptions lie une exception à une vue. Pas de valeur par défaut.
`setExceptionAttribute`	fixe la clé qui sera placée dans le modèle du [ModelAndView] rendu par le gestionnaire d'exceptions et qui sera associée à l'exception. Valeur par défaut : **exception**

Pour illustrer l'usage des gestionnaires d'exceptions, nous utiliserons le projet Eclipse suivant :

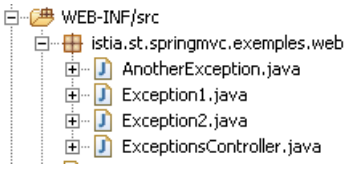


L'application est configurée par [spring-mvc-20-servlet.xml] :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les mappings de l'application-->
5.    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
6.      <property name="mappings">
7.        <props>
8.          <prop key="/exceptions.html">ExceptionsController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="ExceptionsController"
14.     class="istia.st.springmvc.exemples.web.ExceptionsController"/>
15.   <!-- le résolveur de vues -->
16.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
17.     <property name="basename">
18.       <value>vues</value>
19.     </property>
20.   </bean>
21.   <!-- les gestionnaires d'exceptions -->
22.   <bean
23.     class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
24.     <property name="exceptionMappings">
25.       <props>
26.         <prop key="Exception1">exception1</prop>
27.         <prop key="Exception2">exception2</prop>
28.       </props>
29.     </property>
30.     <property name="exceptionAttribute">
31.       <value>monexception</value>
32.     </property>
33.     <property name="defaultStatusCode">
34.       <value>500</value>
35.     </property>
36.     <property name="defaultErrorView">
37.       <value>anotherexception</value>
38.     </property>
39.   </bean>
40. </beans>
```

- lignes 5-11 : l'unique URL gérée est [**/exceptions.html**]. Elle sera traitée par le contrôleur [ExceptionsController] défini lignes 13-14.
- lignes 16-20 : les vues seront définies dans un fichier [**vues.properties**]
- lignes 22-39 : définissent le gestionnaire d'exceptions, ici de type [**SimpleMappingExceptionResolver**]
- ligne 26 : les exceptions de type [**Exception1**] seront traitées par la vue nommée " **exception1** "
- ligne 27 : les exceptions de type [**Exception2**] seront traitées par la vue nommée " **exception2** "
- lignes 36-38 : les exceptions d'un autre type que [Exception1, Exception2] seront traitées par la vue nommée " **anotherexception** "
- lignes 30-32 : le gestionnaire d'exceptions mettra l'exception qui s'est produite dans le modèle qu'il va rendre à [DispatcherServlet] associée à la clé "**monexception**"
- lignes 33-35 : en cas d'exception résolue par le gestionnaire d'exceptions (c.a.d. qu'il y avait une vue prévue pour l'exception), le code HTTP **500** sera envoyé au client.

Les classes référencées par ce fichier de configuration sont définies dans [WEB-INF/src) :



[Exception1] est une classe d'exceptions non contrôlées dérivée de [RuntimeException] :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Exception1 extends RuntimeException {
4.
5.    public Exception1() {
6.      super();
7.    }
8.
9.    public Exception1(String message) {
10.     super(message);
11.   }
12.
13.   public Exception1(String message, Throwable cause) {
14.     super(message, cause);
15.   }
16.
17.   public Exception1(Throwable cause) {
18.     super(cause);
19.   }
20.
21. }
```

La classe n'ajoute rien à sa classe parente. Elle sert simplement à définir un nouveau type d'exceptions. Les classes [exception2] et [AnotherException] sont identiques au nom de la classe près. On a donc défini trois types d'exceptions.

Le contrôleur [ExceptionsController] est le contrôleur chargé de traiter la requête liée à l'url [/exceptions.html] :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Random;
4.
5.  import javax.servlet.http.HttpServletRequest;
6.  import javax.servlet.http.HttpServletResponse;
7.
8.  import org.springframework.web.servlet.ModelAndView;
9.  import org.springframework.web.servlet.mvc.Controller;
10.
11. public class ExceptionsController implements Controller {
12.
13.   private Random random = new Random();
14.
15.   public ModelAndView handleRequest(HttpServletRequest request,
16.       HttpServletResponse response) throws Exception {
17.     int n = random.nextInt(3);
18.     switch (n) {
19.     case 0:
20.       throw new Exception1(
21.           "L'exception [Exception1] s'est produite dans "
22.               + this.getClass().getName());
23.     case 1:
24.       throw new Exception2(
25.           "L'exception [Exception2] s'est produite dans "
26.               + this.getClass().getName());
```

```
27.    default:
28.      throw new AnotherException(
29.          "L'exception [AnotherException] s'est produite dans "
30.              + this.getClass().getName());
31.    }
32.  }
33.
34. }
```

- ligne 11 : la classe implémente l'interface [Controller] et donc son unique méthode [handleRequest] (lignes 15-32)
- ligne 13 : un générateur de nombres aléatoires est initialisé lors de l'instanciation du contrôleur
- ligne 17 : un nombre entier aléatoire est tiré dans l'intervalle [0,2]. Il va servir à déterminer le type d'exception que va lancer le contrôleur
- lignes 19-22 : une exception de type [Exception1] est lancée avec un message permettant d'identifier le type de l'exception.
- lignes 24-26 : idem avec le type [Exception2]
- lignes 19-22 : idem avec le type [AnotherException]

Au final, le contrôleur ne rend aucun [ModelAndView] et lance systématiquement une exception. Celle-ci va donc remonter à [DispatcherServlet] qui va alors utiliser le résolveur d'exceptions configuré par l'application. Nous avons vu que celle-ci associait :

- la vue nommée " exception1 " aux exceptions de type [Exception1]
- la vue nommée " exception2 " aux exceptions de type [Exception2]
- la vue nommée " anotherexception " aux exceptions d'un autre type que les précédents

Dans le fichier [**vues.properties**], ces trois vues sont définies comme suit :

```
1.  #exception1
2.  exception1.class=org.springframework.web.servlet.view.JstlView
3.  exception1.url=/WEB-INF/vues/exception1.jsp
4.  #exception2
5.  exception2.class=org.springframework.web.servlet.view.JstlView
6.  exception2.url=/WEB-INF/vues/exception2.jsp
7.  #anotherexception
8.  anotherexception.class=org.springframework.web.servlet.view.JstlView
9.  anotherexception.url=/WEB-INF/vues/anotherexception.jsp
```

Elles sont toutes les trois associées à des pages JSP. Celles-ci sont les suivantes :

[**exception1.jsp**]

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-20</title>
8.    </head>
9.    <body>
10.     <h2>Exception1.jsp</h2>
11.     Une exception de type [Exception1] s'est produite :
12.     <c:out value="${monexception}"/>
13.   </body>
14. </html>
```

[**exception2.jsp**]

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-20</title>
8.    </head>
9.    <body>
10.     <h2>Exception2.jsp</h2>
11.     Une exception de type [Exception2] s'est produite :
12.     <c:out value="${monexception}"/>
13.   </body>
14. </html>
15.
```

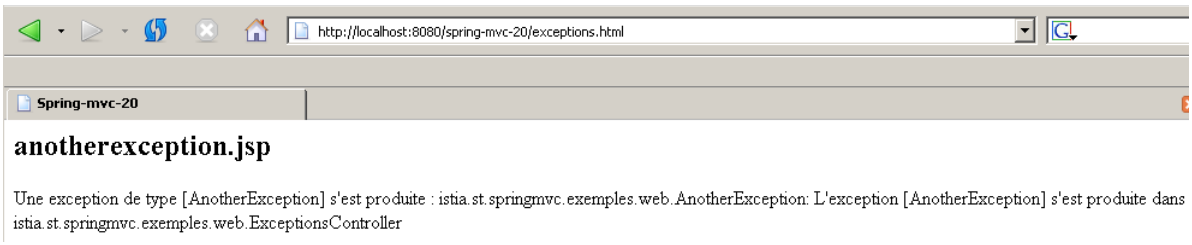
[**anotherexception.jsp**]

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
```

```
5.  <html>
6.    <head>
7.      <title>Spring-mvc-20</title>
8.    </head>
9.    <body>
10.     <h2>anotherexception.jsp</h2>
11.     Une exception de type [AnotherException] s'est produite :
12.     <c:out value="${monexception}"/>
13.   </body>
14. </html>
```

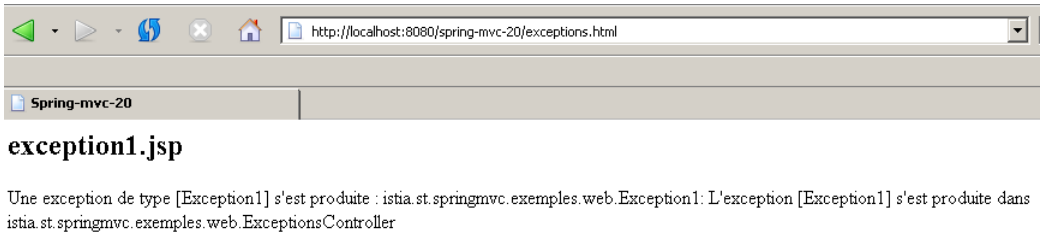
Chaque page affiche l'exception (ligne 12) ainsi qu'un titre indiquant la page JSP affichée (ligne 10). Ainsi nous saurons si l'association [Exception] <-> [Vue] a été faite correctement.

Nous sommes prêts pour les tests. Nous demandons l'url [http://localhost:8080/spring-mvc-20/exceptions.html] :



Ci-dessus, le message indique que l'exception qui s'est produite est de type [AnotherException]. Le titre indique que la page JSP utilisée pour cet affichage est [anotherexception.jsp]. La relation Exception <-> Vue s'est faite comme attendue.

En rechargeant plusieurs fois la page, on obtient au bout d'un moment les autres vues, par exemple la vue " exception1 " :



Un plugin du navigateur Firefox permet de suivre les échanges HTTP entre le navigateur Firefox et le serveur. Ce plugin appelé [LiveHttpHeaders] est disponible à l'url [http://livehttpheaders.mozdev.org/]. Voici un échange HTTP :

```
1.  http://localhost:8080/spring-mvc-20/exceptions.html
2.
3.  GET /spring-mvc-20/exceptions.html HTTP/1.1
4.  Host: localhost:8080
5.  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.7) Gecko/20040614 Firefox/0.9
6.  Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
7.  Accept-Language: fr,en;q=0.7,de;q=0.3
8.  Accept-Encoding: gzip,deflate
9.  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
10. Keep-Alive: 300
11. Connection: keep-alive
12. Cookie: JSESSIONID=146E5A3F58C6E773849D86B70EC2078A
13.
14. HTTP/1.x 500 Erreur Interne de Servlet
15. Content-Type: text/html;charset=ISO-8859-1
16. Content-Language: fr
17. Content-Length: 332
18. Date: Tue, 21 Mar 2006 14:28:48 GMT
19. Server: Apache-Coyote/1.1
20. Connection: close
```

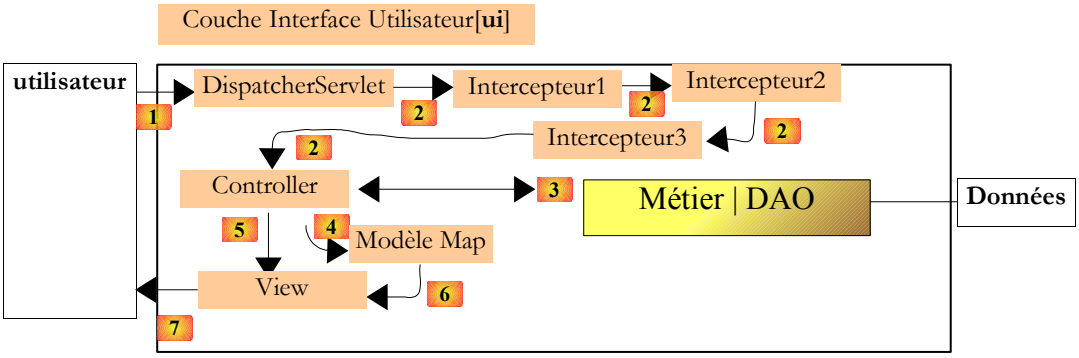
- ligne 1 : l'URL demandée
- lignes 3-12 : les lignes HTTP envoyées par le navigateur
- lignes 14-20 : les lignes HTTP envoyées par le serveur

Ligne 14, on voit que le serveur a envoyé le code HTTP 500, celui qui avait été demandé par configuration dans le cas d'une exception.

5 Gestion des formulaires

5.1 La classe [SimpleFormController]

Revenons sur l'architecture d'une application Spring MVC :



Une requête du client est traitée par d'éventuels intercepteurs puis par un objet implémentant l'interface [Controller] :

org.springframework.web.servlet.mvc

Interface Controller

All Known Implementing Classes:

AbstractCommandController, AbstractController, AbstractFormController, AbstractUrlViewController, AbstractWizardFormController, BaseCommandController, BurlapServiceExporter, CancellableFormController, HessianServiceExporter, HttpInvokerServiceExporter, MultiActionController, ParameterizableViewController, ServletForwardingController, ServletWrappingController, SimpleFormController, UrlFilenameViewController

Nous avons à plusieurs reprises implémenté nous-mêmes l'interface [Controller]. Spring offre plusieurs classes implémentant cette interface. Comme le code d'un contrôleur est spécifique à une application, les contrôleurs de Spring sont destinés à être dérivés pour être adaptés à l'application.

La classe [SimpleFormController] que l'on voit ci-dessus est une classe destinée à servir de classe de base aux contrôleurs gérant des formulaires. Sa lignée est la suivante :

org.springframework.web.servlet.mvc

Class SimpleFormController

```
java.lang.Object
  └ org.springframework.context.support.ApplicationObjectSupport
      └ org.springframework.web.context.support.WebApplicationObjectSupport
          └ org.springframework.web.servlet.support.WebContentGenerator
              └ org.springframework.web.servlet.mvc.AbstractController
                  └ org.springframework.web.servlet.mvc.BaseCommandController
                      └ org.springframework.web.servlet.mvc.AbstractFormController
                          └ org.springframework.web.servlet.mvc.SimpleFormController
```

All Implemented Interfaces:

ApplicationContextAware, Controller

[SimpleFormController] tire l'essentiel de son comportement de sa classe de base [AbstractFormController]. Celle-ci est abstraite et ne peut donc être instanciée. Une instance de [SimpleFormController] permet de gérer :

- le GET qui va envoyer au client le formulaire HTML
- le POST qui va envoyer au serveur les données saisies par l'utilisateur

Considérons l'exemple suivant :

Si on utilise le plugin [LiveHttpHeaders], on découvre que la page ci-dessus a été obtenue avec un GET :

```
1.  GET /spring-mvc-22/formulaire.html HTTP/1.1
2.  Host: localhost:8080
3.  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.7) Gecko/20040614 Firefox/0.9
4.  Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
5.  Accept-Language: fr,en;q=0.7,de;q=0.3
6.  Accept-Encoding: gzip,deflate
7.  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
8.  Keep-Alive: 300
9.  Connection: keep-alive
10. Cookie: JSESSIONID=4C07DC1D2F1C49B5865D20A1D3AA3DD1
```

Le code HTML de la page est le suivant :

```
1.  <html>
2.    <head>
3.     <title>formulaire</title>
4.    </head>
5.    <body>
6.     <h3>Formulaire Spring : boutons radio</h3>
7.      <hr>
8.      <form method="post">
9.        <table border="0">
10.        <!-- boutons radio -->
11.         <tr>
12.           <td>Boutons radio</td>
13.           <td>
14.             <input type="radio" name="opt" value="oui">Oui
15.             <input type="radio" name="opt" value="non" checked>Non
16.           </td>
17.         </tr>
18.       </table>
19.      <input type="submit" value="Envoyer">
20.     </form>
21.   </body>
22. </html>
```

- ligne 8 : on voit que le formulaire sera posté (method= "post "). Comme l'attribut " **action** " est absent, il sera posté à la même Url que le GET.

Remplissons le formulaire et faisons le POST :

Lorsque le bouton [Envoyer] est cliqué, le navigateur envoie le flux HTTP suivant :

```
1.  POST /spring-mvc-22/formulaire.html HTTP/1.1
2.  Host: localhost:8080
3.  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.7) Gecko/20040614 Firefox/0.9
4.  Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
5.  Accept-Language: fr,en;q=0.7,de;q=0.3
6.  Accept-Encoding: gzip,deflate
```

```
7.  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
8.  Keep-Alive: 300
9.  Connection: keep-alive
10. Referer: http://localhost:8080/spring-mvc-22/formulaire.html
11. Cookie: JSESSIONID=4C07DC1D2F1C49B5865D20A1D3AA3DD1
12. Content-Type: application/x-www-form-urlencoded
13. Content-Length: 7
14.
15. opt=oui
```

- ligne 1 : le navigateur fait un POST vers la même URL vers laquelle précédemment il avait fait un GET
- ligne 15 : les valeurs postées

Nous donnons ci-dessous, le flux d'exécution associé à ces deux phases de traitement d'un formulaire, lorsque celui-ci est traité par un contrôleur [SimpleFormController] ou dérivé. La description ci-dessous est tirée de la documentation JavaDoc des classes [AbstractController] et [SimpleFormController]. Elle à la fois précise et technique. Le lecteur la trouvera probablement indigeste mais nous nous appuierons sur elle pour comprendre les cas simples que allons construire ultérieurement.

1. le contrôleur reçoit une requête de type **GET** pour le formulaire
2. la méthode [**formBackingObject**] de la classe de base [AbstractFormController] est appelée. Si elle n'a pas été redéfinie, elle rend une instance de type [**commandClass**] défini en général par configuration. Si elle a été redéfinie, elle doit rendre un objet dont les propriétés seront utilisées pour initialiser le formulaire. Par la suite, nous appellerons **command** l'objet créé à cette étape. Si l'attribut " **sessionForm** " de [SimpleFormController] est à *true*, l'objet **command** est mis en session.
3. la méthode [**initBinder**] de la classe de base [BaseCommandController] est appelée. Si redéfinie, cette méthode permet de déclarer des éditeurs de propriétés. Ces objets permettent de faire des conversions entre le type non String de certaines des propriétés de l'objet [**command**] et le flux HTTP de type String échangé entre le client et le serveur. Par exemple, une date dans un formulaire est envoyée comme une chaîne de caractères au serveur. L'éditeur de propriétés va permettre la trasformation de ce type [String] en un type [java.util.Date] de la propriété de [**command**] qui doit être initialisée par la chaîne.
4. la méthode [**showForm**] de [SimpleFormController] est appelée. C'est elle qui va rendre l'objet [ModelAndView] attendu par [DispatcherServlet]. L'objet [**command**] construit en (2) sera placé dans le modèle sous un nom [**commandName**] généralement fixé par configuration. Par ailleurs, [showForm] va appeler la méthode [**referenceData**] de [SimpleFormController]. Cette méthode rend un dictionnaire [Map] qui est ajouté au modèle. L'implémentation par défaut de [SimpleFormController] rend la référence *null*. Si on veut dans le modèle, d'autres éléments que l'objet de type [**command**] construit en (2), on redéfinira la méthode [referenceData]. La méthode [showForm] rend un objet [ModelAndView] où le nom de la vue est celui défini par l'attribut " **formView** " dans la configuration du contrôleur.
5. maintenant que le [ModelAndView] a été rendu à [DispatcherServlet], celui-ci fait ce qu'il faut pour envoyer le formulaire au client.

L'utilisateur va saisir des données dans le formulaire et poster celles-ci :

1. le contrôleur [SimpleFormController] reçoit une requête de type **POST**
2. si l'attribut " **sessionForm** " de [SimpleFormController] est à *false*, la méthode [**formBackingObject**] de la classe de base [AbstractFormController] est appelée pour créer l'objet **command** qui va servir à enregistrer les données du POST. Si " sessionForm " est à *true*, alors l'objet **command** en question est récupéré dans la session (cf étape 2 du GET).
3. les valeurs postées sont affectées aux champs de même nom de l'objet **command**. Si d'éventuels éditeurs de propriétés avaient été définis à l'étape 3 du GET, ils sont utilisés pour les conversions String -> Type, par exemple java.lang.String -> java.util.Date.
4. la méthode **onBind(**HttpServletRequest request, Object command, BindException Errors**)** est appelée. Elle ne fait rien par défaut. Elle peut être redéfinie pour faire par exemple les conversions String -> Type à la main.
5. si l'attribut **validateOnBinding** de [SimpleFormController] a la valeur *true*, un objet de type [Validator] sera utilisé pour vérifier la validité des données.
6. la méthode **onBindAndValidate**(HttpServletRequest request, Object command, BindException Errors) est appelée. Elle ne fait rien par défaut. Elle peut être redéfinie pour faire par exemple les validations sans objet [Validator].
7. la méthode **processFormSubmission(**HttpServletRequest request, HttpResponse response, Object command, BindException Errors**)** est appelée. Elle vérifie la liste des erreurs contenu dans l'objet [**Errors**] mis à jour par les méthodes 4, 5 et 6 précédentes. Si cette liste d'erreurs est non vide, la méthode [showForm] est appelée. Comme pour le GET, elle rend un objet [ModelAndView] où le nom de la vue est celui défini par l'attribut " **formView** " dans la configuration du contrôleur. Le modèle lui, comprend l'objet **command** initialisé par le POST à l'étape 3 précédente ainsi que l'objet [**Errors**]. L'utilisateur récupère donc le formulaire tel qu'il l'a posté avec de plus les erreurs interceptées par le serveur. Cette méthode n'est normalement pas redéfinie.
8. lorsqu'il n'y a pas d'erreurs, la méthode **processFormSubmission** appelle la méthode **ModelAndView onSubmit(**HttpRequest request, HttpResponse response, Object command, BindException Errors**)** qui par défaut appelle la méthode **ModelAndView onSubmit(**Object command, BindException Errors**)** qui par défaut appelle la méthode **ModelAndView onSubmit(**Object command**)** qui par défaut appelle la méthode **void doSubmitAction(**Object command**)**. Le traitement des valeurs du POST étant spécifique au formulaire, on sera amené le plus souvent à redéfinir l'une des quatre méthodes précédentes. Le choix se fera sur les paramètres qu'on veut utiliser. On peut se demander quelle est la différence entre les deux dernières méthodes qui toutes deux reçoivent l'objet **command** construit à partir des

valeurs postées. La différence dans la valeur rendue par la méthode. Avec **onSubmit**, on peut construire nous-mêmes le [ModelAndView] qu'on va rendre à [DispatcherServlet]. Avec **doSubmitAction**, on ne rend rien. Un objet [ModelAndView] par défaut est construit par la méthode appelante [onSubmit] avec pour nom de vue, l'attribut " **successView** " du contrôleur et pour modèle un objet [Map] contenant l'objet [**command**].

9. [DispatcherServlet] envoie au client le [ModelAndView] que **processFormSubmission** lui a rendu.

Le lecteur est sans doute un peu abasourdi par l'apparente complexité des explications précédentes. Nous allons voir que dans la pratique les choses sont souvent plus simples. Nous allons construire une série de formulaires sur le modèle suivant :

- le formulaire contiendra un unique élément de saisie (boutons radio, cases à cocher, liste, ...)
- le formulaire sera demandé par un GET et affiché pré-initialisé. Nous analyserons alors le flux d'exécution du GET qui a amené cet affichage.
- l'utilisateur modifiera la valeur de l'élément de saisie et postera le formulaire (POST)
- une page de confirmation affichant la valeur saisie lui sera renvoyée en retour. Nous analyserons le flux d'exécution du POST qui a amené cet affichage.

Nous commençons par un formulaire dont l'élément de saisie est constitué de boutons radio.

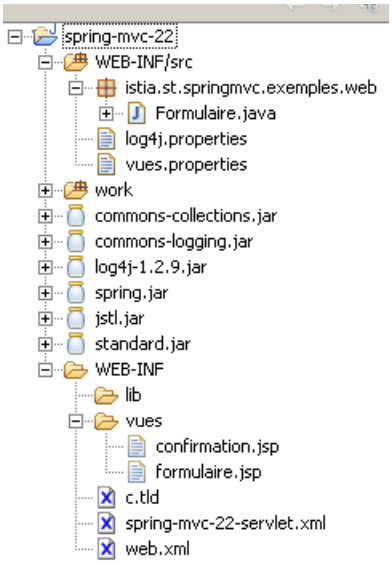
5.2 Formulaire de saisie avec boutons radio

Le formulaire affiché à l'issue du GET sera le suivant :

L'utilisateur fait sa saisie :

Une page de confirmation lui est envoyée :

Le projet Eclipse est le suivant :



Tous nos exemples auront l'architecture ci-dessus :

- [Formulaire.java] : une classe ayant des attributs portant le même nom que les éléments de saisie HTML de la page JSP du formulaire [formulaire.jsp]. Une instance de cette classe est créée au moment du GET. Elle servira à alimenter les champs de la page [formulaire.jsp]. L'instance créée sera mise en session pour être réutilisée au moment du POST. Au moment du POST, l'instance mise en session sera récupérée et mise à jour par les valeurs postées. Lors de cette mise à jour, la valeur d'un paramètre posté est affecté au champ de même nom de l'instance [Formulaire].
- [formulaire.jsp] : présentera le formulaire HTML tel qu'initialisé par l'instance [Formulaire]
- [confirmation.jsp] : affichera les champs de l'instance [Formulaire] une fois que le POST a été réalisé. Cette page reflète donc les valeurs saisies par l'utilisateur.
- [spring-mvc-xx-servlet.xml] : le fichier de configuration de l'application

Le fichier [spring-mvc-22-servlet.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les mappings de l'application-->
5.    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
6.      <property name="mappings">
7.        <props>
8.          <prop key="formulaire.html">FormulaireController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="FormulaireController"
14.     class="org.springframework.web.servlet.mvc.SimpleFormController">
15.     <property name="sessionForm">
16.       <value>true</value>
17.     </property>
18.     <property name="commandClass">
19.       <value>istia.st.springmvc.exemples.web.Formulaire</value>
20.     </property>
21.     <property name="commandName">
22.       <value>formulaire</value>
23.     </property>
24.     <property name="formView">
25.       <value>formulaire</value>
26.     </property>
27.     <property name="successView">
28.       <value>confirmation</value>
29.     </property>
30.   </bean>
31.   <!-- le résolveur de vues -->
32.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
33.     <property name="basename">
34.       <value>vues</value>
35.     </property>
36.   </bean>
37. </beans>
```

- lignes 5-11 : l'unique URL traitée par l'application est [**/formulaire.html**]. Elle est traitée par le bean d'id " formulaireController " défini lignes 13-30.

- lignes 32-36 : les vues seront définies dans un fichier [**vues.properties**].
- lignes 13-14 : le contrôleur qui traitera la requête [/formulaire.html] est de type [SimpleFormController]. Comme [/formulaire.html] est la cible à la fois d'un GET et d'un POST, le contrôleur traitera aussi bien la demande initiale du formulaire (GET) que le traitement des valeurs postées (POST).

Afin de comprendre la configuration du contrôleur [SimpleFormController] des lignes 13-30, revenons sur le rôle de ce contrôleur tel qu'il a été décrit pour le GET (page 27) et pour le POST (page 27). La configuration du contrôleur [SimpleFormController] des lignes 13-30 fait que le GET de l'url [/formulaire.html] va être traité de la façon suivante :

1. Le contrôleur [SimpleFormController] reçoit une requête de type **GET** pour le formulaire
2. Une instance de type [**commandClass**] est créée. Celle-ci est définie lignes 18-20. La méthode [**setCommandClass**] de la classe [BaseCommandController], classe parent de [SimpleFormController] est ici utilisée. Dans cette application, comme dans les suivantes, [Formulaire] est une classe dont les propriétés seront utilisées pour initialiser le formulaire. Les noms des propriétés de la classe [Formulaire] correspondent aux noms (attributs **name**) des éléments de saisie du formulaire HTML.
3. Parce que l'attribut " **sessionForm** " de [SimpleFormController] est à *true* (lignes 15-17), l'objet [Formulaire] créé à l'étape précédente est mis en session. La méthode [**setSessionForm**] de la classe [AbstractController], classe parent de [SimpleFormController] est ici utilisée.
4. La méthode [**initBinder**] de la classe de base [BaseCommandController] est appelée. Ici elle ne fera rien.
5. La méthode [**showForm**] de [SimpleFormController] est appelée. C'est elle qui va rendre l'objet [ModelAndView] attendu par [DispatcherServlet].
6. La méthode [showForm] va appeler la méthode [**referenceData**] de [SimpleFormController]. Ici cette méthode n'étant pas redéfinie, elle ne fera rien. La méthode [showForm] rend un objet [**ModelAndView**] où le nom de la vue est celui défini par l'attribut " **formView** " des lignes 24-26 de la configuration de [SimpleFormController]. L'objet de type [Formulaire] construit en (2) sera placé dans le modèle sous le nom fixé par l'attribut " **commandName** " des lignes 21-23 de la configuration de [SimpleFormController]. La méthode [**setCommandName**] de la classe [BaseCommandController], classe parent de [SimpleFormController] est ici utilisée.
7. Maintenant que le [ModelAndView] a été rendu à [DispatcherServlet], celui-ci fait ce qu'il faut pour afficher la vue du formulaire.

Si on a bien suivi, le GET sur l'url [/formulaire.html] va envoyer une vue nommée " **formulaire** " (lignes 24-26) chargée d'afficher un modèle où il n'y aura qu'un élément de clé " **formulaire** " (lignes 21-23) et de valeur une instance de la classe [**Formulaire**] (lignes 18-20). Cette instance sera mise en cache dans la session (lignes 15-17) pour être réutilisée lors du POST.

Examinons maintenant le traitement du POST vers l'url [/formulaire.html] toujours à la lumière ce de qui a été expliqué page 27.

1. Le contrôleur [SimpleFormController] reçoit une requête de type **POST**
2. L'attribut " **sessionForm** " de [SimpleFormController] étant à *true* (lignes 15-17) l'instance [Formulaire] (lignes 18-20) mis en session créé et mis en session lors du GET est récupérée.
3. Les valeurs des paramètres postés sont affectées aux champs de l'instance [Formulaire] ayant les noms des paramètres.
4. La méthode **onBind(**HttpServletRequest request, Object command, BindException Errors**)** est appelée. Non redéfinie, elle ne fera rien.
5. La méthode **onBindAndValidate**(HttpServletRequest request, Object command, BindException Errors) est appelée. Non redéfinie, elle ne fera rien.
6. La méthode **processFormSubmission** est appelée. Elle vérifie la liste des erreurs contenue dans un objet [**Errors**] mis à jour par les méthodes 4, 5 précédentes. Ces deux méthodes n'ayant rien fait, la liste des erreurs sera vide et le traitement va continuer.
7. La méthode **processFormSubmission(**HttpServletRequest request, HttpResponse response, Object command, BindException Errors**)** appelle la méthode **ModelAndView onSubmit(**HttpRequest request, HttpResponse response, Object command, BindException Errors**)** qui par défaut appelle la méthode **ModelAndView onSubmit(**Object command, BindException Errors**)** qui par défaut appelle la méthode **ModelAndView onSubmit(**Object command**)** qui par défaut appelle la méthode **void doSubmitAction(**Object command**)**. Ici, aucune de ces méthodes ne sera redéfinie. C'est au final la méthode prédéfinie **ModelAndView onSubmit(Object command)** qui va envoyer le [ModelAndView] à [DispatcherServlet]. Un objet [ModelAndView] est construit avec pour nom de vue, l'attribut " **successView** " des lignes 27-29 de la configuration de [SimpleFormController]. C'est la méthode [**setSuccessView**] de la classe [SimpleFormController] qui est ici utilisée. Le modèle lui contiendra la clé définie par [**commandName**] (lignes 21-23) associée à l'objet **command** récupéré en session à l'étape 2.
8. [DispatcherServlet] fait afficher le [ModelAndView] que **processFormSubmission** lui a rendu.

Finalement, le POST sur l'url [/formulaire.html] va envoyer une vue nommée "**confirmation**" (lignes 27-29) chargée d'afficher un modèle où il n'y aura qu'un élément de clé " **formulaire** " (lignes 21-23) et de valeur une instance de la classe [**Formulaire**] (lignes 18-20) dont les champs auront pris pour valeurs les valeurs postées.

Le GET affiche une vue appelée une vue nommée " formulaire ", le POST une vue nommée " confirmation ". Ces deux vues auront une instance de [Formulaire] dans leur modèle. Examinons le fichier [**vues.properties**] pour découvrir ce qui se cache derrière ces deux noms de vues :

```
1.  #formulaire
2.  formulaire.class=org.springframework.web.servlet.view.JstlView
3.  formulaire.url=/WEB-INF/vues/formulaire.jsp
4.  #confirmation
5.  confirmation.class=org.springframework.web.servlet.view.JstlView
6.  confirmation.url=/WEB-INF/vues/confirmation.jsp
```

On voit que les deux vues sont associées à des pages JSP. Avant d'examiner celles-ci, découvrons la classe [Formulaire] qui sera dans le modèle des deux vues :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Formulaire {
4.
5.    // valeur bouton radio
6.    private String opt;
7.
8.    public Formulaire() {
9.      // initialisations du formulaire
10.     this.setOpt("non");
11.   }
12.
13.   public String getOpt() {
14.     return opt;
15.   }
16.
17.   public void setOpt(String opt) {
18.     this.opt = opt;
19.   }
20. }
```

Cette classe sera le modèle d'un formulaire HTML ayant un unique élément de saisie, un groupe de deux boutons radio nommé " **opt** " à deux valeurs possibles " **oui** " et " **non** ". La classe [Formulaire] définit donc un champ **opt** de type String. Celui-ci est intialisé à " **non** " par construction. Lors du [GET /formulaire.html], la valeur du champ [opt] de l'instance [Formulaire] sera utilisée pour cocher le bon bouton radio. Implicitement, la méthode [getOpt] sera utilisée. Lors du [POST /formulaire.html], la valeur du paramètre posté " **opt** " sera affectée au champ [**opt**] de l'instance [Formulaire]. Implicitement, la méthode [setOpt] sera utilisée.

La page [formulaire.jsp] sert à afficher la valeur initiale de l'instance [Formulaire] :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.  <html>
5.    <head>
6.      <title>Formulaire Spring : boutons radio</title>
7.    </head>
8.    <body>
9.      <h3>Formulaire Spring : boutons radio</h3>
10.     <hr>
11.     <form method="post">
12.       <table border="0">
13.       <!-- boutons radio -->
14.         <tr>
15.           <td>Boutons radio</td>
16.           <td>
17.           <c:choose>
18.             <c:when test='${formulaire.opt=="oui"}'>
19.               <input type="radio" name="opt" value="oui" checked>Oui
20.               <input type="radio" name="opt" value="non">Non
21.             </c:when>
22.             <c:otherwise>
23.               <input type="radio" name="opt" value="oui">Oui
24.               <input type="radio" name="opt" value="non" checked>Non
25.             </c:otherwise>
26.           </c:choose>
27.           </td>
28.         </tr>
29.       </table>
30.      <input type="submit" value="Envoyer">
31.     </form>
32.   </body>
33. </html>
```

- ligne 11 : on a bien un formulaire dont les valeurs seront postées.
- lignes 17-26 : le groupe de deux boutons radio exclusifs avec le nom (name) " opt "

- ligne 18 : on se rappelle que l'instance [Formulaire] qui est dans le modèle est associée à la clé " **formulaire** ". Donc l'expression " formulaire.opt " est traduite en [Formulaire.getOpt()]. Le test fait ici vise à savoir quel bouton radio cocher en fonction de la valeur du champ **opt** de [Formulaire].
- lignes 18-21 : le groupe de boutons radio avec le bouton radio [Oui] coché
- lignes 23-24 : le groupe de boutons radio avec le bouton radio [Non] coché

On se rappelle que l'instance [Formulaire] a été créée avec la valeur " non " pour le champ **opt**. Le GET aboutit donc à la page suivante :

Le code HTML reçu est le suivant :

```
1.  <html>
2.    <head>
3.     <title>Formulaire Spring : boutons radio</title>
4.    </head>
5.    <body>
6.     <h3>Formulaire Spring : boutons radio</h3>
7.      <hr>
8.      <form method="post">
9.        <table border="0">
10.        <!-- boutons radio -->
11.         <tr>
12.           <td>Boutons radio</td>
13.           <td>
14.            <input type="radio" name="opt" value="oui">Oui
15.            <input type="radio" name="opt" value="non" checked>Non
16.           </td>
17.         </tr>
18.       </table>
19.      <input type="submit" value="Envoyer">
20.     </form>
21.   </body>
22. </html>
```

La page [confirmation.jsp] sert à afficher la valeur de l'instance [Formulaire] après le POST. C'est l'attribut " successView " du contrôleur [SimpleFormController] (lignes 27-29) qui le dit. Son code est le suivant :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Formulaire Spring : boutons radio</title>
8.    </head>
9.    <body>
10.     <h3>Confirmation des donn&eacute;es saisies</h3>
11.     <table border="1">
12.       <tr>
13.         <td>Bouton radio</td>
14.         <td>${formulaire.opt}</td>
15.       </tr>
16.     </table>
17.   </body>
18. </html>
```

- la ligne 14 affiche la valeur du champ opt de l'objet associé à la clé **formulaire**. On sait que celui-ci est l'instance [Formulaire] créée par le GET et modifiée par le POST. La page affiche donc la valeur du bouton radio cochée par l'utilisateur, c.a.d. soit la chaîne " **oui** " (ligne 14 du code HTML du formulaire), soit la chaîne " **non** " (ligne 15).

Ainsi si la saisie de l'utilisateur est la suivante :

le contrôleur [SimpleFormController] renvoie la page qui suit :

Nous invitons le lecteur à jouer lui-même les tests précédents.

Nous avons essayé d'être assez précis dans cet exemple afin que le lecteur ait une vision à peu près correcte de ce qui se passe côté serveur. Dans les exemples qui suivent, bâtis sur le même format que celui-ci, nous serons plus rapides. La configuration des projets [**spring-mvc-xx-servet.xml**] sera identique à celle du projet qui vient d'être étudié, ainsi que les noms de vues. Seuls trois éléments changeront :

- la classe [Formulaire]
- la page [formulaire.jsp] qui affiche la valeur initiale de [Formulaire] et permet d'en changer les valeurs
- la page [confirmation.jsp] qui affiche la nouvelle valeur de [Formulaire]

5.3 Formulaire de saisie avec cases à cocher

Le formulaire affiché à l'issue du GET sera le suivant :

Le formulaire HTML sous-jacent est le suivant :

```
1.  <html>
2.    <head>
3.     <title>formulaire</title>
4.    </head>
5.    <body>
6.     <h3>Formulaire Spring : cases &agrave; cocher</h3>
7.      <hr>
8.      <form method="post">
9.        <table border="0">
10.        <!-- cases à  cocher -->
11.         <tr>
```

```
12.           <td>Cases &agrave; cocher</td>
13.           <td>
14.            <input type="checkbox" name="chk1" value="un">1
15.            <input type="checkbox" name="chk2" value="deux" checked>2
16.            <input type="checkbox" name="chk3" value="trois">3
17.           </td>
18.         </tr>
19.       </table>
20.      <br>
21.      <hr>
22.      <input type="hidden" name="_chk1">
23.      <input type="hidden" name="_chk2">
24.      <input type="hidden" name="_chk3">
25.      <input type="submit" value="Envoyer">
26.     </form>
27.   </body>
28. </html>
```

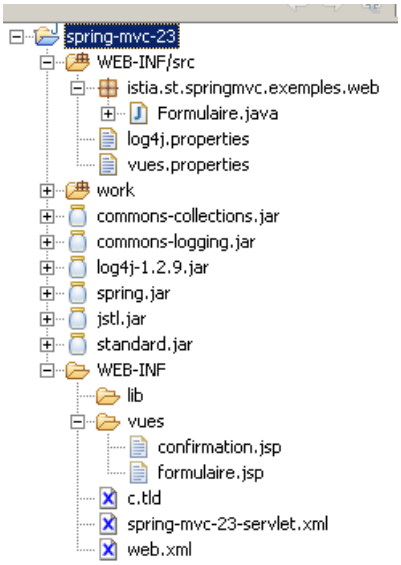
- lignes 14-16 : trois cases à cocher indépendantes de noms " **chk1** ", " **chk2** ", " **chk3** " et de valeurs respectives " **un** ", " **deux** ", " **trois** "
- lignes 22-24 : trois champs cachés, un pour chaque case à cocher. Ils portent le nom des cases à cocher précédé du signe _.

L'utilisateur fait sa saisie :

Une page de confirmation lui est envoyée :

Les valeurs (value) des cases à cocher ont été correctement reproduites.

Le projet Eclipse est le suivant :



Comme il a été expliqué, seuls la classe [Formulaire] et les pages JSP [formulaire.jsp, confirmation.jsp] ont changé.

La classe [Formulaire] doit refléter les trois cases à cocher. Son code est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Formulaire {
4.
5.    // les cases à cocher
6.    private String chk1;
7.
8.    private String chk2;
9.
10.   private String chk3;
11.
12.   // le constructeur
13.   public Formulaire() {
14.     // initialisations du formulaire
15.     this.setChk2("deux");
16.   }
17.
18.   // les getters - setters
19.   public String getChk1() {
20.     return chk1;
21.   }
22.
23.   public void setChk1(String chk1) {
24.     this.chk1 = chk1;
25.   }
26.
27.   public String getChk2() {
28.     return chk2;
29.   }
30.
31.   public void setChk2(String chk2) {
32.     this.chk2 = chk2;
33.   }
34.
35.   public String getChk3() {
36.     return chk3;
37.   }
38.
39.   public void setChk3(String chk3) {
40.     this.chk3 = chk3;
41.   }
42. }
```

- les trois cases à cocher de noms HTML " **chk1** ", " **chk2** " et " **chk3** " sont associées aux champs de mêmes noms dans la classe [Formulaire]
- lignes 13-16 : le constructeur donne la valeur " deux " au champ " chk2 " afin que lors de l'affichage initial, la case de même nom soit cochée, les deux autres ne l'étant pas.

A l'issue du GET, la page JSP [formulaire.jsp] est affichée avec pour modèle une instance de [Formulaire] :

La page [formulaire.jsp] affichant la page ci-dessus est la suivante :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.  <html>
5.    <head>
6.     <title>formulaire</title>
7.    </head>
8.    <body>
9.     <h3>Formulaire Spring : cases &agrave; cocher</h3>
10.     <hr>
11.     <form method="post">
12.       <table border="0">
13.        <!-- cases à cocher -->
14.         <tr>
15.           <td>Cases &agrave; cocher</td>
16.           <td>
17.           <c:choose>
18.             <c:when test='${formulaire.chk1=="un"}'>
19.               <input type="checkbox" name="chk1" value="un" checked>1
20.             </c:when>
21.             <c:otherwise>
22.                 <input type="checkbox" name="chk1" value="un">1
23.             </c:otherwise>
24.           </c:choose>
25.           <c:choose>
26.             <c:when test='${formulaire.chk2=="deux"}'>
27.               <input type="checkbox" name="chk2" value="deux" checked>2
28.             </c:when>
29.             <c:otherwise>
30.                 <input type="checkbox" name="chk2" value="deux">2
31.             </c:otherwise>
32.           </c:choose>
33.           <c:choose>
34.             <c:when test='${formulaire.chk3=="trois"}'>
35.               <input type="checkbox" name="chk3" value="trois" checked>3
36.             </c:when>
37.             <c:otherwise>
38.                 <input type="checkbox" name="chk3" value="trois">3
39.             </c:otherwise>
40.           </c:choose>
41.           </td>
42.         </tr>
43.       </table>
44.      <br>
45.      <hr>
46.      <input type="hidden" name="_chk1">
47.      <input type="hidden" name="_chk2">
48.      <input type="hidden" name="_chk3">
49.      <input type="submit" value="Envoyer">
50.     </form>
51.   </body>
52. </html>
```

- lignes 17-24 : on teste la valeur du champ [chk1] de [Formulaire] pour savoir si on doit cocher ou non la case de nom " chk1 ".
- lignes 25-32 : idem pour la case de nom " chk2 "
- lignes 23-40 : idem pour la case de nom " chk3 "

Plus mystérieuses sont les lignes 46-48. A quoi servent-elles ?

Pour le comprendre, il faut regarder les valeurs réellement postées par le navigateur client. Rappelons-nous que les champs de [Formulaire] ont par construction les valeurs [null,"deux",null]. Le formulaire affiché est donc le suivant :

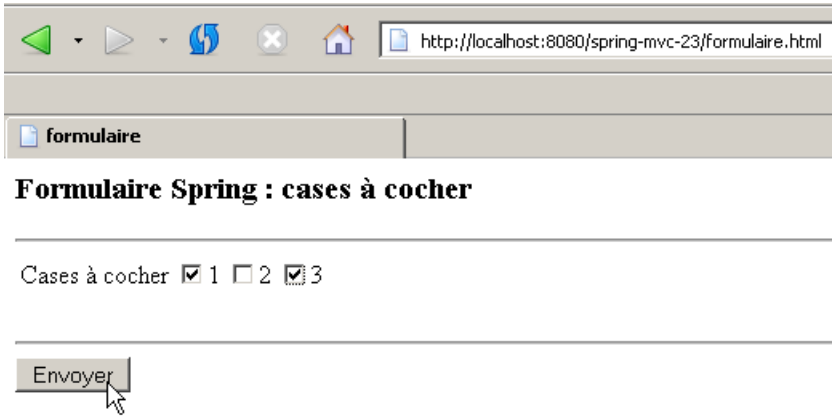
Puis l'utilisateur fait des saisies :

Si on regarde le flux HTTP envoyé par le navigateur avec un outil tel que [LiveHttpHeaders], on trouve la chose suivante :

```
1.  POST /spring-mvc-23/formulaire.html HTTP/1.1
2.  Host: localhost:8080
3.  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.7) Gecko/20040614 Firefox/0.9
4.  Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
5.  Accept-Language: fr,en;q=0.7,de;q=0.3
6.  Accept-Encoding: gzip,deflate
7.  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
8.  Keep-Alive: 300
9.  Connection: keep-alive
10. Referer: http://localhost:8080/spring-mvc-23/formulaire.html
11. Cookie: JSESSIONID=4AF81C5DA7C5444659D7D6CD3AAAEC5A
12. Content-Type: application/x-www-form-urlencoded
13. Content-Length: 39
14.
15. chk1=un&chk3=trois&_chk1=&_chk2=&_chk3=
```

La ligne 15 montre les valeurs postées. Elles vont être affectées à l'instance de [Formulaire] que le GET a mis en session. On se rappelle que les valeurs initiales des champs [chk1, chk2, chk3] étaient [null, "deux ",null]. D'après les valeurs postées, le champ [chk1] de [Formulaire] va recevoir la valeur " un " et le champ [chk3] la valeur " trois ". Le champ [chk2] lui va garder sa valeur " deux " alors qu'il devrait prendre la valeur **null**. Le problème réside dans le fait que la valeur d'une case non cochée n'est pas postée. Spring MVC propose une solution à ce problème. Si dans les valeurs postées, il existe un paramètre **_C** alors le champ **C** de l'objet récupérant les valeurs postées est réinitialisé à **null**. Ainsi ci-dessus, les champs [chk1, chk2, chk3] vont être réinitialisés à **null** avant de recevoir les valeurs postées. Ainsi le champ [chk2] associé à la case qui n'a pas été cochée aura bien la valeur **null** comme on le souhaite.

La page [confirmation.jsp] affiche la valeur de [Formulaire] tel que modifié par le POST :



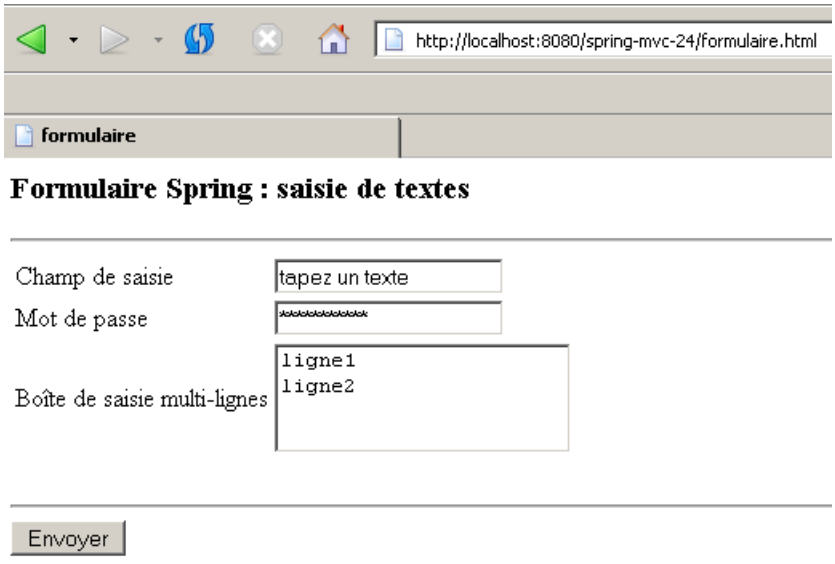
Son code est le suivant :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Formulaire Spring : cases &agrave; cocher</title>
8.    </head>
9.    <body>
10.     <h3>Confirmation des donn&eacute;es saisies</h3>
11.     <table border="1">
12.       <tr>
13.         <td>Case &agrave; cocher chk1</td>
14.         <td>${formulaire.chk1}</td>
15.       </tr>
16.       <tr>
17.         <td>Case &agrave; cocher chk2</td>
18.         <td>${formulaire.chk2}</td>
19.       </tr>
20.       <tr>
21.         <td>Case &agrave; cocher chk3</td>
22.         <td>${formulaire.chk3}</td>
23.       </tr>
24.     </table>
25.   </body>
26. </html>
```

- ligne 14 : la valeur du champ [chk1]
- ligne 18 : la valeur du champ [chk2]
- ligne 22 : la valeur du champ [chk3]

5.4 Formulaire de saisie de textes

Le formulaire affiché à l'issue du GET sera le suivant :

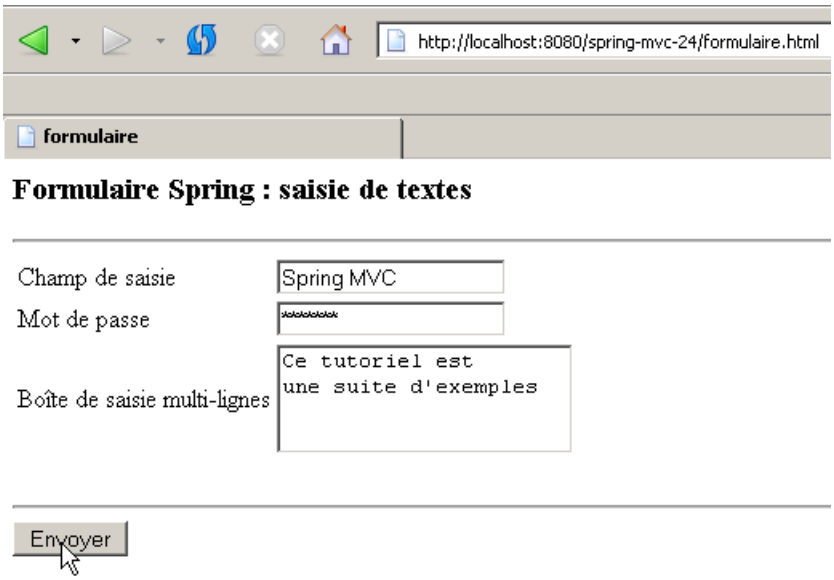


Le formulaire HTML sous-jacent est le suivant :

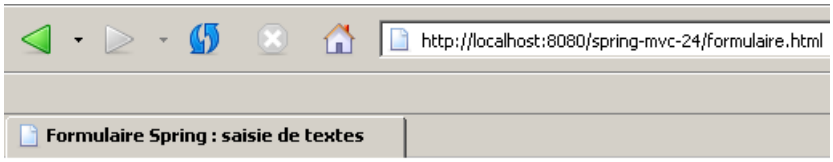
```
1.  <html>
2.    <head>
3.     <title>formulaire</title>
4.    </head>
5.    <body>
6.     <h3>Formulaire Spring : saisie de textes</h3>
7.      <hr>
8.      <form method="post">
9.        <table border="0">
10.        <!-- champ de saisie -->
11.         <tr>
12.           <td>Champ de saisie</td>
13.           <td>
14.           <input type="text" name="champSaisie" value="tapez un texte">
15.           </td>
16.         </tr>
17.        <!-- mot de passe -->
18.         <tr>
19.           <td>Mot de passe</td>
20.           <td>
21.           <input type="password" name="mdp" value="mdporiginel">
22.           </td>
23.         </tr>
24.        <!-- Boîte de saisie multi-lignes -->
25.         <tr>
26.           <td>Bo&icirc;te de saisie multi-lignes</td>
27.           <td>
28.           <textarea name="boiteSaisie" rows="3">ligne1
29. ligne2</textarea>
30.           </td>
31.         </tr>
32.
33.       </table>
34.       <td>
35.        <input type="hidden" name="secret" value="ceci est secret">
36.       </td>
37.      <br>
38.      <hr>
39.      <input type="submit" value="Envoyer">
40.     </form>
41.   </body>
42. </html>
```

- ligne 14 : un champ de saisie HTML nommé " **champSaisie** " de type " **text** "
- ligne 21 : un champ de saisie HTML nommé " **mdp** " de type "**password**"
- lignes 28-29 : un champ de saisie HTML nommé "**boiteSaisie**" de type "**textarea**"
- ligne 35 : un champ caché HTML nommé "**secret**"

L'utilisateur fait ses saisies :



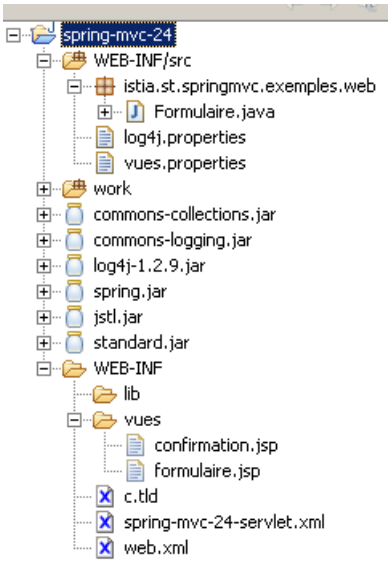
Une page de confirmation lui est envoyée :



Confirmation des données saisies

Champ de saisie	Spring MVC
Mot de passe	qqchose
Boîte de saisie multi-lignes	Ce tutoriel est une suite d'exemples
Champ caché	ceci est secret

Le projet Eclipse est le suivant :



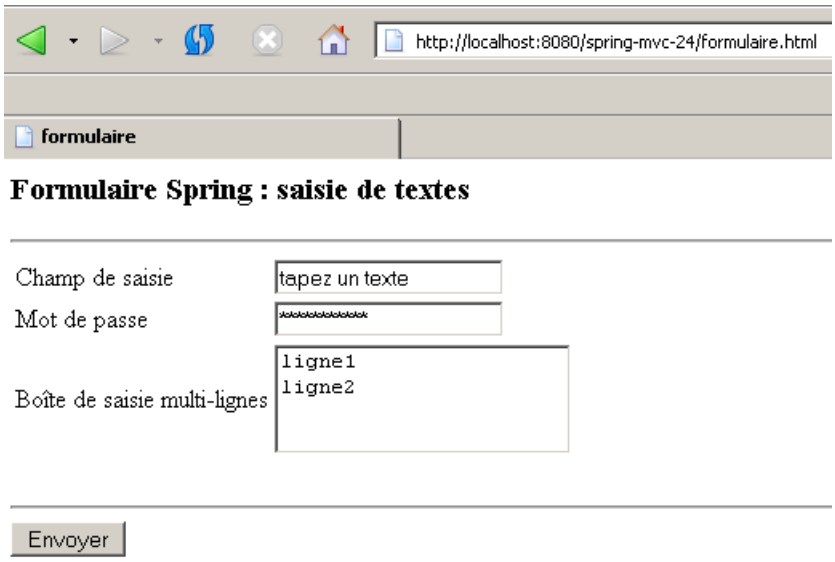
La classe [Formulaire] doit refléter les quatre champs HTML du formulaire. Son code est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Formulaire {
4.
5.    // le champ de saisie
6.    private String champSaisie;
7.    // le mot de passe
8.    private String mdp;
9.    // la boîte de saisie multi-lignes
10.   private String boiteSaisie;
11.   // le champ caché
12.   private String secret;
13.
14.   // constructeur
15.   public Formulaire() {
16.     // initialisations du formulaire
17.     this.setChampSaisie("tapez un texte");
18.     this.setMdp("mdporiginel");
19.     this.setBoiteSaisie("ligne1\nligne2");
20.     this.setSecret("ceci est secret");
21.   }
22.
23.   // getters et setters
24.   public String getBoiteSaisie() {
25.     return boiteSaisie;
26.   }
27.
28.   public void setBoiteSaisie(String boiteSaisie) {
29.     this.boiteSaisie = boiteSaisie;
30.   }
31.
32.   public String getChampSaisie() {
33.     return champSaisie;
34.   }
35.
```

```
36.  public void setChampSaisie(String champSaisie) {
37.    this.champSaisie = champSaisie;
38.  }
39.  public String getMdp() {
40.    return mdp;
41.  }
42.
43.  public void setMdp(String mdp) {
44.    this.mdp = mdp;
45.  }
46.
47.  public String getSecret() {
48.    return secret;
49.  }
50.
51.  public void setSecret(String secret) {
52.    this.secret = secret;
53.  }
54.
55. }
```

- ligne 6 : le champ associé au champ HTML "**champSaisie**"
- ligne 8 : le champ associé au champ HTML "**mdp**"
- ligne 10 : le champ associé au champ HTML "**boiteSaisie**"
- ligne 12 : le champ associé au champ HTML "**secret**"
- lignes 15-21 : les quatre champs reçoivent une valeur initiale lors de la construction de l'objet [Formulaire]

A l'issue du GET, la page JSP [formulaire.jsp] est affichée avec pour modèle une instance de [Formulaire] :



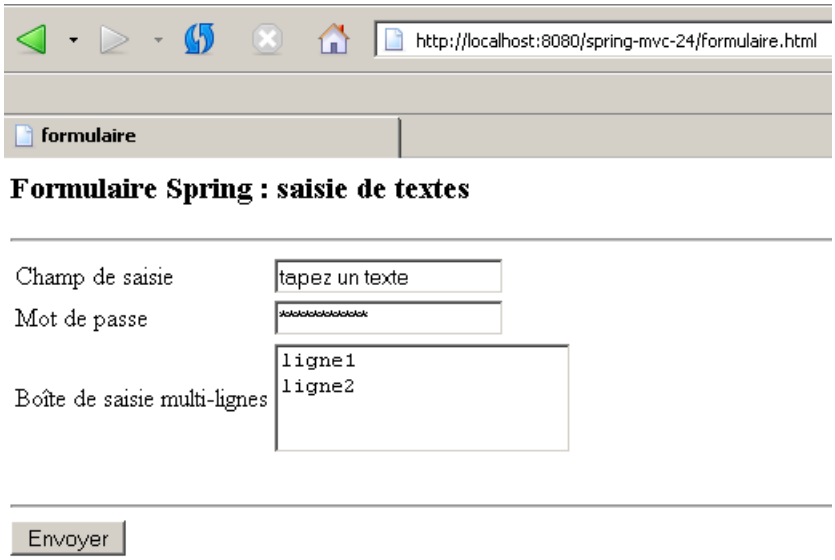
La page [formulaire.jsp] affichant la page ci-dessus est la suivante :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.  <html>
5.    <head>
6.     <title>formulaire</title>
7.    </head>
8.    <body>
9.     <h3>Formulaire Spring : saisie de textes</h3>
10.     <hr>
11.     <form method="post">
12.       <table border="0">
13.        <!-- champ de saisie -->
14.         <tr>
15.           <td>Champ de saisie</td>
16.           <td>
17.            <input type="text" name="champSaisie" value="${formulaire.champSaisie}">
18.           </td>
19.         </tr>
20.        <!-- mot de passe -->
21.         <tr>
22.           <td>Mot de passe</td>
23.           <td>
24.            <input type="password" name="mdp" value="${formulaire.mdp}">
25.           </td>
```

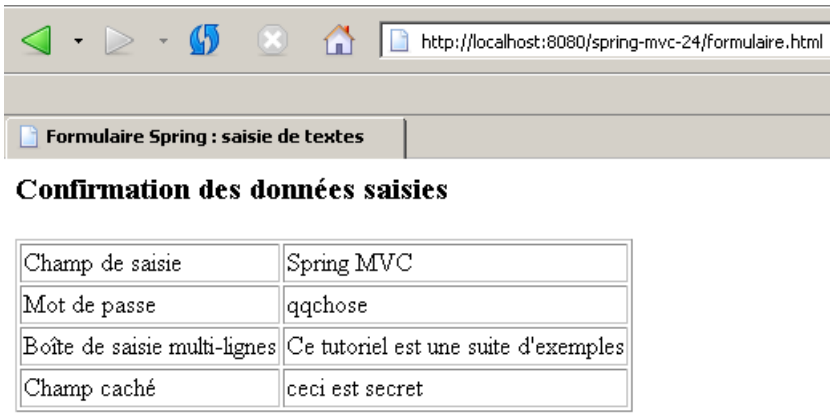
```
26.         </tr>
27.        <!-- Boîte de saisie multi-lignes -->
28.         <tr>
29.           <td>Bo&icirc;te de saisie multi-lignes</td>
30.           <td>
31.           <textarea name="boiteSaisie" rows="3">${formulaire.boiteSaisie}</textarea>
32.           </td>
33.         </tr>
34.       </table>
35.       <td>
36.        <input type="hidden" name="secret" value="${formulaire.secret}">
37.       </td>
38.      <br>
39.      <hr>
40.      <input type="submit" value="Envoyer">
41.     </form>
42.   </body>
43. </html>
```

- ligne 17 : le champ HTML " **champSaisie** " prend sa valeur initiale dans le champ [champSaisie] de [Formulaire], donc la chaîne "tapez un texte"
- ligne 24 : le champ HTML "**mdp**" prend sa valeur initiale dans le champ [mdp] de [Formulaire], donc la chaîne "mdporiginel"
- ligne 31 : le champ HTML " **boiteSaisie** " prend sa valeur initiale dans le champ [boiteSaisie] de [Formulaire], donc la chaîne "ligne1\nligne2"
- ligne 36 : le champ HTML "**secret**" prend sa valeur initiale dans le champ [secret] de [Formulaire], donc la chaîne "ceci est secret"

Ceci explique que la page affichée à la suite du GET soit la suivante :



La page [confirmation.jsp] affiche la valeur de [Formulaire] tel que modifié par le POST :



Son code est le suivant :

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

```
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Formulaire Spring : saisie de textes</title>
8.    </head>
9.    <body>
10.     <h3>Confirmation des donn&eacute;es saisies</h3>
11.     <table border="1">
12.       <tr>
13.         <td>Champ de saisie</td>
14.         <td>${formulaire.champSaisie}</td>
15.       </tr>
16.       <tr>
17.         <td>Mot de passe</td>
18.         <td>${formulaire.mdp}</td>
19.       </tr>
20.       <tr>
21.         <td>Bo&icirc;te de saisie multi-lignes</td>
22.         <td>${formulaire.boiteSaisie}</td>
23.       </tr>
24.       <tr>
25.         <td>Champ cach&eacute;</td>
26.         <td>${formulaire.secret}</td>
27.       </tr>
28.   </body>
29. </html>
30.
```

- ligne 14 : la valeur du champ [champSaisie]
- ligne 18 : la valeur du champ [mdp]
- ligne 22 : la valeur du champ [boiteSaisie]
- ligne 26 : la valeur du champ [secret]

5.5 Formulaire avec liste déroulante

Le formulaire affiché à l'issue du GET sera le suivant :

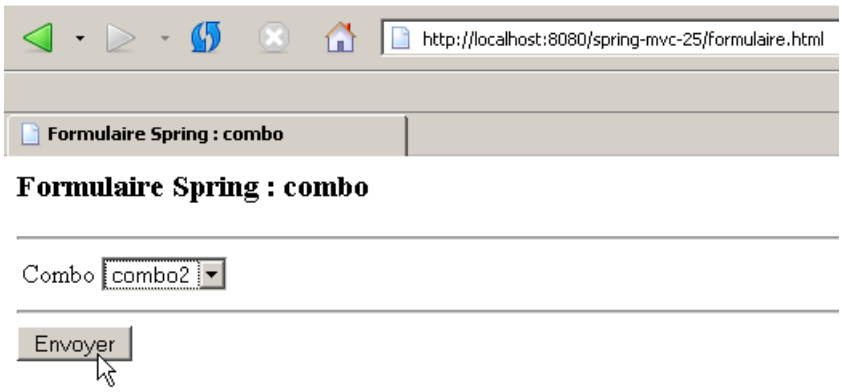
Le formulaire HTML sous-jacent est le suivant :

```
1.  <html>
2.    <head>
3.      <title>Formulaire Spring : combo</title>
4.    </head>
5.    <body>
6.      <h3>Formulaire Spring : combo</h3>
7.       <hr>
8.       <form method="post">
9.         <table border="0">
10.         <!-- Combo -->
11.          <tr>
12.            <td>Combo</td>
13.            <td>
14.           <select name="combo" rows="3">
15.             <option>combo0</option>
16.             <option>combo1</option>
17.             <option>combo2</option>
18.             <option selected>combo3</option>
19.             <option>combo4</option>
20.           </select>
21.             </td>
22.          </tr>
23.        </table>
```

```
24.       <hr>
25.       <input type="submit" value="Envoyer">
26.     </form>
27.   </body>
28. </html>
```

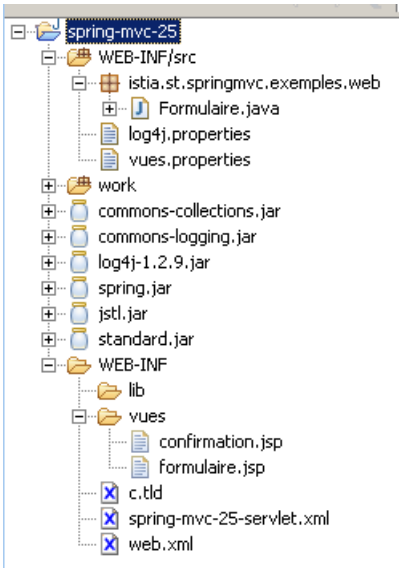
- lignes 14-20 : un champ de saisie HTML nommé "**combo**" de type "**select**"
- lignes 15-19 : les cinq options de la balise <select>. La syntaxe normale d'une option est <option value= "Value ">Texte</option>. Lors du POST, c'est la valeur **Value** de l'élément sélectionné qui est postée. En l'absence d'attribut **value**, c'est la valeur **Texte** qui est postée. Ce sera donc le cas ici.

L'utilisateur fait sa sélection dans le combo :



Une page de confirmation lui est envoyée :

Le projet Eclipse est le suivant :



La classe [Formulaire] doit refléter le champ HTML " **combo** " du formulaire. Son code est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Formulaire {
```

```
4.
5.   // la valeur sélectionnée dans le combo
6.   private String combo;
7.   // la liste des options du combo
8.   private String[] optionsCombo;
9.
10.  // le constructeur
11.  public Formulaire() {
12.    // initialisations du formulaire
13.    this.setOptionsCombo(getOptions(5, "combo"));
14.    this.setCombo("combo3");
15.  }
16.
17.  // getters et setters
18.  public String getCombo() {
19.    return combo;
20.  }
21.
22.  public void setCombo(String combo) {
23.    this.combo = combo;
24.  }
25.
26.  public String[] getOptionsCombo() {
27.    return optionsCombo;
28.  }
29.
30.  public void setOptionsCombo(String[] optionsCombo) {
31.    this.optionsCombo = optionsCombo;
32.  }
33.
34.  // générateur d'options de liste
35.  private String[] getOptions(int taille, String label) {
36.    String[] options = new String[taille];
37.    for (int i = 0; i < taille; i++) {
38.      options[i] = label + i;
39.    }
40.    return options;
41.  }
42.
43. }
```

- ligne 6 : le champ associé au champ HTML "**combo**". Lors du POST, il recevra le texte de l'élément sélectionné dans le combo par l'utilisateur.
- lignes 8 : le tableau dont le contenu sera utilisé pour alimenter le champ HTML "**combo**" en " options ". Une nouveauté apparaît ici. Ce champ est utilisé par le GET mais pas par le POST. [Formulaire] n'est alors pas le reflet exact des champs HTML postés mais un sur-ensemble de ceux-ci. Ultérieurement, on introduira une solution permettant de mettre ce type d'informations, nécessaires au GET mais pas au POST, dans le modèle de la vue affichée par le GET sans pour autant les mettre dans [Formulaire].
- lignes 11-15 : le constructeur
- ligne 13 : le tableau des options du combo est initialisé avec 5 valeurs arbitraires
- ligne 14 : l'une d'elles " **combo3** ", sera la valeur sélectionnée à l'affichage initial de [Formulaire]

La page [formulaire.jsp] à l'origine de la page ci-dessus est la suivante :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.  <html>
5.    <head>
6.     <title>Formulaire Spring : combo</title>
7.    </head>
8.    <body>
9.     <h3>Formulaire Spring : combo</h3>
10.     <hr>
```

```
11.     <form method="post">
12.       <table border="0">
13.        <!-- Combo -->
14.         <tr>
15.           <td>Combo</td>
16.           <td>
17.          <select name="combo" rows="3">
18.            <c:forEach items="${formulaire.optionsCombo}" var="optionCombo">
19.              <c:choose>
20.                <c:when test="${formulaire.combo==optionCombo}">
21.                  <option selected>${optionCombo}</option>
22.                </c:when>
23.                <c:otherwise>
24.                  <option>${optionCombo}</option>
25.                </c:otherwise>
26.              </c:choose>
27.            </c:forEach>
28.          </select>
29.           </td>
30.         </tr>
31.       </table>
32.      <hr>
33.      <input type="submit" value="Envoyer">
34.     </form>
35.   </body>
36. </html>
```

- lignes 18-27 : la boucle de génération des éléments de la liste déroulante
- ligne 18 : on parcourt le tableau [**optionsCombo**] de [Formulaire] qui contient les éléments de la liste déroulante
- lignes 19-26 : pour savoir si l'élément courant **optionCombo** de la boucle doit être sélectionné ou non, on le compare à l'élément à sélectionner qu'on sait trouver dans le champ **combo** de [Formulaire]
- ligne 21 : génération de l'élément sélectionné
- ligne 24 : génération des éléments non sélectionnés

La page [confirmation.jsp] affiche la valeur de [Formulaire] tel que modifié par le POST :

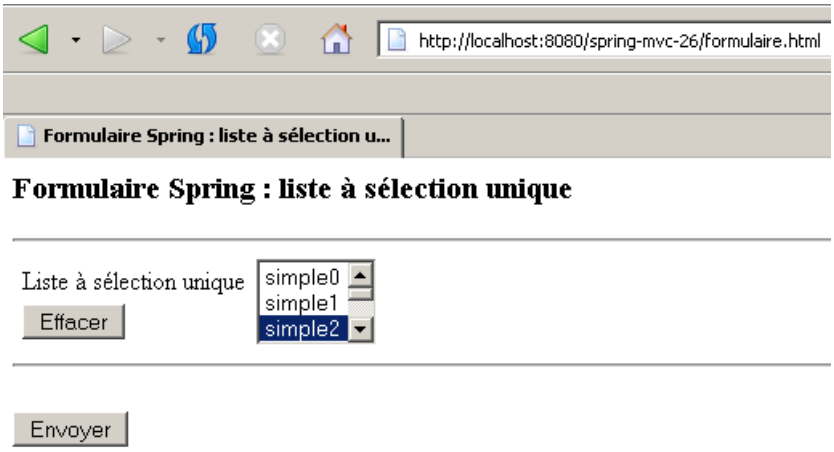
Son code est le suivant :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Formulaire Spring : combo</title>
8.    </head>
9.    <body>
10.     <h2>Confirmation des donn&eacute;es saisies</h2>
11.     <table border="1">
12.       <tr>
13.         <td>Option s&eacute;lectionn&eacute;e dans le combo</td>
14.         <td>${formulaire.combo}</td>
15.       </tr>
16.     </table>
17.   </body>
18. </html>
```

La ligne 14 affiche la valeur du champ [combo] de [Formulaire]. A l'issue du POST du champ HTML " combo ", ce champ contient la valeur sélectionnée par l'utilisateur. C'est ce que les copies d'écran de début de cette section ont montré.

5.6 Formulaire avec liste à sélection unique

Le formulaire affiché à l'issue du GET sera le suivant :

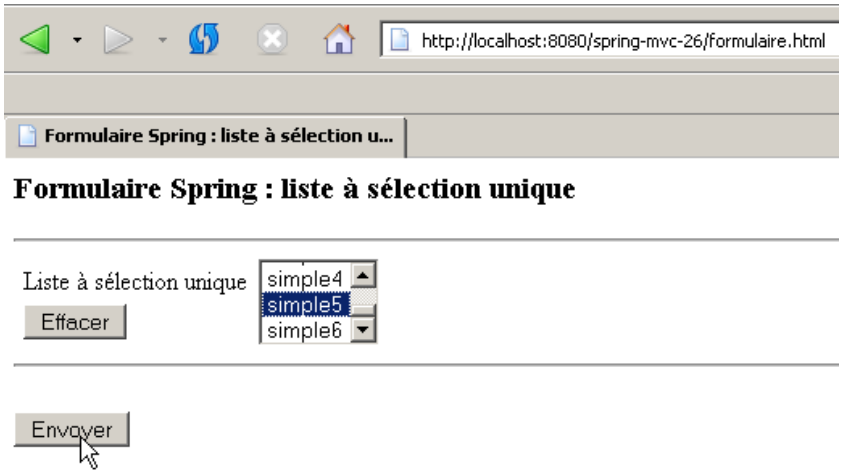


Le formulaire HTML sous-jacent est le suivant :

```
1.  <html>
2.    <head>
3.      <title>Formulaire Spring : liste &agrave; s&eacute;lection unique</title>
4.    </head>
5.    <body>
6.      <h3>Formulaire Spring : liste &agrave; s&eacute;lection unique</h3>
7.       <hr>
8.       <form method="post">
9.         <table border="0">
10.          <!-- Liste à sélection unique -->
11.          <tr>
12.            <td>
13.              <table>
14.                <tr>
15.                  <td>Liste &agrave; s&eacute;lection unique</td>
16.
17.                </tr>
18.                <tr>
19.                  <td>
20.                    <input type="button" value="Effacer" onclick="this.form.listeSimple.selectedIndex=-1"/>
21.                  </td>
22.                </tr>
23.              </table>
24.            </td>
25.            <td>
26.            <select name="listeSimple" size="3">
27.                <option>simple0</option>
28.                <option>simple1</option>
29.                <option selected>simple2</option>
30.                <option>simple3</option>
31.                <option>simple4</option>
32.                <option>simple5</option>
33.                <option>simple6</option>
34.            </select>
35.            </td>
36.          </tr>
37.       </table>
38.       <hr>
39.       <br>
40.       <input type="hidden" name="_listeSimple">
41.       <input type="submit" value="Envoyer">
42.     </form>
43.   </body>
44. </html>
```

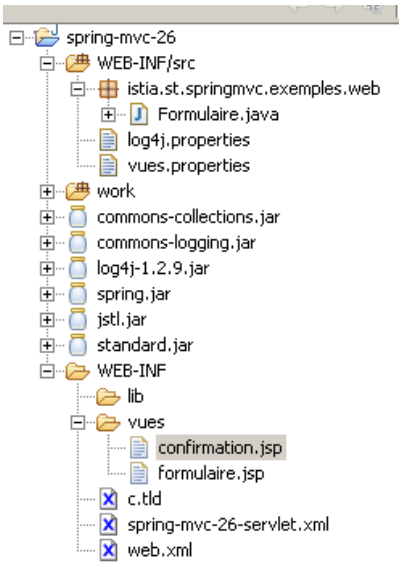
- lignes 26-34 : une liste HTML à sélection unique de type "**select**" nommée "**listeSimple**". L'option " **simple2** " est sélectionnée (ligne 29).
- lignes 27-33 : les options de la balise <select>. La syntaxe normale d'une option est <option value= "Value ">Texte</option>.
- ligne 40 : un champ caché nommé " **_listeSimple** " dont nous expliquerons le rôle ultérieurement.
- ligne 20 : un bouton associé à du code Javascript. Après un clic dessus, la liste HTML " **listeSimple** " n'a plus d'élément sélectionné.

L'utilisateur fait sa sélection dans la liste :



Une page de confirmation lui est envoyée :

Le projet Eclipse est le suivant :

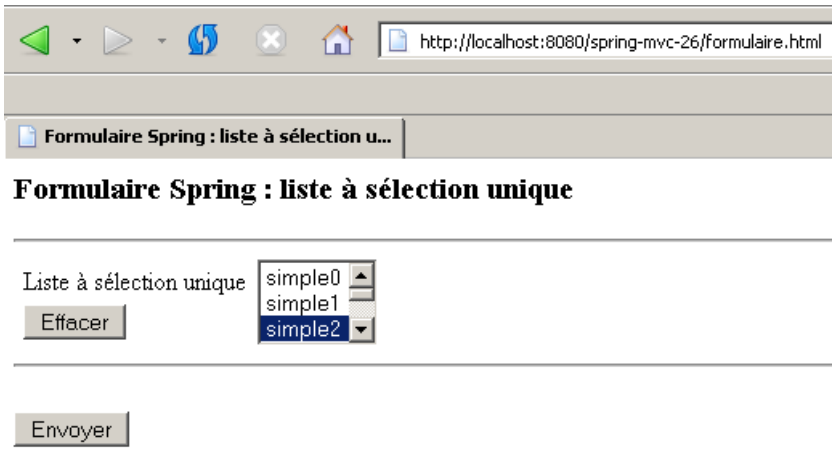


La classe [Formulaire] doit refléter le champ HTML "**listeSimple**" du formulaire. Son code est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Formulaire {
4.
5.    // valeur sélectionnée dans la liste
6.    private String listeSimple;
7.    // les éléments de la liste
8.    private String[] optionsListeSimple;
9.
10.   // constructeur
11.   public Formulaire() {
12.     // initialisations du formulaire
13.     this.setOptionsListeSimple(getOptions(7, "simple"));
14.     this.setListeSimple("simple2");
15.   }
16.
```

```
17.  // getters et setters
18.  public String getListeSimple() {
19.    return listeSimple;
20.  }
21.
22.  public void setListeSimple(String listeSimple) {
23.    this.listeSimple = listeSimple;
24.  }
25.
26.  public String[] getOptionsListeSimple() {
27.    return optionsListeSimple;
28.  }
29.
30.  public void setOptionsListeSimple(String[] optionsListeSimple) {
31.    this.optionsListeSimple = optionsListeSimple;
32.  }
33.
34.  // générateur de liste de valeurs
35.  private String[] getOptions(int taille, String label) {
36.    String[] options = new String[taille];
37.    for (int i = 0; i < taille; i++) {
38.      options[i] = label + i;
39.    }
40.    return options;
41.  }
42.
43. }
```

- ligne 6 : le champ associé au champ HTML "**listeSimple**". Lors du POST, il recevra le texte de l'élément sélectionné dans la liste par l'utilisateur.
- lignes 8 : le tableau dont le contenu sera utilisé pour alimenter le champ HTML "**listeSimple**" en " options ".
- lignes 11-15 : le constructeur.
- ligne 13 : le tableau des éléments de la liste est initialisé avec 7 valeurs arbitraires
- ligne 14 : l'une d'elles, " **simple2** ", sera la valeur sélectionnée à l'affichage initial de [Formulaire]



La page [formulaire.jsp] à l'origine de la page ci-dessus est la suivante :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.  <html>
5.    <head>
6.      <title>Formulaire Spring : liste &agrave; s&eacute;lection unique</title>
7.    </head>
8.    <body>
9.      <h3>Formulaire Spring : liste &agrave; s&eacute;lection unique</h3>
10.     <hr>
11.     <form method="post">
12.       <table border="0">
13.         <!-- Liste à sélection unique -->
14.         <tr>
15.           <td>
16.             <table>
17.               <tr>
18.                 <td>Liste &agrave; s&eacute;lection unique</td>
19.               </tr>
20.               <tr>
21.                 <td>
22.                   <input type="button" value="Effacer" onclick="this.form.listeSimple.selectedIndex=-1"/>
23.                 </td>
24.               </tr>
25.             </table>
26.           </td>
27.           <td>
28.           <select name="listeSimple" size="3">
```

```
29.             <c:forEach items="${formulaire.optionsListeSimple}" var="optionListeSimple">
30.               <c:choose>
31.                 <c:when test="${formulaire.listeSimple==optionListeSimple}">
32.                   <option selected>${optionListeSimple}</option>
33.                 </c:when>
34.                 <c:otherwise>
35.                   <option>${optionListeSimple}</option>
36.                 </c:otherwise>
37.               </c:choose>
38.             </c:forEach>
39.           </select>
40.           </td>
41.         </tr>
42.       </table>
43.       <hr>
44.       <br>
45.       <input type="hidden" name="_listeSimple">
46.       <input type="submit" value="Envoyer">
47.     </form>
48.   </body>
49. </html>
```

- lignes 29-38 : la boucle de génération des éléments de la liste **listeSimple**
- ligne 29 : on parcourt le tableau [**optionsListeSimple**] de [Formulaire] qui contient les éléments de la liste à afficher
- lignes 29-37 : pour savoir si l'élément courant **optionListeSimple** de la boucle doit être sélectionné ou non, on le compare à l'élément à sélectionner qu'on sait trouver dans le champ **listeSimple** de [Formulaire]
- ligne 32 : génération de l'élément sélectionné
- ligne 35 : génération des éléments non sélectionnés

La page [confirmation.jsp] affiche la valeur de [Formulaire] tel que modifié par le POST :

Son code est le suivant :

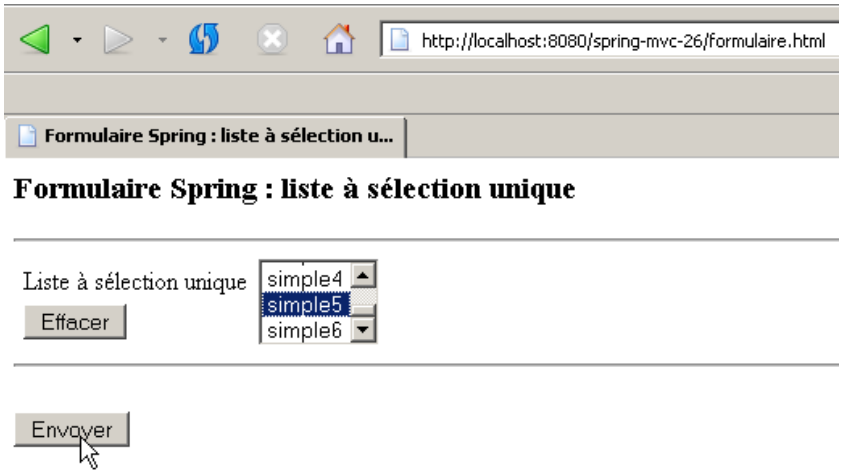
```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Formulaire Spring : liste &agrave; s&eacute;lection unique</title>
8.    </head>
9.    <body>
10.     <h3>Confirmation des donn&eacute;es saisies</h3>
11.     <table border="1">
12.       <tr>
13.         <td>Option s&eacute;lectionn&eacute;e dans la liste &agrave; s&eacute;lection unique</td>
14.         <td>${formulaire.listeSimple}</td>
15.       </tr>
16.     </table>
17.   </body>
18. </html>
19.
```

La ligne 14 affiche la valeur du champ [**listeSimple**] de [Formulaire]. A l'issue du POST du champ HTML "**listeSimple**", ce champ contient la valeur sélectionnée par l'utilisateur. C'est ce que les copies d'écran montrent.

Il nous reste à élucider un point du code HTML, celui du champ caché " **_listeSimple** " (ligne 40 de formulaire.jsp). A quoi sert-il ?

```
      <input type="hidden" name="_listeSimple">
```

Pour comprendre le rôle du champ caché **_listeSimple**, il nous faut connaître le flux HTTP envoyé par le client au serveur lors du POST. Prenons le cas déjà montré :



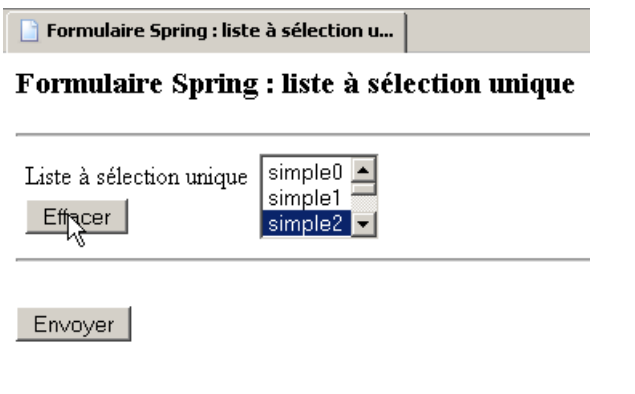
Avec le plugin [LiveHttpHeaders], on peut connaître le flux HTTP envoyé au serveur lorsque l'utilisateur va cliquer sur le bouton [Envoyer] :

```
1.  POST /spring-mvc-26/formulaire.html HTTP/1.1
2.  Host: localhost:8080
3.  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.7) Gecko/20040614 Firefox/0.9
4.  Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
5.  Accept-Language: fr,en;q=0.7,de;q=0.3
6.  Accept-Encoding: gzip,deflate
7.  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
8.  Keep-Alive: 300
9.  Connection: keep-alive
10. Referer: http://localhost:8080/spring-mvc-26/formulaire.html
11. Cookie: JSESSIONID=A6A985341AC243D071FA4B0843D4ECFA
12. Content-Type: application/x-www-form-urlencoded
13. Content-Length: 33
14.
15. listeSimple=simple5&_listeSimple=
```

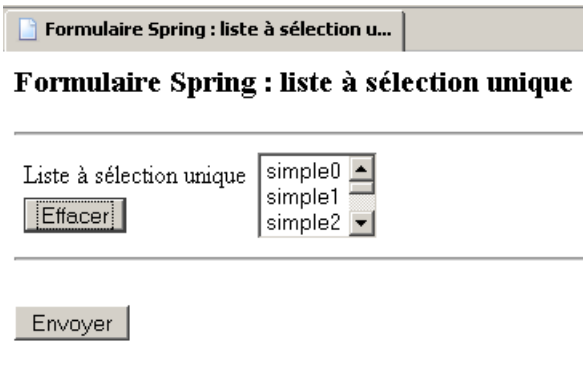
- ligne 1 : on voit le POST qui est fait
- ligne 15 : on voit les valeurs postées. On a deux valeurs postées :
  - la valeur du champ HTML "**listeSimple**". C'est la valeur " **simple5** " sélectionnée par l'utilisateur.
  - la valeur du champ caché "**_listeSimple**". C'est la valeur initiale du champ caché, c.a.d. la chaîne vide.

    Nous savons qu'initialement, le champ " **listeSimple** " de [Formulaire] avait la valeur " **simple2** ". Après le POST, il aura la valeur " **simple5** ". C'est ce qu'ont montré les copies d'écran.

Considérons maintenant, le cas suivant. L'utilisateur déselectionne tout élément de la liste avec le bouton [Effacer] :



avant



après

Maintenant postons le formulaire avec le bouton [Envoyer] et examinons le flux HTTP envoyé par le navigateur :

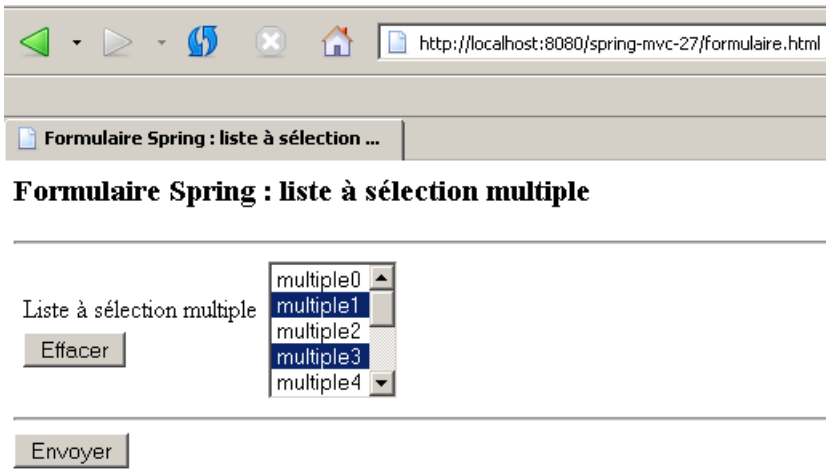
```
1.  POST /spring-mvc-26/formulaire.html HTTP/1.1
2.  Host: localhost:8080
3.  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US; rv:1.7) Gecko/20040614 Firefox/0.9
4.  Accept: text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.5
5.  Accept-Language: fr,en;q=0.7,de;q=0.3
6.  Accept-Encoding: gzip,deflate
7.  Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
8.  Keep-Alive: 300
9.  Connection: keep-alive
10. Referer: http://localhost:8080/spring-mvc-26/formulaire.html
11. Cookie: JSESSIONID=A6A985341AC243D071FA4B0843D4ECFA
12. Content-Type: application/x-www-form-urlencoded
13. Content-Length: 13
14.
```

```
15. _listeSimple=
```

- ligne 15, on constate que le paramètre " **listeSimple** " n'est plus dans les paramètres postés, ceci parce qu'aucun élément n'a été sélectionné dans la liste. Donc si on ne fait rien, le champ **listeSimple** de [Formulaire] ne reçoit pas de nouvelle valeur et garde sa valeur initiale " **simple2** ". La valeur du champ ne refléterait alors plus l'état de la liste HTML **listeSimple**. Pour résoudre ce problème, Spring MVC propose une solution identique à celle des cases à cocher : mettre dans la liste des paramètres postés, un paramètre portant le nom de la liste, précédé du signe _. C'est le rôle ici du paramètre **_listeSimple**. La valeur de ce paramètre n'importe pas. Seule sa présence compte. En détectant la présence d'un paramètre posté **_C**, Spring MVC affecte la valeur **null** au champ **C** associé, avant d'affecter les valeurs postées aux différents champs de [Formulaire]. C'est ainsi qu'ici, le champ **listeSimple** va recevoir la valeur **null**. Ce que montre la page de confirmation :

5.7 Formulaire avec liste à sélection multiple

Le formulaire affiché à l'issue du GET sera le suivant :



Le formulaire HTML sous-jacent est le suivant :

```
1.  <html>
2.    <head>
3.      <title>Formulaire Spring : liste &agrave; s&eacute;lection multiple</title>
4.    </head>
5.    <body>
6.      <h3>Formulaire Spring : liste &agrave; s&eacute;lection multiple</h3>
7.      <hr>
8.      <form method="post">
9.        <table border="0">
10.         <!-- Liste à sélection multiple -->
11.         <tr>
12.           <td>
13.             <table>
14.               <tr>
15.                 <td>Liste &agrave; s&eacute;lection multiple</td>
16.
17.               </tr>
18.               <tr>
19.                 <td>
20.                   <input type="button" value="Effacer" onclick="this.form.listeMultiple.selectedIndex=-1"/>
21.                 </td>
22.               </tr>
23.             </table>
24.           </td>
25.           <td>
26.             <select name="listeMultiple" size="5" multiple>
27.               <option>multiple0</option>
```

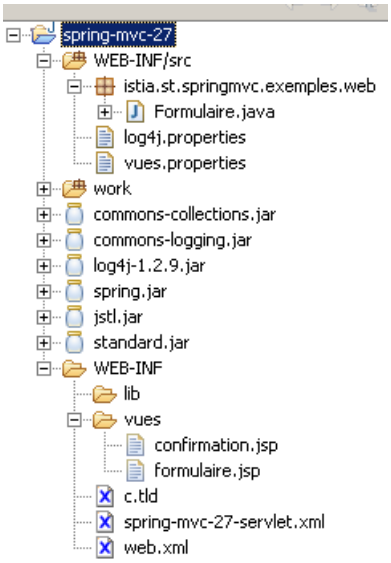
```
28.               <option selected>multiple1</option>
29.               <option>multiple2</option>
30.               <option selected>multiple3</option>
31.               <option>multiple4</option>
32.               <option>multiple5</option>
33.               <option>multiple6</option>
34.               <option>multiple7</option>
35.               <option>multiple8</option>
36.               <option>multiple9</option>
37.           </select>
38.           </td>
39.         </tr>
40.       </table>
41.       <hr>
42.       <input type="hidden" name="_listeMultiple">
43.       <input type="submit" value="Envoyer">
44.     </form>
45.   </body>
46. </html>
```

- ligne 26 : une liste HTML à sélection multiple de type "**select**" nommée "**listeMultiple**".
- lignes 27-36 : les options de la balise <select>. Les options " **multiple1** " et " **multiple3 "** sont sélectionnées (lignes 28, 29).
- ligne 42 : un champ caché nommé " **_listeMultiple** ". Son rôle est identique à celui du paramètre " **_listeSimple** " décrit précédemment.
- ligne 20 : un bouton associé à du code Javascript. Après un clic dessus, la liste HTML " **listeMultiple** " n'a plus d'élément sélectionné.

L'utilisateur fait sa sélection dans la liste :

Les sélections multiples sont faites en maintenant appuyée la touche [Ctrl]. Une page de confirmation lui est envoyée :

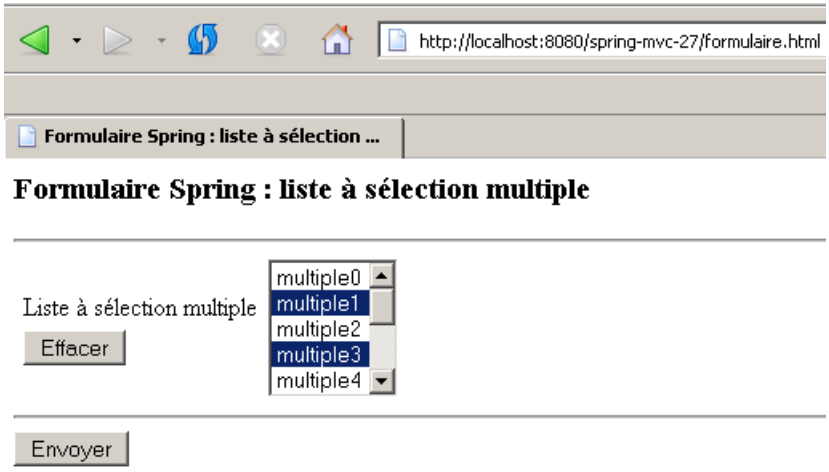
Le projet Eclipse est le suivant :



La classe [Formulaire] doit refléter le champ HTML "**listeMultiple**" du formulaire. Son code est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Formulaire {
4.
5.    // les éléments sélectionnés dans la liste
6.    private String[] listeMultiple;
7.    // les options de la liste
8.    private String[] optionsListeMultiple;
9.
10.   // constructeur
11.   public Formulaire() {
12.     // initialisations du formulaire
13.     this.setOptionsListeMultiple(getOptions(10, "multiple"));
14.     this.setListeMultiple(new String[] { "multiple1", "multiple3" });
15.   }
16.
17.   // getters et setters
18.   public String[] getListeMultiple() {
19.     return listeMultiple;
20.   }
21.
22.   public void setListeMultiple(String[] listeMultiple) {
23.     this.listeMultiple = listeMultiple;
24.   }
25.
26.   public String[] getOptionsListeMultiple() {
27.     return optionsListeMultiple;
28.   }
29.
30.   public void setOptionsListeMultiple(String[] optionsListeMultiple) {
31.     this.optionsListeMultiple = optionsListeMultiple;
32.   }
33.
34.   // générateur de liste de valeurs
35.   private String[] getOptions(int taille, String label) {
36.     String[] options = new String[taille];
37.     for (int i = 0; i < taille; i++) {
38.       options[i] = label + i;
39.     }
40.     return options;
41.   }
42.
43. }
```

- ligne 6 : le champ associé au champ HTML "**listeMultiple**". Lors du POST, il recevra le texte des éléments sélectionnés dans la liste par l'utilisateur. Il nous faut donc un **tableau de chaînes de caractères**.
- ligne 8 : le tableau dont le contenu sera utilisé pour alimenter le champ HTML "**listeMultiple**" en " options ".
- lignes 11-15 : le constructeur.
- ligne 13 : le tableau des éléments de la liste est initialisé avec 10 valeurs arbitraires
- ligne 14 : deux d'entre-elles, "**multiple1**" et "**multiple3**", seront sélectionnées à l'affichage initial de [Formulaire]



La page [formulaire.jsp] à l'origine de la page ci-dessus est la suivante :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.  <html>
5.    <head>
6.      <title>Formulaire Spring : liste &agrave; s&eacute;lection multiple</title>
7.    </head>
8.    <body>
9.      <h3>Formulaire Spring : liste &agrave; s&eacute;lection multiple</h3>
10.     <hr>
11.     <form method="post">
12.       <table border="0">
13.         <!-- Liste à sélection multiple -->
14.         <tr>
15.           <td>
16.             <table>
17.               <tr>
18.                 <td>Liste &agrave; s&eacute;lection multiple</td>
19.               </tr>
20.               <tr>
21.                 <td>
22.                   <input type="button" value="Effacer" onclick="this.form.listeMultiple.selectedIndex=-1"/>
23.                 </td>
24.               </tr>
25.             </table>
26.           </td>
27.           <td>
28.           <select name="listeMultiple" size="5" multiple>
29.             <c:forEach items="${formulaire.optionsListeMultiple}" var="optionListeMultiple">
30.               <c:set var="selection" value="faux"/>
31.               <c:forEach items="${formulaire.listeMultiple}" var="selectionListeMultiple">
32.                 <c:if test="${selectionListeMultiple==optionListeMultiple}">
33.                   <c:set var="selection" value="vrai"/>
34.                 </c:if>
35.                 </c:forEach>
36.               <c:choose>
37.                 <c:when test='${selection=="vrai"}'>
38.                   <option selected>${optionListeMultiple}</option>
39.                 </c:when>
40.                 <c:otherwise>
41.                   <option>${optionListeMultiple}</option>
42.                 </c:otherwise>
43.               </c:choose>
44.             </c:forEach>
45.           </select>
46.           </td>
47.         </tr>
48.       </table>
49.       <hr>
50.       <input type="hidden" name="_listeMultiple">
51.       <input type="submit" value="Envoyer">
52.     </form>
53.   </body>
54. </html>
```

- lignes 29-44 : la boucle de génération des éléments de la liste multiple
- ligne 29 : on parcourt le tableau [**optionsListeMultiple**] de [Formulaire] qui contient les éléments de la liste
- ligne 30 : une variable locale **selection** qui passera à " vrai " si l'élément courant **optionListeMultiple** de la boucle doit être sélectionné ou non dans la liste HTML
- lignes 31-35 : l'élément courant **optionListeMultiple** de la boucle est comparé à tous les éléments du tableau **listeMultiple** de [Formulaire]. Ce tableau contient les éléments à sélectionner. A l'issue de cette boucle interne, la variable locale **selection** est à " vrai " ou " faux " selon que l'élément courant **optionListeMultiple** de la boucle doit être sélectionné ou non dans la liste HTML
- lignes 36-43 : le test qui permet de générer l'élément sélectionné (ligne 38) ou non sélectionné (ligne 41).

La page [confirmation.jsp] affiche la valeur de [Formulaire] tel que modifié par le POST :

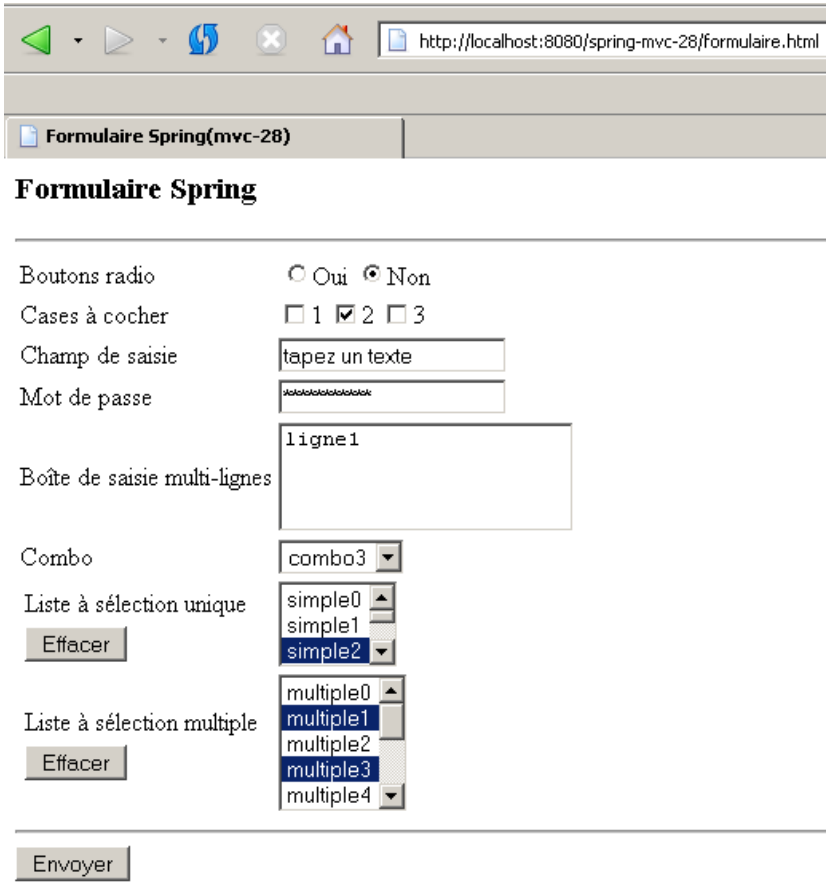
Son code est le suivant :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Formulaire Spring : liste &agrave; s&eacute;lection multiple</title>
8.    </head>
9.    <body>
10.     <h3>Confirmation des donn&eacute;es saisies</h3>
11.     <table border="1">
12.       <tr>
13.         <td>Options s&eacute;lectionn&eacute;es dans la liste &agrave; s&eacute;lection multiple</td>
14.         <c:forEach items="${formulaire.listeMultiple}" var="selectionListeMultiple">
15.           <td>${selectionListeMultiple}</td>
16.         </c:forEach>
17.       </tr>
18.     </table>
19.   </body>
20. </html>
```

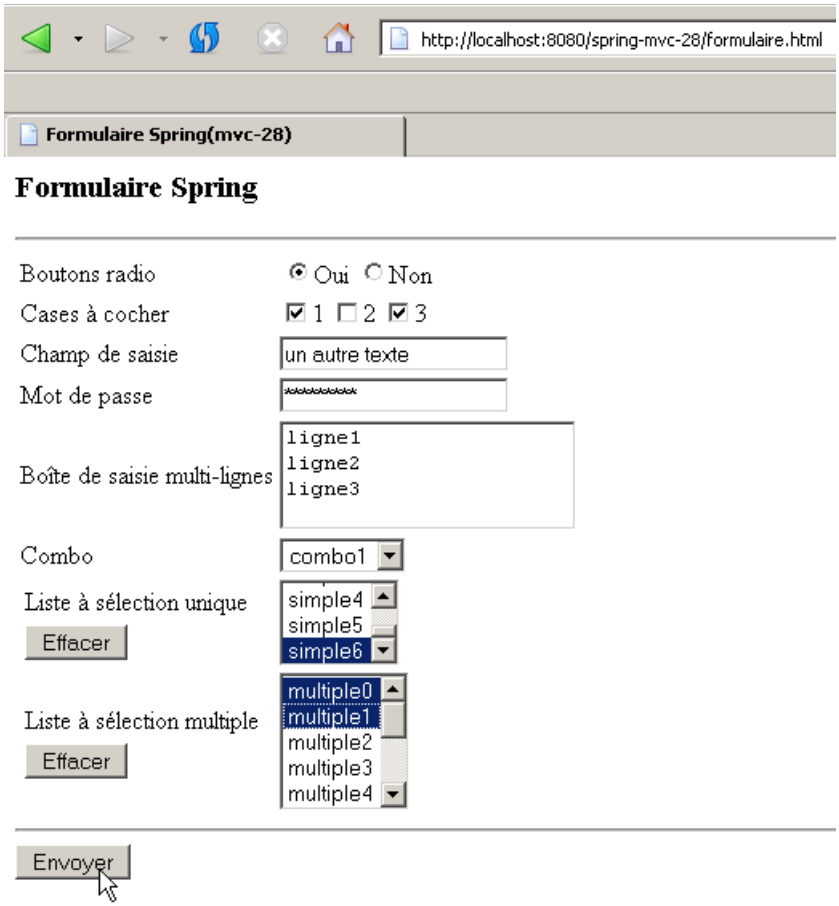
- lignes 14-16 : on boucle sur les éléments du tableau **listeMultiple** de [Formulaire] pour les afficher. Ils représentent les éléments qui ont été sélectionnés par l'utilisateur dans la liste.

5.8 Formulaire complet avec gestion de session

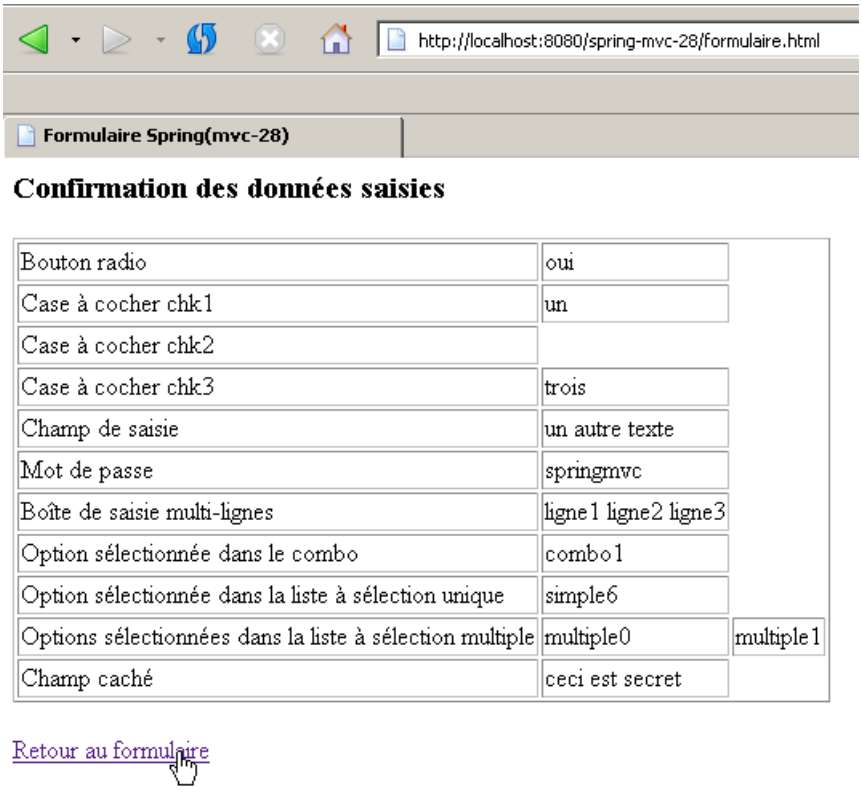
On réunit tout ce qu'on a vu précédemment dans un unique formulaire :



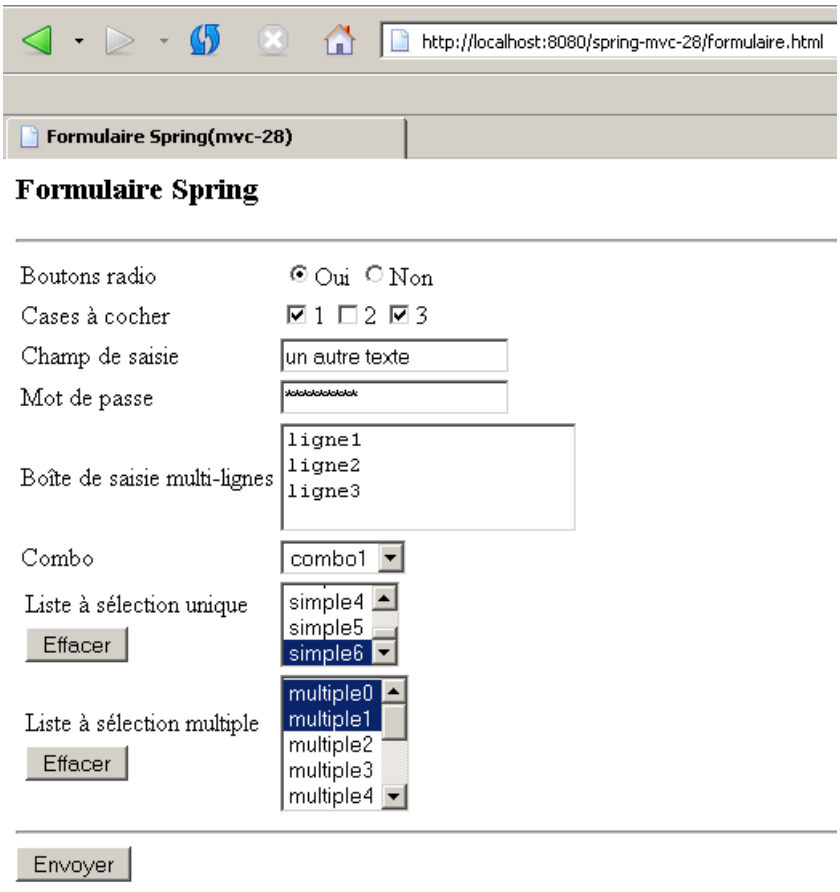
L'utilisateur fait ses saisies :



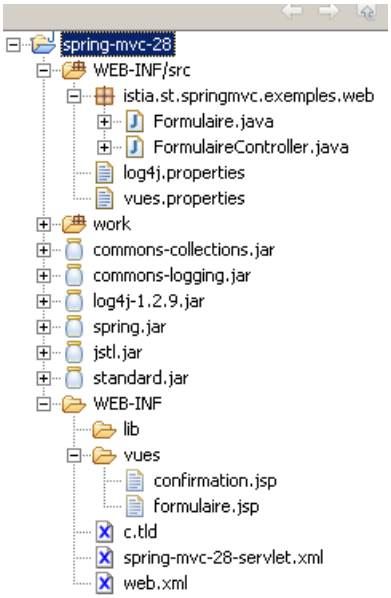
Une page de confirmation lui est envoyée :



Le lien ci-dessus permet de retrouver le formulaire tel qu'il a été posté :



Le projet Eclipse est le suivant :



Une nouveauté apparaît. Dans [WEB-INF/src] on retrouve un contrôleur : **FormulaireController**.

La classe [Formulaire] doit refléter tous les champs de saisie du formulaire HTML. Son code est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Formulaire {
4.
5.    // boutons radio
6.    private String opt;
7.
8.    // cases à cocher
9.    private String chk1;
10.
11.   private String chk2;
12.
13.   private String chk3;
14.
15.   // champs de saisie
16.   private String champSaisie;
17.
18.   private String mdp;
19.
20.   private String boiteSaisie;
21.
22.   // liste déroulante
23.   private String combo;
24.
25.   private String[] optionsCombo;
26.
27.   // liste à sélection simple
28.   private String listeSimple;
29.
30.   private String[] optionsListeSimple;
31.
32.   // liste à sélection multiple
33.   private String[] listeMultiple;
34.
35.   private String[] optionsListeMultiple;
36.
37.   // champ caché
38.   private String secret;
39.
40.   // initialisation formulaire
41.   public Formulaire() {
42.     // initialisations du formulaire
43.     this.setOpt("non");
44.     this.setChk2("deux");
45.     this.setChampSaisie("tapez un texte");
46.     this.setMdp("mdporiginel");
47.     this.setBoiteSaisie("ligne1");
48.     this.setOptionsCombo(getOptions(5, "combo"));
49.     this.setCombo("combo3");
50.     this.setOptionsListeSimple(getOptions(7, "simple"));
51.     this.setListeSimple("simple2");
52.     this.setOptionsListeMultiple(getOptions(10, "multiple"));
53.     this.setListeMultiple(new String[] { "multiple1", "multiple3" });
54.     this.setSecret("ceci est secret");
```

```
  }

  // getters et setters
  public String getOpt() {
    return opt;
  }
...
  public void setOptionsListeMultiple(String[] optionsListeMultiple) {
    this.optionsListeMultiple = optionsListeMultiple;
  }

  // générateur de valeurs
  private String[] getOptions(int taille, String label) {
    String[] options = new String[taille];
    for (int i = 0; i < taille; i++) {
      options[i] = label + i;
    }
    return options;
  }

}
```

Nous ne commentons pas ce code. Il a été obtenu par la réunion des codes des différentes classes [Formulaire] que nous avons construites.

La page [formulaire.jsp] à l'origine de la page affichée lors du GET est également la réunion de toutes les pages [formulaire.jsp] que nous avons construites :

```
<%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
<%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
<%@ page isELIgnored="false" %>
<html>
  <head>
    <title>Formulaire Spring(mvc-28)</title>
  </head>
  <body>
    <h3>Formulaire Spring</h3>
    <hr>
    <form method="post">
      <table border="0">
        <!-- boutons radio -->
        <tr>
          <td>Boutons radio</td>
          <td>
            <c:choose>
              <c:when test='${formulaire.opt=="oui"}'>
                <input type="radio" name="opt" value="oui" checked>Oui
                <input type="radio" name="opt" value="non">Non
              </c:when>
              <c:otherwise>
                <input type="radio" name="opt" value="oui">Oui
                <input type="radio" name="opt" value="non" checked>Non
              </c:otherwise>
            </c:choose>
          </td>
        </tr>
        <!-- cases à cocher -->
        <tr>
          <td>Cases &agrave; cocher</td>
          <td>
            <c:choose>
              <c:when test='${formulaire.chk1=="un"}'>
                <input type="checkbox" name="chk1" value="un" checked>1
              </c:when>
              <c:otherwise>
                  <input type="checkbox" name="chk1" value="un">1
              </c:otherwise>
            </c:choose>
            <c:choose>
              <c:when test='${formulaire.chk2=="deux"}'>
                <input type="checkbox" name="chk2" value="deux" checked>2
              </c:when>
              <c:otherwise>
                  <input type="checkbox" name="chk2" value="deux">2
              </c:otherwise>
            </c:choose>
            <c:choose>
              <c:when test='${formulaire.chk3=="trois"}'>
                <input type="checkbox" name="chk3" value="trois" checked>3
              </c:when>
              <c:otherwise>
                  <input type="checkbox" name="chk3" value="trois">3
              </c:otherwise>
            </c:choose>
          </td>
        </tr>
        <!-- champ de saisie -->
        <tr>
          <td>Champ de saisie</td>
          <td>
            <input type="text" name="champSaisie" value="${formulaire.champSaisie}">
          </td>
        </tr>
```

```
66.         <!-- mot de passe -->
67.         <tr>
68.           <td>Mot de passe</td>
69.           <td>
70.             <input type="password" name="mdp" value="${formulaire.mdp}">
71.           </td>
72.         </tr>
73.         <!-- Boîte de saisie multi-lignes -->
74.         <tr>
75.           <td>Bo&icirc;te de saisie multi-lignes</td>
76.           <td>
77.             <textarea name="boiteSaisie" rows="3">${formulaire.boiteSaisie}</textarea>
78.           </td>
79.         </tr>
80.         <!-- Combo -->
81.         <tr>
82.           <td>Combo</td>
83.           <td>
84.           <select name="combo" rows="3">
85.             <c:forEach items="${formulaire.optionsCombo}" var="optionCombo">
86.               <c:choose>
87.                 <c:when test="${formulaire.combo==optionCombo}">
88.                   <option selected>${optionCombo}</option>
89.                 </c:when>
90.                 <c:otherwise>
91.                   <option>${optionCombo}</option>
92.                 </c:otherwise>
93.               </c:choose>
94.             </c:forEach>
95.           </select>
96.           </td>
97.         </tr>
98.         <!-- Liste à sélection unique -->
99.         <tr>
100.          <td>
101.            <table>
102.              <tr>
103.                <td>Liste &agrave; s&eacute;lection unique</td>
104.              </tr>
105.              <tr>
106.                <td>
107.                  <input type="button" value="Effacer" onclick="this.form.listeSimple.selectedIndex=-1"/>
108.                </td>
109.              </tr>
110.            </table>
111.          </td>
112.          <td>
113.          <select name="listeSimple" size="3">
114.            <c:forEach items="${formulaire.optionsListeSimple}" var="optionListeSimple">
115.              <c:choose>
116.                <c:when test="${formulaire.listeSimple==optionListeSimple}">
117.                  <option selected>${optionListeSimple}</option>
118.                </c:when>
119.                <c:otherwise>
120.                  <option>${optionListeSimple}</option>
121.                </c:otherwise>
122.              </c:choose>
123.            </c:forEach>
124.          </select>
125.          </td>
126.         </tr>
127.         <!-- Liste à sélection multiple -->
128.         <tr>
129.           <td>
130.             <table>
131.               <tr>
132.                 <td>Liste &agrave; s&eacute;lection multiple</td>
133.               </tr>
134.               <tr>
135.                 <td>
136.                   <input type="button" value="Effacer" onclick="this.form.listeMultiple.selectedIndex=-1"/>
137.                 </td>
138.               </tr>
139.             </table>
140.           </td>
141.           <td>
142.           <select name="listeMultiple" size="5" multiple>
143.             <c:forEach items="${formulaire.optionsListeMultiple}" var="optionListeMultiple">
144.               <c:set var="selection" value="faux"/>
145.               <c:forEach items="${formulaire.listeMultiple}" var="selectionListeMultiple">
146.                 <c:if test="${selectionListeMultiple==optionListeMultiple}">
147.                   <c:set var="selection" value="vrai"/>
148.                 </c:if>
149.                 </c:forEach>
150.               <c:choose>
151.                 <c:when test='${selection=="vrai"}'>
152.                   <option selected>${optionListeMultiple}</option>
153.                 </c:when>
154.                 <c:otherwise>
155.                   <option>${optionListeMultiple}</option>
156.                 </c:otherwise>
157.               </c:choose>
158.             </c:forEach>
159.           </select>
160.           </td>
161.         </tr>
162.       </table>
163.       <input type="hidden" name="secret" value="${formulaire.secret}">
```

```
164.      <hr>
165.      <input type="hidden" name="_chk1">
166.      <input type="hidden" name="_chk2">
167.      <input type="hidden" name="_chk3">
168.      <input type="hidden" name="_listeSimple">
169.      <input type="hidden" name="_listeMultiple">
170.      <input type="submit" value="Envoyer">
171.    </form>
172.  </body>
173.</html>
```

La page [confirmation.jsp] à l'origine de la page affichée à l'issue du POST est également la réunion de toutes les pages [confirmation.jsp] que nous avons construites :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Formulaire Spring(mvc-28)</title>
8.    </head>
9.    <body>
10.     <h3>Confirmation des donn&eacute;es saisies</h3>
11.     <table border="1">
12.       <tr>
13.         <td>Bouton radio</td>
14.         <td>${formulaire.opt}</td>
15.       </tr>
16.       <tr>
17.         <td>Case &agrave; cocher chk1</td>
18.         <td>${formulaire.chk1}</td>
19.       </tr>
20.       <tr>
21.         <td>Case &agrave; cocher chk2</td>
22.         <td>${formulaire.chk2}</td>
23.       </tr>
24.       <tr>
25.         <td>Case &agrave; cocher chk3</td>
26.         <td>${formulaire.chk3}</td>
27.       </tr>
28.       <tr>
29.         <td>Champ de saisie</td>
30.         <td>${formulaire.champSaisie}</td>
31.       </tr>
32.       <tr>
33.         <td>Mot de passe</td>
34.         <td>${formulaire.mdp}</td>
35.       </tr>
36.       <tr>
37.         <td>Bo&icirc;te de saisie multi-lignes</td>
38.         <td>${formulaire.boiteSaisie}</td>
39.       </tr>
40.       <tr>
41.         <td>Option s&eacute;lectionn&eacute;e dans le combo</td>
42.         <td>${formulaire.combo}</td>
43.       </tr>
44.       <tr>
45.         <td>Option s&eacute;lectionn&eacute;e dans la liste &agrave; s&eacute;lection unique</td>
46.         <td>${formulaire.listeSimple}</td>
47.       </tr>
48.       <tr>
49.         <td>Options s&eacute;lectionn&eacute;es dans la liste &agrave; s&eacute;lection multiple</td>
50.         <c:forEach items="${formulaire.listeMultiple}" var="selectionListeMultiple">
51.           <td>${selectionListeMultiple}</td>
52.         </c:forEach>
53.       </tr>
54.       <tr>
55.         <td>Champ cach&eacute;</td>
56.         <td>${formulaire.secret}</td>
57.       </tr>
58.     </table>
59.     <br>
60.     <a href="<c:url value="/formulaire.html"/>">Retour au formulaire</a>
61.   </body>
62. </html>
```

Seule nouveauté : le lien de la ligne 60 ci-dessus. Une fois cliqué, il va provoquer un GET vers l'url [/spring-mvc-28/formulaire.html]. Tout se passe comme si le formulaire était demandé pour la première fois. Rappelons la configuration du contrôleur [SimpleFormController] qui avait été faite précédemment :

```
1.<bean id="FormulaireController"
2.    class="istia.st.springmvc.exemples.web.SimpleFormController">
3.    <property name="sessionForm">
4.      <value>true</value>
5.    </property>
6.    <property name="commandClass">
7.      <value>istia.st.springmvc.exemples.web.Formulaire</value>
8.    </property>
9.    <property name="commandName">
10.     <value>formulaire</value>
11.   </property>
12.   <property name="formView">
```

```
13.     <value>formulaire</value>
14.   </property>
15.   <property name="successView">
16.     <value>confirmation</value>
17.   </property>
18. </bean>
```

1. lors d'un GET, le contrôleur [SimpleFormController] crée un nouvel objet [Formulaire] tel que défini ligne 7.
2. à cause de l'attribut **sessionForm** des lignes 3-5, cet objet [Formulaire] sera mis dans la session
3. au moment du POST, il est récupéré dans la session pour être modifié par les valeurs postées. Il est ensuite mis dans le modèle de la vue **successView** défini aux lignes 15-17. Ensuite, il est supprimé de la session.
4. Lorsqu'arrive un GET suivant, d'une part l'objet [Formulaire] n'est plus dans la session, d'autre part l'étape 1 précédente ne l'y aurait pas cherché. C'est bien un nouvel objet [Formulaire] qui est créé par défaut.

Dans cette application, on voudrait que lors du second GET de l'étape 4 ci-dessus, ce soit l'objet [Formulaire] issu du POST de l'étape 3 qui soit mis dans le modèle et non un nouvel objet [Formulaire] tout neuf. Comme le comportement par défaut de [SimpleFormController] ne convient pas, il nous faut dériver cette classe. C'est ce que nous faisons avec le contrôleur de type [FormulaireController] :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import javax.servlet.http.HttpServletRequest;
4.  import javax.servlet.http.HttpServletResponse;
5.
6.  import org.springframework.validation.BindException;
7.  import org.springframework.web.servlet.ModelAndView;
8.  import org.springframework.web.servlet.mvc.SimpleFormController;
9.
10. public class FormulaireController extends SimpleFormController {
11.
12.   protected Object formBackingObject(HttpServletRequest request) {
13.     // on récupère le formulaire dans la session s'il existe
14.     Object formulaire = request.getSession().getAttribute("formulaire");
15.     if (formulaire == null) {
16.       formulaire = new Formulaire();
17.     }
18.     // on rend le formulaire
19.     return formulaire;
20.   }
21.
22.   // traitement du POST
23.   protected ModelAndView onSubmit(HttpServletRequest request,
24.       HttpServletResponse response, Object formulaire,
25.       BindException errors) {
26.     // on met le formulaire dans la session
27.     request.getSession().setAttribute("formulaire", formulaire);
28.     // on rend un [ModelAndView]
29.     return new ModelAndView("confirmation",null);
30.   }
31. }
```

- ligne 10 : la classe dérive de [SimpleFormController]

Pour comprendre le code ci-dessus, le lecteur est invité à relire le fonctionnement de [SimpleFormController] au moment d'un GET (page 27) et au moment du POST (page 27).

GET

Dans la configuration précédente de [SimpleFormController] l'objet créé au moment du GET pour initialiser le formulaire était une instance de la classe définie par l'attribut [**commandClass**] des lignes 6-8 de la configuration de [SimpleFormController] (voir plus haut). Si on ne fournit pas cet attribut, la méthode [**formBackingObject**] de [SimpleFormController] est appelée pour le fournir. C'est ce que nous faisons ci-dessus en redéfinissant, lignes 12-20, cette méthode qui par défaut ne fait rien :

- ligne 14 : on cherche un objet de clé " **formulaire** " dans la session. Si on le trouve, alors l'objet trouvé est le formulaire posté qui a été mis en session au moment du POST par la méthode **onSubmit** que nous allons décrire prochainement.
- lignes 15-17 : si on n'a pas trouvé d'instance [Formualaire] dans la session, alors on en crée une nouvelle.

POST

Au moment du POST, une chaîne de méthodes **onSubmit** est appelée jusqu'à une méthode finale **doSubmit**. Pour modifier le comportement par défaut de [SimpleFormController], il nous faut redéfinir l'une des méthodes de la chaîne. Le choix se fait sur les paramètres des différentes méthodes **onSubmit**. Ici, on veut mettre en session l'objet [Formulaire] qui vient de recevoir les valeurs postées. Il était déjà en session (sessionForm=true) mais on sait qu'il ne va pas y rester. Par ailleurs, on ne connaît pas la clé qui lui est associée. La documentation de Spring ne nous renseigne pas sur ce point. Donc on veut le mettre en session nous-mêmes. Pour

cela, il nous faut l'objet [request] de la requête en cours de traitement. Seule une méthode parmi les trois méthodes **onSubmit** reçoit ce paramètre. Nous utilisons donc cette méthode **onSubmit** particulière, lignes 23-25.

- ligne 27 : on met le formulaire dans la session. Se pose le problème du nom de la clé associée. On sait que la page [confirmation.jsp] attend qu'une clé **formulaire** associée à une instance [**Formulaire**] soit trouvée dans l'un des contextes **request**, **session**, **application**. Dans le comportement par défaut de [SimpleFormController], la chaîne des méthodes **onSubmit** mettait l'instance [**Formulaire**] mise à jour par le POST, dans le contexte de la requête, associée à la clé définie par l'attribut [**commandName**], ici " **formulaire** ". Ce comportement par défaut n'existe plus. Comme la page [confirmation.jsp] attend néanmoins une clé **formulaire**, c'est cette clé que nous utilisons pour l'instance [Formulaire] que nous mettons en session.
- ligne 29 : la méthode [onSubmit] doit rendre un [ModelAndView] à [DispatcherServlet]. Nous voulons afficher la vue nommée " **confirmation** " qui correspond à la page JSP [confirmation.jsp]. Nous ne rendons pas de modèle (second paramètre égal à **null**). Pourquoi ? Parce que c'est inutile. [DispatcherServlet] mettra le modèle que nous allons lui rendre dans le contexte de la requête. Notre modèle est constitué de l'instance [Formulaire] déjà mise en session. Il est donc inutile, qu'en plus, elle aille dans le contexte de la requête.

Il ne nous reste plus qu'à donner le détail du fichier de configuration de l'application [spring-mvc-28-servlet.xml] :

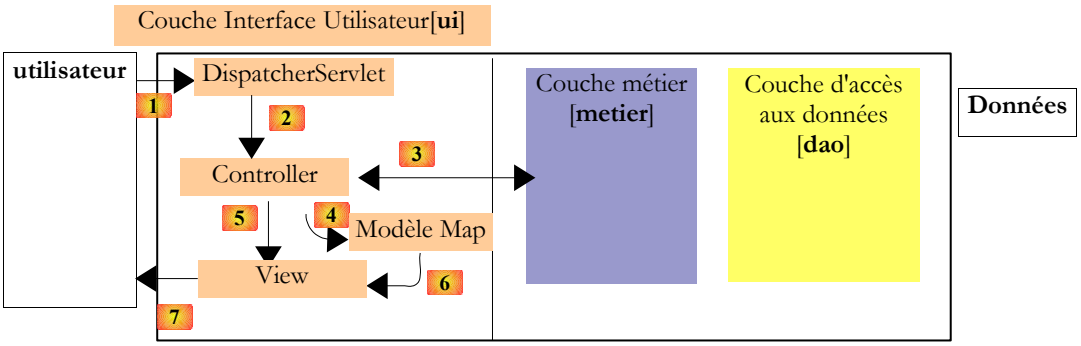
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les mappings de l'application-->
5.    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
6.      <property name="mappings">
7.        <props>
8.          <prop key="/formulaire.html">FormulaireController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="FormulaireController"
14.     class="istia.st.springmvc.exemples.web.FormulaireController">
15.     <property name="sessionForm">
16.       <value>true</value>
17.     </property>
18.     <property name="commandName">
19.       <value>formulaire</value>
20.     </property>
21.     <property name="formView">
22.       <value>formulaire</value>
23.     </property>
24.   </bean>
25.   <!-- le résolveur de vues -->
26.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
27.     <property name="basename">
28.       <value>vues</value>
29.     </property>
30.   </bean>
31. </beans>
```

La configuration du contrôleur, lignes 13-23 a changé :

- ligne 14 : le contrôleur est une instance de [**FormulaireController**] et non plus de [**SimpleFormController**]
- on n'a gardé que certains paramétrages. Ont disparu :
  - l'attribut " **successView** " qui est le nom de la vue à afficher à l'issue du POST. La méthode **onSubmit** fixe elle-même ce nom.
  - l'attribut " **commandClass** " qui fixe le type de l'objet à instancier pour récupérer les valeurs postées. Ici, c'est la méthode [**formBackingObject**] qui crée elle-même cette instance.

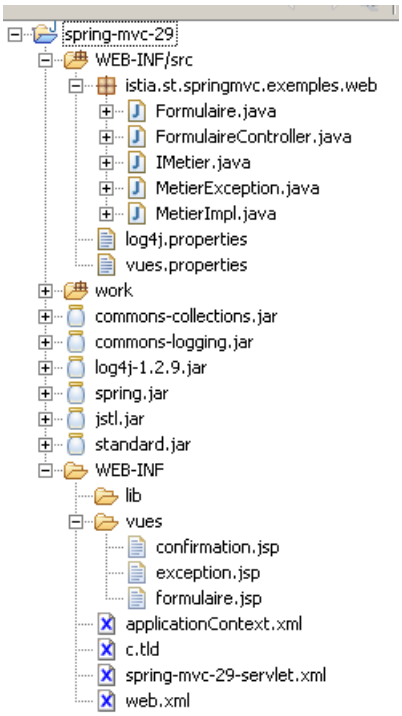
5.9 Formulaire dans une architecture 3tier

Nous reprenons la même application que précédemment mais nous la plaçons dans une architecture 3tier :



Dans l'application précédente, la méthode [**formBackingObject**] du [Controller] créait lui-même l'instance [Formulaire] miroir du formulaire HTML. Ici la méthode [**formBackingObject**] va demander l'instance [Formulaire] à la couche [métier] (étape 3 ci-dessus).

Le projet Eclipse est le suivant :



Dans [WEB-INF/src] on trouve les classes Java du projet :

- **IMetier** : interface de la couche [métier]
- **MetierImpl** : implémentation de l'interface [IMetier]
- **MetierException** : exception lancée par la couche [métier]
- **Formulaire** : la classe miroir du formulaire
- **FormulaireController** : le contrôleur qui va gérer le GET et le POST du formulaire

L'interface [Imetier] a pour rôle de fournir à la couche web, une instance initialisée de [Formulaire]. Celle-ci est demandée au moment du GET afin de fournir à l'utilisateur un formulaire pré-initialisé. L'idée ici, est de reproduire un cas courant d'usage : celui où le formulaire affiché est initialisé avec un objet provenant d'une base de données. Il est alors demandé à la couche [métier] qui le demande à la couche [dao] qui le récupère auprès d'un SGBD. Ici, nous nous contenterons de ne faire travailler que la seule couche [métier].

La classe [Formulaire], miroir du formulaire HTML sera la suivante :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Formulaire {
4.
5.    // boutons radio
6.    private String opt;
7.
8.    // cases à cocher
```

```
9.   private String chk1;
10.  private String chk2;
11.  private String chk3;
12.
13.  // champs de saisie
14.  private String champSaisie;
15.  private String mdp;
16.  private String boiteSaisie;
17.
18.  // liste déroulante
19.  private String combo;
20.
21.  // liste à sélection simple
22.  private String listeSimple;
23.
24.  // liste à sélection multiple
25.  private String[] listeMultiple;
26.
27.  // champ caché
28.  private String secret;
29.
30.  // getters et setters
31.  public String getOpt() {
32.    return opt;
33.  }
34. ...
35.  public void setCombo(String combo) {
36.    this.combo = combo;
37.  }
38.
39. }
```

On a restreint la classe [Formulaire] aux seuls champs qui correspondent à un champ HTML de saisie, c.a.d. aux seuls champs qui vont recevoir une valeur au moment du POST. Les autres données (options des trois listes) ont été externalisées et seront fournies par la couche [métier]. Par ailleurs, la classe [Formulaire] n'a plus de constructeur.

L'interface [Imetier] sera la suivante :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public interface IMetier {
4.    // instance de [Formulaire]
5.    public Formulaire getFormulaire();
6.    // options de la liste déroulante
7.    public String[] getOptionsCombo();
8.    // options de la liste à sélection unique
9.    public String[] getOptionsListeSimple();
10.   // options de la liste à sélection multiple
11.   public String[] getOptionsListeMultiple();
12. }
```

L'interface [IMetier] fournit toutes les données qui doivent être placées dans le modèle de la page JSP [formulaire.jsp] :

1. ligne 5 : une instance [Formulaire]
2. ligne 7 : les options qui vont alimenter la liste déroulante "**combo**" du formulaire
3. ligne 9 : les options qui vont alimenter la liste déroulante "**listeSimple**" du formulaire
4. ligne 11 : les options qui vont alimenter la liste déroulante "**listeMultiple**" du formulaire

L'implémentation [MetierImpl] de l'interface [IMetier] sera la suivante :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Random;
4.
5.  public class MetierImpl implements IMetier {
6.
7.    private Random random=new Random();
8.
9.    public Formulaire getFormulaire() {
10.     if(random.nextInt(2)==0){
11.       // création formulaire
12.       Formulaire formulaire=new Formulaire();
13.       // initialisations du formulaire
14.       formulaire.setOpt("non");
15.       formulaire.setChk2("deux");
16.       formulaire.setChampSaisie("tapez un texte");
17.       formulaire.setMdp("mdporiginel");
18.       formulaire.setBoiteSaisie("ligne1");
19.       formulaire.setCombo("combo3");
20.       formulaire.setListeSimple("simple2");
21.       formulaire.setListeMultiple(new String[]{"multiple1","multiple3"});
```

```
22.      formulaire.setSecret("ceci est secret");
23.      // on rend le formulaire
24.      return formulaire;
25.    }else{
26.      throw new MetierException("Exception dans la couche métier");
27.    }
28.  }
29.
30.  public String[] getOptionsCombo(){
31.    return getOptions(5, "combo");
32.  }
33.
34.  public String[] getOptionsListeSimple(){
35.    return getOptions(7, "simple");
36.  }
37.
38.  public String[] getOptionsListeMultiple(){
39.    return getOptions(10, "multiple");
40.  }
41.
42.  // générateur de liste de valeurs
43.  private String[] getOptions(int taille, String label) {
44.    String[] options = new String[taille];
45.    for (int i = 0; i < taille; i++) {
46.      options[i] = label + i;
47.    }
48.    return options;
49.  }
50.
51.
52. }
```

- ligne 5 : on implémente l'interface [IMetier]
- ligne 7 : un générateur de nombre aléatoires est initialisé lors de la création d'une instance de la classe
- lignes 9-28 : la méthode [getFormulaire] qui rend une instance de la classe [Formulaire] initialisée. De façon aléatoire, en moyenne une fois sur deux (ligne 10), une exception est lancée (ligne 26). Elle est de type [MetierException], un type que nous allons décrire prochainement. Notre application devra gérer cette exception.
- lignes 30-32 : la méthode [**getOptionsCombo**] qui rend le tableau qui va alimenter la liste déroulante "**combo**" du formulaire HTML
- lignes 34-36 : la méthode [**getOptionsListeSimple**] qui rend le tableau qui va alimenter la liste à sélection unqiue "**listeSimple**" du formulaire HTML
- lignes 38-40 : la méthode [**getOptionsListeMultiple**] qui rend le tableau qui va alimenter la liste à sélection multiple "**listeMultiple**" du formulaire HTML
- lignes 43-49 : une méthode privée pour générer les valeurs nécessaires aux trois méthodes précédentes

La classe [MetierException] est une exception non contrôlée dérivée de [RuntimeException]. Elle se contente de faire appel aux constructeurs de sa classe parent.

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class MetierException extends RuntimeException {
4.
5.    public MetierException() {
6.      super();
7.    }
8.
9.    public MetierException(String message) {
10.     super(message);
11.   }
12.
13.   public MetierException(String message, Throwable cause) {
14.     super(message, cause);
15.   }
16.
17.   public MetierException(Throwable cause) {
18.     super(cause);
19.   }
20.
21. }
```

Le contrôleur [FormulaireController] de l'application sera le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.HashMap;
4.  import java.util.Map;
5.
6.  import javax.servlet.http.HttpServletRequest;
7.  import javax.servlet.http.HttpServletResponse;
8.
```

```
9.  import org.springframework.validation.BindException;
10. import org.springframework.web.servlet.ModelAndView;
11. import org.springframework.web.servlet.mvc.SimpleFormController;
12.
13. public class FormulaireController extends SimpleFormController {
14.
15.  // la couche métier
16.  private IMetier metier;
17.
18.  public IMetier getMetier() {
19.    return metier;
20.  }
21.
22.  public void setMetier(IMetier metier) {
23.    this.metier = metier;
24.  }
25.
26.  // données nécessaires au formulaire web
27.  // autres que celles déjà contenues dans l'objet Formulaire
28.  protected Map referenceData(HttpServletRequest request){
29.    // le dictionnaire
30.    Map data=new HashMap();
31.    // les options du combo
32.    data.put("optionsCombo",metier.getOptionsCombo());
33.    // les options de la liste à sélection unique
34.    data.put("optionsListeSimple",metier.getOptionsListeSimple());
35.    // les options de la liste à sélection multiple
36.    data.put("optionsListeMultiple",metier.getOptionsListeMultiple());
37.    // on rend le dictionnaire
38.    return data;
39.  }
40.
41.  // l'objet adossé au formulaire
42.  protected Object formBackingObject(HttpServletRequest request) {
43.    // on récupère le formulaire dans la session s'il existe
44.    Formulaire formulaire = (Formulaire)request.getSession().getAttribute("formulaire");
45.    if (formulaire == null) {
46.      // on demande un objet Formulaire à la couche métier
47.      formulaire = metier.getFormulaire();
48.    }
49.    // on rend le formulaire
50.    return formulaire;
51.  }
52.
53.  // traitement du POST
54.  protected ModelAndView onSubmit(HttpServletRequest request,
55.      HttpServletResponse response, Object formulaire,
56.      BindException errors) {
57.    // on met le formulaire dans la session
58.    request.getSession().setAttribute("formulaire", formulaire);
59.    // on rend un [ModelAndView]
60.    return new ModelAndView("confirmation",null);
61.  }
62. }
```

- ligne 13 : comme précédemment, [FormulaireController] dérive de [SimpleFormController]
- ligne 16 : une référence à la couche [métier]. En effet, le contrôleur doit dialoguer avec cette couche. Le champ [métier] est du type de l'interface [IMetier] et non du type d'une classe d'implémentation particulière. Ce champ sera initialisé par Spring au démarrage de l'application web.
- lignes 42-51 : la méthode [**formBackingObject**] n'a pas changé dans son essence. Elle doit rendre une instance de [Formulaire], instance qui sera mise dans le modèle de [formulaire.jsp]. Elle sera mise également en session (sessionForm=true) pour y être récupérée au moment du POST et recevoir les valeurs postées.
- ligne 44 : la méthode [**formBackingObject**] cherche d'abord une instance de [Formulaire] dans la session, associée à la clé " formulaire ". C'est la méthode [onSubmit] qui l'y mettra sous ce nom.
- lignes 45-48 : si aucune instance de [Formulaire] n'a été trouvée dans la session, elle est demandée à la couche [métier]. C'est la différence avec la version précédente du contrôleur qui créait lui-même l'instance [Formulaire].
- ligne 50 : l'objet [Formulaire] est rendu. Il est destiné à être intégré au modèle de la page [formulaire.jsp]. Ce modèle est insuffisant. On n'y trouve pas les valeurs qui doivent alimenter les trois listes HTML : **combo**, **listeSimple** et **listeMultiple**.

Le lecteur est invité ici à relire le flux d'exécution du GET page 27. L'étape 4 y est définie comme suit :

> La méthode [**showForm**] de [SimpleFormController] est appelée. C'est elle qui va rendre l'objet [ModelAndView] attendu par [DispatcherServlet]. L'objet [**command**] construit en (2) sera placé dans le modèle sous un nom [**commandName**] généralement fixé par configuration. Par ailleurs, [showForm] va appeler la méthode [**referenceData**] de [SimpleFormController]. Cette méthode rend un dictionnaire [Map] qui est ajouté au modèle. L'implémentation par défaut de [**referenceData**] rend la référence *null*. Si on veut dans le modèle, d'autres éléments que l'objet de type [**command**]

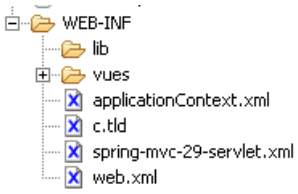
construit en (2), on redéfinira la méthode [referenceData]. La méthode [showForm] rend un objet [ModelAndView] où le nom de la vue est celui défini par l'attribut " **formView** " dans la configuration du contrôleur.

On voit que si le modèle construit par [formBackingObject] est insuffisant, on peut le compléter en redéfinissant la méthode [**referenceData**]. C'est ce qui est fait lignes 28-39 :

- ligne 28 : la méthode [**referenceData**] reçoit l'objet [request] qui encapsule la requête du client en cours de traitement. Ici, nous ne servirons pas de ce paramètre.
- ligne 30 : on crée le conteneur du modèle qu'on veut créer
- ligne 32 : on demande à la couche [métier] les valeurs de la liste déroulante HTML **combo** et on les met dans le modèle
- ligne 34 : on fait de même pour les valeurs de la liste HTML à sélection unique **listeSimple**
- ligne 36 : on fait de même pour les valeurs de la liste HTML à sélection multiple **listeMultiple**
- ligne 38 : on rend le modèle.

Au final, le modèle à l'issue du GET, sera composé de tous les éléments nécessaires à l'affichage de la page JSP [formulaire.jsp]. Il aura été construit à la fois par [**formBackingObject**] et [**referenceData**].

Détaillons maintenant la configuration de l'application :



On voit apparaître le fichier [applicationContext.xml] qui configure le contexte de l'application, celui qui est vu par tous les utilisateurs. On sait que pour que ce fichier soit exploité, un **listener** doit être déclaré dans le fichier [web.xml] :

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.
3.  <!DOCTYPE web-app PUBLIC
4.    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5.    "http://java.sun.com/dtd/web-app_2_3.dtd">
6.  <web-app>
7.    <!-- le chargeur du contexte spring de l'application -->
8.    <listener>
9.      <listener-class>
10.       org.springframework.web.context.ContextLoaderListener</listener-class>
11.   </listener>
12.   <!-- la servlet -->
13.   <servlet>
14.     <servlet-name>spring-mvc-29</servlet-name>
15.     <servlet-class>
16.       org.springframework.web.servlet.DispatcherServlet</servlet-class>
17.   </servlet>
18.   <!-- le mapping des url -->
19.   <servlet-mapping>
20.     <servlet-name>spring-mvc-29</servlet-name>
21.     <url-pattern>*.html</url-pattern>
22.   </servlet-mapping>
23. </web-app>
```

- lignes 8-11 : le listener qui va exploiter le contenu du fichier [applicationContext.xml]

Le fichier [applicationContext.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- couche métier -->
5.    <bean id="metier" class="istia.st.springmvc.exemples.web.MetierImpl"/>
6.  </beans>
```

- ligne 5 : on déclare la classe qui va implémenter l'interface [IMetier]. Si on doit changer d'implémentation, on change cette ligne mais on ne touche pas au code Java. Le bean est déclaré avec l'id " **metier** ". C'est sous cet identifiant qu'il est référencé dans le fichier [spring-mvc-29-servlet.xml] qui configure la couche web de l'architecture 3tier.

Le fichier [spring-mvc-29-servlet.xml] est le suivant :

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

```
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les mappings de l'application-->
5.    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
6.      <property name="mappings">
7.        <props>
8.          <prop key="/formulaire.html">FormulaireController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="FormulaireController"
14.     class="istia.st.springmvc.exemples.web.FormulaireController">
15.     <property name="sessionForm">
16.       <value>true</value>
17.     </property>
18.     <property name="commandName">
19.       <value>formulaire</value>
20.     </property>
21.     <property name="formView">
22.       <value>formulaire</value>
23.     </property>
24.     <property name="metier">
25.       <ref bean="metier"/>
26.     </property>
27.   </bean>
28.   <!-- le résolveur de vues -->
29.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
30.     <property name="basename">
31.       <value>vues</value>
32.     </property>
33.   </bean>
34.   <!-- le gestionnaire d'exceptions -->
35.   <bean
36.     class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
37.     <property name="exceptionAttribute">
38.       <value>exception</value>
39.     </property>
40.     <property name="defaultStatusCode">
41.       <value>500</value>
42.     </property>
43.     <property name="defaultErrorView">
44.       <value>exception</value>
45.     </property>
46.   </bean>
47. </beans>
```

- lignes 5-11 : seule l'URL [**/formulaire.html**] est gérée. Elle le sera par un contrôleur, instance d'id [**FormulaireController**] (ligne 14).
- lignes 29-33 : les vues seront définies dans un fichier [**vues.properties**] placé dans le *ClassPath* de l'application.
- lignes 35-46 : le gestionnaire d'exceptions. Le gestionnaire d'exceptions est (ligne 36) de type [**SimpleMappingExceptionResolver**], un gestionnaire présenté au paragraphe 4, page 19. Ici, le gestionnaire ne gère aucune exception particulière (absence de la propriété **exceptionMappings**). Pour toute exception remontant jusqu'à [DispatcherServlet], ce sera donc l'attribut **defaultErrorView,** ligne 44, qui sera utilisé. L'exception sera mise dans le modèle sous la clé **exception** (attribut **exceptionAttribute** ligne 38) de la vue appelée **exception** (attribut **defaultErrorView** ligne 44). Le client se verra renvoyé le code HTTP 500 (attribut **defaultStatusCode** ligne 41).

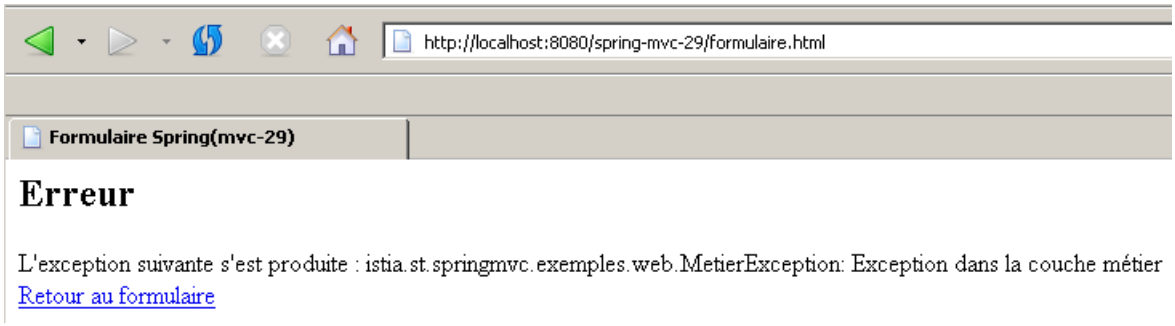
  Un gestionnaire d'exceptions a été inclus dans la configuration car on se rappelle que la couche [métier] lance en moyenne une fois sur deux, une exception de type [MetierException], lorsqu'elle interrogée par le contrôleur au moment du GET sur le formulaire.

- lignes 13-27 : la configuration du contrôleur [**FormulaireController**]. Cette configuration est analogue à celle qui avait été faite dans l'exemple précédent, au détail près des lignes 24-26 qui vont injecter dans le champ [metier] de [FormulaireController] la référence de la couche métier (ligne 25) créée lors de l'initialisation de l'application par exploitation du fichier [applicationContext.xml]. Ainsi lorsque le contrôleur est créé, il détient une référence sur la couche [métier].

Il ne nous reste plus qu'à détailler les vues. Le fichier [**vues.properties**] est le suivant :

```
1.  #exception
2.  exception.class=org.springframework.web.servlet.view.JstlView
3.  exception.url=/WEB-INF/vues/exception.jsp
4.  #formulaire
5.  formulaire.class=org.springframework.web.servlet.view.JstlView
6.  formulaire.url=/WEB-INF/vues/formulaire.jsp
7.  #confirmation
8.  confirmation.class=org.springframework.web.servlet.view.JstlView
9.  confirmation.url=/WEB-INF/vues/confirmation.jsp
```

La seule nouveauté, vis à vis du fichier [**vues.properties**] des applications précédentes est l'apparition de la vue nommée **exception**. Celle-ci est référencée ligne 44 du fichier [spring-mvc-29-servlet.xml] et sert à afficher l'exception de type [MetierException] que va lancer la couche [métier] de façon aléatoire :



La page [exception.jsp] est la suivante :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Formulaire Spring(mvc-29)</title>
8.    </head>
9.    <body>
10.     <h2>Erreur</h2>
11.     L'exception suivante s'est produite :
12.     <c:out value="${exception}"/>
13.     <br>
14.     <a href="<c:url value="/formulaire.html"/>">Retour au formulaire</a>
15.   </body>
16. </html>
```

- ligne 12 : le texte de l'exception. C'est la méthode [toString] de l'exception qui est ici implicitement utilisée.
- ligne 14 : un lien de retour vers le formulaire. Un GET sera émis lors d'un clic sur ce lien.

La page JSP [formulaire.jsp] évolue un peu. En effet, précédemment, la totalité de son modèle était dans l'instance [Formulaire] associée à la clé " formulaire ". Ce n'est plus le cas. Son modèle est désormais éclaté sur quatre clés :

- **formulaire** : associée à l'instance [**Formulaire**]
- **optionsCombo** : associée à un tableau de chaînes de caractères qui seront les options de la liste HTML **combo**
- **optionsListeSimple** : associée à un tableau de chaînes de caractères qui seront les options de la liste HTML **listeSimple**
- **optionsListeMultiple** : associée à un tableau de chaînes de caractères qui seront les options de la liste HTML **listeMultiple**

Aussi, par exemple le code de génération de la liste HTML **listeSimple** devient le suivant :

```
1.  <select name="listeSimple" size="3">
2.  <c:forEach items="${optionsListeSimple}" var="optionListeSimple">
3.    <c:choose>
4.      <c:when test="${formulaire.listeSimple==optionListeSimple}">
5.        <option selected>${optionListeSimple}</option>
6.      </c:when>
7.      <c:otherwise>
8.        <option>${optionListeSimple}</option>
9.      </c:otherwise>
10.   </c:choose>
11. </c:forEach>
12. </select>
```

Dans la version précédente, la ligne 2 était la suivante :

```
<c:forEach items="${formulaire.optionsListeSimple}" var="optionListeSimple">
```

Ainsi le code de [formulaire.jsp] ne subit aucun changement autre que les suivants :

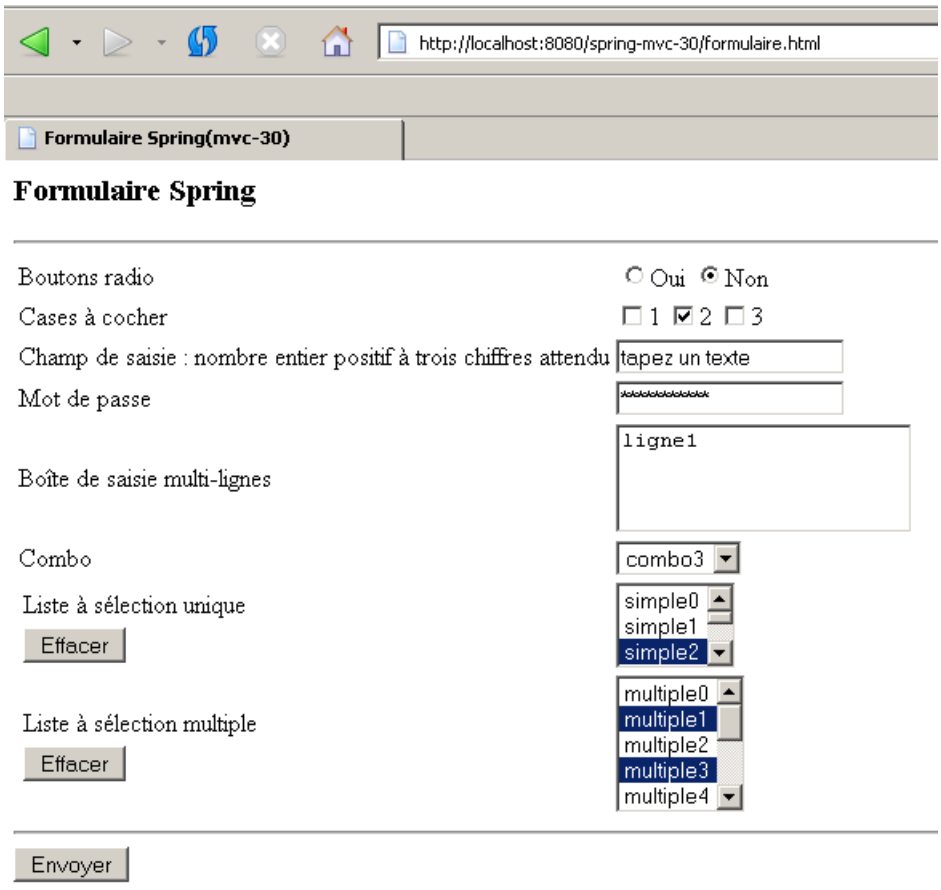
- ${formulaire.optionsCombo} devient ${optionsCombo}
- ${formulaire.optionsListeSimple} devient ${optionsListeSimple}
- ${formulaire.optionsListeMultiple} devient ${optionsListeMultiple}

La page JSP [confirmation.jsp] reste identique à celle de l'application précédente.

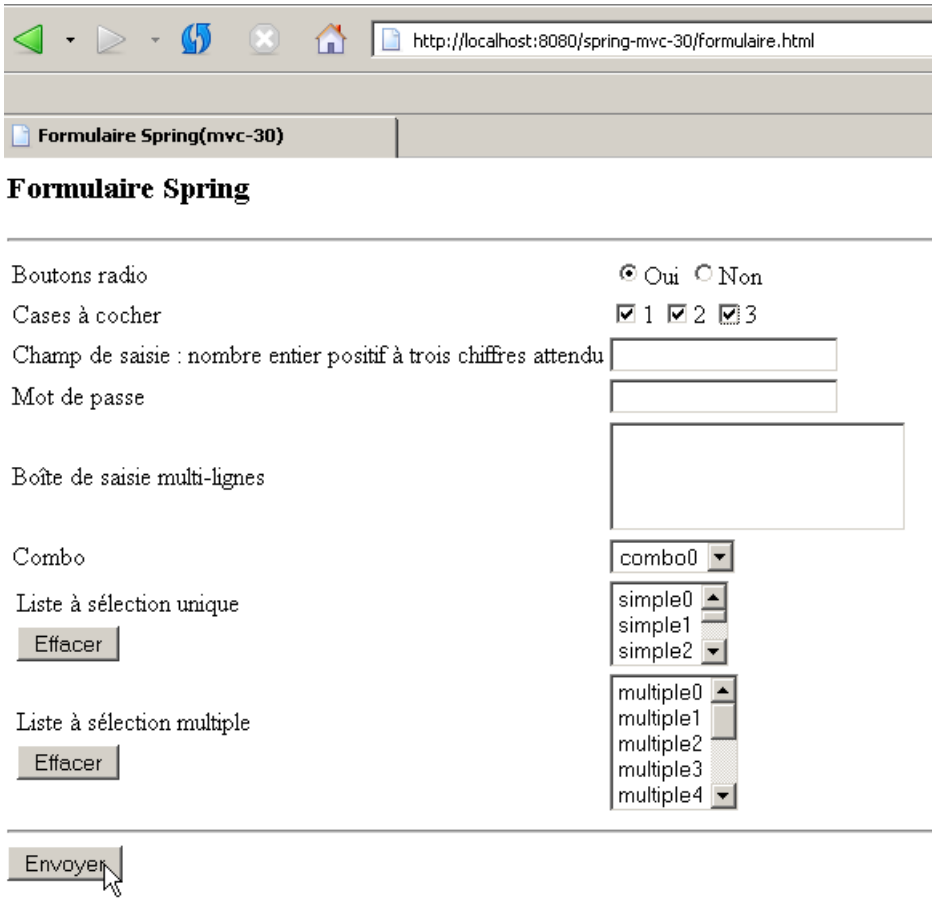
Le lecteur est invité à faire des tests. Pour l'utilisateur, rien ne distingue cette application de la précédente si ce n'est que de temps en temps, il voit apparaître la page d'exception que lance de façon aléatoire la couche métier (uniquement sur un GET) :

5.10 Formulaire avec contrôles de validité

Nous reprenons la même application que précédemment mais nous souhaitons que certaines saisies soient contrôlées. Le formulaire reçu initialement sera le suivant :

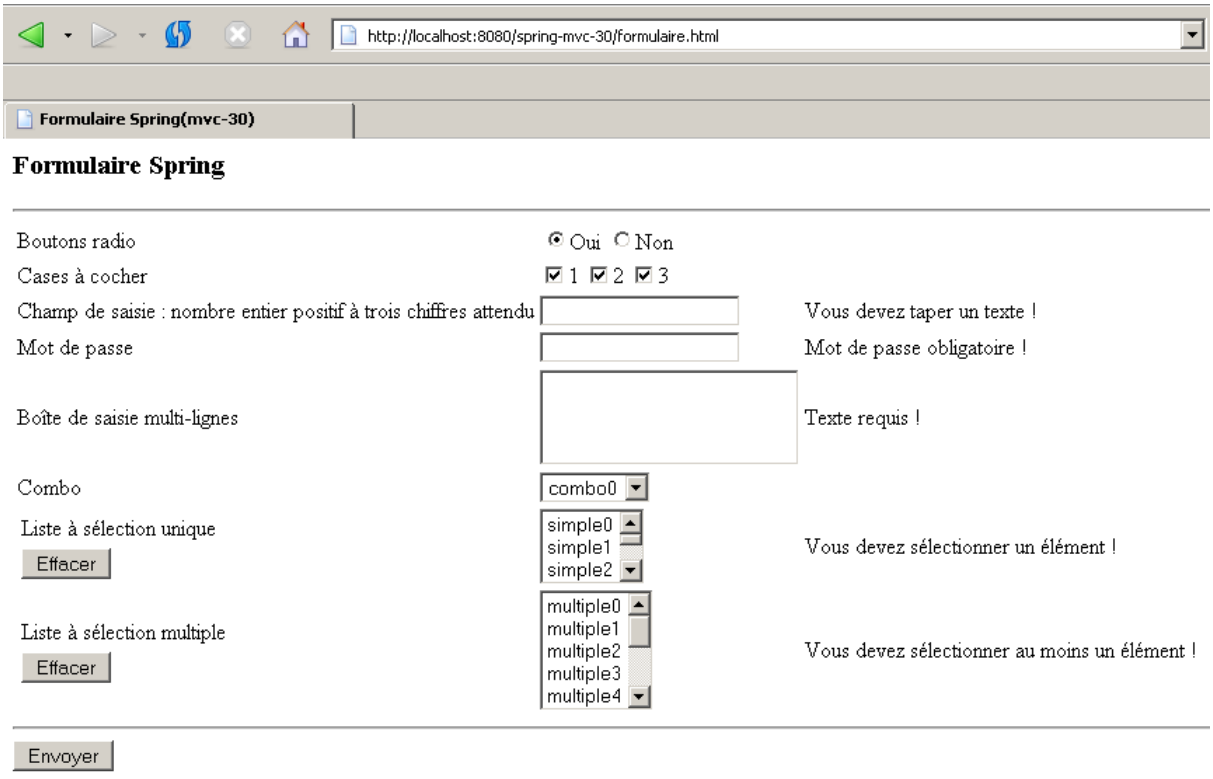


L'utilisateur fait les saisies suivantes :



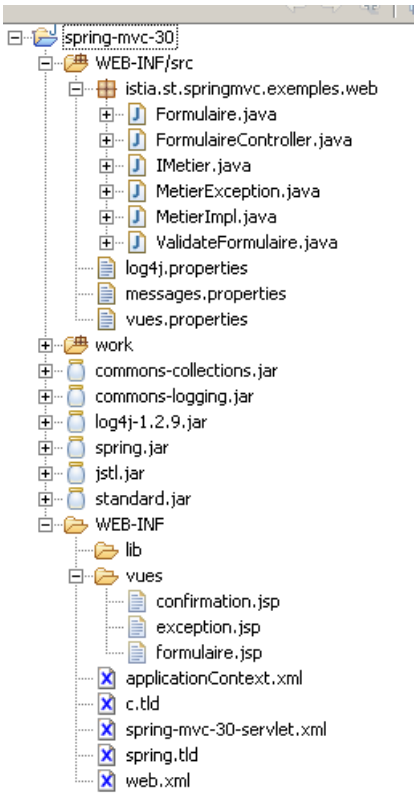
- le champ de saisie est vide
- le mot de passe est vide
- la boîte de saisie multi-lignes est vide
- aucun élément n'est sélectionné dans les listes à sélection simple et multiple

L'utilisateur reçoit alors la réponse suivante :



Il a reçu la page telle qu'il l'a initialement postée (voir les cases à cocher par exemple) avec de plus des messages d'erreurs indiquant que certains champs n'ont pas été remplis correctement. Spring offre des facilités pour faire ces contrôles de validité et c'est ce que nous allons voir maintenant.

Le projet Eclipse de cet exemple est quasi identique au précédent :



Les nouveaux fichiers sont tous liés aux contrôles de validité que nous voulons ajouter.

Commençons par regarder la configuration [applicationContext.xml] du contexte de l'application :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <!-- couche métier -->
5.    <bean id="metier" class="istia.st.springmvc.exemples.web.MetierImpl"/>
6.    <!-- le fichier des messages -->
7.    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
8.      <property name="basename">
9.        <value>messages</value>
10.     </property>
11.   </bean>
12. </beans>
```

- ligne 5 : notre couche [métier] comme précédemment
- lignes 7-10 : une implémentation de l'interface [MessageSource]. Cette interface est destinée à gérer les messages utilisés par l'application et leur internationalisation. Dans notre cas, ce sont les messages d'erreurs affichés lorsqu'une valeur postée est invalide qui seront gérés par cette interface. Spring MVC fournit deux implémentations de l'interface [MessageSource] :
    - **ResourceBundleMessageSource** : cette classe lit, au démarrage de l'application, le ou les fichiers de définition des messages. Ils ne sont plus relus par la suite.
    - **ReloadableResourceBundleMessageSource** : cette classe fait la même chose que la classe précédente mais il est possible de relire les fichiers de définition des messages au cours de l'exécution de l'application. Cela peut permettre par exemple de générer dynamiquement des fichiers de messages en cours d'exécution.

L'implémentation de l'interface [MessageSource] utilisée ligne 7 de [applicationContext.xml] est **ResourceBundleMessageSource** :

org.springframework.context.support

Class ResourceBundleMessageSource

java.lang.Object
  └ org.springframework.context.support.AbstractMessageSource
    └ **org.springframework.context.support.ResourceBundleMessageSource**

All Implemented Interfaces:

HierarchicalMessageSource, MessageSource

Cette classe suppose que le ou les fichiers des messages sont dans le *ClassPath* de l'application. Elle possède deux méthodes de type *setter* pour fixer le ou les noms de ces fichiers :

void	**setBasename**(String basename) 1 Set a single basename, following ResourceBundle conventions: It is a fully-qualified classname.
void	**setBasenames**(String[] basenames) 2 Set an array of basenames, each following ResourceBundle conventions.

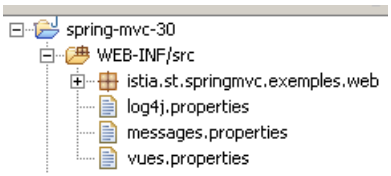
1. la méthode [1] permet de donner le nom du fichier contenant les messages. Ce fichier est cherché dans le *ClassPath* de l'application. S'il est dans un paquetage, il doit être préfixé par le nom du paquetage.
2. la méthode [2] permet de donner dans un tableau les noms des fichiers contenant les messages

Revenons sur la configuration de [ResourceBundleMessageSource] faite dans [applicationContext.xml] :

```
1.  <!-- le fichier des messages -->
2.  <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
3.    <property name="basename">
4.      <value>messages</value>
5.    </property>
6.  </bean>
```

- lignes 3-5 : configuration de la propriété [basename]. Le setter [setBasename] ci-dessus va être appelé.
- ligne 4 : un fichier appelé [**messages.properties**] placé dans le ClassPath contient les messages.

Dans notre application, le fichier [**messages.properties**] est à la racine [WEB-INF/src] :



On sait que dans un projet Eclipse / Tomcat, tous les fichiers de [WEB-INF/src] sont recopiés tels quels dans [WEB-INF/classes] si ce ne sont pas des fichiers source Java. Ainsi en sera-t-il de [**messages.properties**] qui, du coup, se retrouve bien dans le *ClassPath* de l'application.

Le fichier [**messages.properties**] contient des lignes de la forme

```
code=message
```

Notre fichier [**messages.properties**] sera le suivant :

```
1.  # messages de l'application
2.  formulaire.champSaisie.necessaire=Vous devez taper un texte !
3.  formulaire.mdp.necessaire=Mot de passe obligatoire !
4.  formulaire.mdp.tropcourt=Le mot de passe doit avoir au moins 8 caractères !
5.  formulaire.boiteSaisie.necessaire=Texte requis !
6.  formulaire.listeSimple.necessaire=Vous devez sélectionner un élément !
7.  formulaire.listeMultiple.necessaire=Vous devez sélectionner au moins un élément !
8.  formulaire.champSaisie.incorrect=Tapez un entier positif de 3 chiffres !
```

Lorsqu'une application veut écrire un message dans une vue, elle n'indique que le code de celui-ci, par exemple "**formulaire.champSaisie.necessaire**". Ce n'est qu'au moment du rendu de la vue, que le code se transforme en texte "**Vous devez taper un texte !**". Cela comporte divers avantages :

- on peut changer le texte du message sans entrer dans le code Java
- on peut internationaliser le message

Nous avons parlé à diverses reprises de l'internationalisation des vues (paragraphe 3, page 11). Nous avons vu que le fichier [**vues.properties**] pouvait exister en diverses variantes :

- **vues_en.properties** : pour définir les vues en anglais
- **vues_de_AT** : pour définir les vues en allemand d'Autriche
- ...

Le fichier [**messages.properties**] pourra être lui aussi décliné en variantes : **messages_en.properties, messages_de_AT.properties**, ...

Ainsi dans un fichier [**messages_en.properties]** situé au même endroit que [**messages.properties**], le code "**formulaire.champSaisie.necessaire**" pourra être associé à un message en anglais cette fois :

```
1. # messages de l'application en anglais
2. formulaire.champSaisie.necessaire=You must enter a text !
3. ...
```

Les messages peuvent être paramétrés. Ainsi pourrait-on avoir le message suivant dans le fichier [messages.properties] :

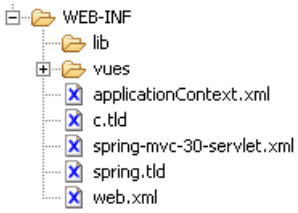
```
formulaire.champSaisie.incorrect=La saisie [{0}] n'est pas un entier positif de 3 chiffres !
```

Lorsque l'application veut afficher dans une vue le message associé au code [formulaire.champSaisie.incorrect], elle devra de plus fournir un argument pour le paramètre {0} du message. Ainsi le message réellement affiché pourra devenir :

```
formulaire.champSaisie.incorrect=La saisie [xx] n'est pas un entier positif de 3 chiffres !
```

On peut utiliser jusqu'à 9 paramètres {0}..{9} dans un message. Nous verrons ultérieurement à quels moments notre application va utiliser le fichier de messages [messages.properties].

Revenons à la configuration de l'application [spring-mvc-30] :



Nous venons d'expliquer le contenu de [applicationContext.xml]. Le fichier de configuration [spring-mvc-30-servlet.xml] de la couche [web] du projet est lui le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les mappings de l'application-->
5.    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
6.      <property name="mappings">
7.        <props>
8.          <prop key="formulaire.html">FormulaireController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="FormulaireController"
14.     class="istia.st.springmvc.exemples.web.FormulaireController">
15.     <property name="sessionForm">
16.       <value>true</value>
17.     </property>
18.     <property name="commandName">
19.       <value>formulaire</value>
20.     </property>
21.     <property name="formView">
22.       <value>formulaire</value>
23.     </property>
24.     <property name="validator">
25.       <ref bean="Formulaire.Validator"/>
26.     </property>
27.     <property name="metier">
```

```
28.       <ref bean="metier"/>
29.     </property>
30.   </bean>
31.   <!-- le validateur de formulaire -->
32.   <bean id="Formulaire.Validator"
33.     class="istia.st.springmvc.exemples.web.ValidateFormulaire"/>
34.   <!-- le résolveur de vues -->
35.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
36.     <property name="basename">
37.       <value>vues</value>
38.     </property>
39.   </bean>
40.   <!-- le gestionnaire d'exceptions -->
41.   <bean
42.     class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
43.     <property name="exceptionAttribute">
44.       <value>exception</value>
45.     </property>
46.     <property name="defaultStatusCode">
47.       <value>500</value>
48.     </property>
49.     <property name="defaultErrorView">
50.       <value>exception</value>
51.     </property>
52.   </bean>
53. </beans>
```

Nous ne revenons pas sur ce qui n'a pas changé vis à vis de l'application précédente.

Un premier changement se trouve dans la configuration du contrôleur [FormulaireController]. Lignes 24-26, une propriété [validator] est définie. Comme il ne s'agit pas d'une propriété de [FormulaireController], il s'agit donc d'une propriété d'une classe parent, très exactement la classe [BaseCommandController] :

setValidator

```
public final void setValidator(Validator validator)
```

Set the primary Validator for this controller. The Validator must support the specified command class. If there are one or more existing validators set already when this method is called, only the specified validator will be kept. Use `setValidators(Validator[])` to set multiple validators.

La méthode [**setValidator**] attend comme unique paramètre un objet de type [**Validator**]. [Validator] est une interface. Les lignes 32-33 ci-dessus indiquent que l'implémentation utilisée pour cette interface est [**ValidateFormulaire**], une classe que nous allons construire. Avant de détailler celle-ci, ils nous faut comprendre à quoi elle sert. Pour cela, il nous faut revenir au flux d'exécution du POST décrit page 27 :

1. le contrôleur [SimpleFormController] reçoit une requête de type **POST**
2. si l'attribut " **sessionForm** " de [SimpleFormController] est à *false*, la méthode [**formBackingObject**] de la classe de base [AbstractFormController] est appelée pour créer l'objet **command** qui va servir à enregistrer les données du POST. Si " sessionForm " est à *true*, alors l'objet **command** en question est récupéré dans la session (cf étape 2 du GET).
3. les valeurs postées sont affectées aux champs de même nom de l'objet **command**. Si d'éventuels éditeurs de propriétés avaient été définis à l'étape 3 du GET, ils sont utilisés pour les conversions String -> Type, par exemple java.lang.String -> java.util.Date.
4. la méthode **onBind(**HttpServletRequest request, Object command, BindException Errors**)** est appelée. Elle ne fait rien par défaut. Elle peut être redéfinie pour faire par exemple les conversions String -> Type à la main.
5. si l'attribut **validateOnBinding** de [SimpleFormController] a la valeur *true*, un objet de type [Validator] sera utilisé pour vérifier la validité des données.
6. la méthode **onBindAndValidate**(HttpServletRequest request, Object command, BindException Errors) est appelée. Elle ne fait rien par défaut. Elle peut être redéfinie pour faire par exemple les validations sans objet [Validator].
7. la méthode **processFormSubmission(**HttpServletRequest request, HttpResponse response, Object command, BindException Errors**)** est appelée. Elle vérifie la liste des erreurs contenu dans l'objet [**Errors**] mis à jour par les méthodes 4, 5 et 6 précédentes. Si cette liste d'erreurs est non vide, la méthode [showForm] est appelée. Comme pour le GET, elle rend un objet [ModelAndView] où le nom de la vue est celui défini par l'attribut " **formView** " dans la configuration du contrôleur. Le modèle lui, comprend l'objet **command** initialisé par le POST à l'étape 3 précédente ainsi que l'objet [**Errors**]. L'utilisateur récupère donc le formulaire tel qu'il l'a posté avec de plus les erreurs interceptées par le serveur. Cette méthode n'est normalement pas redéfinie.
8. lorsqu'il n'y a pas d'erreurs, la méthode **processFormSubmission** appelle la méthode **ModelAndView onSubmit(**HttpRequest request, HttpResponse response, Object command, BindException Errors**)** ...

Aux étapes 4, 5 et 6 diverses méthodes sont appelées qui ont toutes pour rôle de vérifier la validité des valeurs postées. Si elles ne sont pas redéfinies, aucune vérification de validité n'est faite par défaut. Associer un validateur à un contrôleur, comme la configuration de [FormulaireController] le fait, revient à implémenter l'étape 5 ci-dessus. Les valeurs postées vont être examinées

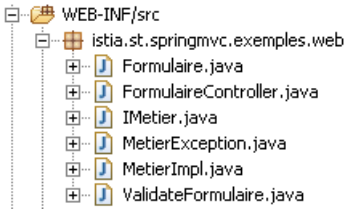
par l'objet [Validator] associé au contrôleur. Si certaines sont déclarées invalides, des messages d'erreur vont être ajoutés à l'objet [BindException Errors], paramètre commun à ces différentes méthodes. A l'étape 7, la méthode [processFormSubmission] de la classe [SimpleFormController] va vérifier cette liste d'erreurs. Si elle est non vide, la méthode va rendre la main à [DispatcherServlet] en lui demandant de réafficher l'objet **command** qui a été posté. L'objet [Errors] fera partie du modèle affiché, permettant ainsi l'affichage des messages d'erreurs. Dans ce cas, l'étape 8 n'est pas faite.

Revenons sur la configuration du contrôleur [FormulaireController] :

```
1.   <!-- les contrôleurs de l'application-->
2.   <bean id="FormulaireController"
3.     class="istia.st.springmvc.exemples.web.FormulaireController">
4.  ...
5.     <property name="validator">
6.       <ref bean="Formulaire.Validator"/>
7.     </property>
8.  ...
9.   </bean>
10.  <!-- le validateur de formulaire -->
11.  <bean id="Formulaire.Validator"
12.    class="istia.st.springmvc.exemples.web.ValidateFormulaire"/>
```

- lignes 5-7 : un validateur est associé au contrôleur
- lignes 11-12 : ce validateur est une instance d'une classe propriétaire [ValidateFormulaire]

La classe [ValidateFormulaire] est dans [WEB-INF/src] avec les autres classes :



La classe [ValidateFormulaire] doit implémenter l'interface [org.springframework.validation.Validator] qui a deux méthodes :

boolean	**supports**(Class clazz) Return whether or not this object can validate objects of the given class. 1
void	**validate**(Object obj, Errors errors) 2 Validate an object, which must be of a class for which the supports() method returned true.

1. la méthode **supports** [1] sert à indiquer quel type d'objet le validateur sait valider. Elle sera appelée avec, comme paramètre, la classe de l'objet **command** initialisé par le POST.
2. la méthode **validate** [2] reçoit l'objet à valider donc l'objet **command** précédent. Il contient les valeurs postées qui vont donc pouvoir être examinées. La méthode **validate** dispose de l'objet [**Errors errors**] grâce auquel elle pourra enregistrer des messages d'erreurs.

[**Errors**] est une interface.

org.springframework.validation

Interface Errors

All Known Implementing Classes:

BindException, EscapedErrors

Pour signaler une erreur, la méthode [validate] du validateur, utilisera l'une des méthodes [reject] de l'interface :

void	**reject**(String errorCode) Reject the current object, using the given error description.
void	**reject**(String errorCode, Object[] errorArgs, String defaultMessage) 2 Reject the current object, using the given error description.
void	**reject**(String errorCode, String defaultMessage) Reject the current object, using the given error description.
void	**rejectValue**(String field, String errorCode) 1 Reject the given field of the current object, using the given error description.
void	**rejectValue**(String field, String errorCode, Object[] errorArgs, String defaultMessage) Reject the given field of the current object, using the given error description. 3
void	**rejectValue**(String field, String errorCode, String defaultMessage) Reject the given field of the current object, using the given error description.

Pour indiquer qu'un champ **C** de l'objet **command** posté est incorrect, nous utiliserons la méthode [1] ci-dessus :

- le paramètre **field** est le nom **C** du champ erroné
- **errorCode** est le code du message d'erreur à utiliser pour signaler cette erreur. Dans notre application, ce code sera l'un des codes définis dans [**messages.properties**].

Nous avons dit qu'un message de [[**messages.properties**] pouvait être paramétré, comme par exemple :

```
formulaire.champSaisie.incorrect=La saisie [{0}] n'est pas un entier positif de 3 chiffres !
```

Les méthodes **reject** [2] et [3] de l'interface [**Errors**], qui ont le paramètre [**Object[] errorArgs**] permettent de passer des arguments au message désigné par le paramètre **errorCode**. L'élément n° **i** du tableau **errorArgs** est référencé par la notation **{i}** dans le message.

Revenons à notre application pour en rappeler certains points :

- l'objet examiné par le validateur est de type [Formulaire]

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Formulaire {
4.
5.    // boutons radio
6.    private String opt;
7.
8.    // cases à cocher
9.    private String chk1;
10.   private String chk2;
11.   private String chk3;
12.
13.   // champs de saisie
14.   private String champSaisie;
15.   private String mdp;
16.   private String boiteSaisie;
17.
18.   // liste déroulante
19.   private String combo;
20.
21.   // liste à sélection simple
22.   private String listeSimple;
23.
24.   // liste à sélection multiple
25.   private String[] listeMultiple;
26.
27.   // champ caché
28.   private String secret;
29.
30.   // getters et setters
31. ...
32. }
```

- les champs à examiner par le validateur [ValidateFormulaire] seront les suivants :
    - **champSaisie** doit être un nombre à trois chiffres
    - **mdp, boiteSaisie** et **listeSimple** doivent être initialisés
    - **listeMultiple** doit être un tableau non vide

Nous pouvons maintenant examiner le code du validateur [**ValidateFormulaire**] :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.regex.Pattern;
4.
5.  import org.springframework.validation.Errors;
6.
7.  public class ValidateFormulaire implements
8.     org.springframework.validation.Validator {
9.
10.  public boolean supports(Class classe) {
11.    return classe.isAssignableFrom(Formulaire.class);
12.  }
13.
14.  public void validate(Object obj, Errors erreurs) {
15.    // on récupère le formulaire posté
16.    Formulaire formulaire = (Formulaire) obj;
17.    // on vérifie le champ de saisie
18.    String champSaisie = formulaire.getChampSaisie();
19.    if (champSaisie == null || champSaisie.trim().length() == 0) {
20.      erreurs.rejectValue("champSaisie",
21.          "formulaire.champSaisie.necessaire");
22.    }
23.    // on veut vérifier qu'on a un entier positif
24.    if(! Pattern.matches("\\s*\\d{3}\\s*",champSaisie)){
25.      erreurs.rejectValue("champSaisie",
26.      "formulaire.champSaisie.incorrect");
27.    }
28.    // on vérifie le mot de passe
29.    String mdp = formulaire.getMdp();
30.    if (mdp == null || mdp.trim().length() == 0) {
31.      erreurs.rejectValue("mdp", "formulaire.mdp.necessaire");
32.    } else {
33.      mdp = mdp.trim();
34.      if (mdp.length() < 8) {
35.        erreurs.rejectValue("mdp", "formulaire.mdp.tropcourt");
36.      }
37.    }
38.    // on vérifie le champ de saisie multi-lignes
39.    String boiteSaisie = formulaire.getBoiteSaisie();
40.    if (boiteSaisie == null || boiteSaisie.trim().length() == 0) {
41.      erreurs.rejectValue("boiteSaisie",
42.          "formulaire.boiteSaisie.necessaire");
43.    }
44.    // on vérifie la liste à sélection unique
45.    String selectionListeSimple = formulaire.getListeSimple();
46.    if (selectionListeSimple == null) {
47.      erreurs.rejectValue("listeSimple",
48.          "formulaire.listeSimple.necessaire");
49.    }
50.    // on vérifie la liste à sélection multiple
51.    String[] selectionListeMultiple = formulaire.getListeMultiple();
52.    if (selectionListeMultiple == null) {
53.      erreurs.rejectValue("listeMultiple",
54.          "formulaire.listeMultiple.necessaire");
55.    }
56.  }
57.
58. }
```

- lignes 7-8 : la classe [ValidateFormulaire] implémente l'interface [Validator] et donc la méthode [**supports**] (lignes 10-12) et la méthode [**validate**] (lignes 14-56).
- lignes 10-12 : la la méthode [**supports**] doit rendre *true* si l'objet [Class classe] reçu en paramètre peut être validé par la méthode [validate]. La méthode [validate] traitant les champs d'un objet de type [Formulaire], on rend *true* si l'objet [Class classe] reçu en paramètre est de type [Formulaire] ou dérivé.
- ligne 14 : la méthode [validate] reçoit en premier paramètre l'objet [Object obj] à valider. Pour nous ce sera un objet de type [Formulaire] ou dérivé (cf méthode **supports**). Elle reçoit en second paramètre, l'objet [Errors erreurs] dans lequel, elle va pouvoir stocker ses messages d'erreurs éventuels.
- ligne 16 : on dit que l'objet reçu est de type [Formulaire]
- ligne 18 : on récupère la valeur du champ **champSaisie** de [Formulaire]
- ligne 19 : on vérifie qu'il n'est ni **null** ni réductible à une chaîne vide
- lignes 20-21 : en cas d'erreur, un message d'erreur est enregistré dans le second paramètre [Errors erreurs]. On utilise la méthode [rejectValue] de l'interface [Validator] qui admet en premier paramètre le nom du champ erroné, ici " **champSaisie** " et en second paramètre le code du message de [**messages.properties**] qu'il faudra utiliser pour cette erreur. Ceci sera répété pour chaque champ erroné de [Formulaire]. On voit qu'au final, l'objet [Errors erreurs] va contenir une liste de couples (champ, code d'erreur). En cas d'erreurs, l'objet [Errors erreurs] est mis dans le modèle de la vue envoyée au client. Nous verrons ultérieurement comment exploiter cet élément du modèle.
- lignes 23-27 : on vérifie que **champSaisie** est un nombre à trois chiffres

- lignes 29-37 : on vérifie que **mdp** est une chaîne d'au moins huit caractères
- lignes 39-43 : on vérifie que **boiteSaisie** a une valeur non vide
- lignes 45-49 : on vérifie que **listeSimple** a une valeur, la valeur **null** signifiant qu'aucun élément n'a été sélectionné dans la liste à sélection unique du formulaire HTML.
- lignes 51-55 : on vérifie que **listeMultiple** a une valeur, la valeur **null** signifiant qu'aucun élément n'a été sélectionné dans la liste à sélection multiple du formulaire HTML.

Si le validateur [ValidateFormulaire] détecte des erreurs, la vue " formulaire " va être affichée avec dans son modèle les éléments suivants :

- l'instance [Formulaire] issue du POST et déclarée invalide par le validateur
- les différents tableaux nécessaires au remplissage des listes HTML
- l'objet [Errors erreurs] construit par le validateur [ValidateFormulaire]

On sait que la vue nommée " formulaire " est associée à la page JSP [formulaire.jsp]. Se pose alors la question : comment [formulaire.jsp] va-t-il avoir accès aux messages d'erreurs créés par le validateur ? C'est ce que nous voyons maintenant.

Le code de la page JSP [formulaire.jsp] est le suivant :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ taglib uri="/WEB-INF/spring.tld" prefix="spring" %>
4.  <%@ page isELIgnored="false" %>
5.  <html>
6.    <head>
7.      <title>Formulaire Spring(mvc-30)</title>
8.    </head>
9.    <body>
10.     <h3>Formulaire Spring</h3>
11.     <hr>
12.     <form method="post">
13.       <table border="0">
14.         <!-- boutons radio -->
15.         <tr>
16.           <td>Boutons radio</td>
17.           <td>
18.             <c:choose>
19.               <c:when test='${formulaire.opt=="oui"}'>
20.                 <input type="radio" name="opt" value="oui" checked>Oui
21.                 <input type="radio" name="opt" value="non">Non
22.               </c:when>
23.               <c:otherwise>
24.                 <input type="radio" name="opt" value="oui">Oui
25.                 <input type="radio" name="opt" value="non" checked>Non
26.               </c:otherwise>
27.             </c:choose>
28.           </td>
29.         </tr>
30.         <!-- cases à cocher -->
31.         <tr>
32.           <td>Cases &agrave; cocher</td>
33.           <td>
34.             <c:choose>
35.               <c:when test='${formulaire.chk1=="un"}'>
36.                 <input type="checkbox" name="chk1" value="un" checked>1
37.               </c:when>
38.               <c:otherwise>
39.                   <input type="checkbox" name="chk1" value="un">1
40.               </c:otherwise>
41.             </c:choose>
42.             <c:choose>
43.               <c:when test='${formulaire.chk2=="deux"}'>
44.                 <input type="checkbox" name="chk2" value="deux" checked>2
45.               </c:when>
46.               <c:otherwise>
47.                   <input type="checkbox" name="chk2" value="deux">2
48.               </c:otherwise>
49.             </c:choose>
50.             <c:choose>
51.               <c:when test='${formulaire.chk3=="trois"}'>
52.                 <input type="checkbox" name="chk3" value="trois" checked>3
53.               </c:when>
54.               <c:otherwise>
55.                   <input type="checkbox" name="chk3" value="trois">3
56.               </c:otherwise>
57.             </c:choose>
58.           </td>
59.         </tr>
60.         <!-- champ de saisie -->
61.         <tr>
62.           <td>Champ de saisie : nombre entier positif &agrave; trois chiffres attendu</td>
63.             <spring:bind path="formulaire.champSaisie">
64.               <td>
65.                   <input type="text" name="${status.expression}" value="${status.value}">
66.               </td>
67.               <td>${status.errorMessage}</td>
68.             </spring:bind>
69.         </tr>
```

```
70.         <!-- mot de passe -->
71.         <tr>
72.           <td>Mot de passe</td>
73.             <spring:bind path="formulaire.mdp">
74.               <td>
75.                   <input type="password" name="${status.expression}" value="${status.value}">
76.               </td>
77.               <td>${status.errorMessage}</td>
78.             </spring:bind>
79.         </tr>
80.         <!-- Boîte de saisie multi-lignes -->
81.         <tr>
82.           <td>Bo&icirc;te de saisie multi-lignes</td>
83.             <spring:bind path="formulaire.boiteSaisie">
84.               <td>
85.                 <textarea name="${status.expression}" rows="3">${status.value}</textarea>
86.               </td>
87.               <td>${status.errorMessage}</td>
88.             </spring:bind>
89.         </tr>
90.         <!-- Combo -->
91.         <tr>
92.           <td>Combo</td>
93.           <td>
94.           <select name="combo" rows="3">
95.             <c:forEach items="${optionsCombo}" var="optionCombo">
96.               <c:choose>
97.                 <c:when test="${formulaire.combo==optionCombo}">
98.                   <option selected>${optionCombo}</option>
99.                 </c:when>
100.                <c:otherwise>
101.                  <option>${optionCombo}</option>
102.                </c:otherwise>
103.              </c:choose>
104.            </c:forEach>
105.          </select>
106.          </td>
107.        </tr>
108.        <!-- Liste à sélection unique -->
109.        <tr>
110.          <td>
111.            <table>
112.              <tr>
113.                <td>Liste &agrave; s&eacute;lection unique</td>
114.              </tr>
115.              <tr>
116.                <td>
117.                  <input type="button" value="Effacer" onclick="this.form.listeSimple.selectedIndex=-1"/>
118.                </td>
119.              </tr>
120.            </table>
121.          </td>
122.            <spring:bind path="formulaire.listeSimple">
123.              <td>
124.                <select name="${status.expression}" size="3">
125.                  <c:forEach items="${optionsListeSimple}" var="optionListeSimple">
126.                    <c:choose>
127.                      <c:when test="${status.value==optionListeSimple}">
128.                        <option selected>${optionListeSimple}</option>
129.                      </c:when>
130.                      <c:otherwise>
131.                        <option>${optionListeSimple}</option>
132.                      </c:otherwise>
133.                    </c:choose>
134.                  </c:forEach>
135.                </select>
136.              </td>
137.              <td>${status.errorMessage}</td>
138.            </spring:bind>
139.        </tr>
140.        <!-- Liste à sélection multiple -->
141.        <tr>
142.          <td>
143.            <table>
144.              <tr>
145.                <td>Liste &agrave; s&eacute;lection multiple</td>
146.              </tr>
147.              <tr>
148.                <td>
149.                  <input type="button" value="Effacer" onclick="this.form.listeMultiple.selectedIndex=-1"/>

150.                </td>
151.              </tr>
152.            </table>
153.          </td>
154.            <spring:bind path="formulaire.listeMultiple">
155.              <td>
156.                <select name="${status.expression}" size="5" multiple>
157.                  <c:forEach items="${optionsListeMultiple}" var="optionListeMultiple">
158.                    <c:set var="selection" value="faux"/>
159.                    <c:forEach items="${status.value}" var="selectionListeMultiple">
160.                      <c:if test="${selectionListeMultiple==optionListeMultiple}">
161.                        <c:set var="selection" value="vrai"/>
162.                      </c:if>
163.                    </c:forEach>
164.                    <c:choose>
165.                      <c:when test='${selection=="vrai"}'>
```

```
166.                       <option selected>${optionListeMultiple}</option>
167.                     </c:when>
168.                     <c:otherwise>
169.                       <option>${optionListeMultiple}</option>
170.                     </c:otherwise>
171.                   </c:choose>
172.                 </c:forEach>
173.               </select>
174.             </td>
175.             <td>${status.errorMessage}</td>
176.           </spring:bind>
177.         </tr>
178.       </table>
179.       <input type="hidden" name="secret" value="${formulaire.secret}">
180.       <hr>
181.       <input type="hidden" name="_chk1">
182.       <input type="hidden" name="_chk2">
183.       <input type="hidden" name="_chk3">
184.       <input type="hidden" name="_listeSimple">
185.       <input type="hidden" name="_listeMultiple">
186.       <input type="submit" value="Envoyer">
187.     </form>
188.   </body>
189. </html>
```

Dans le code, on voit apparaître des balises inconnues. Par exemple pour le champ de saisie :

```
1.   <!-- champ de saisie -->
2.    <tr>
3.      <td>Champ de saisie : nombre entier positif &agrave; trois chiffres attendu</td>
4.      <spring:bind path="formulaire.champSaisie">
5.        <td>
6.          <input type="text" name="${status.expression}" value="${status.value}">
7.        </td>
8.        <td>${status.errorMessage}</td>
9.      </spring:bind>
10.   </tr>
```

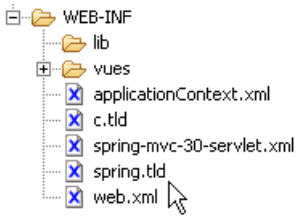
Ligne 4, une balise **<spring:bind>** est ouverte. Elle est fermée en ligne 9.

Ces balises fournies par Spring vont nous permettre d'avoir accès aux messages d'erreurs des champs vérifiés par le validateur HTML. Les balises Spring ne sont pas utilisées pour les champs dont la valeur n'est pas vérifiée.

Cette bibliothèque de balises est déclarée ligne 3 de [formulaire.jsp] :

```
<%@ taglib uri="/WEB-INF/spring.tld" prefix="spring" %>
```

Le fichier [spring.tld] de définition de la bibliothèque de balises a été placé dans [WEB-INF] :



Nous allons expliquer le rôle de la balise **<spring:bind>** sur l'exemple du champ HTML **champSaisie** :

```
1.   <!-- champ de saisie -->
2.    <tr>
3.      <td>Champ de saisie : nombre entier positif &agrave; trois chiffres attendu</td>
4.      <spring:bind path="formulaire.champSaisie">
5.        <td>
6.          <input type="text" name="${status.expression}" value="${status.value}">
7.        </td>
8.        <td>${status.errorMessage}</td>
9.      </spring:bind>
10.   </tr>
```

Ligne 4, l'attribut **path** de <spring:bind> lie la balise à un champ particulier de l'objet [Formulaire] issu du GET ou du POST. Ici c'est le champ [**champSaisie**]. A l'intérieur de la balise, un objet de clé **status** est disponible. Cet objet possède diverses propriété utiles :

- **status.expression** : le nom du champ lié à la balise <spring:bind>, ici **champSaisie**
- **status.value** : la valeur postée pour le champ lié à la balise <spring:bind>
- **status.errorMessage** : le message d'erreur lié au champ, s'il y en a un

Prenons un exemple. Supposons que l'utilisateur ait saisi le texte " xx " dans le champ HTML **champSaisie**. A l'issue du POST, l'instance [Formulaire] a donc la valeur " xx " dans son champ **champSaisie**. Si on suit le code de la méthode [validate] du validateur, on voit que celle-ci va mettre dans l'objet [Errors erreurs] un couple ("champSaisie","formulaire.champSaisie.incorrect") :

```
    erreurs.rejectValue("champSaisie","formulaire.champSaisie.incorrect");
```

D'après [**messages.properties**] :

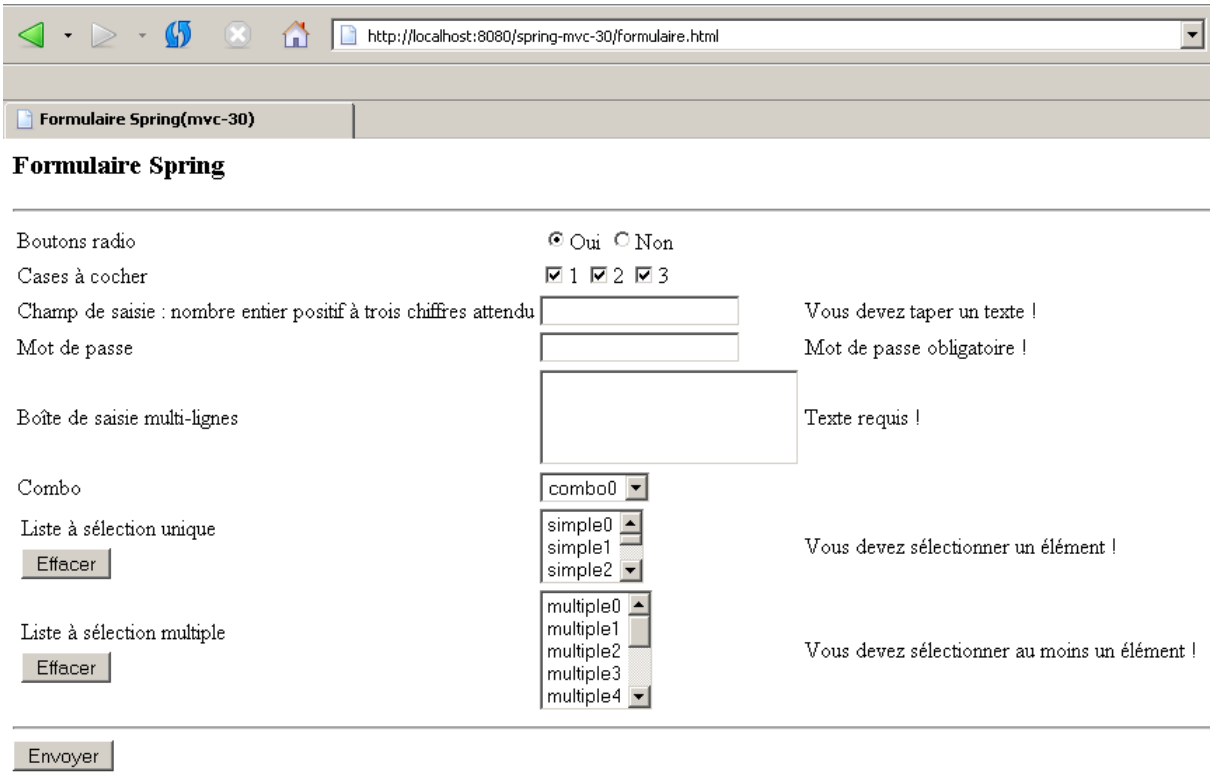
```
formulaire.champSaisie.incorrect=Tapez un entier positif de 3 chiffres !
```

le code "formulaire.champSaisie.incorrect" est associé au message "Tapez un entier positif de 3 chiffres !". Dans la balise <spring:bind> de [champSaisie], on aura donc :

- **status.expression** : *champSaisie*
- **status.value** : *xx*
- **status.errorMessage** : *Tapez un entier positif de 3 chiffres !*

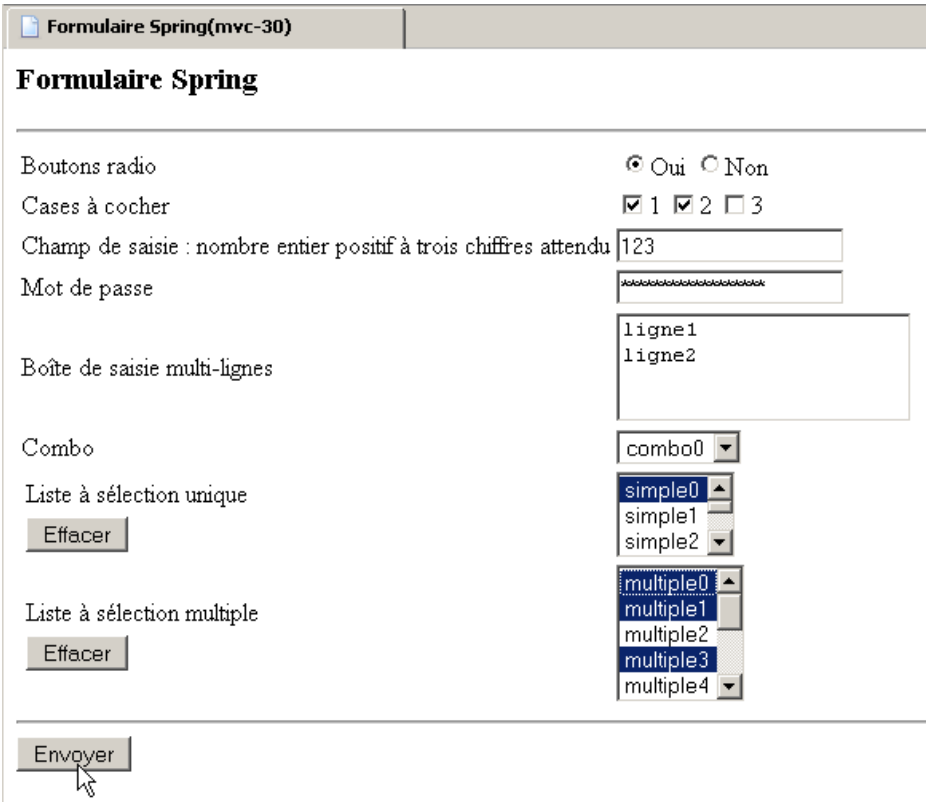
La zone de saisie HTML [champSaisie] sera donc affichée avec d'une part sa valeur postée **xx** et d'autre part le message d'erreur **Tapez un entier positif de 3 chiffres !**.

Le même raisonnement peut être fait avec tous les autres champs dont la validité est vérifiée. Nous laissons au lecteur le soin de cette vérification. Rappelons un exemple de vue affichée après erreurs de valeurs postées :



Lorsqu'il n'y a pas d'erreurs de saisie, cette application a le même fonctionnement que la précédente. La page [confirmation.jsp] reste identique à celle de la version précédente.

Voici un exemple :



La page de confirmation :

5.11 Conversions de types de données

Dans tous les exemples précédents, notre application fonctionnait de la façon suivante :

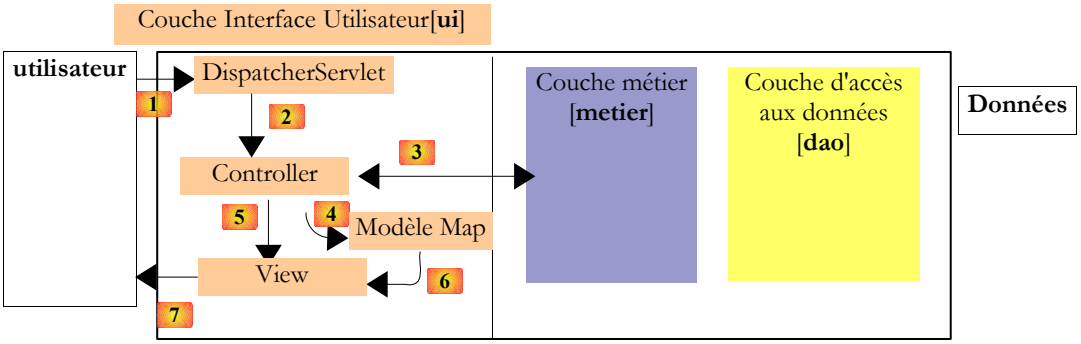
1. l'utilisateur faisait un GET /formulaire.html
2. un objet [Formulaire] était instancié et placé dans le modèle de la page JSP [formulaire.jsp]
3. l'utilisateur faisait des saisies puis les postait par un POST /formulaire.html
4. l'objet [Formulaire] créé à l'étape 2 était récupéré dans la session et les valeurs postées lui étaient injectées
5. l'objet [Formulaire] ainsi mis à jour était éventuellement vérifié par un validateur
6. selon les cas, une page de confirmation [confirmation.jsp] ou la page JSP [formulaire.jsp] initiale complétée de messages d'erreurs était envoyée à l'utilisateur

Le problème de la conversion de données se pose aux étapes 2 et 4 :

- à l'étape 2, les valeurs des champs de [Formulaire] sont utilisées pour initialiser les champs HTML. Ces derniers sont de type chaîne de caractères. Si un champ de [Formulaire] n'est pas de type **String** (de type java.util.Date par exemple), il y a une conversion de données à faire.
- à l'étape 4, les valeurs postées sont de type chaîne de caractères. Si une valeur postée doit être injectée dans un champ qui n'est pas de type **String**, il y a une conversion de données à faire.

Nous avons jusqu'à maintenant, soigneusement évité ces problèmes, en faisant en sorte que tous les champs de [Formulaire] soient de type **String**. Il est maintenant temps de nous intéresser à ce problème.

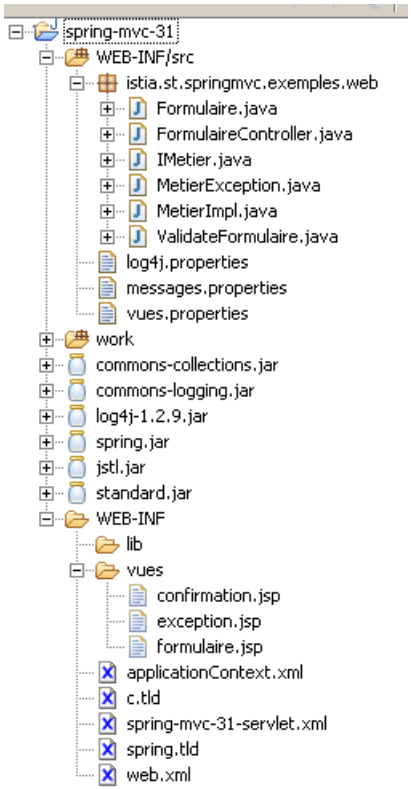
Revenons sur l'architecture 3tier d'une application web :



Lorsqu'à l'étape 1, l'utilisateur demande un formulaire, celui-ci va être initialisé par un objet provenant en général des couches [métier] et [dao] (étape 3). Cet objet est représentatif du domaine de l'application. Il n'y a bien sûr aucune raison que tous ses champs soient de type **String**. Pour que les valeurs des champs de l'objet puissent être placés dans le flux HTTP envoyé à l'étape 7, des conversions **[Type] -> String** devront en général être faites.

Une fois obtenue la page du formulaire, l'utilisateur va faire des saisies et les poster. Les étapes 1-7 vont de nouveau se répéter. L'objet obtenu lors du GET va être mis à jour par des valeurs postées de type **String**. Là des conversions **String -> [Type]** devront être faites. Puis l'objet mis à jour sera transmis de nouveau aux couches [métier] et [dao], peut-être pour mémorisation (étape 3).

Pour illustrer ce problème de conversion de données, nous utiliserons le projet Eclipse suivant :



L'architecture utilisée est la même que précédemment. Une couche [métier] est utilisée pour fournir l'objet [Formulaire] qui va nourrir le formulaire HTML qui va être affiché. L'interface [IMetier] est la suivante :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public interface IMetier {
4.    public Formulaire getFormulaire();
5.  }
```

Elle ne fournit que l'objet [Formulaire] et rien d'autre. Celui-ci sera le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.io.File;
4.  import java.net.URL;
5.  import java.util.Date;
6.  import java.util.Locale;
7.  import java.util.Properties;
8.
9.  public class Formulaire {
10.
11.   private int _int;
12.   private Integer _Integer;
13.   private double _double;
14.   private Double _Double;
15.   private boolean _boolean;
16.   private Boolean _Boolean;
17.   private Class _Class;
18.   private File _File;
19.   private Locale _Locale;
20.   private Properties _Properties;
21.   private String[] _Strings;
22.   private byte[] _bytes;
23.   private URL _URL;
24.   private Date _Date;
25.
26. // getters - setters
27.   public boolean is_boolean() {
28.     return _boolean;
29.   }
30.
31.   public void set_boolean(boolean _boolean) {
32.     this._boolean = _boolean;
33.   }
34. ...
35.   // méthode toString
36.   public String toString() {
37.     return "[int," + _int + "]<br>" + "[Integer," + _Integer + "]<br>"
38.         + "[double," + _double + "]<br>" + "[Double," + _Double
```

```
39.       + "]<br>" + "[boolean," + _boolean + "]<br>" + "[Boolean,"
40.       + _Boolean + "]<br>" + "[Class," + _Class.getName() + "]<br>"
41.       + "[File," + _File.getName() + "]<br>" + "[URL,"
42.       + _URL.toExternalForm() + "]<br>" + "[Locale,"
43.       + _Locale.getDisplayName() + "]<br>" + "[Properties,"
44.       + _Properties + "]<br>" + "[String[],nombre d'éléments="
45.       + _Strings.length + "]<br>" + "[byte[],nombre d'éléments=" + _bytes.length
46.       + "]<br>" + "[Date,"+_Date+"]<br>";
47.   }
48. }
```

- lignes 11-24 : des champs de types divers. Pour tous ces champs, Spring sait faire les conversions **String <-> Type**. Néanmoins, pour le champ de type [**java.util.Date**], nous souhaitons que la chaîne associée soit au format JJ/MM/AAAA. Pour cela, nous aurons besoin de définir un **éditeur de propriété** particulier. Un éditeur de propriété permet de faire les conversions **String <-> Type**, pour un **Type** donné. Spring a un certain nombre d'éditeurs de propriété par défaut pour tous les types ci-dessus. Il faut simplement que les " String " respectent un certain format. Le format par défaut pour le type [java.util.Date] ne nous convenant pas, nous serons amenés à enregistrer un éditeur de propriété particulier.
- ligne 36 : méthode [toString] que nous utiliserons dans la page de confirmation [confirmation.jsp] pour afficher la valeur de l'objet [Formulaire] posté.

La classe d'implémentation [MetierImpl] de l'interface [IMetier] sera la suivante :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Random;
4.
5.  public class MetierImpl implements IMetier {
6.
7.    private Random random=new Random();
8.
9.    public Formulaire getFormulaire() {
10.     if(random.nextInt(2)==0){
11.       // création formulaire
12.       return new Formulaire();
13.     }else{
14.       throw new MetierException("Exception dans la couche métier");
15.     }
16.   }
17.
18. }
```

- ligne 5 : la classe implémente l'interface [IMetier] et donc la méthode [getFormulaire] (lignes 9-16)
- lignes 9-16 : la méthode [getFormulaire] se contente de rendre une instance [Formulaire] non initialisée. De façon aléatoire, elle génère une exception de type [MetierException] comme dans l'application précédente. La classe [MetierException] n'a pas été modifiée.

La configuration de l'application [applicationContext.xml, spring-mvc-31-servlet.xml] est identique à celle de l'application précédente [spring-mvc-30]. L'unique URL acceptée est [/formulaire.html]. Elle est traitée (GET et POST) par le contrôleur [FormulaireController]. Celui-ci a un validateur de type [ValidateFormulaire].

Le contrôleur [FormulaireController] est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.text.SimpleDateFormat;
4.
5.  import javax.servlet.http.HttpServletRequest;
6.  import javax.servlet.http.HttpServletResponse;
7.
8.  import org.springframework.beans.propertyeditors.CustomDateEditor;
9.  import org.springframework.validation.BindException;
10. import org.springframework.web.bind.ServletRequestDataBinder;
11. import org.springframework.web.servlet.ModelAndView;
12. import org.springframework.web.servlet.mvc.SimpleFormController;
13.
14. public class FormulaireController extends SimpleFormController {
15.
16.   // la couche métier
17.   private IMetier metier;
18.
19.   public IMetier getMetier() {
20.     return metier;
21.   }
22.
23.   public void setMetier(IMetier metier) {
24.     this.metier = metier;
25.   }
```

```
26.
27.  protected void initBinder(HttpServletRequest request,
28.      ServletRequestDataBinder binder) throws Exception {
29.    // format attendu pour la date de naissance
30.    SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
31.    // format strict
32.    dateFormat.setLenient(false);
33.    // on enregistre un éditeur de propriétés String (dd/MM/yyyy) -> Date
34.    // CustomDateEditor est fourni par Spring - il sera utilisé par Spring
35.    // pour transformer
36.    // la chaîne saisie dans le formulaire en type java.util.Date
37.    // la date ne pourra être vide (2ième paramètre de CustomDateEditor)
38.    binder.registerCustomEditor(java.util.Date.class, null,
39.        new CustomDateEditor(dateFormat, false));
40.  }
41.
42.  // l'objet adossé au formulaire
43.  protected Object formBackingObject(HttpServletRequest request) {
44.    // on récupère le formulaire dans la session s'il existe
45.    Formulaire formulaire = (Formulaire)request.getSession().getAttribute("formulaire");
46.    if (formulaire == null) {
47.      // on demande un objet Formulaire à la couche métier
48.      formulaire = metier.getFormulaire();
49.    }
50.    // on rend le formulaire
51.    return formulaire;
52.  }
53.
54.  // traitement du POST
55.  protected ModelAndView onSubmit(HttpServletRequest request,
56.      HttpServletResponse response, Object formulaire,
57.      BindException errors) {
58.    // on met le formulaire dans la session
59.    request.getSession().setAttribute("formulaire", formulaire);
60.    // on rend le [ModelAndView]
61.    return new ModelAndView("confirmation",null);
62.  }
63. }
```

- la méthode [**formBackingObject**] qui rend l'objet miroir du formulaire HTML et la méthode [**onSubmit**] qui traite le POST sont inchangées.
- la méthode [**initBinder**] permet d'associer au contrôleur des éditeurs de propriété non définis par défaut.

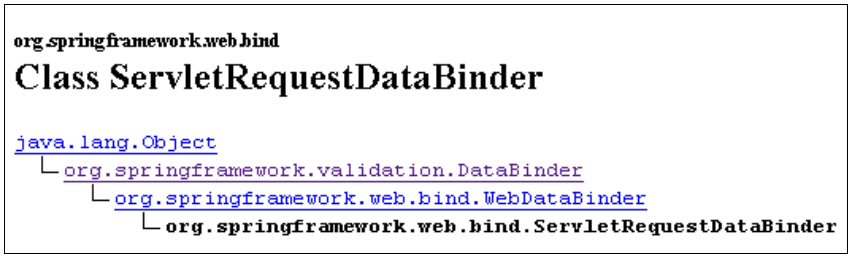
Revenons sur le flux d'exécution du GET défini page 27 :

1. le contrôleur reçoit une requête de type **GET** pour le formulaire
2. la méthode [**formBackingObject**] de la classe de base [AbstractFormController] est appelée. Si elle n'a pas été redéfinie, elle rend une instance de type [**commandClass**] défini en général par configuration. Si elle a été redéfinie, elle doit rendre un objet dont les propriétés seront utilisées pour initialiser le formulaire. Par la suite, nous appellerons **command** l'objet créé à cette étape. Si l'attribut " **sessionForm** " de [SimpleFormController] est à *true*, l'objet **command** est mis en session.
3. la méthode [**initBinder**] de la classe de base [BaseCommandController] est appelée. Si redéfinie, cette méthode permet de déclarer des éditeurs de propriétés. Ces objets permettent de faire des conversions entre le type non String de certaines des propriétés de l'objet [**command**] et le flux HTTP de type String échangé entre le client et le serveur. Par exemple, une date dans un formulaire est envoyée comme une chaîne de caractères au serveur. L'éditeur de propriétés va permettre la trasformation de ce type [String] en un type [java.util.Date] de la propriété de [**command**] qui doit être initialisée par la chaîne.

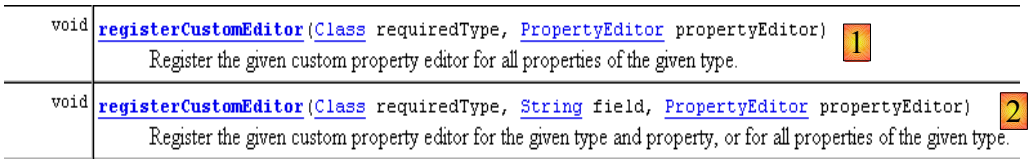
On voit qu'à l'étape 3, la méthode [**initBinder**] du contrôleur est appelée. Si on ne redéfinit pas cette méthode, seuls les éditeurs de propriété prédéfinis dans Spring seront disponibles. Ici nous voulons un éditeur de propriété capable de faire la conversion dans les deux sens de **" JJ/MM/AAAA " <-> java.util.Date**. Nous sommes amenés à redéfinir la méthode [initBinder], lignes 27-40.

- lignes 27-28 : la méthode [initBinder] reçoit la requête [request] en cours de traitement et un objet [**ServletRequestDataBinder** binder] qui va lui permettre d'enregistrer de nouveaux éditeurs de propriété.

La classe [ServletRequestDataBinder] est définie comme suit :



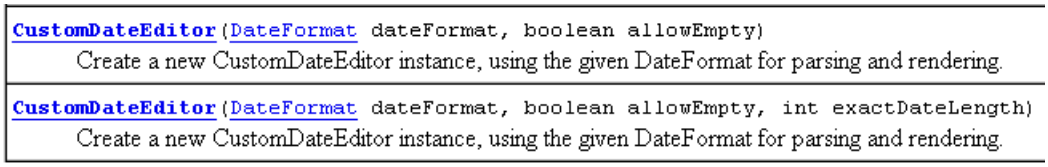
Elle dérive de la classe [DataBinder] qui possède les deux méthodes suivantes pour enregistrer des éditeurs de propriétés :



- la méthode (1) permet d'associer à un type de donnée, un éditeur de propriété particulier. L'éditeur ainsi défini est valable quelque soit le champ concerné de l'objet **command** miroir du formulaire HTML.
- la méthode (2) fait la même chose mais pour un champ précis uniquement (second paramètre). Ainsi dans l'objet **command**, on peut avoir deux champs de même type à qui on appliquera des éditeurs de propriété différents.

Le type [PropertyEditor] (second paramètre de 1, troisième de 2) est une interface. La classe [CustomDateEditor] implémente cette interface :

La classe [CustomDateEditor] possède deux constructeurs :



Nous utiliserons le premier. Le premier paramètre est de type [**DateFormat**] qui est une classe abstraite. Nous utiliserons ici sa classe dérivée [**SimpleDateFormat**].

Revenons au code de la méthode [initBinder] :

```
1.   protected void initBinder(HttpServletRequest request,
2.       ServletRequestDataBinder binder) throws Exception {
3.     // format attendu pour la date de naissance
4.     SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");
5.     // format strict
6.     dateFormat.setLenient(false);
7.     // on enregistre un éditeur de propriétés String (dd/MM/yyyy) -> Date
8.     // CustomDateEditor est fourni par Spring - il sera utilisé par Spring
9.     // pour transformer
10.    // la chaîne saisie dans le formulaire en type java.util.Date
11.    // la date ne pourra être vide (2ième paramètre de CustomDateEditor)
12.    binder.registerCustomEditor(java.util.Date.class,new CustomDateEditor(dateFormat, false));
13.  }
```

- ligne 4 : nous créons une instance de [SimpleDateFormat]. Cette classe permet de gérer des conversions java.util.Date <-> java.lang.String selon un format précis pour le type String. Ce format est le paramètre du constructeur utilisé ligne 4.

- ligne 6 : on impose un strict respect du format.
- ligne 12 : on enregistre un éditeur de propriété pour le type [java.util.Date]. L'éditeur est une instance [CustomDateEditor]. Le premier paramètre du constructeur de cette instance doit être de type [DateFormat]. Nous fournissons l'instance dérivée [SimpleDateFormat] construite en 4. Le second paramètre indique si une date vide " " doit être analysée. La réponse est non. Si ce cas se produisait, une exception de type [IllegalArgumentException] serait lancée.

Au final, bien que la méthode [initBinder] n'ait que quelques lignes, son fonctionnement est complexe à comprendre dans les détails. Le lecteur perdu par cette longue démonstration ne retiendra que le fait que l'utilisateur va pouvoir entrer une date sous la forme JJ/MM/AAAA et que par la magie de l'éditeur de propriété enregistré dans [initBinder], cette chaîne sera convertie en un type [java.util.Date].

Il nous faut maintenant présenter le validateur [ValidateFormulaire]. Rappelons-nous que celui-ci n'intervient qu'une fois l'objet [Formulaire] a reçu les valeurs postées, donc une fois que les conversions String -> Type ont eu lieu. Si lors de ces conversions, des erreurs ont été détectées, le flux d'exécution est arrêté et l'objet [Formulaire] réaffiché. Si le validateur [ValidateFormulaire] intervient, c'est donc que toutes les conversions String -> Type **ont réussi**. Il ne sert donc qu'à faire des vérifications supplémentaires :

Le validateur [ValidateFormulaire] sera le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import org.springframework.validation.Errors;
4.
5.  public class ValidateFormulaire implements
6.     org.springframework.validation.Validator {
7.
8.   public boolean supports(Class classe) {
9.     return classe.isAssignableFrom(Formulaire.class);
10.  }
11.
12.  public void validate(Object obj, Errors erreurs) {
13.    // on récupère le formulaire posté
14.    Formulaire formulaire = (Formulaire) obj;
15.    // on vérifie le champ de saisie _int
16.    if(formulaire.get_int()<0){
17.      erreurs.rejectValue("_int",
18.      "formulaire.int.positif");
19.    }
20.    // on vérifie le champ de saisie _Class
21.    if(formulaire.get_Class()==null){
22.      erreurs.rejectValue("_Class",
23.      "formulaire.Class.requis");
24.    }
25.    // on vérifie le champ de saisie _File
26.    if(formulaire.get_File()==null || formulaire.get_File().getName().trim().length()==0){
27.      erreurs.rejectValue("_File",
28.      "formulaire.File.requis");
29.    }
30.    // on vérifie le champ de saisie _Locale
31.    if(formulaire.get_Locale()==null){
32.      erreurs.rejectValue("_Locale",
33.      "formulaire.Locale.requis");
34.    }
35.    // on vérifie le champ de saisie _Properties
36.    if(formulaire.get_Properties()==null || formulaire.get_Properties().size()==0){
37.      erreurs.rejectValue("_Properties",
38.      "formulaire.Properties.requis");
39.    }
40.    // on vérifie le champ de saisie _Strings
41.    if(formulaire.get_Strings()==null || formulaire.get_Strings().length==0){
42.      erreurs.rejectValue("_Strings",
43.      "formulaire.Strings.requis");
44.    }
45.    // on vérifie le champ de saisie _bytes
46.    if(formulaire.get_bytes()==null){
47.      erreurs.rejectValue("_bytes",
48.      "formulaire.bytes.requis");
49.    }
50.  }
51.
52. }
```

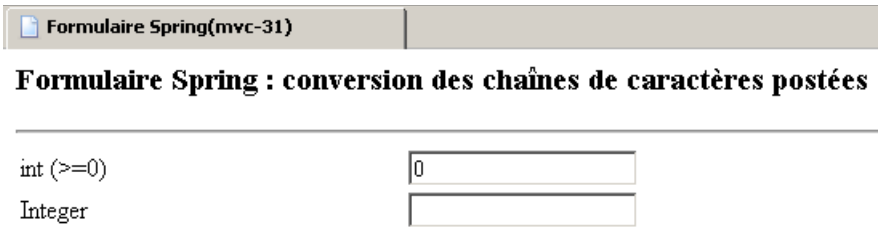
- lignes 8-10 : le validateur valide tout objet de type [Formulaire] ou dérivé
- ligne 14 : on récupère l'objet [**Formulaire**] à valider
- lignes 16-19 : le champ **_int** doit être >=0
- lignes 21-24 : le champ **_Class** doit être non **null**
- lignes 26-29 : le champ **_File** doit être non **null** et le nom du fichier non vide

- lignes 31-34 : le champ **_Locale** doit être non **null**
- lignes 35-39 : le champ **_Properties** doit être non **null** et le nombre de propriétés non nul
- lignes 41-44 : le tableau **_Strings** doit être non **null** et non vide
- lignes 46-49 : le tableau **_bytes** doit être non **null**

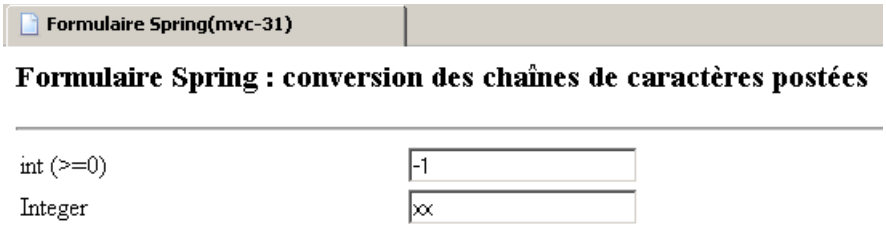
Le validateur utilise des codes d'erreur définis dans le fichier [**messages.properties**] :

```
1.  # messages d'erreur
2.  #typeMismatch=Donnée incorrecte !
3.  formulaire.int.positif=Saisissez un nombre positif ...
4.  formulaire.Class.requis=Nom de la classe requis ...
5.  formulaire.File.requis=Nom de fichier requis ...
6.  formulaire.Locale.requis=Locale requis ...
7.  formulaire.Properties.requis=Indiquez au moins une propriété ...
8.  formulaire.Strings.requis=Indiquez au moins une chaîne de caractères ...
9.  formulaire.bytes.requis=Tapez au moins un caractère ...
```

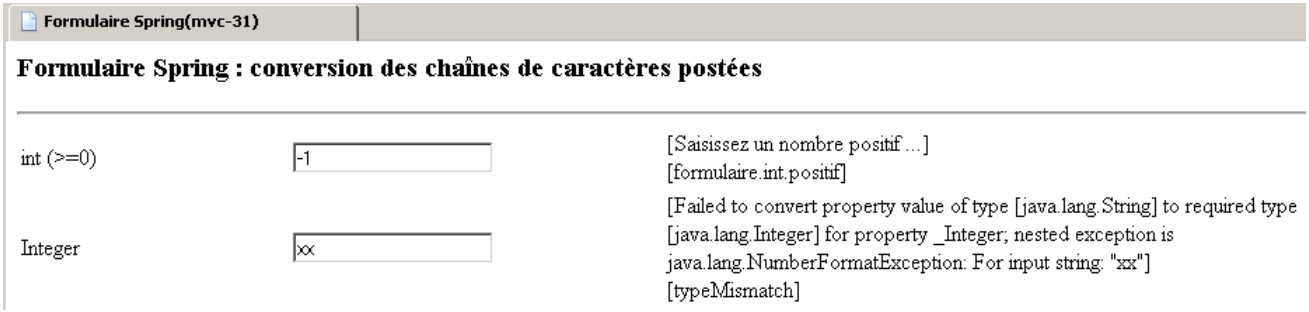
La page affichée à l'issue du GET /formulaire.html est la suivante (vue partielle) :



Si on poste le formulaire suivant :



on reçoit la réponse suivante :



Les deux erreurs ne sont pas de même nature :

- l'erreur sur le champ **_int** a été détectée par le validateur [ValidateFormulaire]. On le reconnaît au message.
- l'erreur sur le champ **_Integer** a elle été détectée par l'éditeur de propriété du type [Integer] qui n'a pas réussi à convertir " xx " en une valeur **Integer**. On n'a pas la maîtrise du message d'erreur et c'est pourquoi il est en anglais.
- outre le message d'erreur, l'écran affiche le code du message : [formulaire.int.positif] pour le premier champ, [typeMismatch] pour le second.

La page JSP [formulaire.jsp] qui a provoqué ces différents affichages est la suivante (vue partielle) :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ taglib uri="/WEB-INF/spring.tld" prefix="spring" %>
4.  <%@ page isELIgnored="false" %>
5.  <html>
6.    <head>
7.     <title>Formulaire Spring(mvc-31)</title>
8.    </head>
```

```
9.    <body>
10.    <h3>Formulaire Spring : conversion des cha&icirc;nes de caract&egrave;res post&eacute;es</h3>
11.     <hr>
12.     <form method="post">
13.       <table border="0">
14.       <!-- champ de saisie -->
15.         <tr>
16.           <td>int (>=0)</td>
17.           <spring:bind path="formulaire._int">
18.              <td>
19.                 <input type="text" name="${status.expression}" value="${status.value}">
20.              </td>
21.              <c:if test="${status.error}">
22.               <td>[${status.errorMessage}]<br>[${status.errorCode}]</td>
23.              </c:if>
24.           </spring:bind>
25.         </tr>
26.         <tr>
27.           <td>Integer</td>
28.           <spring:bind path="formulaire._Integer">
29.              <td>
30.                 <input type="text" name="${status.expression}" value="${status.value}">
31.              </td>
32.              <c:if test="${status.error}">
33.               <td>[${status.errorMessage}]<br>[${status.errorCode}]</td>
34.              </c:if>
35.           </spring:bind>
36.         </tr>
37. ....
38.       </table>
39.      <br>
40.      <hr>
41.      <input type="submit" value="Envoyer">
42.     </form>
43.   </body>
44. </html>
```

Examinons simplement le code lié au champ de saisie **_int** :

```
1.  <tr>
2.            <td>int (>=0)</td>
3.            <spring:bind path="formulaire._int">
4.               <td>
5.                  <input type="text" name="${status.expression}" value="${status.value}">
6.               </td>
7.               <c:if test="${status.error}">
8.                <td>[${status.errorMessage}]<br>[${status.errorCode}]</td>
9.               </c:if>
10.           </spring:bind>
11.         </tr>
```

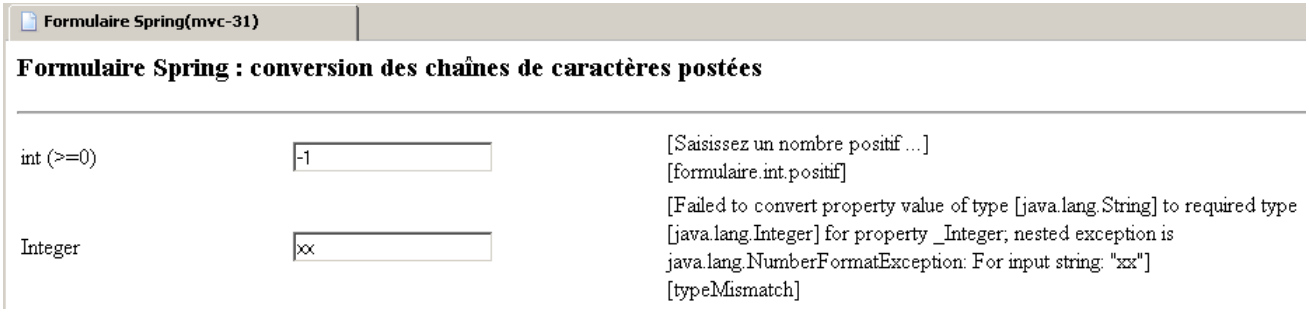
Rappelons qu'à l'intérieur de la balise <spring:bind> :

- **status.expression** est le nom du champ lié à la balise <spring:bind>, ici **_int**
- **status.value** est la valeur postée pour le champ lié à la balise <spring:bind>
- **status.errorMessage** est le message d'erreur lié au champ s'il y a eu erreur

Enfin, ligne 8, une expression non encore rencontrée :

- **status.errorCode** est le code du message d'erreur lié au champ s'il y a eu erreur

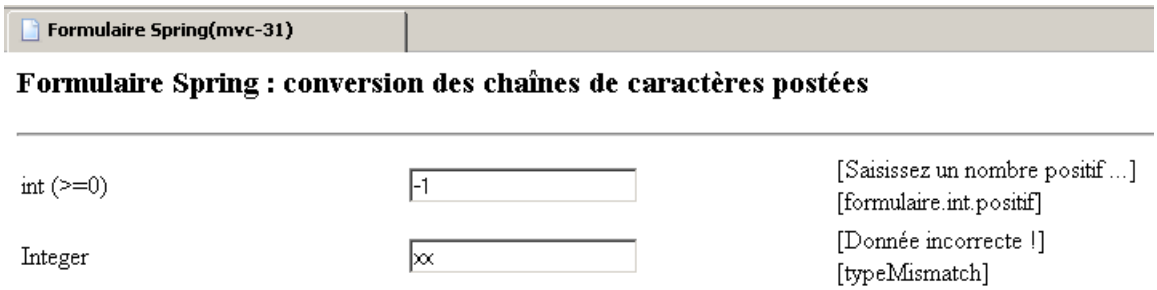
Ce code JSP explique ce qui apparaît dans la page d'erreurs :



On voit que le code d'erreur sur le champ **_Integer** est " typeMismatch ". Mettons ce code dans [**messages.properties**] et associons-le à un message en français :

```
1.  # messages d'erreur
2.  typeMismatch=Donnée incorrecte !
3.  formulaire.int.positif=Saisissez un nombre positif ...
4.  formulaire.Class.requis=Nom de la classe requis ...
5.  formulaire.File.requis=Nom de fichier requis ...
6.  formulaire.Locale.requis=Locale requis ...
7.  formulaire.Properties.requis=Indiquez au moins une propriété ...
8.  formulaire.Strings.requis=Indiquez au moins une chaîne de caractères ...
9.  formulaire.bytes.requis=Tapez au moins un caractère ...
```

Le code [typeMismatch] a été défini ligne 2. Recommençons les deux saisies erronées. On obtient cette fois la page d'erreurs suivante :



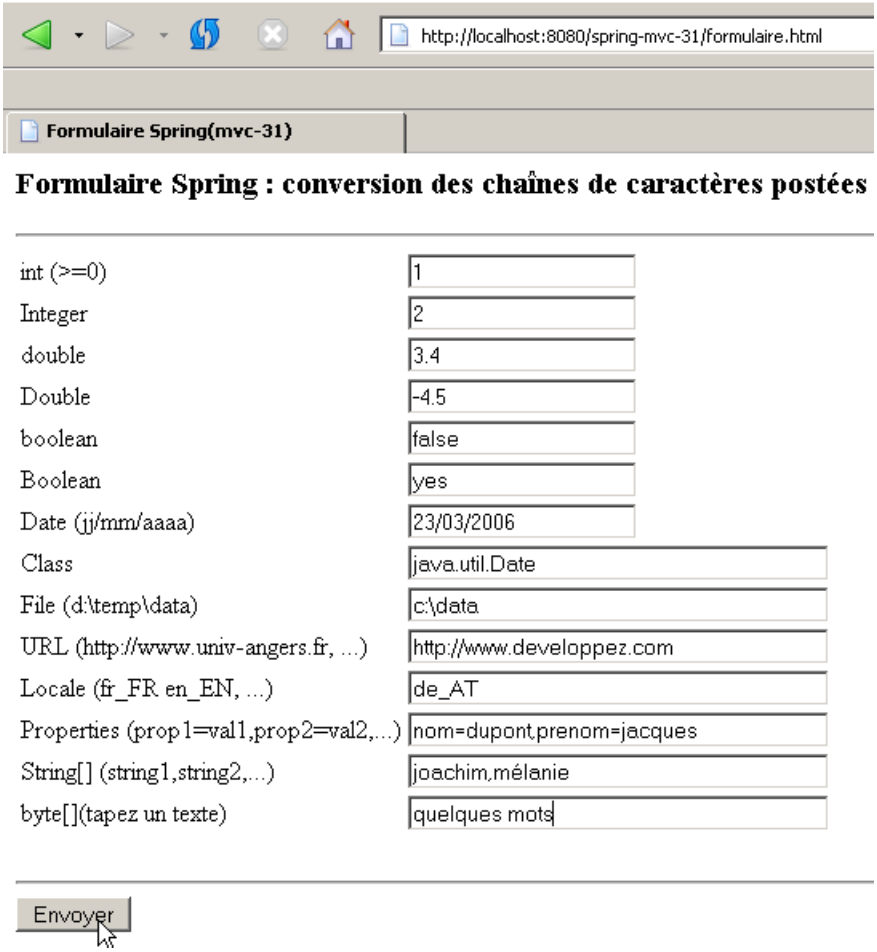
Le message d'erreur lié au code [typeMismatch] a bien été redéfini.

Lorsqu'aucune erreur n'est détectée, la page [confirmation.jsp] est affichée :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Formulaire Spring(mvc-31)</title>
8.    </head>
9.    <body>
10.     <h3>Confirmation des donn&eacute;es saisies</h3>
11.     ${formulaire}
12.     <br>
13.     <a href="<c:url value="/formulaire.html"/>">Retour au formulaire</a>
14.   </body>
15. </html>
```

- ligne 11 : affiche l'objet associé à la clé " **formulaire** ". C'est la méthode [toString] de cet objet qui va implicitement être appelée.

Voici un exemple de saisies :

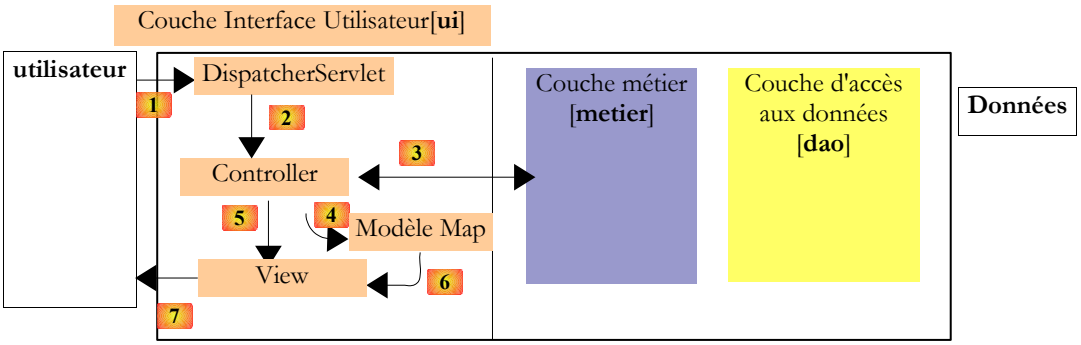


et la page de confirmation :

Beaucoup d'autres détails mériteraient d'être expliqués. Nous pensons cependant que l'essentiel a été dit et nous laissons au lecteur, le soin d'approfondir cet exemple, notamment en procédant lui-même aux tests.

6 Conclusion

Revenons à l'architecture 3tier d'une application web :



Au cours des articles 1 et 2, nous avons passé en revue les principaux morceaux du puzzle qui contribue au traitement d'une requête du client. Ce puzzle, symbolisé par les étapes 1 à 7 ci-dessus, est complexe et sa courbe d'apprentissage assez raide quoiqu'on puisse lire ici et là. Elle est comparable à la courbe d'apprentissage de Struts. Mais comme pour toute technologie, une fois maîtrisée, la méthodologie Spring MVC devient naturelle et peut amener des gains de productivité. Il faut simplement dépasser la phase d'apprentissage. Ces deux articles ont voulu apporter une contribution sur ce point.

Dans un article à venir, nous présenterons d'autres contrôleurs prédéfinis dans Spring. On rappelle qu'un contrôleur implémente l'interface [Controller] :

org.springframework.web.servlet.mvc
Interface Controller

All Known Implementing Classes:

AbstractCommandController, AbstractController, AbstractFormController, AbstractUrlViewController, AbstractWizardFormController, BaseCommandController, BurlapServiceExporter, CancellableFormController, HessianServiceExporter, HttpInvokerServiceExporter, MultiActionController, ParameterizableViewController, ServletForwardingController, ServletWrappingController, SimpleFormController, UrlFilenameViewController

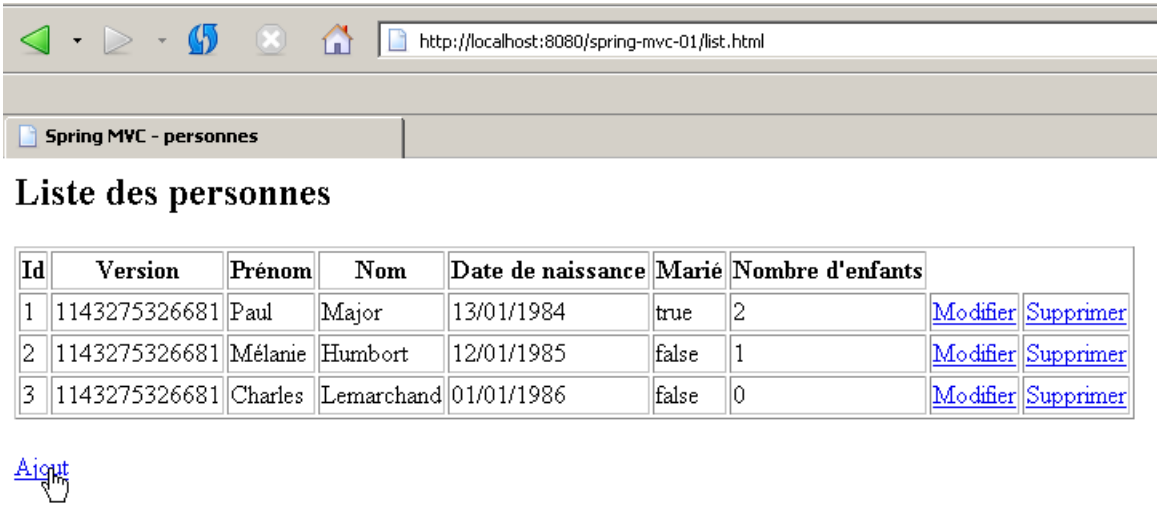
Nous présenterons les implémentations suivantes : **AbstractWizardFormController**, **MultiActionController**, **ParameterizableViewController**.

Ce qui a été fait jusqu'ici est suffisant pour écrire des applications web réalistes. Le lecteur peut s'essayer par exemple sur l'application décrite dans l'article [http://tahe.developpez.com/java/web3tier/]. Cet article décrit une application web de panier électronique implémentée sous trois formes :

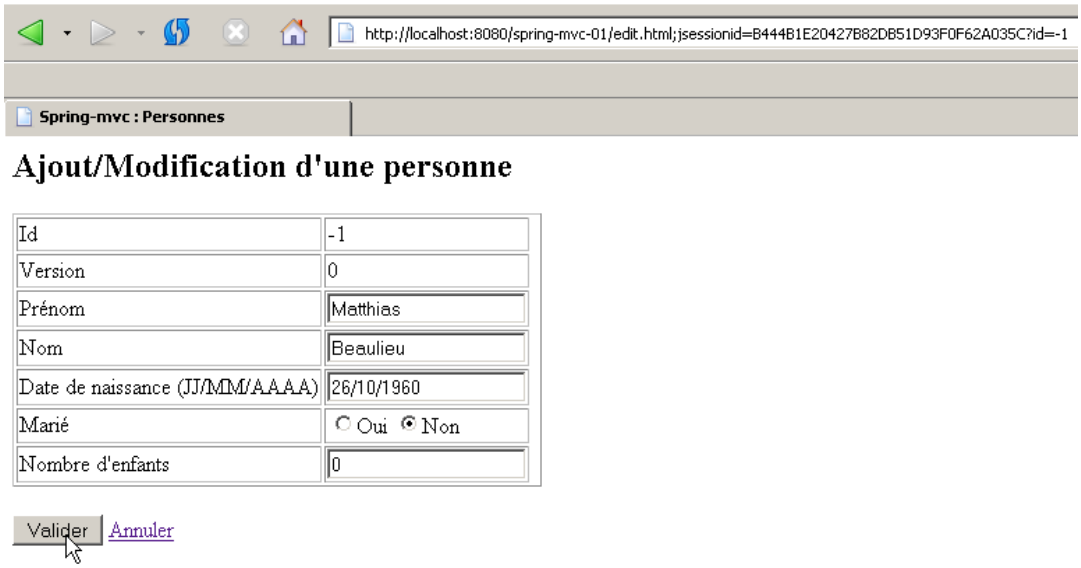
1. avec une servlet propriétaire
2. avec une forme allégée du framework Struts
3. avec une forme allégée du framework Spring MVC

L'implémentation [3] n'utilisait pas toutes les capacités de Spring MVC. Le lecteur est invité à écrire une quatrième implémentation qui utiliserait mieux Spring MVC. Les codes des trois implémentation déjà faites sont disponibles sur le site de l'article.

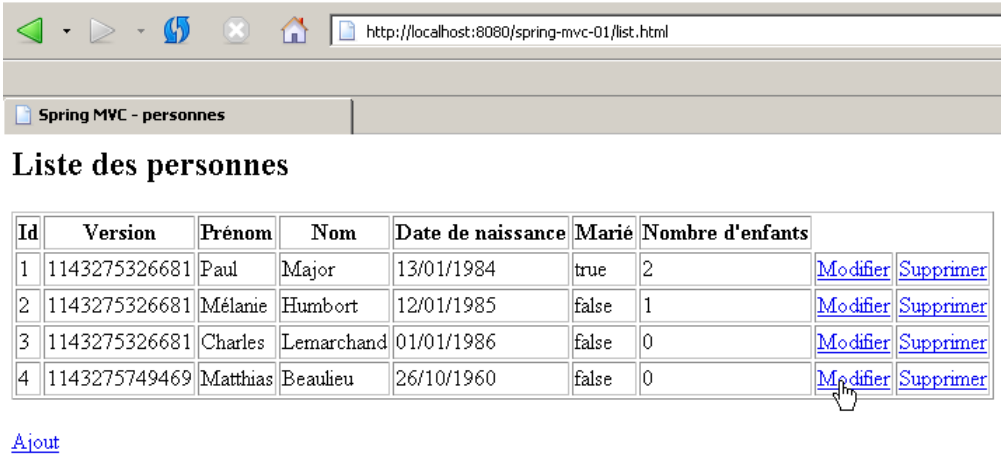
Dans l'article prochain, nous présenterons une application 3tier destinée à gérer une liste de personnes :



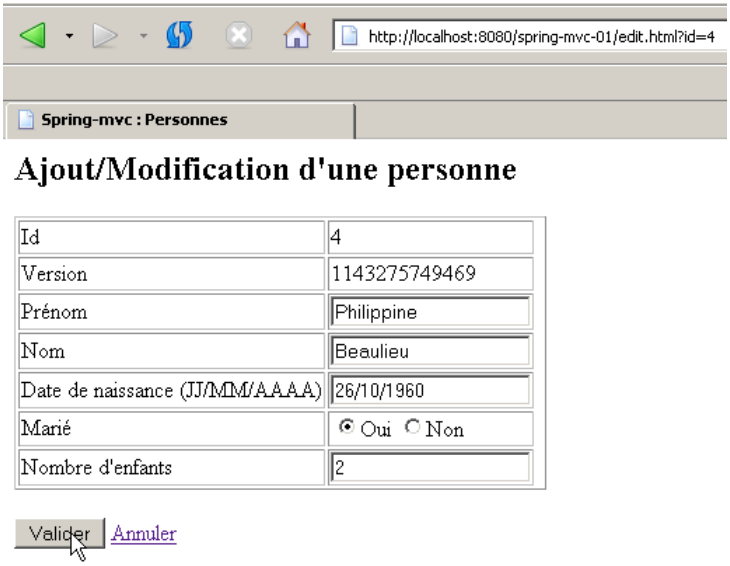
Nous pouvons ajouter une nouvelle personne :



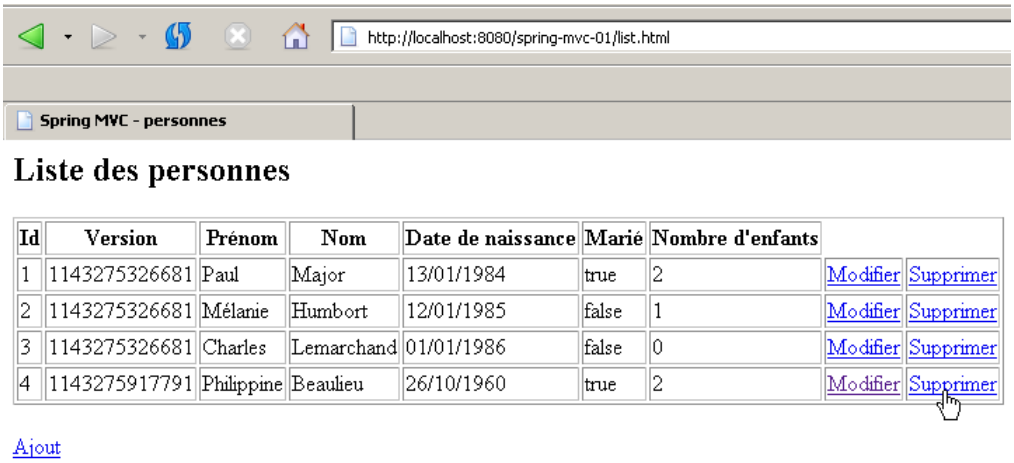
La personne ajoutée est alors intégrée dans la liste :



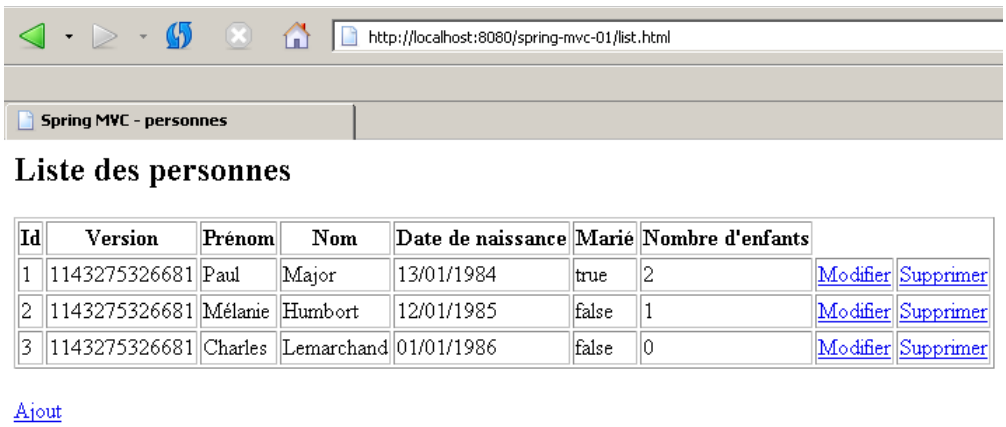
Nous pouvons ensuite modifier cette personne :



La personne modifiée apparaît dans la liste :



On peut alors la supprimer. Elle disparaît alors de la liste :



7 Le code de l'article

Comme pour l'article 1, le lecteur trouvera le code des exemples de l'article 2 sous la forme d'un fichier zippé sur le site de l'article. Les règles de déploiement du fichier zippé sont à relire dans l'article 1. Une fois un projet importé dans [Eclipse] :

- copier le contenu du dossier [lib] du zip dans le dossier [WEB-INF/lib] du projet
- s'assurer que le dossier [work] existe sinon le créer : [clic droit sur projet / Projet Tomcat / Créer le dossier work]
- nettoyer le projet [Project / clean / clean selected projects]

Table des matières