

# Spring

## par la pratique

2<sup>e</sup> édition  
Spring 2.5  
et 3.0

**Arnaud Cogoluègnes**  
**Thierry Templier**  
**Julien Dubois**  
**Jean-Philippe Retailé**

avec la contribution  
de Séverine Templier Roblou  
et de Olivier Salvatori

© Groupe Eyrolles, 2006, 2009,

ISBN : 978-2-212-12421-7

**EYROLLES**

# 7

## Spring MVC

---

La mise en pratique du patron de conception MVC (Model View Controller) offre une meilleure structuration du tiers de présentation des applications Java EE en dissociant les préoccupations de déclenchement des traitements de la construction de la présentation proprement dite. Les principaux frameworks MVC implémentent le type 2 de ce patron, qui instaure un point d'entrée unique ayant pour mission d'aiguiller les requêtes vers la bonne entité de traitement.

Le framework Spring offre une implémentation innovante du patron MVC par le biais d'un framework nommé Spring MVC, qui profite des avantages de l'injection de dépendances (*voir chapitres 2 et 3*) et qui, depuis la version 2.5, offre une intéressante flexibilité grâce aux annotations Java 5. Ce module permet dès lors de s'abstraire de l'API Servlet de Java EE, les informations souhaitées étant automatiquement mises à disposition en tant que paramètres des méthodes des contrôleurs.

De plus, à partir de la version 3.0, Spring MVC intègre un support permettant de gérer la technologie REST, les URL possédant la structure décrite par cette dernière étant exploitable en natif.

Le présent chapitre passe en revue les fonctionnalités et apports de ce module, qui met en œuvre les principes généraux du framework Spring, lesquels consistent à masquer l'API Servlet et à simplifier les développements d'applications Java EE tout en favorisant leur structuration et leur flexibilité.

### Implémentation du pattern MVC de type 2 dans Spring

Cette section décrit brièvement l'implémentation du patron de conception MVC de type 2 dans le framework Spring.

Nous présenterons rapidement les concepts de base du type 2 de ce patron puis nous concentrerons sur les principes de fonctionnement et constituants de Spring MVC, l'implémentation du patron MVC par Spring.

## Fonctionnement du patron MVC 2

Le patron MVC est communément utilisé dans les applications Java/Java EE pour réaliser la couche de présentation des données aussi bien dans les applications Web que pour les clients lourds. Lorsqu'il est utilisé dans le cadre de Java EE, il s'appuie généralement sur l'API servlet ainsi que sur des technologies telles que JSP/JSTL.

Il existe deux types de patrons MVC, celui dit de type 1, qui possède un contrôleur par action, et celui dit de type 2, plus récent et plus flexible, qui possède un contrôleur unique. Nous nous concentrerons sur ce dernier, puisqu'il est implémenté dans les frameworks MVC.

La figure 7-1 illustre les différentes entités du type 2 du patron MVC ainsi que leurs interactions lors du traitement d'une requête.

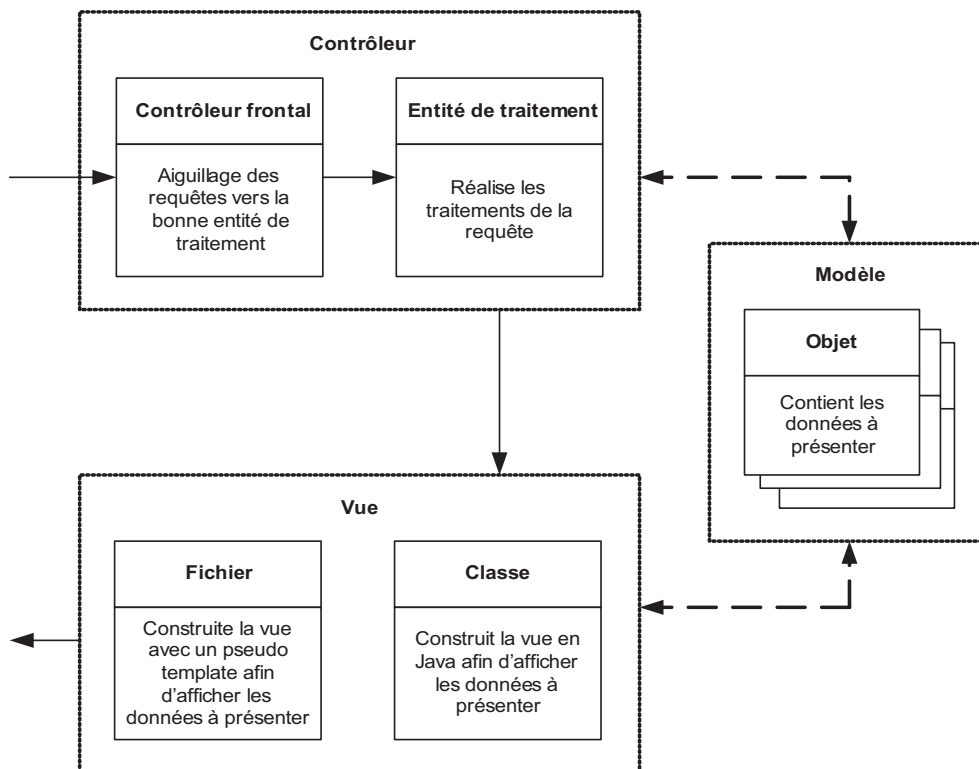


Figure 7-1

Entités mises en œuvre dans le patron MVC de type 2

Les caractéristiques des composants mis en œuvre dans ce patron sont les suivantes :

- **Modèle.** Permet de mettre à disposition les informations utilisées par la suite lors des traitements de présentation. Cette entité est indépendante des API techniques et est constituée uniquement de Beans Java.
- **Vue.** Permet la présentation des données du modèle. Il existe plusieurs technologies de présentation, parmi lesquelles JSP/JSTL, XML, les moteurs de templates Velocity et FreeMarker ou de simples classes Java pouvant générer différents types de formats.
- **Contrôleur.** Gère les interactions avec le client tout en déclenchant les traitements appropriés. Cette entité interagit directement avec les composants de la couche service métier et a pour responsabilité la récupération des données mises à disposition dans le modèle. Lors de la mise en œuvre du type 2 de ce patron, cette partie se compose d'un point d'entrée unique pour toute l'application et de plusieurs entités de traitement. Ce point d'entrée unique traite la requête et dirige les traitements vers l'entité appropriée. Pour cette raison, l'entité de traitement est habituellement appelée contrôleur. Le contrôleur frontal, ou « contrôleur façade », est intégré au framework MVC, et seuls les entités de traitement sont spécifiques à l'application.

Un framework MVC implémente le contrôleur façade, les mécanismes de gestion du modèle, ainsi que ceux de sélection et de construction de la vue. L'utilisateur d'un tel framework a en charge le développement et la configuration des entités de traitements et des vues choisies.

## ***Principes et composants de Spring MVC***

Le framework Spring fournit des intégrations avec les principaux frameworks MVC ainsi que sa propre implémentation. Forts de leur expérience dans le développement d'applications Java EE, ses concepteurs considèrent que l'injection de dépendances offre un apport de taille pour concevoir et structurer des applications fondées sur le patron MVC.

Précisons que Spring MVC ne constitue qu'une partie du support relatif aux applications Web. Le framework Spring offre d'autres fonctionnalités permettant notamment le chargement des contextes applicatifs de manière transparente ainsi que des intégrations avec d'autres frameworks MVC, tels JSF, WebWork ou Tapestry.

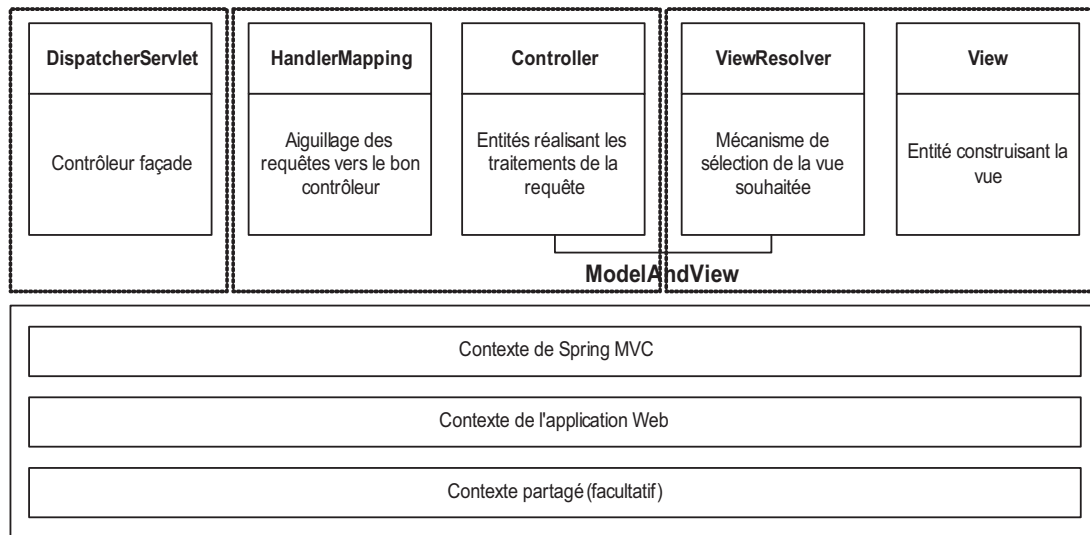
Parmi les principes fondateurs de Spring MVC, remarquons notamment les suivants :

- Utilisation du conteneur léger afin de configurer les différentes entités du patron MVC et de bénéficier de toutes les fonctionnalités du framework Spring, notamment au niveau de la résolution des dépendances.
- Favorisation de la flexibilité et du découplage des différentes entités mises en œuvre grâce à la programmation par interface.
- Utilisation d'une hiérarchie de contextes applicatifs afin de réaliser une séparation logique des différents composants de l'application. Par exemple, les composants des services métier et des couches inférieures n'ont pas accès à ceux du MVC.

Les composants du MVC ont pour leur part les principales caractéristiques suivantes :

- À partir de la version 2.5 de Spring, la gestion de l'aiguillage et la configuration des contrôleurs MVC se réalisent par l'intermédiaire d'annotations. Par ce biais, Spring MVC permet de masquer l'utilisation de l'API servlet et favorise la mise en œuvre des tests unitaires à ce niveau. L'approche fondée sur les classes d'implémentation de contrôleurs est désormais dépréciée.
- Gestion des formulaires en se fondant sur les annotations relatives aux contrôleurs afin non seulement de charger et d'afficher les données du formulaire, mais également de gérer leur soumission. Ces données sont utilisées pour remplir directement un Bean sans lien avec Spring MVC, qui peut être validé si nécessaire. Des mécanismes de mappage et de conversion des données sont intégrés et extensibles selon les besoins.
- Abstraction de l'implémentation des vues par rapport aux contrôleurs permettant de changer de technologie de présentation sans impacter le contrôleur.

Pour mettre en œuvre ces principes et composants, Spring MVC s'appuie sur les entités illustrées à la figure 7-2. Le traitement d'une requête passe successivement par les différentes entités en commençant par la servlet `DispatcherServlet` et en finissant par la vue choisie.



**Figure 7-2**

*Entités de traitement des requêtes de Spring MVC*

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

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

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

## Initialisation du framework Spring MVC

L'initialisation du framework Spring MVC s'effectue en deux parties, essentiellement au sein du fichier **web.xml** puisqu'elle utilise des mécanismes de la spécification Java EE servlet.

### *Gestion des contextes*

Le framework Spring permet de charger automatiquement les contextes applicatifs en utilisant les mécanismes des conteneurs de servlets.

Dans le cadre d'applications Java EE, une hiérarchie de contextes est mise en œuvre afin de regrouper et d'isoler de manière logique les différents composants. De la sorte, un composant d'une couche ne peut accéder à celui d'une couche supérieure.

#### **Contexte applicatif Spring**

Rappelons qu'un contexte applicatif correspond au conteneur léger en lui-même, dont la fonction est de gérer des Beans. Le framework offre également un mécanisme permettant de définir une hiérarchie de contextes afin de réaliser une séparation logique entre des groupes de Beans. Dans le cas du MVC, il s'agit d'empêcher l'utilisation de composants du MVC par des composants service métier ou d'accès aux données.

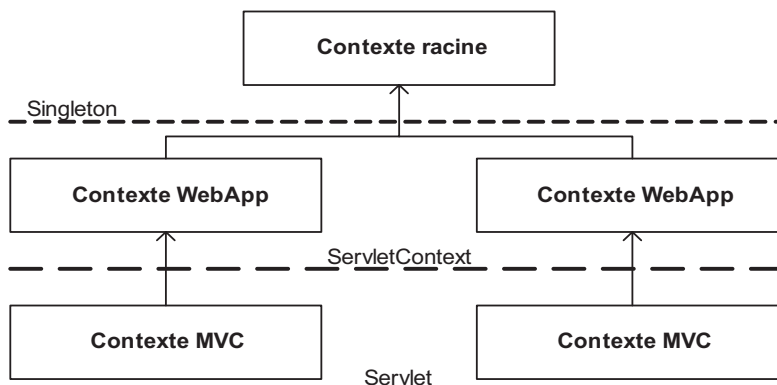
Le framework Spring offre une hiérarchie pour les trois contextes suivants :

- Contexte racine. Ce contexte est très utile pour partager des objets d'une même bibliothèque entre plusieurs modules d'une même application Java EE pour un même chargeur de classes.
- Contexte de l'application Web. Stocké dans le ServletContext, ce contexte doit contenir la logique métier ainsi que celle de l'accès aux données.
- Contexte du framework MVC. Géré par le contrôleur façade du framework, ce contexte doit contenir tous les composants relatifs au framework MVC utilisé.

La figure 7-3 illustre cette hiérarchie ainsi que la portée des différents contextes.

**Figure 7-3**

*Hiérarchie des contextes de Spring pour une application Web*



La mise en œuvre du contexte de l'application Web est obligatoire, tandis que celle du contexte racine est optionnelle et n'est donc pas détaillée ici. Si ce dernier est omis, Spring positionne de manière transparente celui de l'application Web en tant que contexte racine.

L'initialisation du contexte de l'application Web, que nous détaillons dans ce chapitre, est indépendante du framework MVC choisi et utilise les mécanismes du conteneur de servlets. Sa configuration est identique dans le cadre du support d'autres frameworks MVC.

### Chargement du contexte de l'application Web

Le framework Spring fournit une implémentation de la classe `ServletContextListener` de la spécification servlet permettant de configurer et d'initialiser ce contexte au démarrage et de le finaliser à l'arrêt de l'application Web.

Cette fonctionnalité est utilisable avec un conteneur de servlets supportant au moins la version 2.3 de la spécification. Cet observateur paramètre les fichiers de configuration XML du contexte en ajoutant les lignes suivantes dans le fichier **web.xml** de l'application :

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext*.xml</param-value>
</context-param>

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

## Chargement du contexte de Spring MVC

La configuration tout comme le chargement de ce contexte sont liés à ceux de la servlet du contrôleur façade de Spring MVC. Précisons que, par défaut, cette dernière initialise un contexte applicatif fondé sur un fichier **<nom-servlet>-servlet.xml**, lequel utilise le nom de la servlet précédente pour `<nom-servlet>`. Ce fichier se situe par défaut dans le répertoire **WEB-INF** ; la valeur de `<nom-servlet>` est spécifiée grâce à la balise `servlet-name`.

Dans notre étude de cas, la servlet de Spring MVC s'appelle `tudu`. Le fichier **tudu-servlet.xml** contient les différents composants utilisés par ce framework, comme les contrôleurs, les vues et les entités de résolution des requêtes et des vues.

Nous pouvons personnaliser le nom de ce fichier à l'aide du paramètre d'initialisation `contextConfigLocation` de la servlet. Le code suivant montre comment spécifier un fichier **mvc-context.xml** pour le contexte de Spring MVC par l'intermédiaire du paramètre précédemment cité (❶) :

```
<web-app>
  (...)
  <servlet>
    <servlet-name>tudu</servlet-name>
    (...)
    <init-param>❶
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/mvc-context.xml</param-value>
    </init-param>
  </servlet>
</web-app>
```

## Initialisation du contrôleur façade

Le fait que le framework Spring MVC implémente le patron MVC de type 2 entraîne qu'il met en œuvre un contrôleur façade pour diriger les traitements vers des classes désignées par le terme *Controller* dans Spring MVC.

### Le contrôleur façade

Cette entité correspond à l'unique point d'accès de l'application Web. Son rôle est de rediriger les traitements vers le bon contrôleur en se fondant sur l'adresse d'accès pour traiter la requête. Dans le cas d'applications Web, ce contrôleur est implémenté par le biais d'une servlet, qui est généralement fournie par le framework MVC utilisé.

Ce contrôleur façade est implémenté par le biais de la servlet `DispatcherServlet` du package `org.springframework.web.servlet`, cette dernière devant être configurée dans le fichier **WEB-INF/web.xml**.

Le mappage de la ressource (❶) est défini au niveau du conteneur de servlets dans le fichier **web.xml** localisé dans le répertoire **WEB-INF**. Spring ne pose aucune restriction à ce niveau, comme l'illustre le code suivant :

```
<web-app>
  ...
  <servlet>
    <servlet-name>tudu</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>
  <servlet-mapping>←❶
    <servlet-name>tudu</servlet-name>
    <url-pattern>*.do</url-pattern>
  </servlet-mapping>
</web-app>
```

### Support des annotations pour les contrôleurs

Comme indiqué précédemment, à partir de la version 2.5 de Spring, la configuration des contrôleurs se réalise par l'intermédiaire d'annotations. Bien que cette approche soit celle à utiliser, il reste néanmoins nécessaire de l'activer dans la configuration de Spring.

Cet aspect peut être mis en œuvre de deux manières à l'instar de la configuration des composants par l'intermédiaire des annotations dans Spring.

La première approche consiste à spécifier une implémentation de l'interface `HandlerMapping` fondée sur les annotations. Spring MVC propose la classe `DefaultAnnotationHandlerMapping` à cet effet. Cette dernière peut être utilisée conjointement avec la classe `AnnotationMethodHandlerAdapter` afin de configurer les méthodes de traitements des requêtes dans les contrôleurs avec des annotations.

Avec cette approche, les Beans des contrôleurs doivent être configurés en tant que Beans dans le contexte de Spring MVC.

Le code suivant décrit l'utilisation de ces deux classes dans le fichier de configuration Spring associé à la servlet `DispatcherServlet` de Spring MVC :

```
<beans (...)>
  <bean class="org.springframework.web.servlet
    .mvc.annotation.DefaultAnnotationHandlerMapping"/>
  <bean class="org.springframework.web.servlet
    .mvc.annotation.AnnotationMethodHandlerAdapter"/>
</beans>
```

La seconde consiste en l'utilisation de la balise `component-scan` de l'espace de nommage `context` afin de détecter tous les composants présents et notamment les contrôleurs Spring MVC. Ces derniers n'ont plus à être définis en tant que Beans dans la configuration de Spring. Dans ce contexte, l'annotation `Autowired` doit être utilisée pour l'injection de dépendances. Pour plus de précision, reportez-vous au chapitre 2.

La configuration de cet aspect se réalise dans le fichier de configuration Spring associé à la servlet `DispatcherServlet` de Spring MVC :

```
<beans (...)>
  <context:component-scan base-package="tudu.web" />
</beans>
```

Il est recommandé de n'utiliser la première approche que si une personnalisation de la stratégie de mappage des requêtes est envisagée. La seconde approche reste donc celle à utiliser par défaut.

### *En résumé*

Cette section a détaillé les mécanismes généraux de fonctionnement du patron MVC de type 2 ainsi que la structuration utilisée dans le framework Spring MVC afin de le mettre en œuvre. Nous avons pu constater que les entités utilisées permettent de bien modulariser les traitements et d'isoler le contrôleur de la vue.

Les mécanismes de chargement des contextes, de configuration du contrôleur façade et de l'approche dirigée par les annotations du framework Spring MVC ont également été décrits.

Nous allons maintenant détailler la façon dont Spring MVC gère les requêtes et les vues.

## Traitement des requêtes

Comme nous l'avons évoqué précédemment, l'approche de Spring MVC afin de traiter les requêtes est désormais complètement dirigée par des annotations. Ces dernières permettent de configurer aussi bien l'aiguillage des requêtes que les contrôleurs eux-mêmes.

Il est à noter dans ce contexte que l'annotation `Controller` permet de préciser qu'une classe correspond à un contrôleur MVC et qu'elle contient les traitements correspondants.

Une fois cette annotation positionnée, le mappage des URI doit être défini afin de sélectionner le contrôleur de traitement pour une requête Web donnée. Cela se configure également par le biais des annotations, ainsi que nous le détaillons dans la prochaine section.

### *Sélection du contrôleur*

Comme pour tout framework implémentant le patron MVC de type 2, un mécanisme de correspondance entre la classe de traitement appropriée et l'URI de la requête est intégré. Le framework configure cette correspondance en se fondant sur les informations présentes dans les annotations `RequestMapping` des contrôleurs.

Afin de configurer ce mécanisme, il est indispensable de bien comprendre la structure de l'URL d'une requête, qui apparaît toujours sous la forme suivante dans les applications Java EE :

```
http://<machine>:<port>/<alias-webapp>/<alias-ressource-web>
```

L'URI correspond à la fin de l'URL :

```
■ /<alias-webapp>/<alias-ressource-web>
```

L'alias de l'application Web, <alias-webapp>, est configuré au niveau du serveur d'applications, contrairement à celui de la ressource Web, <alias-ressource-web>, qui se réalise au sein de l'application.

Dans un premier temps, l'accès à la servlet `DispatcherServlet` de Spring MVC est paramétré dans le fichier `web.xml` du répertoire **WEB-INF** afin de prendre en compte un ensemble d'URI avec des mappages de la forme `*/quelquechose/*` ou `*.quelquechose`. Nous avons détaillé la configuration de cet aspect à la section « Initialisation du contrôleur façade » précédemment dans ce chapitre.

Avec les annotations, l'implémentation `DefaultAnnotationHandlerMapping` de l'interface `HandlerMapping` est utilisée implicitement ou explicitement suivant la configuration. Elle se fonde sur les informations présentes dans les annotations de type `RequestMapping`. Cette dernière peut être présente aussi bien au niveau de la classe du contrôleur que des méthodes de ce dernier. Les informations spécifiées par ce biais au niveau de la classe lui sont globales, avec la possibilité de les surcharger au niveau des méthodes.

Cet aspect offre d'intéressantes perspectives afin de configurer différents types de contrôleurs, tels que ceux à entrées multiples ou dédiés à la gestion des formulaires. Nous détaillons cet aspect plus loin dans ce chapitre.

Le tableau 7-1 récapitule les différentes propriétés utilisables de l'annotation `RequestMapping`.

**Tableau 7-1. Propriétés de l'annotation `RequestMapping`**

Propriété	Type	Description
<code>method</code>	<code>String[]</code>	Spécifie la ou les méthodes HTTP supportées par le mappage. La spécification d'une méthode se réalise par l'intermédiaire des valeurs de l'énumération <code>RequestMethod</code> .
<code>params</code>	<code>String[]</code>	Permet de réaliser un mappage plus fin en se fondant sur les paramètres de la requête. La présence ou la non-présence (avec l'opérateur <code>!</code> ) d'un paramètre peut être utilisée.
<code>value</code>	<code>String[]</code>	Correspond à la valeur de l'annotation. Cette propriété permet de définir la ou les valeurs définissant le mappage de l'élément. Ces valeurs peuvent éventuellement correspondre à des expressions régulières au format Ant.

L'exemple suivant illustre l'utilisation de l'annotation au niveau de la classe du contrôleur (❶) afin de spécifier la valeur du mappage ainsi qu'au niveau de la méthode dédiée au traitement de la requête (❷) :

```
@Controller
@RequestMapping("/welcome.do")←❶
public class WelcomeController {

    @RequestMapping←❷
    public void welcome() {
        (...)
    }
}
```

Dans l'exemple ci-dessus, l'annotation au niveau de la méthode reprend les valeurs des propriétés de l'annotation positionnée au niveau de la classe.

Il est également possible de ne spécifier le mappage qu'au niveau de la méthode de traitement (❶) :

```
@Controller
public class WelcomeController {

    @RequestMapping("/welcome.do")←❶
    public void welcome() {
        (...)
    }
}
```

Pour finir, il est également possible de spécifier plusieurs mappages (❶) pour une même annotation avec la méthode HTTP d'accès souhaitée (❷) ainsi qu'un filtrage se fondant sur les valeurs des paramètres de la requête (❸), comme l'illustre le code suivant :

```
@Controller
public class WelcomeController {

    @RequestMapping(
        value={"/welcome.do", "/index.do"},←❶
        method=RequestMethod.GET←❷
        params={"auth=true", "refresh", "!authenticate"})←❸
    public void welcome() {
        (...)
    }
}
```

Dans l'exemple ci-dessus, la méthode `welcome` du contrôleur est appelée pour les URI `/<alias-webapp>/welcome.do` ou `/<alias-webapp>/index.do` seulement par la méthode HTTP GET si les conditions sur les paramètres sont vérifiées. Dans notre cas, le paramètre `auth` doit posséder la valeur `true`, le paramètre `refresh` doit être présent, et le paramètre `authenticate` ne doit pas l'être.

## Les types de contrôleurs

Comme indiqué précédemment, Spring MVC fournit l'annotation `Controller` afin de définir une classe en tant que contrôleur. Cette approche est particulièrement flexible à mettre en œuvre, car elle permet de s'abstraire de l'API Servlet et de définir le contenu des contrôleurs et les signatures des méthodes en fonction des besoins.

Les sections qui suivent détaillent les différents mécanismes et déclinaisons utilisables afin de mettre en œuvre des contrôleurs dans Spring MVC.

## Contrôleurs de base

Pour mettre en œuvre les contrôleurs de ce type, l'utilisation de l'annotation `RequestMapping` précédemment décrite est suffisante. Les méthodes sur lesquelles est appliquée cette annotation prennent en paramètres des objets de type `HttpServletRequest` et `HttpServletResponse` et retournent un objet de type `ModelAndView`.

Ces contrôleurs peuvent être à entrées multiples puisqu'il est possible en standard de positionner une annotation `RequestMapping` sur plusieurs de leurs méthodes.

### Point d'entrée

Dans le contexte du pattern MVC, un point d'entrée correspond à une méthode d'un composant qui peut être utilisée par le conteneur ou le framework qui le gère afin de traiter une requête. La signature de cette méthode suit habituellement des conventions spécifiques afin de pouvoir être appelée.

Le code suivant illustre la mise en œuvre d'un contrôleur simple en se fondant sur l'annotation `RequestMapping` (❶) :

```
@Controller
public class ShowTodosController {
    (...)
    @RequestMapping("/showTodos.do") ←❶
    public ModelAndView showTodos(HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        Collection<TodoList> todoLists = new TreeSet<TodoList>(
            userManager.getCurrentUser().getTodoLists());

        String listId = null;
        if (!todoLists.isEmpty()) {
            listId = request.getParameter("listId");
            if (listId != null) {
                listId = todoLists.iterator().next().getListId();
            }
        }

        Map<String, Object> model = new HashMap<String, Object>();
        model.put("defaultList", listId);
        return new ModelAndView("todos", model);
    }
}
```

Aucune autre configuration, si ce n'est l'injection des dépendances avec l'annotation `Autowired`, n'est nécessaire.

Spring MVC offre néanmoins une approche intéressante et flexible afin de supporter différentes signatures de méthodes de traitements des contrôleurs et de spécifier des méthodes de remplissage du modèle. Nous décrivons cette approche, dont l'utilisation est recommandée, aux sections suivantes.

## Support des paramètres et retours des méthodes

En parallèle de la signature type pour les méthodes de traitement des contrôleurs, signature héritée des précédentes versions de Spring MVC, le framework permet si nécessaire de s'abstraire des API Servlet et de Spring MVC. Il est en ce cas possible de spécifier une signature de méthodes en fonction de ses besoins. La détermination des points d'entrée d'un contrôleur est déterminée par la présence de l'annotation `RequestMapping`.

Il est à noter que cette approche est également valable pour les méthodes annotées par `ModelAttribute` et `InitBinder`.

Spring MVC permet de passer directement des paramètres précis soit par type, soit en se fondant sur des annotations supplémentaires. Le framework permet en outre de retourner différents types en fonction de ses besoins. Ces deux possibilités peuvent se combiner pour une plus grande flexibilité.

Le tableau 7-2 récapitule les différents paramètres supportés par Spring MVC pour les méthodes de gestion des requêtes Web, et le tableau 7-3 les types de retours possibles.

**Tableau 7-2. Types de paramètres possibles pour une méthode d'un contrôleur**

Type de paramètre	Description
<code>ServletRequest</code> ou <code>HttpServletRequest</code>	Requête par l'intermédiaire de l'API Servlet.
<code>ServletResponse</code> ou <code>HttpServletResponse</code>	Réponse de la requête par l'intermédiaire de l'API Servlet.
<code>HttpSession</code>	Session de l'initiateur de la requête par l'intermédiaire de l'API Servlet.
<code>WebRequest</code> ou <code>NativeWebRequest</code>	Accès d'une manière générique aux paramètres de la requête sans utiliser l'API Servlet.
Locale	Couple pays et langue associé à la requête.
<code>InputStream</code> ou <code>Reader</code>	Flux d'entrée associé à la requête afin d'avoir accès au contenu de la requête.
<code>OutputStream</code> ou <code>Writer</code>	Flux de sortie associé à la réponse de la requête afin de générer le contenu de la réponse.
Paramètre annoté par <code>RequestParam</code>	Paramètre de la requête dont l'identifiant est celui spécifié dans l'annotation. Spring a la responsabilité de le récupérer dans la requête et de le convertir dans le type attendu.
<code>Map</code> , <code>Model</code> ou <code>ModelMap</code>	Modèle utilisé pour les données présentées dans la vue. Celui-ci offre la possibilité d'avoir accès aux données contenues dans le modèle et de les manipuler.
Type correspondant à un objet de formulaire et annoté par <code>ModelAttribute</code>	Objet de formulaire récupéré dans le modèle en se fondant sur l'identifiant spécifié dans l'annotation.
<code>Errors</code> ou <code>BindingResult</code>	Résultat du mappage et validation d'objets de formulaire. Une validation personnalisée peut se fonder sur ce paramètre afin d'enregistrer les erreurs.
<code>SessionStatus</code>	Dans le cas d'un formulaire mis en œuvre sur plusieurs pages, cet objet offre la possibilité de relâcher les ressources mises en œuvre à cet effet.

Tableau 7-3. Types de retours possibles pour une méthode d'un contrôleur

Type de retour	Description
Map	Objet contenant les données du modèle à utiliser dans la vue. L'identifiant de la vue est implicitement déduit par Spring MVC ( <i>voir le cas void, où aucun objet n'est retourné</i> ).
Model	Identique au précédent.
ModelAndView	Objet regroupant l'identifiant de la vue à utiliser suite aux traitements du contrôleur et le contenu du modèle pour cette dernière.
String	Identifiant de la vue à utiliser suite aux traitements du contrôleur.
View	Vue à utiliser suite aux traitements du contrôleur.
void	Dans le cas où aucun objet n'est retourné, Spring MVC déduit implicitement l'identifiant de la vue à utiliser. Ce mécanisme se fonde sur une implémentation de l'interface <code>RequestToViewNameTranslator</code> . L'implémentation par défaut extrait cet identifiant en enlevant les préfixe et suffixe de l'URI. Par exemple, pour un URI <code>/showTodos.do</code> , l'identifiant de la vue est <code>showTodos</code> .
N'importe quel type annoté par <code>ModelAttribute</code>	Objet à ajouter aux données du modèle après l'exécution de la méthode et avant celle de la vue. L'identifiant utilisé dans l'ajout correspond à celui de l'annotation.

Comme le montrent ces tableaux, deux annotations sont proposées pour le traitement des requêtes, `RequestParam` et `ModelMap`. Nous décrivons dans cette section l'utilisation de la première et détaillerons la seconde à la section « Contrôleur de gestion de formulaire ».

L'annotation `RequestParam` offre la possibilité de référencer un paramètre de la requête par son nom. L'objet correspondant est alors passé en tant que paramètre. À ce niveau, une conversion de type est réalisée si nécessaire afin de coller avec le type attendu pour le paramètre.

Le code suivant illustre l'utilisation de l'annotation `RequestParam` afin d'avoir accès au paramètre de la requête d'identifiant `listId` par l'intermédiaire d'un paramètre d'une méthode de traitement du contrôleur :

```
@Controller
public class ShowTodosController {
    (...)
    @RequestMapping("/showTodos.do")
    public ModelAndView showTodos(
        @RequestParam String listId) throws Exception { ← ❶

        Collection<TodoList> todoLists = new TreeSet<TodoList>(
            userManager.getCurrentUser().getTodoLists());

        if (!todoLists.isEmpty()) {
            if (listId != null) {
                listId = todoLists.iterator().next().getListId();
            }
        }
    }
}
```

```
        Map<String, Object> model = new HashMap<String, Object>();
        model.put("defaultList", listId);
        return new ModelAndView("todos", model);
    }
}
```

Lors de l'omission de la valeur de l'annotation `RequestParam`, le nom du paramètre sur lequel elle porte est utilisé. Ainsi, l'utilisation de l'annotation dans l'exemple précédent est similaire à la suivante (❶) :

```
@Controller
public class ShowTodosController {
    (...)
    @RequestMapping("/showTodos.do")
    public ModelAndView showTodos(
        @RequestParam("listId") String listId) ←❶
        throws Exception {
        (...)
    }
}
```

Par défaut, l'utilisation de l'annotation `RequestParam` nécessite la présence du paramètre dans la requête. L'attribut `required` de l'annotation permet de paramétrer ce comportement afin de rendre le paramètre optionnel, comme le montre le code suivant (❶) :

```
@RequestMapping("/showTodos.do")
public ModelAndView showTodos(
    @RequestParam(value="listId",
        required="false") String listId) ←❶
    throws Exception {
    (...)
}
```

Les méthodes de traitement des contrôleurs acceptent également un paramètre de type `ModelMap`, ce paramètre correspondant aux données du modèle. En utilisant ce paramètre, il est possible de manipuler les données du modèle et d'en ajouter de nouvelles. Dans ce cas, il n'est plus nécessaire de retourner un objet de type `ModelAndView` ; une chaîne de caractères correspondant à l'identifiant de la vue suffit.

Le code suivant illustre l'adaptation de l'exemple précédent (❶) afin d'utiliser ce mécanisme :

```
@Controller
public class ShowTodosController {
    (...)
    @RequestMapping("/showTodos.do")
    public String showTodos(
        @RequestParam String listId,
        ModelMap model) ←❶

        Collection<TodoList> todoLists = new TreeSet<TodoList>(
            userManager.getCurrentUser().getTodoLists());
}
```

```
        if (!todoLists.isEmpty()) {
            if (listId != null) {
                listId = todoLists.iterator().next().getListId();
            }
        }

        model.addAttribute("defaultList", listId); ← ❶
        return "todos"; ← ❶
    }
}
```

Les principaux autres types de paramètres et de retour sont décrits dans les sections suivantes.

### Contrôleur de gestion de formulaire

Spring MVC fournit un support pour l’affichage des données des formulaires et leur soumission à l’aide d’annotations. Ce support se fonde sur les différents concepts et annotations décrits aux sections précédentes.

Bien que ce type de contrôleur utilise un Bean afin de stocker les informations des formulaires, aucune configuration n’est à réaliser pour l’injection de dépendances. Il suffit que ce Bean soit présent dans les données du modèle et que l’identifiant correspondant soit spécifié dans le formulaire.

La section suivante se penche sur la façon d’implémenter la gestion des formulaires HTML au moyen de l’approche orientée annotations de Spring MVC.

### Affichage du formulaire

L’utilisation des annotations `RequestMapping`, `ModelAttribute` et `InitBinding` permet de charger les différentes entités nécessaires à l’affichage du formulaire dans la vue. Les méthodes sur lesquelles sont appliquées ces annotations, prennent alors part au cycle de traitement de la requête et adressent des problématiques distinctes et s’enchaînent dans un ordre bien précis.

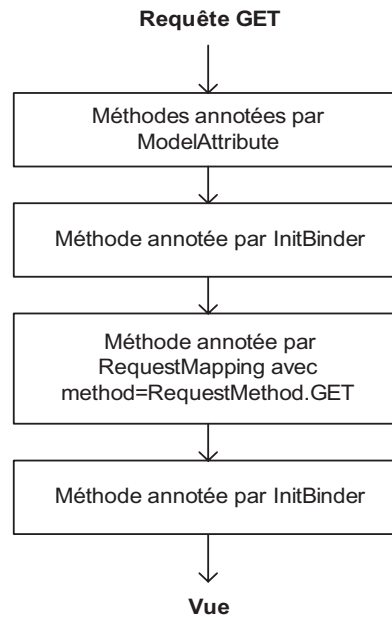
L’affichage du formulaire est réalisé grâce à l’appel d’une méthode de traitement de la requête par le biais de la méthode GET. Le cycle d’enchaînement des méthodes est illustré à la figure 7-4.

Spring MVC permet d’initialiser l’objet de formulaire en se fondant sur une méthode annotée par `ModelAttribute`. Cet objet doit être retourné par la méthode et est automatiquement ajouté dans le modèle. Il peut donc être utilisé par la suite dans la vue pour initialiser le formulaire correspondant.

Le comportement habituel consiste à créer une instance vierge à chaque demande d’affichage du formulaire si aucun paramètre n’est spécifié. Si un paramètre est présent dans la requête, celui-ci peut être alors utilisé, par exemple, afin de récupérer une instance initialisée avec des valeurs de la base de données.

Figure 7-4

*Enchaînement des méthodes permettant l'affichage des données d'un formulaire*



Le code suivant, tiré de Tudu Lists, donne un exemple d'utilisation de la méthode `initFormObject` (❶) de ce type :

```
(...)  
public class MyInfoController {  
    (...)  
    @ModelAttribute("userinfo")  
    public UserInfoData initFormObject(❶  
        HttpServletRequest request) {  
        String login = request.getRemoteUser();  
        User user = userManager.findUser(login);  
        UserInfoData data = new UserInfoData();  
        data.setPassword(user.getPassword());  
        data.setVerifyPassword(user.getPassword());  
        data.setFirstName(user.getFirstName());  
        data.setLastName(user.getLastName());  
        data.setEmail(user.getEmail());  
        return data;  
    }  
    (...)  
}
```

Les `PropertyEditor` personnalisés sont ajoutés par l'intermédiaire d'une méthode annotée par `InitBinder` afin de convertir les propriétés du Bean de formulaire en chaînes de caractères affichables dans des champs. Cette méthode doit posséder un paramètre de type `WebDataBinder` afin de pouvoir les enregistrer.

Le code suivant indique la façon d'ajouter un `PropertyEditor` dans un contrôleur de gestion de formulaire en se fondant sur l'annotation `InitBinder` (❶) :

```
(...)  
public class MyInfoController {  
    (...)  
    @InitBinder  
    public void initBinder(WebDataBinder binder) {←❶  
        binder.registerCustomEditor(MyClass.class,  
                                   new MyPropertyEditor());  
    }  
    (...)  
}
```

Cette méthode est utilisée afin d'afficher les valeurs du Bean de formulaire dans la vue correspondante sous forme de chaînes de caractères.

Il est possible d'ajouter des éléments dans le modèle par l'intermédiaire de l'annotation `ModelAttribute`. Ces éléments peuvent être utilisés dans la construction du formulaire afin notamment d'initialiser des listes de sélection, des boutons radio ou des cases à cocher. Autant de méthodes que d'éléments à ajouter doivent être définies.

Le code suivant montre la façon d'ajouter les données nécessaires afin d'initialiser un formulaire en se fondant sur l'annotation `ModelAttribute` (❶) :

```
(...)  
public class MyInfoController {  
    (...)  
    @ModelAttribute("datas")  
    public List<String> populateDataList() {←❶  
        List<String> datas = new ArrayList<String>();  
        datas.add("my data");  
        return datas;  
    }  
    (...)  
}
```

Pour finir, il convient de définir une méthode de traitement dédiée à l'affichage du formulaire. Cette dernière doit être annotée avec `RequestMapping` et posséder la propriété `method` avec la valeur `RequestMethod.GET`. Le mappage avec l'URI peut être spécifié à ce niveau ou globalement au niveau de la classe. Cette méthode ne possède pas particulièrement de traitements, mais spécifie la vue correspondant au formulaire.

Le code suivant illustre un exemple de méthode de ce type annoté par `RequestMapping` (❶) :

```
(...)  
public class MyInfoController {  
    (...)  
    @RequestMapping(method = RequestMethod.GET)  
    public String showForm() {←❶  
        return "userinfo";  
    }  
    (...)  
}
```

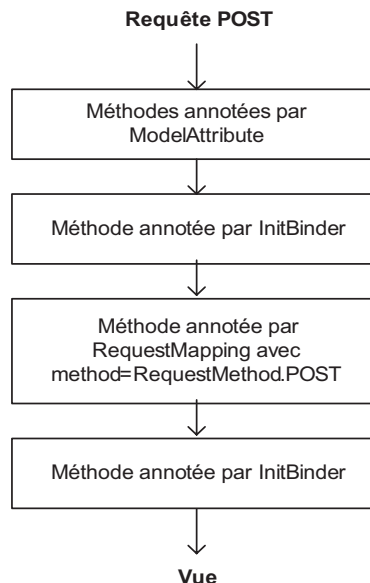
## Soumission du formulaire

L'utilisation des annotations `RequestMapping`, `ModelAttribute` et `InitBinding` permet de définir des méthodes de remplissage de l'objet de formulaire avec les données soumises et de les traiter. Ces méthodes adressent des problématiques distinctes et s'enchaînent dans un ordre précis.

La soumission du formulaire est traitée grâce à l'appel d'une méthode de traitement de la requête par la méthode `POST`. Le cycle d'enchaînement des méthodes est illustré à la figure 7-5.

Figure 7-5

*Enchaînement des méthodes permettant la soumission des données d'un formulaire*



Les premières méthodes annotées avec `RequestMapping` et `InitBinder` (respectivement `initFormObject`, `initBinder`) fonctionnent de la même manière que précédemment

Une validation des données d'un formulaire peut être mise en œuvre si nécessaire. La validation liée au mappage des données du formulaire dans l'objet correspondant est directement intégrée dans le cycle de traitements. Par contre, avec l'approche fondée sur les annotations, Spring MVC n'intègre pas les validations personnalisées dans ce cycle. Néanmoins, il est recommandé d'utiliser l'interface `Validator` afin de regrouper ces traitements. Le code de cette interface est le suivant :

```
public interface Validator {  
    boolean supports(Class clazz);  
    void validate(Object obj, Errors errors);  
}
```

La méthode `supports` permet de spécifier sur quel Bean de formulaire peut être appliquée l'entité de validation. La méthode `validate` doit contenir l'implémentation de la validation et utiliser l'instance de l'interface `Errors` associée.

Le ou les validateurs sont associés au contrôleur soit par injection de dépendances, soit par une instantiation directe, cette dernière étant peu coûteuse.

Le code suivant de la classe `RegisterValidator` montre que l'implémentation du validateur permet de spécifier des erreurs aussi bien à un niveau global (❶) que sur chaque propriété du formulaire (❷) en s'appuyant sur l'interface `Errors` :

```
public class RegisterValidator implements Validator {

    public boolean supports(Class clazz) {
        return RegisterData.class.isAssignableFrom(clazz);
    }

    public void validate(Object command, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "login",
            "errors.required", new Object[] {"login"}, "");←❷
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "password",
            "errors.required", new Object[] {"password"}, "");←❷

        ValidationUtils.rejectIfEmptyOrWhitespace(
            errors, "verifyPassword",
            "errors.required", new Object[] {"verifyPassword"}, "");←❷
        if( !data.getPassword().equals(data.getVerifyPassword()) ) {
            errors.rejectValue("verifyPassword", "errors.required",
                new Object[] {"verifyPassword"}, "");←❷
        }

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName",
            "errors.required", new Object[] {"firstName"}, "");←❷
        ValidationUtils.rejectIfEmptyOrWhitespace(
            errors, "lastName",
            "errors.required", new Object[] {"lastName"}, "");←❷

        if( errors.hasErrors() ) {
            errors.reject("register.info.1");←❶
        }
    }
}
```

La méthode de traitement dédiée à la soumission du formulaire doit être annotée avec `RequestMapping` et posséder la propriété `method` avec la valeur `RequestMethod.POST`. Cette dernière prend en paramètre l'objet de formulaire, objet annoté par `ModelAttribute` et a la responsabilité de traiter cet objet.

Le code suivant illustre la mise en œuvre d'une méthode de ce type dans le cadre du contrôleur `MyInfoController`, méthode nommée `submitForm` (❶) :

```
(...)
public class MyInfoController {
    (...)

    @RequestMapping(method = RequestMethod.POST)
```

```
public String submitForm(←❶
    @ModelAttribute("userinfo") UserInfoData userInfo,
    BindingResult result) {

    User user = userManager.findUser(userInfo.getLogin());
    user.setPassword(userInfo.getPassword());
    user.setFirstName(userInfo.getFirstName());
    user.setLastName(userInfo.getLastName());
    user.setEmail(userInfo.getEmail());
    userManager.updateUser(user);

    return "userinfo";
}
(...)
```

Si une validation est mise en œuvre, cette méthode a en charge la spécification de la vue d'affichage du formulaire en cas d'échec de validation. À cet effet, elle doit prendre un paramètre de type `BindingResult` correspondant au résultat du mappage. Ce paramètre pourra être passé à la méthode `validate` du validateur.

Le code suivant illustre l'intégration d'un validateur (❶) dans les traitements de soumission du contrôleur :

```
(...)
public class MyInfoController {
    (...)

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
        @ModelAttribute("userinfo") UserInfoData userInfo,
        BindingResult result) {

        (new RegisterValidator()).validate(userInfo, result);←❶
        if (!result.hasErrors()) {←❶
            User user = userManager.findUser(userInfo.getLogin());
            user.setPassword(userInfo.getPassword());
            user.setFirstName(userInfo.getFirstName());
            user.setLastName(userInfo.getLastName());
            user.setEmail(userInfo.getEmail());
            userManager.updateUser(user);
        }

        return "userinfo";
    }
    (...)
}
```

Lors de l'utilisation d'une vue fondée sur JSP/JSTL, les balises du taglib `form` de Spring offrent un support à l'affichage des données du formulaire ou des erreurs de validation. Son utilisation est détaillée plus loin dans ce chapitre.

### Support des formulaires sur plusieurs pages

Spring MVC supporte la mise en œuvre d'un formulaire sur plusieurs pages, la session Web devant dans ce cas être utilisée. Le framework offre la possibilité de gérer implicitement le stockage de l'objet de formulaire à ce niveau.

Pour ce faire, il convient de préciser que l'objet de formulaire est stocké en session par l'intermédiaire de l'annotation `SessionAttributes` au niveau de la classe du contrôleur. Cette annotation permet de spécifier l'identifiant correspondant.

Pour libérer les ressources associées lors d'un succès de la soumission du formulaire sur la dernière page, il convient d'utiliser la méthode `setComplete` sur un objet de type `SessionStatus`. Un paramètre de ce type peut être passé directement en tant que paramètre de méthodes de traitement de requêtes dans les contrôleurs.

Le code suivant est un exemple simple d'utilisation de cette approche, à savoir la configuration de l'objet de formulaire pour un stockage en session (❶), le passage d'un paramètre de type `SessionStatus` (❷) et l'utilisation de la méthode `setComplete` (❸) sur cet objet :

```
(...)  
@SessionAttributes("userinfo") ← ❶  
public class MyInfoController {  
    (...)  
  
    @RequestMapping(method = RequestMethod.POST)  
    public String submitForm(  
        @ModelAttribute("userinfo") UserInfoData userInfo,  
        BindingResult result,  
        SessionStatus status) ← ❷ {  
        (new RegisterValidator()).validate(userInfo, result);  
        if (!result.hasErrors()) {  
            status.setComplete(); ← ❸  
            (...)  
        }  
        (...)  
    }  
}
```

## Gestion des exceptions

Par défaut, Spring MVC fait remonter les différentes exceptions levées dans le conteneur de servlets. Il est cependant possible de modifier ce comportement par l'intermédiaire de l'interface `HandlerExceptionResolver`, localisée dans le package `org.springframework.web.servlet` :

```
public interface HandlerExceptionResolver {  
    ModelAndView resolveException(HttpServletRequest request,  
        HttpServletResponse response, Object handler, Exception ex);  
}
```

Le développeur peut choisir d'utiliser ses propres implémentations ou la classe `SimpleMappingExceptionHandler` du package `org.springframework.web.servlet.handler` fournie par le framework. Cette dernière permet de configurer les exceptions à traiter ainsi que les vues qui leur sont associées. L'exception est alors stockée dans la requête avec la clé `exception`, ce qui la rend disponible pour un éventuel affichage.

Cette implémentation se paramètre de la manière suivante :

```
<bean id="exceptionResolver" class="org.springframework.web
    .servlet.handler.SimpleMappingExceptionResolver">
  <property name="exceptionMappings">
    <props>
      <prop key="org.springframework.dao.DataAccessException">
        dataAccessFailure
      </prop>
      <prop key="org.springframework.transaction
        .TransactionException">
        dataAccessFailure
      </prop>
    </props>
  </property>
</bean>
```

### En résumé

Spring MVC met en œuvre une approche intéressante pour les contrôleurs MVC fondée sur les annotations. Elle offre ainsi la possibilité de s'abstraire de l'API Servlet en laissant le framework réaliser cette manipulation. Les signatures des points d'entrée des contrôleurs peuvent être adaptées en fonction des besoins et de l'approche souhaitée.

Au-delà du cadre général proposé par le framework, plusieurs déclinaisons sont possibles, comme l'utilisation de contrôleurs simples ou de formulaire pour la récupération des paramètres de la requête, le remplissage du modèle et la sélection de la vue.

## Spring MVC et la gestion de la vue

Cette section se penche sur la façon dont sont traitées les vues au sein du framework Spring MVC.

### Sélection de la vue et remplissage du modèle

Spring MVC abstrait complètement la vue du contrôleur, masquant ainsi sa technologie et sa mise en œuvre. Au niveau du contrôleur, le développeur a la responsabilité de remplir le modèle avec les instances utilisées dans la vue et de spécifier son identifiant.

Différentes approches sont possibles pour cela, qui visent toutes à faciliter l'utilisation de Spring et la mise en œuvre des contrôleurs.

La première se fonde sur la classe `ModelAndView` dans le package `org.springframework.web.servlet`. Les données véhiculées par cette classe sont utilisées afin de sélectionner la vue, la classe lui fournissant les données du modèle. De ce fait, le développeur ne manipule plus l'API Servlet pour remplir le modèle et passer la main aux traitements de la vue.

Cette classe doit être utilisée en tant que retour d'une méthode de traitement de requêtes annotée avec `RequestMapping`. Une instance de la classe `ModelAndView` doit alors être instanciée et remplie à ce niveau.

Les données du modèle sont stockées sous forme de table de hachage. Le code suivant donne un exemple de mise en œuvre de ce mécanisme (❶), dans lequel l'identifiant `todos` correspond à un nom symbolique de vue configuré dans Spring MVC :

```
@RequestMapping("/showTodos.do")
public ModelAndView showTodos(HttpServletRequest request, ←❶
                             HttpServletResponse response) throws Exception {

    String listId = (...);

    Map<String, Object> model = new HashMap<String, Object>();
    model.put("defaultList", listId);
    return new ModelAndView("todos", model); ←❶
}
```

La seconde approche consiste en l'utilisation de la classe `Map` ou `Model` ou `ModelMap` en tant que paramètre d'une méthode de traitement annotée avec `RequestMapping`. Dans ce cas, ce paramètre correspond à l'entité de stockage des éléments du modèle. L'identifiant de la vue et ces données sont désormais dissociées.

Si aucun identifiant de vue n'est précisé, Spring MVC le déduit de l'URI. Par exemple, si la vue se finit par `/showTodos.do`, l'identifiant déduit est `showTodos`. Il est néanmoins possible de spécifier explicitement l'identifiant de la vue choisie en le faisant retourner sous forme de chaîne de caractères par la méthode.

Le code suivant illustre l'utilisation de la classe `ModelMap` pour gérer les données du modèle, ainsi que la manière de spécifier implicitement (❶) et explicitement (❷) l'identifiant de la vue :

```
@RequestMapping("/welcome.do")
public void welcome(ModelMap model) throws Exception { ←❶
    (...)
    //L'identifiant de la vue est déduit
    //et correspond à welcome
}

@RequestMapping("/showTodos.do")
public String showTodos(ModelMap model) throws Exception { ←❷
    String listId = (...);
    model.addAttribute("defaultList", listId);
    //L'identifiant de la vue est retourné
    return "todos";
}
```

En parallèle des méthodes de traitement des requêtes, il est possible d'ajouter d'autres éléments dans le modèle en utilisant le retour des méthodes annotées par `ModelAttribute`. Dans ce cas, le retour est automatiquement ajouté au modèle, avant même que la méthode de traitement de la requête soit appelée.

Le code suivant illustre l'utilisation de l'annotation `ModelAttribute` (❶) afin d'ajouter le retour d'une méthode dans le modèle et la vérification de la présence de cet objet (❷) dans la méthode de traitement d'une requête :

```
@ModelAttribute("user")←❶
public User getCurrentUser() {
    return userManager.getCurrentUser();
}

@RequestMapping("/showTodos.do")
public String showTodos(ModelMap model) throws Exception {
    User user = (User)model.get("user");←❷
    (...)
    return "todos";
}
```

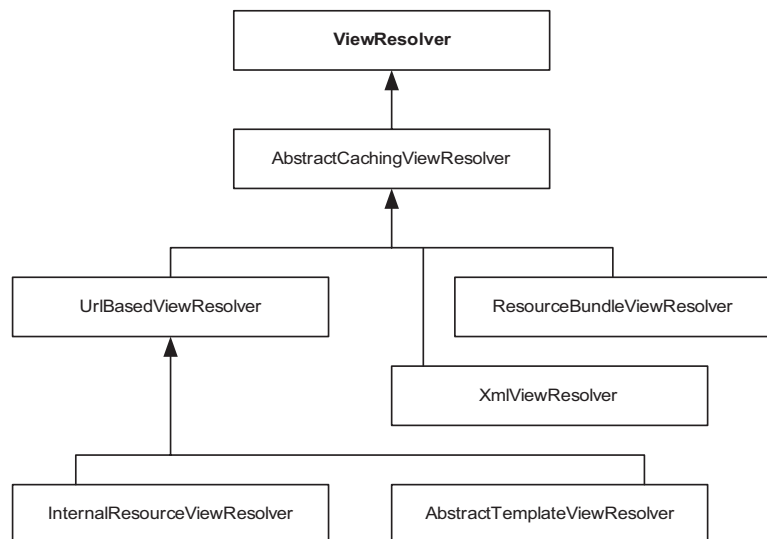
## Configuration de la vue

La sélection des vues dans Spring MVC est effectuée par le biais d'une implémentation de l'interface `ViewResolver` dans le package `org.springframework.web.servlet`, comme le montre le code suivant :

```
public interface ViewResolver {
    View resolveViewName(String viewName, Locale locale);
}
```

Les sections qui suivent détaillent les différentes implémentations de cette interface. La figure 7-6 illustre la hiérarchie de ces classes et interfaces.

**Figure 7-6**  
*Hiérarchie des implémentations de l'interface `ViewResolver`*



### *ResourceBundleViewResolver*

La première implémentation, `ResourceBundleViewResolver`, correspond à une configuration au cas par cas des vues utilisées. Cette approche est particulièrement intéressante pour une utilisation des vues fondées sur différentes technologies de présentation. Sa configuration s'effectue par le biais d'un fichier de propriétés contenant le paramétrage des différentes vues.

Cette classe peut toutefois devenir vite contraignante si la majeure partie des vues utilise la même technologie de présentation. Les applications qui utilisent JSP/JSTL et des vues PDF ou Excel pour afficher des états sont un exemple de cette contrainte. Spring MVC offre une solution fondée sur le chaînage de `ViewResolver` pour optimiser la résolution de ce problème.

Le code suivant montre de quelle manière configurer cette implémentation avec Spring MVC :

```
<bean id="viewResolver" class="org.springframework
    .web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views"/>
</bean>
```

La propriété `basename` permet de spécifier le fichier de propriétés utilisé, qui, dans l'exemple suivant, a pour nom `views.properties` :

```
register_ok.class=org.springframework.web.servlet.view.RedirectView
register_ok.url=welcome.action

recover_password_ok.class
    =org.springframework.web.servlet.view.RedirectView
recover_password_ok.url=welcome.action

todo_lists_report.class=tudu.web.ShowTodoListsPdfView

rssFeed.class=tudu.web.RssFeedView
rssFeed.stylesheetLocation=/WEB-INF/xsl/rss.xsl
```

Ce fichier possède les configurations des vues de redirection ainsi que des vues fondées sur la technologie XSLT et le framework `iText`.

### *XmlViewResolver*

Les vues sont définies par l'intermédiaire de cette implémentation au cas par cas, comme précédemment, mais dans un sous-contexte de Spring. L'utilisation de toutes les fonctionnalités et mécanismes du framework est donc envisageable, de même que l'injection de dépendances sur la classe d'implémentation des vues.

La configuration de cette implémentation se réalise de la manière suivante en utilisant par défaut le fichier `/WEB-INF/views.xml`, tout en n'écartant pas la possibilité d'en spécifier un autre par le biais de la propriété `location` :

```
<bean id="viewResolver"
```

```
        class="org.springframework.web.servlet.view.XmlViewResolver">
        <property name="order" value="2" />
        <property name="location" value="/WEB-INF/views.xml" />
    </bean>
```

### ***InternalResourceViewResolver***

L'implémentation `InternalResourceViewResolver` utilise les URI dans le but de résoudre les vues fondées, par exemple, sur les technologies JSP/JSTL. Ce mécanisme construit l'URI à partir de l'identifiant de la vue puis dirige les traitements vers d'autres ressources gérées par le conteneur de servlets, telles que des servlets ou des JSP, comme dans l'exemple ci-dessous :

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

Cette implémentation générale s'applique à toutes les vues, excepté celles qui sont résolues précédemment par une autre implémentation dans une chaîne de `ViewResolver`.

### **Chaînage d'implémentations de *ViewResolver***

Spring MVC offre la possibilité de chaîner les entités de résolution des vues. Le framework parcourt dans ce cas la chaîne jusqu'à la découverte du `ViewResolver` approprié.

Certaines de ces entités s'appliquant à toutes les vues, une stratégie par défaut de résolution des vues peut être définie. Les implémentations fondées sur `UriBasedViewResolver`, telles que `InternalResourceViewResolver`, fonctionnent sur ce principe.

L'utilisation des vues fondées sur JSP/JSTL peut être spécifiée. D'autres vues, comme des redirections ou des vues générant des flux PDF ou Excel, sont définies ponctuellement dans un fichier.

La figure 7-7 illustre un chaînage d'implémentations de l'interface de `ViewResolver` tiré de *Tudu Lists*.

Sans cette fonctionnalité, la configuration de toutes les vues au cas par cas dans un fichier serait nécessaire, même pour celles ne nécessitant pas de paramétrage spécifique.

L'exemple suivant décrit la configuration du chaînage de `ViewResolver` :

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

```

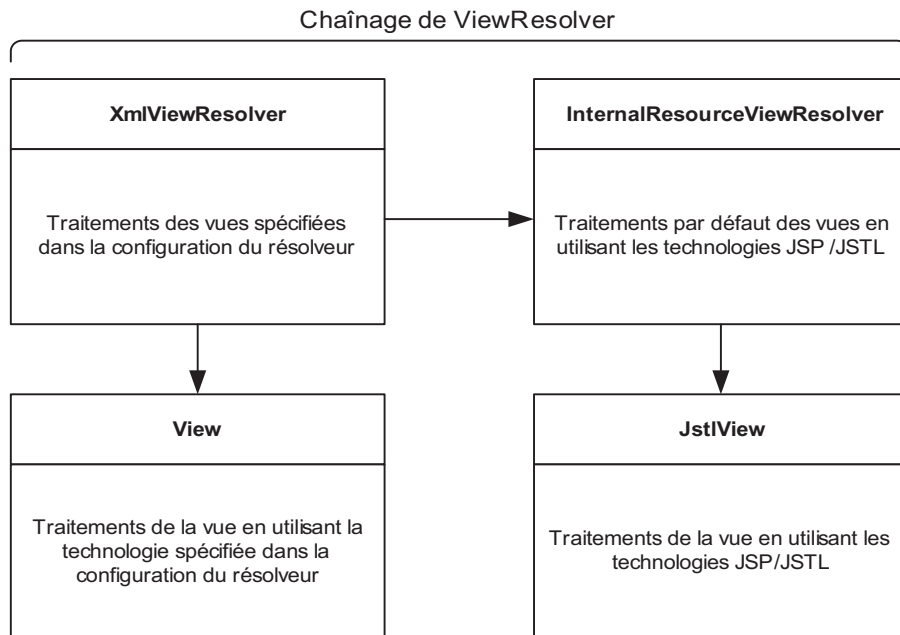
<bean id="viewResolver"
      class="org.springframework.web.servlet.view.XmlViewResolver">
  <property name="order" value="1"/>
  <property name="location" value="/WEB-INF/views.xml"/>
</bean>

```

La propriété `order` permet de spécifier la position du `ViewResolver` dans la chaîne. Cet exemple met en évidence que la classe `InternalResourceViewResolver` ne possède pas cette propriété, ce `ViewResolver` ne pouvant être utilisé qu'en fin de chaîne.

**Figure 7-7**

*Chaînage de  
ViewResolver  
dans Tudu Lists*



## Les technologies de présentation

Spring MVC propose plusieurs fonctionnalités qui simplifient énormément la mise en œuvre des technologies et frameworks de présentation.

Dans Spring MVC, une vue correspond à une implémentation de l'interface `View` du package `org.springframework.web.servlet` telle que décrite dans le code suivant :

```

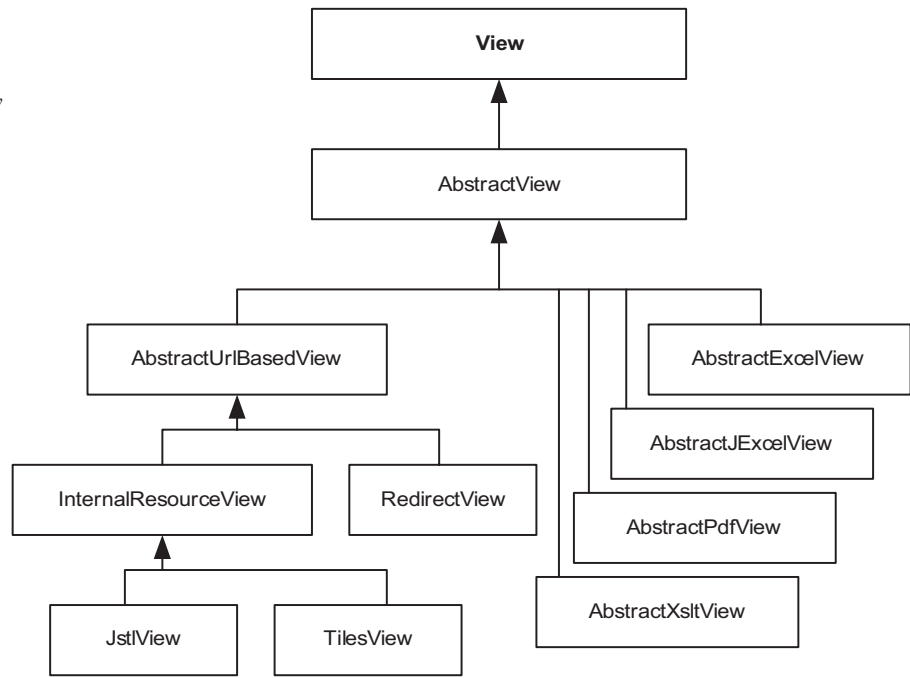
public interface View {
    void render(Map model, HttpServletRequest request,
               HttpServletResponse response);
}

```

Cette interface possède plusieurs implémentations, localisées dans le package `org.springframework.web.servlet.view` ou dans un de ses sous-packages.

La figure 7-8 illustre la hiérarchie de ses classes et interfaces.

**Figure 7-8**  
Hiérarchie des implémentations de l'interface *View*



### Vue de redirection

Spring MVC définit un type de vue particulier afin de rediriger les traitements vers un URI ou une URL par l'intermédiaire de la classe *RedirectView*. Elle se configure avec l'implémentation *ResourceBundleViewResolver* ou *XmlViewResolver* en imposant de définir la propriété *url*.

Le code suivant donne un exemple de sa mise en œuvre dans *Tudu Lists* :

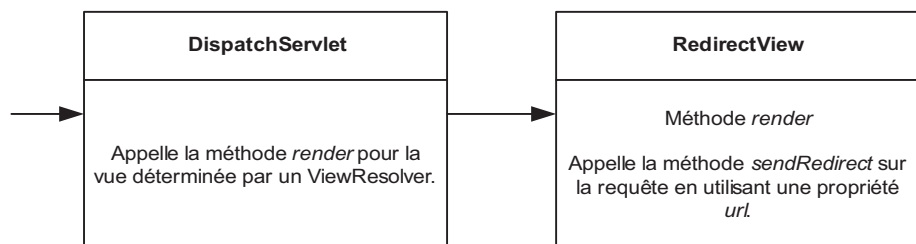
```

register_ok.class=org.springframework.web.servlet.view.RedirectView
register_ok.url=welcome.action
  
```

La propriété *url* permet de spécifier l'adresse de redirection correspondante, laquelle est, dans notre cas, relative au contexte de l'application.

La figure 7-9 illustre l'enchaînement des traitements afin d'utiliser une vue de type *RedirectView*.

**Figure 7-9**  
Enchaînement des traitements pour la vue



Cette vue peut être configurée plus rapidement et directement dans la configuration des contrôleurs grâce au préfixe `redirect` (❶), comme l'illustre le code suivant :

```
public class RestoreTodoListController {
    (...)

    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
        @ModelAttribute("restoredata") RestoreData restoreData,
        BindingResult result,
        SessionStatus status) {
        (...)
        return "redirect:showTodos.action"; ←❶
    }
}
```

### Vue fondée sur JSP/JSTL

Spring MVC fournit une vue fondée sur JSP/JSTL, dirigeant les traitements de la requête vers une page JSP dont l'URI est construit à partir de l'identifiant de la vue.

La figure 7-10 illustre l'enchaînement des traitements afin d'utiliser une vue de type `JstlView`.

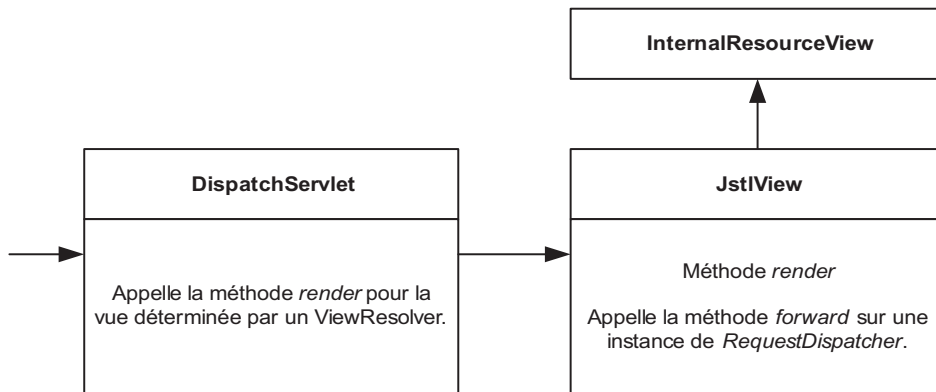


Figure 7-10

Enchaînement des traitements pour la vue

Le développeur prend uniquement en charge le développement de la page JSP, tandis que Spring MVC a la responsabilité de mettre à disposition dans la requête tout le contenu du modèle ainsi qu'éventuellement des informations concernant le formulaire.

Les balises et expressions JSTL peuvent être utilisées d'une manière classique en utilisant les données du modèle, ces dernières étant mises à disposition par Spring MVC pour les pages JSP. Pour une entrée ayant pour clé `maVariable` dans le modèle, la page JSP récupère la valeur de sa propriété `maPropriete` correspondante de la manière suivante :

```
<c:out value="${maVariable.maPropriete}"/>
```

Afin d'utiliser les taglibs JSTL, des importations doivent être placées dans les pages JSP, comme dans le code suivant, tiré de la page **WEB-INF/jspf/header.jsp** :

```
<%@ taglib prefix="c" uri="http://java.sun.com/jstl/core_rt" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jstl/fmt_rt" %>
```

Ainsi, l'affichage de la liste des todos dans une page JSP se réalise de la manière suivante (cette liste ayant été spécifiée dans le modèle) :

```
Affichage de la liste des todos:
<c:forEach items="${todos}" var="todo">
- <c:out value="${todo.id}"/>, <c:out value="${todo.name}"/><br/>
</c:forEach>
```

Au niveau des formulaires, un taglib dédié permet de réaliser le mappage entre les données du formulaire et les champs correspondants. Pour pouvoir l'utiliser, l'importation suivante (tirée de la page **WEB-INF/jspf/header.jsp**) doit être placée dans les pages JSP :

```
<%@ taglib prefix="form"
uri="http://www.springframework.org/tags/form" %>
```

Le tableau 7-4 récapitule les balises proposées par ce taglib pour construire des formulaires.

**Tableau 7-4. Balises du taglib *form* de gestion de formulaires**

Balise	Description
checkbox	Définit un élément de formulaire de type case à cocher pour un attribut du Bean de formulaire.
checkboxes	Définit un ensemble d'éléments de formulaire de type case à cocher pour un attribut de formulaire. L'initialisation des valeurs des éléments se réalise à partir d'une liste présente dans le modèle.
errors	Affiche les erreurs survenues lors de la soumission d'un formulaire à un niveau global ou par champ.
form	Définit un formulaire et le rattache éventuellement à un Bean de formulaire.
hidden	Définit un élément de formulaire caché pour un attribut du Bean de formulaire.
input	Définit un élément de formulaire de type texte pour un attribut du Bean de formulaire.
option	Définit un élément de liste.
options	Définit un ensemble d'éléments de liste. L'initialisation des valeurs des éléments se réalise à partir d'une liste présente dans le modèle.
password	Définit un élément de formulaire de type mot de passe pour un attribut du Bean de formulaire.
radiobutton	Définit un élément de formulaire de type bouton radio pour un attribut du Bean de formulaire.
radiobuttons	Définit un ensemble d'éléments de formulaire de type bouton radio pour un attribut du Bean de formulaire. L'initialisation des valeurs des éléments se réalise à partir d'une liste présente dans le modèle.
select	Définit un élément de formulaire de type liste pour un attribut du Bean de formulaire. Cette balise doit être utilisée conjointement avec les balises <code>option</code> et <code>options</code> afin de spécifier les valeurs de la liste.
textarea	Définit un élément de formulaire de type zone de texte pour un attribut du Bean de formulaire.

Ces balises permettent de référencer les attributs d'un Bean de formulaire mis à disposition dans le modèle à l'aide des techniques décrites précédemment à la section « Contrôleur de gestion de formulaire ».

La correspondance entre le formulaire et le Bean de formulaire se réalise au niveau de la balise `form` par l'intermédiaire du champ `modelAttribute`, qui spécifie le nom de l'entrée dans le modèle.

L'utilisation des balises du taglib `form` permet d'initialiser automatiquement les champs du formulaire avec les données du formulaire et d'afficher les éventuelles erreurs survenues. Le code suivant montre l'utilisation de cette balise dans la page JSP **WEB-INF/jsp/user\_info.jsp** de l'étude de cas :

```
<form:form modelAttribute="userinfo">←①
  <tr class="odd">
    <td>
      <fmt:message key="user.info.first.name"/>
    </td>
    <td>
      <form:input path="firstName" size="15" maxLength="60"/>←②
      &#160;<font color="red">
        <form:errors path="firstName"/></font>←③
      </td>
    </tr>
    (...)
  </form:form>
```

La balise `form` permet de définir l'identifiant de l'élément dans le modèle correspondant au Bean de formulaire en se fondant sur l'attribut `modelAttribute` (①). Elle permet également de délimiter les éléments du formulaire.

Au sein du formulaire, les champs correspondant aux attributs du Bean de formulaire peuvent être spécifiés. La correspondance entre le champ et l'attribut correspondant se réalise par l'intermédiaire de l'attribut `path` de la balise. Dans notre exemple, la balise `input` (②) porte sur la propriété `firstName` du Bean de formulaire ayant pour nom `userinfo`.

L'affichage des erreurs, générales ou associées à un champ, se réalise par l'intermédiaire de la balise `errors`. L'attribut `path` permet de préciser les erreurs à afficher. Avec une valeur `*`, toutes les erreurs relatives au formulaire sont affichées. Avec le nom d'un champ, comme dans l'exemple précédent (③), l'éventuelle erreur correspondant au champ est affichée.

### Autres vues

Spring MVC apporte des supports pour toutes sortes de vues qui ne sont pas nécessairement fondées sur des redirections internes au conteneur de servlets (méthode `forward`) ou des redirections de requêtes (méthode `sendRedirect`), notamment des classes abstraites de base pour les technologies suivantes :

- génération de documents (PDF, Excel, etc.) ;
- génération de rapports avec Jasper Reports ;

- génération de présentations fondées sur des templates (Velocity, FreeMarker) ;
- génération de présentations fondées sur les technologies XML.

Concernant les vues générant des documents et celles fondées sur les technologies XML, le framework Spring MVC instancie les ressources représentant le document par le biais de méthodes génériques. Il délègue ensuite les traitements à une méthode de la vue afin de construire le document ou de convertir le modèle dans une technologie donnée. Le framework reprend ensuite en main ces traitements afin d'exécuter éventuellement une transformation puis d'écrire le résultat sur le flux de sortie.

La figure 7-11 illustre l'enchaînement des traitements d'une vue générant un document avec le support PDF de Spring MVC.

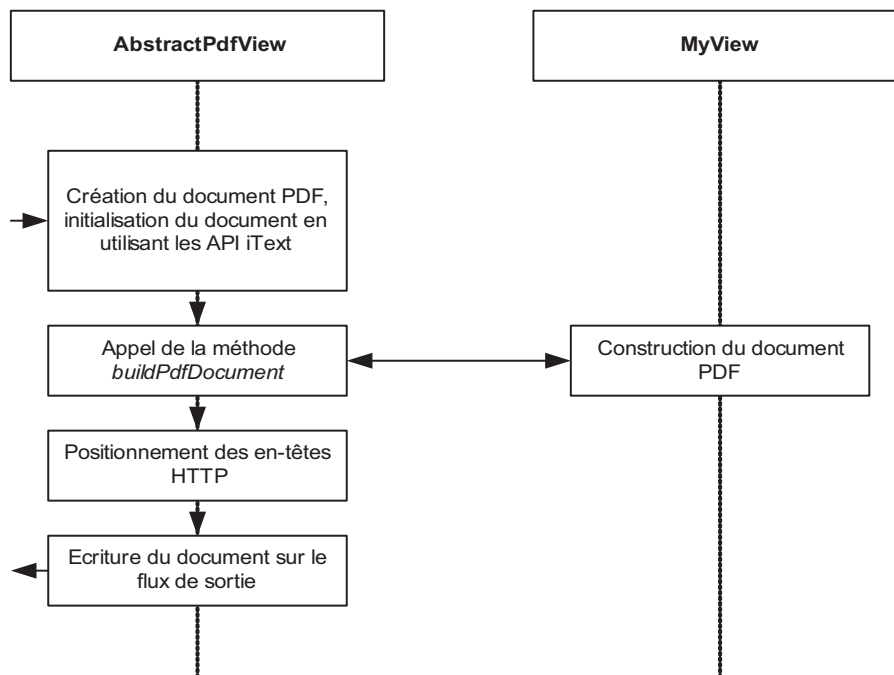


Figure 7-11

*Enchaînement des traitements de la vue*

Dans l'exemple décrit à la figure 7-11, la classe `MyView` étend la classe abstraite `AbstractPdfView` du package `org.springframework.web.servlet.view` afin d'implémenter la méthode abstraite `buildPdfDocument`. C'est pourquoi la classe `MyView` ne contient plus que les traitements spécifiques à l'application pour la construction du document, son instanciation et son renvoi au client étant encapsulés dans la classe `AbstractPdfView`.

## En résumé

Approfondissant les principes de gestion de la présentation au sein du framework Spring MVC, nous avons vu que ce dernier met en œuvre des mécanismes de mise en relation d'un identifiant et de sa vue, tout en favorisant la coexistence de vues fondées sur diverses technologies. Le framework permet l'utilisation d'un nombre important de technologies de présentation.

La mise à disposition des données présentes dans le modèle pour les vues est réalisée automatiquement par le framework.

## Support de REST (Representational State Transfer)

La version 3.0 de Spring introduit le support natif de la technologie REST afin d'utiliser les URI conformes à ce format tout en suivant le modèle de programmation de Spring MVC fondé sur les annotations.

### La technologie REST

Cette technologie correspond à une manière de construire des applications pour les systèmes distribués. REST n'est pas un protocole ou un format, mais correspond au style architectural original du Web, bâti sur les principes simples suivants :

- L'URI est très important puisque, dans ce contexte, connaître ce dernier doit suffire pour accéder à la ressource. Il n'est plus besoin de spécifier des paramètres supplémentaires. De plus, la manipulation de la ressource se fonde sur l'utilisation des opérations du protocole HTTP (GET, POST, PUT et DELETE, essentiellement).
- Chaque opération est autosuffisante, et aucun état n'est stocké au niveau de la ressource. Le client a la responsabilité du stockage de cet état. En parallèle, la technologie utilise des standards hypermédias tels que HTML ou XML afin de réaliser les liens vers d'autres ressources et d'assurer ainsi la navigation dans l'application.
- Il est possible de mettre en œuvre des architectures orientées services de manière simple en utilisant des services Web interapplicatifs. La technologie REST propose ainsi une solution de rechange intéressante et plus simple au mode RPC et, dans la plupart des cas, à SOAP.

Avec la technologie REST, les paramètres font partie intégrante de l'URI, comme l'illustre le code suivant avec un exemple d'URI contenant le paramètre `id` :

```
■ /<alias-webapp>/<ressource-name>/{id}
```

Spring 3.0 offre non seulement la possibilité d'écrire des contrôleurs Web REST, mais propose aussi un composant client permettant d'effectuer des requêtes REST, le `RestTemplate`.

## Contrôleur Web REST

Spring MVC offre la possibilité de passer en paramètres des méthodes des contrôleurs des valeurs en se fondant sur l'annotation `PathVariable`. Cette dernière fonctionne d'une manière

similaire à l'annotation `RequestParam` en référençant les paramètres présents dans l'adresse d'accès, comme l'illustre le code suivant (❶) :

```
@RequestMapping(value = "/todo/{format}/{todoId}")
public ModelAndView getTodo(@PathVariable String format, ←❶
                            @PathVariable String todoId) {
    (...)
}
```

Avec REST, il est désormais possible d'utiliser d'autres méthodes HTTP que celles utilisées habituellement dans les navigateurs par les applications Web classiques, à savoir les méthodes GET et POST. Dans ce contexte, l'appel d'une ressource réalisant la suppression d'une entité s'effectue avec la méthode DELETE. La méthode correspondante du contrôleur doit spécifier la méthode HTTP d'accès au niveau de l'annotation `RequestMapping`, comme dans le code suivant (❶) :

```
@RequestMapping(
    value = "/todo/{todoId}",
    method = RequestMethod.DELETE ←❶
)
public void deleteTodo(@PathVariable String todoId,
                      HttpServletResponse response) {
    todoManager.remove(todoId);
    // pas de rendu de vue
    response.setStatus(HttpServletResponse.SC_OK);
}
```

L'appel de cette méthode peut se réaliser dans une page JSP par le biais de la balise `form` du taglib `form` de Spring, comme dans le code suivant (❶) :

```
<form:form method="delete"> ←❶
    <input type="submit" value="Supprimer un todo"/></p>
</form:form>
```

En gardant le même modèle de programmation que celui de Spring MVC, le support REST de Spring permet de mettre en œuvre REST de manière intéressante. Tous les mécanismes de Spring MVC sont utilisables dans ce contexte, contribuant d'autant à la simplicité de mise en œuvre de cette technologie.

## Le RestTemplate

Le `RestTemplate` est la classe centrale du support client REST dans Spring. Il permet d'effectuer des requêtes HTTP selon les différentes méthodes du protocole, tout en facilitant la gestion des réponses, avec la possibilité de transformer directement le flux de réponses en objets Java.

Le `RestTemplate` fonctionne suivant la philosophie des templates d'accès aux données de Spring, dont les principes sont présentés au chapitre 10. Il peut néanmoins être utilisé directement, sans connaître ces principes sous-jacents, l'essentiel consistant à savoir que le

`RestTemplate` gère les connexions HTTP et laisse le développeur se concentrer sur le code applicatif.

Le `RestTemplate` peut être instancié de façon programmatique, mais il est plus judicieux de le déclarer dans le contexte Spring, afin de profiter de l'injection de dépendances, notamment pour de la configuration plus fine :

```
<bean id="restTemplate"
    class="org.springframework.web.client.RestTemplate">←①
    <property name="requestFactory">
        <bean class="org.springframework.http.client.
            CommonsClientHttpRequestFactory" />←②
    </property>
</bean>
```

Le `RestTemplate` est déclaré comme tout autre Bean, avec la classe correspondante (①). Pour effectuer les accès HTTP, le `RestTemplate` utilise une abstraction, la `ClientHttpRequestFactory`, que l'on positionne avec la propriété `requestFactory` (②). Nous utilisons dans notre exemple une implémentation fondée sur la bibliothèque Jakarta Commons `HttpClient`, qui offre des possibilités intéressantes de configuration (authentification, pool de connexions, etc.).

Une fois configuré, le `RestTemplate` peut être utilisé pour récupérer le résultat d'une requête HTTP, par exemple pour appeler la méthode `getTodo` du contrôleur REST de la section précédente :

```
String xml = restTemplate.getForObject(
    "http://localhost:8080/rest/todo/{format}/{todoId}", ←①
    String.class, ←②
    "xml", "1"); ←③
```

La méthode `getForObject` du `RestTemplate` prend en premier paramètre l'URL à appeler (①). On remarque l'utilisation de la syntaxe `{nomParametre}` pour indiquer les différents paramètres ; dans notre cas, le format souhaité de la réponse (XML, HTML ou PDF) et l'identifiant du `Todo`. Le deuxième paramètre de `getForObject` est la classe de retour attendue (②). Le `RestTemplate` est en effet capable de transformer le flux de la réponse en un objet Java.

Par défaut, les transformations sont limitées (chaînes de caractères ou tableau d'octets), mais nous verrons par la suite que ce mécanisme est paramétrable. Dans notre exemple, nous nous contentons de récupérer la réponse sous forme de chaîne de caractères. La méthode `getForObject` accepte en derniers paramètres une liste variable d'objets, qui constituent les paramètres à insérer dans la requête HTTP (③).

La combinaison de l'URL demandée et des paramètres fera que l'URL appelée sera la suivante :

```
http://localhost:8080/rest/todo/xml/1
```

La réponse est bien sûr gérée par notre contrôleur REST, qui renvoie pour une telle requête un flux XML :

```
<todo>
  <todoId>1</todoId>
```

```

    <description>todo 1</description>
    <priority>1</priority>
    <completed>true</completed>
    <hasNotes>true</hasNotes>
  </todo>

```

La variable locale `xml` dans l'exemple ci-dessus contiendra donc ce code XML. Il est important de noter que l'utilisation du `RestTemplate` n'est pas limitée à l'interrogation de contrôleurs REST implémentés par Spring MVC. La réponse pourrait tout aussi bien être générée par un autre type de contrôleur Web, dans un langage différent de Java.

Nous venons de voir comment faire une requête GET (au sens HTTP du terme) avec le `RestTemplate`. Celui-ci propose une API complète pour effectuer d'autres types de requêtes, suivant les méthodes du protocole HTTP.

Nous pouvons effectuer une requête DELETE, par exemple, sur la méthode `deleteTodo` de notre contrôleur :

```

restTemplate.delete(
    "http://localhost:8080/rest/todo/{todoId}",
    "1");

```

Cet appel va supprimer le `Todo` d'identifiant 1. Aucune réponse n'est attendue.

Le tableau 7-5 récapitule les principales méthodes disponibles dans le `RestTemplate`. Remarquez l'existence de la méthode `execute`, qui, moyennant une utilisation plus complexe, permet de construire des requêtes et d'exploiter les réponses pour des besoins avancés, selon le principe des templates Spring.

**Tableau 7-5. Méthodes du *RestTemplate***

Méthode HTTP	Méthode du <i>RestTemplate</i>
DELETE	<code>delete(String, String, ...)</code>
GET	<code>delete(String, Class, String, ...)</code>
HEAD	<code>headForHeaders(String, String ...)</code>
OPTIONS	<code>optionsForAllow(String, String ...)</code>
POST	<code>postForLocation(String, Object, String ...)</code>
PUT	<code>put(String, Object, String ...)</code>
Toutes	<code>execute(String, HttpMethod, RequestCallback, ResponseExtractor, String ...)</code>

Nous avons vu dans notre première utilisation du `RestTemplate` qu'il facilitait grandement l'appel de services REST, mais que sa gestion de la réponse était relativement limitée : nous avons récupéré un document XML sous forme de chaîne de caractères. Cette réponse contient les informations demandées, mais nécessite une exploitation (analyse du code XML), qui s'avère fastidieuse si elle doit être faite au niveau de l'appel. L'idéal serait de récupérer l'objet métier attendu, c'est-à-dire une instance de `Todo`, le `RestTemplate` prenant à sa charge la transformation.

Le `RestTemplate` propose un mécanisme de convertisseur qui permet de contrôler très finement la conversion des objets Java qui lui sont passés ainsi que ceux qu'il renvoie. Nous allons ici nous intéresser aux objets renvoyés, en convertissant le flux XML reçu précédemment en un objet `Todo`.

Cette conversion nous permettra d'effectuer l'appel suivant, pour lequel nous demandons et récupérons directement un `Todo`, plutôt qu'un document XML :

```
Todo todo = restTemplate.getForObject(
    "http://localhost:8080/rest/todo/{format}/{todoId}",
    Todo.class,
    "xml", "1");
```

Pour arriver à ce résultat, il faut implémenter un `HttpMessageConverter`, qui va se charger de la conversion de la réponse, puis l'assigner au `RestTemplate`.

Voici le code de l'implémentation de `TodoHttpMessageConverter` :

```
package tudu.web.rest;
(...)
import org.springframework.http.HttpInputMessage;
import org.springframework.http.HttpOutputMessage;
import org.springframework.http.MediaType;
import org.springframework.converter.HttpMessageConverter;
import tudu.domain.model.Todo;
import com.thoughtworks.xstream.XStream;

public class TodoHttpMessageConverter
    implements HttpMessageConverter<Todo> {

    public List<MediaType> getSupportedMediaTypes() {
        return Collections.singletonList(
            new MediaType("text", "xml") ← ❶
        );
    }

    public boolean supports(Class<? extends Todo> clazz) {
        return Todo.class.equals(clazz); ← ❷
    }

    public Todo read(Class<Todo> clazz,
                    HttpInputMessage inputMessage)
        throws IOException {
        XStream xstream = new XStream();
        xstream.alias("todo", Todo.class);
        return (Todo) xstream.fromXML(inputMessage.getBody()); ← ❸
    }

    public void write(Todo t, HttpOutputMessage outputMessage)
        throws IOException {
        throw new UnsupportedOperationException(); ← ❹
    }
}
```

Le rôle d'un `HttpMessageConverter` est d'effectuer des transformations d'objets Java vers des messages HTTP et inversement. Il doit donc indiquer le type de média qu'il est capable de gérer (❶) : dans notre cas les réponses de type `text/xml`, ainsi que la hiérarchie de classes qu'il peut convertir (❷), c'est-à-dire des `Todos`.

La méthode `read` nous intéresse particulièrement, car c'est elle qui se charge de la transformation de la réponse en `Todo`. Nous recevons ici une représentation XML d'un `Todo`, et nous utilisons `XStream` pour la convertir en objet Java (❸). `XStream` présente l'intérêt d'être très simple d'utilisation et concis, mais tout autre mécanisme de désérialisation aurait pu tout aussi bien convenir. L'opération inverse, qui consiste à transformer l'objet Java en message HTTP, ne nous intéressant pas, elle n'est pas implémentée (❹).

Le convertisseur étant écrit, il faut maintenant le positionner au sein du `RestTemplate`, grâce à sa propriété `messageConverters`, qui contient la liste de ses différents convertisseurs :

```
<bean id="restTemplate"
    class="org.springframework.web.client.RestTemplate">
    (...)
    <property name="messageConverters">
        <list>
            <bean class="org.springframework.http.converter.
                ByteArrayHttpMessageConverter" />←❶
            <bean class="org.springframework.http.converter.
                StringHttpMessageConverter" />←❶
            <bean class="tudu.web.rest.TODOHttpMessageConverter" />←❷
        </list>
    </property>
</bean>
```

Un `RestTemplate` dispose de convertisseurs positionnés par défaut, qui gèrent les conversions sous forme de chaînes de caractères ou de tableau d'octets. Il faut les positionner explicitement dès que nous paramétrons la propriété `messageConverters`, afin que ces cas simples soient toujours gérés (❶). Notre convertisseur de `Todo` est lui paramétré au repère (❷).

Le mécanisme de convertisseurs apporte au `RestTemplate` une extensibilité et une adaptabilité très intéressantes, lui permettant de gérer la conversion des messages HTTP en objets Java, afin d'obtenir un code applicatif le plus épuré possible.

## En résumé

Spring 3.0 apporte un support REST à Spring MVC. Il devient alors possible d'implémenter des contrôleurs REST en suivant le modèle de programmation de Spring MVC, fondé sur les annotations.

Le support client n'est pas en reste, avec le `RestTemplate`, qui propose une API très simple et extensible afin d'interroger des services REST (indépendamment de leur technologie) et d'exploiter au mieux leur réponse.

## Mise en œuvre de Spring MVC dans Tudu Lists

Les principaux concepts et composants de Spring MVC ayant été introduits, nous pouvons passer à leur mise en pratique dans notre étude de cas.

### Configuration des contextes

Le premier contexte renvoie à l'application Web. Il est chargé avec le support de la classe `ContextLoaderListener` en utilisant les fichiers dont le nom correspond au pattern **/WEB-INF/applicationContext\*.xml**.

Le second contexte est utilisé par Spring MVC et est chargé par la servlet `DispatcherServlet` en utilisant le fichier **action-servlet.xml** localisé dans le répertoire **WEB-INF**. Les différentes entités relatives à Spring MVC sont définies dans ce fichier.

Cette servlet est configurée dans le fichier **web.xml** de l'application Web, comme l'illustre le code suivant :

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app>
  (...)
  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
    <servlet-name>action</servlet-name>
    <url-pattern>*.action</url-pattern>
  </servlet-mapping>
  (...)
</web-app>
```

Dans le fichier **action-servlet.xml** suivant, l'approche fondée sur les annotations a été activée en se fondant sur la balise `component-scan` (❶) de l'espace de nommage `context` (seuls les contrôleurs localisés dans les packages dont le nom commence par `tudu.web` étant utilisés) :

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

  <context:component-scan base-package="tudu.web" />❶

  (...)
</beans>
```

## Commons Validator

Le projet Spring Modules offre le support de Commons Validator, le framework d'Apache visant à spécifier les règles de validation par déclaration communément utilisé avec Struts.

Commons Validator se configure de la manière suivante dans le cadre de Spring MVC :

```
<bean id="validatorFactory" class="org.springframework.modules.commons
    .validator.DefaultValidatorFactory">
  <property name="validationConfigLocations">
    <list>
      <value>/WEB-INF/validator-rules.xml</value>
      <value>/WEB-INF/validation.xml</value>
    </list>
  </property>
</bean>

<bean id="beanValidator" class="org.springframework.modules.commons
    .validator.DefaultBeanValidator">
  <property name="validatorFactory" ref="validatorFactory"/>
</bean>
```

Dans chaque contrôleur de gestion de formulaire, le Bean `beanValidator` doit être injecté par l'intermédiaire de l'annotation `Autowired`. Le nom du Bean de formulaire utilisé par le contrôleur permet de déterminer la règle à utiliser de la configuration de la validation **WEB-INF/validation.xml**. La méthode `validate` de ce Bean est ensuite utilisée afin de réaliser la validation.

## Implémentation des contrôleurs

Au travers de l'application `Tudu Lists`, différents mécanismes et types de contrôleurs sont implémentés par le biais des annotations `RequestMapping`, `ModelAttribute` et `InitBinder` décrites dans les sections précédentes.

Les sections suivantes se penchent sur la façon dont sont mis en œuvre des contrôleurs à entrée multiple et de formulaire réalisant respectivement un affichage de données et une gestion de formulaire HTML. La présentation associée à ces deux contrôleurs utilise les technologies JSP/JSTL.

### Contrôleur simple

Le contrôleur `ShowTodosAction` du package `tudu.web` dispose de traitements pour afficher les todos d'une liste. Comme l'illustre l'extrait de code ci-après, il utilise Spring MVC à l'aide des éléments suivants :

- Annotation `Controller` au niveau de la classe (❶) afin de préciser que cette dernière joue le rôle d'un contrôleur dans Spring MVC.
- Annotation `RequestMapping` au niveau de la méthode `showTodos` (❷) afin de définir le point d'entrée. Plusieurs points d'entrée peuvent être spécifiés à ce niveau.

- Annotation `RequestParam` afin de récupérer directement en tant que paramètres du point d'entrée les paramètres de la requête Web. Dans notre cas, le paramètre optionnel `listId` (❸) est pris directement dans la requête s'il est présent. Dans le cas contraire, sa valeur vaut `null`.
- Paramètre de type `ModelMap` (❹) afin de manipuler les données du modèle. Des ajouts dans ce dernier peuvent être réalisés par l'intermédiaire de la méthode `addAttribute` de la classe (❺).
- Type de retour afin de préciser la vue utilisée pour présenter les données (❻).

```

@Controller←❶
public class ShowTodosController {
    (...)
    @RequestMapping("/secure/showTodos.action")
    public String showTodos(←❷
        @RequestParam(value="listId",←❸
            required=false) String listId,
        ModelMap model←❹
    ) {

        log.debug("Execute show action");
        Collection<TodoList> todoLists = new TreeSet<TodoList>(
            userManager.getCurrentUser().getTodoLists());

        if (!todoLists.isEmpty()) {
            if (listId != null) {
                model.addAttribute("defaultList", listId);←❺
            } else {
                model.addAttribute("defaultList",
                    todoLists.iterator().next().getListId());←❺
            }
        }
        return "todos";←❻
    }
}

```

Aucune configuration n'est requise dans le fichier **WEB-INF/action-servlet.xml** pour le contrôleur. Les configurations relatives à Spring MVC (❶) et à l'injection de dépendances (❷) se réalise directement dans la classe par l'intermédiaire d'annotations, comme l'illustre le code suivant :

```

@Controller←❶
public class ShowTodosController {
    @Autowired←❷
    private UserManager userManager;

    (...)

    @RequestMapping("/secure/showTodos.action")←❶
    public String showTodos(

```

```
        @RequestParam(value="listId", ← ❶  
                      required=false) String listId,  
        ModelMap model  
    (...)  
    }  
}
```

La manière d'accéder aux points d'entrée du contrôleur est définie dans les annotations `RequestMapping`. Aucune configuration supplémentaire n'est nécessaire. Dans le cas précédent, le contrôleur `ShowTodosController` est accessible à partir de l'URI `/secure/showTodos.action`.

Cette étude de cas met également en pratique le mécanisme de chaînage de `ViewResolver`. Ainsi, retrouve-t-on dans le fichier `/WEB-INF/action-servlet.xml` la configuration de plusieurs `ViewResolver` :

- Le premier est implémenté par la classe `ResourceBundleViewResolver`, qui utilise le fichier de propriétés `views.properties` afin de configurer les vues.
- Le deuxième est implémenté par la classe `XmlViewResolver`, qui utilise le fichier `/WEB-INF/views.xml` pour configurer les vues.
- Le dernier est implémenté par la classe `InternalResourceViewResolver`, qui se fonde, dans notre cas, sur une vue JSP/JSTL. Il constitue la stratégie de résolution par défaut et est utilisé dans le cas où un identifiant de vue ne peut être résolu par les deux entités précédemment décrites.

La figure 7-12 illustre cette chaîne en soulignant les identifiants des vues configurées dans les deux premiers `ViewResolver`.

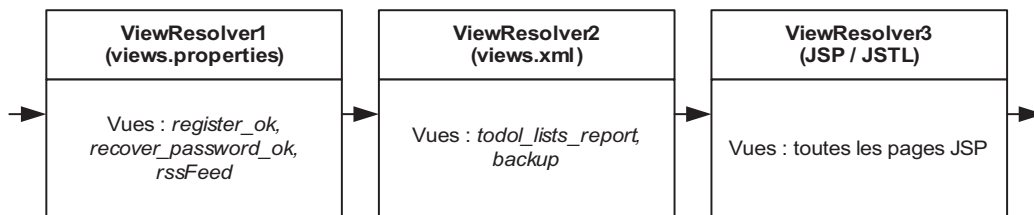


Figure 7-12

Chaînage des `ViewResolver` dans `Tudu Lists`

Le contrôleur utilise la vue `todos` qui est résolue par le dernier `ViewResolver` et correspond donc à la page JSP `todos.jsp` localisée dans le répertoire `/WEB-INF/jsp/`.

### Contrôleur de gestion de formulaire

La mise en œuvre d'un contrôleur de gestion de formulaire n'est guère plus complexe que celle d'un contrôleur simple : un Bean est utilisé pour les données du formulaire, et deux points d'entrée doivent être définis, un pour le chargement du formulaire, l'autre pour sa validation.

Détaillons le contrôleur `MyInfosController` du package `tudu.web`, qui permet de modifier les informations d'un utilisateur de l'application.

Comme l'illustre l'extrait de code ci-après, ce contrôleur utilise Spring MVC à l'aide des éléments suivants :

- Annotations `Controller` et `RequestMapping` au niveau de la classe afin de définir la classe en tant que contrôleur et l'URI d'accès à ce dernier (❶).
- Méthode `showForm` (❷) appelée par l'intermédiaire d'une méthode GET et configurée avec l'annotation `RequestMapping`. Cette méthode a la responsabilité de charger les données du formulaire.
- Méthode `submitForm` (❸) appelée afin de traiter les données soumises par le formulaire avec la méthode HTTP POST et configurée avec l'annotation `RequestMapping`.
- Utilisation explicite de la validation Commons Validator dans le corps de la méthode `submitForm` (❹).
- Spécification des vues utilisées pour l'affichage suite à la soumission du formulaire par l'intermédiaire des retours des deux méthodes précédentes (❺).

```
@Controller←❶
@RequestMapping("/secure/myinfos.action")
public class MyInfoController {
    (...)
    @RequestMapping(method = RequestMethod.GET)
    public String showForm(HttpServletRequest request,←❷
        ModelMap model) {
        String login = request.getRemoteUser();
        User user = userManager.findUser(login);
        UserInfoData data=new UserInfoData();
        data.setPassword(user.getPassword());
        data.setVerifyPassword(user.getPassword());
        data.setFirstName(user.getFirstName());
        data.setLastName(user.getLastName());
        data.setEmail(user.getEmail());
        model.addAttribute("userinfo", data);
        return "userinfo";←❺
    }

    @RequestMapping(method=RequestMethod.POST)
    public String submitForm(←❸
        HttpServletRequest request,
        @ModelAttribute("userinfo") UserInfoData userInfo,
        BindingResult result) {

        beanValidator.validate(userInfo, result);←❹
        if (!result.hasErrors()) {←❹
            String password = userInfo.getPassword();
            String firstName = userInfo.getFirstName();
            String lastName = userInfo.getLastName();
```

```
        String email = userInfo.getEmail();
        String login = request.getRemoteUser();
        User user = userManager.findUser(login);
        user.setPassword(password);
        user.setFirstName(firstName);
        user.setLastName(lastName);
        user.setEmail(email);
        userManager.updateUser(user);
    }

    return "userinfo"; ← 5
}
}
```

Aucune configuration n'est requise dans le fichier **WEB-INF/action-servlet.xml** pour le contrôleur. Comme le montre le code suivant, les configurations relatives à Spring MVC (1) et à l'injection de dépendances (2) se réalisent directement dans la classe par l'intermédiaire d'annotations :

```
@Controller ← 1
public class MyInfoController {
    @Autowired ← 2
    private UserManager userManager;

    @Autowired ← 2
    private DefaultBeanValidator beanValidator;

    (...)

    @RequestMapping(method = RequestMethod.GET) ← 1
    public String showForm(HttpServletRequest request,
                          ModelMap model) {
        (...)
    }

    @RequestMapping(method=RequestMethod.POST) ← 1
    public String submitForm(
        HttpServletRequest request,
        @ModelAttribute("userinfo") UserInfoData userInfo, ← 1
        BindingResult result) {
        (...)
    }
}
```

La manière d'accéder aux points d'entrée du contrôleur est définie dans les annotations `RequestMapping`. Aucune configuration supplémentaire n'est nécessaire. Dans l'exemple précédent, le contrôleur `MyInfoController` est accessible à partir de l'URI `/secure/myinfos.action` par l'intermédiaire des méthodes GET et POST.

Les données du formulaire soumises peuvent être validées par le biais de l'abstraction `Validator` vue précédemment, et plus particulièrement les implémentations constituant le support de `Commons Validator`, qui permet de réutiliser les règles de validation `userinfo`.

Le contrôleur utilise la vue `user_info` dans le but d'afficher le formulaire dans les cas d'échec ou de succès lors de la validation. Cette vue est résolue par le dernier `ViewResolver` de la chaîne et correspond donc à la page JSP `user_info.jsp` localisée dans le répertoire `/WEB-INF/jsp/`. Cette page doit être modifiée afin de remplacer les taglibs de gestion des formulaires de Struts par ceux de Spring MVC.

Comme l'illustre le code suivant, le framework met en œuvre le taglib `form` afin de remplir les champs du formulaire (❶) et d'afficher les éventuelles erreurs de validation (❷) :

```
(...)
<form:form modelAttribute="userinfo" focus="firstName">
  <font color="red">
    <b><form:errors path="*" /></b>
  </font>
  (...)
  <form:input path="firstName" size="15" maxLength="60" /> ←❶
  &#160;<font color="red">
    <form:errors path="firstName" /></font> ←❷
  (...)
  <input type="submit" value="<fmt:message key="form.submit" />" />
  <input type="button"
    onclick="document.location.href='<c:url
      value="../welcome.action" />';"
    value="<fmt:message key="form.cancel" />" />
</form>
```

Le contrôleur affiche la page de formulaire suite au succès de la soumission des données par l'appel de la méthode `showForm` dans la méthode `onSubmit`. L'utilisation d'une autre vue grâce aux méthodes `setSuccessView` et `getSuccessView` serait aussi possible.

## Implémentation de vues spécifiques

Tudu Lists utilise plusieurs technologies de présentation afin de générer différents formats de sortie (HTML, XML, RSS, PDF). Spring MVC offre une infrastructure facilitant leur utilisation conjointement dans une même application.

Dans la mesure où l'entité `ViewResolver` utilisée précédemment pour les vues utilise les technologies JSP et JSTL, elle ne peut servir pour les vues décrites ci-dessous. Dans les sections suivantes, nous utiliserons donc l'implémentation `XmlViewResolver` afin de les configurer.

### XML

L'application Tudu Lists permet de sauvegarder au format XML les informations contenues dans une liste de todos. Cette fonctionnalité est implémentée par le contrôleur `Backup-ToDoListController` du package `tudu.web`, qui a la responsabilité de charger la liste correspondant

à l'identifiant spécifié puis de la placer dans le modèle afin de la rendre disponible lors de la construction de la vue.

Ce contrôleur dirige les traitements vers la vue ayant pour identifiant `backup` à la fin de ses traitements. Cette vue, implémentée par la classe `BackupTodoListView` du package `tudu.web`, est résolue par le second `ViewResolver` de la chaîne.

Sa configuration se trouve dans le fichier **views.xml** localisé dans le répertoire **WEB-INF**. Elle utilise le support de la classe `AbstractXsltView` de Spring MVC afin de générer une sortie XML, comme dans le code suivant :

```
public class BackupTodoListView extends AbstractXsltView {
    (...)
    protected Node createDomNode(Map model,
        String rootName, HttpServletRequest request,
        HttpServletResponse response) throws Exception {

        TodoList todoList = (TodoList)model.get("todoList");
        Document doc = todoListsManager.backupTodoList(todoList);
        return new DOMOutputter().output( doc );
    }
}
```

À la fin de l'exécution de la méthode `createDomNode`, la classe `BackupTodoListView` rend la main à sa classe mère afin de réaliser éventuellement une transformation XSLT et d'écrire le contenu sur le flux de sortie.

L'utilisation de `XmlViewResolver` permet d'injecter une instance du composant `TodoListsManager` dans la classe de la vue, comme décrit dans le fichier **views.xml** suivant :

```
<bean id="backup" class="tudu.web.BackupTodoListView">
    <property name="todoListsManager" ref="todoListsManager"/>
</bean>
```

## PDF

L'application de l'étude de cas permet également de générer un rapport au format PDF avec les informations contenues dans une liste de todos. Cette fonctionnalité est implémentée par l'intermédiaire du contrôleur `ShowTodoListsReportController` du package `tudu.web`, qui a la responsabilité de charger la liste correspondant à l'identifiant spécifié puis de la placer dans le modèle afin de la rendre disponible lors de la construction de la vue.

Ce contrôleur dirige les traitements vers la vue ayant pour identifiant `todo_lists_report` à la fin de ses traitements. Cette vue, implémentée par la classe `ShowTodoListsPdfView` du package `tudu.web`, est résolue par le second `ViewResolver` de la chaîne.

Sa configuration se trouve dans le fichier **views.xml** localisé dans le répertoire **WEB-INF**. Elle utilise le support de la classe `AbstractPdfView` de Spring MVC afin de générer une sortie PDF par le biais du framework `iText`, comme dans le code suivant :

```
public class ShowTodoListsPdfView extends AbstractPdfView {

    protected void buildPdfDocument(
```

```
        Map model, Document doc,  
        PdfWriter writer, HttpServletRequest req,  
        HttpServletResponse resp) throws Exception {  
  
        TodoList todoList=(TodoList)model.get("todoList");  
  
        doc.add(new Paragraph(todoList.getListId()));  
        doc.add(new Paragraph(todoList.getName()));  
        (...)  
    }  
}
```

La classe mère de la classe `ShowTodoListsPdfView` a la responsabilité d'instancier les différentes classes de `iText` afin de les lui fournir. À la fin de l'exécution de la méthode `buildPdfDocument`, la classe `ShowTodoListsPdfView` rend la main à sa classe mère pour écrire le contenu sur le flux de sortie.

Notons que cette vue aurait pu aussi bien être configurée dans le fichier **views.properties**, puisqu'elle ne nécessite pas d'injection de dépendances.

## Conclusion

Pour mettre en œuvre le patron de conception MVC, Spring MVC offre une approche intéressante fondée sur les mécanismes d'injection de dépendances et les métadonnées configurées dans des annotations.

Les principaux atouts du framework résident dans son ouverture ainsi que dans la modularité et l'isolation des composants du patron MVC. Le développement des applications utilisant différentes technologies s'en trouve facilité, et le changement de briques n'impacte pas le reste de l'application ni l'extension du framework.

L'utilisation de l'injection de dépendances de Spring dans l'implémentation du pattern MVC permet de configurer avec une réelle facilité ces différents composants ainsi que leurs dépendances. Cette configuration est centralisée dans le contexte applicatif de Spring, lequel peut de la sorte mettre à disposition toute la puissance du conteneur léger.

Spring MVC propose une approche fondée sur les annotations afin d'implémenter les contrôleurs. Introduite avec la version 2.5 du framework, cette approche d'une grande facilité d'utilisation permet de s'abstraire de l'API Servlet. Le framework propose ainsi une gestion des formulaires d'une flexibilité remarquable.

Le framework Spring MVC propose enfin une séparation claire entre le contrôleur et la vue ainsi qu'un support pour les différentes technologies et frameworks de présentation.

En résumé, ce framework fournit un socle solide pour développer des applications Web sans donner la possibilité d'abstraire la navigation et l'enchaînement des pages. Un framework tel que Spring Web Flow, présenté en détail au chapitre suivant, peut s'appuyer sur ce socle afin de résoudre cette problématique.