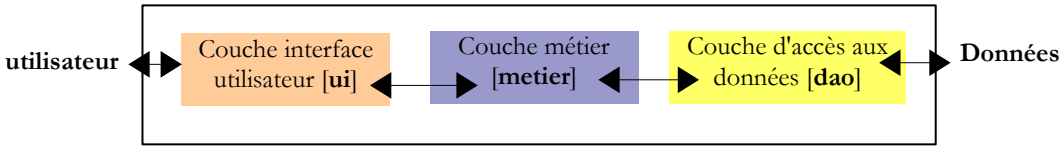
Spring MVC par l'exemple
- Partie 1 -

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

1 Généralités

Dans le développement web, la méthodologie de développement MVC (Modèle-Vue-Contrôleur) est désormais bien ancrée. Rappelons-en le principe.

Une application web a souvent une architecture 3tier :



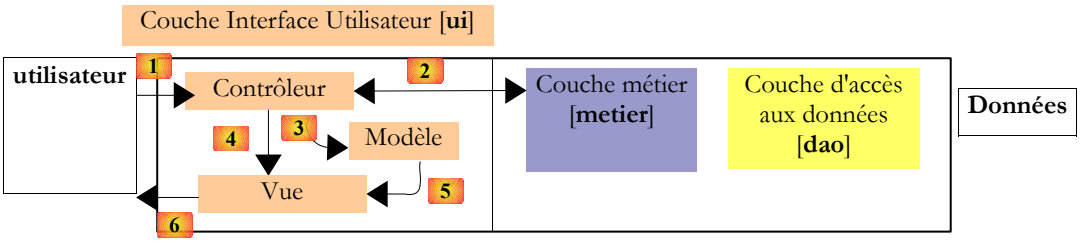
- la couche [**dao**] s'occupe de l'accès aux données, le plus souvent des données persistantes au sein d'un SGBD. Mais cela peut être aussi des données qui proviennent de capteurs, du réseau, ...
- la couche [**metier**] implémente les algorithmes " métier " de l'application. Cette couche est indépendante de toute forme d'interface avec l'utilisateur. Ainsi elle doit être utilisable aussi bien avec une interface console, une interface web, une interface de client riche. Elle doit ainsi pouvoir être testée en-dehors de l'interface web et notamment avec une interface console. C'est généralement la couche la plus stable de l'architecture. Elle ne change pas si on change l'interface utilisateur ou la façon d'accéder aux données nécessaires au fonctionnement de l'application.
- la couche [**interface utilisateur**] qui est l'interface (graphique souvent) qui permet à l'utilisateur de piloter l'application et d'en recevoir des informations.

La communication va de la gauche vers la droite :
- l'utilisateur fait une demande à la couche [**interface utilisateur**]
- cette demande est mise en forme par la couche [**interface utilisateur**] et transmise à la couche [**métier**]
- si pour traiter cette demande, la couche [**métier**] a besoin des données, elle les demande à la couche [**dao**]
- chaque couche interrogée rend sa réponse à la couche de gauche jusqu'à la réponse finale à l'utilisateur.

Les couches [métier] et [dao] sont normalement utilisées via des interfaces Java. Ainsi la couche [métier] ne connaît de la couche [dao] que son ou ses interfaces et ne connaît pas les classes les implémentant. C'est ce qui assure l'indépendance des couches entre-elles : changer l'implémentation de la couche [dao] n'a aucune incidence sur la couche [métier] tant qu'on ne touche pas à la définition de l'interface de la couche [dao]. Il en est de même entre les couches [interface utilisateur] et [métier].

L'architecture MVC prend place dans la couche [**interface utilisateur**] lorsque celle-ci est une interface web. Des articles (par exemple : http://tahe.developpez.com/java/m2vc) ont montré qu'on pouvait également appliquer le paradigme MVC à une couche [**interface utilisateur**] Swing.

Au sein de l'architecture 3tier, l'architecture MVC peut être représentée comme suit :



Le traitement d'une demande d'un client se déroule selon les étapes suivantes :

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.
2. le contrôleur **C** traite cette demande. 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
3. 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 le contrôleur. Ces informations forment ce qu'on appelle le modèle **M** de la vue, le **M** de MVC.

L'étape 3 consiste donc en le choix d'une vue **V** et en la construction du modèle **M** nécessaire à celle-ci.

4. le contrôleur **C** demande à la vue choisie de s'afficher. Il s'agit le plus souvent de faire exécuter une méthode particulière de la vue **V** chargée de générer la réponse au client. Dans ce document, nous appelerons vue, aussi bien l'objet qui génère la réponse au client que cette réponse elle-même. La littérature MVC n'est pas explicite sur ce point. Si c'est la réponse qui devait s'appeler vue, on pourrait appeler générateur de vue, l'objet qui génère cette réponse.
5. le générateur de vue **V** utilise le modèle **M** préparé par le contrôleur **C** pour initialiser les parties dynamiques de la réponse qu'il doit envoyer au client.
6. 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, ...

La méthodologie de développement web MVC ne nécessite pas nécessairement d'outils externes. On peut ainsi développer une application web Java avec une architecture MVC avec un simple JDK et les bibliothèques de base du développement web. Une méthode utilisable pour des applications simples est la suivante :

- le contrôleur est assuré par une servlet unique. C'est le **C** de MVC.
- toutes les requêtes du client contiennent un attribut **action**, par exemple (http://.../appli?**action=liste**).
- selon la valeur de l'attribut **action**, la servlet fait exécuter une méthode interne de type [**doAction**(...)].
- la méthode [**doAction**] exécute l'action demandée par l'utilisateur. Pour cela, si besoin est, elle utilise la couche [métier].
- selon le résultat de l'exécution, la méthode [**doAction**] décide d'une page JSP à afficher. C'est la vue **V** du modèle MVC.
- la page JSP a des éléments dynamiques qui doivent être fournis par la servlet. La méthode [**doAction**] va fournir ces éléments. C'est le modèle de la vue, le **M** de MVC. Ce modèle est placé le plus souvent dans le contexte de la requête (request.setAttribute(" clé ", "valeur "), voire moins fréquemment, dans le contexte de la session ou de l'application. On sait que la page JSP a accès à ces trois contextes.
- la méthode [**doAction**] fait afficher la vue en transmettant le flux d'exécution à la page JSP choisie. Elle utilise pour cela, une instruction du genre [getServletContext() .getRequestDispatcher(" pageJSP ").forward(request, response)].

Pour des applications simples, développées par un individu unique, cette méthode est suffisante. Néanmoins, lorsqu'on a écrit plusieurs applications de ce type, on s'aperçoit que les servlets de deux applications différentes :

1. ont le même mécanisme pour déterminer quelle méthode [doAction] il faut exécuter pour traiter l'action demandée par l'utilisateur
2. ne diffèrent en fait que par le contenu de ces méthodes [doAction]

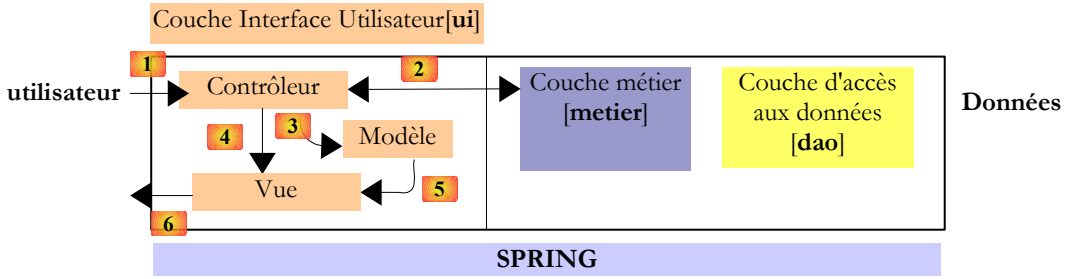
La tentation est alors grande de :

- factoriser le traitement (1) dans une servlet générique ignorante de l'application qui l'utilise
- déléguer le traitement (2) à des classes externes puisque la servlet générique ne sait pas dans quelle application elle est utilisée
- faire le lien entre l'action demandée par l'utilisateur et la classe qui doit la traiter à l'aide d'un fichier de configuration

Des outils, souvent appelés " frameworks ", sont apparus pour apporter les facilités précédentes aux développeurs. Le plus ancien et probablement le plus connu d'entre-eux est **Struts** (http://struts.apache.org/). **Jakarta Struts** est un projet de l'**Apache Software Foundation** (www.apache.org). Ce framework est décrit dans (http://tahe.developpez.com/java/struts/).

Apparu plus récemment, le framework **Spring** (http://www.springframework.org/) offre des facilités analogues à celles de Struts. Le but de cet article est de présenter les possibilités de **Spring MVC**, la branche de Spring consacrée au développement web MVC.

Une différence importante entre Struts et Spring est le champ d'applications de ces deux outils. Struts ne sert qu'à construire le modèle MVC dans la couche [interface web]. Spring offre lui, des outils pour le développement des trois couches d'une application 3tier.



Ci-dessus, on a représenté Spring comme une couche transversale de l'application 3tier afin d'illustrer le fait qu'il offre des outils pour le développement des trois couches. Struts lui, se cantonne uniquement à la couche [interface utilisateur].

On peut utiliser conjointement Spring et Struts :

- Spring pour l'intégration des couches, la gestion des transactions dans la couche [métier], l'accès aux données dans la couche [dao], ...
- Struts dans la couche [interface utilisateur]

Il peut cependant être tentant de n'utiliser qu'un seul " framework " et d'utiliser dans la couche [interface utilisateur], **Spring MVC** en lieu et place de **Struts**.

L'article " Variations autour d'une architecture web à trois couches " ([http://tahe.developpez.com/java/web3tier]), présente une application utilisant les trois architectures évoquées ici :

- une servlet propriétaire traitant les actions avec des méthodes internes
- une architecture Struts
- une architecture Spring MVC

Cet article présente Spring MVC. Nous utiliserons une succession d'exemples simples pour illustrer les différentes facettes du produit et nous terminerons par la construction d'une application web MVC 3tier qui soit un peu réaliste.

2 Outils utilisés et références

Les outils utilisés pour les exemples ont été les suivants :

- **Eclipse 3.01** pour le développement des applications Java, disponible à l'url [http://www.eclipse.org].
- **Plugin Eclipse** : Sysdeo Tomcat pour développer des applications web avec le conteneur de servlets Tomcat, disponible à l'url [http://www.sysdeo.com/eclipse/tomcatplugin].
- **Spring 1.2.4** disponible aux url [http://www.springframework.org/download]. On prendra la version de [Spring] avec dépendances car [Spring] utilise de nombreux outils tiers dont les archives sont contenues dans la version "avec dépendances".
- **Tomcat 5** comme conteneur de servlets, disponible à l'url [http://jakarta.apache.org].

Dans une échelle [débutant-intermédiaire-avancé], ce document est dans la partie [intermédiaire]. Sa compréhension nécessite divers pré-requis. Certains d'entre-eux peuvent être acquis avec des documents que j'ai écrits. Dans ce cas, je les cite. Il est bien évident que ce n'est qu'une suggestion et que le lecteur peut utiliser ses documents favoris.

- [**ref1**] : programmation web en Java [http://tahe.developpez.com/java/web/]
- [**ref2**] : utilisation d'**Eclipse** avec **Tomcat 5** : [http://tahe.developpez.com/java/eclipse/]. Ce document présente la version de Tomcat 5.0. Celle actuellement disponible est la 5.5.15 (mars 2006). Dans cette version, le lien [Tomcat administration] de la page d'accueil [http://localhost:8080] de Tomcat, ne fonctionne plus comme dans la version 5.0. En-dehors de ce point, les explications du document précédent restent valides. Le présent article n'utilise pas le lien [Tomcat administration].
- [**ref3**] : documentation **Spring** : [http://www.springframework.org/documentation]
- [**ref4**] : utilisation de l'aspect IoC de **Spring** : [http://tahe.developpez.com/java/springioc]
- [**ref5**] : applications web 3tier avec une architecture MVC : [http://tahe.developpez.com/java/web3tier]

Pour écrire cet article, je me suis servi de la documentation officielle de Spring ainsi que des deux ouvrages suivants :

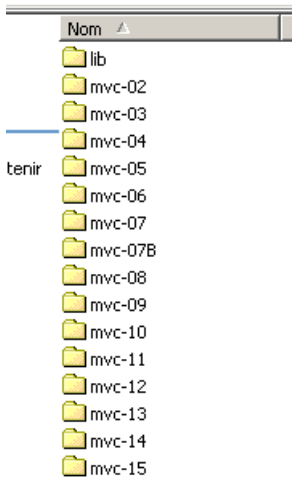
1. **[ref6] : Professional Java Development with the Spring Framework**, par Rod Johnson, Juergen Hoeller, Alef Arendsen, Thomas Risberg, Colin Sampaleanu, chez l'éditeur Wiley
2. **[ref7] : Pro Spring**, par Rob Harrop et Jan Machacek, chez l'éditeur APress

Un livre est paru récemment (février 2006) sur Spring MVC [**ref8**] : **Expert Spring MVC and Web Flow**, par Seth Ladd, Darren Davison, Steven Devijver, Colin Yates aux éditions APress. Je n'ai pas eu l'occasion de le lire. A ma connaissance, c'est le premier livre sorti sur ce sujet.

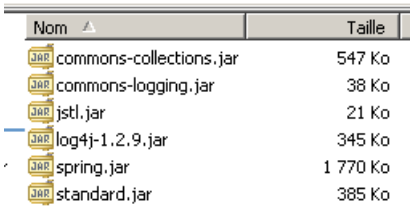
A ceux qui sont fâchés avec l'anglais, je propose de lire l'article qui suit. Pour les autres, les trois références précédentes seront certainement utiles pour approfondir le sujet.

3 Le code des exemples

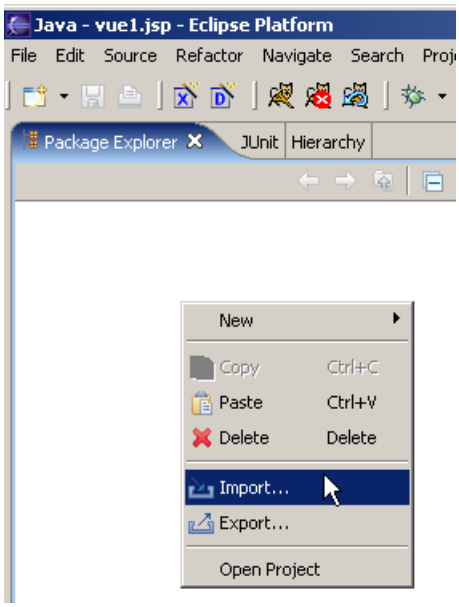
Le code des exemples qui vont suivre au même endroit que cet article sous la forme d'un zip. Une fois celui-ci dézippé, on obtient l'arborescence suivante :



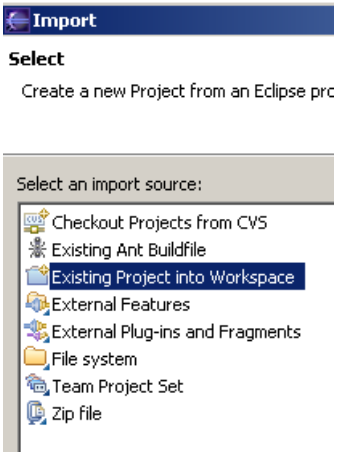
Les dossiers [mvc-xx] sont des projets Eclipse. Ils ont tous un dossier [lib] vide. Celui-ci doit normalement contenir des archives .jar. Celles-ci étant de taille assez importante, on a éviter de les dupliquer en les mettant uniquement dans le dossier [lib] ci-dessus :



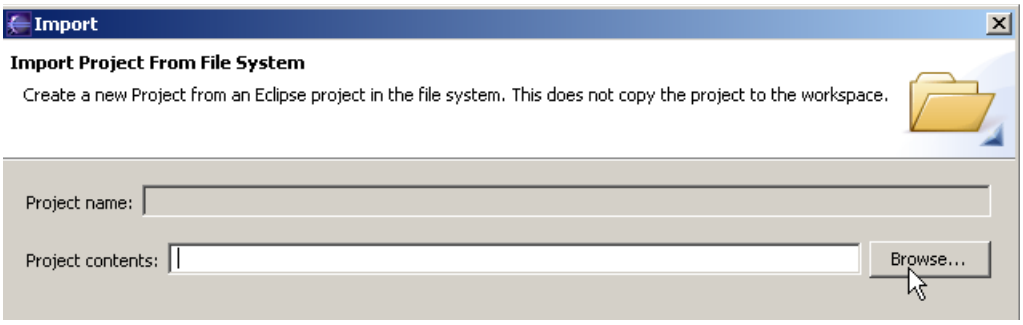
Pour tester les différents projets sous Eclipse, on peut procéder de la façon suivante. Sous Eclipse, importons par exemple le projet [mvc-02]. On clique droit dans [Package Explorer] :



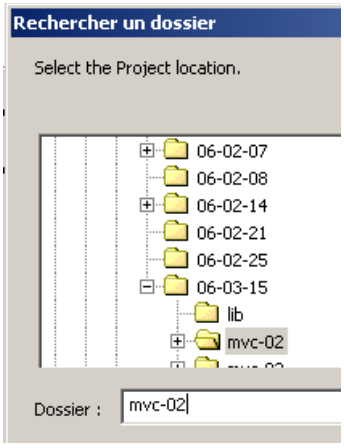
Nous prenons l'option [Import] :



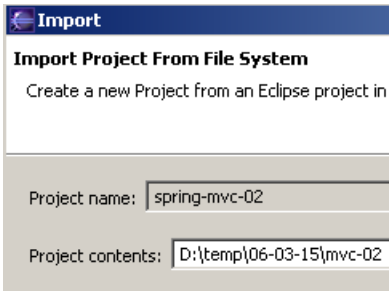
- faire [Next]



- faire [Browse] pour aller désigner le projet [mvc-02] :

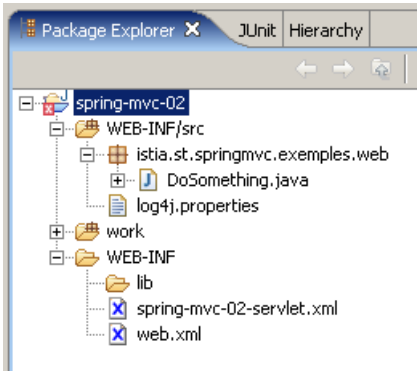


- faire [OK]

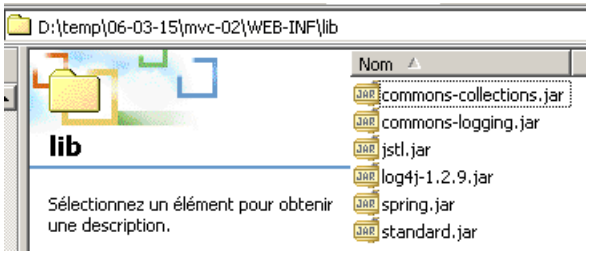


- faire [Finish]

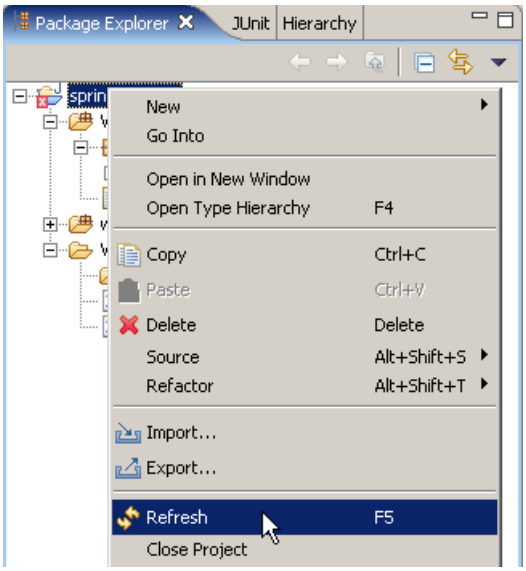
Le projet s'ouvre dans [Package Explorer] :



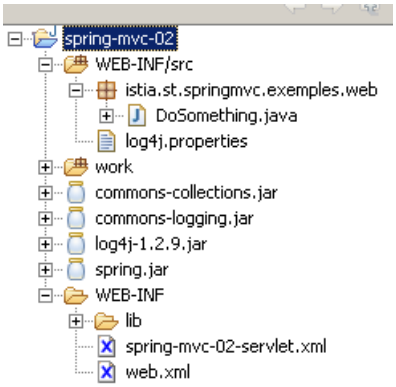
Ci-dessus, la croix rouge indique que le projet est erroné. Ceci parce que le dossier [lib] est vide. Avec l'explorateur windows, copier le contenu du dossier [lib] du fichier dézippé dans [mvc-02/WEB-INF/lib] :



Revenir sous Eclipse. Cliquer droit sur le nom du projet [spring-mvc-02] et prendre [Refresh] :

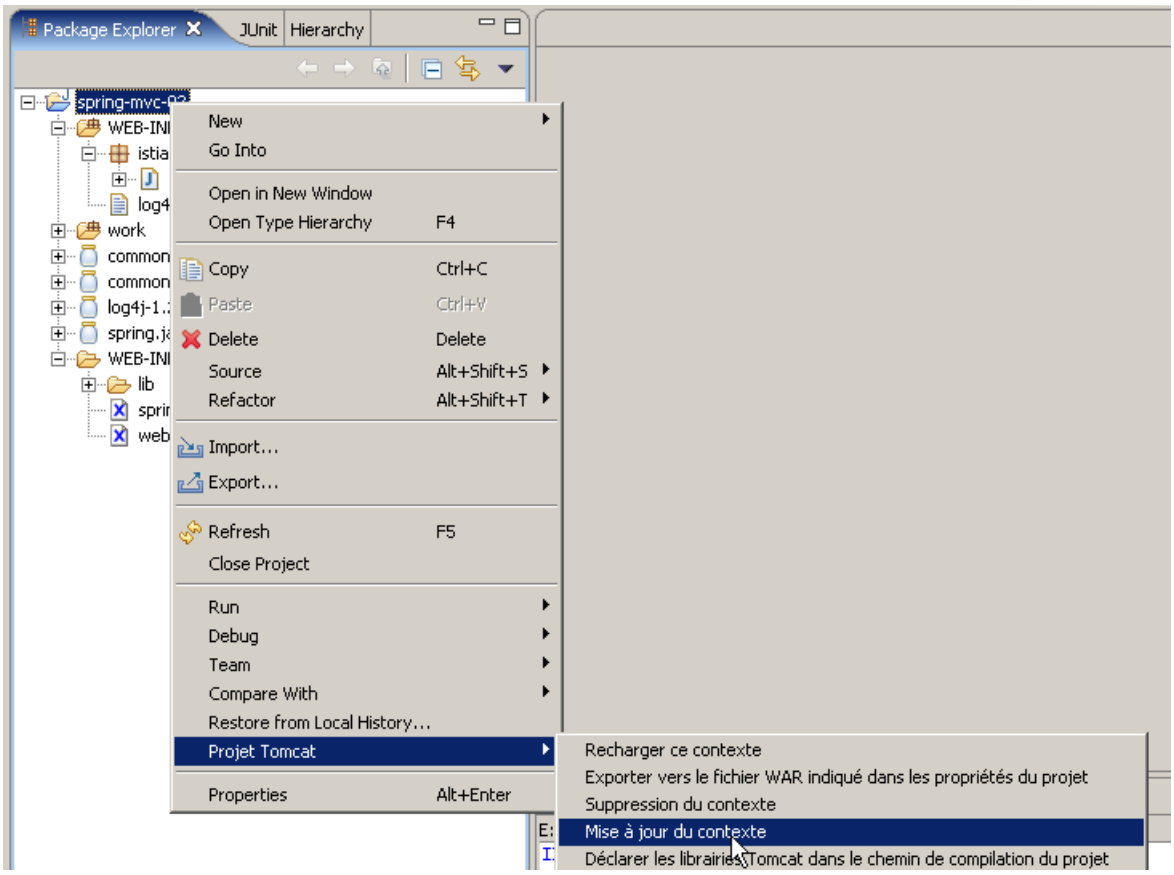


Cela force le projet à se reconstruire :

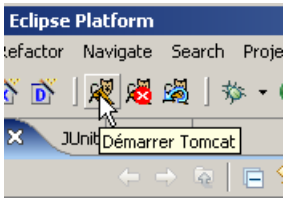


Ci-dessus, le projet ne présente plus d'erreurs.

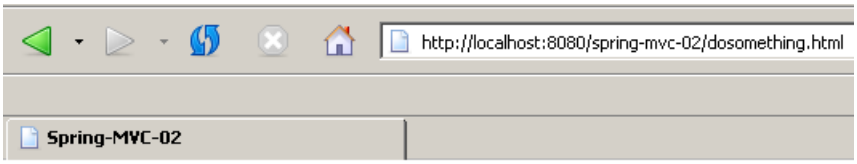
Faire du projet [spring-mvc-02] une application Tomcat (clic droit sur projet) :



Pour le tester, lancer [Tomcat] :

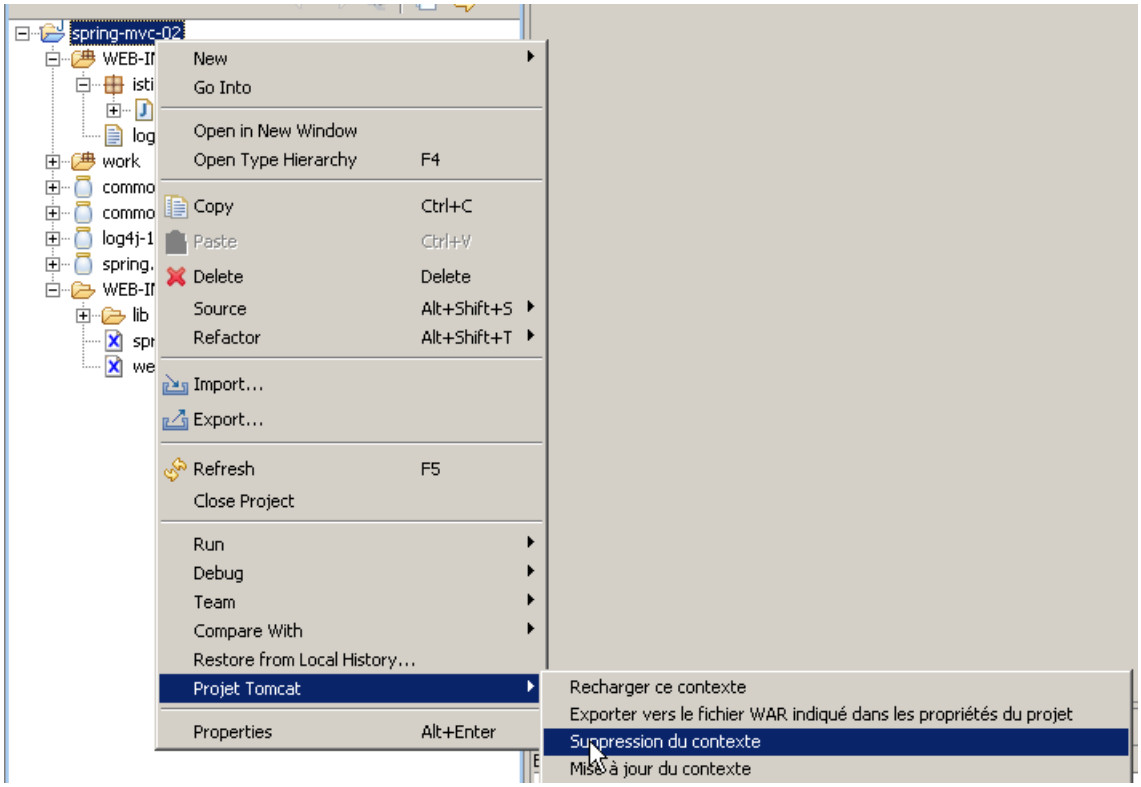


Avec un navigateur, demander l'url [http://localhost:8080/spring-mvc-02/dosomething.html] :



DoSomething exécuté...

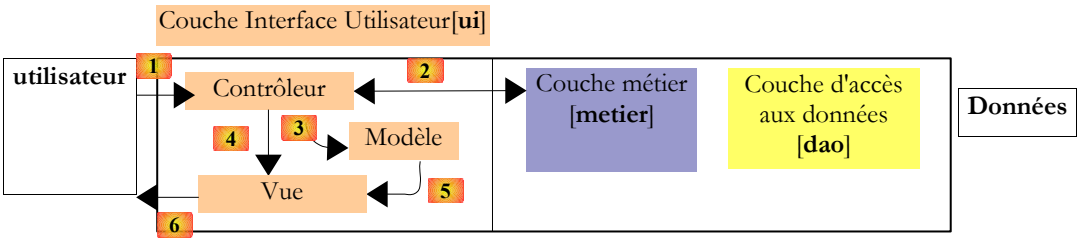
Pour passer d'un projet à l'autre, on réitère la démarche précédente. Pour permettre à Tomcat de démarrer plus vite, il est préférable de supprimer le contexte des projets testés :



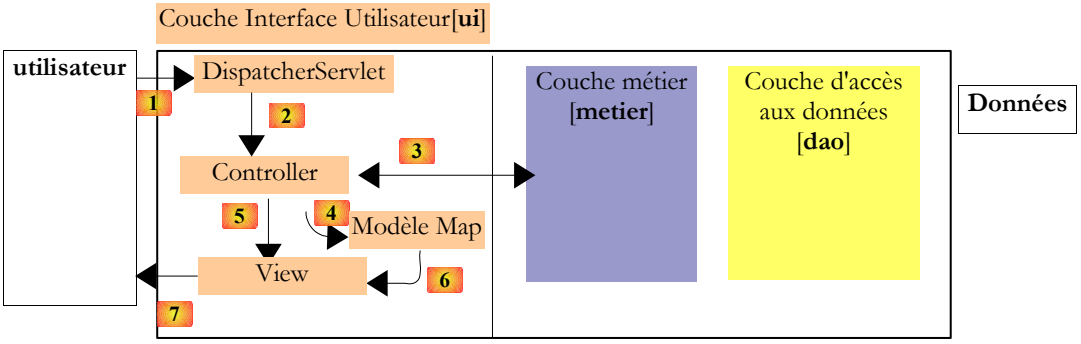
Pour recréer ce contexte ultérieurement, il suffir de prendre l'option [Mise à jour du contexte].

4 Les composantes d'une architecture Spring MVC

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



Spring MVC implémente cette architecture de la façon 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, ...

Examinons le traitement d'une demande du client sous un aspect un peu plus technique :

Etapes 1-2 : choix du contrôleur traitant l'action demandée par l'utilisateur

---

Avec Spring MVC, c'est à partir de l'URL demandée par le client que le contrôleur principal [DispatcherServlet] va choisir l'objet [Controller] qui va traiter l'action. Le lien URL <-> Controller est fait par configuration. Il existe plusieurs méthodes de résolution possibles, c.a.d. plusieurs façons possibles de lier une URL à un objet [Controller].

Etapes 3-5 : traitement de la demande par l'objet [Controller]

L'objet [Controller] chargé de traiter l'action est une instance de classe implémentant l'interface [**org.springframework.web.servlet.mvc.Controller**].

Spring offre plusieurs classes implémentant l'interface [Controller] :

org.springframework.web.servlet.mvc

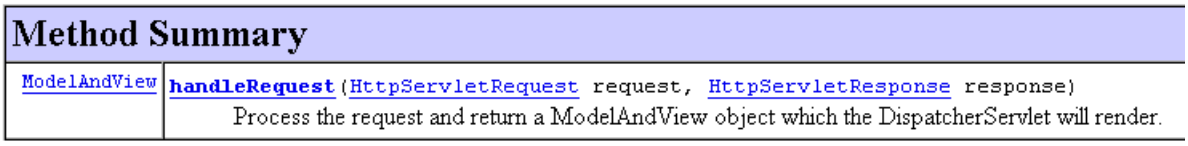
Interface Controller

All Known Implementing Classes:

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

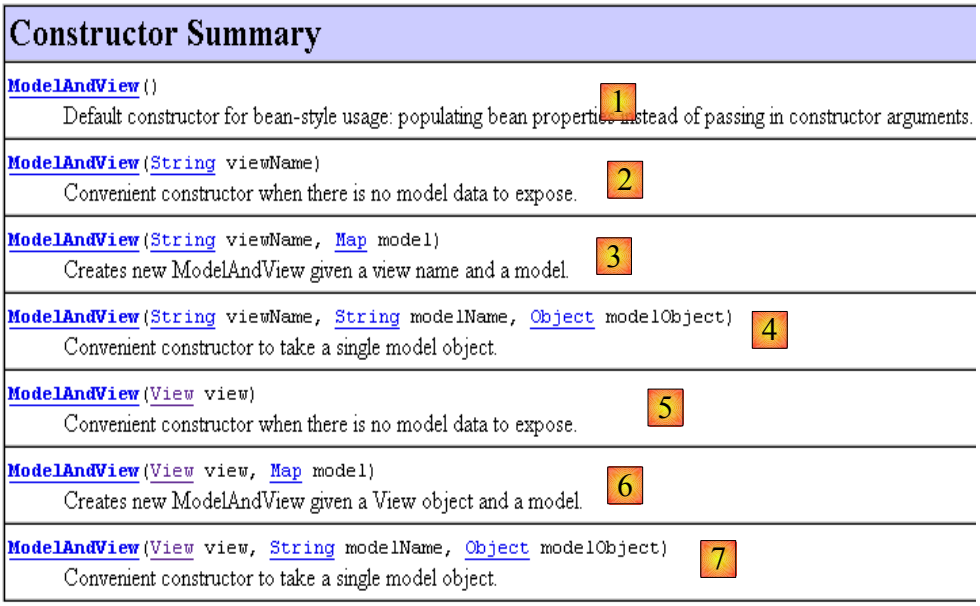
Ces classes sont normalement destinées à être dérivées. On se souvient en effet, que le contrôleur est chargé de traiter une action d'une application spécifique. Ceci ne peut être fait dans les classes génériques ci-dessus sauf lorsqu'il n'y a pas de traitement à faire (ParameterizableViewController, ServletForwardingController). De façon générale, c'est au développeur de construire la classe implémentant l'interface [Controller]. Il peut parfois se faire aider en dérivant cette classe des classes ci-dessus. C'est notamment le cas si l'action demandée est le POST d'un formulaire. Dans ce cas particulier important, la classe [SimpleFormController] apporte des facilités de développement.

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



- l'unique méthode à implémenter par un contrôleur est donc [**handleRequest**] qui a pour paramètres, l'objet [request] qui encapsule la requête HTTP du client, et l'objet [response] qui lui encapsule la réponse HTTP qui sera faite à ce même client.
- comme le montre le schéma plus haut, un contrôleur doit désigner une vue V à afficher en réponse à la demande du client et le modèle M qui sera affiché par cette vue. Ces deux éléments sont rendus par [**handleRequest**] sous la forme d'un objet de type [**ModelAndView**] plus exactement [**org.springframework.web.servlet.ModelAndView**].

Un objet de type [**ModelAndView**] peut être construit de diverses façons :



1. le constructeur [1] ne définit ni modèle, ni vue. Ceux-ci sont alors définis ultérieurement via les méthodes de la classe [ModelAndView]
2. le constructeur [2] définit une vue **V** d'après son nom. Nous savons qu'au final, la vue est un objet implémentant l'interface [**org.springframework.web.servlet.View**]. Le lien nom <-> objet View est alors fait dans un fichier de configuration. Le constructeur [2] ne précise pas de modèle. Il convient donc pour une vue sans modèle. On peut aussi préciser le modèle ultérieurement via des méthodes de [ModelAndView].
3. le constructeur [3] définit une vue **V** d'après son nom et définit également un modèle **M** sous la forme d'un objet implémentant l'interface [java.util.Map], donc un dictionnaire.
4. le constructeur [4] est une variante du constructeur [3] lorsque le modèle n'a qu'un unique élément. Dans ce cas, le dictionnaire n'a qu'un élément dont la clé sera [modelName] et la valeur associée [modelObject].
5. les constructeurs [5, 6, 7] sont des variantes des constructeurs [2, 3, 4] dans lesquelles, la vue V n'est plus désignée par son nom mais directement instanciée par un objet implémentant l'interface [View]. L'association nom vue <-> objet View n'a alors plus à être faite dans le fichier de configuration.

Les méthodes de la classe [ModelAndView] sont les suivantes :

Method Summary	
ModelAndView	**addAllObjects**(Map modelMap) Add all entries contained in the provided map to the model. 1
ModelAndView	**addObject**(String modelName, Object modelObject) Add an object to the model. 2
void	**clear**() Clear the state of this ModelAndView object.
Map	**getModel**() Return the model map.
protected Map	**getModelInternal**() Return the model map.
View	**getView**() Return the View object, or null if we are using a view name to be resolved by the DispatcherServlet via a ViewResolver.
String	**getViewName**() Return the view name to be resolved by the DispatcherServlet via a ViewResolver, or null if we are using a View object.

- les méthodes [1] et [2] permettent d'ajouter des éléments au modèle M

boolean	**isEmpty**() Return whether this ModelAndView object is empty, i.e. whether it does not hold any view and does not contain a model.
boolean	**isReference**() Return whether we use a view reference, i.e. true if the view has been specified via a name to be resolved by the DispatcherServlet via a ViewResolver.
void	**setView**(View view) Set a View object for this ModelAndView. 3
void	**setViewName**(String viewName) Set a view name for this ModelAndView, to be resolved by the DispatcherServlet via a ViewResolver. 4
String	**toString**() Return diagnostic information about this model and view.

- les méthodes [3] et [4] permettent de préciser la vue V soit par une instance View [3], soit par le nom de la vue [4].

Etapes 6-7 : rendu du modèle M par la vue V

Arrivé ici, le flux d'exécution a été transféré à une vue [View]. [View] est une interface avec l'unique méthode suivante :

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

- assez logiquement, l'unique méthode [render] dispose de l'objet [response] à partir duquel la réponse HTTP va être construite et du modèle [model] construit précédemment par le contrôleur. La méthode [render] dispose également de l'objet [request] qui encapsule la requête HTTP du client. Via cet objet, la méthode [render] a accès au contexte de la requête, à la session qui lui est liée, ...
  L'une des implémentations de l'interface [View] est la classe [JstlView] qui implémente la vue à l'aide d'une page JSP utilisant des balises JSTL. La méthode [render] de la classe [JstlView] met les éléments du dictionnaire [model] qu'elle a reçu dans le contexte de la requête [request]. C'est là que la page JSP qui sera envoyée au client trouvera son modèle.

Spring propose plusieurs implémentations de l'interface [View] :

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

Des noms de classes ci-dessus, on peut déduire que Spring offre des classes permettant d'avoir des vues

- de type HTML (JstlView), Excel, PDF
- destinées à des frameworks d'affichage : Velocity, FreeMarker, Tiles

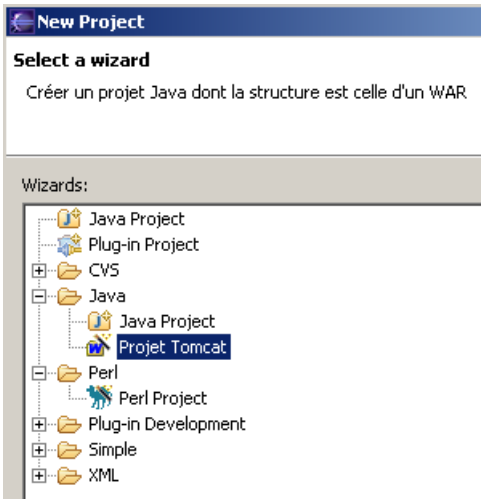
Dans la suite, nous présenterons des exemples illustrant les différentes étapes de traitement d'une requête décrites ci-dessus.

5 Configuration d'une application Spring MVC

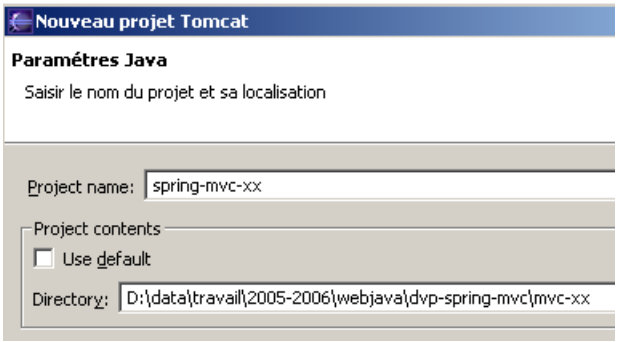
Une application Spring MVC est une application web. Elle obéit donc aux règles de configuration de ce type d'applications. Ci-dessous, nous montrons comment est configurée une application Spring MVC. L'environnement de développement utilisé est Eclipse. Le lecteur fera la part des choses entre ce que l'architecture décrite doit à Eclipse et qui est propre à Eclipse et ce qui ressort de la configuration nécessaire de toute application web, quelque soit l'environnement de développement utilisé.

Les applications qui vont suivre ont été construites sous Eclipse avec le plugin Sysdeo Tomcat afin de pouvoir lancer, arrêter, relancer Tomcat à partir d'Eclipse. Le développement web avec Eclipse et Tomcat est décrit dans [ref2] (cf page 4). Les différents projets que nous allons construire sous Eclipse seront tous des projets de type [Projet Tomcat]. Rappelons comment se crée un tel projet :

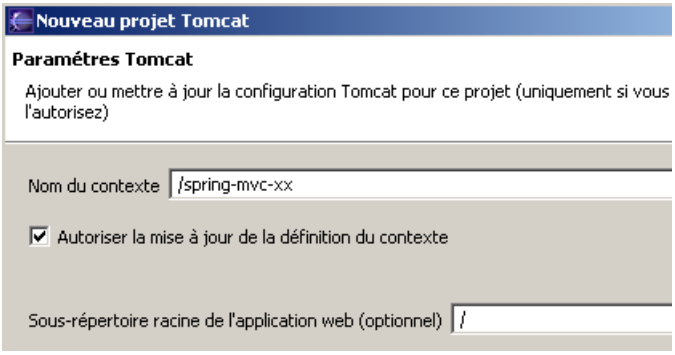
- on démarre l'assistant de création par [File / New / Project] :



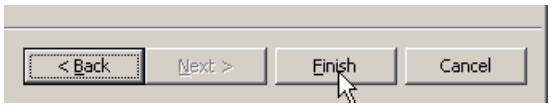
- on choisit le type [Projet Tomcat]
- on indique le nom du projet et son emplacement dans le système de fichiers :



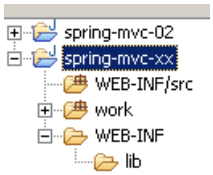
- on donne un nom (contexte) à l'application web :



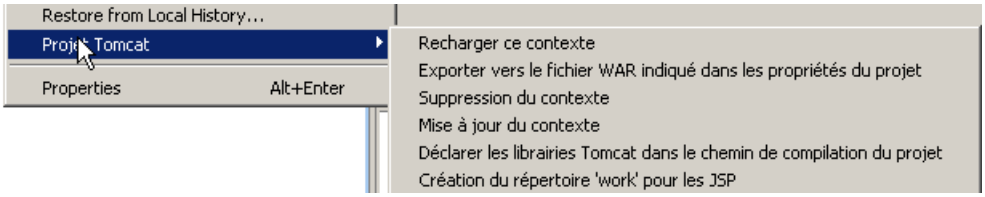
- et c'est fini :



Le projet apparaît dans l'explorateur de projets d'Eclipse :



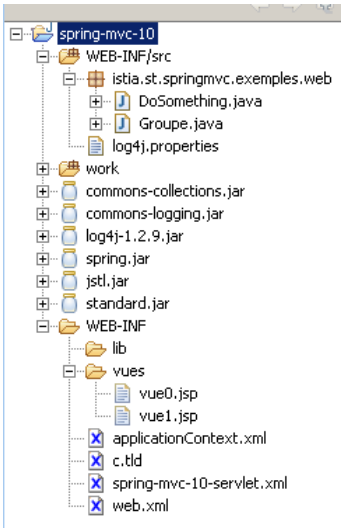
Avec un clic droit sur le nom du projet, on a accès à diverses propriétés du projet Tomcat :



Les options utiles sont les suivantes :

- **Mise à jour du contexte** : crée le fichier de définition de l'application qui indique à Tomcat que cette application fait partie des applications qu'il doit gérer.
- **Suppression du contexte** : supprime le fichier précédent. Cela évite à Tomcat de charger des ressources pour une application qui n'est temporairement plus utilisée par exemple. Cela accélère également le démarrage de Tomcat.
- **Recharger le contexte** : demande à Tomcat de recharger le contexte de l'application parce qu'on y a apporté une modification. L'application est alors rechargée sans avoir à relancer Tomcat.

Nous décrivons maintenant une structure typique des applications Spring MVC que nous allons construire.



WEB-INF/src

Ce dossier contiendra les classes des applications web. C'est le plugin Tomcat qui l'impose. Nous mettrons les classes Java toujours dans le même paquetage [istia.st.springmvc.exemples.web]. Les classes compilées sont automatiquement placées dans le dossier [WEB-INF/classes], là où les attend le serveur web Tomcat.

Tous les fichiers du dossier [WEB-INF/src] autres que les classes .java sont automatiquement recopiées en l'état dans [WEB-INF/classes]. Dans une application web, le dossier [WEB-INF/classes] fait partie du " Classpath " de l'application web. Aussi mettra-t-on dans [WEB-INF/src] tous les fichiers qui doivent être dans le Classpath de l'application. C'est le cas du fichier [log4j.properties] ci-dessus. Ce fichier est recherché dans le Classpath par l'archive [log4j-1.2.9.jar] que l'on voit dans la copie d'écran ci-dessus. Cette archive contient les classes de l'outil **log4j** souvent utilisé pour afficher des logs. **log4j** est configuré à l'aide du fichier [log4j.properties].

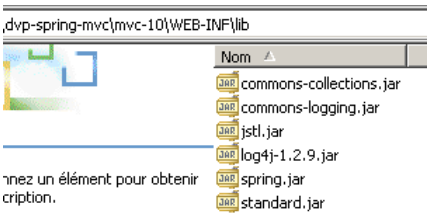
A ma grande honte, j'avoue ne pas bien connaître log4j. J'ai utilisé le fichier [log4j.properties] minimal suivant :

```
1.  # Global logging configuration
2.  log4j.rootLogger=INFO, stdout
3.
4.  # configuration...
5.  org.apache.commons.digester.Digester=INFO
6.
7.  # Console output...
8.  log4j.appender.stdout=org.apache.log4j.ConsoleAppender
9.  log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
10. log4j.appender.stdout.layout.ConversionPattern=%5p [%t] - %m%n
```

En l'absence de ce fichier, Spring affichait sur la console un message disant que log4j n'était pas configuré. Il précisait de plus que c'était la classe [org.apache.commons.digester.Digester] qui voulait faire des logs. Après quelques recherches sur le web, j'ai créé le fichier précédent. Les lignes sont standard : on les retrouve dans tous les exemples de [log4j.properties]. J'ai utilisé le message de Spring pour la ligne 5 qui elle est spécifique. Ceci fait, Spring ne s'est plus plaint et a fait ses logs sur la console. Je m'en suis tenu là et c'est ce même fichier qui sera utilisé dans toutes les applications.

WEB-INF/lib

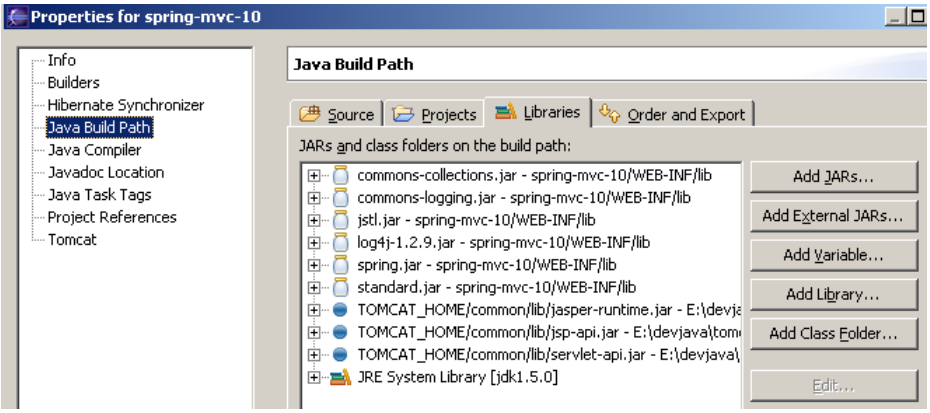
Ce dossier est facultatif. J'y mets toutes les archives .jar dont l'application a besoin :



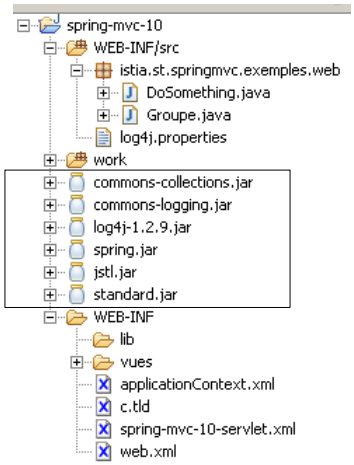
- **spring.jar** : la totalité de l'archive Spring 1.2.4 [http://www.springframework.org]
- **commons-*.jar** : bibliothèques " commons " utilisées par Spring [http://jakarta.apache.org/commons/]
- **jstl.jar, standard.jar** : bibliothèques nécessaires pour l'utilisation des balises JSTL dans les pages JSP [http://jakarta.apache.org/taglibs/]
- **log4j-1.2.9.jar** : bibliothèque de logs [http://logging.apache.org/log4j/docs/]

Le fait de mettre ces archives dans [WEB-INF/lib] n'a aucune signification particulière. Pour que l'application les reconnaisse, il faut les mettre dans son Classpath. Sous Eclipse, on procède comme suit :

1. clic droit sur le nom du projet / Properties / Java Build Path



2. utiliser [Add Jars] pour ajouter les archives désirées. Eclipse indique alors les archives du Classpath dans sa modélisation du projet :



WEB-INF/vues

Nous utiliserons ce dossier pour y mettre les pages JSP qui serviront de vues à l'application. Ces vues peuvent être mises n'importe où dans le dossier de l'application web. En les mettant sous le dossier [WEB-INF] on interdit à un utilisateur extérieur de demander directement une vue JSP. Il devra passer obligatoirement par l'une des actions définies pour l'application.

WEB-INF

Ce dossier est obligatoire dans une application web Java. Il comporte au moins deux éléments :

- [**WEB-INF/classes**] qui contient les fichiers .class nécessaires à l'application
- **web.xml** : le fichier de configuration de l'application

Le fichier [web.xml] sera pour toutes les applications construit sur le modèle suivant :

```
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-10</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-10</servlet-name>
21.     <url-pattern>*.html</url-pattern>
```

```
22.   </servlet-mapping>
23. </web-app>
```

- lignes 8-11 : définissent une classe qui est chargée au démarrage de l'application. La classe [org.springframework.web.context.ContextLoaderListener] est une classe Spring chargée d'initialiser le contexte de l'application web en lisant le contenu du fichier [WEB-INF/applicationContext.xml]. Ce fichier nous servira à instancier les couches [métier] et [dao] lorsqu'il y en a. Lorsque l'application n'aura que la couche [interface utilisateur] et pas les couches [métier] et [dao], le fichier [WEB-INF/applicationContext.xml] n'existera pas, ni les lignes 8-11 ci-dessus. Ce sera souvent le cas dans nos applications exemple où il n'y aura le plus souvent que la seule couche d'interface web.
- lignes 13-14 : définissent une servlet :
  - ligne 14 : nom de la servlet
  - ligne 16 : classe de la servlet. Avec Spring MVC, cette classe est toujours la classe [org.springframework.web.servlet.DispatcherServlet] qui joue le rôle du contrôleur C du MVC.
- lignes 19-22 : désignent les URL gérées par l'application :
  - ligne 21 : indique que toute URL terminée par .html est gérée par l'application
  - ligne 20 : désigne la servlet chargée de gérer les URL définies ligne 21.

Le fichier [web.xml] ci-dessus indique donc que :

- toute URL terminée par .html doit être traitée par la servlet appelée [spring-mvc-10]
- que la servlet [spring-mvc-10] désigne la classe [org.springframework.web.servlet.DispatcherServlet], une classe de Spring

Le contrôleur [org.springframework.web.servlet.DispatcherServlet] associé à la servlet [spring-mvc-10] et chargé de traiter les URL *.html, a besoin d'informations pour traiter les demandes de l'utilisateur. Il a par exemple besoin de connaître pour chaque URL [ur1.html], l'objet [Controller] chargé de la traiter. Nous avons vu qu'un objet [Controller] allait désigner la vue à renvoyer à l'utilisateur par son nom. Le contrôleur [DispatcherServlet] aura donc besoin de savoir qu'à tel nom de vue correspond telle page JSP. D'autres informations seront également nécessaires. Nous les détaillerons progressivement. Le contrôleur [org.springframework.web.servlet.DispatcherServlet] associé à la servlet [spring-mvc-10] trouvera ces informations dans le fichier [WEB-INF/spring-mvc-10-servlet.xml]. Le fichier porte donc le nom de la servlet. Il aura un contenu qui pourra ressembler à ceci :

```
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="/dosomething.html">DoSomethingController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="DoSomethingController"
14.     class="istia.st.springmvc.exemples.web.DoSomething">
15.     <property name="groupe">
16.       <ref bean="groupe"/>
17.     </property>
18.   </bean>
19.   <!-- le résolveur de vues -->
20.   <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
21.     <property name="viewClass">
22.       <value>org.springframework.web.servlet.view.JstlView</value>
23.     </property>
24.     <property name="prefix">
25.       <value>/WEB-INF/vues/</value>
26.     </property>
27.     <property name="suffix">
28.       <value>.jsp</value>
29.     </property>
30.   </bean>
31. </beans>
```

Il est trop tôt pour entrer dans les détails mais voici les grandes lignes de ce fichier :

- lignes 5-11 : indiquent que l'URL se terminant par [/dosomething.html] doit être traité par l'objet [Controller] nommé [DoSomethingController]. Nous avons vu que [web.xml] redirigeait les URL *.html vers [DispatcherServlet]. Le fichier [servlet-mvc-10-servlet.xml] précise les choses : seule l'URL [*/dosomething.html] sera acceptée. Les autres seront refusées.
- lignes 13-18 : définissent la classe associée au contrôleur de nom [DoSomethingController]. Cette classe est écrite par le développeur pour les besoins spécifiques de l'application et implémente l'interface [Controller]. La classe l'implémentant est ici [istia.st.springmvc.exemples.web.DoSomething], une classe propriétaire qu'on trouve dans [WEB-INF/src] sur la copie d'écran du projet Eclipse.

- lignes 20-30 : définissent comment à partir du nom d'une vue, on trouve la page JSP associée. Si [DoSomethingController] demande l'affichage d'une vue appelée V, c'est la page JSP [WEB-INF/vues/V.jsp] qui sera affichée.

Pour terminer, la copie d'écran du projet montre la présence du fichier [c.tld] dans le dossier [WEB-INF]. Ce fichier définit la syntaxe des balises de la bibliothèque JSTL. Les pages JSP utilisent souvent cette bibliothèque de balises qui permet de réduire la présence de code Java dans la page JSP. Les pages JSP auront un contenu analogue au suivant :

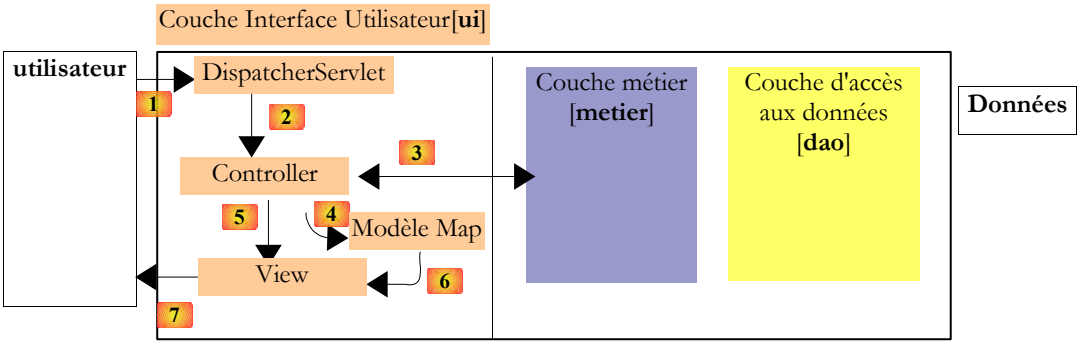
```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-10</title>
8.    </head>
9.    <body>
10.     <h2>Vue n° 0</h2>
11.     Membres du groupe
12.     <table>
13.       <c:forEach var="personne" items="${groupe.membres}">
14.         <tr>
15.           <td>${personne}</td>
16.         </tr>
17.       </c:forEach>
18.     </table>
19.     <br>
20.     DoSomething exécuté en ${durée} ms...
21.   </body>
22. </html>
```

- la ligne 1 est facultative. Elle précise le type de codage utilisé pour les caractères du fichier. ISO-8859-1 autorise les caractères accentués.
- la ligne 2 indique où trouver un fichier de définition de balises. Ici c'est le fichier [c.tld] dont on vient de parler.
- la ligne 3 indique que les expressions ${identificateur} (par exemple ${personne}ci-dessus) doivent être interprétées. Selon la version JSTL utilisée et celle des pages JSP, cette balise est parfois nécessaire, parfois pas. Avec les versions de JSTL et JSP utilisées pour les exemples, elle était nécessaire.

6 La liaison URL - Controller

6.1 Les stratégies de résolution des URL

Reprenons l'architecture d'une application Spring MVC :



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

Comment le contrôleur [**DispatcherServlet]** sait-il quel objet [Controller] doit traiter la demande du client ?

La demande du client est contenue dans l'URL demandée au serveur et éventuellement dans les données qui l'accompagnent (dans la cas du requête HTTP POST ou plus rarement HTTP PUT). Spring se sert de l'URL pour déterminer quel objet [Controller] doit

traiter la demande. Spring met à notre disposition diverses stratégies pour lier une URL à un objet [Controller]. Le choix d'une stratégie particulière est fait dans le fichier de configuration [**S-servlet.xml**] de la servlet **S** de l'application Spring MVC en précisant le nom d'une classe implémentant l'interface [**org.springframework.web.servlet.HandlerMapping**].

Spring offre les classes d'implémentation suivantes :

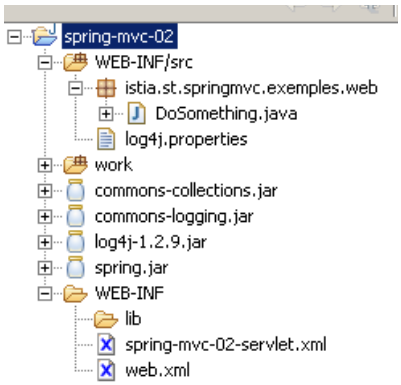
- [**BeanNameUrlHandlerMapping**] : avec cette classe, une URL de la forme [http://machine:port/chemin1/.../**document**] sera traitée par le Controller nommé [**document**]. C'est la stratégie de résolution par défaut si aucune stratégie n'est définie dans le fichier de configuration de la servlet.
- [**SimpleUrlHandlerMapping**] : avec cette classe, chaque URL [http://machine:port/chemin1/.../**document**] devant être traitée par l'application est déclarée dans le fichier de configuration de la servlet. Dans cette déclaration , on associe à [/**document**] le nom de l'objet [Controller] qui doit la traiter.

Les deux stratégies permettent de spécifier des URL génériques, par exemple /erreur*.html. La stratégie [**SimpleUrlHandlerMapping**] permet des URL génériques plus complexes que celles permises par la stratégie [**BeanNameUrlHandlerMapping**]. Par ailleurs, la stratégie [**SimpleUrlHandlerMapping**] permet d'externaliser dans un fichier les associations URL <-> Controller.

Nous présentons tout d'abord un exemple utilisant la stratégie [**SimpleUrlHandlerMapping**].

6.2 La stratégie SimpleUrlHandlerMapping

Le projet Eclipse de l'exemple est le suivant :



L'application est définie par le fichier [web.xml] suivant :

```
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.    <!-- la servlet -->
8.    <servlet>
9.      <servlet-name>spring-mvc-02</servlet-name>
10.     <servlet-class>
11.       org.springframework.web.servlet.DispatcherServlet</servlet-class>
12.   </servlet>
13.   <!-- le mapping des url -->
14.   <servlet-mapping>
15.     <servlet-name>spring-mvc-02</servlet-name>
16.     <url-pattern>*.html</url-pattern>
17.   </servlet-mapping>
18. </web-app>
```

Ce fichier a déjà été expliqué. Simplement notons que la servlet de l'application s'appelle [spring-mvc-02] et que donc son fichier de configuration doit s'appeler [spring-mvc-02-servlet.xml]. Ce fichier est présent sur la copie d'écran ci-dessus. Son contenu 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="/dosomething.html">DoSomethingController</prop>
9.       </props>
10.    </property>
11.  </bean>
12.  <!-- les contrôleurs de l'application-->
13.  <bean id="DoSomethingController"
14.    class="istia.st.springmvc.exemples.web.DoSomething"/>
15. </beans>
```

- lignes 5-11 : définissent la stratégie de résolution des URL, ici la stratégie [SimpleUrlHandlerMapping]. On remarquera que le bean de la ligne 5 n'a pas de d'attribut " id ", attribut qui identifie les beans. On pourrait lui en donner un mais ce n'est pas obligatoire car il n'est pas référencé par ailleurs dans le fichier. Spring MVC instancie automatiquement un certain nombre de beans. C'est le cas des beans définissant la stratégie de résolution, c'est à dire des beans implémentant l'interface [HandlerMapping].
- ligne 6-9 : définissent les liaisons URL <-> Controller de l'application. On y trouve toutes les URL que l'application doit traiter avec, associé, le nom de l'objet Controller qui doit les traiter.
- ligne 8 : indique que l'URL se terminant par [/dosomething.html] doit être traitée par le Controller de nom [DoSomethingController].
- lignes 13-14 : définissent le Controller de nom [DoSomethingController] (attribut id). On y indique que ce Controller est une instance de la classe [istia.st.springmvc.exemples.web.DoSomething], une classe propriétaire que nous allons définir maintenant.

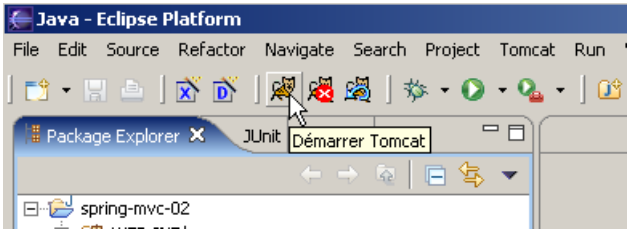
La classe [DoSomething] a été placée dans [WEB-INF/src] comme le montre le projet Eclipse. Sa définition est la suivante :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.io.PrintWriter;
4.  import javax.servlet.http.HttpServletRequest;
5.  import javax.servlet.http.HttpServletResponse;
6.
7.  import org.springframework.web.servlet.ModelAndView;
8.  import org.springframework.web.servlet.mvc.Controller;
9.
10. public class DoSomething implements Controller {
11.
12.   // gestion de la requête
13.   public ModelAndView handleRequest(HttpServletRequest request,
14.       HttpServletResponse response) throws Exception {
15.     // on code en dur
16.     response.setContentType("text/html");
17.     PrintWriter out = response.getWriter();
18.     out.println(
19.           "<html><head><title>Spring-MVC-02</title></head>"+
20.           "<body>DoSomething exécuté...</body></html>");
21.     // pas de ModelAndView
22.     return null;
23.   }
24.
25. }
```

Rappelons tout d'abord qu'un objet [Controller] doit implémenter l'interface [Controller]. C'est ce que fait la classe ci-dessus, ligne 10.

- lignes 13-23 : définissent la méthode [handleRequest] qui est l'unique méthode de l'interface [Controller]. Cette méthode doit rendre un couple (Vue, Modèle) sous la forme d'un objet de type [ModelAndView]. Cet objet est ensuite utilisé par le contrôleur [DispatcherServlet] pour envoyer une réponse au client. Ici nous décidons d'envoyer la réponse nous-mêmes. C'est possible puisque nous disposons du paramètre [HttpServletResponse response] qui encapsule la réponse HTTP destinée au client.
- ligne 16 : précise que le flux qui va être envoyé est un flux HTML
- ligne 17 : récupère le flux d 'écriture vers le client
- lignes 18-20 : le flux HTML est envoyé
- ligne 22 : on retourne au contrôleur [DispatcherServlet] le pointeur **null** comme objet [ModelAndView]. Le contrôleur interprète ce pointeur **null** comme le fait que la réponse au client a été envoyée et qu'il ne doit pas s'en occuper.

Nous sommes prêts pour un test. Nous lançons Tomcat :



Dans la console, Spring affiche de nombreux logs qu'il est souvent intéressant de lire. Ceux liés à l'application [servlet-mvc-02] sont les suivants :

```
INFO [http-8080-Processor23] - Initializing servlet 'spring-mvc-02'
 INFO [http-8080-Processor23] - FrameworkServlet 'spring-mvc-02': initialization started
 INFO [http-8080-Processor23] - Loading WebApplicationContext for Spring FrameworkServlet 'spring-mvc-02'
 INFO [http-8080-Processor23] - Loading XML bean definitions from ServletContext resource [/WEB-INF/spring-mvc-02-
servlet.xml]
 INFO [http-8080-Processor23] - Bean factory for application context [WebApplicationContext for namespace 'spring-mvc-02-
servlet']: org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
[org.springframework.web.servlet.handler.SimpleUrlHandlerMapping,DoSomethingController]; root of BeanFactory hierarchy
 INFO [http-8080-Processor23] - 2 beans defined in application context [WebApplicationContext for namespace 'spring-mvc-
02-servlet']
 INFO [http-8080-Processor23] - JDK 1.4+ collections available
 INFO [http-8080-Processor23] - Commons Collections 3.x available
 INFO [http-8080-Processor23] - Unable to locate MessageSource with name 'messageSource': using default
[org.springframework.context.support.DelegatingMessageSource@1d314cc]
 INFO [http-8080-Processor23] - Unable to locate ApplicationEventMulticaster with name 'applicationEventMulticaster':
using default [org.springframework.context.event.SimpleApplicationEventMulticaster@31d5e2]
 INFO [http-8080-Processor23] - No ThemeSource found for [WebApplicationContext for namespace 'spring-mvc-02-servlet']:
using ResourceBundleThemeSource
 INFO [http-8080-Processor23] - Pre-instantiating singletons in factory
[org.springframework.beans.factory.support.DefaultListableBeanFactory defining beans
[org.springframework.web.servlet.handler.SimpleUrlHandlerMapping,DoSomethingController]; root of BeanFactory hierarchy]
 INFO [http-8080-Processor23] - Creating shared instance of singleton bean
'org.springframework.web.servlet.handler.SimpleUrlHandlerMapping'
 INFO [http-8080-Processor23] - Creating shared instance of singleton bean 'DoSomethingController'
 INFO [http-8080-Processor23] - Using context class [org.springframework.web.context.support.XmlWebApplicationContext]
for servlet 'spring-mvc-02'
 INFO [http-8080-Processor23] - Unable to locate MultipartResolver with name 'multipartResolver': no multipart request
handling provided
 INFO [http-8080-Processor23] - Unable to locate LocaleResolver with name 'localeResolver': using default
[org.springframework.web.servlet.i18n.AcceptHeaderLocaleResolver@1f31ad9]
 INFO [http-8080-Processor23] - Unable to locate ThemeResolver with name 'themeResolver': using default
[org.springframework.web.servlet.theme.FixedThemeResolver@1c0cd80]
 INFO [http-8080-Processor23] - No HandlerAdapters found in servlet 'spring-mvc-02': using default
 INFO [http-8080-Processor23] - No ViewResolvers found in servlet 'spring-mvc-02': using default
 INFO [http-8080-Processor23] - FrameworkServlet 'spring-mvc-02': initialization completed in 1592 ms
 INFO [http-8080-Processor23] - Servlet 'spring-mvc-02' configured successfully
```

Avec un navigateur, nous demandons l'URL [http://localhost:8080/spring-mvc-02/dosomething.html] :

DoSomething exécuté...

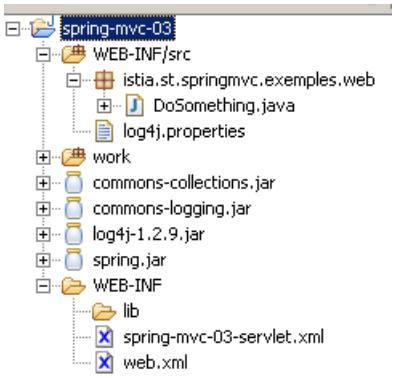
Expliquons ce qui s'est passé :

1. le fichier [web.xml] de l'application [spring-mvc-02] a été consulté. Ce fichier indique que les URL *.html doivent être traitées par la servlet de nom [spring-mvc-02] et instanciée par [DispatcherServlet].
2. le contrôleur [DispatcherServlet] prend la main et exploite le contenu du fichier [spring-mvc-02-servlet.html]. Il voit que l'URL [/dosomething.html] doit être traitée par le contrôleur [istia.st.springmvc.exemples.web.DoSomething]. Celui a été instancié au démarrage de l'application. [DispatcherServlet] appelle la méthode [handleRequest] de ce contrôleur.
3. la méthode [handleRequest] du contrôleur [istia.st.springmvc.exemples.web.DoSomething] envoie le flux HTML que nous voyons ci-dessus et rend le pointeur [null] au contrôleur principal [DispatcherServlet]. Celui-ci comprend que la réponse a été envoyée. Le cycle [demande client - réponse serveur] est terminé.

Voilà... Notre première application Spring a été écrite. Les briques de base de toute application Spring MVC commencent à se mettre en place.

6.3 La stratégie BeanNameUrlHandlerMapping

Le projet Eclipse de l'exemple est le suivant :



L'application est définie par le fichier [web.xml] suivant :

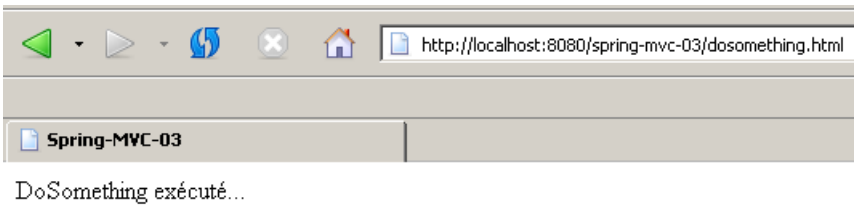
```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.
3.  <!DOCTYPE web-app PUBLIC
4.    "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
5.    "http://java.sun.com/dtd/web-app_2_3.dtd">
6.  <web-app>
7.    <!-- la servlet -->
8.    <servlet>
9.      <servlet-name>spring-mvc-03</servlet-name>
10.     <servlet-class>
11.       org.springframework.web.servlet.DispatcherServlet</servlet-class>
12.   </servlet>
13.   <!-- le mapping des url -->
14.   <servlet-mapping>
15.     <servlet-name>spring-mvc-03</servlet-name>
16.     <url-pattern>*.html</url-pattern>
17.   </servlet-mapping>
18. </web-app>
```

La servlet de l'application s'appelle [spring-mvc-03] et donc son fichier de configuration doit s'appeler [spring-mvc-03-servlet.xml]. Ce fichier est présent sur la copie d'écran ci-dessus. Son contenu 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 contrôleurs de l'application-->
5.    <bean name="/dosomething.html"
6.      class="istia.st.springmvc.exemples.web.DoSomething"/>
7.  </beans>
```

Il n'y a plus de stratégies de résolution. On sait qu'alors c'est la stratégie [BeanNameUrlHandlerMapping] qui est utilisée. C'est à dire qu'une url [/URL] est traitée par le bean contrôleur d'attribut **name** [/URL]. Dans une balise <bean>, l'attribut **name** sert également à identifier un bean comme le fait l'attribut **id**. Seulement l'attribut **id** a des règles de syntaxe qui ne permettent pas d'écrire [id= "/dosomething.html "], à cause du signe /. On utilise donc l'attribut **name**. Ci-dessus, l'url [/dosomething.html] sera traitée par le contrôleur d'attribut **name** [/dosomething.html] et donc par la classe [istia.st.springmvc.exemples.web.DoSomething]. Cette classe est celle de l'application précédente. Nous ne changeons rien sauf le titre de la page HTML que nous changeons en Spring-MVC-03.

Nous sommes prêts pour le test. Nous lançons Tomcat si besoin est, puis demandons l'URL [http://localhost:8080/spring-mvc-03/dosomething.html] :



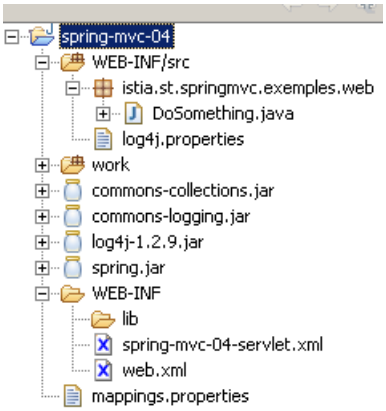
Expliquons ce qui s'est passé :

1. le fichier [web.xml] de l'application [spring-mvc-03] a été consulté. Ce fichier indique que les URL *.html doivent être traitées par la servlet de nom [spring-mvc-03] dont la classe associée est [DispatcherServlet].

2. le contrôleur [DispatcherServlet] prend la main et exploite le contenu du fichier [spring-mvc-03-servlet.html]. Sans stratégie de résolution définie [spring-mvc-03-servlet.html], il utilise la stratégie par défaut [BeanNameUrlhandlerMapping]. Il cherche donc un contrôleur d'id [/dosomething.html]. Il le trouve. [DispatcherServlet] appelle la méthode [handleRequest] de ce contrôleur.
3. la méthode [handleRequest] du contrôleur [istia.st.springmvc.exemples.web.DoSomething] envoie le flux HTML que nous voyons ci-dessus et rend le pointeur [null] au contrôleur principal [DispatcherServlet]. Celui-ci comprend que la réponse a été envoyée. Le cycle [demande client - réponse serveur] est terminé.

6.4 La stratégie SimpleUrlHandlerMapping avec liaisons externalisées

Le projet Eclipse de l'exemple est le suivant :



L'application est définie par un fichier [web.xml] analogue au précédent, mais désormais la servlet s'appelle [spring-mvc-04]. Le fichier de configuration [spring-mvc-04-servlet.xml] est le suivant :

```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les mappings de l'application-->
5.    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
6.      <property name="mappings">
7.        <bean
8.          class="org.springframework.beans.factory.config.PropertiesFactoryBean">
9.          <property name="location">
10.           <value>mappings.properties</value>
11.         </property>
12.       </bean>
13.     </property>
14.   </bean>
15.   <!-- les contrôleurs de l'application-->
16.   <bean id="DoSomethingController"
17.     class="istia.st.springmvc.exemples.web.DoSomething"/>
18. </beans>
```

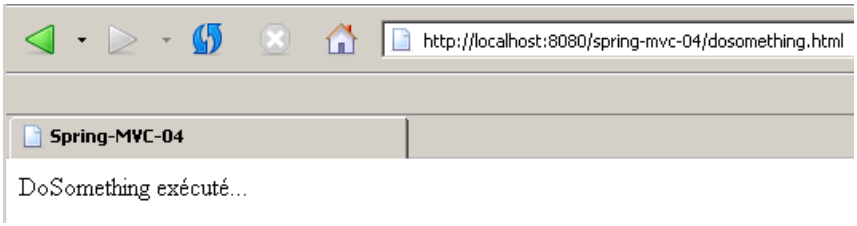
- la ligne 15 indique que nous sommes revenus à la stratégie de résolution [SimpleUrlHandlerMapping]. Cependant, les liens URL <-> Controller ne sont pas dans le même fichier mais dans le fichier de propriétés indiqué ligne 10.

Le fichier [mappings.properties] se trouve à la racine du dossier de l'application (cf copie d'écran) et son contenu est le suivant :

```
## mappings de l'application spring-mvc-04
/dosomething.html=DoSomethingController
```

Ce fichier a des lignes de la forme : Url=Controller, qui associent un objet Controller à une Url. On ne fait ici qu'externaliser des informations qui dans l'application [spring-mvc-02] étaient dans le fichier [spring-mvc-02-servlet.xml]. En-dehors de ce point, les applications spring-mvc-02 et spring-mvc-04 sont identiques.

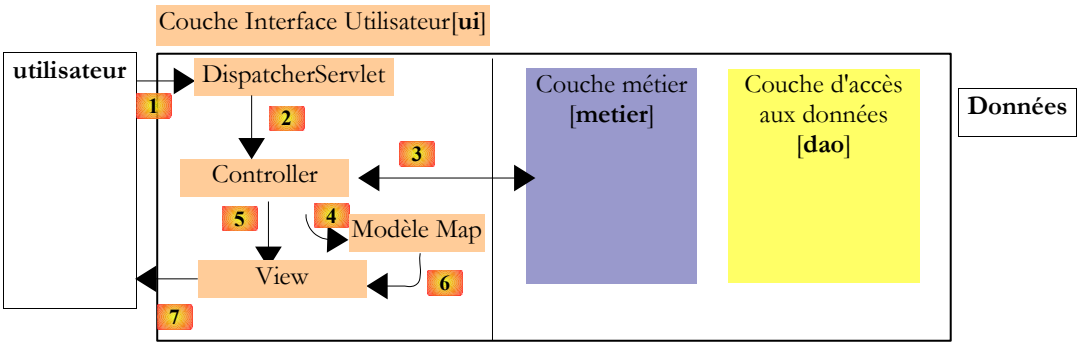
Pour le test, nous demandons l'URL [http://localhost:8080/spring-mvc-04/dosomething.html] :



7 Accès au contexte d'une application Spring MVC

7.1 Exemple 1

Reprenons l'architecture d'une application Spring MVC :



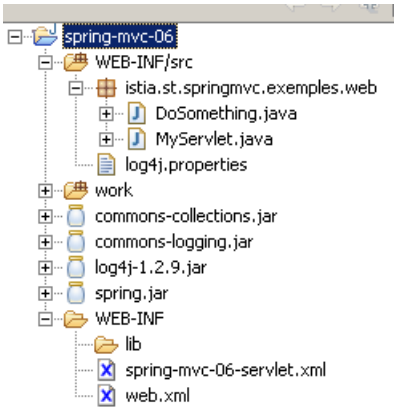
Pour traiter la demande d'un client, l'objet [Controller] a besoin d'informations de différente portée et situés pour cette raison dans des conteneurs différents :

- le **contexte de l'application** (javax.servlet.ServletContext) contient les objets qui sont partagés entre tous les utilisateurs. Parce qu'ils sont partagés entre tous les utilisateurs, ils sont le plus souvent en lecture seule. Leur durée de vie est celle de l'application.
- la **session** (javax.servlet.http.Session) contient les objets appartenant à un utilisateur donné. Celui fait au serveur des requêtes successives qui sont considérées à chaque fois par le serveur comme une nouvelle requête. Le mécanisme de la session permet d'avoir une mémoire entre les différentes requêtes d'un même utilisateur.
- le **contexte de la requête** (javax.servlet.http.HttpServletRequest) contient les objets d'une requête donnée d'un utilisateur donné. L'objet [HttpRequest] qui l'encapsule peut être traité par une chaîne de servlets. Le **contexte de la requête** permet à une servlet de passer des informations à la servlet suivante de la chaîne.

Lors de son démarrage, une application web a besoin de construire son environnement d'exécution. Il s'agit d'une phase d'initialisation qui aboutit à mettre dans le contexte de l'application, des objets partagés par tous les utilisateurs, par exemple ci-dessus, les objets instanciant les couches [métier] et [dao]. Où peut se faire cette initialisation ?

Lorsque la servlet d'une application web est chargée par le conteneur de servlets, sa méthode [init] est exécutée. C'est dans la spécification des servlets. Ce sera donc le cas également pour la servlet [DispatcherServlet] d'une application Spring MVC. Nous n'avons pas la maîtrise du contrôleur [DispatcherServlet]. Pour accéder à sa méthode [init] qui nous permettrait d'initialiser l'application, nous sommes amenés à dériver la classe [DispatcherServlet] afin de redéfinir sa méthode [init]. C'est ce que montre l'exemple que nous décrivons maintenant.

Le projet Eclipse est le suivant :



L'application web [spring-mvc-06] est configurée par le fichier [web.xml] suivant :

```
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.    <!-- la servlet -->
8.    <servlet>
9.      <servlet-name>spring-mvc-06</servlet-name>
10.     <servlet-class>
11.       istia.st.springmvc.exemples.web.MyServlet</servlet-class>
12.     <init-param>
13.       <param-name>groupe</param-name>
14.       <param-value>Paul,Mélanie,Jacques</param-value>
15.     </init-param>
16.   </servlet>
17.   <!-- le mapping des url -->
18.   <servlet-mapping>
19.     <servlet-name>spring-mvc-06</servlet-name>
20.     <url-pattern>*.html</url-pattern>
21.   </servlet-mapping>
22. </web-app>
```

Ce fichier [web.xml] ne diffère des précédents que par sa ligne 11. Le contrôleur C du MVC n'est plus de type [DispatcherServlet] comme jusqu'à maintenant mais de type [istia.st.springmvc.exemples.web.MyServlet], un type propriétaire défini dans [WEB-INF/src] (cf copie d'écran). Pour ne rien perdre des capacités de [DispatcherServlet], la classe [MyServlet] en dérive. Nous dérivons [DispacherServlet] afin d'avoir accès aux paramètres d'initialisation de l'application définis lignes 12-15 ci-dessus. Il s'agit de définir un groupe de personnes.

La lignée de la classe [DispatcherServlet] est la suivante :

org.springframework.web.servlet

Class DispatcherServlet

java.lang.Object
  └ javax.servlet.GenericServlet
    └ javax.servlet.http.HttpServlet
      └ org.springframework.web.servlet.HttpServletBean
        └ org.springframework.web.servlet.FrameworkServlet
          └ **org.springframework.web.servlet.DispatcherServlet**

La classe [DispatcherServlet] ne définit pas elle-même de méthode [init]. Dans la lignée ci-dessus, la méthode [init] apparaît tout d'abord dans [GenericServlet]. Les classes dérivées [HttpServlet, HttpServletBean] la redéfinissent. Seulement la classe [HttpServletBean] la redéfinit en la déclarant [final], ce qui interdit qu'elle soit redéfinie de nouveau dans les classes dérivées de [HttpServletBean]. Nous ne pouvons donc utiliser la méthode [init] pour initialiser l'application Spring MVC. En consultant la définition de la classe [DispatcherServlet], on trouve une méthode [initFrameworkServlet] qui peut être utilisée pour le faire :

initFrameworkServlet

```
protected void initFrameworkServlet()
                            throws ServletException,
                                   BeansException
```

Overridden method, invoked after any bean properties have been set and the WebApplicationContext and BeanFactory for this namespace is available.

Loads HandlerMapping and HandlerAdapter objects, and configures a ViewResolver and a LocaleResolver.

Overrides:

initFrameworkServlet in class FrameworkServlet

Throws:

ServletException - in case of an initialization exception

BeansException - if thrown by ApplicationContext methods

Lorsque cette méthode est appelée, le contexte de l'application [applicationContext.xml] a déjà été exploité. Par ailleurs, la méthode instancie différents objets nécessaires à l'application, notamment sa stratégie de résolution des URL (HandlerMapping). En redéfinissant cette méthode, on peut ajouter de plus des initialisations " propriétaires ".

La classe [MyServlet], que nous allons utiliser comme contrôleur principal de l'application, et dérivée de la classe [DispatcherServlet] est la suivante :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import javax.servlet.ServletException;
4.
5.  import org.springframework.beans.BeansException;
6.  import org.springframework.web.servlet.DispatcherServlet;
7.
8.  public class MyServlet extends DispatcherServlet {
9.
10.  public void initFrameworkServlet() throws BeansException, ServletException {
11.    // init parent
12.    super.initFrameworkServlet();
13.    // on récupère le paramètre de [web.xml]
14.    String[] groupe = (this.getServletConfig().getInitParameter("groupe"))
15.        .split(",");
16.    // on met le tableau dans le contexte de l'application
17.    this.getServletContext().setAttribute("groupe", groupe);
18.  }
19. }
```

- ligne 8 : la classe dérive de [DispatcherServlet]
- lignes 10-18 : redéfinissent la méthode [initFrameworkServlet] de la classe [DispatcherServlet]
- ligne 12 : nous laissons la méthode [initFrameworkServlet] de la classe parent faire son travail
- lignes 13-17 : nous ajoutons nos initialisations propriétaires
- ligne 14 : on récupère dans [web.xml], le paramètre (init-param) qui porte le nom " groupe ". On obtient une chaîne de caractères constituée de sous-chaînes séparées par une virgule. Ces sous-chaînes sont récupérées dans le tableau **groupe**.
- ligne 17 : le tableau **groupe** est mis dans le contexte de l'application afin qu'il soit visible par tous les utilisateurs de l'application.

Le fichier [web.xml] définissait une servlet appelée [servlet-mvc-06]. Le fichier de configuration [servlet-mvc-06-servlet.xml] de celle-ci est le suivant :

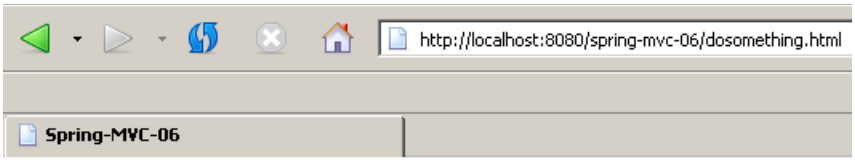
```
1.  <?xml version="1.0" encoding="UTF-8"?>
2.  <!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
    beans.dtd">
3.  <beans>
4.    <!-- les mappings de l'application-->
5.    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
6.      <property name="mappings">
7.        <props>
8.          <prop key="/dosomething.html">DoSomethingController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="DoSomethingController"
14.     class="istia.st.springmvc.exemples.web.DoSomething"/>
15. </beans>
```

Il n'y a rien ici qu'on ne connaisse déjà. L'url [/dosomething.html] va être traitée par le contrôleur [istia.st.springmvc.exemples.web.DoSomething]. Celui-ci est dans [WEB-INF/src] (cf copie d'écran). Le code du contrôleur [DoSomething] est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.io.PrintWriter;
4.  import java.util.Date;
5.
6.  import javax.servlet.http.HttpServletRequest;
7.  import javax.servlet.http.HttpServletResponse;
8.
9.  import org.springframework.web.servlet.ModelAndView;
10. import org.springframework.web.servlet.mvc.Controller;
11.
12. public class DoSomething implements Controller {
13.
14.   // gestion de la requête
15.   public ModelAndView handleRequest(HttpServletRequest request,
16.       HttpServletResponse response) throws Exception {
17.     // début
18.     long début = new Date().getTime();
19.     // attente
20.     Thread.sleep(10);
21.     // on code en dur
22.     response.setContentType("text/html");
23.     PrintWriter out = response.getWriter();
24.     // on prépare le code HTML
25.     String Html = "<html><head><title>Spring-MVC-06</title></head>"
26.         + "<body>";
27.     // on parcourt la liste des membres du groupe
28.     String[] groupe = (String[]) request.getSession().getServletContext().getAttribute("groupe");
29.     for (int i = 0; i < groupe.length; i++) {
30.       Html += groupe[i] + "<br>\n";
31.     }
32.     // fin
33.     long fin = new Date().getTime();
34.     // durée
35.     long durée = fin - début;
36.     Html += "<br>DoSomething exécuté en " + durée + " ms...</body></html>";
37.     // on envoie le flux HTML
38.     out.println(Html);
39.     // pas de ModelAndView
40.     return null;
41.   }
42.
43. }
```

- ligne 12 : [DoSomething] implémente l'interface [Controller].
- lignes 15-42 : redéfinissent la méthode [HandleRequest] qui sera appelée par le contrôleur [DispatcherServlet]
- ligne 18 : on note l'heure de début du traitement
- ligne 20 : on attend 10 ms
- ligne 22 : on indique que la réponse sera un flux HTML
- lignes 23 : le flux d'écriture vers le client
- lignes 25 - 26 : début du flux HTML
- ligne 28 : on récupère l'attribut " groupe " dans le contexte de l'application.
- lignes 29-31 : les membres du groupe sont écrits dans le flux HTML
- ligne 33 : on note l'heure de fin du traitement
- ligne 35 : la durée du traitement
- ligne 36 : la durée est écrite dans le flux HTML
- ligne 38 : envoi du flux HTML au client
- ligne 40 : on rend le pointeur [null] à [DispatcherServlet] pour lui indiquer que la réponse a été envoyée.

Nous sommes prêts pour un test. Après avoir lancé Tomcat, nous demandons l'url [http://localhost:8080/spring-mvc-06/dosomething.html] :

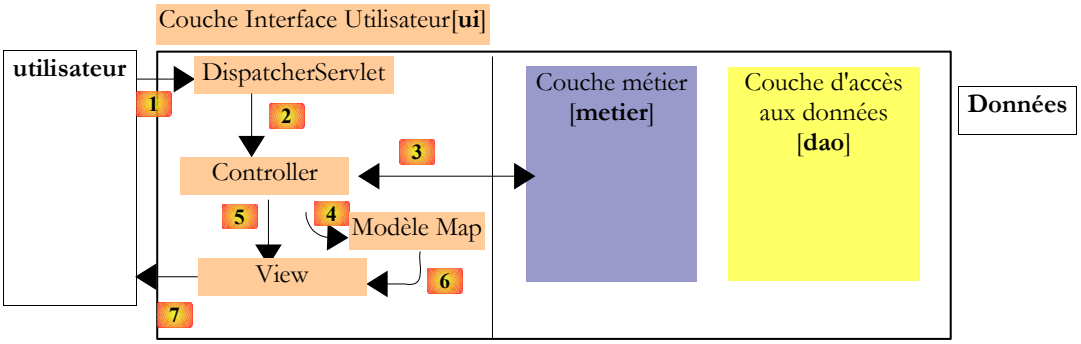


Paul
Mélanie
Jacques

DoSomething exécuté en 20 ms...

7.2 Exemple 2

Reprenons l'architecture d'une application Spring MVC :



Les différents objets [Controller] qui traitent les demandes des utilisateurs ont normalement besoin de communiquer avec la couche métier (opération 3 ci-dessus). Celle-ci est accédée en général via une ou plusieurs interfaces. L'instanciation de celles-ci peut être faite par configuration avec Spring IoC. Les instances créées doivent être placées dans le contexte de l'application car ce sont des singletons partagés par tous les utilisateurs.

Le fichier [web.xml] utilisé précédemment pour créer le contexte de l'application n'est pas un fichier Spring et ne peut donc être utilisé pour instancier les couches [métier] et [dao] avec Spring IoC. Spring MVC propose donc une autre approche.

Nous avons vu que les applications précédentes étaient associées à une servlet S et à un fichier de configuration [S-servlet.xml]. On pourrait vouloir instancier la couche [métier] dans ce dernier fichier. Cependant, une application Spring MVC peut utiliser diverses servlets S1, S2, ... donnant naissance à des fichiers [S1-servlet.xml, S2-servlet.xml, ...]. Afin que les singletons des couches [métier] et [dao] soient accessibles aux différentes servlets, on définit ces singletons dans un fichier de configuration Spring " parent " des fichiers de servlets [Si-servlet.xml]. Spring permet en effet de créer une relation parent - fils entre deux fichiers de configuration. Les beans définis dans le fichier parent sont automatiquement connus du fichier fils.

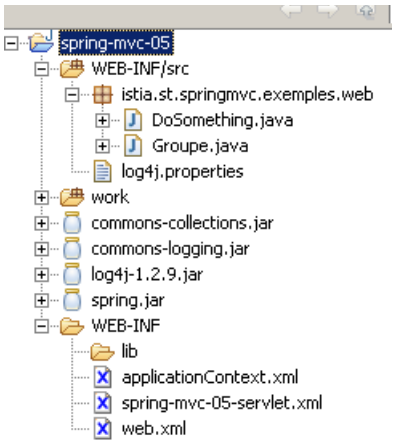
Dans une application Spring MVC, le fichier [applicationContext.xml] joue le rôle de " parent " des fichiers [S1-servlet.xml, S2-servlet.xml, ...] de définition des servlets de l'application. C'est donc dans ce fichier qu'on placera en général la configuration des couches [métier] et [dao] de l'application. Le fichier [applicationContext.xml] doit être placé dans le dossier [WEB-INF].

Pour que le fichier [applicationContext.xml] soit reconnu et exploité, l'application Spring MVC doit charger une classe spéciale de type [org.springframework.web.context.ContextLoaderListener]. Cette classe va exploiter le fichier [applicationContext.xml] et instancier les beans qui y sont définis. Parce que ceux-ci doivent être instanciés avant ceux définis par les fichiers [Si-servlet.xml], la classe [org.springframework.web.context.ContextLoaderListener] doit être instanciée avant la classe [DispatcherServlet] qui elle, va exploiter les fichiers [Si-servlet.xml]. Cela peut être obtenu au moyen d'un fichier [web.xml] analogue au suivant :

```
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-05</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-05</servlet-name>
21.     <url-pattern>*.html</url-pattern>
22.   </servlet-mapping>
23. </web-app>
```

Ce fichier est semblable aux précédents sauf en ce qui concerne les lignes 8-11. Ces lignes définissent un " listener " d'application qui sera instancié dès le démarrage de l'application et avant les classes associées aux servlets. Ceci nous assure que les beans définis dans [applicationContext.xml] seront instanciés avant ceux définis dans les fichiers [Si-servlet.xml].

Pour illustrer le rôle du fichier [applicationContext.xml], nous utiliserons le projet Eclipse suivant :



On notera la présence du fichier [applicationContext.xml] dans [WEB-INF]. Le fichier [web.xml] de l'application est celui qui vient d'être décrit. Le fichier [applicationContext.xml] est, lui, 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.    <!-- une liste de personnes -->
5.    <bean id="groupe" class="istia.st.springmvc.exemples.web.Groupe">
6.      <property name="membres">
7.        <list>
8.          <value>Paul</value>
9.          <value>Mélanie</value>
10.         <value>Jacques</value>
11.       </list>
12.     </property>
13.   </bean>
14. </beans>
```

Il définit un bean " groupe ", objet de type [istia.st.springmvc.exemples.web.Groupe]. Cette classe, définie dans [WEB-INF/src] (cf copie d'écran) est définie comme suit :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.ArrayList;
4.
5.  public class Groupe {
6.
7.    private ArrayList membres;
8.
9.    public ArrayList getMembres() {
10.     return membres;
11.   }
12.
13.   public void setMembres(ArrayList membres) {
14.     this.membres = membres;
15.   }
16. }
```

- ligne 7 : un champ privé de type [ArrayList]
- lignes 9 - 15 : getter / setter associés

Le fichier [applicationContext.xml] définit un bean d'id " groupe " qui est une instance de la classe [Groupe] avec pour éléments du champ privé [membres], les chaînes de caractères : " Paul ", " Mélanie ", " Jacques ".

Le fichier de configuration [servlet-mvc-05-servlet.xml] de la servlet [spring-mvc-05] 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="/dosomething.html">DoSomethingController</prop>
9.        </props>
```

```
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="DoSomethingController"
14.     class="istia.st.springmvc.exemples.web.DoSomething">
15.     <property name="groupe">
16.       <ref bean="groupe"/>
17.     </property>
18.   </bean>
19. </beans>
```

- lignes 5-11 : définissent [SimpleUrlHandlerMapping] comme stratégie de résolution des URL
- lignes 13-18 : définissent le contrôleur chargé de traiter l'url [/dosomething.html]. Ce contrôleur est une instance de la classe [istia.st.springmvc.exemples.web.DoSomething]. On voit (lignes 15-17) que ce contrôleur a une propriété nommée " groupe " qui reçoit pour valeur (ligne 16), la valeur d'un bean d'id " groupe ". Il s'agit du bean instancié par [applicationContext.xml]. La relation parent - fils qui lie les fichiers [applicationContext.xml] et [servlet-mvc-05-servlet.xml] fait que le bean " groupe " défini dans [applicationContext.xml] peut être utilisé dans [servlet-mvc-05-servlet.xml].

Il ne nous reste plus qu'à présenter le contrôleur [istia.st.springmvc.exemples.web.DoSomething] :

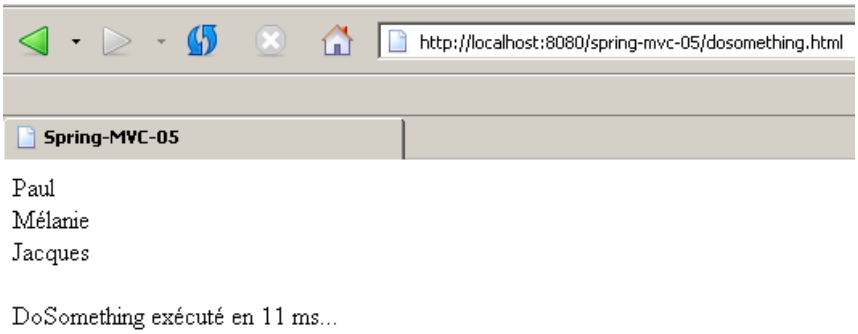
```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.io.PrintWriter;
4.  import java.util.ArrayList;
5.  import java.util.Date;
6.
7.  import javax.servlet.http.HttpServletRequest;
8.  import javax.servlet.http.HttpServletResponse;
9.
10. import org.springframework.web.servlet.ModelAndView;
11. import org.springframework.web.servlet.mvc.Controller;
12.
13. public class DoSomething implements Controller {
14.
15.   // un groupe de personnes fourni par le contexte de l'application
16.   private Groupe groupe;
17.   public Groupe getGroupe() {
18.     return groupe;
19.   }
20.   public void setGroupe(Groupe groupe) {
21.     this.groupe = groupe;
22.   }
23.
24.   // gestion de la requête
25.   public ModelAndView handleRequest(HttpServletRequest request,
26.       HttpServletResponse response) throws Exception {
27.     // début
28.     long début = new Date().getTime();
29.     // attente
30.     Thread.sleep(10);
31.     // on code en dur
32.     response.setContentType("text/html");
33.     PrintWriter out = response.getWriter();
34.     // on prépare le code HTML
35.     String Html="<html><head><title>Spring-MVC-05</title></head>"+
36.             "<body>";
37.     // on parcourt la liste des membres du groupe
38.     ArrayList membres=groupe.getMembres();
39.     for(int i=0;i<membres.size();i++){
40.       Html+=membres.get(i).toString()+"<br>\n";
41.     }
42.     // fin
43.     long fin = new Date().getTime();
44.     // durée
45.     long durée = fin - début;
46.     Html += "<br>DoSomething exécuté en " + durée + " ms...</body></html>";
47.     // on envoie le flux HTML
48.     out.println(Html);
49.     // pas de ModelAndView
50.     return null;
51.   }
52.
53. }
```

Ce contrôleur est quasi identique à celui de l'exemple précédent. Les différences sont les suivantes :

- lignes 15 - 21 : définissent un champ privé [groupe] de type [Groupe] avec ses *getter* et *setter*. Nous avons vu que ce champ était initialisé dans [servlet-mvc-05-servlet.xml] par le bean d'id " groupe " défini dans [applicationContext.xml]. Le champ privé [groupe] du contrôleur va donc être initialisé avec une instance de classe de type [Groupe] dont le champ privé " membres " contiendra les chaînes de caractères : " Paul ", " Mélanie ", " Jacques ". Lorsque la méthode [handleRequest]

est exécutée, le champ privé [groupe] aura donc déjà sa valeur alors que dans la version précédente il fallait récupérer cette valeur dans le contexte de l'application.

Testons. Demandons l'url [http://localhost:8080/spring-mvc-05/dosomething.html] :



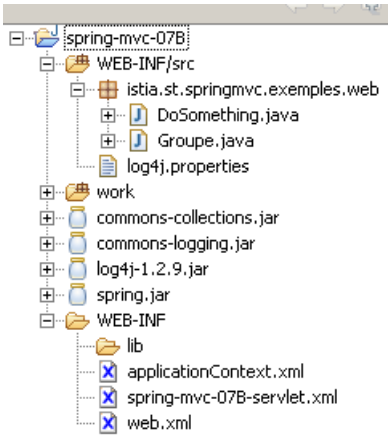
7.3 Exemple 3

Dans les deux versions précédentes, les membres du groupe à afficher :

- exemple 1 : étaient définis dans [web.xml], placés dans le contexte de l'application par le contrôleur principal [MyServlet] puis récupérés par le contrôleur [doSomething] dans le contexte de l'application
- exemple 2 : étaient définis dans [applicationContext.xml], et injectés directement dans le contrôleur [doSomething] par configuration du contrôleur [DoSomething] dans [servlet-mvc-05-servlet.xml]

Dans ce nouvel exemple, les membres du groupe seront définis dans [applicationContext.xml] comme dans l'exemple 2 mais pas injectés dans le contrôleur [DoSomething]. Nous voulons montrer comment le contrôleur [DoSomething] peut avoir accès au contenu du fichier [applicationContext.xml].

Le projet Eclipse est le suivant :



Nous voyons la présence du fichier [applicationContext.xml]. Son contenu est identique à celui de la version précédente. Il en est de même pour le fichier [web.xml] en-dehors du fait que celui-ci référence désormais une servlet appelée [spring-mvc-07B]. Le fichier de configuration [spring-mvc-07B-servlet.xml] de celle-ci a le contenu 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="/dosomething.html">DoSomethingController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="DoSomethingController"
14.     class="istia.st.springmvc.exemples.web.DoSomething">
15.   </bean>
16. </beans>
```

On a là, un contenu analogue à celui de [spring-mvc-05-servlet.xml], si ce n'est que le bean " groupe " n'est plus injecté dans le contrôleur [DoSomething]. Comment celui-ci peut-il avoir accès au contenu de [applicationContext.xml] ?

Pour accéder au contenu du fichier [applicationContext.xml], le contrôleur peut utiliser la classe [org.springframework.web.servlet.support.RequestContextUtils]. Cette classe offre des méthodes utilitaires statiques dont la suivante :

static WebApplicationContext	**getWebApplicationContext**(ServletRequest request) Look for the WebApplicationContext associated with the DispatcherServlet that has initiated request processing.

On voit qu'à partir de la requête du client, on est capable d'avoir accès au contexte de l'application Spring MVC.

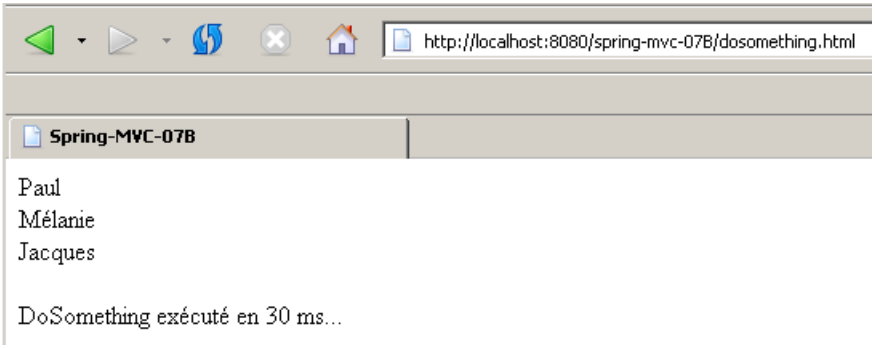
Le contrôleur [DoSomething] sera le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.io.PrintWriter;
4.  import java.util.ArrayList;
5.  import java.util.Date;
6.
7.  import javax.servlet.http.HttpServletRequest;
8.  import javax.servlet.http.HttpServletResponse;
9.
10. import org.springframework.web.servlet.ModelAndView;
11. import org.springframework.web.servlet.mvc.Controller;
12. import org.springframework.web.servlet.support.RequestContextUtils;
13.
14. public class DoSomething implements Controller {
15.
16.  // gestion de la requête
17.  public ModelAndView handleRequest(HttpServletRequest request,
18.      HttpServletResponse response) throws Exception {
19.    // début
20.    long début = new Date().getTime();
21.    // attente
22.    Thread.sleep(10);
23.    // on récupère le bean groupe
24.    Groupe groupe=(Groupe)RequestContextUtils.getWebApplicationContext(request).getBean("groupe");
25.    // on code le HTML en dur
26.    response.setContentType("text/html");
27.    PrintWriter out = response.getWriter();
28.    // on prépare le code HTML
29.    String Html="<html><head><title>Spring-MVC-07</title></head>"+
30.          "<body>";
31.    // on parcourt la liste des membres du groupe
32.    ArrayList membres=groupe.getMembres();
33.    for(int i=0;i<membres.size();i++){
34.      Html+=membres.get(i).toString()+"<br>\n";
35.    }
36.    // fin
37.    long fin = new Date().getTime();
38.    // durée
39.    long durée = fin - début;
40.    Html += "<br>DoSomething exécuté en " + durée + " ms...</body></html>";
41.    // on envoie le flux HTML
42.    out.println(Html);
43.    // pas de ModelAndView
44.    return null;
45.  }
46.
47. }
```

Ce contrôleur est analogue à celui des exemples précédents. Notons les différences :

- ligne 24 : la classe [RequestContextUtils] est utilisée pour retrouver le bean " groupe " défini dans [applicationContext.xml].

Un test d'exécution est le suivant :



7.4 Exemple 4

Spring met à notre disposition diverses classes prédéfinies implémentant l'interface [Controller]. Elles disposent de certaines caractéristiques qu'il peut être intéressant d'exploiter en les dérivant. L'une des classes de Spring implémentant l'interface [Controller] est la classe [AbstractController] :

org.springframework.web.servlet.mvc

Class AbstractController

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**

All Implemented Interfaces:

ApplicationContextAware, Controller

Direct Known Subclasses:

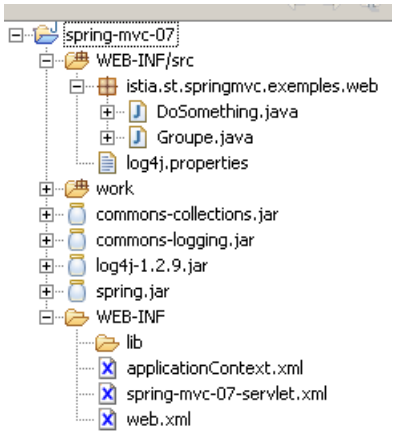
AbstractUrlViewController, BaseCommandController, MultiActionController, ParameterizableViewController, ServletForwardingController, ServletWrappingController

La méthode [handleRequest] de cette classe est appelée par le contrôleur [DispatcherServlet] ou dérivé. Dans son implémentation, la méthode [handleRequest] de [AbstractController] appelle la méthode [handleRequestInternal]. Cette méthode n'a pas de contenu dans [AbstactController] et est déclarée abstraite. Sa signature est la suivante :

protected abstract ModelAndView	**handleRequestInternal**(HttpServletRequest request, HttpServletResponse response) Template method.

Elle a donc la même signature que la méthode [handleRequest].

Le nouveau projet Eclipse est le suivant :



Le contenu du fichier [applicationContext.xml] est identique à celui de la version précédente. Il en est de même pour le fichier [web.xml] en-dehors du fait que celui-ci référence désormais une servlet appelée [spring-mvc-07]. Le contenu de [spring-mvc-07-servlet.xml] est identique à celui de [spring-mvc-07B-servlet.xml].

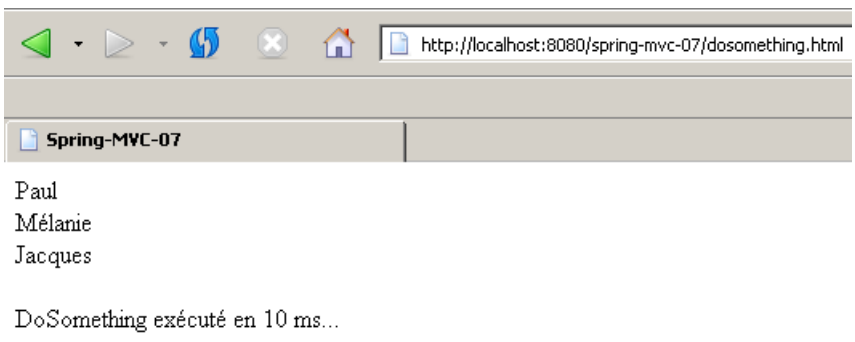
Le contrôleur [DoSomething] sera le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.io.PrintWriter;
4.  import java.util.ArrayList;
5.  import java.util.Date;
6.
7.  import javax.servlet.http.HttpServletRequest;
8.  import javax.servlet.http.HttpServletResponse;
9.
10. import org.springframework.web.servlet.ModelAndView;
11. import org.springframework.web.servlet.mvc.AbstractController;
12.
13. public class DoSomething extends AbstractController {
14.
15.  // un groupe de personnes fourni par le contexte de l'application
16.  private Groupe groupe;
17.  // gestion de la requête
18.  public ModelAndView handleRequestInternal(HttpServletRequest request,
19.      HttpServletResponse response) throws Exception {
20.    // début
21.    long début = new Date().getTime();
22.    // attente
23.    Thread.sleep(10);
24.    // on récupère le bean groupe
25.    groupe=(Groupe)getWebApplicationContext().getBean("groupe");
26.    // on code le HTML en dur
27.    response.setContentType("text/html");
28.    PrintWriter out = response.getWriter();
29.    // on prépare le code HTML
30.    String Html="<html><head><title>Spring-MVC-07</title></head>"+
31.           "<body>";
32.    // on parcourt la liste des membres du groupe
33.    ArrayList membres=groupe.getMembres();
34.    for(int i=0;i<membres.size();i++){
35.      Html+=membres.get(i).toString()+"<br>\n";
36.    }
37.    // fin
38.    long fin = new Date().getTime();
39.    // durée
40.    long durée = fin - début;
41.    Html += "<br>DoSomething exécuté en " + durée + " ms...</body></html>";
42.    // on envoie le flux HTML
43.    out.println(Html);
44.    // pas de ModelAndView
45.    return null;
46.  }
47.
48. }
```

Une nouvelle fois, ce contrôleur est très proche de celui des versions précédentes. Notons les différences :

- ligne 13 : le contrôleur dérive de [AbstractController]. Ceci nous permet ligne 25 de récupérer les membres du groupe définis dans le fichier [applicationContext.xml]. On utilise pour cela la méthode [getWebApplicationContext], une méthode définie dans la classe [org.springframework.context.support.ApplicationObjectSupport], une classe parent de [AbstractController].
- lignes 18-45 : redéfinissent la méthode [handleRequestInternal] de la classe [AbstractController].

Pour le test, nous demandons l'URL [http://localhost:8080/spring-mvc-07/dosomething.html] :

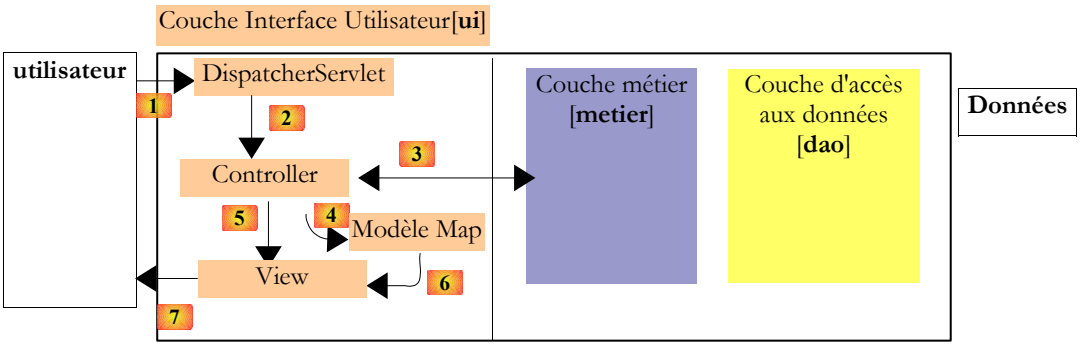


8 Gestion des vues

8.1 La place des vues dans l'architecture Spring MVC

L'architecture Spring MV a été décrite au paragraphe 4, page 9. Nous revenons sur cette architecture afin d'y situer la place des vues. Ce qui suit est un simple copier-coller de ce qui a été présenté au paragraphe 4, page 9 mais nous avons voulu éviter au lecteur de revenir en arrière et attirer son attention sur des points qui, lorsqu'ils ont été exposés, étaient peut-être assez flous dans son esprit. Le lecteur averti peut lui passer directement au paragraphe 8.2, page 38.

L'architecture 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, ...

Etapes 1-2 : choix du contrôleur traitant l'action demandée par l'utilisateur

Avec Spring MVC, c'est à partir de l'URL demandée par le client que le contrôleur principal [DispatcherServlet] va choisir l'objet [Controller] qui va traiter l'action. Le lien URL <-> Controller est fait par configuration. Il existe plusieurs méthodes de résolution possibles, c.a.d. plusieurs façons possibles de lier une URL à un objet [Controller].

Diverses méthodes de résolution des Url (HandlerMapping) ont été présentées.

L'objet [Controller] chargé de traiter l'action est une instance de classe implémentant l'interface [**org.springframework.web.servlet.mvc.Controller**].

Spring offre plusieurs classes implémentant l'interface [Controller] :

org.springframework.web.servlet.mvc

Interface Controller

All Known Implementing Classes:

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

Ces classes sont normalement destinées à être dérivées. On se souvient en effet, que le contrôleur est chargé de traiter une action d'une application spécifique. Ceci ne peut être fait dans les classes génériques ci-dessus sauf lorsqu'il n'y a pas de traitement à faire (ParameterizableViewController, ServletForwardingController). De façon générale, c'est au développeur de construire la classe implémentant l'interface [Controller]. Il peut parfois se faire aider en dérivant cette classe des classes ci-dessus. C'est notamment le cas si l'action demandée est le POST d'un formulaire. Dans ce cas particulier important, la classe [SimpleFormController] apporte des facilités de développement.

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

Method Summary	
ModelAndView	**handleRequest**(HttpServletRequest request, HttpServletResponse response) Process the request and return a ModelAndView object which the DispatcherServlet will render.

- l'unique méthode à implémenter par un contrôleur est donc [**handleRequest**] qui a pour paramètres, l'objet [request] qui encapsule la requête HTTP du client, et l'objet [response] qui, lui, encapsule la réponse HTTP qui sera faite à ce même client.
- comme le montre le schéma plus haut, un contrôleur doit désigner une vue V à afficher en réponse à la demande du client et le modèle M qui sera affiché par cette vue. Ces deux éléments sont rendus par [**handleRequest**] sous la forme d'un objet de type [**ModelAndView**] plus exactement [**org.springframework.web.servlet.ModelAndView**].

Un objet de type [**ModelAndView**] peut être construit de diverses façons :

Constructor Summary	
ModelAndView()	Default constructor for bean-style usage: populating bean properties instead of passing in constructor arguments. 1
ModelAndView(String viewName)	Convenient constructor when there is no model data to expose. 2
ModelAndView(String viewName, Map model)	Creates new ModelAndView given a view name and a model. 3
ModelAndView(String viewName, String modelName, Object modelObject)	Convenient constructor to take a single model object. 4
ModelAndView(View view)	Convenient constructor when there is no model data to expose. 5
ModelAndView(View view, Map model)	Creates new ModelAndView given a View object and a model. 6
ModelAndView(View view, String modelName, Object modelObject)	Convenient constructor to take a single model object. 7

1. le constructeur [1] ne définit ni modèle, ni vue. Ceux-ci sont alors définis ultérieurement via les méthodes de la classe [ModelAndView]
2. le constructeur [2] définit une vue **V** d'après son nom. Nous savons qu'au final, la vue est un objet implémentant l'interface [**org.springframework.web.servlet.View**]. Le lien nom <-> objet View est alors fait dans un fichier de configuration. Le constructeur [2] ne précise pas de modèle. Il convient donc pour une vue sans modèle. On peut aussi préciser le modèle ultérieurement via des méthodes de [ModelAndView].
3. le constructeur [3] définit une vue **V** d'après son nom et définit également un modèle **M** sous la forme d'un objet implémentant l'interface [java.util.Map], donc un dictionnaire.
4. le constructeur [4] est une variante du constructeur [3] lorsque le modèle n'a qu'un unique élément. Dans ce cas, le dictionnaire n'a qu'un élément dont la clé sera [modelName] et la valeur associée [modelObject].
5. les constructeurs [5, 6, 7] sont des variantes des constructeurs [2, 3, 4] dans lesquelles, la vue V n'est plus désignée par son nom mais directement instanciée par un objet implémentant l'interface [View]. L'association nom vue <-> objet View n'a alors plus à être faite dans le fichier de configuration.

Les méthodes de la classe [ModelAndView] sont les suivantes :

Method Summary	
ModelAndView	**addAllObjects**(Map modelMap) Add all entries contained in the provided map to the model. 1
ModelAndView	**addObject**(String modelName, Object modelObject) Add an object to the model. 2
void	**clear**() Clear the state of this ModelAndView object.
Map	**getModel**() Return the model map.
protected Map	**getModelInternal**() Return the model map.
View	**getView**() Return the View object, or null if we are using a view name to be resolved by the DispatcherServlet via a ViewResolver.
String	**getViewName**() Return the view name to be resolved by the DispatcherServlet via a ViewResolver, or null if we are using a View object.

- les méthodes [1] et [2] permettent d'ajouter des éléments au modèle M

boolean	**isEmpty**() Return whether this ModelAndView object is empty, i.e. whether it does not hold any view and does not contain a model.
boolean	**isReference**() Return whether we use a view reference, i.e. true if the view has been specified via a name to be resolved by the DispatcherServlet via a ViewResolver.
void	**setView**(View view) Set a View object for this ModelAndView. 3
void	**setViewName**(String viewName) Set a view name for this ModelAndView, to be resolved by the DispatcherServlet via a ViewResolver. 4
String	**toString**() Return diagnostic information about this model and view.

- les méthodes [3] et [4] permettent de préciser la vue V soit par une instance View [3], soit par le nom de la vue [4].

Etapes 6-7 : rendu du modèle M par la vue V

Arrivé ici, le flux d'exécution a été transféré à une vue [View]. [View] est une interface avec l'unique méthode suivante :

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

- assez logiquement, l'unique méthode [render] dispose de l'objet [response] à partir duquel la réponse HTTP va être construite et du modèle [model] construit précédemment par le contrôleur. La méthode [render] dispose également de l'objet [request] qui encapsule la requête HTTP du client. Via cet objet, la méthode [render] a accès au contexte de la requête, à la session qui lui est liée, ...

L'une des implémentations de l'interface [View] est la classe [JstlView] qui implémente la vue à l'aide d'une page JSP utilisant des balises JSTL. La méthode [render] de la classe [JstlView] met les éléments du dictionnaire [model] qu'elle a reçu dans le contexte de la requête [request]. C'est là que la page JSP qui sera envoyée au client trouvera son modèle.

Spring propose plusieurs implémentations de l'interface [View] :

org.springframework.web.servlet

Interface View

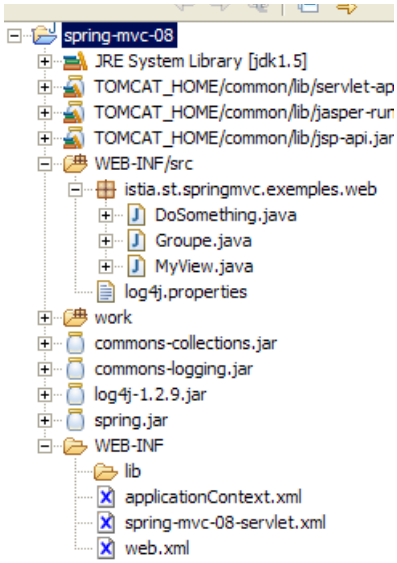
All Known Implementing Classes:

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

8.2 Implémenter l'interface View

Même si Spring offre beaucoup de classes implémentant l'interface **View**, il est intéressant de l'implémenter une fois soi-même afin de mieux comprendre le mécanisme qui lie le contrôleur [Controller] à la vue [View] à qui il demande d'afficher un modèle [Map].

Notre premier exemple Eclipse sera le suivant :



Le fichier [web.xml] du projet est classique :

```
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-08</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-08</servlet-name>
21.     <url-pattern>*.html</url-pattern>
22.   </servlet-mapping>
23. </web-app>
```

On rappelle que les lignes 8-11 sont nécessaires pour charger la classe chargée d'exploiter le fichier [applicationContext.xml] initialisant le contexte de l'application. Le contenu de 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.    <!-- une liste de personnes -->
5.    <bean id="groupe" class="istia.st.springmvc.exemples.web.Groupe">
6.      <property name="membres">
7.        <list>
8.          <value>Paul</value>
9.          <value>Mélanie</value>
10.         <value>Jacques</value>
11.       </list>
12.     </property>
13.   </bean>
14. </beans>
```

C'est un contenu déjà rencontré qui définit le bean "groupe" de type [istia.st.springmvc.exemples.web.Groupe]. Nous ne revenons pas sur ce fichier déjà expliqué.

Le fichier [spring-mvc-08-servlet.xml] de définition de la servlet [spring-mvc-08] 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="/dosomething.html">DoSomethingController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="DoSomethingController"
14.     class="istia.st.springmvc.exemples.web.DoSomething">
15.     <property name="groupe">
16.       <ref bean="groupe"/>
17.     </property>
18.   </bean>
19. </beans>
```

Là encore, rien de neuf. L'URL [/dosomething.html] sera traitée par le contrôleur [istia.st.springmvc.exemples.web.DoSomething]. Ce contrôleur a un champ privé "groupe" qui sera initialisé au démarrage de l'application par le bean "groupe" défini dans [applicationContext.xml].

Le contrôleur [DoSomething] amène un premier changement :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Date;
4.  import java.util.HashMap;
5.
6.  import javax.servlet.http.HttpServletRequest;
7.  import javax.servlet.http.HttpServletResponse;
8.
9.  import org.springframework.web.servlet.ModelAndView;
10. import org.springframework.web.servlet.mvc.Controller;
11.
12. public class DoSomething implements Controller {
13.
14.   // un groupe de personnes fourni par le contexte de l'application
15.   private Groupe groupe;
16.
17.   public Groupe getGroupe() {
18.     return groupe;
19.   }
20.
21.   public void setGroupe(Groupe groupe) {
22.     this.groupe = groupe;
23.   }
24.
25.   // gestion de la requête
26.   public ModelAndView handleRequest(HttpServletRequest request,
27.       HttpServletResponse response) throws Exception {
28.     // début
29.     long début = new Date().getTime();
30.     // on fait qq chose...
31.     Thread.sleep(10);
32.     // fin
```

```
33.    long fin = new Date().getTime();
34.    // durée
35.    long durée = fin - début;
36.    // on crée le modèle de la vue à afficher
37.    Map modèle=new HashMap();
38.    modèle.put("groupe",groupe);
39.    modèle.put("durée",new Long(durée));
40.    // on retourne le ModelAndView
41.    return new ModelAndView(new MyView(),modèle);
42.  }
43.
44. }
```

- ligne 12 : [DoSomething] implémente l'interface [Controller]
- lignes 26-42 : la méthode [handleRequest] de cette interface est définie. Jusqu'à maintenant, le contrôleur avait toujours généré lui-même la réponse au client. Grâce à l'objet [response] qu'elle reçoit en paramètre, la méthode [handleRequest] générait le flux HTML vers le client et rendait un ModelAndView *null* à [DispatcherServlet] afin d'indiquer que la réponse avait été envoyée et que [DispatcherServlet] n'avait plus à le faire. Ligne 41, on voit que la méthode rend cette fois une instance de [ModelAndView].
- ligne 41 : on utilise le constructeur **ModelAndView(View view, Map modèle)**. Le premier paramètre doit être une instance de classe implémentant l'interface **View,** le second le modèle, sous la forme d'un dictionnaire de type [java.util.Map], que la vue (premier paramètre) devra afficher.
- lignes 28-35 : on simule un traitement quelconque et on note la durée de celui-ci.
- ligne 37 : on crée le dictionnaire du modèle
- ligne 38 : on met dedans le champ privé [groupe] du contrôleur. On veut montrer que le contrôleur a bien accès aux données du contexte de l'application (applicationContext.xml).
- ligne 39 : on place également dans le modèle, la durée du traitement.
- ligne 41 : on retourne une instance de [ModelAndView], avec pour premier paramètre une instance d'une classe [MyView] que nous allons présenter, et comme second paramètre, le modèle que cette vue devra afficher.

Il ne nous reste plus qu'à présenter la classe [MyView] (dans WEB-INF/src, cf copie d'écran) :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.io.PrintWriter;
4.  import java.util.ArrayList;
5.  import java.util.Map;
6.
7.  import javax.servlet.http.HttpServletRequest;
8.  import javax.servlet.http.HttpServletResponse;
9.
10. import org.springframework.web.servlet.View;
11.
12. public class MyView implements View {
13.
14.   public void render(Map modèle, HttpServletRequest request, HttpServletResponse response) throws
    Exception {
15.     // on code le HTML en dur
16.     response.setContentType("text/html");
17.     PrintWriter out = response.getWriter();
18.     // on prépare le code HTML
19.     String Html="<html><head><title>Spring-MVC-08</title></head>"+
20.             "<body>";
21.     // on parcourt la liste des membres du groupe
22.     Groupe groupe=(Groupe)modèle.get("groupe");
23.     ArrayList membres=groupe.getMembres();
24.     for(int i=0;i<membres.size();i++){
25.       Html+=membres.get(i).toString()+"<br>\n";
26.     }
27.     // on ajoute la durée d'exécution
28.     long durée=((Long)modèle.get("durée")).longValue();
29.     Html += "<br>DoSomething exécuté en " + durée + " ms...</body></html>";
30.     // on envoie le flux HTML
31.     out.println(Html);
32.   }
33.
34. }
35.
```

- ligne 12 : la classe [myView] implémente l'interface [View]

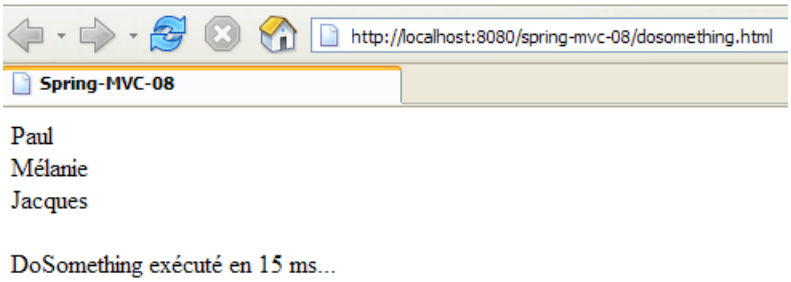
Nous avons vu au paragraphe 8.1, page 37, que l'interface [View] avait une unique méthode :

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

- lignes 14-32 : la méthode [render] de l'interface [View]. Puisqu'on dispose comme paramètre, de l'objet [HttpResponse response], nous sommes capables d'envoyer nous-mêmes la réponse au client. C'est ce qui est fait ici. On notera également que le premier paramètre de la méthode est le modèle à afficher.
- ligne 16 : indique au client qu'il va recevoir un flux HTML
- ligne 17 : on récupère un canal d'écriture
- ligne 19 : début du flux HTML
- ligne 22 : on récupère l'objet "groupe" dans le modèle
- lignes 23-26 : on insère son contenu, c.a.d. les membres du groupe, dans le flux HTML
- lignes 27-29 : on fait de même avec l'objet "durée"
- ligne 31 : on envoie le flux HTML au client

Qu'avons-nous fait ? Nous avons simplement déplacé l'élaboration de la réponse au client, du contrôleur vers la vue. Nous avons continué à construire le flux HTML à la main.

Testons. Nous demandons l'url [http://localhost:8080/spring-mvc-08/dosomething.html] :



8.3 L'implémentation JstlView

Dans l'exemple précédent, nous avons implémenté l'interface [View] avec une classe propriétaire. Dans ce nouvel exemple, l'interface [View] est implémentée par une classe prédéfinie de Spring MVC, la classe [JstlView] (**org.springframework.web.servlet.view.JstlView**) :

org.springframework.web.servlet.view

Class JstlView

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.AbstractUrlBasedView
          └ org.springframework.web.servlet.view.InternalResourceView
            └ **org.springframework.web.servlet.view.JstlView**

All Implemented Interfaces:

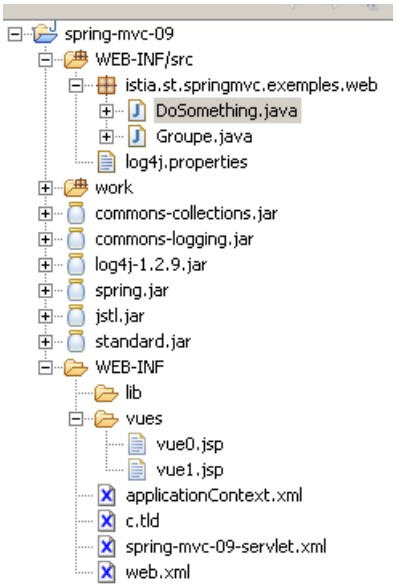
BeanNameAware, InitializingBean, ApplicationContextAware, View

La classe [JstlView] permet d'afficher un modèle [Map] au moyen d'une page JSP utilisant ou non des balises JSTL (Java Standard Tag Library). Cette bibliothèque de balises permet de limiter la présence de code Java dans les pages JSP.

La classe [JstlView] a uniquement un constructeur sans paramètres. Elle sert à afficher une page JSP qu'il faut préciser au moyen d'une URL. Les méthodes *get* et *set* permettant de connaître et de fixer celle-ci sont définies dans la classe [AbstractUrlBasedView], une classe parent de [JstlView] :

String	**getUrl**() Return the URL of the resource that this view wraps.
void	**setUrl**(String url) Set the URL of the resource that this view wraps.

L'exemple Eclipse mettant en oeuvre l'implémentation [JstlView] est le suivant :



L'utilisation de la bibliothèque JSTL amène de nouveaux fichiers dans le projet :

- **c.tld** : décrit la syntaxe des balises JSTL. Elle permet aux programmes exploitant ces balises (parseurs) de vérifier leur bonne syntaxe. Nous avons placé ici ce fichier dans [WEB-INF]. En fait, il peut être en fait un peu n'importe où, sur une machine distante par exemple.
- **standard.jar**, **jstl.jar** : les archives des classes implémentant les balises JSTL. Ces deux archives sont dans le *Classpath* du projet.

Le répertoire [vues] contient les deux pages JSP [vue0.jsp, vue1.jsp] que l'application va afficher.

Ce projet est identique au précédent. Seuls diffèrent le contrôleur [DoSomething] et le mécanisme d'affichage des vues.

Le contrôleur [DoSomething] est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Date;
4.  import java.util.HashMap;
5.
6.  import javax.servlet.http.HttpServletRequest;
7.  import javax.servlet.http.HttpServletResponse;
8.
9.  import org.springframework.web.servlet.ModelAndView;
10. import org.springframework.web.servlet.mvc.Controller;
11. import org.springframework.web.servlet.view.JstlView;
12.
13. public class DoSomething implements Controller {
14.
15.   // un groupe de personnes fourni par le contexte de l'application
16.   private Groupe groupe;
17.
18.   public Groupe getGroupe() {
19.     return groupe;
20.   }
21.
22.   public void setGroupe(Groupe groupe) {
23.     this.groupe = groupe;
24.   }
25.
26.   // gestion de la requête
27.   public ModelAndView handleRequest(HttpServletRequest request,
28.       HttpServletResponse response) throws Exception {
29.     // début
30.     long début = new Date().getTime();
```

```
31.    // on fait qq chose...
32.    Thread.sleep(10);
33.    // fin
34.    long fin = new Date().getTime();
35.    // durée
36.    long durée = fin - début;
37.    // on crée le modèle de la vue à afficher
38.    Map modèle = new HashMap();
39.    modèle.put("groupe", groupe);
40.    modèle.put("duree", new Long(durée));
41.    // on retourne le ModelAndView
42.    int i = (int) (Math.random() * 2);
43.    JstlView vue = new JstlView();
44.    vue.setUrl("/WEB-INF/vues/vue" + i + ".jsp");
45.    return new ModelAndView(vue, modèle);
46.  }
47.
48. }
```

- jusqu'à la ligne 40 incluse, on a le même contrôleur que dans l'exemple précédent
- ligne 42 : un nombre entier i est généré aléatoirement. Il peut avoir deux valeurs 0 et 1.
- ligne 43 : on crée une instance de type [JstlView]
- ligne 44 : on fixe l'URL de la page JSP à afficher au moyen de la méthode [setUrl]. Selon la valeur de i, on affichera la page [WEB-INF/vues/vue0.jsp] ou [WEB-INF/vues/vue1.jsp].
- ligne 45 : on retourne une instance [ModelAndView] où l'interface [View] est représentée par l'instance [JstlView] créée et associée à une page JSP particulière.

Les pages JSP qui doivent afficher le modèle [Map] construit par le contrôleur sont les suivantes :

[**vue0.jsp**]

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-09</title>
8.    </head>
9.    <body>
10.     <h2>Vue n&deg; 0</h2>
11.     Membres du groupe
12.     <table>
13.       <c:forEach var="personne" items="${groupe.membres}">
14.         <tr>
15.           <td>${personne}</td>
16.         </tr>
17.       </c:forEach>
18.     </table>
19.     <br>
20.     DoSomething ex&eacute;cut&eacute; en ${duree} ms...
21.   </body>
22. </html>
```

- les lignes 1-3 ont été expliquées au paragraphe 5, page 18.
- ligne 10 : affiche le nom de la vue sous la forme **Vue n° i**
- lignes 13-17 : affichage de l'élément "groupe" du modèle.
- lige 20 : affichage de l'élément "duree" du modèle

Rappelons comment sont récupérées les valeurs des éléments ${item} d'une page JSP / JSTL.

- ligne 20 : la valeur de l'élément ${duree} est cherchée successivement dans les contextes suivants :
  - **Requête** : request.getAttribute(" duree ")
  - **Session** : session.getAttribute(" duree ")
  - **Application** : application.getAttribute(" duree ")

Considérons par exemple la page JSP suivante :

```
1.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
2.  <%@ page isELIgnored="false" %>
3.
4.  <%
5.    String localContext="contexte page";
6.    request.setAttribute("requestContext","contexte request");
7.    session.setAttribute("sessionContext","contexte session");
8.    application.setAttribute("applicationContext","contexte application");
```

```
9.   session.setAttribute("var1","var1 session");
10.  application.setAttribute("var1","var1 application");
11. %>
12. <html>
13.  <head>
14.    <title>Contextes d'une page JSP</title>
15.  </head>
16.  <body>
17.    <h3>Contextes d'une page JSP</h3>
18.    page : <%=localContext%><br>
19.    request : ${requestContext}<br>
20.    session : ${sessionContext}<br>
21.    application : ${applicationContext}<br>
22.    var1 : ${var1}<br>
23.    var1 : <%=application.getAttribute("var1")%><br>
24.  </body>
25.</html>
```

- ligne 5 : définit une variable locale à la page
- ligne 6 : met une donnée de clé " requestContext " dans le contexte de la requête. [request] est une variable prédéfinie dans une page JSP et désigne la requête en cours de traitement.
- ligne 7 : met une donnée de clé "sessionContext " dans le contexte de la session. [session] est une variable prédéfinie dans une page JSP et désigne la session à laquelle appartient la requête en cours de traitement.
- ligne 8 : met une donnée de clé " applicationContext " dans le contexte de l'application. [application] est une variable prédéfinie dans une page JSP et désigne l'application à laquelle appartient la requête en cours de traitement.
- lignes 9-10 : met une donnée de clé " var1 " à la fois dans le contexte " session " et le contexte " application "
- ligne 18 : affiche la variable [localContext] de portée " page "
- ligne 19 : affiche la variable de clé "requestContext"
- ligne 20 : affiche la variable de clé "sessionContext"
- ligne 21 : affiche la variable de clé "applicationContext"
- ligne 22 : affiche la variable de clé " var1 " de portée " session "
- ligne 23 : affiche la variable de clé " var1 " de portée "application"

Cette page affichée donne le résultat suivant :

Expliquons par exemple le processus de traitement de la ligne 21 de la page JSP :

```
    application : ${applicationContext}<br>
```

La servlet issue de la page JSP recherche une donnée de clé " applicationContext " successivement dans les contextes " request ", " session ", " application ". Elle la trouve dans ce dernier contexte.

Pour la ligne 22 de la page JSP, la même processus se déroule. La clé "var1 " se trouve à la fois dans les contextes " session " et " application ". Elle sera trouvée d'abord dans le contexte " session ". Pour obtenir la clé " var1 " de contexte " application " qui est cachée par la clé " var1 " de contexte "session", on peut utiliser directement l'objet prédéfini [application] comme le montre la ligne 23.

Revenons sur la page JSP / JSTL [vue0.jsp] et considérons maintenant la séquence suivante :

```
1.       <c:forEach var="personne" items="${groupe.membres}">
2.         <tr>
3.           <td>${personne}</td>
4.         </tr>
5.       </c:forEach>
```

- ligne 1 : la donnée ${groupe.membres} est évaluée de la façon suivante :
  - la clé " groupe " est cherchée dans les différents contextes comme il a été expliqué précédemment

- une fois l'objet " groupe " trouvé, l'objet [groupe.membres] est obtenu par [groupe.getMembres()] ou [groupe.get(" membres ")]. Le premier cas réussira si l'objet " groupe " est un Javabean avec le *get* approprié pour son champ " membres ". Le second cas réussira si l'objet " groupe " est un dictionnaire avec une clé " membres ".

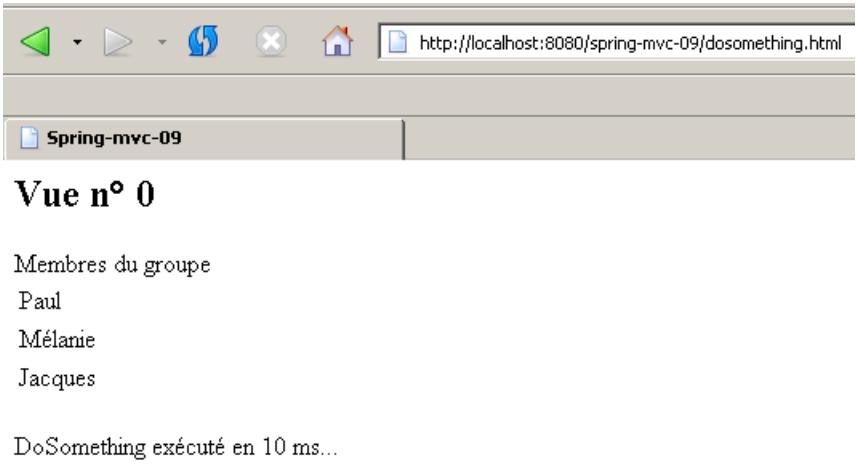
Il nous reste un dernier point à expliquer. Comment les éléments du modèle [Map] construit par le contrôleur [DoSomething] sont-ils arrivés dans un des contextes [" request ", " session ", " application "] ? C'est le contrôleur [DispatcherServlet] qui se charge de mettre les éléments du dictionnaire [Map] qu'il reçoit d'un contrôleur [Controller] dans le contexte " request ". La page JSP retrouve donc dans ce contexte tous les éléments que le contrôleur [Controller] a placés dans le modèle [Map].

La page [**vue1.jsp**] est analogue à la vue [**vue0.jsp**] :

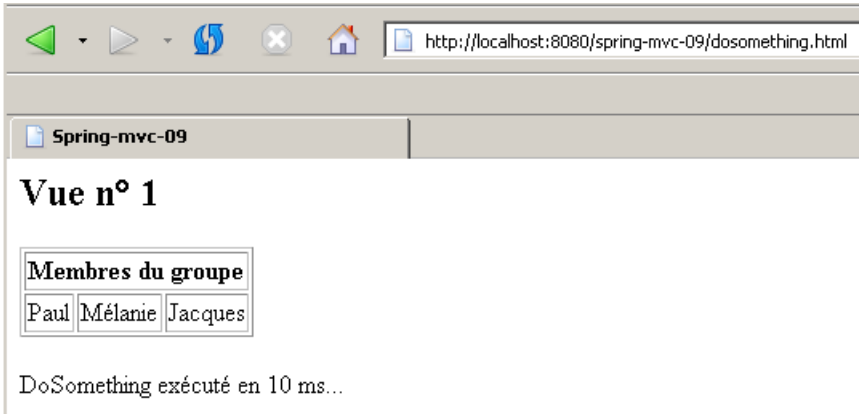
```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-09</title>
8.    </head>
9.    <body>
10.     <h2>Vue n&deg; 1</h2>
11.     <table border="1">
12.       <tr>
13.         <th colspan="3" align="center">Membres du groupe</th>
14.       </tr>
15.       <tr>
16.         <c:forEach var="personne" items="${groupe.membres}">
17.           <td>${personne}</td>
18.         </c:forEach>
19.       </tr>
20.     </table>
21.     <br>
22.     DoSomething ex&eacute;cut&eacute; en ${duree} ms...
23.   </body>
24. </html>
```

- ligne 10 : affichera Vue n° 1
- ligne 11 : les membres du groupe seront mis en ligne dans un tableau alors que dans la vue n° 0, ils sont affichés à raison d'un par ligne

Nous sommes prêts pour les tests. Nous demandons l'url [http://localhost:8080/spring-mvc-09/dosomething.html]. Premier essai :

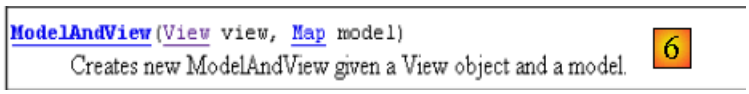


En rechargeant plusieurs fois la page (Page reload), on obtient aléatoirement les vues 0 ou 1. La vue n° 1 est la suivante :



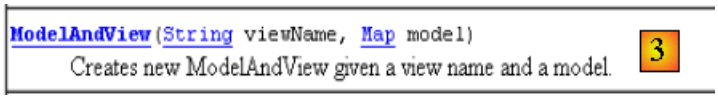
8.4 Le résolveur de noms de vue [InternalResourceViewResolver]

Dans l'exemple précédent, le contrôleur [DoSomething] rendait au contrôleur principal [DispatcherServlet] un objet [ModelAndView] construit avec le constructeur suivant :



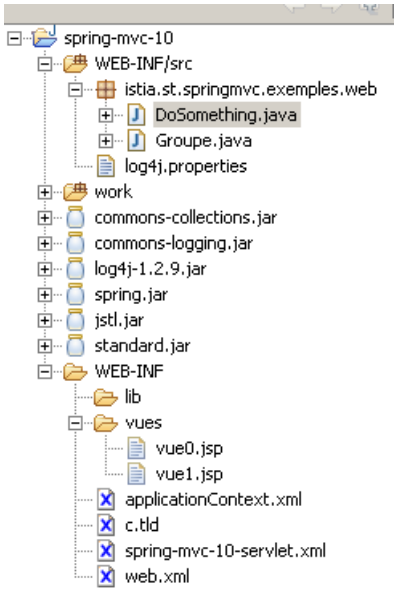
Ce constructeur exige du contrôleur [Controller] qu'il connaisse l'implémentation [View] à utiliser pour afficher le modèle [Map] qu'il a construit. Cela manque de souplesse. S'il est bien normal que le contrôleur [Controller] construise le modèle à afficher pour l'utilisateur, on ne voit pas pourquoi il déciderait du mode de rendu de celui-ci. Si on est amené à changer le mode de rendu (Pdf au lieu de Html), on est obligé de modifier le code. On souhaiterait davantage de souplesse.

On peut alors utiliser le constructeur suivant :



Cette fois-ci, le contrôleur [Controller] ne précise que **le nom de la vue** à utiliser. Par configuration, ce nom sera associé à une classe implémentant l'interface [View]. L'avantage est bien sûr la souplesse apportée par ce mécanisme. Passer d'un rendu Html à un rendu Pdf ne nécessitera qu'un changement dans le fichier de configuration, pas dans le code Java.

Nous illustrons ce mécanisme avec le projet Eclipse suivant :



De nouveau, l'essence des exemples précédents est conservée. Le premier changement intervient dans le contrôleur [DoSomething] :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Date;
4.  import java.util.HashMap;
5.
6.  import javax.servlet.http.HttpServletRequest;
7.  import javax.servlet.http.HttpServletResponse;
8.
9.  import org.springframework.web.servlet.ModelAndView;
10. import org.springframework.web.servlet.mvc.Controller;
11.
12. public class DoSomething implements Controller {
13.
14.  // un groupe de personnes fourni par le contexte de l'application
15.  private Groupe groupe;
16.
17.  public Groupe getGroupe() {
18.    return groupe;
19.  }
20.
21.  public void setGroupe(Groupe groupe) {
22.    this.groupe = groupe;
23.  }
24.
25.  // gestion de la requête
26.  public ModelAndView handleRequest(HttpServletRequest request,
27.      HttpServletResponse response) throws Exception {
28.    // début
29.    long début = new Date().getTime();
30.    // on fait qq chose...
31.    Thread.sleep(10);
32.    // fin
33.    long fin = new Date().getTime();
34.    // durée
35.    long durée = fin - début;
36.    // on crée le modèle de la vue à afficher
37.    HashMap modèle = new HashMap();
38.    modèle.put("groupe", groupe);
39.    modèle.put("durée", new Long(durée));
40.    // on retourne le ModelAndView
41.    int i = (int) (Math.random() * 2);
42.    return new ModelAndView("vue"+i, modèle);
43.  }
44.
45. }
```

- ligne 42 : au lieu de désigner la vue par une référence à un type [View], la vue est désormais désignée par son nom : **vue0** ou **vue1** selon la valeur de i.

C'est le fichier de configuration [spring-mvc-10-servlet.xml] de la servlet [spring-mvc-10] qui va faire le lien entre ce nom et une instance de type [View] :

```
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="/dosomething.html">DoSomethingController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="DoSomethingController"
14.     class="istia.st.springmvc.exemples.web.DoSomething">
15.     <property name="groupe">
16.       <ref bean="groupe"/>
17.     </property>
18.   </bean>
19.   <!-- le résolveur de vues -->
20.   <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
21.     <property name="viewClass">
22.       <value>org.springframework.web.servlet.view.JstlView</value>
23.     </property>
24.     <property name="prefix">
25.       <value>/WEB-INF/vues/</value>
26.     </property>
27.     <property name="suffix">
28.       <value>.jsp</value>
29.     </property>
30.   </bean>
31. </beans>
```

- ligne 20 : on définit le résolveur des noms de vues. Ce nom barbare désigne la classe chargée de faire le lien entre le nom d'une vue et la classe chargée de son rendu. Un résolveur de noms de vues implémente l'interface [ViewResolver] :

org.springframework.web.servlet
Interface ViewResolver

All Known Implementing Classes:
AbstractCachingViewResolver, AbstractTemplateViewResolver, BeanNameViewResolver, FreeMarkerViewResolver, InternalResourceViewResolver, JasperReportsViewResolver, ResourceBundleViewResolver, UrlBasedViewResolver, VelocityLayoutViewResolver, VelocityViewResolver, XmlViewResolver

On voit qu'il y a de nombreuses implémentations disponibles, dont [InternalResourceViewResolver] que nous utilisons dans [spring-mvc-10-servlet.xml]. L'interface [ViewResolver] n'a qu'une méthode :

Method Summary

View	**resolveViewName**(String viewName, Locale locale) Resolve the given view by name.

La méthode [resolveViewName] a pour but d'associer un nom de vue (viewName) à une classe implémentant l'interface [View] (résultat de la méthode). Le second paramètre de la méthode est un objet [Locale] qui identifie la langue et le pays qui seront utilisés par la vue. Par exemple le français de France métropolitaine, le français du Québec, l'anglais de Grande-Bretagne, l'anglais des USA, ...). C'est généralement, le navigateur client qui précise dans quelle [Locale] il veut travailler via un entête HTTP particulier. Nous aurons l'occasion de revenir sur ce point lorsque nous parlerons de l'internationalisation des vues.

Le résolveur des noms de vues [InternalResourceViewResolver] utilisé dans [spring-mvc-10-servlet.xml] permet de préciser :
- la classe à utiliser pour afficher le modèle rendu par le contrôleur [Controller] : **setViewClass**
- le préfixe avec lequel faire précéder le nom de vue rendu par le contrôleur [Controller] pour construire l'url complète de la page JSP : **setPrefix**
- le suffixe à ajouter au nom de vue rendu par le contrôleur [Controller] pour construire l'url complète de la page JSP : **setSuffix**

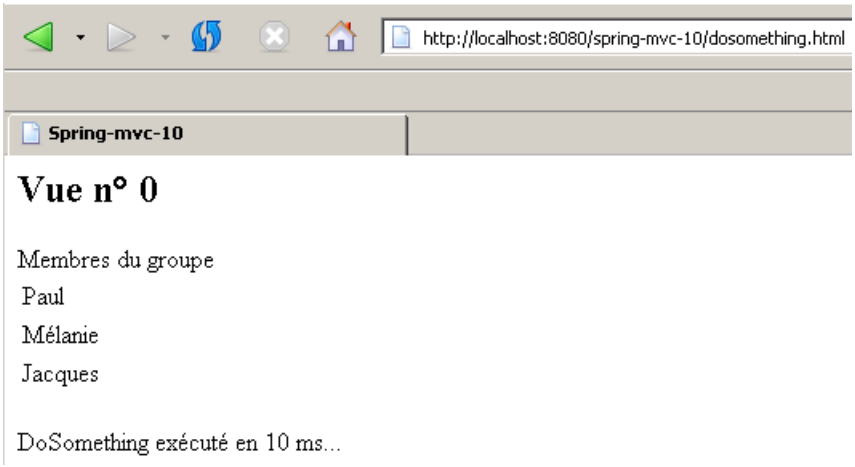
Les lignes suivantes du fichier [spring-mvc-10-servlet.xml] définissent le mode de résolution des noms de vues :

```
1.  <!-- le résolveur de vues -->
2.  <bean class="org.springframework.web.servlet.view.InternalResourceViewResolver">
3.    <property name="viewClass">
4.      <value>org.springframework.web.servlet.view.JstlView</value>
5.    </property>
6.    <property name="prefix">
7.      <value>/WEB-INF/vues/</value>
8.    </property>
9.    <property name="suffix">
10.     <value>.jsp</value>
11.   </property>
12.</bean>
```

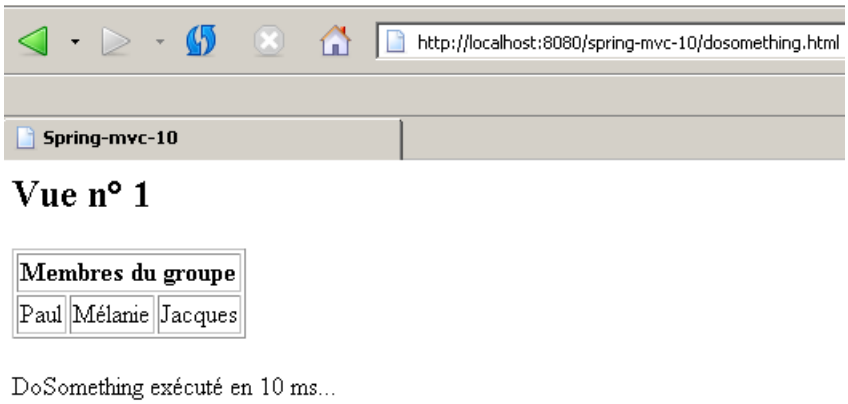
- lignes 3-5 : le modèle rendu par le contrôleur [Controller] sera affiché par une instance de la classe [JstlView] que nous avons étudiée dans l'exemple précédent
- lignes 6-11 : indique que si le contrôleur [Controller] a précisé comme nom de vue, le nom **N**, la classe [JstlView] devra afficher la page JSP [/WEB-INF/vues/**N**.jsp], c.a.d. [préfixe/**N**.suffixe].

Les vues [vue0.jsp, vue1.jsp] sont restées identiques à ce qu'elles étaient dans l'exemple précédent, à l'attribut HTML <title> près.

Nous sommes prêts pour un test. Nous demandons l'url [http://localhost:8080/spring-mvc-10/dosomething.html] :



En rechargeant plusieurs fois la page (Page reload), on obtient aléatoirement les vues 0 ou 1. La vue n° 1 est la suivante :



8.5 Le résolveur de noms de vue [BeanNameViewResolver]

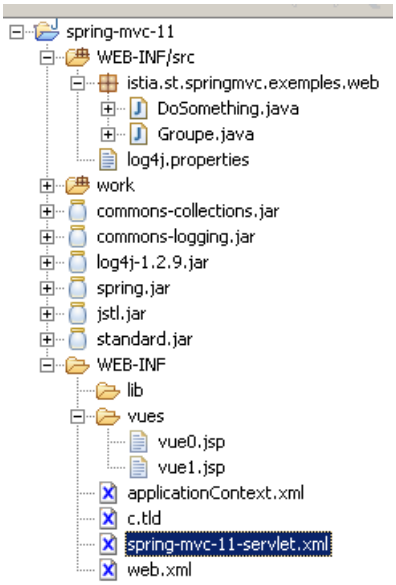
Revenons sur l'interface [ViewResolver] de résolution des noms de vue et ses implémentations :

org.springframework.web.servlet
Interface ViewResolver

All Known Implementing Classes:
AbstractCachingViewResolver, AbstractTemplateViewResolver, BeanNameViewResolver, FreeMarkerViewResolver, InternalResourceViewResolver, JasperReportsViewResolver, ResourceBundleViewResolver, UrlBasedViewResolver, VelocityLayoutViewResolver, VelocityViewResolver, XmlViewResolver

Ci-dessus, on voit qu'une classe d'implémentation est [BeanNameViewResolver]. Le projet suivant illustre l'utilisation de ce résolveur de noms de vue.

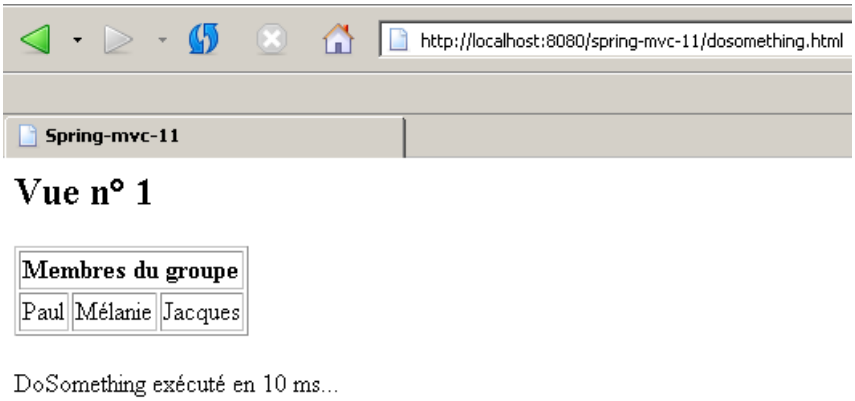
Le projet est identique au précédent. Seul change le résolveur des noms de vue utilisé dans [spring-mvc-11-servlet.xml] :




```
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="/dosomething.html">DoSomethingController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="DoSomethingController"
14.     class="istia.st.springmvc.exemples.web.DoSomething">
15.     <property name="groupe">
16.       <ref bean="groupe"/>
17.     </property>
18.   </bean>
19.   <!-- le résolveur de vues -->
20.   <bean class="org.springframework.web.servlet.view.BeanNameViewResolver"/>
21.   <!-- les vues -->
22.   <bean id="vue0" class="org.springframework.web.servlet.view.JstlView">
23.     <property name="url">
24.       <value>/WEB-INF/vues/vue0.jsp</value>
25.     </property>
26.   </bean>
27.   <bean id="vue1" class="org.springframework.web.servlet.view.JstlView">
28.     <property name="url">
29.       <value>/WEB-INF/vues/vue1.jsp</value>
30.     </property>
31.   </bean>
32. </beans>
```

- le résolveur des noms de vues est défini lignes 20-31. Le résolveur [BeanNameViewResolver] associe un bean à chacun des noms de vue. L'id du bean est le nom de la vue, d'où le nom du résolveur. Le bean associé au nom d'une vue définit :
  - la classe chargée du rendu de la vue
  - les paramètres dont cette classe pourrait avoir besoin
- nous savons que le contrôleur [DoSomething] rend des instances [ModelAndView] référençant les vues portant les noms : **vue0** et **vue1**.
- lignes 22-26 : le bean " vue0 " définit la vue de nom " vue0 ". La classe chargée de rendre cette vue est une instance de [JstlView] (attribut **class**). L'url de la page JSP que doit afficher l'instance [JstlView] est [/WEB-INF/vues/vue0.jsp] (attribut **url**). On a ici toutes les informations nécessaires à l'affichage de la vue.
- lignes 27-31 : de façon analogue, le bean " vue1 " définit la vue de nom " vue1 ".

Nous pouvons tester. Nous demandons l'url [http://localhost:8080/spring-mvc-11/dosomething.html] :



En rechargeant la page, on obtiendra de façon aléatoire les vues 0 et 1.

8.6 Le résolveur de noms de vue [XmlViewResolver]

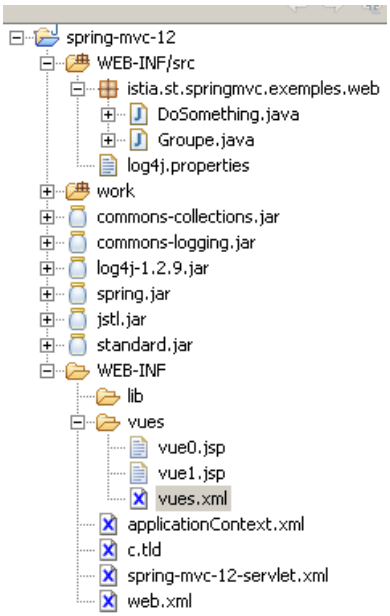
Revenons sur l'interface [ViewResolver] :

org.springframework.web.servlet
Interface ViewResolver

All Known Implementing Classes:
AbstractCachingViewResolver, AbstractTemplateViewResolver, BeanNameViewResolver, FreeMarkerViewResolver, InternalResourceViewResolver, JasperReportsViewResolver, ResourceBundleViewResolver, UrlBasedViewResolver, VelocityLayoutViewResolver, VelocityViewResolver, XmlViewResolver

L'une des classes d'implémentation de l'interface [ViewResolver] est, ci-dessus, la classe [XmlViewResolver]. C'est une variante de l'implémentation [BeanNameViewResolver] que nous venons de voir. Une vue de nom N est configurée par un bean d'id N. Avec [BeanNameViewResolver], ces beans étaient cherchés dans le fichier de configuration [S-servlet.xml] de la servlet S de l'application. Avec [XmlViewResolver], ces beans sont définis dans un fichier XML externe. On trouve dans [S-servlet.xml] le nom de ce fichier.

Pour illustrer cette stratégie, nous utilisons le projet Eclipse suivant :



Ce projet ne diffère du précédent que par ses fichiers [spring-mvc-12-servlet.xml] et [vues.xml].

Dans le fichier [spring-mvc-12-servlet.xml], le résolveur de noms de vue change :

```
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="/dosomething.html">DoSomethingController</prop>
9.        </props>
10.     </property>
11.   </bean>
12.   <!-- les contrôleurs de l'application-->
13.   <bean id="DoSomethingController"
14.     class="istia.st.springmvc.exemples.web.DoSomething">
15.     <property name="groupe">
16.       <ref bean="groupe"/>
17.     </property>
18.   </bean>
19.   <!-- le résolveur de vues -->
20.   <bean class="org.springframework.web.servlet.view.XmlViewResolver">
21.     <property name="location">
22.       <value>/WEB-INF/vues/vues.xml</value>
23.     </property>
24.   </bean>
25. </beans>
```

- lignes 20-24 : définissent le nouveau résolveur de noms de vues
- ligne 20 : la classe d'implémentation [XmlViewResolver] de l'interface [ViewResolver]. Cette classe a un constructeur sans paramètres et une méthode [setLocation] qui permet d'indiquer où se trouve le fichier de définition des beans des vues :

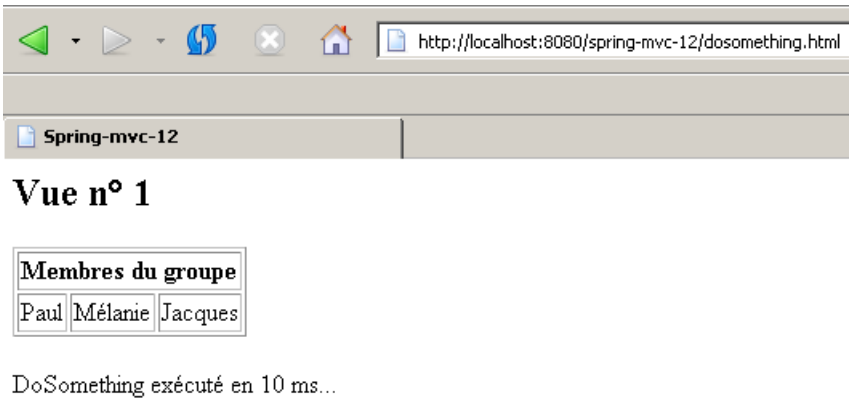
void	**setLocation**(Resource location) Set the location of the XML file that defines the view beans.

- lignes 21-23 : définissent l'emplacement du fichier XML de définition des vues, ici [/WEB-INF/vues/vues.xml]. On peut ne pas définir d'emplacement. Dans ce cas, c'est le fichier [/WEB-INF/vues/**views.xml**] qui est utilisé par défaut.

Dans le fichier [/WEB-INF/vues/vues.xml], on retrouve les beans déjà rencontrés avec la stratégie [BeanNameViewResolver] :

```
1.  <?xml version="1.0" encoding="ISO_8859-1"?>
2.  <!DOCTYPE beans SYSTEM "http://www.springframework.org/dtd/spring-beans.dtd">
3.  <beans>
4.    <bean id="vue0" class="org.springframework.web.servlet.view.JstlView">
5.      <property name="url">
6.        <value>/WEB-INF/vues/vue0.jsp</value>
7.      </property>
8.    </bean>
9.    <bean id="vue1" class="org.springframework.web.servlet.view.JstlView">
10.     <property name="url">
11.       <value>/WEB-INF/vues/vue1.jsp</value>
12.     </property>
13.   </bean>
14. </beans>
```

Nous sommes prêts pour les tests. Nous demandons l'url [http://localhost:8080/spring-mvc-12/dosomething.html] :



En rechargeant la page, on obtiendra de façon aléatoire les vues 0 et 1.

8.7 Le résolveur de noms de vue [ResourceBundleViewResolver]

Revenons sur l'interface [ViewResolver] :

org.springframework.web.servlet

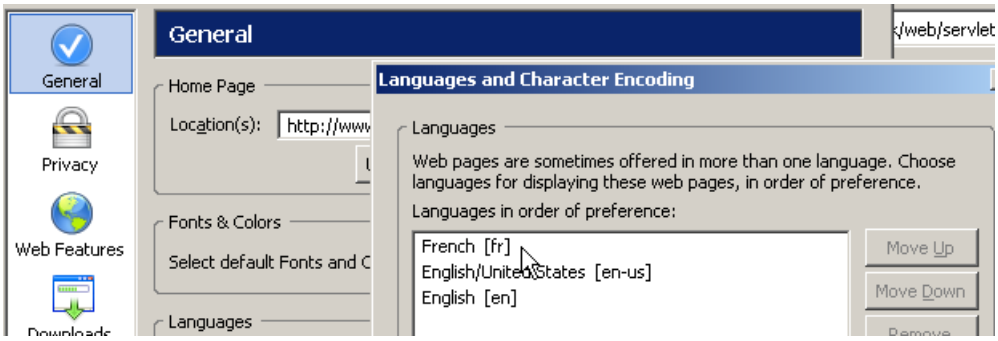
Interface ViewResolver

All Known Implementing Classes:

AbstractCachingViewResolver, AbstractTemplateViewResolver, BeanNameViewResolver, FreeMarkerViewResolver, InternalResourceViewResolver, JasperReportsViewResolver, ResourceBundleViewResolver, UrlBasedViewResolver, VelocityLayoutViewResolver, VelocityViewResolver, XmlViewResolver

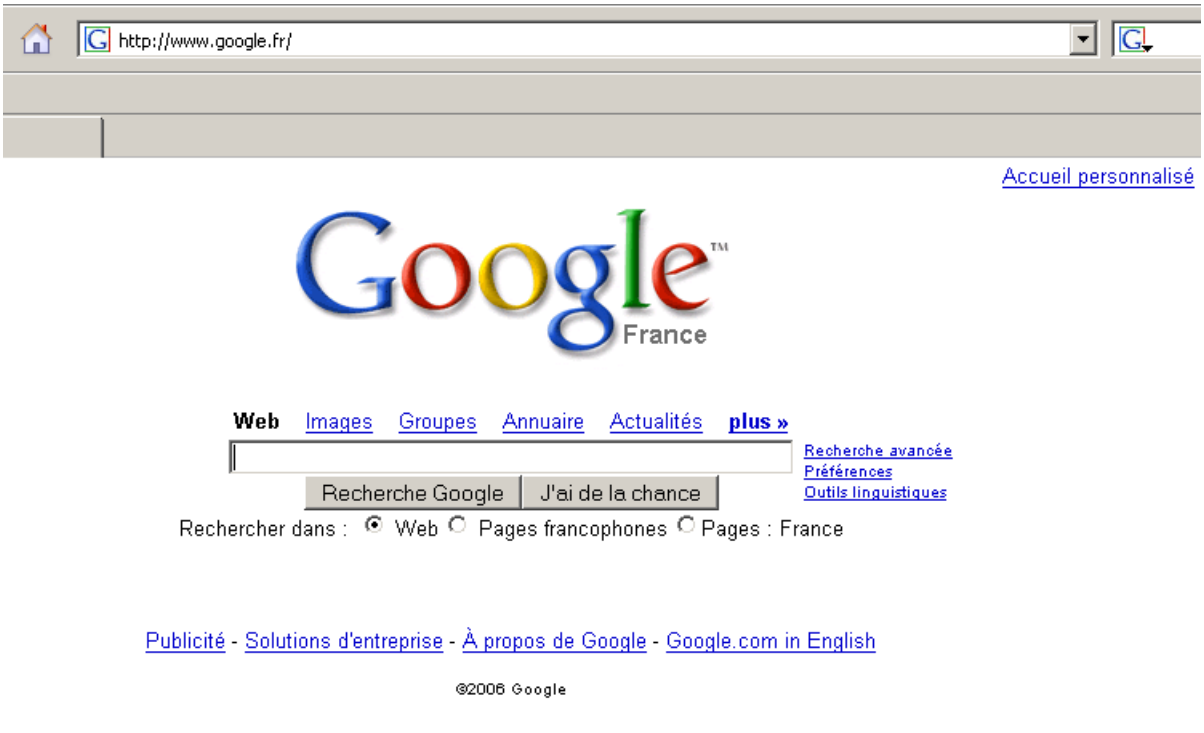
Ci-dessus, l'une des classes d'implémentation de l'interface [ViewResolver] est [ResourceBundleViewResolver]. Comme dans la stratégie [XmlViewResolver] la définition des beans des vues est déportée dans un fichier externe, voire des fichiers externes. L'apport de [ResourceBundleViewResolver] vis à vis de [XmlViewResolver] est ce qu'on appelle, l'internationalisation des pages. Les vues vont pouvoir être proposées en différentes langues selon, en général, la configuration du navigateur client.

Montrons sur un exemple simple comment fonctionne ce mécanisme. La configuration de mon navigateur FireFox est la suivante : Tools/ Options / Languages :

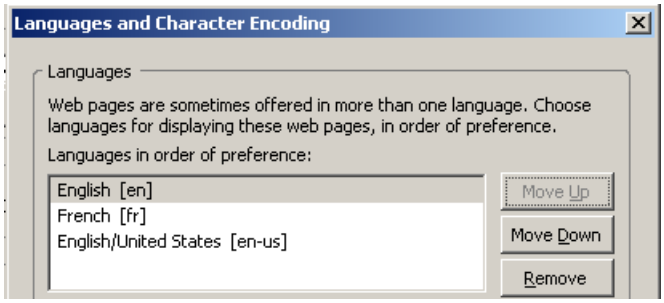


Il est configuré pour trois langues : le français (fr), l'anglais américain (en-us), l'anglais. Lorsque le navigateur fait une demande à un site web, ces informations sont envoyées dans un entête HTTP. Le serveur essaie de satisfaire le navigateur dans l'ordre des langues, ici d'abord (fr), puis (en-us), puis (en).

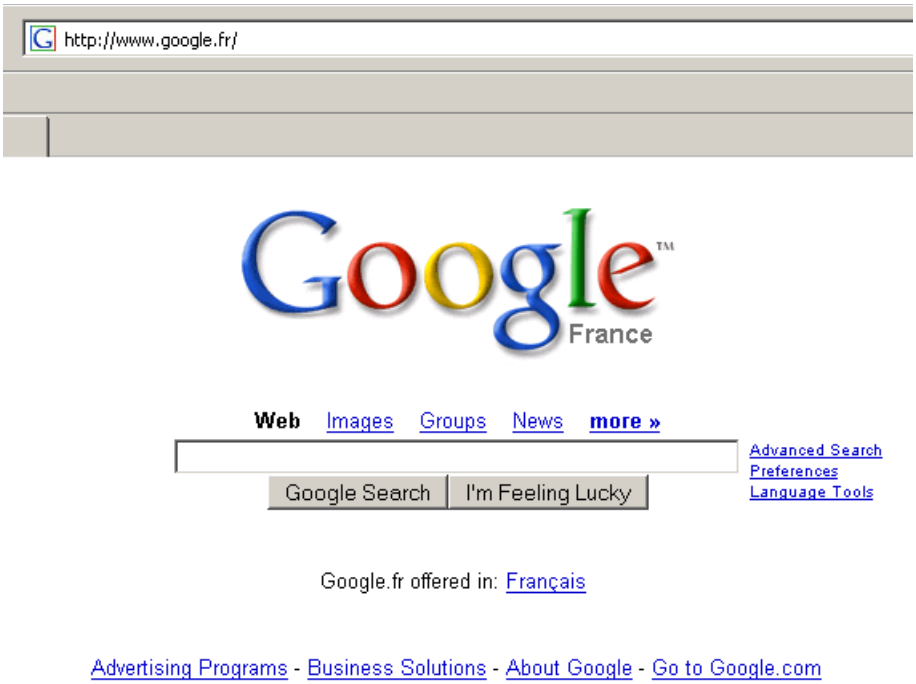
Faisons un test. Avec cette configuration, demandons l'url [http://www.google.fr/]



Maintenant changeons la configuration du navigateur comme suit :

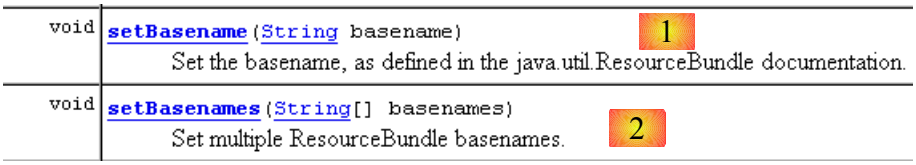


La langue préférée est donc devenue l'anglais. Demandons la même url [http://www.google.fr/].



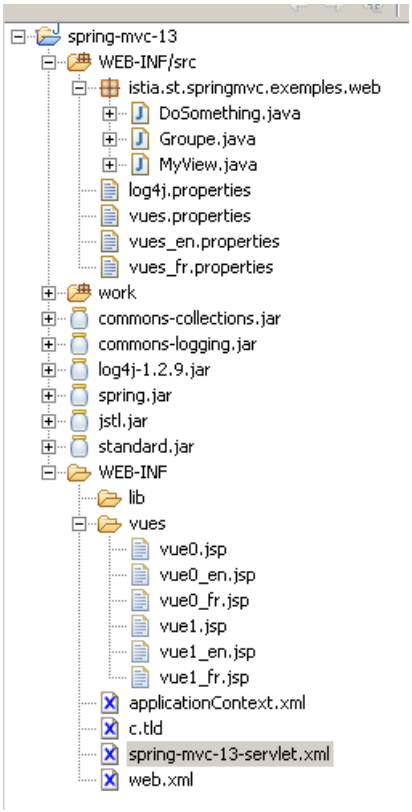
La page nous a été envoyée en anglais.

La classe [ResourceBundleViewResolver] va nous permettre d'obtenir ce fonctionnement. Cette classe a un constructeur sans paramètres et deux méthodes qui permettent de préciser l'emplacement du fichier (ou des fichiers) de définition des beans de vues :



- la méthode [1] permet de définir le nom de base du fichier de définition des vues. Si ce nom est **N**, un fichier **N_locale.properties** sera recherché dans le *classpath* de l'application web. Le suffixe **locale** désigne la langue préférée du navigateur client. Ainsi si celle-ci est l'allemend d'Autriche (de_AT), l'application cherchera un fichier de définition des vues appelé [**N_de_AT.properties**], puis le fichier [**N_de.properties**] et enfin le ficheir [**N.properties**] dans le *classpath* de l'application. Ainsi si on a rédigé des pages en français (fr) et en anglais (en), on aura
  - un fichier **N_fr.properties** pour définir les vues en français
  - un fichier **N_en.properties** pour définir les vues en anglais
  - un fichier **N.properties** pour définir les vues qui seront envoyées aux navigateurs demandant une autre langue, par exemple l'allemand (de).
- la méthode [2] permet de définir plusieurs noms de base plutôt qu'un seul. Ainsi si le paramètre de cette méthode est le tableau {" N1 ", "N2 "}, les fichiers **N1_locale.properties** et **N2_locale.properties** seront recherchés dans le *classpath* de l'application. Cela permet de répartir les définitions de vues dans plusieurs fichiers et de faciliter ainsi le travail séparé d'équipes de développement.

Pour illustrer la stratégie [ResourceBundleViewResolver], nous utiliserons le projet Eclipse suivant :

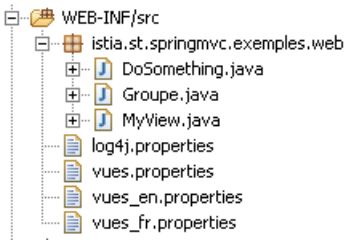


Le fichier de configuration de la servlet [spring-mvc-13-servlet.xml] est le suivant :

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

- les lignes 20-25 définissent le nouveau résolveur de noms des vues de type [ResourceBundleViewResolver]. Le nom du fichier des beans des vues est **vues** (ligne 22). Il y a une valeur par défaut pour ce fichier : **views**. Si donc rien n'est précisé, ce sont des fichiers **views_locale.properties** qui serviront à définir les vues. Ici, nous n'avons pas utilisé la valeur par défaut et ce seront des fichiers **vues_locale.properties** qui serviront à définir les vues. Ils seront recherchés dans le *classpath* de l'application.

Nous avons placé les fichiers **vues_locale.properties** dans [WEB-INF/src]. Nous savons qu'Eclipse les recopiera automatiquement dans [WEB-INF/classes] qui fait partie du *classpath :*



- **vues_fr.properties** : définit les vues en français
- **vues_en.properties** : définit les vues en anglais
- **vues.properties** : définit les vues pour les autres langues

Examinons le contenu de [**vues.properties**] :

```
1.  #vue0
2.  vue0.class=org.springframework.web.servlet.view.JstlView
3.  vue0.url=/WEB-INF/vues/vue0.jsp
4.  #vue1
5.  vue1.class=org.springframework.web.servlet.view.JstlView
6.  vue1.url=/WEB-INF/vues/vue1.jsp
7.  #vue2
8.  vue2.class=istia.st.springmvc.exemples.web.MyView
```

- le fichier est un fichier de propriétés obéissant à la syntaxe de ce type de fichiers donc contenant des lignes de type **attribut=valeur**. Ces attributs doivent être des propriétés de la classe. Ainsi utiliser, ligne 3, l'attribut **url** implique que la classe JstlView ait une méthode [setUrl]. Le lecteur pourra vérifier que c'est bien le cas. Les lignes de [**vues.properties**] donnent les mêmes informations que les beans de la stratégie [XmlViewResolver] de l'exemple précédent. Ainsi, revenons sur la définition de la vue " vue0 " dans la version précédente :

```
1.    <bean id="vue0" class="org.springframework.web.servlet.view.JstlView">
2.      <property name="url">
3.        <value>/WEB-INF/vues/vue0.jsp</value>
4.      </property>
5.    </bean>
```

- les informations ci-dessus sont reprises lignes 1-3 du fichier [**vues.properties**]
- lignes 4-6 : définissent la vue nommée " vue1 "
- lignes 7-8 : définissent la vue nommée " vue2 ". Pour montrer qu'on n'est pas obligés d'utiliser la même classe [View] pour toutes les vues, nous proposons d'afficher la vue " vue2 " à l'aide de la classe [MyView] décrite au paragraphe 8.2, page 40. Cette classe ne demande aucun paramètre pour s'initialiser d'où l'absence d'attributs supplémentaires.

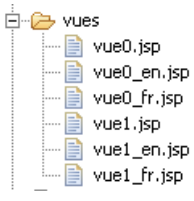
Le fichier [**vues_fr.properties**] est analogue :

```
1.  #vue0
2.  vue0.class=org.springframework.web.servlet.view.JstlView
3.  vue0.url=/WEB-INF/vues/vue0_fr.jsp
4.  #vue1
5.  vue1.class=org.springframework.web.servlet.view.JstlView
6.  vue1.url=/WEB-INF/vues/vue1_fr.jsp
7.  #vue2
8.  vue2.class=istia.st.springmvc.exemples.web.MyView
```

ainsi que le fichier [**vues_en.properties**] :

```
1.  #vue0
2.  vue0.class=org.springframework.web.servlet.view.JstlView
3.  vue0.url=/WEB-INF/vues/vue0_en.jsp
4.  #vue1
5.  vue1.class=org.springframework.web.servlet.view.JstlView
6.  vue1.url=/WEB-INF/vues/vue1_en.jsp
7.  #vue2
8.  vue2.class=istia.st.springmvc.exemples.web.MyView
```

Les pages JSP affichées par les instances [JstlView] sont les suivantes :



- les pages suffixées par **fr** sont celles affichées par [**vues_fr.properties**]
- les pages suffixées par **en** sont celles affichées par [**vues_en.properties**]
- les pages sans suffixe sont celles affichées par [**vues.properties**]

La page [**vue0_fr.jsp**] est une page en français :

```
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-13</title>
8.    </head>
9.    <body>
10.     <h2>Vue n&deg; 0 (fr)</h2>
11.     Membres du groupe
12.     <table>
13.       <c:forEach var="personne" items="${groupe.membres}">
14.         <tr>
15.           <td>${personne}</td>
16.         </tr>
17.       </c:forEach>
18.     </table>
19.     <br>
20.     DoSomething ex&eacute;cut&eacute; en ${duree} ms...
21.   </body>
22. </html>
```

La page [**vue0_en.jsp**] est analogue mais en anglais :

```
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-13</title>
8.    </head>
9.    <body>
10.     <h2>View n&deg; 0 (en)</h2>
11.     Group members
12.     <table>
13.       <c:forEach var="personne" items="${groupe.membres}">
14.         <tr>
15.           <td>${personne}</td>
16.         </tr>
17.       </c:forEach>
18.     </table>
19.     <br>
20.     DoSomething executed in ${duree} ms...
21.   </body>
22. </html>
```

La page par défaut [**vue0.jsp**] envoyée lorsque la langue du navigateur n'est ni le français (fr), ni l'anglais (en) est la suivante :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%@ taglib uri="/WEB-INF/c.tld" prefix="c" %>
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-13</title>
8.    </head>
9.    <body>
10.     <h2>Vue n&deg; 0 (default)</h2>
11.     Membres du groupe
12.     <table>
13.       <c:forEach var="personne" items="${groupe.membres}">
14.         <tr>
15.           <td>${personne}</td>
16.         </tr>
17.       </c:forEach>
18.     </table>
19.     <br>
20.     DoSomething ex&eacute;cut&eacute; en ${duree} ms...
21.   </body>
22. </html>
```

Afin qu'aux tests, on s'y retrouve, la ligne 10 indique (default) pour indiquer que c'est la vue par défaut qui a été envoyée.

La même démarche a été suivie pour la vue appelée " vue1 ".

Il ne nous reste plus qu'à éclaircir un mystère. Jusqu'à maintenant, le contrôleur [DoSomething] demandait l'affichage de deux vues appelées " vue0 " et " vue1 ". On a vu apparaître dans les vues ci-dessus une vue appelée " vue2 " qui est rendue par la classe [MyView]. Pour que cette vue soit affichée, le contrôleur [DoSomething] a été légèrement modifié :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Date;
4.  import java.util.HashMap;
5.
6.  import javax.servlet.http.HttpServletRequest;
7.  import javax.servlet.http.HttpServletResponse;
8.
9.  import org.springframework.web.servlet.ModelAndView;
10. import org.springframework.web.servlet.mvc.Controller;
11.
12. public class DoSomething implements Controller {
13.
14.  // un groupe de personnes fourni par le contexte de l'application
15.  private Groupe groupe;
16.
17.  public Groupe getGroupe() {
18.    return groupe;
19.  }
20.
21.  public void setGroupe(Groupe groupe) {
22.    this.groupe = groupe;
23.  }
24.
25.  // gestion de la requête
26.  public ModelAndView handleRequest(HttpServletRequest request,
27.      HttpServletResponse response) throws Exception {
28.    // début
29.    long début = new Date().getTime();
30.    // on fait qq chose...
31.    Thread.sleep(10);
32.    // fin
33.    long fin = new Date().getTime();
34.    // durée
35.    long durée = fin - début;
36.    // on crée le modèle de la vue à afficher
37.    HashMap modèle = new HashMap();
38.    modèle.put("groupe", groupe);
39.    modèle.put("duree", new Long(durée));
40.    // on retourne le ModelAndView
41.    int i = (int) (Math.random() * 3);
42.    return new ModelAndView("vue"+i, modèle);
43.  }
44.
45. }
```

- ligne 41 : le nombre entier aléatoire i prend maintenant ses valeurs dans [0,1,2] au lieu de [0,1] précédemment. D'où la naissance de la vue appelée " vue2 ".

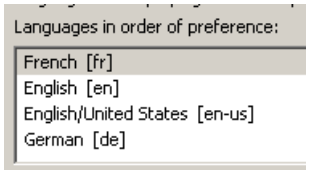
Dans les trois fichiers [vues_locale.properties], la vue nommée " vue2 " est générée par la classe [MyView]. Nous avons déjà expliqué le fonctionnement de cette classe, page 40. Nous ne revenons pas dessus. Elle génère le même flux HTML quelquesoit la langue du navigateur :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.io.PrintWriter;
4.  import java.util.ArrayList;
5.  import java.util.Map;
6.
7.  import javax.servlet.http.HttpServletRequest;
8.  import javax.servlet.http.HttpServletResponse;
9.
10. import org.springframework.web.servlet.View;
11.
12. public class MyView implements View {
13.
14.  public void render(Map modèle, HttpServletRequest request, HttpServletResponse response) throws
    Exception {
15.    // on code le HTML en dur
16.    response.setContentType("text/html");
17.    PrintWriter out = response.getWriter();
18.    // on prépare le code HTML
19.    String Html="<html><head><title>Spring-MVC-13</title></head>"+
20.           "<body><h2>Vue n° 2 (pour toute langue)</h2>";
21.    // on parcourt la liste des membres du groupe
22.    Groupe groupe=(Groupe)modèle.get("groupe");
```

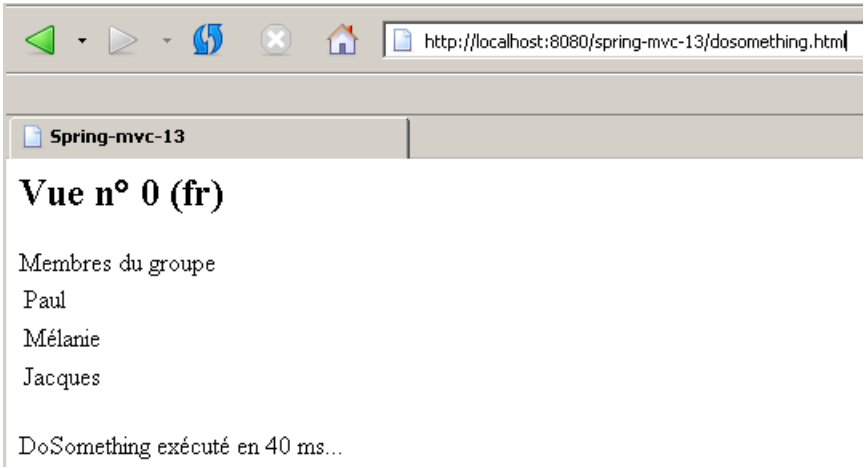
```
23.    long durée=((Long)modèle.get("duree")).longValue();
24.    ArrayList membres=groupe.getMembres();
25.    for(int i=0;i<membres.size();i++){
26.      Html+=membres.get(i).toString()+"<br>\n";
27.    }
28.    // on ajoute la durée d'exécution
29.    Html += "<br>DoSomething exécuté en " + durée + " ms...</body></html>";
30.    // on envoie le flux HTML
31.    out.println(Html);
32.  }
33.
34. }
```

- ligne 20 : on indique par un commentaire (pour toute langue) que la vue ne dépend pas de la langue.

Nous sommes prêts pour les tests. Notre navigateur est tout d'abord configuré pour demander des pages en français :

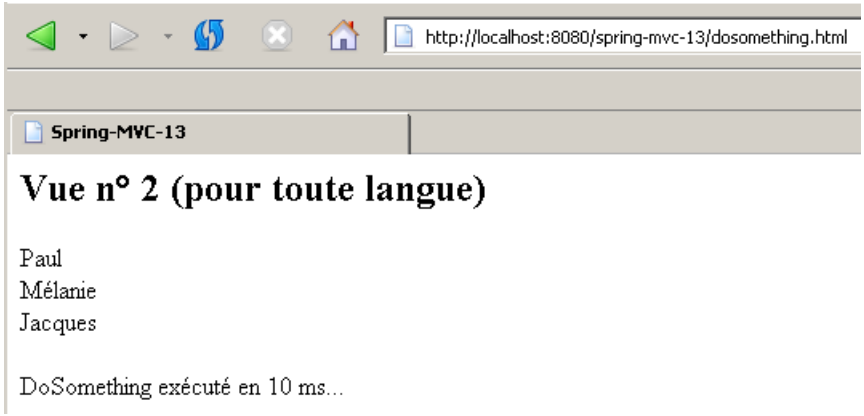


Nous demandons l'url [http://localhost:8080/spring-mvc-13/dosomething.html] :



En rechargeant la page (page reload), nous finissons par avoir les deux autres vues :

La vue " vue2 " générée par la classe [MyView] :



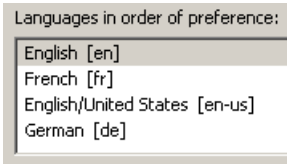
et la vue " vue1 " :

Vue n° 1 (fr)

Membres du groupe		
Paul	Mélanie	Jacques

DoSomething exécuté en 10 ms...

Nous configurons notre navigateur pour qu'il travaille maintenant en anglais :



Nous redemandons l'url [http://localhost:8080/spring-mvc-13/dosomething.html] :

View n° 0 (en)

Group members
Paul
Mélanie
Jacques

DoSomething executed in 10 ms...

Nous avons obtenu la version anglaise de la vue " vue0 ". Nous rechargeons la page autant de fois que nécessaire pour avoir les deux autres vues :

Vue n° 2 (pour toute langue)

Paul
Mélanie
Jacques

DoSomething exécuté en 10 ms...

La vue n° 2 ne change pas car elle est indépendante de la langue.

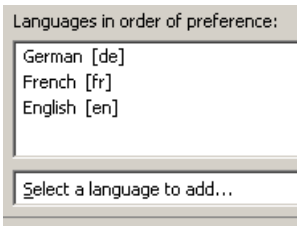
View n° 1 (en)

Group members
Paul	Mélanie	Jacques

DoSomething executed in 10 ms...

On obtient ci-dessus la version anglaise de la vue " vue1 ".

Changeons une nouvelle fois la langue préférée du navigateur pour choisir l'allemand :



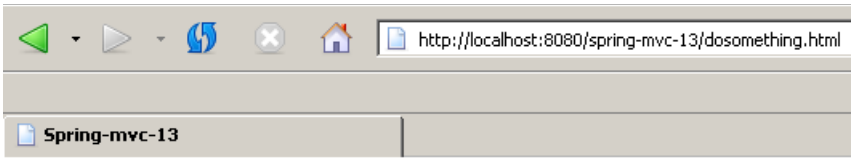
Recommençons le processus précédent. La vue n° 2 reste identique :

Vue n° 2 (pour toute langue)

Paul
Mélanie
Jacques

DoSomething exécuté en 120 ms...

La vue n° 1 obtenue est la suivante :



Vue n° 1 (fr)

Membres du groupe
Paul	Mélanie	Jacques

DoSomething exécuté en 10 ms...

Ceci est anormal. En effet, le fichier [vues_de.properties] n'étant pas défini, Spring MVC aurait du utiliser le fichier par défaut [vues.properties]. Ce n'est pas le cas. C'est le fichier [vues_fr.properties] qui a été utilisé. Après de multiples vérifications, je n'ai pas trouvé l'explication de ce problème. On peut vérifier que l'objet [Locale] associé à la requête est le bon (de). Il y a diverses façons de vérifier ce point. Une façon simple est de créer le fichier [vues_de.properties]. On constate qu'il est alors utilisé.

Pour l'instant, je conclus que je suis passé à côté de quelque chose dans le fonctionnement du résolveur de noms de vues [ResourceBundleViewResolver]. Cela fera l'objet d'un " Errata " si je trouve l'explication.

8.8 Rediriger le navigateur client

Revenons sur l'interface [View] que doivent implémenter les classes chargées d'envoyer une vue au client :

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

Ci-dessus, on voit qu'il existe une classe d'implémentation appelée [RedirectView]. Elle permet d'envoyer en réponse au client, un ordre de redirection vers une URL donnée :

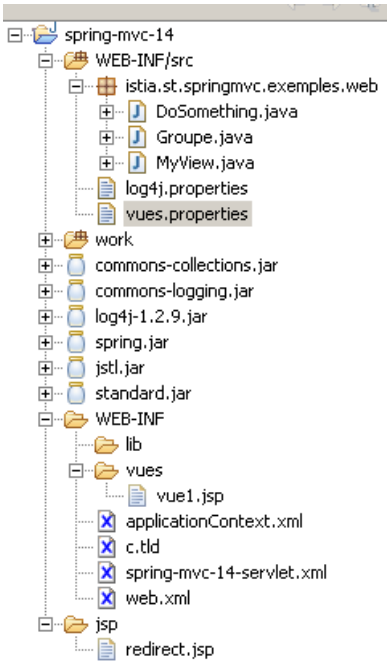
Constructor Summary

RedirectView() Constructor for use as a bean.
RedirectView(String url) Create a new RedirectView with the given URL.
RedirectView(String url, boolean contextRelative) Create a new RedirectView with the given URL.
RedirectView(String url, boolean contextRelative, boolean http10Compatible) Create a new RedirectView with the given URL.

Le dernier constructeur donne les trois paramètres importants pour la redirection :

- **url** : url vers laquelle doit se rediriger le navigateur client
- **contextRelative** (defaut=false) : indique si l'url précédente est relative ou non au contexte de l'application. Si oui, l'url de redirection sera préfixée par le contexte (=nom) de l'application. Ce paramètre n'intervient que si l'url commence par /. Si url=/URL, l'url de redirection sera /C/URL où C est le contexte de l'application si **contextRelative**=true, et simplement /URL si **contextRelative**=false.
- **http10Compatible** (defaut=true) : indique si la redirection doit être comptaible avec le protocole HTTP 1.0. Avec le protocole HTTP 1.1, le code HTTP de redirection est 303 alors qu'avec HTTP 1.0 c'est le code 302. Comme beaucoup de clients HTTP 1.1 traitent le code 302 comme le code 303, il peut être intéressant d'envoyer le code 302 qui conviendra à la fois aux clients HTTP 1.0 et 1.1. C'est ce qui se passe avec **http10Compatible**=true.

Pour illustrer la redirection, nous utiliserons le projet Eclipse suivant :



Ce projet reprend l'essentiel du projet précédent en éliminant toutefois l'aspect internationalisation. On a les mêmes vues quelque soit la langue préférée du navigateur.

Comme précédemment le résolveur de noms de vues est [ResourceBundleViewResolver] et la définition des vues est faite dans [vues.properties] :

```
1.  #vue0
2.  vue0.class=org.springframework.web.servlet.view.RedirectView
3.  vue0.url=/jsp/redirect.jsp
4.  vue0.contextRelative=true
5.  vue0.http10Compatible=false
6.  #vue1
7.  vue1.class=org.springframework.web.servlet.view.JstlView
8.  vue1.url=/WEB-INF/vues/vue1.jsp
9.  #vue2
10. vue2.class=istia.st.springmvc.exemples.web.MyView
```

Comme précédemment, il y aura trois vues : " vue0 ", " vue1 ", " vue2 ". Celles-ci sont définies ci-dessus :

- lignes 1-5 : définissent la vue " vue0 "
- ligne 2 : la classe d'affichage de la vue est [RedirectView]. La vue " vue0 " est donc une demande de redirection.
- ligne 3 : l'url de redirection
- ligne 4 : l'url précédente est relative au contexte. Le navigateur client sera donc redirigé vers l'url /spring-mvc-14/jsp/redirect.jsp. On peut voir plus haut, sur la copie d'écran, cette page JSP. On se rappelle qu'une vue sert à afficher un modèle [Map] construit par un contrôleur [Controller]. Que devient ce modèle [Map] dans le cas d'une redirection ? Il va être ajouté à l'Url de redirection. Ainsi si le modèle [Map] possède les éléments [cle1,valeur1] [cle2,valeur2], l'url de redirection aura la forme [/urlRedirection?cle1=valeur1&cle2=valeur2]. Comme *clei* et *valeuri* sont des objets, ce sont en réalité leurs méthodes [toString] qui sont utilisées pour créer la chaîne des paramètres de la requête. Nous en verrons un exemple bientôt.
- ligne 5 : la redirection sera compatible HTTP 1.0
- lignes 6-8 : définissent la vue " vue1 ". Elle sera affichée par une instance [JstlView].
- lignes 9-10 : définissent la vue " vue2 ". Elle sera affichée par une instance [MyView].

Comme précédemment, le contrôleur [DoSomething] demande de façon aléatoire l'affichage de ces trois vues.

La vue " vue0 " demande la redirection du client vers la vue [redirect.jsp] suivante :

```
1.  <%@ page language="java" pageEncoding="ISO-8859-1" contentType="text/html;charset=ISO-8859-1"%>
2.  <%
3.    String duree=request.getParameter("duree");
4.  %>
5.  <html>
6.    <head>
7.      <title>Spring-mvc-14</title>
8.    </head>
9.    <body>
10.     <h2>Vue n&deg; 0 obtenue par redirection (pour toute langue)</h2>
11.     <br>
```

```
12.    DoSomething ex&eacute;cut&eacute; en <%=duree%> ms...
13.  </body>
14. </html>
```

Le contrôleur [DoSomething] demande à toutes les vues d'afficher un modèle [Map] avec les éléments suivants :

- la durée de l'exécution sous la forme d'un entier long [Long] associé à la clé " duree "
- le groupe de personnes sous la forme d'un objet [Groupe] associé à la clé " groupe ".

L'url de redirection aura alors la forme suivante :

```
http://localhost:8080/spring-mvc-14/jsp/redirect.jsp?groupe=istia.st.springmvc.exemples.web.Groupe%40623367&duree=10
```

On y retrouve :

- l'url de redirection : /spring-mvc-14/jsp/redirect.jsp
- la chaîne des paramètres qui représente le modèle construit par le contrôleur [DoSomething] :

  groupe=istia.st.springmvc.exemples.web.Groupe%40623367&duree=10

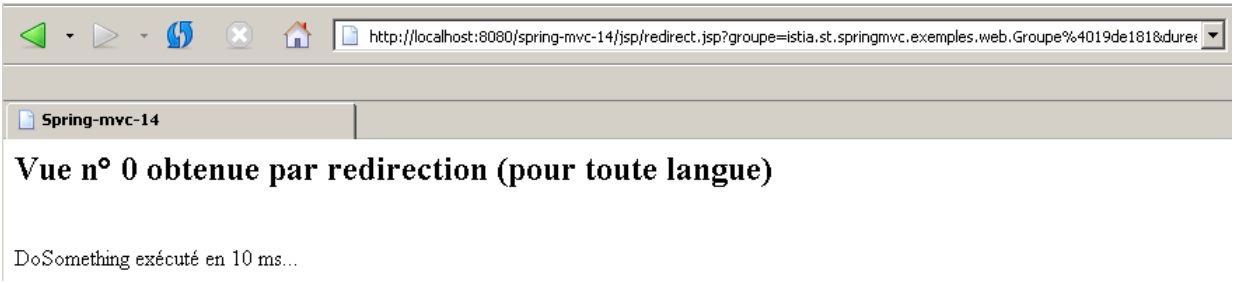
  La clé " duree " étant associée à un type [Long], la méthode [toString] de cette classe a été utilisée et on obtient le 10 de [duree=10]. La clé " groupe " est, elle, associée à un objet de type [Groupe] pour lequel la méthode [toString] n'avait pas été redéfinie. Aussi c'est la méthode [toString] de la classe [Object] qui a été utilisée pour introduire la valeur associée à la clé " groupe " dans la chaîne des paramètres. En redéfinissant une méthode [toString] pour la classe [Groupe], on aurait pu faire en sorte que la valeur soit affichée, par exemple, sous la forme *Paul,Mélanie,Jacques*.

Le seul paramètre exploitable de la chaîne de paramètres étant le paramètre " duree ", la valeur de celui-ci est récupérée ligne 2 de la page [redirect.jsp] et affichée ligne 12.

La vue " vue1 " est la même que dans les exemples précédents, de même que la vue " vue2 " générée par [MyView].

Nous sommes prêts pour les tests. Nous demandons l'url [http://localhost:8080/spring-mvc-14/dosomething.html] :

Nous avons obtenu la vue " vue1 ". En rechargeant la page (Page reload) plusieurs fois, on obtient également les vues " vue0 " et " vue2 " :



La vue " vue0 " a provoqué une redirection vers l'adresse affichée ci-dessus par le navigateur.

8.9 Diriger une vue vers une ressource interne

Revenons sur l'interface [View] que doivent implémenter les classes chargées d'envoyer une vue au client :

org.springframework.web.servlet

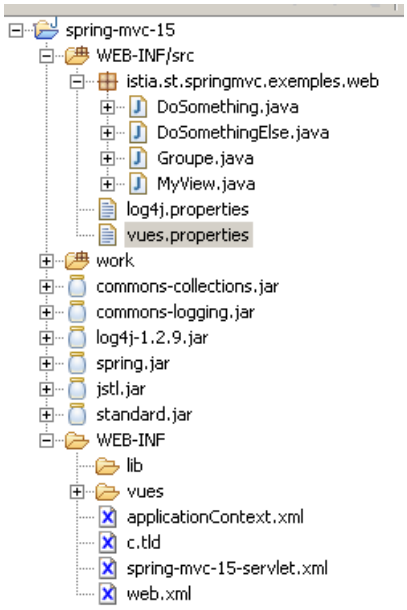
Interface View

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

Ci-dessus, on voit qu'il existe une classe d'implémentation appelée [InternalResourceView]. Elle permet de rediriger le modèle construit par un contrôleur [Controller] vers un autre contrôleur plutôt que vers une vue.

Pour illustrer cette possibilité, nous utiliserons le projet Eclipse suivant :



La configuration [spring-mvc-15-servlet.xml] de la servlet est analogue à celle des deux projets précédents :

```
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="/dosomething.html">DoSomethingController</prop>
9.          <prop key="/dosomethingelse.html">DoSomethingElseController</prop>
10.       </props>
11.     </property>
12.   </bean>
```

```
13.  <!-- les contrôleurs de l'application-->
14.  <bean id="DoSomethingController"
15.    class="istia.st.springmvc.exemples.web.DoSomething">
16.    <property name="groupe">
17.      <ref bean="groupe"/>
18.    </property>
19.  </bean>
20.  <bean id="DoSomethingElseController"
21.    class="istia.st.springmvc.exemples.web.DoSomethingElse">
22.  </bean>
23.  <!-- le résolveur de vues -->
24.  <bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
25.    <property name="basename">
26.      <value>vues</value>
27.    </property>
28.  </bean>
29. </beans>
```

La nouveauté vient du fait que nous ajoutons un second contrôleur [Controller] :

- ligne 9 : l'url [/dosomethingelse.html] est associée au contrôleur d'id [DoSomethingElseController]
- lignes 20-22 : définissent le contrôleur d'id [DoSomethingElseController]
- ligne 21 : la classe d'implémentation est de type [istia.st.springmvc.exemples.web.DoSomethingElse] une classe définie dans [WEB-INF/src] (cf copie d'écran).

Pour comprendre le rôle du contrôleur [DoSomethingElse], il nous faut tout d'abord parler des vues définies dans [vues.properties] :

```
1.  #vue0
2.  vue0.class=org.springframework.web.servlet.view.InternalResourceView
3.  vue0.url=/dosomethingelse.html
4.  #vue1
5.  vue1.class=org.springframework.web.servlet.view.JstlView
6.  vue1.url=/WEB-INF/vues/vue1.jsp
7.  #vue2
8.  vue2.class=istia.st.springmvc.exemples.web.MyView
9.  #vue3
10. vue3.class=org.springframework.web.servlet.view.JstlView
11. vue3.url=/WEB-INF/vues/vue3.jsp
```

- lignes 1-3 : définissent la vue " vue0 "
- ligne 2 : la classe implémentant l'interface [View] est de type [InternalResourceView]
- ligne 3 : la ressource qui doit être instanciée pour traiter le modèle associé à la vue " vue0 ". Cette ressource est ici donnée sous la forme d'une url. La stratégie de résolution des url est utilisée pour trouver la ressource associée. La ligne 9 du fichier [spring-mvc-15-servlet.xml] indique que cette ressource est le contrôleur d'id [DoSomethingElseController]. Il s'agit donc d'une instance de la classe [istia.st.springmvc.exemples.web.DoSomethingElse]. Lorsque le contrôleur [DoSomething] va demander à la vue " vue0 " d'afficher le modèle [Map] qu'il a construit :
  - les éléments du modèle [Map] seront mis un à un dans le contexte de la requête
  - le contrôleur [DoSomethingElse] sera instancié. Il trouvera dans le contexte de la requête, le modèle [Map] construit par le contrôleur [DoSomething]. Il pourra alors travailler avec, l'enrichir, construire un nouveau modèle et demander à son tour l'affichage d'une vue. On a donc, dans cet exemple, un chaînage de contrôleurs. La ressource interne n'est pas nécessairement un objet de type [Controller]. Ce peut être une servlet quelconque capable d'exploiter le contexte de la requête.
- lignes 4-6 : la vue " vue1 " telle qu'on la connaît déjà
- lignes 7-8 : la vue " vue2 " telle qu'on la connaît déjà
- lignes 9-11 : la vue " vue3 " qui sera associée à la page JSP [vue3.jsp]. C'est le contrôleur [DoSomethingElse] qui demande l'affichage de cette vue. Nous y reviendrons.

Le contrôleur [DoSomething] n'a pas changé :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Date;
4.  import java.util.HashMap;
5.
6.  import javax.servlet.http.HttpServletRequest;
7.  import javax.servlet.http.HttpServletResponse;
8.
9.  import org.springframework.web.servlet.ModelAndView;
10. import org.springframework.web.servlet.mvc.Controller;
11.
12. public class DoSomething implements Controller {
13.
14.  // un groupe de personnes fourni par le contexte de l'application
15.  private Groupe groupe;
16.
```

```
17.  public Groupe getGroupe() {
18.    return groupe;
19.  }
20.
21.  public void setGroupe(Groupe groupe) {
22.    this.groupe = groupe;
23.  }
24.
25.  // gestion de la requête
26.  public ModelAndView handleRequest(HttpServletRequest request,
27.      HttpServletResponse response) throws Exception {
28.    // début
29.    long début = new Date().getTime();
30.    // on fait qq chose...
31.    Thread.sleep(10);
32.    // fin
33.    long fin = new Date().getTime();
34.    // durée
35.    long durée = fin - début;
36.    // on crée le modèle de la vue à afficher
37.    HashMap modèle = new HashMap();
38.    modèle.put("groupe", groupe);
39.    modèle.put("duree", new Long(durée));
40.    // on retourne le ModelAndView
41.    int i = (int) (Math.random() * 3);
42.    return new ModelAndView("vue"+i, modèle);
43.  }
44.
45. }
```

Les vues demandées par [DoSomething] sont " vue0 ", " vue1 " et " vue2 ". Les vues " vue1 " et " vue2 " vont être traitées comme dans l'exemple précédent. La vue " vue0 " ne provoque, elle, pas d'affichage de vue au client mais le passage du flux d'exécution au contrôleur [DoSomethingElse]. Celui-ci est le suivant :

```
1.  package istia.st.springmvc.exemples.web;
2.
3.  import java.util.Date;
4.  import javax.servlet.http.HttpServletRequest;
5.  import javax.servlet.http.HttpServletResponse;
6.
7.  import org.springframework.web.servlet.ModelAndView;
8.  import org.springframework.web.servlet.mvc.Controller;
9.
10. public class DoSomethingElse implements Controller {
11.
12.  // gestion de la requête
13.  public ModelAndView handleRequest(HttpServletRequest request,
14.      HttpServletResponse response) throws Exception {
15.    // début
16.    long début = new Date().getTime();
17.    // on fait qq chose...
18.    Thread.sleep(20);
19.    // fin
20.    long fin = new Date().getTime();
21.    // durée
22.    long durée = fin - début;
23.    // on met la durée dans le contexte de la requête
24.    request.setAttribute("duree2",new Long(durée));
25.    // on retourne le ModelAndView
26.    return new ModelAndView("vue3", null);
27.  }
28.
29. }
```

- ligne 13 : la méthode [handleRequest] de [DoSomethingElse] va être appelée par le contrôleur principal [DispatcherServlet]. Elle reçoit en paramètres les mêmes objets [request, response] que le contrôleur [DoSomething]
- les éléments du modèle [Map] créé par le contrôleur [DoSomething] ont été placés par [DispatcherServlet] dans le contexte de la requête. Ainsi doit-on trouver dans ce contexte les clés " duree " et " groupe " du modèle [Map] créé par le contrôleur [DoSomething].
- lignes 15-22 : on calcule la durée de l'exécution
- ligne 24 : qui est mise ensuite dans le contexte de la requête associée à la clé " duree2 "
- ligne 26 : on retourne un [ModelAndView] tel que la vue est " vue3 " et le modèle est vide. Ce qu'on veut montrer ici, c'est que le modèle n'est pas obligatoire. Si on est certain par exemple que la vue va être affichée par la classe [JstlView], le contexte de la requête peut servir de modèle. Néanmoins, cette méthode est à déconseiller car elle fait perdre de la souplesse à l'application.

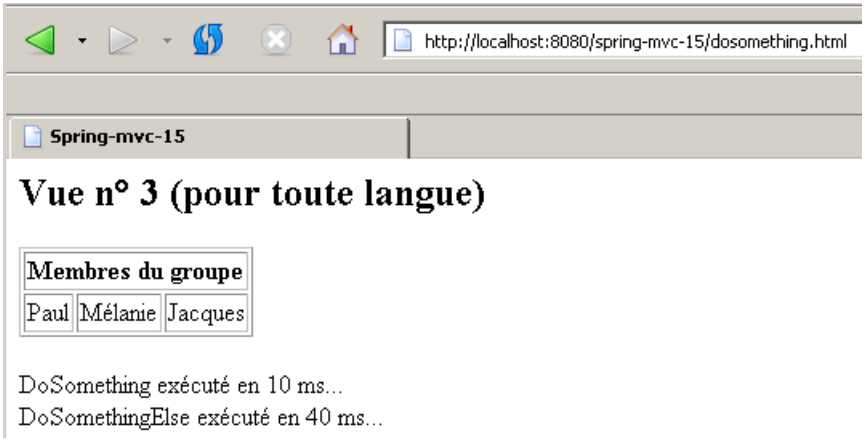
La vue " vue3 " est dans [vues.properties] associée à la page JSP [vue3.jsp]. Celle-ci est la suivante :

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

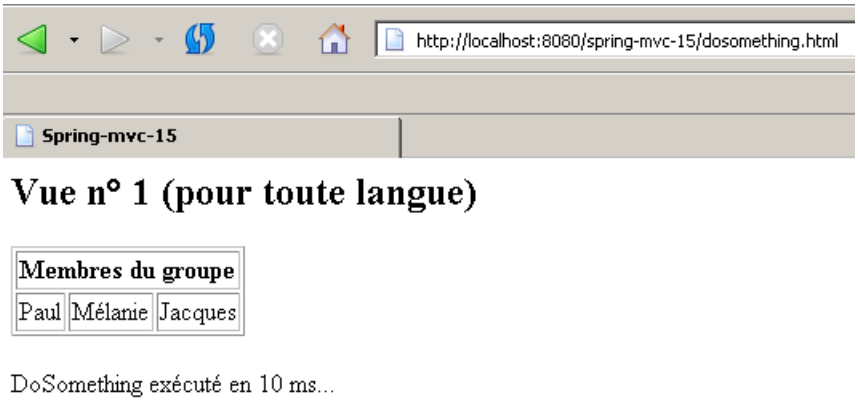
```
3.  <%@ page isELIgnored="false" %>
4.
5.  <html>
6.    <head>
7.      <title>Spring-mvc-15</title>
8.    </head>
9.    <body>
10.     <h2>Vue n&deg; 3 (pour toute langue)</h2>
11.     <table border="1">
12.       <tr>
13.         <th colspan="3" align="center">Membres du groupe</th>
14.       </tr>
15.       <tr>
16.         <c:forEach var="personne" items="${groupe.membres}">
17.           <td>${personne}</td>
18.         </c:forEach>
19.       </tr>
20.     </table>
21.     <br>
22.     DoSomething ex&eacute;cut&eacute; en ${duree} ms...<br>
23.     DoSomethingElse ex&eacute;cut&eacute; en ${duree2} ms...<br>
24.   </body>
25. </html>
```

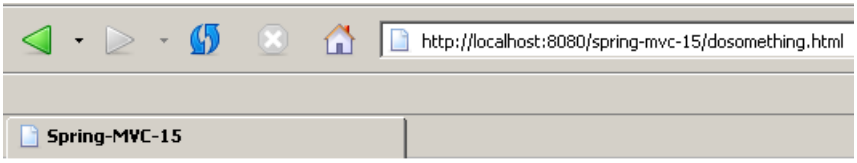
- la page JSP affiche les éléments de clé " groupe " (ligne 16), " duree " (ligne 22) et " duree2 " (ligne 23).
- les éléments de clé " groupe " (ligne 16) et " duree " (ligne 22) ont été tout d'abord placés dans le modèle [Map] créé par le contrôleur [DoSomething]. Lors du passage de témoin entre [DoSomething] et [DoSomethingElse], le contrôleur [DispatcherServlet] a placé les éléments du modèle dans le contexte de la requête. C'est pourquoi la page JSP présente peut les récupérer.
- la clé " duree2 " a elle été placée explicitement dans le contexte de la requête par le contrôleur [DoSomethingElse].

Nous sommes prêts pour des tests. Nous demandons l'url [http://localhost:8080/spring-mvc-15/dosomething.html] :



Nous avons ici obtenu la vue " vue3 ". En rechargeant plusieurs fois la page, on obtient les deux autres vues :





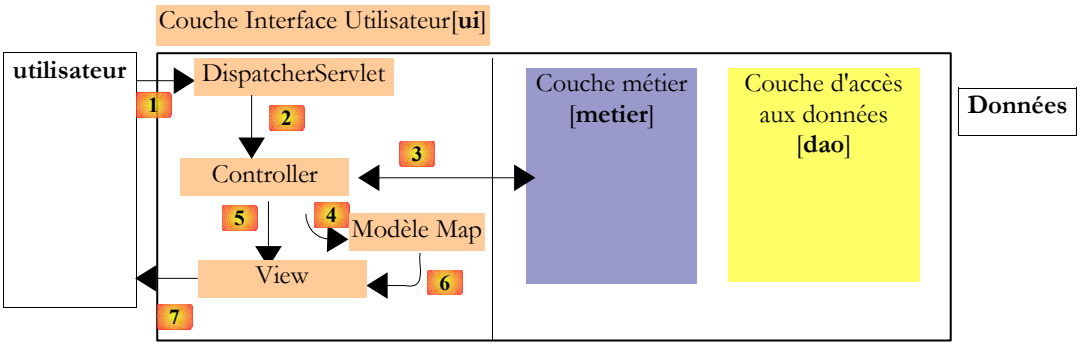
Vue n° 2 (pour toute langue)

Paul
Mélanie
Jacques

DoSomething exécuté en 10 ms...

9 Conclusion de la partie 1

Rappelons l'architecture Spring MVC que nous avons étudiée dans ce document :



Nous avons vu les points suivants :

- les différentes stratégie de résolutions d'URL qui associent à une URL demandée par le client, un contrôleur implémentant l'interface [Controller] qui va traiter la demande du client
- les différentes façons qu'avait un contrôleur [Controller] d'accéder au contexte de l'application, par exemple pour accéder aux instances des couches [métier] et [dao]
- les différentes stratégie 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]

Il nous reste

- à travailler sur les implémentations de l'interface [Controller] que met à notre disposition Spring MVC. L'une d'elles, appelée [SimpleFormController], nous permet d'écrire des contrôleurs de formulaire HTML avec contrôle de validité des données, d'une manière analogue à celle utilisée dans le framework Struts.
- à présenter une application web MVC 3tier un peu réaliste
- à présenter divers compléments tels que par exemple les vues Excel ou PDF

Table des matières

1GÉNÉRALITÉS....2

2OUTILS UTILISÉS ET RÉFÉRENCES....4

3LE CODE DES EXEMPLES....4

4LES COMPOSANTES D'UNE ARCHITECTURE SPRING MVC....9

5CONFIGURATION D'UNE APPLICATION SPRING MVC....13

6LA LIAISON URL - CONTROLLER....18

6.1Les stratégies de résolution des URL....18
6.2La stratégie SimpleUrlHandlerMapping....19
6.3La stratégie BeanNameUrlHandlerMapping....21
6.4La stratégie SimpleUrlHandlerMapping avec liaisons externalisées....23

7ACCÈS AU CONTEXTE D'UNE APPLICATION SPRING MVC....24

7.1Exemple 1....24
7.2Exemple 2....28
7.3Exemple 3....31
7.4Exemple 4....33

8GESTION DES VUES....35

8.1La place des vues dans l'architecture Spring MVC....35
8.2Implémenter l'interface View....38
8.3L'implémentation JstlView....41
8.4Le résolveur de noms de vue [InternalResourceViewResolver]....46
8.5Le résolveur de noms de vue [BeanNameViewResolver]....49
8.6Le résolveur de noms de vue [XmlViewResolver]....51
8.7Le résolveur de noms de vue [ResourceBundleViewResolver]....52
8.8Rediriger le navigateur client....62
8.9Diriger une vue vers une ressource interne....65

9CONCLUSION DE LA PARTIE 1....69