www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

Spring MVC par l'exemple
- Partie 3 -

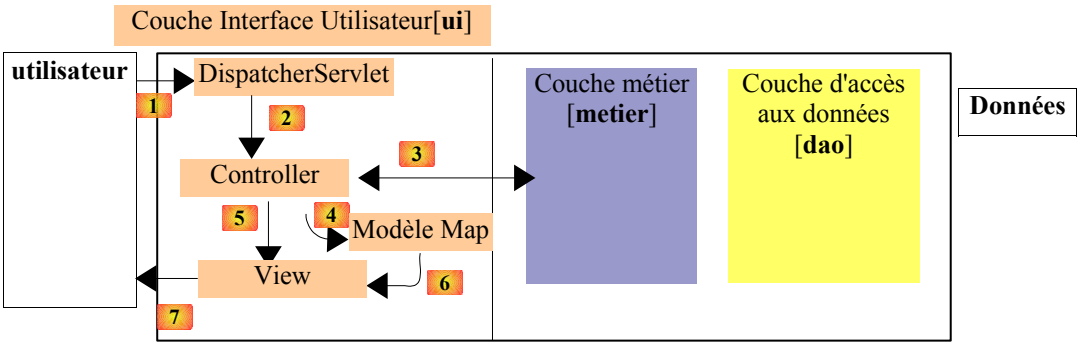
serge.tahe@istia.univ-angers.fr, avril 2006

1 Rappels

Nous poursuivons dans cet article le travail fait dans les précédents :

- Spring MVC par l'exemple – partie 1 : [http://tahe.developpez.com/java/springmvc-part1]
- Spring MVC par l'exemple – partie 2 : [http://tahe.developpez.com/java/springmvc-part2]

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 (1,2)
- 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] (4,5)

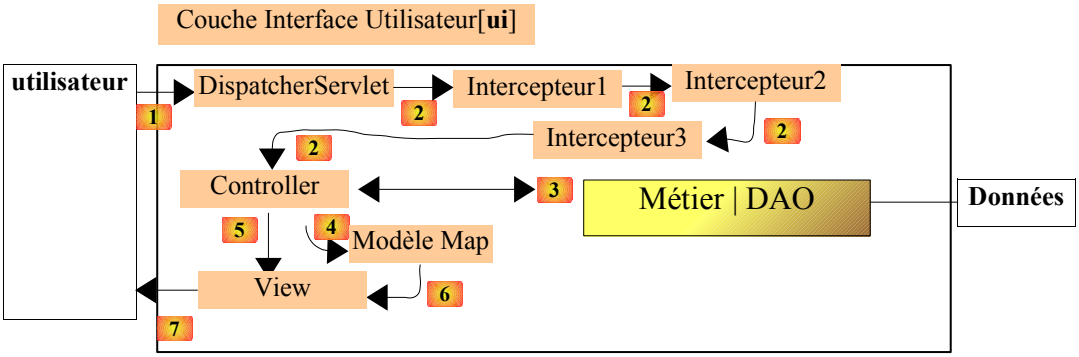
Nous avons vu dans la partie 2 de l'article les points suivants :
- les classes [HandlerInterceptor] qui filtrent la requête avant de la passer au [Controller] qui doit la traiter (2)
- les différentes façons de gérer l'internationalisation (*localisation*) d'une application web
- les gestionnaires des exceptions qui peuvent de produire dans une application web
- la gestion des formulaires grâce à la classe [SimpleFormController]

Dans le présent article, nous présentons :
- quelques implémentations de l'interface [Controller] non encore abordées
- des implémentations de l'interface [View] permettant de générer des documents PDF et Excel
- des outils facilitant l'écriture de formulaires destinés à télécharger des documents du poste client vers le serveur

L'application exemple qui devait être présentée dans cet article le sera dans le prochain.

2 Encore des contrôleurs

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



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. Nous nous proposons ici de présenter de nouveaux objets [Controller].

2.1 Le contrôleur ParameterizableViewController

L'interface [Controller] est la suivante :

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

Parmi les classes implémentant cette interface, on voit ci-dessus, la classe [ParameterizableViewController]. Ce contrôleur ne fait aucun travail sur la requête qu'il reçoit. Il se contente de déterminer la vue qui doit être affichée. Celui-ci est généralement fixé par configuration. Le contrôleur [ParameterizableViewController] ne construit aucun modèle pour la vue qu'il demande d'afficher. Celle-ci doit donc récupérer son modèle, si elle en a un, dans les différents contextes de l'application : requête, session, application.

La classe [ParameterizableViewController] est définie comme suit :

Constructor Summary

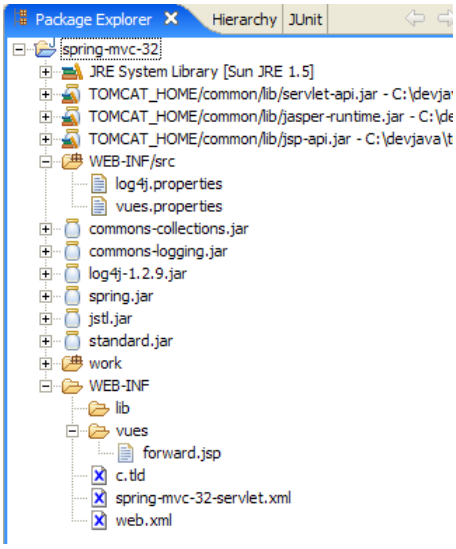
ParameterizableViewController()

Method Summary

String	getViewName() Return the name of the view to delegate to.
protected ModelAndView	handleRequestInternal(HttpServletRequest request, HttpServletResponse response) Return a ModelAndView object with the specified view name.
protected void	initApplicationContext() Subclasses can override this for custom initialization behavior.
void	setViewName(String viewName) Set the name of the view to delegate to.

- la méthode [handleRequestInternal] appelée par [DispatcherServlet] pour créer un objet [ModelAndView] à partir de la requête courante, se contente de rendre le modèle : *new ModelAndView(getViewName())*. Elle ne rend aucun modèle [Map].
- La méthode [setViewName] permet de fixer la vue du [ModelAndView]

Pour illustrer l'usage de cette classe, nous utiliserons le projet Eclipse / Tomcat (spring-mvc-32) suivant :



Le fichier de configuration Spring (spring-mvc-32-servlet.xml) de l'application 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="/accueil.html">ForwardController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- le résolveur de vues -->
13.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
14.     <property name="basename">
15.       <value>vues</value>
16.     </property>
17.   </bean>
18.   <!-- les contrôleurs de l'application-->
19.   <bean id="ForwardController"
20.     class="org.springframework.web.servlet.mvc.ParameterizableViewController">
21.       <property name="viewName" value="forward"/>
22.   </bean>
23. </beans>
```

- lignes 5-11 : définissent les Url gérées par l'application
- ligne 8 : seule l'url */accueil.html* est ici traitée. Elle le sera par le contrôleur d'id [ForwardController] défini aux lignes 19-22.
- lignes 19-22 : le contrôleur est de type [ParameterizableViewController]. Le nom de la vue est fixée par la propriété [viewName] ligne 21, forçant Spring à appeler la méthode [setViewName] de la classe. La vue s'appelera donc *forward*. On se sait pas encore à quoi elle correspond.
- lignes 13-17 : définit le mode de résolution des noms de vues. Ici le résolveur est [ResourceBundleViewResolver] avec l'attribut *basename* égal à *vues*. Ceci signifie que la définition des vues sera trouvée dans le fichier de propriétés [vues.properties] situé dans le *ClassPath* de l'application.

Le fichier [vues.properties] a été placé ici dans [WEB-INF/src]. Le projet Eclipse / Tomcat le recopie automatiquement dans [WEB-INF/classes], le plaçant ainsi dans le *ClassPath* de l'application. Son contenu est le suivant :

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

La vue associée au nom [forward] est une page JSP (ligne 3). Sa classe d'affichage est donc la classe [JstlView] (ligne 2).

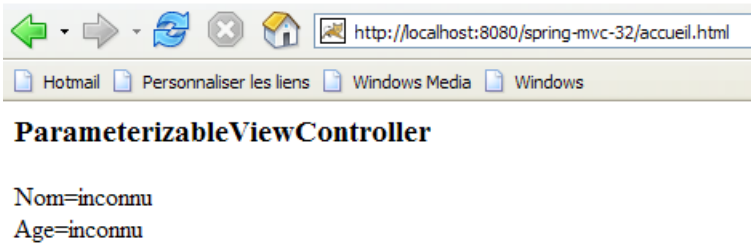
La page JSP [forward.jsp] est la suivante :

```
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.  <%
6.    // on récupère les paramètres de la requête
7.    String nom=request.getParameter("nom");
8.    if(nom==null) nom="inconnu";
9.    String age=request.getParameter("age");
10.   if(age==null) age="inconnu";
11. %>
12. <html>
13.   <head>
14.     <title>Spring(mvc-32)</title>
15.   </head>
16.   <body>
17.     <h3>ParameterizableViewController</h3>
18.     Nom=<%=nom%><br>
19.     Age=<%=age%><br>
20.   </body>
21. </html>
```

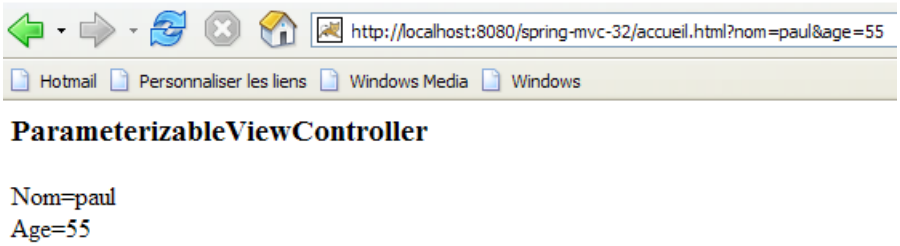
- la requête qui sera traitée par le contrôleur [ParameterizableViewController] sera de la forme */forward.html?nom=XX&age=YY*.
- lignes 7-8 : on récupère le paramètre [nom] de la requête. S'il est absent, on lui donne une valeur arbitraire.
- lignes 9-10 : on fait de même pour le paramètre [age]
- lignes 18-19 : les deux valeurs précédentes sont incluses dans la page

On pourra remarquer que dans le projet Eclipse, il n'y a pas de code Java. On se contente de ce que sait faire [ParameterizableViewController].

Pour les tests, nous demandons l'url [http://localhost:8080/spring-mvc-32/accueil.html] :



ou encore l'url [http://localhost:8080/spring-mvc-32/accueil.html?nom=paul&age=55] :



2.2 Le contrôleur UrlFileNameViewController

Revenons sur 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

Parmi les classes implémentant cette interface, on voit ci-dessus, la classe [UrlFileNameViewController]. Ce contrôleur est très semblable au contrôleur [ParameterizableViewController] si ce n'est que le nom de la vue à afficher est extrait du dernier élément de l'url demandée, plutôt que défini dans un fichier de configuration. Ainsi si l'URL demandée est de la forme [http://machine:port/elmt1/elmt2/.../**elmtn**.xxx], le nom de la vue sera **elmtn**. A ce nom, peuvent être ajoutés un préfixe et un suffixe qui, eux, sont définis par configuration.

La classe [UrlFileNameViewController] est définie comme suit :

Constructor Summary

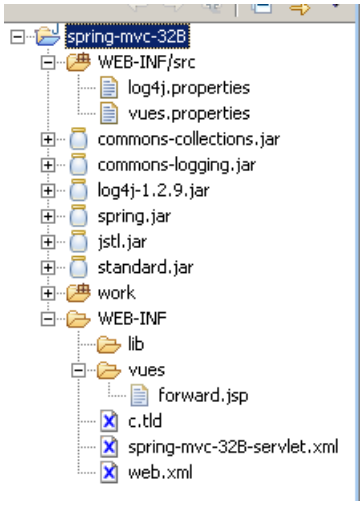
`UrlFilenameViewController()`

Method Summary

protected String	`extractViewNameFromUrlPath(String uri)` Extract the URL filename from the given request URI.
protected String	`getPrefix()` Return the prefix to prepend to the request URL filename.
protected String	`getSuffix()` Return the suffix to append to the request URL filename.
protected String	`getViewNameForUrlPath(String urlPath)` Returns view name based on the URL filename, with prefix/suffix applied when appropriate.
protected String	`postProcessViewName(String viewName)` Build the full view name based on the given view name as indicated by the URL path.
void	`setPrefix(String prefix)` Set the prefix to prepend to the request URL filename to build a view name.
void	`setSuffix(String suffix)` Set the suffix to append to the request URL filename to build a view name.

- les méthodes [setPrefix] et [setSuffix] permettent de définir le préfixe et le suffixe à ajouter aux noms des vues. On notera que ces deux méthodes sont apparues tardivement dans les distributions de Spring. Elles n'existaient par exemple pas dans la distribution 1.2.4 que j'utilisais jusqu'à maintenant dans ces articles. Dans la suite de l'article, j'utilise désormais la version 1.2.7.

Pour illustrer l'usage de cette classe, nous utiliserons le projet Eclipse / Tomcat (spring-mvc-32B) suivant :



Le fichier de configuration Spring (spring-mvc-32B-servlet.xml) de l'application 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="urlMap">
7.        <map>
8.          <entry key="/forward.html">
9.            <ref local="urlFileNameViewController"/>
10.         </entry>
11.       </map>
12.     </property>
13.   </bean>
14.   <!-- le résolveur de vues -->
15.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
16.     <property name="basename">
17.       <value>vues</value>
18.     </property>
19.   </bean>
20.   <!-- les contrôleurs de l'application-->
21.   <bean id="urlFileNameViewController"
22.     class="org.springframework.web.servlet.mvc.UrlFilenameViewController">
23.     <property name="prefix">
24.       <value>-</value>
25.     </property>
26.     <property name="suffix">
27.       <value>-</value>
28.     </property>
29.   </bean>
30. </beans>
```

- ligne 8 : seule l'URL [/forward.html] est acceptée
- ligne 9 : elle est traitée par le contrôleur d'id [urlFileNameViewController]. Ce contrôleur est défini lignes 21-29.
- lignes 21-22 : le contrôleur d'id [urlFileNameViewController] est de type [UrlFilenameViewController] (ligne 22).
- ligne 23 : le signe - sera préfixé au nom de la vue
- ligne 27 : le signe - sera suffixé au nom de la vue
- des lignes précédentes, il ressort que la demande de l'url [/forward.html] va entraîner l'affichage de la vue portant le nom [-forward-].

Les vues sont définies dans le fichier [WEB-INF/src/vues.properties] suivant :

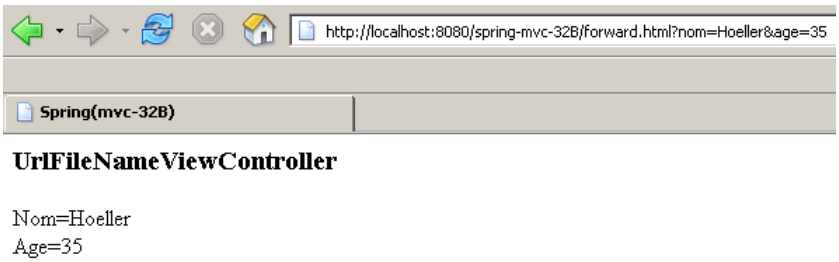
```
1.  #-forward-
2.  -forward-.class=org.springframework.web.servlet.view.JstlView
3.  -forward-.url=/WEB-INF/vues/forward.jsp
```

On y définit l'unique vue de nom [-forward-]. C'est au final, la page JSP [/WEB-INF/vues/forward.jsp] qui sera affichée. C'est la même que dans l'exemple précédent à quelques détails près :

```
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.  <%
6.    // on récupère les paramètres de la requête
7.    String nom=request.getParameter("nom");
```

```
8.    if(nom==null) nom="inconnu";
9.    String age=request.getParameter("age");
10.   if(age==null) age="inconnu";
11. %>
12. <html>
13.   <head>
14.     <title>Spring(mvc-32B)</title>
15.   </head>
16.   <body>
17.     <h3>UrlFileNameViewController</h3>
18.     Nom=<%=nom%><br>
19.     Age=<%=age%><br>
20.   </body>
21. </html>
```

Pour tester, nous demandons l'url [http://localhost:8080/spring-mvc-32B/forward.html?nom=Hoeller&age=35] :



2.3 Le contrôleur MultiActionController

2.3.1 Présentation

Revenons sur 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

Parmi les classes implémentant cette interface, on voit ci-dessus, la classe [MultiActionController]. Cette classe permet de traiter des requêtes différentes avec un unique contrôleur. Qu'entend-on par requêtes différentes ? Une requête est de la forme [/url?param1=val1&param2=val2&....]. [MultiActionController] distingue deux requêtes soit par la composante [url], soit par l'un des paramètres [parami] qui sert de marqueur de requête.

La classe [MultiActionController] est une classe abstraite destinée à être dérivée. Sa lignée est la suivante :

org.springframework.web.servlet.mvc.multiaction

Class MultiActionController

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.multiaction.MultiActionController**

[MultiActionController] dérive de la classe abstraite [AbstractController], classe parent d'un certain nombre d'autres contrôleurs prédéfinis de Spring MVC. Pour qu'elle soit vraiment utile, il faut la dériver. Pour faciliter l'explication qui va suivre, appelons [MyMultiActionController], une classe dérivée de [MultiActionController].

Deux requêtes différentes [req1] et [req2] seront traitées par deux méthodes différentes du contrôleur [MyMultiActionController]. Celles-ci ont la signature suivante :

```
ModelAndView actionName(HttpServletRequest request, HttpServletResponse response);
```

On retrouve une situation connue. La méthode ci-dessus doit exploiter la requête [request] et renvoyer à [DispatchServlet] un [ModelAndView] à afficher. En fait, les méthodes [actionName] peuvent avoir deux autres signatures. Nous renvoyons le lecteur au Javadoc de la classe [MultiActionController].

Les requêtes traitées par [MyMultiActionController] vont d'abord l'être par des méthodes de la classe parent [MultiActionController]. Parmi celles-ci, on trouve la suivante :

protected ModelAndView	**invokeNamedMethod**(String methodName, HttpServletRequest request, HttpServletResponse response) Invokes the named method.

Si la valeur du 1er paramètre [methodName] est **XX**, la classe [MultiActionController] va appeler la méthode suivante de sa classe dérivée [MyMultiActionController] :

```
ModelAndView XX(HttpServletRequest request, HttpServletResponse response);
```

Il nous reste à savoir, comment [MultiActionController] connaît le nom de la méthode qui doit traiter la requête particulière [req1], par exemple. Elle utilise pour cela, un résolveur de noms de méthodes, c.a.d. une classe qui sait associer une requête reçue à la méthode qui doit la traiter. Une méthode de [MultiActionController] permet de fixer le résolveur de noms de méthodes à utiliser :

void	**setMethodNameResolver**(MethodNameResolver methodNameResolver) Set the method name resolver that this class should use.

Comme on peut s'y attendre avec Spring, [MethodNameResolver] est une interface :

org.springframework.web.servlet.mvc.multiaction

Interface MethodNameResolver

All Known Implementing Classes:

AbstractUrlMethodNameResolver, InternalPathMethodNameResolver, ParameterMethodNameResolver, PropertiesMethodNameResolver

L'interface [MethodNameResolver] n'a qu'une méthode :

Method Summary

String	**getHandlerMethodName**(HttpServletRequest request) Return a method name that can handle this request.

La méthode [getHandlerMethodName] rend le nom de la méthode de [MultiActionController] qui doit être appelée pour traiter la requête [request]. Ceci connu, [MultiActionController] va appeler cette méthode qui sera trouvée dans sa classe dérivée [MyMultiActionController].

Il y a plusieurs stratégies pour associer une requête [request] à un nom de méthode. On voit ci-dessus, que Spring propose quatre implémentations de l'interface [MethodNameResolver] :

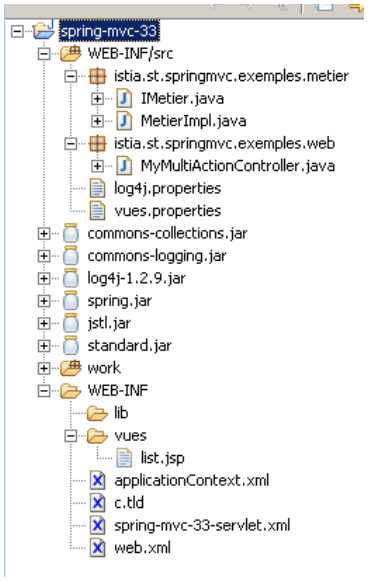
- [**AbstractUrlMethodNameResolver**] : classe abstraite. Sa méthode [getHandlerMethodName] fait appel à la méthode abstraite [String getHandlerMethodNameForUrlPath(String urlPath)] qui doit être implémentée dans une classe dérivée. Cette méthode reçoit le chemin de l'URL de la requête qui doit être traitée. A partir de cette information, elle doit fournir le nom de la méthode du [MultiActionController] chargée de traiter la requête.
- [**InternalPathMethodNameResolver**] : implémentation par défaut utilisée par [MultiActionController]. Une requête d'url [/elmt1/elmt2/.../**elmntN**.extension] est traitée par la méthode du [MultiActionController] nommée [**elmntN**].
- [**ParameterMethodNameResolver**] : cette implémentation supporte deux statégies de résolution de noms de méthodes :
    1. un paramètre particulier de l'url de la requête fixe le nom de la méthode du [MultiActionController] à utiliser. Par défaut, ce paramètre est **action**. Ainsi l'url **/url?action=XX&param1=val1&...** sera traitée par la méthode **XX**. La méthode [**setParamName**] permet de fixer le nom du paramètre qui définit la méthode du [MultiActionController] à utiliser si on veut un autre nom que [action].
    1. le nom du paramètre est lui-même le nom de la méthode à utiliser. Sa valeur est ignorée. Comme il peut y avoir d'autres paramètres dans la requête, la méthode [**setParamNames**] sert à fixer les paramètres qui sont des noms de méthodes. Ainsi un formulaire ayant trois boutons de type Submit appelés [submit1, submit2, submit3] pourra être traité par les méthodes : [submit1, submit2, submit3]. La méthode qui traitera la

requête dépendra du bouton utilisé pour faire le Submit du formulaire. On utilisera la méthode [setParamNames] pour indiquer que la présence d'un des paramètres [submit1, submit2, submit3] dans la requête implique que celle-ci doit être traitée par la méthode du [MultiActionController] portant le nom de ce paramètre.

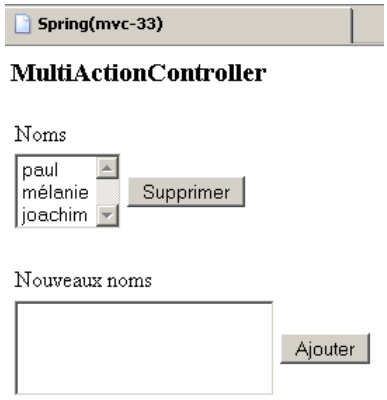
- [**PropertiesMethodNameResolver**] : l'association [requête] <-> [méthode] est faite par un objet [Properties] à l'aide de la méthode [setMappings(Properties props)].

2.3.2 Exemple 1

Pour illustrer l'usage de la classe [MultiActionController], nous utiliserons le projet Eclipse / Tomcat (spring-mvc-33) suivant :



L'application (spring-mvc-33) n'a qu'une vue :



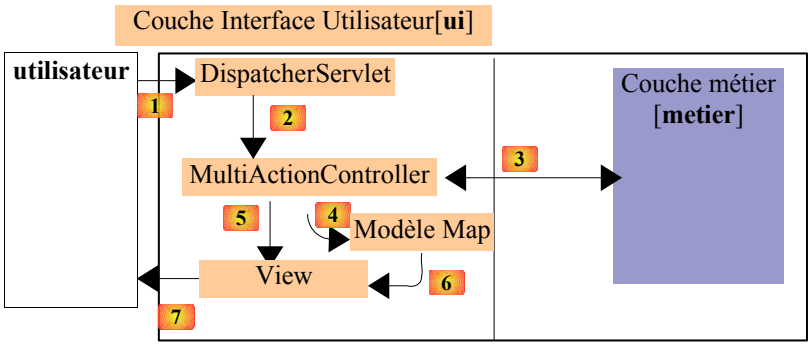
Il s'agit de gérer une liste de noms. Trois actions seront possibles :

- lister tous les noms
- supprimer de noms
- ajouter de noms

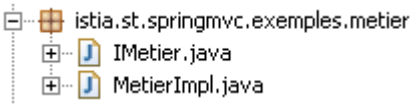
Ces trois actions seront traitées par un unique contrôleur, d'où l'usage de la classe [MultiActionController].

Pour être plus réaliste, nous supposerons que la liste des noms est gérée par une couche [métier]. L'architecture de l'aplication est la suivante :

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com



La couche [métier] est dans le paquetage [istia.st.springmvc.exemples.metier] du projet :



[IMetier] est l'interface que la couche [métier] présente à la couche [web] :

```
1.  package istia.st.springmvc.exemples.metier;
2.
3.  import java.util.List;
4.
5.  public interface IMetier {
6.    // liste de tous les noms
7.    public List getAll();
8.    // ajouter de nouveaux noms
9.    public void add(List noms);
10.   // supprimer certains noms
11.   public void delete(List noms);
12. }
```

- ligne 7 : la méthode [getAll] permet d'avoir les noms sous la forme d'un objet de type List
- ligne 9 : la méthode [add] permet d'ajouter à la liste actuelle des noms, une liste de nouveaux noms
- ligne 11 : la méthode [delete] permet de supprimer certains noms de la liste actuelle des noms

[MetierImpl] est une classe d'implémentation utilisant un objet [ArrayList] pour stocker les noms :

```
1.  package istia.st.springmvc.exemples.metier;
2.
3.  import java.util.ArrayList;
4.  import java.util.List;
5.
6.  public class MetierImpl implements IMetier {
7.
8.    // une liste de noms
9.    private ArrayList noms;
10.
11.   public ArrayList getNoms() {
12.     return noms;
13.   }
14.
15.   public void setNoms(ArrayList noms) {
16.     this.noms = noms;
17.   }
18.
19.   // ajouter de nouveaux noms
20.   public synchronized void add(List nouveaux) {
21.     this.noms.addAll(nouveaux);
22.   }
23.
24.   // supprimer certains noms
25.   public synchronized void delete(List nomsAEnlever) {
26.     this.noms.removeAll(nomsAEnlever);
27.   }
28.
29.   // liste de tous les noms
30.   public synchronized List getAll() {
31.     return this.noms;
32.   }
33.
34. }
```

- lignes 9-17 : la liste des noms avec ses méthodes *get* et *set*. La méthode *set* nous permettra d'initialiser la liste des noms par configuration Spring.
- parce que la classe [MetierImpl] va être instanciée en un unique exemplaire (singleton) qui sera partagé entre tous les utilisateurs de l'application, il nous faut gérer les éventuels conflits d'accès à l'objet [ArrayList noms]. Pour cette raison, nous avons synchronisé toutes les méthodes, assurant par là qu'un seul thread à la fois peut opérer sur la liste des noms.

La couche [métier] définie, nous passons à la couche web. Tout d'abord, examinons la configuration de l'application (spring-mvc-33).

On voit que dans le projet (paragraphe 2.3.2, page 10), il y a un fichier [applicationContext.xml]. Il va nous servir à définir le contexte de l'application et notamment l'implémentation de la couche [métier] à utiliser. Pour que ce fichier soit exploité, il faut que [web.xml] charge le listener [org.springframework.web.context.ContextLoaderListener] (lignes 8-11 ci-dessous) :

```
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-33</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-33</servlet-name>
21.     <url-pattern>*.html</url-pattern>
22.   </servlet-mapping>
23. </web-app>
```

Le contexte de l'application est défini comme suit 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.    <!-- couche métier -->
5.    <bean id="metier" class="istia.st.springmvc.exemples.metier.MetierImpl">
6.      <property name="noms">
7.        <list>
8.          <value>paul</value>
9.          <value>mélanie</value>
10.         <value>joachim</value>
11.       </list>
12.     </property>
13.   </bean>
14. </beans>
```

- lignes 5-13 : définit le singleton de la couche [métier] avec un bean d'id " metier "
- ligne 5 : la classe d'implémentation de la couche [métier]
- lignes 6-12 : initialisation de l'attribut [noms] de cette classe. La méthode [setNoms] de la classe est ici implicitement utilisée.

Le fichier [spring-mvc-33-servlet.xml] configure la couche web de l'application :

```
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="multiactions.html">MultiActionsController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- le résolveur de vues -->
13.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
14.     <property name="basename">
15.       <value>vues</value>
16.     </property>
```

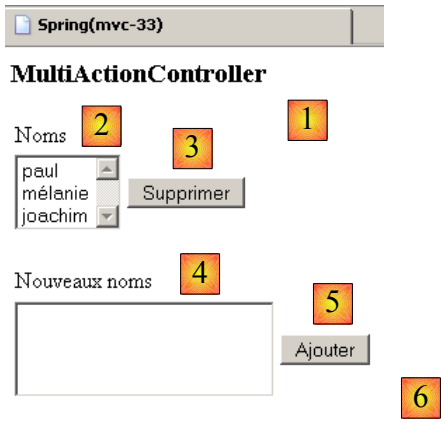
```
17.  </bean>
18.  <!-- les contrôleurs de l'application-->
19.  <bean id="MultiActionsController"
20.    class="istia.st.springmvc.exemples.web.MyMultiActionController">
21.      <property name="methodNameResolver">
22.        <ref local="myMethodNameResolver"/>
23.      </property>
24.      <property name="metier">
25.        <ref bean="metier"/>
26.      </property>
27.  </bean>
28.  <!-- le résolveur des noms de méthodes -->
29.  <bean id="myMethodNameResolver"
    class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver">
30.    <property name="defaultMethodName"><value>list</value></property>
31.    <property name="paramName"><value>action</value></property>
32.  </bean>
33. </beans>
```

- ligne 8 : l'unique url acceptée est [/multiactions.html]. Elle est traitée par le contrôleur d'id [MultiActionController] défini lignes 19-27.
- ligne 20 : le contrôleur d'id [MultiActionsController] est de type [MyMultiActionsControler]. Nous verrons ultérieurement que ce type est dérivé du contrôleur [MultiActionController].
- lignes 21-23 : définissent le résolveur de noms de méthodes du contrôleur [MultiActionController]. Ce résolveur d'id "*myMethodNameResolver*" est défini lignes 29-32.
- lignes 24-26 : la référence de la couche métier est injectée dans le contrôleur
- lignes 29-32 : définissent le résolveur de noms de méthodes d'id "*myMethodNameResolver*"
- ligne 29 : il est de type [ParameterMethodNameResolver]. Nous avons vu que ce résolveur permettait deux stratégies de résolution de méthodes (cf page 9). Nous utilisons celle où le nom de la méthode à utiliser est fixé par la valeur d'un paramètre particulier de la requête.
- ligne 31 : ce paramètre est [action]. C'est en fait la valeur par défaut qui est utilisée lorsqu'aucun nom de paramètre n'est défini. Nous pouvions donc ne pas définir la propriété [paramName] de [ParameterMethodNameResolver].
- ligne 30 : la propriété [defaultMethodName] de [ParameterMethodNameResolver] permet d'indiquer quelle méthode exécuter lorsque le paramètre [action] n'est pas trouvé dans la requête. Ici, ce sera la méthode [list].

- lignes 13-17 : le résolveur des noms de vues, ici [ResourceBundleViewResolver]. Les vues sont définies dans le fichier [WEB-INF/src/vues.properties] :

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

Il n'y a qu'un nom de vue : **list**, associé à la page JSP [/WEB-INF/vues/list.jsp]. L'aspect visuel de cette vue a déjà été présenté :



Les composants du formulaire HTML sous-jacent sont les suivants :

N°	Type HTML	Nom	Rôle
1	<form>	frmNoms	formulaire
2	<select multiple>	noms	liste actuelle des noms
3	<input type= "submit ">		pour supprimer les noms sélectionnés dans (2)
4	<textarea>	nouveaux	liste de noms à ajouter, à raison d'un nom par

			ligne
5	<input type= "submit ">		pour ajouter dans (2) les noms de (4)
6	<input type= "hidden ">	action	sert à préciser l'action à faire.

Le code JSP de la page 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>Spring(mvc-33)</title>
8.      <!-- Javascript -->
9.      <script language="javascript">
10.       function submitAction(uneAction){
11.         with(document.frmNoms){
12.           action.value=uneAction;
13.           submit();
14.         }
15.       }
16.     </script>
17.   </head>
18.   <!-- HTML -->
19.   <body>
20.     <h3>MultiActionController</h3>
21.     <form name="frmNoms" method="post">
22.       <table border="0">
23.         <tr>
24.           <td>Noms</td>
25.         </tr>
26.         <tr>
27.           <td>
28.             <select name="noms" multiple>
29.               <c:forEach items="${noms}" var="nom">
30.                 <option>${nom}</option>
31.               </c:forEach>
32.             </select>
33.           </td>
34.           <td>
35.             <input type="button" value="Supprimer" onclick='submitAction("delete");'>
36.           </td>
37.         </tr>
38.       </table>
39.       <br>
40.       <table border="0">
41.         <tr>
42.           <td>Nouveaux noms</td>
43.         </tr>
44.         <tr>
45.           <td>
46.             <textarea name="nouveaux" rows="3"></textarea>
47.           </td>
48.           <td>
49.             <input type="button" value="Ajouter" onclick='submitAction("add");'>
50.           </td>
51.         </tr>
52.       </table>
53.       <input type="hidden" name="action">
54.     </form>
55.   </body>
56. </html>
```

- lignes 10-15 : un script javascript qui donne une valeur au champ caché [**action**] de la ligne 53 puis poste le formulaire. Ce script est appelé ligne 35 lorsque l'utilisateur appuie sur le bouton [Supprimer]. On voit qu'alors, action=**delete**. Il est appelé également ligne 49 lorsque l'utilisateur appuie sur le bouton [Ajouter]. On voit qu'alors, action=**add**.
- le POST du formulaire se fera donc avec un paramètre **action** prenant ses valeurs dans [**add**,**delete**]. On sait que ce paramètre **action** sera utilisé par le résolveur de noms de méthodes pour déterminer quelle méthode du contrôleur traitera la requête. Lors de l'appel initial du formulaire via l'url [/multiactions.html], le paramètre [action] sera absent. C'est alors la propriété [defaultMethodName] du résolveur de noms de méthodes qui sera utilisée pour déterminer quelle méthode du contrôleur traitera la requête. Par configuration, (ligne 30 de spring-mvc-33-servlet.xml), ce sera la méthode **list**. Notre contrôleur [MultiActionController] devra donc avoir les méthodes [**add**, **delete**, **list**].

- lignes 29-31 : les options de la liste sont trouvées dans un attribut **noms** qui sera cherché dans les différents contextes de l'application. Le contrôleur [MultiActionController] devra donc placer cet attribut dans le modèle de la vue.

Nous avons désormais les éléments pour comprendre le code du contrôleur [MyMultiActionController] :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Arrays;
4.  import java.util.HashMap;
5.  import java.util.List;
6.  import java.util.Map;
7.
8.  import istia.st.springmvc.exemples.metier.IMetier;
9.
10. import javax.servlet.http.HttpServletRequest;
11. import javax.servlet.http.HttpServletResponse;
12.
13. import org.springframework.web.servlet.ModelAndView;
14. import org.springframework.web.servlet.mvc.multiaction.MultiActionController;
15.
16. public class MyMultiActionController extends MultiActionController {
17.
18.  // couche métier
19.  IMetier metier;
20.
21.  public IMetier getMetier() {
22.    return metier;
23.  }
24.
25.  public void setMetier(IMetier metier) {
26.    this.metier = metier;
27.  }
28.
29.  // liste des noms
30.  public ModelAndView list(HttpServletRequest request,
31.      HttpServletResponse response) {
32.    // on demande les noms à la couche [métier]
33.    List noms = metier.getAll();
34.    // on les met dans le modèle de la vue
35.    Map modèle = new HashMap();
36.    modèle.put("noms", noms);
37.    // on rend le nom de la vue et son modèle
38.    return new ModelAndView("list", modèle);
39.  }
40.
41.  // supprimer des noms
42.  public ModelAndView delete(HttpServletRequest request,
43.      HttpServletResponse response) {
44.    // liste des noms sélectionné dans la liste
45.    String[] noms = request.getParameterValues("noms");
46.    // on demande à la couche [métier] de les supprimer de la liste existante
47.    metier.delete(Arrays.asList(noms));
48.    // on rend le nom de la vue et son modèle
49.    return list(request,response);
50.  }
51.
52.  // ajouter des noms
53.  public ModelAndView add(HttpServletRequest request,
54.      HttpServletResponse response) {
55.    // chaîne des noms à ajouter
56.    String nouveaux = request.getParameter("nouveaux");
57.    // ces noms sont mis dans un tableau
58.    String[] nouveauxNoms = nouveaux.split("\r\n");
59.    // on crée la liste des noms à ajouter
60.    ArrayList alNouveauxNoms = new ArrayList();
61.    String nouveauNom;
62.    for (int i = 0; i < nouveauxNoms.length; i++) {
63.      // on ne garde le nom que s'il est non vide
64.      nouveauNom = nouveauxNoms[i].trim();
65.      if (nouveauNom.length() != 0) {
66.        alNouveauxNoms.add(nouveauNom);
67.      }
68.    }
69.    // on demande à la couche [métier] de les ajouter à la liste existante
70.    if (alNouveauxNoms.size() != 0) {
71.      metier.add(alNouveauxNoms);
72.    }
73.    // on rend le nom de la vue et son modèle
74.    return list(request, response);
75.  }
76. }
```

- ligne 16 : le contrôleur [MyMultiActionController] dérive de la classe [MultiActionController]

- lignes 19-27 : référence sur la couche métier de type IMetier. On notera qu'on utilise l'interface et non le type d'une classe d'implémentation particulière. Nous avons vu que le champ [metier] était initialisé par (spring-mvc-33-servlet.xml).
- lignes 30-39 : la méthode **list** qui traite l'action **list** ou l'absence d'action
- lignes 42-50 : la méthode **delete** qui traite l'action **delete**
- lignes 53-63 : la méthode **add** qui traite l'action **add**

Commentons tout d'abord la méthode [list]. Elle est appelée notamment lorsque l'utilisateur appelle directement l'url [/multiactions.html]. Comme le paramètre [action] n'est pas présent, c'est la méthode par défaut [list] qui est chargée de traiter la requête. Il s'agit de faire afficher la page [list.jsp] avec la liste des noms. De ce qu'on a vu précédemment lors de l'étude de [list.jsp], la méthode doit construire un [ModelAndView] avec :

- pour nom de vue : **list** (cf vue.properties)
- dans le modèle, la liste des noms associée à un attribut nommé "**noms**".

- ligne 33 : la liste des noms est demandée à la couche [métier]
- ligne 35 : un modèle vide est construit
- ligne 36 : la liste des noms est placée dans le modèle, associée à un attribut nommé "noms"
- ligne 38 : la méthode rend un [ModelAndView] avec " list " comme nom de vue et le modèle qui vient d'être construit.

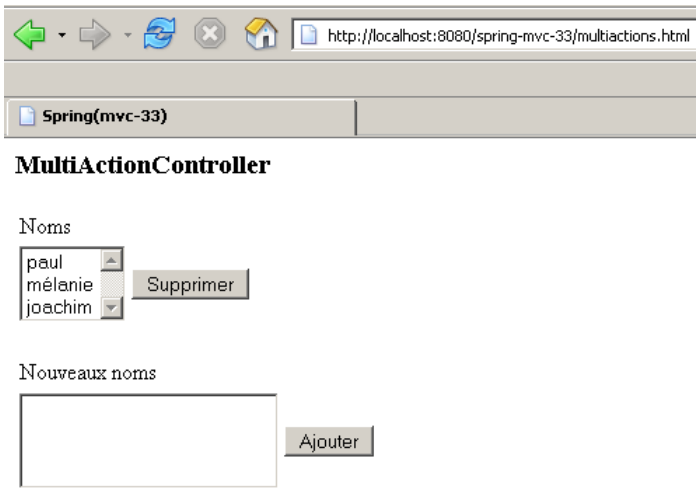
La méthode [delete] (lignes 42-50) est appelée lorsque l'utilisateur a appuyé sur le bouton [Supprimer]. Il faut donc supprimer de la liste des noms, les noms sélectionnés dans la liste HTML " noms " :

- ligne 45 : on récupère dans un tableau les noms sélectionnés dans la liste HTML appelée "noms"
- ligne 47 : on demande à la couche [métier] de les supprimer
- ligne 49 : on veut réafficher la liste mise à jour. Pour cela, on se contente de faire appel à la méthode [list] qui sait faire cela.

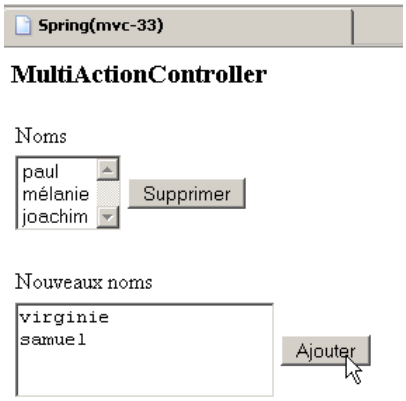
La méthode [add] (lignes 53-74) est appelée lorsque l'utilisateur a appuyé sur le bouton [Ajouter]. Il faut alors ajouter à la liste des noms, ceux qui ont été placés dans le <textarea> HTML appelé "nouveaux" :

- ligne 56 : on récupère dans une chaîne le contenu du <textarea> HTML appelé "nouveaux"
- la valeur d'un <textarea> est une chaîne de type " ligne1\r\nligne2\r\nligne3... ", où lignei est une ligne de texte. Ligne 58, on récupère les différentes lignes dans un tableau.
- ligne 60 : on crée une liste vide dans laquelle on mettra les noms à ajouter.
- lignes 62-67 : on débarrasse les noms des éventuels espaces qui peuvent les précéder ou les suivre et on ne garde le nom que s'il est non vide.
- lignes 70-72 : on demande à la couche [métier] d'ajouter les nouveaux noms à la liste actuelle
- ligne 74 : on veut réafficher la liste mise à jour. Pour cela, on se contente de faire appel à la méthode [list].

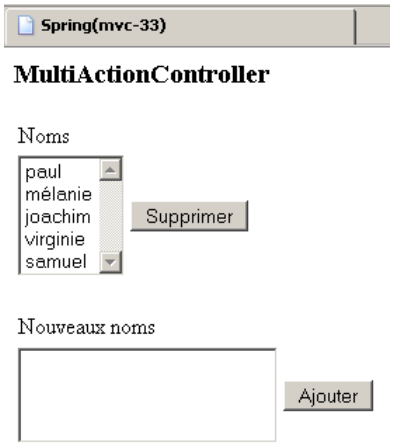
Voici un exemple d'exécution. On demande tout d'abord l'url [http://localhost:8080/spring-mvc-33/multiactions.html] :



On ajoute des noms :

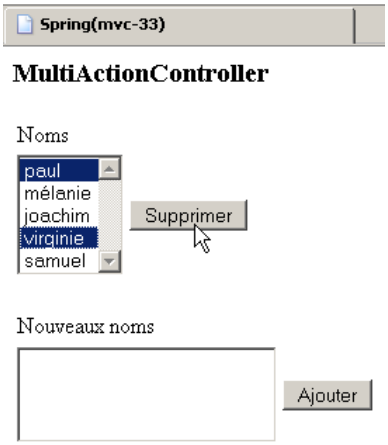


requête

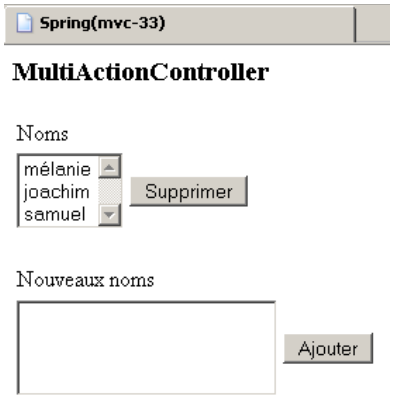


réponse

On supprime des noms :



requête



réponse

2.3.3 Exemple 2 (spring-mvc-33B)

Nous reprenons l'exemple précédent dans un nouveau projet Eclipse (spring-mvc-33B). Celui-ci est identique au projet (spring-mvc-33) mais nous utilisons une autre stratégie de résolution de noms de méthodes. La page [list.jsp] est modifiée comme suit :

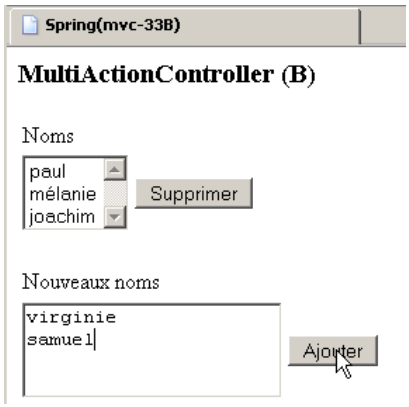
```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring(mvc-33B)</title>
8.    </head>
9.    <body>
10.     <h3>MultiActionController (B)</h3>
11.     <form method="post">
12. ...
13.           <input type="submit" name="delete" value="Supprimer">
14. ...
15.           <input type="submit" name="add" value="Ajouter">
16. ...
17.     </form>
18.   </body>
19. </html>
```

- le code Javascript a disparu ainsi que le champ caché nommé " **action** ".

- les boutons [Supprimer] et [Ajouter] qui étaient des composants HTML de type [Button] deviennent des composants HTML de type [Submit]. Alors que précédemment, ils n'avaient pas de noms, ils en ont désormais un, respectivement [**delete**] et [**add**].

La conséquence principale de ces changements est dans les paramètres postés au serveur :
- il n'y a plus de paramètre [action] posté
- lorsque le formulaire est posté via le bouton [Supprimer], le paramètre [delete] sera posté avec la valeur [Supprimer]
- lorsque le formulaire est posté via le bouton [Ajouter], le paramètre [add] sera posté avec la valeur [Ajouter]

C'est ce que montrent l'outil [LiveHttpHeaders] déjà utilisé dans les articles précédents :

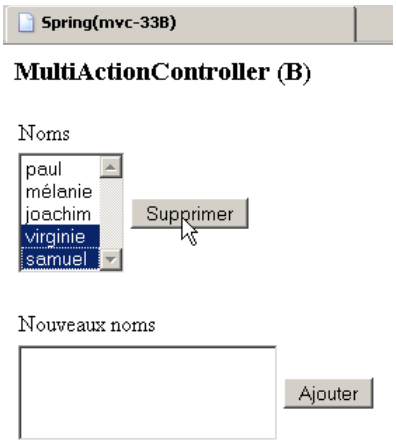


demande

```
1.  POST /spring-mvc-33B/multiactions.html HTTP/1.1
2.  Host: localhost:8080
3.  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US;
    rv:1.8.0.1) Gecko/20060111 Firefox/1.5.0.1
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-
    33B/multiactions.html
11. Cookie: JSESSIONID=A83C308074C7E21EA2304993F2030A3D
12. Content-Type: application/x-www-form-urlencoded
13. Content-Length: 41
14.
15. nouveaux=virginie%0D%0Asamuel&add=Ajouter
```

flux HTTP envoyé par le navigateur client

- ci-dessus, ligne 15, le paramètre **add** est posté.



demande

```
1.  POST /spring-mvc-33B/multiactions.html HTTP/1.1
2.  Host: localhost:8080
3.  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; en-US;
    rv:1.8.0.1) Gecko/20060111 Firefox/1.5.0.1
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-
    33B/multiactions.html
11. Cookie: JSESSIONID=A83C308074C7E21EA2304993F2030A3D
12. Content-Type: application/x-www-form-urlencoded
13. Content-Length: 52
14.
15. noms=virginie&noms=samuel&delete=Supprimer&nouveaux=
```

flux HTTP envoyé par le navigateur client

- ci-dessus, ligne 15, le paramètre **delete** est posté.

L'unique URL traitée par l'application reste [/multiactions.html]. On a trois cas :
- l'url ne contient aucun des paramètres [delete,add] : on veut que la méthode [list] du contrôleur [MultiActionController] soit exécutée.
- l'url contient le paramètre [delete] ou [add] : on veut que la méthode [delete] ou [add] du contrôleur [MultiActionController] soit exécutée.

Ce fonctionnement est obtenu en utilisant le résolveur de noms de méthodes suivant (spring-mvc-33-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.  ....
5.    <!-- les contrôleurs de l'application-->
6.    <bean id="MultiActionsController"
7.      class="istia.st.springmvc.exemples.web.MyMultiActionController">
8.        <property name="methodNameResolver">
9.          <ref local="myMethodNameResolver"/>
10.       </property>
11.       <property name="metier">
12.         <ref bean="metier"/>
13.       </property>
14.   </bean>
15.   <!-- le résolveur des noms de méthodes -->
16.   <bean id="myMethodNameResolver"
    class="org.springframework.web.servlet.mvc.multiaction.ParameterMethodNameResolver">
17.     <property name="defaultMethodName"><value>list</value></property>
18.     <property name="methodParamNames">
19.       <list>
20.         <value>delete</value>
21.         <value>add</value>
22.       </list>
23.     </property>
24.   </bean>
25. </beans>
```

- ligne 16 : le résolveur de noms de méthodes est comme dans l'exemple précédent [ParameterMethodNameResolver]
- ligne 17 : lorsque le résolveur de noms de méthodes ne réussit pas à lier une requête à une méthode, c'est la méthode [list] qui sera utilisée.
- lignes 18-23 : définissent les paramètres qui, lorsque leur seule présence est détectée dans la requête, provoque l'exécution de la méthode qui porte leur nom.

Le lecteur est invité à tester cette nouvelle version.

2.3.4 Exemple 3 (spring-mvc-33C)

Nous reprenons le même exemple dans un nouveau projet Eclipse (spring-mvc-33C). Nous utilisons une autre stratégie de résolution de noms de méthodes (spring-mvc-33-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="/list.html">MultiActionsController</prop>
9.          <prop key="/delete.html">MultiActionsController</prop>
10.         <prop key="/add.html">MultiActionsController</prop>
11.       </props>
12.     </property>
13.   </bean>
14.   <!-- le résolveur de vues -->
15.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
16.     <property name="basename">
17.       <value>vues</value>
18.     </property>
19.   </bean>
20.   <!-- les contrôleurs de l'application-->
21.   <bean id="MultiActionsController"
22.     class="istia.st.springmvc.exemples.web.MyMultiActionController">
23.       <property name="metier">
24.         <ref bean="metier"/>
25.       </property>
26.   </bean>
27. </beans>
```

- lignes 8-10 : l'application admet désormais trois URL
- lignes 21-26 : le contrôleur est toujours le même que précédemment mais aucun résolveur de noms de méthodes n'est défini. Dans ce cas, c'est le résolveur [InternalPathMethodNameResolver] qui est utilisé. Avec ce résolveur, une URL [/M.html] est traitée par la méthode [M]. Ainsi, l'url [/list.html] sera traitée par la méthode [list], l'url [/delete.html] par la méthode [delete], l'url [/add.html] par la méthode [add].

La page [list.jsp] est transformée comme suit :

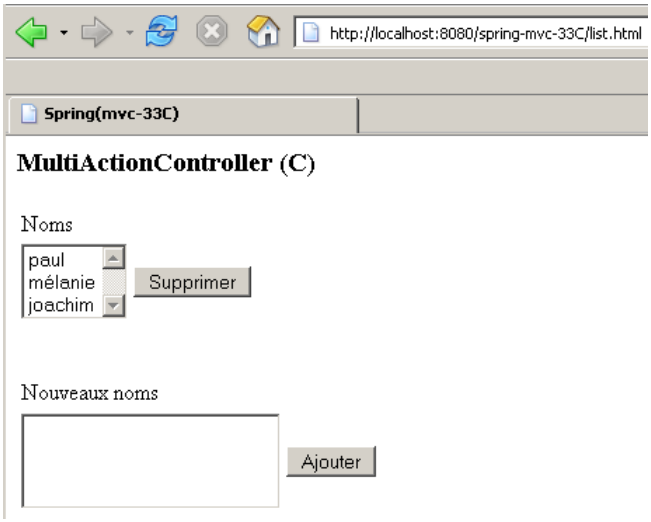
www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

```
1.  ...
2.  <html>
3.    <head>
4.      <title>Spring(mvc-33C)</title>
5.    </head>
6.    <body>
7.      <h3>MultiActionController (C)</h3>
8.      <form method="post" action="<c:url value="/delete.html"/>">
9.  ...
10.           <select name="noms" multiple>
11. ...
12.           </select>
13. ...
14.           <input type="submit" name="delete" value="Supprimer">
15. ...
16.     </form>
17.       <br>
18.     <form method="post" action="<c:url value="/add.html"/>">
19. ...
20.           <textarea name="nouveaux" rows="3"></textarea>
21. ...
22.           <input type="submit" name="add" value="Ajouter">
23. ...
24.     </form>
25.   </body>
26. </html>
```

La page comporte désormais deux formulaires :

- le 1er formulaire est défini lignes 8-16 et sert à poster les noms à supprimer
- ligne 8 : les paramètres seront postés à l'url [/delete.html]. Comme nous l'avons vu, cela va entraîner qu'ils seront traités par la méthode [delete] du contrôleur.
- le second formulaire est défini lignes 18-24 et sert à poster les noms à ajouter
- ligne 18 : les paramètres seront postés à l'url [/add.html]. Comme nous l'avons vu, cela va entraîner qu'ils seront traités par la méthode [add] du contrôleur.

Pour demander la liste initiale des noms, il faut demander l'url [/list.html] comme montré ci-dessous :



Le lecteur est invité à tester cette nouvelle version.

2.3.5 Exemple 4 (spring-mvc-33D)

Nous reprenons le même exemple dans un nouveau projet Eclipse (spring-mvc-33D). Nous utilisons une autre stratégie de résolution de noms de méthodes (spring-mvc-33-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="/list.html">MultiActionsController</prop>
9.          <prop key="/delete.html">MultiActionsController</prop>
```

```
10.       <prop key="/add.html">MultiActionsController</prop>
11.     </props>
12.   </property>
13. </bean>
14. <!-- le résolveur de vues -->
15. <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
16.   <property name="basename">
17.     <value>vues</value>
18.   </property>
19. </bean>
20. <!-- les contrôleurs de l'application-->
21. <bean id="MultiActionsController"
22.   class="istia.st.springmvc.exemples.web.MyMultiActionController">
23.     <property name="methodNameResolver">
24.       <ref local="myMethodNameResolver"/>
25.     </property>
26.     <property name="metier">
27.       <ref bean="metier"/>
28.     </property>
29. </bean>
30. <!-- le résolveur des noms de méthodes -->
31. <bean id="myMethodNameResolver"
    class="org.springframework.web.servlet.mvc.multiaction.PropertiesMethodNameResolver">
32.   <property name="mappings">
33.     <props>
34.       <prop key="/list.html">list</prop>
35.       <prop key="/delete.html">delete</prop>
36.       <prop key="/add.html">add</prop>
37.     </props>
38.   </property>
39. </bean>
40. </beans>
```

- lignes 8-10 : l'application admet les mêmes trois URL que précédemment
- lignes 31-39 : le résolveur de noms de méthodes est désormais [PropertiesMethodNameResolver]. Ce résolveur permet de définir, une à une, les associations URL <-> méthode, grâce à sa propriété [mappings] (ligne 32). Lignes 34-36, nous associons à chacune des URL traitées par le contrôleur, le nom de la méthode du contrôleur qui doit la traiter.

Le lecteur est invité à tester cette nouvelle version.

2.4 Le contrôleur AbstractWizardFormController

2.4.1 Présentation

Revenons sur l'interface [Controller] :

org.springframework.web.servlet.mvc

Interface Controller

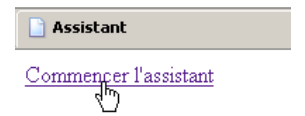
All Known Implementing Classes:

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

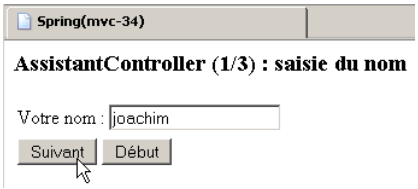
Parmi les classes implémentant cette interface, on voit ci-dessus, la classe [AbstractWizardFormController]. Cette classe permet de construire des formulaires formés de plusieurs pages. Ce type d'application est parfois appelée assistant (wizard) d'où le nom de la classe. Celle-ci est abstraite et doit donc être dérivée. La classe [AbstractWizardFormController] fournit la " mécanique " de gestion de l'assistant, la classe dérivée se contenant de traiter les parties spécifiques à l'assistant particulier que l'on construit.

Avant d'étudier la classe [AbstractWizardFormController], montrons des copies d'écran de l'application exemple que nous allons construire. L'assistant aura trois pages :

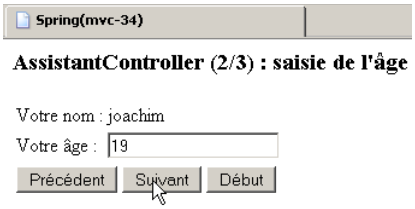
- la page n° 0 demandera un nom. Celui-ci devra être non vide pour être accepté.
- la page n° 1 demandera un âge. Celui-ci devra être un nombre entier dans l'intervalle [1,150] pour être accepté.
- la page n° 2 demandera à l'utilisateur de valider ces deux saisies.
- l'utilisateur passe d'une page à l'autre avec le bouton [Suivant].
- tant qu'il n'a pas validé la dernière page de l'assistant (page n° 2), l'utilisateur peut revenir en arrière (jusqu'à la page 0) avec le bouton [Précédent]. Il retrouve à chaque fois les valeurs saisies précédemment.
- à tout moment (pages n° 0 à 2), l'utilisateur peut annuler l'assistant avec le bouton [Début]



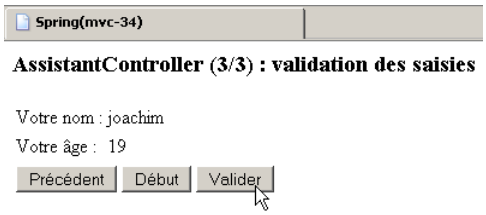
cette page ne fait pas partie de l'assistant



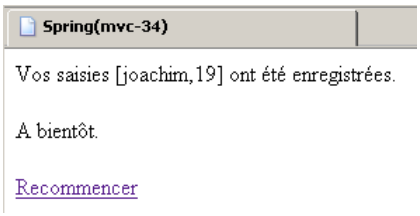
page n° 0 de nom " page1 "



page n° 1 de nom " page2 "

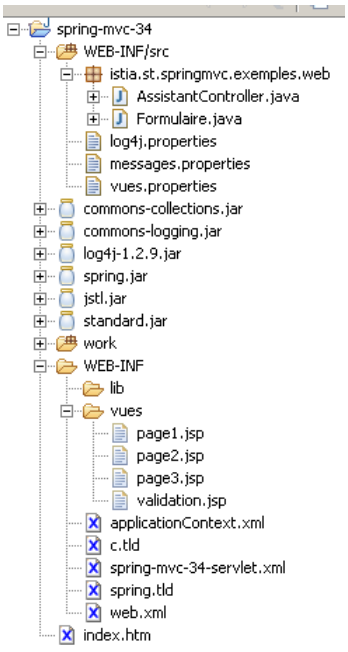


page n° 2 de nom " page3 "



cette page ne fait pas partie de l'assistant

Le projet Eclipse / Tomcat de cette application est le suivant :



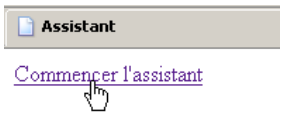
L'application est configurée par [web.xml] :

```
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-34</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-34</servlet-name>
21.     <url-pattern>*.html</url-pattern>
22.   </servlet-mapping>
23.   <!-- fichier d'index -->
24.   <welcome-file-list>
25.     <welcome-file>index.htm</welcome-file>
26.   </welcome-file-list>
27. </web-app>
```

Il y a une nouveauté par rapport aux fichiers [web.xml] des applications précédentes :

- lignes 24-26 : définissent un fichier d'accueil. Lorsque l'url [/spring-mvc-24] sera demandée, le serveur transformera l'url en [/spring-mvc-24/index.htm]. [index.htm] est placé à la racine du dossier [spring-mvc-24] (cf projet Eclipse). C'est une page HTML qui contient le lien qui va lancer l'assistant :



Les lignes 9-11 de [web.xml] charge le *listener* capable d'exploiter le contenu du fichier [applicationContext.xml]. Celui-ci 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.    <!-- le fichier des messages -->
5.    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
6.      <property name="basename">
7.        <value>messages</value>
8.      </property>
9.    </bean>
10. </beans>
```

Il définit un fichier de messages sur lequel nous aurons l'occasion de revenir.

Le fichier (spring-mvc-34-servlet.xml) qui configure l'application Spring 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="assistant.html">AssistantController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- le résolveur de vues -->
13.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
14.     <property name="basename">
15.       <value>vues</value>
16.     </property>
17.   </bean>
18.   <!-- le contrôleur de l'application-->
19.   <bean id="AssistantController"
20.     class="istia.st.springmvc.exemples.web.AssistantController">
21.     <property name="sessionForm">
22.       <value>true</value>
23.     </property>
24.     <property name="commandClass">
25.       <value>istia.st.springmvc.exemples.web.Formulaire</value>
26.     </property>
27.     <property name="commandName">
28.       <value>formulaire</value>
29.     </property>
30.     <property name="pages">
31.       <list>
32.         <value>page1</value>
33.         <value>page2</value>
34.         <value>page3</value>
35.       </list>
```

```
36.     </property>
37.     <property name="allowDirtyBack">
38.       <value>true</value>
39.     </property>
40.     <property name="allowDirtyForward">
41.       <value>false</value>
42.     </property>
43.   </bean>
44. </beans>
```

- ligne 8 : l'unique URL traitée sera [/assistant.html]. Elle le sera par le contrôleur d'id [AssistantController]. Celui-ci est défini lignes 19-43. La classe [AssistantController] dérivera de la classe [AbstractWizardFormController].

La classe [AbstractWizardFormController] a l'ascendance suivante :

La classe [AbstractWizardFormController] dérive de la classe [AbstractFormController]. C'était déjà le cas de la classe [SimpleFormController] que nous avons utilisée dans l'article précédent pour gérer des formulaires mono-page. Aussi allons-nous trouver des similarités dans la configuration de ces deux types de contrôleur. Celle du contrôleur utilisé dans l'application exemple est la suivante :

```
1.     <!-- le contrôleur de l'application-->
2.   <bean id="AssistantController"
3.     class="istia.st.springmvc.exemples.web.AssistantController">
4.     <property name="sessionForm">
5.       <value>true</value>
6.     </property>
7.     <property name="commandClass">
8.       <value>istia.st.springmvc.exemples.web.Formulaire</value>
9.     </property>
10.    <property name="commandName">
11.      <value>formulaire</value>
12.    </property>
13.    <property name="pages">
14.      <list>
15.        <value>page1</value>
16.        <value>page2</value>
17.        <value>page3</value>
18.      </list>
19.    </property>
20.    <property name="allowDirtyBack">
21.      <value>true</value>
22.    </property>
23.    <property name="allowDirtyForward">
24.      <value>false</value>
25.    </property>
26. </bean>
```

- lignes 7-9 : définissent l'attribut [**commandClass**] qui fixe la classe qui doit être instanciée pour recevoir les saisies de l'utilisateur. Si cet attribut n'est pas défini, alors c'est la méthode [**formBackingObject**] qui doit fournir ce conteneur des saisies (ce ne sera pas le cas ici). Le conteneur mémorisera le **nom** et l'**âge** saisis. Il est ici de type [Formulaire].
- lignes 4-6 : définissent l'attribut [sessionForm] qui décide du maintien ou non en session de l'objet [Formulaire] entre les différentes pages. Si on veut permettre à l'utilisateur de revenir en arrière dans l'assistant et qu'il retrouve les valeurs qu'il a précédemment saisies, on mettra l'attribut [sessionForm] à vrai.
- lignes 10-12 : le nom de l'attribut à utiliser dans les pages JSP / JSTL pour avoir accès à l'objet [Formulaire], conteneur des saisies. Ici, ce sera [**formulaire**].

Les attributs précédents avaient déjà été rencontrés dans la configuration de [SimpleFormController]. Les attributs définis lignes 13-25 sont propres au contrôleur [AbstractWizardFormController] :

- lignes 13-19 : l'attribut [**pages**] définit les noms des vues formant l'assistant. Ici l'assistant est formé de trois vues appelées respectivement : **page1**, **page2**, **page3**. Ces noms de vues seront associés à des page JSP / JSTL grâce au résolveur de noms de vues.

- lignes 20-22 : l'attribut [**allowDirtyBack**] indique si l'utilisateur pourra ou non revenir en arrière alors même que les saisies dans la page courante sont invalides. Si la propriété [allowDirtyBack] est à vrai, alors l'assistant néglige les valeurs postées par la page courante et fait afficher la page précédente . Si elle est à faux, alors l'assistant vérifie les valeurs postées par la page courante. Si elles sont incorrectes, la page courante est réaffichée. La valeur par défaut de la propriété [allowDirtyBack] est la valeur *true*
- lignes 23-25 : l'attribut [**allowDirtyForward**] indique si l'utilisateur peut passer à la page suivante alors même que les saisies dans la page courante sont invalides. Si la propriété [allowDirtyForward] est à vrai, alors l'assistant néglige les valeurs postées par la page courante et fait afficher la page suivante. Si elle est à faux, alors l'assistant vérifie les valeurs postées par la page courante. Si elles sont incorrectes, la page courante est réaffichée. La valeur par défaut de la propriété [allowDirtyForward] est la valeur **false**.

Les lignes 13-19 ne précisent que les noms des vues formant l'assistant. L'association du nom de la vue à sa classe d'affichage est faite par le résolveur de noms de vues, ici [ResourceBundleViewResolver] :

```
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] (ligne 4) est défini dans le dossier [WEB-INF/src] comme suit :

```
1.  #page1
2.  page1.class=org.springframework.web.servlet.view.JstlView
3.  page1.url=/WEB-INF/vues/page1.jsp
4.  #page2
5.  page2.class=org.springframework.web.servlet.view.JstlView
6.  page2.url=/WEB-INF/vues/page2.jsp
7.  #page3
8.  page3.class=org.springframework.web.servlet.view.JstlView
9.  page3.url=/WEB-INF/vues/page3.jsp
10. #index
11. index.class=org.springframework.web.servlet.view.JstlView
12. index.url=/index.htm
13. #validation
14. validation.class=org.springframework.web.servlet.view.JstlView
15. validation.url=/WEB-INF/vues/validation.jsp
```

Cinq vues JSP / JSTL sont définies :
- les trois vues [page1, page2, page3] qui forment l'assistant
- la vue [index] qui contient le lien à partir duquel on lancera l'assistant
- la vue [validation] qui va confirmer les saisies faites dans l'assistant

Le cheminement normal de l'utilisateur dans ces vues est le suivant : index -> page1 -> page2 -> page3 -> validation

Le comportement de l'assistant est gouverné par la présence de certains paramètres dans la requête qu'il traite. La valeur de ceux-ci n'a aucune importance. Seule importe leur présence. Parmi ces paramètres on trouve les suivants :

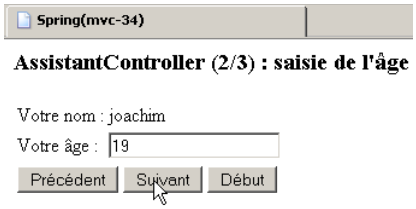
- **_page** : le n° de la page dont on demande la validation. L'assistant va vérifier la validité des saisies de la page n° [**_page**] en exécutant la méthode

```
protected void validatePage(Object formulaire, Errors errors, int page)
```

  - **formulaire** : le conteneur des saisies. A noter qu'il contient les saisies faites dans l'ensemble des pages de l'assistant et non seulement les saisies de la page n° [**_page**].
  - **page** : le n° de page courant de l'assistant. Avec cette information, la méthode **validatePage** pourra ne vérifier la validité que des seules saisies de cette page particulière.
  - **errors** : la liste des erreurs rencontrées. Vide au départ, elle doit être remplie par la méthode **validatePage** au fur et à mesure que celle-ci détecte des données invalides. Si, à l'issue de la méthode **validatePage**, cette liste est non vide, la page invalide est réaffichée. On s'arrange en général pour qu'elle affiche outre les valeurs précédemment saisies, les messages des erreurs détectées.
- **_targetx** : **x** est un n° de page commençant à 0. La présence de ce paramètre dans la requête indique à l'assistant qu'il doit passer à la page n° **x**. Il s'agit dans l'assistant, de passer de la page n° [**_page**] à la page n° [**x**]. Ceci ne sera possible que si la page n° [**_page**] n'a pas été déclarée invalide par la méthode [**validatePage**].
  La présence du paramètre [_target1] indique ainsi à l'assistant qu'il doit passer à la page n° 1. Dans notre exemple, le tableau des pages est [page1,page2,page3]. Les pages sont numérotées à partir de 0. Ainsi la page n° 1 est-elle la vue " page2 " et donc la page JSP / JSTL [page2.jsp] (cf vues.properties).

- **_cancel** : la présence de cet attribut doit être interprétée comme une demande de sortie de l'assistant avec abandon des saisies déjà faites. A la rencontre de ce paramètre, l'assistant va exécuter la méthode [**processCancel**]. Cette méthode doit rendre un [ModelAndView] traduisant l'abandon de l'assistant.
- **_finish** : la présence de cet attribut doit être interprétée comme une demande de sortie de l'assistant avec validation des saisies faites. A la rencontre de ce paramètre, l'assistant va exécuter la méthode [**processFinish**]. Cette méthode doit rendre un [ModelAndView] traduisant la validation des saisies. Cependant avant d'exécuter cette méthode, l'assistant fait une validation de toutes les pages de l'assistant, c.a.d. qu'il appelle la méthode **validatePage** pour chacune des pages du tableau des pages. Si la validation des pages a été faite comme expliqué précédemment, cela peut ressembler redondant. En effet, il semble que si on est arrivé à la dernière page de l'assistant c'est que les précédentes étaient correctes. Ce fonctionnement n'est cependant pas le seul possible. Un fonctionnement différent qui permet d'avancer dans l'assistant sans vérification des données est également possible (allowDirtyForward=true). Celles-ci sont alors toutes vérifiées d'un seul coup lorsque le paramètre **_finish** est reçu.

Prenons un exemple. La 2ième page de notre assistant est la vue " page2 " suivante :



- le bouton [**Suivant**] doit servir à passer à la page suivante de l'assistant. Parmi les paramètres postés, il faudra mettre les suivants :
  - **_page=1** : pour refléter le fait que la page courante " page2 " est la page d'indice 1 dans le tableau des pages [page1, page2, page3] de l'assistant.
  - **_target2** : pour refléter le fait que, si la page courante est valide, on veut passer à la page d'indice 2 dans le tableau des pages [page1, page2, page3] de l'assistant, c.a.d à la vue " page3 ".

Cela peut être obtenu avec le code HTML suivant :

```
1.  <form method="post">
2.  ...
3.        <input type="hidden" name="_page" value="1">
4.  ...
5.        <input type="submit" value="Suivant" name="_target2"></td>
6.  ...
7.  </form>
```

- le bouton [**Précédent**] doit servir à revenir à la page précédente de l'assistant. Parmi les paramètres postés, il faudra mettre les suivants :
  - **_page=1** : pour refléter le fait que la page courante " page2 " est la page d'indice 1 dans le tableau des pages [page1, page2, page3] de l'assistant.
  - **_target0** : pour refléter le fait qu'on veut passer à la page d'indice 0 dans le tableau des pages [page1, page2, page3] de l'assistant, c.a.d à la vue " page1 ". A noter qu'alors la validation de la page [**_page**] n'aura pas lieu (allowDirtyBack=true) et les saisies de la page courante seront abandonnées.

Cela peut être obtenu avec le code HTML suivant :

```
1.  <form method="post">
2.  ...
3.        <input type="hidden" name="_page" value="1">
4.  ...
5.        <input type="submit" value="Précédent" name="_target0"></td>
6.  ...
7.  </form>
```

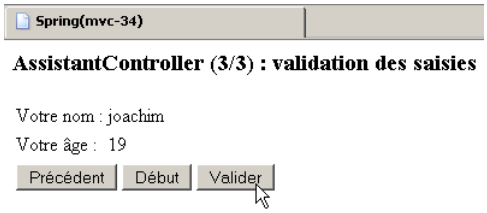
- le bouton [**Début**] doit permettre d'abandonner l'assistant. Parmi les paramètres postés, il faudra mettre le suivant :
  - **_cancel** : pour refléter le fait que l'utilisateur veut abandonner l'assistant.

Cela peut être obtenu avec le code HTML suivant :

```
1.  <form method="post">
2.  ...
3.        <input type="hidden" name="_page" value="1">
4.  ...
5.        <input type="submit" value="Début" name="_cancel"></td>
6.  ...
```

```
7.  </form>
```

Considérons maintenant la dernière page de l'assistant :



Le bouton [**Valider**] doit provoquer la sortie de l'assistant. Il faut donc mettre le paramètre **_finish** dans les paramètres postés. Cela peut être obtenu avec le code HTML suivant :

```
1.     <form method="post">
2.       <table>
3.         <tr>
4.           <td><input type="submit" value="Précédent" name="_target1"></td>
5.           <td><input type="submit" value="Début" name="_cancel"></td>
6.           <td><input type="submit" value="Valider" name="_finish"></td>
7.         </tr>
8.       </table>
9.  </form>
```

Ici, le paramètre **_page** est inutile car il n'y a aucune validation de données à faire.

2.4.2 Le conteneur des saisies

Au fil des pages de l'assistant, les saisies seront enregistrées dans une instance de la classe [Formulaire] suivante :

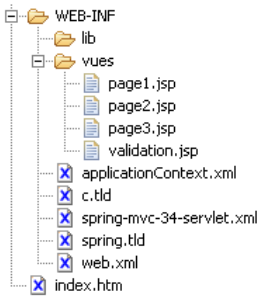
```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Formulaire {
4.    // nom
5.    private String nom;
6.
7.    // age
8.    private int age;
9.
10.   // getters - setters
11. ....
12. }
```

Lors de l'affichage de la première page de l'assistant, une instance de cette classe est créée. Elle va rester ensuite en session jusqu'à la fin de l'assistant. Elle sera placée dans le modèle de chacune des vues [page1, page2, page3] avant affichage de celles-ci sous le nom " **formulaire** ". Elle servira à mémoriser les données saisies lors des POST des vues [page1, page2]. Ce fonctionnement découle de la configuration du contrôleur faite dans (spring-mvc-34-servlet.xml, page 23) et rappelée ci-dessous :

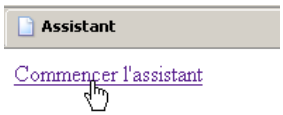
```
1.  <!-- les contrôleurs de l'application-->
2.    <bean id="AssistantController"
3.      class="istia.st.springmvc.exemples.web.AssistantController">
4.      <property name="sessionForm">
5.        <value>true</value>
6.      </property>
7.      <property name="commandClass">
8.        <value>istia.st.springmvc.exemples.web.Formulaire</value>
9.      </property>
10.     <property name="commandName">
11.       <value>formulaire</value>
12.     </property>
13.     <property name="pages">
14.       <list>
15.         <value>page1</value>
16.         <value>page2</value>
17.         <value>page3</value>
18.       </list>
19.     </property>
20.     <property name="allowDirtyBack">
21.       <value>true</value>
22.     </property>
23.     <property name="allowDirtyForward">
24.       <value>true</value>
25.     </property>
26.   </bean>
```

2.4.3 Les vues du projet

Les vues du projet sont dans le dossier [vues] du projet ainsi que sous la racine du projet (index.htm) :



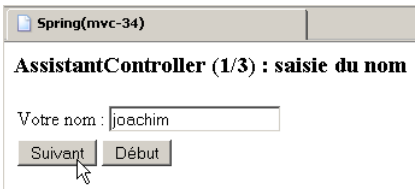
L'application démarre par la demande de l'url [/spring-mvc-34]. Nous avons vu que cela allait se traduire par la demande de l'url [/spring-mvc-34/index.htm] (cf web.xml page 22). La page obtenue est la suivante :



Le code de cette page est le suivant :

```
1.  <html>
2.    <head>
3.      <title>Assistant</title>
4.    </head>
5.    <body>
6.      <a href="assistant.html">Commencer l'assistant</a>
7.    </body>
8.  </html>
```

Le lien pointe sur l'url [assistant.html]. D'après l'étude du fichier [spring-mvc-34-servlet.xml] (page 23), nous savons que cette url va être traitée par l'assistant [AbstractWizardFormController]. Parce que c'est un GET, l'assistant va afficher la première page, c.a.d. la vue " page1 " associée à la page JSP [page1.jsp] :



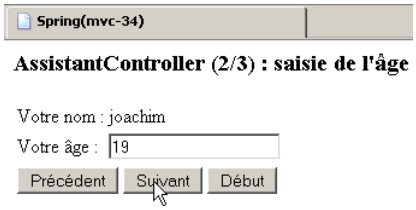
Le code de [page1.jsp] est le suivant :

```
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.
6.  <html>
7.    <head>
8.      <title>Spring(mvc-34)</title>
9.    </head>
10.   <body>
11.     <h3>AssistantController (1/3) : saisie du nom</h3>
12.     <form method="post">
13.       <table border="0">
14.         <tr>
15.           <td>Votre nom :</td>
16.           <spring:bind path="formulaire.nom">
17.             <td>
18.               <input type="text" name="${status.expression}" value="${status.value}">
19.             </td>
20.             <td>${status.errorMessage}</td>
21.           </spring:bind>
22.         </tr>
```

```
23.       </table>
24.       <table>
25.         <tr>
26.           <td><input type="submit" name="_target1" value="Suivant"></td>
27.           <td><input type="submit" name="_cancel" value="D&eacute;but"></td>
28.         </tr>
29.       </table>
30.       <input type="hidden" name="_page" value="0">
31.     </form>
32.   </body>
33. </html>
```

- lignes 16-21 : la saisie du nom est contrôlée par une balise <spring:bind>.
- ligne 16 : le champ HTML de saisie du nom est associé au champ [nom] de l'objet de clé " formulaire ". On sait par [spring-mvc-34-servlet.xml] (page 23) que cet attribut est associé à un objet de type [Formulaire] qui a deux champs : [nom, age].
- ligne 18 : ${status.value} permet d'afficher la valeur actuelle de [formulaire.nom]
- ligne 20 : ${status.errorMessage} permet d'afficher l'éventuel message d'erreur lié à la saisie du champ [formulaire.nom]. Ce message sera construit par la méthode [validatePage] de l'assistant appliquée à la page n° 0 (attribut _page ligne 30).
- ligne 26 : le bouton [Suivant] qui va provoquer la validation de la page n° 0 (attribut _page ligne 30) et le passage à la page n° 1 (attribut name= "_target1 ")
- ligne 27 : le bouton [Début] qui va provoquer l'abandon de l'assistant (attribut name= "_cancel ") et l'exécution de la méthode [processCancel] de l'assistant.

Si le nom est valide, l'assistant va afficher la page n° 1, c.a.d. la vue nommée " page2 " associée à la page JSP [page2.jsp] :



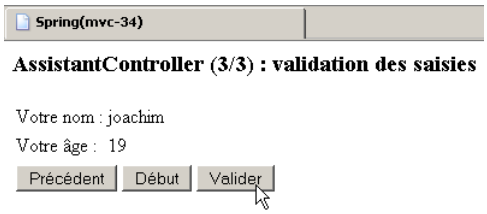
Le code de [page2.jsp] est le suivant :

```
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.
6.  <html>
7.    <head>
8.      <title>Spring(mvc-34)</title>
9.    </head>
10.   <body>
11.     <h3>AssistantController (2/3) : saisie de l'&acirc;ge</h3>
12.     <form method="post">
13.       <table border="0">
14.         <tr>
15.           <td>Votre nom :</td>
16.           <td>${formulaire.nom}</td>
17.         </tr>
18.         <tr>
19.           <td>Votre &acirc;ge :</td>
20.           <spring:bind path="formulaire.age">
21.             <td>
22.               <input type="text" name="${status.expression}" value="${status.value}">
23.             </td>
24.             <td>${status.errorMessage}</td>
25.           </spring:bind>
26.         </tr>
27.       </table>
28.       <input type="hidden" name="_page" value="1">
29.       <table>
30.         <tr>
31.           <td><input type="submit" value="Pr&eacute;c&eacute;dent" name="_target0"></td>
32.           <td><input type="submit" value="Suivant" name="_target2"></td>
33.           <td><input type="submit" value="D&eacute;but" name="_cancel"></td>
34.         </tr>
35.       </table>
36.     </form>
37.   </body>
38. </html>
```

- ligne 16 : on affiche le nom saisi à l'étape précédente

- lignes 20-25 : la saisie de l'âge est contrôlée par une balise <spring:bind>.
- ligne 20 : le champ HTML de saisie de l'âge est associé au champ [age] de l'objet de clé " formulaire ". Cet objet de type [Formulaire] est le même qui a servi à la page n° 0 (sessionForm=true dans la configuration de l'assistant).
- ligne 22 : ${status.value} permet d'afficher la valeur actuelle de [formulaire.age]
- ligne 24 : ${status.errorMessage} permet d'afficher l'éventuel message d'erreur lié à la saisie du champ [formulaire.age]. Ce message sera construit par la méthode [validatePage] de l'assistant appliquée à la page n° 1 (attribut _page ligne 28).
- ligne 31 : le bouton [Précédent] qui va provoquer le retour à la page n° 0 (attribut name= "_target0 ") sans validation des saisies de la page courante n° 1 (allowDirtyForward=true dans la configuration de l'assistant).
- ligne 32 : le bouton [Suivant] qui va provoquer la validation de la page n° 1 (attribut _page ligne 28) et le passage à la page n° 2 (attribut name= "_target2 ")
- ligne 33 : le bouton [Début] qui va provoquer l'abandon de l'assistant (attribut name= "_cancel ") et l'exécution de la méthode [processCancel] de celui-ci.

Si l'âge est valide, l'assistant va afficher la page n° 2, c.a.d. la vue nommée " page3 " associée à la page JSP [page3.jsp] :



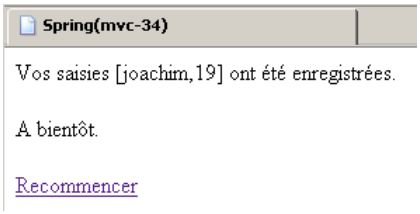
Le code de [page3.jsp] est le suivant :

```
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.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring(mvc-34)</title>
8.    </head>
9.    <body>
10.     <h3>AssistantController (3/3) : validation des saisies</h3>
11.       <table border="0">
12.         <tr>
13.           <td>Votre nom :</td>
14.           <td>${formulaire.nom}</td>
15.         </tr>
16.         <tr>
17.           <td>Votre &acirc;ge :</td>
18.           <td>${formulaire.age}</td>
19.         </tr>
20.       </table>
21.     <!-- form -->
22.     <form method="post">
23.       <table>
24.         <tr>
25.           <td><input type="submit" value="Pr&eacute;c&eacute;dent" name="_target1"></td>
26.           <td><input type="submit" value="D&eacute;but" name="_cancel"></td>
27.           <td><input type="submit" value="Valider" name="_finish"></td>
28.         </tr>
29.       </table>
30.     </form>
31.   </body>
32. </html>
```

- ligne 13 : confirme le nom saisi
- ligne 18 : confirme l'âge saisi
- ligne 25 : le bouton [Précédent] qui va provoquer le retour à la page n° 1 (attribut name= "_target1 ").
- ligne 27 : le bouton [Suivant] qui va provoquer la validation de l'ensemble des pages de l'assistant (attribut name= "_finish ") puis l'exécution de la méthode [processFinish] de celui-ci.
- ligne 26 : le bouton [Début] qui va provoquer l'abandon de l'assistant (attribut name= "_cancel ") et l'exécution de la méthode [processCancel] de celui-ci.

Dans notre cas, il n'y aura pas de surprise. L'ensemble des pages sera validé puisque :
- on passe d'une page à l'autre seulement si la page est valide (allowDirtyForward=false dans la configuration de l'assistant)
- la dernière page ne comporte pas de saisies

La méthode [processFinish] de l'assistant envoie la page de confirmation suivante :



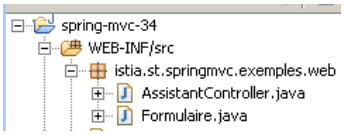
Cette page a " validation " comme nom de vue et est associée à la page [validation.jsp] suivante :

```
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-34)</title>
8.    </head>
9.    <body>
10.     Vos saisies [${formulaire.nom},${formulaire.age}] ont &eacute;t&eacute; enregistr&eacute;es.
11.     <br><br>
12.     A bient&ocirc;t.
13.     <br><br>
14.     <a href="index.htm">Recommencer</a>
15.   </body>
16. </html>
```

- ligne 10 : on affiche les saisies de l'objet associé à la clé " formulaire ". Il faut faite attention ici, que cette page ne fait pas partie des pages de l'assistant qui est désormais terminé. L'assistant maintenait dans la session un conteneur [Formulaire] qu'il rendait accessible aux pages JSP via la clé " formulaire ". L'assistant étant terminé, le conteneur [Formulaire] n'existe plus. Il faudra donc que la méthode [processFinish] mette elle-même le conteneur [Formulaire] dans le modèle de la vue " validation " afin que celle-ci y ait accès.

2.4.4 Les classes du projet

Revenons au projet Eclipse / Tomcat pour étudier ses classes :



- [**Formulaire.java**] est le conteneur des saisies de l'assistant
- [**AssistantController.java**] est l'assistant

La classe [**Formulaire.java**] a été décrite au paragraphe 2.4.2, page 27.

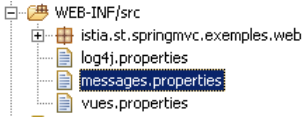
Le contrôleur [**AssistantController.java**] est le suivant :

```
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.validation.Errors;
11. import org.springframework.web.servlet.ModelAndView;
12. import org.springframework.web.servlet.mvc.AbstractWizardFormController;
13.
14. public class AssistantController extends AbstractWizardFormController {
15.
16.   protected ModelAndView processFinish(HttpServletRequest request,
17.       HttpServletResponse response, Object command, BindException errors)
18.       throws Exception {
19.     // confirmation des saisies
20.     Map modèle=new HashMap();
21.     modèle.put("formulaire",command);
22.     return new ModelAndView("validation",modèle);
```

```
23.     }
24.
25.   protected ModelAndView processCancel(HttpServletRequest request,
26.             HttpServletResponse response,
27.             Object command,
28.             BindException errors)
29.       throws Exception{
30.     // retour à la case départ
31.     return new ModelAndView("index");
32.   }
33.
34.   // validation d'une page
35.   protected void validatePage(Object formulaire, Errors errors, int page) {
36.     // on récupère le formulaire
37.     Formulaire form=(Formulaire)formulaire;
38.     // quelle page ?
39.     switch (page) {
40.     case 0:
41.       validatePage0(form, errors);
42.       break;
43.     case 1:
44.       validatePage1(form, errors);
45.       break;
46.     }
47.   }
48.
49.   // validation page 0
50.   protected void validatePage0(Formulaire form, Errors errors) {
51.     // le nom doit être non vide
52.     String nom=form.getNom();
53.     if(nom==null || nom.trim().length()==0){
54.       errors.rejectValue("nom","formulaire.nom.obligatoire");
55.     }
56.   }
57.
58.   // validation page 1
59.   protected void validatePage1(Formulaire form, Errors errors) {
60.     // l'âge doit être positif
61.     int age=form.getAge();
62.     if(age<1 || age>150){
63.       errors.rejectValue("age","formulaire.age.invalide", new String[]{"1","150"},"Donnée
    invalide");
64.     }
65.   }
66. }
```

- ligne 14 : [AssistantController] dérive de [AbstractWizardFormController]
- lignes 35-47 : la méthode [**validatePage**] qui sert à valider les données postées par la page n° [page] de l'assistant. Ce cas se produit lorsque l'utilisateur clique le bouton [Suivant] des pages [page1, page2] donc pour les pages n° 0 et 1.
- lignes 40-42 : si page=0, alors la méthode [validatePage] appelle la méthode [validatePage0] qui vérifie que le nom n'est pas vide.
- lignes 43-45 : si page=1, alors la méthode [validatePage] appelle la méthode [validatePage1] qui vérifie que l'âge est dans l'intervalle [1,150]. Il faut rappeler que l'âge a été saisi dans la page n° 1 tout d'abord comme chaîne de caractères. Spring MVC a essayé de transformer celle-ci en nombre entier pour la mettre dans le champ [age] de type [int] de l'instance [Formulaire] qui sert de conteneur aux saisies de l'assistant. Si cette transformation échoue, Spring MVC
    - n'appelle pas la méthode [validatePage] de l'assistant
    - construit un objet [Errors] avec dedans une erreur associée à la clé "**typeMismatch**"
    - provoque le réaffichage de la page n° 1

Les codes d'erreurs utilisés par les différentes méthodes de validation sont définis dans le fichier [messages.properties] du projet :



Ce fichier a été défini comme fichier des messages dans le fichier de configuration [applicationContext.xml] (cf page 23). Son contenu est le suivant :

```
1.  typeMismatch=Donnée incorrecte !
2.  formulaire.nom.obligatoire=Le nom est obligatoire ...
3.  formulaire.age.invalide=Indiquez un âge dans l''intervalle [{0},{1}] ...
```

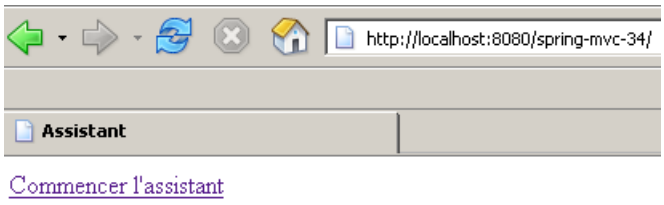
On notera ligne 3, l'utilisation de paramètres positionnels dans le message. Ces paramètres sont initialisés ligne 63 de l'assistant avec respectivement les chaînes 1 et 150.

- lignes 25-32 : la méthode [**processCancel**] qui sert à traiter le cas où l'utilisateur a demandé l'abandon de l'assistant. Dans ce cas, on demande l'affichage de la vue nommée " index " (ligne 31). Dans [vues.properties] (page 25), ce nom est associée à la page HTML " index.htm " déjà présentée. C'est la page qui présente le lien de démarrage de l'assistant.

- lignes 16-24 : la méthode [**processFinish**] qui sert à traiter le cas où l'utilisateur a demandé la validation de l'ensemble des pages de l'assistant.
  - on veut afficher la vue nommée " validation " associée à la page JSP [validation.jsp] présentée page 31. Nous avons alors expliqué que la méthode [processFinish] devait mettre le conteneur de saisies de type [Formulaire] dans le modèle de la vue " validation " associé à la clé " formulaire ". C'est ce qui est fait lignes 20-21.

2.4.5 Conclusion

La présentation du contrôleur [AbstractWizardFormController] a été complexe. Heureusement, ce contrôleur est, dans la pratique, plus facile à utiliser qu'à expliquer. Le lecteur est invité à tester l'application en demandant l'url [http://localhost:8080/spring-mvc-34/] :



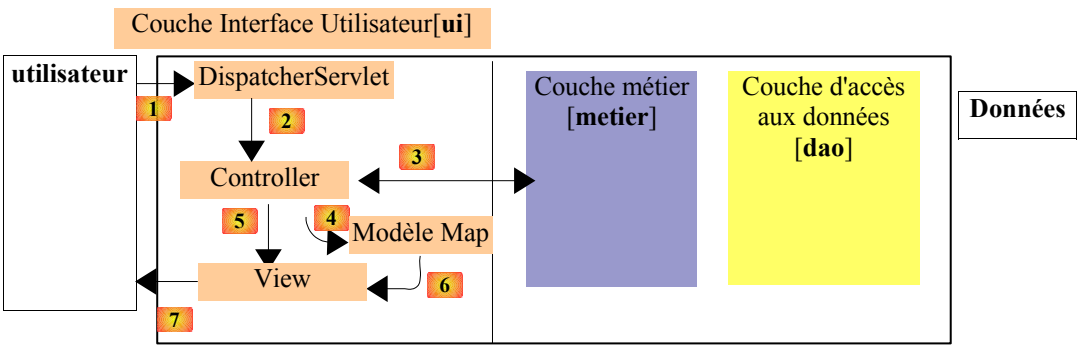
Lorsque différents tests auront été faits, il est conseillé de modifier dans (spring-mvc-34-servlet.xml) les attributs [allowDirtyBack] et [allowDirtyForward] de l'assistant afin de voir l'influence de ces deux attributs sur son comportement :

```
1.<!-- le contrôleur de l'application-->
2.  <bean id="AssistantController"
3.    class="istia.st.springmvc.exemples.web.AssistantController">
4.  ...
5.    <property name="allowDirtyBack">
6.      <value>true</value>
7.    </property>
8.    <property name="allowDirtyForward">
9.      <value>true</value>
10.   </property>
11. </bean>
```

3 Produire des vues Excel ou PDF

3.1 Introduction

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



On sait que l'opération (7) de rendu d'une vue est faite par une classe implémentant l'interface [View] de Spring :

org.springframework.web.servlet
Interface View

All Known Implementing Classes:

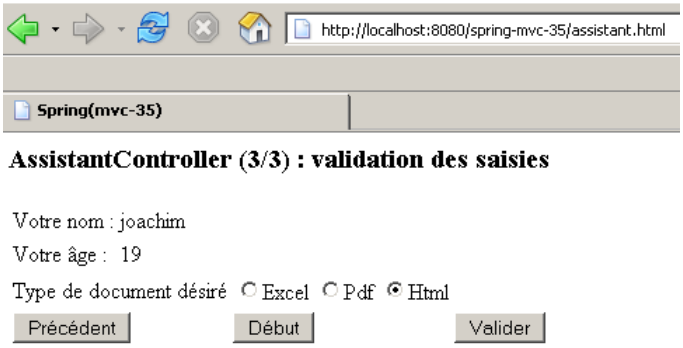
AbstractExcelView, AbstractJasperReportsSingleFormatView, AbstractJasperReportsView, AbstractJExcelView, AbstractPdfView, AbstractTemplateView, AbstractUrlBasedView, AbstractView, AbstractXsltView, FreeMarkerView, InternalResourceView, JasperReportsCsvView, JasperReportsHtmlView, JasperReportsMultiFormatView, JasperReportsPdfView, JasperReportsXlsView, JstlView, RedirectView, TilesJstlView, TilesView, VelocityLayoutView, VelocityToolboxView, VelocityView

Cette interface n'a qu'une unique méthode chargée d'envoyer une réponse au client. La forme exacte de celle-ci dépend de la classe d'implémentation utilisée. Parmi les classes d'implémentation prédéfinies de Spring et présentées ci-dessus, nous en avons rencontré deux :

- **JstlView** : la classe chargée de rendre des pages JSP / JSTL
- **RedirectView** : une classe qui envoie au client une demande de redirection

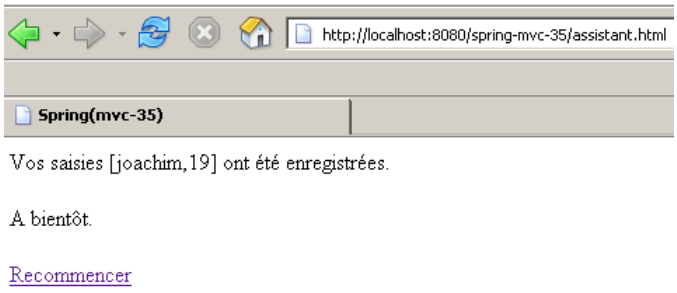
Nous allons présenter dans cet exemple deux nouvelles implémentations : **AbstractExcelView** et **AbstractPdfView**, deux classes abstraites qui une fois dérivées, permettent de générer respectivement un flux Excel ou un flux PDF comme réponse au client.

Pour illustrer l'utilisation de ces deux classes, nous allons reprendre l'assistant que nous venons d'étudier en changeant simplement sa dernière page. Celle-ci devient la suivante :

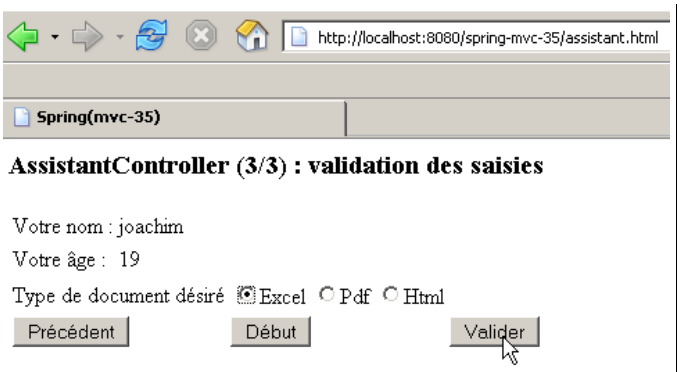


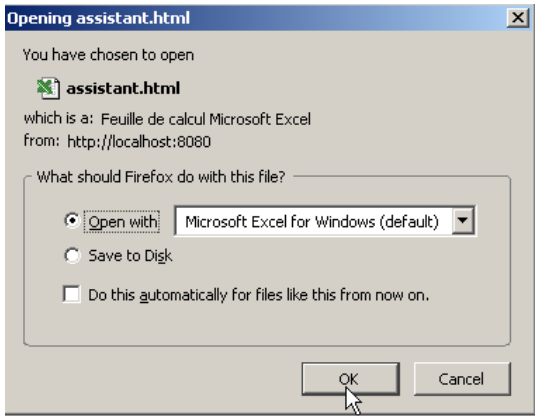
L'utilisateur peut choisir trois formats de confirmation de ses saisies.

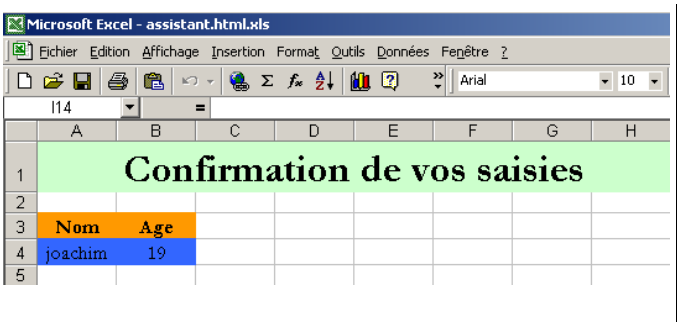
Le choix [Html] correspond à la page de confirmation de l'assistant (spring-mvc-34) :



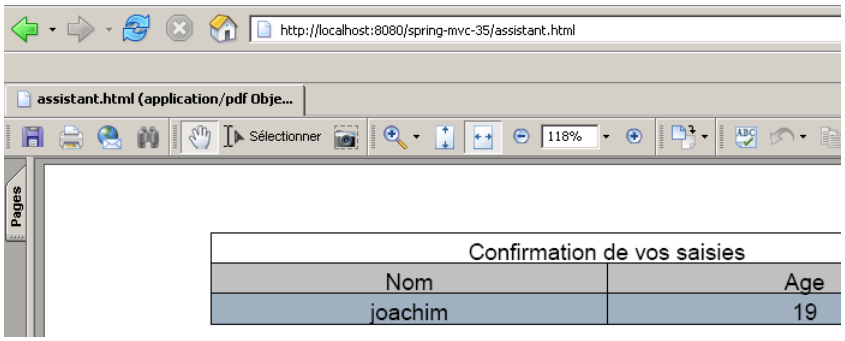
Le choix [Excel] envoie au client une confirmation des saisies sous la forme d'un flux Excel :



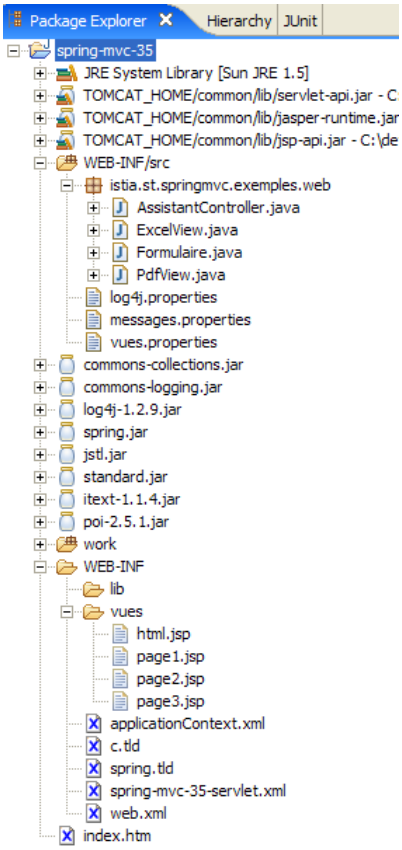




Le choix [Pdf] envoie au client une confirmation des saisies sous la forme d'un flux PDF :



Le nouveau projet Eclipse / Tomcat s'appelle [spring-mvc-35]. Son organisation est la suivante :



Le projet [spring-mvc-35] est, pour une large part, identique au projet [spring-mvc-34]. On a le même assistant. Seule sa dernière page [page3.jsp] change :

Le code JSP de la page 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>Spring(mvc-35)</title>
8.    </head>
9.    <body>
10.     <h3>AssistantController (3/3) : validation des saisies</h3>
11.       <table border="0">
12.         <tr>
13.           <td>Votre nom :</td>
14.           <td>${formulaire.nom}</td>
15.         </tr>
16.         <tr>
17.           <td>Votre &acirc;ge :</td>
18.           <td>${formulaire.age}</td>
19.         </tr>
20.       </table>
21.     <!-- forms -->
22.     <form method="post">
```

```
23.      <table>
24.        <tr>
25.          <td>Type de document d&eacute;sir&eacute;</td>
26.          <td>
27.            <input type="radio" value="excel" name="doc">Excel
28.            <input type="radio" value="pdf" name="doc">Pdf
29.            <input type="radio" value="html" name="doc" checked>Html
30.          </td>
31.        </tr>
32.        <tr>
33.          <td><input type="submit" value="Pr&eacute;c&eacute;dent" name="_target1"></td>
34.          <td><input type="submit" value="D&eacute;but" name="_cancel"></td>
35.          <td><input type="submit" value="Valider" name="_finish"></td>
36.        </tr>
37.      </table>
38.    </form>
39.  </body>
40. </html>
```

- la différence avec la page [page3.jsp] du projet précédent se trouve lignes 27-30. La valeur d'un groupe de boutons radio nommé " **doc** " va être postée. Les trois valeurs possibles de ce groupe de boutons sont " **excel** " (ligne 27), " **pdf** " (ligne 28) et " **html** " (ligne 29).

Le traitement des valeurs postées par les boutons [Précédent] et [Début] est le même que précédemment. Celui du POST provoqué par le bouton [Valider] (ligne 35) doit être modifié. En effet, il nous faut récupérer la valeur postée pour le groupe de boutons radio nommé " doc ". Parce que le nom du bouton [Valider] est " _finish ", nous savons que le POST est traité par la méthode [processFinish] de l'assistant. Cette méthode devient la suivante :

```
1.  protected ModelAndView processFinish(HttpServletRequest request,
2.      HttpServletResponse response, Object command, BindException errors)
3.      throws Exception {
4.    // confirmation des saisies
5.    Map modèle = new HashMap();
6.    modèle.put("formulaire", command);
7.    // type de confirmation souhaitée
8.    String vue = RequestUtils.getStringParameter(request, "doc", "html");
9.    // on rend le [ModelAndView]
10.   return new ModelAndView(vue, modèle);
11.}
```

- lignes 5-6 : le conteneur des saisies, de type [Formulaire] est mis dans le modèle de la vue à afficher, associé à la clé " formulaire ".
- ligne 8 : on récupère la valeur postée pour le groupe de boutons radio nommé " doc ". Nous aurions pu la récupérer en écrivant simplement :

```
String vue=(String)request.getParameter(" doc ");
```

Ici, nous avons utilisé la classe utilitaire [RequestUtils] :

org.springframework.web.bind

Class RequestUtils

java.lang.Object
  └ org.springframework.web.bind.RequestUtils

Cette classe offre un certain nombre de méthodes statiques facilitant l'extraction d'informations de la chaîne de paramètres de la requête : getIntParameter, getLongParameter, getDoubleParameter, getStringParameter, ... Ces méthodes offrent plusieurs versions surchargées. L'une d'elles permet de donner une valeur par défaut au paramètre demandé si celui-ci est absent de la requête. Ici l'instruction :

```
String vue = RequestUtils.getStringParameter(request, "doc", "html");
```

donne à [vue] la valeur du paramètre " doc " ou à défaut la valeur " html " si le paramètre " doc " est absent de la requête. Ce dernier cas n'est pas possible dans le fonctionnement normal de notre exemple. Au final, la variable [vue] a donc l'une des valeurs [" html ", "excel ", "pdf "].

On notera que puisque le groupe de boutons radio nommé " doc " est une saisie, on aurait pu ajouter au conteneur des saisies [Formulaire] un nouveau champ :

```
String doc;
```

et alors on aurait écrit :

```
String vue=((Formulaire)command).doc;
```

Nous avons voulu montrer qu'on pouvait parfois utiliser directement l'objet [request] pour y récupérer des paramètres postés.

- ligne 10 : la variable [vue] est utilisé comme nom de vue du [ModelAndView] rendu par la méthode [processFinish].

Le contrôleur général [DispatcherServlet] va demander à la classe chargée de rendre la vue de s'exécuter. Cette classe est trouvée grâce au résolveur de noms de vues, ici [ResourceBundleViewResolver]. C'est donc le fichier [vues.properties] qui va être utilisé. Le contenu de celui-ci diffère de ce qu'il était dans le projet précédent :

```
1.  #page1
2.  page1.class=org.springframework.web.servlet.view.JstlView
3.  page1.url=/WEB-INF/vues/page1.jsp
4.  #page2
5.  page2.class=org.springframework.web.servlet.view.JstlView
6.  page2.url=/WEB-INF/vues/page2.jsp
7.  #page3
8.  page3.class=org.springframework.web.servlet.view.JstlView
9.  page3.url=/WEB-INF/vues/page3.jsp
10. #index
11. index.class=org.springframework.web.servlet.view.JstlView
12. index.url=/index.htm
13. #excel
14. excel.class=istia.st.springmvc.exemples.web.ExcelView
15. #pdf
16. pdf.class=istia.st.springmvc.exemples.web.PdfView
17. #html
18. html.class=org.springframework.web.servlet.view.JstlView
19. html.url=/WEB-INF/vues/html.jsp
```

- lignes 1-12 : les vues " page1 ", " page2 ", " page3 " et " index " restent inchangées
- lignes 17-19 : la vue " html " n'est rien d'autre que la vue " validation " du projet précédent.
- ligne 14 : définit la classe de rendu vue " excel ". Cette classe [ExcelView] mettra le modèle de la vue dans un flux Excel.
- ligne 16 : définit la classe de rendu vue "pdf". Cette classe [PdfView] mettra le modèle de la vue dans un flux PDF.

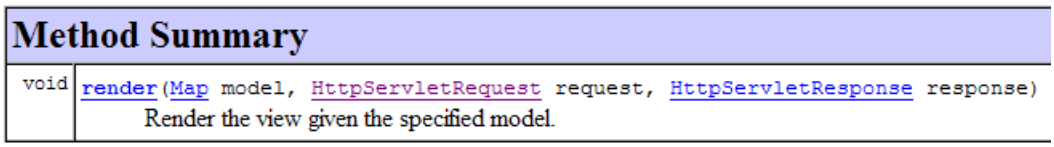
3.2 La vue HTML

Le fichier [vues.properties] indique que la vue nommée " html " est la page JSP [html.jsp]. Cette page n'est autre que la page [validation.jsp] du projet précédent qui a été renommée.

3.3 La vue Excel

Le fichier [vues.properties] indique que la vue nommée "excel" doit être rendue par un objet de type [ExcelView]. Cette classe, que nous allons construire, dérive de la classe prédéfinie [AbstractExcelView] :

On voit que la classe [AbstractExcelView] implémente l'interface [View], condition que doit remplir toute classe chargée de rendre une vue. On sait que cette interface n'a qu'une méthode :



La méthode [render] de la classe [AbstractExcelView] va :

1. construire un flux Excel et mettre le modèle [model] dedans
2. encapsuler ce flux Excel dans un flux HTTP
3. envoyer le flux HTTP au client grâce à l'objet [response]

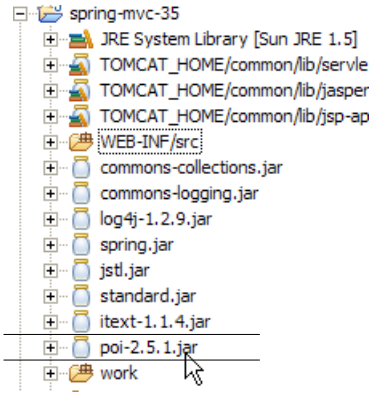
La classe [AbstractExcelView] assure elle-même les étapes 2 et 3. L'étape 1 est assurée par la méthode abstraite [buildExcelDocument] de la classe [AbstractExcelView] :

```
protected abstract void buildExcelDocument(Map model, HSSFWorkbook workbook, HttpServletRequest request, HttpServletResponse response)
        Subclasses must implement this method to create an Excel HSSFWorkbook document, given the model.
```

On y retrouve les trois paramètres de la méthode [render]. Le paramètre [**HSSFWorkbook workbook**] représente un classeur Excel vide. Le modèle [model] doit être inséré dans celui-ci. La classe générique [AbstractExcelView] ne peut savoir comment car cela dépend bien évidemment de chaque application. La classe [buildExcelDocument] est donc déclarée abstraite et doit être redéfinie dans une classe dérivée. Celle-ci n'a que cette méthode à redéfinir.

Le type [**HSSFWorkbook**] n'est pas un type Spring. Spring s'appuie ici sur un projet Apache : **Jakarta POI** disponible à l'url [http://jakarta.apache.org/poi/] :

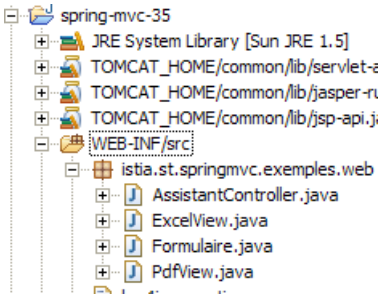
Les classes de ce projet sont disponibles dans une archive que nous devons inclure dans le *ClassPath* du projet [spring-mvc-35] :



La classe dérivée de [AbstractExcelView] chargée d'afficher la vue nommé " excel " est de type [ExcelView] comme indiqué dans le fichier [vues.properties] :

```
1.  #excel
2.  excel.class=istia.st.springmvc.exemples.web.ExcelView
```

Cette classe est définie dans [WEB-INF/src] :



Son code est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Map;
4.
5.  import javax.servlet.http.HttpServletRequest;
6.  import javax.servlet.http.HttpServletResponse;
7.
8.  import org.apache.poi.hssf.usermodel.HSSFCell;
9.  import org.apache.poi.hssf.usermodel.HSSFCellStyle;
10. import org.apache.poi.hssf.usermodel.HSSFFont;
11. import org.apache.poi.hssf.usermodel.HSSFSheet;
12. import org.apache.poi.hssf.usermodel.HSSFWorkbook;
13. import org.apache.poi.hssf.util.HSSFColor;
14. import org.apache.poi.hssf.util.Region;
15. import org.springframework.web.servlet.view.document.AbstractExcelView;
16.
17. public class ExcelView extends AbstractExcelView {
18.
19.  protected void buildExcelDocument(Map modèle, HSSFWorkbook classeur,
20.      HttpServletRequest request, HttpServletResponse response)
21.      throws Exception {
22.    // on récupère le formulaire dans le modèle
23.    Formulaire form = (Formulaire) modèle.get("formulaire");
24.    // on créée une feuille dans le classeur
25.    HSSFSheet feuille = classeur.createSheet("confirmation");
26.    // les polices de caractères
27.    // font 0
28.    HSSFFont font0 = classeur.createFont();
29.    font0.setFontHeightInPoints((short) 24);
30.    font0.setFontName("Garamond");
31.    font0.setBoldweight(HSSFFont.BOLDWEIGHT_BOLD);
32.    // font 1
33.    HSSFFont font1 = classeur.createFont();
34.    font1.setFontHeightInPoints((short) 12);
35.    font1.setFontName("Garamond");
36.    font1.setBoldweight(HSSFFont.BOLDWEIGHT_BOLD);
37.    // font 2
38.    HSSFFont font2 = classeur.createFont();
39.    font2.setFontHeightInPoints((short) 12);
40.    font2.setFontName("Garamond");
41.    font2.setBoldweight(HSSFFont.BOLDWEIGHT_NORMAL);
42.
43.    // les styles utilisés
44.    // style 0
45.    HSSFCellStyle style0 = classeur.createCellStyle();
46.    style0.setAlignment(HSSFCellStyle.ALIGN_CENTER);
47.    style0.setFillPattern(HSSFCellStyle.SOLID_FOREGROUND);
48.    style0.setFillForegroundColor(HSSFColor.LIGHT_GREEN.index);
49.    style0.setFont(font0);
50.    // style 1
51.    HSSFCellStyle style1 = classeur.createCellStyle();
52.    style1.setAlignment(HSSFCellStyle.ALIGN_CENTER);
53.    style1.setFillPattern(HSSFCellStyle.SOLID_FOREGROUND);
54.    style1.setFillForegroundColor(HSSFColor.LIGHT_ORANGE.index);
55.    style1.setFont(font1);
56.    // style 2
57.    HSSFCellStyle style2 = classeur.createCellStyle();
58.    style2.setAlignment(HSSFCellStyle.ALIGN_CENTER);
59.    style2.setFillPattern(HSSFCellStyle.SOLID_FOREGROUND);
60.    style2.setFillForegroundColor(HSSFColor.LIGHT_BLUE.index);
61.    style2.setFont(font2);
62.
63.    // contenu de la feuille
64.    // ligne 0
65.    int ligne = 0;
66.    HSSFCell cell = getCell(feuille, ligne, 0);
67.    cell.setCellValue("Confirmation de vos saisies");
68.    // cellule 0 étendue jusqu'à la cellule 7
69.    feuille.addMergedRegion(new Region(0, (short) 0, 0, (short) 7));
70.    // style de la cellule 0
71.    cell.setCellStyle(style0);
```

```
72.
73.     // ligne 2
74.     ligne += 2;
75.     cell = getCell(feuille, ligne, 0);
76.     cell.setCellValue("Nom");
77.     cell.setCellStyle(style1);
78.     cell = getCell(feuille, ligne, 1);
79.     cell.setCellValue("Age");
80.     cell.setCellStyle(style1);
81.
82.     // ligne 3
83.     ligne++;
84.     cell = getCell(feuille, ligne, 0);
85.     cell.setCellValue(form.getNom());
86.     cell.setCellStyle(style2);
87.     cell = getCell(feuille, ligne, 1);
88.     cell.setCellValue(form.getAge());
89.     cell.setCellStyle(style2);
90.   }
91. }
```

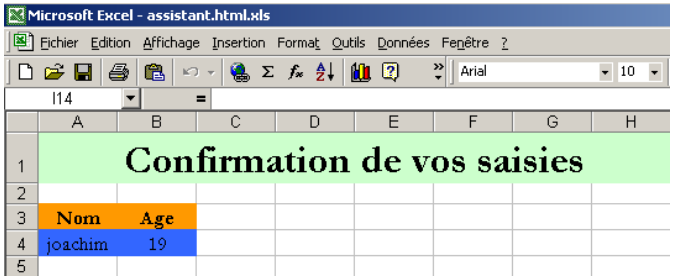
- ligne 17 : la classe [ExcelView] dérive de la classe [AbstractExcelView] de Spring
- lignes 19-21 : la méthode abstraite [buildExcelDocument] de [AbstractExcelView] est redéfinie

Rappelons que la méthode [buildExcelDocument] reçoit un classeur Excel vide qu'elle doit remplir avec le modèle. Ce classeur est le paramètre [HSSFWorkbook classeur] de la méthode. Pour construire le classeur Excel, il faut consulter la documentation du projet Jakarta POI. Un bon point de départ est l'url [http://jakarta.apache.org/poi/hssf/quick-guide.html] (avril 2006) :

Index of Features

- How to create a new workbook
- How to create a sheet
- How to create cells
- How to create date cells
- Working with different types of cells
- Aligning cells
- Working with borders
- Fills and color
- Merging cells
- Working with fonts
- Custom colors
- Reading and writing
- Use newlines in cells.
- Create user defined data formats
- Fit Sheet to One Page
- Set print area for a sheet
- Set page numbers on the footer of a sheet
- Shift rows
- Set a sheet as selected
- Set the zoom magnification for a sheet
- Create split and freeze panes
- Repeating rows and columns
- Headers and Footers
- Drawing Shapes
- Styling Shapes
- Shapes and Graphics2d
- Outlining
- Images

Nous commentons les grandes lignes de la méthode [buildExcelDocument] qui vise à construire le classeur Excel suivant :

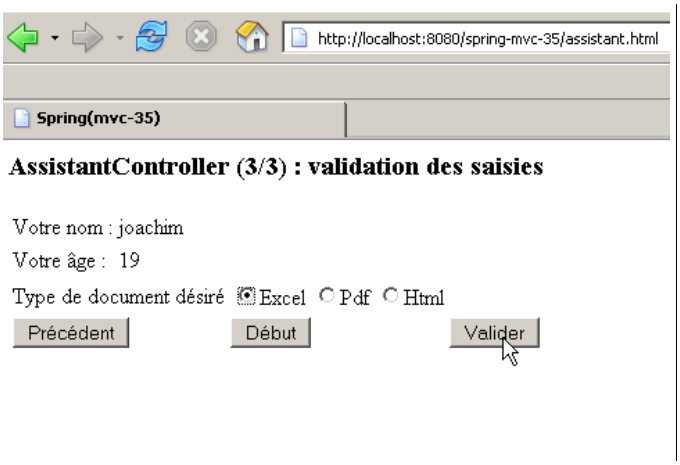


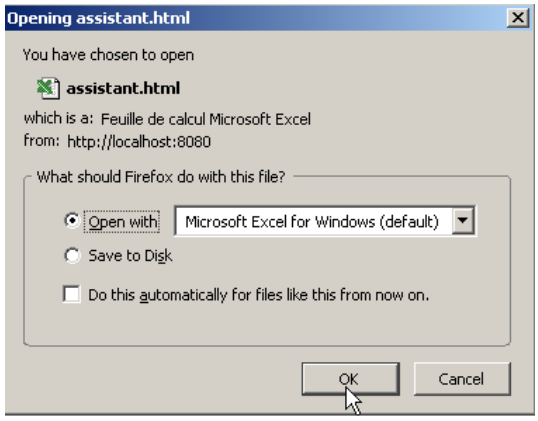
- ligne 23 : récupère le modèle à rendre. Nous savons que c'est une instance de la classe [Formulaire].
- ligne 25 : crée une feuille dans le classeur Excel et lui donne le nom " confirmation "
- lignes 28-31 : définit un type de caractères " font0 "
- ligne 28 : l'instance " font0 " est créée
- ligne 29 : fixe la taille de la police à 24 points

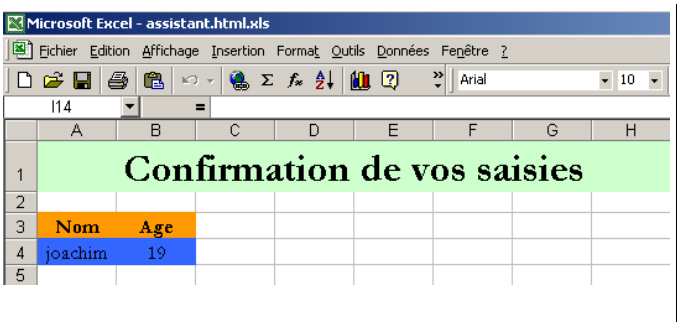
- ligne 30 : fixe à " Garamond " la police utilisée
- ligne 31 : fixe le style des caractères à " gras "
- ligne 33-36 : une police " font1 " (12 pts, Garamond, " gras ") est définie
- ligne 38-41 : une police " font2 " (12 pts, Garamond, " normal ") est définie
- lignes 45-49 : définissent un style " style0 " de cellule
- ligne 45 : le style " style0 " est créé
- ligne 46 : fixe l'alignement horizontal du style à " centré "
- ligne 47 : fixe le motif de remplissage des cellules du style à " solid ", c.a.d. un motif plein couvrant la totalité de la cellule
- ligne 48 : fixe la couleur de remplissage du style à " vert clair "
- ligne 49 : fixe la police de caractères du style à " font0 ", le style défini lignes 28-31
- lignes 51-55 : un style " style1 " est créé
- lignes 57-61 : un style " style2 " est créé
- ligne 66 : crée une référence sur la cellule (0,0) de la feuille courante
- ligne 67 : met le texte " Confirmation de vos saisies " dans cette cellule
- ligne 69 : crée une région, c.a.d. une zone rectangulaire de cellules. Celle-ci peut être définie par les coordonnées des cellules de début et de fin d'une des diagonales du rectangles. Ici Region(0,0,0,7) définit la région allant de la cellule (0,0) à la cellule (0,7), donc une région de 8 cellules dans le coin supérieur gauche de la feuille. Les cellules de cette région sont fusionnées (feuille.addMergedRegion). Ceci a pour conséquence que la cellule (0,0) occupe désormais toute la région créée.
- ligne 71 : le style " style0 " est appliqué à la cellule (0,0)
- lignes 75-80 : le texte " Nom " est mis dans la cellule (2,0), le texte " Age " dans la cellule (2,1). Le style " style1 " est appliqué aux deux cellules.
- lignes 84-89 : la valeur du champ **nom** du formulaire est mis dans la cellule (3,0), la valeur du champ **age** dans la cellule (3,1). Le style " style2 " est appliqué aux deux cellules.

C'est tout. Le modèle [model] a été mis dans un classeur Excel. La méthode [render] de [AbstractExcelView] va se charger d'encapsuler celui-ci dans la réponse HTTP envoyée au client.

Aux tests, cela donne les choses suivantes :







Si on regarde le flux HTTP envoyé au client, avec un outil tel que LiveHTTPHeaders, on obtient la chose suivante :

```
1.  HTTP/1.x 200 OK
2.  Server: Apache-Coyote/1.1
3.  Pragma: No-cache
4.  Expires: Thu, 01 Jan 1970 00:00:00 GMT
```

```
5.  Cache-Control: no-cache, no-store
6.  Content-Type: application/vnd.ms-excel;charset=ISO-8859-1
7.  Content-Language: fr-FR
8.  Transfer-Encoding: chunked
9.  Date: Sat, 15 Apr 2006 14:03:25 GMT
```

- la ligne 6 indique qu'après les entêtes HTTP (lignes 1-9), le serveur va envoyer un document de type MIME [application/vnd.ms-excel]. A réception de cet entête, le navigateur sait quel type de document il va recevoir.
- la ligne 8 indique que le document va être envoyé par morceaux. On ne les voit pas ici. Pour chacun d'eux, le serveur indique d'abord combien d'octets il va envoyer puis il envoie ceux-ci.
- lorsque le navigateur a reçu l'intégralité du document, il peut soit le copier sur le disque soit proposer sa visualisation s'il existe sur la machine de réception un programme capable de visualiser le type de document reçu. C'est pourquoi ci-dessus, le navigateur Firefox a proposé soit la visualisation du document par Excel qui est présent sur la machine cliente, soit la sauvegarde du document sur le disque.

Le lecteur pourra vérifier que le document Excel reçu est bien celui construit par la méthode [buildExcelDocument] que nous avons commentée.

3.4 La vue PDF

Le fichier [vues.properties] (cf page 38) indique que la vue nommée "pdf" doit être rendue par un objet de type [PdfView]. Cette classe, que nous allons construire, dérive de la classe prédéfinie [AbstractPdfView] :

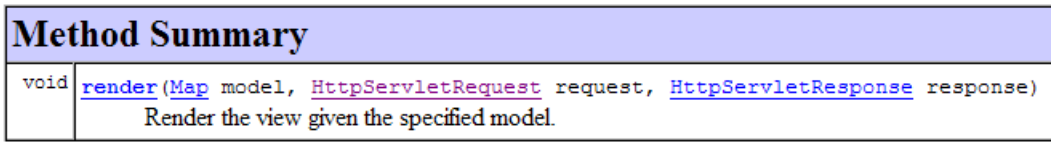

org.springframework.web.servlet.view.document

Class AbstractPdfView

java.lang.Object
└ org.springframework.context.support.ApplicationObjectSupport
└ org.springframework.web.context.support.WebApplicationObjectSupport
└ org.springframework.web.servlet.view.AbstractView
└ org.springframework.web.servlet.view.document.AbstractPdfView

All Implemented Interfaces:
BeanNameAware, ApplicationContextAware, View

On voit que la classe [AbstractPdfView] implémente l'interface [View], condition que doit remplir toute classe chargée de rendre une vue. On sait que cette interface n'a qu'une méthode :

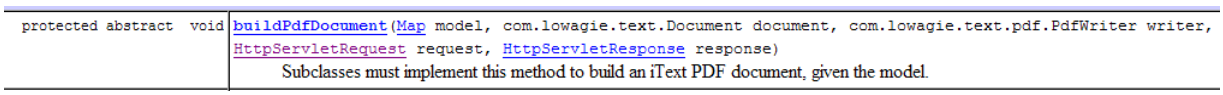

Method Summary

void	render(Map model, HttpServletRequest request, HttpServletResponse response) Render the view given the specified model.

La méthode [render] de la classe [AbstractPdfView] va :

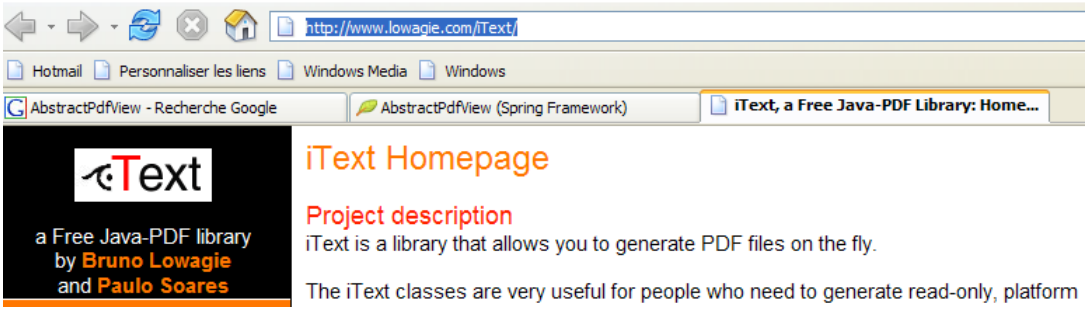
1. construire un flux PDF et mettre le modèle [model] dedans
2. encapsuler ce flux PDF dans un flux HTTP
3. envoyer le flux HTTP au client grâce à l'objet [response]

La classe [AbstractPdfView] assure elle-même les étapes 2 et 3. L'étape 1 est assurée par la méthode abstraite [buildPdfDocument] de la classe [AbstractPdfView] :

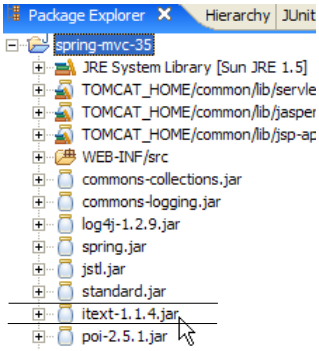

protected abstract void	buildPdfDocument(Map model, com.lowagie.text.Document document, com.lowagie.text.pdf.PdfWriter writer, HttpServletRequest request, HttpServletResponse response) Subclasses must implement this method to build an iText PDF document, given the model.

On y retrouve les trois paramètres de la méthode [render]. Le paramètre [com.lowagie.text.Document document] représente un document PDF vide. Le modèle [model] doit être inséré dans celui-ci. La classe générique [AbstractPdfView] ne peut savoir comment, car cela dépend bien évidemment de chaque application. La classe [buildPdfDocument] est donc déclarée abstraite et doit être redéfinie dans une classe dérivée. Celle-ci n'a que cette méthode à redéfinir.

Le type [com.lowagie.text.Document] n'est pas un type Spring. Spring s'appuie ici sur un projet tiers appelé **iText** et disponible à l'url [http://www.lowagie.com/iText/] :



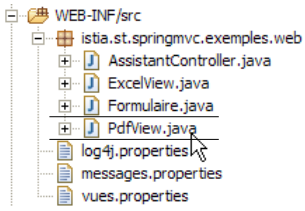
Les classes de ce projet sont disponibles dans une archive que nous devons inclure dans le *ClassPath* du projet [spring-mvc-35] :



La classe dérivée de [AbstractPdfView] chargée d'afficher la vue nommé "pdf" est de type [PdfView] comme indiqué dans le fichier [vues.properties] (cf page 38):

```
1.  #pdf
2.  pdf.class=istia.st.springmvc.exemples.web.PdfView
```

Cette classe est définie dans [WEB-INF/src] :



Son code est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.awt.Color;
4.  import java.util.Map;
5.
6.  import javax.servlet.http.HttpServletRequest;
7.  import javax.servlet.http.HttpServletResponse;
8.
9.  import org.springframework.web.servlet.view.document.AbstractPdfView;
10.
11. import com.lowagie.text.Document;
12. import com.lowagie.text.Element;
13. import com.lowagie.text.Paragraph;
14. import com.lowagie.text.pdf.PdfPCell;
15. import com.lowagie.text.pdf.PdfPTable;
16. import com.lowagie.text.pdf.PdfWriter;
17.
18. public class PdfView extends AbstractPdfView {
19.
20.  protected void buildPdfDocument(Map modèle, Document doc, PdfWriter writer, HttpServletRequest
    request, HttpServletResponse response) throws Exception {
21.    // on récupère le formulaire dans le modèle
22.    Formulaire form=(Formulaire)modèle.get("formulaire");
23.    // une table à 2 colonnes
```

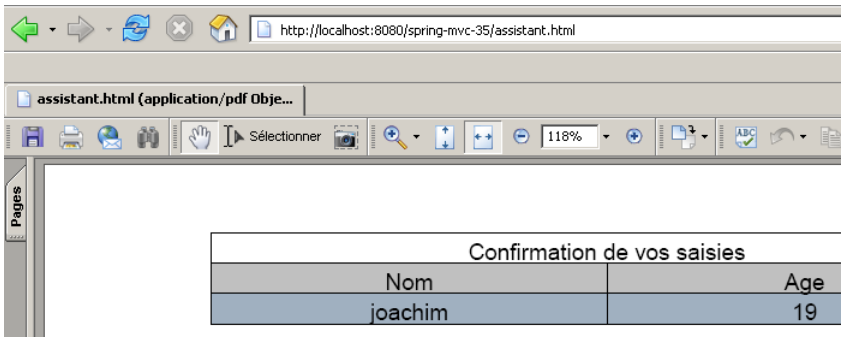
```
24.     PdfPTable table = new PdfPTable(2);
25.     // une cellule
26.     PdfPCell cell = new PdfPCell(new Paragraph("Confirmation de vos saisies"));
27.     // qui prendra deux colonnes
28.     cell.setColspan(2);
29.     // et sera centré
30.     cell.setHorizontalAlignment(Element.ALIGN_CENTER);
31.     // ajout de la cellule à la table
32.     table.addCell(cell);
33.     // 1ère ligne, 1ère colonne
34.     cell=new PdfPCell(new Paragraph("Nom"));
35.     cell.setBackgroundColor(new Color(0xC0, 0xC0, 0xC0));
36.     cell.setHorizontalAlignment(Element.ALIGN_CENTER);
37.     table.addCell(cell);
38.     // 1ère ligne, 2ième colonne
39.     cell=new PdfPCell(new Paragraph("Age"));
40.     cell.setBackgroundColor(new Color(0xC0, 0xC0, 0xC0));
41.     cell.setHorizontalAlignment(Element.ALIGN_CENTER);
42.     table.addCell(cell);
43.     // 2ième ligne, 1ère colonne
44.     cell=new PdfPCell(new Paragraph(form.getNom()));
45.     cell.setBackgroundColor(new Color(0xA0, 0xB0, 0xC0));
46.     cell.setHorizontalAlignment(Element.ALIGN_CENTER);
47.     table.addCell(cell);
48.     // 2ième ligne, 2ième colonne
49.     cell=new PdfPCell(new Paragraph(""+form.getAge()));
50.     cell.setBackgroundColor(new Color(0xA0, 0xB0, 0xC0));
51.     cell.setHorizontalAlignment(Element.ALIGN_CENTER);
52.     table.addCell(cell);
53.     // ajout de la table au document
54.     doc.add(table);
55.     }
56. }
```

- ligne 18 : la classe [PdfView] dérive de la classe [AbstractPdfView] de Spring
- ligne 20 : la méthode abstraite [buildPdfDocument] de [AbstractPdfView] est redéfinie

Rappelons que la méthode [buildPdfDocument] reçoit un document PDF vide qu'elle doit remplir avec le modèle. Ce document est le paramètre [Document doc] de la méthode. Pour construire le document PDF, il faut consulter la documentation du projet **iText**. Un bon point de départ est l'url [http://itextdocs.lowagie.com/tutorial/] (avril 2006) :

- Part I: General Use of iText
- Part II: Using High Level Objects
- Part III: Fonts
- Part IV: Direct Content
- Part V: Interactive Features (AcroForms)
- Part VI: the iText Toolbox explained
- Part VII: RTF
- Part VIII: HTML
- Part IX: XML
- Part X: Under the hood

Nous commentons les grandes lignes de la méthode [buildPdfDocument] qui vise à construire le document PDF suivant :

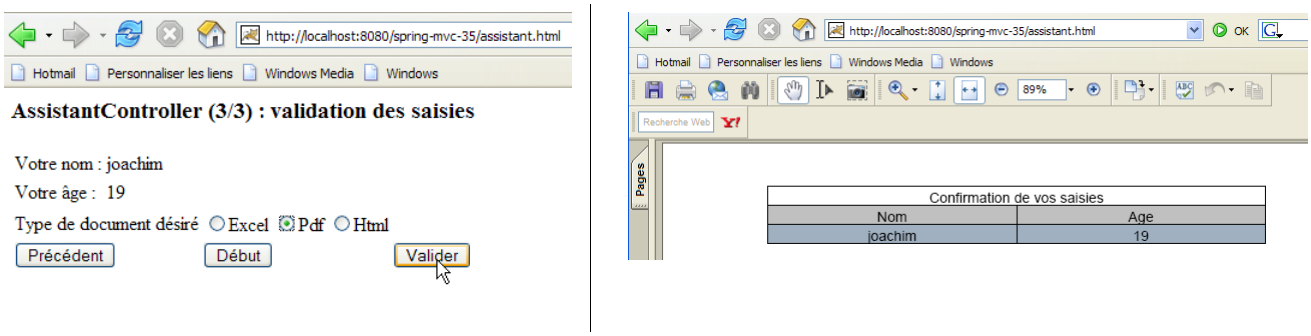


- ligne 22 : récupère le modèle à rendre. Nous savons que c'est une instance de la classe [Formulaire].
- ligne 24 : crée la table à deux colonnes qui va contenir le modèle. On ne précise pas le nombre de lignes de la table. La table va être remplie de cellules. L'ordre de remplissage par défaut est un remplissage ligne par ligne, de la droite vers la gauche. Une cellule est ajoutée au premier emplacement libre de la ligne courante. Si celle-ci est pleine, une nouvelle ligne est créée et la cellule est placée dans la première colonne de cette ligne.
- ligne 26 : crée une cellule PDF et met dedans un texte sous la forme d'un paragraphe.

- ligne 28 : la cellule précédente va s'étaler sur deux colonnes
- ligne 30 : le texte de la cellule sera centré horizontalement
- ligne 32 : la cellule créée ligne 26 est ajoutée à la table. Elle en forme la première ligne.
- lignes 34-37 : la cellule (1,0) de la table est définie – contient le texte "Nom"
- lignes 39-42 : la cellule (1,1) de la table est définie – contient le texte "Age"
- lignes 44-47 : la cellule (2,0) de la table est définie – contient la valeur du champ [formulaire.nom]
- lignes 49-52 : la cellule (2,1) de la table est définie – contient la valeur du champ [formulaire.age]
- ligne 54 : la table créée ligne 24 est ajoutée au document PDF

C'est tout. Le modèle [model] a été mis dans un document PDF. La méthode [render] de [AbstractPdfView] va se charger d'encapsuler celui-ci dans la réponse HTTP envoyée au client.

Aux tests, cela donne les choses suivantes :



Si on regarde le flux HTTP envoyé au client, avec un outil tel que LiveHTTPHeaders, on obtient la chose suivante :

```
1.  HTTP/1.x 200 OK
2.  Server: Apache-Coyote/1.1
3.  Pragma: No-cache
4.  Expires: Thu, 01 Jan 1970 00:00:00 GMT
5.  Cache-Control: no-cache, no-store
6.  Content-Type: application/pdf;charset=ISO-8859-1
7.  Content-Language: fr-FR
8.  Content-Length: 1162
9.  Date: Mon, 17 Apr 2006 12:57:08 GMT
```

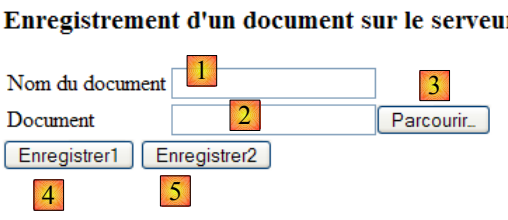
- la ligne 6 indique qu'après les entêtes HTTP (lignes 1-9), le serveur va envoyer un document de type MIME [application/pdf]. A réception de cet entête, le navigateur sait quel type de document il va recevoir.
- la ligne 8 indique que le document PDF comporte 1162 octets.
- lorsque le navigateur a reçu l'intégralité du document, il peut soit le copier sur le disque soit proposer sa visualisation s'il existe sur la machine de réception un programme capable de visualiser le type de document reçu. Pour les documents PDF, les navigateurs récents ont la plupart du temps un plugin permettant de visualiser le document PDF à l'intérieur du navigateur.

Le lecteur pourra vérifier que le document PDF reçu est bien celui construit par la méthode [buildPdfDocument] que nous avons commentée.

4 Téléchargement de documents vers le serveur

4.1 Introduction

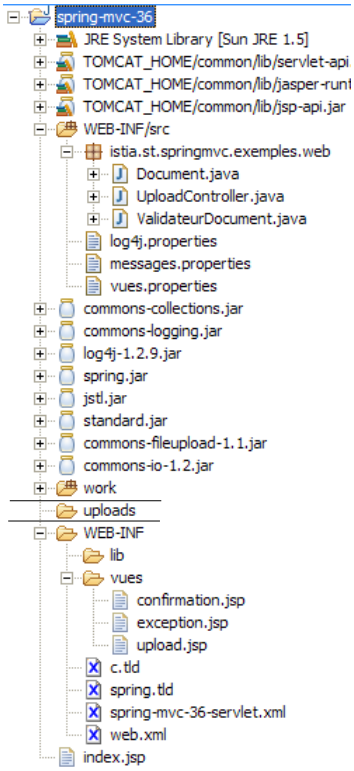
Spring offre des outils pour faciliter le téléchargement de documents du client vers le serveur. Pour les présenter, nous allons étudier l'application suivante :



La page principale de l'application est un formulaire destiné à désigner le document local (c.a.d. un document présent sur le disque du client) à télécharger sur le serveur. Les éléments HTML du formulaire ci-dessus sont les suivants :

n°	type HTML	nom	rôle
1	<input type= "text ">	nom	donner un nom au document qui va être téléchargé sur le serveur
2-3	<input type= "file ">	contenu	l'ensemble boîte de saisie – bouton permettant de désigner le document à télécharger. Son chemin peut être indiqué soit en : • le tapant dans [2] • le désignant avec le bouton [3]
4	<input type= "submit ">	action	provoque le téléchargement sur le serveur du document précisé en [2] avec l'identifiant précisé en [1].
4	<input type= "submit ">	action	idem que précédemment mais côté serveur, la méthode utilisée pour enregistrer le document sera différente.

Le projet Eclipse / Tomcat s'appellera (spring-mvc-35) et sera le suivant :



Le dossier [uploads] sera le dossier où seront placés les documents téléchargés sur le serveur. Le formulaire de l'application présenté page 47 peut être traité par un formulaire de type [MultiActionController] puisqu'il comporte deux boutons de type [Submit]. On peut aussi le traiter avec un simple contrôleur de type [SimpleFormController] et c'est ce que nous faisons ici.

Le fichier de configuration [spring-mvc-36-servlet.xml] de l'application 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="upload.html">UploadController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- le résolveur de vues -->
13.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
14.     <property name="basename">
15.       <value>vues</value>
16.     </property>
```

```
17.   </bean>
18.   <!-- le contrôleur de l'application-->
19.   <bean id="UploadController"
20.     class="istia.st.springmvc.exemples.web.UploadController">
21.     <property name="sessionForm">
22.       <value>true</value>
23.     </property>
24.     <property name="commandClass">
25.       <value>istia.st.springmvc.exemples.web.Document</value>
26.     </property>
27.     <property name="commandName">
28.       <value>document</value>
29.     </property>
30.     <property name="formView">
31.       <value>upload</value>
32.     </property>
33.     <property name="validator">
34.       <ref local="validateur"/>
35.     </property>
36.   </bean>
37.   <!-- validateur -->
38.   <bean id="validateur" class="istia.st.springmvc.exemples.web.ValidateurDocument"/>
39.   <!-- le gestionnaire d'exceptions -->
40.   <bean
41.     class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
42.     <property name="exceptionAttribute">
43.       <value>exception</value>
44.     </property>
45.     <property name="defaultStatusCode">
46.       <value>200</value>
47.     </property>
48.     <property name="defaultErrorView">
49.       <value>exception</value>
50.     </property>
51.   </bean>
52.   <!-- upload de fichiers -->
53.   <bean id="multipartResolver"
54.     class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
55.     <property name="maxUploadSize">
56.       <value>1000000</value>
57.     </property>
58.   </bean>
59.   <!-- le fichier des messages -->
60.   <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
61.     <property name="basename">
62.       <value>messages</value>
63.     </property>
64.   </bean>
65. </beans>
```

- ligne 8 : l'unique url traitée par l'application est [/upload.html]. Cette url est traitée par un contrôleur défini lignes 19-39.
- lignes 19-39 : définissent l'unique contrôleur de l'application
- ligne 20 : le contrôleur est de type [UploadController], un type propriétaire dérivé du type Spring [SimpleFormController]. Nous avons déjà étudié ce type de contrôleur dans l'article 2.
- lignes 24-26 : définissent le conteneur des saisies du formulaire. Il est de type [Document] un type propriétaire qui stockera deux informations :
    - l'identifiant donné au document téléchargé (saisie [1])
    - le contenu du document téléchargé (saisie [2])
- lignes 27-29 : la clé via laquelle sera accessible le document téléchargé dans la vue définie par la propriété [formView] (lignes 30-32) qui est la vue du formulaire de saisies.
- lignes 30-32 : le nom de la vue qui contient le formulaire de saisies. Ici la vue s'appelle " upload ".
- lignes 33-35 : le validateur chargé de vérifier la validité des données postées
- lignes 21-23 : la mise en session ou non du conteneur de saisies entre son GET et son POST. Ici, il restera en session.

- ligne 38 : le validateur référencé par le contrôleur [UploadController]. C'est une classe propriétaire que nous serons amenés à écrire.

- lignes 40-51 : le gestionnaire des exceptions qui remonteront jusqu'à [DispatcherServlet]. Ici, nous indiquons que la vue nommée " exception " devra être affichée (ligne 43). On notera ligne 46, le code d'état de la réponse HTTP. Dans les exemples de l'article 2, on mettait 500 comme code qui est le code " Internal server error " signalant que le serveur a rencontré un problème. Si le navigateur [Firefox] accepte d'afficher une page envoyée avec le code 500, ce n'est pas le cas d'Internet Explorer qui lui, affiche sa propre page d'erreur au lieu de celle envoyée par le serveur. Donc certains exemples de l'article 2 ne fonctionnent pas avec IE. Ici, le code 200 signifiant "OK", la page envoyée par le serveur sera correctement affichée par IE.

- lignes 13-17 : le résolveur de noms de vues, ici [ResourceBundleViewResolver]. Les vues seront définies dans un fichier [vues.properties]. Ce sera le cas des vues nommées " upload " et " confirmation " définies par le contrôleur [UploadController] ainsi que de la vue " exception " définie par le gestionnaire d'exceptions [SimpleMappingExceptionResolver].

- lignes 60-64 : le gestionnaire des messages, notamment des messages d'erreurs. Comme d'habitude, c'est [ResourceBundleMessageSource] et les messages seront placés dans un fichier [messages.properties] qui sera cherché dans le *Classpath* de l'application. Jusqu'à maintenant, cette ressource était définie dans [applicationContext.xml]. Nous innovons en le plaçant dans le fichier [spring-mvc-36-servlet.xml]. Quelle est la différence ? Dans une architecture 3tier, les couches [métier] et [dao] seront définies dans le fichier [applicationContext.xml]. Il se peut que ces couches aient également besoin d'une référence sur le fichier des messages. Si c'était le cas, celui-ci devrait être défini dans [applicationContext.xml].

Tout ce qui précède a déjà été rencontré au moins une fois dans l'une ou l'autre des applications étudiées jusqu'à maintenant. La seule nouveauté est dans les lignes 53-58 qui définissent l'objet chargé de gérer les téléchargements de fichiers du client vers le serveur.

```
1.   <!-- upload de fichiers -->
2.   <bean id="multipartResolver"
3.     class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
4.     <property name="maxUploadSize">
5.       <value>1000000</value>
6.     </property>
7.   </bean>
```

Ces lignes définissent un intercepteur de type [MultipartResolver] qui va participer au traitement de la requête du client, si celle-ci est de type " multipart ". Nous étudions maintenant ce type d'intercepteur.

4.2 Les intercepteurs de type [MultipartResolver]

Le formulaire HTML d'un téléchargement de fichiers vers le serveur a un type spécial :

```
<form method="post" enctype="multipart/form-data">
```

L'attribut [enctype] indique au serveur qu'il va recevoir un document en plusieurs parties. Dans ce document, il y aura :
- les valeurs postées, ici les paramètres " nom ", " contenu " et " action "
- le contenu du document téléchargé

Lorsque ce type de formulaire est posté au serveur, la requête HTTP faite au serveur par le client a la forme suivante :

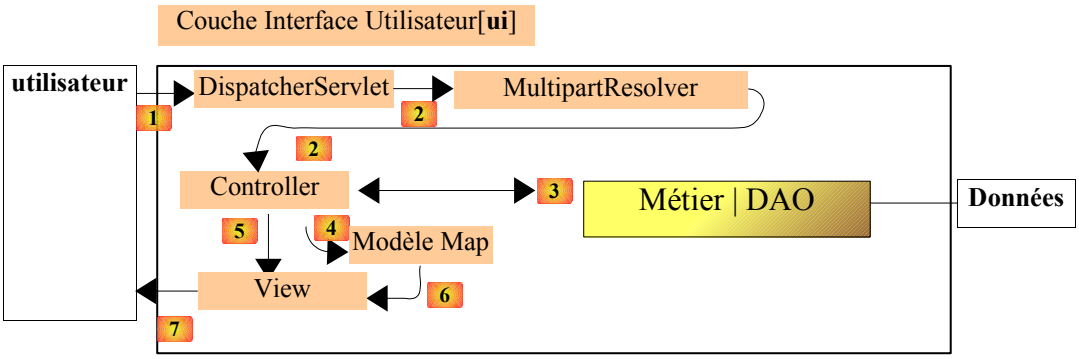
```
1.  POST /spring-mvc-36/upload.html HTTP/1.1
2.  Host: localhost:8080
3.  User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.1; fr; rv:1.8.0.2) Gecko/20060308
    Firefox/1.5.0.2
4.  Accept:
    text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image/png,*/*;q=0.
    5
5.  Accept-Language: fr-fr,fr;q=0.8,en;q=0.6,en-us;q=0.4,de;q=0.2
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-36/upload.html
11. Cookie: JSESSIONID=4950824B472EEEB6B95751E537DB4D3F
12. Content-Type: multipart/form-data; boundary=---------------------------23281168279961
13. Content-Length: 1543
14.
15. -----------------------------23281168279961
16. Content-Disposition: form-data; name="nom"
17.
18. upload
19. -----------------------------23281168279961
20. Content-Disposition: form-data; name="contenu"; filename="upload.jsp"
21. Content-Type: application/octet-stream
22.
23. ....
24. -----------------------------23281168279961
25. Content-Disposition: form-data; name="action"
26.
27. Enregistrer1
28. -----------------------------23281168279961--
```

- ligne 12 : le serveur est averti qu'il va recevoir un document au format " multipart ". Ce format ce caractérise par des envois de paquets délimités par un marqueur. Celui-ci est la valeur de l'attribut " boundary " de la ligne 12 qu'on retrouve aux lignes 15, 19, 24 et 28.
- lignes 16-18 : envoi du paramètre [nom] et de sa valeur [upload]
- lignes 20-23 : envoi du flux d'octets du document téléchargé.
    - ligne 20 : envoi du paramètre [contenu] et de sa valeur [upload.jsp]. Si le chemin du document est [/elmt1/elmt2/.../**elmntN**], seule la dernière partie du chemin [**elmntN**] est envoyée au serveur.
    - ligne 21 : indique au serveur qu'il va recevoir un flux d'octets
    - lignes 22-23 : le flux d'octets du document [upload.jsp]
- lignes 25-27 : envoi du paramètre [action] et de sa valeur [Enregistrer1]

Lorsque le contrôleur général [DispatcherServlet] reçoit une requête HTTP avec l'attribut [Content-type] suivant :

```
Content-Type: multipart/form-data; boundary=---------------------------23281168279961
```

il cherche si l'application a défini un bean d'id [multipartResolver]. Si oui, ce bean sera chargé de traiter la requête de type " multipart ". On peut voir le bean [multipartResolver] comme une classe intercepteur :



- [DispatcherServlet] transmet la requête du client sous la forme d'un objet [HttpServletRequest] à l'intercepteur [MultipartResolver]
- [MultipartResolver] transforme cet objet en un objet [MultipartHttpServletRequest] et le transmet à l'objet [Controller] chargé de traiter la requête du client. Celui-ci, va alors avoir un accès simplifié au document téléchargé

Spring offre deux classes capables d'implémenter l'intercepteur [MultipartResolver] ci-dessus :

org.springframework.web.multipart

Interface MultipartResolver

All Known Implementing Classes:

CommonsMultipartResolver, CosMultipartResolver

Les classes [CommonsMultiPartResolver] et [CosMultiPartResolver] implémentent l'interface [MultiPartResolver] suivante :

Method Summary	
void	cleanupMultipart(MultipartHttpServletRequest request) Cleanup any resources used for the multipart handling, like a storage for the uploaded files.
boolean	isMultipart(HttpServletRequest request) Determine if the request contains multipart content.
MultipartHttpServletRequest	resolveMultipart(HttpServletRequest request) Parse the given HTTP request into multipart files and parameters, and wrap the request inside a MultipartHttpServletRequest object that provides access to file descriptors and makes contained parameters accessible via the standard ServletRequest methods.

La méthode la plus importante est [resolveMultipart] qui à partir de la requête multipart reçue par [DispatcherServlet] va créer une instance de type [MultipartHttpServletRequest] :

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

org.springframework.web.multipart

Interface MultipartHttpServletRequest

All Superinterfaces:

HttpServletRequest, ServletRequest

All Known Implementing Classes:

AbstractMultipartHttpServletRequest, CosMultipartHttpServletRequest, DefaultMultipartHttpServletRequest

[MultipartHttpServletRequest] est une interface dérivée de l'interface [HttpServletRequest]. Nous décrivons certaines des méthodes utiles pour obtenir des informations sur le document téléchargé :

MultipartFile	getFile(String name) Return the contents plus description of an uploaded file in this request, or null if it does not exist.

- **name** : le nom du champ HTML de type " file " qui a servi au téléchargement. Dans notre exemple c'est " contenu ".
- **MultipartFile** : une interface qui donne accès au contenu du document téléchargé ainsi qu'à son nom

Method Summary	
byte[]	getBytes() Return the contents of the file as an array of bytes.
String	getContentType() Return the content type of the file.
InputStream	getInputStream() Return an InputStream to read the contents of the file from.
String	getName() Return the name of the parameter in the multipart form.
String	getOriginalFilename() Return the original filename in the client's filesystem.
long	getSize() Return the size of the file in bytes.
boolean	isEmpty() Return whether the uploaded file is empty in the sense that no file has been chosen in the multipart form.
void	transferTo(File dest) Transfer the received file to the given destination file.

Ainsi nous pourrons :

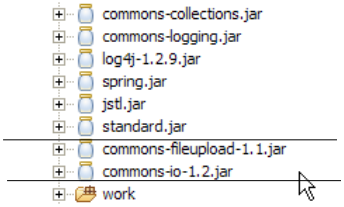
- obtenir le nom primitif du document téléchargé par la méthode [getOriginalFileName]
- savoir si l'utilisateur a choisi un document non vide par la méthode [isEmpty]
- copier le document téléchargé dans le système de fichiers du serveur avec la méthode [transferTo]
- obtenir le contenu du document téléchargé par la méthode [getBytes]

Revenons sur la configuration de l'intercepteur [MultipartResolver] de notre application :

```
1.   <!-- upload de fichiers -->
2.   <bean id="multipartResolver"
3.     class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
4.     <property name="maxUploadSize">
5.       <value>1000000</value>
6.     </property>
7.   </bean>
```

- ligne 2 : d'après le Javadoc, la valeur de l'id est importante. Sa valeur " multipartResolver " indique que le bean correspondant est un intercepteur [MultipartResolver].
- ligne 3 : l'implémentation choisie ici pour l'intercepteur est [CommonsMultipartResolver], une classe qui s'appuie sur le projet Apache " commons fileUpload " disponible à l'url [http://jakarta.apache.org/commons/fileupload/] :

L'utilisation du projet " commons fileUpload " nécessite l'ajout d'archives dans le *Classpath* de l'application, [commons-fileupload] et [commons-io] :



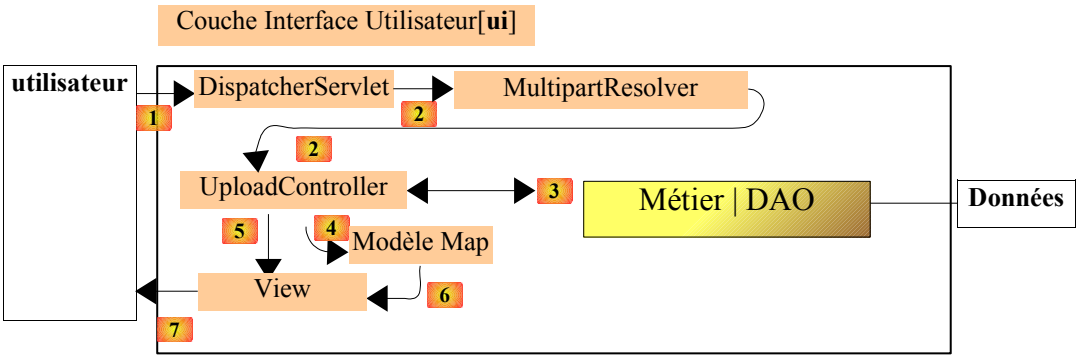
La classe [CommonsMultipartResolver] dispose de plusieurs méthodes *set* pour configurer l'intercepteur [MultipartResolver] :

void	**setMaxInMemorySize**(int maxInMemorySize) Set the maximum allowed size (in bytes) before uploads are written to disk.
void	**setMaxUploadSize**(long maxUploadSize) Set the maximum allowed size (in bytes) before uploads are refused
void	**setServletContext**(ServletContext servletContext) Set the ServletContext that this object runs in.
void	**setUploadTempDir**(Resource uploadTempDir) Set the temporary directory where uploaded files get stored.

La méthode [setMaxUploadSize] sert à fixer la taille maximale des documents téléchargés. Dans notre application, nous avons fixé à 1 Mo cette taille (ligne 5 de la configuration du MultipartResolver).

4.3 Ecriture du contrôleur [UploadController]

Replaçons le contrôleur [UploadController] dans la chaîne de traitement de la requête " multipart " qu'il est chargé de traiter :



et revenons sur sa configuration :

```
1.<!-- le contrôleur de l'application-->
2.  <bean id="UploadController"
3.    class="istia.st.springmvc.exemples.web.UploadController">
4.    <property name="sessionForm">
5.      <value>true</value>
6.    </property>
7.    <property name="commandClass">
8.      <value>istia.st.springmvc.exemples.web.Document</value>
9.    </property>
10.   <property name="commandName">
11.     <value>document</value>
12.   </property>
```

```
13.   <property name="formView">
14.     <value>upload</value>
15.   </property>
16.   <property name="validator">
17.     <ref local="validateur"/>
18.   </property>
19. </bean>
```

Lignes 7-10, nous voyons que le conteneur des saisies est de type [Document]. Ce type défini dans [WEB-INF/src] est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  public class Document {
4.    // le nom du document
5.    String nom;
6.
7.    // son contenu
8.    byte[] contenu;
9.
10.   // getters et setters
11. ...
12. }
```

Rappelons les deux champs de saisie du formulaire HTML :

```
1.      <form method="post" enctype="multipart/form-data">
2.        ...
3.        <input name="nom" type="text" value="">
4.        ...
5.        <input name="contenu" type="file" size="80">
6.        ...
7.        <input type="submit" name="action" value="Enregistrer1"></td>
8.        <input type="submit" name="action" value="Enregistrer2"></td>
9.  </form>
```

Si on rapproche ce formulaire du conteneur [Document], on voit que la valeur associée au champ HTML :

1. " nom " sera mémorisée dans le champ [nom] de type String de [Document].
2. " contenu " sera mémorisée dans le champ [contenu] de type byte[] de [Document].

On se rappelle que Spring MVC dispose de conversions implicites des valeurs postées, qui sont de simples chaînes de caractères (sauf dans le cas du champ HTML de type [file]), vers des types autres que String, telles que String -> File, String -> Class, etc... Lorsqu'il n'existe pas de conversion implicite, on peut déclarer des classes de conversion dans la méthode [initBinder] du contrôleur chargé de traiter la requête. Certaines de ces classes sont fournies par Spring. C'est ainsi que dans un exemple de l'article 2, nous avions déclaré dans [initBinder], la classe Spring [CustomDateEditor] capable de faire les conversions String -> Date :

```
binder.registerCustomEditor(java.util.Date.class, null,new CustomDateEditor(dateFormat, false));
```

On appelle ces classes de conversion des éditeurs de propriété, une dénomination provenant de la norme Javabean. Dans le cas d'une requête " multipart ", il nous faut un éditeur de propriétés particulier qui soit capable d'extraire les valeurs postées du flux multipart envoyé par le client et de les affecter aux champs de mêmes noms dans le conteneur de saisies. Cet éditeur est fourni par Spring MVC et s'appelle [ByteArrayMultipartFileEditor]. Avec cet éditeur, le flux " multipart " envoyé par le client que nous avons étudié précédemment :

```
1.  -----------------------------23281168279961
2.  Content-Disposition: form-data; name="nom"
3.
4.  upload
5.  -----------------------------23281168279961
6.  Content-Disposition: form-data; name="contenu"; filename="upload.jsp"
7.  Content-Type: application/octet-stream
8.
9.  ....
10. -----------------------------23281168279961
11. Content-Disposition: form-data; name="action"
12.
13. Enregistrer1
14. -----------------------------23281168279961--
```

va affecter :

- la valeur " upload " au champ [Document].nom (lignes 2-4)
- le contenu du document téléchargé (upload.jsp) au champ [Document].contenu (lignes 6-9)

Le code du contrôleur [UploadController] est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.io.DataOutputStream;
4.  import java.io.File;
5.  import java.io.FileOutputStream;
6.  import java.io.IOException;
7.  import java.util.HashMap;
8.  import java.util.Map;
9.
10. import javax.servlet.http.HttpServletRequest;
11. import javax.servlet.http.HttpServletResponse;
12.
13. import org.springframework.validation.BindException;
14. import org.springframework.web.bind.RequestUtils;
15. import org.springframework.web.bind.ServletRequestDataBinder;
16. import org.springframework.web.multipart.MultipartFile;
17. import org.springframework.web.multipart.MultipartHttpServletRequest;
18. import org.springframework.web.multipart.support.ByteArrayMultipartFileEditor;
19. import org.springframework.web.servlet.ModelAndView;
20. import org.springframework.web.servlet.mvc.SimpleFormController;
21.
22. public class UploadController extends SimpleFormController {
23.
24.  // upload de document
25.  protected ModelAndView onSubmit(HttpServletRequest request,
26.      HttpServletResponse response, Object command, BindException errors)
27.      throws Exception {
28.    // on récupère le document
29.    Document document = (Document) command;
30.    // on récupère la requête multipart
31.    MultipartHttpServletRequest multipartRequest = (MultipartHttpServletRequest) request;
32.    // on récupère une référence sur le fichier chargé
33.    MultipartFile fichier = multipartRequest.getFile("contenu");
34.    // fichier désigné ?
35.    if (fichier.isEmpty()) {
36.      // le fichier n'a pas été désigné
37.      errors.rejectValue("contenu", "document.contenu.obligatoire");
38.      // on réaffiche le formulaire avec les erreurs
39.      return showForm(request, response, errors);
40.    }
41.    // on récupère le nom du fichier chargé
42.    String nomDocument = fichier.getOriginalFilename();
43.    // le chemin d'enregistrement du document
44.    String chemin = request.getSession().getServletContext().getRealPath(
45.        "/")
46.        + "uploads\\" + nomDocument;
47.    // on récupère le nom de l'action en cours
48.    String action = RequestUtils.getStringParameter(request, "action",
49.        "enregistrer1").toLowerCase();
50.    // selon l'action, deux méthodes différentes d'enregistrement
51.    if (action.equals("enregistrer1")) {
52.      doEnregistrer1(fichier, chemin);
53.    } else {
54.      doEnregistrer2(document, chemin);
55.    }
56.    // confirmation de l'enregistrement
57.    Map modèle = new HashMap();
58.    modèle.put("document", document);
59.    modèle.put("nomDocument", nomDocument);
60.    return new ModelAndView("confirmation", modèle);
61.  }
62.
63.  // enregistrement - méthode 1 - à partir de la requête
64.  private void doEnregistrer1(MultipartFile fichier, String chemin)
65.      throws IllegalStateException, IOException {
66.    // on enregistre le flux [fichier] sur le serveur à l'emplacement
67.    // [chemin]
68.    fichier.transferTo(new File(chemin));
69.  }
70.
71.  // enregistrement - méthode 2 - à partir du document
72.  private void doEnregistrer2(Document document, String chemin)
73.      throws IOException {
74.    // on enregistre le document [document] sur le serveur à l'emplacement
75.    // [chemin]
76.    DataOutputStream out = new DataOutputStream(
77.        new FileOutputStream(chemin));
78.    out.write(document.getContenu());
79.    out.close();
80.  }
81.
82.  // enregistrement éditeur de propriété
83.  protected void initBinder(HttpServletRequest request,
```

```
84.      ServletRequestDataBinder binder) throws Exception {
85.    // pour que le flux multipart posté soit directement affecté au champ de type byte[]
86.    binder.registerCustomEditor(byte[].class,
87.        new ByteArrayMultipartFileEditor());
88.  }
89.
90. }
```

- ligne 22 : [UploadController] dérive de [SimpleFormController]
- lignes 83-88 : la méthode [initBinder] dans laquelle nous déclarons l'éditeur de propriétés [ByteArrayMultipartFileEditor] pour le type byte[]
- lignes 25-61 : la méthode [onSubmit] qui sera exécutée si le conteneur de saisies [Object command] a été déclaré valide par le validateur de type [ValidateurDocument] lié au contrôleur :

```
1.   <!-- le contrôleur de l'application-->
2.   <bean id="UploadController"
3.     class="istia.st.springmvc.exemples.web.UploadController">
4.  ...
5.     <property name="validator">
6.       <ref local="validateur"/>
7.     </property>
8.   </bean>
9.   <!-- validateur -->
10.  <bean id="validateur" class="istia.st.springmvc.exemples.web.ValidateurDocument"/>
```

Nous aurons l'occasion de revenir sur ce validateur de données. Rappelons que l'objectif de la méthode [onSubmit] est de :

- terminer le traitement la requête [HttpServletRequest request]. Ici il s'agira de sauvegarder le document téléchargé dans le dossier [uploads] de l'application web, avec le même nom qu'il avait sur la machine cliente.
- retourner un objet [ModelAndView] au contrôleur général [DispatcherServlet]. Ici nous afficherons soit une page de confirmation de la sauvegarde soit le formulaire de téléchargement avec un message d'erreur s'il y a eu un problème.

- ligne 29 : on récupère le conteneur des saisies de type [Document]. Celui-ci contient les champs [nom] et [contenu] contenant respectivement l'identifiant donné au document téléchargé et le contenu de celui-ci.
- ligne 31 : on récupère la requête dont a été extrait [Document]. Cette requête est de type [MultipartHttpServletRequest] et contient des informations supplémentaires à celles présentes dans [Document] et qui vont nous être utiles.
- ligne 33 : on récupère l'objet [MultipartFile fichier] décrivant le document téléchargé via le champ HTML " contenu "
- lignes 35-40 : on vérifie si l'utilisateur a bien désigné un fichier et s'il l'a désigné qu'il est non vide. Si ce n'est pas le cas, un message d'erreur de code " document.contenu.obligatoire " et associé au champ HTML " contenu " est ajouté à la liste d'erreurs [BindException errors] reçue en paramètre (ligne 37). Puis (ligne 39), on redemande l'affichage du formulaire. La méthode [showForm] rend un [ModelAndView] où la vue est celle définie par la propriété [formView] du contrôleur et où le modèle est formé de la liste des erreurs [BindException errors] qui lui est passée en paramètre ainsi que du conteneur de saisies défini par la propriété [commandClass] du contrôleur. Parce que le contrôleur a sa propriété [sessionForm] à *true*, le conteneur de saisies sera cherché dans la session. C'est donc le même objet que l'objet [Object command] reçu par la méthode [onSubmit].
- ligne 42 : on récupère le nom qu'avait le document sur le poste client. Si le document avait un chemin [elt1/elt2/.../eltN], le nom récupéré est [eltN].
- ligne 44 : on construit le chemin où sera sauvegardé le document sur le serveur. Il s'agit du chemin <contexte>/uploads/eltN, où <contexte> est le dossier racine de l'application web. Celle-ci est obtenue à partir de la requête [request] par

```
request.getSession().getServletContext().getRealPath("/")
```

- ligne 48 : on récupère la valeur du paramètre HTML nommé " action ". Rappelons où apparaît ce paramètre dans le formulaire HTML :

```
1.     <form method="post" enctype="multipart/form-data">
2.       ...
3.       <input name="nom" type="text" value="">
4.       ...
5.       <input name="contenu" type="file" size="80">
6.       ...
7.       <input type="submit" name="action" value="Enregistrer1"></td>
8.       <input type="submit" name="action" value="Enregistrer2"></td>
9.  </form>
```

La valeur du paramètre " action " sera " Enregistrer1 " ou " Enregistrer2 " selon le bouton utilisé pour valider le formulaire. On a voulu montrer deux techniques pour enregistrer le document sur le serveur. Le bouton [Enregistrer1]

va utiliser la requête [MultipartHttpServletRequest] alors que le bouton [Enregistrer2] va, lui, utiliser le conteneur de saisies.

- ligne 52 : enregistrement du document téléchargé à partir de l'objet [MultipartFile] extrait de la requête [MultipartHttpServletRequest]
- ligne 54 : enregistrement du document téléchargé à partir du conteneur de saisies de type [Document]

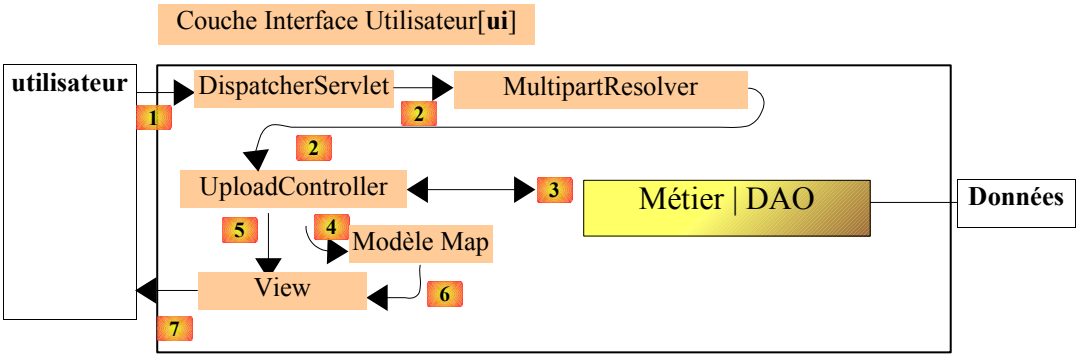
- lignes 57-60 : construction d'un objet [ModelAndView] à destination de [DispatcherServlet]. Le nom de la vue est " confirmation " (ligne 60). Deux éléments sont mis dans le modèle de cette vue :
  - le conteneur de saisies associé à la clé " document ", ligne 58
  - le nom qu'avait le document sur le poste client, ligne 9

- lignes 64-69 : enregistrement du document téléchargé encapsulé dans l'objet [MultipartFile fichier] à l'emplacement [String chemin]. On utilise la méthode [MultipartFile].transferTo pour faire ce travail.

- lignes 72-80 : enregistrement du document téléchargé encapsulé dans l'objet [Document document] à l'emplacement [String chemin]. On passe par l'intermédiaire d'un flux d'écriture [DataOutputStream] qui a une méthode [write] capable de placer un tableau d'octets (byte[]) dans son flux d'écriture.

Où en sommes-nous ?

- l'architecture du projet a été décrite :



- la configuration [spring-mvc-36-servlet.xml] du projet a été décrite :

```
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="upload.html">UploadController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- le résolveur de vues -->
13.   <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
14.     <property name="basename">
15.       <value>vues</value>
16.     </property>
17.   </bean>
18.   <!-- le contrôleur de l'application-->
19.   <bean id="UploadController"
20.     class="istia.st.springmvc.exemples.web.UploadController">
21. ...
22.     <property name="validator">
23.       <ref local="validateur"/>
24.     </property>
25.   </bean>
26.   <!-- validateur -->
27.   <bean id="validateur" class="istia.st.springmvc.exemples.web.ValidateurDocument"/>
28.   <!-- le gestionnaire d'exceptions -->
29.   <bean
30.     class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
31. ...
32.   </bean>
```

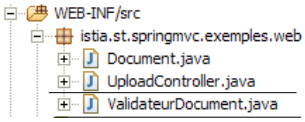
```
33.   <!-- upload de fichiers -->
34.   <bean id="multipartResolver"
35.     class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
36. ...
37.   </bean>
38.   <!-- le fichier des messages -->
39.   <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
40.     <property name="basename">
41.       <value>messages</value>
42.     </property>
43.   </bean>
44. </beans>
```

Il nous reste à expliciter :

- le validateur de données défini ligne 27
- les différentes vues de l'application définies dans [vues.properties] (lignes 13-17)
- le fichier des messages (lignes 39-43)

4.4 Le validateur des données postées

La classe [ValidateurDocument] est une classe propriétaire définie dans [WEB-INF/src] :



Son code est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import org.springframework.validation.Errors;
4.  import org.springframework.validation.Validator;
5.
6.  public class ValidateurDocument implements Validator {
7.
8.    public boolean supports(Class classe) {
9.      // valide les classes de type Document ou dérivé
10.     return classe.isAssignableFrom(Document.class);
11.   }
12.
13.   public void validate(Object objet, Errors erreurs) {
14.     // on récupère le document
15.     Document document = (Document) objet;
16.     // on vérifie son nom
17.     String nom = document.getNom();
18.     if (nom == null || nom.trim().length() == 0) {
19.       erreurs.rejectValue("nom", "document.nom.obligatoire");
20.     }
21.   }
22.
23. }
```

- lignes 8-11 : indiquent que le validateur ne sait valider que les types [Document] ou dérivés.
- lignes 13-20 : la méthode [validate] qui sera appelée par [DispatcherServlet] avant que la requête ne soit passée au contrôleur [UploadController].
- ligne 15 : le conteneur de saisies est récupéré en tant que type [Document]
- ligne 17 : récupère la valeur postée pour le champ HTML " nom "
- lignes 18-20 : vérifie que cette valeur est non vide. Si ce n'est pas le cas, une erreur de code "document.nom.obligatoire" est associée au champ HTML " nom " et mise dans la liste des erreurs [Errors erreurs] reçue en paramètre. On sait que lorsque [DispatcherServlet] récupère cette liste non vide, il réaffiche alors le formulaire initial en ajoutant à son modèle la liste des erreurs.

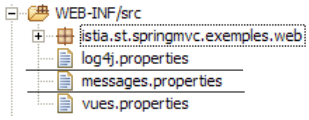
4.5 Le fichier des messages

D'après la configuration :

```
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>
```

les messages d'erreur seront trouvés dans un fichier " messages.properties " (ligne 4) situé dans le *ClassPath* de l'application web. Il a été placé dans [WEB-INF/src] :



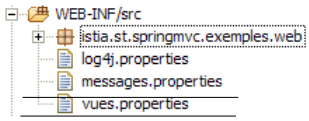
Son contenu est le suivant :

```
1.  typeMismatch=Erreur au chargement ...
2.  document.nom.obligatoire=Le nom est obligatoire ...
3.  document.contenu.obligatoire=Vous devez désigner un fichier non vide ...
```

Le code (2) est généré par le contrôleur [UploadController] et le code (3) par le validateur [ValidateurDocument]. Le code (1) est généré par Spring MVC lorsqu'il n'arrive pas à " caster " les valeurs postées dans les champs du conteneur [Document]. Normalement ce cas ne devrait pas se produire ici. On le voit apparaître lorsqu'on supprime la méthode [initBinder] du contrôleur [UploadController] qui définit l'éditeur de propriétés capable de " caster " les valeurs postées dans les champs du conteneur [Document]. Le lecteur est invité à faire ce test.

4.6 Les vues de l'application

Les vues sont définies dans [vues.properties] :

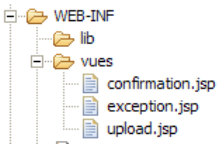


de la façon suivante :

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

- lignes 2-3 : la vue " upload " est la vue du formulaire de téléchargement
- lignes 5-6 : la vue " confirmation " est la vue qui confirme le succès du téléchargement
- lignes 8-9 : la vue " exception " utilisée par le gestionnaire d'exceptions lorsqu'une exception non gérée se produit.

Les pages JSP de ces trois vues ont été placées dans le dossier [WEB-INF/vues] :



Le code de la page [upload.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.
6.  <html>
7.  <head>
8.    <title>Upload (mvc-36)</title>
9.  </head>
```

```
10.  <body>
11.    <h3>Enregistrement d'un document sur le serveur</h3>
12.    <form method="post" enctype="multipart/form-data">
13.      <table>
14.        <tr>
15.          <td>Nom du document</td>
16.          <spring:bind path="document.nom">
17.            <td>
18.              <input name="${status.expression}" type="text" value="${status.value}">
19.            </td>
20.            <td>
21.              ${status.errorMessage}
22.            </td>
23.          </spring:bind>
24.        </tr>
25.        <tr>
26.          <td>Document</td>
27.          <spring:bind path="document.contenu">
28.            <td>
29.              <input name="${status.expression}" type="file" size="80">
30.            </td>
31.            <td>
32.              ${status.errorMessage}
33.            </td>
34.          </spring:bind>
35.        </tr>
36.      </table>
37.      <input type="submit" name="action" value="Enregistrer1"></td>
38.      <input type="submit" name="action" value="Enregistrer2"></td>
39.    </form>
40.  </body>
41.</html>
```

- on se rappellera que, par configuration, la clé "document " désigne le conteneur de saisies de type [Document]
- ligne 12 : le formulaire " multipart "
- lignes 16-23 : le champ HTML " nom " associé au champ " nom " du conteneur de saisies. Ce champ fait l'objet d'une vérification de validité d'où l'usage de la balise <spring:bind> qui nous permet notamment d'avoir accès au message d'erreur éventuel associé au champ. La vérification est faite dans la méthode [validate] du validateur [ValidateurDocument] (ligne 32).
- lignes 27-34 : le champ HTML "contenu" associé au champ "contenu" du conteneur de saisies qui fait lui aussi l'objet d'une vérification de validité. Celle-ci est faite dans la méthode [onSubmit] du contrôleur [UploadController].

Le code de la page [confirmation.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.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Upload (mvc-36)</title>
8.    </head>
9.    <body>
10.     Le document [${document.nom},${nomDocument}] a &eacute;t&eacute; enregistr&eacute; sur le
    serveur.
11.     <br><br>
12.     <a href"upload.html">Recommencer</a>
13.   </body>
14. </html>
```

La méthode [onSubmit] du contrôleur [UploadController] met dans le modèle de la page [confirmation.jsp], le conteneur des saisies sous la clé " document " et le nom du document sur le poste client sous la clé " nomDocument ". La ligne 10 l'identifiant du document [document.nom] et son nom primtif sur le poste client [nomDocument].

Le code de la page [exception.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.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Upload (mvc-36)</title>
8.    </head>
9.    <body>
10.     <h2>Erreur</h2>
11.     L'exception suivante s'est produite :
12.     <c:out value="${exception.message}"/>
13.     <br>
14.   </body>
```

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com

```
15. </html>
```

Cette page n'est utilisée que par le gestionnaire d'exceptions. Celui-ci est configuré (spring-mvc-36-servlet.xml) de la façon suivante :

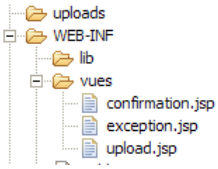
```
1.<!-- le gestionnaire d'exceptions -->
2.  <bean
3.    class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
4.    <property name="exceptionAttribute">
5.      <value>exception</value>
6.    </property>
7.    <property name="defaultStatusCode">
8.      <value>500</value>
9.    </property>
10.   <property name="defaultErrorView">
11.     <value>exception</value>
12.   </property>
13. </bean>
```

- la ligne 5 indique que la vue par défaut utilisée pour les exceptions qui remontent jusqu'à [DispatcherServlet] est la vue " exception ", donc la page JSP [exception.jsp].
- la ligne 11 indique que l'exception sera mise dans le modèle de la vue sous le nom " exception "

La ligne 12 de [exception.jsp] affiche donc le texte de l'exception. On sait en effet que [exception.message] sera traduit par [request.get("exception").getMessage()].

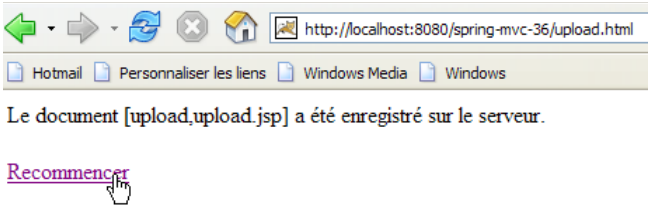
4.7 Les tests

Après ces longues explications, nous sommes prêts pour des tests. Ici, le client et le serveur sont sur le même poste. Nous prendrons les documents téléchargés dans le dossier [vues] du projet Eclipse. Nous savons que [uploadController] les range ensuite dans le dossier [uploads] du même projet. Au départ, la situation dans le projet Eclipse est la suivante :



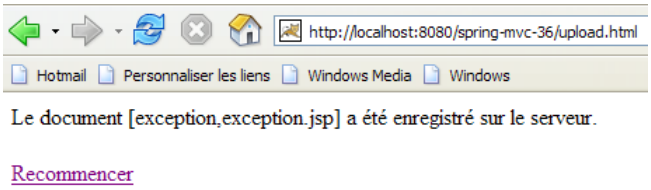
Nous demandons l'url [http://localhost:8080/spring-mvc-36/upload.html] :

Ici la méthode 1 d'enregistrement va être utilisée. La page obtenue en retour est la suivante :

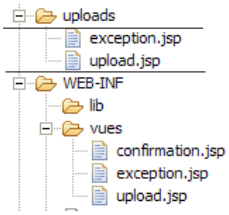


L'identifiant " upload " du document téléchargé ainsi que le nom originel de celui-ci " upload.jsp " ont été correctement récupérés. Nous utilisons le lien [Recommencer] pour faire un second téléchargement :

Ici la méthode 2 d'enregistrement va être utilisée. La page obtenue en retour est la suivante :



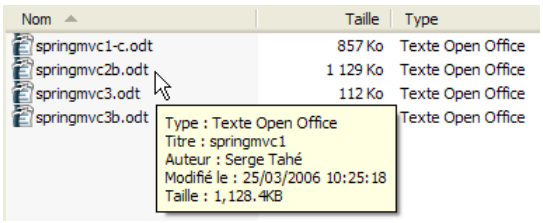
Si nous revenons au projet Eclipse et que nous le rafraîchissons (F5), nous pouvons voir que les documents précédents ont été sauvegardés dans le dossier [uploads] :



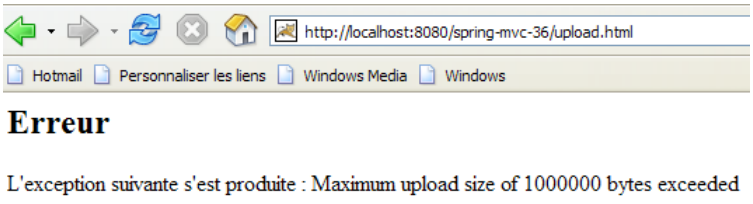
Rappelons la configuration de l'intercepteur [MultipartResolver] :

```
1.<!-- upload de fichiers -->
2.  <bean id="multipartResolver"
3.    class="org.springframework.web.multipart.commons.CommonsMultipartResolver">
4.    <property name="maxUploadSize">
5.      <value>1000000</value>
6.    </property>
7.  </bean>
```

Les lignes 4-6 indiquent que la taille du document téléchargé ne peut excéder 1 Mo. Essayons :



La réponse est la suivante :



L'implémentation [CommonsMultipartResolver] utilisée a lancé une exception qui est remontée jusqu'à [DispatcherServlet]. Celui-ci a alors fait appel au gestionnaire d'exceptions qui a fait afficher la vue " exception " et donc la page JSP [exception.jsp].

5 Conclusion

Cet article a présenté les points suivants du framework Spring MVC :

- le contrôleur [**MultiActionController**] qui permet de rassembler dans un même contrôleur le traitement de diverses Url. Il convient à des cas simples.
- le contrôleur [**AbstractWizardFormController**] qui permet de gérer des formulaires multi-pages (assistant)
- l'intercepteur [**MultipartResolver**] qui permet de gérer les documents téléchargés sur le serveur
- les générateurs de vues [**AbstractExcelView**] et [**AbstractPdfView**] qui permettent de produire respectivement des vues Excel ou PDF

Les principaux points de Spring MVC ont été désormais exposés. Dans le prochain article, nous mettrons en oeuvre Spring MVC dans une architecture 3tier [web, metier, dao] sur un exemple basique décliné en plusieurs versions. Ce sera l'occasion de revenir sur les outils offerts par Spring pour gérer l'ensemble des couches d'une architecture ntier et qui le distinguent notamment du framework Struts qui lui, ne s'occupe que de la seule couche [web].

6 Le code de l'article

Comme pour les précédents articles, le lecteur trouvera le code des exemples de ce document 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

www.Mcours.com
Site N°1 des Cours et Exercices Email: contact@mcours.com