Spring 1.2 et 2.0

Spring par la pratique

Mieux développer ses applications Java/J2EE avec Spring, Hibernate, Struts, Ajax...

Julien Dubois
Jean-Philippe Retaillé
Thierry Templier

Préface de Rod Johnson, fondateur de Spring

EYROLLES

CHEZ LE MÊME ÉDITEUR

Des mêmes auteurs

T. TEMPLIER, A. GOUGEON. – **JavaScript pour le Web 2.0**.
N°12009, 2007, 492 pages.

J.-P. RETAILLÉ. – **Refactoring des applications Java/J2EE**.
N°11577, 2005, 390 pages.

R. PAWLAK, J.-P. RETAILLÉ, L. SEINTURIER. – **Programmation orientée aspect pour Java/J2EE**.
N°11408, 2004, 462 pages.

Dans la même collection

D. THOMAS *et al*. – **Ruby on Rails**.
N°12079, 2e édition, 2007, 750 pages.

R. GOETTER. – **CSS 2 : pratique du design web**.
N°11976, 2e édition, 2007, 350 pages.

C. PORTENEUVE – **Bien développer pour le Web 2.0 – Bonnes pratiques Ajax**.
N°12028, 2007, 580 pages.

M. PLASSE. – **Développez en Ajax**.
N°11965, 2006, 314 pages.

E. DASPET et C. PIERRE de GEYER. – **PHP 5 avancé**.
N°12004, 3e édition 2006, 804 pages.

A. PATRICIO. – **Hibernate 3.0**.
N°11644, 2005, 336 pages.

K. DJAFAAR. – **Eclipse et JBoss**.
N°11406, 2005, 656 pages + CD-Rom.

J. WEAVER, K. MUKHAR, J. CRUME. – **J2EE 1.4**.
N°11484, 2004, 662 pages.

L. DERUELLE. **– Développement Java/J2EE avec JBuilder**.
N°11346, 2003, 726 pages + CD-Rom.

Autres ouvrages sur Java/J2EE

J. MOLIÈRE. – **Cahier du programmeur J2EE**. *Conception et déploiement J2EE.*
N°11574, 2005, 234 pages.

E. PUYBARET. – **Cahier du programmeur Swing**.
N°12019, 2007, 500 pages.

R. FLEURY. – **Cahier du programmeur Java/XML**. *Méthodes et frameworks : Ant, Junit, Eclipse, Struts-Stxx, Cocoon, Axis, Xerces, Xalan, JDom, XIndice…*
N°11316, 2004, 228 pages.

E. PUYBARET. – **Cahier du programmeur Java 1.4 et 5.0**.
N°11916, 3e édition, 2006, 380 pages.

C. DELANNOY. – **Programmer en Java**.
N°11748, 4e édition, 2006, 774 pages + CD-Rom.

Spring
par la pratique

Mieux développer ses applications Java/J2EE
avec Spring, Hibernate, Struts, Ajax...

Julien Dubois
Jean-Philippe Retaillé
Thierry Templier

avec la contribution
de Séverine Templier Roblou
et de Olivier Salvatori

2e tirage 2007

EYROLLES

ÉDITIONS EYROLLES
61, bd Saint-Germain
75240 Paris Cedex 05
www.editions-eyrolles.com

Le code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée notamment dans les établissements d'enseignement, provoquant une baisse brutale des achats de livres, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.

En application de la loi du 11 mars 1957, il est interdit de reproduire intégralement ou partiellement le présent ouvrage, sur quelque support que ce soit, sans autorisation de l'éditeur ou du Centre Français d'Exploitation du Droit de Copie, 20, rue des Grands-Augustins, 75006 Paris.
© Groupe Eyrolles, 2006, ISBN : 978-2-212-11710-3

Préface

Rod JOHNSON
Founder, Spring Framework, CEO, Interface21

French readers have had to wait longer than most for a book on Spring in their native language. However, the wait has not been in vain, and they are fortunate in this, the first book on Spring in French.

It is almost five years since I wrote the first code towards what would later become the Spring Framework. The open source project formally began in February 2003, soon making the product far more than any individual could achieve. Since that time, Spring has become widely used worldwide, powering applications as diverse as retail banking portals; airline reservation systems; the French online tax submission system; payment engines handling inter-bank transfers, salaries and utility bills; search engines; government agency portals including that of the European Patent Office; critical scientific research systems; logistics solutions; and football web sites. In that time, Spring also spawned a rich literature, in a variety of languages.

This book does an excellent job, not merely of describing what Spring does, and how, but the central issue of why. Excellent examples illustrate the motivation for important Spring concepts and capabilities, making it not merely a book about a particular product, but a valuable book about writing effective server-side Java applications.

While this book is ideal as an introduction to Spring and modern concepts such as Dependency Injection and Aspect Oriented Programming, it always respects the reader. The authors never write down to their readership. While their experience stands out, and they offer clear guidance as to best practice, the reader feels involved in their discussion of architectural choices and trade-offs.

The content is not only up to date, but broad in scope and highly readable. Enterprise Java is a dynamic area, and open source projects are particularly rapidly moving targets. Spring has progressed especially rapidly in the last six months, with work leading up to the final release of Spring 2.0. The authors of this book have done a remarkable job of writing about Spring 2.0 features as soon as they have stabilized. The coverage of AJAX is also welcome.

The writing style is clear and to the point, making the book a pleasure to read.

Finally, the authors' commitment to providing a realistic sample application (rather than the simplistic effort that mars many books), is shown by the fact that Tudu Lists has become a viable open source project in its own right.

I highly recommend this book to all those new to the Spring Framework or wishing to deepen their understanding of it, as well as those who wish to understand the current state of enterprise Java development.

Rod JOHNSON,
fondateur du framework Spring et président d'Interface21

Si les lecteurs francophones ont dû patienter plus que d'autres pour avoir accès à un livre sur Spring écrit dans leur langue, leur attente n'aura pas été vaine, puisque ce premier ouvrage en français dédié à Spring est une grande réussite.

Voici bientôt cinq ans que j'ai écrit les premières lignes du code de ce qui allait devenir le framework Spring. Le projet Open Source lui-même n'a réellement débuté qu'en février 2003, pour aboutir rapidement à un produit outrepassant de beaucoup ce qu'une seule personne aurait pu réaliser. Aujourd'hui, Spring est largement utilisé à travers le monde, dans des applications aussi diverses que des portails bancaires publics, des systèmes de réservation de billets d'avion, le système français de déclaration de revenus en ligne, des moteurs de paiements assurant les transferts interbancaires ou la gestion de la paie et des factures, des moteurs de recherche, des portails de services gouvernementaux, dont celui de l'Office européen des brevets, des systèmes critiques de recherche scientifique, des solutions de logistique ou des sites… dédiés au football. Durant toute cette période, Spring a fait l'objet d'une abondante littérature, dans un grand nombre de langues.

Au-delà de la description de ce que fait Spring et de la façon dont il le fait, toute l'originalité de ce livre réside dans sa façon de répondre à la question centrale du *pourquoi*. Les très bons exemples qui illustrent les motivations ayant conduit à l'élaboration des concepts et des fonctionnalités fondamentales de Spring en font, bien plus qu'un simple manuel de prise en main, un ouvrage de référence pour quiconque souhaite réaliser efficacement des applications Java côté serveur.

Idéal pour une introduction à Spring et à des concepts aussi modernes que l'injection de dépendances ou la programmation orientée aspect, ce livre respecte en outre toujours le lecteur, les auteurs s'étant fait un point d'honneur de ne jamais le prendre de haut. Tout en profitant de leur vaste expérience et de leur clair exposé des meilleures pratiques, le lecteur se sent continuellement impliqué dans leur présentation critique des choix d'architecture et des compromis qui en découlent.

Le contenu de l'ouvrage est parfaitement à jour et couvre une large gamme de sujets. J2EE est un domaine très actif, dans lequel les projets Open Source évoluent de manière extrêmement rapide. Spring lui-même a fortement progressé au cours des six derniers mois, pour atteindre sa version finalisée Spring 2.0. Les auteurs de ce livre ont accompli une véritable prouesse pour traiter des fonctionnalités de cette version de Spring 2.0 dès qu'elles ont pu être stabilisées. La couverture de la technologie AJAX est en outre particulièrement bienvenue.

Pour finir, les auteurs ont fait l'effort d'adjoindre au livre une application exemple réaliste plutôt qu'une étude de cas simpliste, comme on en trouve dans trop d'ouvrages. Cette application, Tudu Lists, est même devenue un projet Open Source à part entière, avec déjà de nombreux utilisateurs.

Ajoutons que le style d'écriture est clair et pragmatique, rendant le parcours du lecteur très agréable.

Pour toutes ces raisons, je ne saurais trop recommander la lecture de cet ouvrage à ceux qui débutent dans l'utilisation du framework Spring ou qui souhaitent en approfondir la maîtrise comme à ceux qui ont à cœur de mieux comprendre l'état de l'art du développement Java d'entreprise.

Remerciements

Nous remercions Éric Sulpice, directeur éditorial d'Eyrolles, et Olivier Salvatori pour leurs multiples relectures et conseils.

Nous remercions également les personnes suivantes de la communauté Spring, pour leur confiance, leur accessibilité et leur gentillesse : Rod Johnson, Juergen Hoeller, Rob Harrop, Dmitriy Kopylenko, Steven Devijver, Jan Machacek et Costin Leau.

Enfin, merci à toutes les personnes qui ont eu la gentillesse de nous soutenir et de nous relire, notamment Stéphane Labbé, Erwan Perigault, Pascal Poussard et Jérôme Morille.

Julien Dubois :

Merci à ma famille et à mes proches pour leur soutien tout au long de cette aventure.

Jean-Philippe Retaillé :

Merci à Audrey et à ma famille pour m'avoir soutenu dans l'écriture de cet ouvrage.

Thierry Templier :

Merci à ma femme, Séverine, pour son soutien et son aide précieuse tout au long de ce projet. Merci également à toutes les personnes de Paris et de Nantes avec qui j'ai eu l'occasion d'échanger sur Spring (David Fiou, Fabrice Legrand, Bruno Rizzi, Louis-Gilles Ovize, Didier Girard, Laurent Guérin, Gérald Bureau et Arnaud Gougeon), ainsi qu'aux membres de l'OSSGTP (Open Source Software Get-Together Paris) pour leur accueil.

Table des matières

PARTIE I

Les fondations de Spring

PARTIE IV

Technologies d'intégration

CHAPITRE 13

Avant-propos

Spring est un framework Open Source rendant l'utilisation de J2EE à la fois plus simple et plus productive. Tout au long de cet ouvrage, nous nous efforçons de dégager les bonnes pratiques de développement d'applications Java/J2EE, dont une large part ne sont pas spécifiques de Spring, mais dont la mise en œuvre est grandement simplifiée et rendue plus consistante grâce à son utilisation.

Spring s'appuie sur des concepts modernes, tels que l'inversion de contrôle ou la programmation orientée aspect, afin d'améliorer l'architecture des applications Java/J2EE en les rendant tout à la fois plus souples, plus agiles et plus facilement testables.

S'intégrant avec les grands frameworks Open Source tels que Struts ou Hibernate, ainsi qu'avec les standards J2EE, Spring propose un modèle d'application cohérent, complet et simple d'emploi.

Recommandé par de nombreux architectes et développeurs expérimentés, Spring commence à se diffuser au sein des SSII et des entreprises françaises. Une bonne connaissance de ce produit est donc essentielle dans le monde très concurrentiel de l'informatique d'entreprise d'aujourd'hui.

Objectifs de cet ouvrage

Cet ouvrage se veut un guide pratique pour le développement d'applications Java/J2EE fondées sur Spring.

Nous avons voulu le rendre accessible au plus grand nombre afin de permettre aux développeurs Java/J2EE d'être plus productifs et de mieux réussir leurs projets grâce à l'utilisation de Spring.

C'est la raison pour laquelle nous n'entrons pas dans la description d'API complexes. Il s'agit avant tout d'un ouvrage didactique, destiné à rendre le lecteur directement opérationnel.

Cette volonté d'accessibilité ne signifie pas pour autant que l'ouvrage soit d'une lecture simple et peu technique. Lorsque c'est nécessaire, nous abordons aussi des thèmes complexes, comme les transactions avec JTA ou l'intégration avec JCA.

Convaincus que l'on apprend mieux par la pratique, nous adjoignons à l'ouvrage une étude de cas pratique, l'application Tudu Lists, développée pas à pas tout au long des chapitres. Le lecteur a de la sorte sous les yeux, au fur et à mesure de sa progression dans l'ouvrage, des exemples de mise en œuvre concrète, dans une application réelle, des sujets traités.

Organisation de l'ouvrage

L'ouvrage commence par décrire des concepts et des problèmes courants des applications Java/J2EE, avant d'aborder en douceur l'utilisation du conteneur Spring. Des sujets classiques sont également traités, tels que les frameworks MVC ou le mappage objet/relationnel, en montrant de quelle manière Spring permet d'être plus efficace dans la mise en œuvre de ces techniques.

L'ouvrage comporte cinq grandes parties :

- La première partie introduit les principaux concepts de Spring. L'injection de dépendances et la programmation orientée aspect y sont décrits et détaillés, d'abord de manière globale puis de manière spécifique à Spring.
- La partie II concerne la couche de présentation d'une application Web. Nous y présentons le très classique Struts, ainsi que deux frameworks spécifiques de Spring : Spring MVC et Spring Web Flow. Nous passons également en revue les deux concepts très populaires que sont les portlets et les technologies AJAX.
- La partie III est dédiée à la couche de persistance des données, essentiellement le mappage objet/relationnel et la gestion des transactions.
- Une application ayant souvent besoin d'interagir avec d'autres systèmes, la partie IV s'intéresse aux technologies d'intégration. Cette intégration peut être réalisée en Java, avec les technologies JCA ou JMS, mais également en XML, en particulier *via* des services Web. Cette partie se conclut par un chapitre dédié à la sécurité des applications.
- La partie V présente deux technologies connexes, qui permettent d'améliorer la qualité d'une application dans son ensemble : la supervision, avec JMX, et les tests unitaires, avec JUnit.

À propos de l'application Tudu Lists

L'application Tudu Lists, qui nous sert d'étude de cas tout au long de l'ouvrage, est un exemple concret d'utilisation des technologies Spring. Il s'agit d'un projet Open Source réel, qui a été réalisé spécifiquement pour cet ouvrage, et qui permet d'illustrer par l'exemple les techniques décrites dans chacun des chapitres.

Loin de n'être qu'un simple exemple, cette application est utilisée en production depuis quelques mois dans plusieurs entreprises. Le principal serveur Tudu Lists possède ainsi plus de 5 000 utilisateurs.

Cette application étant Open Source, le lecteur est invité à participer à son développement. Elle est disponible sur le site de SourceForge, à l'adresse *http://tudu.sourceforge.net.*

À qui s'adresse l'ouvrage?

Cet ouvrage s'adresse à tout développeur J2EE souhaitant améliorer sa productivité et ses méthodes de développement et s'intéressant à l'architecture des applications.

Il n'est nul besoin d'être expert dans les différentes technologies présentées. Chaque chapitre présente clairement chacune d'elles puis montre comment elle est implémentée dans Spring avant d'en donner des exemples de mise en œuvre dans l'application Tudu Lists.

Pour toute question concernant cet ouvrage, vous pouvez contacter les auteurs sur la page Web dédiée à l'ouvrage du site d'Eyrolles, à l'adresse *www.editions-eyrolles.com.*

1

Introduction

Les développements Java/J2EE, notamment ceux qui utilisent les EJB, sont réputés complexes, tant en terme de développement que de tests et de maintenance. La productivité des développeurs Java/J2EE ne disposant que des standards constituant la plateforme est faible, comparée à celle obtenue avec d'autres technologies, comme les langages de type RAD (Rapid Application Development). Sur les projets Java/J2EE de taille conséquente, une couche d'abstraction est généralement développée afin de simplifier le travail des développeurs et leur permettre de se concentrer davantage sur la réelle valeur ajoutée d'une application, à savoir le métier.

C'est à partir de ce constat que les concepteurs de Spring, experts J2EE de renommée mondiale, ont imaginé une solution permettant de simplifier et structurer les développements J2EE de manière à respecter les meilleures pratiques d'architecture logicielle.

Spring est à ce titre un véritable cadre de travail. Non content de structurer les développements spécifiques d'un projet, il propose une intégration des frameworks tiers les plus répandus. Grâce aux communautés Open Source très actives du monde Java, un grand nombre de frameworks spécialisés sur différentes problématiques sont apparus, notamment Struts pour la couche présentation des applications et Hibernate pour la persistance des données. Ainsi, il n'est pas rare d'utiliser au sein d'un même projet plusieurs frameworks spécialisés.

Spring simplifie l'utilisation de ces frameworks en les insérant dans son cadre de travail. La vocation de Spring est d'offrir, non pas toutes les fonctionnalités dont un développeur pourrait rêver, mais une architecture applicative générique capable de s'adapter à ses choix technologiques en terme de frameworks spécialisés.

Cet ouvrage ambitionne de proposer une vision globale de Spring, afin d'en présenter toute la richesse et d'en faciliter la mise en œuvre dans des projets. Nous commençons dans ce chapitre par aborder les problématiques rencontrées dans les projets Java/J2EE classiques, à savoir la séparation des préoccupations techniques et fonctionnelles, la productivité des développements, l'indépendance du code vis-à-vis de la plate-forme d'exécution et les tests.

Spring apporte des réponses à ces problématiques en se reposant sur la notion de conteneur léger et sur la POA (programmation orientée aspect) et en introduisant les meilleures pratiques en matière d'architecture applicative et de développement d'applications.

En fin de chapitre, nous introduisons l'étude de cas Tudu Lists, tirée d'un projet Open Source, qui nous servira de fil conducteur tout au long de l'ouvrage. Au travers de sa mise en œuvre concrète, nous illustrerons tous les principes fondamentaux de Spring ainsi que l'intégration des différentes technologies manipulées par les projets Java J2EE (persistance des données, gestion des transactions, etc.).

Problématiques des développements Java/J2EE

J2EE constitue une infrastructure complexe, qui demande un niveau technique non négligeable pour être bien maîtrisée. Par rapport au développement d'applications métier, cette infrastructure soulève un certain nombre de difficultés, auxquelles tout développeur Java/J2EE est confronté un jour ou l'autre, notamment les quatre problèmes majeurs suivants :

- J2EE n'encourage pas intrinsèquement une bonne séparation des préoccupations, c'est-à-dire l'isolation des différentes problématiques qu'une application doit gérer (typiquement, les problématiques techniques, d'une part, et les problématiques métier, d'autre part).
- J2EE est une plate-forme complexe à maîtriser, qui pose des problèmes de productivité importants, la part des développements consacrés aux problématiques techniques étant disproportionnée par rapport à celle vouée aux problématiques fonctionnelles, qui est pourtant la véritable valeur ajoutée d'une application.
- J2EE impose une plate-forme d'exécution lourde, qui pose des problèmes d'interopérabilité entre différentes implémentations. Elle peut aussi nuire à la réutilisation des services composant une application, si ceux-ci doivent fonctionner sur du matériel ne pouvant supporter un serveur d'applications dans le cadre d'une utilisation nomade (assistants personnels, téléphones, etc.).
- Les développements utilisant J2EE s'avèrent souvent difficiles à tester du fait de leur forte dépendance vis-à-vis de cette plate-forme et de la lourdeur de celle-ci.

Nous détaillons dans cette section ces différentes problématiques, qui nous serviront de référence dans le reste de l'ouvrage pour illustrer tout l'intérêt d'utiliser un framework tel que Spring dans nos développements J2EE.

La séparation des préoccupations

Le concept de séparation des préoccupations consiste à isoler au sein des applications les différentes problématiques qu'elles ont à traiter. Deux catégories de préoccupations parmi les plus évidentes sont les préoccupations d'ordre technique, à l'image des mécanismes de persistance des données, et les préoccupations d'ordre métier, comme la gestion des données de l'entreprise.

L'idée sous-jacente à la séparation des préoccupations est de garantir une meilleure évolutivité des applications grâce à une bonne isolation des différentes problématiques. Chaque préoccupation est traitée de la manière la plus indépendante possible des autres afin d'en pérenniser au maximum le code, tout en limitant les effets de bord liés à l'évolution d'une préoccupation.

En spécifiant J2EE (Java 2 Enterprise Edition), Sun Microsystems visait à transformer le langage Java en un véritable langage d'entreprise, permettant une séparation nette entre les développeurs « techniques » et les développeurs « métier ». L'objectif avoué de cette séparation était de permettre à des non-informaticiens de prendre en charge directement le développement des composants implémentant la logique métier, en l'occurrence les EJB (Enterprise JavaBeans), en connaissant le strict minimum de programmation nécessaire. Force est de constater que les EJB n'ont pas tenu leurs promesses.

L'une des raisons de cet échec est que la séparation des préoccupations au sein d'une architecture J2EE se fondant uniquement sur les EJB n'est pas suffisante pour isoler complètement les développements métier des problématiques techniques. La conception même des composants métier est directement influencée par l'architecture technique sous-jacente. Combien de projets J2EE ont échoué pour avoir utilisé des EJB d'une manière ingérable par le serveur d'applications, par exemple en définissant des EJB Session ayant une granularité trop fine ?

Par ailleurs, la séparation des préoccupations est limitée à des préoccupations spécifiques, et J2EE ne propose pas de mécanisme générique permettant d'élargir ce périmètre. En l'occurrence, la séparation des préoccupations porte essentiellement sur les EJB, alors qu'une application ne se résume pas, loin s'en faut, à ces derniers. Les préoccupations techniques liées à la persistance des données, aux transactions ou à la sécurité peuvent concerner bien d'autres composants de l'application.

La productivité des développements

Si les normes J2EE définissent une infrastructure technique de base pour développer des applications, celle-ci s'avère insuffisante en terme de productivité par rapport à d'autres technologies. Ces normes occultent le problème de la productivité, laissant place à des solutions complémentaires masquant la complexité de la technologie mais faisant perdre en même temps les gains attendus de la standardisation.

La mise en place de ces solutions, régulièrement développées spécifiquement pour un projet, a impacté fortement les budgets d'investissement, souvent aux dépens des développements strictement métier. Si l'utilisation de technologies standards est un gage de

pérennité, la seule standardisation ne suffit pas à répondre aux besoins des clients, dont l'objectif est toujours d'avoir plus pour moins cher.

Pour combler les lacunes de la plate-forme J2EE, la lenteur de son instance de standardisation, le JCP (Java Community Process), et éviter d'avoir à développer et maintenir des solutions maison, la communauté Java, a dû s'appuyer sur des offres complémentaires, notamment Open Source.

Pour gagner en productivité, les projets Java/J2EE actuels abandonnent les solutions maison au profit de frameworks tiers implémentant les meilleures pratiques disponibles. L'exemple qui vient immédiatement à l'esprit est celui de Struts. Ce framework, ou cadre de travail, structure les développements Web selon un modèle de conception éprouvé depuis le langage SmallTalk, le modèle MVC (Model View Controller) et fournit une boîte à outils rudimentaire pour structurer et accélérer les développements.

Malheureusement, les frameworks sont des outils spécialisés dans une problématique donnée, à l'image de Struts pour le Web ou Log4J pour les traces applicatives. Pour parvenir à un niveau de productivité satisfaisante, il est donc nécessaire de recourir à plusieurs frameworks. Cela pose d'évidents problèmes d'intégration de ces frameworks au sein des applications, problèmes auxquels J2EE n'apporte aucune réponse.

L'indépendance vis-à-vis de la plate-forme d'exécution

Pour respecter un de ses principes fondateurs, le fameux « Write Once, Run Anywhere » (écrire une fois, exécuter partout), Java repose sur une machine virtuelle permettant de s'abstraire des plates-formes matérielles sous-jacentes.

Avec J2EE, ce principe fondateur se heurte à deux écueils : la difficulté de standardisation d'une technologie complexe et le choix d'une architecture exclusivement orientée serveur.

La standardisation est un processus long et difficile. Si plusieurs acteurs sont impliqués, il est nécessaire de trouver un consensus sur le contenu du standard. Par ailleurs, ce standard doit laisser le moins de place possible à l'ambiguïté et à l'imprécision, autant de niches engendrant des implémentations incompatibles.

Au finale, les premières implémentations du standard J2EE se sont avérées assez liées au serveur d'applications sous-jacent. La situation s'est nettement améliorée depuis, avec la mise en place d'implémentations de référence et d'un processus de certification très rigoureux. Cependant, certains aspects, comme la sécurité, ne sont que partiellement couverts par la norme J2EE, ouvrant la porte aux solutions propriétaires.

Le deuxième écueil vient de l'orientation exclusivement serveur de J2EE, un choix « politique » de la part de Sun Microsystems, selon qui « the Network is the Computer » (le réseau est l'ordinateur). Malheureusement, cette orientation réseau est préjudiciable à certains types d'applications, notamment celles qui doivent fonctionner sans réseau et qui nécessitent généralement des services de persistance des données et de gestion des transactions. Le code métier reposant sur les EJB nécessite un serveur d'applications pour s'exécuter. Or l'installation d'un serveur d'applications J2EE directement sur un poste de travail nomade ou un assistant personnel est difficilement justifiable.

Finalement, J2EE répond à des problématiques fortes mais pose des problèmes d'interopérabilité au niveau des EJB, qui nuisent à la réutilisation des composants construits sur cette architecture. Par ailleurs, l'orientation réseau de J2EE gêne son utilisation dans certains domaines comme le nomadisme.

Les tests

La problématique des tests est fondamentale dans tout développement d'application. La technologie J2EE s'avère complexe à tester, car elle nécessite un serveur d'applications pour s'exécuter. De cette nécessité résultent une exécution des tests lourde et une difficulté d'automatisation.

Dans les faits, il s'agit davantage de tests d'intégration que de tests réellement unitaires. Des solutions telles que Cactus, de la communauté Apache Jakarta, restent lourdes à mettre en œuvre par rapport à l'outillage disponible pour les classes Java classiques (frameworks dérivés de JUnit, Mock Objects, etc.).

La difficulté de tester une application Java/J2EE selon une granularité plus fine que les tests d'intégration nuit fortement à la qualité des applications construites sur cette plate-forme. D'où des problèmes de productivité et de maintenance évidents.

En résumé

Nous avons vu que J2EE posait des problèmes pénalisants pour le développement des applications. J2EE n'encourage pas intrinsèquement de bonnes pratiques de conception, comme la séparation des préoccupations. La lourdeur de cette plate-forme rend de surcroît l'application dépendante du serveur d'applications sur lequel elle fonctionne, et les tests de ses composants s'avèrent difficiles à mettre en œuvre.

Face à ce constat, les concepteurs de Spring ont développé un framework permettant d'adresser ces problématiques de manière efficace.

Réponses de Spring

Pour résoudre les problèmes que nous venons d'évoquer, des solutions ont émergé. En rupture avec les conteneurs J2EE, disqualifiés par de nombreux experts pour leur lourdeur, sont apparus des conteneurs dits légers. Le cœur de Spring entre dans cette catégorie de solutions.

Ce cœur a été étendu de manière à supporter la POA (programmation orientée aspect), ou AOP (Aspect Oriented Programming), un nouveau paradigme de programmation permettant d'aller au-delà de l'approche objet en terme de modularisation des composants. Au-dessus de ce socle, des modules optionnels sont proposés afin de faciliter l'intégration de frameworks spécialisés.

Les sections qui suivent détaillent la façon dont Spring résout les problèmes soulevés par l'utilisation de J2EE.

La notion de conteneur léger

La notion de conteneur EJB est une technologie lourde à mettre en œuvre, inadaptée pour les objets à faible granularité, intrusive dans le code et en partie dépendante du serveur d'applications, ne serait-ce qu'en terme de configuration.

En fournissant une infrastructure de gestion de l'ensemble des composants de l'application, les conteneurs légers adoptent une approche foncièrement différente. Cela s'effectue au travers de mécanismes de configuration, comme les fichiers XML permettant d'initialiser les composants, de gestion du cycle de vie des composants et de gestion des dépendances entre les composants. Les conteneurs légers sont indépendants de la technologie J2EE et peuvent fonctionner sur tout type de serveur d'applications, voire sans eux en utilisant une simple machine virtuelle Java.

Les conteneurs légers rendent les applications qui les utilisent à la fois plus flexibles et mieux testables, car le couplage entre les composants est géré par le conteneur, et non plus directement à l'intérieur du code. Par ailleurs, les conteneurs légers encouragent l'utilisation intensive d'interfaces afin de rendre l'application indépendante de leurs implémentations. Cela se révèle notamment utile pour les tests, dans lesquels il est souvent nécessaire de remplacer une implémentation réelle par un simulacre d'objet, ou Mock Object. Le débogage est facilité, puisqu'il n'est pas nécessaire d'exécuter le code au sein d'un serveur d'applications et que le conteneur léger est peu intrusif dans le code de l'application.

Pour nous faire une idée plus précise de la nature d'un conteneur léger, nous pouvons établir un parallèle avec les frameworks gérant les interfaces graphiques, comme Swing ou SWT (Standard Widget Toolkit). Avec ces outils, nous définissons les composants graphiques à utiliser et nous nous connectons aux événements qu'ils sont susceptibles de générer, un clic sur un bouton, par exemple. Ces frameworks prennent en charge le cycle de vie des composants graphiques de manière complètement transparente pour l'application concernée. Un conteneur léger peut offrir des services similaires, mais avec des objets de toute nature.

Nous détaillons au chapitre 2 le fonctionnement des conteneurs légers et abordons les modèles de conception, ou design patterns, sur lesquels ils se fondent.

À la différence d'autres conteneurs légers du marché, notamment HiveMind ou PicoContainer, Spring propose bien d'autres fonctionnalités, comme le support de la POA ou l'intégration de frameworks tiers.

Le support de la POA

Outre son conteneur léger, Spring supporte la POA (programmation orientée aspect), ou AOP (Aspect-Oriented Programming). Ce support est utilisé en interne par Spring pour offrir des services similaires à ceux des conteneurs EJB, mais sans leur lourdeur, car ils sont applicables à de simples JavaBeans, ou POJO (Plain Old Java Objects). Ce support peut aussi être utilisé par les utilisateurs de Spring pour leurs propres besoins.

La POA est un nouveau paradigme de programmation, dont les fondations ont été définies au centre de recherche Xerox, à Palo Alto, au milieu des années 1990. Par paradigme, nous entendons un ensemble de principes qui structurent la manière de modéliser les applications et, en conséquence, la façon de les développer.

Elle a émergé à la suite de différents travaux de recherche, dont l'objectif était d'améliorer la modularité des logiciels afin de faciliter la réutilisation et la maintenance.

La POA ne remet pas en cause les autres paradigmes de programmation, comme l'approche procédurale ou l'approche objet, mais les étend en offrant des mécanismes complémentaires, afin de mieux séparer les différentes préoccupations d'une application, et une nouvelle dimension de modularisation, l'*aspect*.

Dans son principe, la POA consiste à modulariser les éléments logiciels mal pris en compte par les paradigmes classiques de programmation. En l'occurrence, la POA se concentre sur les éléments transversaux, c'est-à-dire ceux qui se trouvent dupliqués ou utilisés dans un grand nombre d'entités, comme les classes ou les méthodes, sans pouvoir être centralisés au sein d'une entité unique avec les concepts classiques. Ainsi, grâce à la notion d'aspects, qui capturent en leur sein les préoccupations transversales, la séparation des préoccupations est nettement améliorée.

Avec les EJB, notamment les EJB Entity CMP, un premier niveau de séparation des préoccupations a été atteint. Un grand nombre d'aspects techniques sont en effet pris en charge sous forme de descripteurs de déploiement (mapping objet-relationnel, sécurité et transactions, etc.) et n'apparaissent plus dans le code métier. La maintenance en est d'autant facilitée.

La POA va au-delà en offrant des mécanismes génériques pour modulariser un grand nombre d'éléments transversaux des logiciels. Les limites des outils de POA pour capturer les préoccupations transversales tiennent essentiellement à leur capacité à exprimer le périmètre de celles-ci et à la façon dont elles sont liées au reste de l'application.

Spring n'implémente pas toutes les notions de la POA, mais l'essentiel est présent et facilement utilisable. Pour des besoins plus poussés, l'utilisation d'un framework tiers est permise. Spring s'intègre notamment avec AspectJ, l'outil précurseur de la POA et qui demeure le plus populaire à ce jour, mais cette intégration se limite aux aspects de type singleton.

Diverses fonctionnalités avancées de Spring reposent sur la POA, notamment les suivantes :

- gestion des transactions ;
- gestion de cache ;
- notification d'événements.

Grâce à la POA, Spring offre des services proches de ceux fournis par J2EE, mais sans se limiter aux seuls EJB. Ces services peuvent de la sorte être utilisés avec tout objet Java.

L'intégration de frameworks tiers

L'intégration de frameworks tiers dans Spring repose sur la notion de conteneur léger et, dans une moindre mesure, sur la POA.

Cette intégration peut s'effectuer aux trois niveaux suivants :

- intégration des composants du framework tiers au sein du conteneur léger et configuration de ses ressources ;
- mise à disposition d'un template, ou modèle, d'utilisation et de classes de support ;
- abstraction de l'API.

En fonction du framework, l'intégration de celui-ci est plus ou moins poussée, le minimum étant l'intégration des composants du framework dans le conteneur léger. Cette intégration consiste à configurer et réaliser une injection de dépendances sur ces composants. Elle est généralement peu intrusive du point de vue de l'application qui utilise le framework tiers, c'est-à-dire qu'aucune dépendance forte n'est créée à l'égard de Spring.

Le deuxième degré d'intégration consiste en la mise à disposition d'un template d'utilisation et de classes de support. Il s'agit ici de faciliter l'utilisation du framework tiers en simplifiant les appels à son API et en implémentant les meilleures pratiques.

Pour cela, différentes techniques sont utilisées. L'une d'elles consiste en la transformation systématique des exceptions vérifiées *(checked exceptions)*, c'est-à-dire les exceptions devant être obligatoirement prises en charge par une clause `catch` dans le code et les clauses `throws` spécifiées dans les signatures des méthodes, en exceptions d'exécution *(runtime exceptions)*, plus simples à gérer de manière centralisée. Dans ce cas, l'utilisation d'un template lie le code de l'application explicitement à Spring.

Le troisième degré d'intégration consiste à s'abstraire de l'API du framework tiers. L'objectif est ici de normaliser l'utilisation d'un ensemble de frameworks répondant à des besoins similaires. C'est typiquement le cas du support des DAO (Data Access Objects), ou objets d'accès aux données, par Spring, qui normalise l'utilisation de ce concept quel que soit le framework de persistance utilisé (Hibernate, OJB, etc.).

L'abstraction d'API ne permet toutefois pas de s'abstraire complètement des frameworks sous-jacents, car cela reviendrait à choisir le plus petit dénominateur commun et à perdre ainsi la richesse des fonctionnalités avancées de ces frameworks. Le parti pris de Spring consiste à abstraire les points communs sans pour autant masquer le framework tiers. C'est un choix pragmatique, qui diminue les coûts de remplacement d'un framework par un autre, sans occulter les gains de productivité induits par leur utilisation poussée.

Grâce à ces différents degrés d'intégration, Spring constitue le ciment permettant de lier les différents frameworks mis en œuvre avec le code applicatif dans un ensemble cohérent et implémentant les meilleures pratiques. La productivité et la maintenabilité des applications en sont d'autant améliorées.

Architecture globale de Spring

À partir de ces trois principes que sont le conteneur léger, la POA et l'intégration de frameworks tiers, Spring propose un cadre de développement permettant de construire des applications de manière modulaire.

La figure 1.1 illustre les différentes briques constituant l'architecture de Spring, ainsi que les principaux frameworks tiers (représentés en pointillés) dont elles assurent l'intégration.

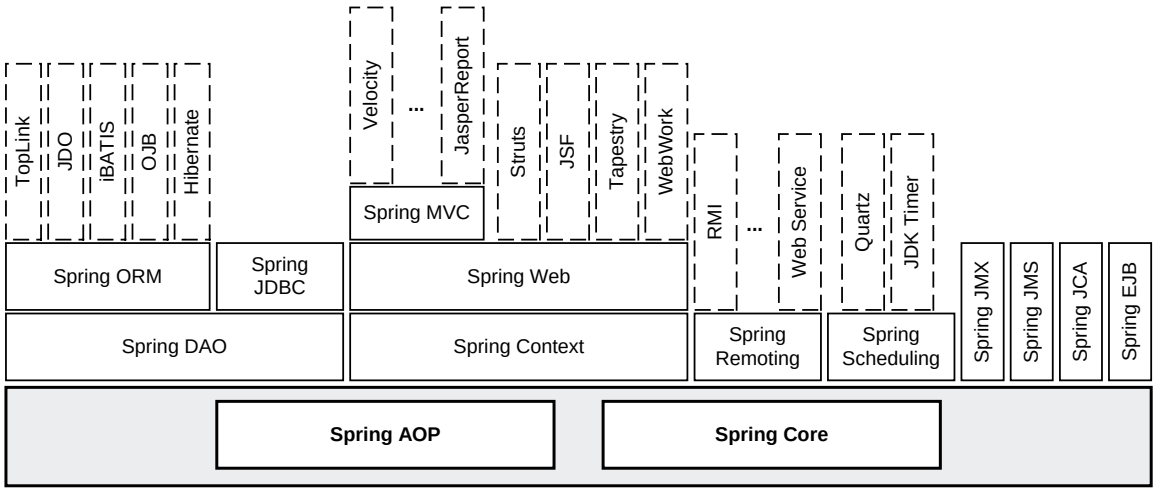


Figure 1.1
Architecture de Spring

Comme nous pouvons le voir, Spring repose sur un socle technique constitué des modules :

- Spring Core, le conteneur léger ;
- Spring AOP, le framework de POA.

Sur ce socle sont construits des modules de plus haut niveau destinés à intégrer des frameworks tiers ou à fournir des fonctions de support. Ces modules sont les suivants :

- Modules d'intégration de la persistance des données (Spring DAO pour l'abstraction de la notion d'objets d'accès aux données, Spring ORM pour l'intégration de frameworks de persistance, Spring JDBC pour simplifier l'utilisation de JDBC). Les principaux frameworks de persistance du marché sont supportés.
- Module de gestion de contexte applicatif (Spring Context), qui assure le dialogue entre Spring et l'application, indépendamment de la plate-forme technique sur laquelle fonctionne cette dernière (au sein d'un serveur d'applications, d'une simple JVM, etc.).

- Module d'intégration de frameworks Web (Spring Web), qui supporte les principaux frameworks Open Source du marché. Spring propose en outre son propre framework Web, conçu sur le modèle MVC, sous le nom de Spring MVC.
- Module de distribution d'objets (Spring Remoting), qui permet de transformer de simples classes Java en objets distribués RMI, Web Services ou autres.
- Module d'intégration d'ordonnanceur de tâches, qui supporte actuellement Quartz d'OpenSymphony et le Timer fourni par le JDK.
- Modules de support des différentes technologies J2EE (JMX, JMS, JCA et EJB).

Comme nous pouvons le constater, l'offre de Spring couvre l'essentiel des frameworks utilisés par les applications J2EE actuelles, ce qui en fait une solution pertinente pour tout développement. De plus, Spring n'est pas une solution fermée, comme nous le verrons à la section suivante.

Spring propose enfin les sous-projets suivants, qui viennent compléter ses fonctionnalités :

- Acegi Security System for Spring, qui permet de gérer l'authentification et les autorisations au sein d'une application développée avec Spring.
- Spring Web Flow, qui permet de gérer la navigation au sein de pages Web.
- Spring Rich Client, qui permet d'accélérer le développement d'applications Swing en se fondant sur Spring.
- Spring BeanDoc, qui permet de documenter l'utilisation des artefacts de Spring utilisés dans les applications.
- Spring IDE, un plug-in Eclipse qui permet d'accélérer les développements en gérant les composants de Spring au sein de cet environnement de développement.

Le projet Spring Modules (voir *https://springmodules.dev.java.net/*) a pour objectif d'intégrer d'autres frameworks et outils sans modifier le cœur de Spring. À ce jour, Spring Modules intègre des frameworks tels que OSWorkflow, Commons Validator, Drools, etc.

Spring dispose aussi d'une implémentation pour le monde Microsoft, avec Spring.Net *(http://www.springframework.net/).*

En résumé

Spring est une solution élégante qui répond à un ensemble de problématiques essentielles des développements Java/J2EE, notamment la flexibilité du code, l'intégration de frameworks tiers et l'implémentation des meilleures pratiques de programmation.

Du fait de son indépendance vis-à-vis de la plate-forme J2EE, il est utilisable sur tout type de développement Java reposant sur J2SE. Il convient aussi bien aux applications de type Web (grâce à Spring MVC ou à son intégration des principaux frameworks MVC) ou Swing (notamment grâce au sous-projet Spring Rich Client).

Certains reprochent à Spring de ne pas être un standard, à la différence de J2EE. Force est pourtant de constater qu'un développement J2EE est souvent difficile à migrer d'un serveur d'applications à un autre, alors que les développements qui utilisent Spring et s'abstiennent d'utiliser les EJB le sont beaucoup moins. Par ailleurs, Spring laisse le choix au développeur du niveau d'adhérence de son code vis-à-vis du framework, ce qui est moins le cas de J2EE. Par exemple, en n'utilisant que le conteneur léger et la POA, l'impact du remplacement de Spring par une autre solution est relativement faible.

Présentation de l'étude de cas Tudu Lists

Tout au long de l'ouvrage, nous illustrons notre propos au moyen d'une étude de cas, Tudu Lists, qui emploie les fonctionnalités majeures de Spring. Au travers de cette étude de cas, nous donnons une vision globale de l'architecture d'une application fondée sur Spring, ainsi que de nombreux exemples de mise en œuvre.

Dans le contexte de cet ouvrage, nous utilisons une préversion 2.0 de Spring, la version finale n'étant pas disponible au moment où nous mettons sous presse. L'API de Spring étant remarquablement stable, la version finale ne devrait pas introduire de dysfonctionnements majeurs. Si tel devait néanmoins être le cas, un addendum précisant les changements à effectuer serait mis à la disposition des lecteurs sur la page Web dédiée à l'ouvrage du site des éditions Eyrolles, à l'adresse *http://www.editions-eyrolles.com.*

Tudu Lists est un projet Open Source créé par Julien Dubois, l'un des auteurs du présent ouvrage, et auquel les autres auteurs de l'ouvrage participent. Ce projet consiste en un système de gestion de listes de choses à faire *(todo lists)* sur le Web. Il permet de partager des listes entre plusieurs utilisateurs et supporte le protocole RSS (Really Simple Syndication). Les listes de choses à faire sont des outils de gestion de projet simples mais efficaces. Tudu Lists est hébergé sur le site communautaire SourceForge *(http://tudu.sourceforge.net/)* pour sa partie développement.

L'utilisation de Tudu Lists est très simple comme nous allons le voir. La page d'accueil se présente de la manière illustrée à la figure 1.2.

Pour créer un nouvel utilisateur, il suffit de cliquer sur le lien « register » et de remplir le formulaire. Une fois authentifié, l'utilisateur peut gérer ses listes de todos.

Par commodité, nous utilisons à partir de maintenant le terme « todo » pour désigner une chose à faire, comme illustré à la figure 1.3.

Pa le biais des onglets disposés en haut de la page, nous pouvons gérer notre compte (My info), nos listes de todos (My Todo Lists), nos todos (My Todos) ou nous déloguer (Log out).

La création d'une liste est on ne peut plus simple. Il suffit de cliquer sur l'onglet My Todo Lists puis sur le lien Add a new Todo List et de remplir le formulaire et cliquer sur le lien Submit, comme illustré à la figure 1.4.

La création d'un todo suit le même principe. Dans l'onglet My Todos, il suffit de cliquer sur la liste dans laquelle nous allons ajouter un todo (les listes sont affichées dans la partie gauche de la page) puis de cliquer sur Add a new Todo.

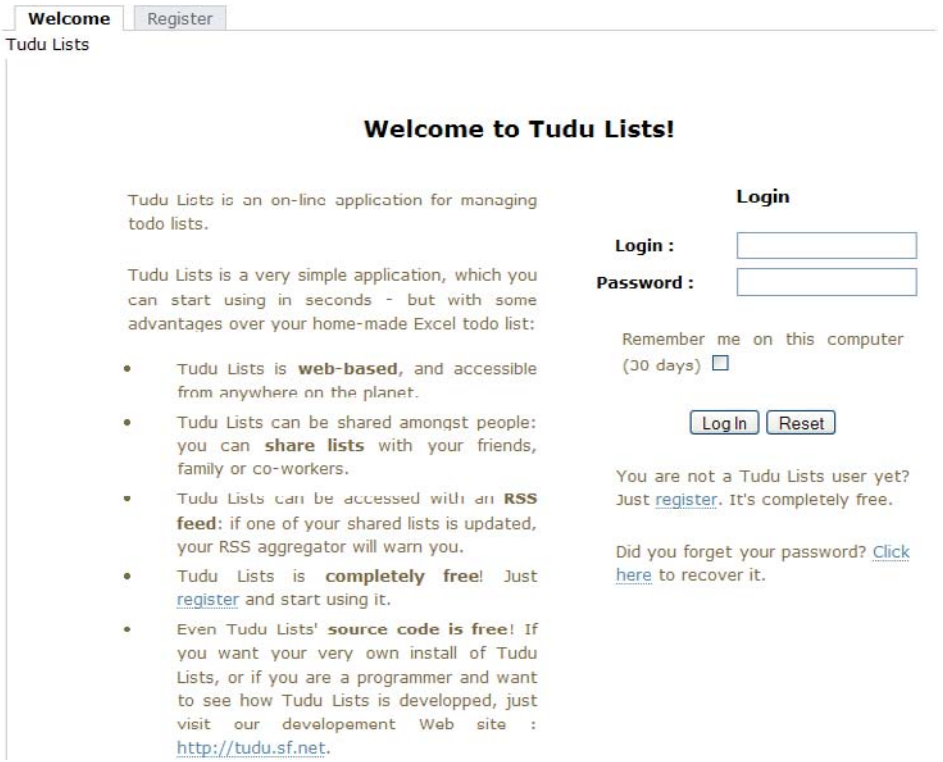


Figure 1.2
Page d'accueil de Tudu Lists

Figure 1.3
Liste de todos

Figure 1.4
Création d'une liste de todos

Pour finir, nous pouvons sauvegarder le contenu d'une liste en cliquant sur le lien Backup. De même, nous pouvons restaurer le contenu d'une liste en cliquant sur le lien Restore.

Nous avons vu l'essentiel des fonctions de Tudu Lists. Pour une description plus détaillée de son utilisation, voir la documentation en ligne, sur le site *http://tudu.sourceforge.net.*

Architecture du projet Tudu Lists

Tudu Lists est une application Web, conçue pour démontrer que l'utilisation de Spring et de frameworks spécialisés permet d'obtenir sans développement lourd une application correspondant à l'état de l'art en terme de technologie.

Dans cette section, nous décrivons de manière synthétique les principes architecturaux de Tudu Lists. Ces informations seront utiles pour manipuler l'étude de cas tout au long de l'ouvrage.

Nous listons dans un premier temps les frameworks utilisés par Tudu Lists puis abordons la gestion des données, notamment la modélisation de ces dernières. Nous terminons avec les principes de la modélisation objet de Tudu Lists.

Les frameworks utilisés

Outre Spring, Tudu Lists utilise les frameworks Open Source suivants :

- Hibernate, le célèbre framework de mapping objet-relationnel, pour assurer la persistance des données.
- Struts, pour assurer l'implémentation du modèle de conception MVC. Dans le cadre de cet ouvrage, nous proposons aussi une version spéciale de Tudu Lists utilisant Spring MVC en lieu et place de Struts.
- Struts Menu, pour gérer les menus de l'application.
- DWR (Direct Web Remoting) pour implémenter les fonctionnalités AJAX (Asynchronous JavaScript And XML) de Tudu Lists. AJAX est une technologie fondée sur JavaScript destinée à améliorer l'expérience de l'utilisateur devant une interface homme-machine (IHM) en limitant les échanges entre le navigateur Web et le serveur d'applications à des messages XML au lieu de pages Web complètes. Nous aurons l'occasion de décrire plus en détail cette technologie au chapitre 9, consacré à AJAX et DWR.
- Acegi Security, pour gérer les authentifications et les autorisations.
- JAMon (Java Application Monitor), pour surveiller les performances de Tudu Lists.
- Log4J, pour gérer les traces applicatives.
- Rome, pour gérer les flux RSS.

Comme nous le constatons, bien que Tudu Lists implémente des fonctionnalités simples à appréhender, il met en œuvre un grand nombre de frameworks.

Grâce aux services rendus par ces différents frameworks et à l'intégration assurée par Spring, le nombre de lignes de code de Tudu Lists reste raisonnable (3 200 lignes pour

une quarantaine de classes et interfaces, avec une moyenne de 6 lignes par méthode) en comparaison de l'équivalent développé uniquement avec J2EE.

Modèle conceptuel de données

Tudu Lists utilise une base de données relationnelle pour stocker les informations sur les utilisateurs et les listes de todos.

Le modèle conceptuel de données de Tudu Lists s'articule autour de quelques tables, comme l'illustre la figure 1.5.

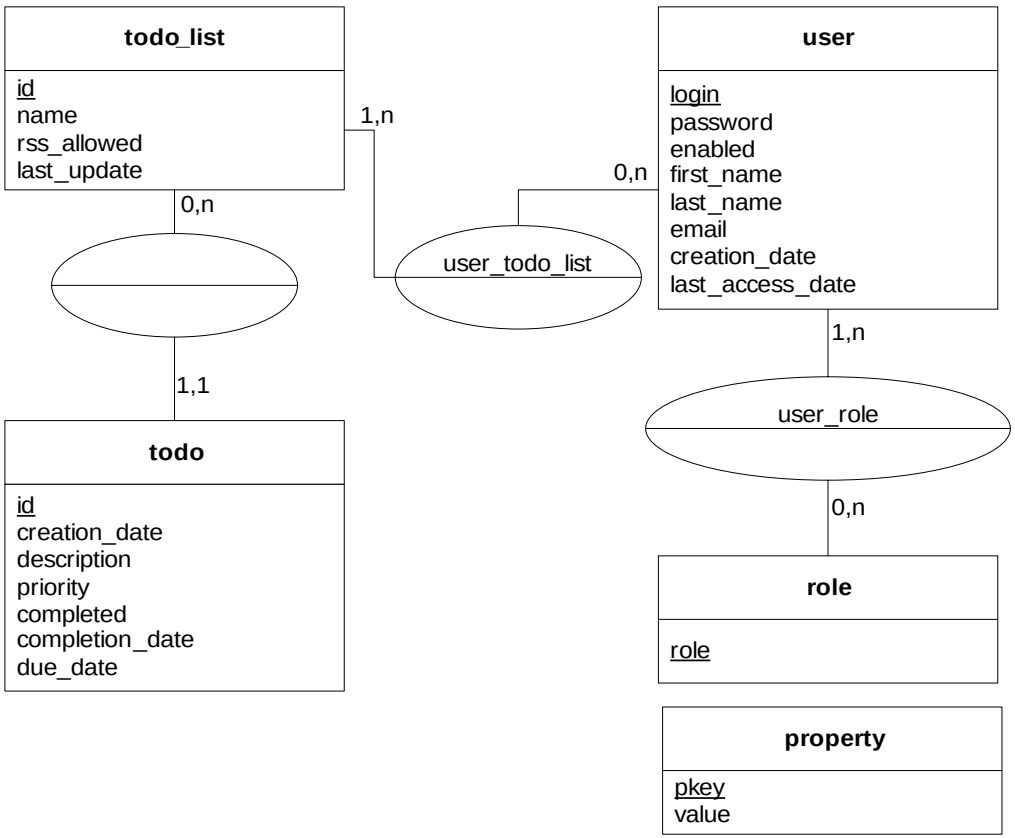


Figure 1.5

Modèle conceptuel de données de Tudu Lists

Remarquons que la table `property` sert à stocker les paramètres internes de Tudu Lists, notamment l'adresse du serveur SMTP nécessaire à l'envoi d'e-mail. Par ailleurs, la table `role` ne contient que deux lignes, chacune représentant les deux rôles gérés par Tudu Lists : administrateur et utilisateur.

Pour implémenter ce modèle de données, nous utilisons le SGBDR Open Source HSQLDB *(http://hsqldb.org/).* À l'origine, Tudu Lists était prévu pour fonctionner avec le SGBDR MySQL, mais nous avons remplacé ce dernier pour simplifier l'installation de l'étude de cas. En effet, HSQLDB est un SGBDR écrit en Java, dont l'exécution peut se faire au sein même de l'application Web. Il est donc directement embarqué dans l'application, simplifiant grandement la problématique du déploiement.

Pour utiliser Tudu Lists dans un environnement de production, nous conseillons d'utiliser la version de MySQL disponible sur le site de Tudu Lists, car ce SGBDR est prévu pour supporter de fortes charges, contrairement à HSQLDB.

HSQLDB et MySQL

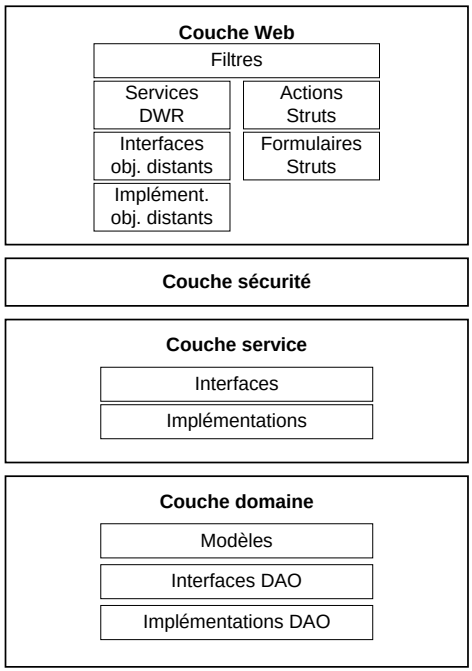
Le remplacement de MySQL est facilité par l'architecture de Tudu Lists. L'essentiel du travail consiste à adapter le script SQL de création des tables, celui de MySQL contenant des spécificités propres à ce SGBDR. Il suffit d'apporter ensuite des modifications à quelques fichiers de configuration portant sur la définition de la source de données et le réglage du dialecte SQL utilisé par Hibernate.

Les principes du modèle objet

Nous ne détaillons pas ici le modèle objet de Tudu Lists, car il est décrit tout au long des chapitres de l'ouvrage. Ce modèle suit la philosophie de conception préconisée par Spring, à savoir un découpage clair des couches composant l'application.

La figure 1.6 illustre les couches de Tudu Lists.

Figure 1.6
Couches de Tudu Lists



La couche domaine

La couche domaine prend en charge le modèle objet métier de Tudu Lists. C'est dans cette couche que nous trouvons les classes modélisant et implémentant les notions de liste de todos, de todos, etc.

Cette couche est constituée de trois sous-ensembles :

- Le sous-ensemble des modèles contient les objets qui sont manipulés par les autres couches de l'application. C'est dans ce sous-ensemble, offrant le plus haut niveau d'abstraction, que se trouvent les objets métier.
- Le sous-ensemble des interfaces DAO contient les interfaces des DAO (Data Access Objects), qui spécifient les services destinés à assurer la persistance des objets métier.
- Le sous-ensemble des implémentations DAO contient les implémentations des interfaces précédentes. Dans le cadre de l'étude de cas, nous fournissons une implémentation réalisée avec Hibernate.

L'utilisation des DAO est ainsi conforme au principe de Spring visant à séparer les interfaces des implémentations. Bien entendu, les DAO de Tudu Lists reposent sur Spring DAO et Spring ORM.

La couche service

La couche service contient les services permettant de manipuler les objets métier du modèle. Une nouvelle fois, les interfaces et leurs implémentations sont clairement séparées.

Les services sont au nombre de quatre :

- configuration de Tudu Lists ;
- gestion des utilisateurs ;
- gestion des listes de todos ;
- gestion des todos.

La couche sécurité

La couche sécurité est assurée par le framework Acegi Security for Spring. Pour Tudu Lists, cette couche se résume à fournir une implémentation d'un DAO permettant à Acegi Security d'accéder aux informations concernant les utilisateurs et de paramétrer le framework *via* un fichier de configuration Spring.

La couche Web

La couche Web est constituée de plusieurs sous-ensembles :

- Le sous-ensemble des filtres contient, comme son nom l'indique, les filtres (au sens J2EE du terme) utilisés pour analyser les requêtes HTTP. Tudu Lists possède les filtres suivants : filtre pour JAMon, filtre « session-in-view » et filtre Acegi Security.

- Les sous-ensembles des Actions Struts et des formulaires Struts contiennent les composants Web nécessaires au framework Struts pour gérer l'IHM de Tudu Lists.
- Les sous-ensembles des services DWR des interfaces des objets distants et des implémentations des objets distants contiennent respectivement les services AJAX et les interfaces et implémentations des objets métier disponibles pour ces services.

Installation de l'environnement de développement

Pour les besoins de l'étude de cas, nous avons limité au maximum les logiciels nécessaires à son fonctionnement.

Tudu Lists a simplement besoin des suivants :

- Java 5
- Eclipse 3.1+, avec WTP 1.0.1+ (Web Tools Platform)
- Tomcat 5.5+

Nous fournissons en annexe les instructions nécessaires à leur installation.

Nous supposons dans la suite de cette section que ces outils sont correctement installés sur le poste de travail.

Installation de l'étude de cas

L'étude de cas se présente sous la forme de cinq fichiers Zip, chacun contenant un projet Eclipse complet et indépendant des autres :

- **Tudu.zip :** ce fichier contient le projet qui est utilisé pour la majorité des chapitres de cet ouvrage. Il s'agit d'une version légèrement adaptée de Tudu Lists.
- **Tudu-SpringMVC.zip :** ce fichier contient le projet utilisé par les chapitres 7, 13 et 16. Il s'agit d'une version de Tudu Lists utilisant le framework Spring MVC au lieu de Struts.
- **Tudu-WebFlow.zip :** ce fichier contient le projet utilisé par le chapitre 8. Il s'agit d'une version de Tudu Lists utilisant Spring Web Flow pour gérer sa couche présentation.
- **Tudu-Portlet.zip :** ce fichier contient le projet utilisé au chapitre 10. Il s'agit d'une version de Tudu Lists transformée en portlet utilisable au sein d'un portail J2EE.
- **Tudu-Client.zip :** ce fichier contient le projet utilisé au chapitre 14. Il s'agit d'une application Java Swing capable de communiquer avec l'application Web Tudu Lists en utilisant un Web Service.

Il est possible de télécharger ces fichiers à partir soit de la page Web dédiée à l'ouvrage du site des éditions Eyrolles *(http://www.editions-eyrolles.com).*

L'installation des quatre premiers projets au sein d'Eclipse suit la même procédure quel que soit le fichier. Nous décrivons ci-après l'installation du projet contenu dans le fichier **Tudu.zip :**

1. Une fois le fichier téléchargé, lancer Eclipse, puis créer un projet Web Dynamique par le biais du menu File/Project… en sélectionnant l'option Dynamic Web Project dans le dossier Web.
2. Dans la boîte de dialogue de création de projet, définir le nom du projet (Tudu en respectant la majuscule) puis le serveur d'applications cible (en l'occurrence Tomcat) en cliquant sur le bouton New. Le serveur d'applications doit impérativement être défini à cette étape, faute de quoi le projet ne se compilera pas. Il est inutile de cocher la case Add Project to an EAR.
3. Dans la boîte de dialogue permettant de spécifier les noms de répertoires du projet, changer **src** en **JavaSource.**
4. Importer le fichier Zip *via* les menus File et Import.
5. Dans la fenêtre d'importation, sélectionner le type Archive File, et cliquer sur le bouton Next.
6. Dans la fenêtre de sélection, cliquer sur le bouton Browse en haut de la fenêtre afin de sélectionner le fichier Zip sur le disque dur.
7. Cliquer sur le bouton Finish.
8. Pour recompiler le projet, sélectionner Project et Clean, puis choisir le projet Tudu et cocher la case Start a build immediately.
9. Si des problèmes de compilation apparaissent, s'assurer que le projet utilise bien le compilateur Java 5 (les propriétés du projet sont accessibles *via* son menu contextuel).
10. Pour terminer, exporter les fichiers **hsqldb-1.8.0.1.jar** et **jta.jar** depuis la racine du projet dans le répertoire **common/lib** du répertoire d'installation de Tomcat. Pour cela, sélectionner les fichiers dans la vue et utiliser le menu contextuel Export…

Une fois l'opération terminée, il est possible de configurer et d'exécuter chaque projet, comme expliqué à la section suivante.

Pour Tudu-Client, la procédure est plus simple. Il suffit d'utiliser le menu File/Import…, de sélectionner Existing Projects into Workspace et de cliquer sur Next. Dans la boîte de dialogue d'import, il faut ensuite sélectionner l'option Select archive file puis cliquer sur Browse… pour choisir le fichier **Tudu-Client.zip.** Pour achever, l'importation du projet, cliquer sur Finish.

Configuration et exécution de l'application

Une fois l'application Tudu Lists installée, il faut configurer la connexion à la base de données puis tester son bon fonctionnement. Le projet Tudu-Client ne contenant pas l'application Tudu Lists mais une simple application Swing, il n'est pas concerné par ce qui suit.

La base de données de Tudu Lists est installée dans le répertoire **db** de chaque projet Eclipse :

1. Pour connaître le chemin physique de ce répertoire sur le disque dur, sélectionner par clic droit le dossier **db,** et cliquer sur Properties dans le menu contextuel.
2. Dans la fenêtre qui s'affiche, le chemin physique est fourni sous le nom Location. Mémoriser ce chemin, car il sera nécessaire pour configurer la connexion à la base.
3. Fermer la fenêtre.
4. Pour configurer la connexion, éditer le fichier **context.xml,** qui se trouve dans le répertoire **WebContent/META-INF** du projet. Ce fichier se présente de la manière suivante :

```
<?xml version='1.0' encoding='utf-8'?>
<Context>
<Resource auth="Container" description="Tudu database"
    driverClassName="org.hsqldb.jdbcDriver"
    maxActive="100" maxIdle="30"
    maxWait="10000" name="jdbc/tudu" password=""
    type="javax.sql.DataSource"
    url="jdbc:hsqldb:C:\Eclipse\workspace\tudu\db\tudu"
    username="sa"
/>
</Context>
```

5. Remplacer le texte en gras par le chemin physique de la **db** en veillant à ne pas supprimer le tudu final.

Nous pouvons maintenant exécuter Tudu Lists dans Tomcat (les étapes qui suivent ne sont pas valables pour le projet Tudu-Portlet, qui doit s'exécuter dans un portail ; la procédure dépendant du portail utilisé, nous ne la décrivons pas) :

1. Pour lancer le projet Tudu-SpringMVC, il est nécessaire de lancer préalablement le script **build.xml,** qui se trouve à la racine du projet. Pour cela, il suffit de le sélectionner dans la vue Package Explorer et d'utiliser le menu contextuel Run As/Ant Build. Ce script lance le logiciel ActiveMQ, nécessaire spécifiquement pour ce projet.
2. Pour lancer Tomcat, accéder au menu contextuel du projet par clic droit sur son dossier dans la vue Package Explorer.
3. Dans ce menu, sélectionner Run As et Run on Server puis Tomcat dans la liste des serveurs disponibles et cliquer sur Next.
4. Il convient de s'assurer que le projet se trouve bien dans la zone Configured Projects en utilisant au besoin le bouton Add et en cliquant sur Finish.

Tomcat s'exécute, et une fenêtre de navigation Web s'affiche dans Eclipse avec la page d'accueil de Tudu Lists *(voir figure 1.2)*. Dans le cas du projet Spring-WebFlow, l'interface présentée a été fortement simplifiée pour les besoins du chapitre 8.

Nous pouvons maintenant nous connecter à Tudu Lists en utilisant l'identifiant **test** et le mot de passe **test.**

Organisation des projets dans Eclipse

La figure 1.7 illustre l'organisation des projets (à l'exception de Tudu-Client) telle qu'elle doit apparaître dans l'environnement Eclipse.

Figure 1.7
Organisation des projets Tudu Lists dans Eclipse

Le répertoire **JavaSource** contient l'ensemble du code de Tudu Lists. Il comprend aussi les fichiers de configuration d'Hibernate et de Log4J.

Le répertoire **Tests** contient l'ensemble des tests unitaires de Tudu Lists. Le répertoire **db** contient pour sa part la base de données de l'application.

Le répertoire **WebContent** dispose de plusieurs sous-répertoires :

- **css** contient la feuille de style utilisée par Tudu Lists.
- **images** contient l'ensemble des images du projet.
- **META-INF** contient le fichier de paramétrage de la connexion à la base de données.
- **scripts** contient les fichiers JavaScript nécessaires à l'IHM.
- **WEB-INF** contient les fichiers de configuration de l'application Web, ainsi que les pages et fragments JSP (respectivement dans les sous-répertoires **jsp** et **jspf**) et les taglibs (sous-répertoire **tld**).

Conclusion

Ce chapitre a introduit les grands principes de fonctionnement de Spring et décrit ses apports fondamentaux au développement d'applications Java/J2EE, que ces dernières utilisent ou non des frameworks tiers. Il a aussi présenté l'étude de cas Tudu Lists, qui guidera le lecteur tout au long de sa progression dans l'ouvrage.

Au cours des deux chapitres suivants, nous plongeons au cœur de Spring afin de mettre en valeur les notions essentielles de conteneur léger et de POA, qui en forment l'ossature.

Le reste de l'ouvrage est consacré aux principales intégrations de frameworks tiers proposées par Spring et aux outils permettant d'accélérer les développements avec Spring.

Partie I

Les fondations de Spring

Comme indiqué au chapitre 1, Spring est bâti sur deux piliers : son conteneur léger et son framework de POA.

Pour utiliser Spring de manière efficace dans nos développements et bénéficier pleinement de sa puissance, il est fondamental de bien comprendre les concepts liés à ces deux piliers et de modéliser nos applications en conséquence.

L'objectif de cette partie est double : fournir les bases conceptuelles nécessaires à la bonne mise en œuvre de Spring et initier à l'utilisation de son conteneur léger et de son framework de POA.

Le chapitre 2 fournit les bases conceptuelles des conteneurs légers et illustre leur valeur ajoutée au travers d'une étude de cas fondée sur l'application Tudu Lists.

Le chapitre 3 fait le lien entre les aspects fondamentaux des conteneurs légers et leur implémentation par Spring.

Sur le modèle du chapitre 2, le chapitre 4 fournit les bases conceptuelles de la POA en reprenant l'étude de cas précédente. L'objectif de ce chapitre n'est pas d'être exhaustif sur le sujet mais de présenter les notions essentielles de la POA.

Construit sur le modèle du chapitre 3, le chapitre 5 montre comment Spring implémente les notions essentielles de la POA et en tire parti pour fournir aux applications des fonctionnalités à valeur ajoutée sans intrusivité.

2

Les concepts des conteneurs légers

Les conteneurs légers sont devenus très populaires ces dernières années. Au chapitre précédent, nous avons donné les principales raisons expliquant cet engouement. Dans le présent chapitre, nous décrivons les concepts qui sous-tendent les conteneurs légers et font tout leur intérêt par rapport aux techniques classiques de développement.

Pour bien comprendre l'intérêt des conteneurs légers, nous abordons la conception d'un modèle métier simplifié de l'application Tudu Lists selon différents points de vue conceptuels, allant du plus naïf au plus évolué, en faisant ressortir leurs faiblesses respectives.

Nous verrons ensuite comment les conteneurs légers répondent aux problèmes de conception soulevés par ces différentes approches et détaillerons les notions d'inversion de contrôle et d'injection de dépendances, qui sont au fondement des conteneurs légers.

Problématiques de conception d'une application

Pour concevoir une application, plusieurs approches sont possibles. Une première approche consiste à se concentrer sur le besoin immédiat sans se préoccuper de l'avenir. C'est ce que nous appelons l'approche naïve.

Une approche plus mature consiste à utiliser les services offerts par la technologie pour gagner en productivité. Malheureusement, la technologie n'est pas suffisante en elle-même pour lutter contre les défauts de conception compromettant la pérennité de l'application. Nous nous plaçons volontairement ici dans le cadre d'applications à forte longévité.

La troisième approche respecte les règles de l'art en s'appuyant sur des modèles de conception, ou design patterns, éprouvés, tels ceux proposés par le Gang of Four

(Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) dans leur célèbre ouvrage *Design patterns : catalogue de modèles de conception réutilisables*.

Nous tenterons d'appliquer ces différentes approches pour créer une version volontairement simplifiée du modèle métier de Tudu Lists afin de faire ressortir les problèmes conceptuels qu'elles soulèvent.

Pour pouvoir analyser les défauts de ces approches, il est nécessaire de se doter de critères objectifs permettant de les comparer à un modèle idéal, lequel n'existe malheureusement pas.

Dans son article *The Depency Inversion Principle*, publié dans le magazine *C++ Report*, Robert Cecil Martin propose les trois critères suivants pour identifier les faiblesses d'une modélisation :

- *Rigidité :* les changements dans l'application sont difficiles à effectuer, car ils affectent trop de composants du système.
- *Fragilité :* lorsque des changements sont opérés dans l'application, des erreurs inattendues se produisent.
- *Immobilité :* il est difficile de réutiliser les composants dans une autre application, car ils ne peuvent fonctionner indépendamment les uns des autres.

Pour compléter ces trois critères, nous en utilisons un quatrième, *l'invérifiabilité*, qui pose que les composants du modèle ne peuvent être vérifiés selon différents niveaux de granularité (tests unitaires, tests d'intégration, tests des interfaces, etc.).

Périmètre de la modélisation

Avant de nous lancer dans la conception du modèle métier de Tudu Lists, commençons par décrire les besoins fonctionnels à couvrir. Il ne s'agit pas de concevoir l'ensemble de l'application mais de nous intéresser à un sous-ensemble de besoins suffisamment significatifs pour faire ressortir les faiblesses des différentes approches.

Nous allons modéliser les deux notions fonctionnelles de Tudu Lists, à savoir les todos et les listes de todos. Nous faisons ici abstraction de la notion d'utilisateur, de la sécurité, de la gestion des transactions, etc. Nous ne nous intéressons pas non plus aux problématiques d'IHM et nous concentrons exclusivement sur la modélisation des todos et de leurs listes, ainsi que des fonctions permettant de gérer ces deux entités.

Dans ce contexte fonctionnel simplifié à l'extrême, les todos se présentent sous la forme d'une structure de données comportant les informations suivantes :

- identifiant ;
- date de création ;
- description ;
- priorité ;
- indicateur de réalisation de la chose à faire ;
- date de réalisation prévue ;
- date de réalisation effective.

Les listes de todos ne comportent pour leur part qu'un identifiant, un nom et une liste de todos.

Le modèle métier devra proposer les fonctionnalités classiques de création, suppression, modification et consultation, généralement identifiées par l'acronyme CRUD (Create, Retrieve, Update, Delete).

Pour assurer la persistance de ce modèle, une base de données relationnelle est utilisée, dont la modélisation conceptuelle est illustrée à la figure 2.1.

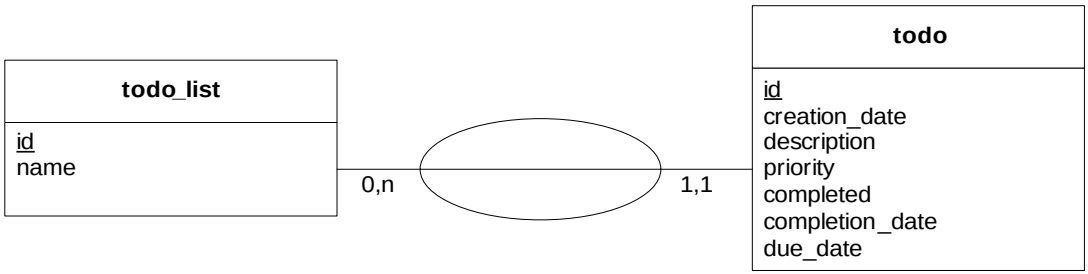


Figure 2.1
Modèle conceptuel de la base de données de Tudu Lists

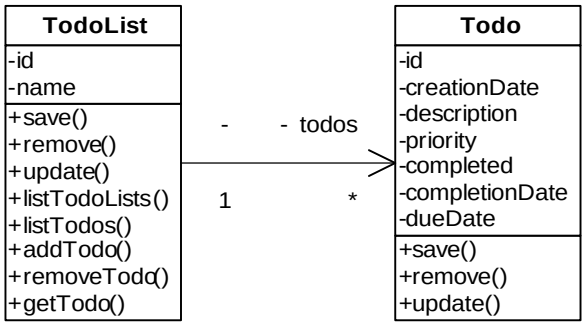
Approche naïve

Comme indiqué en début de chapitre, l'approche naïve consiste à se focaliser sur les besoins immédiats sans se préoccuper de l'avenir de l'application.

Modélisation

À partir du modèle conceptuel de la base de données, il est naturel pour cette approche de proposer un modèle métier comprenant deux classes calquant les entités `todo_list` et `todo`, comme l'illustre le diagramme de classes UML de la figure 2.2.

Figure 2.2
Diagramme de classes de l'approche naïve



Sur ce schéma UML simplifié, nous n'avons fait figurer ni les accesseurs (getters) et modificateurs (setters) des attributs des deux classes ni les constructeurs.

Dans cette conception, la classe `TodoList` est la porte d'accès aux deux notions fonctionnelles évoquées à la section précédente. Outre ses propres propriétés, `TodoList` pilote la classe `Todo` en lui sous-traitant les aspects strictement spécifiques aux todos.

Critique de la modélisation

Pour critiquer cette modélisation, utilisons les quatre critères d'analyse décrits précédemment :

- **Rigidité.** La classe `TodoList` est étroitement liée à la classe `Todo`, puisqu'elle prend directement en charge leur gestion (ajout, suppression, consultation). Ces opérations ensemblistes alourdissent le code de `TodoList` alors qu'elles pourraient être prises en charge par d'autres classes plus génériques, comme les `List` de l'API Java. Ce choix de conception impose une représentation interne du stockage des todos, rendant le code rigide dès lors que cette représentation ne s'avère plus adaptée. Par ailleurs, les fonctionnalités riches proposées par les deux classes encouragent les mauvaises pratiques. En l'occurrence, il est tentant de disperser dans les différentes couches de l'application finale l'appel aux méthodes de gestion des listes et de leurs todos. Ce type de mauvaise pratique rend le code plus rigide, car toute modification sur ces fonctionnalités impacte un grand nombre de composants.

- **Fragilité.** Nous constatons un mélange entre les notions strictement fonctionnelles que sont les todos et leurs listes, d'une part, et la problématique de leur gestion, en l'occurrence leur persistance, d'autre part. Cette absence de séparation des préoccupations aboutit à des effets de bord. Par exemple, si un todo n'est pas enregistré en base de données, il est nécessaire de passer en revue à la fois la gestion des todos dans la classe `TodoList` et le code de la classe `Todo`.

- **Immobilité.** Les classes `TodoList` et `Todo` supposent la présence d'une base de données relationnelle. Si le besoin exprimé aujourd'hui est bien d'avoir un SGBDR pour assurer la persistance de données, nous pouvons très bien imaginer une version de Tudu Lists pour téléphone mobile. Dans ce cas, l'utilisation d'un SGBDR n'est pas forcément le choix le plus judicieux, ce type de terminal ayant des capacités limitées. Il nous faut donc développer une nouvelle version de ces deux classes pour supporter ces nouveaux besoins, nous privant ainsi de toute réutilisation des autres services indépendants de la persistance proprement dite.

- **Invérifiabilité.** Il est clair qu'une conception aussi rigide, fragile et immobile ne facilite pas les tests. Les classes `TodoList` et `Todo` mélangeant différentes préoccupations fonctionnelles (manipulation des notions fonctionnelles) et techniques (persistance de données dans un SGBDR), les tests unitaires sont impossibles, puisque nous ne pouvons faire fonctionner ces deux classes indépendamment de l'infrastructure technique nécessaire à l'ensemble de l'application, en l'occurrence le SGBDR. La granularité la plus fine en terme de tests est fournie par les tests d'intégration, à la fois coûteux à réaliser et peu efficaces pour détecter les erreurs d'implémentation simples.

En résumé

Nous pouvons constater que l'approche naïve, bien que proposant une solution simple et immédiate à notre besoin de gérer des listes de todos, n'est pas satisfaisante du point de vue conceptuel, comme le montre l'analyse de nos quatre critères.

Cette modélisation s'avère rigide, fragile et immobile du fait d'une modularisation insuffisante des problématiques techniques. Cette mauvaise modularisation a un impact direct sur l'invérifiabilité de l'application, puisque ses composants ne peuvent être testés efficacement de manière unitaire.

Approche technologique

L'approche technologique va plus loin que l'approche naïve en ce qu'elle s'appuie sur les fonctionnalités de la plate-forme J2EE pour gagner en productivité dans l'implémentation des notions métier.

Fondamentalement, la conception reste la même mais prend en compte le modèle de composants métier de J2EE.

Modélisation

Les EJB (Enterprise JavaBeans) du standard J2EE ont pour vocation d'implémenter les objets métier d'une application. Il est donc naturel de s'y intéresser pour l'implémentation des todos et de leurs listes.

Les EJB peuvent être de différents types. Les EJB Session implémentent les traitements fonctionnant selon un mode conversationnel, avec ou sans état. Les EJB Entité implémentent les objets métier persistants. Quant aux EJB Message, ils implémentent les traitements fonctionnant en mode asynchrone.

Dans le cas qui nous intéresse, il est évident que les EJB Entité répondent à nos attentes. Le modèle de données étant simple, nous utilisons des EJB Entité CMP (Container Managed Persistence), dont la persistance est gérée par le conteneur, à la différence des EJB BMP (Bean Managed Persistence), gérés par programmation.

L'utilisation des EJB Entité aboutit au diagramme UML illustré à la figure 2.3.

Nous avons ici deux EJB Entité, que nous considérons comme étant de type CMP :

- `TodoListBean`, qui représente la notion fonctionnelle de liste de todos.
- `TodoBean`, qui représente la notion fonctionnelle de todo.

La principale évolution de ce modèle par rapport à celui de l'approche naïve est que le code des objets métier est expurgé des traitements liés à la persistance des données, ceux-ci étant pris en charge directement par le conteneur d'EJB du serveur d'applications.

La modélisation s'est complexifiée afin de supporter le fonctionnement du conteneur : les classes `TodoListBean` et `TodoBean` sont abstraites (contrainte imposée aux EJB CMP), et différentes interfaces doivent être définies pour chaque EJB afin de permettre aux clients d'y accéder.

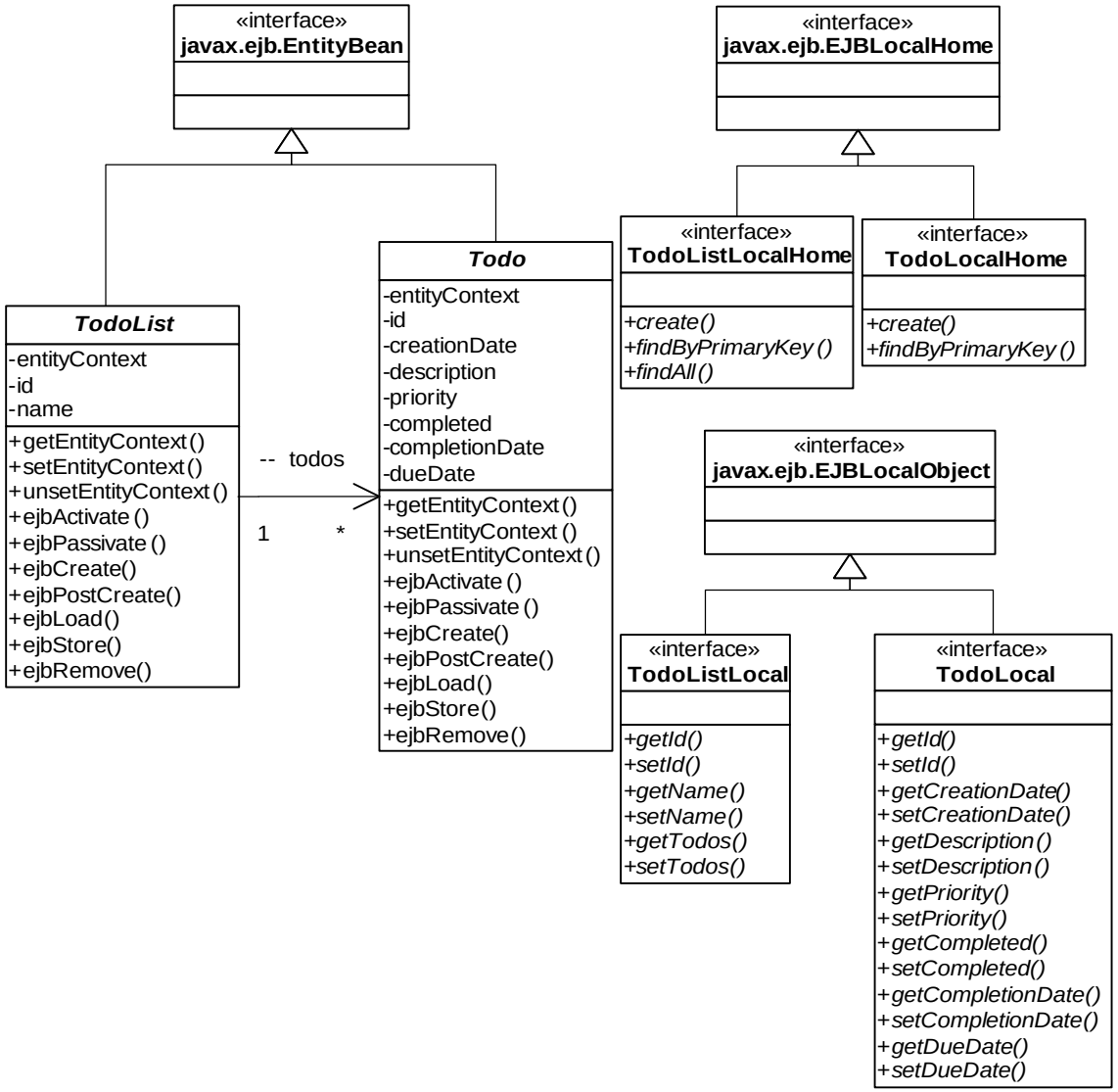


Figure 2.3
Diagramme de classes de l'approche technologique

Nous ne faisons apparaître ici que les interfaces locales. Les interfaces distantes sont similaires aux exceptions près et dérivent de `javax.ejb.EJBObject` et `javax.ejb.EJBHome`.

Pour bénéficier au maximum de la puissance des EJB, nous avons choisi de faire gérer par le conteneur la relation entre l'EJB `TodoList` et l'EJB `Todo`.

Critique de la modélisation

Comme indiqué précédemment, l'approche technologique ne révolutionne pas la modélisation naïve mais solutionne un certain nombre de problèmes en déléguant la gestion de la persistance au conteneur d'EJB.

À l'égard de nos quatre critères d'analyse, nous pouvons adresser les critiques suivantes à la modélisation :

- **Rigidité.** La situation est améliorée par rapport à la précédente, car la gestion de la relation entre `TodoListBean` et `TodoBean` est directement prise en charge par le conteneur d'EJB, allégeant ainsi le code de `TodoListBean`. Les todos sont représentés dans la classe `TodoListBean` sous la forme d'une collection (au sens Java du terme). Malheureusement, la prise en charge par le conteneur d'EJB de la problématique de la persistance ne corrige en rien le problème de la richesse fonctionnelle de ces deux classes, lequel encourage les mauvaises pratiques, comme nous l'avons montré avec l'approche naïve.
- **Fragilité.** Le bilan est mitigé. D'un côté, nous bénéficions d'une meilleure séparation des préoccupations, le conteneur d'EJB prenant à sa charge, moyennant paramétrage dans le fichier **ejb-jar.xml,** toute la problématique de la persistance. Mais d'un autre côté, les EJB ne sont pas aussi facilement accessibles que des classes Java standards. De ce fait, les classes clientes des EJB doivent se doter de traitements techniques spécifiques pour accéder aux EJB (accès à l'annuaire JNDI, etc.).
- **Immobilité.** La situation est détériorée : les EJB requièrent non seulement la présence d'une base de données relationnelle mais aussi celle d'un serveur d'applications J2EE doté d'un conteneur d'EJB. La réutilisation dans des contextes techniques différents est donc compromise.
- **Invérifiabilité.** Pas d'amélioration non plus : l'infrastructure technique nécessaire pour tester les deux notions fonctionnelles de notre étude de cas est prépondérante. Nous ne pouvons toujours pas en faire abstraction lors des tests, rendant impossibles les tests unitaires.

En résumé

Fondamentalement, la conception de l'approche technologique reste la même que celle de l'approche naïve. Elle hérite donc d'une grande partie des défauts de cette dernière.

Si la situation s'est améliorée en terme de séparation des préoccupations du fait de la prise en charge de la persistance par le conteneur EJB, elle s'est en revanche dégradée en terme de réutilisation. L'application repose entièrement sur le modèle EJB, ce qui implique nécessairement la présence d'un serveur d'applications en plus de la base de données.

Approche par les modèles de conception

L'approche par les modèles de conception se veut plus mature que les deux précédentes en ce qu'elle repose sur des modèles génériques éprouvés et formalisés.

La technologie retrouve ici sa vraie place par rapport à la modélisation : elle devient un moyen et non une finalité.

Modélisation

Avant d'identifier les modèles de conception qui nous seront utiles, il est important de préciser dès à présent les fondations de notre modélisation. Celles-ci sont constituées de deux piliers principaux :

- Séparation claire entre les notions fonctionnelles manipulées par l'application, leur gestion et leur persistance. L'idée est d'améliorer la réutilisation et l'évolution des services autour des notions fonctionnelles. Il s'agit donc d'une architecture en couches : une couche modèle contenant les notions fonctionnelles, une couche service contenant les gestionnaires de todos et de leurs listes et une couche de persistance.
- Séparation claire entre les interfaces et les implémentations pour rendre transparentes les modifications des implémentations qui n'impactent pas les interfaces.

À partir de ces deux choix fondamentaux, nous pouvons sélectionner les modèles de conception, ou design patterns, suivants :

- La fabrique (factory) : ce modèle de conception permet d'associer une implémentation à une interface. En effet, si nous désirons rendre notre application indépendante des implémentations, il est nécessaire de faire appel à une classe spécifique, la fabrique, qui saura instancier l'implémentation correspondant à une interface donnée.
- Le singleton : ce modèle de conception définit des classes n'ayant qu'une seule instance pour l'ensemble de l'application. Cela sera le cas notamment des fabriques précédemment évoquées, puisqu'il y a plus de désagréments que d'intérêt en terme de ressources d'en avoir plusieurs instances.
- L'objet d'accès aux données, ou DAO (Data Access Object) : ce modèle de conception isole la problématique de l'accès aux données dans des classes prévues exclusivement à cet effet. Nous évitons ainsi la prolifération du code d'accès aux données au sein de l'application en le concentrant dans quelques classes spécialisées, plus faciles à gérer.

Forts de ces choix de conception, nous pouvons proposer la modélisation illustrée à la figure 2.4.

Pour les besoins de la conception, nous avons défini un stéréotype UML « singleton » pour dénoter les classes implémentant le modèle de conception singleton. Nous n'avons pas représenté l'implémentation de ce design pattern au sein des classes puisqu'il s'agit davantage d'une problématique de codage que de conception, comme nous le verrons plus loin.

Les flèches en pointillés montrent les relations d'utilisation des implémentations se substituant aux relations physiques concernant leurs interfaces.

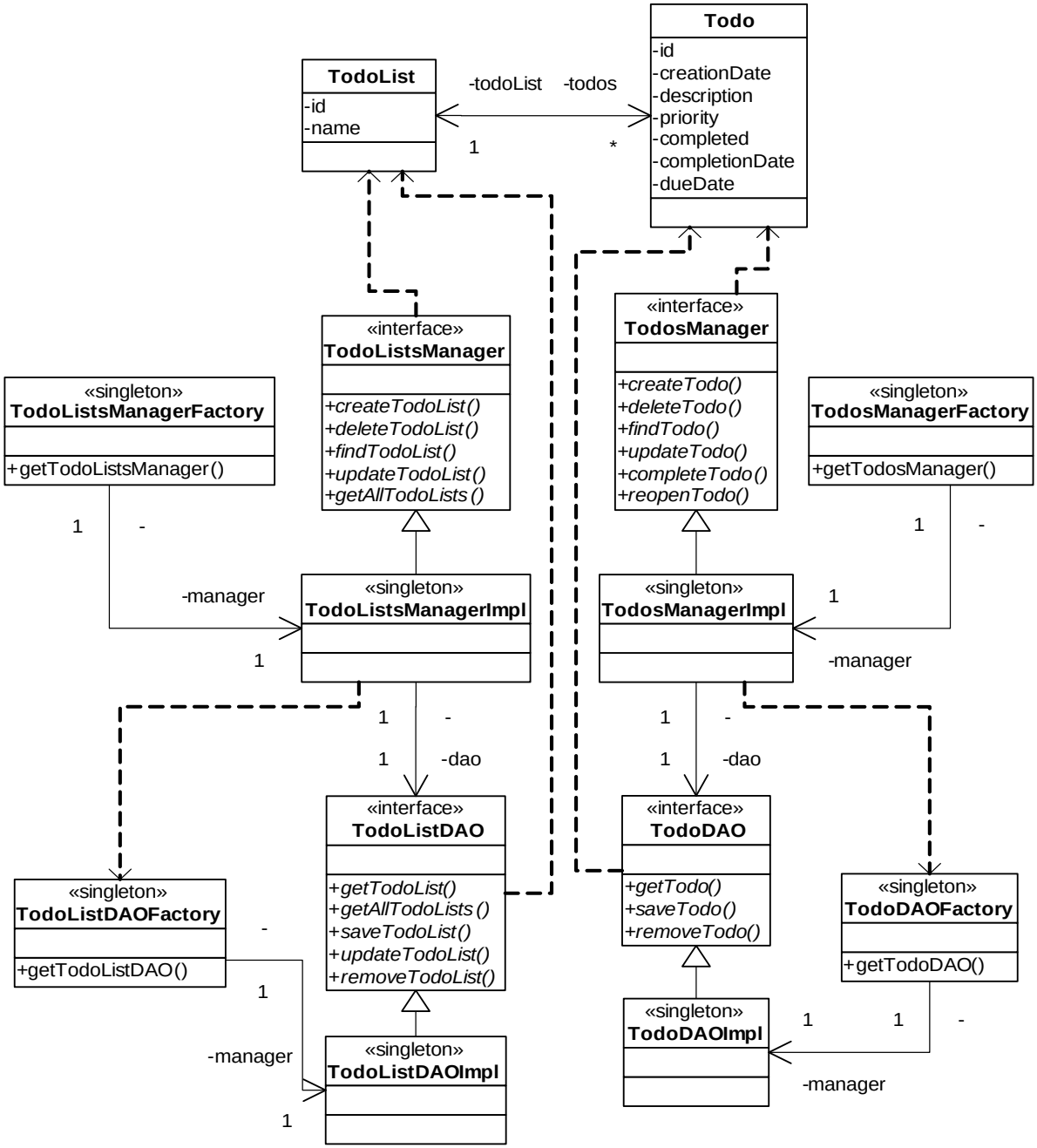


Figure 2.4

Diagramme de classes de l'approche par les modèles de conception

Nous remarquons que les deux piliers de notre conception ont abouti aux résultats suivants :

- Les deux classes `Todo` et `TodoList` sont expurgées des problématiques associées à leur gestion. Il s'agit donc de la matérialisation des deux notions fonctionnelles, sans interférence de la part de la technique.
- La gestion des todos et de leurs listes se matérialise sous forme de deux types de classes : les classes « managers » assurant la gestion et les classes « DAO » prenant en charge la communication avec le support de stockage.
- Du fait de la séparation systématique entre les interfaces et les implémentations des classes « managers » et « DAO », nous avons créé des fabriques permettant d'associer à chacune des interfaces l'implémentation correspondante.

Nous avons choisi de séparer la notion de manager de la notion de DAO afin d'améliorer la séparation des préoccupations au sein de notre modèle. Les DAO sont exclusivement réservés à l'accès aux données, les managers prenant en charge des problématiques de plus haut niveau, comme la sécurité et les transactions. Les DAO pourront être implémentés de plusieurs manières sans impacter le reste de l'application, par exemple, sous forme d'appels JDBC classiques, d'EJB ou encore d'appels à un framework de mapping objet-relationnel.

Nous constatons aussi que les classes métier `TodoList` et `Todo` n'ont pas d'interface séparée de leur implémentation. Cela se justifie par le fait qu'il n'y a aucun intérêt pratique à fournir des implémentations différentes pour des classes métier qui s'avèrent être de simples structures de données.

De même, les fabriques ne possèdent pas d'interface distincte de leur implémentation pour la bonne raison que, dans le cas contraire, celles-ci devraient bénéficier d'une fabrique, ce qui n'apporterait aucune valeur ajoutée au modèle.

Critique de la modélisation

Une critique qui vient immédiatement à l'esprit lorsque nous comparons cette modélisation avec les approches précédentes est qu'elle fait intervenir un plus grand nombre de classes et d'interfaces, rendant le modèle un peu plus complexe à comprendre et le code plus long à écrire.

Cependant, si nous nous attachons aux quatre critères d'analyse que nous avons définis, nous constatons que la situation s'améliore très nettement :

- **Rigidité.** La séparation claire des préoccupations permet de faire évoluer les problématiques techniques telles que la persistance des données de manière transparente vis-à-vis des classes métier. De même, les classes « managers » peuvent offrir de nouveaux services ou implémenter de nouveaux contrôles sans mettre en cause les classes métier.
- **Fragilité.** Chaque couche étant clairement isolée des autres, du moment que les contrats fixés par les interfaces sont respectés, l'évolution des implémentations associées

se déroule de manière totalement transparente et ne crée normalement pas d'effets de bord sur le reste du code.

- **Immobilité.** La situation est là aussi nettement améliorée du fait de l'architecture en couches et de la séparation claire des interfaces. Les classes métier sont réutilisables quel que soit le contexte technique sous-jacent. Les classes managers et DAO peuvent évoluer dans leur implémentation pour s'adapter au contexte technique, comme l'utilisation d'un fichier XML au lieu d'un SGBDR pour la persistance, sans pour autant remettre en cause leurs interfaces, et donc le modèle dans son intégralité.
- **Invérifiabilité.** Nous pouvons enfin tester chaque classe indépendamment des autres. En effet, les classes `TodoList` et `Todo` se suffisent à elles-mêmes, puisqu'elles n'ont aucune dépendance technique, et peuvent donc être testées unitairement. De même, les classes managers peuvent être testées de manière unitaire en remplaçant l'implémentation des DAO utilisant le SGBDR par des « objets simulacres », ou mock objects, développés spécifiquement pour les tests à l'aide de frameworks tels qu'EasyMock, qui facilitent grandement leur création. Pour cela, il est nécessaire de modifier les fabriques de manière qu'elles génèrent des simulacres et non des implémentations réelles.

En résumé

L'utilisation des design patterns a permis d'augmenter de manière significative la qualité de note conception pour tous nos critères d'analyse. Cette qualité s'obtient cependant au prix d'une plus grande complexité de conception du fait de l'implémentation des design patterns.

Bilan des différentes approches

Comme nous venons de le voir, la technologie seule ne suffit pas pour bâtir un modèle évolutif et testable. En se fondant sur l'expérience des meilleurs experts, l'approche par modèle de conception parvient à améliorer significativement la situation.

Cependant, si nous observons le modèle de conception de cette dernière approche, nous pouvons nous interroger sur la possibilité d'implémenter de manière générique, et donc réutilisable, les fabriques et les singletons, et plus généralement le cycle de vie des objets. En effet, leur mode de fonctionnement ne varie pas en fonction des interfaces et des implémentations.

Plus généralement, la gestion des dépendances entre les objets reste encore présente dans le code, en partie prise en charge par les fabriques. Plusieurs dépendances sont gérées au sein des classes mais ne sont pas visibles au niveau conceptuel, à l'exemple des sources de données nécessaires aux DAO.

Ce sont ces problèmes que proposent de résoudre les conteneurs légers, dont nous allons étudier les mécanismes dans les sections suivantes.

L'inversion de contrôle

Les conteneurs légers sont aussi souvent appelés conteneurs d'inversion de contrôle, ou IoC (Inversion of Control) Container. Avant d'étudier le fonctionnement de ces conteneurs, il est important de bien comprendre ce qu'est le concept d'inversion de contrôle.

Par contrôle, nous entendons le contrôle du flot d'exécution d'une application. Comme nous le verrons, l'inversion de contrôle est un concept générique ancien, que l'on rencontre dans des solutions logicielles autres que les conteneurs légers. Nous montrerons que l'inversion de contrôle proposée par les conteneurs légers s'avère être une version spécialisée de cette notion.

Contrôle du flot d'exécution

Lorsque nous programmons de manière procédurale, nous maîtrisons totalement le flux d'exécution de notre programme *via* des instructions, des conditions et des boucles. C'est généralement le cas au sein d'une méthode, par exemple.

Si nous considérons une application utilisant Swing (l'API graphique standard de Java), par exemple, nous ne pouvons pas dire que le flot d'exécution est maîtrisé de bout en bout par l'application. En effet, Swing utilise un modèle événementiel qui déclenche les traitements de l'application en fonction des interactions de l'utilisateur avec l'IHM (clic sur un bouton, etc.).

En nous reposant sur Swing pour déclencher les traitements de l'application, nous opérons une inversion du contrôle du flot d'exécution, puisque ce n'est plus notre programme qui se charge de le contrôler de bout en bout. Au contraire, le programme s'insère dans le cadre de fonctionnement de Swing (son modèle événementiel) en « attachant » ses différents traitements aux événements générés par Swing suite aux actions de l'utilisateur.

La figure 2.5 illustre l'effet de l'inversion de contrôle sur le flot d'exécution.

Inversion de contrôle : frameworks et bibliothèques logicielles

Comme le fait remarquer Martin Fowler dans son célèbre article sur les conteneurs légers *(voir références en annexe),* l'inversion de contrôle est un principe qui permet de distinguer les frameworks des bibliothèques logicielles.

Les bibliothèques sont de simples boîtes à outils, dont les fonctions s'insèrent dans le flot d'exécution du programme. Il n'y a donc pas d'inversion de contrôle. À l'inverse, les frameworks prennent à leur charge l'essentiel du flot d'exécution dans leur domaine de spécialité, le programme devant s'insérer dans le cadre qu'ils fixent.

Ainsi, le principe de l'inversion de contrôle est au moins aussi ancien que celui de la programmation événementielle. Il s'agit d'un principe générique utilisé par de nombreux frameworks apparus bien avant la notion de conteneur léger. L'inversion de contrôle est

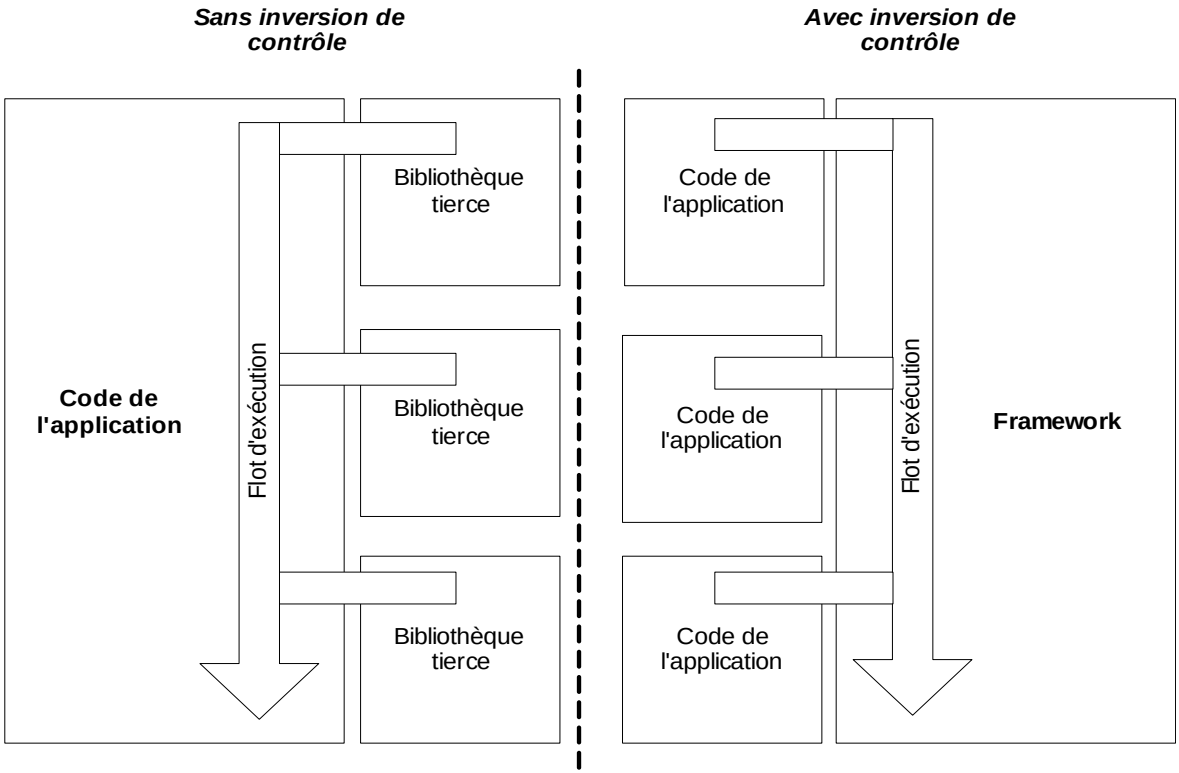


Figure 2.5
Effet de l'inversion de contrôle sur le flot d'exécution

aussi appelée principe « Hollywood » en référence à la phrase mythique « ne nous appelez pas, nous vous appellerons ».

Dans le cas du framework Web Struts, il est évident qu'il implémente une inversion de contrôle puisqu'il se charge d'appeler lui-même les actions de l'application en fonction des requêtes envoyées par les navigateurs Web. Cependant, cette inversion de contrôle est limitée à la couche de présentation, comme nous le verrons au chapitre 6, dédié à l'intégration de Struts par Spring.

De la même manière, les conteneurs d'EJB implémentent une inversion de contrôle puisque la gestion du cycle de vie des EJB (*via* les méthodes `ejbActivate`, `ejbPassivate`, etc.) ainsi que l'appel à leurs différentes méthodes sont pris en charge directement par le conteneur.

Maintenant que nous connaissons la signification de l'inversion de contrôle et ses nombreuses implémentations, nous pouvons nous interroger sur la nouveauté introduite par les conteneurs légers en la matière.

L'inversion de contrôle au sein des conteneurs légers

Les conteneurs légers proposent une version spécialisée de l'inversion de contrôle. Ils se concentrent en fait sur le problème évoqué plus haut de la gestion des dépendances entre objets et leur instanciation, notamment dans le cadre d'une dissociation des interfaces et des implémentations.

Si nous reprenons notre étude de cas, la modélisation proposée par l'approche par les modèles de conception est la meilleure selon nos critères d'appréciation (rigidité, fragilité, immobilité et invérifiabilité). Cependant, si nous la comparons à l'approche naïve, nous constatons que ce gain s'effectue au détriment de la taille et de la complexité du code : la flexibilité du modèle a un coût.

Dans notre cas, ce coût est assez faible. Les classes et interfaces étant peu nombreuses, leurs dépendances sont facilement gérables. Cependant, si nous envisageons ce type de modélisation pour une application plus complexe, il devient évident que le grand nombre de fabriques va rapidement poser problème en terme de gestion. Il est donc intéressant de chercher une solution de rechange à cette multitude de fabriques, d'autant qu'elles fonctionnent toutes selon le même modèle, qui consiste à associer une implémentation X à une interface Y à la demande d'une classe utilisatrice. De même, bien que les singletons fonctionnent tous selon le même principe, quelle que soit la classe considérée, ils doivent être implémentés systématiquement. Il serait judicieux de proposer un mode de fonctionnement réellement générique.

Pour résoudre ce problème, il faut modulariser ces comportements au sein d'une entité ou d'un groupe d'entités unique afin de bénéficier d'un effet de réutilisation maximal. C'est ici que l'inversion de contrôle intervient : plutôt que de laisser l'application maîtriser la fabrication des objets dont elle a besoin et la gestion de leurs dépendances, un framework de création d'objets doit pouvoir prendre en charge cette problématique de manière générique.

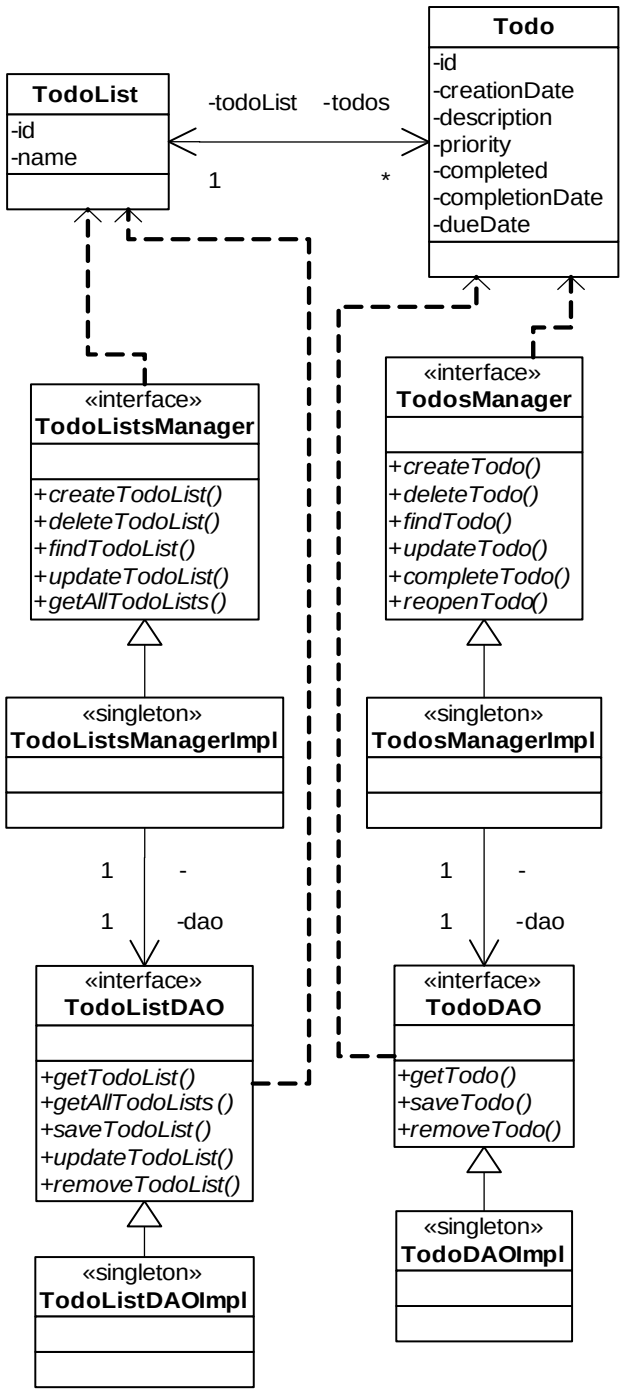
Le résultat attendu en terme de modélisation serait la disparition pure et simple des fabriques et de l'implémentation de la notion de singleton dans notre modélisation, comme l'illustre la figure 2.6.

En sous-traitant la gestion des dépendances et l'instanciation des objets, nous rendons le modèle plus lisible, mais au prix d'une perte d'information puisque la gestion des dépendances n'apparaît plus explicitement dans le modèle. Il faut donc veiller à documenter correctement les modèles pour pallier cette carence.

Grâce au conteneur léger, nous remplaçons des mécanismes spécifiques de fabrication des objets et de gestion des dépendances par un mécanisme générique. En règle générale, les conteneurs légers sont paramétrables de manière déclarative *via* des fichiers de configuration. Nous définissons ainsi un véritable référentiel des dépendances entre objets.

Figure 2.6

Modélisation avec une inversion de contrôle sur la gestion des dépendances des objets



Si nous reprenons nos quatre critères d'analyse, nous pouvons constater que la situation s'améliore encore par rapport à l'approche par les modèles de conception :

- **Rigidité.** La gestion générique des dépendances offre une plus grande souplesse dans l'évolution des interfaces et de leurs implémentations. En effet, si le conteneur léger est paramétré *via* des fichiers de configuration, la modification de ces derniers s'avère moins coûteuse que la modification d'une fabrique développée spécifiquement.
- **Fragilité.** S'il n'y a pas ici d'apport majeur, nous pouvons toutefois noter que le conteneur léger fournit une infrastructure éprouvée, qui ajoute un élément de confiance dans la gestion des dépendances, notamment des dépendances complexes, comme les dépendances cycliques, et contribue à isoler plus facilement d'éventuels dysfonctionnements applicatifs.
- **Immobilité.** Le modèle étant simplifié du fait d'une plus grande abstraction apportée par le conteneur léger, nous pouvons dire que la réutilisation des concepts est facilitée.
- **Invérifiabilité.** Elle est minimisée, car le conteneur léger facilite l'utilisation des simulacres par rapport à l'approche précédente, qui nécessite la réécriture des fabriques.

Comme nous le voyons, l'inversion de contrôle pour la gestion des dépendances offre des avantages non négligeables en matière de conception des applications. Cette inversion de contrôle peut être assurée de deux façons différentes : par la recherche de dépendances ou par l'injection de dépendances, cette dernière étant privilégiée par Spring.

La section suivante aborde ces deux techniques en les comparant.

L'injection de dépendances

Les conteneurs légers peuvent être classés en deux catégories en fonction de la façon dont ils gèrent les dépendances entre objets :

- Les conteneurs légers qui utilisent la recherche de dépendances.
- Les conteneurs légers qui utilisent l'injection de dépendances.

Spring utilise essentiellement l'injection de dépendances mais propose aussi la recherche de dépendances pour des cas spécifiques où l'injection ne fonctionne pas. Nous décrirons donc les deux modes de fonctionnement.

L'injection de dépendances existe sous trois formes :

- injection de dépendances *via* le constructeur ;
- injection de dépendances *via* les modificateurs (setters) ;
- injection de dépendances *via* une interface.

Les deux premières formes sont les plus utilisées et sont proposées toutes deux par Spring. Nous n'aborderons donc pas la troisième forme, qui est marginale.

Recherche de dépendances

La recherche de dépendances consiste pour un objet donné à interroger le conteneur pour trouver ses dépendances. Ce mode de fonctionnement est typiquement celui proposé par les EJB.

Pour accéder à ces derniers, il est nécessaire d'appeler un annuaire JNDI, comme le montre le code suivant :

```
(...)
ctx = new InitialContext(proprietes);
ref = ctx.lookup("MonEJB");
home = (MonEJBHome) javax.rmi.PortableRemoteObject.narrow(ref,
   MonEJBHome.class);
monEJB = home.create();
```

Dans notre étude de cas, chaque objet appelle le conteneur léger en lieu et place des fabriques spécifiques pour récupérer ses dépendances. Par exemple, l'initialisation d'une instance de la classe `TodoListsManagerImpl` pourrait être la suivante :

```
public class TodoListsManagerImpl implements TodoListsManager {
   (...)
   TodoListsDAO dao ;

   public TodoListsManager() {
      (...)
      dao = (TodoListsDAO)Conteneur.getDependance("TodoListsDAO");
      (...)
   }

   (...)
}
```

La méthode `getDependance` de notre conteneur fictif instancie la classe Java référencée en tant que `"TodoListsDAO"` dans son référentiel interne. Cette méthode étant générique, le typage de sa valeur de retour est forcément `java.lang.Object`, ce qui explique la nécessite du transtypage en `TodoListsDAO`. Bien entendu, il est supposé ici que la classe associée à l'identifiant `"TodoListsDAO"` implémente bien l'interface `TodoListsDAO`. Il faut noter que le respect de cette règle ne pourra malheureusement être vérifié qu'à l'exécution puisque notre haut niveau d'abstraction nous fait perdre la sécurité du typage fort.

La recherche de dépendances contextualisée (Contextualized Dependency Lookup) est une autre façon de procéder conforme au principe de l'inversion de dépendances. Elle se présente sous la forme d'une interface à implémenter par tout objet géré par le contrôleur :

```
public interface InitDependances {
   public void resoudreDependances(Conteneur conteneur) ;
}
```

Le conteneur léger prenant en charge l'instanciation des objets, il appelle systématiquement la méthode `resoudreDependances` juste après leur création.

Dans le cas de notre classe `TodoListsManagerImpl`, le code est maintenant le suivant :

```
public class TodoListsManagerImpl implements
 TodoListsManager,InitDependances {
   (...)
   TodoListsDAO dao ;

   public TodoListsManager() {
      (...)
   }

   public void resoudreDependances(Conteneur conteneur) {
      dao = (TodoListsDAO)conteneur.getDependance("TodoListsDAO");
   }
   (...)
}
```

La recherche de dépendances est une solution simple au problème de gestion des dépendances. Cependant, elle rend le code explicitement dépendant du conteneur léger, et ce quel que soit le type de recherche que nous effectuons. Cela présente deux inconvénients majeurs : la dépendance de l'application vis-à-vis du conteneur léger est plus forte, et les tests unitaires sur les objets nécessitent la présence systématique du conteneur (du fait des appels directs à celui-ci).

Avec la recherche de dépendances, la modélisation représentée à la figure 2.6 n'est pas atteinte, puisqu'un lien explicite reste présent entre les classes et le conteneur qui les gère.

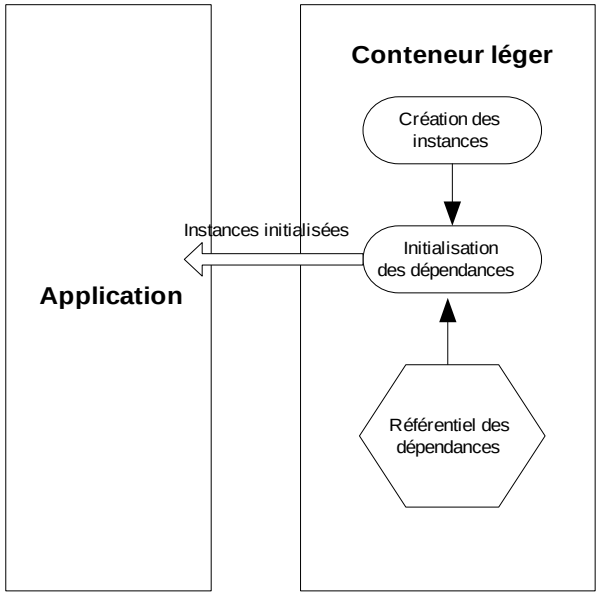
Injection de dépendances

L'injection vise à rendre l'inversion de contrôle de la gestion des dépendances la plus transparente possible vis-à-vis des classes concernées. Comme nous l'avons vu à la section précédente, la recherche de dépendances crée un lien explicite entre les classes et le conteneur léger en charge de la gestion de leurs dépendances. Grâce à l'injection de dépendances, nous allons pouvoir transformer ce lien explicite en lien implicite.

Pour procéder à l'injection des dépendances, le conteneur léger initialise directement les objets (à partir d'un référentiel), libérant ainsi l'application de cette charge. Au lieu d'utiliser l'opérateur `new`, le conteneur léger injecte dans l'application les instances dont elle a besoin, comme l'illustre la figure 2.7.

Pour effectuer cette initialisation, le conteneur peut implémenter deux méthodes : l'injection de dépendances *via* le constructeur ou l'injection de dépendances *via* les modificateurs. Spring offre la possibilité de combiner l'injection *via* les constructeurs avec celle fondée sur les modificateurs.

Figure 2.7
Principe général de l'injection de dépendances



Injection de dépendances *via* le constructeur

En Java, toute classe dispose d'un constructeur, qu'il soit explicite, c'est-à-dire implémenté dans le code de la classe, ou par défaut, c'est-à-dire fourni par la machine virtuelle. Si nous suivons les bonnes pratiques de programmation orientée objet, le rôle d'un constructeur est de construire une instance d'une certaine classe dans un état valide. Si cet objet possède des dépendances (sous forme d'attributs) avec d'autres objets, ces dépendances sont initialisées par le constructeur.

Si nous reprenons l'exemple de la classe `TodoListsManagerImpl` de l'approche par les modèles de conception, nous aurions donc le code suivant :

```
public class TodoListsManagerImpl implements TodoListsManager {
   (...)
   TodoListsDAO dao ;

   public TodoListsManager() {
      (...)
      dao = TodoListsDAOManager.getTodoListsDAO();← ❶
      (...)
   }

   (...)
}
```

Avec la recherche de dépendances présentée à la section précédente, nous avons remplacé une dépendance vis-à-vis d'une fabrique spécifique par une dépendance vis-à-vis du conteneur.

L'idée de l'injection de dépendances *via* le constructeur est d'externaliser l'appel ❶ afin qu'il soit pris en charge par le conteneur léger, charge à celui-ci de transmettre au constructeur l'instance de la dépendance désirée. Le lien explicite avec le conteneur léger est remplacé par un lien implicite, rendant notre code plus indépendant du framework dans lequel il s'exécute.

Pour arriver à ce résultat, il suffit de transformer le constructeur pour qu'il accepte de recevoir en paramètres les dépendances. Ainsi, notre code se présente de la manière suivante :

```
public class TodoListsManagerImpl implements TodoListsManager {
   (...)
   TodoListsDAO dao ;

   public TodoListsManager(TodoListsDAO pDAO) {
      (...)
      dao = pDAO;
      (...)
   }

   (...)
}
```

Nous avons cassé la dépendance explicite avec le conteneur. L'instanciation de `TodoListsManagerImpl` se faisant au sein du conteneur léger, et non par l'opérateur `new`, celui-ci emploie le constructeur pour initialiser (injecter) les dépendances de l'instance de la classe.

Le principal avantage de l'injection de dépendances *via* le constructeur est que cette technique respecte les bonnes pratiques de programmation orientée objet (le constructeur doit être utilisé pour créer un objet dans un état valide). Cependant, son inconvénient majeur, en tout cas avec Java, est que les paramètres du constructeur sont non pas nommés, mais ordonnés. C'est l'ordre des paramètres qui permet de savoir quelle dépendance est initialisée.

Ce mode de fonctionnement s'avère rapidement peu lisible et source d'erreurs dès qu'augmente le nombre de dépendances à gérer pour une classe. Par ailleurs, l'injection de dépendances *via* le constructeur pose problème pour l'héritage. En effet, puisqu'un constructeur est spécifique de chaque classe et que ses paramètres ne sont pas nommés, nous perdons la capacité d'hériter de la configuration d'une classe mère à une classe fille au niveau du conteneur léger.

Injection de dépendances *via* les modificateurs

Outre le constructeur, il est possible d'utiliser les modificateurs (setters) pour initialiser les attributs d'un objet. L'injection de dépendances *via* les modificateurs utilise ces derniers pour initialiser les dépendances d'un objet. Bien entendu, cela suppose que la classe dispose des modificateurs permettant de le faire.

Si nous reprenons notre classe `TodoListsManagerImpl`, nous obtenons le code suivant :

```
public class TodoListsManagerImpl implements TodoListsManager {
  (...)
  TodoListsDAO dao ;

  public TodoListsManager() {
     (...)
  }

  public void setDao(TodoListsDAO pDAO) {
     dao = pDAO ;
  }
  (...)
}
```

Le conteneur léger utilise la méthode `setDao` pour initialiser la dépendance de `TodoListsManagerImpl` vis-à-vis de `TodoListsDAO` au lieu d'utiliser le constructeur comme précédemment.

Il faut noter qu'un modificateur n'est pas forcément associé à un seul attribut de la classe et qu'il peut effectuer des traitements complexes pour initialiser l'objet.

Le principal avantage de cette forme d'injection est que les modificateurs sont nommés, puisqu'il s'agit de méthodes. Les noms apparaîtront de la sorte explicitement dans le paramétrage du conteneur léger, contrairement à l'injection *via* le constructeur, où seul l'ordre apparaît. En contrepartie, cette façon de procéder constitue une entorse à la bonne pratique consistant à utiliser le constructeur pour initialiser une instance d'objet. Nous ne bénéficions plus de la sécurité offerte par le constructeur, dont les paramètres indiquent explicitement toutes les dépendances à initialiser.

Malgré ce défaut, l'injection *via* les modificateurs est généralement préférée du fait de sa meilleure lisibilité, facilitant la maintenance des dépendances. De plus, cette méthode permet à une classe fille d'hériter de la configuration de sa classe mère au niveau du conteneur léger.

Gestion du cycle de vie des objets

Comme nous venons de le voir, l'inversion de contrôle dans la gestion des dépendances implique une mainmise du conteneur léger lors de l'initialisation des objets. Ce contrôle permet par extension au conteneur d'assurer la gestion du cycle de vie des objets.

Cette gestion du cycle de vie des objets comporte généralement pour les conteneurs légers deux volets :

- la gestion des singletons ;
- la génération d'événements indiquant le changement d'état de l'objet.

Gestion des singletons

Classiquement, le design pattern singleton est implémenté dans chaque classe ne devant avoir qu'une seule instance au sein de l'application. Un singleton est souvent implémenté de la manière suivante :

```
public class TodoListsManagerImpl implements TodoListsManager {
   (...)
   private static TodoListsManagerImpl instance =
    new TodoListsManagerImpl();

   public TodoListsManagerImpl getInstance() {
      return instance;
   }

   private TodoListsManagerImpl() {
      (...)
   }
}
```

Le constructeur a une portée privée afin d'éviter que des instances de la classe puissent être créées en dehors de celle-ci par des appels à l'opérateur `new`. Seule la méthode `getInstance` peut être utilisée pour récupérer l'instance unique de la classe.

Cette implémentation du design pattern singleton a beau être très générique, elle est malheureusement dépendante de la classe à instancier, empêchant toute réutilisation. Par ailleurs, il n'existe pas de norme de codage de ce design pattern. Ainsi, il est possible de trouver au sein d'une même application plusieurs implémentations différentes, utilisant des noms de méthodes différents, etc.

Puisque le conteneur léger assure l'instanciation des objets, il est en mesure de contrôler le nombre d'instances créées et de solutionner ainsi le problème de la duplication de la mécanique du singleton au sein des classes devant implémenter ce design pattern.

Il suffit de déclarer une classe comme étant un singleton auprès du conteneur léger pour que celui-ci ne génère qu'une instance unique. Il n'est de la sorte plus nécessaire d'implémenter la mécanique du singleton au sein de chaque classe devant implémenter ce design pattern. En contrepartie, un transtypage est nécessaire.

Génération d'événements

Au sein des conteneurs légers, le cycle de vie des objets est souvent réduit à sa plus simple expression, à savoir la vie puis la mort. Le cycle de vie qu'ils proposent est donc des plus simples en comparaison de celui des EJB – il n'y a pas, par exemple, de notion d'activation ou de désactivation – mais répond néanmoins aux besoins des objets qu'ils manipulent.

La génération d'événements consiste à appeler des méthodes spécifiques de l'objet en fonction de ses changements d'états. Ces méthodes spécifiques peuvent être formalisées sous forme d'interfaces à implémenter ou spécifiées par paramétrage du conteneur léger.

Nous aurons ainsi une méthode qui sera appelée par le conteneur léger après l'instanciation de l'objet permettant, par exemple, d'effectuer des contrôles de cohérence sur l'état de l'objet. Une autre méthode sera appelée par le conteneur léger juste avant la destruction de l'objet. Ce type de méthode est souvent utile pour effectuer des opérations de nettoyage ou de libération de ressources.

Conclusion

Ce chapitre s'est efforcé de comparer plusieurs approches permettant de concevoir une version simplifiée du modèle métier de Tudu Lists. Les approches naïve et technologique se sont avérées peu satisfaisantes car trop rigides, fragiles, immobiles et invérifiables.

La situation s'est nettement améliorée avec l'approche par les modèles de conception, même si cette dernière souffre de certaines faiblesses, notamment une plus grande complexité du modèle du fait de l'utilisation dans certains cas de singletons et de fabriques. Chacun de ces cas a nécessité une implémentation spécifique de ces modèles.

Grâce à l'injection de dépendances et à la gestion du cycle de vie des objets, les conteneurs légers permettent d'atteindre la modélisation idéale illustrée à la figure 2.6, qui offre un résultat nettement amélioré selon nos critères d'analyse (rigidité, fragilité, immobilité et invérifiabilité). Les modèles de conception sont implémentés directement au sein du conteneur léger, libérant ainsi l'application de leur implémentation spécifique.

Cependant, il est important de garder à l'esprit que l'injection de dépendances n'est pas la pierre philosophale de la programmation. L'injection de dépendances est un outil permettant de rendre une modélisation plus flexible. Comme nous l'avons vu, cette flexibilité a un coût. Il faut donc s'assurer systématiquement que la flexibilité du modèle est justifiée. Par exemple, des abstractions fondamentales comme les classes `Todo` et `TodoList` n'ont nullement besoin de flexibilité puisqu'elles représentent les invariants de notre application.

Maintenant que nous avons explicité les principes de fonctionnement des conteneurs légers, nous pouvons aborder le fonctionnement de Spring.

3

Le conteneur léger de Spring

Le chapitre précédent a détaillé les principes de fonctionnement des conteneurs légers et montré qu'ils amélioraient de manière significative la qualité d'une application. C'est à partir de ces bases conceptuelles que nous abordons ici le conteneur léger de Spring.

Spring propose une multitude de façons d'utiliser son conteneur léger, ce qui est souvent déroutant pour le développeur découvrant cet outil. Afin de permettre au lecteur d'utiliser efficacement le framework, nous nous concentrons dans le présent chapitre sur les fonctionnalités utilisées par la grande majorité des projets.

Dans un premier temps, nous abordons la notion de fabrique de Beans (les Beans sont les objets gérés par le conteneur léger selon la terminologie Spring) et de contexte d'application *(application context)*, qui permettent de configurer et dialoguer avec le conteneur léger.

Nous décrivons ensuite les différentes techniques dont nous disposons pour définir nos objets, ainsi que leurs dépendances au sein du conteneur léger. Nous détaillons en outre le cycle de vie des objets et les différentes interfaces disponibles pour leur permettre de communiquer avec le conteneur léger.

En fin de chapitre, nous évoquons diverses fonctionnalités avancées du conteneur, souvent utilisées dans les projets.

Fabrique de Bean et contexte d'application

La fabrique de Bean *(Bean Factory)* est l'interface de base permettant aux applications reposant sur le conteneur léger de Spring d'accéder à ce dernier. Elle définit les fonctionnalités minimales dont dispose l'application pour dialoguer avec le conteneur.

Cette interface comporte plusieurs implémentations prêtes à l'emploi. Spring étant un framework ouvert (du fait du découplage systématique entre interfaces et implémentations), il est possible de définir notre propre implémentation pour répondre à des besoins spécifiques, non couverts par les implémentations proposées par Spring.

Le contexte d'application *(application context)* est une interface qui étend la notion de fabrique de Bean en ajoutant des fonctionnalités supplémentaires très utiles pour les applications. La majorité des projets fondés sur Spring utilisent des implémentations de cette interface plutôt que des implémentations directes de la fabrique de Bean, conformément aux recommandations de la documentation de Spring.

La fabrique de Bean

La notion de fabrique de Bean, qui offre un moyen d'accès au conteneur léger, est manipulable par l'application au travers de deux interfaces complémentaires : l'interface `BeanFactory`, qui définit l'accès aux Beans gérés par le conteneur, et l'interface `BeanDefinitionRegistry`, qui formalise la définition des Beans que le conteneur léger doit gérer.

Ces interfaces disposent de plusieurs implémentations fournies en standard par Spring, mais elles peuvent aussi être implémentées spécifiquement dans le cadre d'un projet.

L'interface *BeanFactory*

L'interface `BeanFactory` permet au reste de l'application d'accéder aux objets gérés par le conteneur léger.

Cette interface est très succincte, comme le montre son code, reproduit partiellement ci-dessous :

```
package org.springframework.beans.factory;

(...)

public interface BeanFactory {
(...)
   Object getBean(String name) throws BeansException;
   Object getBean(String name, Class requiredType)
      throws BeansException;
   boolean containsBean(String name);
   boolean isSingleton(String name)
      throws NoSuchBeanDefinitionException;
   Class getType(String name) throws NoSuchBeanDefinitionException;
   String[] getAliases(String name)
      throws NoSuchBeanDefinitionException;
}
```

Les méthodes proposées par cette interface permettent uniquement d'interroger le conteneur léger à propos des objets dont il a la charge. Les objets sont désignés par un nom sous forme de chaîne de caractères (par exemple, le paramètre `id` permettant de définir l'identifiant unique du Bean).

La méthode `containsBean` vérifie pour un nom donné qu'un objet correspondant est bien géré dans le conteneur léger.

Les méthodes `getBean` permettent de récupérer l'instance d'un objet à partir de son nom. L'une d'elle prend un paramètre supplémentaire, `requiredType`, afin de contraindre le type d'objet renvoyé par `getBean` pour plus de sécurité. Si le nom fourni ne correspond pas à un objet dont le type est celui attendu, une exception est générée. Par ailleurs, la méthode `getType` permet de connaître le type d'un objet à partir de son nom.

Un Bean renvoyé par la fabrique peut être un singleton ou non. Nous parlons dans ce dernier cas de prototype dans la terminologie Spring. La méthode `isSingleton` permet de savoir, à partir du nom d'un objet, s'il s'agit ou non d'un singleton.

Tout objet dans Spring peut avoir des noms multiples, ou alias. La méthode `getAlias` fournit l'ensemble des alias associés à un objet dont nous connaissons le nom.

L'interface `BeanFactory` est étendue par d'autres interfaces définissant des fonctionnalités supplémentaires. Citons notamment l'interface `ConfigurableBeanFactory`, qui définit le protocole pour configurer le conteneur léger, l'interface `ListableBeanFactory`, qui fournit des fonctions avancées pour consulter les objets gérés par le conteneur ou encore l'interface `HierarchicalBeanFactory`, qui permet de définir une hiérarchie de fabriques de Bean.

L'interface *BeanDefinitionRegistry*

Si toutes ces fonctions sont nécessaires pour l'utilisation du conteneur léger, elles ne sont cependant pas suffisantes pour le faire fonctionner. Notons notamment l'absence totale de fonctions permettant d'indiquer au conteneur léger les objets qu'il gère et leurs dépendances.

Ces fonctions sont formalisées par l'interface `BeanDefinitionRegistry`, qui spécifie le protocole pour définir les objets à gérer par le conteneur ainsi que leurs dépendances.

Ce protocole est réduit au strict nécessaire, comme le montre son code, reproduit partiellement ci-dessous :

```
package org.springframework.beans.factory.support;

(...)

public interface BeanDefinitionRegistry {
   (...)
   int getBeanDefinitionCount();
   String[] getBeanDefinitionNames();
   boolean containsBeanDefinition(String beanName);
   BeanDefinition getBeanDefinition(String beanName)
      throws NoSuchBeanDefinitionException;
   void registerBeanDefinition(String beanName, BeanDefinition
                                                beanDefinition)
      throws BeansException;
   void registerAlias(String beanName, String alias)
      throws BeansException;
}
```

L'opération d'enregistrement d'un objet s'effectue par le biais de la méthode `registerBeanDefinition`, qui prend en paramètres le nom de l'objet et sa définition. Cette dernière est formalisée par l'interface `BeanDefinition`, qui fournit au conteneur les informations nécessaires pour gérer l'objet et ses propriétés.

La méthode `registerAlias` permet pour sa part d'enregistrer les différents noms qu'un objet possède au sein du conteneur léger. Les autres fonctions sont destinées à la consultation des définitions des objets qui sont gérés par le conteneur.

Implémentations de *BeanFactory* et *BeanDefinitionRegistry*

Grâce aux interfaces `BeanFactory` et `BeanDefinitionRegistry`, les concepteurs de Spring ont formalisé les différentes manières de faire dialoguer l'application avec le conteneur léger.

À partir de cette formalisation sous forme d'interfaces, plusieurs implémentations sont possibles, permettant de choisir celle qui est la mieux adaptée. Spring fournit plusieurs implémentations directement utilisables, qui suffisent dans la grande majorité des cas. Bien entendu, nous avons toujours la possibilité de développer notre propre implémentation, mais nous n'aborderons pas ce genre de besoin dans le contexte de cet ouvrage.

Une des implémentations les plus simples mises à la disposition des applications est la classe `DefaultListableBeanFactory`. Située, comme les autres implémentations fournies par Spring, dans le package `org.springframework.beans.factory.support`, cette classe fournit un accès simple (*via* les interfaces `BeanFactory` et `BeanDefinitionRegistry`) au conteneur léger mais sans valeur ajoutée.

La définition des objets à gérer pour une application donnée doit donc être développée spécifiquement. Nous pouvons utiliser pour cela un fichier de propriétés, qui sera lu par l'application pour déclarer chaque objet à gérer auprès du conteneur léger *via* la méthode `registerBeanDefinition` de l'interface `BeanDefinitionRegistry`.

`DefaultListableBeanFactory` fournit le minimum de fonctionnalités nécessaires et peut servir de base pour développer nos propres implémentations. Dans la majorité des cas, les implémentations plus évoluées fournies par Spring sont toutefois préférables à cette classe.

L'implémentation `XmlBeanFactory` supporte la définition des objets sous forme de fichiers XML. Spring dispose en effet d'un langage XML de définition d'objets formalisé par des schémas XML. Précédemment, dans les versions 1.*x* de Spring, il s'agissait d'une DTD beaucoup moins satisfaisante en terme de vérification du respect du langage.

Le schéma est consultable à l'URL *http://www.springframework.org/schema/beans*. Le langage spécifié par ce schéma est identique à celui de l'ancienne DTD. Les fichiers de configuration fondés sur la version 1.2 de Spring n'ont de la sorte pas à être modifiés pour fonctionner avec la version 2.0. La définition des objets sous forme XML est celle qui est la plus utilisée dans les projets utilisant Spring. Dans la suite de cet ouvrage, nous nous concentrons donc sur cette forme.

Pour aller plus loin en matière de fonctionnalités, Spring définit la notion de contexte d'application, qui englobe les services du conteneur léger définis précédemment et ajoute

des services additionnels, utiles aux applications mais sans rapport direct avec la notion de conteneur léger.

Le contexte d'application

La notion de contexte d'application étend celle de fabrique et de registre de Bean en ajoutant de nouvelles fonctionnalités très utiles pour les applications, notamment les suivantes :

- Support des messages et de leur internationalisation.
- Support avancé du chargement de fichiers (appelés ressources), que ce soit dans le système de fichiers ou dans le classpath de l'application.
- Support de la publication d'événements permettant à des objets de l'application de réagir en fonction d'eux.
- Possibilité de définir une hiérarchie de contextes. Cette fonctionnalité est très utile pour isoler les différentes couches de l'application (les Beans de la couche présentation ne sont pas visibles de la couche service, par exemple). Cette possibilité est évoquée dans la partie II de l'ouvrage.

Ces fonctionnalités additionnelles seront présentées plus en détail en fin de chapitre, car elles ne concernent pas à proprement parler le conteneur léger.

L'interface `ApplicationContext` dispose, comme les précédentes, de plusieurs implémentations, permettant de choisir celle qui correspond le mieux aux besoins de l'application. Elle dispose ainsi de deux implémentations équivalentes à `XmlBeanFactory` : `FileSystemXmlApplicationContext`, pour lire les fichiers de configuration dans le système de fichiers, et `ClassPathXmlApplicationContext`, pour les lire dans le classpath.

Le code ci-dessous montre comment utiliser un contexte d'application dans une application Java exécutable en ligne de commande :

```
public static void main(String[] args) throws IOException {
   ApplicationContext context = new
      FileSystemXmlApplicationContext("applicationContext.xml");
   UnBean monBean = (UnBean) context.getBean("monBean");
   (...)
}
```

Nous commençons par initialiser un contexte d'application avec les objets définis dans le fichier **applicationContext.xml,** stocké dans le répertoire courant de l'application. Nous interrogeons ensuite le conteneur léger *via* la méthode `getBean` pour récupérer une instance de la classe `UnBean`.

Par convention, les noms des fichiers de définition des objets s'appellent **applicationContext.xml.** Si la définition des objets de l'application est séparée en plusieurs fichiers, une pratique recommandée dès lors que l'application manipule un grand nombre d'objets, les noms des fichiers sont préfixés par **applicationContext-** et conservent l'extension **.xml.** Bien entendu, le respect de cette norme n'est pas obligatoire.

Pour notre application exemple Tudu Lists, les fichiers suivants sont créés dans le répertoire **WEB-INF :**

- **applicationContext.xml :** contient la définition des services de Tudu Lists.
- **applicationContext-hibernate.xml :** contient la définition des DAO.
- **appplicationContext-security.xml :** contient la définition des objets gérant la sécurité de l'application.
- **applicationContext-dwr.xml :** contient la définition des objets utilisés par le framework AJAX DWR.
- **action-servlet.xml :** contient la définition des actions Struts.

Dans le cadre de ce chapitre, nous nous intéressons essentiellement au premier et au deuxième fichier de configuration. Le contenu des autres sera précisé au fil de l'ouvrage.

En plus de `FileSystemXmlApplicationContext` et `ClasspathXmlApplicationContext`, d'autres implémentations sont disponibles spécifiquement pour les applications Web, comme `XmlWebApplicationContext`, mais celles-ci sont abordées dans la partie II.

Notons que, dans le cas des applications Web utilisant Struts ou Spring MVC, la définition du contexte d'application s'effectue *via* le fichier de configuration standard **web.xml.** Il n'est donc pas nécessaire de le créer de manière programmatique, comme dans l'exemple précédent.

Par ailleurs, les objets de l'application développés pour ces frameworks, comme les actions Struts, n'ont pas besoin d'utiliser la méthode `getBean`. Ces objets étant eux-mêmes gérés par le conteneur, lequel est exécuté au lancement de l'application Web, l'injection de leurs dépendances se fait automatiquement.

En résumé

La notion de fabrique de Bean formalise l'accès aux objets gérés par le conteneur léger. La définition des objets gérés est formalisée pour sa part par la notion de registre de Bean.

Ces deux notions constituent le minimum requis pour utiliser le conteneur léger de Spring dans une application. Le contexte d'application étend ces deux notions en fournissant non seulement l'ensemble des services définis précédemment mais aussi des fonctionnalités supplémentaires utiles aux applications (comme l'internationalisation ou l'accès générique aux ressources).

La majorité des projets utilisant Spring utilisent ainsi la notion de contexte d'application dans une de ses implémentations. Dans tous les cas, la définition des objets à gérer se réalise de manière privilégiée sous forme XML à partir d'un langage formalisé par Spring sous forme de schéma.

Les sections suivantes se penchent sur la façon d'utiliser ce langage.

Définition d'un Bean

Dans Spring, la notion de Bean correspond à celle d'instance de classe. Cette classe peut être un JavaBean, mais ce n'est pas obligatoire, et il peut s'agir aussi d'une classe quelconque. L'essentiel est de fournir un constructeur ou un ensemble de modificateurs pour permettre à Spring d'initialiser l'objet correctement. Il est aussi possible d'utiliser des fabriques spécifiques pour les instanciations complexes, comme nous le verrons un peu plus loin avec les techniques avancées.

La définition d'un Bean peut s'effectuer de manière programmatique, *via* la méthode `registerBeanDefinition` de l'interface `BeanDefinitionRegistry`, mais il est nettement plus commode la plupart du temps d'externaliser cette définition dans un fichier de configuration XML. Dans cette section, nous nous concentrons sur cette dernière façon.

Les informations de base

La définition d'un Bean nécessite au minimum la fourniture de son type (sa classe) et de son nom. Il est aussi nécessaire de savoir si le Bean en question est un singleton ou non.

Nous allons voir comme indiquer au conteneur léger ces informations primordiales.

Structure du fichier de configuration

La définition des Beans s'effectue dans un fichier XML utilisant les schémas spécifiés par Spring. L'ensemble des définitions est encadré par le tag `beans`. C'est dans ce tag que sont spécifiés les différents schémas nécessaires. Ici, nous n'en utilisons qu'un, mais nous verrons qu'il existe deux autres schémas, un pour la POA et un autre pour la gestion des transactions.

Voici une illustration de l'utilisation de ce tag `beans` dans le fichier **applicationContext.xml** du projet Tudu Lists :

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

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"

       xsi:schemaLocation="
       http://www.springframework.org/schema/beans
       http://www.springframework.org/schema/beans/spring-beans.xsd
       http://www.springframework.org/schema/aop
       http://www.springframework.org/schema/aop/spring-aop.xsd
       http://www.springframework.org/schema/tx
       http://www.springframework.org/schema/tx/spring-tx.xsd">

   (...)<!-- définitions des beans -->

</beans>
```

Les fichiers de configuration des contextes d'application utilisent plusieurs espaces de nommage, avec chacun un schéma spécifique. L'espace de nommage par défaut est celui de la définition des Beans (première ligne en gras dans le code). Les espaces de nommage `aop` et `tx` concernent respectivement la POA et la gestion des transactions. Nous aurons l'occasion d'y revenir dans le cours des chapitres suivants.

La localisation des schémas est spécifiée par le paramètre `xsi:schemaLocation`. Ainsi, le schéma par défaut *http://www.springframework.org/schema/beans* se situe à l'URL *http://www.springframework.org/schema/beans/spring-beans.xsd* (deuxième et troisième lignes en gras).

Nommage des Beans

Une fois cette structure mise en place, nous pouvons définir nos Beans.

Nous utilisons pour cela le tag `bean` (à ne pas confondre avec le tag `beans` qui le contient) et fournissons au minimum deux informations : la classe et le nom du Bean.

Pour définir le Bean nommé `userManager`, de type `tudu.service.impl.UserManagerImpl`, nous pouvons écrire :

```
<bean id="userManager" class="tudu.service.impl.UserManagerImpl"/>
```

Le résultat de cette configuration est une instance `userManager`, dont les propriétés ne sont pas initialisées, car elles nécessitent une configuration spécifique, que nous détaillons plus loin dans ce chapitre. Le paramètre `id` spécifie l'identifiant unique (le nom) du Bean.

Il peut être utile de définir des alias. Il suffit pour cela d'utiliser le paramètre `name`, qui accepte plusieurs noms séparés par des virgules ou des espaces.

Dans notre exemple, nous pouvons définir les alias suivants :

```
<bean id="userManager" class="tudu.service.impl.UserManagerImpl"
      name="userManagerService,gestionUtilisateurs"/>
```

Sélection du mode d'instanciation

Par défaut, Spring considère que les Beans sont des singletons. Si nous appelons plusieurs fois la méthode `getBean` du contexte d'application, nous obtenons donc toujours le même objet.

Ce mode de fonctionnement est réglable *via* le paramètre `singleton`. Comme expliqué précédemment, dans la terminologie de Spring, les Beans qui ne sont pas des singletons sont appelés des prototypes. Si nous voulons signaler explicitement que `userManager` est un singleton, il nous suffit de compléter notre code de la façon suivante :

```
<bean id="userManager" class="tudu.service.impl.UserManagerImpl"
      singleton="true"/>
```

Pour le transformer en prototype, il suffit de remplacer la valeur `true` par `false`.

Les méthodes d'injection

Nous avons vu à la section précédente que, grâce à Spring, les applications n'ont plus à se préoccuper d'implémenter le design pattern fabrique, puisqu'il est fourni directement par le framework.

Cependant, l'essentiel de la puissance de Spring réside dans l'injection de dépendances, c'est-à-dire dans l'initialisation des propriétés d'un Bean, qu'elles soient simples (entier, réel, chaîne de caractères, etc.) ou qu'elles fassent référence à d'autres Beans gérés par le conteneur (nous parlons alors de collaborateurs).

Pour initialiser les propriétés d'un Bean, Spring propose deux méthodes d'injection : soit en utilisant les modificateurs du Bean, s'il en a, soit en utilisant un de ses constructeurs. Dans la plupart des cas, l'injection par modificateur est préférée à l'injection par constructeur, car elle présente l'avantage de fournir explicitement le nom de chacune des propriétés initialisées, ce qui n'est pas le cas avec un constructeur, puisque, en Java, les paramètres ne sont pas nommés.

L'injection par modificateur s'avère par ailleurs plus pratique pour l'héritage de configuration, que nous abordons en fin de chapitre. Par contre, l'utilisation de l'injection par constructeur est plus sécurisante, puisqu'elle assure que l'objet est correctement initialisé (en supposant que le constructeur soit correct). Il est toutefois possible de combiner les deux types d'injections au sein d'une même application.

L'injection par modificateur

L'injection par modificateur se paramètre au sein d'une définition de Bean en utilisant le tag `property`. Ce tag possède un paramètre `name` spécifiant le nom de la propriété à initialiser. Rappelons qu'un modificateur ne correspond pas forcément à un attribut de l'objet à initialiser et qu'il peut s'agir d'un traitement d'initialisation plus complexe.

Le tag `property` s'utilise en combinaison avec le tag `value`, qui sert à spécifier la valeur à affecter à la propriété lorsqu'il s'agit d'une propriété canonique, ou avec le tag `ref`, s'il s'agit d'un collaborateur.

Dans le fichier **applicationContext-hibernate.xml,** nous initialisons la propriété canonique `configLocation` du Bean `sessionFactory` de la manière suivante :

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

   <property name="configLocation">
      <value>classpath:hibernate.cfg.xml</value>
   </property>

</bean>
```

Il est possible d'avoir une écriture plus concise de cette configuration en utilisant le paramètre `value` du tag `property` en lieu et place du tag de même nom :

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

   <property name="configLocation"
             value="classpath:hibernate.cfg.xml"/>

</bean>
```

L'injection par constructeur

L'injection par constructeur se paramètre au sein d'une définition de Bean en spécifiant les paramètres du constructeur par le biais du tag `constructor-arg`.

Supposons que nous ayons une classe `UnBean` codée de la manière suivante :

```
public class UnBean {
   private String chaine;
   private int entier;

   public UnBean(String chaine, int entier) {
      this.chaine = chaine;
      this.entier = entier;
   }

   (...)
}
```

La configuration de ce Bean s'effectue ainsi :

```
<bean id="monBean" class="UnBean">
   <constructor-arg>
      <value>valeur</value>
   </constructor-arg>
   <constructor-arg>
      <value>10</value>
   </constructor-arg>
</bean>
```

Il est possible de changer l'ordre de définition des paramètres du constructeur en utilisant le paramètre `index` du tag `constructor-arg`. L'indexation se fait à partir de 0.

La configuration suivante est équivalente à la précédente, bien que nous ayons inversé l'ordre de définition des paramètres du constructeur :

```
<bean id="monBean" class="UnBean">
   <constructor-arg index="1">
      <value>10</value>
   </constructor-arg>
```

```
    <constructor-arg index="0">
       <value>valeur</value>
    </constructor-arg>
</bean>
```

Dans certains cas, il peut y avoir ambiguïté dans la définition du constructeur, empêchant Spring de choisir correctement ce dernier. Pour illustrer ce problème, ajoutons les deux constructeurs à la classe `UnBean` :

```
    public UnBean(String chaine) {
       this.chaine = chaine;
       this.entier = 0;
    }

    public UnBean(int entier) {
       this.chaine = "";
       this.entier = entier;
        }
```

Si nous écrivons la définition de Bean suivante, une ambiguïté apparaît, puisque les deux constructeurs peuvent être utilisés à partir de ce paramétrage :

```
<bean id="monBean" class="UnBean">
   <constructor-arg>
      <value>10</value>
   </constructor-arg>
</bean>
```

Par défaut, Spring sélectionne le premier constructeur supportant cette configuration, en l'occurrence celui qui initialise l'attribut `chaîne`. Pour lever l'ambiguïté, il est possible d'utiliser le paramètre `type` du tag `constructor-arg` :

```
<bean id="monBean" class="tudu.UnBean">
   <constructor-arg type="int">
      <value>10</value>
   </constructor-arg>
</bean>
```

Grâce à ce paramètre, nous sélectionnons le constructeur qui initialise `entier`. Pour sélectionner l'autre constructeur explicitement, nous pouvons écrire :

```
<bean id="monBean" class="tudu.UnBean">
   <constructor-arg type="java.lang.String">
      <value>10</value>
   </constructor-arg>
</bean>
```

Lorsque le type de l'argument est une classe, il est nécessaire de le qualifier complètement (c'est-à-dire en précisant son package), comme nous pouvons le constater pour le type `String`.

Comme pour l'injection par modificateur, le tag `constructor-arg` dispose d'une forme plus concise, ne nécessitant pas l'utilisation du tag `value`. Si nous reprenons notre dernière configuration, nous pouvons l'écrire de la manière suivante :

```
<bean id="monBean" class="UnBean">
   <constructor-arg type="java.lang.String" value="10"/>
</bean>
```

Injection des propriétés

Les exemples fournis à la section précédente nous ont permis de nous familiariser avec la configuration des propriétés d'un Bean. Nous allons maintenant voir plus en détail quelles sont les possibilités de Spring en la matière.

L'injection des propriétés est le mécanisme fondamental du conteneur léger. Nous allons voir dans cette section comment sont gérées les valeurs simples et les structures de données.

Injection de valeurs simples

Spring supporte l'injection de valeurs simples en convertissant les chaînes de caractères fournies au tag ou au paramètre `value` dans le type de la propriété à initialiser.

Outre les chaînes de caractères et les nombres, les types de propriétés supportés par Spring sont les suivants :

- Booléens.
- Type `char` et `java.lang.Character`.
- Type `java.util.Properties`.
- Type `java.util.Locale`.
- Type `java.net.URL`.
- Type `java.io.File`.
- Type `java.lang.Class`.
- Tableaux de bytes (la chaîne de caractères est transformée *via* la méthode `getBytes` de `String`).
- Tableaux de chaînes de caractères (chaque chaîne doit être séparée par une virgule selon le format CSV).

Afin d'illustrer l'utilisation de ces différentes valeurs, modifions notre classe `UnBean` en conséquence :

```
public class UnBean {
   private String chaine;
   private int entier;
   private float reel;
   private boolean booleen;
```

```
    private char caractere;
    private java.util.Properties proprietes;
    private java.util.Locale localisation;
    private java.net.URL url;
    private java.io.File fichier;
    private java.lang.Class classe;
    private byte[] tab2bytes;
    private String[] tab2chaines;

    // définition des accesseurs et modificateurs de chaque attribut
    (...)
}
```

Si nous utilisons l'injection par modificateur, plus explicite, la définition du Bean est la suivante :

```
<bean id="monBean" class="UnBean">
 <property name="chaine" value="valeur"/>
 <property name="entier" value="10"/>
 <property name="reel" value="10.5"/>
 <property name="booleen" value="true"/>
 <property name="caractere" value="a"/>
 <property name="proprietes"
   value="log4j.rootLogger=DEBUG,CONSOLE\nlog4j.logger.tudu=WARN"/>
 <property name="localisation" value="fr_FR"/>
 <property name="url" value="http://tudu.sf.net"/>
 <property name="fichier" value="file:c:\\temp\\test.txt"/>
 <property name="classe" value="java.lang.String"/>
 <property name="tab2bytes" value="valeur"/>
 <property name="tab2chaines" value="valeur1,valeur2"/>
</bean>
```

Pour les attributs `proprietes` et `localisation`, le format de la chaîne de caractères respecte le format attendu pour ces types de données (ce format est spécifié dans l'API J2SE). C'est notamment la raison pour laquelle `\n` figure dans l'initialisation de `proprietes`, car son format impose la définition d'une propriété par ligne.

Pour la propriété `fichier`, le format utilisé est celui des URL. Ce format supporte le protocole `classpath` introduit par Spring pour accéder aux fichiers se trouvant dans le classpath *(voir en fin de chapitre l'abstraction des accès aux ressources).*

Si Spring supporte les types les plus utilisés, pour des besoins spécifiques, il peut être nécessaire de supporter de nouveaux types. Nous étudions cette possibilité à la section dédiée aux techniques d'injection avancées.

Injection de la valeur *null*

Dans certaines situations, il est nécessaire d'initialiser explicitement une propriété à `null`. Pour cela, Spring propose le tag `null`.

La configuration suivante initialise `attr1` de `UnBean` à `null` en utilisant le tag `null` en lieu et place du tag `value` :

```
<bean id="monBean" class="tudu.UnBean">
   <constructor-arg type="java.lang.String">
      <null/>
   </constructor-arg>
</bean>
```

Bien entendu, le tag `null` est utilisable quelle que soit la méthode d'injection.

Injection de structures de données

Outre les valeurs simples, Spring supporte l'injection de structures de données. Ces dernières peuvent stocker soit des ensembles de valeurs simples (tag `value`), soit des objets gérés par le conteneur (tag `ref`), dont nous verrons la définition un peu plus loin.

Les structures de données supportées sont `java.util.Map`, `java.util.Set` et `java.util.List`. Leur initialisation utilise des tags spécifiques.

Outre ces trois types, Spring fournit des tags spécifiques pour initialiser les propriétés du type `java.util.Properties`. Ces tags sont plus lisibles que la configuration que nous avons utilisée précédemment. Notons que, contrairement aux structures de données précédentes, la classe `Properties` n'accepte que les chaînes de caractères, puisqu'il s'agit du seul type qu'elle est capable de manipuler.

Le type *java.util.Map*

L'interface `java.util.Map` représente un objet qui fait correspondre des clés à des valeurs. Une Map ne peut contenir de clés dupliquées (elles sont uniques), et une clé ne peut correspondre qu'à une seule valeur.

L'initialisation d'une propriété ayant ce type s'effectue grâce aux tags `map` et `entry`, à utiliser en combinaison avec le tag `property`.

Si nous ajoutons un attribut `map` de type `java.util.Map` à notre Bean `UnBean`, nous pouvons ajouter la ligne suivante dans notre configuration pour l'initialiser :

```
<property name="map">
   <map>
      <entry key="cle1">
         <value>valeur1</value>
      </entry>
      <entry key="cle2">
         <value>valeur2</value>
      </entry>
   </map>
</property>
```

Le tag `map` encadre la définition du contenu de la propriété, et le tag `entry` spécifie un couple clé-valeur contenu dans la propriété.

Il est possible de se passer de l'utilisation du tag `value` en utilisant le paramètre de même nom dans le tag `entry` :

```
        <entry key="cle1" value="valeur1"/>
        <entry key="cle2" value="valeur2"/>
```

Le type *java.util.Set*

L'interface `java.util.Set` est une collection d'éléments qui ne contient aucun dupliqua et au maximum une fois la valeur `null`. Ce type correspond à la notion mathématique d'ensemble.

L'initialisation d'une propriété ayant ce type s'effectue grâce au tag `set`, à utiliser en combinaison avec le tag `property`.

Si nous ajoutons un attribut `set` de type `java.util.Set` à notre Bean `UnBean`, nous devons ajouter la ligne suivante dans notre configuration :

```
<property name="set">
   <set>
      <value>valeur1</value>
      <value>valeur2</value>
   </set>
</property>
```

Le tag `set` encadre la définition du contenu de la propriété, et le tag `value` spécifie chaque valeur contenue dans la propriété.

Le type *java.util.List*

L'interface `java.util.List` est une collection ordonnée d'éléments. À ce titre, ce type permet un contrôle précis de la façon dont chaque élément est inséré dans la liste.

L'initialisation d'une propriété ayant ce type s'effectue grâce au tag `list`, à utiliser en combinaison avec le tag `property`.

Si nous ajoutons un attribut `list` de type `java.util.List` à notre Bean `UnBean`, nous devons ajouter la ligne suivante dans notre configuration :

```
<property name="list">
   <list>
      <value>valeur1</value>
      <value>valeur2</value>
   </list>
</property>
```

Le tag `list` encadre la définition du contenu de la propriété, et le tag `value` spécifie chaque valeur contenue dans la propriété. Comme il s'agit d'une collection ordonnée d'éléments, l'ordre de définition des valeurs est pris en compte.

Le type *java.util.Properties*

Comme nous l'avons vu précédemment, il est possible d'initialiser une propriété de type `java.util.Properties` directement à partir d'une chaîne de caractères. Spring propose cependant une autre méthode, plus lisible, à base de tags.

L'initialisation d'une propriété ayant ce type s'effectue grâce aux tags `props` et `prop`, à utiliser en combinaison avec le tag `property`.

Nous pouvons donc remplacer l'initialisation de l'attribut `proprietes` par la configuration suivante :

```
<property name="proprietes">
    <props>
       <prop key="log4j.rootLogger">
          DEBUG,CONSOLE
       </prop>
       <prop key="log4j.logger.tudu">
          WARN
       </prop>
    </props>
 </property>
```

Injection des collaborateurs

Nous venons de voir comment initialiser les valeurs simples ainsi que les structures de données au sein d'un objet géré par le conteneur léger. Spring supporte en standard les types les plus fréquemment rencontrés pour les attributs d'une classe. Il fournit en outre les transformateurs nécessaires pour convertir la configuration effectuée sous forme de chaînes de caractères en une valeur ayant le type convenu.

Nous allons nous intéresser à présent à l'injection des propriétés particulières que sont les collaborateurs.

Comme nous l'avons déjà indiqué, la terminologie de Spring désigne par le terme *collaborateur* une propriété d'un Bean étant elle-même un Bean géré par le conteneur léger. Pour réaliser l'injection des collaborateurs, Spring propose deux méthodes, l'une explicite, chaque collaborateur étant défini dans le fichier de configuration, et l'autre automatique, le conteneur léger décidant lui-même des injections à effectuer *(autowiring)* par introspection des Beans.

Une fois l'injection effectuée, il est possible de procéder à une vérification des dépendances afin de s'assurer que les Beans ont été correctement initialisés.

Injection explicite des collaborateurs

L'injection explicite des collaborateurs est la manière la plus sûre de gérer les dépendances entre les Beans gérés par le conteneur léger, car toute la mécanique d'injection est sous contrôle du développeur (contrairement à l'injection automatique, où Spring prend des décisions).

Comme expliqué précédemment, les collaborateurs sont des propriétés. À ce titre, ils se configurent à l'aide du tag `property`. Cependant, le tag ou le paramètre `value` sont ici remplacés par le tag ou le paramètre `ref`, signifiant référence.

Ainsi, la classe `tudu.service.impl.ConfigurationManagerImpl`, qui ne possède qu'une propriété collaborateur, `propertyDAO`, est définie de la manière suivante dans le fichier **applicationContext.xml :**

```
<bean id="configurationManager"
      class="tudu.service.impl.ConfigurationManagerImpl">
   <property name="propertyDAO">
      <ref bean="propertyDAO"/>
   </property>
</bean>
```

La ligne en gras montre comment utiliser le tag `ref`. Celui-ci possède un paramètre `bean` qui contient le nom de l'objet à injecter dans la propriété `propertyDAO`. Ce Bean `propertyDAO` est défini non pas dans le fichier **applicationContext.xml,** mais dans le fichier **applicationContext-hibernate.xml.** Si les deux Beans avaient été définis dans le même fichier de configuration, nous aurions pu remplacer le paramètre `bean` par le paramètre `local`, qui sert à spécifier les Beans locaux à un fichier.

Comme le tag `value`, le tag `ref` peut être remplacé par le paramètre `ref` dans le tag `property` pour une écriture plus concise. Ce paramètre est l'équivalent de `ref bean`. Il n'y a pas de raccourci pour `ref local`. Nous pouvons donc modifier l'exemple précédent de la manière suivante :

```
<bean id="configurationManager"
      class="tudu.service.impl.ConfigurationManagerImpl">
   <property name="propertyDAO" ref="propertyDAO"/>
</bean>
```

Pour injecter un Bean, il n'est pas nécessaire qu'il soit défini séparément auprès du conteneur léger. Il est possible de le déclarer pour une injection unique à l'intérieur d'un tag `property`. Pour `configurationManager`, par exemple, nous pouvons remplacer le tag `ref` par une définition interne du Bean `propertyDAO` :

```
<bean id="configurationManager"
      class="tudu.service.impl.ConfigurationManagerImpl">
   <property name="propertyDAO">
     <bean class="tudu.domain.dao.hibernate3.PropertyDAOHibernate">
        <property name="sessionFactory">
           <ref bean="sessionFactory" />
        </property>
     </bean>
    </property>
</bean>
```

Il n'est pas utile de nommer un Bean interne, puisqu'il n'est pas visible en dehors de la définition dans laquelle il s'inscrit.

Injection automatique des collaborateurs

Nous avons vu que l'injection explicite impliquait l'écriture de plusieurs lignes de configuration. Sur des projets de grande taille, les configurations peuvent rapidement devenir imposantes. Pour réduire de manière drastique le nombre de lignes de configuration, Spring propose un mécanisme d'injection automatique, appelé *autowiring.*

Ce mécanisme utilise des algorithmes de décision pour savoir quelles injections réaliser. L'autowiring est activé Bean par Bean par le biais du paramètre `autowire` du tag `bean`, ce qui permet d'utiliser ce mode d'injection de manière ciblée.

Par défaut, l'autowiring n'est pas activé. Son activation implique le choix de l'algorithme de décision parmi les quatre proposés par Spring :

- `byName` : Spring va rechercher un Bean ayant le même nom que la propriété pour réaliser l'injection.
- `byType` : Spring va rechercher un Bean ayant le même type que la propriété pour réaliser l'injection. S'il y en a plus d'un, une exception est générée. Si aucun Bean n'est trouvé, la propriété est initialisée à `null`.
- `constructor` : similaire à l'algorithme, mais fondé cette fois sur les paramètres du constructeur de la propriété.
- `autodetect` : sélectionne automatiquement la recherche par le type ou par le constructeur. La première est utilisée s'il existe un constructeur par défaut (c'est-à-dire sans arguments).

Par exemple, dans Tudu Lists, nous pourrions activer l'autowiring pour nos Beans manager déclarés dans **applicationContext.xml.** Nous pourrions utiliser indifféremment la méthode par nom (les propriétés ont le même nom que le Bean à injecter) ou par type (il n'y a qu'une seule implémentation par type). En revanche, la méthode par constructeur n'est pas adaptée, puisque les DAO n'ont que des constructeurs par défaut, donc indifférentiables. L'autodétection peut aussi être utilisée, mais elle revient au même que d'utiliser la méthode par type.

Si nous reprenons le Bean `configurationManager`, nous pouvons écrire sa définition de la manière suivante, plus concise que la précédente :

```
<bean id="configurationManager" autowire="byName"
          class="tudu.service.impl.ConfigurationManagerImpl"/>
```

Ainsi, l'utilisation de l'injection automatique permet de réduire de manière significative le volume du fichier de configuration. Cependant, nous conseillons de ne pas abuser de ce mode d'injection, car il ne fait plus apparaître explicitement les liaisons entre les différents Beans du conteneur.

Vérification des dépendances

Spring offre la possibilité de s'assurer que les propriétés des Beans ont été initialisées, soit de manière explicite, dans le fichier de configuration, soit de manière implicite, avec l'autowiring.

Comme pour l'autowiring, la vérification des dépendances se règle au niveau du tag `bean`, grâce au paramètre `dependency-check`, permettant ainsi de cibler l'utilisation de cette fonctionnalité.

La vérification des dépendances peut prendre l'une des trois formes suivantes :

- `simple` : seules les propriétés simples (int, float, etc.) et les structures de données sont vérifiées. Les collaborateurs ne le sont pas.
- `object` : seuls les collaborateurs sont vérifiés.
- `all` : combinaison des deux formes précédentes.

Si nous reprenons le Bean `configurationManager`, nous pouvons enclencher la vérification des dépendances de la manière suivante :

```
<bean id="configurationManager" dependency-check="object"
      class="tudu.service.impl.ConfigurationManagerImpl">
   <property name="propertyDAO" ref="propertyDAO"/>
</bean>
```

Techniques avancées

Nous venons de parcourir les différentes possibilités offertes par Spring pour l'injection des collaborateurs et la vérification des dépendances.

Nous abordons à présent des techniques avancées proposées par Spring pour aller encore plus loin dans la définition des Beans d'une application, notamment pour l'expression des dépendances et la création de Beans.

Définitions abstraites de Beans et héritage

Pour remédier au problème de duplication de lignes de configuration d'un Bean à un autre, Spring propose un mécanisme d'héritage de configuration, comme pour l'approche orientée objet. Il permet en outre de définir des Beans abstraits, c'est-à-dire qui ne sont pas instanciés par le conteneur, de façon à concentrer les lignes de configuration réutilisables.

Si nous analysons le fichier **applicationContext-hibernate.xml,** nous constatons que les DAO ont tous la même propriété, initialisée avec le même collaborateur. Nous pouvons donc utiliser avantageusement l'héritage de configuration pour rendre ce fichier plus concis.

Dans un premier temps, nous définissons un Bean abstrait, `abstractDAO`, destiné à recevoir la configuration de la propriété commune à tous les Beans DAO :

```
<bean id="abstractDAO" abstract="true">
   <property name="sessionFactory">
      <ref bean="sessionFactory" />
   </property>
</bean>
```

Il n'est pas utile de préciser un type de Bean puisque celui-ci n'est pas destiné à être instancié par le conteneur.

Il suffit ensuite de faire hériter chaque DAO de ce Bean pour se dispenser de configurer chaque fois la propriété `sessionFactory`. Le code du DAO `userDAO` devient ainsi :

```
<bean id="userDAO" parent="abstractDAO"
        class="tudu.domain.dao.hibernate3.UserDAOHibernate"/>
```

Le principe est exactement le même pour tous les autres DAO du fichier.

Notons que l'héritage de configuration ne nécessite pas de structure d'héritage au niveau objet (nos DAO sont indépendants les uns des autres). Ce mécanisme est strictement interne à la configuration des Beans.

Support de nouveaux types pour les valeurs simples

Les classes utilisées comme des types de propriétés n'ont pas forcément toutes vocation à être gérées sous forme de Beans par le conteneur léger. Il peut être plus intéressant de les traiter comme des valeurs simples, initialisées *via* une chaîne de caractères.

Inconnus de Spring, ces types doivent être accompagnés d'un transcodeur, à même de faire la conversion entre la chaîne de caractères et les attributs du type.

Nous devons donc créer un éditeur de propriétés, un concept issu du standard JavaBean, correspondant à notre transcodeur. C'est ce concept qu'utilise Spring pour supporter les différents types de valeurs simples que nous avons vus précédemment.

Supposons que nous définissions de la manière suivante une classe `UnType` destinée à être utilisée dans le Bean `monBean` par la propriété `attr` :

```
public class UnType {
   private String info;

   public String getInfo() {
      return info;
   }

   public void setInfo(String info) {
      this.info = info;
   }
}
```

Après avoir ajouté la nouvelle propriété dans `monBean` ainsi que son accesseur et son modificateur, nous écrivons la ligne de configuration suivante dans la définition du Bean :

```
<property name="attr" value="test"/>
```

Bien entendu, Spring ne connaissant pas le type `UnType`, une exception sera générée à l'exécution.

Pour créer un éditeur, il suffit de dériver la classe `org.springframework.beans.propertyeditors.PropertiesEditor`. La nouvelle classe surcharge la méthode `setAsText`, qui effectue la conversion proprement dite. Cette méthode reçoit pour unique paramètre la chaîne de caractères spécifiée dans la configuration. Une fois la conversion terminée, elle doit enregistrer le résultat obtenu, c'est-à-dire l'instance de `UnType` correctement initialisée, avec la méthode `setValue`.

Pour `UnType`, l'éditeur a la forme suivante :

```
import org.springframework.beans.propertyeditors.PropertiesEditor;

public class UnTypeEditor extends PropertiesEditor {

   public void setAsText(String arg)
     throws IllegalArgumentException {
       UnType instance = new UnType();
       instance.setInfo(arg);
       setValue(instance);
   }
}
```

Dans notre cas, la conversion consiste tout simplement à créer une instance de `UnType` et à initialiser son attribut `info` avec la chaîne de caractères issue du fichier de configuration.

Pour que Spring sache où trouver nos éditeurs de propriétés spécifiques, il existe deux possibilités : soit l'éditeur se trouve dans le même package que le type, et son nom est alors de la forme `TypeEditor` (où `Type` correspond au nom du type, dans notre exemple `UnTypeEditor`), soit en ajoutant ces quelques lignes dans le fichier de configuration :

```
<bean id="customEditorConfigurer"
class="org.springframework.beans.factory.config.CustomEditorConfigurer">
   <property name="customEditors">
      <map>
         <entry key="package1.UnType">
            <bean class="package2.UnTypeEditor"/>
         </entry>
      </map>
   </property>
</bean>
```

Support de fabriques de Bean spécifiques

Les exemples que nous avons donnés jusqu'à présent laissaient au conteneur léger la charge d'instancier et d'initialiser directement les Beans de notre application. Dans certains cas, il peut être nécessaire de ne pas déléguer cette création, afin de réaliser des traitements spécifiques, non exprimables *via* le langage XML de configuration, par exemple. Ce support est souvent utilisé pour intégrer des applications existantes non fondées sur Spring et l'injection de dépendances.

Afin de répondre à ce besoin, Spring propose plusieurs méthodes pour implémenter des fabriques de Bean spécifiques.

Utilisation d'une fabrique sous forme de Bean classique

La méthode la plus simple consiste à créer une classe fabrique disposant d'une méthode sans paramètre renvoyant une instance du Bean attendu.

Par exemple, pour notre Bean `monBean`, nous pouvons développer une classe `UneFabrique`, dont la méthode `creeInstance` renvoie une instance vide de `UnBean` :

```
public class UneFabrique {

   public UnBean creeInstance() {
      return new UnBean();
   }

}
```

Il suffit ensuite d'indiquer la classe préalablement définie sous forme de Bean et la méthode de fabrication dans la configuration du Bean cible grâce aux paramètres `factory-bean` et `factory-method` :

```
<bean id="maFabrique" class="UneFabrique"/>

<bean id="monBean" class="UnBean"
      factory-bean="maFabrique" factory-method="creeInstance">
   (...)<!-- Initialisation des propriétés si nécessaire-->
</bean>
```

Bien entendu, `maFabrique` étant un Bean, ses propriétés peuvent être initialisées par Spring.

Utilisation de l'interface *FactoryBean*

Une méthode plus complexe consiste à implémenter l'interface `FactoryBean` du package `org.springframework.beans.factory`. Cette interface est très utilisée par Spring en interne pour définir des fabriques spécialisées pour certains types complexes.

Par convention, le nom des implémentations de cette interface est suffixé par `FactoryBean`. Cette technique s'avère pratique pour récupérer des instances qui ne peuvent être créées directement avec un `new`.

Par exemple, une fabrique de ce type est utilisée pour créer une fabrique de session Hibernate (`SessionFactory`), l'outil de mapping objet-relationnel que nous aurons l'occasion d'aborder au chapitre 11 :

```
<bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
   <property name="configLocation"
             value="classpath:hibernate.cfg.xml"/>
</bean>
```

La fabrique se déclare sous la forme d'un Bean, comme avec la méthode précédente, la seule différence étant qu'il n'est pas nécessaire de spécifier la fonction de création des instances du Bean. En fait, la fabrique se substitue au type du Bean.

En effet, les références à ce Bean correspondent non pas à l'instance de la fabrique, mais aux instances qu'elle crée. C'est la raison pour laquelle `sessionFactory` est utilisé comme un Bean classique par les DAO, comme le montre l'extrait du fichier **applicationContext-hibernate.xml** ci-dessous :

```
<bean id="todoDAO"
      class="tudu.domain.dao.hibernate3.TodoDAOHibernate">
    <property name="sessionFactory">
        <ref bean="sessionFactory" />
    </property>
</bean>
```

Si nous reprenons notre exemple précédent, la fabrique prend la forme suivante :

```
import org.springframework.beans.factory.FactoryBean;

public class UneFabrique implements FactoryBean {
   public Object getObject() throws Exception {
      return new UnBean();
   }

   public Class getObjectType() {
      return UnBean.class;
   }

   public boolean isSingleton() {
      return false;
   }
}
```

La méthode `getObject` renvoie l'instance demandée à la fabrique, tandis que la méthode `getObjectType` renvoie le type de Bean créé par la fabrique. La fonction `isSingleton` indique pour sa part si la fabrique crée des singletons ou des prototypes.

Nous pouvons maintenant configurer `monBean` pour utiliser cette nouvelle fabrique :

```
<bean id="monBean" class="UneFabrique" />
```

Notons la disparition de l'initialisation de propriétés de `UnBean`. L'initialisation qui doit figurer à la place est celle de la fabrique du Bean, et non du Bean en lui-même. Or `UneFabrique` n'ayant pas de propriétés, il n'y a pas lieu à avoir d'initialisation dans cette définition.

En conclusion, par rapport à la méthode précédente, la fabrique de Bean a ici l'entière responsabilité de la création de l'instance du Bean ainsi que de son initialisation. Remarquons que nous pouvons récupérer l'instance du `FactoryBean` lui-même en préfixant le nom du Bean avec `&`.

Support des méthodes de recherche

Certaines classes ont besoin de méthodes spécifiques pour rechercher un composant de l'application (ou externe) dont elles ont besoin. Plutôt que de les développer directement au sein de ces classes, Spring permet de les externaliser au sein de Beans dédiés. Ces Beans dédiés sont ensuite sollicités *via* une méthode définie dans les classes utilisatrices. Cette méthode peut être abstraite ou concrète. Dans le dernier cas, le traitement associé est remplacé par celui du Bean dédié.

Supposons que `monBean` ait besoin d'une instance de `UnService`, dont la récupération nécessite un traitement particulier, plus complexe qu'un simple `new` ou qu'une injection. Pour cela, nous définissons une méthode abstraite, appelée `getService`. Le code de cette méthode est externalisé dans le Bean `serviceFinder`, qui possède une méthode concrète de même nom.

Grâce à la configuration suivante, utilisant le tag `lookup-method`, Spring remplace la méthode abstraite définie dans `monBean` par la méthode concrète définie dans `serviceFinder` :

```
<bean id="serviceFinder" class="ServiceFinder"/>

<bean id="monBean" class="UnBean">
   <lookup-method name="getService" bean="serviceFinder"/>
</bean>
```

Cette substitution s'effectuant directement au niveau du bytecode de l'application, la JVM considère que `monBean` est une classe concrète, alors que son code source mentionne l'existence d'une méthode abstraite.

En résumé

Nous avons vu dans cette section l'essentiel des outils fournis par Spring pour initialiser les objets et gérer leurs dépendances.

En supportant plusieurs modèles d'injection et d'instanciation (injection par modificateur, par constructeur, explicite, automatique ou utilisation d'une fabrique spécifique), Spring est capable de gérer tout type de Bean au sein de son conteneur léger.

Cycle de vie des Beans et interactions avec le conteneur

Au sein du conteneur léger, chaque Bean suit un cycle de vie qui va de sa création jusqu'à sa destruction. La gestion de ce cycle de vie est de la responsabilité du conteneur léger. Spring propose cependant divers moyens pour que les Beans puissent être avertis en fonction des différentes étapes de leur cycle de vie.

Spring permet en outre aux objets d'avoir un accès direct à la fabrique de Bean ou au contexte d'application grâce à des interfaces spécifiques, que les Beans intéressés doivent implémenter.

Enfin, Spring offre la possibilité de mettre en place des post-processeurs destinés à effectuer des traitements, suite aux opérations exécutées par le conteneur léger.

Cycle de vie des Beans

Le cycle de vie de chaque Bean comporte une naissance et une mort. Dans le cadre du conteneur léger de Spring, la naissance de l'ensemble des Beans s'effectue au démarrage de celui-ci par défaut. Cela induit un temps de chargement plus long de l'application mais présente l'avantage de s'assurer dès le démarrage que la création des Beans ne posera pas de problème. Ce comportement peut être modifié en positionnant le paramètre `lazy-init` du tag `bean` à `false`.

Si nous désirons signaler que le Bean `userManager` ne doit être créé qu'au dernier moment, il suffit d'écrire le code suivant :

```
<bean id="userManager" class="tudu.service.impl.UserManagerImpl"
      lazy-init="true"/>
```

La mort des Beans dépend de leur nature. S'il s'agit de prototypes, ceux-ci disparaissent dès lors que plus aucun objet ne les référence et que le ramasse-miettes a fait son œuvre. Spring ne conservant pas de référence en interne pour les prototypes, il n'a pas « conscience » de leur mort. Par contre, il conserve une référence pour chaque singleton dont il a la charge. Il a donc « conscience » de leur mort, ce qui permet de réaliser des traitements lorsque celle-ci survient.

Traitement lors de la création d'un Bean

Pour exécuter un traitement lors de la création d'un Bean spécifique, Spring propose deux méthodes. La première consiste à utiliser le paramètre `init` du tag `bean`, et la seconde à faire implémenter au Bean l'interface `InitializingBean` du package `org.springframework.beans.factory`.

Nous conseillons d'utiliser la première méthode, car elle est moins intrusive dans le code de l'application que la seconde, qui oblige à créer une dépendance explicite entre l'application et l'API de Spring.

Si nous voulons exécuter un traitement lors de la création de `monBean` en utilisant la première méthode, il suffit d'initialiser le paramètre `init` du tag `bean` en lui indiquant le nom de la méthode `UnBean` à appeler :

```
<bean id="monBean" class="UnBean" init="initialise">
   (...)
</bean>
```

Ici, la méthode `initialise` de la classe `UnBean` est appelée dès que `monBean` est créé. Cette méthode ne prend aucun paramètre et ne renvoie aucun résultat (`void`).

La seconde méthode nécessite que la classe `UnBean` implémente l'interface `InitializingBean` :

```
import org.springframework.beans.factory.InitializingBean;
(...)

public class UnBean implements InitializingBean {

   (...)
   public void afterPropertiesSet() {
      // intialise
   }
}
```

La méthode `afterPropertiesSet` est l'équivalent de la méthode `initialise` abordée précédemment.

Traitement lors de la destruction d'un Bean

Comme pour la création d'un Bean, Spring propose deux méthodes pour exécuter des traitements lors de sa destruction. La première consiste à utiliser le paramètre `destroy-method` du tag `bean`, et la seconde à faire implémenter au Bean l'interface `DisposableBean` du package `org.springframework.beans.factory`. Là encore, nous conseillons d'utiliser la première méthode plutôt que la seconde.

Si nous voulons exécuter un traitement lors de la destruction de `monBean` en utilisant la première méthode, il suffit d'initialiser le paramètre `destroy-method` du tag `bean` en lui indiquant le nom de la méthode `UnBean` à appeler :

```
<bean id="monBean" class="UnBean" destroy-method="detruit">
   (...)
</bean>
```

Ici, la méthode `detruit` de la classe `UnBean` est appelée dès que `monBean` est en passe d'être détruit. Cette méthode ne prend aucun paramètre.

La seconde méthode nécessite que la classe `UnBean` implémente l'interface `DisposableBean` :

```
import org.springframework.beans.factory.DisposableBean;
(...)

public class UnBean implements DisposableBean {

   (...)
```

```
    public void destroy() {
       // detruit
    }
}
```

La méthode `destroy` est l'équivalent de la méthode `detruit` abordée précédemment.

Récupération du nom du Bean

Dans certains cas, il peut être utile que la classe d'un Bean connaisse le nom de ce dernier. Pour permettre cela, Spring dispose de l'interface `BeanNameAware` du package `org.springframework.beans.factory`.

Cette interface dispose d'une méthode unique, `setBeanName`, qui est appelée par le conteneur léger pour indiquer à l'implémentation le nom du Bean auquel elle correspond. Nous conseillons de limiter au maximum l'utilisation de cette interface, car elle crée une dépendance explicite avec l'API Spring.

Nous pouvons modifier la classe `UnBean` afin qu'elle ait accès au nom du Bean correspondant à ses instances :

```
import org.springframework.beans.factory.BeanNameAware;
(...)

public class UnBean implements BeanNameAware {
   private String beanName;
   (...)
   public void setBeanName(String name) {
      this.beanName = name;
   }
}
```

Accès à la fabrique de Bean ou au contexte d'application

Certains Beans peuvent avoir besoin d'accéder à la fabrique de Bean ou au contexte d'application pour leurs traitements. Pour cela, Spring fournit deux interfaces, `BeanFactoryAware` dans le package `org.springframework.beans.factory` pour la fabrique de Bean et `ApplicationContextAware` dans le package `org.springframework.context` pour le contexte d'application.

Nous pouvons modifier la classe `UnBean` afin qu'elle ait accès à la fabrique de Bean :

```
import org.springframework.beans.factory.BeanFactoryAware;
(...)

public class UnBean implements BeanFactoryAware {
   private BeanFactory beanFactory;
   (...)
   public void setBeanFactory(BeanFactory beanFactory) {
      this.beanFactory = beanFactory;
   }
}
```

Pour accéder aux fonctionnalités supplémentaires offertes par les différentes implémentations de cette interface, il est bien entendu possible d'effectuer un transtypage *(cast)*.

De la même manière, la classe `UnBean` peut avoir accès au contexte d'application :

```
import org.springframework.beans.context.ApplicationContextAware;
(...)

public class UnBean implements ApplicationContextAware {
   private ApplicationContext applicationContext;
   (...)
   public void setApplicationContext(ApplicationContext ac) {
      this.applicationContext = ac;
   }
```

Là encore, il est possible d'effectuer un transtypage pour accéder aux fonctionnalités spécifiques des implémentations de cette interface.

Les post-processeurs

Spring définit plusieurs interfaces permettant de créer des post-processeurs. Ces post-processeurs sont appelés à la fin de certaines opérations spécifiques effectuées par le conteneur léger. Ils sont ainsi en mesure d'influer sur le résultat de ces opérations.

Nous détaillons dans cette section les deux post-processeurs de base de Spring : le post-processeur de Bean et le post-processeur de fabrique de Bean.

Les post-processeurs sont automatiquement chargés par les contextes d'application dès lors qu'ils sont définis sous forme de Bean dans le fichier de configuration. Si l'application utilise seulement une fabrique de Bean, la solution est spécifique du type de post-processeur *(voir plus loin)*.

Le post-processeur de Bean

Un post-processeur de Bean est appelé par le conteneur léger dès qu'un nouveau Bean est créé. Deux moments sont identifiés dans le processus de création : avant l'initialisation des propriétés du Bean et après. Cette notion est matérialisée par l'interface `BeanPostProcessor`, définie dans le package `org.springframework.beans.factory.config`.

Pour illustrer l'utilisation de cette interface, nous pouvons créer un post-processeur de Bean, qui signalera sur la console toutes les créations de Beans, avant et après leur initialisation :

```
import org.springframework.beans.factory.config.BeanPostProcessor;

public class PostProcBean implements BeanPostProcessor {

   public Object postProcessAfterInitialization(Object bean
                                                ,String beanName) {
      System.out.println("Après initialisation de "+beanName);
      return bean;
   }
```

```
    public Object postProcessBeforeInitialization(Object bean
                                                  ,String beanName) {
        System.out.println("Avant initialisation de "+beanName);
        return bean;
    }
}
```

Les deux méthodes `postProcess` renvoient l'instance du Bean qu'elles reçoivent en paramètre. En effet, puisqu'elles sont en mesure de modifier ce `bean`, elles doivent renvoyer le résultat de leurs opérations.

Pour que le contexte d'application charge ce post-processeur, il est nécessaire de le définir sous forme de Bean :

```
<bean id="postProcBean" class="PostProcBean"/>
```

Notre post-processeur sera ainsi appelé avant et après chaque initialisation d'un Bean au sein du conteneur léger.

Si l'application n'utilise pas de contexte d'application, il est nécessaire d'enregistrer le post-processeur auprès de la fabrique de Bean. Pour cela, nous utilisons la méthode `addBeanPostProcessor` de l'interface `ConfigurableBeanFactory`. Bien entendu, cela suppose que l'implémentation de la fabrique de Bean implémente cette interface, mais c'est généralement le cas, notamment pour `XmlBeanFactory`.

Le post-processeur de fabrique de Bean

Le post-processeur de fabrique de Bean est utilisé pour modifier la configuration de la fabrique de Bean suite à sa création. Cette notion est matérialisée par l'interface `BeanFactoryPostProcessor` du package `org.springframework.beans.factory.config`.

Comme il s'agit ici de configuration, seules les fabriques implémentant l'interface `ConfigurableBeanFactory` sont supportées. Le code suivant montre comment créer un post-processeur de fabrique de Bean :

```
import org.springframework.beans.factory.config.BeanFactoryPostProcessor;
import org.springframework.beans.factory.config
.ConfigurableListableBeanFactory;

public class PostProcFactBean implements BeanFactoryPostProcessor {

    public void postProcessBeanFactory
    (ConfigurableListableBeanFactory beanFactory) {
        //modification des paramètres de la fabrique
    }
}
```

À nouveau, il est nécessaire de définir ce post-processeur sous forme de Bean dans le fichier de configuration de Spring afin que le contexte d'application le charge automatiquement.

Si l'application n'utilise pas de contexte d'application, il est nécessaire d'exécuter manuellement le post-processeur après que la fabrique de Bean a été créée, en appelant la méthode `postProcessBeanFactory` dans le code de l'application.

Fonctionnalités additionnelles du contexte d'application

Nous avons passé en revue l'essentiel des fonctionnalités du conteneur léger. La notion de contexte d'application, qui englobe le conteneur léger, offre toutefois des fonctions utilitaires supplémentaires, qui sont utiles aux applications.

Nous détaillons dans cette section les deux principales d'entre elles que sont le support de l'internationalisation et l'abstraction des accès aux ressources.

Support de l'internationalisation

L'internationalisation des applications avec Java est assurée *via* des fichiers de propriétés (un par langue) associant une clé et un message. En fonction de la localisation de l'application, la JVM sélectionne le fichier de propriétés adéquat, qui doit se trouver dans le classpath.

Elle se fonde pour cela sur le nom du fichier, qui doit être suffixé par le code ISO du pays (par exemple **messages_FR.properties** pour les messages en français). Si le fichier de propriétés ne spécifie pas de code ISO, il est considéré comme le fichier par défaut qui sera utilisé si l'application n'est pas en mesure d'identifier la langue dans laquelle elle doit fonctionner.

Pour en savoir plus sur le format de ces fichiers, nous invitons le lecteur à lire la javadoc concernant la classe `java.util.ResourceBundle`, qui est celle utilisée pour accéder aux messages.

Pour Tudu Lists, le seul fichier de message fourni est **messages.properties,** qui se trouve dans le répertoire **JavaSource.** Dans notre application exemple, les messages sont uniquement utilisés dans les pages JSP. Nous n'avons donc pas besoin du support de Spring dans ce domaine, la taglib standard JSTL étant suffisante.

Si nous avions eu besoin d'utiliser un fichier de propriétés au sein même de nos Beans, une alternative se serait présentée : soit utiliser la classe `ResourceBundle` directement, soit bénéficier du support de Spring. L'avantage de la première solution est qu'elle est parfaitement standard, tandis que celui de la seconde est d'offrir davantage de possibilités, principalement l'agrégation de `ResourceBundles`.

Spring dispose de sa propre notion de `ResourceBundle`, qui permet d'agréger le contenu de plusieurs fichiers de propriétés. Pour effectuer cette opération, il suffit de créer un Bean de type `ResourceBundleMessageSource` (package `org.springframework.context.support`) de la manière suivante :

```
<bean id="messageSource"
class="org.springframework.context.support.ResourceBundleMessageSource">
   <property name="basenames">
```

```
        <list>
          <value>messages</value>
          <value>exceptions</value>
        </list>
      </property>
    </bean>
```

Ce type possède une propriété `basenames`, qui contient la liste des noms des `ResourceBundles` à agréger (sans le code ISO). Dans cet exemple, nous avons spécifié deux `ResourceBundles`, `msg` et `exceptions`. Pour la langue française, Spring charge donc **messages_FR.properties** et **exceptions_FR.properties** ou les fichiers par défaut (c'est-à-dire sans le code ISO en suffixe), si les précédents n'existent pas.

Pour récupérer un message, il suffit d'appeler l'une des méthodes `getMessage` du contexte d'application. Pour rappel, nous pouvons obtenir le contexte d'application en implémentant l'interface `ApplicationContextAware`.

Le code suivant illustre l'utilisation des deux méthodes `getMessage` avec le contenu de **messages.properties** de Tudu Lists :

```
String welcome1 = applicationContext
                  .getMessage("menu.welcome",null,locale);
String welcome2 = applicationContext
                 .getMessage("menu.welcome",null,"welcome",locale);
```

Le paramètre `locale` (de type `java.util.Locale`) spécifie la localisation à utiliser pour sélectionner la langue du message. S'il vaut `null`, le choix de la langue est réalisé par la JVM.

Le dernier appel permet de définir une valeur par défaut si le message correspondant à la clé n'est pas trouvé. Le second génère une exception si nous nous trouvons dans ce cas de figure.

Le paramètre à `null` dans les deux appels correspond aux paramètres à substituer dans le message. Il s'agit d'un tableau d'objets, généralement des chaînes de caractères. En effet, Java permet de définir dans un texte des variables qui seront spécifiées au moment de l'exécution.

Dans **messages.properties,** le message `register.user.already.exists` prend typiquement un paramètre qui est, en l'occurrence, le nom de l'utilisateur qui existe déjà :

```
register.user.already.exists=User login "{0}" already exists.
```

Pour récupérer ce message correctement paramétré, il suffit d'écrire :

```
String erreur = applicationContext
                .getMessage("register.user.already.exists"
                            ,new Object[] {username},locale);
```

La variable `username` contient bien entendu le nom de l'utilisateur qui tente de s'enregistrer.

Abstraction des accès aux ressources

Les ressources (fichiers) utilisées par une application peuvent avoir de multiples supports. Il peut s'agir de ressources disponibles sur un serveur Web *via* le protocole HTTP, sur le système de fichiers de la machine, dans le classpath, etc.

Java ne propose malheureusement pas de mécanisme d'accès unique à ces ressources en faisant abstraction du support qu'elles utilisent. Par exemple, pour accéder à un fichier sur un serveur Web, nous disposons de la classe `java.net.URL`, mais celle-ci n'est pas utilisable pour les ressources accessibles depuis le classpath.

Pour combler ce manque, Spring définit deux notions : la ressource et le chargeur de ressources. Comme d'habitude, ces deux notions sont matérialisées sous forme d'interfaces.

L'interface `Resource` possède plusieurs implémentations, disponibles dans le package `org.springframework.core.io` : `FileSystemResource` pour les ressources stockées dans le système de fichiers, `ClassPathSystemResource` pour celles qui sont accessibles depuis le classpath, etc. La notion de ressource ne présuppose pas l'existence de la ressource. C'est la raison pour laquelle, elle dispose de la fonction booléenne `exists`.

La notion de chargeur de ressource est implémentée par les contextes d'application qui disposent de la méthode `getResource`. Cette méthode, définie par l'interface `ResourceLoader`, prend pour unique paramètre le chemin de la ressource sous forme d'une chaîne de caractères. Ce chemin peut être soit relatif, soit absolu. Dans le second cas, le format de ce paramètre est celui d'une URL. Cette URL supporte `HTTP`, `file` et `classpath`, afin de désigner respectivement les ressources disponibles sur un serveur Web, dans le système de fichiers ou depuis le classpath.

Spring propose une interface `ResourceLoaderAware` pour les Beans ayant besoin de charger des ressources. Il leur suffit de l'implémenter pour que le contexte d'application se charge de leur fournir l'instance de chargeur dont ils ont besoin *via* la méthode `setResourceLoader`. Bien entendu, si le Bean a déjà accès au contexte d'application (*via* `ApplicationContextAware`), l'implémentation de cette interface n'est pas nécessaire, puisque ce dernier l'implémente directement.

Si notre Bean `monBean` a besoin de charger des ressources et qu'il n'ait pas accès au contexte d'application, nous pouvons écrire :

```
import org.springframework.context.ResourceLoaderAware;
(...)

public class UnBean implements ResourceLoaderAware {
   private ResourceLoader loader;
   (...)

   public void setResourceLoader(ResourceLoader loader) {
      this.loader = loader;
   }
}
```

Une fois un chargeur disponible, nous utilisons ses services pour accéder aux ressources.

Le code suivant utilise le contexte d'application pour récupérer une ressource disponible sur le Web, une ressource stockée dans le système de fichiers et une ressource accessible depuis le classpath :

```
Resource r1 = applicationContext.getResource("http://tudu.sf.net/index.html");

Resource r2 = applicationContext.getResource("file:c:/temp/fichier.txt");

Resource r3 = applicationContext.getResource("classpath:tudu/domain/model/User.hbm.xml");
```

Une fois les ressources récupérées, elles peuvent être manipulées à l'aide des méthodes `getFile` et `getURL` de l'interface `Resource`, puisque ces deux méthodes renvoient respectivement un objet de type `java.io.File` ou un objet de type `java.net.URL`.

Ajoutons que Spring offre la possibilité de transformer des instances de la classe `java.io.InputStream` et des tableaux de bytes en ressources. Pour cela, il suffit d'instancier respectivement les classes `InputStreamResource` et `ByteArrayResource` avec le flux ou le tableau en paramètre. Il n'est pas ici nécessaire d'utiliser un chargeur de ressource.

Publication d'événements

Le contexte d'application a la possibilité de publier des événements destinés à des observateurs. Cette publication s'effectue *via* la méthode `publishEvent` définie par l'interface `org.springframework.context.ApplicationEventPublisher` et implémentée par les contextes d'application.

Un événement est un objet héritant de la classe `ApplicationEvent` du package `org.springframework.context`. Cette classe définit deux méthodes : `getTimeStamp`, qui donne l'instant auquel l'événement a été généré, et `getSource`, qui donne l'objet source de l'événement. La source de l'événement est initialisée *via* l'unique paramètre du constructeur de cette classe.

Nous pouvons définir un événement de la manière suivante :

```
import org.springframework.context.ApplicationEvent;

public class UnEvenement extends ApplicationEvent {

   public UnEvenement(Object source) {
      super(source);
   }

}
```

Un observateur est un objet dont la classe implémente l'interface `ApplicationListener`. Cet observateur reçoit tous les événements publiés par le contexte d'application. Cette interface spécifie une méthode `onApplicationEvent`, qui est appelée dès qu'un événement est généré. Cette méthode prend en unique paramètre l'événement en question.

Pour traiter l'événement précédent, nous pouvons développer l'observateur suivant :

```
import org.springframework.context.ApplicationListener;
import org.springframework.context.ApplicationEvent;

public class UnObservateur implements ApplicationListener {

   public void onApplicationEvent(ApplicationEvent evt) {
      if(evt instanceof UnEvenement) {
         // traitement de l'événement
      }
   }
}
```

Cet observateur doit ensuite être déclaré sous forme de Bean dans le fichier de configuration, afin que le contexte d'application puisse l'avertir à chaque publication d'événement :

```
<bean id="unObservateur" class="UnObservateur"/>
```

Pour générer un événement, il suffit de créer une instance de la classe `UnEvenement` et de la publier *via* la méthode `publishEvent` du contexte d'application :

```
applicationContext.publishEvent(new UnEvenement());
```

Bien entendu, pour que cette opération soit possible, il est nécessaire que le Bean générant cet événement ait accès au contexte de l'application. Sinon, il a la possibilité d'implémenter l'interface `ApplicationEventPublisherAware` du package `org.springframework.context`, qui fonctionne selon les mêmes principes que les précédentes interfaces de type `Aware`.

Pour illustrer l'utilisation de cette interface, nous pouvons modifier notre classe `UnBean` de la manière suivante :

```
import org.springframework.context.ApplicationEventPublisherAware;
import org.springframework.context.ApplicationEventPublisher;

public class UnBean implements ApplicationEventPublisherAware {
   (...)
   private ApplicationEventPublisher publisher;

   public void setApplicationEventPublisher
   (ApplicationEventPublisher publisher) {
      this.publisher = publisher;
   }
   (...)
}
```

Le Bean `monBean` peut de la sorte générer des événements en utilisant la méthode `publishEvent` de son attribut `publisher`.

L'application Tudu Lists utilise l'interface `ApplicationListener` pour afficher un message au démarrage de l'application Web. Cet affichage est géré dans la classe

`tudu.service.impl.TodoListsManagerImpl`, dont nous reproduisons une partie du code ci-dessous :

```
(...)
import org.springframework.context.ApplicationListener;
import org.springframework.context.event.ContextRefreshedEvent;

public class TodoListsManagerImpl implements TodoListsManager,
                                             ApplicationListener {

   (...)
   public void onApplicationEvent(ApplicationEvent event) {
      if (event instanceof ContextRefreshedEvent) {
         log.warn("Spring context is starting up : "
                  + event.toString());
      }
    }

   (...)
}
```

Cette classe utilise l'événement standard de Spring `ContextRefreshedEvent` pour détecter le chargement du contexte d'application signifiant le démarrage de Tudu Lists.

Conclusion

Nous avons vu dans ce chapitre comment utiliser les différentes fonctionnalités du conteneur léger de Spring.

Nous avons étudié les différentes méthodes d'injection, ainsi que la façon dont sont créés les Beans gérés par le conteneur léger et leur cycle de vie. Nous avons vu en outre comment les Beans peuvent s'interfacer étroitement avec le conteneur *via* l'implémentation d'interfaces. Ce dernier point doit cependant être limité au strict nécessaire, car il crée une dépendance explicite entre les Beans de l'application et l'API de Spring.

Pour terminer, nous avons décrit les fonctionnalités additionnelles qu'apporte le contexte d'application à la fabrique de Bean.

Si nous reprenons la modélisation proposée pour Tudu Lists au chapitre 2 *(voir figure 2.7)*, nous constatons que le contenu des fichiers de configuration des projets **applicationContext.xml** et **applicationContext-hibernate.xml** la reproduit exactement. Spring nous a donc permis d'atteindre la modélisation optimale que nous avons définie.

Ce chapitre a effectué une plongée dans le cœur même de Spring, sur lequel repose une grande partie de l'édifice. Certaines activités, comme la gestion des transactions, ne peuvent toutefois se satisfaire uniquement du conteneur léger. C'est la raison pour laquelle Spring s'est doté d'un support de la POA, que nous présentons au chapitre suivant.

4

Les concepts de la POA

La POA (programmation orientée aspect), ou AOP (Aspect-Oriented Programming), est un nouveau paradigme, dont les fondations ont été définies au centre de recherche Xerox, à Palo Alto, au milieu des années 1990. Par paradigme, nous entendons un ensemble de principes qui structurent la manière de modéliser les applications informatiques et, en conséquence, la façon de les développer.

La POA a émergé à la suite de différents travaux de recherche, dont l'objectif était d'améliorer la modularité des logiciels afin de faciliter la réutilisation et la maintenance. Elle ne remet pas en cause les autres paradigmes de programmation, comme l'approche procédurale ou l'approche objet, mais les étend en offrant des mécanismes complémentaires pour mieux modulariser les différentes préoccupations d'une application et améliorer ainsi leur séparation.

Le conteneur léger de Spring étant de conception orientée objet, il ne peut aller au-delà des limites fondamentales de ce paradigme. C'est la raison pour laquelle Spring dispose de son propre framework de POA, qui lui permet d'aller plus loin dans la séparation des préoccupations.

Pour les lecteurs qui ne connaîtraient pas ce nouveau paradigme de programmation, ce chapitre donne une vision synthétique des notions clés de la POA disponibles avec Spring. Nous invitons ceux qui désireraient en savoir plus sur la POA à lire l'ouvrage *Programmation orientée aspect pour Java/J2EE*, publié en 2004 aux éditions Eyrolles.

Comme nous l'avons fait au chapitre 2 pour les concepts des conteneurs légers, nous commençons par décrire les problématiques rencontrées par les approches classiques de modélisation puis montrons en quoi la POA propose une solution plus élégante, avec ses notions d'aspect, de point de jonction, de coupe, de greffon et d'introduction.

Limites de l'approche orientée objet

Comme indiqué au chapitre 2, une bonne conception est une conception qui minimise la rigidité, la fragilité, l'immobilité et l'invérifiabilité d'une application.

L'idéal pour une modélisation logicielle est d'aboutir à une séparation totale entre les différentes préoccupations d'une application afin que chacune d'elles puisse évoluer sans impacter les autres, pérennisant ainsi au maximum le code de l'application.

Typiquement, une application recèle deux sortes de préoccupations : les préoccupations d'ordre fonctionnel et les préoccupations d'ordre technique. Une séparation claire entre ces deux types de préoccupations est souhaitable à plus d'un titre : le code métier est ainsi pérennisé par rapport aux évolutions de la technique, les équipes de développement peuvent être spécialisées (équipe pour le fonctionnel distincte de l'équipe s'occupant des préoccupations techniques), etc.

Grâce à l'inversion de contrôle, les conteneurs légers apportent une solution élégante à la gestion des dépendances et à la création des objets. Ils encouragent la séparation claire des différentes couches de l'application, aidant ainsi à la séparation des préoccupations, comme nous avons pu le voir au chapitre 2.

La modélisation illustrée à la figure 2.6, que nous reproduisons ici *(voir figure 4.1)* montre clairement que les préoccupations d'ordre technique, en l'occurrence la persistance des données centralisée dans les DAO, sont clairement isolées des préoccupations fonctionnelles, représentées par les classes métier `Todo` et `TodoList` ainsi que les classes managers.

Cependant, les conteneurs légers restent de conception orientée objet et souffrent des insuffisances de cette approche. Ainsi, la séparation des préoccupations techniques et fonctionnelles n'est pas toujours étanche. Nous allons montrer que l'approche orientée objet ne fournit pas toujours de solution satisfaisante pour aboutir à des programmes clairs et élégants. C'est notamment le cas de l'intégration des fonctionnalités transversales, problématique directement adressée par la POA.

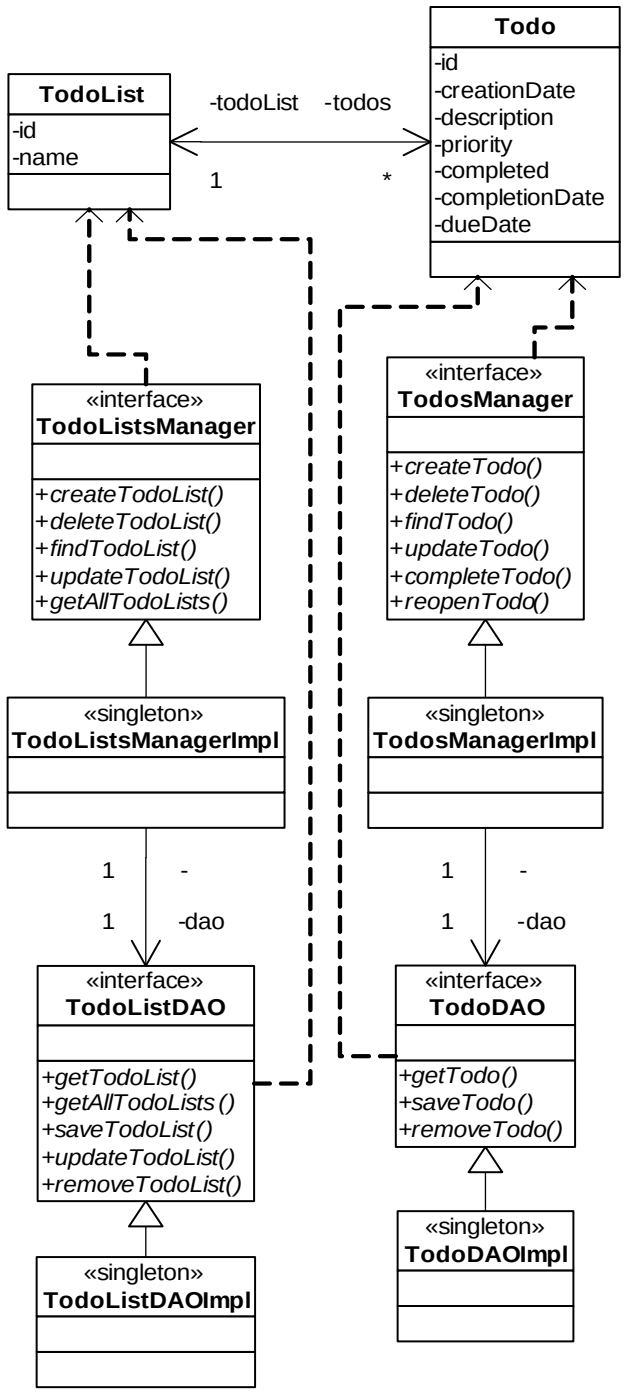
Intégration de fonctionnalités transversales

Nous qualifions de transversales les fonctionnalités devant être offertes de manière similaire par plusieurs classes d'une application. Parmi les fonctionnalités transversales que nous rencontrons souvent dans les applications, citons notamment les suivantes :

- **Sécurité.** L'objet doit s'assurer que l'utilisateur a les droits suffisants pour utiliser ses services ou manipuler certaines données.
- **Intégrité référentielle.** L'objet doit s'assurer que ses relations avec les autres sont cohérentes par rapport aux spécifications du modèle métier.
- **Gestion des transactions.** L'objet doit interagir avec le contexte transactionnel en fonction de son état (valide : la transaction continue ; invalide : la transaction est invalidée, et les effets des différentes opérations sont annulés).

Figure 4.1

Modélisation avec une inversion de contrôle sur la gestion des dépendances des objets



Avec l'approche orientée objet, ces fonctionnalités sont implémentées dans chaque classe concernée, au moins sous forme d'appels à une bibliothèque ou un framework spécialisés. Une évolution de ces fonctionnalités transversales implique la modification de plusieurs classes. Par ailleurs, nous pouvons constater que ces fonctionnalités transversales sont des préoccupations en elles-mêmes. Le fait qu'elles soient prises en charge par des classes destinées à répondre à d'autres préoccupations n'est donc pas satisfaisant.

Exemple de fonctionnalité transversale dans Tudu Lists

Pour mieux appréhender ce problème, imaginons que nous désirions faire évoluer Tudu Lists de telle sorte que l'application supporte le déclenchement de traitements en fonction d'événements techniques. L'idée est de permettre à des objets de s'inscrire auprès d'un DAO afin de réagir en fonction des appels à ses méthodes. Ainsi, il sera possible de créer des objets enregistrant ces événements à des fins statistiques, par exemple.

Cette fonctionnalité transversale concerne les interfaces DAO de notre modèle métier simplifié, à savoir `TodoDAO` et `TodoListDAO`, ainsi que leurs implémentations. L'approche par les modèles de conception apporte une solution générique éprouvée sous la forme du design pattern observateur.

Le design pattern observateur permet à un objet de signaler un changement de son état à d'autres objets, appelés observateurs. Ce design pattern est simple à implémenter avec Java puisque l'API standard de J2SE fournit une interface (`java.util.Observer`) et une classe (`java.util.Observable`) offrant la base nécessaire.

L'interface `Observer` doit être implémentée par les classes des observateurs. Cette interface ne comprend qu'une seule méthode, `update`, qui est appelée pour notifier l'observateur d'un changement au niveau du sujet d'observation.

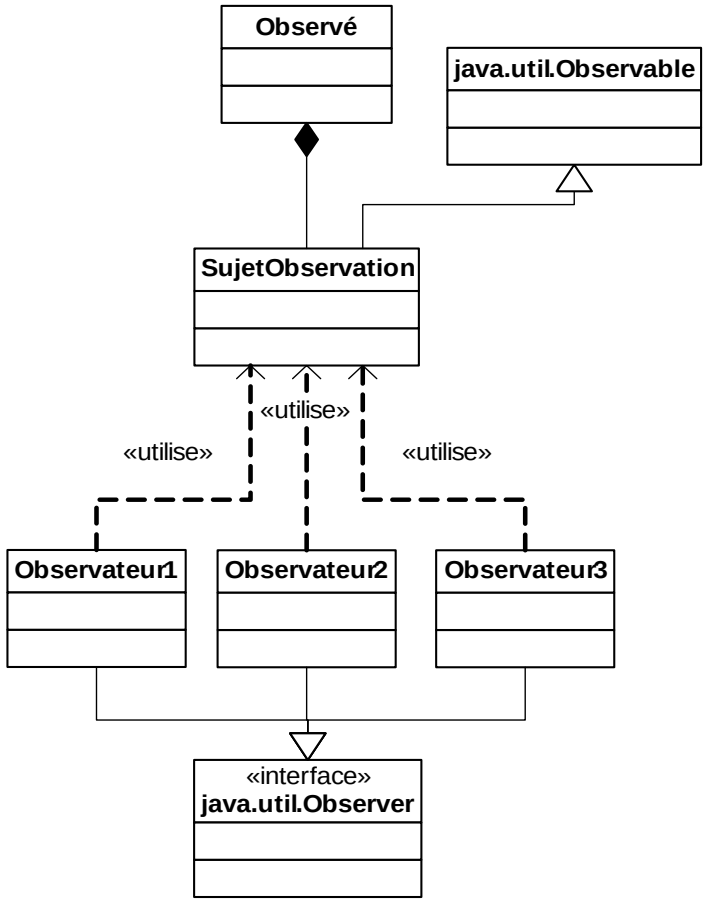
La classe `Observable` doit être utilisée par la classe observée. Cette classe fournit la mécanique d'inscription et de désinscription des observateurs (méthodes `addObserver` et `deleteObserver`) ainsi que la mécanique de notification (méthodes `notifyObservers`).

L'utilisation d'`Observable` par la classe observée peut se faire de deux manières différentes. La première revient à employer l'héritage, avec toutes les contraintes que cela impose (impossibilité d'hériter d'autres classes). La seconde consiste à créer une classe interne héritant d'`Observable`. Cette dernière nous semble la meilleure en ce qu'elle offre davantage de flexibilité dans le cas où la classe observée comporte plusieurs sujets d'observation – il suffit de créer un attribut d'un type dérivant d'`Observable` par sujet – et permet de mieux isoler la logique du design pattern du reste de la classe.

La figure 4.2 illustre sous forme de schéma UML la mise en œuvre de ce design pattern avec l'API Java.

Il est aussi possible d'utiliser la publication d'événements proposée par le contexte d'application de Spring, mais cette publication est davantage adaptée aux événements impactant l'ensemble de l'application (rafraîchissement de contexte par exemple) qu'aux événements dont la granularité est fine.

Figure 4.2
Utilisation du design pattern observateur en Java



L'implémentation générique du design pattern observateur étant définie, nous allons l'appliquer pour développer notre fonctionnalité transversale et analyser ses effets sur la qualité de conception de Tudu Lists.

Implémentation du design pattern observateur dans Tudu Lists

Les événements fonctionnels susceptibles de nous intéresser correspondent aux méthodes de nos managers. Pour simplifier l'exposé, nous ne fournissons ici que l'implémentation du design pattern observateur pour `TodoDAO` et `TodoDAOHibernte`, sachant que le principe reste le même pour `TodoListDAO` et `TodoListDAOHibernate`.

Nous pouvons définir la correspondance entre les méthodes de l'interface `TodoDAO` et les événements fonctionnels associés de la manière suivante :

- `saveTodo` : événement « enregistrement d'un todo » ;
- `removeTodo` : événement « suppression d'un todo ».

Nous avons volontairement écarté la fonction `getTodo`, car elle est utilisée en consultation et générerait un nombre très important d'événements sans qu'ils aient un intérêt fonctionnel évident, à la différence des autres événements.

Afin de profiter des avantages du conteneur léger de Spring pour la configuration des objets et la gestion des dépendances, le design pattern observateur a été implémenté de manière légèrement différente que le modèle générique présenté en début de chapitre, comme le montre le diagramme UML illustré à la figure 4.3.

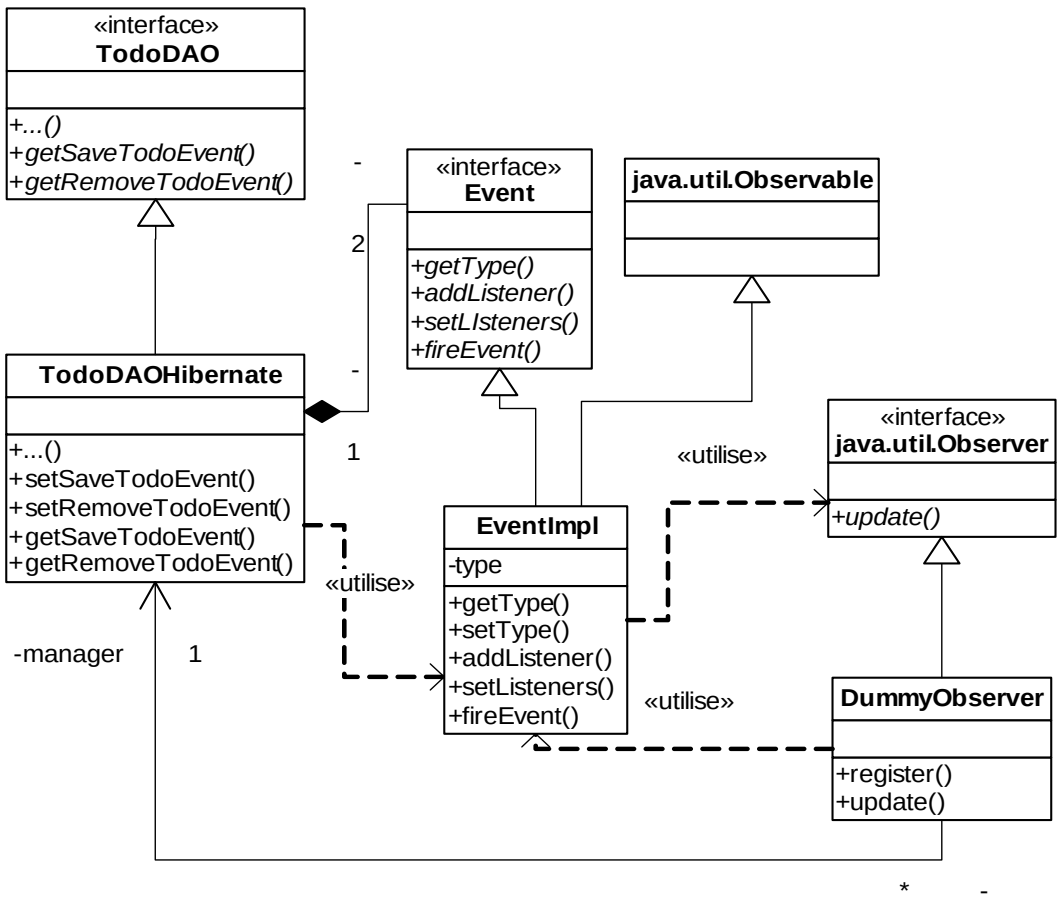


Figure 4.3
Implémentation du design pattern observateur dans Tudu Lists

Pour améliorer la lisibilité de cette figure, nous n'avons pas mentionné l'ensemble des méthodes de `TodoDAO` et `TodoDAOHibernate`. Seules sont représentées celles qui doivent être spécialement créées pour implémenter les sujets d'observation.

Pour implémenter les sujets d'observation, nous avons créé la notion d'événement, représentée par l'interface `Event`, conformément à l'esprit de Spring, qui sépare interfaces et implémentations. Un événement dispose d'un attribut `type` de type `String` pouvant prendre deux valeurs, `SaveTodoEvent` et `RemoveTodoEvent`, représentant chacune un des événements que nous avons mentionné en début de section.

Un événement dispose de trois méthodes servant d'abstraction à la classe `java.util.Observable` : `addListener` et `setListeners`, pour inscrire les observateurs du sujet d'observation, et `fireEvent`, pour avertir les observateurs que l'événement observé s'est réalisé.

Le code de l'interface `Event` est le suivant :

```
package tudu.service.events;

import java.util.Observer;

public interface Event {

   String getType();

   void fireEvent();

   void addListener(Observer obs);
}
```

L'implémentation de cette interface, la classe `EventImpl`, dérive de `java.util.Observable`, dont elle utilise les services pour gérer l'inscription des observateurs et leur avertissement en cas de changement d'état.

Le code de la classe `EventImpl` est le suivant :

```
package tudu.service.events.impl;

import java.util.Iterator;
import java.util.Observable;
import java.util.Observer;
import java.util.Set;

import tudu.service.events.Event;

public class EventImpl extends Observable implements Event {
   private String type;

   public void fireEvent() {
      super.setChanged(); ← 1
      super.notifyObservers();
   }
```

```
    public void addListener(Observer obs) {
       super.addObserver(obs);
    }

    public void setListeners(Set listeners) {
       Iterator i = listeners.iterator();
          while(i.hasNext()){
             addObserver((Observer)i.next());
          }
    }

    public String getType() {
       return type;
        }


    public void setType(String type) {
       this.type = type;
    }
}
```

L'appel à la méthode `setChanged` (repère ❶) de `java.util.Observable` est nécessaire pour que la notification des observateurs fonctionne. Cette méthode signale à la classe mère qu'un changement d'état s'est opéré dans le sujet d'observation.

Nous pouvons maintenant faire évoluer l'interface `TodoDAO` et son implémentation `TodoDAOHibernate` pour prendre en compte cette nouvelle notion d'événement. Pour cela, nous faisons évoluer l'interface `TodoDAO` en lui ajoutant les fonctions suivantes :

```
package tudu.domain.dao;

import tudu.domain.model.Todo;
import tudu.service.events.Event;

public interface TodoDAO {
   (...)
   Event getSaveTodoEvent();
   Event getRemoveTodoEvent();

}
```

Ces deux méthodes permettent de récupérer les différents sujets d'observation correspondant aux événements générés par la classe manager. Au niveau de l'implémentation fournie par `TodoDAOHibernate`, cela se matérialise par la création de cinq attributs ainsi que des accesseurs et modificateurs associés :

```
package tudu.domain.dao.hibernate;

(...)
import tudu.service.events.Event;

public class TodoDAOHibernate implements TodoDAO {
```

```
    (...)
    private Event saveTodoEvent;
    private Event removeTodoEvent;


    public Event getSaveTodoEvent() {
       return saveTodoEvent;
    }

    public void setSaveTodoEvent(Event saveTodoEvent) {
       this.saveTodoEvent = saveTodoEvent;
    }

    // Les autres attributs suivent la même logique
    (...)
}
```

Il est nécessaire de modifier le code des autres méthodes de `TodoDAOHibernate` afin qu'elles appellent la méthode `fireEvent` de l'événement qui leur correspond. Le code ci-dessous reproduit la modification (en gras dans le code) effectuée pour la méthode `saveTodo` :

```
public void saveTodo(Todo todo) {
   getHibernateTemplate().saveOrUpdate(todo);
   if (logger.isDebugEnabled()) {
      logger.debug("todoId set to: " + todo.getTodoId());
   }
   updateTodoEvent.fireEvent();
}
```

Pour `removeTodo` de la classe `TodoDAOHibernate`, la logique est exactement la même.

Pour initialiser les deux nouveaux attributs, nous utilisons l'injection de dépendances fournie par Spring. Pour cela, nous définissons dans le fichier **WEB-INF/applicationContext.xml** de Tudu Lists deux nouveaux Beans :

```
<bean id="saveTodoEvent"
 class="tudu.service.events.impl.EventImpl">
 <property name="type">
    <value>saveTodoEvent</value>
 </property>
</bean>

<bean id="removeTodoEvent"
 class="tudu.service.events.impl.EventImpl">
 <property name="type">
    <value>removeTodoEvent</value>
 </property>
</bean>
```

Ces Beans sont ensuite injectés dans notre classe `TodoDAOHibernate` telle que définie dans **WEB-INF/applicationContext.xml :**

```
<bean id="todoDAO"
 class="tudu.domain.dao.hibernate.TodoDAOHibernate">
  (...)
  <property name="saveTodoEvent">
     <ref local="saveTodoEvent"/>
   </property>
   <property name="removeTodoEvent">
      <ref local="removeTodoEvent"/>
 </property>
</bean>
```

Nous pouvons maintenant développer un observateur, par exemple une classe chargée de calculer des statistiques sur les todos, que nous appelons `Stats`. Celle-ci se présente de la manière suivante :

```
package tudu.domain.model;

import java.util.Observable;
import java.util.Observer;

import tudu.service.events.Event;

public class Stats implements Observer {

       public void update(Observable observe, Object args) {← ❸
      if (observe instanceof Event) {
         Event event = (Event) observe;
         String lEventType = event.getType();
         if ("SaveTodoEvent".equals(lEventType)) {
            System.out.println("Enregistrement d'un todo");
            (...)
         } else if ("RemoveTodoEvent".equals(lEventType)) {
            System.out.println("Suppression d'un todo");
            (...)
         }
      }
   }
}
```

Pour inscrire cet observateur auprès des sujets d'observation, nous le ferons injecter par le conteneur dans les événements *via* leur méthode `setListeners` :

```
<bean id="stats" class="tudu.domain.model.Stats"/>

<bean id="saveTodoEvent"
 class="tudu.service.events.impl.EventImpl">
  <property name="type">
     <value>saveTodoEvent</value>
  </property>
```

```
<property name="listeners">
   <set>
      <ref local="stats"/>
    </set>
 </property>
</bean>

<bean id="removeTodoEvent"
 class="tudu.service.events.impl.EventImpl">
   <property name="type">
         <value>removeTodoEvent</value>
   </property>
   <property name="listeners">
    <set>
       <ref local="stats"/>
    </set>
  </property>
</bean>
```

Lorsqu'un événement est généré (*via* l'appel à la méthode `fireEvent`), la méthode `update` est appelée (repère ❸).

Si nous exécutons maintenant l'application, nous constatons qu'en manipulant des todos, nous générons des événements affichés dans la console.

Critique de l'implémentation orientée objet

L'implémentation du design pattern observateur n'est guère compliquée, quoiqu'un peu fastidieuse. Si nous nous intéressons à la qualité de la conception, nous nous apercevons que celle-ci n'est pas totalement satisfaisante, bien qu'elle respecte l'état de l'art en terme de conception orientée objet.

Le problème conceptuel réside dans la modélisation des sujets d'observation. Idéalement, l'observation devrait être transparente pour la classe observée, c'est-à-dire qu'elle ne devrait pas avoir à gérer le fait d'être observée. L'observation du comportement d'une classe au travers de l'appel de ses méthodes est une problématique transversale (nous pourrions, par exemple, vouloir observer de la même façon l'implémentation `TodoListDAO`). Or, comme nous pouvons le constater en analysant l'implémentation orientée objet que nous venons de fournir, toute nouvelle classe observée nécessite une nouvelle implémentation des sujets d'observation, alors même qu'ils suivent le même principe générique.

Par ailleurs, le moment où sont notifiés les observateurs est défini en dur dans le code de la classe observée. Si nous désirons changer ce moment, par exemple, en notifiant avant un traitement plutôt qu'après, il est nécessaire de modifier directement la classe.

Enfin, l'ajout de nouveaux sujets d'observation à une classe observée implique une plus grande complexité de son code. Une classe simple ayant beaucoup de sujets d'observation différents, par exemple, peut voir une partie non négligeable de son code vouée aux sujets d'observation.

Si nous reprenons nos cinq critères d'analyse d'une conception, nous constatons une dégradation de la qualité de la modélisation :

- Le modèle est devenu plus rigide et fragile, car la génération d'événements est étroitement liée au code de chaque méthode observée. Il faut donc veiller qu'une modification de ce code n'entraîne pas de dysfonctionnement de la génération des événements (suppression malencontreuse de l'appel à la méthode `fireEvent`, par exemple).
- L'immobilité du modèle est augmentée puisque l'observation devient partie intégrante du DAO. Si nous désirons faire une version de Tudu Lists sans sujet d'observation, nous sommes contraints de conserver la mécanique d'observation, sauf à la supprimer directement dans le code.
- La vérification du fonctionnement du DAO est rendue plus difficile, augmentant d'autant l'invérifiabilité du modèle, dans la mesure où il est nécessaire de prendre en compte dans les tests les sujets d'observation, en fournissant des simulacres, par exemple *(voir le chapitre 17 consacré aux tests).*

Alors que la séparation des interfaces et des implémentations couplée avec l'injection de dépendances a pour objectif de rendre les classes les plus indépendantes possible, nous constatons que le design pattern observateur n'est pas correctement modularisé et génère un phénomène de dispersion du code au fur et à mesure de son utilisation pour différentes classes de Tudu Lists.

Analyse du phénomène de dispersion

Partant d'un tel constat, il est légitime de se demander si une meilleure conception et un autre découpage des classes de l'application ne permettraient pas de faire disparaître cette dispersion du code. La réponse est malheureusement le plus souvent négative.

La raison qui tend à prouver que la dispersion du code est inéluctable est liée à la différence entre service offert et service utilisé. Une classe fournit, *via* ses méthodes, un ou plusieurs services. Il est aisé de rassembler tous les services fournis dans un même endroit, c'est-à-dire dans une même classe. Cependant, rien dans l'approche objet ne permet de rassembler les utilisations de ce service. Il n'est donc pas surprenant qu'un service général et d'utilisation courante soit utilisé partout.

La dispersion d'une fonctionnalité dans une application est un frein à son développement, à sa maintenance et à son évolutivité. Lorsque plusieurs fonctionnalités sont dispersées, la situation empire. Le code ressemble alors à un plat de spaghettis, avec de multiples appels à diverses API. Il devient embrouillé (en anglais *tangled*). Ce phénomène se manifeste dans de nombreuses applications.

Il devient donc évident qu'une nouvelle dimension de modularisation doit être créée afin de capturer les fonctionnalités transversales au sein d'entités spécifiques préservant la flexibilité de la conception de l'application.

En résumé

L'approche orientée objet a apporté un niveau de modularisation important grâce à la notion d'objet et aux mécanismes associés (héritage, polymorphisme, etc.). Cependant, cette approche est grevée de limitations fondamentales, qui apparaissent dès qu'il s'agit de modulariser des préoccupations transversales d'une application. Celles-ci sont généralement dispersées au sein du code, rendant leur maintenance et leur évolution délicates.

La POA introduit de nouvelles notions offrant une dimension supplémentaire de modularisation adaptée à ces problématiques.

Notions de base de la POA

Pour répondre au besoin de modularisation des fonctionnalités transversales, la POA a introduit de nouvelles notions, qui viennent en complément de celles de l'approche objet.

Dans cette section, nous présentons ces nouvelles notions en montrant comment elles interviennent pour améliorer l'intégration de fonctionnalités transversales, comme le design pattern observateur.

Outre la notion d'aspect, équivalent de la notion de classe en POO, nous détaillons les notions de point de jonction, de coupe, de greffon, aussi appelé code advice, et d'introduction, qui sont au cœur de la notion d'aspect, au même titre que les attributs ou les méthodes le sont pour la notion de classe.

La notion d'aspect

Pour appréhender la complexité d'un programme, nous cherchons généralement à le découper en sous-programmes de taille moins importante. Les critères à appliquer pour arriver à cette séparation ont fait l'objet de nombreuses études, visant à faciliter la conception, le développement, la maintenance et l'évolutivité des programmes.

La programmation procédurale induit un découpage en fonction des traitements à implémenter, tandis que la programmation objet induit un découpage en fonction des données qui seront encapsulées dans les classes avec les traitements associés. Comme nous l'avons vu, certaines fonctionnalités s'accommodent mal de ce découpage, et les instructions correspondant à leur utilisation se retrouvent dispersées dans l'ensemble de l'application. Tout changement dans l'utilisation de ces fonctionnalités implique de devoir consulter et modifier un grand nombre de fichiers.

L'apport essentiel de la POA est de fournir un moyen de rassembler dans une nouvelle entité, l'aspect, le code d'une fonctionnalité transversale, habituellement dispersé au sein de l'application avec les approches classiques de programmation.

Aspect
Entité logicielle qui capture une fonctionnalité transversale à une application.

La définition d'un aspect est presque aussi générale que celle d'une classe. Une classe est un élément du problème à modéliser (la clientèle, les commandes, les fournisseurs, etc.), auquel nous associons des données et des traitements. De même, un aspect est une fonctionnalité à mettre en œuvre dans une application (la sécurité, la persistance, etc.), dont l'implémentation comprendra les données et les traitements relatifs à cette fonctionnalité.

En POA, une application comporte des classes et des aspects. Un aspect se différencie d'une classe par le fait qu'il implémente une fonctionnalité transversale à l'application, c'est-à-dire une fonctionnalité qui, en programmation orientée objet ou procédurale, serait dispersée dans le code de cette application.

La présence de classes et d'aspects dans une même application introduit donc deux dimensions de modularité : celle des fonctionnalités implémentées par les classes et celles des fonctionnalités transversales, implémentées par les aspects.

La figure 4.4 illustre l'effet d'un aspect sur le code d'une application. La partie gauche de la figure schématise une application composée de trois classes. Les filets horizontaux représentent les lignes de code correspondant à une fonctionnalité, par exemple, la gestion des traces. Cette fonctionnalité est transversale à l'application, car elle affecte toutes ses classes. La partie droite de la figure montre la même application après ajout d'un aspect de gestion des traces (rectangle noir). Le code de cette fonctionnalité est maintenant entièrement localisé dans l'aspect, et les classes sont vierges de toute intrusion. Une application ainsi conçue avec un aspect est plus simple à écrire, maintenir et faire évoluer qu'une application sans aspect.

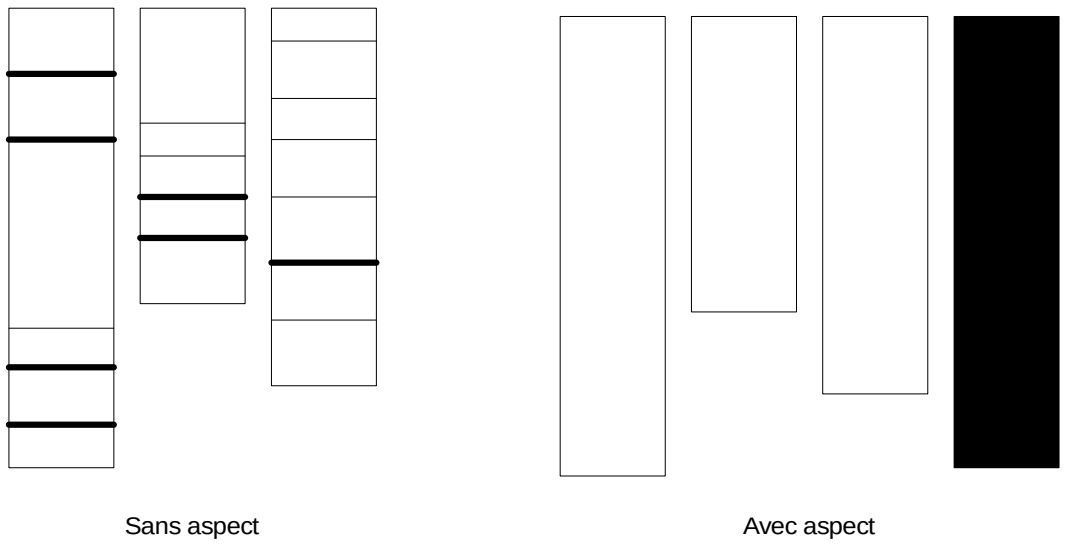


Figure 4.4

Impact d'un aspect sur la localisation d'une fonctionnalité transversale

Dans notre exemple d'implémentation du design pattern observateur, l'utilisation d'un aspect permet de laisser inchangé le code des classes observées (interface et implémentation, en l'occurrence `TodoDAO` et `TodoDAOHibernate`). Par ailleurs, les observateurs ne sont plus liés à la classe manager, mais directement à l'aspect.

Le diagramme de classes illustré à la figure 4.5 montre les effets de la modularisation de l'implémentation du design pattern observateur dans Tudu Lists.

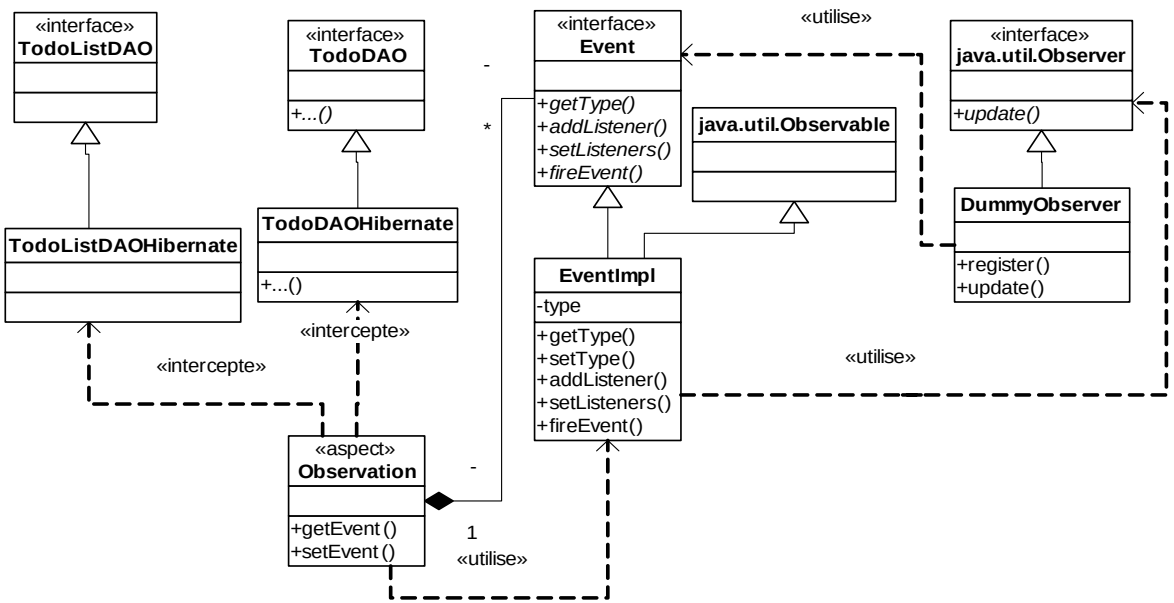


Figure 4.5

Modélisation du design pattern observateur sous forme d'aspect dans Tudu Lists

Grâce à cette nouvelle dimension de modularisation, la séparation des préoccupations est maintenue, avec un modèle métier qui conserve son expressivité originelle et sur lequel s'interfacent des extensions clairement identifiées apportées par l'aspect. Ces extensions, transparentes pour les managers – notons l'inversion de contrôle au profit de l'aspect –, permettent une réutilisation de la fonctionnalité transversale sans dispersion du code.

Nous verrons dans la suite de ce chapitre que deux éléments entrent dans l'écriture d'un aspect, la *coupe* et le *greffon*, ou *code advice*. La coupe définit le caractère transversal de l'aspect, c'est-à-dire les endroits de l'application dans lesquels la fonctionnalité transversale doit s'intégrer, tandis que le greffon fournit le code proprement dit de cette dernière.

Les points de jonction

Nous avons vu qu'un aspect était une entité logicielle implémentant une fonctionnalité transversale à une application. La définition de cette structure transversale passe par la notion de point de jonction.

> **Point de jonction *(join point)***
> Point dans l'exécution d'un programme autour duquel un ou plusieurs aspects peuvent être ajoutés.

Nous verrons à la section suivante qu'il existe différents types de points de jonction. Il peut s'agir, par exemple, de points dans l'exécution d'un programme où une méthode est appelée.

La notion de point de jonction est très générale. Elle peut être comparée à celle de point d'arrêt d'exécution, qui, lors de la mise au point d'un programme à l'aide d'un débogueur, désigne un endroit du code source où nous souhaitons voir l'exécution s'arrêter. Dans le cadre de la POA, un point de jonction désigne un endroit du programme où nous souhaitons ajouter un aspect. L'analogie s'arrête là. L'emplacement du point d'arrêt est fourni de façon interactive par le développeur *via* un numéro de ligne dans le code source. En POA, le point de jonction est fourni de manière « programmatique » par le développeur.

Si, en théorie, rien n'empêche d'utiliser les numéros de ligne du code source pour définir un point de jonction, aucun des outils de POA existants ne le permet. Le coût à payer, en terme de perte de performance à l'exécution et de complexité d'implémentation de l'outil, est jugé prohibitif. De surcroît, la moindre modification dans les numéros de ligne du code source imposerait une mise à jour coûteuse de la définition des points de jonction.

Les différents types de points de jonction

En faisant référence à l'exécution d'un programme, la notion de point de jonction révèle son caractère éminemment dynamique. Il s'agit d'événements qui surviennent une fois le programme lancé. Lorsqu'il s'agit de définir concrètement et de manière « programmatique » un point de jonction, il est nécessaire de s'appuyer sur la structure des programmes. Dans 80 % des cas, nous nous appuyons sur des méthodes.

Dans notre implémentation du design pattern observateur, les points de jonction à utiliser pour l'aspect d'observation sont les appels aux différentes méthodes de `TodosManager` auxquels correspondent nos événements.

Les méthodes ne sont pas les seuls éléments qui structurent les programmes orientés objet. Classes, interfaces, exceptions, attributs, blocs de code et instructions (`for`, `while`, `if`, `switch`, etc.) en font également partie. Avec Spring AOP, un seul point de jonction existe (exécution de méthodes) en standard. Cependant, d'autres outils de POA, comme AspectJ, JAC ou JBoss AOP, supportent d'autres points de jonction, comme les attributs, ou plus exactement leurs accès en lecture ou en écriture.

Aussi simple soit-il, un programme comporte de nombreux points de jonction potentiels. La tâche du programmeur d'aspects consiste à sélectionner les points de jonction pertinents pour son aspect. La section suivante s'intéresse à la façon dont s'opère cette sélection. Celle-ci passe par la notion de coupe.

Les coupes

Nous avons vu que les points de jonction étaient des éléments liés à l'exécution d'un programme, autour desquels nous souhaitons greffer un aspect. En ce qui concerne l'écriture du code de l'aspect, il est nécessaire de disposer d'un moyen pour désigner de manière concrète les points de jonction à prendre en compte. Ce moyen est fourni par la coupe.

> **Coupe *(crosscut)***
> Désigne un ensemble de points de jonction.

Une coupe est définie à l'intérieur d'un aspect. Dans les cas simples, une seule coupe suffit pour définir la structure transversale d'un aspect. Dans les cas plus complexes, un aspect est associé à plusieurs coupes.

Les notions de coupes et de points de jonction sont liées par leur définition. Pourtant, leur nature est très différente. Une coupe est un élément de code défini dans un aspect, alors qu'un point de jonction est un point dans l'exécution d'un programme. Si une coupe désigne un ensemble de points de jonction, un point de jonction donné peut appartenir à plusieurs coupes d'un même aspect ou d'aspects différents. Dans ce cas, comme nous le verrons dans la suite de ce chapitre, il est nécessaire de définir l'ordre d'application des différentes coupes et des différents aspects autour des points de jonction.

Avec Spring AOP, la définition des coupes repose sur l'utilisation de deux opérateurs ensemblistes, l'union et l'intersection. Dans la majorité des cas, c'est l'union qui est utilisée pour lier les différents points de jonction composant la coupe. C'est notamment le cas pour notre exemple d'implémentation du design pattern observateur, où nous avons cinq points de jonction (un par méthode observée de `TodoDAO`) qui doivent être pris en compte par notre aspect.

Les greffons

Nous avons vu qu'une coupe définissait où un aspect devait être greffé dans une application. La coupe désigne pour cela un ensemble de points de jonction. Le greffon, ou code advice, définit quant à lui ce que l'aspect greffe dans l'application, autrement dit les instructions ajoutées par l'aspect.

> **Greffon *(code advice)***
> Bloc de code définissant le comportement d'un aspect.

Un aspect comporte un ou plusieurs greffons. Chaque greffon définit un comportement particulier pour son aspect. Le greffon joue en quelque sorte le même rôle qu'une méthode. À la différence des méthodes, cependant, les greffons sont associés à une coupe, et donc à des points de jonction, et ont un type. D'un point de vue plus abstrait, il convient de noter que si une méthode définit une fonctionnalité à part entière, un greffon définit plutôt l'intégration d'une fonctionnalité *a priori* transversale.

Chaque greffon est associé à une coupe. La coupe fournit l'ensemble des points de jonction autour desquels sera greffé le bloc de code du greffon. Une même coupe peut être utilisée par plusieurs greffons. Dans ce cas, différents traitements sont à greffer autour des mêmes points de jonction. Cette situation pose le problème de la composition d'aspect, que nous n'abordons pas ici.

Pour notre exemple, le greffon se réduit à l'équivalent de l'appel de la méthode `fireEvent`, réalisé précédemment directement dans le code de `TodoDAOHibernate`. Les attributs `saveTodoEvent` et `removeTodoEvent` sont bien entendu déplacés dans l'aspect, qui est, dans le cas de Spring AOP, une classe Java.

Les différents types de greffon

Avec Spring, il existe quatre types de greffons, qui se différencient par la façon dont le bloc de code est exécuté lorsqu'un point de jonction de la coupe à laquelle ils sont associés apparaît :

- `before` : le code est exécuté avant les points de jonction, c'est-à-dire avant l'exécution des méthodes.
- `after returning` : le code est exécuté après les points de jonction, c'est-à-dire après l'exécution des méthodes.
- `after throwing` : le code est exécuté après les points de jonction si une exception a été générée.
- `around` : le code est exécuté avant et après les points de jonction.

Dans le cas des greffons `around`, il est nécessaire de délimiter la partie de code qui doit être exécutée avant le point de jonction et celle qui doit l'être après. Les outils de POA fournissent pour cela une instruction ou une méthode spéciale, nommée `proceed` (procéder, continuer). La méthode `proceed` permet de revenir à l'exécution du programme, autrement dit d'exécuter le point de jonction.

Le déroulement d'un programme avec un greffon `around` peut être résumé de la façon suivante :

1. Exécution normale du programme.
2. Juste avant un point de jonction appartenant à la coupe, exécution de la partie *avant* du greffon.

3. Appel à `proceed`. Cela déclenche l'exécution du code correspondant au point de jonction.
4. Exécution de la partie *après* du greffon.
5. Reprise de l'exécution du programme juste après le point de jonction.

L'appel à `proceed` est facultatif, un code advice `around` pouvant parfaitement ne jamais appeler `proceed`. Dans ce cas, le code correspondant au point de jonction n'est pas exécuté, et le bloc du greffon remplace le point de jonction. Après l'exécution du greffon, le programme reprend son exécution juste après le point de jonction.

Un greffon `around` peut aussi appeler `proceed` dans certaines situations et pas dans d'autres. C'est le cas, par exemple, d'un aspect de sécurité qui contrôle l'accès à une méthode. Si l'utilisateur est correctement authentifié, l'appel de la méthode est autorisé, et l'aspect invoque `proceed`. Si l'utilisateur n'a pas les bons droits, l'aspect de sécurité n'invoque pas `proceed`, ce qui a pour effet de ne pas exécuter le point de jonction et donc de ne pas appeler la méthode.

Dans certains cas, `proceed` peut être appelé plusieurs fois. Cela peut s'avérer utile pour des aspects qui ont à faire plusieurs tentatives d'exécution du point de jonction, suite, par exemple, à des pannes ou à des erreurs d'exécution.

L'instruction `proceed` ne concerne pas les greffons `before` et `after`. Par définition, un code advice `before` n'a qu'une partie *avant*. Il n'y a donc pas de partie *après* à délimiter. Un code advice `before` est automatiquement greffé avant le point de jonction. Il en va de même des codes advice `after`, qui n'ont qu'une partie *après*.

Notons que le type `around` est l'union des types `before` et `after`. Il est donc tentant de n'utiliser que le type `around`, puisqu'il est en mesure d'offrir les mêmes services que `before` et `after`. Nous déconseillons toutefois fortement cette pratique, car cela nuit à la bonne compréhension du rôle et des effets du greffon dans l'application.

Dans notre exemple, le type de greffon à utiliser est `after returning`. En effet, l'utilisation du type `around` est trop générique pour notre besoin. Quant à l'utilisation du type `before`, elle s'avère risquée puisque la méthode peut ne pas réaliser l'opération observée, par exemple en générant une exception.

Le mécanisme d'introduction

Les mécanismes de coupe et de greffon que nous avons vus jusqu'à présent permettent de définir des aspects qui étendent le comportement d'une application. Les coupes désignent des points de jonction dans l'exécution de l'application, tandis que les greffons ajoutent du code avant ou après ces points.

Dans tous les cas, il est nécessaire que les codes correspondant aux points de jonction soient exécutés pour que les greffons le soient également. Si les points de jonction n'apparaissent jamais, les greffons ne sont jamais exécutés, et l'aspect n'a aucun effet sur l'application.

Le mécanisme d'introduction permet d'étendre le comportement d'une classe en modifiant sa structure, c'est-à-dire en lui ajoutant des éléments, essentiellement des attributs ou des méthodes. Dans le cas de Spring AOP, nous ne pouvons introduire que de nouvelles interfaces avec l'implémentation associée. Ces changements structurels sont visibles des autres classes, et celles-ci peuvent les utiliser au même titre que les éléments originels de la classe. Contrairement aux greffons, qui étendent le comportement d'une application si et seulement si certains points de jonction sont exécutés, le mécanisme d'introduction est sans condition, et l'extension est réalisée dans tous les cas. Le terme introduction renvoie au fait que ces éléments sont introduits, c'est-à-dire ajoutés à la classe.

Comme le mécanisme d'héritage des langages de programmation orientée objet, le mécanisme d'introduction de la POA permet d'étendre une classe. Néanmoins, contrairement à l'héritage, l'introduction ne permet pas de redéfinir une méthode. L'introduction est donc un mécanisme purement additif, qui ajoute de nouveaux éléments à une classe.

Dans notre exemple, nous pouvons transformer une classe quelconque de Tudu Lists en observateur. En effet, grâce au mécanisme d'introduction, nous pouvons lui faire implémenter l'interface `java.util.Observer` sans modifier son code source. L'implémentation de la méthode `update` de cette interface est assurée au sein de l'aspect par un greffon d'un type particulier, appelé `mix-in`.

Le tissage d'aspect

Une application utilisant la POA est composée d'un ensemble de classes et de un ou plusieurs aspects. Une opération automatique est nécessaire pour obtenir une application opérationnelle, intégrant les fonctionnalités des classes et celles des aspects.

Cette opération est désignée sous le terme de tissage (en anglais *weaving*). Le programme qui la réalise est un tisseur d'aspects (en anglais *aspect weaver*). L'application obtenue à l'issue du tissage est dite tissée.

> **Tisseur d'aspects *(aspect weaver)***
> Programme qui réalise une opération d'intégration entre un ensemble de classes et un ensemble d'aspects.

L'opération de tissage peut être effectuée à la compilation ou à l'exécution. Dans le cas de Spring AOP, le tissage s'effectue uniquement à l'exécution, au sein du conteneur léger. Le conteneur léger contrôlant l'instanciation des classes de l'application, il est en mesure de déclencher le tissage des aspects si nécessaire. Bien entendu, lorsqu'une classe est directement instanciée dans l'application, sans passer par le conteneur léger, le tissage ne peut s'effectuer.

Le tissage effectué par Spring AOP repose sur l'utilisation de proxy dynamiques créés avec l'API standard J2SE (voir la classe `java.lang.reflect.Proxy`). Ces proxy dynamiques interceptent les appels aux méthodes de l'interface, appellent les greffons

implémentés par les aspects et redirigent, en fonction du type de greffon, les appels à la classe implémentant l'interface.

Pour illustrer ce mode de fonctionnement, supposons que nous ayons une interface appelée `UneInterface`, une classe appelée `UneImpl` implémentant cette interface et un aspect dont la coupe porte sur l'appel à la méthode `uneMethode` de `UneInterface`.

Lors de l'injection des dépendances, au lieu de renvoyer directement une instance de `UneImpl`, le conteneur léger génère un proxy dynamique de l'interface `UneInterface`. Ce proxy dynamique intercepte les appels à l'ensemble des méthodes définies dans `UneInterface`. Quand `uneMethode` est appelée, le proxy exécute le greffon associé à la coupe. Si le greffon est de type `before`, il est exécuté en premier, puis l'appel est redirigé vers `UneImpl`. Si le greffon est de type `after`, l'appel est redirigé vers `UneImpl`, puis le greffon est exécuté. Si le greffon est de type `around`, seul le greffon est appelé par le proxy, charge au premier de rediriger l'appel à `UneImpl` *via* l'instruction `proceed`.

L'utilisation de proxy dynamiques par le biais de l'API J2SE ne fonctionne qu'avec des interfaces. Pour effectuer un tissage sur une classe sans interface, Spring AOP utilise la bibliothèque open source CGLIB (Code Generation Library) pour générer le proxy.

Grâce à son mode de fonctionnement, Spring AOP peut être utilisé sans difficulté au sein d'un serveur d'applications, ce qui n'est pas toujours le cas des autres outils de POA effectuant un tissage à l'exécution. Du fait de leur fonctionnement en dehors d'un conteneur léger, ils ont besoin de contrôler le chargeur de classes pour effectuer le tissage à l'exécution. Malheureusement, cette opération n'est pas toujours possible avec les serveurs d'applications.

Utilisation de la POA

Comme tout nouveau paradigme de programmation, la POA nécessite un temps d'apprentissage non négligeable avant de pouvoir être utilisée de manière efficace. Elle ne bénéficie pas encore d'assez de retours d'expérience pour être employée de manière généralisée dans les projets. Par ailleurs, les outils de POA ne jouissent pas d'un support poussé dans les environnements de développement, à l'exception notable d'AspectJ, l'outil pionnier de la POA, ce qui ne facilite pas le travail des développeurs, notamment dans la mise au point des programmes.

Spring AOP, non content de fournir une implémentation des concepts fondamentaux de la POA, propose un certain nombre d'aspects prêts à l'emploi. Dans le package `org.springframework.aop.interceptor`, nous disposons d'aspects de débogage, de trace et de monitoring de performance. Dans le package `org.springframework.transaction.interceptor`, nous trouvons l'implémentation en POA de la gestion des transactions. Enfin, dans le package `org.springframework.orm.hibernate3`, un aspect de gestion des sessions Hibernate est proposé comme solution de rechange à l'approche par callback du template Hibernate fourni par Spring.

Pour une initiation à l'utilisation de la POA dans des projets, nous recommandons de commencer par ces aspects, qui ont fait leurs preuves. Nous aurons l'occasion de les

aborder plus en détail au fil de cet ouvrage. Pour aller plus loin et développer de nouveaux aspects, nous conseillons de commencer par des problématiques simples, sans impact direct sur la pérennité de l'application, comme l'implémentation d'un système de conception par contrat.

Il est aussi possible d'implémenter simplement certains design patterns en POA, avec à la clé de réels bénéfices. Outre le design pattern observateur, citons notamment les modèles de conception commande, chaîne de responsabilité et proxy. L'ouvrage *Programmation orientée aspect pour Java/J2EE* en fournit des implémentations pour AspectJ, mais elles sont aisément transposables avec Spring AOP.

En résumé

La POA introduit de nouvelles notions, telles que aspect, point de jonction, coupe, greffon et introduction, qui permettent de modulariser correctement les fonctionnalités transversales au sein d'entités spécifiques, les aspects. Ces notions viennent en complément de celles de l'approche orientée objet mais ne les remplacent pas.

Au final, cette nouvelle dimension de modularisation apporte un degré d'inversion de contrôle supplémentaire, puisqu'elle décharge les classes de l'implémentation de la fonctionnalité au profit d'entités extérieures, en l'occurrence les aspects.

Conclusion

Nous avons vu que l'approche orientée objet, même dotée de concepts poussés, comme ceux des conteneurs légers, n'apportait pas de solution satisfaisante à l'intégration de fonctionnalités transversales. Ces fonctionnalités rendent le code moins flexible et gênent la séparation claire des préoccupations du fait du phénomène de dispersion. La POA répond de manière élégante à ces problématiques en introduisant le concept fondamental d'aspect, complémentaire de ceux utilisés par les langages de programmation orientés objet.

Spring AOP fournit une implémentation plus limitée de la POA que des outils tels qu'AspectJ (dont il intègre cependant une partie des fonctionnalités) mais offre suffisamment de fonctionnalités dans son tisseur à l'exécution pour répondre à la plupart des besoins de modularisation sous forme d'aspects. Des fonctionnalités majeures de Spring sont d'ailleurs implémentées avec Spring AOP, comme la gestion des transactions *(voir le chapitre 12)*.

Spring AOP supporte les aspects créés avec AspectJ dans le conteneur léger afin de répondre aux besoins complexes en POA ou s'il est nécessaire d'opérer le tissage à la compilation. L'un des grands avantages d'AspectJ est son support par Eclipse, qui le dote, *via* le plug-in AJDT, d'outils facilitant le développement des aspects. Spring ne fournit pas d'intégration pour les autres outils de POA que sont JAC et JBoss AOP.

Nous décrivons en détail Spring AOP au chapitre suivant, dans lequel nous implémentons sous forme d'aspect le design pattern observateur afin d'illustrer concrètement les bénéfices apportés par la POA.

5

Spring AOP

Le chapitre précédent a rappelé les concepts de la POA et montré comment ils permettaient d'améliorer de manière significative la qualité d'une application en apportant une nouvelle dimension de modularisation. C'est à partir de ces bases conceptuelles que nous abordons ici le support de la POA offert par Spring.

Spring propose un framework 100 % Java, appelé Spring AOP, permettant d'utiliser les concepts de la POA dans nos applications. Après le succès d'AspectJ, pionnier en matière de POA, Spring intègre depuis sa version 2.0 les fonctionnalités majeures de cet outil dans son framework. Cependant, le support de la POA proposé par Spring est plus limité que celui d'AspectJ en terme de points de jonction pour définir les coupes (Spring ne supporte qu'un seul type de point de jonction). En dépit de ce support limité, Spring AOP couvre la majorité des besoins.

Spring a la capacité de s'interfacer avec les aspects développés directement avec AspectJ, mais nous n'abordons pas cette possibilité dans le cadre de cet ouvrage, l'intégration partielle d'AspectJ disponible avec Spring AOP se révélant suffisante.

Les sections qui suivent proposent un tour d'horizon complet de la POA avec Spring et détaillent la mise en œuvre de chaque concept de la POA, tel que aspect, coupe, greffon et introduction.

Implémentation de l'aspect observateur avec Spring AOP

Avant d'aborder l'implémentation de chacun des concepts de la POA avec Spring, nous allons entrer dans le vif du sujet en développant un premier aspect implémentant en POA le design pattern observateur, que nous avons décrit au chapitre précédent.

Ce premier aspect nous servira de fil directeur dans le reste du chapitre.

Nous avons vu au chapitre précédent que l'implémentation du design pattern observateur n'était pas pleinement satisfaisante en recourant exclusivement à l'approche orientée objet. Nous avons alors introduit les concepts de la POA et montré qu'ils permettaient d'implémenter ce design pattern de manière non intrusive afin d'assurer une meilleure séparation des préoccupations au sein de l'application.

Pour réaliser notre design pattern, nous conservons la majeure partie du code fourni au chapitre 4, à savoir l'interface `Event` et son implémentation `EventImpl`. Par contre, les développements réalisés dans la classe `TodoDAO` sont maintenant gérés sous forme d'aspect et doivent donc être reconsidérés en utilisant les concepts de la POA.

Un aspect comprend au minimum une coupe et un greffon. Dans notre exemple, la coupe porte sur l'exécution des méthodes de `TodoDAO`, tandis que le greffon se réduit à appeler la méthode `fireEvent` de `EventImpl`. L'inscription préalable des observateurs s'effectue dans le fichier de configuration du conteneur léger *via* la propriété `listeners` de `EventImpl` (matérialisée grâce au modificateur `setListeners`).

Pour l'observateur, nous ne créons qu'une implémentation simple, dont le code est reproduit ci-dessous :

```
package tudu.aspects.observer;

import java.util.Observable;
import java.util.Observer;

import tudu.event.Event;

public class DummyObserver implements Observer {

   public void update(Observable target, Object arg) {
      if(target instanceof Event) {
         System.out.println("Evénement détecté : "
                         +((Event)target).getType());
      }
   }
}
```

Comme indiqué en début de chapitre, Spring AOP permet de développer un aspect de deux manières : en utilisant le framework AOP de Spring ou en utilisant le support d'AspectJ proposé par Spring.

Implémentation avec Spring AOP

Spring AOP étant un framework 100 % Java, donc orienté objet, il calque les notions orientées aspect sur celles orientées objet. Ainsi, pour réaliser des greffons, Spring AOP utilise des Beans devant implémenter une interface spécifique en fonction du type de greffon concerné.

Dans notre exemple, nous devons réaliser un greffon de type `afterReturning`, c'est-à-dire qu'un événement ne sera publié que si le code de la coupe s'exécute sans générer d'exception. Pour implémenter ce type de greffon, Spring AOP dispose de l'interface

`AfterReturningAdvice`, définie dans le package `org.springframework.aop`, comme les autres types de greffons. Cette interface spécifie une méthode `afterReturning`, qui est appelée pour exécuter le code du greffon.

Nous pouvons implémenter notre greffon de la manière suivante :

```
package tudu.aspects.observer;

import java.lang.reflect.Method;
import org.springframework.aop.AfterReturningAdvice;
import tudu.service.events.Event;

public class ObserverAdvice implements AfterReturningAdvice {

  private Event event;

  public Event getEvent() {
      return event;
  }

  public void setEvent(Event event) {
      this.event = event;
  }

public void afterReturning(Object returnValue, Method method,
                  Object[] args, Object target) throws Throwable {
  event.fireEvent();
 }
}
```

Notre greffon est un Bean possédant une propriété `event`, qui est initialisée par le conteneur léger, comme nous le verrons plus loin. Cette propriété implémentant toute la logique de gestion des observateurs, le seul travail de notre greffon est d'appeler sa méthode `fireEvent`. Notons que les paramètres de la méthode `afterReturning` permettent d'accéder aux informations concernant la méthode interceptée (valeur de retour, arguments, etc.).

La configuration de ce greffon dans le conteneur léger s'effectue de la manière suivante :

```
<bean id="observerAdvice"
      class="tudu.aspects.observer.ObserverAdvice">
   <property name="event">
      <bean class="tudu.service.events.impl.EventImpl">
         <property name="listeners">
            <set>
               <bean class="tudu.aspects.observer.DummyObserver"/>
            </set>
         </property>
         <property name="type" value="execTodoDAO"/>
      </bean>
   </property>
</bean>
```

Comme nous pouvons le constater, l'événement associé au greffon est créé sous forme de Bean interne. Seul l'observateur `DummyObserver` que nous avons précédemment défini est enregistré auprès de l'événement.

Les moments auxquels notre greffon sera exécuté sont spécifiés sous forme de coupe. Dans notre exemple, ces moments sont les exécutions des différentes méthodes de l'interface `TodoDAO`, à savoir `getTodo`, `saveTodo` et `removeTodo`. Pour définir ces coupes, Spring AOP propose différentes classes à utiliser sous forme de Beans gérés par le conteneur léger. Ainsi, le framework propose la classe `NameMatchMethodPointcut`, qui permet de définir une coupe sur une liste de noms de fonctions.

L'utilisation de cette classe donne le résultat suivant dans le fichier de configuration **applicationContext-springaop.xml** du répertoire **WEB-INF :**

```
<bean id="observerPointcut"
 class="org.springframework.aop.support.NameMatchMethodPointcut">

 <property name="mappedNames" value="getTodo,saveTodo,removeTodo"/>

</bean>
```

Cette classe dispose d'une propriété `mappedNames`, qui reçoit la liste des noms de méthodes dont l'exécution doit être interceptée.

Maintenant que nous avons défini le greffon et la coupe associée, nous pouvons les intégrer au sein d'un aspect. Cette opération s'effectue en utilisant une des classes de Spring AOP prévue à cet effet et configurable dans le conteneur léger.

Pour notre exemple, nous utilisons la classe `DefaultPointcutAdvisor`, qui permet d'associer un greffon et une coupe :

```
<bean id="observerAdvisor"
 class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="advice" ref="observerAdvice"/>
    <property name="pointcut" ref="observerPointcut"/>
</bean>
```

Cette classe dispose de deux propriétés : `advice`, contenant la référence au Bean greffon, et `pointcut`, contenant la référence au Bean coupe.

Pour finir, il est nécessaire d'indiquer à Spring AOP de réaliser le tissage en instanciant un tisseur d'aspect dans le conteneur léger. Ce tissage peut être manuel ou automatique (à partir des informations fournies par les coupes).

Dans notre exemple, nous réalisons un tissage automatique en nous aidant du tisseur `DefaultAdvisorAutoProxyCreator` :

```
<bean class="
org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
```

Grâce à ce tisseur, toutes les classes possédant des méthodes ayant les noms spécifiés dans la coupe seront interceptées. Il faut toutefois prendre garde au périmètre de l'interception. Heureusement, dans Tudu Lists, les noms fournis ne sont utilisés que par `TodoDAO`.

Si nous lançons l'application Web et que nous manipulions des todos en création, modification ou suppression, nous constatons sur la console associée au serveur d'applications que les messages de notre observateur sont bel et bien affichés.

Implémentation avec le support AspectJ de Spring AOP

Depuis sa version 2.0, Spring permet d'utiliser certains éléments d'AspectJ pour créer des aspects sans nécessiter l'utilisation directe de cet outil. Cette intégration est disponible sous deux formes : une forme XML, utilisant un schéma spécifique, et une forme utilisant les annotations Java 5 définies par AspectJ. La première forme a l'avantage d'être compatible avec les anciennes versions de Java, tandis que la seconde bénéficie de la compatibilité avec AspectJ (un aspect défini sous forme d'annotations est directement compilable avec AspectJ).

Pour notre exemple, nous utilisons la première forme, la seconde étant abordée ultérieurement dans ce chapitre. Quelle que soit la forme utilisée, le support d'AspectJ utilise des Beans gérés dans le conteneur léger pour implémenter les différentes notions de la POA. Le mapping entre Beans et notions de POA s'effectue au sein du fichier de configuration.

Contrairement à Spring AOP, le support AspectJ ne nécessite pas d'interfaces spécifiques pour les Beans composant l'aspect. Notre greffon se code donc plus simplement que précédemment :

```
package tudu.aspects.observer.aspectj;

import tudu.service.events.Event;

public class ObserverAdvice {
   private Event event;

   public Event getEvent() {
      return event;
   }

   public void setEvent(Event event) {
      this.event = event;
   }

   public void greffon() {
      event.fireEvent();
   }
}
```

L'équivalent de la méthode `afterReturning` de l'exemple précédent est la méthode `greffon`. Notons que cette méthode ne possède pas de paramètre permettant d'obtenir des informations sur la méthode ayant été interceptée par la coupe, contrairement à la méthode précédente. Il s'agit d'un choix d'implémentation, et non d'un manque de fonctionnalités de la part du support d'AspectJ, comme nous le verrons plus loin.

Le reste de la définition de l'aspect s'effectue dans le fichier de configuration **WEB-INF/EC-aspectj.xml.**

Nous commençons par déclarer le schéma XML utilisé pour la POA (en gras) :

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

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

Le schéma par défaut étant déjà utilisé, nous spécifions un préfixe pour les tags de la POA. Par convention, ce préfixe est `aop`.

Comme pour la version Spring AOP pure, nous déclarons le Bean correspondant au greffon. Sa configuration est identique à la précédente, au package près :

```
<bean id="observerAdvice"
      class="tudu.aspects.observer.aspectj.ObserverAdvice">
   <property name="event">
      <bean class="tudu.service.events.impl.EventImpl">
         <property name="listeners">
            <set>
               <bean class="tudu.aspects.observer.DummyObserver"/>
            </set>
         </property>
      </bean>
   </property>
</bean>
```

Nous définissons ensuite l'aspect grâce aux balises dédiées à la POA :

```
<aop:config>
   <aop:aspect id="observerAspect" ref="observerAdvice">
      <aop:pointcut id="coupe"
          expression="execution(* tudu.domain.dao.TodoDAO.*(..))"/>
      <aop:advice
           kind="afterReturning" method="greffon"
           pointcut-ref="coupe"/>
   </aop:aspect>
</aop:config>
```

Le tag `aop:config` encapsule les définitions d'aspects. Son paramètre `ref` spécifie la classe contenant le(s) greffon(s) associé(s) à l'aspect. Un aspect est défini par le tag `aop:aspect`. La coupe est définie par le tag `aop:poincut`, dont l'attribut `expression` permet de spécifier sous forme d'expression régulière (dans un format spécifique d'AspectJ) les méthodes à intercepter. Ici, il s'agit d'intercepter toutes les exécutions des méthodes de l'interface `TodoDAO`. Nous détaillons plus loin la syntaxe à respecter pour initialiser ce paramètre.

Le paramètre `id` de la coupe permet de la nommer pour l'associer à un ou plusieurs greffons. Le tag `aop:advice` permet d'associer le greffon et la coupe. Le paramètre `kind` spécifie le type du greffon, le paramètre `method` la méthode correspondant au greffon et le paramètre `pointcut-ref` la référence à la coupe.

Pour exécuter cet exemple, nous devons au préalable renommer le fichier **applicationContext-springaop.xml** en **EC-springaop.xml** puis renommer le fichier **EC-aspectj.xml** en **applicationContext-aspectj.xml.** Ainsi, il n'y aura pas d'interférence entre les deux aspects que nous venons d'appeler.

Si nous lançons de nouveau l'application Web et que nous manipulions des todos en création, modification ou suppression, nous constatons sur la console associée au serveur d'applications que les messages de notre observateur sont bel et bien affichés.

Après ces premiers pas dans la POA avec Spring, les sections suivantes détaillent les fonctionnalités dédiées proposées par le framework. Nous commencerons par celles offertes par Spring AOP sans support d'AspectJ. En utilisant Spring AOP de cette façon, nous pouvons créer des aspects fonctionnant avec les versions 1.2 et 2.0 de Spring. Nous verrons ensuite le support d'AspectJ, une nouveauté introduite par la version 2.0 du framework.

Utilisation de Spring AOP sans AspectJ

Spring supporte la POA depuis sa version 1.0. Ce support se présente sous la forme d'un framework 100 % Java implémentant les notions de POA sous forme de Beans gérables par le conteneur léger.

Nous détaillons dans les sections qui suivent la façon dont les différentes notions introduites par la POA sont utilisables avec Spring AOP.

Définition d'un aspect

Comme expliqué précédemment, en POA, la notion d'aspect est l'équivalent de celle d'objet en POO. Nous allons voir comment cette notion est implémentée par Spring AOP.

Pour désigner la notion d'aspect, Spring AOP emploie le terme *advisor,* en référence à la notion d'advice (que nous avons francisé en greffon, plus parlant pour le lecteur). Spring AOP propose plusieurs types d'advisors, dont l'implémentation de base est `DefaultPointcutAdvisor`, que nous avons eu l'occasion d'utiliser dans notre exemple introductif.

Cette implémentation de base fixe le principe fondamental de l'advisor, qui consiste à associer une coupe et un greffon. Les types dérivant cette implémentation sont des classes permettant de paramétrer directement la coupe plutôt que de la spécifier sous forme de Bean spécifique, comme nous l'avons fait dans l'exemple introductif. Deux types d'advisor directement prêts à l'emploi sont proposés par le framework.

Les aspects de type *NameMatchMethodPointcutAdvisor*

`NameMatchMethodPointcutAdvisor` est un advisor spécialisé, qui évite de devoir définir une coupe fondée sur les noms de méthodes sous forme de Bean dédié, comme nous l'avons fait dans notre exemple.

Nous pouvons donc supprimer la définition du Bean `observerPointcut` en remplaçant l'advisor actuel par le suivant :

```
<bean id="observerAdvisor" class="org.springframework.aop.support.
NameMatchMethodPointcutAdvisor">
    <property name="advice" ref="observerAdvice"/>
    <property name="mappedNames"
              value="getTodo,saveTodo,removeTodo"/>
</bean>
```

Les aspects de type *RegexpMethodPointcutAdvisor*

`RegexpMethodPointcutAdvisor` permet d'utiliser des expressions régulières pour définir les noms des méthodes à intercepter. Il est donc beaucoup plus puissant que l'advisor précédent, pour lequel la liste complète des méthodes doit être fournie, ce qui peut devenir rapidement rébarbatif.

Cet advisor dispose de deux propriétés, `pattern` et `patterns`. La première doit être initialisée quand une seule expression régulière est nécessaire pour spécifier la coupe. La seconde est un tableau de chaînes de caractères à utiliser lorsque plusieurs expressions régulières sont nécessaires.

En reprenant notre exemple, nous utilisons cet advisor de la manière suivante :

```
<bean id="observerAdvisor" class="org.springframework.aop.support.
RegexpMethodPointcutAdvisor">
    <property name="advice" ref="observerAdvice"/>
    <property name="pattern"
              value="*Todo"/>
</bean>
```

Nous devons être très prudents en utilisant cet advisor, car nous courons le risque d'intercepter plus de méthodes que nécessaire. Typiquement, `TodoDAO` n'est pas la seule classe ayant des méthodes suffixées par `Todo`. Pour éviter ce type de problème, il est possible de restreindre la portée des advisors en paramétrant un proxy spécifique *(voir la section consacrée au tissage des aspects).*

Portée des aspects

Dans Spring AOP, la portée des aspects est limitée par rapport à des outils de POA supportant l'ensemble des concepts de ce nouveau paradigme. Spring AOP se limite à intercepter l'exécution de méthodes des Beans gérés par le conteneur léger. Il n'est donc pas possible d'intercepter une classe de l'application qui ne serait pas définie sous forme de Bean. Cette limitation n'est toutefois guère pénalisante, car l'essentiel des développements impliquant des aspects repose sur des coupes de ce type.

Par défaut, les aspects de Spring AOP sont des singletons, c'est-à-dire qu'ils sont partagés par l'ensemble des classes et des méthodes qu'ils interceptent. Il est possible de changer ce mode d'instanciation en utilisant la propriété `singleton` (à ne pas confondre avec le paramètre `singleton` du tag `bean`) des proxy spécifiques, que nous abordons plus loin.

Les coupes

Dans Spring AOP, la notion de coupe est formalisée sous la forme de l'interface `Pointcut`, définie dans le package `org.springframework.aop`. Cette interface spécifie deux méthodes, `getClassFilter` et `getMethodMatcher`, permettant de savoir quelles sont les classes et méthodes concernées par la coupe.

Cette interface dispose de plusieurs implémentations, utilisables directement pour définir les coupes de nos aspects. Les deux principales sont `NameMatchPointcut`, que nous avons utilisée dans notre exemple introductif, et `JdkRegexpMethodPointcut`, version plus évoluée de la précédente.

Spring AOP offre en outre la possibilité de réaliser des combinaisons de coupes en supportant les opérations ensemblistes union et intersection.

Les coupes de type *NameMatchPointcut*

La classe `NameMatchPointcut` permet de définir des coupes sur des noms de méthodes. Elle possède deux propriétés, `mappedName` et `mappedNames`, dont l'une des deux doit être initialisée par le conteneur léger lors de la déclaration de la coupe sous forme de Bean.

La première, de type `String`, doit être utilisée lorsqu'il n'y a qu'un seul nom de méthode à spécifier pour la coupe. La seconde, que nous utilisons dans notre exemple introductif, doit être utilisée lorsque plusieurs noms de méthodes doivent être pris en compte. Les noms spécifiés doivent être des noms exacts. Si aucun filtre de classe n'est spécifié, tous les Beans concernés par le tissage de l'aspect sont pris en compte.

Bien qu'il soit possible de spécifier un filtre pour ce type de coupe, il est préférable d'utiliser l'implémentation plus puissante proposée par `JdkRegexpMethodPointcut`, que nous détaillons à la section suivante. L'utilisation de `NameMatchPointcut` doit être réservée aux besoins simples.

Les coupes de type *JdkRegexpMethodPointcut*

La classe `JdkRegexpMethodPointcut` permet de spécifier les méthodes à intercepter sous forme d'expression régulière. Cette spécification porte sur le nom pleinement qualifié de la méthode, c'est-à-dire en mentionnant le package et la classe auxquels elle appartient. Nous pouvons donc effectuer un filtrage sur ces deux critères.

Les expressions régulières peuvent utiliser le symbole `*` pour remplacer zéro ou plusieurs caractères. Ainsi, nous pouvons spécifier l'expression régulière suivante pour notre exemple :

```
tudu.domain.dao.TodoDAO.*
```

Pour avoir une expression régulière plus concise, nous pouvons aussi écrire :

```
.*TodoDAO.*
```

Bien entendu, cette expression régulière intercepte toutes les méthodes de toutes les classes ou interfaces suffixées par `TodoDAO`. Le choix entre classe et interface dépend du mode de tissage choisi. Par défaut, Spring ne prend en compte que les interfaces, mais il est possible de ne prendre en compte que les classes, comme nous le verrons à la section consacrée au tissage.

Cette classe dispose de deux propriétés, `pattern` et `patterns`. La première, de type `String`, doit être utilisée lorsque la coupe est exprimable sous la forme d'une expression régulière unique. La seconde, qui est un tableau de `String`, doit être utilisée lorsque la coupe nécessite plusieurs expressions régulières dont l'union produit le résultat attendu.

Dans notre exemple introductif, nous pouvons remplacer l'utilisation de `NameMatchPointcut` par cette classe, afin d'avoir une définition de coupe plus générique :

```
<bean id="observerPointcut"
 class="org.springframework.aop.support.JdkRegexpMethodPointcut">

 <property name="pattern" value="tudu.domain.dao.TodoDAO.*"/>

</bean>
```

Notons que cette classe impose l'utilisation de J2SE en version 1.4 au minimum. Si nous utilisons une version antérieure, nous devons remplacer cette classe par la classe `Perl5RegexpMethodPointcut`, qui offre le même service sans dépendre de la version de Java utilisée.

Combinaison de coupes

Spring AOP offre la possibilité d'utiliser les opérateurs ensemblistes union et intersection avec les coupes afin de spécifier des coupes composites. L'opérateur union étant le plus utilisé, il bénéficie d'une classe implémentant l'interface `Pointcut`, facilement utilisable sous forme de Bean.

La classe `UnionPointcut` permet de faire l'union de deux coupes spécifiées par ailleurs sous forme de Beans. Notons que cette classe ne supporte que l'injection par constructeur. Son constructeur prend en paramètres les deux coupes à unir.

En reprenant notre exemple, nous pouvons définir deux coupes dont l'union produit la coupe utile pour notre aspect :

```
<bean id="observerPointcut1"
class="org.springframework.aop.support.NameMatchMethodPointcut">
   <property name="mappedName"
             value="getTodo "/>
</bean>

<bean id="observerPointcut2"
 class="org.springframework.aop.support.NameMatchMethodPointcut">
   <property name="mappedNames"
             value="saveTodo,removeTodo"/>
</bean>

<bean id="observerPointcut"
class="org.springframework.aop.support.UnionPointcut">
   <constructor-arg ref="observerPointcut1"/>
   <constructor-arg ref="observerPointcut2"/>
</bean>
```

Pour bénéficier de l'opérateur intersection, il est nécessaire d'utiliser la classe `ComposablePointcut`, qui propose les deux opérateurs ensemblistes. Malheureusement, cette classe n'est pas instanciable directement par le conteneur léger. Il est donc nécessaire de créer une fabrique spécifique pour ce faire. L'utilisation de l'opérateur étant marginale pour la définition de coupes, nous ne l'abordons pas dans cet ouvrage.

Les greffons

Les greffons sont spécifiés sous forme de Beans. Avec Spring AOP, ces Beans doivent implémenter des interfaces spécifiques. Afin de permettre aux greffons de réaliser des traitements en fonction de leur contexte d'exécution, ces interfaces leur permettent d'accéder à des informations détaillées sur les méthodes interceptées.

Spring AOP supporte nativement quatre types de greffons. Chacun d'eux est formalisé par une interface spécifique. Cette interface spécifie une méthode devant déclencher le traitement du greffon. Cette méthode contient plusieurs paramètres, fournissant des informations sur l'interception.

Les greffons de type *around*

Les greffons de type `around`, appelés aussi intercepteurs, doivent implémenter l'interface `MethodInterceptor` du package `org.aopalliance.intercept`. Elle fait partie de l'API générique

définie par l'AOP Alliance, dont l'objectif est d'assurer l'interopérabilité des outils de POA. Cette interface spécifie la méthode `invoke`, dont la signature est la suivante :

```
public Object invoke(MethodInvocation invocation) throws Throwable
```

Le paramètre `invocation` permet d'accéder aux informations concernant la méthode interceptée (nom, classe d'appartenance, arguments, etc.). Ce paramètre dispose d'une méthode `proceed`, à appeler pour déclencher l'exécution de la méthode interceptée.

Nous pouvons réécrire le greffon de notre exemple introductif en utilisant cette interface :

```
(...)
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;

public class ObserverAdvice implements MethodInterceptor {
   (...)

   public Object invoke(MethodInvocation invocation)
                                    throws Throwable {
      Object ret = invocation.proceed();
      event.fireEvent();
      return ret;
   }
}
```

Les greffons de type *before*

Les greffons de type `before` doivent implémenter l'interface `MethodBeforeAdvice` du package `org.springframework.aop`. Cette interface spécifie la méthode `before`, dont la signature est la suivante :

```
public void before(Method method, Object[] args, Object target)
                    throws Throwable
```

Le paramètre `method` permet d'accéder aux informations concernant la méthode interceptée. Les arguments de la méthode sont contenus dans le paramètre `args`. Le paramètre `target` contient la référence à l'objet possédant la méthode interceptée.

Nous pouvons réécrire le greffon de notre exemple introductif en utilisant cette interface :

```
(...)
import java.lang.reflect.Method;
import org.springframework.aop.MethodBeforeAdvice;

public class ObserverAdvice implements MethodBeforeAdvice {
   (...)
```

```
    public void before(Method method, Object[] args, Object target)
                                                    throws Throwable {
        event.fireEvent();
    }
}
```

En utilisant ce type de greffon, la génération s'effectue avant l'exécution de la méthode interceptée, et non plus après, comme dans l'exemple introductif.

Les greffons de type *after returning*

Les greffons de type `after returning` doivent implémenter l'interface `AfterReturningAdvice` du package `org.springframework.aop`. Cette interface spécifie la méthode `afterReturning`, dont la signature est la suivante :

```
public void afterReturning(Object returnValue, Method method,
                    Object[] args, Object target) throws Throwable
```

Le paramètre `returnValue` donne accès à la valeur de retour qui a été renvoyée par la méthode interceptée. Les autres paramètres ont la même signification que ceux des greffons de type `before`.

C'est ce type de greffon que nous utilisons dans notre exemple introductif.

Les greffons de type *after throwing*

Les greffons de type `after throwing` doivent implémenter l'interface `ThrowsAdvice` du package `org.springframework.aop`. Contrairement aux interfaces précédentes, celle-ci ne spécifie pas de méthode. Spring AOP utilise ici la réflexivité pour détecter une méthode sous la forme suivante :

```
public void afterThrowing(Method method, Object[] args,
                          Object target, Throwable exception)
```

L'utilisation de la réflexivité permet de remplacer le type du dernier paramètre `exception` par n'importe quelle classe ou interface dérivant de `Throwable`. Il s'agit généralement du type de l'exception générée par les méthodes interceptées contenues dans `exception`.

L'implémentation suivante de notre greffon déclenche l'événement uniquement si une des méthodes de `TodoDAO` génère une exception :

```
(...)
import java.lang.reflect.Method;
import org.springframework.aop.ThrowsAdvice;

public ObserverAdvice implements ThrowsAdvice {

    (...)
    public void afterThrowing(Method method, Object[] args,
                    Object target, Throwable e) {
        event.fireEvent();
    }
}
```

Utilisation de Spring AOP avec AspectJ

Depuis la version 2.0 de Spring, le support de la POA par le framework progresse de manière significative grâce au support natif d'une partie des fonctionnalités d'AspectJ.

Il s'agit essentiellement du support du langage de spécification de coupe propre à AspectJ, ainsi que des annotations Java 5, qui permettent de définir un aspect à partir d'une simple classe Java.

Fondamentalement, les capacités du framework sont les mêmes. Les progrès se situent dans le moindre effort de programmation nécessaire et dans une plus grande indépendance du code des aspects vis-à-vis de Spring AOP, puisqu'il n'y a plus d'interfaces du framework à implémenter.

Définition d'un aspect

Le support d'AspectJ permet de définir un aspect selon deux formats. Le premier est fondé sur XML, et le second sur les annotations Java 5. Les sections suivantes détaillent ces deux formats.

Le format XML

Comme indiqué dans l'exemple introductif, la définition d'un aspect sous forme XML s'effectue *via* le tag `aop:aspect`. La notion d'aspect est associée à un Bean devant contenir l'ensemble des greffons utilisés par l'aspect *via* le paramètre `ref` du tag. Chaque greffon est matérialisé sous la forme d'une méthode.

La définition de la coupe s'effectue directement dans le fichier de configuration, sans nécessiter la création d'un Bean spécifique. La définition de la coupe s'effectue *via* une expression régulière spécifique d'AspectJ, que nous décrivons à la section dédiée aux coupes.

Dans notre exemple introductif, la coupe est définie séparément de l'association avec le greffon effectuée par le tag `aop:advice`. Elle est alors référencée par le paramètre `pointcut-ref`. L'avantage de cette solution est que cette forme permet de réutiliser cette coupe avec un autre greffon.

Si cela n'est pas nécessaire, il est possible de définir la coupe directement dans le tag, avec le paramètre `pointcut` :

```
<aop:config>
   <aop:aspect id="observerAspect" ref="observerAdvice">
      <aop:advice
           kind="afterReturning" method="greffon"
           pointcut="execution(* tudu.domain.dao.TodoDAO.*(..))"/>
   </aop:aspect>
    </aop:config>
```

Utilisation des annotations Java 5

Les annotations Java 5 permettent de définir un aspect directement à partir du Bean contenant les greffons.

Ainsi, nous pouvons créer la classe `ObserverAspect`, qui implémente notre aspect en copiant la classe `ObserverAdvice` et en lui ajoutant les annotations et le code en gras :

```
package tudu.aspects.observer.aspectj;

import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;

import tudu.service.events.Event;

@Aspect
public class ObserverAspect {
   private Event event;

   public Event getEvent() {
      return event;
   }

   public void setEvent(Event event) {
      this.event = event;
   }

   @AfterReturning("execution(* tudu.domain.dao.TodoDAO.*(..))")
   public void greffon() {
      event.fireEvent();
   }
}
```

Le marquage de la classe comme étant un aspect est assuré par l'annotation `@Aspect`. La définition du greffon et de son type ainsi que de la coupe associée est effectuée par l'annotation `@AfterReturning`. Il existe une annotation par type de greffon, comme nous le verrons à la section dédiée à ces derniers.

Même si les annotations prennent en charge l'ensemble de la définition de l'aspect, il reste nécessaire d'avertir le conteneur léger de la présence d'aspects définis de la sorte.

Il faut utiliser le tag `aop:aspectj-autoproxy` pour les tisser et déclarer les aspects sous forme de Bean :

```
<aop:aspectj-autoproxy/>

<bean id="observerAspect"
      class="tudu.aspects.observer.aspectj.ObserverAspect">
```

```
    <property name="event">
       <bean class="tudu.service.events.impl.EventImpl">
          <property name="listeners">
             <set>
                <bean class="tudu.aspects.observer.DummyObserver"/>
             </set>
          </property>
       </bean>
    </property>
</bean>
```

Portée des aspects

Les aspects définis avec le support d'AspectJ sont des singletons par défaut. Il est possible de contrôler leur instanciation s'ils sont définis en utilisant les annotations. Seuls les Beans gérés par le conteneur léger peuvent être tissés avec les aspects.

L'annotation `@Aspect` accepte les paramètres `perthis` et `pertarget` à cette fin. Leur utilisation assez complexe dépassant le périmètre de notre introduction à la POA, nous ne l'abordons pas dans le cadre de cet ouvrage.

Les coupes

La spécification des coupes en utilisant le support d'AspectJ ne nécessite pas l'instanciation de Beans spécifiques, contrairement à l'approche précédente.

Les coupes sont spécifiées sous forme d'expressions régulières selon un format propre à AspectJ.

Le support d'AspectJ proposé par Spring AOP ne comprend qu'un seul type de point de jonction, `execution`, qui intercepte les exécutions de méthodes. À ce point de jonction est associée une expression régulière spécifiant les méthodes à intercepter.

AspectJ fournit un mécanisme permettant, à l'aide de symboles spécifiques, de créer des expressions englobant plusieurs méthodes. Ces symboles sont appelés des wildcards. Le mécanisme de wildcard fournit une syntaxe riche, permettant de décrire de nombreuses coupes. En contrepartie, cette richesse peut conduire à des expressions de coupe complexes et subtiles. Nous décrivons ici les usages les plus courants de ces wildcards.

Noms de méthodes et de classes

Le symbole `*` peut être utilisé pour remplacer des noms de méthodes ou de classes. Nous verrons qu'il peut l'être également pour des signatures de méthodes et des noms de packages.

En ce qui concerne les méthodes, le symbole `*` désigne tout ou partie des méthodes d'une classe ou d'une interface.

L'expression suivante désigne l'exécution de toutes les méthodes publiques de l'interface `TodoDAO` prenant en paramètre une chaîne de caractères et retournant `void` :

```
execution(public void tudu.domain.dao.TodoDAO.*(String))
```

Le symbole `*` peut être combiné avec des caractères afin de désigner, par exemple, toutes les méthodes qui contiennent la sous-chaîne `Todo` dans leur nom. Dans ce cas, l'expression s'écrit `*Todo*`.

Le symbole `*` peut aussi être utilisé pour les noms de classes ou d'interfaces. L'expression suivante désigne l'exécution de toutes les méthodes publiques de toutes les classes ou interfaces du package `tudu.domain.dao`, qui prennent en paramètre une `String` et qui retournent `void` :

```
execution(public void tudu.domain.dao.*.*(String))
```

Signatures de méthodes

Au-delà des noms, les paramètres, types de retour et attributs de visibilité (`public`, `protected`, `private`) jouent un rôle dans l'identification des méthodes. AspectJ offre la possibilité de les inclure dans les expressions de coupes.

Les paramètres de méthodes peuvent être omis grâce au symbole `..`. L'expression suivante désigne l'exécution de toutes les méthodes publiques retournant `void` de l'interface `TodoDAO`, quel que soit le profil de leurs paramètres :

```
execution(public void tudu.domain.dao.TodoDAO.*(..))
```

Le type de retour et la visibilité d'une méthode peuvent quant à eux être omis à l'aide du symbole `*`.

L'expression suivante désigne l'exécution de toutes les méthodes de l'interface `TodoDAO`, quels que soient leurs paramètres, type de retour et visibilité :

```
execution(* * tudu.domain.dao.TodoDAO.*(..))
```

De façon complémentaire, le symbole `*` pour remplacer l'attribut de visibilité peut être omis, sans que cela change la portée de la coupe.

L'expression suivante est identique à la précédente :

```
execution(* tudu.domain.dao.TodoDAO.*(..))
```

Noms de packages

Les noms de packages peuvent être remplacés par le symbole `..`

Par exemple, l'expression suivante désigne l'exécution de toutes les méthodes de toutes les classes `TodoDAO` dans n'importe quel package de la hiérarchie `tudu`, quel que soit le niveau de sous-package de cette hiérarchie :

```
execution(* tudu..TodoDAO.*(..))
```

Combinaisons de coupes

Il est possible avec AspectJ de spécifier des coupes composites en utilisant les opérateurs `and` (intersection), `or` (union) et `not` (exclusion).

Nous pouvons ainsi définir une coupe sur toutes les méthodes de l'interface `TodoDAO`, sauf la méthode `getTodo` :

```
execution(* tudu.domain.dao.TodoDAO.*(..)) and
not execution(* tudu.domain.dao.TodoDAO.getTodo(..))
```

Les greffons

Avec le support d'AspectJ, les greffons sont simples à définir, car ils ne supposent pas l'implémentation d'interfaces spécifiques. Cette définition peut s'effectuer de deux manières : en utilisant le fichier de configuration XML du conteneur léger ou en utilisant les annotations.

Rappelons que, pour utiliser les annotations, il est nécessaire de disposer d'une JVM version 5 au minimum. À l'instar des interfaces de Spring AOP, les greffons AspectJ ont la capacité d'accéder à des informations sur la méthode interceptée selon deux méthodes, que nous abordons dans cette section.

Lorsque le fichier de configuration est utilisé pour créer les aspects de l'application, la définition du type des greffons s'effectue *via* le paramètre `kind` du tag `aop:advice`. Ce paramètre peut prendre cinq valeurs différentes, correspondant aux différents types de greffons : `before`, `after`, `around`, `afterReturning` et `afterThrowing`. Pour tous ces types sauf `around`, il suffit de changer ce paramètre sans autre modification.

Pour le type `around`, il est nécessaire de modifier la signature du greffon afin qu'il reçoive en paramètre un objet de type `ProceedingJoinPoint`, qui donne accès à la méthode `proceed`. Par ailleurs, le greffon doit pouvoir transmettre les exceptions et les valeurs de retour susceptibles d'être générées par `proceed`.

Si nous transformons le greffon de notre exemple d'introduction en type `around`, nous obtenons le résultat suivant :

```
(...)
import org.aspectj.lang.ProceedingJoinPoint;

(...)
   public Object greffon(ProceedingJoinPoint thisJoinPoint)
                                          throws Throwable {
      Object ret = thisJoinPoint.proceed();
      event.fireEvent();
      return ret;
   }
```

Lorsque la définition des aspects s'effectue par annotation, nous disposons d'une annotation par type de greffon.

En plus de `@AfterReturning`, que nous avons déjà rencontré, nous disposons de `@Before`, `@After`, `@Around` et `@AfterThrowing`, qui fonctionnent tous selon le même principe.

À l'instar de la configuration XML, les greffons de type `around` doivent être modifiés pour prendre en paramètre l'objet donnant accès à la méthode `proceed` :

```
(...)
import org.aspectj.lang.ProceedingJoinPoint;

(...)
   @Around("execution(* tudu.domain.dao.TodoDAO.*(..))")
   public Object greffon(ProceedingJoinPoint thisJoinPoint)
                                        throws Throwable {
      Object ret = thisJoinPoint.proceed();
      event.fireEvent();
      return ret;
   }
```

Notons que le support d'AspectJ propose le type `after`, non spécifié sous forme d'interface par Spring AOP. Ce type est l'équivalent de `afterReturning` et de `afterThrowing` réunis. Il est appelé quel que soit le résultat de l'exécution de la méthode interceptée (valeur de retour ou exception).

Introspection et typage des coupes

Les greffons d'AspectJ ont la possibilité d'accéder aux informations concernant la méthode interceptée. Cet accès peut se faire de deux manières : en utilisant un objet de type `JoinPoint` (dont le type `ProceedingJoinPoint` utilisé par les greffons de type `around` dérive), permettant l'introspection de la méthode interceptée, ou en typant les coupes.

Pour pouvoir réaliser une introspection de la méthode interceptée, il est nécessaire de spécifier comme premier paramètre (obligatoire) un argument de type `JointPoint` (ou dérivé). Cette modification est valable en configuration XML et avec les annotations.

Si nous désirons introspecter la méthode interceptée par notre exemple d'observateur, nous pouvons modifier notre greffon de la manière suivante :

```
(...)
import org.aspectj.lang.JoinPoint;

(...)
   public void greffon(JoinPoint thisJoinPoint) {
      System.out.println("Appel de la méthode : "
           +thisJoinPoint.getSignature().getName());
      event.fireEvent();
       }
```

La méthode `getSignature` de la classe `JoinPoint` permet d'avoir des informations sur la signature de la méthode (nom, portée, etc.) sous la forme d'un objet de type `org.aspectj.lang.Signature`. Dans notre exemple, nous utilisons la méthode `getName` pour obtenir le nom de la méthode.

La classe `JoinPoint` possède d'autres méthodes, souvent utiles pour les greffons :

- `getThis` : renvoie la référence vers le proxy utilisé pour l'interception *(voir la section consacrée au tissage).*
- `getTarget` : renvoie la référence vers l'objet auquel appartient la méthode interceptée.
- `getArgs` : renvoie la liste des arguments passés en paramètres à la méthode interceptée sous forme d'un tableau d'objets.

Typage de la coupe

La récupération des informations sur la méthode interceptée peut s'effectuer de manière fortement typée (contrairement à l'introspection, que nous venons de voir). Pour cela, nous pouvons associer ces informations à des paramètres formels du greffon.

Ainsi, les résultats de `getThis`, `getTarget` et `getArgs` peuvent être directement accessibles depuis des paramètres du greffon. Pour la dernière méthode, il est possible de spécifier une liste fixe et typée d'arguments, évitant ainsi d'utiliser la réflexivité dans le code du greffon.

Les greffons de types `afterReturning` et `afterThrowing` peuvent avoir accès respectivement à la valeur de retour ou à l'exception renvoyée par la méthode interceptée.

this et *target*

La spécification des paramètres `this` et `target` s'effectue à deux niveaux : dans la coupe, en complément de l'expression régulière associée au point de jonction `execution`, et au niveau du greffon, sous la forme d'un paramètre.

Si nous voulons récupérer dans le greffon de notre exemple une référence à l'objet dont la méthode a été interceptée, nous devons modifier notre coupe de la manière suivante :

```
execution(* tudu.domain.dao.TodoDAO.*(..)) and target(todoDAO)
```

Le nom `todoDAO` donné ici doit correspondre au nom du paramètre correspondant dans le greffon.

Notre greffon peut s'écrire maintenant de la manière suivante :

```
(...)
import tudu.domain.dao.TodoDAO;

(...)
   public void greffon(TodoDAO todoDAO) {
      (...)
      event.fireEvent();
   }
```

L'utilisation de `this` est identique à celle de `target`. Le type de paramètre doit être `Object`, car il s'agit d'une référence à un proxy dynamique dont la classe ne peut être connue à l'avance.

args

Comme avec `this` et `target`, les arguments à récupérer doivent être spécifiés dans la coupe à l'aide du paramètre `args` et au niveau des arguments du greffon.

Si nous voulons récupérer l'identifiant du todo (qui est une chaîne de caractères), nous devons spécifier la coupe de la manière suivante :

```
execution(* tudu.domain.dao.TodoDAO.*(..)) and args(id)
```

Notre greffon peut s'écrire maintenant de la manière suivante :

```
(...)
   public void greffon(String id) {
      (...)
      event.fireEvent();
 }
```

Le fait de spécifier de manière formelle les arguments récupérés par le greffon exclut d'office les méthodes dont les arguments ne correspondent pas à ceux attendus. Ainsi, la méthode `saveTodo` précédemment interceptée ne l'est plus avec la nouvelle coupe, car son unique argument est de type `Todo` et non `String`.

Il est possible de spécifier une liste d'arguments dans le paramètre `args` en les séparant par des virgules. Les symboles `*` et `..` sont autorisés pour spécifier des arguments dont le type est indéfini et qui ne doivent pas être associés avec les arguments du greffon.

returning

Les greffons de type `afterReturning` doivent pouvoir accéder à la valeur de retour de la méthode interceptée. Pour ce faire, il est possible de déclarer un argument du greffon comme étant destiné à recevoir cette information.

Pour la configuration XML, il faut utiliser le paramètre `returning` dans le tag `aop:advice` en lui fournissant le nom de l'argument du greffon :

```
<aop:advice
   kind="afterReturning"
   method="greffon"
   pointcut-ref="coupe"
   returning="returnValue"
/>
```

Pour les annotations, il faut utiliser le paramètre `returning` de `@AfterReturning` :

```
@AfterReturning(
   pointcut="execution(* tudu.domain.dao.TodoDAO.*(..))"
  ,returning="returnValue")
```

Dans les deux cas, le greffon devient le suivant :

```
(...)
import tudu.domain.model.Todo;

(...)
   public void greffon(Todo returnValue) {
      (...)
      event.fireEvent();
   }
```

Avec cette nouvelle coupe, seule la méthode `getTodo` de `TodoDAO` est interceptée puisque les autres ne renvoient pas de valeur.

throwing

Les greffons de type `afterThrowing` doivent pouvoir accéder à l'exception générée par la méthode interceptée. Là encore, il est possible pour ce faire de déclarer un argument du greffon comme étant destiné à recevoir cette information.

Pour la configuration XML, il faut utiliser le paramètre `throwing` dans le tag `aop:advice` en lui fournissant le nom de l'argument du greffon :

```
<aop:advice
   kind="afterReturning"
   method="greffon"
   pointcut-ref="coupe"
   throwing="exception"
/>
```

Pour les annotations, il faut utiliser le paramètre `returning` de `@AfterReturning` :

```
@AfterThrowing(
   pointcut="execution(* tudu.domain.dao.TodoDAO.*(..))"
  ,throwing="exception")
```

Dans les deux cas, le greffon devient le suivant :

```
(...)
import tudu.domain.model.Todo;

(...)
   public void greffon(Throwable exception) {
      (...)
      event.fireEvent();
   }
```

L'exception en paramètre d'un greffon de type `afterThrowing` peut être d'un type dérivant de `Throwable`. Nous utilisons ici `Throwable`, car nous ne pouvons préjuger des exceptions générées par la méthode interceptée (elle ne spécifie pas d'exception *via* la clause `throws`, par exemple).

Le mécanisme d'introduction

Spring AOP dispose d'un mécanisme d'introduction utilisant un type de greffon (`DelegatingIntroductionInterceptor`) et un type d'advisor (`DefaultIntroductionAdvisor`) spécifiques. Avec le support d'AspectJ apporté par la version 2.0 de Spring, nous avons la possibilité de nous reposer sur cette technologie pour réaliser des introductions. La solution d'AspectJ étant plus élégante que la solution native de Spring AOP, nous ne présentons ici que la première.

Le mécanisme d'introduction d'AspectJ est uniquement utilisable sous forme d'annotations à l'heure où nous écrivons ces lignes. Il repose sur l'annotation `@DeclareParents`, qui prend deux paramètres : `value`, qui désigne la classe cible à étendre, et `defaultImpl`, qui spécifie la classe fournissant l'implémentation de l'introduction. Cette classe doit bien entendu implémenter l'interface qui est introduite dans la classe cible. La spécification de cette interface est fournie par l'attribut statique auquel s'applique l'annotation `@DeclareParents`.

Si nous désirons étendre les fonctionnalités de la classe `TodoListDAOHibernate` du package `tudu.domain.dao.hibernate3` afin qu'elle implémente l'interface `java.util.Observer`, nous pouvons spécifier l'aspect suivant :

```
package tudu.aspects.observer.aspectj;

import java.util.Observer;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

import tudu.aspects.observer.DummyObserver;

@Aspect
public class ObserverIntroduction {

@DeclareParents(
        value="tudu.domain.dao.hibernate3.TodoListDAOHibernate"
        ,defaultImpl=DummyObserver.class)
 public static Observer mixin;

}
```

Nous spécifions comme implémentation de l'introduction la classe `DummyObserver` de notre exemple introductif, qui implémente l'interface `java.util.Observer`.

Pour que l'introduction soit effective, il est nécessaire de déclarer `ObserverIntroduction` sous forme de Bean et d'utiliser le tag `aop:aspectj-autoproxy` pour que le tissage s'effectue :

```
<aop:aspectj-autoproxy/>

<bean id="observerIntroduction"
      class="tudu.aspects.observer.aspectj.ObserverIntroduction"/>
```

Nous pouvons maintenant modifier le greffon `observerAdvice` afin qu'il utilise le Bean `todoListDAO` comme observateur, puisque celui-ci implémente l'interface `java.util.Observer` grâce à l'introduction :

```
<bean id="observerAdvice" class="tudu.aspects.observer.aspectj.ObserverAdvice">
   <property name="event">
      <bean class="tudu.service.events.impl.EventImpl">
         <property name="listeners">
            <set>
               <ref bean="todoListDAO"/>
            </set>
         </property>
      </bean>
   </property>
</bean>

<aop:config>
   <aop:aspect id="observerAspect" ref="observerAdvice">
      <aop:advice
            kind="afterReturning" method="greffon"
            pointcut="execution(* tudu.domain.dao.TodoDAO.*(..))"
      />
   </aop:aspect>
</aop:config>
```

Si nous exécutons l'application Web avec ces modifications, nous constatons que le Bean `todoListDAO` se comporte effectivement en observateur en lieu et place du Bean interne de type `DummyObserver`.

Le tissage des aspects

Le tissage des aspects avec Spring AOP et son support d'AspectJ s'effectue grâce à la création dynamique de proxy pour les Beans dont les méthodes doivent être interceptées. Les utilisateurs de ces méthodes obtiennent ainsi une référence à ces proxy (dont les méthodes reproduisent rigoureusement celles des Beans qu'ils encapsulent), et non une référence directe aux Beans tissés.

La génération de ces proxy est contrôlée au niveau du fichier de configuration du conteneur léger. Cette génération peut être automatique pour les besoins courants ou spécifique pour les besoins plus complexes.

Avec AspectJ, le tissage est uniquement automatique. Le seul contrôle dont dispose le développeur est de tisser ou non les aspects à base d'annotations *via* le tag `aop:aspectj-autoproxy`.

Les proxy automatiques

Dans notre exemple introductif, nous avons utilisé le mécanisme de proxy automatique afin de simplifier notre configuration. Spring AOP propose des modes de tissage automatique matérialisés par des Beans spécifiques à instancier.

Le mode le plus simple est celui utilisé dans notre exemple. Il utilise la classe `DefaultAdvisorAutoProxyCreator`, qui doit être instanciée sous forme de Bean dans le conteneur léger :

```
<bean class="
org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator"/>
```

Ce mode se fonde sur les coupes gérées par l'ensemble des advisors de l'application pour identifier les proxy à générer automatiquement. Dans notre exemple, les Beans ayant une ou plusieurs méthodes spécifiées dans la coupe `observerPointcut` sont automatiquement encapsulés dans un proxy.

Le second mode permet de spécifier la liste des Beans à encapsuler dans des proxy. Ce mode utilise la classe `BeanNameAutoProxyCreator`, qui doit être instanciée sous forme de Bean dans le conteneur léger :

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames" value="todoDAO"/>
    <property name="interceptorNames">
        <list>
            <value>observerAdvisor</value>
        </list>
    </property>
</bean>
```

Ce tisseur prend en paramètre la liste des Beans à tisser sous forme d'un tableau de chaînes de caractères *via* la propriété `beanNames` (l'utilisation du symbole `*` est autorisée) et la liste des advisors à y appliquer *via* la propriété `interceptorNames`. Cette propriété accepte également les greffons dans sa liste. Dans ce cas, les greffons interceptent l'ensemble des méthodes des Beans concernés par le tissage.

Dans notre exemple, comme nous interceptons l'ensemble des méthodes du Bean `todoDAO`, nous pouvons utiliser directement le greffon `observerAdvice`, sans passer par l'advisor :

```
<bean class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames" value="todoDAO"/>
    <property name="interceptorNames">
        <list>
            <value>observerAdvice</value>
        </list>
    </property>
</bean>
```

Les proxy spécifiques

Les proxy spécifiques offrent un contrôle plus étroit sur leur mode de fonctionnement. Ils permettent de définir s'il faut créer un proxy par rapport à l'interface du Bean ou par rapport à sa classe, ainsi que le mode d'instanciation de l'aspect (singleton ou par instance de Bean).

Les proxy spécifiques sont définis sous forme de Beans avec la classe `ProxyFactoryBean`, qui, comme son nom l'indique, est une fabrique de proxy. Ce sont ces Beans qu'il faut utiliser en lieu et place des Beans qu'ils encapsulent. Nous perdons ainsi la transparence des proxy automatiques du point de vue de la configuration.

Si nous reprenons notre exemple introductif, la première étape consiste à renommer le Bean `todoDAO` en `todoDAOTarget` (convention de nommage pour désigner les Beans devant être tissés spécifiquement). Ensuite, nous définissons un nouveau Bean `todoDAO` de la classe `ProxyFactoryBean` :

```
<bean id="todoDAO" class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="todoDAOTarget"/>
    <property name="interceptorNames">
        <list>
            <value>observerAdvisor</value>
        </list>
    </property>
</bean>
```

La propriété `target` spécifie le Bean devant être encapsulé dans le proxy (ici, `todoDAOTarget`). La propriété `interceptorNames` spécifie la liste des advisors ou des greffons à tisser.

`ProxyFactoryBean` propose trois propriétés optionnelles :

- `proxyInterfaces` : tableau de chaînes de caractères spécifiant la liste des interfaces du Bean à tisser. Lorsque cette propriété n'est pas spécifiée, la détection s'effectue automatiquement.
- `singleton` : booléen spécifiant le mode d'instanciation de l'aspect. S'il vaut `false`, l'aspect est instancié pour chaque instance du Bean tissé. Par défaut, cette propriété vaut `true`.
- `proxyTargetClass` : booléen spécifiant si le proxy doit être généré sur la classe au lieu des interfaces qu'elle implémente. Par défaut, cette propriété vaut `false`.

Modifications de cibles

Comme indiqué à la section précédente consacrée au tissage, un proxy encapsule un Bean afin d'intercepter les appels à ses méthodes. Spring AOP propose un niveau d'indirection supplémentaire grâce à la notion de source de cible, modélisée par l'interface `TargetSource` du package `org.springframework.aop`.

Cette notion est proche de celle de fabrique de Bean spécifique. L'idée est que l'entité de résolution de la cible fournit au proxy la référence au Bean. Elle est donc en mesure de contrôler cette dernière. Cette capacité permet de remplacer des Beans à chaud ou de gérer des pools d'instances de Beans.

La fabrique `ProxyFactoryBean` utilise une implémentation par défaut, qui renvoie la référence du Bean spécifiée par sa propriété `target`. Il est possible d'utiliser une des deux

implémentations que nous allons voir grâce à sa propriété `targetSource`. L'utilisation des sources de cibles est compatible avec le tissage d'aspect au sein de la fabrique.

Remplacement à chaud des cibles

Le remplacement à chaud des cibles permet de changer d'instance pour un Bean pendant l'exécution de l'application. Ce remplacement à chaud est assuré par la classe `HotSwappableTargetSource` du package `org.springframework.aop.target`.

Pour bénéficier de cette fonctionnalité pour le Bean `todoDAO`, il suffit de le renommer en `todoDAOTarget` et d'instancier l'entité de résolution de la cible ainsi que la fabrique `ProxyFactoryBean` de la manière suivante :

```
<bean id="swapper" class="org.springframework.aop.target.HotSwappableTargetSource">
   <constructor-arg ref="todoDAOTarget"/>
</bean>

<bean id="todoDAO" class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="targetSource" ref="swapper"/>
</bean>
```

Le Bean `todoDAO` peut être transtypé en `HotSwappableTargetSource`. Ce transtypage permet d'utiliser la méthode `swap` prenant en unique paramètre la nouvelle implémentation de l'interface `TodoDAO` à utiliser.

Pooling des cibles

Spring AOP permet de créer un pool d'instances d'un Bean, similaire à celui proposé par les serveurs d'applications avec les EJB Session sans état. Pour cela, comme pour le remplacement à chaud des cibles, il faut utiliser une implémentation spécifique de `TargetSource`, ici `CommonsPoolTargetSource`, fondée sur la bibliothèque Commons Pool de la communauté Apache Jakarta.

Pour pouvoir bénéficier de cette fonctionnalité pour le Bean `todoDAO`, il suffit de le transformer en prototype (`singleton="false"`), de le renommer en `todoDAOTarget` et d'instancier l'entité de résolution de la cible ainsi que la fabrique `ProxyFactoryBean` de la manière suivante :

```
<bean id="pool" class="org.springframework.aop.target.CommonsPoolTargetSource">
   <property name="targetBeanName" value="todoDAOTarget"/>
   <property name="maxSize" value="4"/>
</bean>

<bean id="todoDAO" class="org.springframework.aop.framework.ProxyFactoryBean">
   <property name="targetSource" ref="pool"/>
</bean>
```

Dans cet exemple, nous créons un pool d'instances du Bean `todoDAOTarget` ayant au maximum quatre instances disponibles.

Conclusion

Nous avons vu dans ce chapitre comment utiliser Spring AOP et son support d'AspectJ pour faire de la POA. Nous avons implémenté à cet effet le design pattern observateur afin de démontrer concrètement la puissance de ce nouveau paradigme.

Grâce à la richesse de Spring AOP, le développeur dispose d'une large palette d'outils pour définir les aspects qui lui sont nécessaires. Cependant, Spring AOP n'offre pas toute la richesse fonctionnelle des outils de POA spécialisés. Il ne peut, par exemple, intercepter que les exécutions de méthodes des Beans gérés par le conteneur léger.

La couverture fonctionnelle de ce framework reste cependant suffisante pour couvrir la plupart des besoins. La gestion des transactions, que nous traitons en détail au chapitre 12, repose d'ailleurs sur Spring AOP, preuve concrète que ce framework est en mesure de supporter des aspects complexes.

Ce chapitre clôt la première partie de cet ouvrage, qui a fourni les bases nécessaires à l'utilisation de Spring. Dans les parties suivantes, nous abordons l'ensemble des fonctionnalités à valeur ajoutée proposées par ce framework et reposant sur les fondations conceptuelles que nous avons introduites.

Partie II

Intégration des frameworks de présentation

Cette partie se penche sur les principes que Spring met en œuvre afin de résoudre les préoccupations liées aux applications Web. Spring couvre un large spectre de technologies Web, allant des servlets aux portlets. Le framework s'efforce de garder une homogénéité de ses différents supports Web en utilisant des mécanismes similaires.

Spring n'imposant pas l'utilisation de son framework MVC, nous verrons au chapitre 6 comment l'utilisation de Spring en conjonction avec Struts, un des frameworks MVC les plus utilisés actuellement, permet à ce dernier de s'interfacer de manière optimale avec la couche métier des applications Java/J2EE.

Le chapitre 7 traite de Spring MVC, le framework MVC de Spring. Ce dernier vise à fournir un cadre souple et robuste afin d'implémenter des applications Web fondées sur la technologie servlet tout en tirant parti au maximum des fonctionnalités du conteneur léger de Spring.

Le chapitre 8 est consacré à Spring Web Flow, un framework de gestion des flots Web, qui offre des mécanismes robustes afin de configurer et contrôler les enchaînements de pages dans les applications Web.

Le chapitre 9 aborde la technologie émergente AJAX, qui permet de mettre en œuvre des applications Web riches possédant une interface graphique plus élaborée que celles des applications Web classiques.

Le chapitre 10 clôt cette partie en abordant le support portlet, qui permet de développer des applications de portail tout en réutilisant les principes de Spring MVC et les fonctionnalités de Spring.

6

Intégration de Struts

Struts est certainement le framework de présentation le plus réputé dans le monde J2EE. Cette notoriété le fait souvent demander dans les projets afin de respecter l'état de l'art.

En termes techniques, Struts est un framework MVC relativement élémentaire. Il gère des formulaires HTML et les actions des utilisateurs, mais ne va pas beaucoup plus loin. Cette simplicité fait aussi sa force : facile à prendre en main, il permet d'effectuer efficacement les tâches pour lesquelles il est conçu.

L'intérêt principal de Struts réside aussi dans le très grand nombre de développeurs qui connaissent son fonctionnement. Utiliser Struts sur un projet, c'est donc s'assurer de ne pas avoir de problèmes de ressources humaines.

Ce framework est cependant vieillissant et se trouve aujourd'hui sous le feu des critiques. Nous verrons dans le cours de ce chapitre quels sont ses défauts et dans quelles situations Struts n'est plus la meilleure solution disponible.

Une fois l'architecture de Struts et ses principales classes explicitées, nous verrons de quelle manière Spring intègre Struts dans une application Java/J2EE, et à quelles fins. Cette intégration pouvant être réalisée de trois manières, nous étudierons chacune d'elles et dégagerons leurs forces et faiblesses respectives.

En fin de chapitre, nous détaillerons au travers de l'étude de cas Tudu Lists la configuration de Struts et de Spring, ainsi que le développement d'actions et de formulaires spécifiques.

Fonctionnement de Struts

Cette section se penche sur l'architecture interne de Struts et présente sa configuration et ses classes principales.

Struts représentant une implémentation classique du framework MVC, il est important de commencer par rappeler brièvement les caractéristiques de ce modèle.

Le pattern MVC (Model View Controller)

Le pattern MVC est communément utilisé dans les applications Java/J2EE 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 J2EE, il s'appuie généralement sur l'API servlet et des technologies telles que JSP/JSTL.

Il existe deux types de patterns MVC, le pattern MVC dit de type 1, qui possède un contrôleur par action, et le pattern MVC dit de type 2, plus récent, qui possède un contrôleur unique. Nous nous concentrerons sur ce dernier, puisque c'est celui sur lequel s'appuie Struts.

La figure 6.1 illustre les différentes entités du type 2 du pattern MVC ainsi que leurs interactions lors du traitement d'une requête.

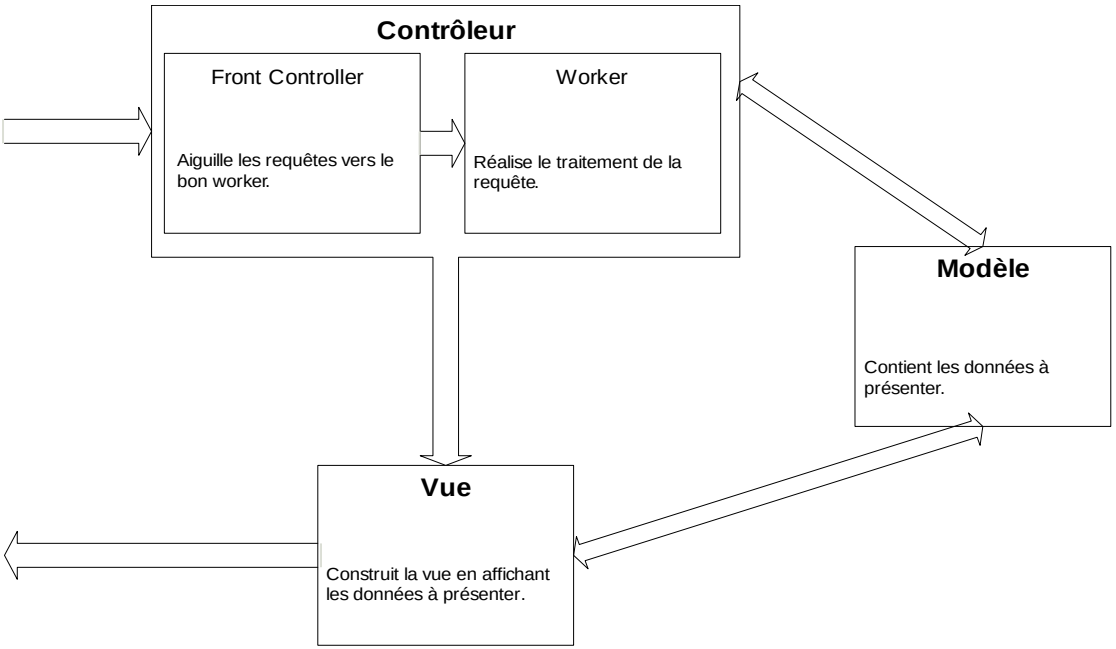


Figure 6.1

Composants du pattern MVC type 2

Les caractéristiques des composants de ce pattern 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 et XML, 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 pattern, 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 Worker appropriée. Pour cette raison, l'entité Worker est habituellement appelée contrôleur. Le Front Controller, ou « contrôleur façade », est intégré au framework MVC, et seuls les workers 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 workers.

Architecture et concepts de Struts

Comme expliqué précédemment, Struts est une implémentation classique du framework MVC de type 2, dont les caractéristiques sont les suivantes :

- **Modèle.** Constitué de JavaBeans standards. Dans notre cas, ils proviennent des couches inférieures de l'application (couche de service ou couche de domaine).
- **Vue.** Classiquement constituée de pages JSP. Cette partie peut aussi être prise en charge par d'autres technologies de présentation, telles que Velocity *(http://jakarta.apache.org/velocity/)* ou Cocoon *(http://cocoon.apache.org/),* tous deux de la fondaction Apache Jakarta.
- **Contrôleur.** Constitué de la servlet Struts principale, appelée `ActionServlet`.

Dans son implémentation, Struts utilise deux notions principales, qui sont représentées par les classes `org.apache.struts.action.Action`, appelées actions, et `org.apache.struts.action.ActionForm`, appelées `FormBeans`, ou formulaires :

- **Action.** Représente une action d'un utilisateur, telle que valider une sélection, mettre un produit dans un panier électronique, payer en ligne, etc. Cette classe proche du contrôleur sert de liant entre le formulaire envoyé par l'utilisateur, l'exécution de traitements dans les couches métier de l'application et l'affichage d'une page Web résultant de l'opération. Elle représente le worker introduit à la section précédente.

- **Formulaire.** Représente un formulaire HTML, tel qu'il a été rempli par l'utilisateur. Typiquement, ce formulaire a été codé en HTML avec des balises classiques de type `<input type='...'/>`. Struts reçoit et interprète en réalité les paramètres passés en GET ou en POST d'une requête HTTP. Il est donc possible de forger des URL, en JavaScript, par exemple, que Struts comprendra. Voici un exemple de paire clé/valeur passée en paramètre d'une Action Struts : *http://localhost:8080/example/test.do?clef=valeur.*

L'ensemble formé par les actions, formulaires et JSP est relié *via* le fichier de configuration de Struts, **struts-config.xml,** lequel est généralement stocké dans le répertoire **WEB-INF** de l'application Web. Dans le cas de Tudu Lists, ce fichier se trouve à l'emplacement **WEB-INF/struts-config.xml.**

Configuration de Struts

Struts se fonde sur une servlet très évoluée pour assurer sa fonction de contrôleur générique. Comme pour toute servlet, il faut la configurer dans le fichier **web.xml :**

```
( ... )
<servlet>
  <servlet-name>action</servlet-name>
  <servlet-class>
    org.apache.struts.action.ActionServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
( ... )
<servlet-mapping>
  <servlet-name>action</servlet-name>
  <url-pattern>*.do</url-pattern>
</servlet-mapping>
( ... )
```

L'usage veut que l'on fasse correspondre les actions Struts à l'URL ***.do,** comme dans l'exemple ci-dessus. Cette extension en **.do** provient du verbe anglais *to do* (faire). Les actions Struts ont donc toutes une URL se terminant en **.do,** par exemple, *http://localhost/example/test.do,* de façon à être renvoyées à `ActionServlet`, qui les traite. Comme il s'agit d'une convention de nommage arbitraire, pour Tudu Lists nous avons préféré suffixer les actions par ***.action,** que nous trouvons plus parlant.

`ActionServlet`, quant à elle, est configurée *via* le fichier **struts-config.xml,** que nous avons présenté précédemment. Voici un exemple de ce fichier :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE struts-config PUBLIC "-//Apache Software Foundation//DTD Struts
➥Configuration 1.1//EN" "http://jakarta.apache.org/struts/dtds/struts-
➥config_1_1.dtd">
<struts-config>
  <form-beans>
    <form-bean
```

```
      name="exampleForm"
      type="tudu.web.form.ExampleForm"/>
  </form-beans>
  <global-exceptions>
    <exception  handler="tudu.web.TuduExceptionHandler"
                key="error.general"
                 path="/WEB-INF/jsp/error.jsp"
                 type="java.lang.Throwable"/>
  </global-exceptions>
  <global-forwards>
    <forward name="error" path="/WEB-INF/jsp/error.jsp"/>
  </global-forwards>
  <action-mappings>
    <action path="/test/example"
            name="exampleForm"
            type="tudu.web.action.ExampleAction"
            validate="true">

      <forward name="success"
               path="/WEB-INF/jsp/example.jsp"/>
    </action>
  </action-mappings>
  <message-resources parameter="messages"/>
</struts-config>
```

Ce fichier XML comporte les grandes parties suivantes :

- `<form-beans>`. Contient la liste des formulaires Struts. Nous donnons un nom à chaque formulaire, comme `exampleForm`, ainsi que le nom de la classe Java à laquelle il correspond. Il existe un type particulier de formulaire, dit `DynaBean` ou `DynaForm`, entièrement décrit en XML. Tudu Lists est majoritairement constitué de `DynaForms`, mais nous conseillons dans un premier temps l'utilisation de JavaBeans standards codés en Java. Presque aussi rapides à écrire, ils sont moins sujets à erreur au début.
- `<global-exceptions>`. En fonction de leur type, les exceptions lancées par les actions sont renvoyées par Struts à des classes spécifiques, qui doivent être spécifiquement implémentées. Cette partie est facultative.
- `<global-forwards>`. Donne des noms génériques à des pages JSP. Ces noms peuvent ensuite être utilisés depuis n'importe quelle action Struts. Dans notre exemple, nous pouvons renvoyer la page nommée `erreur` depuis toutes les actions, ce qui peut se révéler très utile. Cette partie est également facultative.
- `<action-mappings>`. Cette partie est la plus importante et la plus complexe. Elle permet de lier une URL à une action. Dans notre exemple, nous utilisons l'URL */test/example.action,* en supposant que le suffixe Struts tel que décrit plus haut est **.action.** Toute requête envoyée à cette URL est traitée par la classe `tudu.web.action.ExampleAction`, si elle est validée par le formulaire `exampleForm`.
- `<message-resources>`. Définit un fichier de propriétés, dans notre cas `messages.properties`. Situé à la racine du répertoire **JavaSource** de notre projet Eclipse, ce fichier

contient les messages utilisés dans les pages JSP et les actions. Ce fichier est recherché à la racine du classpath (typiquement dans **WEB-INF/classes**).

Ce fichier de configuration relativement complexe peut être édité graphiquement. Struts étant un standard du marché, de nombreux IDE supportent son utilisation. C'est le cas notamment de JDeveloper d'Oracle, un IDE gratuit qui propose une configuration graphique assez intuitive.

Si vous utilisez Eclipse, ou si vous n'utilisez pas d'IDE, nous vous conseillons l'utilisation de Struts Console, un utilitaire gratuit très bien conçu disponible à l'adresse *http://www.jamesholmes.com/struts/console/index.html*.

Actions et formulaires

Les actions et les formulaires sont les classes de base utilisées dans Struts.

Les actions Struts héritent toutes de `org.apache.struts.action.Action`. Cette classe représente une action standard, mais Struts est fourni avec un certain nombre d'actions plus évoluées, par exemple `org.apache.struts.actions.DispatchAction`, lesquelles restent cependant proches dans leur fonctionnement.

Voici un exemple d'action simple, provenant de Tudu Lists :

```
package tudu.web;

( ... )

/**
 * The Log out action.
 */
public class LogoutAction extends Action {

    private final Log log = LogFactory.getLog(LogoutAction.class);

    public final ActionForward execute(ActionMapping mapping,
        ActionForm form,
            HttpServletRequest request,
            HttpServletResponse response) {

        log.debug("Execute action");
        request.getSession().invalidate();
        Cookie terminate = new Cookie(TokenBasedRememberMeServices.ACEGI_SECURITY-
        ➥_HASHED_REMEMBER_ME_COOKIE_KEY, null);
        terminate.setMaxAge(0);
        response.addCookie(terminate);
        return mapping.findForward("logout");
    }
}
```

Comme toutes les actions Struts, `LogoutAction` implémente la méthode `execute()`, laquelle prend quatre arguments :

- `mapping` : correspond à la description faite précédemment dans le fichier **struts-config.xml** et permet de retrouver les vues telles quelles y ont été décrites dans les balises `<forward>`. Si nous reprenons la configuration décrite précédemment, nous pouvons rediriger l'utilisateur vers la page d'erreur en faisant un `mapping.findForward("error")`.
- `form` : correspond au formulaire Struts, tel qu'il a été décrit dans le fichier de configuration. Dans cet exemple très simple, il n'y a pas de formulaire associé, ce qui est une configuration correcte.
- `request/response` : correspondent aux objets classiques `HttpServletRequest` et `HttpServletResponse`, tels qu'ils existent pour les servlets.

L'objectif de l'action est d'effectuer un traitement, généralement réalisé par la couche métier, avec un Bean Spring ou un EJB, par exemple, puis de renvoyer l'utilisateur vers une nouvelle page Web. Cette page est représentée par l'objet `ActionForward`, que retourne l'action.

Les formulaires Struts héritent tous de la classe `org.apache.struts.action.ActionForm` et sont des JavaBeans standards.

Il s'agit de formulaires HTML créés dans une page Web. Lors de la validation d'un formulaire, Struts transforme la requête HTTP envoyée en cet objet Java, que nos pouvons ensuite plus facilement manipuler. Chaque paramètre de la requête HTTP est renseigné dans l'attribut du formulaire qui porte son nom.

Par exemple, pour la requête HTTP suivante :

```
http://localhost/Tudu/exampleAction.do?listId=001&name=test
```

le formulaire `ExampleForm` a ses attributs `listId` et `name` renseignés (méthodes `setListId` et `setName`), et nous pouvons accéder à ces valeurs *via* les méthodes `getListId` et `getName`.

Il faut cependant être conscient que ce formulaire est l'exacte représentation de ce qu'a envoyé l'utilisateur *via* son navigateur et que les données qu'il contient ne sont donc pas forcément valides. C'est pourquoi les formulaires Struts possèdent une méthode `validate`, qui est automatiquement appelée si, dans le fichier **struts-config.xml,** l'attribut `validate="true"` est mis à une action donnée, ce qui est le cas dans l'exemple de configuration ci-dessus.

Voici un exemple de formulaire inspiré de Tudu Lists :

```
package tudu.web.form;

( ... )

public class ExampleForm extends ActionForm {

    private String listId;
```

```
    private String name;

    public ActionErrors validate(ActionMapping mapping,    HttpServletRequest request) {

        ActionErrors errors = new ActionErrors();
        if (this.listId == null || this.listId.equals("")) {
            ActionMessage message = new ActionMessage(
                    "errors.required",  "Todo List ID");

            errors.add(ActionMessages.GLOBAL_MESSAGE, message);
        }
     if (this.name == null || this.name.equals("")) {
            ActionMessage message = new ActionMessage(
                    "errors.required",  "name");

            errors.add(ActionMessages.GLOBAL_MESSAGE, message);
        }
        return errors;
    }

    public String getListId() {
        return listId;
    }

    public void setListId(String listId) {
        this.listId = listId;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }
}
```

Dans cet exemple, la méthode `validate` force l'utilisateur à renseigner les champs `listId` et `name` de son formulaire. Sans cela, la requête HTTP envoyée n'atteindrait même pas l'action.

Les bibliothèques de tags

Afin d'aider à l'affichage des pages JSP, Struts propose en standard les cinq bibliothèques de tags *(tag libraries)* suivantes :

- **Bean.** Sert à afficher et manipuler des JavaBeans dans une JSP. Développée avant l'existence des bibliothèques de tags standards, ou JSTL (JavaServer Pages Standard Tag Library), cette bibliothèque présente aujourd'hui peu d'intérêt. Rappelons que

depuis J2EE 1.4, nous pouvons utiliser les JavaBeans directement dans les Pages JSP, de la manière suivante :

```
${todo.todoList.name}
```

- **HTML.** D'excellente qualité, cette bibliothèque est l'une des raisons du succès de Struts. Très proche dans sa syntaxe des éléments de formulaire HTML, elle est intuitive et permet de réaliser facilement des formulaires Struts. Bien entendu, les formulaires HTML peuvent toujours être réalisés en HTML classique, Struts interprétant uniquement la requête HTTP envoyée. Cette bibliothèque n'est donc pas obligatoire mais fournit une aide intéressante.
- **Logic.** Sert à réaliser des boucles et des branchements conditionnels. Comme la bibliothèque Bean, elle n'est plus d'actualité depuis l'apparition de JSTL.
- **Nested.** Reprend l'ensemble des bibliothèques Struts, en leur donnant la capacité de s'imbriquer les unes dans les autres. Cela simplifie considérablement l'utilisation d'arbres d'objets complexes.
- **Tiles.** Permet l'inclusion et le paramétrage de fragments Tiles *(voir ci-après)*.

En résumé, une grande partie des bibliothèques de tags de Struts ont été détrônées par JSTL. Reste principalement la bibliothèque HTML, qui permet de construire plus facilement des formulaires utilisés avec Struts. En voici un exemple, issu d'une simplification de la page **user_info.jsp** de Tudu Lists :

```
<h3><fmt:message key="user.info.title"/></h3>
<html:form action="/secure/myInfo" focus="firstName">
 <c:if test="${success eq 'true'}">
  <span class="success"><fmt:message key="form.success"/></span>
 </c:if>
 <html:errors/>
 <hr/>
  <fmt:message key="user.info.first.name"/> :
  <html:text property="firstName" size="15" maxlength="60"/>
 <br/>
  <fmt:message key="user.info.last.name"/> :
  <html:text property="lastName" size="15" maxlength="60"/>
 <br/>
  <fmt:message key="user.info.email"/> :
  <html:text property="email" size="30" maxlength="100"/>
 <hr/>
 <html:submit>
  <fmt:message key="form.submit"/>
 </html:submit>
 <html:submit><fmt:message key="form.cancel"/></html:submit>
</html:form>
```

La technologie Tiles

Tiles est une technologie de découpage de pages JSP qui s'appuie sur les `include` des JSP en les améliorant significativement. En particulier, Tiles permet d'effectuer les tâches suivantes :

- Créer aisément des modèles de pages réutilisables, avec un support de l'héritage entre différents modèles.
- Nommer les pages au lieu de donner leur chemin complet, dans l'esprit des tags `<forward>` vus précédemment dans la configuration de Struts.
- Réutiliser des composants de présentation, y compris des composants internationalisés (nous n'affichons pas le même composant suivant la langue de l'utilisateur).

Il y a donc de grandes chances pour que vous souhaitiez mettre Tiles en œuvre dans vos projets, d'autant plus que ce n'est guère compliqué.

Tiles fait partie intégrante du code source de Struts, bien qu'il s'agisse d'un composant optionnel, qui n'est pas activé par défaut. Pour l'activer, il suffit d'ajouter les lignes suivantes à la fin du fichier **struts-config.xml** (juste avant la balise de fermeture `</struts-config>`) :

```
<plug-in className=
"org.apache.struts.tiles.TilesPlugin">
  <set-property property=
  "definitions-config"
   value="/WEB-INF/tiles-defs.xml"/>
</plug-in>
```

Il suffit ensuite de créer le fichier **tiles-def.xml** dans le répertoire **WEB-INF.** Voici un exemple de ce fichier, dans lequel nous définissons un modèle simple (`layout`) et deux pages qui en héritent :

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE tiles-definitions PUBLIC "-//Apache Software Foundation//DTD
➥Tiles Configuration 1.1//EN" "http://jakarta.apache.org/struts/dtds/
➥tiles-config_1_1.dtd">

<tiles-definitions>
 <definition name="layout" path="/WEB-INF/jsp/layout.jsp">
  <put name="title" value="Titre par defaut" />
  <put name="body" />
 </definition>
 <definition name="index" extends="layout">
  <put name="body" value="/WEB-INF/jsp/index.jsp" />
  <put name="title" value="Bienvenue" />
 </definition>
 <definition name="user" extends="layout">
  <put name="body" value="/WEB-INF/jsp/user.jsp" />
  <put name="title" value="Gestion des utilisateurs" />
 </definition>
</tiles-definitions>
```

Pour utiliser ces composants dans Struts, il faut modifier les `<forward>` dans **struts-config.xml.** Dans l'exemple précédent, pour rediriger vers le composant `index`, il faut mettre dans la configuration de l'action :

```
<forward name="success" path="index"/>
```

Nous utilisons comme chemin *(path)* le nom de la définition Tiles configurée dans le fichier **tiles-def.xml.** Bien entendu, nous pouvons toujours utiliser des pages JSP classiques à la place de Tiles. En effet, à chaque `forward`, Struts vérifie si une définition Tiles correspond au chemin demandé. S'il n'en trouve pas, il bascule en fonctionnement normal et peut alors rediriger vers une JSP classique.

Points faibles et problèmes liés à Struts

Struts a été le premier framework MVC à rencontrer un fort succès auprès de la communauté Java. Depuis sa création, fin 2000, ce framework a cependant relativement mal vieilli, et un certain nombre de problèmes sont apparus, notamment les suivants :

- Trop grande adhérence avec les classes `HttpServletRequest` et `HttpServletResponse`, qui sont passées en paramètre de la méthode `execute` des actions. Cela limite Struts à être utilisé avec des servlets et des pages JSP, si bien qu'il est impossible de réutiliser une couche Struts dans un client Swing.
- Utilisation de classes concrètes, et non d'interfaces, ce qui limite considérablement la possibilité de tester des actions Struts. En particulier, il est impossible d'utiliser des simulacres d'objets, ou mock objects, pour tester des actions Struts.
- Absence de mécanisme standard d'interception des actions. Une chaîne d'interception permettrait, par exemple, de gérer le logging, la sécurité ou l'ouverture des accès en base de données dans des objets spécialisés et transversaux aux actions.
- Erreurs de typage dans les formulaires, qui lancent des exceptions techniques critiques. Au final, les seuls attributs réellement utilisables pour les formulaires sont de type String. Cela ajoute un niveau de conversion avec les objets métier d'une complexité inutile.
- Prise en compte des seules données envoyées par les formulaires. Les données de référence, par exemple une liste à sélectionner, doivent être traitées séparément et mises en attribut de requête ou de session, de la même manière qu'avec les servlets. Cela lie encore plus le contrôleur à une couche de présentation codée en JSP.
- Nécessité, pour la bibliothèque de tags HTML, d'utiliser un éditeur de JSP supportant Struts. Sa manière de fonctionner l'empêche en effet d'être comprise par des éditeurs HTML couramment utilisés, tels que DreamWeaver.
- Séparation entre actions et formulaires peu justifiée, qui augmente le nombre de classes et de lignes à écrire.
- Absence de mécanisme d'accès à une couche métier. Dans la pratique, les actions Struts sont forcées d'aller elles-mêmes se connecter à la couche métier, par exemple en

utilisant JNDI dans le cadre d'une couche métier codée en EJB. Cela présente les inconvénients supplémentaires suivants :

– Gêne encore plus la testabilité des actions, puisqu'il est impossible d'exécuter un test JUnit si l'objet testé appelle un EJB dans ses méthodes.

– Lie les actions Struts à une implémentation de la couche métier et non à des interfaces.

– Limite l'utilisation de fonctionnalités avancées lors de l'accès à la couche métier, comme des proxy dynamiques ou de la POA pour gérer les transactions.

Ces désavantages, minimes au début de la vie de Struts, sont devenus aujourd'hui plus visibles : de nouveaux frameworks, mieux conçus, ont mis en lumière ces manquements. Citons parmi eux Spring MVC, que nous présentons au chapitre suivant, mais aussi Tapestry ou WebWork, avec lequel Struts est destiné à fusionner depuis fin 2005.

Struts et JSF (Java Server Faces)

JSF est l'API standard (JSR 127 du Java Community Process) d'affichage des composants dans des pages Web. Il ne s'agit pas d'un concurrent direct de Struts, puisqu'il se préoccupe uniquement de la partie présentation. Par ailleurs, rien n'empêche d'utiliser Struts en tant que contrôleur, au sens MVC, de l'application. Il n'y a donc rien de surprenant à la bonne intégration de ces deux technologies, qui sont de surcroît l'œuvre d'un même homme, Craig McClanahan.

JSF est davantage concurrent de la bibliothèque de tags HTML de Struts, mais rien n'empêche d'utiliser ces deux technologies conjointement.

Concernant l'avenir de Struts, il est certain que sa très grande popularité lui vaudra de rester encore longtemps d'actualité. L'un des objectifs de l'équipe qui l'a développé a toujours été d'assurer une bonne compatibilité entre les versions. C'est pourquoi elle a préféré une API stable à une évolution vers des concepts plus modernes. Dans la pratique, le succès de Struts lui a donné raison.

Cependant, Struts pourrait être détrôné à long terme par plusieurs autres technologies, notamment les suivantes :

- **Shale.** Nouvelle version de Struts, toujours développée par la fondation Apache et Craig McClanahan, résolvant la majorité des problèmes soulevés à l'encontre de Struts. Ne semble pas avoir encore provoqué d'engouement particulier.
- **Spring MVC.** Excellente implémentation du pattern MVC, entièrement intégrée à Spring.
- **AJAX.** Technique de codage d'une page Web très en vogue utilisant JavaScript pour charger à la demande des morceaux d'information insérés à chaud dans la page en cours. Tudu Lists en fait une utilisation intensive pour gérer les listes de todos et les todos eux-mêmes, et nous la présentons en détail au chapitre 9. Struts est mal adapté à ce type de programmation, qui possède ses propres frameworks, comme DWR, que nous présentons également au chapitre 9. Utilisé dans Tudu Lists, ce dernier permet d'utiliser directement des Beans Spring en JavaScript, ce qui court-circuite complètement Struts.

En résumé

Bien que toujours très populaire, Struts est un framework vieillissant. Parmi les nombreuses solutions de rechange plus évoluées techniquement disponibles sur le marché, citons Spring MVC, Tapestry ou WebWork.

Ces solutions ont en commun d'être fondées sur des conteneurs légers et de proposer des mécanismes d'interception. Nous verrons dans les sections qui suivent qu'en couplant Spring et Struts, il est possible d'obtenir les mêmes avantages.

Intégration de Struts à Spring

Bien que Spring possède son propre framework MVC, il s'intègre parfaitement avec Struts.

Nous verrons dans cette section de quelle manière cette intégration est réalisée et tenterons de dégager les bénéfices que nous pouvons en retirer, en particulier du point de vue des utilisateurs de Struts.

Intérêt de l'intégration de Struts à Spring

L'intérêt de l'intégration de Struts à Spring est d'ajouter à Struts un accès performant à la couche métier de l'application, Spring se chargeant de gérer la couche métier. Ce faisant, les actions Struts ne sont plus liées à des implémentations d'objets métier mais à leurs interfaces. Toutes les fonctionnalités de Spring sont disponibles lors de l'accès à cette couche métier, notamment l'utilisation de la POA pour gérer les transactions.

Spring vient ainsi combler une importante lacune de Struts, contribuant à moderniser ce framework.

Cette intégration est disponible sous trois formes différentes, chacune ayant ses avantages et ses inconvénients. Les sections qui suivent passent en revue chacune d'elles.

Configuration commune

Quel que soit le type d'intégration choisi, il faut tout d'abord ajouter un contexte Spring à Struts. Il s'agit en réalité d'un sous-contexte du contexte Spring général, qui se charge en tant que plug-in Struts.

Pour cela, il faut ajouter les lignes suivantes dans le fichier **struts-config.xml :**

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">

    <set-property  property="contextConfigLocation"
                   value="/WEB-INF/action-servlet.xml"/>
</plug-in>
```

Cette configuration provient de Tudu Lists, qui fournit un exemple d'intégration de Spring et de Struts. Habituellement, ce fichier de configuration Spring est nommé **action-servlet.xml.**

*Utilisation d'*ActionSupport

La manière la plus simple de connecter Struts à Spring consiste à utiliser la classe `org.springframework.web.struts.ActionSupport`. Il suffit pour cela de faire hériter ses actions de `ActionSupport` au lieu de `org.apache.struts.action.Action`.

L'action Struts en cours hérite alors de la méthode `getWebApplicationContext`, qui permet d'accéder au contexte d'application Spring configuré précédemment (en tant que plug-in Struts) :

```
package tudu.web;

( ... )

/** Backup a Todo List. */
public class BackupTodoListAction
    extends org.springframework.web.struts.ActionSupport {

    public final ActionForward execute(
ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
            throws Exception {

        DynaActionForm todoListForm = (DynaActionForm) form;
        String listId = (String) todoListForm.get("listId");
        ApplicationContext ctx = getWebApplicationContext();
        TodoListsManager todoListsManager =
        (TodoListsManager) ctx.getBean("todoListsManager");
        TodoList todoList = todoListsManager.findTodoList(listId);
        Document doc = todoListsManager.backupTodoList(todoList);
        request.getSession()
        .setAttribute("todoListDocument", doc);
        return mapping.findForward("backup");
    }
}
```

Dans cet exemple, nous recherchons le Bean Spring `todoListsManager` dans le contexte Spring.

Spring propose des équivalents plus évolués des actions Struts, comme `DispatchAction` (`DispathActionSupport`). Le code reste proche de celui d'une action Struts classique et est simple à mettre en œuvre dans une application existante. Il ne permet toutefois pas de bénéficier pleinement des capacités de Spring au niveau de la couche MVC, comme l'inversion de contrôle pour injecter des Beans métier dans les actions. Pour cette raison, ce n'est pas la méthode que nous recommandons, en dépit de sa facilité d'utilisation attrayante.

Le DelegationRequestProcessor

Changer le `RequestProcessor` de Struts permet de découpler les actions Struts de Spring. Il n'y a donc plus d'import de classes Spring dans les actions, et les actions Struts deviennent de véritables Beans Spring, qui bénéficient de toute la puissance de ce framework (injection de dépendances, POA, etc.).

Pour changer le `RequestProcessor`, il faut ajouter un élément dans le fichier **struts-config.properties,** entre l'élément `<message-resources>` et l'élément `<plug-in>` :

```
<controller processorClass="org.springframework.web.struts.
   DelegatingRequestProcessor"/>
```

Une action Struts doit être définie en tant qu'action dans le fichier **struts-config.properties** et en tant que Bean Spring dans le fichier **action-servlet.xml.**

Dans **struts-config.properties :**

```
<action path="/secure/backupTodoList"
    name="todoListForm"
    type="tudu.web.BackupTodoListAction">

</action>
```

Dans **action-servlet.xml :**

```
<bean name="/secure/backupTodoList"
class="tudu.web.BackupTodoListAction">

   <property name="todoListsManager">
      <ref bean="todoListsManager" />
    </property>
</bean>
```

Notons que le nom du chemin *(path)* Struts est identique à celui du Bean Spring.

L'action vue précédemment bénéficie maintenant de l'inversion de contrôle et peut être réécrite ainsi :

```
package tudu.web;

( ... )

/** Backup a Todo List. */
public class BackupTodoListAction
    extends org.apache.struts.action.Action {

    private TodoListsManager todoListsManager = null;

    public final void setTodoListsManager(
    TodoListsManager todoListsManager) {

        this.todoListsManager = todoListsManager;
    }
```

```
    public final ActionForward execute(
    ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response)
            throws Exception {

        DynaActionForm todoListForm = (DynaActionForm) form;
        String listId = (String) todoListForm.get("listId");
        TodoList todoList = todoListsManager.findTodoList(listId);
        Document doc = todoListsManager.backupTodoList(todoList);
        request.getSession()
        .setAttribute("todoListDocument", doc);
        return mapping.findForward("backup");
    }
}
```

La seule faiblesse de cette méthode est d'utiliser un `RequestProcessor` spécifique. Cela pose des problèmes aux applications qui utilisent leur propre `RequestProcessor`, une technique couramment utilisée pour étendre Struts.

Ainsi, pour l'utilisation de Tiles, qui utilise lui aussi son propre `RequestProcessor`, une classe `DelegatingTilesRequestProcessor` est fournie par Spring. Mais cet exemple met en lumière la nécessité de coder des `RequestProcessor` très spécifiques afin de lier Struts et Spring.

La délégation d'actions

La délégation d'actions est la méthode que nous conseillons et que nous utilisons avec Tudu Lists.

Proche de la méthode précédente, elle permet de gérer les actions Struts depuis Spring comme des JavaBeans, mais sans impact sur la configuration de Struts. L'idée est d'utiliser une classe spécifique, `org.springframework.web.struts.DelegatingActionProxy`, afin de déléguer la gestion de l'action Struts à Spring.

Dans la pratique, nous continuons à coder des actions Struts normalement (ces classes peuvent même hériter d'actions complexes, comme les `DispatchAction`). La seule différence est que, dans **struts-config.xml,** nous donnons `org.springframework.web.struts.DelegatingActionProxy` comme étant la classe de l'action.

Par exemple, dans Tudu Lists, l'action `"/register"` est configurée de la façon suivante :

```
<actionpath="/register"
    name="registerForm"
    type="org.springframework.web.struts.DelegatingActionProxy"
    parameter="method"
    validate="false"
    input="/WEB-INF/jsp/register.jsp">

<forward name="register" path="/WEB-INF/jsp/register.jsp"/>
<forward name="success" path="/WEB-INF/jsp/register_ok.jsp"/>
     <forward name="cancel" path="/welcome.action" redirect="true"/>
</action>
```

Dans le fichier de configuration Spring **action-servlet.xml,** cette action est configurée de la façon suivante :

```
<bean name="/register" class="tudu.web.RegisterAction">
    <property name="userManager">
        <ref bean="userManager" />
    </property>
</bean>
```

Spring réalise le lien avec l'action Struts *via* le nom du Bean, identique au chemin de l'action Struts `"/register"`.

De même que pour la méthode précédente, `RegisterAction` est à la fois une action Struts complète, qui hérite de `DispatchAction`, et un Bean Spring à part entière, bénéficiant de l'inversion de contrôle.

En résumé

Il est relativement facile d'intégrer Struts et Spring et de combiner ainsi les avantages de chaque framework :

- Struts apporte une couche MVC simple et bien connue des développeurs.
- Spring apporte un accès simple et non intrusif à une couche métier performante.

Nous avons vu qu'il était possible d'avoir des classes étant à la fois des actions Struts et des Beans Spring. Ces « nouvelles actions Struts » ont des capacités bien supérieures aux actions classiques, réduisant ainsi les faiblesses de Struts.

Grâce à la délégation d'actions, ces nouvelles classes bénéficient des avantages suivants :

- Elles ne lient plus la couche de présentation à une implémentation donnée de la couche métier.
- Elles sont plus facilement testables de manière unitaire du point de vue de l'accès à la couche de service. Elles dépendent toutefois toujours des API servlet et Struts et restent donc difficiles à tester.
- Elles disposent d'un mécanisme d'interception. Ces classes étant des Beans Spring, nous pouvons utiliser la POA pour intercepter leurs méthodes. Cela peut permettre d'ajouter des mécanismes transversaux de sécurité, de monitoring, de logging, de cache, etc.

Tudu Lists : intégration de Struts

Reprenant notre étude de cas Tudu Lists, nous allons mettre en pratique l'intégration de Spring et de Struts en utilisant la délégation d'actions.

Pour Tudu Lists, nous avons fait le choix d'une couche de présentation MVC avec Struts, enrichie de l'intégration avec Spring. Malgré ses défauts, nous avons préféré utiliser

Struts en standard avec Tudu Lists en raison de sa très large diffusion et parce que nous en connaissons bien les rouages.

Pour cette application particulière, nous n'avons pas besoin d'une couche de présentation très élaborée, capable, par exemple, de gérer des workflows complexes. Si cela avait été le cas, nous aurions opté pour l'utilisation conjointe de Struts et d'une technologie plus adaptée, par exemple Spring Web Flow.

Concernant la partie vue de l'application, nous avons choisi d'utiliser des pages JSP dans le cas général, en utilisant au maximum la JSTL (JavaServer Pages Standard Tag Library). Nous utilisons aussi les nouvelles notations apportées par J2EE 1.4, de type `${javabean.attribut}`, ce qui réduit et clarifie considérablement le code. Dans certains cas très particuliers, nous avons utilisé des servlets : c'est le cas de la génération de flux RSS. Ce flux étant généré par Rome, un framework Open Source spécialisé dans la gestion de ce type de flux, l'utilisation d'une JSP n'a pas d'intérêt.

Tudu Lists utilise aussi la technologie AJAX (Asynchronous Javascript And XML), qui permet de communiquer en XML avec le serveur sans recharger la page Web en cours. Une partie de la couche de présentation a donc été réalisée avec DWR, un framework permettant d'utiliser des Beans Spring côté serveur directement en JavaScript côté client. Cette technique est détaillée au chapitre 9, dédié à AJAX.

Les fichiers de configuration

La configuration de la partie présentation de Tudu Lists est répartie dans les fichiers suivants :

- **WEB-INF/web.xml.** Fichier standard de configuration des applications Web J2EE servant ici à configurer la servlet `ActionServlet` de Struts.
- **WEB-INF/struts-config.xml.** Fichier de configuration de Struts que nous avons étudié précédemment. Tudu Lists utilisant la délégation d'actions, l'ensemble des actions décrites dans ce fichier est de type `org.springframework.web.struts.DelegatingActionProxy`. Afin de limiter le nombre de formulaires Struts, nous utilisons des `DynaForm`. Uniquement décrits en XML dans le fichier **struts-config.xml,** ces formulaires n'ont pas d'implémentation Java.
- **WEB-INF/validation.xml.** Fichier de configuration du Validator de Struts. Ce dernier est un plug-in Struts aidant à la validation de formulaires en fournissant des implémentations Java et JavaScript des principales règles de validation trouvées dans une application Web (champ obligatoire, tailles minimale et maximale, champ e-mail, etc.).
- **WEB-INF/validator-rules.xml.** Fichier de configuration interne du Validator, n'ayant normalement pas à être modifié.
- **WEB-INF/action-servlet.xml.** Fichier contenant une configuration Spring standard, dans laquelle les actions Struts sont définies en tant que Beans Spring. Cela permet de configurer l'injection de dépendances dont elles bénéficient.

- **JavaSource/messages.properties.** Fichier stockant les messages utilisés dans l'application. Il s'agit d'un fichier de propriétés, aussi appelé *resource bundle*, permettant de gérer plusieurs langues.

Exemple d'action Struts avec injection de dépendances

Pour cet exemple, nous allons utiliser `tudu.web.MyInfoAction`, une action qui permet à l'utilisateur de gérer ses informations personnelles.

Lors de l'utilisation de cette action, le parcours de l'utilisateur est le suivant :

1. L'utilisateur exécute l'action `MyInfoAction` sans lui envoyer de paramètre.
2. Les informations concernant l'utilisateur en cours sont recherchées en base de données.
3. La page **WEB-INF/jsp/user_info.jsp** est affichée. Elle contient les informations récupérées précédemment, prêtes à être modifiées.
4. L'utilisateur modifie ces informations. Il peut ensuite soit annuler ses changements (retour à l'étape 1), soit les valider et passer à l'étape 5.
5. Si les informations envoyées ne sont pas correctes, l'utilisateur est renvoyé à l'étape 3.
6. Les informations sont sauvegardées en base de données, et nous revenons à l'étape 1.

La classe `MyInfoAction` est un groupement de trois actions différentes : l'affichage de la page, l'annulation du formulaire et sa validation.

Dans les premières versions de Struts, il fallait trois classes différentes pour représenter ces trois actions, ce qui obligeait à écrire beaucoup de code. C'est désormais simplifié grâce à la classe `org.apache.struts.actions.DispatchAction`, dont `MyInfoAction` hérite *via* `tudu.web.TuduDispatchAction`.

Ainsi, `MyInfoAction` est plus un groupe d'actions qu'une action, ce qui a pour effet de réduire le nombre de classes et la taille du fichier de configuration.

La signature de ses méthodes est la suivante :

```
package tudu.web;

( ... )

public class MyInfoAction extends TuduDispatchAction {

    public final void setUserManager(UserManager userManager) {
        ( ... )
    }

    public final ActionForward display(
    ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response) {
```

```
    ( ... )
    }

    public final ActionForward update(
    ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response) {

        ( ... )
    }
}
```

La première méthode, `setUserManager`, est utilisée pour l'injection de dépendances avec Spring.

Les deux autres méthodes, `display` et `update`, correspondent aux actions Struts dont nous avons précédemment parlé, qui servent à afficher et à mettre à jour les informations utilisateur. Il existe également une méthode `cancel`, dont `MyInfoAction` hérite, qui correspond à l'annulation de l'action. En réalité, cette méthode ne fait que rediriger vers la méthode `display`. En effet, une annulation effectue un nouvel affichage de la page initiale, avec les données d'origine.

Ces actions sont configurées *via* le fichier **struts-config.xml :**

```
<form-bean name="userForm"
        type="org.apache.struts.validator.DynaValidatorForm">

<form-property name="password" type="java.lang.String"/>
<form-property name="verifyPassword" type="java.lang.String"/><form-property
➥name="firstName" type="java.lang.String"/>
<form-property name="lastName" type="java.lang.String"/>
<form-property name="email" type="java.lang.String"/>
</form-bean>

( ... )
<action
    path="/secure/myInfo"
    name="userForm"
    type="org.springframework.web.struts.DelegatingActionProxy"
    parameter="method"
    validate="false"
    input="/WEB-INF/jsp/user_info.jsp">

<forward name="user.info" path="/WEB-INF/jsp/user_info.jsp"/>
</action>
```

`MyInfoAction` peut exécuter chacune des méthodes `display`, `update` et `cancel` en fonction d'un paramètre qui lui est envoyé. Ce paramètre, configuré plus haut en tant qu'attribut de l'action, est défini à `"method"`. Cela signifie qu'un paramètre HTTP `"method"` va être envoyé avec le formulaire et qu'en fonction de ce paramètre une méthode sera exécutée.

On retrouve ce paramètre dans la page JSP **WEB-INF/jsp/user_info.jsp** (le code suivant est volontairement simplifié afin de mettre en valeur l'essentiel) :

```
<html:form action="/secure/myInfo" focus="firstName">
<html:errors/>
<html:hidden property="method" value="cancel"/>

<fmt:message key="user.info.first.name"/>
<html:text property="firstName" size="15" maxlength="60"/>
<br/>
<fmt:message key="user.info.last.name"/>
<html:text property="lastName" size="15" maxlength="60"/>
<br/>
<fmt:message key="user.info.email"/>
<html:text property="email" size="30" maxlength="100"/>
<br/>
<fmt:message key="user.info.password"/>
<html:password property="password" size="15" maxlength="32"/>
<br/>
<html:submit
onclick="document.forms[0].elements['method'].value='update';">

<fmt:message key="form.submit"/>
</html:submit>
<html:submit><fmt:message key="form.cancel"/></html:submit>
</html:form>
```

L'utilisation du champ HTML `"method"`, qui est un champ caché, ainsi que celle de JavaScript dans l'événement `"onClick"` du bouton `"Submit"`, permettent d'envoyer avec le formulaire une variable supplémentaire.

L'envoi du formulaire permet ainsi d'envoyer les variables `"firstName"`, `"lastName"`, `"email"`, `"password"` et `"method"`. C'est cette dernière variable qui sera utilisée par Struts pour déterminer quelle méthode utiliser dans `MyInfoAction`.

Intégration de Spring et de Struts

Dans le fichier **struts-config.xml,** l'action est configurée en tant que `DelegatingActionProxy`. Nous utilisons donc la troisième forme d'intégration Struts-Spring présentée précédemment, à savoir la délégation d'actions.

Nous retrouvons par conséquent le JavaBean `MyInfoAction` configuré dans le fichier **WEB-INF/action-servlet.xml :**

```
<bean name="/secure/myInfo" class="tudu.web.MyInfoAction">
<property name="userManager">
    <ref bean="userManager" />
</property>
</bean>
```

Grâce à cette configuration, le JavaBean `userManager` est injecté dans `MyInfoAction`. Nous pouvons l'utiliser directement dans `MyInfoAction`, par exemple dans la méthode `display` :

```
public final ActionForward display(
    ActionMapping mapping, ActionForm form,
    HttpServletRequest request, HttpServletResponse response) {

        log.debug("Execute display action");
        String login = request.getRemoteUser();
        User user = userManager.findUser(login);
        DynaActionForm userForm = (DynaActionForm) form;
        userForm.set("password", user.getPassword());
        userForm.set("verifyPassword", user.getPassword());
        userForm.set("firstName", user.getFirstName());
        userForm.set("lastName", user.getLastName());
        userForm.set("email", user.getEmail());
        return mapping.findForward("user.info");
    }
```

Dans cet exemple, nous avons donc bien une action Struts complète relativement évoluée, notamment grâce à l'utilisation de `DispatchAction`, qui s'intègre parfaitement dans Spring en tant que JavaBean standard et bénéficie ainsi de l'injection de dépendances.

Utilisation conjointe des DynaBeans et du Validator

Tudu Lists utilise deux fonctionnalités avancées de Struts que nous n'avons pas encore détaillées, les DynaBeans et le Validator. Utilisées conjointement, elles réduisent la complexité de l'application en supprimant la majeure partie du code nécessaire au codage des formulaires.

Coder des formulaires Struts devient rapidement répétitif puisque nous nous retrouvons avec un grand nombre de JavaBeans qui n'ont pas grande utilité. De plus, la validation de ces formulaires est fastidieuse, car il faut chaque fois coder la méthode `validate()` des formulaires, un travail le plus souvent long, même pour un résultat simple.

Voici un exemple de méthode `validate()` qui teste si un champ obligatoire a été renseigné :

```
public ActionErrors validate(
ActionMapping mapping, HttpServletRequest request) {

ActionErrors errors = new ActionErrors();
if (this.todoId == null || this.todoId.equals("")) {
    ActionMessage message = new ActionMessage(
                    "errors.required",  "Todo ID");

    errors.add(ActionMessages.GLOBAL_MESSAGE, message);
}
return errors;
}
```

Ce code est incomplet, car il ne représente qu'une validation côté serveur, alors que bien souvent une validation côté client en JavaScript est nécessaire afin de ne pas surcharger le serveur de requêtes contenant des informations potentiellement non valides.

Les formulaires Struts se révélant longs et répétitifs à coder, les DynaBeans et le Validator ont été conçus pour soulager le développeur.

Un DynaBean est un formulaire Struts décrit uniquement en XML dans le fichier **struts-config.xml.** Nous en avons déjà rencontré un dans le code décrivant le formulaire `userForm` :

```
<form-bean name="userForm"
        type="org.apache.struts.validator.DynaValidatorForm">

<form-property name="password" type="java.lang.String"/>
<form-property name="verifyPassword" type="java.lang.String"/><form-property
name="firstName" type="java.lang.String"/>
<form-property name="lastName" type="java.lang.String"/>
<form-property name="email" type="java.lang.String"/>
</form-bean>
```

Malheureusement, une des limitations de Struts vient de ce que le type des propriétés des DynaBeans est presque obligatoirement `java.lang.String`. Struts renvoie des erreurs fatales en cas de non-correspondance entre une propriété envoyée par le navigateur client et son type.

Le Validator permet quant à lui de valider automatiquement le formulaire côté serveur comme côté client.

Cette validation est définie dans le fichier **WEB-INF/validation.xml :**

```
<form name="userForm">
<field property="password" depends="required">
    <arg0 key="user.info.password" />
</field>
<field property="verifyPassword" depends="validwhen">
    <arg0 key="user.info.password.not.matching" />
    <var>
        <var-name>test</var-name>
        <var-value>(*this* == password)</var-value>
    </var>
</field>
<field property="firstName" depends="required">
    <arg0 key="user.info.first.name" />
</field>
<field property="lastName" depends="required">
    <arg0 key="user.info.last.name" />
</field>
<field property="email" depends="email">
    <arg0 key="user.info.email" />
</field>
</form>
```

Dans cet exemple, nous pouvons définir des champs comme `"required"` (obligatoire), `"validwhen"` (valable en fonction d'un test) ou `"email"` (s'il s'agit d'un e-mail valide). Un certain nombre de validations standards, en particulier sur le type et la taille des champs, sont en outre configurées dans le fichier **WEB-INF/validator-rules.xml.** Il est bien entendu possible d'ajouter des validations supplémentaires, mais Struts fournit par défaut une bibliothèque de règles de validation assez complète.

Cette validation fonctionne automatiquement côté serveur et peut être ajoutée côté client en insérant simplement le tag suivant dans la page JSP de formulaire :

```
<html:javascript formName="userForm"/>
```

L'ajout de ce tag permet de générer à la volée le JavaScript nécessaire à la validation du formulaire passé en paramètre.

Dans Tudu Lists, nous utilisons cette double technique des DynaBeans et du Validator afin de réduire la taille du code à écrire et réaliser l'application le plus rapidement possible.

Cette solution n'est toutefois pas à recommander dans tous les cas, notamment pour les raisons suivantes :

- Le temps de codage des formulaires n'est pas toujours très long. Par exemple, coder un JavaBean avec Eclipse est très rapide, en particulier grâce au menu Source/Generate Getters and Setters.
- Les DynaBeans font perdre la garantie de la compilation et ne permettent plus d'utiliser les fonctionnalités de refactoring d'Eclipse pour renommer un champ.
- Très utile pour les validations simples, le Validator devient soit trop complexe pour les validations plus avancées (plus simples à réaliser en Java), soit complètement inefficace pour les validations métier (qui nécessitent de toute manière d'être exécutées au niveau de l'action).

Dans une application, la technique décrite précédemment n'est valable que pour les formulaires simples, pour lesquels elle apporte des gains de temps et de taille de code.

Dans le cas particulier de Tudu Lists, la majorité des formulaires est assez simple. Il s'agit soit de pages aux fonctionnalités élémentaires (comme l'action `MyInfoAction`), soit de pages plus complexes dans lesquelles les fonctionnalités avancées sont déportées dans la couche AJAX.

Création d'un intercepteur sur les actions Struts

Outre l'injection de dépendances, l'un des intérêts d'intégrer Spring et Struts est d'utiliser des intercepteurs Spring sur les actions Struts.

Pour cet exemple, nous allons utiliser l'un des intercepteurs fourni en standard avec Spring, `org.springframework.aop.interceptor.DebugInterceptor`.

Cet intercepteur est configuré dans le fichier **WEB-INF/action-servlet.xml :**

```
<bean id="debugInterceptor"
 class="org.springframework.aop.interceptor.DebugInterceptor">

  <property name="loggerName">
    <value>tudu.interceptor.debug</value>
  </property>
</bean>

<bean    name="proxyCreator"
 class="org.springframework.aop.framework
            .autoproxy.BeanNameAutoProxyCreator">

  <property name="beanNames" value="/*"/>
  <property name="interceptorNames">
    <list>
        <value>debugInterceptor</value>
    </list>
  </property>
</bean>
```

Concernant le Bean `debugInterceptor`, il s'agit de la configuration classique d'un intercepteur Spring. Le `"proxyCreator"` est configuré pour correspondre au nom `"/*"`, ce qui permet d'intercepter l'ensemble des appels aux actions Struts.

Cette configuration permet d'obtenir des messages de log particulièrement intéressants pour déboguer une application, comme ici :

```
DEBUG tudu.interceptor.debug  - Exiting invocation: method 'execute',
arguments [ActionConfig[path=/secure/admin/administration,name=adminis-
trationForm,parameter=method,scope=request,type=org.springframe-
work.web.struts.DelegatingActionProxy,
DynaActionForm[dynaClass=administrationForm,smtpHost=smtp.test.com,smtpFr
om=email@test.com,smtpPassword=password,smtpPort=25,smtpUser=user],
net.sf.acegisecurity.wrapper.ContextHolderAwareRequestWrapper@676316,
org.apache.catalina.connector.ResponseFacade@9a6c56]; target is of class
[tudu.web.AdministrationAction]; count=4
```

Nous retrouvons dans ce message de log l'URL de l'action appelée, ainsi que l'ensemble des paramètres envoyés, la classe exécutée et le nombre d'appels à cette URL.

Ces messages apparaissant en mode debug, il faut, pour les observer, configurer le fichier **log4j.properties** (dans le répertoire **JavaSource**) de la manière suivante :

```
log4j.logger.tudu.interceptor.debug=DEBUG
```

Des intercepteurs peuvent être ajoutés ou enlevés très facilement. Ils servent généralement à monitorer l'application.

Points forts et points faibles de la solution

Cette solution permet de combiner les forces de Struts et de Spring :

- Elle est la plus rapide et la moins coûteuse à mettre en œuvre, car tout développeur maîtrisant Struts est presque immédiatement opérationnel.
- Du fait de l'injection de dépendances et des intercepteurs, l'utilisation de Struts est considérablement améliorée par Spring.

Cependant, elle ne résout pas tous les problèmes liés à Struts :

- Une grande partie des problèmes provenant de Struts restent présents, notamment l'utilisation de classes et non d'interfaces, la dépendance directe à l'API des servlets, etc.
- La configuration nécessaire est complexe et nécessite une bonne connaissance du fichier **struts-config.xml** et de la configuration de Spring.
- Les objets ainsi créés sont de mauvaise qualité du point de vue de Spring. Ils dépendent à la fois de l'API servlet et de l'API Struts, les rendant difficiles à tester.

Conclusion

Après une rapide présentation de Struts, ce chapitre a détaillé les nombreux problèmes liés à ce framework populaire mais vieillissant.

L'intégration de Spring et de Struts permet de bénéficier des avantages de chacune de ces deux technologies, notamment la vaste communauté d'utilisateurs de Struts et l'injection de dépendances et la POA proposées par Spring. La combinaison de ces deux technologies permet de réaliser une couche MVC de bonne qualité, sans nécessiter la maîtrise des nouveaux et complexes frameworks MVC à la mode.

Nous avons étudié les trois méthodes classiques d'intégration de Spring et de Struts, en constatant que la méthode de délégation d'action était généralement la plus intéressante. Au prix d'une configuration un peu lourde, il est possible d'avoir des objets qui soient à la fois des actions Struts et des Beans Spring, bénéficiant de la sorte des deux technologies.

Ce choix reste un compromis puisqu'il ne résout pas tous les problèmes liés à Struts et ne permet pas de créer des Beans Spring de bonne qualité, ces derniers dépendant des API servlet et Struts et restant difficilement testables unitairement. Il permet néanmoins de combiner intelligemment les forces de ces deux frameworks et d'obtenir un résultat rapide sans prendre trop de risques.

Il s'agit donc d'un bon compromis pour les équipes possédant déjà des compétences Struts et ne désirant pas passer à un autre framework MVC dans l'immédiat.

7

Spring MVC

La mise en pratique du pattern MVC (Model View Controller) offre une meilleure structuration du tiers présentation des applications J2EE 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 pattern, qui consiste à instaurer un point d'entrée unique ayant pour mission d'aiguiller les requêtes vers la bonne entité de traitement.

Malgré des applications structurées et une large utilisation, certains frameworks MVC, tels que Struts, nuisent à la flexibilité des applications. En effet, ce dernier contraint le développeur à se lier fortement à ses API et ne favorise pas la mise en place de technologies de présentation différentes. De plus, il ne fournit pas de solution d'intégration à une architecture en couches complète.

Le framework Spring offre une implémentation innovante du pattern MVC par le biais d'un module nommé Spring MVC, qui profite des avantages de l'injection de dépendances *(voir chapitres 2 et 3).*

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 simplifier les développements d'applications J2EE tout en favorisant leur structuration et leur flexibilité.

Implémentation du pattern MVC de type 2 dans Spring

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

Actuellement, le framework MVC le plus utilisé est incontestablement Struts, mais ce dernier connaît des limitations qui freinent son utilisation et incitent de plus en plus d'architectes et de développeurs à trouver des solutions de rechange. Un sondage non officiel publié sur le blog de Matt Raible concernant la popularité des frameworks MVC confirme cette tendance et place Spring MVC en première place devant Struts.

Nous ne reviendrons pas sur les concepts de base du pattern MVC, abordés au chapitre 6, et nous concentrerons sur les principes de fonctionnement et constituants de Spring MVC, l'implémentation du pattern MVC de type 2 par Spring.

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 J2EE, ses concepteurs considèrent que l'injection de dépendances offre un apport de taille pour concevoir et structurer des applications fondées sur le pattern 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 d'application de manière transparente ainsi que des intégrations avec d'autres frameworks MVC, tels Struts, 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 pattern 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 d'application 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 :

- Modélisation des contrôleurs sous forme d'interface et non de classes concrètes. Ce choix de conception permet d'implémenter facilement de nouveaux types de contrôleurs tout en bénéficiant des avantages de l'héritage. Ce choix favorise la mise en œuvre des tests unitaires des contrôleurs.
- Gestion des formulaires à l'aide d'un contrôleur spécifique permettant 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.

- Abstraction de l'implémentation des vues par rapport aux contrôleurs permettant de changer de technologie de présentation sans impacter le contrôleur.
- Abstraction par rapport aux API servlet. Grâce aux différentes implémentations des contrôleurs, les implémentations des contrôleurs ne se lient pas systématiquement à ces API, lesquelles sont en outre masquées lors de l'implémentation du passage du contrôleur à la vue.
- Possibilité d'implémenter et de configurer des intercepteurs directement au niveau du MVC sans passer par la POA.

Pour mettre en œuvre ces principes et composants, Spring MVC s'appuie sur les entités illustrées à la figure 7.1.

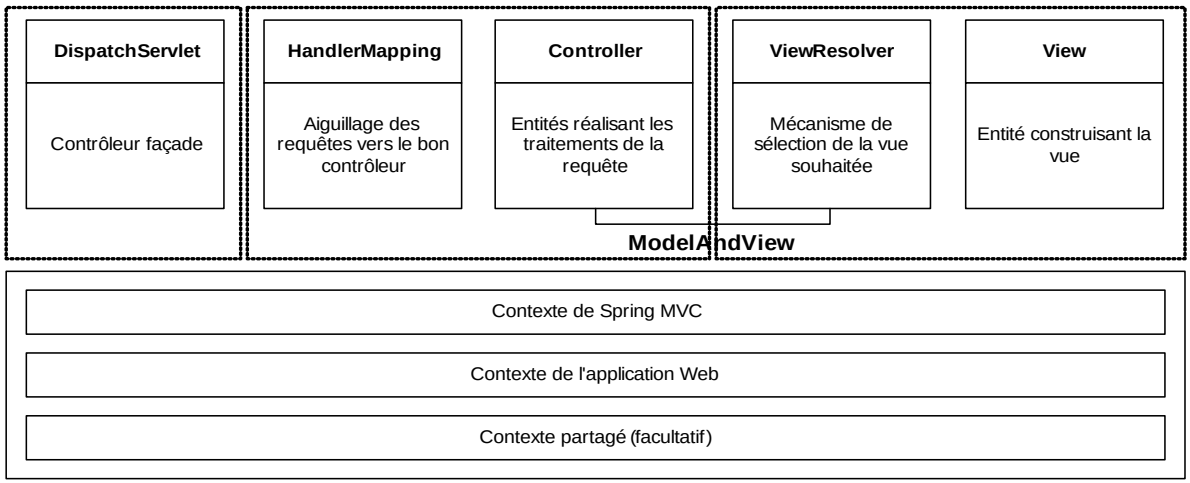


Figure 7.1

Entités de traitement des requêtes de Spring MVC

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

- **Gestion du contrôleur façade et des contextes d'application.** 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.
- **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'elles utilisent des mécanismes de la spécification J2EE servlet.

Gestion des contextes

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

Dans le cadre d'applications J2EE, 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**
>
> Rappelons qu'un contexte correspond au conteneur léger en lui-même, dont la fonction est de gérer des Beans *(voir chapitres 2 et 3)*. 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 très utile pour partager des objets d'une même bibliothèque entre plusieurs modules d'une même application Java/J2EE peut être partagé au niveau du 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 7.2 illustre cette hiérarchie ainsi que la portée des différents contextes.

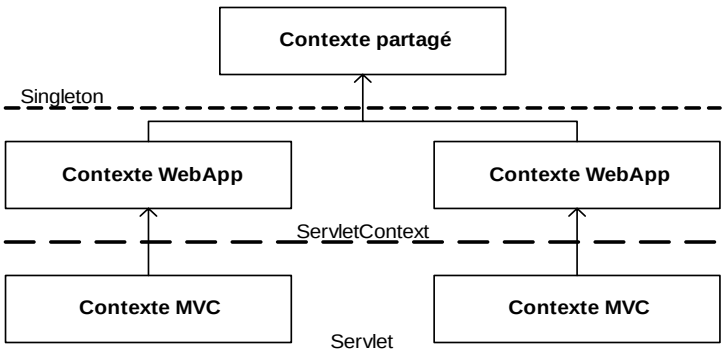


Figure 7.2
Hiérarchie des contextes de Spring pour une application Web

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 de Struts abordé au chapitre précédent.

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 écouteur se paramètre dans 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>
```

Au cas où le serveur d'applications utilisé supporte une ancienne version (antérieure à la 2.2 incluse), le framework Spring fournit une servlet afin de réaliser les mêmes traitements de configuration et d'initialisation. Les lignes suivantes doivent en ce cas être ajoutées dans le fichier **web.xml** à la place des précédentes :

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

<servlet>
    (...)
    <servlet-class>
        org.springframework.web.context.ContextLoaderServlet
    </servlet-class>
</servlet>
```

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 d'application fondé sur un fichier **<nom-servlet>-servlet.xml,** lequel utilise le nom de la servlet précédente pour `<nom-servlet>`, ce fichier se situant par défaut dans le répertoire **WEB-INF** et la valeur de `<nom-servlet>` étant spécifiée grâce à la balise `servlet-name`. Dans l'étude de cas, la servlet de Spring MVC s'appelle `action`, et le fichier **action-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 la façon de spécifier un fichier **mvc-context.xml** pour le contexte de Spring MVC :

```
<web-app>
    (...)
    <servlet>
        <servlet-name>example</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
        <init-param>
            <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 pattern MVC de type 2 signifie qu'il met en œuvre un contrôleur façade qui dirige 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 *(voir le chapitre 6, dédié à Struts).*

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

Le mappage de la ressource est défini au niveau du conteneur de servlets dans le fichier **WEB-INF/web.xml** (Spring ne pose aucune restriction à ce niveau) :

```
<web-app>
    (...)
    <servlet>
        <servlet-name>example</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>
    <servlet-mapping>
        <servlet-name>example</servlet-name>
        <url-pattern>*.html</url-pattern>
    </servlet-mapping>
</web-app>
```

Notons que Spring MVC n'impose pas le mapping pour la servlet principale. Dans notre cas, nous avons choisi arbitrairement ***.html.**

En résumé

Cette section a détaillé les mécanismes de chargement des contextes et de configuration du contrôleur façade du framework Spring MVC puis a montré comment configurer et implémenter les différentes entités de ce framework.

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

Traitement des requêtes

La mise en place de la partie contrôleur du MVC commence par la configuration de la correspondance des URI avec des classes de traitement.

Elle se poursuit éventuellement par des interceptions sur des requêtes puis par l'implémentation de ces contrôleurs de traitement.

Sélection du contrôleur

Comme pour tout framework implémentant le pattern 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 la stratégie choisie grâce au fichier de configuration du contexte de Spring MVC.

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 J2EE :

```
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 `DispatchServlet` de Spring MVC est paramétré dans le fichier **WEB-INF/web.xml** afin de prendre en compte un ensemble d'URI avec des mappages de la forme `*`, `/quelquechose/*` ou `*.quelquechose`. Nous détaillons la configuration de la première partie à la section « Initialisation du contrôleur façade », plus loin dans ce chapitre.

L'étape suivante consiste à configurer Spring MVC afin qu'il sélectionne l'entité traitant l'URI. Spring MVC dispose à cet effet de l'interface `HandlerMapping` du package `org.springframework.web.servlet`, laquelle possède différentes implémentations suivant le mécanisme de mappage souhaité. Cette interface est décrite ci-dessous (les implémentations sont localisées dans le package `org.springframework.web.servlet.handler`) :

```
public interface HandlerMapping {
    HandlerExecutionChain getHandler(
                    HttpServletRequest request) throws Exception;
}
```

Passons en revue les diverses implémentations de cette interface ainsi que leurs configurations et fonctionnements.

Tout d'abord, l'implémentation `BeanNameUrlHandlerMapping` permet d'utiliser le nom du Bean comme correspondance afin de réaliser un mappage simple et efficace.

Dans ce cas, l'attribut `name` identifie le Bean en lieu et place de l'attribut `id` :

```
<beans>
  <bean id="handlerMapping" class="org.springframework.web
                    .servlet.handler.BeanNameUrlHandlerMapping"/>

  <bean name="/welcome.action" class="tudu.web.WelcomeController"/>
<beans>
```

La configuration précédente dirige les traitements vers le contrôleur ayant le nom `welcome.action` pour les URL de la forme suivante :

```
http://<machine>:<port>/<alias-webapp>/welcome.action
```

Spring MVC fournit également l'implémentation `SimpleUrlHandlerMapping`, plus flexible, qui permet d'associer des Beans à des requêtes sans utiliser leur attribut `name`. Il définit le contrôleur correspondant à un ou plusieurs URI par l'intermédiaire d'une expression régulière spécifiée dans la propriété `mapping`. Le code suivant montre l'adaptation de la configuration précédente à cette implémentation :

```
<beans>
  <bean id="handlerMapping" class="org.springframework.web
                        .servlet.handler.SimpleUrlHandlerMapping">
```

```
    <property name="mappings">
      <props>
        <prop key="/welcome.action">welcomeController</prop>
      </props>
    </property>
  </bean>

  <bean id="welcomeController" class="tudu.web.WelcomeController"/>
<beans>
```

Par l'intermédiaire de cette stratégie, le lien avec les contrôleurs se réalise par nom et non par référence.

Interception des requêtes

Spring MVC offre la possibilité de réaliser des interceptions au niveau du traitement des requêtes. Ces interceptions utilisent des mécanismes spécifiques du framework, qui, sans relever des concepts de la POA, permettent d'exécuter des traitements avant et après l'exécution du contrôleur associé à la requête ainsi qu'à la fin de la construction de la vue.

Un intercepteur de Spring MVC doit implémenter l'interface suivante `HandlerInterceptor` du package `org.springframework.web.servlet` ou étendre la classe `HandlerInterceptorAdapter` afin de ne pas avoir à redéfinir les méthodes non utilisées :

```
public interface HandlerInterceptor {
    boolean preHandle(HttpServletRequest request,
                      HttpServletResponse response,
                      Object handler) throws Exception;

    void postHandle(HttpServletRequest request,
                    HttpServletResponse response,
                    Object handler,
                    ModelAndView modelAndView) throws Exception;

    void afterCompletion(HttpServletRequest request,
                         HttpServletResponse response,
                         Object handler,
                         Exception ex) throws Exception;
```

Le code suivant montre une implémentation d'un intercepteur de mesure des temps de réponse des contrôleurs fondé sur le framework JAMon :

```
public class JAMonInterceptor extends HandlerInterceptorAdapter {

    public boolean preHandle(HttpServletRequest request,
                HttpServletResponse response,
                Object handler) throws Exception {
        Monitor monitor =
               MonitorFactory.start(handler.getClass().getName());
```

```
        request.setAttribute("monitor",monitor);
        return true;
    }

    public void postHandle(HttpServletRequest request,
                HttpServletResponse response, Object handler,
                ModelAndView modelAndView) throws Exception {
        Monitor monitor = (Monitor)request.getAttribute("monitor");
        monitor.stop();
        request.removeAttribute("monitor");
    }
}
```

Dans le cadre de l'utilisation de la classe `SimpleUrlHandlerMapping`, l'intercepteur se configure grâce à la propriété `interceptors`, comme ci-dessous :

```
<beans>
    <bean id="handlerMapping" class="org.springframework
                    .web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="interceptors">
            <list>
                <ref bean="jamonInterceptor"/>
            </list>
        </property>
        <property name="mappings">
            <props>
                <prop key="/welcome.action">
                    welcomeController
                </prop>
                (...)
            </props>
        </property>
    </bean>

    <bean id="jamonInterceptor" class="samples.JAMonInterceptor"/>
<beans>
```

Un intercepteur arrête la chaîne d'exécution des traitements si la valeur `false` est retournée par la méthode `preHandle`.

Les types de contrôleurs

Spring MVC fournit une interface de base pour les contrôleurs que le développeur est libre d'implémenter directement. Il peut également utiliser les implémentations des contrôleurs fournis par le framework qui adressent des problématiques spécifiques en mettant souvent en œuvre un cycle de traitement des requêtes.

Les sections qui suivent détaillent l'interface de base des contrôleurs ainsi que ses principales implémentations fournies par Spring MVC.

Contrôleur simple

L'abstraction de base est l'interface `Controller`, située dans le package `org.springframework.web.mvc`, qui définit la méthode `handleRequest` comme point d'entrée :

```
public interface Controller {
    ModelAndView handleRequest(HttpServletRequest request,
                    HttpServletResponse response) throws Exception;
}
```

Ce contrôleur de base convient bien pour mettre en œuvre un traitement spécifique sans gestion de formulaire. Le code suivant donne un exemple d'utilisation de cette interface issu de Tudu Lists :

```
public class ShowTodosController implements Controller {
    (...)
    ModelAndView handleRequest(HttpServletRequest request,
               HttpServletResponse response) throws Exception {
        Collection<TodoList> todoLists = new TreeSet<TodoList>(
                   userManager.getCurrentUser().getTodoLists());
        Map<String,Object> model=new HashMap<String,Object>();
        model.put("defaultList", listId);
        return new ModelAndView("todos",model);
    }
}
```

Il se configure comme un Bean dans le contexte de Spring MVC et doit être référencé par l'implémentation de mappage choisie, comme ci-dessous :

```
<beans>
    <bean id="showTodosController"
          class="tudu.web.ShowTodosController"/>

    <bean id="handlerMapping" class="org.springframework
                    .web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/showTodos.action">
                    showTodosController
                </prop>
                (...)
            </props>
        </property>
    </bean>
</beans>
```

Contrôleur à entrée multiple

Spring MVC fournit également dans le package `org.springframework.web.mvc.multiaction` l'implémentation `MultiActionController`, qui permet de mettre en œuvre des contrôleurs avec plusieurs points d'entrée. Les méthodes de ces contrôleurs doivent répondre à des

critères stricts de signature et posséder la même signature que la méthode `handleRequest` au nom près.

> **Point d'entrée**
>
> Dans le contexte du pattern MVC, un point d'entrée correspond à une méthode d'un composant qui peut être appelé par le conteneur ou le framework qui le gère. Cette méthode suit habituellement des conventions spécifiques quant à sa signature.

L'une ou l'autre des deux approches suivantes sont envisageables pour mettre en œuvre cette implémentation :

- Étendre la classe `MultiActionController`, qui contient dès lors la logique permettant de sélectionner la méthode du contrôleur à exécuter. Les méthodes possibles doivent avoir une signature normalisée au niveau des paramètres d'entrée et de retour.
- Définir une classe indépendante contenant les méthodes des points d'entrée auxquelles la classe `MultiActionController` délègue les traitements. L'implémentation n'a aucune adhérence avec Spring MVC, mais les signatures des méthodes doivent être normalisées, comme précédemment.

L'exemple suivant illustre l'adaptation du code du contrôleur précédent afin de supporter plusieurs points d'entrée :

```
public class ShowTodosController extends MultiActionController {
    (...)
    ModelAndView showTodos(HttpServletRequest request,
                HttpServletResponse response) throws Exception {
        Collection<TodoList> todoLists = new TreeSet<TodoList>(
                    userManager.getCurrentUser().getTodoLists());
        Map<String,Object> model=new HashMap<String,Object>();
        model.put("defaultList", listId);
        return new ModelAndView("todos",model);
    }
}
```

Une fois le contrôleur implémenté, la stratégie d'appel doit être spécifiée afin de faire correspondre la requête avec une méthode du contrôleur. Spring MVC fournit différentes stratégies pour cela, configurables par le biais de la propriété `methodNameResolver` du contrôleur de type `MethodNameResolver`.

Les approches décrites dans la suite de cette section correspondent à différentes implémentations de l'interface `MethodNameResolver` et sont localisées dans le package `org.springframework.web.servlet.mvc.multiaction`.

Le code suivant décrit cette interface :

```
public interface MethodNameResolver {
    String getHandlerMethodName(HttpServletRequest request);
}
```

La classe `ParameterMethodNameResolver`, symbolisant la première implémentation, utilise un paramètre de la requête devant être présent dans l'URL de cette dernière ou dans un formulaire HTML. Cette approche se configure de la manière suivante :

```
<bean id="paramResolver" class="org.springframework.web.servlet
                    .mvc.multiaction.ParameterMethodNameResolver">
  <property name="paramName"><value>method</value></property>
</bean>

<bean id="showTodosController"
      class="tudu.web.ShowTodosController">
  <property name="methodNameResolver" ref="paramResolver"/>
</bean>
```

Seconde implémentation et stratégie par défaut de la classe `MultiActionController`, la classe `InternalPathMethodNameResolver` travaille directement sur l'URI de la requête. Cette implémentation est la stratégie par défaut de `MultiActionController`, et aucune configuration spécifique n'est nécessaire.

La classe `PropertiesMethodNameResolver` est la dernière implémentation. Elle utilise une propriété de type `Properties` pour définir la correspondance entre l'URI des requêtes et un nom de méthode du contrôleur. Elle se configure de la manière suivante :

```
<bean id="paramResolver" class="org.springframework.web.servlet
                  .mvc.multiaction.PropertiesMethodNameResolver">
  <property name="mappings">
    <props>
      <prop key="/secure/showTodos.action">showTodos</prop>
    </props>
  </property>
</bean>

<bean id="showTodosController"
      class="tudu.web.ShowTodosController">
  <property name="methodNameResolver" ref="paramResolver"/>
</bean>
```

Contrôleur de gestion de formulaire

Spring MVC fournit un support pour l'affichage des données des formulaires et leur soumission au travers de l'implémentation `SimpleFormController`.

Ce type de contrôleur utilise un objet désigné par Spring MVC par le terme `Command`, qui est configuré au travers des propriétés `commandName` et `commandClass`. Cet objet correspond au modèle du pattern MVC. Il est implémenté grâce à un simple Bean Java, totalement indépendant du framework, ce qui lui permet d'être utilisé par les composants des couches inférieures.

Voici la configuration du Bean de type `RegisterData` pour le contrôleur `RegisterController` :

```
<bean id="registerController" class="tudu.web.RegisterController">
  <property name="commandName" value="register"/>
  <property name="commandClass"
            value="tudu.web.bean.RegisterData"/>
  (...)
</bean>
```

La section suivante se penche sur la façon dont l'implémentation `SimpleFormController` gère les formulaires HTML.

Affichage du formulaire

L'implémentation `SimpleFormController` offre la possibilité de charger les différentes entités nécessaires à l'affichage du formulaire dans la vue. Elle utilise pour cela différentes méthodes de rappel que le développeur peut surcharger suivant ses besoins.

L'affichage du formulaire est réalisé grâce à l'appel du contrôleur par la méthode GET. Le cycle d'enchaînement des méthodes est illustré à la figure 7.3.

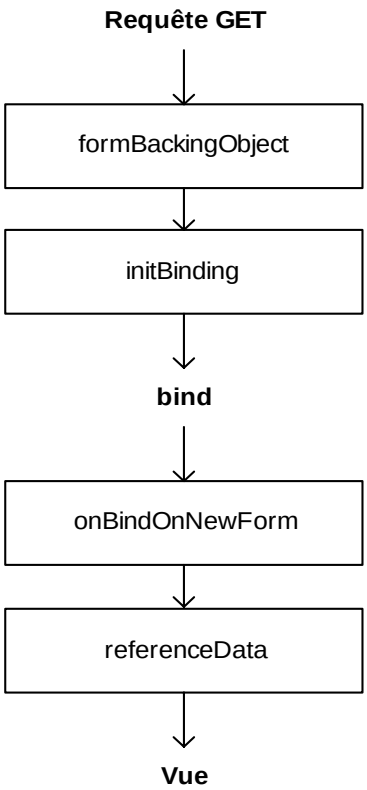


Figure 7.3
Enchaînement des méthodes permettant l'affichage des données d'un formulaire

Spring MVC utilise d'abord la méthode `formBackingObject` dans le but de récupérer une instance de l'objet sur laquelle le formulaire va se fonder. Son comportement par défaut crée une nouvelle instance à chaque demande d'affichage du formulaire, laquelle peut être surchargée afin de renvoyer une instance initialisée avec des valeurs de la base de données.

Le code suivant, tiré de Tudu Lists, donne un exemple de surcharge de la méthode `formBackingObject` :

```
public class MyInfoController extends SimpleFormController {
    (...)
    protected Object formBackingObject(HttpServletRequest request)
                                        throws ServletException {
        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 ajoutés grâce à la méthode `initBinder` permettent de convertir les propriétés du Bean de formulaire en chaînes de caractères affichables dans des champs.

Le code suivant indique la façon d'ajouter un `PropertyEditor` dans un contrôleur :

```
public class MyInfoController extends SimpleFormController {
    (...)
    protected void initBinder(HttpServletRequest request,
                ServletRequestDataBinder binder) throws Exception {
        binder.registerCustomEditor(MyClass.class,
                                    new MyPropertyEditor());
    }
    (...)
}
```

La méthode `onBindOnNewForm` permet de positionner des valeurs sur le Bean de formulaire une fois celui-ci rempli automatiquement avec des données passées dans l'URI.

Pour finir, la méthode `referenceData` offre la possibilité d'ajouter au modèle des données nécessaires à la construction du formulaire, comme les valeurs d'une liste de sélection.

Le code suivant montre la façon de surcharger cette méthode :

```
public class MyInfoController extends SimpleFormController {
    (...)
    protected Map referenceData(HttpServletRequest request)
                                                throws Exception {
```

```
        Map model=new HashMap();
        model.put("data","my data");
        return model;
    }
    (...)
}
```

Soumission du formulaire

L'implémentation `SimpleFormController` permet également de traiter les données soumises par un formulaire grâce à différentes méthodes de rappel, que le développeur peut surcharger comme précédemment.

Avec Spring MVC, la soumission d'un formulaire doit être réalisée par l'appel du contrôleur avec la méthode POST. Le cycle d'enchaînement des méthodes correspondant est illustré à la figure 7.4.

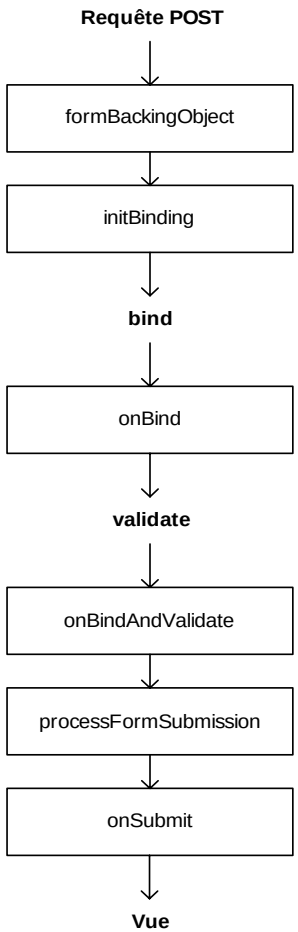


Figure 7.4
Enchaînement des méthodes permettant la soumission des données d'un formulaire

Les premières méthodes (`formBackingObject`, `initBinder`) fonctionnent de la même manière que précédemment, et la méthode `onBind` comme `onBindOnNewForm`. Cette dernière est particulièrement utile afin de positionner une propriété de type complexe en s'appuyant sur des données annexes.

Une validation des données d'un formulaire peut être mise en œuvre si nécessaire. Spring MVC introduit à cet effet l'interface `Validator`, dont le code 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 par l'intermédiaire de sa propriété `validator`, comme dans le code ci-dessous :

```
<bean id="registerController" class="tudu.web.RegisterController">
  (...)
  <property name="validator" value="register"/>
  (...)
</bean>

<bean id="registerValidator"
      class="tudu.web.validator.RegisterValidator"/>
```

Le code suivant de la classe `RegisterValidator` montre que l'implémentation du validateur permet de spécifier des erreurs aussi bien à un niveau global (repère ❷) que sur chaque propriété du formulaire (repère ❶) 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"}, "");← 2
        }

        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "firstName",
               "errors.required", new Object[] {"firstName"}, "");← 2
         ValidationUtils.rejectIfEmptyOrWhitespace(
                errors, "lastName",
                "errors.required", new Object[] {"lastName"}, "");← 2

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

Pour finir, la méthode `processFormSubmission` permet de traiter les données soumises et de les réafficher si des erreurs de conversion ou de validation se sont produites.

En cas de succès, les traitements sont délégués à la méthode `onSubmit` à surcharger, comme le montre le code suivant de la classe `MyInfoController` issue de l'étude de cas :

```
public class MyInfoController extends SimpleFormController {
    (...)
    protected ModelAndView onSubmit(HttpServletRequest request,
                  HttpServletResponse response, Object command,
                  BindException errors) throws Exception {
        UserInfoData data = (UserInfoData) command;
        String password = data.getPassword();
        String firstName = data.getFirstName();
        String lastName = data.getLastName();
        String email = data.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 showForm(request,response,errors);
    }
    (...)
}
```

Lors de l'utilisation d'une vue fondée sur JSP/JSTL, la balise `<bind>` de Spring offre un support à l'affichage des données du formulaire ainsi des erreurs de validation. Son utilisation est détaillée plus loin dans ce chapitre.

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 `SimpleMappingExceptionResolver` 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>
```

Spring MVC et la gestion de la vue

Cette section se penche sur la façon dont sont traitées les vues.

Nous commençons par détailler la manière dont sont mis en correspondance les identifiants de ces vues et leurs implémentations. Nous abordons ensuite les différents supports des technologies de présentation.

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.

Spring MVC fournit pour cela la classe `ModelAndView` dans le package `org.springframework.web.servlet`, dont une instance est renvoyée par la méthode `handleRequest` des

contrôleurs. 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.

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, dans lequel l'identifiant `todos` correspond à un nom symbolique de vue configuré dans Spring MVC :

```
Collection<TodoList> todoLists = new TreeSet<TodoList>(
                    userManager.getCurrentUser().getTodoLists());
Map<String,Object> model=new HashMap<String,Object>();
model.put("defaultList", listId);
ModelAndView model=new ModelAndView("todos",model);
```

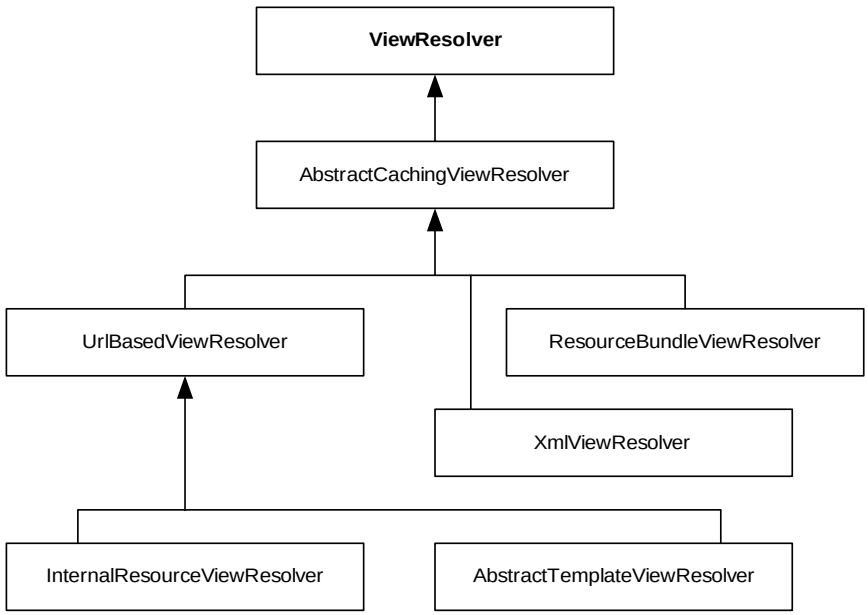
Configuration de la vue

La sélection des vues dans Spring MVC est réalisée 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 7.5 illustre la hiérarchie de ces classes et interfaces.

Figure 7.5
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 de 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 cependant 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 facilités 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="localtion" value="/WEB-INF/views.xml" />
</bean>
```

Le code suivant donne un exemple de fichier **views.xml** tiré de Tudu Lists :

```
<beans>
    <bean id="todo_lists_report"
          class="tudu.web.ShowTodoListsPdfView"/>

    <bean id="backup" class="tudu.web.BackupTodoListView">
        <property name="stylesheetLocation"
                  value="/WEB-INF/xsl/backup.xsl"/>
        <property name="todoListsManager" ref="todoListsManager"/>
    </bean>
</beans>
```

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, exceptées celles qui sont résolues précédemment par une autre implémentation dans une chaîne de `ViewResolver`.

Chaînage d'implémentations de *ViewResolver*

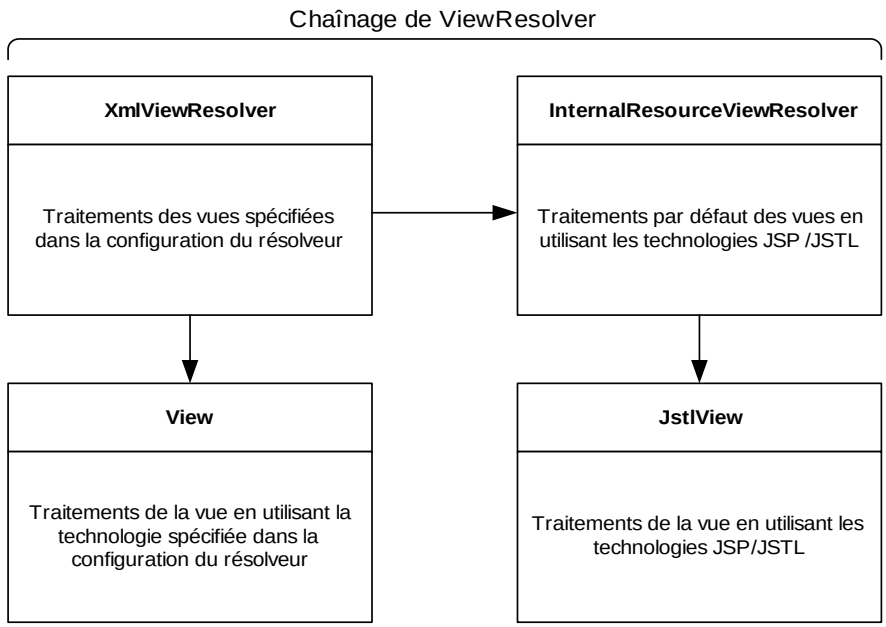
Spring MVC offre la possibilité de chaîner les entités de résolution des vues. Le framework parcourt en ce cas la chaîne jusqu'à la découverte du `ViewResolver` approprié.

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 `UrlBasedViewResolver`, 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 7.6 illustre un exemple de chaînage d'implémentations de l'interface de `ViewResolver` tiré de Tudu Lists.

Figure 7.6
Chaînage de ViewResolver *dans 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.

Les technologies de présentation

Spring MVC offre plusieurs fonctionnalités qui simplifient énormément la mise en œuvre des différentes 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 7.7 illustre la hiérarchie de ses classes et interfaces.

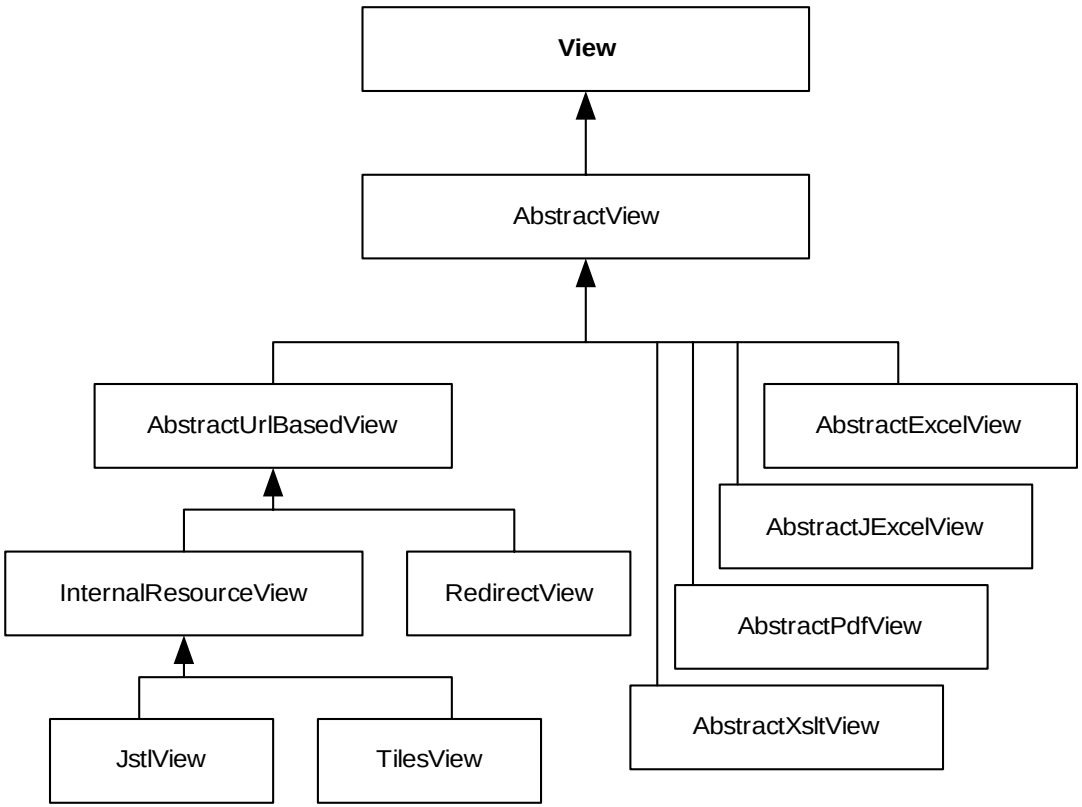


Figure 7.7
Hiérarchie des implémentations de l'interface View

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 7.8 illustre l'enchaînement des traitements afin d'utiliser une vue de type `RedirectView`.

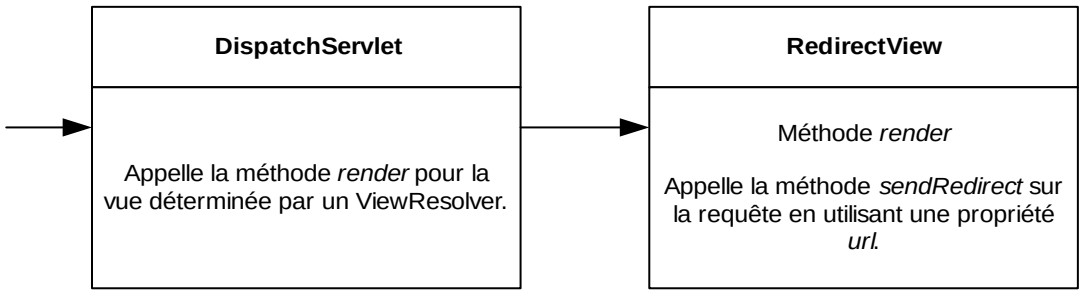


Figure 7.8
Enchaînement des traitements pour la vue

Cette vue peut être configurée plus rapidement et directement dans la configuration des contrôleurs grâce au préfixe `redirect` :

```
<bean id="restoreTodoListController"
                        class="tudu.web.RestoreTodoListController">
    <property name="todoListsManager" ref="todoListsManager" />
    <property name="formView" value="restore"/>
    <property name="successView"
              value="redirect:showTodos.action"/>
    <property name="commandName" value="restore"/>
    <property name="commandClass"
              value="tudu.web.bean.RestoreData"/>
</bean>
```

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 7.9 illustre l'enchaînement des traitements afin d'utiliser une vue de type `JstlView`.

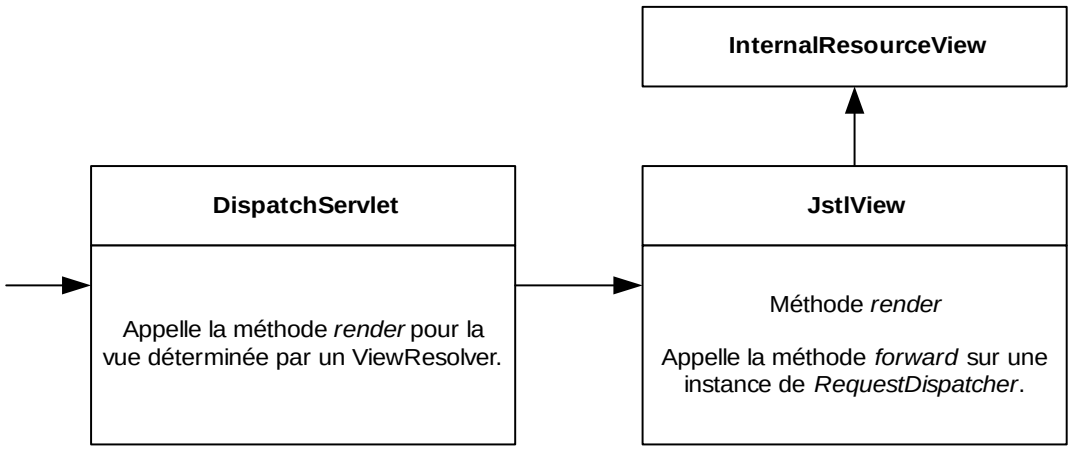


Figure 7.9
Enchaînement des traitements pour la vue

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.

Lors de la mise en œuvre d'un contrôleur fondé sur l'implémentation `SimpleFormController`, Spring MVC stocke dans la requête les deux instances suivantes, qui sont utilisables dans la page JSP :

- `<commandName>`. Cette clé correspond au Bean contenant les données initiales du formulaire, une instance de cet élément étant récupérée grâce à la méthode `formBackingObject` du contrôleur.
- `org.springframework.validation.BindException. <commandName>`. Cette clé renvoie aux éventuelles erreurs de chargement ou de validation du Bean de formulaire. Cet élément est initialisé avec les données envoyées lors de la soumission du formulaire.

La valeur de `<commandName>` est définie dans la configuration du contrôleur à l'aide de la propriété `commandName`.

Spring MVC fournit une balise `bind` permettant d'initialiser les champs 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 :

```
(...)
<tr class="odd">
```

```
 <td>
  <fmt:message key="user.info.first.name"/>
 </td>
 <td>
  <spring:bind path="userinfo.firstName">← ❶
   <input type="text" name="firstName"
          value="<c:out value="${status.value}"/>"← ❷
          size="15" maxlength="60"/>
    <font color="red">
   <c:out value="${status.errorMessage}"/></font>← ❸
  </spring:bind>← ❹
 </td>
</tr>
(...)
```

La balise `bind` permet de définir un contexte quant au code imbriqué par l'intermédiaire de la variable `status` donnant accès aux informations de l'entité. Dans notre exemple, sa portée est délimitée par les repères ❶ et ❹.

L'entité cible définie dans l'attribut `path` de la balise peut correspondre aussi bien au Bean de données lui-même qu'à une de ses propriétés. La balise `bind` porte sur la propriété `firstName` du Bean de formulaire ayant pour nom `userinfo`.

Le contexte permet de récupérer la valeur et un message d'erreur associé éventuel. La valeur n'est renseignée que si la balise `bind` porte sur un champ.

Les balises et les expressions JSTL peuvent être utilisées comme dans l'exemple précédent. Spring MVC met également à disposition les données présentes dans le modèle. Pour une entrée ayant pour clé `maVariable` dans le modèle, la JSP récupère la valeur de sa propriété `maPropriete` correspondante de la manière suivante :

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

Afin d'utiliser les taglibs JSTL et de Spring, 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" %>
<%@ taglib prefix="spring"
           uri="http://www.springframework.org/tags" %>
```

Autres vues

Spring MVC apporte également des supports pour toutes sortes de vues qui ne sont pas nécessairement fondées sur des forwards ou des redirections de requêtes.

Le framework offre ainsi des classes abstraites de base pour les différentes technologies suivantes :

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

- génération de la présentation fondée sur des templates (Velocity, FreeMaker) ;
- génération de la présentation fondée 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 7.10 illustre l'enchaînement des traitements d'une vue générant un document avec le support PDF de Spring MVC.

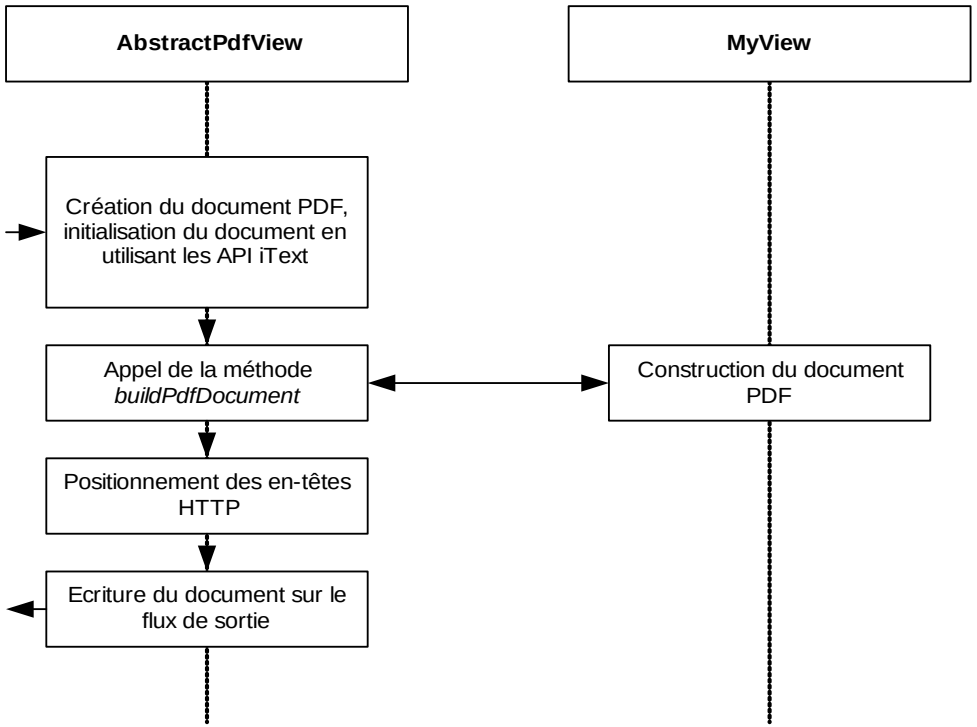


Figure 7.10
Enchaînement des traitements de la vue

Dans l'exemple décrit à la figure 7.10, 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

Dans cette section, nous avons approfondi les principes de gestion de la présentation dans Spring MVC.

Ce framework met en œuvre des mécanismes de mise en relation d'un identifiant et de sa vue correspondante, 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 section suivante détaille la mise en œuvre de Spring MVC dans l'étude de cas Tudu Lists.

Tudu Lists : utilisation de Spring MVC

Maintenant que les concepts et composants mis en œuvre dans Spring MVC sont éclaircis, nous pouvons passer à la description de la façon dont nous le mettons en pratique dans notre étude de cas.

Pour Spring MVC, nous avons créé un projet, `Tudu-SpringMVC`, indépendant de celui fondé sur le framework MVC Struts, dont la migration est détaillée dans cette section. Ce projet étant aussi utilisé au chapitre 13, qui nécessite un fournisseur JMS actif, il faut exécuter le script **build.xml,** comme indiqué au chapitre 1, avant de lancer l'application Web.

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. Il est chargé par la servlet `DispatchServlet` en utilisant le fichier **WEB-INF/action-servlet.xml.** Les différentes entités relatives à Spring MVC sont définies dans ce fichier.

La migration a donc un impact sur le fichier **web.xml** puisque la servlet configurée est remplacée :

```
<?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 cadre de la migration, les fichiers du contexte principal restent inchangés.

Commons Validator

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

Commons Validator se configure de la manière suivante :

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

Dans chaque contrôleur de gestion de formulaire, la propriété `validator` doit être ensuite positionnée avec le Bean `beanValidator`. Ce dernier utilise la valeur de la propriété `commandClass` afin de déterminer le nom du bloc `form` dans la configuration de la validation **WEB-INF/validation.xml.** La reprise des règles de validation devient de la sorte possible.

Implémentation des contrôleurs

Au travers de Struts, les contrôleurs sont implémentés par le biais de la classe `org.apache.struts.action.Action`. Plusieurs types d'actions permettent de résoudre diverses problématiques.

Le tableau 7.1 fournit la correspondance entre les actions de Struts et les contrôleurs de Spring MVC suivant la problématique à résoudre.

Tableau 7.1. Correspondance entre les actions Struts et les contrôleurs Spring MVC

Problématique	Action Struts	Contrôleur Spring MVC
Contrôleur simple	Action	Controller
Contrôleur multiple	DispatchAction	MultiActionController
Contrôleur avec gestion de formulaire	`Action` ou `DispatchAction` avec une instance d'`ActionForm`	`SimpleFormController` avec un Bean indépendant

La section suivante se penche sur la façon de migrer deux actions réalisant respectivement un affichage de données et une gestion de formulaire. La présentation associée à ces deux contrôleurs utilise les technologies JSP/JSTL.

Contrôleur simple

L'adaptation de l'action `ShowTodosAction` du package `tudu.web` permet d'afficher les todos d'une liste. La migration consiste à réaliser les modifications suivantes *(voir le code ci-après) :*

- Implémentation de l'interface `Controller` du package `org.springframework.web.servlet.mvc` (repère ❶). Comme nous l'avons vu précédemment, les traitements présents dans la méthode `execute` se trouveront désormais dans la méthode `handleRequest` (repère ❷).
- Modification de la façon d'ajouter les éléments dans le modèle (repères ❸ et ❹). Contrairement à Struts, Spring MVC les stocke dans une table de hachage et n'utilise pas les API servlet.
- Modification de la façon de spécifier la vue à utiliser. Spring MVC nécessite uniquement l'utilisation d'une instance de la classe `ModelAndView` contenant l'identifiant de la vue et le modèle (repère ❺). La vue est configurée avec l'implémentation du `ViewResolver`.
- Configuration du contrôleur et de la vue utilisée.

```
public class ShowTodosController implements Controller {←❶
    (...)
    public ModelAndView handleRequest(HttpServletRequest request,
              HttpServletResponse response) throws Exception {←❷

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

        Map<String,Object> model=new HashMap<String,Object>();←❸
        if (!todoLists.isEmpty()) {
            String listId =
               RequestUtils.getStringParameter(request,"listId");
            if (listId != null) {
               model.put("defaultList", listId);←❹
            } else {
               model.put("defaultList",
                     todoLists.iterator().next().getListId());←❹
            }
        }
        return new ModelAndView("todos",model);←❺
    }
}
```

Le contrôleur se configure dans le fichier **WEB-INF/action-servlet.xml.** Avec Spring MVC, les configurations des propriétés relatives au MVC et la résolution des dépendances sont réalisées dans ce fichier :

```
<bean id="showTodosController"
      class="tudu.web.ShowTodosController">
    <property name="userManager" ref="userManager" />
</bean>
```

La manière d'accéder au contrôleur est définie dans le même fichier en ajoutant une entrée dans la propriété `mapping` de la configuration de la stratégie de mappage. Afin de conserver les mêmes URI pour les différents traitements, nous avons choisi l'implémentation `SimpleUrlHandlerMapping`, qui minimise l'impact sur les vues.

Pour le contrôleur `showTodosController`, l'entrée `/secure/showTodos.action` doit être ajoutée :

```
<bean id="handlerMapping" class="org.springframework.web
                        .servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            (...)
            <prop key="/secure/showTodos.action">
                showTodosController
            </prop>
            (...)
        </props>
    </property>
</bean>
```

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 7.11 illustre cette chaîne en soulignant les identifiants des vues configurées dans les deux premiers `ViewResolver`.

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/.** Dans le cas d'une migration vers Spring MVC, aucune modification n'est nécessaire dans cette page.

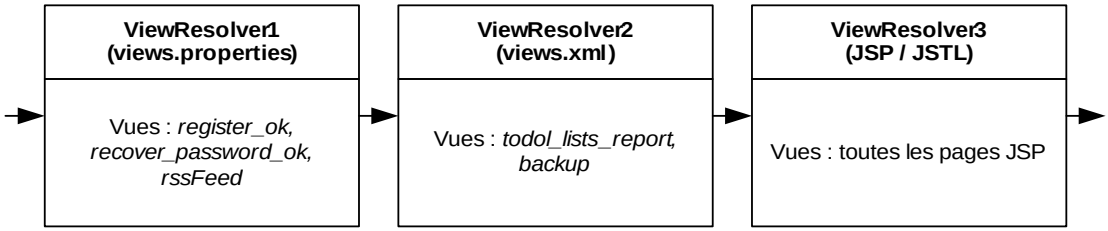


Figure 7.11
Chaînage des ViewResolver *dans Tudu Lists*

Contrôleur de gestion d'un formulaire

La migration d'une entité de gestion d'un formulaire est plus complexe, les mécanismes correspondants étant opposés à ceux de Struts. Arrêtons-nous un instant sur l'adaptation de l'action `MyInfosAction` du package `tudu.web`, qui permet de modifier les informations d'un utilisateur de l'application.

La migration consiste à réaliser les modifications suivantes *(voir le code ci-après) :*

- Étendre la classe `SimpleFormController`. Avec Spring MVC, la gestion d'un formulaire ne peut être réalisée avec un contrôleur à entrée multiple tel que `MultiActionController` (repère ❶).
- Surcharger la méthode `formBackingObject` afin de charger les données du formulaire (repère ❷).
- Redéfinir la méthode `onSubmit` afin de traiter les données soumises par le formulaire (repère ❸) avec la méthode HTTP POST.
- Spécifier et configurer la validation des données soumises. La migration utilise Commons Validator ainsi que les mêmes règles de validation.
- Configurer le contrôleur ainsi que les vues utilisées.

```
public class MyInfoController extends SimpleFormController {←❶
    (...)
    protected Object formBackingObject(HttpServletRequest request)
                                    throws ServletException {←❷
        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;
    }
```

```
    protected ModelAndView onSubmit(HttpServletRequest request,
                    HttpServletResponse response, Object command,
                      BindException errors) throws Exception {←❸
        UserInfoData data = (UserInfoData) command;
        String password = data.getPassword();
        String firstName = data.getFirstName();
        String lastName = data.getLastName();
        String email = data.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 showForm(request,response,errors);
    }
}
```

Comme pour le contrôleur précédent, le contrôleur `MyInfoController` doit être configuré dans le fichier **action-servlet.xml** de la manière suivante :

```
<bean id="myInfoController" class="tudu.web.MyInfoController">
    <property name="userManager" ref="userManager" />
    <property name="formView" value="user_info"/>
    <property name="validator" ref="userInfoValidator"/>
    <property name="commandName" value="userinfo"/>
    <property name="commandClass"
              value="tudu.web.bean.UserInfoData"/>
</bean>
```

Une entrée doit également être ajoutée au Bean `handlerMapping` afin que le contrôleur puisse être accédé à partir d'un URI :

```
<bean id="handlerMapping" class="org.springframework.web
                        .servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            (...)
            <prop key="/secure/myInfo.action">
                myInfoController
            </prop>
            (...)
        </props>
    </property>
</bean>
```

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'échecs 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.

Le framework met en œuvre le taglib `bind` afin de remplir les champs du formulaire et d'afficher les éventuelles erreurs de validation, comme dans le code suivant :

```
(...)
<spring:bind path="userinfo">
 <font color="red">
   <b><c:out value="${status.value}"/></b>
   <b><c:out value="${status.errorMessage}"/></b>
 </font>
</spring:bind>

<form action="<c:url value="/secure/myInfo.action"/>"
                          method="POST" focus="firstName">
(...)
  <spring:bind path="userinfo.firstName">
    <input type="text" name="firstName"
           value="<c:out value="${status.value}"/>"
           size="15" maxlength="60"/>
     <font color="red">
      <c:out value="${status.errorMessage}"/>
    </font>
  </spring:bind>
(...)
  <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 le `ViewResolver` utilisé précédemment pour les vues utilise les technologies JSP et JSTL, il ne peut servir pour les vues décrites ci-dessous. Dans les sections suivantes, nous utilisons 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 totos. Cette fonctionnalité est implémentée par l'intermédiaire du contrôleur `BackupTodoListController` 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 à la vue ayant pour identifiant `backup` à la fin de ses traitements. Cette dernière, 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 le montre le code suivant :

```
public class BackupTodoListView extends AbstractXsltView {
(...)
    protected Source createXsltSource(
        Map model, String root, HttpServletRequest request,
        HttpServletResponse response) throws Exception {

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

À la fin de l'exécution de la méthode `createXsltSource`, la classe `BackupTodoListView` rend la main à sa classe mère afin de réaliser 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 et de spécifier le fichier **backup.xsl** en tant que feuille de style XSLT, comme décrit dans le fichier **views.xml** suivant :

```
<bean id="backup" class="tudu.web.BackupTodoListView">
    <property name="stylesheetLocation"
              value="/WEB-INF/xsl/backup.xsl"/>
    <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 le biais de la classe `ShowTodoLists-PdfView` 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 le montre 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.

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

Conclusion

Spring et son framework MVC ont su tirer parti des limitations d'autres frameworks MVC tels que Struts. La modularité, l'isolation des différents composants du pattern MVC ainsi que l'ouverture du framework sont les objectifs visés. Dès lors, le développement des applications utilisant différentes technologies est facilité, de même que le changement de briques sans impact sur le reste de l'application et l'extension du framework.

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

Spring MVC propose en outre des implémentations robustes de contrôleurs visant à résoudre les problématiques techniques des applications Web. Dans le cadre de la gestion des formulaires, le framework affiche une grande flexibilité d'utilisation et de validation des données du formulaire.

Le framework Spring MVC offre 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 toutefois 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.

8

Spring Web Flow

Certaines applications Web nécessitent un enchaînement défini d'écrans afin de réaliser leurs traitements. Elles doivent, de plus, empêcher l'utilisateur de s'écarter des enchaînements possibles et garder un contexte relatif à l'exécution. Lorsque notre application répond à ces critères, la mise en œuvre d'outils tels que Spring Web Flow est pertinente. Ce type de framework ne présente en revanche aucun intérêt dans le cas d'applications pour lesquelles la navigation est libre.

Le framework Spring Web Flow se positionne en concurrent direct de la fonctionnalité Page Flow du framework Beehive d'Apache, qui vise également à résoudre cette problématique. Les mises en œuvre diffèrent cependant puisque le choix de Beehive porte sur l'utilisation massive des annotations alors que Spring Web Flow s'appuie sur des fichiers de configuration XML ou une API de définition des flots.

Concepts des Web Flows

L'objectif des Web Flows est d'implémenter le traitement de flots Web qui comprennent plusieurs étapes et diverses contraintes pour passer d'une page à une autre.

Dans le cas d'applications J2EE, ces étapes se matérialisent par des interactions avec le serveur J2EE.

La conception et la mise en œuvre d'applications pour lesquelles la navigation se trouve restreinte et prédéfinie par des règles précises se révèlent particulièrement complexes, et ce pour les raisons suivantes :

- La configuration des enchaînements de traitements est difficile.
- La vérification de la validité des enchaînements est complexe à mettre en œuvre.

- La session HTTP n'est pas entièrement adaptée pour stocker les données d'un flot de traitements.
- La réutilisation des différents flots Web est complexe à implémenter.

Ces constats mettent en exergue les enjeux suivants, que l'architecture et la conception de ce type d'application doivent résoudre de manière optimale :

- découplage des flots et de leur implémentation ;
- gestion des données des états du flot de traitements ;
- respect des contraintes du flot par une entité autonome ;
- favorisation de la réutilisation des flots Web ;
- intégration des flots Web dans l'environnement technique existant.

Pour résoudre ces problématiques et adresser au mieux les enjeux précédemment décrits, l'approche adéquate consiste à intégrer un moteur d'exécution de flots Web utilisant une description des flots.

L'utilisation de ce type de moteur dans une application offre les avantages suivants :

- définition centralisée des éléments du flot et de leurs enchaînements ;
- configuration du flot fondée sur une grammaire XML dédiée, lisible aussi bien par un concepteur, un développeur ou un environnement de développement (IDE) ;
- notion de transitions mise en œuvre dans la configuration du flot afin de cadrer et sécuriser la navigation ;
- gestion systématique par le moteur des états en interne ;
- suppression des dépendances avec la technologie sous-jacente et le protocole utilisé ;
- configuration des flots facilement intégrable dans un IDE ;
- gestion du cycle de vie du flot facilitée et intégrée au moteur sans aucun impact sur les flots Web.

Détaillons maintenant plus en détail les problématiques liées aux flots et automates à états finis ainsi que leurs diverses composantes.

Définition d'un flot Web

Un flot Web s'inscrit dans la problématique générale des AEF (automate à état fini). Ces automates permettent de spécifier les différents états accessibles des flots ainsi que la manière de passer des uns aux autres. Ils peuvent être facilement décrits en UML dans des diagrammes d'activité avec des entités stéréotypées, comme l'illustre la figure 8.1.

Un flot est donc caractérisé par un ensemble fini d'états possibles, dont certains peuvent être initiaux et d'autres finaux. L'exécution ne peut se poursuivre que par le biais de transitions d'un état vers une liste d'états connue et finie.

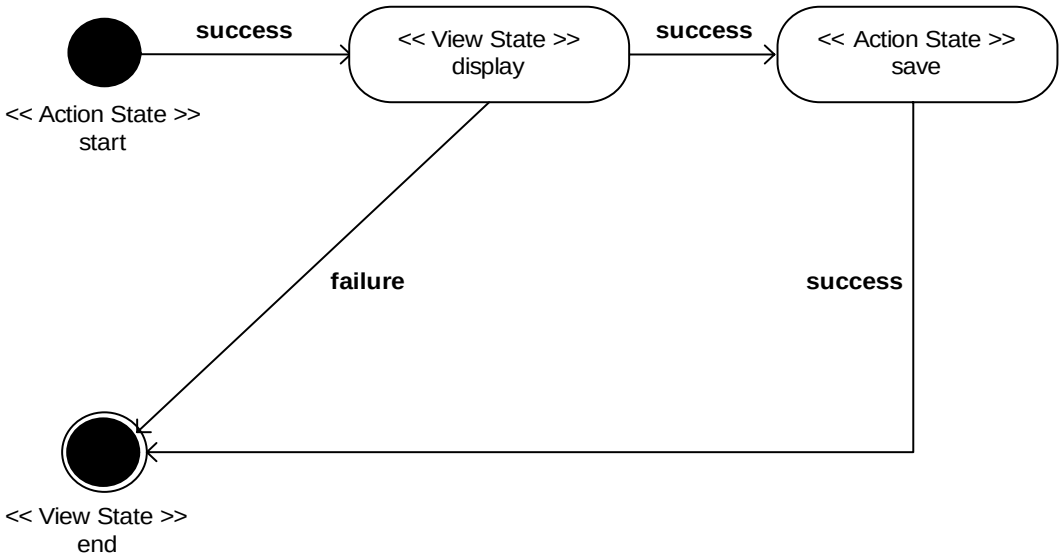
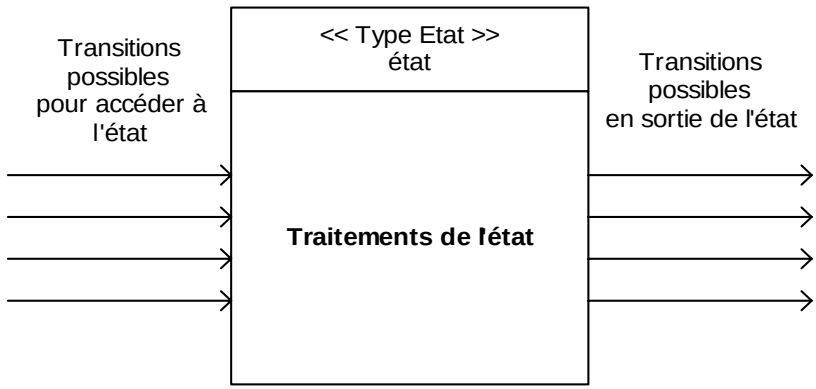


Figure 8.1
Diagramme UML d'un flot Web

Les états initiaux et finaux exceptés, un état est accessible depuis un ensemble d'états suite au déclenchement de différents événements et peut transiter vers d'autres états suite au déclenchement d'autres événements.

La figure 8.2 illustre les différentes composantes d'un état.

Figure 8.2
Composantes d'un état



Le passage d'un état à un autre s'effectue par le biais d'une transition, qui décrit l'état à accéder suite au déclenchement d'un événement sur un autre état. Propriété d'un état, une transition regroupe l'événement déclencheur ainsi que l'état à exécuter dans ce cas.

Les types d'états

Au sein d'un flot d'exécution, plusieurs types d'états peuvent être mis en œuvre afin d'adresser différentes problématiques :

- **Début et fin.** Les états de début et de fin d'un flot correspondent à des états particuliers. Un état de début ne peut avoir d'état l'appelant, et un état de fin ne doit pas posséder de transition en sortie. Ces types d'états peuvent aussi bien être des états d'affichage de vue que de déclenchement d'actions, ne possédant donc pas de stéréotype spécifique.
- **Affichage de vue.** Un état peut correspondre à l'affichage d'une vue. Il doit dans ce cas lui faire référence. Les différents événements déclenchés par la vue doivent être définis sur l'état en tant que transitions. Ce type d'état est désigné par le terme « View State ». Nous utilisons le stéréotype `View State` pour le représenter.
- **Exécution d'action.** Un état peut correspondre au déclenchement d'une méthode d'une action, quel que soit son type. Il doit en ce cas définir les différents événements à déclencher que peuvent retourner ces méthodes. Ce type d'état est désigné par le terme « Action State ». Nous utilisons le stéréotype `Action State` pour le représenter.
- **Aiguillage.** Un état peut correspondre à un aiguillage fondé sur une ou plusieurs conditions afin d'accéder à d'autres états. Ce type d'état est désigné par le terme « Decision State ». Nous utilisons le stéréotype `Decision State` pour le représenter.
- **Lancement de sous-flots d'exécution.** Un état peut déclencher l'exécution d'un sous-flot de traitement et permettre le passage de paramètres d'un flot à un autre. Ce type d'état est désigné par le terme « Sub Flow State ». Nous utilisons le stéréotype `Sub Flow State` pour le représenter.

En résumé

Nous avons décrit dans cette section les différents concepts des flots Web dont Spring Web Flow constitue une implémentation. Nous avons également abordé les notions d'état et de transition utilisées dans ces flots.

Les sections suivantes décrivent les mécanismes du framework Spring Web Flow permettant d'implémenter cette problématique.

Mise en œuvre de Spring Web Flow

Spring Web Flow n'est pas intégré pour le moment dans la distribution de Spring. Il correspond à l'un de ses sous-projets, dont les différentes versions sont téléchargeables à l'adresse suivante :

http://sourceforge.net/project/showfiles.php?group_id=73357&package_id=148517

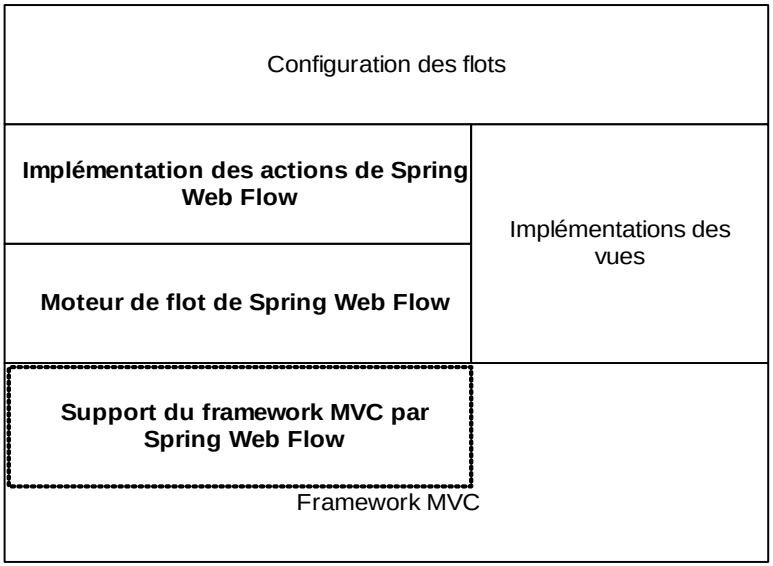
La page d'accueil du projet est quant à elle accessible par le biais du wiki du projet Spring, à l'adresse suivante :

http://opensource2.atlassian.com/confluence/spring/display/WEBFLOW/Home.

Spring Web Flow offre une implémentation afin de résoudre les problématiques décrites dans les sections précédentes. Il fournit pour cela un niveau d'abstraction ainsi que différentes entités permettant de concevoir et d'implémenter de manière optimale des flots Web.

La figure 8.3 illustre l'architecture générale de Spring Web Flow.

Figure 8.3
Architecture de Spring Web Flow



Les sections qui suivent détaillent la configuration de Spring Web Flow ainsi que la conception et l'implémentation des flots Web fondés sur ce framework.

Configuration du moteur

Spring Web Flow s'abstrait des frameworks MVC sous-jacents et fournit différents supports pour utiliser son moteur avec divers frameworks MVC.

La configuration de ce moteur diffère suivant que nous utilisions Spring MVC ou Struts.

Configuration avec Spring MVC

La configuration du moteur s'effectue en deux étapes :

1. Configuration de la servlet de Spring MVC, `DispatchServlet`, ainsi que des emplacements des configurations des divers contextes de Spring dans le fichier **web.xml,** localisé dans le répertoire **WEB-INF/** :

```
<web-app>
  <display-name>SpringWebFlow</display-name>

  <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>

  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
      org.springframework.web.servlet.DispatcherServlet
    </servlet-class>
  </servlet>

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

La configuration dans ce fichier est spécifique non de Spring Web Flow mais de Spring MVC. Elle doit être mise en œuvre afin que le support de Spring MVC par Spring Web Flow soit utilisé.

2. Configuration du moteur de Spring Web Flow lui-même. Spring Web Flow utilise un contrôleur Spring MVC dédié, implémenté par l'intermédiaire de la classe `FlowController`, localisée dans le package `org.springframework.webflow.executor.mvc`. Ce contrôleur représente le point d'entrée permettant d'utiliser le moteur. Il est rattaché à un URI par le biais d'une implémentation de l'interface `HandlerMapping` de Spring MVC.

Ce contrôleur s'appuie sur une entité d'exécution des flots utilisant un registre de stockage des définitions des flots.

L'entité d'exécution correspond à une implémentation de l'interface `FlowExecutor` du package `org.springframework.webflow.executor`. Nous utilisons l'implémentation `FlowExecutorImpl` du même package. Pour sa part, le registre correspond à l'entité de stockage des définitions des flots et est initialisé par l'intermédiaire de la classe `XmlFlowRegistryFactoryBean` du package `org.springframework.webflow.registry` dans le cas de fichiers XML de définition.

Notons qu'il est possible de spécifier plusieurs noms de fichiers XML par l'intermédiaire d'une expression régulière ou d'une liste de noms. L'identifiant du flot correspond au nom du fichier dans lequel il est défini, sans tenir compte de l'extension.

Le code suivant illustre la configuration complète du moteur dans le fichier **action-servlet.xml** localisé dans le répertoire **WEB-INF/ :**

```
<bean id="urlMapping" class="org.springframework.web.servlet
                                .handler.SimpleUrlHandlerMapping">
    <property name="mappings">
```

```
        <props>
            <prop key="/flow/todolists.html">flowController</prop>
         </props>
    </property>
</bean>

<bean name="flowController" class="org.springframework.webflow
                                    .executor.mvc.FlowController">
    <property name="flowExecutor" ref="flowExecutor"/>
    <property name="cacheSeconds" value="5"/>
</bean>

<bean id="flowExecutor" class="org.springframework.webflow
                                      .executor.FlowExecutorImpl">
    <constructor-arg>
        <bean id="repositoryFactory" class="org.springframework
                    .webflow.execution.repository.continuation
                    .ContinuationFlowExecutionRepositoryFactory">
             <constructor-arg ref="flowRegistry"/>
        </bean>
    </constructor-arg>
</bean>

<bean id="flowRegistry" class="org.springframework.webflow
                            .registry.XmlFlowRegistryFactoryBean">
    <property name="flowLocations">
        <list>
            <value>/WEB-INF/flows-*.xml</value>
         </list>
    </property>
</bean>
```

La résolution des vues du flot est dans ce cas réalisée avec les facilités de Spring MVC par le biais des implémentations de `ViewResolver` configurées dans l'application.

Configuration avec Struts

De même que pour Spring MVC, la configuration du moteur de Spring Web Flow avec le framework Struts se réalise en deux étapes :

1. Configuration de la servlet Struts `ActionServlet` et des emplacements des configurations des différents contextes de Spring dans le fichier **web.xml**, localisé dans le répertoire **WEB-INF/ :**

```
<web-app>
  <display-name>SpringWebFlow</display-name>

  <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>

  <servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>
      org.apache.struts.action.ActionServlet
    </servlet-class>
  </servlet>

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

La configuration dans ce fichier est spécifique non de Spring Web Flow mais de Struts. Elle doit être mise en œuvre afin que le support de Struts par Spring Web Flow soit utilisé.

2. Configuration du moteur de Spring Web Flow lui-même. Spring Web Flow fournit et utilise l'action de Struts `FlowAction` localisée dans le package `org.springframework.webflow.executor.struts`. Cette action doit être configurée dans le fichier **struts-config.xml** du framework localisé dans le répertoire **WEB-INF/,** comme dans le code suivant :

```
<struts-config>

    <form-beans>
        <form-bean name="actionForm" type="org.springframework
                            .web.struts.SpringBindingActionForm"/>
    </form-beans>

    <action-mappings>
        <action path="/flows-todolist"
            name="actionForm"
            scope="request"
            type="org.springframework.webflow.executor.struts.FlowAction"/>
    </action-mappings>

</struts-config>
```

Cette action Struts s'appuie sur la classe `SpringBindingActionForm` de Spring pour bénéficier des mécanismes de binding, ou mappage de données de la requête, et de validation du framework Spring.

Cette action se fonde sur le gestionnaire d'exécution des flots de Spring Web Flow et le recherche au moyen de la clé `flowExecutor` comme identifiant de Bean dans le contexte de Spring.

La configuration de cette entité dans le fichier **applicationContext.xml** est la suivante :

```
<bean id="flowExecutor" class="org.springframework.webflow
                                        .executor.FlowExecutorImpl">
    <constructor-arg>
        <bean id="repositoryFactory" class="org.springframework
                    .webflow.execution.repository.continuation
                    .ContinuationFlowExecutionRepositoryFactory">
             <constructor-arg ref="flowRegistry"/>
        </bean>
    </constructor-arg>
</bean>

<bean id="flowRegistry" class="org.springframework.webflow
                            .registry.XmlFlowRegistryFactoryBean">
    <property name="flowLocations">
        <list>
            <value>/WEB-INF/flows-*.xml</value>
         </list>
    </property>
</bean>
```

Notons que la classe `FlowAction` recherche le Bean du gestionnaire d'exécution dans le contexte de l'application Web et non dans le contexte de Spring associé à Struts.

Fichier XML de configuration de flots

Spring Web Flow offre une grammaire XML pour décrire des flots d'exécution en mode Web.

Nous détaillons ci-après les balises et attributs permettant de modéliser les différentes entités décrites précédemment.

L'identifiant du flot doit correspondre au nom du Bean qui le configure dans Spring par l'intermédiaire de la classe `XmlFlowFactoryBean`.

La balise de base du flot est `webflow`, laquelle doit posséder l'attribut `start-state` afin de définir l'état initial du flot. L'identifiant du flot est déterminé par le nom du fichier dans lequel il est défini. Il n'est donc pas nécessaire de le spécifier sur cette balise.

Le code suivant permet de définir un flot en utilisant le fichier DTD du framework, accessible depuis le site de Spring :

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

<!DOCTYPE flow PUBLIC "-//SPRING//DTD WEBFLOW 1.0//EN"
      "http://www.springframework.org/dtd/spring-webflow-1.0.dtd">

<flow start-state="start">
    (...)
</flow>
```

Un flot peut contenir les différents éléments suivants :

- attributs permettant de lui spécifier des propriétés ;
- actions exécutées au démarrage du flot ;
- éléments spécifiques aux développeurs (état fondé sur une vue, état fondé sur une action, etc.) ;
- actions exécutées à la fin du flot ;
- entités de gestion des exceptions ;
- définitions de flots.

Le code suivant illustre la mise en œuvre de tous ces éléments :

```
<flow start-state="startingPoint">
    <attribute .../>
    <start-actions>(...)</start-actions>
    <-- définitions de nos états -->
    <global-transitions>(...)</global-transitions>
    <end-actions>(...)</end-actions>
    <exception-handler .../>
    <inline-flow>(...)</inline-flow>
</flow>
```

Les sections qui suivent décrivent, selon la grammaire XML de Spring Web Flow, la configuration des différents types d'états des flots : transition, état fondé sur une action, état fondé sur une vue, état de décision, état d'exécution d'un sous-flot et état de fin de flot.

Transitions

D'une manière générale, tous les types d'états se doivent de définir les transitions possibles vers d'autres états. Ils possèdent tous pour cela une ou plusieurs sous-balises `transition`.

Ces dernières peuvent se présenter sous la forme la plus simple possible afin de spécifier l'état à atteindre (attribut `to`) après le déclenchement d'un événement (attribut `on`), comme l'illustre le code suivant :

```
<transition on="success" to="show.todolists" />
<transition on="detail" to="todolist" />
```

Spring Web Flow rend également possible l'exécution de traitements sur une transition en définissant une sous-balise `action` pour la balise `transition`.

Le code suivant permet d'exécuter la méthode `bindAndValidate` de l'action `todoListAction` sur l'événement `save` avant d'atteindre l'état `save.todolist` :

```
<transition on="save" to="save.todolist">
    <action bean="todoListAction" method="bindAndValidate" />
</transition>
```

Cette fonctionnalité facilite la mise en œuvre des formulaires dans un flot Web, comme nous le verrons plus loin. Nous verrons aussi à la section décrivant les entités de Spring Web Flow comment déclencher un événement afin de transiter vers un autre état du flot.

Dans les différents types d'états décrits ci-après, les noms des balises associées sont tous suffixés par –`state`.

Spring Web Flow offre la possibilité de définir des transitions globales, appliquées automatiquement à tous les états. Ces transitions peuvent être configurées directement au niveau du flot, comme dans l'exemple suivant :

```
<flow (...)>
    (...)
    <global-transitions>
        <transition on="globalEvent1" to="state1"/>
        <transition on="globalEvent2" to="state2"/>
    </global-transitions>
</flow>
```

Actions

Dans le contexte de Spring Web Flow, une action correspond à des traitements qui peuvent être exécutés au cours d'un flot. Elle peut être déclenchée aux différents moments suivants :

- Démarrage d'un flot.
- Moment où le flot entre dans un état.
- Moment où le flot sort d'un état.
- Moment où le flot exécute une transition.
- Moment de la finalisation d'un flot.

Une action est configurée par l'intermédiaire de la balise `action`, dont les principales propriétés sont le nom, avec l'attribut `name`, le Bean utilisé, avec l'attribut `bean`, et le nom de la méthode du Bean utilisé, avec l'attribut `method`.

Propriétés générales d'un état

D'une manière similaire à un flot, les différents états de Spring Web Flow possèdent les propriétés communes suivantes :

- attributs permettant de lui spécifier des propriétés ;
- actions exécutées au démarrage du flot ;
- transitions ;
- actions exécutées à la fin du flot ;
- entités de gestion des exceptions.

Ces propriétés peuvent être configurées pour tous les types d'états, comme le montre l'exemple de code suivant :

```
<some-state id="stateId">
    <attribute (...)/>
    <entry-actions>(...)</entry-actions>
    <transition .../>
    <exit-actions>(...)</exit-actions>
    <exception-handler (...)/>
</some-state>
```

État fondé sur une action

Ce type d'état correspond à l'exécution d'une méthode d'un objet Java. Il est défini par le biais de la balise `action-state`, comprenant l'attribut `id` afin de spécifier son identifiant.

De par son type, cette balise contient une sous-balise `action` permettant de rattacher l'état à l'une des méthodes d'action, comme le montre le code suivant, tiré du fichier **flows-todolist.xml** du répertoire **WEB-INF/ :**

```
<action-state id="todolist">
    <action bean="todoListAction" method="setupForm"/>
    <transition on="success" to="show.todolist"/>
</action-state>
```

La valeur de l'attribut `bean` de la balise `action` correspond à l'identifiant d'un Bean configuré dans le contexte de Spring.

État fondé sur une vue

Ce type d'état est défini par le biais de la balise `view-state`, qui comprend l'attribut `id` afin de spécifier son identifiant et l'attribut `view` afin de relier l'état à la vue qu'il doit afficher.

Le code suivant est tiré du fichier **flows-todolist.xml** du répertoire **WEB-INF/ :**

```
<view-state id="show.todolist" view="todolist">
    <transition on="list" to="todolists"/>
    <transition on="save" to="save.todolist">
        <action bean="todoListAction" method="bindAndValidate"/>
    </transition>
    <transition on="todos" to="todos"/>
    <transition on="end" to="end"/>
</view-state>
```

La valeur de l'attribut `view` de la balise `view-state` correspond à l'identifiant d'une vue. Ce dernier est résolu suivant les fonctionnalités de la technologie sous-jacente utilisée. Dans le cas de Spring MVC, il est résolu par l'intermédiaire des implémentations de l'interface `ViewResolver` *(voir le chapitre 7)*.

État de décision

Spring Web Flow permet de définir des éléments d'aiguillage fondés sur une ou plusieurs conditions. Le framework fournit cet état par le biais de la balise `decision-state`, qui comprend un attribut `id` correspondant à l'identifiant de l'état.

Cette balise peut comporter une ou plusieurs sous-balises `if`, afin de configurer les différents tests. La balise `if` comprend un attribut `test` pour définir la condition, un attribut `then` pour spécifier l'identifiant de l'état dans le cas d'une vérification du test et un attribut optionnel `else` pour spécifier l'identifiant de l'état dans le cas contraire.

Le code suivant illustre la mise en œuvre d'un tel état :

```
<decision-state id="isTodolistRss">
    <if test="${requestScope.todolist == null}"
        then="todolist"/>
    <if test="${requestScope.todolist.rssAllowed}"
        then="configureTodoListRss"
        else="configureTodoList"/>
</decision-state>
```

Les conditions suivent la syntaxe OGNL et peuvent utiliser des variables du contexte d'exécution du flot.

OGNL (Object-Graph Navigation Language)

OGNL est un langage fondé sur des expressions destiné à positionner et récupérer les propriétés d'objets Java. Il consiste à évaluer des expressions qui peuvent avoir la forme suivante : `${requestScope.todolist.rssAllowed}`. Le projet relatif au langage est accessible depuis les adresses *http://www.ognl.org* et *http://www.opensymphony.com/ognl/*.

État d'exécution d'un sous-flot

Les flots d'exécution étant par essence réutilisables, il est possible de les faire interagir les uns avec les autres. Un flot principal peut appeler des sous-flots et, dans ce cas, suspendre son exécution le temps de l'exécution du sous-flot.

Spring Web Flow fournit un état à cet effet, défini par le biais de la balise `subflow-state`. Cette balise comprend un attribut `id` correspondant à l'identifiant de l'état, ainsi qu'un attribut `flow` permettant de relier l'état au flot à exécuter.

Le code suivant est tiré du fichier **flows-todolist.xml,** localisé dans le répertoire **WEB-INF/ :**

```
<subflow-state id="todos" flow="flows-todo">
    (...)
    <transition on="show.end" to="todolist"/>
</subflow-state>
```

La valeur de l'attribut `id` correspond à un sous-flot défini pour le registre de stockage des définitions de flots. De ce fait, il fait référence à un fichier définissant un flot, dont le nom correspond à l'identifiant, suffixé par **.xml.**

Dans notre cas, l'identifiant fait référence à un flot défini dans le fichier **flows-todo.xml,** localisé dans le répertoire **WEB-INF/ :**

```
(...)
<flow start-state="todos">
(...)
</flow>
```

Spring Web Flow offre la possibilité de mapper des attributs du contexte du flot principal dans des attributs du contexte du sous-flot, comme dans le code suivant :

```
<subflow-state id="todos" flow="flows-todo">
    <attribute-mapper>
        <input-mapping value="${requestScope.id}" as="todoListId" />
    </attribute-mapper>
    <transition on="show.end" to="todolist" />
</subflow-state>
```

État de fin de flot

Ce type d'état permet de terminer un flot d'exécution et de libérer toutes les ressources associées à son contexte d'exécution. Il est défini par le biais de la balise `view-state`, comprenant l'attribut `id` pour spécifier son identifiant et l'attribut `view` pour relier l'état à la vue qu'il doit afficher.

Le code suivant est tiré du fichier **flows-todolist.xml,** localisé dans le répertoire **WEB-INF/ :**

```
<end-state id="show.end" view="end" />
```

Implémentation des entités

Les différentes entités d'un flot Web à implémenter sont les actions et les vues.

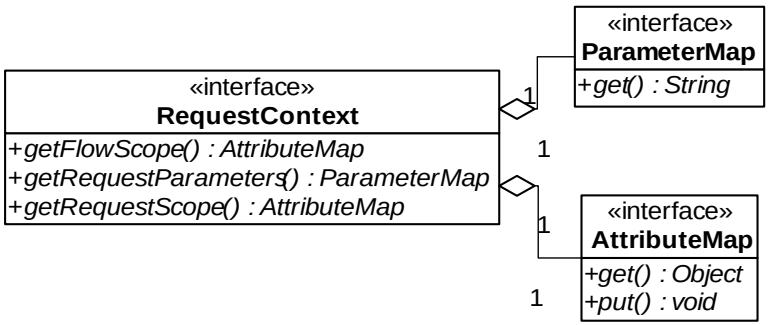
Les actions correspondent à des classes Java particulières, qui peuvent être de différents types suivant la problématique à résoudre. Les vues peuvent quant à elles utiliser des technologies diverses, dépendant du framework MVC utilisé conjointement avec Spring Web Flow.

Le contexte du flot

Chaque méthode d'action spécifique de Spring Web Flow prend en paramètre un contexte permettant d'accéder entre autres aux paramètres de la requête et des attributs du flot. Ces mécanismes s'appuient principalement sur les interfaces `RequestContext`, `ParameterMap` et `AttributeMap`, toutes localisées dans le package `org.springframework.webflow`.

La figure 8.4 illustre les interfaces précédemment introduites ainsi que leurs méthodes principales.

Figure 8.4
Entités globales de Spring Web Flow



Comme le montre le code suivant, toutes les méthodes d'actions utilisables dans un flot Web prennent en paramètre un objet de type `RequestContext` afin d'avoir accès aux informations des différents scopes et de l'événement déclencheur :

```
public Event myMethod(RequestContext context) throws Exception {
    //Récupération d'un paramètre de la requête
    ParameterMap parameters = context.getRequestParameters();
    String myParameter=(String)parameters.get ("myParameter");

    //Récupération d'un attribut dans le scope du flot
    AttributeMap attributes = context.getScopeFlow();
    String myAttribute=(String)attributes.get("myAttribute");
    (...)
}
```

Ces méthodes retournent un objet de type `Event` afin de spécifier l'événement qui doit être pris en compte lors de la transition, comme dans le code suivant :

```
public Event myMethod(RequestContext context) throws Exception {
    (...)
    //Retourne par exemple l'événement « success »
    return success();
}
```

Le type *Action*

Il existe trois types d'actions principaux, les actions simples, les multiactions et les actions de gestion de formulaires.

L'action ayant pour nom `Action` correspond à la forme la plus simple d'action ainsi qu'à l'abstraction de base de toutes les actions. Ces dernières doivent nécessairement l'implémenter. Localisée dans le package `org.springframework.webflow`, elle possède un unique point d'entrée, comme l'illustre le code suivant :

```
public class Action {
    public Event execute(RequestContext context) throws Exception;
}
```

La méthode de l'interface prend en paramètre le contexte courant du flot exécuté et renvoie un événement permettant de déterminer la transition vers l'action suivante.

Spring Web Flow fournit, en complément de cette interface, la classe `AbstractAction` du package `org.springframework.webflow.action`. Cette classe correspond à une implémentation abstraite de base permettant d'utiliser et d'implémenter des actions. Elle dispose notamment des méthodes retournant des événements prédéfinis et très souvent utilisés, comme le récapitule le tableau 8.1.

Tableau 8.1. Événements prédéfinis par la classe *AbstractAction*

Identifiant de l'événement	Méthode	Description

`success`	`Event success()`
	`Event success(Object result)`
	`Event success(String resultParameterName, Object result)`
		Événement correspondant au succès d'un traitement

`error`	`Event error ()` `Event error(Exception e)`	Événement correspondant à une erreur lors d'un traitement
`yes`	`Event yes()` `Event yesOrNo(boolean booleanResult)`	Événement correspondant à une affirmation
`no`	`Event no()` `Event yesOrNo(boolean booleanResult)`	Événement correspondant à une négation

`-`	`Event result(String eventId)`
	`Event result(String eventId, Map parameters)`
	`Event result(String eventId, String parameterName, Object parameterValue)`
		Construction personnalisée d'événements à partir d'un identifiant

La classe `AbstractAction` fournissant sa propre implémentation de la méthode `execute`, le développeur peut utiliser les méthodes suivantes :

- `doPreExecute`, qui permet d'exécuter des traitements avant l'éventuelle exécution de la méthode `doExecute`. Si la méthode `doPreExecution` renvoie un événement non `null`, la méthode `doExecute` n'est pas exécutée, et cet événement est renvoyé.
- `doExecute`, qui permet d'implémenter les traitements de l'action.
- `doPostExecute`, qui permet d'exécuter des traitements après la méthode `doExecute`, à moins qu'une exception n'ait été levée dans celle-ci.

Le code suivant met en œuvre une action simple fondée sur la classe `AbstractAction` :

```
public class MyAbstractAction extends AbstractAction {
    protected Event doPreExecute(
                        RequestContext context) throws Exception {
        System.out.println("Exécution de la méthode doPreExecute");
        return null;
    }

    protected Event doExecute(
                        RequestContext context) throws Exception {
```

```
        (...)
    }

    protected void doPostExecute(
                        RequestContext context) throws Exception {
        System.out.println(
                         "Exécution de la méthode doPostExecute");
    }
}
```

Le type *MultiAction*

Spring Web Flow fournit un type d'action plus évolué, permettant de définir plusieurs points d'entrée. Ce mécanisme peut être mis en relation avec la balise `bean` dans les fichiers XML de description de flots, laquelle supporte un attribut `method` afin de définir la méthode à appeler sur l'action.

Les points d'entrée doivent être de la forme suivante, où `myMethod` est un nom de méthode quelconque :

```
public Event myMethod(RequestContext context) throws Exception {
    (...)
}
```

Lorsque l'attribut `method` n'est pas spécifié, le framework utilise l'identifiant de l'`action-state` englobante comme nom par défaut de la méthode à appeler.

La classe de base de ce type d'action est `MultiAction`, localisée dans le package `org.springframework.webflow.action`. Cette classe étend la classe `AbstractAction` afin d'implémenter les mécanismes de sélection de la méthode à appeler.

Le code suivant met en œuvre ce type d'action avec la classe `TodoListsAction`, localisée dans le package `tudu.web` de l'application Tudu Lists :

```
public class TodoListsAction extends MultiAction {
    (...)

    public Event list(RequestContext context) throws Exception {
        (...)
        return success();
    }

    public Event detail(RequestContext context) throws Exception {
        (...)
        return success();
    }

    public Event end(RequestContext context) throws Exception {
        (...)
        return success();
    }

    (...)
}
```

Si nous considérons que l'instance de l'action ci-dessus est configurée dans le contexte de Spring avec l'identifiant `myMultiAction`, l'état utilisant cette action peut être configuré de l'une des deux manières suivantes :

```
<action-state id="test">
    <action bean="myMultiAction" method="myMethod"/>
    (...)
</action-state>
```

Ou :

```
<action-state id="myMethod">
    <action bean="myMultiAction"/>
    (...)
</action-state>
```

Utilisation de Beans en tant qu'actions

Spring Web Flow offre la possibilité d'utiliser n'importe quel Bean en tant qu'action. Dans ce cas, le composant n'a aucune adhérence avec le framework. L'objectif de cette fonctionnalité est d'appeler directement une classe d'une application, telle qu'un service métier dans un flot Web. Ainsi, l'implémentation d'une action réalisant simplement l'appel au Bean n'est plus nécessaire.

Spring Web Flow se charge de récupérer un objet dans un contexte du flot et de le passer en paramètre de la méthode. Si aucune erreur ne survient durant l'exécution de cette méthode, l'objet de retour est placé dans le contexte avec une clé configurée dans le flot, et un événement `success` est renvoyé. Dans le cas contraire, un événement `error` est renvoyé.

Le code suivant donne un exemple de mis en œuvre de cette fonctionnalité :

```
<action-state id="searchTodos">
    <action bean="searchTodosManager"
            method=" searchTodos (${flowScope.todoCriteria})"
            resultName="results"/>
    <transition on="success" to="displayTodos"/>
</action-state>
```

Le Bean doit contenir une méthode `searchTodos`, avec un paramètre de type `TodosCriteria`, contenant les informations pour la recherche. Afin de garantir cette règle, le Bean peut implémenter, par exemple, l'interface suivante :

```
public interface SearchTodosManager {
    List searchTodos(TodosCriteria criteria);
}
```

Le type *FormAction*

Un des types d'actions les plus utilisés est celui gérant les formulaires. Il correspond à une spécialisation du type d'action précédent par le biais de la classe `FormAction` du package `org.springframework.webflow.action`. Il possède un cycle de vie de gestion inspiré du framework Spring MVC, que nous allons détailler.

La classe `FormAction` fournit les différentes propriétés récapitulées au tableau 8.2.

Tableau 8.2. Propriétés de la classe *FormAction*

Méthode	Valeur par défaut	Description
`FormObjectName`	`formObject`	Définit le nom de l'objet représentant le formulaire.
`FormObjectClass`	`null`	Définit la classe de l'objet représentant le formulaire.
`formObjectScope`	`ScopeType.REQUEST`	Définit l'endroit de stockage de l'objet représentant le formulaire.
`ErrorsScope`	`ScopeType.REQUEST`	Définit l'endroit de stockage de l'objet contenant les erreurs de validation d'un objet de formulaire.
`propertyEditorRegistrar`	`null`	Définit l'implémentation de l'interface `PropertyEditorRegistar` afin d'enregistrer des `PropertyEditor` pour l'action. Ces enregistrements peuvent également être réalisés en surchargeant la méthode `initBinder` de l'action.
`Validator`	`null`	Spécifie l'instance du validateur à utiliser afin de valider l'objet représentant le formulaire.
`bindOnSetupForm`	`false`	Spécifie si l'objet représentant le formulaire doit être rempli avec les paramètres de la requête durant l'exécution de la méthode `setupForm`.
`validateOnBinding`	`true`	Spécifie si l'objet représentant le formulaire doit être validé lors de son remplissage avec les paramètres de la requête.
`messageCodesResolver`	`null`	Spécifie la stratégie à utiliser afin de convertir les codes d'erreurs en messages.

La classe `FormAction` fournit six méthodes prédéfinies pouvant être appelées lors de l'exécution du flot par le biais de la balise `action`. Le tableau 8.3 récapitule les fonctions de ces méthodes prédéfinies.

Tableau 8.3. Méthodes prédéfinies de la classe *FormAction*

Méthode	Description
`exposeFormObject(RequestContext)`	Charge l'objet représentant le formulaire et le place dans le contexte tout en enregistrant les différents `PropertyEditor`. Retourne l'événement `success` en cas du succès du chargement, et `error` dans le cas contraire.
`setupForm(RequestContext)`	Se comporte de la même manière que la méthode précédente en ajoutant la possibilité de réaliser du binding, ou mappage, de données de la requête. La méthode se comporte de la manière concernant les événements retournés.
`bindAndValidate(RequestContext)`	Utilise les différents paramètres de la requête afin de remplir l'objet de formulaire puis le valide en utilisant le validateur configuré. Retourne l'événement `success` s'il n'y a pas d'erreurs de binding ou de validation, et `error` dans le cas contraire.
`bind(RequestContext)`	Utilise les différents paramètres de la requête afin de remplir l'objet de formulaire. Retourne l'événement `success` s'il n'y a pas d'erreurs de binding, et `error` dans le cas contraire.

Tableau 8.3. Méthodes prédéfinies de la classe *FormAction (suite)*

`validate(RequestContext)`	Valide l'objet de formulaire associé à l'action en utilisant le validateur configuré. Retourne l'événement `success` s'il n'y a pas d'erreurs de validation, et `error` dans le cas contraire.
`resetForm(RequestContext)`	Réinitialise l'objet de formulaire associé à l'action. Retourne l'événement `success` si les traitements se passent bien, et `error` dans le cas contraire.

L'utilisation de la classe `FormAction` se configure sur différentes entités d'un flot, puisqu'elle impacte forcément plusieurs états.

L'état de vue affichant le formulaire ainsi que ses éventuelles erreurs doit initialiser correctement l'action puis spécifier la méthode de l'action à exécuter sur la transition, comme le montre le code suivant, tiré du fichier **flows-todolist.xml,** localisé dans le répertoire **WEB-INF/ :**

```
<view-state id="show.todolist" view="todolist">
    <action bean="todoListAction" method="setupForm"/>
    <transition on="list" to="todolists" />
    <transition on="save" to="save.todolist">
        <action bean="todoListAction" method="bindAndValidate"/>
    </transition>
    <transition on="todos" to="todos"/>
    <transition on="end" to="end"/>
</view-state>
```

L'état `save.todolist` implémente une méthode `save` afin de traiter les données du formulaire, comme dans le code suivant, tiré du même fichier que précédemment :

```
<action-state id="save.todolist">
    <action bean="todoListAction" method="save"/>
    <transition on="success" to="todolist"/>
</action-state>
```

La classe `FormAction` fournit également différentes méthodes de gestion de son cycle de vie interne, qu'il est possible de surcharger afin de le personnaliser. Ces méthodes principales sont récapitulées au tableau 8.4.

Tableau 8.4. Méthodes surchargeables du cycle de vie de la classe *FormAction*

Méthode	Description
`initBinder(RequestContext, DataBinder)`	Permet de configurer l'instance de `DataBinder` passée en paramètre. L'objectif est d'enregistrer des `ProrpertyEditor` spécifiques pour certaines propriétés de l'objet de formulaire. Cette méthode est appelée par la méthode `createBinder`, qu'il est éventuellement possible de surcharger afin de créer une implémentation de `DataBinder` personnalisée.
`loadFormObject(RequestContext)`	Permet de charger l'objet de formulaire. Par défaut, elle appelle la méthode `createFormObject`, qu'il est possible de surcharger, bien qu'il soit préférable de surcharger la méthode `loadFormObject`.

Contrairement à Spring MVC, la classe `FormAction` ne fournit pas de méthode similaire à la méthode `referenceData` afin de charger des données utilisées pour l'affichage d'un formulaire. Le développeur doit l'implémenter si nécessaire afin de placer des données dans le scope du flot et l'appeler explicitement dans un état du flot Web.

Le code suivant donne un exemple d'implémentation de cette méthode :

```
public Event setupReferenceData(
                        RequestContext context) throws Exception {
    Scope requestScope = context.getRequestScope();
    requestScope.setAttribute("refData", myDao.loadData());
    return success();
}
```

Dans le cas de l'action implémentée par la classe `TodoListAction`, nous chargeons l'objet de formulaire à partir de la base de données. Nous devons pour cela surcharger la méthode `createFormObject` et implémenter une méthode `save` afin de traiter les données du formulaire.

Le code de cette classe est le suivant :

```
public class TodoListAction extends FormAction {

    private TodoListsManager todoListsManager;

    public TodoListAction() {
        setFormObjectName("todoList");
        setFormObjectClass(TodoList.class);
        setFormObjectScope(ScopeType.FLOW);
    }

    public Object createFormObject(RequestContext context)
                                              throws Exception {
        String listId = (String)context
                    .getSourceEvent().getParameter("id");
        return todoListsManager.findTodoList(listId);
    }

    public Event save(RequestContext context) throws Exception {
        TodoList todoList = (TodoList)context
                    .getFlowScope().getAttribute("todoList");
        todoListsManager.updateTodoList(todoList);
        return success();
    }

    (...)
}
```

Remarquons dans cet exemple qu'un paramètre de requête pour un état est accessible par le biais du contexte et de sa propriété `sourceEvent`, qui possède une méthode nommée `getParameter`. L'objet de formulaire est de plus accessible sur le scope configuré pour l'action. Dans notre cas, il s'agit du scope du flot accessible à partir du contexte et de sa propriété `flowScope`, qui met à disposition une méthode nommée `getAttribute`.

Exécution des flots

L'exécution des flots se réalise en utilisant le contrôleur configuré pour les différents frameworks MVC.

Pour démarrer un flot, le paramètre `flotId` doit être utilisé afin de spécifier le flot choisi. L'appel par le biais, par exemple, de la balise HTML `<a href></a>` se réalise de la manière suivante :

```
<a href="<c:url value="/flow/todolists.html
            ?_flowId=flows-todolist"/>">Démarrage du flot </a>
```

Par la suite, le contexte du flot doit être précisé conjointement avec l'événement déclenché par l'intermédiaire des paramètres `_flowExecutionKey` et `_eventId`. L'appel par le biais, par exemple, de la balise HTML `<a href></a>` se réalise de la manière suivante :

```
<a href="<c:url value="/flow/todolists.html?_flowExecutionKey
              =${flowExecutionKey}&_eventId=list"/>">Mon lien</a>
```

Tudu Lists : utilisation de Spring Web Flow

Telle qu'elle est développée, l'application Tudu Lists ne permet pas d'intégrer facilement Spring Web Flow, car elle est fondée sur la technologie AJAX, qui n'est pas supportée pour le moment par le framework. C'est la raison pour laquelle nous utilisons un projet spécifique, `Tudu-WebFlow`.

Nous migrons donc les fonctionnalités de gestion des listes de todos et des todos avec Spring Web Flow dans un sous-projet annexe et indépendant. Nous concevons à cet effet deux flots gérant respectivement les listes de todos et les todos. Ces flots sont utilisés avec le moteur de Spring Web Flow fondé sur Spring MVC. Ils utilisent les mêmes concepts, adaptés aux différentes entités concernées.

Nous allons tout d'abord décrire les deux flots Web, puis nous implémenterons les différentes actions des flots et les configurerons ainsi que le moteur.

Conception des flots

L'application se compose d'un flot principal, nommé `flows-todolist`, décrit dans le fichier **flows-todolist.xml** du répertoire **WEB-INF,** et d'un sous-flot nommé `flows-todo`, décrit dans le fichier **flows-todo.xml,** localisé dans le même répertoire.

Le flot principal, `flows-todolist`, correspond à un flot Web d'affichage et de modification des listes de todos, comme l'illustre la figure 8.5. La première étape consiste à afficher l'ensemble des listes de todos pour un utilisateur. Il est ensuite possible d'afficher le détail d'un des éléments de cette liste afin de le modifier. À partir de ce détail, le flot peut passer la main au flot de gestion des todos.

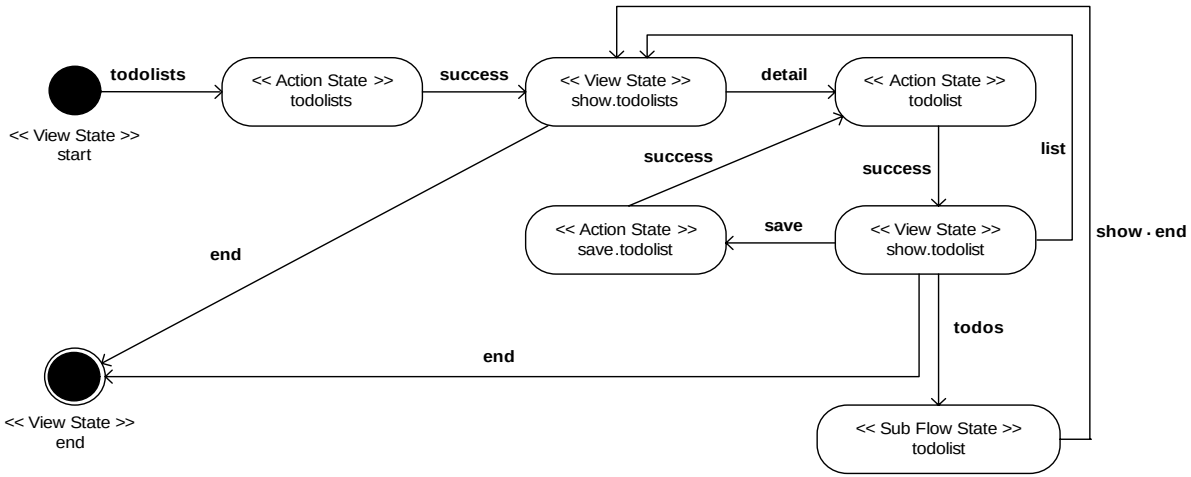


Figure 8.5

Flot de gestion des listes de todos

Le code suivant fournit le contenu du fichier **flows-todolist.xml,** correspondant à la configuration du flot Web :

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

<!DOCTYPE flow PUBLIC "-//SPRING//DTD WEBFLOW 1.0//EN"
       "http://www.springframework.org/dtd/spring-webflow-1.0.dtd">

<flow start-state="start">
    <view-state id="start" view="start">
        <transition on="todolists" to="todolists"/>
    </view-state>

    <action-state id="todolists">
        <action bean="todoListsAction" method="list"/>
        <transition on="success" to="show.todolists"/>
    </action-state>

    <view-state id="show.todolists" view="todolists">
        <transition on="detail" to="todolist"/>
        <transition on="end" to="end"/>
    </view-state>

    <action-state id="todolist">
        <action bean="todoListAction" method="setupForm"/>
        <transition on="success" to="show.todolist"/>
    </action-state>

    <view-state id="show.todolist" view="todolist">
        <transition on="list" to="todolists"/>
        <transition on="save" to="save.todolist">
```

```
            <action bean="todoListAction"
                    method="bindAndValidate"/>
        </transition>
        <transition on="todos" to="todos"/>
        <transition on="end" to="end"/>
    </view-state>

    <action-state id="save.todolist">
        <action bean="todoListAction" method="save"/>
        <transition on="success" to="todolist"/>
    </action-state>

    <subflow-state id="todos" flow="flows-todo">
        <attribute-mapper>
            <input-mapping
                 value="${requestScope.id}" as="todoListId"/>
        </attribute-mapper>
        <transition on="show.end" to="todolist"/>
    </subflow-state>

    <action-state id="end">
        <action bean="todoListsAction" method="end"/>
        <transition on="success" to="show.end"/>
    </action-state>

    <end-state id="show.end" view="end"/>
</flow>
```

Le sous-flot `flows-todo` correspond à un flot Web d'affichage et de modification des todos, comme l'illustre la figure 8.6. Celui-ci peut être appelé depuis le flot principal. Il suit le même principe que le flot précédent, en affichant tout d'abord la liste des todos pour un identifiant de liste, puis en permettant l'affichage du détail d'un todo en vue d'une éventuelle modification.

Le contenu du fichier **flows-todo.xml** correspondant à la configuration du flot Web est le suivant :

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

<!DOCTYPE flow PUBLIC "-//SPRING//DTD WEBFLOW 1.0//EN"
       "http://www.springframework.org/dtd/spring-webflow-1.0.dtd">

<flow start-state="todos">
    <action-state id="todos">
        <action bean="todosAction" method="list"/>
        <transition on="success" to="show.todos"/>
    </action-state>

    <view-state id="show.todos" view="todos">
        <transition on="detail" to="todo"/>
        <transition on="end" to="end"/>
    </view-state>
```

```
    <action-state id="todo">
        <action bean="todoAction" method="setupForm"/>
        <transition on="success" to="show.todo"/>
    </action-state>

    <view-state id="show.todo" view="todo">
        <transition on="list" to="todos"/>
        <transition on="save" to="save.todo">
            <action bean="todoAction" method="bindAndValidate"/>
        </transition>
        <transition on="end" to="end"/>
    </view-state>

    <action-state id="save.todo">
        <action bean="todoAction" method="save"/>
        <transition on="success" to="todo"/>
    </action-state>

    <action-state id="end">
        <action bean="todosAction" method="end"/>
        <transition on="success" to="show.end"/>
    </action-state>

    <end-state id="show.end"/>
</flow>
```

Les deux flots s'appuient sur différents types d'états, utilisant aussi bien des actions que des vues.

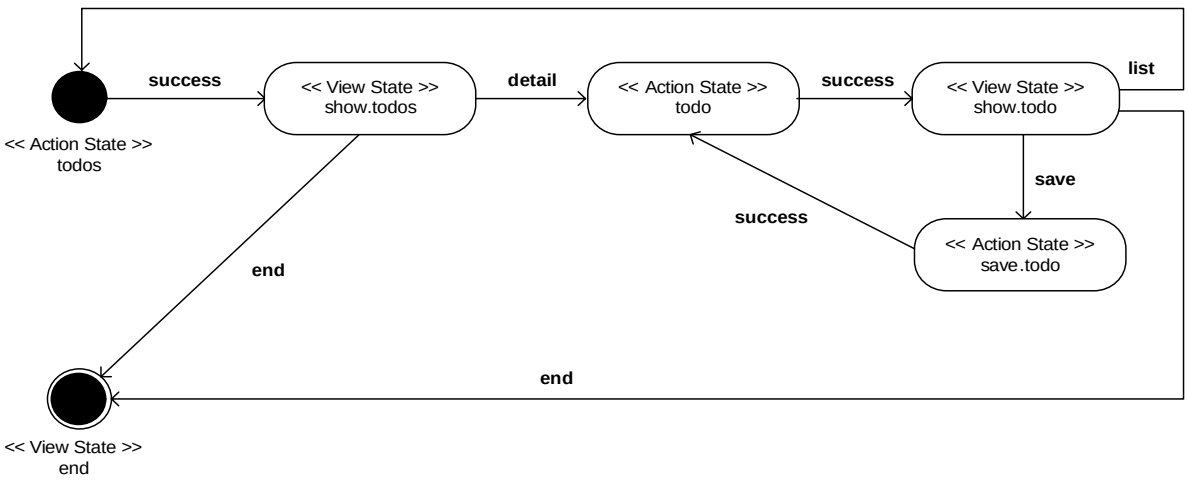


Figure 8.6

Flot de gestion des todos

Implémentation des entités

Les différentes entités à implémenter sont les actions et les vues. Concernant les vues, nous faisons le choix d'utiliser les technologies JSP et JSTL, ainsi que les fonctionnalités de gestion des vues du framework Spring MVC.

Décrivons tout d'abord les différents types d'actions implémentés.

Les types d'actions

Tudu Lists utilise des actions à entrées multiples et des actions de gestion de formulaires, afin d'adresser respectivement l'affichage des listes de todos et celui des todos, ainsi que leurs modifications.

Les actions d'affichage des listes s'appuient sur une action de type `MultiAction` du package `org.springframework.webflow.action.MultiAction`.

Prenons l'exemple de la classe `TodoListsAction`, localisée dans le package `tudu.web` et de sa méthode `list`. Cette dernière charge la liste des `todolists` en utilisant une instance de la classe `TodoListsManager` injectée par Spring et la place dans le contexte du flot, comme dans le code suivant :

```
public class TodoListsAction extends MultiAction {
    private UserManager userManager;
    private TodoListsManager todoListsManager;

    public Event list(RequestContext context) throws Exception {
        User user = userManager.getCurrentUser();
        Collection todoLists = user.getTodoLists();
        context.getRequestScope().put("todoLists", todoLists);
        return success();
    }

    public UserManager getUserManager() {
        return userManager;
    }

    public void setUserManager(UserManager userManager) {
        this.userManager = userManager;
    }

    public TodoListsManager getTodoListsManager() {
        return todoListsManager;
    }

    public void setTodoListsManager(
                    TodoListsManager todoListsManager) {
        this.todoListsManager = todoListsManager;
    }
}
```

Tudu Lists utilise également des actions de gestion de formulaires afin de sauvegarder les modifications sur des entités telles que les listes de todos avec la classe `TodoListAction`, localisée dans le package `tudu.web`. Cette classe surcharge la méthode `createFormObject`

afin de charger une instance de la classe `TodoList` à partir de la base de données et implémente une méthode `save` afin de sauvegarder les modifications :

```
public class TodoListAction extends FormAction {
    private TodoListsManager todoListsManager;

    public TodoListAction() {
        setFormObjectName("todoList");
        setFormObjectClass(TodoList.class);
        setFormObjectScope(ScopeType.FLOW);
    }

    public Object createFormObject(RequestContext context)
                                                throws Exception {
        ParameterMap parameters = context.getRequestParameters();
        String listId = parameters.get("id");
        return todoListsManager.findTodoList(listId);
    }

    public Event save(RequestContext context) throws Exception {
        TodoList todoList = (TodoList)context
                            .getFlowScope().get("todoList");
        todoListsManager.updateTodoList(todoList);
        return success();
    }

    public TodoListsManager getTodoListsManager() {
        return todoListsManager;
    }

    public void setTodoListsManager(
                        TodoListsManager todoListsManager) {
        this.todoListsManager = todoListsManager;
    }
}
```

Notons que les propriétés sont configurées dans le constructeur de l'action mais qu'elles auraient très bien pu l'être dans le contexte applicatif de Spring.

Les vues

Puisque nous utilisons Spring MVC comme framework MVC, nous retrouvons les principes détaillés à la section « Spring MVC et la gestion de la vue » du chapitre 7.

La seule différence consiste dans l'appel du contrôleur de Spring Web Flow, qui requiert les paramètres suivants :

- identifiant du flot à exécuter avec le paramètre ayant pour clé `_flotId`. Cet élément doit être utilisé afin d'identifier le flot à démarrer.
- clé relative à l'exécution du flot courant avec le paramètre ayant pour clé `_flowExecutionKey` ;
- identifiant de l'événement déclenché avec le paramètre ayant pour clé `_eventId`.

Le démarrage d'un flot avec la balise HTML `<a href></a>` se réalise de la façon suivante :

```
<a href="<c:url value="/flow/todolists.html?_flowId
          =flows-todolist"/>">Démarrage du flot flows-todolist</a>
```

Un appel d'un état du flot, par le biais de la balise HTML `<a href></a>`, se réalise de la manière suivante :

```
<a href="<c:url value="/flow/todolists.html?_flowExecutionKey
            =${flowExecutionKey}&_eventId=list"/>">Mon lien</a>
```

L'appel peut également être réalisé par l'envoi d'un formulaire HTML, qui doit contenir en champs cachés les deux paramètres précédemment cités, comme dans le code suivant :

```
<form action="<c:url value="/flow/todolists.html"/>">
    <input type="hidden" name="_flowExecutionKey"
               value="<c:out value="${flowExecutionKey}"/>"/>
    <input type="hidden" name="_eventId "
               value="<c:out value="list"/>"/>
    (...)
    <input type="submit" value="Envoyer"/>
</form>
```

Spring Web Flow permet également de spécifier l'événement au niveau des boutons de soumission, comme le montre le code suivant, équivalant au précédent :

```
<form action="<c:url value="/flow/todolists.html"/>">
    <input type="hidden" name="_flowExecutionKey"
               value="<c:out value="${flowExecutionKey}"/>"/>
    (...)
    <input name="_eventId_list" type="submit" value="Envoyer"/>
</form>
```

Configuration de Tudu Lists

La configuration de l'application Tudu Lists comporte deux parties : celle du moteur de Spring Web Flow et celle des entités utilisées.

Configuration du moteur de Spring Web Flow

Puisque nous avons fait le choix d'utiliser le support permettant d'utiliser Spring Web Flow avec Spring MVC, le moteur est accessible par l'intermédiaire d'un élément de type `Controller` de Spring MVC.

Toute la configuration du moteur est réalisée dans le fichier **action-servlet.xml,** qui correspond à la configuration du contexte de Spring MVC.

La première partie de ce fichier sert à configurer ce `Controller`, puis son accès (mapping) et la stratégie de résolution des vues :

```
<beans>
    (...)

    <bean id="messageSource" class="org.springframework.context
```

```
                            .support.ResourceBundleMessageSource">
        <property name="basename" value="messages" />
    </bean>

    <bean id="exceptionResolver"
          class="tudu.web.FlowExceptionResolver" />

    <bean id="urlMapping" class="org.springframework.web
                        .servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
            <props>
                <prop key="/flow/todolists.html">
                                    flowController</prop>
                <prop key="/login.html">loginAction</prop>
            </props>
        </property>
    </bean>

    <bean name="flowController" class="org.springframework.webflow
                                    .executor.mvc.FlowController">
        <property name="flowExecutor" ref="flowExecutor"/>
        <property name="cacheSeconds" value="5"/>
    </bean>

    <bean id="viewResolver" 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>

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

Le contrôleur s'appuie sur un gestionnaire d'exécution de flot correspondant au moteur en lui-même. Ce Bean permet de spécifier la stratégie de stockage des informations des flots ainsi que l'emplacement des différents fichiers XML de description des flots, comme le montre le code suivant, tiré du même fichier **action-servlet.xml :**

```
<beans>
    (...)

    <bean id="flowExecutor" class="org.springframework
                            .webflow.executor.FlowExecutorImpl">
        <constructor-arg>
            <bean id="repositoryFactory" class="org.springframework
                      .webflow.execution.repository.continuation
                      .ContinuationFlowExecutionRepositoryFactory">
                <constructor-arg ref="flowRegistry"/>
            </bean>
        </constructor-arg>
    </bean>
```

```
    <bean id="flowRegistry" class="org.springframework
                    .webflow.registry.XmlFlowRegistryFactoryBean">
        <property name="flowLocations">
            <list>
                <value>/WEB-INF/flows-*.xml</value>
            </list>
        </property>
    </bean>

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

Notons que le sous-flot (Bean d'identifiant `flows-todo`) doit également être configuré en tant Bean, même s'il n'est pas directement rattaché au moteur dans la configuration de Spring Web Flow.

Configuration des entités utilisées

Les différentes actions doivent être configurées dans le fichier **action-servlet.xml,** correspondant à la configuration du contexte de Spring MVC, afin de les faire bénéficier de l'injection de dépendances.

Ce fichier contient la configuration des Beans récapitulés au tableau 8.5.

Tableau 8.5. Actions de Spring Web Flow utilisées dans Tudu Lists

Identifiant	Classe	Description
`todoListsAction`	`tudu.web.TodoListsAction`	Permet de charger la liste des listes de todos d'un utilisateur et de la rendre disponible par le biais du contexte du flot. Elle utilise des instances des classes `UserManager` et `TodoListsManager` devant être injectées.
`todoListAction`	`tudu.web.TodoListAction`	Permet de manipuler une entité de type `TodoList` correspondant à une liste de todos. Elle utilise une instance de la classe `TodoListsManager` devant être injectée.
`todosAction`	`tudu.web.TodosAction`	Permet de charger les todos d'une liste de todos et de les rendre disponibles par le biais du contexte du flot. Elle utilise une instance de la classe `TodoListsManager` devant être injectée.
`todoAction`	`tudu.web.TodoAction`	Permet de manipuler une entité de type `Todo` correspondant à une liste de todos. Elle utilise une instance de la classe `TodosManager` devant être injectée.

Le code suivant illustre la configuration de ces actions dans le fichier **action-servlet.xml :**

```
<beans>
    (...)

    <bean id="todoListsAction" class="tudu.web.TodoListsAction">
        <property name="userManager" ref="userManager"/>
        <property name="todoListsManager" ref="todoListsManager"/>
    </bean>
```

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

    <bean id="todosAction" class="tudu.web.TodosAction">
        <property name="todoListsManager" ref="todoListsManager"/>
    </bean>

    <bean id="todoAction" class="tudu.web.TodoAction">
        <property name="todosManager" ref="todosManager"/>
    </bean>

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

Les définitions des Beans de dépendances sont localisées dans les fichiers XML dont le nom commence par **applicationContext,** dans le répertoire **WEB-INF/.**

En résumé

Les éléments du flot et entités utilisées, qu'il s'agisse de vues ou de méthodes d'actions au niveau des états et des transitions, sont récapitulés à la figure 8.7 pour le flot `flow-todolist` décrit dans le fichier **flow-todolist.xml** du répertoire **WEB-INF/.**

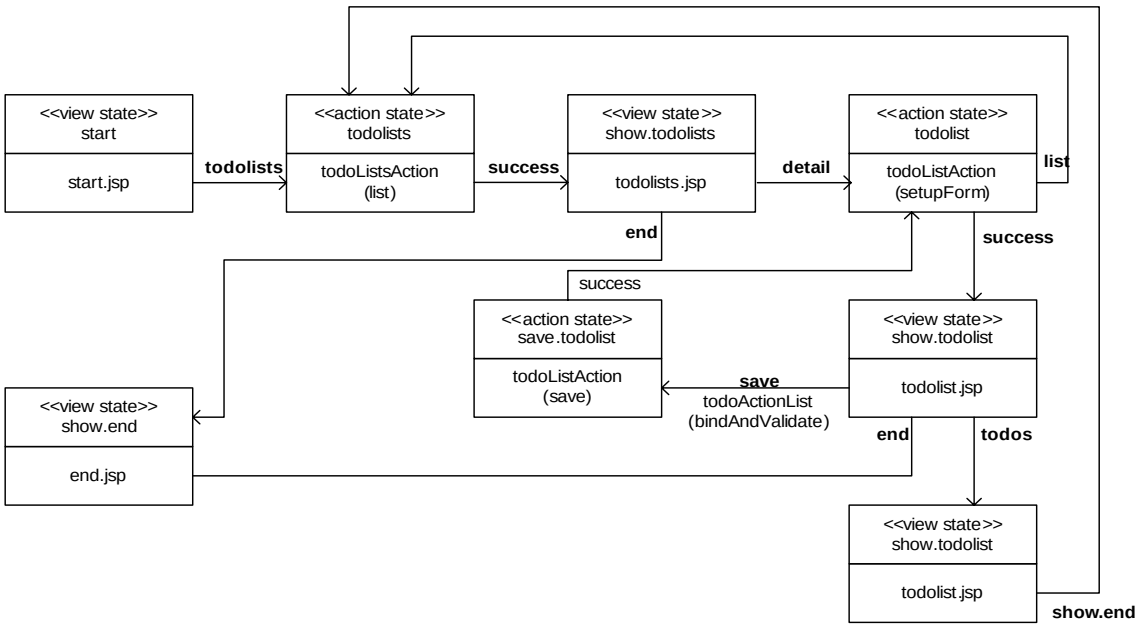


Figure 8.7
Récapitulatif des éléments du flot flow-todolist

La figure 8.8 récapitule ces différents éléments pour le flot `flow-todo` décrit dans le fichier **flow-todo.xml** du répertoire **WEB-INF/.**

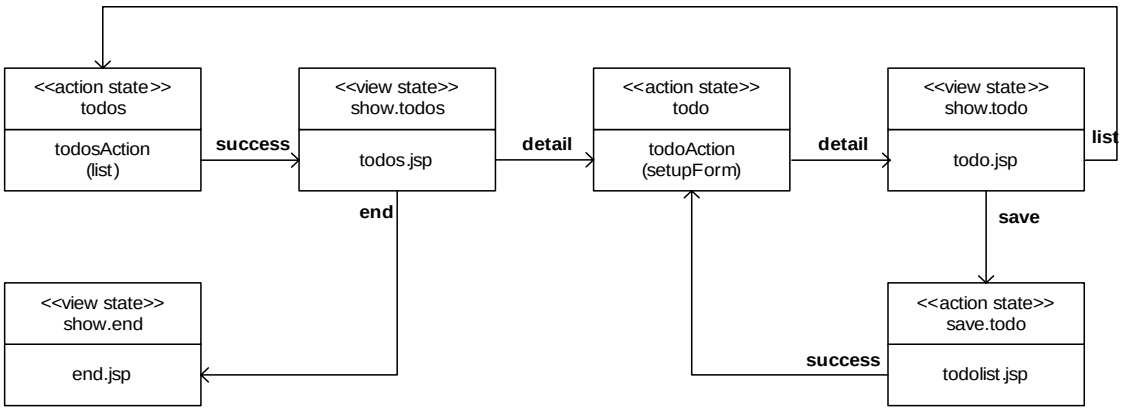


Figure 8.8
Récapitulatif des éléments du flot flow-todo

Conclusion

Le framework Spring Web Flow a pour principale fonction d'adresser les problématiques liées à la navigation des applications Web. Sa mise en œuvre n'est toutefois pas appropriée à tous les types d'applications Web. Certaines applications laissent à l'utilisateur le loisir de naviguer librement alors que d'autres imposent des règles strictes de navigation. Spring Web Flow vise à faciliter le développement de ces dernières.

Ce framework découple et abstrait les différentes briques de mise en œuvre des flots Web, telles que flot, moteur d'exécution de flots et architecture existante (framework MVC, par exemple). Cette approche favorise la réutilisation des flots dans différents environnements. Le moteur peut également être paramétré sans impacter les flots et utilisé sur différentes architectures.

Spring Web Flow fournit un cadre robuste et flexible afin de définir un flot, d'implémenter les actions et les vues, de les relier au flot et d'intégrer le flot dans une architecture existante par le biais du framework Spring. Il reprend d'ailleurs les concepts mis en œuvre dans ce framework ainsi que son implémentation MVC.

Spring Web Flow peut également être utilisé avec le support de Spring des portlets. Au moment de l'écriture de cet ouvrage, le framework fournit un exemple expérimental permettant d'utiliser les flots Web en mode AJAX. L'objectif affirmé du framework est d'étendre le spectre de ces intégrations et de devenir par ce biais un standard de fait pour la gestion des flots Web.

9

Utilisation d'AJAX avec Spring

Depuis leurs débuts, les applications Web n'offrent à l'utilisateur final qu'une expérience relativement pauvre. Le cycle traditionnel requête/réponse sur lequel est fondé le Web ne permet pas la création d'interfaces client élaborées, telles que nous pouvons en observer dans les applications bureautiques traditionnelles.

Le problème vient essentiellement du fait que chaque action de l'utilisateur requiert un rechargement complet de la page HTML. Or c'est sur ce point qu'AJAX intervient en permettant le rechargement à chaud de portions d'une page HTML.

Le terme AJAX (Asynchronous JavaScript And XML) renvoie à un ensemble de technologies différentes, toutes existantes depuis longtemps mais qui, combinées intelligemment, permettent de changer radicalement notre mode de création des interfaces Web.

Le concept d'AJAX a été lancé début 2005 par Jesse James Garrett dans un article devenu célèbre, intitulé *AJAX: A New Approach to Web Applications*, disponible sur le site Web de sa société, Adaptive Path, à l'adresse *http://www.adaptivepath.com/publications/essays/archives/000385.php*.

Les technologies utilisées par AJAX, telles que JavaScript et XML, sont connues depuis longtemps, et, en réalité, plusieurs sociétés faisaient déjà de l'AJAX dès 2000. La révolution actuelle provient de l'arrivée à maturité de ces technologies désormais largement répandues et livrées en standard avec les navigateurs Internet récents. Pour preuve, des entreprises comme Google et Microsoft se mettent à utiliser AJAX de manière intensive. Des produits tels que GMail ou Google Maps sont des exemples de technologies AJAX mises à la disposition du grand public.

Ce chapitre se penche en détail sur AJAX et introduit plusieurs techniques permettant de créer une application en suivant ses principes. Nous aborderons ainsi DWR, l'un des frameworks AJAX les plus populaires et qui a la particularité de s'intégrer facilement à Spring. En particulier, ce framework nous permettra de publier des Beans Spring en JavaScript afin de manipuler directement des objets gérés par Spring depuis un navigateur Internet.

Nous soulignerons au passage les principaux écueils qui guettent le développeur AJAX et conclurons le chapitre par une présentation de script.aculo.us, une bibliothèque de scripts Open Source qui permet de réaliser aisément des effets spéciaux.

AJAX et le Web 2.0

Cette section entre dans le détail des technologies et concepts utilisés avec AJAX.

Nous commencerons par démystifier le terme marketing « Web 2.0 », très souvent associé à AJAX. Nous aborderons ensuite le fameux objet `XMLHttpRequest`, qui est au fondement d'AJAX et qui nous permettra de créer une première fonction AJAX simple.

Le Web 2.0

AJAX est souvent associé au Web 2.0. S'il s'agit là d'une campagne marketing très bien orchestrée, force est de reconnaître que derrière les mots se cache une façon radicalement nouvelle de construire des applications Web.

Le Web 2.0 est un concept plus global qu'AJAX, qui regroupe un ensemble d'aspects fonctionnels et métier. L'expression a été conçue par les sociétés O'Reilly et MediaLive International et a été définie par Tim O'Reilly dans un article disponible à l'adresse *http://www.oreillynet.com/lpt/a/6228.*

Nous pouvons retenir de cet article que les applications Web 2.0 s'appuient sur les sept principes suivants :

- Offre d'un service, et non plus d'une application packagée.
- Contrôle d'une banque de données complexe, difficile à créer et qui s'enrichit au fur et à mesure que des personnes l'utilisent.
- Implication des utilisateurs dans le développement de l'application.
- Utilisation de l'intelligence collective, les choix et les préférences des utilisateurs étant utilisés en temps réel pour améliorer la pertinence du site.
- Choix de laisser les internautes utiliser l'application comme un self-service, sans contact humain nécessaire.
- Fonctionnement de l'application sur un grand nombre de plates-formes, et plus seulement sur l'ordinateur de type PC (il y a aujourd'hui plus de terminaux mobiles que de PC ayant accès à Internet).

- Simplicité des interfaces graphiques, des processus de développement et du modèle commercial.

Nous comprendrons mieux cette philosophie en observant des sites emblématiques de ce Web 2.0, tels que les suivants :

- Wikipedia, une encyclopédie en ligne gérée par les utilisateurs eux-mêmes. Chacun est libre d'y ajouter ou de modifier du contenu librement, des groupes de bénévoles se chargeant de relire les articles afin d'éviter les abus.
- Les blogs, qui permettent à chacun de participer et de lier (mécanisme des *trackbacks*) des articles.
- del.icio.us et Flickr, des sites collaboratifs dans lesquels les utilisateurs gèrent leurs données en commun (des liens pour del.icio.us, des images pour Flickr). Ainsi, chacun participe à l'enrichissement d'une base de données géante.

Le Web 2.0 est donc davantage un modèle de fonctionnement collaboratif qu'une technologie particulière. Pour favoriser l'interactivité entre les utilisateurs, ces sites ont besoin d'une interface graphique riche et dynamique. C'est ce que leur propose AJAX, avec sa capacité à recharger à chaud des petits morceaux de pages Web. Ces mini-mises à jour permettent à l'utilisateur d'enrichir la base de données de l'application sans pour autant avoir à recharger une page HTML entière.

Tudu Lists, notre application étude de cas, est un modeste représentant du Web 2.0, puisqu'elle permet le partage de listes de tâches entre utilisateurs, à la fois en mode Web (avec AJAX) et sous forme de flux RSS.

Les technologies d'AJAX

AJAX met en œuvre un ensemble de technologies relativement anciennes, que vous avez probablement déjà rencontrées.

Un traitement AJAX se déroule de la manière suivante :

1. Dans une page Web, un événement JavaScript se produit : un utilisateur a cliqué sur un bouton (événement `onClick`), a modifié une liste déroulante (événement `onChange`), etc.
2. Le JavaScript qui s'exécute alors utilise un objet particulier, le `XMLHttpRequest` ou, si vous utilisez Microsoft Internet Explorer, le composant ActiveX `Microsoft.XMLHTTP`. Cet objet n'est pas encore standardisé, mais il est disponible sur l'ensemble des navigateurs Internet récents. Situé au cœur de la technique AJAX, il permet d'envoyer une requête au serveur en tâche de fond, sans avoir à recharger la page.
3. Le résultat de la requête peut ensuite être traité en JavaScript, de la même manière que lorsque nous modifions des morceaux de page Web en DHTML. Ce résultat est généralement du contenu renvoyé sous forme de XML ou de HTML, que nous pouvons ensuite afficher dans la page en cours.

AJAX est donc un mélange de JavaScript et de DHTML, des technologies utilisées depuis bien longtemps. L'astuce vient essentiellement de ce nouvel objet `XMLHttpRequest`, lui aussi présent depuis longtemps, mais jusqu'à présent ignoré du fait qu'il n'était pas disponible sur suffisamment de navigateurs Internet.

Voici un exemple complet de JavaScript utilisant cet objet `XMLHttpRequest` pour faire un appel de type AJAX à un serveur. C'est de cette manière que fonctionnaient les anciennes versions de Tudu Lists (avant l'arrivée de DWR, que nous détaillons plus loin dans ce chapitre). Cet exemple illustre le réaffichage à chaud d'une liste de tâches, ainsi que l'édition d'une tâche :

```
<script language="JavaScript">
var req;
var fragment;

/**
 * Fonction AJAX générique pour utiliser XMLHttpRequest.
 */
function retrieveURL(url) {
  if (window.XMLHttpRequest) {
    req = new XMLHttpRequest();
    req.onreadystatechange = miseAJourFragment;
    try {
      req.open("GET", url, true);
    } catch (e) {
      alert("<fmt:message key="todos.ajax.error"/>");
    }
    req.send(null);
  } // Utilisation d'ActiveX, pour Internet Explorer
  else if (window.ActiveXObject) {
    req = new ActiveXObject("Microsoft.XMLHTTP");
    if (req) {
      req.onreadystatechange = miseAJourFragment;
      req.open("GET", url, true);
      req.send();
    }
  }
}

/**
 * Met à jour un fragment de page HTML.
 */
function miseAJourFragment() {
  if (req.readyState == 4) { // requête complétée
    if (req.status == 200) { // réponse OK

      document.getElementById(fragment)
        .innerHTML = req.responseText;

    } else {
```

```
      alert("<fmt:message key="todos.ajax.error"/> " +
        req.status);
    }
  }
}

/**
 * Affiche la liste des tâches.
 */
function afficheLesTaches() {
  fragment='todosTable';
  retrieveURL('${ctx}/ajax/manageTodos.action?' +
    'listId=${todoList.listId}&method=render');
}

/**
 * Edite une tâche.
 */
function editeUneTache(id) {
  fragment='editFragment';
  retrieveURL('${ctx}/ajax/manageTodos.action?todoId=' +
    escape(id) + '&method=edit');
}

( ... )

</script>
```

Dans cet exemple, des fragments de page sont mis à jour à chaud, avec du contenu HTML renvoyé par le serveur.

Comme nous pouvons le deviner en regardant les URL envoyées au serveur, ces requêtes HTTP sont traitées par des actions Struts, le paramètre `method` étant utilisé par des actions Struts de type `DispatchAction` *(voir le chapitre 6 pour plus d'informations sur le traitement de ces requêtes).*

Le HTML renvoyé par les actions Struts est simplement affiché dans des éléments HTML. En l'occurrence, la méthode `innerHTML` utilisée plus haut permet de changer le HTML présent dans des éléments `<span>` :

```
<span id="editFragment"></span>

<span id="todosTable"></span>
```

Cette utilisation d'AJAX reste très simple mais réclame une relativement bonne connaissance de JavaScript. Suffisante pour un site Web simple, elle souffre cependant des quelques lacunes suivantes :

- Elle nécessite beaucoup de code JavaScript et devient difficile à utiliser dès lors que nous voulons faire plus que simplement afficher du HTML.
- Elle pose des problèmes de compatibilité d'un navigateur à un autre.

Afin de faciliter l'utilisation de ce type de technique, un certain nombre de frameworks ont vu le jour, tels Dojo, DWR, JSON-RPC, MochiKit, SAJAX, etc.

Nous présentons DWR, l'un des plus populaires, dans la suite de ce chapitre.

Le framework AJAX DWR (Direct Web Remoting)

Parmi la multitude de frameworks AJAX, nous avons choisi DWR pour sa popularité et son intégration à Spring. DWR permet très facilement de présenter en JavaScript des JavaBeans gérés par Spring. Il est ainsi possible de manipuler dans un navigateur Internet des objets s'exécutant côté serveur.

DWR est un framework Open Source développé par Joe Walker, qui permet d'utiliser aisément des objets Java (côté serveur) en JavaScript (côté client).

Fonctionnant comme une couche de transport, DWR permet à des objets Java d'être utilisés à distance selon un principe proche du RMI (Remote Method Invocation). Certains autres frameworks AJAX, comme Rico, proposent des bibliothèques beaucoup plus complètes, avec des effets spéciaux souvent très impressionnants. Pour ce type de résultat, DWR peut être utilisé conjointement avec des bibliothèques JavaScript non spécifiques à J2EE. Nous abordons plus loin dans ce chapitre l'utilisation conjointe de DWR et de script.aculo.us, une bibliothèque d'effets spéciaux très populaire parmi les utilisateurs de Ruby On Rails.

Signe de la reconnaissance de DWR par la communauté Java, des articles ont été publiés sur les sites développeur de Sun, IBM et BEA. Dans le foisonnement actuel de frameworks AJAX, nous pouvons raisonnablement avancer que DWR est l'un des plus prometteurs.

Publié sous licence Apache 2.0, ce framework peut être utilisé sans problème, quelle que soit l'application que vous développez.

Principes de fonctionnement

DWR est composé de deux parties : du JavaScript, qui s'exécute côté client dans le navigateur de l'utilisateur, et une servlet Java, laquelle est chargée de traiter les requêtes envoyées par le JavaScript.

DWR permet de présenter des objets Java en JavaScript et de générer dynamiquement le JavaScript nécessaire à cette fonctionnalité. Les développeurs JavaScript ont ainsi la possibilité d'utiliser de manière transparente des objets Java, qui tournent dans le serveur d'applications et qui donc ont la possibilité d'accéder à une base de données, par exemple. Cela augmente considérablement les possibilités offertes à l'interface graphique d'une application.

La figure 9.1 illustre l'appel d'une fonction Java depuis une fonction JavaScript contenue dans une page Web.

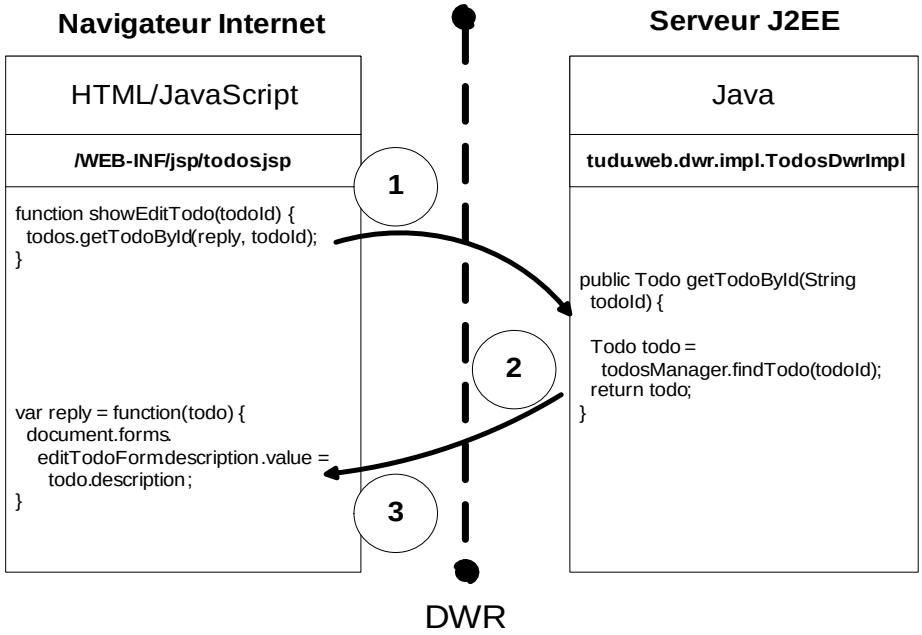


Figure 9.1

Fonctionnement de DWR

Afin de mieux illustrer ce propos, voici comment fonctionne une version simplifiée du code permettant d'éditer une tâche à chaud dans Tudu Lists.

Suite au clic de souris d'un utilisateur sur le texte « edit », à droite d'une tâche, le JavaScript suivant s'exécute :

```
function showEditTodo(todoId) {
 Effect.Appear('editTodoDiv');
 todos.getTodoById(reply, todoId);
 document.forms.editTodoForm.description.focus();
}
```

Ce code affiche le layer DHTML d'édition d'une page (il s'agit d'un effet spécial de script.aculo.us), appelle un objet Java distant et donne le focus à la description de la tâche éditée.

Le code d'appel à la fonction Java est composé d'un objet `todo`, généré automatiquement par DWR, qui prend les deux paramètres suivants en fonction :

- `reply`, la fonction callback JavaScript. Il s'agit donc non pas d'une variable passée en argument, mais d'une fonction JavaScript qui s'exécute après l'exécution de l'appel à l'objet Java.
- `todoId`, l'identifiant de la tâche à éditer, qui est passé en argument à la fonction Java distante.

Ce code appelle la fonction Java distante `getTodoById` de l'objet `todo` présenté par DWR :

```
package tudu.web.dwr.impl;

( ... )

/**
 * Implementation du service tudu.service.TodosManager.
 */
public class TodosDwrImpl implements TodosDwr {

  ( ... )
  public RemoteTodo getTodoById(String todoId) {
    Todo todo = todosManager.findTodo(todoId);
    RemoteTodo remoteTodo = new RemoteTodo();
    remoteTodo.setDescription(todo.getDescription());
    remoteTodo.setPriority(todo.getPriority());
    if (todo.getDueDate() != null) {
      SimpleDateFormat formatter =
        new SimpleDateFormat("MM/dd/yyyy");

      String formattedDate = formatter.format(todo.getDueDate());
      remoteTodo.setDueDate(formattedDate);
    } else {
      remoteTodo.setDueDate("");
    }
    return remoteTodo;
  }
}
```

Ce code Java recherche la tâche demandée dans la couche de service de l'application, laquelle fait à son tour un appel à la couche de persistance afin qu'Hibernate retrouve les informations en base de données. Pour des raisons de sécurité, que nous verrons plus tard dans ce chapitre, ce code crée un objet Java spécifique pour la couche DWR, et cet objet est retourné au client.

Au retour d'exécution du code Java, la fonction callback `reply`, que nous avons vue plus haut, s'exécute :

```
var reply = function(todo) {
 document.forms.editTodoForm.description.value = todo.description;
 document.forms.editTodoForm.priority.value = todo.priority;
 document.forms.editTodoForm.dueDate.value = todo.dueDate;
}
```

Étant une fonction callback, elle prend automatiquement un seul argument, l'objet retourné par la fonction JavaScript `getTodoById`. Cela revient à dire qu'elle utilise l'objet Java retourné par la méthode `getTodoById`, qui représente une tâche. Il est ensuite aisé de recopier les attributs de cet objet dans les champs HTML du layer présentant la tâche ainsi éditée.

Configuration

Vu de l'extérieur, DWR se compose d'une servlet et d'un fichier de configuration, nommé **dwr.xml.** Sa configuration n'est donc pas particulièrement complexe, d'autant que le framework fournit une excellente interface de test.

La servlet DWR

DWR est tout d'abord une servlet, qu'il faut configurer dans l'application Web en cours. Les différents JAR fournis dans la distribution de DWR doivent donc être copiés dans le répertoire **WEB-INF/lib,** et la servlet DWR doit être configurée dans le fichier **WEB-INF/web.xml,** comme toutes les servlets :

```
( ... )
<servlet>
<servlet-name>dwr-invoker</servlet-name>
<servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
<init-param>
  <param-name>debug</param-name>
  <param-value>false</param-value>
</init-param>
</servlet>

( ... )

<servlet-mapping>
  <servlet-name>dwr-invoker</servlet-name>
  <url-pattern>/secure/dwr/*</url-pattern>
</servlet-mapping>
( ... )
```

Dans cette configuration, le paramètre `debug` peut être mis à `true` afin d'avoir des informations plus précises sur le fonctionnement de DWR. Par ailleurs, l'URL d'accès à DWR est `/secure/dwr/*`. Cette URL est bien entendu librement configurable. Dans Tudu Lists, nous préférons la mettre derrière l'URL `/secure/*`, afin qu'elle soit protégée par Acegi Security.

Le fichier *dwr.xml*

DWR est configuré *via* un fichier, qui est par défaut **WEB-INF/dwr.xml.** Ce fichier est configurable grâce aux paramètres d'initialisation de la servlet DWR :

```
<servlet>
  <servlet-name>dwr-user-invoker</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
  <init-param>
    <param-name>config-user</param-name>
    <param-value>WEB-INF/dwr-user.xml</param-value>
  </init-param>
</servlet>
```

Le nom du paramètre doit impérativement commencer par `config`. C'est pourquoi, dans l'exemple ci-dessus, nous l'avons appelé `config-user`. L'intérêt de cette manipulation est d'autoriser plusieurs fichiers de configuration, et ainsi plusieurs instances différentes de la servlet DWR : une instance pour les utilisateurs normaux (`config-user`) et une pour les administrateurs (`config-admin`). En leur donnant des URL différentes, nous pouvons utiliser la sécurité standard de J2EE pour ne permettre qu'aux utilisateurs ayant un rôle donné d'utiliser une instance de DWR.

Le fichier **dwr.xml** est de la forme suivante :

```
<dwr>
  <init>
    <creator id="..." class="..."/>
    <converter id="..." class="..."/>
  </init>

  <allow>
    <create creator="..." javascript="..." scope="...">
      <param name="..." value="..."/>
    </create>

    <convert convertor="..." match="..."/>
  </allow>

  <signatures>
    ( ... )
  </signatures>
</dwr>
```

La partie `<init></init>` n'est généralement pas utilisée. Elle sert à instancier les classes utilisées plus bas dans le fichier de configuration, dans le cas où le développeur souhaiterait utiliser ses propres classes au lieu de celles fournies en standard par DWR. Nous allons voir que les classes livrées avec DWR sont cependant largement suffisantes pour une utilisation de l'outil, fût-elle avancée.

La section `<allow></allow>` indique à DWR quelles classes il a le droit d'instancier et de convertir. Cette section est la plus importante du fichier. Nous allons détailler les éléments Creator et Convertor qui la composent.

Pour instancier une classe, DWR utilise un Creator, défini dans le fichier de configuration par la balise `<create>`. Ce Creator peut instancier une classe de plusieurs manières et les associer à un JavaScript généré à la volée :

```
<allow>
  <create creator="new" javascript="Example">
    <param name="class" value="tudu.web.dwr.Example"/>
  </create>
</allow>
```

Dans cet exemple, le Creator `new` permet d'instancier une classe et de l'associer à un JavaScript nommé `Example`. Il s'agit du Creator le plus simple, mais il en existe d'autres,

en particulier pour accéder à des Beans Spring, ce que nous verrons dans l'étude de cas. L'objet `Example` une fois présenté ainsi en JavaScript, ses méthodes peuvent être appelées depuis le navigateur client. Cependant, il est tout à fait possible que lesdites méthodes utilisent comme arguments, ou comme objets de retour, des classes complexes, telles que des collections ou des JavaBeans. Ces classes vont donc devoir être converties d'un langage à l'autre.

Pour convertir une classe Java en JavaScript et *vice versa,* DWR utilise des Converters. En standard, DWR est fourni avec des Converters permettant la conversion des types Java primaires, des dates, des collections et des tableaux, des JavaBeans, mais aussi d'objets plus complexes, comme des objets DOM ou Hibernate.

Pour des raisons de sécurité, certains Converters complexes, comme celui gérant les JavaBeans, ne sont pas activés par défaut. Pour convertir un JavaBean en JavaScript, il faut donc autoriser sa conversion :

```
<allow>
  <convert converter="bean"
           match="tudu.web.dwr.bean.RemoteTodoList"/>

  <convert converter="bean"
           match="tudu.web.dwr.bean.RemoteTodo"/>
</allow>
```

Utilisation du JavaScript dans les JSP

Une fois la servlet DWR correctement configurée, il est nécessaire d'accéder au code JavaScript généré dynamiquement par le framework depuis les pages JSP. Pour cela, il faut importer dans les JSP les deux bibliothèques propres à DWR :

```
<script type="text/javascript"
        src="${ctx}/secure/dwr/engine.js"></script>

<script type="text/javascript"
        src="${ctx}/secure/dwr/util.js"></script>
```

Dans Tudu Lists, cet import est réalisé dans le fichier **WEB-INF/jspf/header.jsp.**

Notons que ces deux fichiers JavaScript sont fournis par la servlet DWR et qu'ils n'ont pas à être ajoutés manuellement dans l'application Web.

Il faut ensuite importer les fichiers JavaScript générés dynamiquement par DWR et qui représentent les classes Java décrites dans **dwr.xml.** Voici l'exemple de la classe `tudu.web.dwr.impl.TodosDwrImpl`, qui est configurée dans **dwr.xml** pour être présentée en tant que todos :

```
<script type='text/javascript'
        src='${ctx}/secure/dwr/interface/todos.js'></script>
```

Ce fichier doit donc être importé depuis la servlet DWR mappée dans Tudu Lists sur l'URL `/secure/dwr/*`, à laquelle nous ajoutons le suffixe `interface`.

Test de la configuration

La configuration étudiée ci-dessus est relativement complexe, l'import et l'utilisation du JavaScript généré posant généralement problème aux nouveaux utilisateurs. C'est pourquoi DWR est fourni avec une interface de tests très bien conçue. Pour la mettre en place, il faut modifier la configuration de DWR dans le fichier **web.xml,** afin de passer en mode debug :

```
<servlet>
  <servlet-name>dwr-invoker</servlet-name>
  <servlet-class>uk.ltd.getahead.dwr.DWRServlet</servlet-class>
  <init-param>
    <param-name>debug</param-name>
    <param-value>true</param-value>
  </init-param>
</servlet>
```

Notons bien que cette configuration ne doit pas être utilisée en production, car elle faciliterait la tâche d'une personne malveillante.

Une fois ce paramétrage effectué, nous pouvons accéder à des pages de test *via* l'URL de la servlet DWR, qui, dans notre étude de cas, est *http://127.0.0.1:8080/Tudu/secure/dwr/* (si vous avez changé la racine de contexte du projet, remplacez *Tudu* dans l'URL par la nouvelle racine, en respectant les majuscules). Ces pages listent les objets présentés par DWR, ainsi que leurs méthodes. Ces méthodes peuvent être appelées directement. Le code source de ces pages fournit une aide précieuse pour l'utilisation de DWR.

La figure 9.2 illustre la page affichant la classe `TodosDWRImpl`, l'une des deux classes configurées avec DWR dans Tudu Lists.

Figure 9.2
Interface de test de DWR

Utilisation de l'API servlet

DWR permet de s'abstraire de l'API servlet, ce qui simplifie généralement grandement le travail à effectuer. En comparaison de Struts, nous utilisons directement des arguments passés en paramètres au lieu de les extraire depuis un objet de formulaire.

L'API servlet reste toutefois incontournable pour peu que nous voulions stocker des attributs dans la session HTTP, utiliser JAAS pour la sécurité ou accéder au contexte de l'application Web. Pour toutes ces utilisations, DWR stocke dans une variable `ThreadLocal` les objets `HttpServletRequest`, `HttpServletResponse`, `HttpSession`, `ServletContext` et `ServletConfig`.

Pour accéder à la requête HTTP en cours, il suffit d'écrire :

```
HttpServletRequest request
  = uk.ltd.getahead.dwr.WebContextFactory.get()
                .getHttpServletRequest();
```

L'accès à cette variable en `ThreadLocal` n'est bien entendu possible que pour les fils d'exécution correspondant à des requêtes traitées avec DWR.

Gestion des performances

À première vue, l'utilisation d'AJAX améliore les performances. Au lieu de demander au serveur de recharger des pages complètes, nous nous contentons de lui demander le strict minium.

Cependant, dès que nous maîtrisons cette technique, nous sommes rapidement poussés à multiplier les requêtes côté serveur. Le site devient alors plus riche, plus « Web 2.0 », mais, en contrepartie, les requêtes HTTP se trouvent multipliées.

Coût de DWR en matière de performances

En lui-même, DWR n'ajoute qu'un impact négligeable en matière de performances côté serveur. Le serveur d'applications, le temps réseau ou les traitements métier sont normalement bien plus importants.

Joe Walker, l'auteur de DWR, a effectué plusieurs tests qui lui permettent de proposer les optimisations suivantes :

- Les appels de moins de 1 500 octets peuvent tenir dans un seul paquet TCP, ce qui améliore sensiblement l'utilisation du réseau. Il est donc important de n'échanger que le strict minimum de données.
- L'utilisation de paramètres de la JVM privilégiant les objets à faible durée de vie apporte un gain de performances, tout du moins avec Tomcat. Avec une JVM Sun, il s'agit de l'option -XX:NewSize.

Utilisation de batchs pour les requêtes

Il est possible de grouper des requêtes DWR afin de diminuer le nombre d'allers-retours entre le client et le serveur. Nous retrouvons cette optimisation à plusieurs endroits dans Tudu Lists, en particulier lors de l'ajout d'un todo :

```
DWREngine.beginBatch();
todos.addTodo(replyRenderTable,
  listId, description, priority, dueDate);

todos.getCurrentTodoLists(replyCurrentTodoLists);
DWREngine.endBatch();
```

La fonction `beginBatch()` permet de placer les requêtes suivantes en queue et de ne les envoyer qu'une fois la méthode `endBatch()` exécutée.

Étude des performances avec JAMon

Afin de mieux étudier les performances des appels à DWR, nous avons utilisé JAMon, un outil Open Source de monitoring des performances disponible à l'adresse *http://www.jamo-napi.com/.*

Pour cela, nous avons créé un filtre de servlet, disponible dans Tudu Lists mais que vous pouvez réutiliser sans modification dans votre propre application. Cette classe, `tudu.web.filter.JAMonFilter`, permet d'accumuler des statistiques sur l'ensemble des requêtes HTTP envoyées au serveur. Ces statistiques sont ensuite visualisables *via* **JAMonAdmin.jsp,** une JSP fournie sur le site de JAMon et que nous avons intégrée dans Tudu Lists. Il s'agit de l'onglet Monitoring, qui est uniquement accessible aux administrateurs de l'application.

Grâce à ce filtre et à la JSP permettant d'étudier les statistiques, il est possible de trouver les requêtes AJAX qui prennent le plus de temps ou celles qui sont appelées le plus souvent.

Dans le cas particulier de Tudu Lists, la requête la plus utilisée en production est l'appel à la méthode `renderTodos` du Bean Spring `todosDwr`. Elle est appelée plusieurs dizaines de milliers de fois par jour et a un temps de réponse inférieur à 10 millisecondes.

Intégration de Spring et de DWR

DWR propose une ingénieuse intégration à Spring, qui permet d'utiliser directement des Beans Spring en JavaScript.

Il faut pour cela utiliser un Creator particulier, nommé `spring`, qui indique à DWR que le Bean en question est géré par Spring. Ce créateur prend en paramètre l'ID du Bean Spring. C'est cette configuration que nous retrouvons dans Tudu Lists :

```
<dwr>
  <allow>
    <create creator="spring"
```

```
            javascript="todo_lists" scope="application">

      <param name="beanName" value="todoListsDwr" />
    </create>
    <create creator="spring"
            javascript="todos" scope="application">

      <param name="beanName" value="todosDwr" />
    </create>

( ... )

  </allow>
</dwr>
```

Dans cet exemple, nous publions en JavaScript les Beans Spring nommés `todoListsDwr` et `todosDwr`, par l'intermédiaire de DWR. Ces Beans ont bien entendu été configurés au préalable dans un contexte Spring, nommé **WEB-INF/applicationContext-dwr.xml.**

Voici la configuration du premier Bean, `todoListsDwr` :

```
<bean id="todoListsDwr"
      class="tudu.web.dwr.impl.TodoListsDwrImpl">

  <property name="todoListsManager">
    <ref bean="todoListsManager" />
  </property>
  <property name="userManager">
    <ref bean="userManager" />
  </property>
</bean>
```

Les Beans Spring ainsi configurés sont accessibles en JavaScript. En voici un exemple d'utilisation, tiré de la page JSP **WEB-INF/jsp/todo_lists.jsp,** qui sert à gérer les listes de tâches :

```
<script type='text/javascript'
        src='${ctx}/secure/dwr/interface/todo_lists.js'>
</script>

<script type="text/javascript">
( ... )

// Ajout d'un utilisateur à la liste sélectionnée.
function addTodoListUser() {
  var listId = document.forms.editListForm.listId.value;
  var login = document.forms.editListForm.login.value;
  todo_lists.addTodoListUser(replyAddTodoListUser, listId, login);
}
```

Tudu Lists : utilisation d'AJAX

Nous utilisons la technologie AJAX dans les deux pages principales de l'étude de cas Tudu Lists, la page de gestion des listes de todos et la page de gestion des todos.

Tudu Lists utilise DWR, intégré à Spring de la manière étudiée précédemment, afin d'éditer, ajouter, supprimer ou afficher des entités gérées dans la couche de service de l'application et persistées avec Hibernate.

Fichiers de configuration

Les fichiers de configuration sont ceux étudiés précédemment :

- **WEB-INF/web.xml,** qui sert à configurer la servlet DWR.xml.
- **WEB-INF/applicationContext-dwr.xml,** qui gère les Beans Spring présentés avec DWR.
- **WEB-INF/dwr.xml,** qui est le fichier de configuration de DWR.

Chargement à chaud d'un fragment de JSP

Nous allons commencer par l'exemple le plus simple, le chargement à chaud d'un fragment HTML généré côté serveur par une JSP. L'utilisation d'AJAX ne nécessite pas obligatoirement l'utilisation et la transformation de données en XML. Il est possible de générer une chaîne de caractères côté serveur et de l'afficher directement dans la page HTML en cours. Dans le monde J2EE, le moyen naturel pour générer du HTML étant un fichier JSP, voici de quelle manière transférer une JSP en AJAX.

Cette technique s'applique aux deux fichiers **WEB-INF/jspf/todo_lists_table.jsp** et **WEB-INF/jspf/todos_table.jsp.** Nous allons étudier plus précisément ce deuxième fichier, qui est essentiel à la génération de la principale page de l'application. Voyons pour cela le JavaScript contenu dans la page **WEB-INF/jsp/todos.jsp** qui va utiliser ce fragment de JSP :

```
function renderTable() {
  var listId = document.forms.todoForm.listId.value;
  todos.renderTodos(replyRenderTable, listId);
}

var replyRenderTable = function(data) {
  DWRUtil.setValue('todosTable',
    DWRUtil.toDescriptiveString(data, 1));
}
```

La fonction `renderTable()` utilise l'objet `todos`, qui est un Bean Spring présenté par DWR, pour générer du HTML. La variable `replyRenderTable` est une fonction callback prenant automatiquement en paramètre l'argument de retour de la fonction `renderTodos`.

Elle utilise des méthodes utilitaires fournies par DWR pour afficher cet argument de retour dans l'entité HTML possédant l'identifiant `todosTable`.

Ces fonctions utilitaires, fournies par le fichier **util.js,** ont été importées *via* le header de la JSP **WEB-INF/jspf/header.jsp.** Ce ne sont pas des fonctions AJAX, et elles ne servent qu'à faciliter l'affichage sans être pour autant obligatoires. Les deux fonctions utilisées ici sont les suivantes :

- `DWRUtil.setValue(id, value)`, qui recherche un élément HTML possédant l'identifiant donné en premier argument et lui donne la valeur du second argument.
- `DWRUtil.toDescriptiveString(objet, debug)`, une amélioration de la fonction `toString` qui prend en paramètre un niveau de debug pour plus de précision.

La partie la plus importante du code précédent concerne la fonction `todos.renderTodos()`, qui prend en paramètre la fonction callback que nous venons d'étudier ainsi qu'un identifiant de liste de todos.

Comme nous pouvons le voir dans le fichier **WEB-INF/dwr.xml,** cette fonction est en fait le Bean Spring `todosDwr` :

```
<create creator="spring" javascript="todos" scope="application">
  <param name="beanName" value="todosDwr" />
</create>
```

Ce JavaBean est lui-même configuré dans **WEB-INF/applicationContext-dwr.xml :**

```
<bean id="todosDwr" class="tudu.web.dwr.impl.TodosDwrImpl">
```

La fonction appelée est au final la fonction `renderTodos(String listId)` de la classe `TodosDwrImpl`.

Cette méthode utilise la méthode suivante pour retourner le contenu d'une JSP :

```
return WebContextFactory.get().forwardToString(
                    "/WEB-INF/jspf/todos_table.jsp");
```

Cette méthode permet donc de recevoir le résultat de l'exécution d'une JSP sous forme de chaîne de caractères, que nous pouvons ensuite insérer dans un élément HTML de la page grâce à la fonction `DWRUtil.setValue()`, que nous avons vue précédemment.

Cette technique évite d'utiliser du XML, qu'il faudrait parser et transformer côté client. Elle permet d'utiliser un résultat qui a en fait été obtenu côté serveur. En ce sens, ce n'est donc pas une technique AJAX « pure ». Elle a cependant l'avantage d'être simple et pratique.

Modification d'un tableau HTML avec DWR

Dans l'étape suivante de notre étude de cas, nous utilisons plus complètement l'API de DWR en modifiant les lignes d'un tableau HTML. Cette technique permet de créer des tableaux éditables à chaud par l'utilisateur. Ce dernier peut en changer les lignes et les cellules sans pour autant avoir à subir le rechargement de la page Web en cours.

La page utilisée pour gérer les todos, **WEB-INF/jsp/todos.jsp,** possède un menu sur la gauche, qui contient un tableau contenant les listes de todos possédées par l'utilisateur. Ce tableau est géré en AJAX grâce au JavaScript suivant :

```
function renderMenu() {
  todos.getCurrentTodoLists(replyCurrentTodoLists);
}

var replyCurrentTodoLists = function(data) {
  DWRUtil.removeAllRows("todoListsMenuBody");
  DWRUtil.addRows("todoListsMenuBody", data,
    [ selectTodoListLink ]);
}

function selectTodoListLink(data) {
  return "<a href=\"javascript:renderTableListId('"
    + data.listId + "')\">" + data.description + "</a>";
}
```

Nous utilisons les deux éléments importants suivants :

- `renderMenu()`, qui est la fonction appelée à d'autres endroits de l'application pour générer le tableau. En l'occurrence, cette fonction est appelée au chargement de la page, lors de la mise à jour des todos, ainsi que par une fonction lui demandant un rafraîchissement toutes les deux minutes. En effet, les listes de todos pouvant être partagées avec d'autres utilisateurs, les données doivent être régulièrement rafraîchies. Cette fonction appelle un Bean Spring présenté avec DWR, dont nous connaissons maintenant bien le fonctionnement, et utilise une variable callback.
- `replyCurrentTodoLists`, qui est une fonction callback prenant comme paramètre le résultat de la méthode `todos.getCurrentTodoLists()`. Cette méthode, qui appartient à la classe `tudu.web.dwr.impl.TodosDwrImpl`, retourne un tableau de JavaBeans, de type `tudu.web.dwr.bean.RemoteTodoList`. Afin de retourner un tableau de listes de todos, nous n'utilisons pas directement le JavaBean `tudu.domain.model.TodoList`. Présenter en JavaScript un objet de la couche de domaine pourrait en effet présenter un risque en matière de sécurité. Pour cette raison, seuls des Beans spécifiques à la présentation sont autorisés dans DWR, *via* la section `<allow></allow>` du fichier **WEB-INF/dwr.xml.**

Cette dernière fonction fait appel aux classes utilitaires de DWR suivantes, qui permettent de gérer les éléments de tableau :

- `DWRUtil.removeAllRows(id)`, qui enlève toutes les lignes du tableau HTML possédant l'identifiant passé en argument.
- `DWRUtil.addRows(id, data, cellFuncs)`, qui ajoute des lignes au tableau HTML possédant l'identifiant passé en premier argument. Les données utilisées pour construire ce tableau sont passées dans le deuxième paramètre de la fonction et sont un tableau d'objets. Le troisième paramètre est un tableau de fonctions. Chacune de ces fonctions prend en paramètre un élément du tableau, ce qui permet de créer les cellules.

Dans notre exemple, le tableau n'ayant qu'une colonne, une seule fonction, `selectTodoListLink(data)`, prend donc en paramètre un JavaBean, `tudu.web.dwr.bean.RemoteTodoList`, converti au préalable en JavaScript.

Nous pouvons ainsi utiliser le JavaScript pour obtenir l'identifiant et la description de la liste à afficher dans la cellule du tableau.

Utilisation du pattern session-in-view avec Hibernate

Le pattern session-in-view, présenté au chapitre 11, permet d'utiliser l'initialisation tardive, ou lazy-loading, pour de meilleures performances. Cette méthode, utile dans le cadre des JSP, reste entièrement valable pour des composants AJAX.

C'est de cette manière que le Bean Spring `tudu.web.dwr.impl.TodoListsDwrImpl` peut rechercher la liste des utilisateurs dans sa méthode `getTodoListsUsers()`. En effet, la liste des utilisateurs est une collection en initialisation tardive, c'est-à-dire qu'elle n'est réellement recherchée en base de données qu'au moment où elle appelée.

Pour ce faire, il nous faut configurer le filtre de servlets d'Hibernate afin qu'il traite les requêtes envoyées à DWR de la même manière qu'il traite les requêtes envoyées à Struts ou à Spring MVC. Cela se traduit par un double mapping dans le fichier **WEB-INF/web.xml :**

```
<filter>
  <filter-name>Hibernate Session In View Filter</filter-name>
  <filter-class>
org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
  </filter-class>
</filter>

<!-- configuration pour Struts -->
<filter-mapping>
  <filter-name>Hibernate Session In View Filter</filter-name>
  <url-pattern>*.action</url-pattern>
</filter-mapping>

<!-- configuration pour DWR -->
<filter-mapping>
  <filter-name>Hibernate Session In View Filter</filter-name>
  <url-pattern>/secure/dwr/*</url-pattern>
</filter-mapping>
```

Améliorations apportées par script.aculo.us

script.aculo.us est une bibliothèque d'effets spéciaux qui n'est pas spécifique à DWR ou même à J2EE, mais qui s'intègre bien à ces technologies.

Le site Web officiel de script.aculo.us *(http://script.aculo.us)* propose de nombreux exemples et démonstrations, dont beaucoup sont réalisées avec Ruby On Rails, un framework aujourd'hui très populaire, qui intègre script.aculo.us en standard.

Nous allons voir comment intégrer script.aculo.us à DWR dans le cadre des Tudu Lists afin de réaliser un site Web esthétiquement plus agréable.

Installation

Pour installer script.aculo.us, il faut en télécharger la dernière version et l'installer dans l'application Web comme n'importe quel fichier JavaScript. Pour Tudu Lists, nous avons choisi de l'installer dans le répertoire **WebContent/scripts/scriptaculous.** Afin de suivre les évolutions de cette bibliothèque, nous avons ajouté un fichier **version.txt**, qui stocke la version de la distribution utilisée.

Nous avons ensuite importé les deux fichiers principaux de script.aculo.us dans le header de Tudu Lists, le fichier **WEB-INF/jspf/header.jsp :**

```
<script type="text/javascript"
        src="${ctx}/scripts/scriptaculous/prototype.js">
</script>
<script type="text/javascript"
        src="${ctx}/scripts/scriptaculous/scriptaculous.js">
</script>
```

Le premier fichier, **prototype.js,** correspond à Prototype, un framework simplifiant le développement JavaScript sur lequel script.aculo.us est fondé.

Le deuxième fichier, **scriptaculous.js,** importe l'ensemble des fichiers nécessaires à l'utilisation de script.aculo.us.

Nous avons choisi de placer ces deux fichiers dans le header de Tudu Lists afin de simplifier le développement, sachant qu'une fois téléchargés ils seront stockés dans le cache du navigateur client. Les importer dans chaque page JSP n'est donc pas un handicapant en terme de charge serveur.

Utilisation des effets spéciaux

L'utilisation des effets fournis par script.aculo.us est très simple. En voici un exemple, provenant de la page de gestion des todos, **WEB-INF/jsp/todos.jsp :**

```
function renderMenu() {
  todos.getCurrentTodoLists(replyCurrentTodoLists);
  Effect.Appear("todoListsMenuBody");
}
```

La bibliothèque d'effets spéciaux s'utilise avec l'objet `Effect`, lequel possède un grand nombre d'effets, qui prennent en paramètre l'identifiant du composant HTML à impacter. Dans l'exemple ci-dessus, nous demandons un effet d'apparition sur le composant HTML `todoListsMenuBody`.

La liste complète de ces effets est disponible sur le site de script.aculo.us. Il existe notamment des effets de fondu (Effect.Fade) et de disparition (Effect.Puff), qui ressemblent à ce que nous trouvons dans Microsoft PowerPoint pour afficher ou faire disparaître des éléments de présentation.

Ces effets peuvent aussi être utilisés dans les événements JavaScript, comme dans l'exemple suivant :

```
<div onclick="new Effect.Fade(this)">
    Cliquez ici pour que cet élément disparaisse.
</div>
```

Ces effets spéciaux présentent les deux utilités majeures suivantes dans la construction d'une page AJAX :

- Ils permettent à l'utilisateur d'avoir une notification visuelle de ses actions. Les personnes habituées aux sites Web classiques peuvent être désorientées devant un site Web AJAX. Grâce à des effets tels que `Effect.Shake`, `Effect.Pulsate` ou `Effect.Highlight`, elles ont mieux conscience de ce qui se passe dans l'application.
- Ils permettent à l'utilisateur d'attendre que les fonctions AJAX s'exécutent. Lorsque nous éditons un todo, il est plus agréable d'avoir une boîte de dialogue qui s'affiche en douceur, avec les éléments déjà prérenseignés. Sans effet spécial, nous n'aurions le choix qu'entre un site semblant peu réactif, si nous recherchons les informations avant d'afficher la boîte de dialogue, ou un site avec des boîtes de dialogues vides se remplissant avec un temps de retard, dans le cas inverse.

Ces effets spéciaux sont donc très importants pour la qualité de l'expérience utilisateur.

Utilisation avancée

script.aculo.us permet de proposer des solutions qui n'étaient jusqu'à présent disponibles que dans les applications de type client lourd. Parmi ces fonctionnalités avancées, script.aculo.us propose le glisser-déplacer, les listes à trier graphiquement et les champs HTML qui se remplissent *via* des appels côté serveur.

Dans sa version 1.5, que nous utilisons dans le cadre de cet ouvrage, script.aculo.us s'intègre mal à DWR pour ces fonctionnalités avancées. C'est la raison pour laquelle nous allons les étudier en dehors de l'étude de cas et de DWR.

Édition de texte à chaud

Cette fonctionnalité permet de transformer une chaîne de caractères normale en un champ de saisie HTML. Nous pouvons ainsi modifier certains éléments d'une page en cliquant simplement dessus.

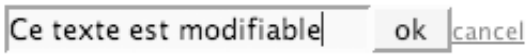
En voici un exemple de code :

```
<p id="texteModifiable">Ce texte est modifiable</p>
```

```
<script language="JavaScript">
 new Ajax.InPlaceEditor('texteModifiable',
  '${ctx}/tudu/example/saveText');
</script>
```

Le résultat de ce code est illustré à la figure 9.3.

Figure 9.3
Texte modifiable à chaud



La fonction `Ajax.InPlaceEditor` prend en premier paramètre l'identifiant du bloc HTML à modifier (un paragraphe dans notre exemple) et en second une URL qui sera chargée de traiter la requête. Cette fonction contourne donc DWR, comme indiqué précédemment.

L'URL appelée a pour rôle de sauvegarder le changement de texte. Elle reçoit en POST le paramètre `value`, contenant le nouveau texte à enregistrer, et doit renvoyer la chaîne de caractères à afficher. Cette dernière peut donc être différente du texte envoyé, si celui-ci n'est pas valide, par exemple. Une simple servlet est parfaitement adaptée pour ce type de traitement.

Création d'une liste déroulante dynamique

Cette fonctionnalité est souvent appelée Google Suggest, Google ayant popularisé ce type de liste déroulante dans une version bêta de son site.

La figure 9.4 en illustre un exemple avec une recherche sur le mot-clé « j2ee ».

Figure 9.4
Interface de Google Suggest

De même que la fonction précédente, cet exemple court-circuite DWR et s'utilise de la manière suivante :

```
<input type="text" id="utilisateurs" name="utilisateurs"/>
<div id="utilisateurs_autocomplete"></div>
```

```
<script language="JavaScript">
  new Ajax.Autocompleter("utilisateurs",
    "utilisateurs_autocomplete",
    "${ctx}/tudu/example/findUsers",
    {});
</script>
```

La fonction `Ajax.AutoCompleter` prend en premier paramètre l'identifiant de la zone de texte éditable, en deuxième paramètre l'identifiant du layer DHTML servant à afficher les valeurs dynamiques, en troisième paramètre l'URL chargée de traiter la requête et en quatrième paramètre des options.

L'URL traitant la requête va recevoir en méthode POST le contenu du champ HTML. La clé utilisée en paramètre est l'attribut `name` de la balise `input`. Cette URL devra retourner le contenu affiché dans le layer DHTML, nommé `utilisateurs_autocomplete` dans notre exemple, sous forme de liste HTML :

```
<ul>
  <li>j2ee</li>
  <li>j2ee tutorial</li>
</ul>
```

Cette fonctionnalité améliore l'utilisabilité du site, mais elle est également extrêmement gourmande en ressources. Nous ne pouvons que conseiller l'utilisation d'un cache côté serveur. Il existe plusieurs solutions adaptées pour ce type de besoin, en particulier Ehcache (*http://ehcache.sourceforge.net),* que nous présentons au chapitre 11, ou OSCache *(http://www.opensymphony.com/oscache/),* qui propose un filtre de servlet intéressant pour cette situation.

Si notre but est la création d'un vrai moteur de recherche, l'utilisation de bases de données n'est pas recommandée, car elles sont lentes et mal adaptées à ce type de traitement. L'utilisation de Lucene *(http://lucene.apache.org/)* paraît plus appropriée, d'autant que cette bibliothèque peut être intégrée à Spring. Nous avons pu rencontrer l'architecte d'un site Internet français à fort trafic, qui utilise la combinaison Spring/Lucene pour obtenir des temps de réponse inférieurs à 100 millisecondes par requête. De tels résultats rendent possible la création d'un site Web de type Google Suggest en utilisant uniquement des technologies Java Open Source.

Utilisation de Prototype

Prototype est une bibliothèque JavaScript, disponible à l'adresse *http://prototype.conio.net/.* Elle est utilisée en interne par script.aculo.us, car elle fournit un ensemble d'utilitaires simplifiant la manipulation de l'arbre DOM et les appels AJAX. Pour un développeur JavaScript, cette bibliothèque d'un investissement minime en apprentissage est certainement utile.

Ses fonctions les plus simples et les plus utiles, qui ne sont pas spécifiques à J2EE ou même à AJAX, sont détaillées dans les sections suivantes.

La fonction *$()*

La fonction `$()` est un raccourci vers la fonction JavaScript `document.getElementById()`, qui est très souvent utilisée en AJAX. En voici un exemple d'utilisation :

```
<div id="exempleDiv">
 <p>Bonjour!</p>
</div>

<script language="JavaScript">
 var exemple = $('exempleDiv');
 alert(exemple.innerHTML);
</script>
```

Cette fonction est capable de retourner un tableau d'éléments si nous lui donnons en paramètres plusieurs identifiants :

```
var exemples = $('exempleDiv', 'exempleDiv2', 'exempleDiv3');

for(i=0; i<exemples.length; i++) {
  alert(exemples[i].innerHTML);
}
```

La fonction *$F()*

Il s'agit là aussi d'un raccourci, capable de retrouver n'importe quel élément d'un formulaire HTML, quel que soit son type.

Cela donne avec l'exemple précédent :

```
<input type="text" id="exempleInput" value="Bonjour!"/>

<script language="JavaScript">
 var exemple = $F('exempleInput');
 alert(exemple);
</script>
```

La fonction *Try.these()*

Cette fonction est utile pour créer des sites Web robustes, capables d'afficher correctement des pages Web malgré les problèmes de compatibilité posés par le JavaScript. Nous en aurons certainement besoin si nous voulons utiliser AJAX sur un site disponible sur Internet.

La fonction `Try.these()` permet de tester plusieurs fonctions à la suite, jusqu'à ce que l'une d'elles marche correctement. En voici un exemple d'utilisation :

```
function exemple(){
  return Try.these(
    function() {
      // fonctionnement normal.
    },
```

```
    function() {
      // mode dégradé, utilisé si la première fonction a échoué
    )
  );
}
```

Conclusion

Après une introduction à AJAX et au Web 2.0, nous nous sommes arrêtés sur deux grandes bibliothèques JavaScript complémentaires, DWR et script.aculo.us. DWR permet d'utiliser côté client des Beans Spring provenant d'une application Java/J2EE en back-end. script.aculo.us permet d'ajouter des effets spéciaux aux pages Web. La combinaison de ces différentes technologies permet de développer des sites Web attractifs, radicalement différents de ceux d'avant 2005. C'est là un changement important en terme de fonctionnalités et d'utilisabilité, qu'il va être important de maîtriser dans les années à venir.

Nous avons vu qu'avec cette boîte à outils il n'était pas très difficile de créer des pages Web utilisant AJAX. Les difficultés rencontrées se trouvent plutôt du côté de l'utilisation d'AJAX, qui présente deux inconvénients : cette technique bombarde les serveurs de requêtes HTTP, mettant en danger leur montée en charge, et risque de troubler les utilisateurs habitués à des applications plus traditionnelles. À ces deux problèmes, nous avons vu qu'il existait des solutions : monitoring des performances et tuning pour le premier, utilisation d'effets spéciaux pour le second.

AJAX est une technique simple d'emploi et bien outillée. Nous ne saurions trop conseiller aux concepteurs de sites Web de l'ajouter à leur arsenal technique.

10

Support des portlets

Un portail est un site Web qui regroupe sur une même page un grand nombre d'informations intéressant ses utilisateurs. Ces informations provenant de canaux différents, les pages des portails gèrent différentes entités de présentation et de traitement autonomes, avec lesquelles elles interagissent. Ces solutions sont de plus en plus utilisées en entreprise afin d'offrir sur une même page les informations et services mis à disposition des utilisateurs.

J2EE adresse la problématique des applications de portail par le biais de la spécification portlet. Son composant de base, la portlet, présente des analogies avec celui de la spécification servlet, la servlet. Les portlets s'appuient d'ailleurs sur cette dernière spécification en l'enrichissant, afin de gérer plusieurs composants indépendants dans une même page Web.

À partir de la version 2.0, Spring fournit un support pour faciliter le développement d'applications fondées sur la technologie portlet. Il reprend et transpose à cet effet les mécanismes de Spring MVC.

Ce chapitre détaille ces mécanismes et composants de Spring MVC repris dans le support des portlets par Spring. Afin de faciliter la compréhension de ce support de la spécification portlet, nous reprenons volontairement le plan du chapitre 7, relatif à Spring MVC.

La spécification portlet

La spécification portlet offre un cadre de développement afin de créer des composants intégrables dans des portails. Comme expliqué précédemment, un portail permet de regrouper sur une même page Web plusieurs applications indépendantes, utilisant différentes sources de données.

Les outils de portail intègrent un conteneur de portlets afin de pouvoir utiliser les composants applicatifs fondés sur les portlets. Le conteneur de portlets le plus utilisé dans les portails est Pluto, de la fondation Apache, disponible à l'adresse *http://portals.apache.org/pluto/*. Il correspond à l'implémentation de référence de la spécification portlet.

Les portails les plus utilisés dans le monde Open Source sont JetSpeed, GridSphere et uPortal. Ces projets sont accessibles respectivement aux adresses *http://portals.apache.org/jetspeed-1/ et http://www.gridsphere.org/gridsphere/gridsphere* et *http://www.uportal.org/*.

La figure 10.1 donne un exemple de site utilisant un outil de portail. Les différentes zones correspondent à différentes portlets affichant divers canaux d'information.

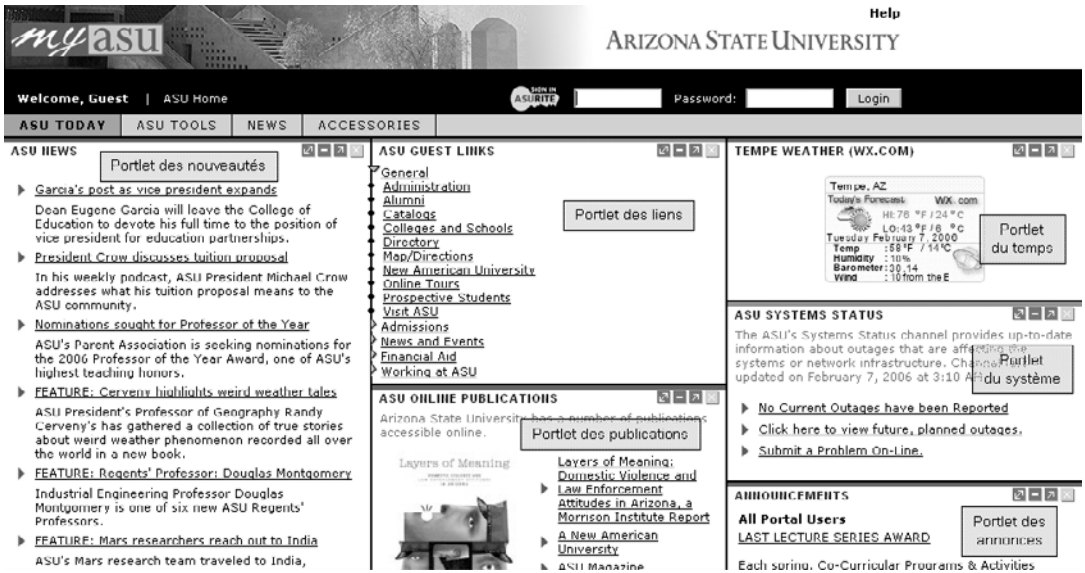


Figure 10.1
Portail de l'université de l'État d'Arizona utilisant uPortal

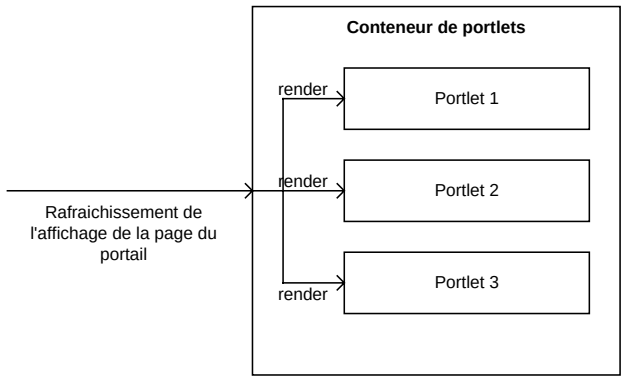
Du fait que cette spécification enrichit la spécification servlet, les applications fondées sur les portlets doivent être packagées sous forme d'archives WAR.

Les portlets définissent deux points d'entrée, dont le premier, `render`, adresse les problématiques de présentation. Il est appelé par le conteneur de portlets en cas de rafraîchissement global de l'affichage de la page du portail. Cet événement peut survenir suite à une

modification des données affichées par une portlet, ce qui entraîne un appel par ce point d'entrée de toutes les portlets affichées dans la page. De ce fait, des mécanismes de cache sont couramment mis en œuvre afin d'optimiser les temps de rafraîchissement de l'affichage des pages du portail.

La figure 10.2 illustre l'impact d'un rafraîchissement de l'affichage d'une page sur toutes les portlets qu'elle contient.

Figure 10.2
Mécanisme de rafraîchissement d'une page de portail



Le second point d'entrée, `processAction`, permet d'interagir avec une portlet afin d'effectuer des modifications, par exemple en utilisant des données envoyées par l'intermédiaire d'un formulaire. Ce point d'entrée n'adresse pas les problématiques de présentation des données et est donc immédiatement suivi par le rafraîchissement de la page du portail, par le biais du point d'entrée précédent.

Ce rafraîchissement est automatiquement déclenché par le conteneur de portlets, comme l'illustre la figure 10.3.

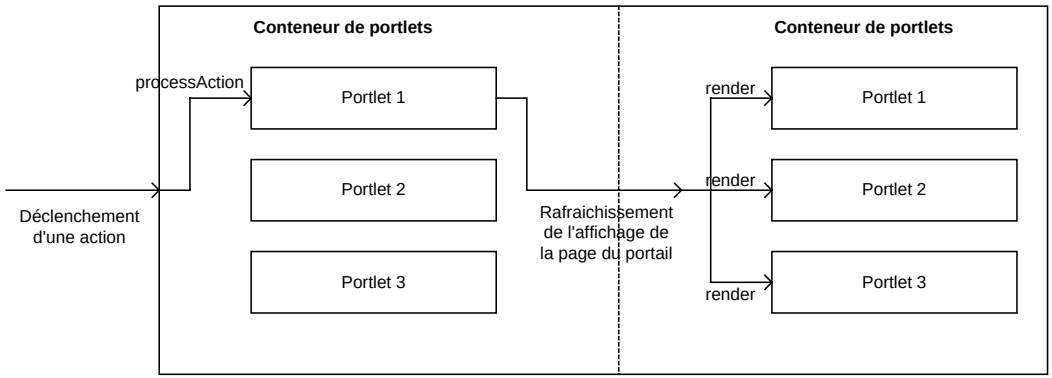


Figure 10.3
Mécanisme de traitement d'une action pour une portlet

L'interface `Portlet` du package `javax.portlet` comporte deux méthodes, correspondant aux deux points d'entrée précédemment décrits, comme le montre le code suivant :

```
public interface Portlet {
    void destroy();
    void init(PortletConfig config);

    void processAction(ActionRequest request,
                       ActionResponse response);
    void render(RenderRequest request,
                       RenderResponse response);
}
```

Les méthodes `init` et `destroy` adressent la gestion du cycle de vie du composant. La méthode `init` peut être implémentée afin d'ajouter des traitements d'initialisation fondés sur une instance de `PortletConfig`, qui matérialise la configuration de la portlet dans le fichier **portlet.xml,** localisé dans le répertoire **WEB-INF/.** La méthode `destroy` permet quant à elle de libérer les ressources utilisées avant que la portlet soit détruite.

Les points d'entrée de la portlet (méthodes `processAction` et `render`) prennent en paramètres des spécialisations des classes `PortletRequest` et `PortletResponse` afin de pouvoir interagir respectivement avec la requête et la réponse.

Notons l'existence de la classe `GenericPortlet` du package `javax.portlet`, similaire à la classe `GenericServlet` de la spécification servlet. Elle fournit des méthodes dédiées afin de gérer les différents modes d'affichage de la portlet.

Le code suivant montre l'implémentation d'une portlet fondée sur cette classe :

```
public class TodoPortlet extends GenericPortlet {
    public void init(PortletConfig config)
                throws PortletException, UnavailableException {
        super.init(config);
        String myParam = config.getInitParameter("myparam");
        (...)
    }

    (...)

    public void doView(RenderRequest request,
                       RenderResponse response)
                          throws PortletException, IOException {
        request.setContentType("text/html");
        // Récupération d'un todo à partir des paramètres de
        //la requête
        Todo todo=getTodo(request);
        request.setAttribute("todo",todo);

        // Affichage du todo dans une JSP
        PortletRequestDispatcher rd =
               getPortletContext().getRequestDispatcher(
                      "/todo.jsp");
        rd.include(rReq,rRes);
    }
}
```

Les applications fondées sur la technologie portlet ne peuvent utiliser directement les adresses pour la navigation. En effet, le conteneur de portlets utilisant sa propre stratégie de gestion des adresses, la spécification portlet impose l'utilisation des taglibs `actionURL` et `renderURL` afin d'interagir avec les différentes ressources gérées par le conteneur de portlets.

Le code suivant illustre leur utilisation dans une page JSP :

```
<%@ taglib prefix="portlet" uri="http://java.sun.com/portlet" %>

(...)

<!- Exécution d'une action pour la portlet -->
<form method="post" action="<portlet:actionURL>
      <portlet:param name="action" value="saveTodo"/>
      <portlet:param name="todoId" value="10"/>
    </portlet:actionURL>">
    (...)
</form>

<!-- Affichage de la portlet -->
<a href="<portlet:renderURL>
           <portlet:param name="action" value="showTodo"/>
           <portlet:param name="todoId" value="10"/>
         </portlet:renderURL>">Visualisation d'un Todo</a>
```

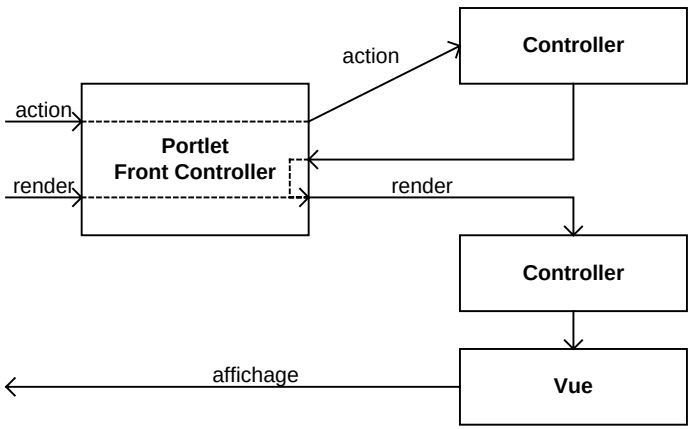
Le support des portlets de Spring

Le support des portlets de Spring implémente un framework MVC de type 2, qui s'appuie sur la technologie portlet afin d'adresser au mieux la résolution de ce pattern dans les applications de portail.

Dans le contexte de la technologie portlet, l'implémentation de ce pattern se divise en deux parties, traitant respectivement les requêtes de modification et celles d'affichage, comme l'illustre la figure 10.4.

Figure 10.4

Principe de fonctionnement du pattern MVC type 2 pour la technologie portlet



Le pattern MVC de type 2 reprend les concepts de base de Spring MVC, mais les adapte à la technologie portlet. Il utilise pour cela toutes les fonctionnalités du conteneur léger de Spring, notamment pour configurer les composants du MVC ainsi que leurs dépendances.

Les points clés de Spring MVC repris et adaptés pour le support des portlets sont les suivants :

- utilisation d'un contrôleur façade implémenté par une portlet et utilisant un sous-contexte de celui de l'application Web ;
- spécification de différentes stratégies d'aiguillage des requêtes vers les contrôleurs ;
- fonctionnalités permettant de réaliser le mappage de données dans des Beans ainsi que leur validation ;
- définition de différents types de contrôleurs afin de résoudre au mieux les différentes situations ;
- spécification de différentes stratégies de résolution des vues ;
- support de différents types de vues.

Dans la suite du chapitre, nous désignons le framework Web de Spring par Spring MVC et le framework portlet par support portlet.

Puisque le support portlet reprend les concepts de Spring MVC, sa prise en main est facile pour les utilisateurs maîtrisant ce framework. Le support portlet réutilise de fait différents composants de Spring MVC, notamment pour la gestion des vues.

La figure 10.5 illustre les différentes briques du support, avec en gris les parties de Spring MVC réutilisées.

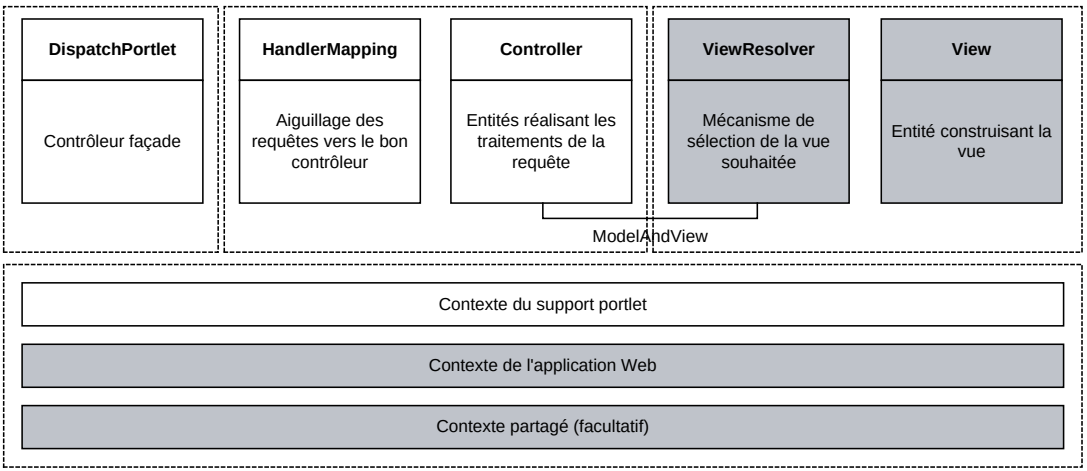


Figure 10.5

Entités de traitement des requêtes du support portlet

Initialisation du support portlet

L'initialisation du support portlet se réalise en deux parties et s'effectue essentiellement dans les fichiers **web.xml** et **portlet.xml,** localisés dans le répertoire **WEB-INF/.** Le framework utilise des mécanismes issus des spécifications servlet et portlet.

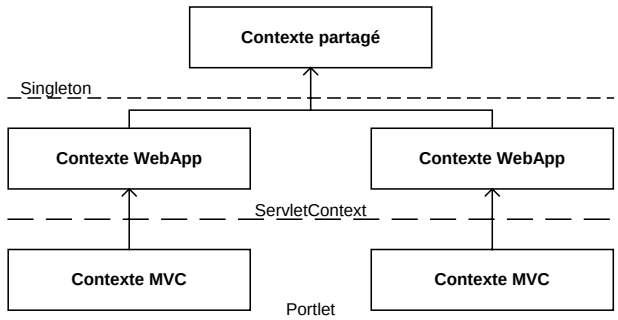
Gestion des contextes

Dans les applications utilisant la technologie portlet, les contextes sont gérés de la même manière que dans les technologies Web classiques.

Une hiérarchie de contextes est mise en œuvre, à la différence près que le contexte situé au bas de la hiérarchie correspond non plus à celui de Spring MVC, fondé sur les servlets, mais à celui du support portlet, comme l'illustre la figure 10.6.

Figure 10.6

Hiérarchie des contextes de Spring pour une application portlet



Le contexte partagé et celui de l'application Web sont chargés de la même manière qu'avec Spring MVC et Struts *(pour plus de détails, voir les sections concernées des chapitres 6 et 7).*

Le chargement du contexte MVC portlet est rattaché à la portlet principale du framework d'une manière analogue à Spring MVC. Le nom du fichier de configuration de ce contexte doit être spécifié par l'intermédiaire du paramètre d'initialisation `contextConfigLocation` pour la classe `DispatchPortlet`, comme le montre le code suivant, issu du fichier **portlet.xml :**

```
<portlet-app>
    <portlet>
        (...)
        <portlet-class>
            org.springframework.web.portlet.DispatcherPortlet
        </portlet-class>
        <init-param>
            <name>contextConfigLocation</name>
            <value>/WEB-INF/tudu-portlet.xml</value>
        </init-param>
        (...)
    </portlet>
</portlet-app>
```

Initialisation des entités de base

Contrairement à Spring MVC, le support portlet nécessite la mise en œuvre de deux entités afin de permettre respectivement l'accès à l'application et l'affichage des données.

Contrôleur façade

Le framework Spring MVC implémentant le pattern MVC de type 2, il doit mettre en œuvre un contrôleur façade afin de diriger les traitements vers des classes désignées par le terme `Controller` dans le support portlet et Spring MVC.

L'objectif du support est de transposer ce mécanisme pour la technologie portlet. De cette manière, le développement d'applications de portail suit les mêmes principes que ceux de Spring MVC.

De ce fait, le contrôleur façade n'est plus implémenté par une servlet mais par une portlet. Son type correspond désormais à la classe `DispatchPortlet` localisée dans le package `org.springframework.web.portlet`.

Le code suivant donne une configuration possible de cette entité dans le fichier **portlet.xml :**

```
<portlet-app>
    <portlet>
        <portlet-name>tudu</portlet-name>
        <portlet-class>
            org.springframework.web.portlet.DispatcherPortlet
        </portlet-class>
        <init-param>
            <name>contextConfigLocation</name>
            <value>/WEB-INF/tudu-portlet.xml</value>
        </init-param>
        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
            <portlet-mode>help</portlet-mode>
        </supports>
        <portlet-info>
            <title>Tudu</title>
        </portlet-info>
    </portlet>
</portlet-app>
```

Comme avec Spring MVC, il est possible de définir dans la même application plusieurs contrôleurs façade afin d'implémenter différents modules indépendants.

Gestion des vues

Pour mettre en œuvre les vues, le support portlet utilise une servlet dédiée, dont nous détaillons l'utilité dans la suite du chapitre.

Cette servlet est de type `ViewRendererServlet` pour le package `org.springframework.web.servlet` et est configurée dans le fichier **web.xml** de la manière suivante :

```
<web-app>
    (...)
```

```
    <servlet>
        <servlet-name>ViewRendererServlet</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.ViewRendererServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>ViewRendererServlet</servlet-name>
        <url-pattern>/WEB-INF/servlet/view</url-pattern>
    </servlet-mapping>
    (...)
</web-app>
```

La valeur spécifiée dans la balise `url-pattern` pour la servlet doit correspondre à celle utilisée par la portlet `DispatchPortlet`. Dans notre exemple, nous avons utilisé la valeur par défaut `/WEB-INF/servlet/view`. Il est cependant possible d'utiliser une autre valeur en utilisant la propriété d'initialisation `viewRendererUrl` de la portlet précédemment décrite.

Traitements des requêtes

Le support portlet de Spring met en œuvre des mécanismes similaires à ceux de Spring MVC afin de rediriger les requêtes vers la bonne entité de traitement.

Il doit pour cela prendre en compte les spécificités de la technologie portlet et ne peut s'appuyer sur les URL, à la différence de Spring MVC.

Sélection du contrôleur

Le support portlet de Spring offre une entité de base permettant de sélectionner le contrôleur à utiliser en fonction de la requête. L'interface correspond au type `HandlerMapping` du package `org.springframework.web.portlet`, telle que définie par le code suivant :

```
public interface HandlerMapping {
    HandlerExecutionChain getHandler(
                        PortletRequest request) throws Exception;
}
```

La différence principale avec Spring MVC vient du paramètre de la méthode `getHandler`, qui est désormais de type `PortletRequest`.

Le support dispose de deux implémentations de cette interface, localisées dans le package `org.springframework.web.portlet.handler`.

Contrairement aux servlets, l'aiguillage ne peut s'appuyer sur une adresse d'accès pour déterminer le contrôleur puisque le conteneur de portlets a la responsabilité de les gérer en interne. L'adresse ne peut donc constituer un critère de sélection.

Le support portlet peut cependant utiliser les modes d'affichage d'une portlet et/ou un paramètre afin de déterminer le contrôleur impacté. Le support portlet définit pour cela deux implémentations de l'interface `HandlerMapping`. La première utilise uniquement le mode comme moyen de sélection. Cette implémentation correspond à la classe `PortletModeHandlerMapping` et se configure par l'intermédiaire d'une table de hachage établissant la correspondance entre les modes et les contrôleurs.

Le code suivant, issu du fichier **tudu-portlet.xml,** localisé dans le répertoire **WEB-INF/context/portlet/,** donne un exemple de mise en œuvre de cette première implémentation :

```
<beans>
    (...)
    <bean id="portletModeHandlerMapping" class="org.springframework
                  .web.portlet.handler.PortletModeHandlerMapping">
    <property name="portletModeMap">
       <map>
           <entry key="view">
                    <ref bean="showTodoListsController"/></entry>
           <entry key="help">
                    <ref bean="showTodoListsController"/></entry>
       </map>
    </property>
    </bean>
    (...)
</beans>
```

La seconde implémentation est plus fine et combine le mode avec un paramètre de requête afin de sélectionner le contrôleur. Elle correspond à la classe `PortletModeParameterHandlerMapping` et se configure par le biais de deux tables de hachage imbriquées, comme dans le code suivant, tiré du même fichier que précédemment :

```
<beans>
    (...)
    <bean id="portletModeParamHandlerMapping" class="org
                              .springframework.web.portlet.handler
                              .PortletModeParameterHandlerMapping">
        <property name="portletModeParameterMap">
            <map>
            <entry key="view">
                 <map>
                     <entry key="showTodoLists">
                         <ref bean="showTodoListsController"/></entry>
                     <entry key="editTodoList">
                         <ref bean="editTodoListController"/></entry>
                     (...)
                 </map>
             </entry>
           </map>
        </property>
    </bean>
    (...)
</beans>
```

Le support portlet offre la possibilité d'utiliser conjointement plusieurs implémentations de l'interface `HandlerMapping` en mettant en œuvre un mécanisme de chaînage de `HandlerMapping`. Cette fonctionnalité reprend les principes de Spring MVC pour les `ViewResolver` en les adaptant pour l'aiguillage des requêtes.

La configuration de l'ordre des implémentations de `HandlerMapping` se réalise par le biais de la propriété `order`, comme dans le code suivant :

```
<beans>
    (...)
    <bean id="portletModeHandlerMapping" class="org.springframework
                  .web.portlet.handler.PortletModeHandlerMapping">
        <property name="order" value="20"/>
        (...)
    </bean>

    <bean id="portletModeParamHandlerMapping" class="org
                              .springframework.web.portlet.handler
                              .PortletModeParameterHandlerMapping">
        <property name="order" value="10"/>
        (...)
    </bean>
    (...)
</beans>
```

Interception des requêtes

Dans le support portlet, les interceptions se fondent sur les mêmes principes que dans Spring MVC. L'entité de base des contrôleurs suit la même structure que l'interface `HandlerInterceptor`, localisée dans le package `org.springframework.web.servlet`. La seule différence provient des paramètres des différentes méthodes, puisque la technologie portlet possède ses propres entités de représentation des requêtes et des réponses.

L'entité de base des intercepteurs pour le support portlet est l'interface `HandlerInterceptor`, localisée désormais dans le package `org.springframework.web.portlet`, comme le montre le code suivant :

```
public interface HandlerInterceptor {
    boolean preHandle(PortletRequest request,
       PortletResponse response, Object handler) throws Exception;
    void postHandle(RenderRequest request, RenderResponse response,
       Object handler, ModelAndView modelAndView) throws Exception;
    void afterCompletion(PortletRequest request,
       PortletResponse response, Object handler, Exception ex)
                                                  throws Exception;
}
```

La classe `HandlerInterceptorAdapter` du package `org.springframework.web.portlet.handler` facilite l'utilisation de cette interface en fournissant une implémentation par défaut fondée sur le pattern adaptateur.

Comme dans Spring MVC, la configuration des intercepteurs se réalise au niveau du paramétrage des différentes implémentations de `HandlerMapping`, comme le montre le code suivant, tiré du fichier **tudu-portlet.xml :**

```
<beans>
    (...)
    <bean id="parameterMappingInterceptor" class="org
                              .springframework.web.portlet.handler
                              .ParameterMappingInterceptor"/>
    <bean id="portletModeParamHandlerMapping" class="org
                              .springframework.web.portlet.handler
                              .PortletModeParameterHandlerMapping">
        (...)
        <property name="interceptors">
            <list>
                <ref bean="parameterMappingInterceptor"/>
            </list>
        </property>
        (...)
    </bean>
    (...)
</beans>
```

Le support fournit également l'intercepteur `ParameterMappingInterceptor` afin d'appeler automatiquement la méthode `setRenderParameter` de l'instance de la classe `ActionResponse`. Ce mécanisme est particulièrement intéressant en cas d'exécution d'actions par les contrôleurs afin que le même contrôleur soit utilisé pour afficher le résultat de la modification.

Dans ce contexte de passage d'une requête d'action à une requête d'affichage, notons les intéressantes possibilités fournies par la classe utilitaire `PortletUtils` du package `org.springframework.web.portlet.util`.

Les différents types de contrôleurs

Le support portlet définit plusieurs implémentations de contrôleurs afin d'adresser différents cas d'utilisation, tels que la gestion de la soumission des formulaires.

Bien que ce support reprenne les mécanismes de Spring MVC, le panel de types de contrôleurs offerts est bien moins riche que celui du framework.

Les contrôleurs simples

Le support portlet s'appuie sur une interface de base, nommée `Controller` et localisée dans le package `org.springframework.web.portlet.mvc`. Cette dernière est similaire à celle de Spring MVC, son unique différence provenant de son adhérence à la technologie portlet.

Cette interface définit deux points d'entrée, comme dans le code suivant :

```
public interface Controller {
    ModelAndView handleRenderRequest(RenderRequest request,
                          RenderResponse response) throws Exception;
```

```
    void handleActionRequest(ActionRequest request,
                        ActionResponse response) throws Exception;
}
```

La première méthode, `handleRenderRequest`, réalise les traitements d'affichage et est appelée en cas de rafraîchissement de l'affichage. Elle utilise en paramètre des objets implémentant `RenderRequest` et `RenderResponse`. Elle renvoie un objet de type `ModelAndView`, localisé dans le package `org.springframework.web.portlet` afin de sélectionner la vue à utiliser et de spécifier le contenu du modèle à utiliser.

La seconde méthode, `handleActionRequest`, prend en compte les traitements de modification et est appelée lors d'une action. Elle utilise des objets implémentant `ActionRequest` et `ActionResponse` mais ne possède pas de type de retour, contrairement à la méthode précédente.

Lors de l'utilisation de cette seconde méthode, une ligne de traitement peut être ajoutée afin de spécifier le paramètre `action`, qui permet de sélectionner le contrôleur à utiliser afin de rafraîchir l'affichage de la portlet.

Le code suivant donne un exemple de mise en œuvre de ce mécanisme :

```
    public void handleActionRequest(ActionRequest request,
                        ActionResponse response) throws Exception {
    (...)
    response.setRenderParameter("action","showTodos");
}
```

Le support portlet offre également une classe de base abstraite afin de faciliter l'implémentation des contrôleurs simples. Elle se nomme `AbstractController` et est localisée dans le package `org.springframework.web.portlet.mvc`. Le développeur doit alors utiliser les méthodes `handleActionRequestInternal` et `handleRenderRequestInternal`, qui correspondent respectivement aux méthodes `handleActionRequest` et `handleRenderRequest` décrites précédemment. Cette classe offre des supports relatifs au contrôle des sessions et au cache.

Le code suivant donne un exemple d'utilisation d'un contrôleur qui l'utilise :

```
public class ShowTodoListsController extends AbstractController {
    (...)

    public ModelAndView handleRenderRequestInternal(
                                RenderRequest request,
                                RenderResponse response)
                                        throws Exception {
        User user = userManager.findUser(getUser());
        Collection<TodoList> todoLists = user.getTodoLists();
        return new ModelAndView("todolists","todolists",todoLists);
    }

    (...)
}
```

Les contrôleurs de gestion de formulaire

Le support portlet met en œuvre les mêmes mécanismes que Spring MVC pour afficher les données des formulaires et les traiter. Il s'appuie pour cela sur l'implémentation `SimpleFormController`, localisée dans le package `org.springframework.web.portlet`.

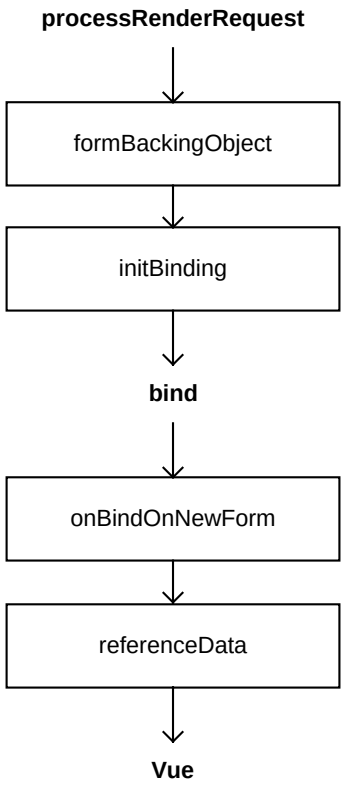
Ce contrôleur doit prendre en compte les spécificités de la technologie portlet en matière de rafraîchissement de l'affichage et de récupération des données de formulaire en vue d'un traitement.

Affichage du formulaire

Le support portlet offre le même cycle de traitements quant à l'affichage du formulaire que Spring MVC. Tous les traitements sont réalisés au cours d'une unique requête, de type `render` pour le contrôleur.

La figure 10.7 illustre les différentes étapes du cycle de traitements permettant d'afficher le formulaire.

Figure 10.7
Enchaînement des méthodes d'affichage des données d'un formulaire



Les noms de méthodes sont les mêmes que ceux utilisés par Spring MVC et correspondent aux mêmes problématiques *(pour plus de détail sur ces méthodes, voir le chapitre 7,*

relatif à Spring MVC). Leurs signatures peuvent cependant légèrement différer, du fait des spécificités de la technologie portlet.

Le code suivant illustre la mise en œuvre de la partie du contrôleur permettant d'afficher un formulaire :

```
public class EditTodoListController extends SimpleFormController {
    (...)

    protected void initBinder(PortletRequest request,
              PortletRequestDataBinder binder) throws Exception {
        (...)
    }

    protected Object formBackingObject(
                        PortletRequest request) throws Exception {
        String listId = request.getParameter("listId");
        return todoListsManager.findTodoList(listId);
    }

    protected Map referenceData(PortletRequest request,
                    Object cms, Errors errors) throws Exception {
        (...)
    }

    (...)
}
```

Soumission du formulaire

L'enchaînement des traitements de gestion de la soumission d'un formulaire est quasiment identique à celle de Spring MVC, à l'exception du nom de la méthode `onSubmit`, qui devient `onSubmitAction` avec le support portlet, et de la façon d'afficher les vues par le biais d'une requête de type `render`.

La soumission se réalise en deux étapes : un premier appel à un contrôleur pour traiter les données soumises et un second appel au même ou à un autre contrôleur pour l'affichage du résultat *(voir figure 10.8).*

Le code suivant illustre la mise en œuvre de ces mécanismes pour traiter les données du formulaire :

```
public class EditTodoListController extends SimpleFormController {
    (...)

    public void onSubmitAction(ActionRequest request,
                          ActionResponse response,Object command,
                          BindException errors) throws Exception {
        TodoList todoList = (TodoList)command;
        todoListsManager.updateTodoList(todoList);
    }
```

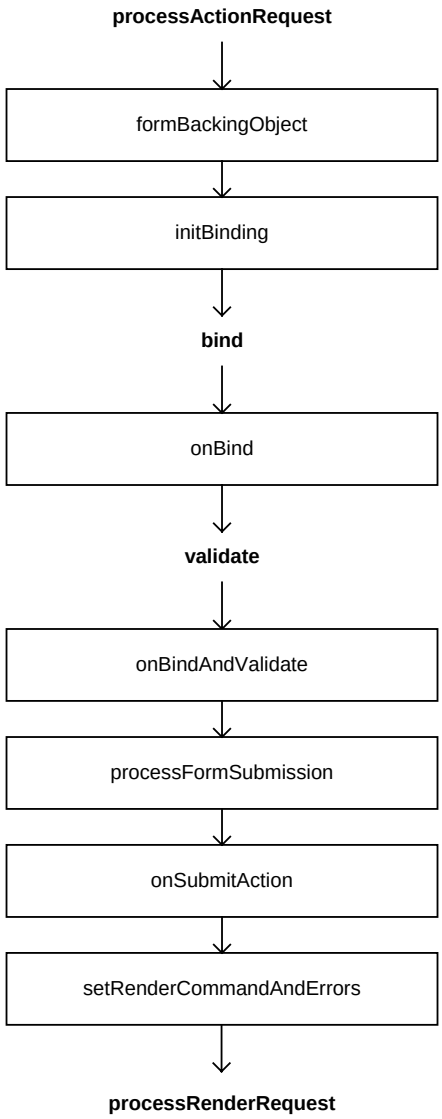


Figure 10.8
Enchaînement des méthodes de soumission des données d'un formulaire

En cas de succès des traitements d'une soumission de données de formulaire, le contrôleur laisse la main au développeur afin de spécifier le contrôleur qui doit poursuivre les traitements. Si rien n'est spécifié, le contrôleur par défaut est utilisé, mais il est possible d'en spécifier un explicitement par l'intermédiaire de la méthode `setRenderParameter` de la classe `PortletResponse`, comme dans le code suivant :

```
public void onSubmitAction(ActionRequest request,
                           ActionResponse response,
                           Object command,
```

```
                            BindException errors) throws Exception {
    TodoList todoList = (TodoList)command;
    todoListsManager.updateTodoList(todoList);
    response.setRenderParameter("action","editTodoList");
}
```

L'intercepteur `ParameterMappingInterceptor` peut également être mis en œuvre à cet effet.

Gestion des exceptions

Le support portlet de Spring reprend les mêmes principes que Spring MVC ainsi que les mêmes noms pour les entités relatives. Les interfaces de Spring MVC ne peuvent toutefois être reprises en l'état, puisque les conteneurs de portlets imposent respectivement l'utilisation des interfaces `PortletRequest` et `PortletResponse` (et leurs dérivées) alors que les conteneurs de servlets s'appuient sur les interfaces `ServletRequest` et `ServletResponse` (et leurs dérivées).

Par défaut, les exceptions sont remontées directement dans le conteneur de portlets, mais il est possible de modifier ce comportement par l'intermédiaire de l'interface `HandlerExceptionResolver`, localisée dans le package `org.springframework.web.portlet`, comme le montre le code suivant :

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

Le développeur peut choisir d'utiliser ses propres implémentations ou de mettre en œuvre la classe `SimpleMappingExceptionResolver` du package `org.springframework.web.portlet.handler` fournie par le framework. Cette dernière permet de configurer les vues à utiliser en fonction des exceptions levées. L'exception est alors stockée dans la requête avec la clé `exception`, la rendant ainsi disponible pour un éventuel affichage.

Ce mécanisme se configure de la manière suivante dans le fichier **tudu-portlet.xml :**

```
<bean id="exceptionResolver" class="org.springframework.web
                .portlet.handler.SimpleMappingExceptionResolver">
    <property name="defaultErrorView" value="defError"/>
    <property name="exceptionMappings">
        <props>
            <prop key="javax.portlet.PortletSecurityException">
                notAuthorized
            </prop>
            <prop key="javax.portlet.UnavailableException">
                notAvailable
            </prop>
        </props>
    </property>
</bean>
```

Gestion de la vue

La gestion de la vue reprend entièrement les mécanismes de Spring MVC, en utilisant les interfaces `ViewResolver` et `View` du package `org.springframework.web.servlet.view` ainsi que leurs implémentations. La configuration des vues se réalise donc de la même manière que dans Spring MVC.

Le code suivant donne un exemple de configuration tiré de Tudu Lists et fondé sur les technologies JSP et JSTL :

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

La figure 10.9 illustre l'interaction entre la portlet `DispatchPortlet` et la servlet `ViewRendererServlet`.

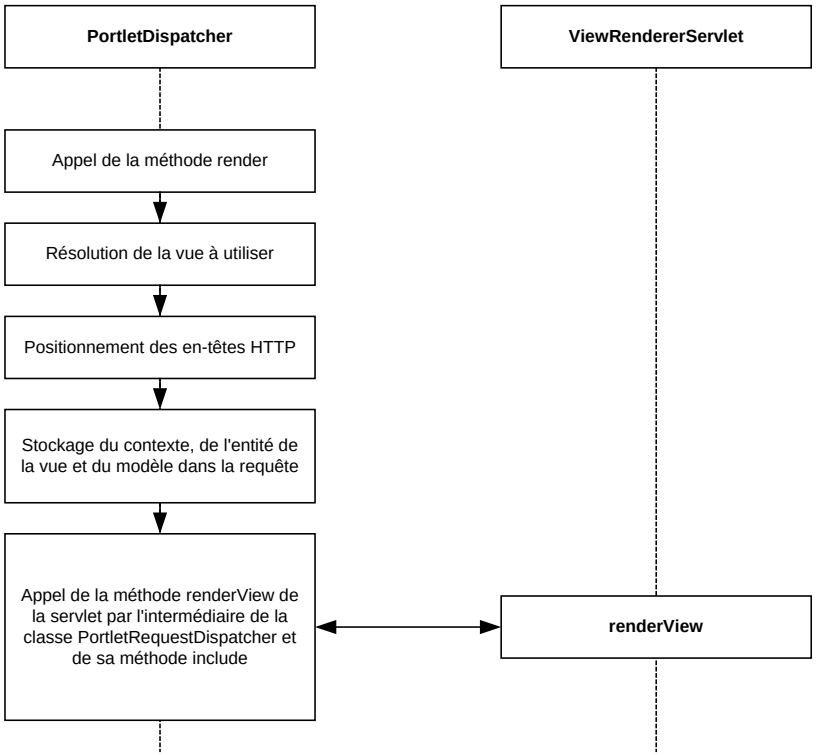


Figure 10.9
Mécanismes de traitement de la vue

Comme le support portlet réutilise les mécanismes et classes de Spring MVC relatives à la gestion des vues, l'exécution de la vue sélectionnée doit être exécutée en dehors de la classe `DispatcherPortlet`, puisque l'implémentation de la vue attend en paramètre des instances de `HttpServletRequest` et `HttpServletResponse`. Cette portlet résout la vue à utiliser puis délègue les traitements d'exécution de la vue à la servlet `ViewRendererServlet`, développée spécifiquement à cet effet.

Notons que les traitements de résolution de la vue se déroulent quand même dans la portlet `DispatchPortlet`. Cette dernière stocke ensuite dans la requête le contexte MVC relatif à la portlet, ainsi que l'instance de la vue sélectionnée et les éléments du modèle, afin de les rendre disponibles dans la servlet précédemment citée.

Pour finir, remarquons que toutes les taglibs de Spring MVC sont utilisables par le support portlet, notamment ceux relatifs à la gestion des formulaires.

Tudu Lists : utilisation d'un conteneur de portlets

Dans le cadre de notre étude de cas, nous décidons de migrer une partie de l'application Tudu Lists afin de la faire fonctionner dans un portail ou un conteneur de portlets. Pour cela, nous utilisons un projet spécifique, `Tudu-Portlet`.

Pour simplifier l'application et rendre l'exemple plus simple, l'application n'utilise ni la technologie AJAX, ni Acegi Security. Nous repartons de l'application fondée sur Spring MVC afin de réutiliser au maximum son code.

L'application migrée met en œuvre les fonctionnalités d'affichage et de modification de todos dans une portlet en réutilisant les composants services métier et d'accès aux données de Tudu Lists.

Configuration de l'application

L'application migrée nécessite la configuration des différents contextes de Spring, ainsi que la mise en œuvre d'une servlet dédiée afin d'afficher la présentation, en réutilisant les composants de Spring MVC.

Les contextes

Le premier contexte correspond à celui de l'application Web. Il est chargé par l'intermédiaire de la classe `ContextLoaderListener` et est configuré avec les fichiers dont le nom correspond à **/WEB-INF/applicationContext*.xml.**

Ce contexte est configuré de la même manière que dans Spring MVC dans le fichier **web.xml** du répertoire **WEB-INF/**, comme le montre le code suivant :

```
<web-app id="WebApp_ID">
    <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>
    (...)
</web-app>
```

Le second contexte est spécifique du support portlet. Il est configuré et chargé par le biais de la portlet `DispatchPortlet`. Le fichier utilisé à cet effet se nomme **portlet.xml** et est localisé dans le même répertoire que précédemment. Le code de la configuration relative à ce contexte est le suivant :

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

<portlet-app
     xmlns="http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
     xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
     xsi:schemaLocation="
            http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd
            http://java.sun.com/xml/ns/portlet/portlet-app_1_0.xsd"
     version="1.0">

    <portlet>
        <portlet-name>tudu</portlet-name>
        <portlet-class>
            org.springframework.web.portlet.DispatcherPortlet
        </portlet-class>
        <init-param>
            <name>contextConfigLocation</name>
            <value>/WEB-INF/tudu-portlet.xml</value>
        </init-param>
        <supports>
            <mime-type>text/html</mime-type>
            <portlet-mode>view</portlet-mode>
            <portlet-mode>help</portlet-mode>
        </supports>
        <portlet-info>
            <title>Tudu</title>
        </portlet-info>
    </portlet>
</portlet-app>
```

La servlet de présentation

Comme expliqué précédemment, le support portlet nécessite la mise en œuvre de la servlet `ViewRendererServlet` afin de réutiliser les mécanismes de gestion des vues de Spring MVC.

Cette servlet doit être configurée dans le fichier **web.xml** précédemment cité, comme indiqué dans le code suivant :

```
<web-app id="WebApp_ID">
    (...)

    <servlet>
        <servlet-name>ViewRendererServlet</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.ViewRendererServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    (...)

    <servlet-mapping>
        <servlet-name>ViewRendererServlet</servlet-name>
        <url-pattern>/WEB-INF/servlet/view</url-pattern>
    </servlet-mapping>
</web-app>
```

Implémentation des contrôleurs

Les différents contrôleurs du support portlet de Spring permettent d'afficher et de modifier des listes de todos et des todos et de réaliser aussi bien un affichage de données qu'une gestion de formulaires.

Le contrôleur simple

Tudu Lists utilise un contrôleur simple pour afficher l'ensemble des listes des todos pour un utilisateur. Il est implémenté par l'intermédiaire de la classe `ShowTodoListsController` du package `tudu.web` et s'appuie sur la classe `AbstractController`, comme le montre le code suivant :

```
public class ShowTodoListsController extends AbstractController {

    private UserManager userManager;
    private TodoListsManager todoListsManager;

    public ModelAndView handleRenderRequestInternal(
              RenderRequest request, RenderResponse response)
                                                throws Exception {
        User user = userManager.findUser(getUser());
        Collection<TodoList> todoLists = user.getTodoLists();
        return new ModelAndView("todolists","todolists",todoLists);
    }

    (...)
}
```

Ce contrôleur correspond à l'identifiant `showTodoLists` et est configuré pour être accessible en mode « view ». Il correspond également au contrôleur par défaut utilisé par la portlet.

Le mode d'accès ainsi que le contrôleur sont configurés dans le fichier **tudu-portlet.xml :**

```
<bean id="showTodoListsController"
      class="tudu.web.ShowTodoListsController">
    <property name="userManager" ref="userManager"/>
    <property name="todoListsManager" ref="todoListsManager"/>
</bean>

(...)

<bean id="portletModeParameterHandlerMapping"
              class="org.springframework.web.portlet
                     .handler.PortletModeParameterHandlerMapping">
    <property name="order" value="10"/>
    <property name="portletModeParameterMap">
        <map>
            <entry key="view">
                <map>
                    <entry key="showTodoLists">
                        <ref bean="showTodoListsController"/>
                    </entry>
                    (...)
                </map>
            </entry>
        </map>
    </property>
</bean>

<bean id="portletModeHandlerMapping" class="org.springframework
                  .web.portlet.handler.PortletModeHandlerMapping">
    <property name="order" value="20"/>
    <property name="portletModeMap">
        <map>
            <entry key="view">
                <ref bean="showTodoListsController"/></entry>
        </map>
    </property>
</bean>
```

Le contrôleur utilise la vue d'identifiant `todolists`. Cette dernière est résolue par l'intermédiaire de la classe `InternalResourceViewResolver`, implémentation de l'interface `ViewResolver`, et correspond donc à la page JSP **todolists.jsp** localisée dans le répertoire **WEB-INF/jsp.**

Le code suivant illustre le contenu de cette page, qui permet d'afficher la liste des listes de todos et définit les liens vers les pages de détail :

```
<%@ page contentType="text/html" isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="portlet" uri="http://java.sun.com/portlet" %>
```

```
List of todolists:<br/><br/>
<c:forEach items="${todolists}" var="todolist">
  - <a href="<portlet:renderURL>
                <portlet:param name="action" value="showTodos"/>
                <portlet:param name="id"
                               value="${todolist.listId}"/>
            </portlet:renderURL>">
       <c:out value="${todolist.listId}"/>
    </a>: <c:out value="${todolist.name}"/><br/><br/>
</c:forEach>
```

Le contrôleur de gestion de formulaire

La mise en œuvre d'une entité de gestion de formulaire comporte les mêmes étapes qu'avec Spring MVC.

Rappelons les différentes tâches à réaliser :

- étendre la classe `SimpleFormController` (repère ❶) ;
- surcharger la méthode `formBackingObject` afin de charger les données du formulaire (repère ❷) ;
- redéfinir la méthode `onSubmitAction` afin de traiter les données soumises par le formulaire (repère ❸) avec la méthode HTTP POST par l'intermédiaire d'une requête de type `action` ;
- configurer le contrôleur ainsi que les vues utilisées.

Le code suivant montre l'implémentation du contrôleur de formulaire `EditTodoListController` du package `tudu.web`, qui permet de modifier les données d'une liste de todos :

```
public class EditTodoListController
                          extends SimpleFormController {←❶
    (...)

    public void onSubmitAction(ActionRequest request,
                       ActionResponse response,Object command,
                       BindException errors) throws Exception {←❸
        TodoList todoList = (TodoList)command;
        todoListsManager.updateTodoList(todoList);
    }

    protected Object formBackingObject(
                      PortletRequest request) throws Exception {←❷
        String listId = request.getParameter("listId");
        return todoListsManager.findTodoList(listId);
    }

    (...)
}
```

Ce contrôleur doit être configuré dans le fichier **tudu-portlet.xml** afin de spécifier ses dépendances ainsi que ses propriétés relatives au MVC, comme le montre le code suivant :

```
<bean id="editTodoListController"
      class="tudu.web.EditTodoListController">
    <property name="todoListsManager" ref="todoListsManager"/>
    <property name="commandName" value="todolist"/>
    <property name="commandClass"
              value="tudu.domain.model.TodoList"/>
    <property name="formView" value="editTodoList"/>
</bean>
```

Une entrée doit également être ajoutée au Bean `portletModeParameterHandlerMapping` afin que le contrôleur puisse être accédé à partir d'une page JSP :

```
<bean id="parameterMappingInterceptor" class="org.springframework
               .web.portlet.handler.ParameterMappingInterceptor"/>

<bean id="portletModeParameterHandlerMapping" class="org
                            .springframework.web.portlet.handler
                            .PortletModeParameterHandlerMapping">
    <property name="order" value="10"/>
    <property name="interceptors">
        <list><ref bean="parameterMappingInterceptor"/></list>
    </property>
    <property name="portletModeParameterMap">
        <map>
            <entry key="view">
                <map>
                    (...)
                    <entry key="editTodoList">
                        <ref bean="editTodoListController"/>
                    </entry>
                    (...)
                </map>
            </entry>
        </map>
    </property>
</bean>
```

La mise en œuvre de l'intercepteur `ParameterMappingInterceptor` permet d'utiliser le même contrôleur afin de construire l'affichage de la portlet après un traitement correct des données soumises par le formulaire.

Notons qu'il serait également possible de mettre en œuvre les mécanismes de validation des données du formulaire de la même manière qu'avec Spring MVC en utilisant l'abstraction `Validator`.

Le contrôleur utilise l'identifiant de vue `editTodoList` pour afficher le formulaire. Cette vue est résolue par le même `ViewResolver` que précédemment et correspond donc à la page JSP **editTodoList.jsp** localisée dans le répertoire **WEB-INF/jsp/.**

Comme avec Spring MVC, le support portlet met en œuvre le taglib `bind` afin de remplir les champs du formulaire et d'afficher les éventuelles erreurs de validation, comme dans le code suivant :

```
<%@ page contentType="text/html" isELIgnored="false" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<%@ taglib prefix="fn" uri="http://java.sun.com/jsp/jstl/functions" %>
<%@ taglib prefix="spring"
           uri="http://www.springframework.org/tags" %>

<%@ taglib prefix="portlet" uri="http://java.sun.com/portlet" %>

<form method="post" action="<portlet:actionURL>
          <portlet:param name="action" value="editTodoList"/>
          <portlet:param name="listId" value="${todolist.listId}"/>
       </portlet:actionURL>">

    Id : <c:out value="${todolist.listId}"/><br/>
    Name : <spring:bind path="todolist.name">
       <input type="text" name="name"
              value="<c:out value="${status.value}"/>"
              size="25" maxlength="60"/>
        <font color="red">
       <c:out value="${status.errorMessage}"/></font>
    </spring:bind>

    <br/><br/>

    <input type="submit" value="Save"/>

</form>
```

Conclusion

Le support des portlets de Spring offre une solution innovante pour développer des applications de portail en offrant une implémentation MVC. Il permet l'utilisation du conteneur léger au cœur de ce type d'application afin de gérer les contrôleurs relatifs aux portlets et de les relier aux composants des différentes couches (services métier et accès aux données).

Pour sélectionner le contrôleur à utiliser et gérer les vues, il reprend les concepts fondamentaux de Spring MVC. Cela favorise le découplage des composants contrôleur et des composants de vue. De plus, il réutilise les mécanismes et entités de ce framework ainsi que sa façon de gérer les contextes applicatifs.

De ce fait, la prise en main du support par un développeur connaissant Spring MVC et la technologie portlet est aisée. Spring offre d'ailleurs un support Web cohérent grâce à son framework MVC servlet, Spring Web Flow, et son support portlet. Les principes et mécanismes mis en œuvre ainsi que la gestion des contextes sont ainsi homogènes et cohérents. Spring Web Flow offre également un support afin d'utiliser son moteur d'exécution.

Le tableau 10.1 récapitule les différentes entités du support portlet en les mettant en correspondance avec celles de Spring MVC..

Tableau 10.1. Correspondance des entités du support portlet et de celles de Spring MVC

Entité	Spring MVC	Support portlet
Package de base	`org.springframework.web.servlet`	`org.springframework.web.portlet`
Contrôleur façade	`ServletDispatcher`	`PortletDispatcher`
Sélection des contrôleurs	`HandlerMapping` du package `org.springframework.web.servlet`	`HandlerMapping` du package `org.springframework.web.portlet`
Mappage de données dans des Beans	Oui	Oui
Chaînage de `HandlerMapping`	Non	Oui
Sélection des vues	`ViewResolver` du package `org.springframework.web.servlet.view`	`ViewResolver` du package `org.springframework.web.servlet.view`
Chaînage de `ViewResolver`	Oui	Oui

Partie III

Gestion des données

La gestion des données couvre deux sujets : la persistance des données et la gestion des transactions. Nous étudions dans cette partie de quelle manière stocker des données de manière fiable et pérenne, en particulier à l'aide d'une base de données relationnelle.

Les deux chapitres qui la composent traitent des bases conceptuelles de chaque sujet et présentent les bonnes pratiques et le vocabulaire généralement utilisés dans chaque domaine. Nous verrons ainsi ce qu'est le mappage objet/relationnel et ce que signifie une transaction ACID.

Spring simplifie l'utilisation de ces deux domaines par rapport aux standards J2EE et permet d'avoir des applications mieux découpées, plus simples et plus rapides à développer. Nous verrons en particulier l'utilisation d'Hibernate avec Spring, ainsi que celle de la POA afin de gérer les transactions de manière transversale.

11

Persistance des données

La persistance des données est généralement une problématique majeure pour les applications J2EE, car les données manipulées par une application sont souvent d'une importance stratégique.

La principale technologie de persistance utilisée est celle des bases de données relationnelles. Les rares solutions concurrentes, comme les bases de données orientées objet, sont peu utilisées, et nous ne nous attarderons pas sur elles.

Les bases de données relationnelles permettent de traiter et de stocker de larges volumes de données, et ce d'une manière totalement indépendante du langage de programmation utilisé pour accéder à ces données (Java, PHP, C#, etc.). Cette ouverture est une des raisons principales du succès de cette technologie.

Cela a toutefois un prix pour le programmeur Java, puisque ce dernier est contraint de faire cohabiter deux mondes conceptuellement différents, le monde objet et le monde relationnel. En conséquence, les applications Java s'appuyant sur des bases de données relationnelles sont très souvent lourdes à coder, difficiles à faire évoluer et peu performantes. Cela provient essentiellement d'une utilisation abusive de JDBC (Java DataBase Connectivity), la technologie Java standard de bas niveau pour accéder aux bases de données.

Ce chapitre commence par une introduction aux bonnes pratiques classiques en terme de persistance des données et introduit les concepts fondamentaux généralement admis dans ce domaine.

Dans un deuxième temps, nous étudions les principales solutions de mapping objet/relationnel, ou ORM (Object Relational Mapping). Ces dernières proposent une couche d'abstraction supérieure à JDBC, sur laquelle elles reposent, et se proposent de combler le fossé entre bases de données relationnelles et objets Java.

Nous montrons ensuite de quelle manière l'intégration de Spring avec ces technologies d'ORM répond au besoin de créer une couche de persistance à la fois simple à développer, performante et facile à faire évoluer.

En fin de chapitre, nous illustrons à l'aide de notre étude de cas la façon d'obtenir la meilleure productivité possible en utilisant des outils et des frameworks qui ôtent au développeur le poids de la majorité des tâches de bas niveau qu'il a traditionnellement à effectuer.

Stratégies et design patterns classiques

La persistance des données est un sujet aujourd'hui relativement bien théorisé. La plupart des entreprises utilisent des pratiques et un vocabulaire communs, rassemblés sous forme de design patterns, ou modèles de conception.

Nous allons commencer par présenter cette base commune, qui est un prérequis à la compréhension d'API de plus haut niveau, telles que la fameuse intégration Spring/Hibernate, que nous étudions en fin de chapitre.

Le design pattern script de transaction

Le design pattern script de transaction *(transaction script)* correspond au développement d'un code d'accès à la base de données par cas d'utilisation. Dans le domaine de la persistance, il s'agit du modèle de conception le plus simple. On le retrouve souvent dans les applications non découpées en couches, à l'intérieur des servlets ou des actions Struts.

Si l'on prend le cas de Struts, chaque action correspond à un cas d'utilisation. Il y a donc une relation 1-1 entre l'action Struts et le script de transaction. C'est pour cette raison que les applications qui utilisent ce modèle de conception ne sont généralement pas conçues en couches. Le code d'accès à la base de données est directement inclus dans l'action Struts, comme dans l'exemple suivant :

```
public class ManageTodosAction extends DispatchAction {

    public final ActionForward render(
        ActionMapping mapping, ActionForm form,
        HttpServletRequest req, HttpServletResponse res) {

        DynaActionForm manageTodosForm = (DynaActionForm) form;
        String listId = (String) manageTodosForm.get("listId");

        Connection conn = null;
        Statement stmt = null;
        Collection todos = new ArrayList();
        try {
            conn = ConnectionHelper.getConnection();
            stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery("SELECT * " +
                "FROM todo WHERE todo.todo_list_id='" +
                listId + "'";
```

```
            while (rs.hasNext()) {
                Todo todo = new Todo();
                todo.setTodoId(rs.getString("id"));
                todo.setDescription(rs.getString("description"));
                todos.add(todo);
            }
            req.setAttribute("todos", todos);
        } catch (Exception e) {
            return mapping.findForward("error");
        } finally {
            ConnectionHelper.closeConnection(stmt, conn);
        }
        return mapping.findForward("render");
    }
```

Les points forts du pattern script de transaction sont les suivants :

- Grande simplicité d'utilisation.
- Les accès à la base de données peuvent être optimisés au maximum.
- Plusieurs développeurs peuvent travailler en parallèle sur des cas d'utilisation différents, avec une gestion de projet très simplifiée.

Ses points faibles sont les suivants :

- Aucune réutilisation du code : ce motif de conception s'essouffle rapidement dès lors que l'application grandit un peu (plus de 10-15 tables en base de données).
- La maintenance devient vite un cauchemar, chaque cas d'utilisation ayant été développé de manière procédurale. Changer une table de la base de données revient cher.

En conclusion, ce motif de conception est utile dans certains cas spécifiques, où les accès en base de données doivent être optimisés au maximum. Cependant, fonder une application complète sur cette conception serait une erreur, car les coûts d'évolution et de maintenance de l'application deviendraient rapidement élevés.

Le design pattern DAO

Les DAO (Data Access Object), ou objets d'accès aux données, ont la tâche de créer (Create), récupérer (Retrieve), mettre à jour (Update) et effacer (Delete) des objets Java, d'où l'expression associée pattern CRUD (Create, Retrieve, Update, Delete).

Ce sont des classes utilitaires qui gèrent la persistance des objets métier. Nous séparons ainsi les données, stockées dans des JavaBeans, et le traitement de ces données. Les DAO ont généralement le squelette suivant :

```
package tudu.domain.dao;

import java.util.Collection;

import tudu.domain.model.Todo;
```

```
/** DAO pour le JavaBean Todo */
public interface TodoDAO {

    /** Trouve tous les Todos appartenant à une liste. */
    public Collection<Todo> getTodosByListId(String listId);

    /** trouve un Todo par son identifiant */
    public Todo getTodo(String todoId);

    /** Crée un Todo. */
    public void createTodo(Todo todo);

    /** Met à jour un Todo. */
    public void updateTodo(Todo todo);

    /** Efface un Todo */
    public void removeTodo(String todoId);
}
```

Un DAO est un objet *threadsafe*, c'est-à-dire qu'il est accessible en simultané par plusieurs threads. À l'inverse d'un JavaBean, qui contient des données spécifiques de l'action en cours, un DAO ne fait que du traitement. Il peut, par conséquent, être commun à plusieurs actions parallèles. Pour cette raison, les DAO peuvent être implémentés comme des singletons, avec une seule instance partagée par l'ensemble des objets de l'application.

Les points forts des DAO sont les suivants :

- Réutilisation du code : un DAO peut être utilisé par plusieurs objets métier différents.
- Simplicité : leur compréhension et leur bonne utilisation ne nécessitent qu'un investissement léger.
- Bonne structuration du code : séparation de l'accès aux données du reste du code.

Leurs points faibles sont les suivants :

- Ils sont potentiellement moins performants que du code SQL réalisé sur mesure pour chaque besoin métier.
- Ils sont longs et fastidieux à coder en JDBC.

Le design pattern couche de domaine et le mappage objet/relationnel

Le design pattern couche de domaine *(domain layer)* propose de modéliser le domaine fonctionnel avec des objets Java, d'où son nom. Il va donc plus loin que le simple pattern DAO, car il apporte la volonté de modéliser un métier en langage informatique. Il représente une approche objet, que nous recommandons vivement pour les applications Web de moyenne ou grande taille.

L'étude de cas Tudu Lists est conçue suivant ce principe. Le fonctionnel dicte qu'un utilisateur possède plusieurs listes de todos et que chacune de ces listes possède à son tour plusieurs todos. Les JavaBeans `User`, `TodoList` et `Todo` (dans le package `tudu.domain.model`) ont été conçus en conséquence. Ainsi, le JavaBean `User` possède en attribut une collection de `TodoLists`. Il s'agit donc bien d'une modélisation du domaine fonctionnel de l'application en Java.

Bien entendu, il est primordial de pouvoir enregistrer ces objets en base de données. Ils sont donc sauvegardés dans des tables les représentant. Le modèle de données de Tudu Lists est explicite pour cela. L'objet `User` est sauvegardé dans une table `User`, possédant en champs les attributs du JavaBean. De la même manière, les relations entre les JavaBeans, qui sont des collections en Java, deviennent des clés étrangères en base de données.

La persistance de cette couche de domaine peut certes être codée en JDBC pur, mais nous nous rendons vite compte des limitations de ce code artisanal dès que nous voulons utiliser des jointures complexes entre objets. C'est le cas, par exemple, des relations many-to-many, qui nécessitent l'utilisation d'une table de jointure intermédiaire. Dans Tudu Lists, nous utilisons la table `USER_TODO_LIST` pour effectuer la liaison entre les tables `USER` et `TODO_LIST`. En effet, un utilisateur possède plusieurs listes, et une liste est possédée par plusieurs utilisateurs. Coder manuellement ce type de relation entre tables est fastidieux et peu efficace.

Le mapping objet/relationnel, ou ORM (Object Relational Mapping), désigne l'ensemble des technologies permettant de faire correspondre un modèle objet et une base de données relationnelle. Comme nous l'avons vu précédemment, cette correspondance est loin d'être évidente. Par exemple, comment modéliser l'héritage ou le polymorphisme, deux notions objet qui n'existent pas du tout en base de données ?

Ce concept de mapping objet/relationnel autorise donc la réalisation aisée d'une couche de domaine grâce à l'utilisation d'outils spécialisés. C'est pour cette raison qu'il connaît un succès grandissant depuis plusieurs années. JDO, Hibernate ou les EJB Entité sont des exemples de technologies de mapping objet/relationnel populaires, que nous étudions ultérieurement dans ce chapitre.

Les points forts du pattern couche de domaine sont les suivants :

- Très bonne évolutivité du code et coûts de maintenance réduits.
- La réutilisation du code est possible en de nombreux points de l'application.

Ses points faibles sont les suivants :

- Nécessite de solides compétences en programmation orientée objet et en gestion des transactions.
- Pousse à l'utilisation d'un outillage évolué.

- Nécessite une bonne compréhension du fonctionnement du moteur de mapping sélectionné.
- Fonctionne difficilement sur un modèle relationnel existant trop éloigné des concepts objets, par exemple, avec un schéma de base de données dénormalisé.

En conclusion, l'utilisation d'une couche de domaine devient nécessaire dès lors qu'une application dépasse la taille critique de 10 à 15 tables en base de données. Le passage à cette étape nécessite généralement de bonnes connaissances en technologie objet et un outillage adéquat. Ces investissements nécessaires sont rapidement rentabilisés.

En résumé

Les design patterns simples tels que le script de transaction et les DAO sont aujourd'hui largement répandus au sein des applications d'entreprise. Il s'agit de concepts simples à mettre en œuvre, mais qui doivent être impérativement maîtrisés avant de se lancer dans la conception d'une application.

L'apparition au cours des cinq dernières années de technologies de mapping objet/relationnel efficaces et bon marché change la donne, puisqu'elles permettent de concevoir aisément des couches de domaine complètes.

Nous verrons dans la suite de ce chapitre comment ces technologies, couplées à Spring, réduisent considérablement le code d'une application, tout en améliorant ses performances, sa maintenabilité et sa montée en charge.

Les solutions d'ORM

Historiquement, le marché de l'ORM a vu le jour en 1994 avec l'apparition de TopLink, un framework alors révolutionnaire développé par la société The Object People (le « TOP » de TopLink). La plupart des solutions actuelles d'ORM sont directement issues de TopLink. À titre d'exemple, les fondateurs de Kodo JDO travaillaient avec TopLink il y a quelques années.

Au fil du temps, un grand nombre de solutions d'ORM sont apparues sur le marché, certaines standardisées (EJB, JDO), d'autres Open Source (Hibernate, OJB), dans un foisonnement créatif qui, au final, a plutôt nui à leur adoption.

Les sections qui suivent présentent les principales de ces solutions d'ORM et donnent les clés pour choisir la plus adaptée en fonction des besoins.

Les EJB Entité 2.x

Les EJB Entité sont, avec JDO, l'une des deux solutions d'ORM standardisées au sein de J2EE. Complexes, longs à développer, avec finalement peu de fonctionnalités, ils ont engendré beaucoup de déception au début de leur utilisation. Le marketing originel a laissé un goût amer aux développeurs et architectes qui s'étaient lancés dans cette aventure au début des années 2000.

La version 2.*x*, aujourd'hui stable et performante, ainsi que l'outillage disponible, par exemple XDoclet, ont considérablement réduit ces problèmes, au point d'en faire une excellente solution de persistance. Cette technologie continue cependant de souffrir d'une mauvaise réputation.

Concepts et architecture

Un EJB Entité est une forme de JavaBean qui est mappé sur une table. Chaque accès à cet objet, par le biais des méthodes `get()` et `set()`, par exemple, entraîne une requête en base de données.

Les EJB 2.*x* ne proposent qu'une solution de mapping rudimentaire. Ils gèrent la persistance des objets dans des tables, ainsi que les jointures one-to-many et many-to-many, mais ne vont pas beaucoup plus loin. Ils sont suffisants dans la majorité des cas, leur principal atout étant leur simplicité d'utilisation. Pour les cas plus complexes, il est préférable de passer par JDBC.

Le fait, par exemple, qu'ils ne soient pas dotés des concepts d'agrégation, de projection, de requête imbriquée et de requête dynamique les rend inutilisables pour des applications de reporting.

Au final, les EJB Entité 2.*x* sont des objets assez lourds à coder, à moins de disposer du bon outillage, qui ne résolvent que 80 % du problème de la persistance. Ils sont cependant plus rapides à développer qu'une solution à base de JDBC classique et plus performants en production, du fait de l'utilisation fréquente d'un cache en supplément.

Le fait que les EJB Entité 2.*x* sont standardisés leur offre un avantage évident. Dans la pratique, il faut cependant un travail assez conséquent pour migrer une application d'un serveur d'applications à un autre, d'IBM WebSphere à BEA WebLogic, par exemple.

Exemple d'utilisation simple

Voici un exemple d'implémentation simple de `TodoDAO` utilisant des EJB Entité 2.1 :

```
package tudu.domain.dao.ejb;

(...)

public class TodoDaoJBoss implements TodoDAO {

    public Collection<Todo> getTodosByListId(String listId) {
        try {
            Collection<TodoEjb> todoEjbs = TodoUtil
                .getLocalHome().findByListId(listId);

            Collection<Todo> todos = new ArrayList();
            for (TodoEjb todoEjb : todoEjbs) {
                todos.add(todoEjb.getTodoValue());
            }
            return todos;
```

```
        } catch (Exception e) {
            throw new DataRetrievalFailureException(e
                .getMessage(), e);
        }
    }

    public void createTodo(Todo todo) {
        try {
            TodoEjb todoEjb = TodoUtil
                .getLocalHome().create(todo);
        } catch (Exception e) {
            throw new DataRetrievalFailureException(e
                .getMessage(), e);
        }
    }

(...)
}
```

Ce code soulève un certain nombre de problèmes, notamment les suivants :

- Il n'est pas facile de retrouver l'EJB Entité `TodoEjb` à l'aide du contexte JNDI. c'est pourquoi ce code a été externalisé dans une classe d'utilitaire.
- Les exceptions posent un problème similaire à celles traitées en JDBC. N'étant pas des exceptions dérivant de `RuntimeException`, elles forcent le développeur à coder un grand nombre de `try/catch` au sein desquels il ne peut pas faire grand-chose d'autre que logger l'erreur et la renvoyer.
- L'EJB `TodoEjb` recopie l'ensemble de ses champs dans le JavaBean `Todo`. En effet, utiliser un EJB directement serait dommageable en terme de performance puisque, à chaque appel à une méthode `get()` ou `set()` de cet EJB, une requête équivalente en base de données est effectuée. Il est préférable de recopier l'EJB dans un JavaBean dit *value object* (objet de valeur), même si cette bonne pratique a pour inconvénient de faire écrire un objet supplémentaire et le code de recopiage.
- La majeure partie du code métier, en particulier la recherche de `Todos`, n'est pas dans le DAO mais se trouve dans le fichier de configuration **ejb-jar** de l'EJB. Ce fichier est particulièrement complexe et peut difficilement être écrit manuellement dès que nous dépassons la dizaine d'EJB Entité. Cette complexité ne peut être résolue qu'imparfaitement *via* l'une ou l'autre des solutions suivantes :
  - Utilisation d'un IDE tel que WSAD d'IBM ou JBuilder Enterprise de Borland. Ces environnements sont toutefois coûteux et lient fortement le projet à un éditeur.
  - Utilisation de XDoclet *(http://xdoclet.sf.net)*, un générateur de code permettant de créer le fichier **ejb-jar.xml.** Le problème de cette solution est que le développement de XDoclet est quasiment stoppé, le focus des développeurs étant aujourd'hui davantage sur Java 5 et les annotations. C'est donc une solution que nous déconseillons pour cause de pérennité non garantie.

EJB QL (Query Language)

EJB Query Language est le langage de requête des EJB Entité. C'est grâce à lui que le DAO précédent a pu exécuter la méthode `getTodosByListId()`.

Mélange de Java et de SQL, son apprentissage est très rapide pour un développeur ayant déjà des compétences dans ces deux langages.

À titre d'illustration, voici le code EJB QL représentant la méthode `getTodosByListId()` :

```
SELECT DISTINCT OBJECT(todo)
FROM TodoEjb AS todo
WHERE todo.list.id=?1
```

Ce code EJB QL est inclus dans le fichier de configuration **ejb-jar.xml** et est validé lors du déploiement de l'application, offrant ainsi une garantie de bon fonctionnement bien supérieure à JDBC.

Un grand nombre de fonctions lui font cependant défaut, notamment les suivantes :

- Il ne supporte pas les requêtes dynamiques.
- Il ne propose que des fonctionnalités de requêtage extrêmement limitées (pas de support des agrégations, gestion trop simple des types `Date`, etc.).

Les EJB 2.*x* sont-ils trop complexes ?

Après cette introduction, nous pouvons aisément conclure que l'utilisation des EJB Entité 2.*x* est lourde, sans pour autant apporter une solution de persistance complète. Si ce n'est donc pas une solution recommandée, il convient de garder à l'esprit qu'elle reste la principale option si nous recherchons une solution de persistance standardisée. En effet, JDO n'a pas vraiment décollé, et les EJB 3.0 sont encore très jeunes.

Il faut donc faire un choix entre la standardisation, le temps de développement, la stabilité et la performance, et ce en fonction des priorités de chaque projet.

État du marché

Il existe une multitude d'offres sur le marché des EJB 2.*x*. Aux côtés des grands acteurs commerciaux, tels que BEA, avec WebLogic, et IBM, avec WebSphere, quelques outsiders, comme IronFlare ou Caucho, et un petit nombre de serveurs Open Source (JBoss, Geronimo, Jonas), gagnent de plus en plus en popularité.

Les EJB 2.*x* étant des composants complexes et longs à développer, l'avantage a longtemps été dans le camp des grands acteurs commerciaux, qui étaient les seuls à proposer des environnements de développement adaptés. Aujourd'hui, la donne a changé, en particulier grâce au générateur de code Java XDoclet et de XML.

Les serveurs d'applications Open Source sont pour leur part arrivés à un niveau de maturité suffisant pour pouvoir être utilisés pour les applications les plus critiques. Il ne faut donc pas craindre d'utiliser un serveur tel que JBoss en cluster pour des applications majeures.

Ajoutons que si de nombreuses entreprises choisissent d'utiliser en production un serveur d'applications commercial, elles laissent souvent leurs équipes développer avec des produits Open Source, qui sont moins coûteux et plus agréables à utiliser en terme de rapidité, de débogage et de configuration.

JDO (Java Data Object)

JDO, le mécanisme standard de persistance transparente des objets Java, entre en partie en concurrence avec les EJB Entité mais s'en distingue par quatre points clés :

- JDO n'est pas lié à un serveur d'applications J2EE. Les applications JDO sont par conséquent plus facilement portables d'un serveur à un autre et sont même utilisables dans des applications de type client lourd, sans aucun serveur d'applications.
- JDO gère la persistance de JavaBeans simples, ou POJO (Plain Old Java Objects). Cela permet de créer des applications moins complexes et plus facilement testables.
- JDO propose des fonctionnalités évoluées en terme de mapping objet/relationnel et de gestion des requêtes.
- JDO n'est pas limité aux bases de données relationnelles. Il peut aussi bien enregistrer des objets sur d'autres types de supports, tels que bases de données objet, fichiers XML, etc.

Malgré ses qualités indéniables, JDO n'a pas connu le succès à grande échelle. Les grands éditeurs, comme IBM et BEA, ont principalement axé leur stratégie sur les EJB. Quant au monde Open Source, il s'est focalisé sur des solutions concurrentes, telles qu'Hibernate. Au final, JDO est resté confiné à quelques petits éditeurs commerciaux qui n'ont pas eu la puissance marketing nécessaire pour faire décoller le marché.

Concepts et architecture

JDO propose une persistance transparente pour les JavaBeans. Il suffit de coder des JavaBeans simples et d'utiliser un outil de mapping pour les faire correspondre aux tables d'une base de données.

Par la suite, ces JavaBeans sont modifiés (on parle *d'instrumentation*) par l'implémentation de JDO afin d'être rendus persistants. Cette phase d'instrumentation a généralement lieu à la compilation, soit par génération de code Java à la volée, soit par modification des classes Java. Elle est entièrement automatisée, si bien qu'en théorie le développeur n'a pas à s'en préoccuper.

L'utilisation du pattern DAO est toujours recommandée, en JDO comme en JDBC. JDO propose un objet unique, `javax.jdo.PersistenceManager`, pour rechercher et lire les JavaBeans sauvegardés. Il s'utilise de la manière suivante :

```
PersistenceManager pm = pmFactory.getPersistenceManager ();
Todo todo = (Todo) pm.getObjectById (todoId, false);
TodoList todoList = todo.getTodoList();

pm.currentTransaction().begin();
```

```
todoList.setName("Liste terminée");
todo.setCompletionDate(Calendar.getInstance().getTime());
todo.setCompleted(true);
pm.currentTransaction().commit();

pm.close();
```

Dans cet exemple, l'objet `pmFactory` est de type `javax.jdo.PersistenceManagerFactory`. Il s'obtient *via* une classe spécifique de l'implémentation JDO utilisée.

Nous retrouvons ici un fonctionnement proche de celui des EJB. Nous pouvons suivre les liens entre les objets, par exemple, `todo.getTodoList()`, et les objets modifiés sont automatiquement sauvegardés lors du commit de la transaction.

JDOQL (JDO Query Language)

À l'instar des EJB, JDO dispose de son propre langage de requête, le JDOQL (JDO Query Language), qui reste proche de Java et du SQL :

```
SELECT FROM tudu.domain.model.Todo
WHERE list.id == "1234"
ORDER BY priority ASCENDING
```

À la différence des EJB 2.*x*, ce code est utilisé directement à l'intérieur du code Java et n'est validé qu'à l'exécution.

JDO étant indépendant de la méthode de persistance, ce langage reste plus proche de Java que du SQL, ce qui, dans la pratique, le rend légèrement plus difficile à appréhender que ses concurrents.

Utilisation dans un serveur d'applications J2EE

Les solutions JDO sont normalement totalement indépendantes des serveurs d'applications. Kodo JDO, par exemple, fonctionne de manière identique avec WebSphere, WebLogic, JBoss, JRun, SunONE, etc. Cela ne veut pas dire que Kodo JDO soit totalement séparé du serveur d'applications. Il peut utiliser les pools de connexions JDBC du serveur, s'enrôler dans des transactions JTA et, bien entendu, être utilisé par des servlets ou des EJB Session.

Cette indépendance facilite la portabilité des applications d'un serveur à un autre. C'est là un avantage important si une application est destinée à être vendue à de nombreux clients.

État du marché

Le marché de JDO est dominé par quelques solutions commerciales :

- Kodo JDO, acheté fin 2005 par la société BEA, qui est la solution la plus répandue et la plus robuste. C'est cette solution qu'utilise TheServerSide *(http://www.theserverside.com),* l'une des principales sources d'information dans le domaine J2EE.
- LiDO, un produit plus confidentiel, mais qui a l'avantage d'être développé par des Français.

Les solutions Open Source, telles que JPOX, Triactive JDO ou Speedo, n'offrent clairement pas le même niveau de service.

JDO se rapproche des EJB dans leur version 3.0, puisque la spécification de persistance des EJB 3.0 a été faite conjointement. Il est donc possible d'utiliser les dernières versions des outils JDO pour persister des EJB 3.0.

Les solutions non standardisées

Outre les solutions standardisées que sont les EJB et JDO, il existe une offre pléthorique de solutions d'ORM non standardisées.

Généralement assez proches d'utilisation les unes des autres, les concepts à l'œuvre étant toujours similaires, elles proposent des solutions de rechange souvent séduisantes aux standards présentés précédemment.

TopLink

TopLink est historiquement le premier produit d'ORM à être apparu sur le marché. D'abord développée en SmallTalk, cette solution a été recodée en Java et appartient aujourd'hui à la société Oracle. La plupart des autres frameworks s'inspirent des concepts de TopLink, qui reste une des références majeures du monde de la persistance.

Réputé pour sa robustesse, TopLink jouit de surcroît du soutien d'Oracle, qui lui garantit un support à long terme. Il s'agit cependant d'une solution propriétaire, avec une API spécifique. La choisir revient à se lier à long terme avec Oracle.

Courant 2005, Oracle a fait prendre un tournant intéressant à cette solution, en en faisant la base de son implémentation des EJB 3.0. Il s'agissait alors de la deuxième implémentation présentée au grand public, juste après celle de JBoss. Il y a donc chez Oracle une fusion en cours entre les technologies EJB, JDO et TopLink, ce qui devrait rendre cette solution très attrayante à l'avenir.

OJB (Object Relational Bridge)

Projet de la fondation Apache, OJB (Object Relational Bridge) est un logiciel libre, qui possède sa propre API. À moyen terme, il devrait offrir une couche JDO complète.

Pour l'heure, il pâtit de son manque de standardisation et de son positionnement trop proche d'Hibernate sur le marché des solutions Open Source d'ORM.

iBATIS

Encore un projet de la fondation Apache, iBATIS n'est pas une solution d'ORM à proprement parler. Cette technologie reste très proche de la base de données et permet

d'utiliser des requêtes ou des procédures stockées en Java à l'aide d'un fichier de mapping simple. Il ne s'agit donc pas de mapper des tables sur des objets.

iBATIS possède des implémentations en .NET et en Ruby, ce qui permet de réutiliser des mappings iBATIS dans des applications codées dans ces différents langages.

Castor JDO

Contrairement à ce que son nom laisse supposer, Castor JDO n'est pas une implémentation du standard JDO que nous avons vu précédemment.

C'est une solution de mapping Open Source relativement ancienne et populaire, qui se rencontre assez fréquemment dans les applications d'entreprise françaises. Cela vient du fait que Castor a proposé très tôt une solution Open Source de mapping Java/XML, ce qui a largement contribué à son succès. C'était l'avantage du premier entrant, que Castor JDO n'a cependant pas réussi à maintenir aujourd'hui.

État du marché

Le marché de la persistance a été florissant au cours des dix dernières années, qui ont vu naître un très grand nombre de solutions propriétaires et Open Source. Ces solutions se répandant en entreprise, nous arrivons à une période de maturité, qui engendre des besoins de standardisation et de professionnalisation.

Les solutions possédant des API propriétaires, comme TopLink ou OJB, sont amenées à se standardiser ou à disparaître.

Hibernate

Hibernate est aujourd'hui une des solutions d'ORM les plus utilisées. Sa simplicité d'utilisation, sa gratuité et son excellente documentation, traduite qui plus est en plusieurs langues, dont le français, l'ont fait retenir par de nombreuses équipes de développement.

En particulier, Hibernate est la solution recommandée en interne dans plusieurs grandes SSII françaises, dans une optique de baisse des coûts et d'amélioration de la qualité des développements au forfait.

Ce succès se retrouve dans le marché de l'emploi. À l'heure où nous écrivons ces lignes, le site *monster.fr* publie 85 offres d'emploi utilisant le mot-clé Hibernate, pour 4 offres JDO. À titre de comparaison, 93 offres d'emploi concernent le mot-clé Struts.

Aujourd'hui, Hibernate est au cœur du nouveau standard EJB 3.0, dont Gavin King, le fondateur d'Hibernate, est l'un des créateurs.

Choisir Hibernate revient donc à adopter une solution éprouvée, reconnue, peu onéreuse et pérenne. Qui plus est, l'engouement actuel pour cette technologie est un facteur favorable à la création d'équipes dynamiques et motivées.

Concepts et architecture

Hibernate est une solution classique, mappant des POJO, autrement dit des JavaBeans simples, sur les tables d'une base de données. Comparé à d'autres solutions, ce mapping est particulièrement avancé. Il permet, par exemple, d'utiliser aisément des fonctionnalités objet complexes, telles que l'héritage ou le polymorphisme, et d'utiliser des bases de données préexistantes avec des modèles de données complexes.

Ce mapping entre les objets Java et la base de données est traditionnellement effectué à l'aide de fichiers XML de mapping, qui ont la forme suivante :

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

<!DOCTYPE hibernate-mapping PUBLIC
    "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping>
    <class name="tudu.domain.model.Todo"
           table="todo">

        <cache usage="read-write" />

        <id name="todoId"
            column="id"
            type="java.lang.String">

            <generator class="uuid.hex">
            </generator>
        </id>

        <property
            name="description"
            type="java.lang.String"
            update="true"
            insert="true"
            column="description"/>

(...)

        <many-to-one
            name="todoList"
            class="tudu.domain.model.TodoList"
            cascade="none"
            outer-join="auto"
            update="true"
            insert="true"
            column="todo_list_id"/>

        <property
            name="completed"
```

```
            type="boolean"
            update="true"
            insert="true"
            column="Completed"/>

    </class>
</hibernate-mapping>
```

Ces documents XML de mapping sont relativement simples à écrire à l'aide d'un éditeur XML. Des outils plus évolués accroissent encore la productivité, notamment des outils graphiques, tels les Hibernate Tools disponibles sur le site d'Hibernate, des générateurs de code, comme XDoclet, ou plus récemment les annotations.

Nous recommandons les deux bonnes pratiques suivantes :

- Utiliser un éditeur XML lorsque le nombre de tables est réduit (moins de 50) et le turn-over de l'équipe important. Il est en effet coûteux de réinstaller les outils et de former les développeurs.
- Investir dans une solution plus productive, Hibernate Tools en particulier, en cas de mapping long et complexe à réaliser par un ou deux développeurs expérimentés.

Une fois les JavaBeans mappés sur les tables de la base de données, Hibernate propose le mécanisme classique suivant pour utiliser le motif CRUD (Create, Retrieve, Update , Delete), tel que présenté précédemment dans ce chapitre :

```
public void saveTodo(String description, int priority) {
    Session session = HibernateUtil.currentSession();
    Transaction tx = session.beginTransaction();

    Todo todo = new Todo();
    todo.setDescription(description);
    todo.setPriority(priority);
    session.save(todo);

    tx.commit();
    HibernateUtil.closeSession();
}
```

HQL (Hibernate Query Language)

Pour des requêtes plus complexes, Hibernate dispose de son propre langage de requête, le HQL (Hibernate Query Language), mélange de Java et de SQL, dont voici un exemple :

```
select user.todoLists
from User user
where user.userId=?
```

Cette requête trouve toutes les listes de todos partagées pour un utilisateur donné. Remarquons que, bien que nous travaillions ici directement avec les noms des objets Java (`userId` et non `user_id`), la syntaxe reste très proche du SQL.

De même que le JDOQL, le HQL s'utilise directement dans le code Java.

État du marché

Depuis 2004, Hibernate est engagé dans la voie de la standardisation et de la professionnalisation. L'embauche d'une partie de l'équipe d'Hibernate par le groupe JBoss, dont Gavin King, le fondateur du projet, a fortement influencé cette professionnalisation, même si des critiques se font jour sur le risque de mainmise du groupe JBoss sur le projet.

Dans la pratique, nous constatons qu'Hibernate est utilisé chez plusieurs opérateurs de télécoms et grandes banques françaises et que cette solution semble largement employée au sein des applications d'entreprise.

D'ici à cinq ans, il est possible qu'Hibernate soit complètement remplaçable par les EJB Entité version 3.0. Étant donné que la communauté Hibernate vient d'aider à redéfinir en profondeur ce standard, une migration d'Hibernate vers des EJB Entité ne devrait pas constituer un problème.

Nous étudierons plus en détail l'utilisation d'Hibernate, en particulier avec Spring, dans l'étude de cas de la fin du chapitre.

Les EJB 3.0

La version 3.0 des EJB incarne le triomphe de deux conceptions que nous défendons tout au long de cet ouvrage :

- Les solutions de persistance fondées sur des POJO, telles qu'Hibernate, plutôt que des solutions impliquant des objets complexes, à l'image des EJB 2.*x*.
- L'injection de dépendances, telle que pratiquée dans Spring, afin d'améliorer la testabilité du code et de simplifier les appels entre EJB, sans passer par JNDI.

Grâce à l'utilisation intensive des annotations, une nouveauté du JDK 5.0, les EJB 3.0 ne sont plus que des JavaBeans simples. Ces annotations, qui viennent enrichir les classes et les méthodes, permettent de définir, d'une part, la persistance et, d'autre part, les liens des JavaBeans entre eux (injection de dépendances).

Un certain nombre de nouvelles fonctionnalités de requêtage viennent en outre combler le fossé existant entre les EJB 2.*x* et les solutions d'ORM concurrentes, notamment les requêtes dynamiques et imbriquées.

Comme il sera possible de déployer des applications EJB 2.*x* sur des serveurs d'applications à la norme EJB 3.0, il n'y a pas lieu de redouter des coûts de migration importants.

Concepts et architecture

Les EJB 3.0 permettent de coder des POJO persistants simplement et rapidement, réduisant d'autant les coûts de développement. La configuration de ces POJO s'effectue par le biais d'annotations, et non plus d'un fichier XML monolithique, estimé trop complexe.

Les concepts à l'œuvre dans les EJB 3.0 sont finalement assez proches de ceux de TopLink, Hibernate ou JDO.

Les EJB Entité 3.0 sont rendus persistants grâce à l'objet `javax.persistence.EntityManager`, qui permet l'utilisation des méthodes CRUD classiques de la même manière que la session Hibernate. Voici de quelle manière il peut être utilisé pour persister un objet :

```
@Stateless
public class TodoManagerSessionEJB implements TodoManagerSession {

    @Inject protected EntityManager entityManager;

    (...)
    public void createTodo(String description, int priority) {
        Todo todo = new Todo();
        todo.setDescription(description);
        todo.setPriority(priority);
        entityManager.persist(todo);
    }
    (...)
```

Nous constatons par cet exemple que les EJB 3.0 s'inscrivent dans la lignée des technologies de persistance étudiées précédemment et qu'ils seront facilement utilisables par les développeurs ayant déjà une expérience d'Hibernate ou de JDO. L'utilisation des annotations (lignes commençant par `@`) et de l'inversion de contrôle, que nous observons ici pour injecter l'objet `entityManager`, est aussi simple qu'intuitive, et elle devrait rapidement séduire.

À titre indicatif, l'ancienne méthode de recherche des EJB (JNDI) est toujours disponible pour trouver l'objet `entityManager` et nécessite autant de code qu'auparavant :

```
if (entityManager == null} (
    try{
        entityManager = (EntityManager)(new InitialContext())
            .lookup("java:comp/ejb/EntityManager");
    } catch (Exception e){};
}
```

Voici un exemple d'EJB 3.0 simple, inspiré de Tudu Lists :

```
package tudu.domain.model.ejb3;

import javax.persistence.*;

@Entity
@Table(name = "TODO")
public class Todo implements java.io.Serializable {

    private String todoId;
    private String description;
    private TodoList todoList;
    private int priority;
```

```
    private boolean completed;
    private Date completionDate;

    public Todo() {
    }

    @Id(generate = GeneratorType.AUTO)
    public String getTodoId() {
        return todoId;
    }

    public void setTodoId(string todoId) {
       this.todoIdd = todoId;
    }

    @Column(name = "DESCRIPTION")
    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }

    @ManyToOne
    @JoinColumn(name = "todo_list_id")
    public TodoList getTodoList() {
        return todoList;
    }

    public void setTodoList(TodoList todoList) {
        this.todoList = todoList;
    }

(...)
}
```

Cet EJB inclut sa configuration sous la forme d'annotations et est dès lors beaucoup plus simple à développer que son équivalent en EJB 2.*x*. L'utilisation du fichier de configuration **ejb-jar.xml** rendait en effet très difficile l'écriture d'un tel EJB en version 2.*x* si nous ne disposions pas d'un outillage approprié. En EJB 3.0, ce fichier a totalement disparu.

De plus, remarquons que nous n'utilisons les annotations que lorsque la configuration par défaut n'est pas satisfaisante. Cette notion de configuration par exception permet de réduire davantage encore le nombre d'instructions nécessaires pour développer un EJB complet.

Coûts et risques

Contrairement aux autres technologies présentées dans ce chapitre, les EJB 3.0 ne sont pas encore une solution reconnue et éprouvée à l'heure où nous écrivons ces lignes. Il faudra sans doute quelque temps avant qu'elle soit utilisée massivement en production.

Le passage aux EJB 3.0 induit des coûts de formation peu importants pour les équipes maîtrisant déjà les EJB 2.*x* ou Hibernate. Il s'agit en fait d'une évolution ou d'une amélioration, qui est attendue depuis longtemps par les développeurs. Pour les développeurs débutants, les EJB 3.0 introduisent une simplification et une standardisation qui leur permettront d'être productifs plus rapidement qu'avec les technologies concurrentes. Le passage aux EJB 3.0 semble quasiment toujours avantageux financièrement.

Même si cette technologie est récente, le passage aux EJB 3.0 se révèle aisément maîtrisable et ne présente que peu de risque. Il s'agit du socle technique standard J2EE, pour lequel existent déjà plusieurs implémentations.

État du marché

Les EJB 3.0 s'annoncent comme une solution de persistance standardisée, supportée par les grands noms de l'industrie et prenant en compte les dernières évolutions en matière de mapping objet/relationnel, telles que la persistance transparente, l'inversion de contrôle et les annotations. Dès 2005, des implémentations exemples des EJB 3.0 ont été fournies par JBoss et Oracle. Il est clair que tous les grands acteurs du monde EJB (IBM, BEA) ou JDO vont rapidement fournir des implémentations stables de ce standard. C'est donc un marché jeune, mais extrêmement compétitif, qui est en train de se former.

L'avenir dira si les apports si prometteurs des EJB 3.0 ne seront pas gâchés par des problèmes dont les normes EJB 1.0 et JDO 1.0 avaient donné le pénible exemple. Faute de pouvoir se mettre d'accord, les entreprises partenaires avaient alors défini des normes trop complexes, peu compatibles entre elles et trop peu pertinentes.

En résumé

Nous venons d'étudier les principales solutions d'ORM du marché. Les EJB 2.*x* et 3.0, JDO et Hibernate en sont les principaux acteurs. Ces technologies restent assez proches dans leur utilisation et se sont engagées vers une standardisation commune. Dans l'avenir, nous pouvons donc espérer pouvoir utiliser un socle technique robuste et mature, fondé sur la spécification EJB 3.0.

Cependant, cette perspective va nécessiter encore quelques années avant de se concrétiser. L'architecte logiciel est donc aujourd'hui contraint de bien connaître les différentes solutions existantes afin de choisir la mieux adaptée au projet qu'il est en train de réaliser.

Apports de Spring au monde de la persistance

Spring réduit considérablement le nombre de lignes de code nécessaires au développement des applications utilisant les principales technologies d'ORM du marché, que nous venons de passer en revue.

En particulier, Spring propose une aide au développement d'applications fondées sur les technologies d'ORM suivantes :

- Hibernate 2.*x* et 3.0
- iBATIS
- JDO
- OJB
- TopLink

Spring propose en outre une aide à l'utilisation des EJB. Bien que cette aide ne soit pas centrée sur la persistance, nous pouvons considérer que cette technologie est aussi couverte par Spring.

Nous retrouvons ici les technologies étudiées précédemment, dont Spring aide l'utilisation des quatre manières différentes suivantes :

- Spring facilite la création d'une couche de DAO bien construite grâce à l'inversion de contrôle et à la séparation en couches.
- Spring simplifie et standardise l'utilisation de transactions dans la couche d'accès aux données.
- Spring propose une hiérarchie d'exceptions standardisées, de manière à avoir les mêmes exceptions dans les couches supérieures, et ce quelle que soit la technologie de persistance utilisée.
- Spring propose des classes d'utilitaires simplifiant considérablement le nombre de lignes de code à écrire pour chacune de ces technologies.

Si les deux premiers points ne portent pas spécialement à débat, les deux suivants (gestion des exceptions et classes d'utilitaires) font généralement l'objet de discussions, car elles impliquent l'utilisation de classes propres à Spring.

Un DAO héritant de la classe `org.springframework.orm.hibernate3.support.HibernateDaoSupport` se lie en effet à Spring. C'est contraire à l'esprit initial d'utilisation d'un framework d'inversion de contrôle, censé permettre de ne pas dépendre de ce framework et donc de n'importer aucune de ses classes.

Dans la pratique, nous pouvons utiliser Spring soit comme un simple moteur d'inversion de contrôle, soit comme un framework complet facilitant l'intégration de différentes technologies. La plupart des projets optent pour la deuxième option et choisissent d'utiliser Spring à son maximum. C'est ce que nous allons faire dans l'étude de cas Tudu Lists.

Tudu Lists : persistance des données

Nous avons choisi de développer Tudu Lists avec Hibernate, car ce framework bénéficie d'une très bonne intégration avec Spring.

Cependant, Spring supportant de manière relativement similaire les différentes technologies d'ORM que nous avons étudiées, ce cas pratique reste pertinent, même si vous choisissez d'utiliser une autre solution qu'Hibernate.

Nous avons fait le choix de tirer le meilleur parti des capacités de Spring en l'utilisant tout à la fois pour simplifier le code de persistance et gérer les transactions et les exceptions.

Création des fichiers de mapping XML

Dans notre chaîne de construction de code, nous créons les fichiers de mapping Hibernate manuellement, de façon à ne pas dépendre d'outils trop complexes à mettre en œuvre.

Nous recommandons simplement l'utilisation de l'éditeur XML Web Tools Platform, qui ajoute un support XML de bon niveau à Eclipse.

Les fichiers de mapping utilisés n'ont rien de particulier. Pour le lecteur qui souhaiterait les étudier, nous conseillons la lecture de la documentation en ligne d'Hibernate, récemment traduite en Français, à l'adresse *http://www.hibernate.org/hib_docs/reference/fr/html/.*

Nous pourrions aussi choisir d'intégrer XDoclet, Hibernate et Ant afin d'avoir un environnement de travail plus productif. Si ces choix permettent de générer très rapidement des fichiers de configuration et donc de démarrer un projet sans délai, ils obligent toutefois à utiliser le JDK 1.4 et Hibernate 2.*x*, car le développement de XDoclet semble avoir été stoppé depuis l'arrivée des annotations (JDK 5.0).

Création des POJO

Nous codons nos POJO en Java à l'aide d'Eclipse. Dans la mesure où ces POJO représentent le domaine fonctionnel de l'application, une analyse doit d'abord être menée afin de spécifier de quels objets nous avons besoin et selon quelles liaisons.

Le codage de ces POJO en Java est particulièrement rapide, d'autant plus qu'Eclipse permet de générer automatiquement les méthodes `get()` et `set()`. Il suffit d'écrire les champs privés correspondant aux attributs de chaque objet puis de sélectionner dans le menu Source la fonction Generate Getters and Setters.

Au final, nos POJO sont de la forme suivante :

```
package tudu.domain.model;

import java.io.Serializable;
import java.util.Date;
```

```
/**
 * Un Todo.
 */
public class Todo implements Serializable, Comparable {

    private String todoId;

    private TodoList todoList;

    private Date creationDate;

    private String description;

    private int priority;

    private boolean completed;

    private Date completionDate;

    private Date dueDate;

    public String getTodoId() {
        return todoId;
    }

    public void setTodoId(String todoId) {
        this.todoId = todoId;
    }

    public boolean isCompleted() {
        return completed;
    }

    public void setCompleted(boolean completed) {
        this.completed = completed;
    }

    public Date getCompletionDate() {
        return completionDate;
    }

    public void setCompletionDate(Date completionDate) {
        this.completionDate = completionDate;
    }

    ( … )

    public TodoList getTodoList() {
        return todoList;
    }
```

```
    public void setTodoList(TodoList todoList) {
        this.todoList = todoList;
    }

    public int compareTo(Object o) {
        Todo that = (Todo) o;
        int order = that.getPriority() - this.getPriority();
        if (this.isCompleted()) {
            order += 10000;
        }
        if (that.isCompleted()) {
            order -= 10000;
        }
        if (order == 0) {
            order = (this.getDescription() + this.getTodoId()).compareTo(that
                    .getDescription()
                    + that.getTodoId());
        }
        return order;
    }

    public boolean equals(Object o) {
        if (this == o) {
            return true;
        }
        if (!(o instanceof Todo)) {
            return false;
        }

        final Todo that = (Todo) o;

        if (todoId != null ? !todoId.equals(that.todoId) : that.todoId != null) {
            return false;
        }

        return true;
    }

    public int hashCode() {
        return (todoId != null ? todoId.hashCode() : 0);
    }

}
```

Les todos devant pouvoir être triés automatiquement par priorité et date, le JavaBean `Todo` doit implémenter l'interface `Comparable`, qui force l'écriture de la méthode `compareTo`. C'est à l'intérieur de cette méthode qu'est inscrite la logique permettant de trier les todos dans le bon ordre.

Notons également que les todos étant des objets Java standards, les méthodes `equals()` et `hashcode()` doivent être correctement implémentées. Dans ce cas précis, cela reste relativement simple, car les todos sont simplement identifiés par leur attribut `todoId`.

Implémentation des DAO

Pour une bonne séparation des couches, une pratique encouragée par l'utilisation de Spring, nous créons des interfaces pour chacun de nos DAO.

Parmi les multiples avantages de ces interfaces, citons notamment les suivants :

- Spécification, lors de la phase de conception, des appels à la base de données qui seront utiles.
- Établissement d'un contrat entre les développeurs de la couche de persistance et ceux des couches supérieures.
- Augmentation de la testabilité de l'application *(voir le chapitre 17)*.
- Meilleure migration future vers un autre système de persistance ou une version supérieure de l'outil en cours.

Il est important d'utiliser Eclipse ou un IDE comparable afin de générer automatiquement une implémentation à partir d'une interface. Cela rend presque nul le coût supplémentaire induit par le développement des interfaces.

Nous avons vu, lors de la présentation d'Hibernate, que l'écriture du DAO Hibernate nécessitait l'ouverture d'une session Hibernate et d'une transaction, puis leur fermeture.

Voici maintenant un DAO codé en utilisant Spring :

```
package tudu.domain.dao.hibernate3;

import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

(...)

public class TodoDAOHibernate extends HibernateDaoSupport implements TodoDAO {

    /**
     * Trouve un Todo par son identifiant.
     */
    public Todo getTodo(String todoId) {
        return (Todo) getHibernateTemplate()
            .get(Todo.class, todoId);
    }

    /**
     * Sauvegarde un Todo.
     */
```

```
    public void saveTodo(Todo todo) {
        getHibernateTemplate().saveOrUpdate(todo);
    }

    /**
     * Efface un Todo.
     */
    public void removeTodo(String todoId) {
        getHibernateTemplate().delete(getTodo(todoId));
    }
}
```

La première chose que nous remarquons dans ce code est son extrême concision. Il suffit pour s'en convaincre de le comparer au design pattern script de transaction, écrit en JDBC en début de chapitre. Par rapport à du code Hibernate standard, nous n'avons pas eu à coder les ouvertures et les fermetures de session ni la gestion des transactions.

Il existe cependant encore du code superflu ici, la méthode `saveTodo` n'étant pas utile. L'un des atouts d'Hibernate est de permettre de sauvegarder automatiquement un ensemble complet d'objets. Dans Tudu Lists, la création d'un objet `Todo` revient à ajouter un nouveau todo dans une liste :

```
TodoList todoList = todoListsManager.findTodoList(listId);
todoList.getTodos().add(todo);
todo.setTodoList(todoList);
```

L'ensemble de ce code fait partie d'une transaction gérée par Spring. Lors du commit de cette transaction, la liste `todoList` est sauvegardée, ce qui entraîne automatiquement la sauvegarde du nouvel objet `Todo`, qui en fait désormais partie.

*Utilisation d'*HibernateDAOSupport

Les DAO utilisés dans Tudu Lists héritent tous de `org.springframework.orm.hibernate3.support.HibernateDaoSupport`, une classe donnant accès à `HibernateTemplate`. Cette dernière permet de réduire considérablement le code à écrire pour utiliser Hibernate. Nous ne pouvons que conseiller une lecture attentive de la javadoc de cette classe, qui reprend les méthodes d'Hibernate pour sauvegarder, mettre à jour, effacer, etc.

Tudu Lists fournit un exemple d'utilisation de l'ensemble des méthodes couramment nécessaires lors de la persistance d'un objet ou d'une collection d'objets.

À nouveau, l'héritage de cette classe lie fortement l'application en cours au framework Spring, ce qui n'est pas recommandé par l'équipe développant Hibernate. Il est bien entendu toujours possible d'utiliser Hibernate de manière traditionnelle, c'est-à-dire sans hériter de cette classe, et de continuer d'utiliser Spring uniquement pour gérer l'inversion de contrôle et les transactions.

Cependant, nous considérons que les avantages de cette classe sont bien supérieurs à ses inconvénients, tant elle offre un gain de temps et une simplification du code appréciables.

De plus, si nous devions cesser de l'utiliser, un refactoring de la couche de persistance serait simple et sans risque, grâce au bon découpage de l'application.

Configuration de Spring

La configuration de Spring pour la couche de persistance est stockée dans le fichier **WEB-INF/applicationContext-hibernate.xml.**

Il s'agit d'un fichier de configuration Spring standard, qui crée les différentes implémentations des DAO sous la forme de Beans Spring, en les liant (par inversion de contrôle) avec la `SessionFactory` d'Hibernate et un gestionnaire de transactions :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
  <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">

    <property name="configLocation">
      <value>classpath:hibernate.cfg.xml</value>
    </property>
  </bean>

  <bean id="transactionManager"
class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory">
      <ref bean="sessionFactory" />
    </property>
  </bean>

  <bean id="userDAO"
        class="tudu.domain.dao.hibernate3.UserDAOHibernate">
    <property name="sessionFactory">
      <ref bean="sessionFactory" />
    </property>
  </bean>

  <bean id="roleDAO"
        class="tudu.domain.dao.hibernate3.RoleDAOHibernate">
    <property name="sessionFactory">
      <ref bean="sessionFactory" />
    </property>
  </bean>

  <bean id="todoListDAO"
        class="tudu.domain.dao.hibernate3.TodoListDAOHibernate">
    <property name="sessionFactory">
      <ref bean="sessionFactory" />
```

```
    </property>
  </bean>

    ( … )
</beans>
```

La seule configuration importante à ce stade consiste à donner une session Hibernate à chaque DAO Hibernate.

Les transactions sont quant à elles gérées dans la couche supérieure, qui est capable d'orchestrer l'utilisation conjointe de plusieurs JavaBeans. Nous avons ainsi des transactions qui peuvent porter sur plusieurs DAO.

Voici un exemple tiré du fichier **WEB-INF/applicationContext.xml** (cette configuration est détaillée au chapitre 12, dédié aux transactions) :

```
<bean id="userManagerTarget"
    class="tudu.service.impl.UserManagerImpl">
    <property name="userDAO">
        <ref bean="userDAO" />
    </property>
    <property name="roleDAO">
        <ref bean="roleDAO" />
    </property>
    <property name="todoListDAO">
        <ref bean="todoListDAO" />
    </property>
    <property name="todoDAO">
        <ref bean="todoDAO" />
    </property>
    <property name="propertyDAO">
        <ref bean="propertyDAO" />
    </property>
</bean>

<bean id="userManager"
class="org.springframework.transaction.interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="transactionManager" />
    </property>
    <property name="target">
        <ref local="userManagerTarget" />
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="create*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="delete*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

Utilisation du pattern session-in-view

Le design pattern session-in-view est particulièrement important si nous voulons utiliser Hibernate de manière simple et performante. Cependant, son apparente complexité et son utilisation particulière nécessitent une explication plus détaillée.

L'idée sur laquelle repose ce pattern est d'associer une session Hibernate au fil d'exécution, ou thread, de la requête HTTP en cours. Cela présente les deux avantages suivants :

- L'ouverture d'une seule session Hibernate apporte un léger gain de performance.
- La session Hibernate est accessible à partir de n'importe quel endroit de l'application.

Cette dernière assertion prête toutefois à confusion. Si l'application est découpée en couches avec Spring, n'est-ce pas précisément pour éviter d'avoir des accès à la couche de persistance depuis n'importe où ?

En réalité, l'accès à la session Hibernate n'est utile que pour une seule raison : l'utilisation maximale des fonctionnalités de lazy-loading, ou initialisation tardive, d'Hibernate. En effet, afin de limiter le nombre et la taille des requêtes effectuées en base de données, Hibernate ne va pas toujours chercher l'intégralité des données réclamées et ne charge les jointures que lorsqu'elles sont directement demandées par l'application. Optionnel dans Hibernate 2.*x,* ce fonctionnement est devenu le mode par défaut d'Hibernate 3.0, car il améliore nettement les performances.

Voici comment fonctionne le retrait d'une jointure classique one-to-many dans un exemple fictif emprunté à Tudu Lists :

1. Couche de persistance : recherche d'un objet `TodoList`, qui représente une collection de todos.
2. Une requête en base est exécutée sur la table `TODO_LIST`, mais pas sur la table `TODO`.
3. Cet objet `TodoList` est remonté jusqu'à la couche Struts, qui le stocke en tant qu'attribut de la requête HTTP.
4. L'action Struts renvoie vers une page JSP.
5. Dans la JSP, une bibliothèque de tags prend l'objet `TodoList` et itère sur la collection de todos qu'il possède.
6. Au moment de cette itération, Hibernate exécute une nouvelle requête SQL sur la table `TODO` afin de récupérer les informations concernant les todos.

Le lazy-loading désigne ainsi le fait de ne charger qu'au dernier moment les informations provenant de la base de données.

Ce principe permet de réduire les accès à la base de données. Nous n'avons généralement pas besoin d'un graphe d'objets complet, mais juste de quelques objets. Il est donc très fortement recommandé.

Malheureusement, pour utiliser ce procédé efficacement, il est nécessaire d'avoir accès à la session Hibernate à partir de n'importe quelle partie de l'application, y compris, comme dans notre exemple, des JSP.

C'est ce que propose de faire le pattern session-in-view en stockant la session Hibernate dans la thread en cours (nous ne détaillons pas ici son implémentation, qui nécessite une connaissance de l'objet `ThreadLocal`).

Spring propose pour cela un filtre de servlet prêt à l'emploi, qui s'insère dans le fichier **WEB-INF/web.xml** de la manière suivante :

```
<filter>
    <filter-name>Hibernate Session In View Filter</filter-name>
    <filter-class>
org.springframework.orm.hibernate3.support.OpenSessionInViewFilter
    </filter-class>
</filter>

( … )

<filter-mapping>
    <filter-name>Hibernate Session In View Filter</filter-name>
    <url-pattern>*.action</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>Hibernate Session In View Filter</filter-name>
    <url-pattern>/secure/dwr/*</url-pattern>
</filter-mapping>

<filter-mapping>
    <filter-name>Hibernate Session In View Filter</filter-name>
    <url-pattern>/j_acegi_security_check</url-pattern>
</filter-mapping>
```

Ce mapping est utilisé à trois endroits différents :

- Actions Struts.
- DWR, le framework utilisé pour les fonctionnalités AJAX de l'application : les POJO qui utilisent le lazy-loading sont en effet présentés en JavaScript. Nous étudions cette partie plus au chapitre 9, dédié à AJAX.
- Phase de login, Acegi Security utilisant bien entendu les POJO `User` et `Role` pour gérer l'authentification et les autorisations.

Afin de tester l'utilisation du pattern session-in-view, nous proposons d'enlever certains de ces mappings et d'observer les erreurs qui ne manqueront pas de se produire dans l'application.

Utilisation du cache d'Hibernate

Pour obtenir de bonnes performances, l'utilisation d'un cache conjointement avec Hibernate est essentielle. Pour ce faire, Hibernate s'intègre à un certain nombre de solutions de cache, dont nous ne retenons pour notre démonstration que deux solutions :

- **JBoss Cache.** Solution complète offrant un cache transactionnel en cluster. Il est recommandé de ne l'utiliser que si nous avons besoin d'un cache en cluster, une option généralement inutile, étant donné les performances déjà très élevées d'Hibernate.
- **Ehcache.** Ce cache très simple d'utilisation et très performant est la solution de cache la plus utilisée avec Hibernate.

C'est Ehcache que nous avons choisi d'utiliser avec Tudu Lists. Sa configuration est définie dans le fichier **JavaSource/ehcache.xml :**

```
<ehcache>

    ( ... )

     <cache name="tudu.domain.model.Todo"
        maxElementsInMemory="50000"
        eternal="false"
        overflowToDisk="true"
        timeToIdleSeconds="600"
        />

     <cache name="tudu.domain.model.TodoList"
        maxElementsInMemory="20000"
        eternal="false"
        overflowToDisk="true"
        timeToIdleSeconds="600"
        />

    ( ... )

</ehcache>
```

Chaque POJO et collection de POJO enregistrables doivent être définis dans ce fichier de configuration, qui comporte les éléments de configuration suivants :

- `maxInMemory` : nombre maximal d'objets dans le cache en mémoire.
- `eternal` : précise si les objets vivent éternellement dans le cache.
- `timeToIdleSeconds` : temps de vie des objets, lorsqu'ils ne sont pas utilisés.
- `timeToLiveSeconds` : temps de vie des objets.
- `overflowToDisk` : précise si les objets peuvent être sauvegardés sur le disque lorsqu'il n'y a plus de place en mémoire.

Dans l'exemple ci-dessus, les todos peuvent être 50 000 en mémoire avant d'être sauvegardés sur le disque dur et ne vivent pas plus de dix minutes.

Cette configuration est bien entendu entièrement dépendante des besoins fonctionnels.

Conclusion

À l'heure actuelle, l'utilisation directe de JDBC est surpassée par celles des frameworks d'ORM, que ce soit en terme de qualité, de productivité ou de performance.

Ces frameworks ne nécessitent pas pour autant des compétences moindres en matière de base de données. Bien au contraire, pour les utiliser efficacement, les développeurs doivent être parfaitement à l'aise avec les mondes objet aussi bien que relationnel. Cela induit des coûts supplémentaires de formation et de recrutement, qui ne sont rentabilisés qu'à moyen terme.

Par ailleurs, la profusion actuelle de solutions d'ORM peut prêter à confusion. Une vue d'ensemble de ces différentes solutions est nécessaire pour choisir la mieux adaptée à un projet précis.

Nous ne saurions trop insister sur l'extrême productivité du couple Spring-Hibernate. Il est intéressant de constater que cette productivité ne joue pas au détriment de la qualité, puisque l'application finale est proprement découpée en couches, ni de la performance, Hibernate générant des requêtes optimisées et disposant de caches performants.

12

Gestion des transactions

Pour stocker des données et échanger des messages, les applications d'entreprise mettent en jeu de nombreuses ressources, telles que bases de données, serveurs de messagerie applicative, gros systèmes, etc. Ces applications doivent maintenir une cohérence entre ces différentes ressources afin de garantir la consistance des données en cas d'annulation de traitement, d'erreur, de bogue ou de panne. Au cœur des technologies Java/J2EE, le concept de transaction vise à résoudre ces problématiques de cohérence.

Ce chapitre passe en revue les principales notions inhérentes aux transactions et explique comment le framework Spring permet de les mettre en œuvre de manière optimale dans une architecture, et ce, quels que soient les technologies ou frameworks sous-jacents employés.

Rappels sur les transactions

Oublier de gérer les transactions dans une application n'engendre pas nécessairement de dysfonctionnements visibles. Cependant, cet oubli ne permettant pas de garantir la consistance des données manipulées, des erreurs délicates à détecter peuvent survenir. Ces erreurs sont souvent difficilement reproductibles sans données de production et peuvent impacter tous les systèmes d'information utilisés par l'application. Par exemple, les données relatives à un client n'ont pas été correctement enregistrées dans les différentes sources de données de l'entreprise ; de ce fait, des données erronées concernant ce client se retrouvent dans certaines applications tierces.

La notion de transaction est donc primordiale. Elle doit être mise en œuvre dans toute application réalisant des mises à jour dans des sources de données.

Propriétés des transactions

Prenons un exemple concret tiré de notre étude de cas Tudu Lists. L'importation d'une liste de todos en mode modification est particulièrement appropriée, puisqu'il s'agit de fusionner les valeurs des éléments de la nouvelle liste avec les valeurs de l'ancienne.

Imaginons que ce traitement échoue à sa moitié. Les données de la liste ne sont dès lors pas homogènes. La première partie correspond à des données de la nouvelle liste, et la seconde à des données de l'ancienne. Ces dernières données n'ont pas été modifiées puisque l'exécution s'est arrêtée avant leur traitement.

Les transactions permettent de pallier ce problème, un de leurs objectifs étant d'assurer l'atomicité d'un traitement. Si une erreur se produit dans l'importation, les modifications ne sont pas répercutées en base, et les données de la liste restent dans leur état initial. Par contre, la liste est effectivement modifiée en base si le traitement se déroule correctement jusqu'à son terme.

Le rôle des transactions est donc de valider les modifications réalisées par des traitements d'une méthode si aucune erreur ne se produit et de les annuler dans le cas contraire. De cette manière, les données restent cohérentes. Ajoutons qu'une fois les modifications sur les données validées, celles-ci doivent être durables.

L'exemple précédent montre qu'une transaction doit être une unité de traitements *(unit of work)*. De la sorte, toutes les modifications réalisées par ces traitements doivent être validées ou annulées en même temps.

Une transaction doit en outre vérifier les propriétés fondamentales suivantes, communément désignées sous l'acronyme ACID (atomicité, consistance, isolation, durabilité) :

- **Atomicité.** Tous les traitements réalisés dans une transaction sont vus par l'application comme une seule unité et sont donc indivisibles.
- **Consistance.** Les données des différents systèmes ne peuvent rester dans des états incohérents. Ces états sont, par exemple, des transitions nécessaires au cours des différents traitements de la transaction, ces derniers pouvant n'être pas visibles au moment de son achèvement.
- **Isolation.** Définit la visibilité des états internes de la transaction par le reste de l'application au cours de sa réalisation. Suivant les sources de données et les technologies utilisées, l'isolation peut avoir plusieurs niveaux.
- **Durabilité.** Lorsqu'une transaction est validée, les modifications réalisées sur les différentes données sont définitives et entièrement visibles par le reste des applications qui utilisent la source de données.

Granularité des transactions

Les transactions peuvent avoir plusieurs niveaux de granularité transactionnelle, allant des transactions simples aux transactions imbriquées :

- **Transactions simples.** En vertu des propriétés ACID, une transaction simple est un bloc indivisible, qui possède donc une granularité importante. Elle correspond à une boîte noire contenant les différents traitements.

- **Transactions imbriquées.** Pour les systèmes compatibles, il est possible de réaliser des transactions imbriquées, à la granularité plus fine. Elles consistent en des sous-blocs transactionnels au sein d'une même transaction. Ces blocs peuvent être annulés indépendamment de la transaction ou validés en même temps que cette dernière.

Isolation transactionnelle

Certains systèmes d'information autorisent différents niveaux d'isolation afin de permettre aux applications de maîtriser complètement les données visibles à l'extérieur de la transaction. Les applications peuvent de la sorte jouer indirectement sur les performances d'accès aux données puisque cette propriété s'appuie généralement sur des mécanismes de gestion de la concourance d'accès et de verrous.

Les bases de données relationnelles sont un exemple classique de système d'information utilisant ce mécanisme. Ces bases de données fournissent plusieurs niveaux d'isolation des données lors de leur manipulation au cours d'une transaction, qui ne correspondent pas tous à une isolation stricte des données et peuvent donc entraîner des problèmes, comme le récapitule le tableau 12.1.

Tableau 12.1. Typologie des problèmes induits par les transactions

Type de problème	Description
Dirty Read (lecture sale)	Un fil d'exécution voit des données d'une transaction sans que celle-ci soit terminée. De ce fait, les données se trouvent dans un état transitoire au moment de la lecture. L'utilisation et la manipulation des valeurs de ces données peuvent entraîner des incohérences au niveau des données. Pour notre exemple, tiré de l'étude de cas, une autre transaction peut voir les valeurs intermédiaires lors du rafraîchissement d'une liste de todos. Cela signifie que certaines valeurs correspondent à l'ancienne liste non rafraîchie et d'autres à la nouvelle rafraîchie. La liste n'est donc pas cohérente dans sa globalité.
Non-Repeatable Read (lecture non reproductible)	Une lecture récupère des données dans un certain état dans la transaction. Si une application modifie ces données parallèlement, la même lecture renvoie ces données modifiées. La transaction n'est donc pas complètement isolée vis-à-vis des données modifiées par les autres transactions. De ce fait, les lectures ne sont pas reproductibles. Le cas se produit si une transaction travaille sur un todo qui est modifié par une autre transaction de manière concourante dans le cadre d'une importation d'une liste de todos. Cette modification est répercutée avant la fin de la première transaction, et l'enregistrement n'a pas la même valeur tout au long de l'autre transaction.
Phantom Read (lecture fantôme)	Contrairement aux lectures non reproductibles, qui impliquent une modification des données, les lectures fantômes sont la conséquence d'insertions ou de suppressions de données entre leur lecture et leur utilisation. Le cas se produit si une transaction travaille sur un todo qui est supprimé par une autre transaction de manière concourante dans le cadre de l'importation d'une liste de todos. Cette suppression est répercutée avant la fin de la première transaction, et l'enregistrement n'existe plus dans l'autre.

Il est important de bien cerner les problèmes éventuels pour chaque niveau d'isolation et de positionner ces niveaux avec précaution. Ces problèmes étant tous issus d'accès

concourants aux données, le développeur de l'application doit identifier avec précision les différents traitements possibles accédant simultanément aux données.

Les bases de données relationnelles définissent plusieurs niveaux d'isolation. Le tableau 12.2 les récapitule, du plus large au plus restrictif.

Tableau 12.2. Niveaux d'isolation des bases de données relationnelles

Niveau d'isolation	Description
`TRANSACTION_NONE`	Les transactions ne sont pas supportées.
`TRANSACTION_READ_UNCOMMITTED`	L'application peut rencontrer les trois problèmes évoqués au tableau 12.1. En cas d'absence d'accès concourant aux données, ce niveau peut être utilisé afin d'améliorer les performances. Ce n'est toutefois pas conseillé, car la cohérence des données n'est pas complètement garantie.
`TRANSACTION_READ_COMMITTED`	Permet de résoudre les lectures non reproductibles, mais l'application peut éventuellement rencontrer les deux autres problèmes.
`TRANSACTION_REPEATABLE_READ`	Permet de résoudre les lectures non reproductibles et les lectures sales. L'application peut toutefois rencontrer des lectures fantômes. Il s'agit du niveau d'isolation par défaut de beaucoup de serveurs de bases de données, dont Oracle.
`TRANSACTION_SERIALIZABLE`	Permet de résoudre les trois problèmes, mais au prix d'une dégradation des performances du fait des verrous posés sur les données accédées.

Pour garantir l'intégrité des données lors d'une transaction, il est impératif de positionner le niveau d'isolation des données de manière appropriée. Il convient pour cela d'évaluer avec précision les différents types d'accès aux données, notamment les accès concourants en modification.

Plus le niveau d'isolation est restrictif, plus la qualité et la cohérence des données sont garanties. Cependant, comme nous allons le voir par la suite, une gestion de la concourance est souvent nécessaire au niveau applicatif.

Types de transactions

Il existe deux grands types de transactions, les transactions locales et les transactions globales. Leurs principales propriétés sont les suivantes :

- **Transactions locales.** Adaptées lorsqu'une seule ressource transactionnelle est utilisée. Dans ce cas, le gestionnaire de transactions est la ressource elle-même. Dans l'exemple précédent d'importation, toutes les données sont importées dans la même source de stockage des données.
- **Transactions globales.** Utilisables si une ou plusieurs ressources sont présentes. À partir de deux ressources, nous sommes toujours en présence de transactions globales. En cas d'utilisation des transactions globales, le gestionnaire de transactions est externalisé par rapport aux ressources et est capable de dialoguer avec elles grâce à des interfaces normalisées. Dans notre exemple, les données sont importées dans différentes sources de stockage des données.

Ressource transactionnelle

On appelle ressource transactionnelle tout module d'accès à un système d'information, gérant des transactions et offrant une interface de programmation afin de les configurer et de les utiliser.

L'utilisation d'une seule base de données n'implique pas nécessairement des transactions locales. Par exemple, si l'application utilise en plus JMS comme middleware de messagerie applicative, les transactions globales sont nécessaires afin de réaliser des transactions utilisant les deux ressources.

Les transactions globales sont cependant beaucoup plus difficiles à mettre en œuvre. En effet, ce type de transaction nécessite la mise en œuvre d'un gestionnaire de transactions externe. Ce dernier est souvent fourni par le serveur d'applications. L'inconvénient de ce type de mécanisme vient de la synchronisation des états transactionnels des ressources et de la résolution des incidents. Les systèmes peuvent se retrouver dans des états incohérents, lesquels doivent être résolus par le biais de choix arbitraires.

Une autre grande différence entre les transactions locales et globales est que ces dernières se fondent sur l'API uniformisée JTA (Java Transaction API), alors que les transactions locales s'appuient directement sur les API des technologies ou frameworks utilisés. Nous retrouvons cependant toujours des méthodes similaires pour démarrer une transaction, ainsi qu'une méthode pour la valider et une méthode pour l'annuler.

Cette section ne détaille pas les différentes façons de gérer les transactions locales suivant les technologies et frameworks utilisés mais se concentre sur la façon de les utiliser avec le framework Spring. Ce dernier masque toute cette complexité et cette diversité derrière une API générique.

Gestion des transactions

Plusieurs opérations peuvent être réalisées afin de manipuler les transactions. Il est en outre généralement possible de détecter les événements du cycle de vie d'une transaction.

Démarquer une transaction

Une transaction est limitée par un début et une fin. Cette dernière peut survenir aussi bien lors d'une validation, en cas de succès, que d'une annulation, en cas d'échec. La démarcation consiste à spécifier le commencement et la fin de la transaction.

Le code suivant montre comment démarquer une transaction avec JDBC :

```
Connection connection=null;
try {
   //Récupération de la connexion
   connection=getConnection();
```

```
   //Démarrage de la transaction
   connection.setAutoCommit(false);

   //Exécution des traitements JDBC de l'application
   (...)

   //Validation de la transaction
   connection.commit();
   connection.setAutoCommit(true);
} catch(SQLException ex) {
   //Gestion des exceptions

   //Annulation de la transaction
   try {
      connection.rollback();
      connection.setAutoCommit(true);
   } catch(SQLException ex) { }
} finally {
   //Libération des ressources JDBC
   try {
      if( connection!=null ) {
         connection.close();
      }
   } catch(SQLException ex) { }
}
```

En cas de succès d'une transaction, le terme « commit » est communément employé. Dans le cas contraire, on parle de « rollback ». Toutes les modifications entre le début et la validation ou annulation de la transaction sont atomiques.

Suspendre et reprendre une transaction

Certains frameworks ou technologies permettent de suspendre une transaction dans le but de réaliser des traitements et de la reprendre une fois ceux-ci terminés. L'implémentation de ces actions se fait de manière différente suivant le type de transaction choisi.

Dans le cas des transactions locales, la suspension consiste à ne plus utiliser la connexion attachée à la transaction tout en la mémorisant, puisqu'elle sera réutilisée quand la transaction reprendra. Lors de la suspension, une nouvelle connexion est utilisée. Nous pouvons en déduire que la suspension est logique et non physique et qu'il incombe au framework qui offre cette fonctionnalité de prendre en charge l'implémentation du mécanisme de suspension et de reprise.

En ce qui concerne les transactions globales, la suspension tout comme la reprise se réalisent directement en utilisant le gestionnaire de transactions et les méthodes `suspend` et `resume`. La première renvoie la transaction courante suspendue. Pour la reprendre, la méthode `resume` est appelée avec pour paramètre cette transaction. L'application doit donc avoir accès au gestionnaire de transactions globales.

Le code suivant évoque la manière de suspendre et de reprendre une transaction avec JTA, l'API J2EE de gestion des transactions globales :

```
// Récupération du gestionnaire de transactions
TransactionManager tm=getTransactionManager();
// Suspension de la transaction courante
Transaction transaction=tm.suspend();
(...)
// Reprise de cette transaction
tm.resume(transaction);
```

Gérer le cycle de vie d'une transaction

Certains frameworks ou technologies offrent la possibilité de déclencher des traitements à différents moments du cycle de vie de la transaction. Le terme de synchronisation est communément utilisé pour désigner ce mécanisme.

Les événements du cycle de vie pris en compte se produisent essentiellement au moment de la validation ou de la finalisation de la transaction. Cela permet de libérer des ressources si nécessaire. On distingue à cet effet les événements avant la validation d'une transaction pour une ressource transactionnelle (« commit ») et ceux avant ou après la finalisation de la transaction. Des événements sur la suspension/reprise d'une transaction peuvent également être supportés.

Dans certains cas, comme avec JTA, ce mécanisme est directement intégré à l'API de gestion des transactions, tandis que des frameworks tels que Spring déclenchent les événements au cours du cycle de vie de la transaction.

Les transactions définissent de manière classique différents événements. Ces événements, récapitulés au tableau 12.3, ne sont toutefois pas tous supportés par toutes les technologies ou tous les frameworks.

Tableau 12.3. Comportements transactionnels

Événement	Description
Sur la suspension	Déclenché lorsque la transaction est suspendue par l'application.
Sur la reprise	Déclenché lorsque l'application reprend le déroulement de la transaction.
Avant la validation	Déclenché avant qu'une ressource transactionnelle soit validée.
Avant la finalisation	Déclenché avant que la transaction soit validée dans sa globalité.
Après la finalisation	Déclenché après que la transaction a été validée dans sa globalité.

Types de comportements transactionnels

Lorsqu'un composant est rendu transactionnel, il peut être configuré afin de spécifier le comportement transactionnel de ses méthodes. La plupart des frameworks ou technologies gérant les transactions définissent des mots-clés pour cela.

Le tableau 12.4 récapitule ces mots-clés. Il ne s'agit pas à proprement parler d'une spécification, même si ces mots-clés sont utilisés plus ou moins entièrement par des frameworks ou technologies tels que Spring ou les EJB.

Tableau 12.4. Mots-clés de comportements transactionnels

Mot-clé de comportement transactionnel	Description
`REQUIRED`	La méthode doit forcément être exécutée dans un contexte transactionnel existant. S'il n'existe pas lors de l'appel, il est créé.
`SUPPORTS`	La méthode peut être exécutée dans un contexte transactionnel s'il existe. Dans le cas contraire, la méthode est tout de même exécutée, mais hors d'un contexte transactionnel.
`MANDATORY`	La méthode doit forcément être exécutée dans un contexte transactionnel. Si tel n'est pas le cas, une exception est levée.
`REQUIRED_NEW`	La méthode impose la création d'une nouvelle transaction pour la méthode.
`NOT_SUPPORTED`	La méthode ne supporte pas les transactions. Si un contexte transactionnel existe lors de son appel, celui-ci est suspendu.
`NEVER`	La méthode ne doit pas être exécutée dans un contexte transactionnel. Si tel est le cas, une exception est levée.
`NESTED`	La méthode est exécutée dans une transaction imbriquée si un contexte transactionnel existe lors de son appel.

Ces mots-clés sont utiles lors d'appels entre des méthodes de différents composants dans un contexte transactionnel, comme des appels entre services métier.

La figure 12.1 illustre ce qui se passe au niveau des transactions pour ce type d'appel. La méthode des premier et troisième composants nécessite une transaction (`REQUIRED`), et celle du second aucune (`NOT_SUPPORTED`). La transaction initiée par le premier composant est donc suspendue le temps d'exécuter la méthode du second puis reprend pour exécuter le troisième.

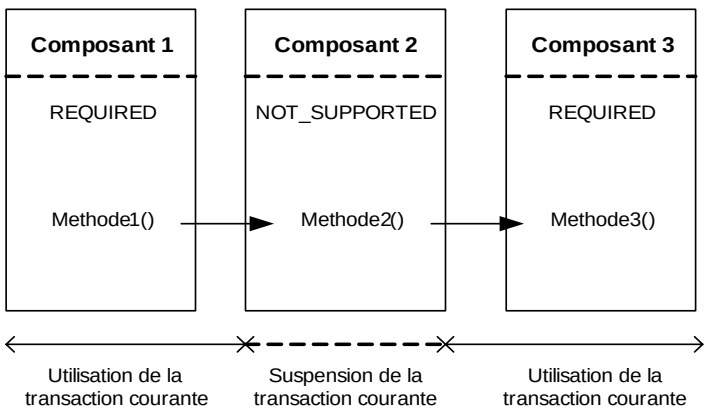


Figure 12.1
Exemple de mécanisme transactionnel fondé sur les comportements

Le tableau 12.5 récapitule la transaction utilisée par un composant suivant le type de comportement transactionnel spécifié et la transaction en cours, T1 et T2 désignant des transactions différentes.

Tableau 12.5. Types de comportements et transactions

Type de comportement transactionnel	Transaction initiale	Transaction utilisée
`REQUIRED`	Aucune T1	T1 T1
`SUPPORTS`	Aucune T1	Aucune T1
`MANDATORY`	Aucune T1	Erreur T1
`REQUIRED_NEW`	Aucune T1	T1 T2
`NOT_SUPPORTED`	Aucune T1	Aucune Aucune
`NEVER`	Aucune T1	Aucune Erreur
`NESTED`	Aucune T1	T1 T1 (imbriquée)

Ressources transactionnelles exposées

Afin de pouvoir utiliser les transactions globales, les fournisseurs de services étendent leurs fabriques de connexions afin de les rendre compatibles XA. La spécification XA (eXtended Architecture) standardise les interactions entre les gestionnaires de ressources et le gestionnaire de transactions afin de mettre en œuvre des protocoles transactionnels. Cela permet de faire participer ces fabriques à une transaction globale.

Le point fondamental pour le composant utilisant la technologie ou le framework est que la fabrique exposée est toujours la fabrique simple, et non celle compatible XA. Un composant qui utilise les transactions locales ou globales ne voit donc pas de différence quant aux entités qu'il manipule. Le fournisseur de services ou le serveur d'applications a pour sa part la responsabilité de masquer la fabrique XA derrière une fabrique classique, de fournir une connexion classique et d'ajouter et d'enlever la ressource du contexte transactionnel.

Le passage des transactions locales aux transactions globales n'a donc aucun impact sur l'utilisation des ressources. La modification est localisée au niveau de la gestion de la démarcation. Si celle-ci est déclarative, le composant n'est pas impacté par un changement de type de transaction. Cela favorise fortement la réutilisation des composants dans différentes architectures.

Concourance d'accès et transactions

Les applications J2EE utilisant par essence plusieurs fils d'exécution pour gérer leurs différents traitements, plusieurs accès ou requêtes de clients peuvent être concourants.

Afin d'éviter les écrasements de données entre utilisateurs, les techniques de verrouillage suivantes peuvent être mises en œuvre (les techniques décrites ci-après sont orientées base de données relationnelle) :

- **Verrouillage pessimiste.** Ce mécanisme de verrouillage fort est géré directement par le système de stockage des données. Pendant toute la durée du verrou, aucune autre application ou fil d'exécution ne peut accéder à la donnée. Pour les bases de données relationnelles, cela se gère directement au niveau du langage SQL. À l'image d'Hibernate, plusieurs frameworks facilitent l'utilisation de ce type de verrou. Une requête SQL de type `select for update` est alors utilisée.

  Ce verrouillage particulièrement restrictif peut impacter les performances des applications. En effet, ces dernières ne peuvent accéder à l'enregistrement tant que le verrou n'est pas levé. Des fils d'exécution peuvent donc rester en attente et pénaliser les traitements de l'application.

- **Verrouillage optimiste.** Ce type de verrouillage est plus large et doit être implémenté dans l'application elle-même. Il ne nécessite pas de verrou dans le système de stockage des données. Pour les bases de données relationnelles, il est généralement implémenté en ajoutant une colonne aux différentes tables impactées. Cette colonne représente une version ou un indicateur de temps indiquant l'état de l'enregistrement lorsqu'il est lu. De ce fait, si cet état change entre la lecture et la modification, nous nous trouvons dans le cas d'un accès concourant.

  L'application peut implémenter plusieurs stratégies pour résoudre ce problème. L'utilisateur peut être averti, par exemple, une interface conviviale lui offrant la possibilité de voir les modifications réalisées par un autre utilisateur conjointement avec les siennes. Il peut dès lors effectuer ses mises à jour. L'application peut aussi ne pas notifier l'utilisateur et implémenter un mécanisme afin de fusionner les différentes données.

En résumé

Situées au cœur des applications d'entreprise, les transactions visent à garantir l'intégrité des données des systèmes d'information. Du fait qu'elles mettent en jeu de nombreuses notions qui peuvent être complexes, il est préférable de les mettre en œuvre en utilisant des technologies ou frameworks encapsulant toute leur complexité technique.

La section suivante détaille la façon de les mettre en œuvre, ainsi que les pièges à éviter et la solution fournie par le framework Spring.

Mise en œuvre des transactions

Depuis le début de ce chapitre, nous avons rappelé les différentes propriétés des transactions. Nous abordons à présent leur mise en œuvre optimale et de la façon la plus transparente possible pour les composants des applications Java/J2EE. L'objectif visé n'est pas d'incorporer les mécanismes transactionnels dans le code des composants applicatifs mais de le spécifier au moment de leur assemblage.

Intégrer les notions transactionnelles dans l'architecture d'une application n'est pas chose aisée, tant les concepts en jeu sont nombreux. Les principaux défis sont de conserver la modularité des composants, l'isolation des couches et la séparation des codes technique et métier.

Afin de concilier les bonnes pratiques abordées au cours des chapitres précédents et la gestion transactionnelle, la démarcation doit être correctement appliquée et le comportement transactionnel des composants judicieusement utilisé. Pour cela, l'utilisation de frameworks ou de technologies appropriés est indispensable.

Gestion de la démarcation

La démarcation transactionnelle doit être réalisée au niveau des services métier. Ces derniers peuvent s'appuyer sur plusieurs composants d'accès aux données. Plusieurs appels à des méthodes de ces composants sous-jacents peuvent donc être réalisés dans une même transaction.

Le support des transactions doit de plus être suffisamment flexible pour permettre de gérer les appels entre services. La définition de types de comportement transactionnel rend ce support flexible et déclaratif. Spring implémente ces stratégies dans sa gestion des transactions.

La figure 12.2 illustre les couches applicatives impactées par la gestion des transactions, qui est une problématique transversale.

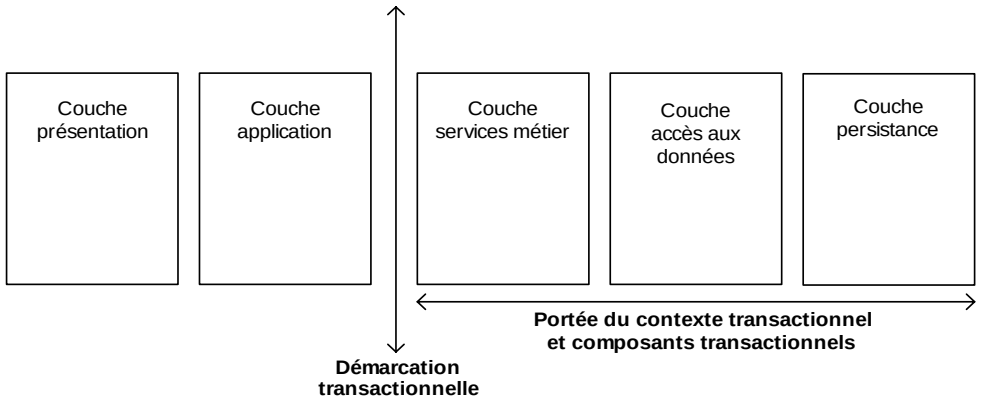


Figure 12.2

Impact de la gestion transactionnelle sur les couches applicatives

Dans notre étude de cas, l'application des comportements transactionnels est mise en œuvre sur les composants du package `tudu.service.impl`.

Mauvaises pratiques et anti-patterns

La structuration des préoccupations en couches constitue une bonne pratique élémentaire, chaque couche ne devant avoir connaissance que de la couche immédiatement inférieure.

Les composants services métier ne doivent s'appuyer que sur les composants d'accès aux données et ne peuvent en aucun cas avoir connaissance des API utilisées par ces composants pour accéder aux données. Cependant, la tentation est grande d'utiliser les API de persistance pour démarquer les transactions au niveau de la couche métier et de passer ensuite ces instances aux couches inférieures. Par exemple, l'utilisation d'une connexion JDBC ou d'une session d'un ORM dans la couche service métier est un anti-pattern.

Le code suivant est un bon exemple de mise en œuvre de cet anti-pattern. Il montre un composant de la couche service métier qui s'appuie sur une connexion JDBC afin de débuter et valider ou d'annuler une transaction locale. Cette connexion est ensuite passée au composant d'accès aux données utilisé afin d'inclure les traitements du composant dans la transaction.

Nous considérons dans ce code que l'instance `myDao` du composant d'accès aux données a été correctement récupérée précédemment :

```
Connection connection=null;
try {
   //Récupération de la connexion
   connection=getConnection();

   //Démarrage de la transaction
   beginTransaction(connection);

   //Exécution des traitements du DAO utilisé
   myDao.update(connection,myEntity);

   //Validation de la transaction
   commitTransaction(connection);
} catch(SQLException ex) {
   //Gestion des exceptions

   //Annulation de la transaction
   rollbackTransaction(connection);
} finally {
   //Libération des ressources JDBC
   closeConnection(connection);
}
```

Il existe un couplage fort entre les technologies utilisées par les composants de la couche d'accès aux données et les services métier, ces derniers utilisant ces technologies explicitement. Cette pratique nuit grandement à la flexibilité, à la modularité, à l'évolutivité et à la séparation des préoccupations.

Une bonne pratique consiste à masquer l'utilisation de ces API derrière une API générique de gestion transactionnelle. Cette API doit être programmée à l'aide d'interfaces afin de cacher l'implémentation de ce gestionnaire. Ce dernier peut éventuellement s'appuyer sur un contexte transactionnel pour le fil d'exécution et être stocké dans une instance de type `ThreadLocal`. Cette classe de base de Java permet de garder une instance pouvant rester accessible pour tout un fil d'exécution.

La section suivante détaille comment Spring permet de gérer facilement et de façon modulaire les transactions dans les applications Java-J2EE à l'aide de bonnes pratiques de conception et de fonctionnalités permettant de spécifier des comportements transactionnels de manière déclarative.

L'approche de Spring

L'une des fonctionnalités les plus attractives de Spring est indiscutablement celle qui concerne la gestion des transactions, car elle offre une souplesse et une facilité d'utilisation sans égales dans le monde Java/J2EE.

La stratégie de Spring est en outre entièrement configurable et modulaire. Nous verrons qu'il existe deux approches pour gérer la démarcation, s'appuyant toutes deux sur une API de démarcation générique, et plusieurs implémentations de la stratégie transactionnelle en fonction des ressources utilisées. Spring permet ainsi de s'adapter aux besoins de l'application et de maîtriser le degré d'intrusivité ainsi que les performances de la gestion des transactions en utilisant le type de transaction approprié.

Spring permet en outre de définir des comportements transactionnels sur les composants par déclaration.

Une API générique de démarcation

Les concepteurs de Spring ont identifié le besoin d'une API générique afin de gérer la démarcation transactionnelle et de spécifier le comportement transactionnel d'un composant.

Cette API comporte deux grandes parties, la partie cliente et la partie fournisseur de services.

Partie cliente

La partie cliente permet de démarquer explicitement la transaction, soit directement avec les API transactionnelles de Spring, soit indirectement avec un template transactionnel ou un intercepteur.

Toutes les classes et interfaces décrites dans cette section sont localisées dans le package `org.springframework.transaction` du framework.

Le premier élément du support transactionnel est l'interface `PlatformTransactionManager`, qui permet de démarquer une transaction, et ce, quelles que soient les ressources et stratégies transactionnelles utilisées. Cette interface fournit des méthodes de validation et d'annulation pour la transaction courante :

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(
             TransactionDefinition definition)
                     throws TransactionException;
    void commit(TransactionStatus status)
                     throws TransactionException;
    void rollback(TransactionStatus status)
                     throws TransactionException;
}
```

Pour commencer une transaction, les propriétés et comportements transactionnels suivants doivent être spécifiés :

- isolation transactionnelle ;
- type de comportement transactionnel ;
- temps d'expiration des transactions ;
- statut lecture seule.

Ces propriétés sont contenues dans l'interface `TransactionDefinition` suivante :

```
public interface TransactionDefinition {
    int PROPAGATION_REQUIRED = 0;
    int PROPAGATION_SUPPORTS = 1;
    int PROPAGATION_MANDATORY = 2;
    int PROPAGATION_REQUIRES_NEW = 3;
    int PROPAGATION_NOT_SUPPORTED = 4;
    int PROPAGATION_NEVER = 5;
    int PROPAGATION_NESTED = 6;

    int ISOLATION_DEFAULT = -1;
    int ISOLATION_READ_UNCOMMITTED =
                        Connection.TRANSACTION_READ_UNCOMMITTED;
    int ISOLATION_READ_COMMITTED =
                        Connection.TRANSACTION_READ_COMMITTED;
    int ISOLATION_REPEATABLE_READ =
                        Connection.TRANSACTION_REPEATABLE_READ;
    int ISOLATION_SERIALIZABLE =
                        Connection.TRANSACTION_SERIALIZABLE;
```

```
    int getPropagationBehavior();
    int getIsolationLevel();
    int getTimeout();
    boolean isReadOnly();
    String getName();
}
```

Spring supporte tous les types de comportements transactionnels décrits précédemment. Les mots-clés correspondants diffèrent toutefois légèrement des mots-clés généraux, comme le montre le tableau 12.6.

Tableau 12.6 Comportements transactionnels de Spring

Comportement transactionnel général	Mot-clé Spring
REQUIRED	PROPAGATION_REQUIRED
SUPPORTS	PROPAGATION_SUPPORTS
MANDATORY	PROPAGATION_MANDATORY
REQUIRED_NEW	PROPAGATION_REQUIRED_NEW
NOT_SUPPORTED	PROPAGATION_NOT_SUPPORTED
NEVER	PROPAGATION_NEVER
NESTED	PROPAGATION_NESTED

Une fois la transaction démarrée, Spring impose de conserver une instance de son statut, matérialisée par l'interface `TransactionStatus` suivante :

```
public interface TransactionStatus {
    boolean isNewTransaction();
    void setRollbackOnly();
    boolean isRollbackOnly();
}
```

En plus des méthodes de visualisation de propriétés de la transaction courante (méthode `isNewTransaction` et `isRollbackOnly`), cette interface définit une méthode `setRollbackOnly`, qui permet de spécifier que la transaction doit être annulée quels que soient les traitements ultérieurs. Ce mécanisme est semblable à celui des EJB, si ce n'est que, dans le cas de Spring, il est uniformisé pour tous les types de transactions, locales comme globales, ou de ressources pour ce qui concerne les seules transactions locales.

Lors de l'utilisation directe de l'API cliente de Spring, la gestion des exceptions et du comportement transactionnel est de la responsabilité de l'application, alors que, en cas d'utilisation du template transactionnel, la levée d'une exception implique forcément, par défaut, une annulation de la transaction.

Nous verrons en détail l'utilisation de cette API à la section décrivant la façon de démarquer les transactions.

Partie fournisseur de services

La partie fournisseur de services de l'API générique est constituée par les implémentations de l'interface `PlatformTransactionManager`. Spring les désigne sous la dénomination de gestionnaire de transactions et fournit une multitude d'intégrations avec différentes technologies et frameworks d'accès aux données, qui se fondent sur les ressources de la technologie ou du framework.

Les tableaux 12.7 à 12.10 récapitulent ces différentes implémentations technologie par technologie. Ces implémentations sont localisées dans les packages correspondant aux technologies ou frameworks utilisés.

Tableau 12.7. Implémentation fondée sur JDBC

Technologie	Gestionnaire de transactions	Ressource utilisée
JDBC	`DataSourceTransactionManager`	`DataSource`

Tableau 12.8. Implémentations fondées sur des frameworks ORM

Framework	Gestionnaire de transactions	Ressource utilisée
Hibernate 2.1	`HibernateTransactionManager`	`SessionFactory`
Hibernate 3	`HibernateTransactionManager`	`SessionFactory`
iBATIS	`DataSourceTransactionManager`	`DataSource`
JDO	`JdoTransactionManager`	`PersistenceManagerFactory`
OJB	`PersistenceBrokerTransaction-Manager`	`DataSource`
TopLink	`TopLinkTransactionManager`	`SessionFactory` (dans le cas de TopLink, cette classe est spécifique de Spring et non de l'outil), `DataSource`.

Tableau 12.9. Implémentations fondées sur des middlewares

Middleware	Gestionnaire de transactions	Ressource utilisée
JMS 1.02	`JmsTransactionManager102`	`ConnectionFactory` (dans le cas de JMS 1.02, il est nécessaire de spécifier le type de ressource, `Queue` ou `Topic`, avec la propriété `pubSubDomain`).
JMS 1.1	`JmsTransactionManager`	ConnectionFactory
JCA	`CciLocalTransactionManager`	ConnectionFactory

Tableau 12.10. Implémentation fondée sur les transactions XA

Technologie	Gestionnaire de transactions	Ressource utilisée
XA	`JtaTransactionManager`	`UserTransaction`, `TransactionManager`

Injection du gestionnaire de transactions

Pour utiliser la gestion des transactions de Spring, le choix du gestionnaire de transactions est primordial. En fonction du type de démarcation utilisé, ce gestionnaire n'est pas injecté sur les mêmes entités.

Dans le cas de la démarcation par programmation, le composant service utilise ce gestionnaire directement ou *via* le template transactionnel. Le gestionnaire de transactions doit donc être injecté sur le composant. Cette injection est configurée dans le fichier de configuration **applicationContext.xml** localisé dans le répertoire **WEB-INF :**

```
<bean id="transactionManager"
   class="org.springframework.orm.
                      hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
        <ref bean="sessionFactory"/>
    </property>
</bean>

<bean id="todosManager" class="tudu.service.impl.TodosManagerImpl">
    (...)
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
</bean>
```

Dans le cas de la démarcation déclarative, cette responsabilité incombe désormais à l'intercepteur transactionnel de Spring. Le gestionnaire doit donc être injecté sur l'intercepteur :

```
<bean id="transactionManager"
   class="org.springframework.orm.
                      hibernate.HibernateTransactionManager">
    <property name="sessionFactory">
        <ref bean="sessionFactory"/>
    </property>
</bean>

<bean id="todoListsManager"
      class="org.springframework.transaction
                       .interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
    (...)
</bean>
```

L'étude de cas Tudu Lists utilise cette approche afin de spécifier les comportements transactionnels sur les composants. L'injection du gestionnaire de transactions se fait donc de cette manière. Cette configuration est réalisée dans le fichier **applicationContext.xml,** localisé dans le répertoire **/WEB-INF.**

Gestion de la démarcation

Maintenant que nous avons décrit les principes de la gestion des transactions avec Spring, nous pouvons détailler les différentes approches de gestion de la démarcation, par programmation et par déclaration.

Gestion de la démarcation par programmation

Spring offre deux façons de réaliser cette démarcation. La première utilise directement les API génériques de Spring, dont l'interface principale est `PlatformTransactionManager`. La gestion des exceptions est à la charge du développeur. La seconde utilise la classe `TransactionTemplate`, le template de Spring dédié à la gestion des transactions fournissant une méthode de rappel à implémenter avec les traitements de la transaction. Dans ce cas, le développeur n'a qu'à se concentrer sur les traitements spécifiques de l'application.

La définition des propriétés transactionnelles est effectuée grâce aux implémentations de l'interface `TransactionDefinition`. La plus communément utilisée est la classe `DefaultTransactionDefinition`, mais il en existe d'autres, comme la classe `RuleBasedTransactionAttribute`. Ces différentes implémentations se trouvent toutes dans le package `org.springframework.transaction.support`.

Cette interface permet de spécifier différentes constantes concernant la propagation transactionnelle et les niveaux d'isolation. Elle offre également des accesseurs sur le niveau d'isolation, le nom de la transaction, le type de propagation, le délai d'expiration, l'attribut lecture seule, etc. Cette interface a déjà été abordée à la section « Partie cliente » de ce chapitre.

La définition des comportements face aux différentes exceptions ne peut être configurée avec cette approche. La responsabilité en incombe à l'application, qui utilise directement les API transactionnelles de Spring. Nous verrons avec l'approche déclarative que Spring offre la possibilité de configurer ce comportement et de rendre ainsi la démarcation des transactions particulièrement flexible.

Le code suivant illustre l'utilisation des API génériques de Spring afin de réaliser une démarcation transactionnelle. Dans cet exemple, le gestionnaire de transactions (instance `transactionManager`) de Spring est injecté à l'aide des fonctionnalités dédiées de Spring et est donc disponible pour un composant de la couche service métier tel que l'implémentation `TodosManagerImpl` (package `tudu.service.impl`) de l'étude de cas. Ce composant contient désormais le code suivant :

```
1  DefaultTransactionDefinition def=
2                      new DefaultTransactionDefinition();
3  def.setPropagationBehavior(
4                      TransactionDefinition.PROPAGATION_REQUIRED);
5
6  TransactionStatus status=transactionManager.getTransaction(def);
7
```

```
 8  try {
 9     // Différents traitements métier de l'application ou
10    // utilisation de composants d'accès aux données
11 } catch (BusinessException ex) {
12     transactionManager.rollback(status);
13     throw ex;
14 }
15 transactionManager.commit(status);
```

Les lignes 1 à 4 permettent de spécifier les propriétés et le comportement transactionnels des traitements. La ligne 6 marque le début de la transaction en s'appuyant sur le gestionnaire de transactions injecté avec Spring. La fin de la transaction peut être réalisée de deux manières, toujours en s'appuyant sur le gestionnaire.

Si tout se passe bien, la transaction est validée à la ligne 15 ; si une exception se produit, la transaction est annulée à la ligne 12.

Tous les traitements du bloc `try/catch` fondés sur la même technologie que le gestionnaire sont automatiquement inclus dans la transaction. Si l'approche d'enregistrement automatique dans le contexte transactionnel n'est pas utilisée, les composants d'accès aux données doivent nécessairement utiliser les méthodes utilitaires de Spring pour récupérer et relâcher les connexions ou sessions.

Le code suivant illustre l'utilisation du template transactionnel de Spring pour réaliser une démarcation transactionnelle dans un composant de la couche service métier tel que l'implémentation `TodosManagerImpl` (package `tudu.service.impl`) de l'étude de cas. Dans cet exemple, le gestionnaire de transactions (instance `transactionManager`) de Spring est également injecté à l'aide de l'injection de dépendance :

```
 1  TransactionTemplate template=new TransactionTemplate();
 2  template.setTransactionManager(transactionManager);
 3
 4  Object result=template.execute(new TransactionCallback() {
 5      public Object doInTransaction(TransactionStatus status) {
 6          // Différents traitements métier de l'application ou
 7          // utilisation de composants d'accès aux données
 8          return (...);
 9      }
10 });
```

L'appel au template se fait à la ligne 4, lors de l'invocation de la méthode `execute`. Cette dernière attend comme paramètre une implémentation de l'interface `TransactionCallback`, spécifiant les traitements à réaliser dans la transaction. En effet, la méthode `execute` démarre tout d'abord une transaction en se fondant sur le gestionnaire spécifié puis appelle la méthode de rappel `doInTransaction` et valide ou annule la transaction suivant le résultat des traitements (exceptions non vérifiées levées).

Gestion de la démarcation par déclaration

La démarcation par déclaration est à utiliser en priorité, car elle n'est pas intrusive pour le composant service métier. Le code technique des transactions est en effet externalisé par rapport au code métier du composant. De plus, le comportement transactionnel peut être spécifié à l'assemblage des composants.

Cette démarcation consiste d'abord à injecter dans l'intercepteur de gestion des transactions les propriétés de la transaction sous forme de chaînes de caractères dont les différents éléments sont séparés par des virgules. Spring implémente sur ces éléments un mécanisme permettant de reconnaître le type de la propriété :

- Si l'élément commence par `PROPAGATION_`, il désigne un type de comportement transactionnel.
- Si l'élément commence par `ISOLATION_`, il désigne le niveau d'isolation.
- Si l'élément commence par `timeout_`, il spécifie le délai d'attente de la transaction.
- Si l'élément correspond à `readOnly`, la transaction est en lecture seule.
- Si l'élément commence par le caractère `+`, la levée de l'exception dont le nom suit ce caractère provoque une validation de la transaction.
- Si l'élément commence par le caractère `-`, la levée de l'exception dont le nom suit ce caractère provoque une annulation de la transaction.

Les chaînes de description des propriétés transactionnelles sont de la forme :

```
PROPAGATION_REQUIRED,-MyCheckedException
PROPAGATION_REQUIRED,ISOLATION_REPEATABLE_READ
PROPAGATION_REQUIRED,readOnly
```

Voyons maintenant comment associer ces propriétés aux intercepteurs transactionnels de Spring.

Spring permet d'injecter ces propriétés transactionnelles grâce à deux types de propriétés :

- propriété de type `Properties` ;
- propriété de type `TransactionAttributeSource`.

Dans le premier cas, l'attribut se nomme `transactionAttributes` pour la classe `TransactionProxyFactoryBean` ou l'intercepteur `TransactionInterceptor`. Son utilisation se réalise par le biais du support du type `Properties` par le framework :

```
<bean id="todoListsManager"
      class="org.springframework.transaction
                      .interceptor.TransactionProxyFactoryBean">
    (...)
    <property name="transactionAttributes">
        <props>
```

```
            <prop key="insert*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

La clé des différentes propriétés constitue une expression régulière permettant de déterminer la ou les méthodes concernées. Dans l'exemple ci-dessus, la classe `TransactionInterceptor` peut être utilisée en lieu et place de la classe `TransactionProxyFactoryBean`.

Dans le second cas, l'attribut se nomme `transactionAttributeSource` pour la classe `TransactionProxyFactoryBean` ou l'intercepteur `TransactionInterceptor`. Comme Spring fournit le gestionnaire de propriétés (`PropertyEditor`) dédié, `TransactionAttributeEditor`, les attributs transactionnels peuvent être configurés sous forme de chaînes de caractères ou d'instances de la classe `TransactionAttributeSource`.

L'exemple suivant montre comment spécifier cette propriété en tant que chaîne de caractères :

```
<bean id="myTransactionInterceptor" class="org.springframework.
                  transaction.interceptor.TransactionInterceptor">
    (...)
    <property name="transactionAttributeSource">
        <value>
          MaClasse.maMethode*=PROPAGATION_REQUIRED
          MonAutreClasse.monAutreMethode=PROPAGATION_MANDATORY
        </value>
    </property>
</bean>
```

Dans cet exemple, la classe `TransactionProxyFactoryBean` peut être utilisée à la place de la classe `TransactionInterceptor`.

Pour configurer une instance de `TransactionAttributeSource`, une de ses nombreuses implémentations peut être utilisée :

- `AttributesTransactionAttributeSource`. S'appuie sur des annotations `common attributes` dans le ou les composants impactés.
- `MatchAlwaysTransactionAttributeSource`. S'applique à toutes les méthodes du ou des composants impactés.
- `MethodMapTransactionAttributeSource`. Utilise une table de hachage pour définir les attributs pour les méthodes.
- `NameMatchTransactionAttributeSource`. S'appuie sur les noms de méthodes pour déterminer les attributs transactionnels correspondants. L'approche fondée sur la classe `Properties` utilise implicitement cette classe.
- `AnnotationTransactionAttributeSource`. S'appuie sur les annotations Java 5 placées dans les composants impactés.

L'exemple suivant montre comment configurer les attributs transactionnels en se fondant explicitement sur une implémentation de `TransactionAttributeSource` :

```
<!-- Propriétés de la transaction -->
<bean id="matchAllWithPropReq" class="org.springframework.
  transaction.interceptor.MatchAlwaysTransactionAttributeSource">
    <property name="transactionAttribute">
        <value>PROPAGATION_REQUIRED</value>
    </property>
</bean>

<!-- Configuration de l'intercepteur transactionnel -->
<bean id="matchAllTxInterceptor" class="org.springframework.
                transaction.interceptor.TransactionInterceptor">
    (...)
    <property name="transactionAttributeSource">
        <ref bean="matchAllWithPropReq"/>
    </property>
</bean>
```

Nous allons voir comment utiliser les intercepteurs transactionnels de Spring, le framework offrant un certain nombre de fonctionnalités pour masquer la complexité de la POA dans le cadre des transactions. Comme le support des transactions par déclaration s'appuie sur le framework POA de Spring, l'application du comportement transactionnel sur des composants peut être réalisée composant par composant ou sur un ensemble de composants.

Utilisation de *TransactionProxyFactoryBean*

Dans ce cas, un proxy est défini devant le composant service métier impliqué. Ce proxy masque l'intercepteur transactionnel et simplifie donc l'utilisation de la POA pour les transactions.

Pour configurer ce mécanisme, il faut d'abord renommer l'identifiant du Bean service métier dans Spring. Une méthode classique consiste à le suffixer avec `Target`. L'identifiant originel du Bean est alors donné au proxy lui-même. Le proxy doit évidemment être relié au Bean cible par la propriété `target`. Il ne faut pas non plus oublier de spécifier le gestionnaire de transactions utilisé à l'aide de la propriété `transactionManager`.

L'application utilise désormais de manière transparente le proxy, lequel se fonde sur le Bean cible. Les composants qui utilisent ce dernier n'ont pas connaissance de ce changement de configuration.

Le code suivant illustre la mise en œuvre de cette approche dans l'étude de cas. Il est extrait du fichier de configuration XML **applicationContext.xml** situé dans le répertoire **WEB-INF :**

```
<bean id="todoListsManagerTarget"
      class="tudu.service.impl.TodoListsManagerImpl">
```

```
    <property name="todoListDAO">
        <ref bean="todoListDAO"/>
    </property>
    <property name="todoDAO">
        <ref bean="todoDAO"/>
    <property name="userManager">
        <ref bean="userManager"/>
    </property>
</bean>

<bean id="todoListsManager"
      class="org.springframework.transaction
                      .interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
    <property name="target">
        <ref bean="todoListsManagerTarget"/>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="create*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="delete*">PROPAGATION_REQUIRED</prop>
            <prop key="add*">PROPAGATION_REQUIRED</prop>
            <prop key="restore*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

Utilisation de *TransactionInterceptor*

Dans ce second cas, le framework POA de Spring est utilisé explicitement. Cette mise en œuvre est plus complexe, car elle nécessite une connaissance des notions de la POA (point de jonction, coupe, code advice et intercepteur) et l'utilisation de l'intercepteur transactionnel de Spring matérialisé par la classe `TransactionInterceptor`. Elle utilise la fonctionnalité Auto Proxy Creator de Spring pour une liste de Beans spécifiée ou la classe `ProxyFactoryBean` pour un Bean particulier.

Pour information, la classe `TransactionProxyFactoryBean` précédente s'appuie elle aussi sur cet intercepteur mais le masque, facilitant d'autant son utilisation.

Commençons par détailler l'utilisation du mécanisme Auto Proxy Creator. Une de ses implémentations est la classe `BeanNameAutoProxyCreator`, qui utilise le nom des Beans pour le tissage. Comme ce dernier est automatique, la modification du nom des Beans service métier n'est plus nécessaire.

Le code suivant illustre la façon de mettre en œuvre une configuration équivalente à la précédente, mais avec le mécanisme Auto Proxy Creator. Cette configuration est à placer dans le même fichier que précédemment :

```
<!-- Configuration de l'intercepteur transactionnel -->
<bean id="transactionInterceptor" class="org.springframework.
                  transaction.interceptor.TransactionInterceptor">
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="create*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="delete*">PROPAGATION_REQUIRED</prop>
            <prop key="add*">PROPAGATION_REQUIRED</prop>
            <prop key="restore*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>

<!-- Tissage de l'intercepteur -->
<bean id="autoProxyCreator" class="org.springframework.aop.
                    framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="interceptorNames">
        <list><idref local="transactionInterceptor"/></list>
    </property>
    <property name="beanNames">
        <list><idref local="todoListsManager"/></list>
    </property>
</bean>
```

Comme expliqué précédemment, des implémentations de l'interface `TransactionAttributeSource` peuvent être utilisées explicitement par le biais de la propriété `transactionAttributeSource`.

Utilisation des espaces de nommage

Comme nous l'avons vu précédemment, la version 2.0 du framework Spring introduit le support des espaces de nommage dans les fichiers de configuration des contextes.

Des espaces de nommage spécifiques ont été ajoutés afin d'adresser les problématiques relatives à la POA, ainsi qu'à la gestion des transactions et de simplifier leur configuration.

L'exemple suivant illustre l'adaptation de la configuration des transactions de la section précédente avec le support des espaces de nommage :

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

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:aop="http://www.springframework.org/schema/aop"
```

```
        xmlns:tx="http://www.springframework.org/schema/tx"
        xsi:schemaLocation=
             "http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans.xsd
             http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop.xsd
             http://www.springframework.org/schema/tx
  http://www.springframework.org/schema/transaction/spring-tx.xsd">

    (...)

    <aop:config>
        <aop:advisor
              pointcut="execution(* *..TodoListsManagerImpl.*(..))"
              advice-ref="txAdvice"/>
    </aop:config>

    <tx:advice id="txAdvice">
        <tx:attributes>
            <tx:method name="create*"/>
            <tx:method name="update*"/>
            <tx:method name="delete*"/>
            <tx:method name="add*"/>
            <tx:method name="restore*"/>
            <tx:method name="*" read-only="true"/>
        </tx:attributes>
    </tx:advice>

    (...)

</beans>
```

La configuration se réalise en deux étapes. La première consiste à configurer le tissage de l'aspect transactionnel avec la balise `config` pour l'espace de nommage `aop`. La seconde met en œuvre la configuration transactionnelle par le biais du code advice matérialisé par la balise `advice` pour l'espace de nommage `tx`. Afin d'alléger la configuration, cette balise utilise des valeurs par défaut pour les propriétés vues précédemment.

Nous détaillons dans la suite de ce chapitre une implémentation permettant de spécifier des comportements transactionnels avec des annotations Java 5. Pour plus de détails sur les autres implémentations, se reporter aux javadocs du framework Spring.

Synchronisation des transactions

Comme pour les EJB, Spring offre la possibilité à une classe d'être notifiée à certains moments du cycle de vie de la transaction courante. Il suffit d'utiliser l'interface `TransactionSynchronization` pour spécifier l'implémentation de la synchronisation et la classe `TransactionSynchronizationManager` pour l'enregistrer dans la transaction courante.

Le code suivant en donne un exemple d'utilisation :

```
TransactionSynchronizationManager.registerSynchronization(
    new TransactionSynchronization() {
        public void suspend() { }
        public void resume() { }
        public void beforeCommit(boolean readOnly) {
            System.out.println("before commit");
        }
        public void beforeCompletion() { }
        public void afterCompletion(final int status) {
            System.out.println("after completion");
        }
    });
```

Gestion des exceptions

Spring offre un mécanisme intéressant pour spécifier la manière de terminer une transaction lorsqu'une exception est levée. Le framework propose un comportement par défaut similaire à celui des EJB (validation pour les exceptions vérifiées et annulation pour les exceptions non vérifiées) mais peut aussi être complètement paramétré en utilisant les attributs de la transaction. Ce mécanisme ne peut être mis en œuvre qu'avec l'approche de gestion des transactions par déclaration.

Si, par exemple, la levée d'une exception vérifiée doit entraîner une annulation de la transaction, il suffit d'utiliser la configuration suivante :

```
(...)
<property name="transactionAttributes">
  <props>
    <prop key="create*">
       PROPAGATION_REQUIRED,-CheckedException
    </prop>
    <prop key="update*">PROPAGATION_REQUIRED</prop>
    <prop key="delete*">PROPAGATION_REQUIRED</prop>
    <prop key="add*">PROPAGATION_REQUIRED</prop>
    <prop key="restore*">PROPAGATION_REQUIRED</prop>
    <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
  </props>
</property>
(...)
```

Le signe - permet de définir une annulation lors de la levée de l'exception correspondante, et le signe + une validation.

Fonctionnalités avancées

Spring fournit quelques fonctionnalités intéressantes facilitant et allégeant la mise en œuvre des transactions dans une application Java/J2EE, comme l'utilisation transparente du contexte transactionnel, l'héritage des configurations transactionnelles ou les annotations.

Utilisation transparente du contexte transactionnel

Spring est devenu maître dans l'art d'intégrer des frameworks tiers dans une application de la manière la plus optimale et modulaire possible. Il fournit également un mécanisme intéressant afin d'ajouter un composant dans un contexte transactionnel géré par Spring de manière transparente, le composant n'ayant pas forcément besoin d'avoir été développé avec Spring.

Spring fournit ces types de proxy sur les ressources transactionnelles pour différentes technologies. Ces proxy ont par convention un nom commençant par `TransactionAware`. Ils intègrent automatiquement les connexions ou sessions dans le contexte transactionnel de Spring, si bien que les composants peuvent ne plus utiliser les classes utilitaires de Spring pour récupérer et relâcher ces ressources.

Le code suivant montre comment configurer ce mécanisme avec JDBC (cette mise en œuvre ne dépend pas de l'implémentation de l'interface `DataSource` choisie) :

```
<bean id="dataSourceTarget" class="org.springframework.jdbc.
                          datasource.DriverManagerDataSource">
    <property name="driverClassName">
        <value>org.hsqldb.jdbcDriver</value>
    </property>
    <property name="url">
        <value>jdbc:hsqldb:hsql://localhost:9001</value>
    </property>
    <property name="username"><value>sa</value></property>
    <property name="password"><value/></property>
</bean>

<bean id="dataSource" class="org.springframework.jdbc.
                  datasource.TransactionAwareDataSourceProxy">
    <property name="dataSource">
        <ref local="dataSourceTarget"/>
    </property>
</bean>
```

Héritage des configurations transactionnelles

Dans une entrée de son blog *(http://blog.exis.com/colin/),* Colin Sampaleanu, un des principaux développeurs de Spring, décrit une façon intéressante de configurer les transactions dans Spring.

Il montre comment utiliser l'héritage dans la configuration des Beans afin d'alléger la configuration transactionnelle. Il distingue une configuration transactionnelle globale et une configuration au cas par cas, héritant de la précédente. Le fichier XML de configuration de Spring est beaucoup plus concis dans le cas où beaucoup de composants transactionnels sont configurés :

```
<!-- Proxy transactionnel générique -->
<bean id="transactionProxy" abstract="true"
      class="org.springframework.transaction
                        .interceptor.TransactionProxyFactoryBean">
```

```
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="insert*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>

<!-- Bean transactionnel cible -->
<bean id="todoListsManagerTarget"
      class="tudu.service.impl.TodoListsManagerImpl">
    <property name="todoListDAO">
        <ref bean="todoListDAO"/>
    </property>
    <property name="todoDAO">
        <ref bean="todoDAO"/>
    <property name="userManager">
        <ref bean="userManager"/>
    </property>
</bean>

<!-- proxy transactionnel sur la cible -->
<bean id="todoListsManager" parent="transactionProxy">
    <property name="target">
        <ref bean="todoListsManagerTarget"/>
    </property>
</bean>
```

Cette approche est à utiliser essentiellement avec l'approche fondée sur la classe `TransactionProxyFactoryBean` et n'offre que peu d'intérêt avec le mécanisme Auto Proxy Creator.

Les annotations

La version 5 de Java introduit de nouveaux éléments de langages dont les annotations font partie. Les versions précédentes ne les supportent pas, mais il est éventuellement possible d'utiliser le framework Commons Attributes d'Apache afin de mettre en œuvre un mécanisme similaire. Spring supporte ces deux types de métadonnées incluses dans les sources.

Spring offre la possibilité de définir les comportements transactionnels des composants grâce à des annotations Java 5. Ce mécanisme permet d'alléger le fichier XML de configuration de Spring et de spécifier le comportement transactionnel aussi bien au niveau du contrat du composant que de son implémentation. Il est toutefois préférable de les définir au niveau du contrat du composant.

Spring permet de définir un comportement général pour une interface. Ses différentes méthodes héritent de ce comportement par défaut mais peuvent le surcharger au cas par cas. Ce mécanisme est similaire entre l'interface et l'implémentation. Cette dernière peut surcharger les comportements au niveau de la classe et des méthodes.

Les types de comportements sont ajoutés dans les services métier (interface ou implémentation) grâce à l'annotation `Transactional`. Le tableau 12.11 récapitule les différentes propriétés possibles pour cette annotation.

Tableau 12.11 Propriétés de l'annotation *Transactional*

Propriété	Type	Description
`Propagation`	`enum:Propagation`	Spécifie le type de propagation de la transaction. La valeur par défaut est `PROPAGATION_REQUIRED`.
`Isolation`	`enum:Isolation`	Spécifie le niveau d'isolation de la transaction. La valeur par défaut est `ISOLATION_DEFAULT`.
`ReadOnly`	`boolean`	Spécifie si la transaction est en lecture seule. La valeur par défaut est `false`.
`RollbackFor`	Tableau d'objets de type `Class`	Spécifie la liste des exceptions qui causeront une annulation de la transaction.
`RollbackForClassname`	Tableau d'objets de type `String`	Spécifie la liste des exceptions qui ne causeront pas d'annulation de la transaction.
`NoRollbackFor`	Tableau d'objets de type `Class`	Spécifie la liste des exceptions qui ne causeront pas d'annulation de la transaction.
`NoRollbackForClassname`	Tableau d'objets de type `String`	Spécifie la liste des exceptions qui ne causeront pas d'annulation de la transaction.

Le code suivant donne un exemple de mise en œuvre de l'annotation `Transactional` sur l'interface `TodoListsManager` :

```
@Transactional
public interface TodoListsManager {

    void createTodoList(TodoList todoList);

    @Transactional(readOnly=true)
    TodoList findTodoList(String listId);

    TodoList unsecuredFindTodoList(String listId);
    void updateTodoList(TodoList todoList);
    void deleteTodoList(String listId);
    void addTodoListUser(String listId, String login);
    void deleteTodoListUser(String listId, String login);
    Document backupTodoList(TodoList todoList);
    void restoreTodoList(String restoreChoice,
           String listId, InputStream todoListContent)
           throws JDOMException, IOException;
}
```

Les annotations permettent de spécifier le comportement transactionnel, mais il faut toujours utiliser des proxy pour les appliquer. La configuration des annotations se réalise au niveau du proxy ou de l'intercepteur POA de gestion des transactions grâce à la propriété `transactionAttributeSource`. Spring fournit une implémentation spécifique de l'interface `TransactionAttributeSource` pour les annotations, `AnnotationTransactionAttributeSource`.

L'extrait de configuration suivant montre comment le réaliser avec la classe `ProxyFactoryBean` :

```
<!-- Bean transactionnel cible -->
<bean id="todoListsManagerTarget"
      class="tudu.service.impl.TodoListsManagerImpl">
    <property name="todoListDAO">
        <ref bean="todoListDAO"/>
    </property>
    <property name="todoDAO">
        <ref bean="todoDAO"/>
    <property name="userManager">
        <ref bean="userManager"/>
    </property>
</bean>

<!-- proxy transactionnel sur la cible -->
<bean id="todoListsManager" parent="transactionProxy">
    <property name="transactionManager">
        <ref bean="transactionManager"/>
    </property>
    <property name="target">
        <ref bean="todoListsManagerTarget"/>
    </property>
  <property name="transactionAttributeSource">
    <bean class="org.springframework.transaction.
              annotation.AnnotationTransactionAttributeSource"/>
  </property>
</bean>
```

Approches personnalisées

Les approches décrites précédemment peuvent ne pas convenir complètement à une application en raison de leur intégration à d'autres composants ou frameworks utilisés dans l'architecture.

D'une manière générale, le support standard des transactions de Spring suffit largement. Cependant, il peut se révéler utile de combiner plusieurs technologies pour en tirer le meilleur parti, notamment dans les cas suivants :

- Le besoin d'uniformiser la démarcation transactionnelle conduit à utiliser une API générique. Spring fournit ce type d'API, mais cette dernière peut être utilisée sans pour autant recourir à l'injection de dépendances implémentée dans Spring.

- Le souci de modularité des composants amène à vouloir externaliser les problématiques techniques induites par les transactions. L'utilisation de technologies d'interception des traitements telles que la POA permet d'atteindre ce but. Le choix du type de tisseur POA peut s'effectuer en fonction des besoins de l'application et ne pas être celui de Spring.
- La spécification de comportements transactionnels plus fins pour les composants transactionnels peut être réalisée à l'assemblage ou au déploiement de manière déclarative afin d'offrir encore plus de flexibilité. La combinaison de technologies telles que les EJB ou Spring permet d'offrir cette fonctionnalité.

Il est donc possible de combiner Spring avec d'autres technologies ou frameworks afin de répondre au plus près aux besoins de l'application, notamment les technologies EJB et POA.

Combiner Spring et EJB

Dans la communauté J2EE, il est parfois de bon ton de mettre en concurrence la technologie EJB et le framework Spring. Les auteurs de cet ouvrage estiment pour leur part que les deux peuvent être utilisés de manière complémentaire, car ils ne couvrent pas complètement les mêmes domaines.

Comme un contexte d'application de Spring peut être embarqué dans un EJB Session, il est possible d'obtenir un niveau de granularité plus fin dans celui-ci. Cela apporte en outre des avantages en terme de gestion des transactions.

Si une CMT (Container Managed Transaction) est utilisée, les transactions des EJB seront gérées par le conteneur. Un contexte d'application de Spring peut être embarqué dans un EJB afin de gérer plus finement les comportements transactionnels des différents composants utilisés par l'EJB. Comme, dans ce cas, JTA est utilisé, le gestionnaire de transactions `JtaTransactionManager` de Spring doit être utilisé.

Si une BMT (Bean Managed Transaction) est utilisée, les transactions peuvent être gérées de la manière souhaitée dans Spring, qui les contrôle complètement.

Combiner Spring et AspectJ

Les concepteurs de l'application peuvent ne pas vouloir utiliser le tisseur POA de Spring et choisir un tisseur statique tel qu'AspectJ. Il est possible de développer un aspect AspectJ utilisant les API de gestion transactionnelle de Spring. Il est à noter qu'AspectJ fournit non pas un mécanisme de gestion transactionnelle, mais un cadre propice à sa mise en place.

Spring offre une intégration avec AspectJ permettant de réaliser de l'injection de dépendances de composants tiers sur un aspect de type singleton. Dans notre cas, le gestionnaire de transactions ainsi que les comportements transactionnels désirés pour les composants tissés peuvent être injectés dans l'aspect. Le code advice de l'aspect a dès lors la responsabilité d'utiliser les API de Spring et de définir les coupes sur les composants transactionnels.

Comme Spring a en charge la gestion du contexte transactionnel, l'aspect peut être de type singleton, ce framework le stockant dans un `ThreadLocal`.

En résumé

Spring offre un mécanisme de gestion des transactions particulièrement flexible, qui permet de répondre au mieux aux exigences des applications. Ce framework fournit en outre un mécanisme de gestion des transactions déclaratif indépendant des conteneurs J2EE, aussi bien pour les transactions locales que globales.

De ce fait, Spring n'impose pas l'utilisation du service transactionnel du serveur d'applications, lequel s'appuie sur les transactions globales, sans l'interdire pour autant. Il est donc possible de l'utiliser à bon escient.

Spring permet d'externaliser la gestion des transactions des composants. Ces derniers n'ont de la sorte pas conscience de la stratégie transactionnelle qui va être utilisée. Celle-ci est réalisée lors de l'assemblage des composants dans les fichiers de configuration. Les composants sont de la sorte de plus en plus découplés des technologies et frameworks sous-jacents. Utiliser directement leurs API constituerait une bien mauvaise pratique puisque les composants seraient alors fortement liés à la technologie et donc très difficiles à faire évoluer.

Le tableau 12.12 récapitule les avantages et inconvénients de Spring et des EJB pour la gestion des transactions.

Tableau 12.12. Avantages et inconvénients de Spring et des EJB pour la gestion transactionnelle

Technologie	Avantage	Inconvénient
EJB	– Gestion des transactions au niveau des composants – Choix du niveau d'intrusivité de la mise en œuvre (par programmation ou déclarative)	– Serveur d'applications J2EE avec un conteneur d'EJB nécessaire – Utilisation des transactions globales obligatoire dans le cas d'une gestion par le conteneur – Solution peu flexible
Spring	– Gestion des transactions au niveau des composants – Choix du niveau d'intrusivité de la mise en œuvre (par programmation ou déclarative) – Solution très flexible quant aux types de transactions, à leur configuration et à la gestion des exceptions – Utilisation possible en dehors d'un serveur d'applications	– Complexité déportée au niveau du fichier de configuration du conteneur léger – Notions de POA souhaitables

Tudu Lists : gestions des transactions

Pour notre étude de cas Tudu Lists, nous avons choisi de mettre en œuvre l'approche au cas par cas fondée sur la classe `TransactionProxyFactoryBean` précédemment décrite. Cette utilisation est adaptée à notre application puisque nous n'avons que quatre composants services métier auxquels s'applique un comportement transactionnel.

Comme l'application utilise Hibernate pour interagir avec la base de données, nous avons mis en œuvre la classe `HibernateTransactionManager` du package `org.springframework.orm.hibernate3`, implémentation de l'interface `PlatformTransactionManager` pour Hibernate 3.0.

Cette classe s'appuie sur l'instance de la `SessionFactory` configurée, comme l'illustre le code suivant, tiré du fichier **applicationContext-hibernate.xml** situé dans le répertoire **WEB-INF :**

```
<bean id="sessionFactory" class="org.springframework.orm
                                .hibernate3.LocalSessionFactoryBean">
    (...)
</bean>

<bean id="transactionManager" class="org.springframework.orm
                            .hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
```

La configuration transactionnelle utilise cette entité afin de spécifier par déclaration, dans le fichier **applicationContext.xml** du répertoire **WEB-INF,** les comportements transactionnels des composants.

Ces comportements, qui doivent être spécifiés pour tous les composants service métier de l'étude de cas, sont récapitulés au tableau 12.13.

Tableau 12.13. Composants services métier de l'étude de cas

Composant	Comportement transactionnel
`userManager`	– Méthode `create` : `PROPAGATION_REQUIRED` – Méthode `update` : `PROPAGATION_REQUIRED` – Méthode `delete` : `PROPAGATION_REQUIRED` – Autres méthodes : `PROPAGATION_REQUIRED` et `readOnly`
`todoListsManager`	– Méthode `create` : `PROPAGATION_REQUIRED` – Méthode `update` : `PROPAGATION_REQUIRED` – Méthode `delete` : `PROPAGATION_REQUIRED` – Méthode `add` : `PROPAGATION_REQUIRED` – Méthode `restore` : `PROPAGATION_REQUIRED` – Autres méthodes : `PROPAGATION_REQUIRED` et `readOnly`
`todosManager`	– Méthode `create` : `PROPAGATION_REQUIRED` – Méthode `update` : `PROPAGATION_REQUIRED` – Méthode `delete` : `PROPAGATION_REQUIRED` – Méthode `completeTodo` : `PROPAGATION_REQUIRED` – Méthode `reopenTodo` : `PROPAGATION_REQUIRED` – Autres méthodes : `PROPAGATION_REQUIRED` et `readOnly`
`configurationManager`	– Méthode `update` : `PROPAGATION_REQUIRED` – Autres méthodes : `PROPAGATION_REQUIRED` et `readOnly`

Le code suivant illustre la configuration du comportement transactionnel du composant ayant pour identifiant `userManager` dans le fichier **applicationContext.xml** précédemment cité :

```
<bean id="userManagerTarget"
      class="tudu.service.impl.UserManagerImpl">
    (...)
</bean>

<bean id="userManager" class="org.springframework.transaction
                        .interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager" ref="transactionManager"/>
    <property name="target" ref="userManagerTarget"/>
    <property name="transactionAttributes">
        <props>
            <prop key="create*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="delete*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

Conclusion

La gestion des transactions est une des problématiques les plus importantes et les plus complexes à mettre en œuvre dans une application. Elle peut impliquer plusieurs ressources transactionnelles fondées sur des technologies différentes, avoir à prendre en compte des accès concourants et savoir réagir aux erreurs ou aux pannes afin de garantir la cohérence des données. La mise en place de ces mécanismes au sein des applications est indispensable afin de garantir leur robustesse.

Les types de transactions utilisés doivent être adaptés aux besoins de l'application. Par exemple, l'utilisation de transactions globales peut amener une complexité inutile si une seule ressource transactionnelle est utilisée. Des frameworks tels que Spring fournissent désormais des mécanismes permettant de gérer les transactions locales de manière déclarative.

La mise en place des transactions dans les applications ne doit pas être négligée, car il s'agit d'une problématique de conception à part entière. Les différents concepts propres aux transactions décrits dans ce chapitre doivent être appliqués aux bons composants et les polluer le moins possible avec du code technique. Cette mise en place ne doit pas non plus être réalisée au détriment de la modularité ni de la flexibilité de l'architecture applicative.

L'utilisation de mécanismes déclaratifs pour spécifier les comportements transactionnels doit être privilégiée, de même que le choix de frameworks ou de technologies offrant cette fonctionnalité.

Partie IV

Technologies d'intégration

Cette partie décrit les différentes technologies permettant d'intégrer des applications Java/J2EE à des systèmes d'information d'entreprise, systèmes fondés sur une multitude de mécanismes de communication et de technologies hétérogènes.

Les technologies d'intégration de Java/J2EE comportent deux grandes familles. La première utilise les spécifications J2EE JMS et JCA afin de communiquer de manière synchrone ou asynchrone avec les systèmes d'information d'entreprise par le biais de middlewares. La seconde s'appuie sur les technologies XML, par l'intermédiaire de leurs supports dans Java/J2EE, afin d'échanger des informations contenues dans des messages normalisés. Dans ce dernier cas, les systèmes accédés doivent nécessairement exposer leurs services selon des technologies XML, tels les services Web.

Le chapitre 13 traite des mécanismes des technologies JMS, relative aux middlewares orientés message, et JCA, relative aux communications avec les systèmes d'information d'entreprise. Il décrit ensuite la façon dont Spring facilite l'utilisation de ces technologies dans les applications Java/J2EE

Le chapitre 14 se penche sur la façon d'utiliser XML afin d'intégrer des systèmes d'information hétérogènes. Nous détaillons les différents supports du framework Spring permettant d'utiliser les technologies XML de manière optimale dans les applications Java/J2EE.

Le chapitre 15 aborde les problématiques liées à la sécurité dans les applications Java/J2EE ainsi que la façon dont Acegi Security, un framework de sécurité complet et intégré utilisant Spring, permet de les résoudre de façon optimale.

13

Technologies d'intégration Java

Les applications Java/J2EE doivent s'intégrer dans les systèmes d'information des entreprises, communément désignés par le terme EIS et correspondant aux différentes applications et infrastructures de stockage des données. L'objectif est de pouvoir réutiliser des services applicatifs existants et de minimiser les duplications de données dans différents systèmes d'information.

Cela passe par l'interaction entre des applications pouvant être séparées physiquement au sein de l'entreprise et utilisant des mécanismes ou des technologies hétérogènes. Cette interaction peut vite devenir complexe, puisqu'il n'est pas toujours possible de les réécrire ou de les modifier. Or cette réécriture n'est pas forcément la meilleure solution pour les applications répondant aux besoins et fonctionnant correctement. Dans ce cas, l'interaction avec elles est la solution la plus appropriée.

Cette interaction peut s'insérer dans différents types de traitements et mettre en œuvre des mécanismes de communication complexes, synchrones ou asynchrones. Ce chapitre se penche sur les technologies et mécanismes fournis par J2EE afin d'intégrer des applications dans des systèmes d'information d'entreprise par le biais des spécifications JMS (Java Messaging Service) et JCA (Java Connector Architecture). La figure 13.1 schématise ces échanges.

Nous verrons que Spring fournit des supports pour ces deux technologies afin d'alléger leur mise en œuvre et leur intégration au sein des applications Java/J2EE.

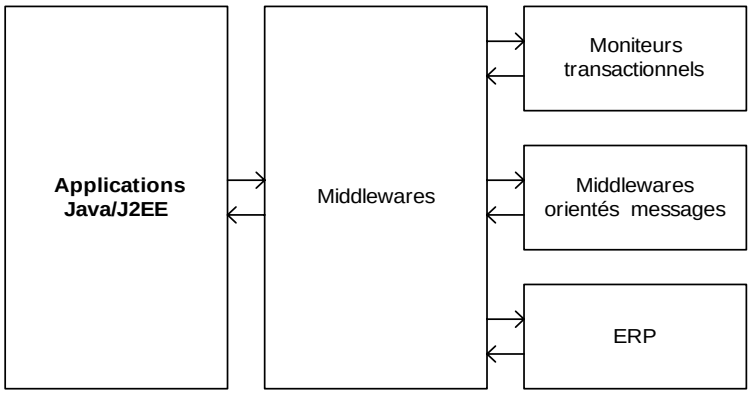


Figure 13.1
Interactions entre les applications Java/J2EE et les applications d'entreprise

Les sections qui suivent détaillent les différentes technologies permettant aux applications Java/J2EE de s'intégrer aux applications d'entreprise, ainsi que les supports de Spring simplifiant leur utilisation. Nous commençons par présenter la technologie JMS, afin de mieux appréhender son support par Spring. Nous utilisons ensuite la même approche pour la technologie JCA.

La spécification JMS (Java Messaging Service)

La spécification JMS vise à résoudre les préoccupations générales des messageries applicatives avec Java.

JMS adresse la problématique générale des MOM (Message-Oriented Middleware), ou middlewares orientés messages, en Java. Ces outils permettent en effet de faire communiquer des applications par l'intermédiaire de messages applicatifs contenant diverses informations applicatives ou de routage réseau.

Ces systèmes garantissent la distribution des messages aux applications tout en fournissant des fonctionnalités de tolérance aux pannes, d'équilibrage de charge, d'évolutivité et de support transactionnel. Ils utilisent à cet effet des canaux de communication, désignés par le terme *destination*, qui peuvent être utilisés afin de mettre en œuvre des mécanismes de communication asynchrones.

Les fournisseurs JMS

Chaque fabricant JMS propose une implémentation de l'API JMS cliente permettant d'interagir avec le serveur JMS. Le fabricant JMS offre également un module serveur de gestion de messages, qui implémente le routage et la distribution des messages. Ces deux entités sont collectivement désignées par le terme fournisseur JMS. Quelle que soit l'architecture utilisée par un fournisseur JMS, les parties logiques d'un système JMS sont identiques. Un fournisseur JMS correspond donc à un middleware ayant la responsabilité de recevoir et de distribuer les messages applicatifs. Il implémente à cet effet des mécanismes complexes, qui garantissent l'envoi et la réception de ces messages.

La spécification JMS adressant la messagerie applicative fournit un cadre générique pour envoyer et recevoir des messages de manière synchrone et asynchrone. Elle fournit de surcroît un niveau d'abstraction normalisé afin d'interagir avec différents systèmes de messagerie applicative, la plupart d'entre eux supportant désormais cette spécification. On désigne les systèmes de messagerie applicative compatibles JMS par le terme *fournisseurs JMS*.

JMS distingue deux domaines de messagerie. Le premier domaine, appelé file, ou *queue*, correspond à une distribution point-à-point. Un message envoyé sur un domaine de ce type est distribué une seule fois et à un seul observateur enregistré, comme l'illustre la figure 13.2.

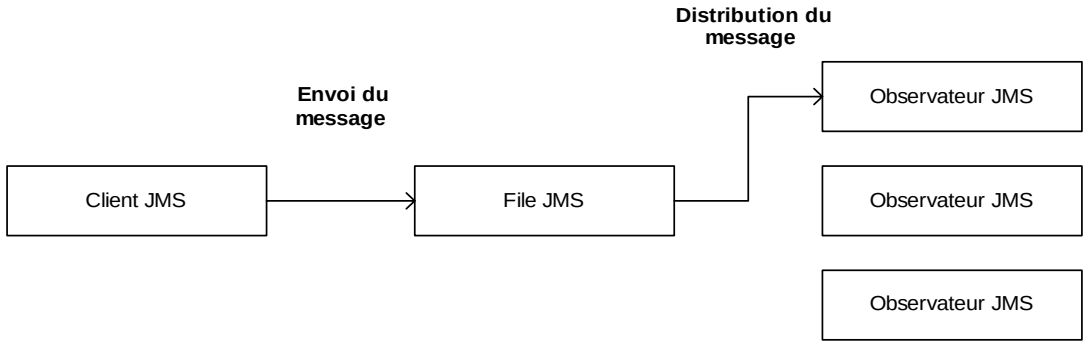


Figure 13.2
Mécanisme de distribution des messages pour le domaine file

Le second domaine, appelé sujet, ou *topic*, fonctionne sur le principe des listes de diffusion. Tous les observateurs enregistrés sur le domaine reçoivent le message envoyé, comme l'illustre la figure 13.3.

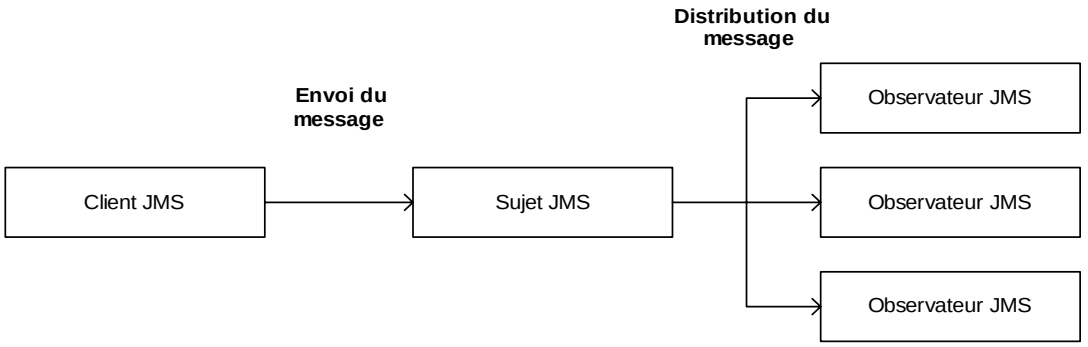


Figure 13.3
Mécanisme de distribution des messages pour le domaine sujet

Notons qu'une file ou un sujet est communément désignée dans la technologie JMS par le terme *destination.* Ces entités sont représentées par les interfaces `Queue` et `Topic`, héritant toutes deux de l'interface `Destination` du package `javax.jms`.

La spécification JMS comporte deux versions majeures. La version 1.0.2, la plus ancienne, dissocie dans ses API les deux domaines de messagerie. Elle comporte toutefois des limitations, notamment pour la gestion transactionnelle des messages. La version 1.1 adresse ces problèmes en uniformisant et homogénéisant les différentes API. Dans la suite du chapitre, nous nous fondons sur cette version 1.1, qui est couramment utilisée dans les applications d'entreprise Java/J2EE.

Interaction avec le fournisseur JMS

L'interaction avec le fournisseur JMS se réalise en plusieurs étapes :

1. Création de la fabrique de connexions.
2. Récupération d'une connexion à partir de la fabrique précédente. Il est possible de spécifier des propriétés pour la connexion, telles que l'identifiant du client.
3. Création d'une session à partir de la connexion. Au moment de cette création, il est possible de définir des propriétés transactionnelles, ainsi que le type d'acquittement d'envoi des messages.

La fabrique de connexion JMS est normalisée par l'intermédiaire de l'interface `ConnectionFactory` du package `javax.jms`. Son unique fonction est de créer des connexions pour un fournisseur JMS, comme le montre son code ci-dessous :

```
public interface ConnectionFactory {
    Connection createConnection();
    Connection createConnection(String userName, String password);
}
```

La connexion JMS correspond à la connexion physique avec le fournisseur JMS. Cette entité est normalisée par l'intermédiaire de l'interface `Connection` du package `javax.jms`. Sa création nécessite une authentification de la part de l'utilisateur. Elle offre la possibilité de créer une session d'utilisation permettant d'envoyer et de recevoir des messages, ainsi que de créer différents types de messages.

Il est possible de positionner un identifiant pour la connexion par l'intermédiaire de sa méthode `setClientID`. Le code suivant donne la définition de cette entité :

```
public interface Connection {
    void close();
    (...)
    Session createSession(boolean transacted, int acknowledgeMode);
    String getClientID();
    ExceptionListener getExceptionListener();
```

```
    ConnectionMetaData getMetaData();
    void setClientID(String clientID);
    void setExceptionListener(ExceptionListener listener);
    void start();
    void stop();
}
```

Notons la présence de l'interface `ExceptionListener` et de la méthode `setExceptionListener` de l'interface `Connection`, qui permettent d'enregistrer des observateurs et de récupérer les exceptions survenant lors de l'utilisation de la connexion.

Lorsqu'une connexion est créée, elle se retrouve en mode non actif et ne peut donc recevoir de messages. Il est par contre possible d'envoyer des messages par l'intermédiaire de l'entité `MessageProducer`, que nous décrivons plus loin dans ce chapitre.

Les méthodes `start` et `stop` de l'interface précédente permettent de changer ce mode. Une fois la méthode `start` invoquée, l'application peut utiliser l'entité `MessageConsumer`, que nous décrivons également plus loin, afin de recevoir des messages. La méthode `close` permet quant à elle de fermer la connexion avec le fournisseur JMS.

Notons qu'une connexion JMS est *thread safe* et que plusieurs fils d'exécution peuvent donc utiliser la même connexion simultanément, ce qui n'est pas le cas de la session décrite par la suite.

La session JMS permet d'interagir directement avec le fournisseur JMS afin d'envoyer, de recevoir et de créer des messages. Cette entité est normalisée par l'interface `Session` du package `javax.jms`. Au moment de sa création, l'utilisateur peut spécifier si elle doit être transactionnelle ainsi que le type d'acquittement des messages. Le code suivant donne la définition de cette entité :

```
public interface Session {
    (...)
    //Création de consommateur de messages
    MessageConsumer createConsumer(Destination destination);
    MessageConsumer createConsumer(Destination destination,
                                        String messageSelector);
    MessageConsumer createConsumer(Destination destination,
                        String messageSelector, boolean NoLocal);

    //Gestion des souscriptions durables
    TopicSubscriber createDurableSubscriber(
                                      Topic topic, String name);
    TopicSubscriber createDurableSubscriber(Topic topic,
           String name, String messageSelector, boolean noLocal);
    void unsubscribe(String name);

    //Création de producteur de messages
    MessageProducer createProducer(Destination destination);

    //Création de destinations
    Topic createTopic(String topicName);
```

```
    Queue createQueue(String queueName);
    TemporaryTopic createTemporaryTopic();
    TemporaryQueue createTemporaryQueue();

    //Création de messages
    TextMessage createTextMessage();
    TextMessage createTextMessage(String text);
    Message createMessage();
    ObjectMessage createObjectMessage();
    ObjectMessage createObjectMessage(Serializable object);
    BytesMessage createBytesMessage();
    MapMessage createMapMessage();
    StreamMessage createStreamMessage();

    //Propriétés de la session
    int getAcknowledgeMode();
    boolean getTransacted();

    //Observateurs enregistrés
    MessageListener getMessageListener();
    void setMessageListener(MessageListener listener);

    //Gestion des transactions
    void commit();
    void recover();
    void rollback();

    //Gestion de la session
    void close();
    (...)
}
```

Dans la définition de l'interface précédente, nous remarquons plusieurs types de méthodes. Ces dernières correspondent aux fonctionnalités récapitulées au tableau 13.1.

Tableau 13.1. Fonctionnalités de la session JMS

Fonctionnalité	Description
Envoi de messages	Permet de créer les entités nécessaires à la réception de messages.
Réception de messages	Permet de créer les entités nécessaires à l'émission de messages.
Gestion des souscriptions durables	Fournit des méthodes afin de gérer les souscriptions durables à des sujets. Ces dernières permettent à un utilisateur de recevoir tous les messages JMS, y compris ceux publiés pendant une période où celui-ci est inactif.
Création de destinations	Permet de créer des destinations (file ou sujet) en dehors des outils d'administration du fournisseur.
Création de messages JMS	Offre plusieurs méthodes permettant de créer les différents types de messages supportés par la spécification. Leur nom suit la règle `create<TYPE>Message()`.
Propriétés de la session	Permet de récupérer les valeurs des propriétés `transacted` et `acknowledgeMode` correspondant respectivement aux propriétés transactionnelles et d'acquittement.

Tableau 13.1. Fonctionnalités de la session JMS *(suite)*

Fonctionnalité	Description
Enregistrement d'un observateur JMS	Permet l'enregistrement d'un observateur JMS afin de recevoir et de traiter les messages par l'intermédiaire de la méthode `setListener`.
Gestion des transactions	Offre deux méthodes afin de finaliser une transaction JMS : `commit` en cas de succès et `rollback` en cas d'annulation.
Gestion de la session	La méthode `close` permet de fermer la session JMS.

Le code suivant met en œuvre ces différentes entités afin d'interagir avec un fournisseur JMS :

```
ConnectionFactory connectionFactory=null;
Connection connection=null;
Session session=null;

try {
    connectionFactory=getConnectionFactory();
    connection=connectionFactory.createConnection();
    connection.start();

    boolean transacted=false;
    int acknowledgeMode=Session.AUTO_ACKNOWLEDGE;
    session=connection.createSession(transacted,acknowledgeMode);
} catch(Exception ex) {
    convertJmsException(ex);
} finally {
    closeSession(session);
    stopAndCloseConnection(connection);
}
```

Constituants d'un message JMS

Avant d'envoyer des informations au fournisseur JMS, il faut d'abord déterminer le type de message puis le créer. Un message JMS est structuré comme décrit au tableau 13.2.

Tableau 13.2. Constituants d'un message JMS

Constituant	Description
En-tête du message	Permet de spécifier des informations interprétables aussi bien par le client que par le fournisseur afin de définir le message et de l'acheminer. La plupart de ces en-têtes (`JMSDestination`, `JMSDeliveryMode`, `JMSExpiration`, `JMSPriority`, `JMSMessageID`, `JMSTimestamp`) sont positionnés automatiquement par les méthodes `send` ou `publish` de la session JMS. Seuls les en-têtes `JMSCorrelationID`, `JMSReplyTo` et `JMSType` peuvent être utilisés par l'application cliente.
Propriétés du message	Permet de spécifier des informations applicatives dans le message.
Corps du message	Contient les données spécifiques de l'application. Elles peuvent prendre différentes formes au sein de cette partie.

JMS définit plusieurs types de messages, comme indiqué au tableau 13.3.

Tableau 13.3. Types de messages JMS

Type	Description
`StreamMessage`	Permet de stocker séquentiellement des informations de type primitif dans le message. Cette interface étend l'interface `Message` afin de fournir des méthodes de lecture et d'écriture de données par type.
`MapMessage`	Permet de stocker les informations du message sous forme de table de hachage. Cette interface étend l'interface `Message` afin de fournir les méthodes permettant d'accéder aux différents éléments suivant leurs types.
`TextMessage`	Permet de stocker des informations de type texte, aussi bien texte simple que XML, dans un message JMS. Cette interface étend l'interface `Message` afin de fournir des méthodes `getText` et `setText` pour accéder au texte du message et le spécifier.
`ObjectMessage`	Permet de stocker un objet Java sérialisable dans un message JMS. Cette interface étend l'interface `Message` afin de fournir des méthodes `getObject` et `setObject` pour accéder à l'objet du message et le spécifier.
`BytesMessage`	Permet de stocker un tableau d'octets dans un message JMS. Cette interface étend l'interface `Message` afin de fournir des méthodes pour lire et écrire des octets.

La création des messages se réalise à partir d'une instance de la session courante, comme le montre le code suivant :

```
Session session = createSession(connection);
//Création d'un message de type texte
TextMessage txtMessage = session.createTextMessage();
message.setText("Le texte de mon message");
(...)
//Création d'un message de type Map
MapMessage mapMessage = session.createMapMessage();
mapMessage.setString("description","Description de mon message");
mapMessage.setInt("taille",26);
```

Il est également possible de positionner des en-têtes et des paramètres sur le message, comme dans le code suivant :

```
txtMessage.setJMSCorrelationID("monId");
txtMessage.setStringProperty("maPropriete","maValeur");
```

Envoi de messages

L'entité clé pour envoyer des messages avec JMS est l'interface `MessageProducer` du package `javax.jms`, dont le code est le suivant :

```
public interface MessageProducer {
    void close();
    int getDeliveryMode();
    Destination getDestination();
    boolean getDisableMessageID();
```

```
    boolean getDisableMessageTimestamp();
    int getPriority();
    long getTimeToLive();
    void send(Destination destination, Message message) ;
    void send(Destination destination, Message message,
                int deliveryMode, int priority, long timeToLive);
    void send(Message message) ;
    void send(Message message, int deliveryMode,
                                int priority, long timeToLive) ;
    void setDeliveryMode(int deliveryMode);
    void setDisableMessageID(boolean value);
    void setDisableMessageTimestamp(boolean value);
    void setPriority(int defaultPriority);
    void setTimeToLive(long timeToLive);
}
```

Une entité `MessageProducer` possède les propriétés décrites au tableau 13.4.

Tableau 13.4. Propriétés de l'entité *MessageProducer*

Propriété	Description
`destination`	Destination (file ou sujet) sur laquelle le message doit être envoyé.
`deliveryMode`	Mode de distribution du message. Les valeurs possibles sont `DeliveryMode.NON_PERSISTENT` et `DeliveryMode.PERSISTENT`. Le mode persistant garantit une distribution du message, même en cas de panne du fournisseur JMS, ce qui n'est pas le cas en mode non persistant.
`disableMessageID`	Désactivation de la génération d'identifiant de messages par le fournisseur JMS
`disableMessageTimestamp`	Désactivation du calcul de l'estampille temporelle par le client JMS
`priority`	Priorité du message
`timeToLive`	Date d'expiration du message

Toutes ces propriétés peuvent être spécifiées de manière globale au moment de la création du `MessageProducer`. Dans ce cas, les messages utilisent ces valeurs lors de leur expédition, comme dans le code suivant :

```
(...)
Session session = createSession(connection);
Destination destination=getDestination();
TextMessage message=session.createTextMessage();
message.setText("texte du message");
MessageProducer messageProducer
                      = session.createProducer(destination);
messageProducer.setDeliveryMode(Message.DEFAULT_DELIVERY_MODE);
messageProducer.setPriority(Message.DEFAULT_PRIORITY);
messageProducer.setTimeToLive(Message.DEFAULT_TIME_TO_LIVE);
messageProducer.send(message);
messageProducer.close();
```

Il est possible de définir des valeurs spécifiques pour un message lors de son envoi. Ces dernières remplacent les valeurs définies au niveau du `MessageProducer`, comme dans le code suivant :

```
(...)
Session session = createSession(connection);
Destination destination=getDestination();
TextMessage message=session.createTextMessage();
message.setText("texte du message");
MessageProducer messageProducer
                        = session.createProducer(destination);
messageProducer.send(message, Message.DEFAULT_DELIVERY_MODE,
                              Message.DEFAULT_PRIORITY,
                              Message.DEFAULT_TIME_TO_LIVE);
messageProducer.close();
```

Réception de messages

L'entité clé pour recevoir des messages avec JMS est l'interface `MessageProducer` du package `javax.jms`, dont le code est le suivant :

```
public interface MessageConsumer {
    void close();
    MessageListener getMessageListener();
    String getMessageSelector();
    Message receive();
    Message receive(long timeout);
    Message receiveNoWait();
    void setMessageListener(MessageListener listener);
}
```

Cette interface offre la possibilité de recevoir des messages JMS de manière synchrone, et donc la plupart du temps bloquante, par l'intermédiaire de ses différentes méthodes `receive`.

Ces méthodes permettent de se mettre en attente d'un message indéfiniment, durant un temps fini ou non bloquant. Dans ce dernier cas, la méthode renvoie `null` si aucun message n'est disponible au moment de son exécution. Le code suivant en donne un exemple d'utilisation :

```
(...)
Session session = createSession(connection);
Destination destination=getDestination();
MessageConsumer messageConsumer
                        = session.createConsumer(destination);
int timeout = 60;
Message message = messageConsumer.receive(timeout);
messageConsumer.close();
```

Lorsque l'entité `MessageConsumer` est utilisée conjointement avec l'interface `MessageListener`, elle permet de recevoir des messages JMS de manière asynchrone. Dans ce cas, lors de la réception d'un message, la méthode `onMessage` de l'observateur est appelée, avec le message en paramètre. Le code suivant décrit l'interface `MessageListener` du package `javax.jms` :

```
public interface MessageListener {
    void onMessage(Message message);
}
```

Voici un exemple de mise en œuvre de réception asynchrone de message avec JMS :

```
(...)
Session session = createSession(connection);
Destination destination=getDestination();
MessageConsumer messageConsumer
                    = session.createConsumer(destination);
MessageListener listener = new MyMessageListener();
MessageConsumer.setMessageListener(listener);
Thread.sleep(60);
messageConsumer.close();
```

Versions de JMS

JMS 1.0.2 fait la distinction entre les différents domaines de messagerie, entraînant de sérieuses limitations dans la gestion des transactions et l'uniformisation des API de la technologie.

La version 1.1 uniformise ces deux domaines mais reste compatible avec les API de la version précédente. De ce fait, la plupart des entités de la version 1.0.2 sont désormais des sous-classes des entités de la version 1.1.

Le tableau 13.5 récapitule les différentes entités de ces deux versions de JMS.

Tableau 13.5. Entités des versions 1.0.2 et 1.1 de JMS

Entité JMS 1.1	Entité JMS 1.0.2 (file)	Entité JMS 1.0.2 (sujet)
`ConnectionFactory`	`QueueConnectionFactory`	`TopicConnectionFactory`
`Connection`	`QueueConnection`	`TopicConnection`
`Session`	`QueueSession`	`TopicSession`
`MessageProducer`	`QueueSender`	`TopicPublisher`
`MessageConsumer`	`QueueReceiver`	`TopicSubscriber`
`Destination`	`Queue`	`Topic`

Support JMS de Spring

Le support JMS de Spring, facilite l'utilisation de cette technologie aussi bien pour son paramétrage que pour son utilisation.

Nous détaillons dans cette section la configuration des entités JMS, ainsi que la classe centrale du support et la façon dont le framework gère l'envoi et la réception de messages JMS.

Le support JMS de Spring concerne les versions 1.0.2 et 1.1, mais nous ne détaillons dans cet ouvrage que le support de cette dernière.

Configuration des entités JMS

La première chose à mettre en place afin d'utiliser les API de JMS est la fabrique de connexions. La plupart du temps, les fournisseurs JMS les rendent disponibles aux applications clientes par le biais de JNDI. Ces entités doivent être configurées préalablement par l'intermédiaire d'outils d'administration.

La classe `JndiObjectFactoryBean` peut être mise en œuvre afin de les utiliser, comme dans le code suivant :

```
<bean id="jmsConnectionFactory"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="myConnectionFactory"/>
    <property name="jndiEnvironment">
        (...)
    </property>
</bean>
```

Notons qu'il est indispensable de spécifier l'environnement JNDI associé au fournisseur JMS. Le support JMS laisse la possibilité de configurer la fabrique en tant que Bean.

La deuxième étape consiste à déterminer les différentes destinations que l'application utilise et la manière dont elle y accède. Le support JMS définit l'abstraction `DestinationResolver` dans le package `org.springframework.jms.support.destination` dans le but de récupérer une instance à partir d'un nom par l'intermédiaire de la méthode `resolveDestinationName` décrite ci-dessous :

```
public interface DestinationResolver {
    Destination resolveDestinationName(Session session,
                                       String destinationName,
                                       boolean pubSubDomain)
                                       throws JMSException;
}
```

Cette interface possède deux implémentations, localisées dans le même package que précédemment : `JndiDestinationResolver`, qui résout le nom en utilisant JNDI, et `DynamicDestinationResolver`, qui utilise les méthodes `createQueue` et `createTopic` de la session JMS afin de créer dynamiquement des files et des sujets JMS. Elles doivent être utilisées

lorsque les différentes entités du support sont configurées avec des noms de destination et non des instances.

ActiveMQ

ActiveMQ est un fournisseur JMS Open Source particulièrement léger et performant. Entièrement écrit en Java, il peut être utilisé en mode autonome ou embarqué dans une application ou des tests unitaires. Ce projet est devenu récemment un sous-projet du projet Geronimo d'Apache, correspondant à une implémentation complète de J2EE. Il est accessible à l'adresse *http://www.activemq.org/*.

Il est toujours possible d'utiliser l'entité `JndiObjectFactoryBean` afin de récupérer une instance de la destination, comme dans le code suivant pour le fournisseur JMS ActiveMQ :

```
<bean id="jmsQueue"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName"><value>queue</value></property>
    <property name="jndiEnvironment">
        <props>
            <prop key="java.naming.factory.initial">
                org.activemq.jndi.ActiveMQInitialContextFactory
            </prop>
            <prop key="java.naming.provider.url">
                tcp://localhost:61616
            </prop>
            <prop key="queue.queue">tudu.queue</prop>
        </props>
    </property>
  </bean>
```

Le template JMS

Le template JMS est la classe centrale du support JMS de Spring puisqu'elle facilite l'interaction entre le fournisseur JMS et l'application.

Il existe deux versions de cette entité, correspondant aux différentes versions de la spécification JMS, mais nous ne détaillons que celle relative à la version 1.1.

Ce template s'appuie sur une fabrique de connexions JMS et une destination configurées de la même manière que précédemment. Étant donné qu'il possède différentes propriétés de paramétrage, il est recommandé de le configurer dans Spring de la façon suivante :

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="jmsConnectionFactory"/>
    <property name="" ref="destination" ref="jmsQueue"/>
    (...)
</bean>
```

Le template peut être configuré en tant que singleton, puisqu'il se fonde directement sur une fabrique de connexions JMS, et être injecté ensuite dans les composants de l'application.

Le tableau 13.6 récapitule les différents paramètres de configuration du template JMS (classe `JmsTemplate`).

Tableau 13.6. Propriétés de la classe *JmsTemplate*

Propriété	Utilisation	Description
`destinationResolver`	Envoi et réception (par défaut `null`)	Correspond à l'entité utilisée afin de récupérer l'instance de la destination dont le nom a été configuré par l'intermédiaire de la propriété `defaultDestination`. Elle est de type `DestinationResolver`.
`sessionTransacted`	Envoi et réception (par défaut `false`)	Permet de déterminer si les sessions JMS créées sont transactionnelles.
`sessionAcknowledgeMode`	Envoi (par défaut `Session.AUTO_ACKNOWLEDGE`)	Correspond au mode d'acquittement des messages envoyés.
`defaultDestination`	Envoi et réception (par défaut `null`)	Correspond à la destination par défaut. Elle peut être renseignée aussi bien avec l'instance de la destination qu'avec son nom. Cette propriété est utilisée par les méthodes du template ne possédant pas d'information de destination en paramètre.
`messageConverter`	Envoi et réception (par défaut `null`)	Correspond à l'entité utilisée afin de construire un message à partir d'un objet et de récupérer un objet à partir d'un message. Elle est de type `MessageConverter` et est décrite à la section suivante.
`messageIdEnabled`	Envoi (par défaut `true`)	Détermine si la génération des identifiants des messages JMS est activée.
`messageTimestampEnabled`	Envoi (par défaut `true`)	Détermine si la génération des estampilles temporelles des messages JMS est activée.
`pubSubNoLocal`	Envoi et réception (par défaut `false`)	Est nécessaire pour ce template afin d'utiliser la création dynamique de destinations.
`receiveTimeout`	Réception (par défaut `-1`)	Correspond au temps d'attente de réception de messages. Si sa valeur est supérieure ou égale à 0, cette propriété est passée en paramètre de la méthode `receive` de la session JMS. Dans le cas contraire, la méthode `receive` est bloquée jusqu'à l'arrivée d'un message.
`explicitQosEnabled`	Envoi (par défaut `false`)	Permet d'activer l'utilisation des paramètres `deliveryMode`, `priority` et `timeToLive` lors de l'envoi de messages JMS. Si la valeur de cette propriété est `true`, les propriétés précédentes sont passées en paramètres de la méthode `send` de la session JMS.
`deliveryMode`	Envoi (par défaut `Message.DEFAULT_DELIVERY_MODE`)	Correspond au mode de distribution des messages envoyés.
`priority`	Envoi (par défaut `Message.DEFAULT_PRIORITY`)	Correspond à la priorité des messages envoyés.
`timeToLive`	Envoi (par défaut `Message.DEFAULT_TIME_TO_LIVE`)	Correspond à la configuration de la date d'expiration des messages envoyés.

Ces paramètres peuvent être spécifiés sur l'instance du template grâce aux fonctionnalités de Spring relatives à l'injection de dépendances.

Envoi de messages

La classe `JmsTemplate` facilite l'envoi de messages au fournisseur JMS en intégrant toute la manipulation des API JMS, tout en laissant la possibilité au développeur de spécifier les parties propres à son application.

Afin d'envoyer des messages JMS, le template permet de travailler directement sur des ressources JMS qu'il gère par le biais de méthodes de rappel, et ce aux niveaux à la fois de la session et du producteur. Il s'appuie pour cela sur les interfaces `SessionCallback` et `ProducerCallback`.

L'interface `SessionCallback` met à disposition la session grâce à la méthode `doInJms`, comme ci-dessous :

```
public interface SessionCallback {
    Object doInJms(Session session) throws JMSException;
}
```

L'interface `ProducerCallback` enrichit cette signature de méthode afin de rendre disponible le producteur de message, comme ci-dessous :

```
public interface ProducerCallback {
    Object doInJms(Session session,
                MessageProducer producer) throws JMSException;
}
```

Le template JMS utilise ces interfaces par l'intermédiaire de méthodes `execute` de la manière suivante :

```
Final Destination destination = getDestination();
JmsTemplate template = getJmsTemplate();
template.execute(new ProducerCallback() {
    public Object doInJms(Session session,
                MessageProducer producer) throws JMSException {
        TextMessage message = session.createTextMessage();
        message.setText("Le texte du message.");
        producer.send(destination,message);
    }
});
```

Création et envoi de messages

Le template JMS peut être paramétré avec une implémentation de l'interface `MessageCreator` du package `org.springframework.jms.core` afin de spécifier la façon de créer le message à envoyer. Le code suivant donne sa définition :

```
public interface MessageCreator {
    Message createMessage(Session session) throws JMSException;
}
```

Ce mécanisme peut être mis en œuvre avec toutes les méthodes `send` du template JMS possédant un paramètre de type `MessageCreator`, comme ci-dessous :

```
JmsTemplate template = getJmsTemplate();
template.send(new MessageCreator() {
    public Message createMessage(Session session)
                                    throws JMSException {
        TextMessage message = session.createTextMessage();
        message.setText("Le texte du message.");
        return message;
    }
}
```

Conversion et envoi de messages

Le support JMS de Spring fournit l'interface `MessageConverter` dans le package `org.springframework.jms.support.converter` afin de généraliser le mécanisme précédent à la réception (conversion d'un message en objet) et à l'envoi (conversion d'un objet en message) de messages.

Contrairement au créateur, un convertisseur de messages est global au template JMS. Son code est le suivant :

```
public interface MessageConverter {
    Message toMessage(Object object, Session session)
                throws JMSException, MessageConversionException;
    Object fromMessage(Message message)
                throws JMSException, MessageConversionException;
}
```

L'utilisateur doit spécifier dans une classe la façon de passer d'un message à un objet, et inversement. Le code suivant décrit son utilisation avec des messages de type `TextMessage` :

```
public MonConvertisseur implements MessageConverter {
    public Message toMessage(Object object, Session session)
                throws JMSException, MessageConversionException {
        TextMessage message = session.createTextMessage();
        if( object instanceof String ) {
            message.setText((String)object);
        } else {
            message.setText(object.toString());
        }
        return message;
    }

    public Object fromMessage(Message message)
                throws JMSException, MessageConversionException {
```

```
        if( message instanceof TextMessage ) {
            return ((TextMessage)message).getText();
        } else {
            throw new MessageConversionException(
                                "Type de message non supporté.");
        }
    }
```

Cette implémentation est rattachée au template précédent de la manière suivante :

```
<bean id="jmsMessageConverter" class="MonConvertisseur"/>

<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="messageConverter" ref="jmsMessageConverter"/>
    (...)
</bean>
```

Ce mécanisme peut être mis en œuvre avec toutes les méthodes `convertAndSend` du template JMS, comme ci-dessous :

```
JmsTemplate template = getJmsTemplate();
template.convertAndSend("Le texte du message.");
```

Notons que ce code utilise le convertisseur configuré précédemment sur le template JMS.

Postprocessing des messages

L'envoi de messages avec conversion offre la possibilité de réaliser des traitements sur les messages immédiatement avant qu'ils soient envoyés. Ce mécanisme se fonde sur l'interface `MessagePostProcessor` suivante du package `org.springframework.jms.core` :

```
public interface MessagePostProcessor {
    Message postProcessMessage(Message message)
                                    throws JMSException;
}
```

L'exemple suivant décrit une mise en œuvre possible de cette entité afin d'ajouter automatiquement un identifiant de corrélation à chaque message envoyé :

```
JmsTemplate template = getJmsTemplate();
template.convertAndSend("Le texte du message.",
           new MessagePostProcessor() {
    public Message postProcessMessage(Message message)
                                        throws JMSException {
        String correlationId = getCorrelationId();
        message.setJMSCorrelationID(correlationId);
    }
});
```

Réception de messages

Le support JMS de Spring permet de recevoir des messages JMS aussi bien de manière synchrone qu'asynchrone. La mise en œuvre de ces fonctionnalités se fonde sur des entités différentes, dont certaines sont apparues avec la version 2.0 du framework.

La réception synchrone implique une action de l'application cliente JMS afin de récupérer les messages, action pouvant être bloquante. Cette fonctionnalité utilise la classe centrale du support JMS, à savoir la classe `JmsTemplate`.

La réception asynchrone met en œuvre des observateurs JMS. Le support JMS fournit un cadre, dénommé conteneur de gestion des observateurs, afin d'enregistrer ces observateurs auprès des ressources JMS appropriées.

Réception synchrone de messages

Puisque le support JMS offre la possibilité de travailler sur une session gérée par le template par l'intermédiaire de l'interface `SessionCallback`, il est possible de créer une instance de `MessageConsumer` afin de recevoir des messages.

Notons que Spring ne définit pas d'interface `ConsumerCallback`, à la manière de l'interface `ProducerCallback`. La mise en œuvre de l'interface `SessionCallback` n'est pas recommandée dans ce cas, car le développeur a la responsabilité de gérer l'instance de consommation des messages. Le template JMS fournit des méthodes afin d'encapsuler toute cette logique technique.

Comme pour l'envoi de messages, la réception se décompose en deux parties, dont la première récupère directement le message reçu. Elle s'appuie pour cela sur les méthodes `receive` et `receiveSelected`. Cette dernière offre la possibilité d'utiliser un sélecteur de message afin de cibler les messages désirés. Le code suivant en donne un exemple d'utilisation :

```
JmsTemplate template = getJmsTemplate();
Message message = template.receive();
```

Notons que la méthode `receive` de cet exemple utilise tous les paramétrages du template réalisés précédemment.

La seconde méthode permet d'utiliser le mécanisme de conversion abordé lors de l'envoi de message, toutes les méthodes nommées `receiveAndConvert` et `receiveSelectedAndConvert` mettant en œuvre ce mécanisme.

Ce dernier type de méthode offre la possibilité d'utiliser un sélecteur de message afin de cibler les messages désirés. Le code suivant en donne un exemple d'utilisation :

```
JmsTemplate template = getJmsTemplate();
String txtMessage = (String)template.receiveAndConvert();
```

Notons que la méthode `receive` de cet exemple utilise également tous les paramétrages réalisés précédemment sur le template ainsi que sur le convertisseur `MonConvertisseur`.

Réception asynchrone de messages

À partir de la version 2.0 de Spring, le support JMS intègre un cadre robuste afin de mettre en œuvre des mécanismes de réception asynchrones de JMS. Spring implémente pour cela deux approches. La première utilise l'entité `MessageConsumer` et sa méthode `setListener`, et la seconde l'entité `ServerSession`.

Ces deux approches ont en commun les propriétés récapitulées au tableau 13.7.

Tableau 13.7. Propriétés communes des conteneurs JMS de Spring

Propriété	Description
`destinationResolver`	Correspond à l'entité utilisée afin de récupérer l'instance de la destination dont le nom a été configuré par l'intermédiaire de la propriété `defaultDestination`. Elle est de type `DestinationResolver`.
`connectionFactory`	Correspond à la fabrique de connexions à utiliser.
`sessionTransacted`	Permet de déterminer si les sessions JMS créées sont transactionnelles.
`sessionAcknowledgeMode`	Correspond au mode d'acquittement des messages envoyés.
`messageSelector`	Correspond à l'entité utilisée afin de filtrer les messages à recevoir.
`messageListener`	Correspond à l'instance de l'observateur JMS utilisé.
`exposeListenerSession`	Permet de spécifier si la session à fournir aux observateurs JMS de type `SessionAwareMessageListener` est celle utilisée pour la réception des messages.
`autoStartup`	Spécifie si la méthode `start` de la connexion JMS doit être appelée au chargement du conteneur. Si sa valeur est `false`, il est possible de le démarrer ultérieurement avec la méthode `start` du conteneur. L'arrêt de la réception des messages est réalisé avec la méthode `stop`.
`destination`	Correspond à la destination à utiliser. Elle peut être renseignée avec l'instance de la destination ou son nom.

Le premier conteneur JMS de Spring est implémenté par l'intermédiaire de la classe `SimpleMessageListenerContainer` du package `org.springframework.jms.listener`. Cette classe correspond à la forme la plus simple et offre une approche multithreadée.

Il est possible de paramétrer le nombre de sessions utilisées pour la réception des messages par l'intermédiaire de la propriété `concurrentConsumers`. Ce conteneur ne permet pas de modifier dynamiquement sa configuration au cours de l'exécution. Le code suivant donne un exemple de sa mise en œuvre :

```
<bean id="connectionFactory"
      class="org.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

```
<bean id="asynchTuduJmsListener"
      class="tudu.jms.AsynchTuduJmsListener"/>

<bean id="jmsContainer" class="org.springframework.jms
                        .listener.SimpleMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="destinationName" value="tudu.queue"/>
    <property name="messageListener" ref="asynchTuduJmsListener"/>
</bean>
```

Le second conteneur, implémenté par l'intermédiaire de la classe `ServerSessionMessageListenerContainer` du même package, est beaucoup plus évolué. Il se fonde sur les API JMS `ServerSessionPool`, généralement mises en œuvre par les serveurs d'applications. Il permet de réaliser la réception de messages de manière multithreadée et se configure de la même manière que le précédent.

En résumé

Le support JMS de Spring réduit la complexité liée à l'utilisation de cette technologie. Il permet de configurer facilement les différentes entités de JMS dans le conteneur léger et de se fonder sur une entité principale gérant les interactions avec le fournisseur JMS.

Le support offre également, à partir de la version 2.0, une façon élégante et simple de mettre en œuvre des observateurs JMS afin d'utiliser les mécanismes asynchrones de communication.

La spécification JCA (Java Connector Architecture)

Cette section se penche sur les préoccupations générales des interactions avec les systèmes d'information d'entreprise ainsi que sur les concepts de la spécification JCA visant à les résoudre avec Java.

La spécification JCA définit les mécanismes permettant d'accéder de manière uniformisée à des systèmes d'information d'entreprise hétérogènes. Ces derniers peuvent utiliser diverses technologies, dont Java/J2EE, et correspondre, par exemple, à des moniteurs transactionnels pour mainframes (CICS, IMS) ou à des ERP (SAP, Peoplesoft, etc.).

La spécification JCA comprend deux versions :

- **Version 1.0.** Adresse la façon d'interagir avec des systèmes d'information d'entreprise par l'intermédiaire de requêtes tout en recourant aux services J2EE. Elle standardise une API cliente, appelée Common Client Interface, afin d'interagir avec ses systèmes d'une manière similaire à JDBC mais plus générique.

- **Version 1.5.** Enrichit la version précédente avec le support des connexions entrantes sur les connecteurs. Les systèmes d'information peuvent envoyer des messages au connecteur, lequel notifie les observateurs enregistrés. Cet ajout est fortement lié à la spécification JMS permettant de normaliser ses interactions avec les services J2EE.

L'objectif de la spécification est également de standardiser les interactions entre les connecteurs et un fournisseur de services. Nous faisons ici la distinction entre fournisseur de services et serveur d'applications, puisque les deux ne sont pas forcément équivalents. Un serveur d'applications correspond à un fournisseur de services, mais la réciproque n'est pas vérifiée.

Les mécanismes de JCA peuvent être utilisés afin de faire interagir des ressources Java/J2EE avec des services tels qu'un gestionnaire de transactions compatible JTA.

La spécification n'impose pas l'API cliente à utiliser, celle-ci étant déterminée par l'implémentation du connecteur. Ce dernier peut nécessiter l'utilisation de Common Client Interface, comme c'est le cas pour des connecteurs tels que ceux d'IBM, permettant d'accéder à des mainframes *via* CICS ou IMS, mais peut également permettre la configuration de fabriques de connexions ou de sessions, telles que celles de JDBC, JMS ou Hibernate.

La technologie JCA est de plus en plus utilisée dans les serveurs d'applications afin de configurer les différentes ressources qu'ils utilisent et de les faire interagir avec leurs services. Geronimo, par exemple, le serveur J2EE de la fondation Apache, configure les pools de connexions JDBC à partir de JCA et de son connecteur générique, TranQL.

La spécification JCA comporte deux parties distinctes :

- **Partie cliente.** Utilisable dans les applications Java/J2EE afin d'interagir avec les systèmes d'information d'entreprise.
- **Partie fournisseur de services.** Utilisable par les serveurs applications ou les frameworks afin de configurer l'environnement d'exécution de JCA aussi bien pour les communications sortantes qu'entrantes.

Gestion des communications sortantes

JCA offre un cadre afin de configurer la fabrique de connexions ou de sessions de différents frameworks ou technologies. Le connecteur a la responsabilité de travailler sur les connexions ou sessions physiques et fournit à l'application cliente des connexions logiques.

Ce mécanisme permet de réaliser des interceptions de traitements afin d'insérer des fonctionnalités transversales de manière transparente, telles que les transactions et la sécurité.

La figure 13.4 illustre les différentes interactions entre le connecteur et l'application ainsi que celles avec le fournisseur de services.

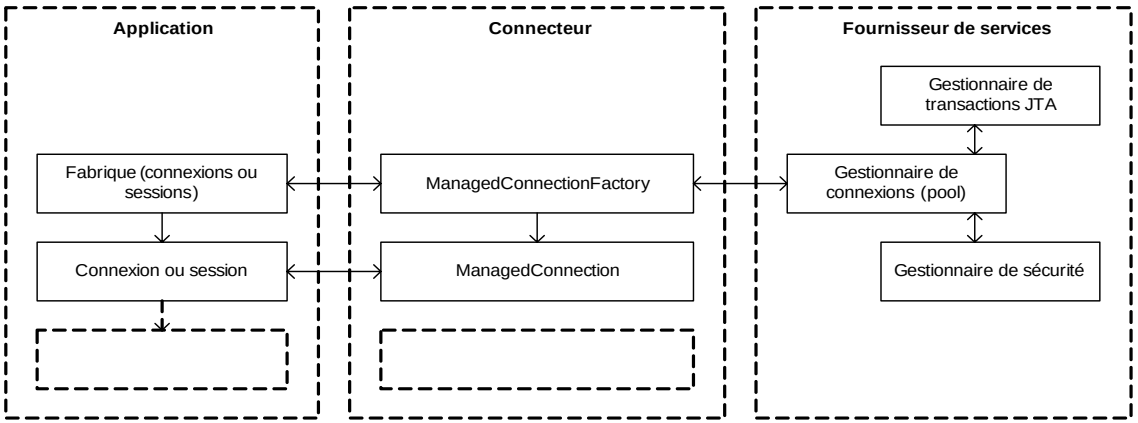


Figure 13.4

Interactions du connecteur avec l'application et le fournisseur de services

Sur cette figure, les différentes entités manipulées par l'application peuvent être aussi bien celles de l'API Common Client Interface de JCA que celles d'autres technologies ou frameworks, tels que JDBC, JMS ou Hibernate. De plus, le fournisseur de services peut aussi bien être un serveur d'applications que des composants autonomes.

Le connecteur peut fonctionner en mode autonome, sans utiliser de fournisseur de services. Dans ce cas, il utilise son gestionnaire de connexions interne mais ne peut plus alors utiliser les transactions globales.

L'implémentation de l'interface `ManagedConnectionFactory` doit dans ce cas être instanciée explicitement par l'application afin d'avoir accès à la fabrique, comme dans le code suivant :

```
ManagedConnectionFactory managedConnectionFactory
          = createAndConfigureManagedConnectionFactory();
Object connectionFactory
          = managedConnectionFactory.createConnectionFactory();
```

Notons l'existence de la méthode `createConnectionFactory`, possédant un paramètre de type `ConnectionManager`, qui permet de spécifier un gestionnaire de connexions pour le connecteur. Elle permet ainsi d'utiliser le connecteur avec un fournisseur de services tel que Jencks. Dans ce cas, l'utilisation des transactions globales est envisageable.

JCA est souvent utilisé pour configurer des fabriques de connexions ou de sessions afin de les faire participer à des transactions globales, la spécification normalisant cette interaction quelle que soit la technologie employée.

Jencks

Ce projet permet de configurer le gestionnaire de transactions JTA et le conteneur JCA du serveur d'applications Geronimo de manière autonome dans n'importe quelle application Java/J2EE, application n'ayant plus la nécessité d'utiliser un serveur d'applications J2EE complet. Il est dès lors possible d'utiliser les technologies JTA et JCA dans des applications Java autonomes ou dans des applications Web déployées dans Tomcat. Ce projet est en marge du support JCA de Spring et du fournisseur JMS ActiveMQ. Il est disponible à l'adresse *http://jencks.org/*.

L'API Common Client Interface permet aux applications Java/J2EE d'interagir avec des systèmes d'information d'entreprise selon les étapes suivantes :

1. Création de la fabrique de connexions.
2. Récupération d'une connexion à partir de la fabrique précédente. Elle est paramétrée par l'intermédiaire d'une implémentation de la classe `ConnectionSpec` relative au connecteur.
3. Création d'une interaction à partir de la connexion précédente.
4. Exécution de la requête, qui peut être paramétrée par le biais d'une implémentation de la classe `InteractionSpec` relative au connecteur. Les paramètres ainsi que les retours sont décrits par des implémentations de la classe `Record` et de ses dérivées.

Notons que les interfaces `ConnectionSpec` et `InteractionSpec` sont des classes marqueurs ne définissant aucune méthode.

La fabrique de connexions JCA est normalisée par l'intermédiaire de l'interface `ConnectionFactory` du package `javax.resource`. Son unique fonction est de créer des connexions pour un système d'information d'entreprise, comme le montre le code suivant de l'interface :

```
public interface ConnectionFactory {
    Connection getConnection();
    Connection getConnection(ConnectionSpec properties);
    ResourceAdapterMetaData getMetaData();
    RecordFactory getRecordFactory();
}
```

La connexion correspond à l'entité de communication avec le système. Sa création peut être paramétrée par l'intermédiaire d'une implémentation de `ConnectionSpec` du connecteur, lequel permet, de manière classique, de spécifier des paramètres d'authentification afin d'établir la connexion.

Le code suivant donne la définition de cette entité :

```
public interface Connection {
    void close();
    Interaction createInteraction();
    LocalTransaction getLocalTransaction();
    ConnectionMetaData getMetaData();
    ResultSetInfo getResultSetInfo();
}
```

Cette entité offre un support afin de démarquer les transactions locales en rendant accessible une implémentation de l'interface `LocalTransaction` pour le connecteur.

L'interaction permet d'exécuter une requête vers le système. Elle offre deux approches à cet effet, dont une seule est supportée par les connecteurs. La première consiste à passer les paramètres d'entrée sous forme de `Record` et de récupérer les résultats sous la même forme. La seconde attend en paramètres un premier `Record` contenant les paramètres d'entrée et un autre `Record` rempli suite à l'exécution avec les paramètres de retour.

Le code suivant donne la définition de cette entité :

```
public interface Interaction {
    void clearWarnings();
    void close();
    Record execute(InteractionSpec ispec, Record input);
    boolean execute(InteractionSpec ispec,
                          Record input, Record output);
    Connection getConnection();
    ResourceWarning getWarnings();
}
```

La mise en œuvre de ces différentes entités afin d'interagir avec un système d'information d'entreprise avec la première approche se déroule de la façon suivante :

```
Connection connection = null;
Interaction interaction = null;

try {
    ConnectionFactory connectionFactory = getConnectionFactory();

    ConnectionSpec connectionSpec = createConnectionSpec();
    connection = connectionFactory.getConnection(connectionSpec);

    interaction interaction = connection.createInteraction();
    InteractionSpec interactionSpec = createInteractionSpec();
    Record inputRecord = createInputRecord();
    Record outputRecord = interaction.execute(
                                    interactionSpec, inputRecord);
    (...)
} catch(Exception ex) {
    convertResourceException(ex);
} finally {
    closeInteraction(interaction);
    closeConnection(connection);
}
```

Gestion des communications entrantes

Comme nous l'avons vu précédemment, la version 1.5 de JCA enrichit la spécification avec le support des communications entrantes. Le système d'information d'entreprise peut rappeler le connecteur afin de lui envoyer des messages. Ce mécanisme est souvent

utilisé conjointement avec la spécification JMS et trouve toute son utilité avec les outils de messagerie asynchrone tels que les fournisseurs JMS.

La figure 13.5 illustre les différentes interactions entre le connecteur et l'application ainsi que celles avec le fournisseur de services.

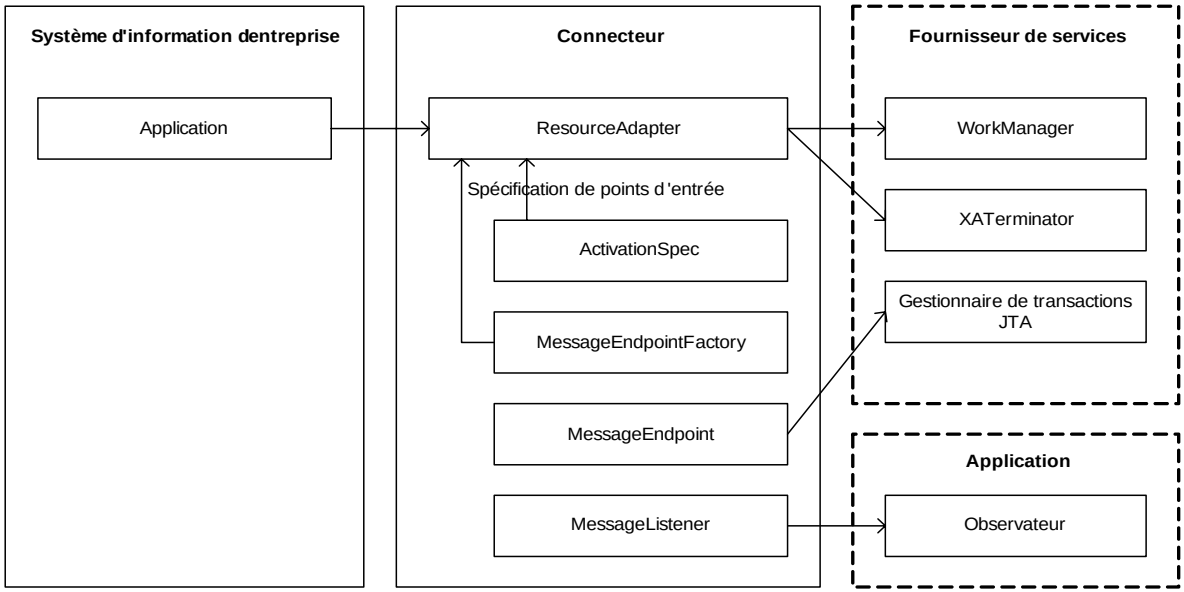


Figure 13.5

Interactions du connecteur avec l'application et le fournisseur de services

La spécification définit l'interface `ResourceAdapter` décrivant le connecteur et permettant d'enregistrer des points d'accès, matérialisés par des implémentations de l'interface `ActivationSpec`. Le connecteur doit avoir été préalablement démarré et initialisé avec des entités du fournisseur de services et arrêté à la fin de son utilisation.

Le code suivant fournit les différentes méthodes de l'interface `ResourceAdapter` :

```
public interface ResourceAdapter {
    (...
    //Enregistrement et désenregistrement de points d'entrée
    void endpointActivation(MessageEndpointFactory endpointFactory,
                                            ActivationSpec spec);
    void endpointDeactivation(
                          MessageEndpointFactory endpointFactory,
                          ActivationSpec spec);
    //Démarrage et arrêt du connecteur
    void start(BootstrapContext ctx);
    void stop();
}
```

Lors de l'activation d'un point d'accès, l'implémentation de l'interface `ResourceAdapter` utilise la fabrique `MessageEndpointFactory` afin d'instancier une unité de traitement de type `MessageEndpoint`. Cette entité met en œuvre les mécanismes transactionnels en se fondant sur le cycle de vie du traitement : avant la réception d'un message (`beforeDelivery`), après sa réception (`afterDelivery`) et au moment de la finalisation du traitement (`release`).

Nous constatons que cette partie de JCA offre des similitudes avec la réception asynchrone de messages de la spécification JMS, tout en fournissant un cadre plus générique. Dans le cas de l'utilisation de fournisseurs JMS, les `MessageEndpoint` peuvent être reliés à des observateurs JMS, lesquels seront notifiés lors de la réception de messages. Nous verrons par la suite comment le framework Jencks permet de mettre en œuvre cette fonctionnalité.

Support JCA de Spring

L'objectif du support JCA de Spring est de faciliter l'utilisation de l'API Common Client Interface, ainsi que la configuration des connecteurs en mode non managé, et en dehors des serveurs d'applications, et celle des communications entrantes asynchrones.

Dans le cas des communications sortantes, le support JCA reprend les concepts du support JDBC de Spring en l'adaptant à l'API Common Client Interface. Il intègre dans le framework la manipulation de ses entités tout en s'intégrant avec le support transactionnel du framework.

Le support JCA n'implémente pas actuellement les communications entrantes. Jencks offre cependant cette fonctionnalité en s'appuyant sur le framework Spring.

Communications sortantes

Le framework Spring intègre, à partir de sa version 1.2, un support complet de la partie relative aux connexions sortantes permettant d'envoyer des requêtes vers des systèmes d'information d'entreprise en utilisant l'API Common Client Interface.

Afin de décrire ce support, nous utilisons le connecteur `BlackBox`, disponible dans le SDK de J2EE. Nous avons retravaillé le code de ce connecteur afin de le rendre davantage paramétrable au niveau de la spécification du pilote JDBC et de la requête SQL utilisés. Le code modifié ainsi que le fichier jar correspondant sont disponibles dans l'étude de cas Tudu Lists.

Comme indiqué précédemment, un connecteur peut être configuré de deux manières. La première consiste à le déployer dans un serveur d'applications afin de rendre disponible par le biais de JNDI une instance de la fabrique de connexions correspondante.

Dans ce cas, aucune configuration particulière n'est nécessaire pour le connecteur. L'application doit se fonder sur une instance de la fabrique en utilisant son support JNDI

par l'intermédiaire de la classe `JnidObjectFactoryBean` de Spring, comme dans le code suivant :

```
<bean id="connectionFactory"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="eis/monConnecteur"/>
    <property name="jndiEnvironment">
        (...)
    </property>
</bean>
```

Il est également possible de configurer un connecteur en dehors d'un serveur d'applications dans une application. Pour ce faire, il convient d'instancier explicitement l'implémentation de l'interface `ManagedConnectionFactory` du connecteur puis d'utiliser le support de la classe `LocalConnectionFactoryBean` du package `org.springframework.jca.support` afin de configurer une instance de la fabrique de connexions, comme dans le code suivant :

```
<!-- Fabrique de connexion interne au connecteur -->
<bean id="managedConnectionFactory" class="com.sun.connector
                .cciblackbox.CciLocalTxManagedConnectionFactory">
    <property name="connectionURL"
              value="jdbc:mysql://localhost:3306/tudu"/>
    <property name="driverName" value="com.mysql.jdbc.Driver"/>
</bean>

<!-- Fabrique de connexion associée -->
<bean id="connectionFactory" class="org.springframework.jca
                            .support.LocalConnectionFactoryBean">
    <property name="managedConnectionFactory"
              ref="managedConnectionFactory"/>
</bean>
```

Il est possible d'utiliser le connecteur en mode managé hors d'un serveur d'applications en injectant une instance d'un gestionnaire de connexions JCA par l'intermédiaire de la propriété `connectionManager`. Le projet Jencks offre un support afin de configurer facilement dans Spring le gestionnaire de connexions JCA ainsi que le gestionnaire de transactions de Geronimo de manière autonome.

Avec ces deux approches, nous pouvons configurer avec JCA des fabriques de connexions de connecteurs. Il est important de bien comprendre que ces fabriques ne sont pas nécessairement compatibles avec l'API cliente CCI. En effet, il est possible de configurer une `DataSource` JDBC ou une `SessionFactory` d'Hibernate de cette manière, à condition de posséder le connecteur correspondant. Puisque JCA normalise les interactions avec les gestionnaires de transactions JTA, cette approche permet de rendre des fabriques de connexions facilement compatibles XA afin de les faire participer à des transactions globales *(pour plus d'informations à ce sujet, voir le chapitre 12, relatif à la gestion des transactions).*

Support de l'API Common Client Interface

Spring offre un support de la partie cliente normalisée, Common Client Interface. Il reprend les concepts du support JDBC du framework afin de simplifier l'utilisation de cette API, d'encapsuler le code répétitif, de masquer sa complexité et de configurer la participation aux transactions.

CCI donne la possibilité de paramétrer la connexion grâce à l'implémentation de l'interface `ConnectionSpec` du connecteur. Le support CCI de Spring permet la mise en œuvre de cette fonctionnalité par l'intermédiaire de la classe `ConnectionSpecConnectionFactoryAdapter` du package `org.springframework.jca.cci.connection`, correspondant à un proxy sur la fabrique de connexions cible.

Le code suivant donne la configuration de ce mécanisme :

```
(...)
<bean id="targetConnectionFactory" class="org.springframework.jca
                              .support.LocalConnectionFactoryBean">
    (...)
</bean>

<bean id="myConnectionFactory" class="org.springframework.jca.cci
              .connection.ConnectionSpecConnectionFactoryAdapter">
    <property name="targetConnectionFactory"
              ref="targetConnectionFactory"/>
    <property name="connectionSpec">
        <bean
           class="com.sun.connector.cciblackbox.CciConnectionSpec">
            <property name="username" value="root"/>
            <property name="password" value=""/>
        </bean>
    </property>
</bean>
```

Le support CCI de Spring propose deux approches afin d'exécuter des requêtes, l'une fondée sur le template, l'autre étant une approche objet.

La classe centrale de l'approche fondée sur le template est `CciTemplate`, localisée dans le package `org.springframework.jca.cci.core`. Elle permet de travailler directement sur les ressources CCI qu'elle gère par le biais de méthodes de rappel, aux niveaux de la connexion et de l'interaction.

Elle s'appuie pour cela sur les interfaces `ConnectionCallback` et `InteractionCallback`. La première met à disposition la connexion grâce à la méthode `doInConnection`, comme ci-dessous :

```
public interface ConnectionCallback {
    Object doInConnection(Connection connection,
                          ConnectionFactory connectionFactory)
                          throws ResourceException,
                                 SQLException, DataAccessException;
}
```

La seconde met à disposition l'interaction grâce à la méthode `doInInteraction` :

```
public interface InteractionCallback {
    Object doInInteraction(Interaction interaction,
                           ConnectionFactory connectionFactory)
                           throws ResourceException,
                               SQLException, DataAccessException;
}
```

Le template CCI utilise ces interfaces par l'intermédiaire des méthodes `execute` de la manière suivante :

```
CciTemplate template = getCciTemplate();
template.execute(new InteractionCallback() {
    public Object doInInteraction(Interaction interaction,
                           ConnectionFactory connectionFactory)
                           throws ResourceException,
                               SQLException, DataAccessException {
        IndexedRecord input =
                    recordFactory.createIndexedRecord("input");
        input.add(new Integer(id));
        CciInteractionSpec interactionSpec =
                                     new CciInteractionSpec();
        interactionSpec.setSql(
                     "select * from todo where id=?");
        Record output = interaction.execute(
                                      interactionSpec, input);
        return extractDataFromRecord(output);
    }
});
```

Le template CCI offre des fonctionnalités permettant de masquer tout le code d'invocation d'une requête par l'intermédiaire de ses autres méthodes `execute`. Ces dernières s'appuient sur l'interface `RecordCreator` afin de définir la façon de créer une instance de la classe `Record` à partir d'un objet. Le code de cette interface est le suivant :

```
public interface RecordCreator {
    Record createRecord(RecordFactory recordFactory)
                    throws ResourceException, DataAccessException;
}
```

Ces méthodes utilisent également l'interface `RecordExtractor` afin de récupérer les données contenues dans un `Record` dans un objet Java simple. Le code de cette interface est le suivant :

```
public interface RecordExtractor {
    Object extractData(Record record) throws ResourceException,
                                  SQLException, DataAccessException;
}
```

Ces deux interfaces sont localisées dans le package `org.springframework.jca.cci.core` et peuvent être mises en œuvre de la manière suivante afin d'exécuter une requête avec des classes anonymes :

```
final int id = 10;
CciInteractionSpec interactionSpec=new CciInteractionSpec();
interactionSpec.setSql("select * from todo where id=?");

List people=(List)getCciTemplate().execute(
               interactionSpec, new RecordCreator() {
    public Record createRecord(RecordFactory recordFactory)
                    throws ResourceException, DataAccessException {
        IndexedRecord input =
                 recordFactory.createIndexedRecord("input");
        input.add(new Integer(id));
        return input;
    }
}, new RecordExtractor() {
    public Object extractData(Record record)
      throws ResourceException, SQLException, DataAccessException {
        List todos=new ArrayList();
        ResultSet rs=(ResultSet)record;
        while( rs.next() ) {
            Todo todo=new Todo();
            todo.setTodoId(rs.getString("id"));
            todo.setDescription(rs.getString("description"));
            (...)
            todos.add(todo);
        }
        return todo;
    }
});
```

Puisque CCI définit une méthode `execute` attendant le `Record` de retour, le template implémente un mécanisme de création automatique de celui-ci. Il s'appuie pour cela sur sa propriété `outputRecordCreator` de type `RecordCreator`, permettant de spécifier la façon de le créer. Ce mécanisme est particulièrement intéressant pour les connecteurs ne supportant que cette façon de réaliser des requêtes.

En parallèle du template, le support CCI offre la possibilité de définir les requêtes CCI sous forme d'objets afin de bénéficier de toutes les possibilités du langage objet. Cette approche se fonde sur la classe abstraite `MappingRecordOperation` du package `org.springframework.jca.cci.object`. Elle doit être étendue afin d'implémenter les méthodes `createInputRecord` et `extractOutputData` décrivant respectivement la façon de créer un `Record` et d'extraire les données du `Record` renvoyé.

La classe suivante décrit l'adaptation de l'exemple de la section précédente à l'approche objet :

```
public class TodosOperation extends MappingRecordOperation {
    public TodosOperation(ConnectionFactory connectionFactory,
```

```
                            InteractionSpec interactionSpec) {
        super(connectionFactory,interactionSpec);
    }

    protected Record createInputRecord(
             RecordFactory recordFactory, Object inputObject)
                  throws ResourceException, DataAccessException {
        Integer id=(Integer)inputObject;
        IndexedRecord input =
                 recordFactory.createIndexedRecord("input");
        input.add(id);
        return input;
    }

    protected Object extractOutputData(
                 Record outputRecord) throws ResourceException,
                               SQLException, DataAccessException {

        List todos=new ArrayList();
        ResultSet rs=(ResultSet)outputRecord;
        while( rs.next() ) {
            Todo todo=new Todo();
            todo.setTodoId(rs.getString("id"));
            todo.setDescription(rs.getString("description"));
            (...)
            todos.add(todo);
        }
        return todos;
    }
}
```

Cette classe peut être utilisée de la manière suivante dans un composant d'accès aux données :

```
CciInteractionSpec interactionSpec=new CciInteractionSpec();
interactionSpec.setSql("select * from todo where id=?");

TodosOperation operation=new TodosOperation(
                        getConnectionFactory(), interactionSpec);
operation.execute(new Integer(10));
```

Notons la présence de la classe `MappingCommAreaOperation`, qui étend la classe `MappingRecordOperation` afin de travailler directement sur des tableaux d'octets. Cette classe facilite l'utilisation des connecteurs tels que celui utilisé dans le cas de CICS ECI et qui échange des données en se fondant sur une COMMAREA (correspondant à un tableau d'octets).

Communications entrantes

La version 2.0 de Spring ne fournit pas, au moment de l'écriture de cet ouvrage, de support afin de configurer les communications entrantes, mais cette problématique devrait être adressée avec la version 2.1.

Par contre, le projet Jencks offre la possibilité de les mettre en œuvre en utilisant les fonctionnalités de Spring.

Configuration du conteneur

Tout d'abord, il convient de configurer le connecteur ainsi que l'entité de gestion du connecteur pour les connexions entrantes, entité désignée par Jencks sous le terme conteneur JCA et implémentée par la classe `JCAContainer` du package `org.jencks`.

Ce conteneur se charge de démarrer le connecteur avec le contexte spécifié et de l'arrêter en se fondant sur le cycle de vie des Beans configurés dans Spring. Le connecteur est configuré en tant que Bean par l'intermédiaire de son implémentation de l'interface `ResourceAdapter`. La configuration de ce conteneur se réalise de la manière suivante avec le connecteur du fournisseur JMS ActiveMQ :

```
<bean id="jencks" class="org.jencks.JCAContainer">
    <!-- Configuration du contexte de démarrage -->
    <property name="bootstrapContext">
        <bean
           class="org.jencks.factory.BootstrapContextFactoryBean">
            <property name="threadPoolSize" value="25"/>
        </bean>
    </property>

    <!-- Configuration du ResourceAdapter du connecteur -->
    <property name="resourceAdapter">
        <bean id="activeMQResourceAdapter"
              class="org.activemq.ra.ActiveMQResourceAdapter">
            <property name="serverUrl"
                      value="tcp://localhost:61616"/>
        </bean>
    </property>
</bean>
```

Définition de points d'activation

Une fois le conteneur mis en œuvre, les différents points d'accès au connecteur peuvent être spécifiés afin que le système d'information d'entreprise puisse les utiliser et remonter des informations.

L'implémentation de l'interface `ActivationSpec` relative au connecteur JCA doit être utilisée à cet effet. Puisque nous avons choisi d'utiliser le fournisseur JMS ActiveMQ,

l'implémentation correspond à la classe `ActiveMQActivationSpec`. Cette classe permet de spécifier le domaine du point d'entrée ainsi que son nom.

Dans l'exemple ci-dessous, nous créons un point d'accès sur le connecteur de type `javax.jms.Queue`, dont le nom est `tudu.queue` :

```
<bean id="inboundConnector" class="org.jencks.JCAConnector">
    <property name="jcaContainer" ref="jencks"/>

    <property name="activationSpec">
        <bean class="org.activemq.ra.ActiveMQActivationSpec">
            <property name="destination" value="tudu.queue"/>
            <property name="destinationType"
                      value="javax.jms.Queue"/>
        </bean>
    </property>

    <property name="ref" value="asynchTuduJmsListener"/>
</bean>

<bean id="asynchTuduJmsListener"
      class="tudu.jms.AsynchTuduJmsListener"/>
```

Cette configuration utilise un observateur JMS classique afin de recevoir les messages JMS dont l'identifiant est `asynchrTuduJmsListener`.

En résumé

Le support JCA de Spring réduit la complexité liée à l'utilisation de l'API Common Client Interface pour les communications sortantes, tout en permettant d'utiliser les connecteurs dans et hors des serveurs d'applications.

Le framework Jencks offre un complément permettant de configurer les connexions entrantes pour des connecteurs. Par ce biais, il est notamment possible de mettre en œuvre des observateurs JMS afin d'utiliser les mécanismes asynchrones de communication.

Tudu Lists : utilisation de JMS et JCA

La mise en œuvre de JMS et JCA dans l'application Tudu Lists permet d'implémenter l'envoi de messages JMS lors de la réalisation de tâches par le biais d'une file JMS nommée `tudu.queue`. Des applications autonomes sont à l'écoute de ces messages afin d'afficher leurs contenus. Ces fonctionnalités sont disponibles dans le projet `Tudu-SpringMVC`.

Tudu Lists propose deux approches afin d'implémenter cette fonctionnalité. La première s'appuie sur JMS et la seconde sur JCA. La figure 13.6 illustre les mécanismes mis en œuvre.

Toutes les classes relatives à la mise en œuvre de ces technologies sont localisées dans le fichier **applicationContext-jms.xml** du répertoire **WEB-INF.**

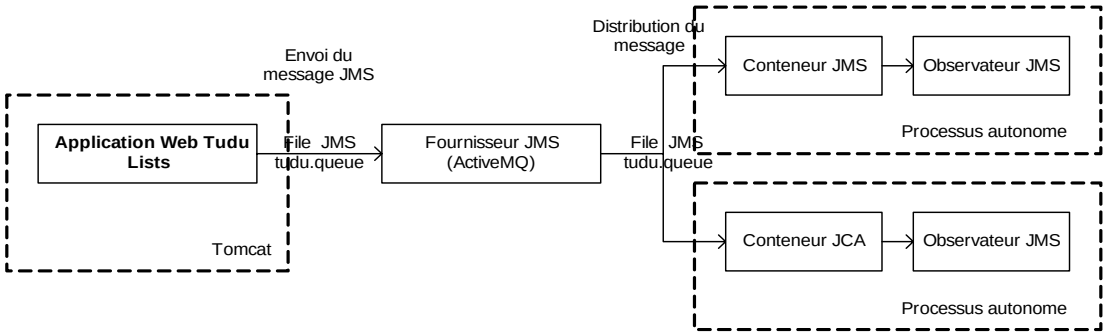


Figure 13.6

Implémentation des mécanismes JMS et JCA dans Tudu Lists

L'application nécessite la mise en œuvre d'un fournisseur JMS. Nous avons choisi d'utiliser ActiveMQ à cet effet. Afin de garantir le bon fonctionnement de l'application, il est nécessaire de démarrer ce fournisseur JMS de manière autonome par l'intermédiaire du script **build.xml,** localisé à la racine du projet. Ce script peut être exécuté dans Eclipse en le sélectionnant et en choisissant le menu contextuel Run As/Ant Build.

Configuration de l'intercepteur

Afin de ne pas impacter le code de l'application existante, nous créons un intercepteur en POA afin d'ajouter le mécanisme d'envoi de messages JMS. Nous détaillons l'implémentation de ce composant à la section relative à l'envoi des messages JMS.

Nous appliquons cette entité sur le composant de la couche service métier `TodoManager`. Le tissage de l'intercepteur se réalise de la manière suivante dans le fichier **applicationContext.xml** du répertoire **WEB-INF :**

```
<bean id="todosManager" class="org.springframework.transaction
                        .interceptor.TransactionProxyFactoryBean">
    (...)
    <property name="target" ref="todosManagerTarget" />
    <property name="preInterceptors">
        <list>
            <ref bean="jmsInterceptor"/>
        </list>
    </property>
    (...)
</bean>
```

Dans la mesure où nous utilisons le fournisseur JMS ActiveMQ afin d'échanger des messages JMS, il est nécessaire de configurer la fabrique de connexions relative à cet

outil ainsi que le template JMS de Spring et l'intercepteur. Cette configuration est localisée dans le fichier **applicationContext-jms.xml,** comme ci-dessous :

```
<bean id="connectionFactory"
      class="org.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>

<bean id="template"
      class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="connectionFactory"/>
    <property name="defaultDestinationName" value="tudu.queue"/>
</bean>
```

Envoi des messages

L'envoi des messages est déclenché par l'intercepteur `JmsInterceptor`, localisé dans le package `tudu.jms`.

Il permet de créer un objet de type `TuduMessage`, suite à l'exécution d'un traitement pour un todo. Cet objet contient les informations relatives au todo impacté ainsi que l'opération réalisée. Il est créé de la manière suivante par le biais de la méthode `createTuduMessage` de l'intercepteur :

```
    private TuduMessage createTuduMessage(
                                MethodInvocation invocation) {
    String methodName = invocation.getMethod().getName();
    Object[] args = invocation.getArguments();
    if( args[0] instanceof String ) {
        String todoId = (String)args[0];

        TuduMessage message = new TuduMessage();
        message.setOperationType(methodName);
        message.setTodoId(todoId);
        return message;
    } else {
        Todo todo = (Todo)args[0];

        TuduMessage message = new TuduMessage();
        message.setOperationType(methodName);
        message.setTodoId(todo.getTodoId());
        message.setTodoCompleted(todo.isCompleted());
        message.setTodoCreationDate(todo.getCreationDate());
        message.setTodoDescription(todo.getDescription());
        message.setTodoPriority(todo.getPriority());
        return message;
    }
}
```

Notons que la méthode précédente gère les méthodes utilisant des paramètres de types `String` (cas de la méthode `findTodo` de l'interface `TodosManager`) et `Todo` (cas des méthodes `updateTodo` de cette même interface).

Une fois cet objet créé, le support JMS de Spring est utilisé afin de l'encapsuler dans un message JMS de type `ObjectMessage` et de l'envoyer. Ces traitements sont réalisés de la façon suivante par l'intermédiaire de la méthode `sendMessage` de l'intercepteur :

```
private void sendMessage(final TuduMessage message) {
    template.send(new MessageCreator() {
        public Message createMessage(
                            Session session) throws JMSException {
            ObjectMessage objectMessage =
                                session.createObjectMessage();
            objectMessage.setObject(message);
            return objectMessage;
        }
    });
}
```

L'intercepteur décrit dans cette section est configuré dans le fichier **applicationContext-jms.xml** de la manière suivante :

```
(...)
<bean id="jmsInterceptor" class="tudu.jms.JmsInterceptor">
    <property name="template" ref="template"/>
</bean>
```

Réception des messages

Tudu Lists offre deux possibilités pour recevoir les messages JMS précédents, toutes deux de manière asynchrone.

La première s'appuie sur le support JMS de Spring et la seconde sur le conteneur JCA de Jencks.

La configuration de la première se réalise de la manière suivante dans le fichier **applicationContext-listenerJms.xml :**

```
<bean id="connectionFactory"
      class="org.activemq.ActiveMQConnectionFactory">
    <property name="brokerURL" value="tcp://localhost:61616"/>
</bean>

<bean id="jmsContainer" class="org.springframework.jms
                    .listener.SimpleMessageListenerContainer">
    <property name="connectionFactory" ref="connectionFactory"/>
```

```
    <property name="destinationName" value="tudu.queue"/>
    <property name="messageListener" ref="asynchTuduJmsListener"/>
</bean>

<bean id="asynchTuduJmsListener"
      class="tudu.jms.AsynchTuduJmsListener"/>
```

La configuration de la seconde se réalise de la manière suivante dans le fichier **application-Context-jencks.xml :**

```
<bean id="jencks" class="org.jencks.JCAContainer">
    <property name="bootstrapContext">
        <bean
           class="org.jencks.factory.BootstrapContextFactoryBean">
            <property name="threadPoolSize" value="25" />
         </bean>
    </property>

    <property name="resourceAdapter">
        <bean id="activeMQResourceAdapter"
              class="org.activemq.ra.ActiveMQResourceAdapter">
            <property name="serverUrl"
                      value="tcp://localhost:61616" />
        </bean>
    </property>
</bean>

<bean id="inboundConnector" class="org.jencks.JCAConnector">
    <property name="jcaContainer" ref="jencks" />

    <property name="activationSpec">
        <bean class="org.activemq.ra.ActiveMQActivationSpec">
            <property name="destination" value="tudu.queue"/>
            <property name="destinationType"
                      value="javax.jms.Queue"/>
        </bean>
    </property>

    <property name="ref" value="asynchTuduJmsListener"/>
</bean>

<bean id="asynchTuduJmsListener"
      class="tudu.jms.AsynchTuduJmsListener"/>
```

L'une et l'autre approches peuvent être utilisées directement dans une application Java autonome par l'intermédiaire d'une classe du type suivant :

```
public class AsynchTuduListenerMain {
    (...)

    public static void main(String[] args) {
        ClassPathXmlApplicationContext context=null;
```

```
        try {
            context=new ClassPathXmlApplicationContext(
                            getApplicationContextFile(););
            Thread.sleep(60000);
        } catch(Exception ex) {
            (...)
        } finally {
            if( context!=null ) {
                context.close();
            }
        }
        System.exit(0);
    }
}
```

La méthode `getApplicationContextFile` renvoie le nom du fichier XML de Spring suivant l'approche souhaitée : **applicationContext-listenerJms.xml** pour l'utilisation du support JMS de Spring et **applicationContext-jencks.xml** pour l'utilisation de Jencks.

Ces deux fichiers sont localisés dans le répertoire **JavaSource** du projet **Tudu-SpringMVC.**

Conclusion

J2EE propose différentes spécifications pour intégrer les applications Java/J2EE dans les systèmes d'information d'entreprise. Les technologies correspondantes sont JMS, afin d'interagir avec des middlewares orientés messages, et JCA, afin d'accéder à des applications d'entreprise utilisant diverses technologies.

Ces spécifications ne sont pas d'une utilisation facile dans les applications Java/J2EE, car elles impliquent le recours conjoint à des mécanismes synchrones et asynchrones, tout en gérant des transactions plus complexes sur plusieurs ressources.

Spring offre un support pour ces deux technologies permettant de les configurer simplement en masquant la complexité de leur utilisation dans les applications. Leur mise en œuvre avec la POA permet de faire interagir des composants existants dans des systèmes d'information d'entreprise sans impacter le code de ces composants et simplement au moment de l'assemblage de l'application.

14

Technologies d'intégration XML

Après les technologies d'intégration Java, abordées au chapitre précédent, nous allons détailler de quelle manière le XML permet d'intégrer des systèmes hétérogènes, distants et faiblement couplés.

Présentée à ses débuts comme une solution miracle pour l'intégration des données, le XML se révèle une technologie d'usage complexe, souvent peu performante, mais surtout surchargée d'une pléthore de normes obscures.

Nous verrons dans un premier temps de quelle manière faire du XML simplement et efficacement, en utilisant des bibliothèques génériques (JDOM) ou spécialisées (Rome). Nous verrons notamment comment intégrer notre application Tudu Lists à la page d'accueil de Google *via* un flux RSS.

L'informatique d'entreprise est envahie de concepts tels que le SOA ou les services Web adjoints au XML. Nous allons définir ces termes, ainsi qu'en montrer les aspects utiles en matière d'architecture logicielle et en détailler les principales normes, SOAP et WSDL en particulier.

En fin de chapitre, nous donnerons un exemple de mise en œuvre de service Web en publiant dans Tudu Lists un Bean Spring en tant que service Web à l'aide de deux frameworks Open Source populaires, XFire et Axis. Nous insisterons sur leurs avantages et inconvénients et construirons une application Swing simple permettant d'appeler ce service Web, afin d'illustrer son fonctionnement.

Le XML sur HTTP

Bien que nous n'utilisions dans cette section aucune API Spring spécifique, les exemples qui la parsèment sont réalisés à l'aide de Beans Spring standards. Spring offre une architecture permettant de gérer aisément la problématique du XML sur HTTP, sans ajout d'une surcouche supplémentaire. Nous verrons que cela n'est pas nécessaire, car il ne s'agit pas d'un sujet si complexe.

Envoyer du XML *via* un flux HTTP est une méthode utilisée depuis de nombreuses années afin de faire communiquer entre elles deux applications distantes. Cette solution est d'autant plus séduisante qu'elle est simple à mettre en œuvre et que les grands langages de programmation offrent tous un excellent support de la technologie XML.

Pour peu que nous nous mettions d'accord au préalable sur le format d'échange, il est donc aisé de faire communiquer des applications développées dans des langages différents.

Cette technique se heurte toutefois aux trois écueils suivants :

- La lecture de documents XML est complexe si nous utilisons les API Java standards. De plus, l'écriture de documents XML est souvent mal supportée. Une API permettant d'écrire du XML a donc son utilité si nous voulons produire un document complexe en suivant une DTD.
- Faire correspondre un ensemble d'objets avec un document XML n'est pas simple. Il est possible de parler de mappage objet/XML, à la manière du mappage objet/relationnel.
- Le XML étant un métalangage, il n'existe pas de norme définie pour spécifier un format d'échange. Cette souplesse exige en contrepartie un temps important de spécification. Comment, par exemple, spécifier un champ de date ?

Ces difficultés se retrouvent dans la création de services Web, que nous détaillons plus loin dans ce chapitre.

Nous allons pour l'instant nous armer d'outils efficaces, qui nous permettront de résoudre le plus simplement possible la problématique de l'envoi de données XML *via* HTTP.

Lecture et écriture avec JDOM

Il existe des API standards pour lire le XML, nommément DOM (Document Object Model) et SAX (Simple API for XML), mais la première est réputée trop lente, et la seconde trop complexe. Nous estimons pour notre part que la réalité confirme largement ce jugement et que ces API ne sont pas utilisables par quiconque souhaite délivrer son projet à temps. Par ailleurs, elles ne règlent pas le problème de l'écriture du XML, qui est pour l'instant notre première tâche : avant de lire un flux XML, il faut pouvoir l'écrire.

Nous rencontrons souvent en entreprise des développeurs qui, suite à ces difficultés, fondent leurs applications sur des concaténations et des recherches de chaînes de caractères.

Il s'agit là d'une erreur majeure au vu du grand nombre d'outils déjà disponibles sur le marché, qui permettent de lire et d'écrire du XML de manière simple et conviviale.

Dans Tudu Lists, nous utilisons une bibliothèque Open Source très populaire, nommée JDOM, disponible à l'adresse *http://www.jdom.org.* Sans être particulièrement performante, cette bibliothèque permet de lire et d'écrire sans difficulté du XML et répond à tous les besoins d'intégration simple *(pour des besoins plus particuliers, voir la section « Les services Web », plus loin dans ce chapitre).*

Dans Tudu Lists, nous avons besoin d'une solution de backup et de récupération des données. Le XML nous paraissant un système de stockage pertinent pour cela, nous créons deux méthodes spécifiques dans le Bean `todoListsManager` : `backupTodoList()`, pour écrire une liste au format XML, et `restoreTodoList()`, pour le lire.

Voici une version simplifiée de la première méthode :

```
public Document backupTodoList(TodoList todoList) {
  Document doc = new Document();
  Element todoListElement = new Element("todolist");
  todoListElement.addContent(new Element("title")
    .addContent(todoList.getName()));

  Element todosElement = new Element("todos");
  for (Todo todo : todoList.getTodos()) {
    Element todoElement = new Element("todo");
    todoElement.setAttribute("id", todo.getTodoId());
    todoElement.addContent(new Element("description")
      .addContent(todo.getDescription()));
    todoElement.addContent(new Element("priority")
      .addContent(Integer.toString(todo.getPriority())));
    todoElement.addContent(new Element("completed")
      .addContent(Boolean.toString(todo.isCompleted())));

    todosElement.addContent(todoElement);
  }
  todoListElement.addContent(todosElement);

  doc.addContent(todoListElement);
  return doc;
}
```

Ce code crée un document XML de la forme suivante :

```
<todolist>
  <title>Liste de tâches</title>
  <todos>
    <todo id="4028944508eb569d0108ef0c72c30023">
      <description>Première tâche à faire</description>
      <priority>100</priority>
      <completed>false</completed>
    </todo>
    <todo id="4028944508eb569d0108ef0c3aac0022">
```

```
      <description>Deuxième tâche à faire</description>
      <priority>50</priority>
      <completed>false</completed>
    </todo>
  </todos>
</todolist>
```

Un `Element` est un nœud XML que nous ajoutons à l'aide de la méthode `addContent()`. L'ajout d'un attribut à un nœud existant s'effectue grâce à la méthode `setAttribute()`. L'API JDOM est donc en pratique extrêmement simple.

Pour la lecture, Tudu Lists propose des fonctionnalités assez complètes de création, remplacement et fusion de listes. Voici, une nouvelle fois en version simplifiée, le code lisant le flux XML :

```
import org.jdom.Document;
import org.jdom.Element;
import org.jdom.input.SAXBuilder;

( ... )

SAXBuilder saxBuilder = new SAXBuilder();
Document doc = saxBuilder.build(inputStream);
Element rootElement = doc.getRootElement();

String title = rootElement.getChildText("title");
TodoList todoList = new TodoList();
todoList.setName(title);
todoListDAO.saveTodoList(todoList);

Element todosElement = rootElement.getChild("todos");
List todos = todosElement.getChildren();
for (Object todoObject : todos) {
  Element todoElement = (Element) todoObject;
  Todo todo = new Todo();
  todo.setDescription(todoElement.getChildText("description"));
  todo.setPriority(Integer.valueOf(
    todoElement.getChildText("priority")));
  todo.setCompleted(Boolean.parseBoolean(
    todoElement.getChildText("completed")));
  }
  todo.setTodoList(todoList);
  todoList.getTodos().add(todo);
  todoDAO.saveTodo(todo);
}
```

De même que pour l'écriture, l'API utilisée est très claire. Nous commençons par construire un `Document` JDOM à partir d'un objet de type `java.io.InputStream`. Une fois cet objet créé, nous pouvons lire le nœud racine grâce à la méthode `getRootElement`, lire les enfants

d'un nœud avec la méthode `getChildren()` ou lire le contenu d'un nœud avec la méthode `getChildText()`. Voilà qui est nettement plus pratique que les API DOM et SAX.

Les deux méthodes que nous venons de mettre en œuvre autorisent l'écriture et la lecture d'un flux XML. Elles sont contenues dans la couche métier de l'application. Il est possible, dans les couches supérieures de l'application, de les transporter de différentes manières. Nous ne sommes en aucun cas liés au protocole HTTP. Par exemple, nous venons de voir que la méthode de lecture avait comme point de départ un objet de type `java.io.InputStream`, ce qui est très générique. Le HTTP étant cependant une méthode de transfert de données particulièrement simple d'utilisation, c'est elle que nous retenons pour les phases d'envoi et de réception des documents XML traités.

Pour l'envoi du document XML, nous utilisons une Action Struts, `tudu.web.BackupTodoListAction`, dans laquelle Spring injecte le Bean `todoListsManager`. Cette Action renvoie vers une servlet faisant office de vue, `tudu.web.servlet.BackupServlet`, qui envoie elle-même le XML grâce à `XMLOutputter`, un objet JDOM spécifique qui formate le XML :

```
XMLOutputter outputter =
  new XMLOutputter(Format.getPrettyFormat());

Writer writer = response.getWriter();
outputter.output(doc, writer);
writer.close();
```

Pour la réception du document XML, nous utilisons la fonctionnalité d'upload de fichiers de Struts et transformons ce fichier en objet `java.io.InputStream`.

En résumé, nous constatons que l'utilisation de JDOM, en conjugaison avec une couche de présentation classique fondée sur des Actions Struts et des servlets, permet de réaliser simplement et rapidement une interface d'échange XML sur HTTP. De plus, le code résultant est concis et rapidement compréhensible.

Publication d'un flux RSS

Le RSS (Really Simple Syndication) est un format largement répandu permettant l'agrégation de contenu simple, essentiellement des informations telles que des nouvelles ou des actualités. Ce format se retrouve notamment sur les blogs ou les wikis.

Sous l'ombrelle RSS se cache en réalité une multitude de formats, si bien que l'utilisation directe de JDOM peut se révéler complexe. Heureusement, la bibliothèque Open Source Rome nous aide dans cette tâche. Elle est disponible à l'adresse *http://wiki.java.net/bin/view/Javawsxml/Rome*.

Rome illustre à merveille la pertinence de l'approche que nous venons de présenter puisqu'elle est elle-même fondée sur JDOM.

Pour Tudu Lists, l'ajout d'un flux RSS présente l'intérêt tout particulier de permettre aux utilisateurs d'agréger leurs listes de tâches et, dans le cadre de listes partagées, d'être rapidement informés des actions des autres utilisateurs.

La méthode de publication du flux RSS est similaire à celle que nous avons déjà vue pour la publication d'un flux XML classique. Nous utilisons Spring pour injecter le Bean `todoListsManager` dans l'Action Struts `tudu.web.ShowRssFeedAction` et renvoyons vers une servlet jouant le rôle de vue.

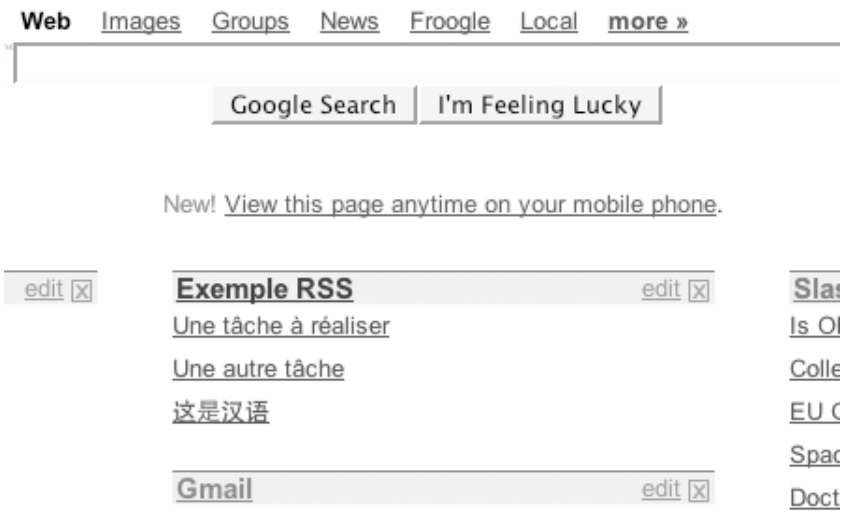
Cette servlet utilise l'API de Rome pour générer un flux RSS au sein de sa méthode `doGet()` (précisons que les packages utilisés commencent par `com.sun` car il s'agit à l'origine d'une API développée par des ingénieurs travaillant chez Sun) :

```
import com.sun.syndication.feed.synd.SyndContent;
import com.sun.syndication.feed.synd.SyndContentImpl;
import com.sun.syndication.feed.synd.SyndEntry;
import com.sun.syndication.feed.synd.SyndEntryImpl;
import com.sun.syndication.feed.synd.SyndFeed;
import com.sun.syndication.feed.synd.SyndFeedImpl;
import com.sun.syndication.io.FeedException;
import com.sun.syndication.io.SyndFeedOutput;

SyndFeed feed = new SyndFeedImpl();
feed.setFeedType("rss_2.0");
feed.setTitle(todoList.getName());
feed.setLink(link);
feed.setDescription("Tudu Lists | " + todoList.getName());
List<SyndEntry> entries = new ArrayList<SyndEntry>();
for (Todo todo : todos) {
  SyndEntry entry = new SyndEntryImpl();
  entry.setLink(link + "?listId="
    + todoList.getListId() + "#todoId" + todo.getTodoId());

  entry.setTitle(todo.getDescription());
  entry.setPublishedDate(todo.getCreationDate());
  SyndContent description = new SyndContentImpl();
  description.setType("text/plain");
  description.setValue(todo.getDescription());
  entry.setDescription(description);
  entries.add(entry);
}
feed.setEntries(entries);
response.setContentType("application/xml; charset=UTF-8");

SyndFeedOutput output = new SyndFeedOutput();
try {
  output.output(feed, response.getWriter());
} catch (FeedException fe) {
  String msg = "The RSS feed could not be generated.";
  log.error("Error while generating the RSS feed : " +
    fe.getMessage());
  response.sendError(HttpServletResponse.
    SC_INTERNAL_SERVER_ERROR,msg);
}
```

Ce code montre bien que Rome propose de renseigner chaque élément du flux RSS en tant qu'objet `Entry`, lequel possède les attributs classiques que nous retrouvons en RSS : un lien, un titre, la date de publication et une description.

Ce travail permet d'intégrer des listes de todos dans tout logiciel proposant le format RSS, ce qui est le cas de la page d'accueil personnalisée de Google. La figure 14.1 illustre une copie d'écran de Google affichant la liste « Exemple RSS », laquelle comporte même une tâche écrite en chinois, afin de bien montrer que l'ensemble est parfaitement intégré.

Figure 14.1
Intégration d'un flux RSS de Tudu Lists dans la page d'accueil de Google



En résumé

Nous avons vu avec quelle facilité nous pouvions utiliser le XML pour intégrer une application Java à d'autres systèmes. Que ce soit avec JDOM ou Rome, le code écrit reste concis et lisible. Ajoutons que le découpage en couches de l'application et l'utilisation de Spring nous ont permis d'avoir des interfaces métier claires, ce qui facilite grandement le travail.

Cependant, nous n'avons pour l'instant traité que des exemples simples, avec des interfaces bien définies, comme le RSS. Nous allons voir à la section suivante comment intégrer des systèmes plus complexes, grâce à l'utilisation des services Web.

Les services Web

Les services Web, ou Web Services, sont un ensemble de spécifications, particulièrement complexes, visant à faire communiquer des applications distantes *via* un format XML standardisé.

La Web Services Interoperability Organization propose une spécification complète, nommée Basic Profile, qui garantit qu'un service Web peut être utilisé sans problème, quel que soit le langage de programmation utilisé. Cette spécification, ainsi qu'un outil permettant de vérifier la conformité d'un service Web, est disponible sur le site Web de l'organisation, à l'adresse *http://www.ws-i.org/*.

Cette technique d'intégration de systèmes est souvent couplée au concept de SOA (Service Oriented Architecture). Le style d'architecture SOA prône la création d'applications développées en couches, avec une couche métier mise à la disposition d'autres applications. En ce sens, une application Spring est une candidate toute désignée pour faire du SOA, puisqu'il s'agit d'offrir un accès à la couche métier (les Beans `todosManager` et `todoListsManager` dans notre exemple).

Notons bien que les services Web ne sont qu'un moyen parmi d'autres de publier des services métier distants. Si nos applications sont toutes développées en Java, par exemple, l'utilisation de RMI est à la fois plus simple et plus efficace que celle des services Web.

Spring offre également le support d'autres protocoles, comme Burlap et Hessian, qui sont eux aussi plus performants que les services Web classiques.

Reste que l'utilisation de services Web est un bon moyen de publier une couche métier de manière indépendante du langage d'implémentation, tout en suivant les grands standards actuels du marché. C'est ce que nous allons voir dans le reste de cette section.

Concepts des services Web

Les services Web s'appuient sur une trentaine de spécifications, dont nous ne détaillons ici que les deux principales, SOAP et WSDL. Ces spécifications sont toutes émises par le W3C (World-Wide Web Consortium) et sont par conséquent des standards internationaux largement reconnus.

SOAP signifiait à l'origine « Simple Object Access Protocol » (le nom complet n'est plus utilisé). Il s'agit d'un protocole permettant de décrire des objets dans un format XML, afin de pouvoir échanger ces objets *via* Internet.

Les applications SOAP utilisent essentiellement le protocole HTTP pour effectuer ces échanges d'objets, mais elles peuvent également utiliser d'autres protocoles, comme SMTP ou JMS. Malgré son nom d'origine, SOAP n'est pas un standard simple d'utilisation. Il a de plus le défaut d'être particulièrement lent, la transformation d'objets en XML étant un mécanisme intrinsèquement coûteux en ressources.

WSDL (Web Services Description Language) — prononcez *wizzdle* — décrit de quelle manière il est possible d'utiliser un service Web (protocole à utiliser, méthodes accessibles, objets utilisés). L'obtention d'un document WSDL est essentielle pour utiliser un service Web, car c'est ce document qui en donne le « mode d'emploi ». Il peut être lu par un logiciel pour générer automatiquement un client au service Web qu'il décrit, faisant ainsi office de spécification technique.

Les deux spécifications que nous venons de décrire sont relativement complexes. Heureusement, il n'est nul besoin de les apprendre pour pouvoir les utiliser. Un ensemble d'outils performants permettent en effet de lire et d'écrire du SOAP et du WSDL à notre place. Ces outils ont le double avantage de faire gagner beaucoup de temps (il faut compter une heure pour lire ou écrire un service Web simple) et de garantir le respect des spécifications.

Les grands éditeurs d'environnements de développement Java proposent ce type d'outils intégrés dans leur IDE. C'est le cas notamment de Sun Java Studio Entreprise et d'Oracle JDeveloper, pour n'en citer que deux, téléchargeables gratuitement.

Ces outils présentent toutefois les inconvénients de nous lier à l'implémentation SOAP de l'éditeur et de n'être généralement pas Open Source. Un service Web développé avec les outils Sun, par exemple, ne peut être déployé que sur le serveur d'applications Sun Java System Application Server. De même, les outils Oracle utilisent nécessairement l'implémentation SOAP d'Oracle lorsqu'ils génèrent un client au service Web.

Pour réaliser des services Web de manière indépendante de l'éditeur, nous optons pour notre part pour des outils Open Source dans le cadre de l'application Tudu Lists. Nous verrons qu'ils sont également très efficaces pour réaliser un document WSDL ou échanger des objets en suivant la norme SOAP.

En résumé

Les services Web se fondent sur une multitude de normes complexes, dont nous venons de décrire les deux principales, SOAP et WSDL.

L'utilisation des services Web n'est pas évidente et entraîne des contraintes importantes en terme d'interopérabilité et de performance.

La section suivante se penche sur la mise en œuvre pratique d'un service Web dans le cadre de l'application Tudu Lists.

Tudu Lists : utilisation de services Web

Dans Tudu Lists, nous voulons présenter une partie de la couche métier en SOA, de manière que des outils externes puissent consulter des listes de todos ainsi que les tâches afférentes.

Nous créons pour cela un Bean Spring simple, `tudu.web.ws.impl.TuduListsWebServiceImpl`, chargé de présenter les deux méthodes suivantes :

- `getAllTodoLists()`, qui retournera un ensemble de Value Objects représentant des listes de todos.
- `getTodosByTodoList(String listId)`, qui représentera un ensemble de todos appartenant à une liste.

Ce Bean implémente l'interface `tudu.web.ws.TuduListsWebService`, dont voici le code :

```
package tudu.web.ws;

import tudu.web.ws.bean.WsTodo;
import tudu.web.ws.bean.WsTodoList;

public interface TuduListsWebService {

  WsTodoList[] getAllTodoLists();

  WsTodo[] getTodosByTodoList(String listId);
}
```

Pour ces deux méthodes, nous créons des Value Objects spécifiques, `WsTodo` et `WsTodoList`, car nous voulons éviter de présenter notre modèle de données à travers un service Web. Une autre raison à ce choix est qu'il aurait été difficile de modéliser un objet `TodoList` classique dans un service Web, car il possède une dépendance cyclique avec l'objet `User`.

Nous pouvons voir ainsi une première différence entre les services Web et une technologie de type RMI, puisqu'il s'agit de mapper des objets Java en XML. Or cela n'est pas toujours évident, puisque les langages XML sont par nature hiérarchiques et ne permettent donc pas facilement de représenter tous les types de relations pouvant lier les objets entre eux.

La création de ces objets spécifiques nous évite par ailleurs de devoir publier des champs de type `java.util.Date` dans le service Web. Ces champs sont effet problématiques, car ils peuvent ne pas être renseignés dans l'application. L'implémentation SOAP fournie par Microsoft avec .Net ne supporte pas les champs de type `Date` ayant une valeur nulle, par exemple.

Ce type d'incompatibilité complexifie considérablement l'utilisation des services Web, engendrant une deuxième différence de taille par rapport à la technologie RMI : des objets de base, comme `java.util.Date`, ou les structures de données du package `java.util`, comme les collections, ne sont que très mal supportés par les services Web. C'est également pour cette raison que le service Web que nous sommes en train de créer renvoie des tableaux d'objets et non une collection d'objets.

En résumé, nous avons vu que le mappage XML/objet n'était pas évident et que l'utilisation d'un service Web était très différente d'une technologie d'accès distant, comme le RMI. Le plus simple est donc de n'utiliser que des JavaBeans classiques et des tableaux d'objets.

Nous allons maintenant publier ce Bean à l'aide de deux frameworks Open Source populaires, XFire et Axis.

Publication d'un Bean avec XFire et Spring

XFire est un projet Open Source de la communauté Codehaus, disponible à l'adresse *http://xfire.codehaus.org/*.

Il représente une implémentation SOAP aussi performante que simple d'utilisation, qui propose de surcroît une bonne intégration à Spring. Son seul véritable défaut est sa jeunesse. En l'utilisant, nous prenons donc un risque plus important qu'avec des frameworks plus anciens, tels que Axis, que nous détaillons par la suite.

XFire utilise la réflexion pour générer à la volée un service Web. Il ne nécessite aucun outil spécifique pour cette génération, à la différence de la plupart des solutions concurrentes (nous en aurons un exemple avec Axis à la section suivante).

Nous commençons par ajouter notre Bean dans le fichier **WEB-INF/applicationContext.xml :**

```
<bean id="tuduListsWebService"
      class="tudu.web.ws.impl.TuduListsWebServiceImpl">

  <property name="todoListsManager">
    <ref bean="todoListsManager" />
  </property>
  <property name="userManager">
    <ref bean="userManager" />
  </property>
</bean>
```

Nous définissons ensuite un contexte Spring spécifique pour XFire. Nous ajoutons pour cela la configuration de XFire à celle de Spring et configurons un `DispatcherServlet` spécifique dans le fichier **WEB-INF/web.xml :**

```
( ... )
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/applicationContext*.xml;
               classpath:org/codehaus/xfire/spring/xfire.xml
  </param-value>
</context-param>
( ... )
<servlet>
  <servlet-name>xfire</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
</servlet>
( ... )
<servlet-mapping>
  <servlet-name>xfire</servlet-name>
  <url-pattern>/ws/xfire/*</url-pattern>
</servlet-mapping>
```

Cette configuration associe les URL commençant par `/ws/xfire` à une servlet Spring, laquelle redirige ensuite les requêtes envoyées à XFire. La configuration que nous avons chargée depuis le classpath est fournie dans le JAR de XFire afin de configurer un contexte Spring spécifique pour les Beans de XFire. Ce fichier de configuration ressemble à celui de Struts, que nous avons présenté au chapitre 6.

Ce contexte est configuré *via* un fichier nommé **xfire-servlet.xml,** qui se trouve dans le répertoire **WEB-INF :**

```
<beans>
  <bean name="/TuduListsWebService" parent="xfireServiceTemplate">
    <property name="serviceBean">
      <ref bean="tuduListsWebService" />
    </property>
    <property name="serviceInterface">
      <value>tudu.web.ws.TuduListsWebService</value>
    </property>
  </bean>

  <bean id="xfireServiceTemplate"
        class="org.codehaus.xfire.spring.remoting.XFireExporter"
        abstract="true">

    <property name="serviceFactory" ref="xfire.serviceFactory" />
    <property name="xfire" ref="xfire" />
    <property name="style" value="document" />
    <property name="use" value="literal" />
  </bean>
</beans>
```

Dans cet exemple, nous utilisons l'héritage de Beans afin de simplifier la création de nouveaux Beans sur le même modèle.

Le Bean parent, `xfireServiceTemplate`, permet de configurer le fonctionnement de XFire.

Les Beans enfants servent à exporter des Beans Spring sous forme de service Web. Nous devons pour cela référencer un service et l'interface qu'il implémente, de manière que XFire puisse utiliser la réflexion sur cette interface.

Il n'y a rien d'autre à réaliser pour publier un Bean Spring sous forme de service Web.

Il suffit de faire pointer un navigateur Internet sur l'URL suivante pour récupérer le fichier WSDL correspondant au service Web publié :

http://127.0.0.1:8080/Tudu/ws/xfire/TuduListsWebService?wsdl

Le service Web est quant à lui disponible à l'adresse :

http://127.0.0.1:8080/Tudu/ws/xfire/TuduListsWebService

Notons que les URL sont sensibles à la casse, d'où l'importance de respecter les majuscules, notamment pour la racine du contexte (ici *Tudu*).

Pour un utilisateur de Spring, XFire permet donc de réaliser extrêmement facilement des services Web.

Pour observer en détail les performances de ce service Web, nous pouvons utiliser JAMon, que nous avons déjà présenté au chapitre 9, consacré à la technologie AJAX. Ces tests fournissent des temps de réponse excellents, inférieurs à 30 millisecondes.

En conclusion, XFire représente une couche SOAP simple et performante, qui n'a pour seul défaut que son relatif manque de maturité.

Publication d'un Bean avec Axis et Spring

Axis est l'implémentation SOAP de la fondation Apache. Très largement utilisé, ce framework mature utilise des scripts Ant afin de générer du code source, un code qui doit ensuite être intégré à l'application.

Nous allons implémenter avec Axis le service Web que nous venons de créer avec XFire, afin d'en détailler les différences d'utilisation.

Il nous faut tout d'abord télécharger Axis sur le site Web d'Apache, à l'adresse *http://ws.apache.org/axis/*.

Nous utilisons ensuite Ant pour générer un fichier WSDL à partir de code Java. Il s'agit de produire par script ce que XFire génère automatiquement par réflexion.

La génération du fichier WSDL n'est toutefois qu'une première étape. Il faut ensuite générer à partir de ce fichier le service Web complet. Nous utilisons pour cela deux tâches Ant, de la manière suivante :

```
<target name="generate-web-service">
  <path id="axis.classpath">
    <pathelement path="${basedir}/classes"/>
    <fileset dir="${basedir}/lib/">
      <include name="**/*.jar" />
    </fileset>
  </path>
  <taskdef resource="axis-tasks.properties"
           classpathref="axis.classpath" />

  <axis-java2wsdl classname="tudu.web.ws.axis.AxisTuduLists"
                  output="${basedir}/AxisTuduListsWebService.wsdl"
                  namespace="http://axis.ws.web.tudu"
                  location="http://127.0.0.1:8080/Tudu/ws/axis/TuduListsWebService">

  </axis-java2wsdl>

  <axis-wsdl2java output="${basedir}/generated-sources"
                  testcase="false"
                  serverside="true"
                  url="${basedir}/AxisTuduListsWebService.wsdl"
                  verbose="true">

  </axis-wsdl2java>
</target>
```

Dans ce script, nous commençons par définir un classpath comprenant les classes de l'application (répertoire **classes**) et les bibliothèques utilisées par Axis (dans le répertoire **lib**). Les JAR d'Axis ajoutent à ce classpath un fichier **axis-tasks.properties,** lequel contient deux tâches Ant, la première pour générer un fichier WSDL depuis des classes Java **(java2wsdl),** et la seconde pour générer du code source Java depuis un fichier WSDL **(wsdl2java).**

Pour générer le service Web, nous créons une interface qui le représente, que nous appelons `tudu.web.ws.axis.AxisTuduLists`. Étant donné que nous voulons que cette interface présente les mêmes méthodes que le Bean `tuduListsWebService`, nous lui en faisons hériter :

```
package tudu.web.ws.axis;

public interface AxisTuduLists extends
  tudu.web.ws.TuduListsWebService {

}
```

Une fois cette interface compilée, nous exécutons le script Ant précédemment créé. Ce dernier génère dans un premier temps un fichier WSDL puis du code source Java fondé sur le framework Axis.

Ces classes générées peuvent ensuite être copiées dans l'arborescence du projet. Remarquons que les tâches Ant d'Axis remplacent alors un certain nombre de classes existantes :

- Les JavaBeans servant à échanger des données, `WsTodo` et `WsTodoList`, sont enrichis avec du code spécifique d'Axis. Ils restent cependant des JavaBeans normaux, qui peuvent être utilisés en parallèle par XFire.
- L'interface `AxisTuduLists`, que nous venons de créer, est complètement modifiée. Nous l'avions fait hériter de `TuduListsWebService` de manière que, par réflexion, Axis pût recréer cette interface avec ces mêmes méthodes. Les méthodes générées présentent cependant une différence, puisqu'elles lancent des exceptions de type `java.rmi.RemoteException`. De son côté, l'interface `AxisTuduLists` n'étend plus l'interface `TuduListsWebService` suite au changement de signature des méthodes.

Axis a ainsi modifié un certain nombre de classes mais a aussi généré un squelette de service Web, qu'il nous faut maintenant implémenter. Il s'agit d'écrire le code de la classe `TuduListsWebServiceSoapBindingImpl`, qui implémente `AxisTuduLists`, l'interface que nous venons d'étudier.

Cette implémentation va publier avec Axis les méthodes du Bean Spring `tuduListsWebService`. C'est d'ailleurs pour cela que nous avons fondé la génération du code sur une interface implémentant `TuduListsWebService`.

Pour cela, Spring fournit la classe `org.springframework.remoting.jaxrpc.ServletEndpointSupport`, qui permet l'intégration de Spring et d'Axis. En étendant cette classe, l'implé-

mentation que nous sommes en train d'écrire peut accéder au contexte Spring, et donc au Bean `tuduListsWebService` :

```
package tudu.web.ws.axis;

import org.springframework.remoting.jaxrpc.ServletEndpointSupport;

import tudu.web.ws.TuduListsWebService;

public class TuduListsWebServiceSoapBindingImpl
  extends ServletEndpointSupport
  implements tudu.web.ws.axis.AxisTuduLists {

  private TuduListsWebService service;

  public final void onInit() {
    this.service =
      (TuduListsWebService) getWebApplicationContext()
      .getBean("tuduListsWebService");
  }

  public tudu.web.ws.bean.WsTodoList[] getAllTodoLists()
    throws java.rmi.RemoteException {

    return service.getAllTodoLists();
  }

  public tudu.web.ws.bean.WsTodo[] getTodosByTodoList(String in0)
    throws java.rmi.RemoteException {

    return service.getTodosByTodoList(in0);
  }
}
```

Axis impose donc la génération d'un certain nombre de classes, qui doivent par la suite être retravaillées en fonction du service Web que nous souhaitons proposer. Ce mode de fonctionnement relativement lourd exige un coût supplémentaire lors de toute évolution du service Web, ainsi que lors d'une montée en version d'Axis. Il faut, dans ces deux cas, générer une nouvelle fois les classes et y répercuter à nouveau les développements précédemment effectués.

Pour cette raison, il est recommandé de mettre le moins de code possible dans la classe implémentant le service Web, comme nous venons de le faire en laissant le code métier à l'intérieur du Bean Spring `tuduListsWebService`.

Maintenant que le service Web est développé, il faut encore configurer le moteur d'Axis afin qu'il puisse publier ce service. De la même manière que pour XFire, il s'agit d'une servlet à configurer dans le fichier **WEB-INF/web.xml :**

```
<servlet>
  <servlet-name>axis</servlet-name>
  <servlet-class>
```

```
    org.apache.axis.transport.http.AxisServlet
  </servlet-class>
  <load-on-startup>10</load-on-startup>
</servlet>

( ... )

<servlet-mapping>
  <servlet-name>axis</servlet-name>
  <url-pattern>/ws/axis/*</url-pattern>
</servlet-mapping>
```

Cette servlet lit un fichier de configuration, qui, par défaut, se trouve à l'emplacement **WEB-INF/server-config.wsdd.** Ce fichier de configuration est fourni par Axis, mais nous pouvons également nous appuyer sur celui de Tudu Lists. C'est en éditant ce fichier que nous pouvons configurer Axis afin qu'il publie notre service Web.

Lors de l'exécution du script Ant, outre le code source Java généré, deux fichiers finissant en **.wsdd** sont créés. L'un de ces fichiers, **deploy.wsdd,** sert à déployer le service Web dans un serveur Axis.

La méthode la plus simple pour déployer ce service consiste à copier-coller le contenu de ce fichier dans le fichier **server-config.wsdd :**

```
<?xml version="1.0" encoding="UTF-8"?>
<deployment xmlns="http://xml.apache.org/axis/wsdd/"
  xmlns:wsdd="http://xml.apache.org/axis/wsdd/"
  xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">

  <globalConfiguration>
    ( ... )
  </globalConfiguration>

  <handler
        type="java:org.apache.axis.transport.local.LocalResponder"
        name="LocalResponder" />

  ( ... )
  <service provider="java:RPC" name="Version">
    <parameter value="getVersion" name="allowedMethods" />
    <parameter value="org.apache.axis.Version" name="className" />
  </service>

  <service name="TuduListsWebService"
           provider="java:RPC"
           style="rpc"
           use="encoded">

    ( ... )
```

```
  </service>

  <transport name="http">
    ( ... )
  </transport>
</deployment>
```

Une fois l'application Web configurée, le service Web Axis peut être publié dans l'application. D'une manière très similaire à ce que nous avons vu avec XFire, nous pouvons accéder au WSDL de ce service Web en utilisant l'URL suivante :

http://127.0.0.1:8080/Tudu/ws/axis/TuduListsWebService?wsdl

Le service Web est quant à lui disponible à l'adresse :

http://127.0.0.1:8080/Tudu/ws/axis/TuduListsWebService

Le service Web ainsi généré fonctionne de manière différente de celui publié par XFire, et ce pour deux raisons : les paramètres utilisés pour créer ce service Web ne sont pas les mêmes dans les deux implémentations, et ces deux moteurs n'interprètent pas toujours les spécifications SOAP et WSDL de la même manière.

Au final, l'utilisation d'Axis se révèle relativement lourde. La génération de code et l'utilisation du fichier **server-config.wsdd** sont fastidieuses, tandis que les performances finales du service Web se révèlent plus que moyennes, de l'ordre de 50 millisecondes dans nos tests. (Nous préférons établir notre propre mesure de performance, en fonction de notre configuration matérielle et logicielle, en utilisant nous-même JAMon, comme nous l'avons vu précédemment pour XFire.) Cependant, Axis étant une implémentation particulièrement répandue et stable, le service Web ainsi généré garantit une excellente interopérabilité, et ce quel que soit le client utilisé pour y accéder.

Enfin, Axis s'intègre avec Spring de manière assez simple, ce qui lui permet d'offrir un accès efficace à la couche métier de l'application.

Utilisation d'un service Web avec Axis, sans Spring

Afin d'illustrer le fonctionnement général d'un service Web, nous allons détailler le développement d'une application Swing, nommée Tudu Lists Client, capable de se connecter à Tudu Lists *via* le service Web Axis que nous venons de développer.

Dans cette section, nous nous concentrons uniquement sur l'utilisation d'Axis.

Lorsqu'elle est utilisée côté client, cette bibliothèque présente de nombreux atouts, notamment les suivants :

- Axis est fiable, respecte les standards et ne nécessite pas de version élevée du JDK (version 1.3 au minimum), ce qui est important pour une application cliente.
- Les problèmes de performance d'Axis sont nettement moins importants côté client, car il y a moins d'appels que côté serveur.
- L'API d'Axis est particulièrement simple d'utilisation.

Nous allons voir dans l'exemple suivant comment utiliser une application Spring en mode SOA. L'application Swing n'est ici qu'une couche graphique, qui utilise les méthodes métier que publie Tudu Lists.

L'installation de Tudu Lists Client a été détaillée au chapitre 1. Une fois configuré dans Eclipse, le projet Tudu Lists Client doit pouvoir être lancé en faisant un clic droit sur la classe `tudu.client.TuduClient` et en sélectionnant Run as puis Java Application dans le menu contextuel.

Pour écrire cette application, nous commençons par reprendre les classes générées par les tâches Ant d'Axis, que nous avons déjà utilisées pour créer le service Web.

Parmi ces classes se trouve un objet permettant d'accéder au service Web, en l'occurrence `AxisTuduListsServiceLocator`, qui s'utilise de la manière suivante :

```
AxisTuduListsServiceLocator locator =
  new AxisTuduListsServiceLocator();

client = locator.getTuduListsWebService(url);
WsTodoList[] wsTodoLists = client.getAllTodoLists();
```

Ce code permet d'accéder à un service Web localisé à l'URL passée en paramètre et d'en appeler directement les méthodes.

Dans l'exemple suivant, nous récupérons une liste de todos en utilisant l'objet `client` que nous venons de créer :

```
String listId =
  wsTodoLists[todoLists.getSelectedIndex()].getListId();

WsTodo[] wsTodos = client.getTodosByTodoList(listId);
for (int i = 0; i < wsTodos.length; i++) {
  Object[] row = {wsTodos[i].getDescription(),
    wsTodos[i].getPriority(), wsTodos[i].isCompleted()};

  todosTableModel.addRow(row);
}
```

L'accès aux différentes méthodes du service Web se fait donc simplement en Java, et nous pouvons ainsi utiliser à distance la couche métier de Tudu Lists pour afficher les listes de todos et les todos qui leur sont associés.

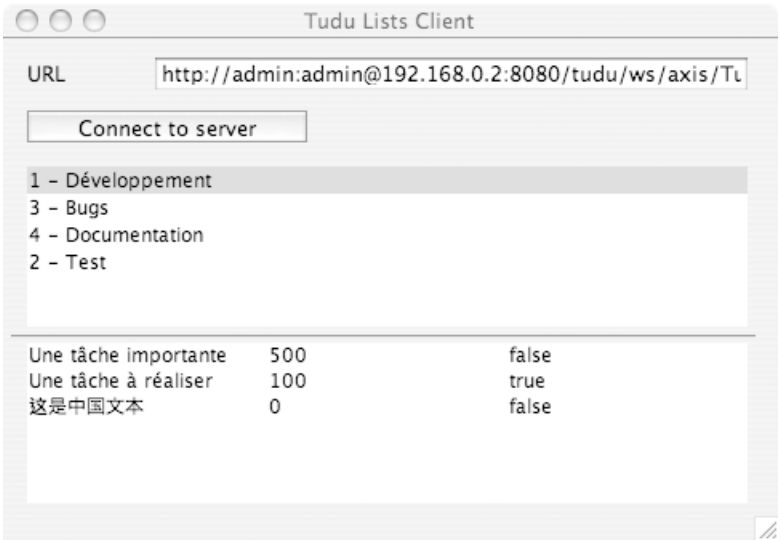
La figure 14.2 illustre une copie d'écran de l'application Tudu Lists Client, alors qu'elle accède à Tudu Lists. Remarquons en haut de la figure l'URL de l'application fournissant le service Web, ainsi que, en dessous, deux tableaux : celui du haut affiche les listes de todos, et celui du bas les tâches associées à la liste sélectionnée.

La partie Swing de cette application est simple à développer à l'aide d'un éditeur graphique, tel que Oracle JDeveloper ou Netbeans 5.0.

Dans la mesure où cette application Swing permet de présenter des informations provenant de Tudu Lists, elle peut être considérée comme un concurrent des méthodes de

présentation présentées précédemment dans l'ouvrage, qu'elles soient fondées sur Struts, Spring MVC ou AJAX.

Figure 14.2
Application Swing se connectant à Tudu Lists



Utilisation d'un service Web avec Axis et Spring

Dans l'exemple précédent, nous n'avons pas utilisé Spring, puisque, pour une couche de présentation simple, sans couche de persistance ou métier, cela ne présentait guère d'intérêt. Par ailleurs, cela nous a permis de créer cette application très rapidement à l'aide d'un IDE.

Il est cependant tout à fait possible d'utiliser un service Web à l'intérieur d'une application d'entreprise plus complexe. Dans ce cas, Spring propose un système simplifiant grandement l'accès aux services Web, grâce à la classe `org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean`. Cette classe permet de transformer un service Web en un Bean Spring classique, ne lançant même pas d'exceptions de type `java.rmi.RemoteException`.

En voici un exemple d'utilisation simple :

```
<bean id="tuduListsServiceLocator"
      class=" tudu.web.ws.axis.AxisTuduListsServiceLocator"/>

<bean id="tuduListsService"
      class="org.springframework.remoting.jaxrpc.
                                        JaxRpcPortProxyFactoryBean">

  <property name="jaxRpcService">
    <ref bean="tuduListsServiceLocator"/>
  </property>
```

```
    <property name="portName">
      <value>TuduListsWebService</value>
    </property>
    <property name="serviceInterface">
      <value>tudu.web.ws.axis.AxisTuduLists</value>
    </property>
  </bean>
```

Nous utilisons ici un Bean nommé `tuduListsServiceLocator`, dont l'implémentation a été générée par les tâches Ant d'Axis, et que nous avons déjà utilisé dans l'exemple Tudu Lists Client.

Ce Bean est utilisé par un deuxième Bean, nommé `tuduListsService`, qui représente notre service Web. Ce Bean possède deux autres propriétés :

- `portName`, qui est le nom du port du service Web défini dans le WSDL.
- `serviceInterface`, qui est l'interface du service Web que nous avons utilisée depuis le début de cette section.

Il aurait été tout aussi possible de créer ce Bean sans utiliser de Bean `tuduListsServiceLocator` en passant en paramètre au Bean `tuduListsService` les propriétés supplémentaires suivantes :

- `wsdlDocumentUrl`, qui est l'URL du WSDL utilisé (correspondant à `"http://127.0.0.1:8080/Tudu/ws/axis/TuduListsWebService?wsdl"` dans notre exemple).
- `namespaceUri`, qui est l'espace de nommage utilisé pour le service Web (`http://axis.ws.web.tudu` dans notre exemple).
- `serviceName`, qui est le nom du service Web (`"AxisTuduListsService"` dans notre exemple).

Nous conseillons cependant l'utilisation de la première configuration, avec le Bean `tuduListsServiceLocator`, car ce dernier est entièrement généré par les tâches Ant d'Axis, réduisant ainsi la complexité de la configuration.

Le Bean que nous venons de configurer, `tuduListsService`, peut maintenant être utilisé à l'intérieur de l'application, comme n'importe quel autre Bean, mais à une différence près : l'interface utilisée, `tudu.web.ws.axis.AxisTuduLists`, lance des exceptions de type `java.rmi.RemoteException`. Dans la mesure où toute méthode présentée par un service Web lance ce type d'exception, c'est donc parfaitement normal.

Nous pouvons toutefois nous interroger sur le bien-fondé de cette pratique. Du fait de cette exception à traiter, le reste de l'application sait qu'il accède à un service distant, chose anormale dans une application découpée en couches distinctes. Si, par exemple, le service métier auquel nous accédons aujourd'hui *via* un service Web doit être accessible demain *via* un EJB local, il sera nécessaire de recoder les couches supérieures de l'application afin de ne plus traiter les `RemoteException`.

Spring permet de résoudre ce problème et d'utiliser une interface métier normale, ne lançant pas de `RemoteException`, pour accéder à un service Web. Pour cela, la classe Spring

que nous venons d'utiliser (`JaxRpcPortProxyFactoryBean`) est capable de transformer automatiquement les exceptions de type `java.rmi.RemoteException` en des exceptions de type `org.springframework.remoting.RemoteAccessException`, qui sont des `RuntimeException` n'ayant pas à être traitées.

Pour utiliser cette fonctionnalité, il faut créer une interface identique à celle du service Web, mais ne lançant pas de `RemoteException`, et l'utiliser à la place de l'autre interface dans la configuration du Bean, comme ci-dessous :

```
<bean id="tuduListsService"
      class="org.springframework.remoting.jaxrpc.
                                        JaxRpcPortProxyFactoryBean">

  <property name="jaxRpcService">
    <ref bean="tuduListsServiceLocator"/>
  </property>
  <property name="portName">
    <value>TuduListsWebService</value>
  </property>
  <property name="serviceInterface">
    <value>tudu.web.ws.TuduListsWebService</value>
  </property>
</bean>
```

Nous utilisons ici l'interface que nous avons définie en début de chapitre pour l'implémentation du service Web avec XFire. Nous pouvons ainsi retrouver la même interface métier côté client et côté serveur.

Ce terme d'« interface métier » est particulièrement important. Il s'agit réellement d'un contrat permettant à un système d'utiliser des objets métier présents dans autre système, et ce sans que ce contrat soit pollué par des contraintes techniques (telles que les exceptions de type `java.rmi.RemoteException`).

Analyse des échanges SOAP

Nous venons de détailler deux types de services Web, XFire et Axis, ainsi que la manière d'accéder à ces services, avec ou sans Spring. Il est probable qu'à ce niveau du développement nous rencontrions des problèmes pour nous connecter à un service Web. Nous allons aussi probablement vouloir étudier plus en détail les communications entre notre service Web et ses clients à des fins de débogage, d'optimisation ou de calcul du trafic réseau.

Il existe pour cela un outil simple, nommé tcpmon, disponible à l'URL *https://tcpmon.dev.java.net.*

tcpmon permet de capturer les échanges réseau entre un client et un serveur, en se positionnant en tant que serveur intermédiaire. Dans notre exemple, il s'agit de configurer tcpmon pour se connecter au serveur (Server Name) **127.0.0.1**, sur le port (Server Port) **8080**. tcpmon redirige alors les requêtes et les réponses échangées avec ce serveur sur le port (Local Port) **8081**.

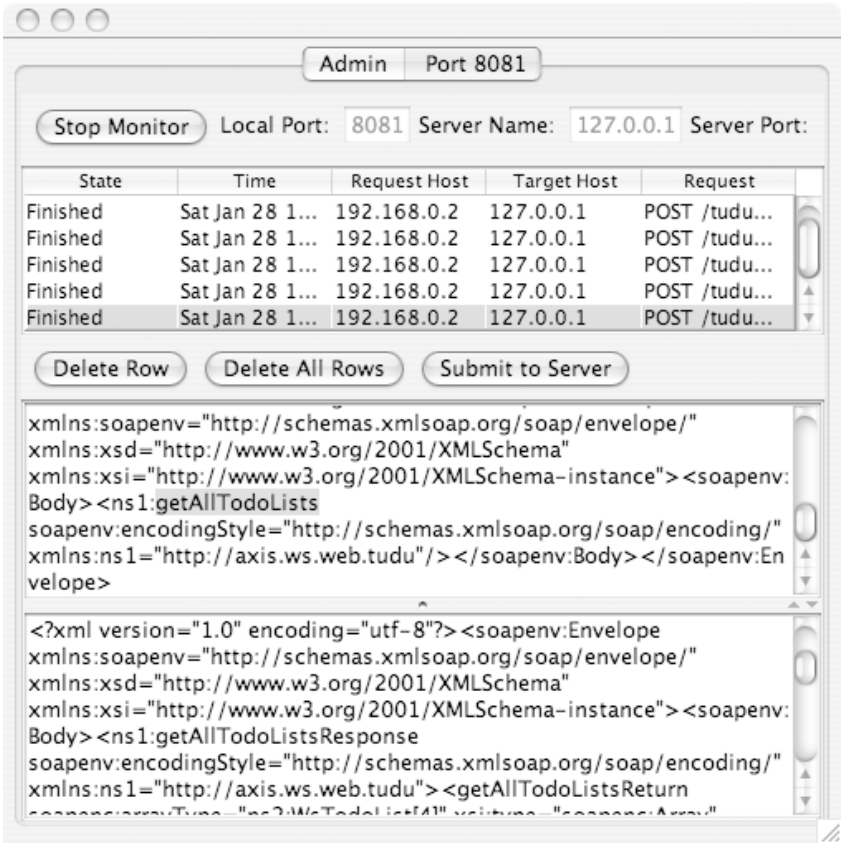
Ainsi, pour se connecter au service Web que nous avons développé, nous utilisons le port **8081** pour passer par tcpmon :

http://127.0.0.1:8081/Tudu/ws/axis/TuduListsWebService

Nous pouvons de cette manière étudier les échanges SOAP entre Tudu Lists et le client Swing.

La figure 14.3 illustre une analyse par tcpmon des échanges entre en client et un serveur SOAP, en l'occurrence l'application Tudu Lists Client accédant à un serveur Tudu Lists et effectuant une demande d'envoi de l'ensemble des listes de todos grâce à la méthode `getAllTodoLists`.

Figure 14.3
Utilisation de tcpmon pour étudier les flux SOAP



Ce logiciel présente, dans sa zone de texte supérieure, la requête envoyée, et, dans sa zone de texte inférieure, la réponse retournée par le serveur. Le format SOAP étant assez facilement compréhensible, cette méthode permet de comprendre les problèmes d'interopérabilité que nous ne manquons pas de rencontrer entre un client et un serveur codés avec des frameworks différents.

Conclusion

Ce chapitre a montré comment publier des Beans Spring grâce aux technologies XML disponibles et créer ainsi des applications en mode SOA.

Nous avons vu qu'il était relativement simple de développer nous-même notre propre format XML, avec JDOM, ou d'utiliser le format RSS.

Nous avons également détaillé deux implémentations Open Source nous permettant de réaliser des services Web :

- XFire, un framework très bien intégré à Spring, rapide d'utilisation, performant et efficace, mais manquant de maturité et de reconnaissance.
- Axis, un framework assez bien intégré à Spring, robuste, fiable et très largement utilisé, mais souffrant de certaines lourdeurs, qui affectent le développement, et de lenteur lors de son exécution.

Tout au long de ce chapitre, nous avons vu que l'utilisation d'un format XML était très différente de l'utilisation directe d'un objet Java. De nombreux problèmes de performance et de mappage objet/XML limitent l'utilisation de cette solution au strict nécessaire.

N'oublions pas la première règle concernant les objets distribués énoncée par Martin Fowler : « N'utilisez pas d'objets distribués ! », puisqu'il s'agit d'une des premières causes de lenteur des applications.

Pour une application Spring typique, et donc proprement découpée en couches, l'utilisation d'une couche de publication d'objets en XML représente réellement une couche de présentation spécifique, dont le rôle est la simple publication d'objets métier. Il est possible d'ajouter des services Web à une application existante et de les proposer en complément d'autres méthodes : accès local aux objets et RMI en particulier. Cela permet d'utiliser XML et les services Web de façon raisonnée, en n'utilisant ces derniers que lorsqu'ils sont réellement nécessaires.

Nous avons volontairement passé sous silence l'aspect sécurité, car il fait l'objet du chapitre suivant, dans lequel nous montrons comment utiliser Acegi Security pour ajouter une authentification HTTP aux services Web que nous venons de développer.

15

La sécurité avec Acegi Security

Sur Internet, mais aussi au cœur même d'une entreprise, le risque de compromission de données sensibles peut conduire à des catastrophes en termes économiques ou légaux. La sécurité est donc un aspect important du développement d'une application. Sujet complexe, s'il en est, elle ne doit pas être traitée à la légère. Pour ces raisons, ce chapitre commence par rappeler les besoins couramment exprimés en matière de sécurité, ainsi que les concepts clés généralement utilisés dans ce domaine.

En Java, nous disposons de JAAS (Java Authentication and Authorization Service) et de la spécification J2EE pour nous aider dans cette tâche. Il s'agit de standards largement utilisés, qu'il est important de connaître, puisqu'un grand nombre de solutions s'appuient dessus. D'une manière plus générale, leur maîtrise est un prérequis à l'utilisation de frameworks plus complexes. Nous verrons les limitations de ces deux spécifications et les raisons pour lesquelles elles ne sont pas suffisantes pour développer des applications d'entreprise un tant soit peu complexes.

Acegi Security est un projet Open Source qui propose une solution de sécurité intégrée complète aux systèmes utilisant Spring. Très largement utilisée au sein de la communauté Spring, elle peut *de facto* être considérée comme un standard. Nous détaillons dans ce chapitre les principes de fonctionnement et l'API d'Acegi Security et insistons sur les avantages qu'elle procure par rapport aux autres solutions.

En suivant notre étude de cas, nous verrons de quelle manière utiliser Acegi Security afin d'obtenir une solution de sécurité simple mais suffisante dans la majorité des cas pour gérer un formulaire de login, sécuriser des services Web et gérer des droits d'utilisateur et des login automatisés en fonction d'un cookie.

La sécurité dans les applications Web

Cette section rappelle les besoins habituellement exprimés pour la sécurisation d'une application, ainsi que les concepts utilisés pour répondre à ces besoins. Nous pourrons de la sorte mieux comprendre les spécificités fonctionnelles et techniques induites par la sécurité.

Nous nous intéresserons en particulier aux applications d'entreprise, et plus spécifiquement aux applications Web fondées sur J2EE, qui sont la cible même de cet ouvrage.

Les besoins

La grande majorité des applications ont les mêmes besoins en terme de sécurité. C'est d'ailleurs pour cette raison que la création d'un mécanisme spécifique de gestion de la sécurité est une aberration et que l'utilisation d'un framework préexistant est largement préférable.

Gestion des utilisateurs

Les utilisateurs, leur mot de passe et leurs droits doivent être gérés dans un système spécialisé. Dans les entreprises, il est courant que ce système soit un référentiel centralisé, tel un annuaire LDAP. Il s'agit d'une sorte de base de données optimisée pour les requêtes en lecture et possédant une structure arborescente permettant de stocker des hiérarchies. Nous rencontrons aussi des systèmes plus simples, reposant sur des bases de données relationnelles. Nous abordons ces dernières en détail dans notre étude de cas.

Ces systèmes ont, de préférence, les caractéristiques suivantes :

- Gestion d'attributs liés à l'utilisateur (nom, numéro de téléphone, etc.).
- Vérification du mot de passe. Il est préférable qu'un système externe gère les mots de passe, plutôt que chaque application utilisatrice.
- Gestion des droits des utilisateurs.
- Gestion des groupes d'utilisateurs, qui facilite la gestion des droits.

Il va de soi que les entreprises ne fournissent pas toujours un système de gestion des droits aussi complet. Les développeurs doivent régulièrement faire face à des situations dégradées, dans lesquelles certains de ces besoins ne peuvent être pris en compte ou doivent être développés spécifiquement.

Sécurisation des URL

La sécurisation la plus simple à mettre en place est celle des requêtes HTTP. Nous décidons de ne permettre qu'à un certain type d'utilisateur d'accéder à une URL donnée. Dans Tudu Lists, par exemple, les URL se terminant par `/secure/**` sont réservées aux utilisateurs authentifiés et possédant le rôle `ROLE_USER`.

Ce type de sécurisation peut être mis en place aux niveaux suivants :

- Réseau, avec utilisation d'un répartiteur de charge (de type Alteon).
- Serveur Web, avec, par exemple, utilisation de la sécurité dans Apache.
- Serveur J2EE, que nous détaillons plus en détail par la suite, car il propose une intégration bien plus avancée de la sécurité au sein de l'application développée.

Notons que le principal défaut de cette solution est qu'il est nécessaire de disposer d'une bonne stratégie dans le nommage des URL. Cette politique contraignante est difficilement imposable à une application existante. C'est pourquoi elle doit être mise en place dès la conception de l'application.

Sécurisation de la couche de service

La couche de service n'est normalement pas accessible aux utilisateurs finals de l'application. Cependant, le fait de sécuriser des EJB Session ou des Beans Spring présente un double intérêt :

- Cela renforce la sécurité de l'application en empêchant un utilisateur malveillant d'y avoir accès. Si un service est sensible et ne doit être utilisé que par l'administrateur, mieux vaut valider qu'il ne peut être exécuté que par des personnes ayant le rôle adéquat.
- Cela permet l'accès distant à ces objets. Il est ainsi possible de permettre à des applications externes d'utiliser un service sensible, à condition que leurs utilisateurs soient dûment autorisés à y accéder.

Cette sécurisation peut se faire au niveau d'une classe ou d'une méthode.

Sécurisation des objets de domaine

Plus rarement exprimé, ce besoin consiste à sécuriser certaines instances d'objets métier. Dans l'exemple de Tudu Lists, un utilisateur a le droit d'effacer des todos mais ne doit pouvoir effacer que les todos qui lui appartiennent. Il a donc le droit d'exécuter la méthode de suppression des données (sécurisation de la couche de service), mais pas sur l'ensemble des objets métier.

Ce besoin se fait sentir dans la plupart des applications multiutilisateurs. Dans une boutique en ligne, par exemple, un utilisateur n'a le droit de gérer que son panier d'achat.

Nous verrons cependant dans la suite de ce chapitre que cette sécurisation est généralement un luxe inutile.

Contrôle du code exécuté

Ce besoin, qui se rencontre également rarement dans le cadre d'une application Web J2EE, consiste à n'autoriser l'exécution que de code qui a été validé. L'exemple le plus courant est celui des applets. Dans la mesure où nous exécutons du code téléchargé depuis Internet, nous ne pouvons avoir une confiance absolue dans ce programme.

Dans le cadre d'une application Java/J2EE s'exécutant sur un serveur d'entreprise, il n'est pas utile de prendre en compte ce type de considération. Si une personne malveillante accède au serveur et est capable d'intervenir directement sur du code, elle sera également capable d'en modifier les règles de sécurité. Mieux vaut donc mettre l'accent sur la sécurisation des serveurs (et de leurs locaux) et éviter ce type de complexité aux développeurs d'applications.

Rappel des principales notions de sécurité

La sécurité s'appuie sur deux notions principales, l'authentification des utilisateurs et la gestion de leurs autorisations, auxquelles se greffe un vocabulaire spécialisé, que nous allons rappeler :

- **Authentification.** Vérification qu'un utilisateur est bien la personne qu'il prétend être. Cette vérification s'effectue généralement à l'aide d'un couple identifiant/mot de passe.
- **Autorisation.** Vérification qu'un utilisateur authentifié a la permission de réaliser une action. Un utilisateur possède généralement un ensemble d'autorisations, qui peuvent être gérées par groupes pour plus de facilité.
- **Les objets `Subject` et `Principal`.** Spécifiques de Java, ces objets se retrouvent dans l'ensemble des implémentations que nous allons étudier par la suite. Un `Subject` représente un utilisateur, tel qu'il est vu par l'application en cours. Ce `Subject` peut posséder plusieurs `Principal`, chacun de ces objets étant une représentation de cette personne. Login, numéro de Sécurité sociale et adresse e-mail peuvent être autant de `Principal` d'un même `Subject`.
- **Ressources et permissions.** Une ressource représente une entité protégée. Il peut s'agir d'un fichier, d'une URL ou d'un objet. Une permission correspond au droit d'accéder à cette ressource. Pour qu'un utilisateur accède à une URL protégée, il faut que ses autorisations lui donnent la permission d'y accéder.

La sécurité Java

Java propose deux API pour gérer la sécurité : JAAS, pour les projets Java standards (J2SE), et une API spécifique incluse dans la norme J2EE.

Nous allons voir que ces API ne répondent que de manière fragmentaire aux besoins que nous avons rappelés précédemment. Il est cependant essentiel de les connaître, car elles composent le socle technique standard de J2EE.

JAAS

JAAS (Java Authentication and Authorization Service) a été intégré à J2SE 1.4, après avoir été un package optionnel.

Il s'agit d'une API de bas niveau, permettant en particulier de gérer les privilèges du code qui s'exécute. Peu adaptée à une application Java/J2EE, elle s'adresse à un domaine différent : les applets et les applications graphiques autonomes fondées sur AWT ou Swing.

La gestion des utilisateurs en base de données ou la création de formulaires Web de login sont des sujets bien éloignés de cette spécification, qui n'est pas conçue pour cela. Par ailleurs, les fonctionnalités qu'elle propose en matière de gestion des privilèges d'exécution du code ne correspondent pas à une application Web classique.

Nous ne nous attarderons donc pas sur cette spécification et nous pencherons plutôt sur les extensions spécifiques à J2EE.

La spécification J2EE

La spécification J2EE inclut plusieurs objets et méthodes dédiés à la sécurité. Généralement simples, ces derniers ont l'avantage d'être bien intégrés dans les frameworks existants. Struts, par exemple, utilise cette API.

Si cette spécification pose un certain nombre de problèmes, que nous allons détailler, elle n'en présente pas moins le grand avantage d'être un standard officiel.

Les servlets

La spécification servlet permet de définir des règles de sécurité au niveau du fichier **WEB-INF/web.xml** afin de protéger des URL en fonction de leur nom. Elle supporte des méthodes d'authentification simples (par formulaire, authentification HTTP basique ou certificat) et fournit une API rudimentaire. Cette API se trouve essentiellement dans deux méthodes de l'objet `javax.servlet.http.HttpServletRequest` :

- `getRemoteUser()`, qui permet d'obtenir le login de l'utilisateur en cours sous forme de chaîne, à charge pour le développeur d'appeler la couche de service de son application pour retrouver un objet représentant l'utilisateur en fonction de ce login.
- `isUserInRole(String role)`, qui vérifie si l'utilisateur en cours possède l'autorisation demandée. Il est ainsi facile de vérifier si l'utilisateur a le rôle « administrateur » et de lui afficher un écran spécifique d'administration.

Ces deux méthodes aussi simples qu'efficaces sont suffisantes pour des applications Web peu complexes.

La spécification dans son ensemble souffre toutefois d'un grand nombre d'inconvénients :

- Elle n'est pas portable d'un serveur d'applications à un autre, chaque serveur J2EE en possédant sa propre implémentation. Dans le meilleur des cas, migrer vers un nouveau serveur peut simplement consister en une configuration spécifique de ce module, mais cela peut être nettement plus compliqué.

- La gestion des URL dans le fichier **WEB-INF/web.xml** est rudimentaire. Il n'est pas possible, par exemple, d'utiliser des expressions régulières pour définir une URL.
- Les services fournis sont généralement élémentaires. Il n'y a pas de service d'authentification automatique par cookie, par exemple, ni de système pour empêcher deux utilisateurs d'utiliser le même login en même temps, etc.
- Cette spécification reposant sur l'objet `HttpServletRequest`, elle requiert la présence de cet objet et n'est donc utilisable qu'au plus près de la couche de présentation. Elle pose en outre des problèmes en matière de tests unitaires puisqu'il faut simuler cet objet.

Les EJB

Les EJB mettent en œuvre une API similaire à celle des servlets et proposent ainsi une sécurité au niveau des classes et des méthodes de la couche de service. L'intérêt de cette sécurité est qu'elle se propage depuis l'application Web (ou le client Swing) à l'ensemble des EJB utilisés.

Le fait de pouvoir sécuriser des EJB au niveau de leur classe comme de leurs méthodes, *via* le fichier **ejb-jar.xml,** offre une grande finesse à la politique de sécurité. Le nom de l'utilisateur et ses rôles peuvent être utilisés de la même manière que pour les servlets, *via* les méthodes `getCallerPrincipal()` et `isCallerInRole()` de l'objet `javax.ejb.EJBContext`.

Cette API étant proche de l'API servlet, elle en conserve le vice originel, et les applications ainsi développées sont généralement difficiles à porter d'un serveur d'applications à un autre.

Utilisation d'Acegi Security

Acegi Security est un framework Open Source dont l'objectif est de proposer un système complet de gestion de la sécurité. Il peut être utilisé de manière indépendante mais fonctionne de manière optimale avec Spring. Il s'agit d'ailleurs d'une solution très répandue au sein de la communauté Spring.

Cette section présente les avantages fournis par Acegi Security par rapport à une solution J2EE classique. Nous verrons aussi comment l'installer et le configurer.

Acegi Security est disponible à l'adresse *http://acegisecurity.sourceforge.net/.*

Principaux avantages

Le premier avantage d'Acegi Security est sa portabilité. Ne dépendant pas d'un serveur d'applications particulier, son utilisation est identique quel que soit le serveur utilisé. C'est grâce à cela que Tudu Lists peut être utilisé sans reconfiguration sur Tomcat, JBoss, Geronimo ou WebLogic.

Cette portabilité est particulièrement importante si l'application développée doit pouvoir être vendue à un grand nombre de clients possédant des systèmes hétérogènes. Dans le cadre d'une application développée en interne, il n'en reste pas moins dommage de se trouver bloqué sur un serveur d'applications uniquement à cause d'une problématique de sécurité.

Le deuxième avantage d'Acegi Security est qu'il fournit en standard un nombre de fonctionnalités beaucoup plus important qu'un serveur J2EE classique. Parmi les plus simples, et qui manquent cruellement dans la spécification J2EE, citons l'authentification automatique par cookie pour un nombre donné de jours, ainsi que la vérification qu'un utilisateur n'est pas déjà authentifié avec le login demandé.

Acegi Security propose en outre des fonctionnalités avancées, telles que le Single Sign-On (une authentification unique pour l'ensemble des applications de l'entreprise) ou la sécurisation des objets de domaine, fournissant ainsi une aide considérable au développement d'applications ayant des besoins complexes en matière de sécurité.

Enfin, Acegi Security propose une excellente intégration avec les applications Web et les Beans Spring. Concernant les applications Web, il propose des filtres de servlets bien plus fins que ceux proposés par la norme J2EE. Pour les Beans Spring, il permet d'utiliser la POA afin de sécuriser la couche métier de manière efficace et transparente.

Si nous reprenons les besoins de sécurité rappelés en début de chapitre, Acegi Security apporte les avantages suivants :

- Gestion des utilisateurs : solution intégrée, complète et portable.
- Sécurisation des requêtes HTTP : disponible de manière plus fine que dans la norme J2EE.
- Sécurisation de la couche de service : API complète, qui s'intègre dans les frameworks courants de POA (AOP Alliance, une API implémentée par Spring AOP, et AspectJ).
- Sécurisation de la couche de domaine : listes de contrôle d'accès, qui peuvent être spécifiées au niveau de chaque objet de domaine.
- Contrôle du code exécuté : non pris en compte par Acegi Security. Comme nous l'avons vu, cette fonction est rarement utile dans une application d'entreprise.

Installation

Acegi Security se compose d'un fichier JAR principal et nécessite la présence d'un certain nombre de dépendances. Ces différents JAR sont fournis dans l'application exemple livrée avec Acegi Security, ainsi que dans Tudu Lists. Acegi Security utilisant Maven, une bonne manière d'étudier ces JAR est de passer en revue le rapport Maven de gestion des dépendances, disponible dans la documentation du framework.

Une fois ces bibliothèques déposées dans le répertoire **WEB-INF/lib,** la configuration d'Acegi Security s'effectue *via* le descripteur de déploiement et la configuration Spring.

Dans le fichier **WEB-INF/web.xml,** nous définissons un filtre de servlets, qui sera responsable de la sécurité de l'application (Acegi Security propose en interne des filtres bien plus fins que ceux proposés par la spécification J2EE) :

```
<filter>
  <filter-name>Acegi Security Filter</filter-name>
  <filter-class>
    org.acegisecurity.util.FilterToBeanProxy
  </filter-class>
  <init-param>
    <param-name>targetClass</param-name>
    <param-value>
      org.acegisecurity.util.FilterChainProxy
    </param-value>
  </init-param>
</filter>

( ... )

<filter-mapping>
  <filter-name>Acegi Security Filter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

Cette configuration redirige l'ensemble des requêtes HTTP vers Acegi Security.

Contexte de sécurité et filtres

Acegi Security fournit un certain nombre de Beans Spring, qu'il s'agit de relier entre eux *via* une configuration Spring classique.

Dans Tudu Lists, nous avons opté pour un fichier séparé du reste de la configuration Spring **(WEB-INF/applicationContext-security.xml).** Nous recommandons cette pratique, car ce fichier va avoir une taille assez conséquente.

À la base du fichier de configuration se trouve la classe donnée en paramètre au filtre défini précédemment, `org.acegisecurity.util.FilterChainProxy`. Cette classe a pour fonction de traiter les requêtes HTTP avec des filtres fournis par Acegi Security. Il s'agit d'un filtre HTTP unique, défini dans le descripteur de déploiement de l'application Web, qui renvoie vers des filtres spécifiques traités en interne par Acegi Security.

Cette manœuvre permet de simplifier la configuration du descripteur de déploiement en réalisant la vraie configuration des filtres de sécurité dans le contexte Spring. Cela offre une configuration plus fine qu'avec le standard J2EE.

Cette classe permet de lier les filtres de sécurité de la manière suivante (tirée de Tudu Lists) :

```
<bean id="filterChainProxy"
      class="org.acegisecurity.util.FilterChainProxy">
```

```
<property name="filterInvocationDefinitionSource">
  <value>
    CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
    PATTERN_TYPE_APACHE_ANT
    /secure/**=httpSessionContextIntegrationFilter,
              contextHolderAwareRequestFilter,
              rememberMeProcessingFilter,
              securityEnforcementFilter
    /j_acegi_security_check*=
              httpSessionContextIntegrationFilter,
              authenticationProcessingFilter
    /welcome.action=
              httpSessionContextIntegrationFilter,
              contextHolderAwareRequestFilter,
              rememberMeProcessingFilter
      </value>
  </property>
  </bean>
```

Pour des raisons de lisibilité, nous avons ajouté des retours chariot, mais chaque liste de filtres, séparés par des virgules, vient normalement sur une seule ligne.

Nous voyons dans cet exemple que les URL vont être validées suivant leur nom en minuscule (premier paramètre) et avec une expression régulière de validation de type Ant (deuxième paramètre). Ant est un outil de construction de code très populaire (disponible à l'adresse *http://ant.apache.org/),* dont les motifs de validation sont plus efficaces que ceux du standard J2EE.

Nous pouvons ainsi valider des URL telles que `/secure/catalogue/admin` et `/secure/paiement/admin` en utilisant le motif `"/secure/**/admin/"`, ce qui ne serait pas possible en mode standard.

Dans l'exemple, nous voyons trois URL, auxquelles sont appliqués un certain nombre de filtres. Ces filtres ont un ordre défini, qu'il est important de respecter.

Voici l'ordre d'utilisation des filtres fournis par Acegi Security :

1. `ChannelProcessingFilter` : permet de rediriger l'URL vers un autre protocole (essentiellement de HTTP vers HTTPS).
2. `ConcurrentSessionFilter` : vérifie si l'utilisateur en cours n'a pas déjà une session ouverte, et, si oui, bloque la requête afin d'éviter d'avoir deux utilisateurs authentifiés par un même login.
3. `HttpSessionContextIntegrationFilter` : permet de stocker le contexte de sécurité d'Acegi Security dans la session utilisateur. Il est essentiel d'avoir ce filtre en place pour qu'Acegi Security puisse fonctionner.
4. Filtres d'authentification, tels que `AuthenticationProcessingFilter`. Dans l'exemple ci-dessus, ce filtre n'est utilisé que sur les URL servant à authentifier un utilisateur.

5. `ContextHolderAwareRequestFilter` : permet d'utiliser l'API servlet standard pour gérer la sécurité, par exemple les méthodes `getRemoteUser` ou `isUserInRole`.
6. `RememberMeProcessingFilter` : gère l'authentification automatique des utilisateurs en fonction d'un cookie.
7. `AnonymousProcessingFilter` : si, à la suite des filtres précédents, l'utilisateur n'est toujours pas authentifié, ce filtre lui donne une authentification anonyme.
8. `SecurityEnforcementFilter` : protège les URL. Si un utilisateur n'est pas authentifié ou n'a pas les bonnes autorisations, ce filtre rejette ses requêtes.

Comme nous avons pu le voir dans l'exemple, nous n'utilisons généralement pas l'ensemble de ces filtres. Seuls trois d'entre eux sont nécessaires pour utiliser Acegi Security de manière simple : `HttpSessionContextIntegrationFilter`, `AuthenticationProcessingFilter` et `SecurityEnforcementFilter`. Nous les présentons plus en détail dans les sections suivantes.

Gestion de l'authentification

Acegi Security propose plusieurs filtres d'authentification, par exemple `JbossIntegrationFilter`, pour que l'authentification soit gérée en interne par le serveur JBoss, mais le filtre normalement utilisé est `AuthenticationProcessingFilter`.

Ce filtre offre un grand nombre d'options de configuration, dont nous ne présentons que les principales. En voici une configuration simple :

```
<bean id="authenticationProcessingFilter"
    class=
     "org.acegisecurity.ui.webapp.AuthenticationProcessingFilter">

  <property name="authenticationManager">
    <ref bean="authenticationManager" />
  </property>
  <property name="authenticationFailureUrl">
    <value>/welcome.action?login_error=true</value>
  </property>
  <property name="defaultTargetUrl">
    <value>/secure/showTodos.action</value>
  </property>
  <property name="filterProcessesUrl">
    <value>/j_acegi_security_check</value>
  </property>
</bean>

<bean id="authenticationManager"
      class="org.acegisecurity.providers.ProviderManager">
```

```
  <property name="providers">
    <list>
      <ref local="authenticationProvider" />
    </list>
  </property>
</bean>
```

Les trois propriétés les plus courantes du Bean `authenticationProcessingFilter` sont les suivantes :

- `authenticationFailureUrl` : URL vers laquelle l'utilisateur est redirigé en cas d'erreur d'authentification. Dans cet exemple, on ajoute un paramètre à la requête de manière à afficher un message d'erreur sur la page d'accueil.
- `defaultTargetUrl` : URL vers laquelle l'utilisateur est redirigé par défaut si son authentification réussit.
- `filterProcessesUrl` : URL utilisée pour traiter l'authentification.

L'authentification à proprement parler est déléguée à un deuxième Bean, `authenticationManager`. Ce dernier liste un certain nombre de méthodes d'authentification. Dans notre cas, il n'y en a qu'une, qui correspond de loin au cas le plus courant, mais il pourrait y avoir plusieurs objets à la suite, chacun gérant un type d'authentification différent.

Nous allons maintenant aborder les deux méthodes d'authentification les plus couramment utilisées : *via* la base de données et *via* un serveur LDAP. Une authentification spécifique sera réalisée dans le cadre de l'étude de cas.

Authentification *via* une base de données

Si l'application que nous développons n'est pas connectée à un système de sécurité fourni par l'entreprise, le plus simple est de stocker les données d'authentification au sein de la base de données.

Pour ce faire, Acegi Security propose un schéma de base de données par défaut. Si ce dernier ne convient pas aux besoins, le DAO utilisé pour l'interroger est entièrement configurable :

```
<bean id="authenticationProvider"
      class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">

  <property name="userDetailsService">
    <ref local="userDetailsService" />
  </property>
</bean>

<bean id="userDetailsService"
      class="org.acegisecurity.userdetails.jdbc.JdbcDaoImpl">

  <property name="dataSource">
    <ref bean="dataSource"/>
  </property>
```

```
  <property name="usersByUsernameQuery">
    <value>
      SELECT login,password,enabled FROM user WHERE login = ?
    </value>
  </property>

  <property name="authoritiesByUsernameQuery">
    <value>
      SELECT login,role FROM role WHERE login = ?
    </value>
  </property>
</bean>
```

Dans cet exemple, la première requête doit retourner trois valeurs : le login, le mot de passe et si l'utilisateur est activé ou non. La seconde requête a pour but de retrouver des couples login-rôle, un utilisateur pouvant avoir plusieurs rôles.

Ces requêtes doivent permettre de mapper tout type de schéma de base de données sur ce DAO. Si cela n'est toujours pas suffisant, il reste possible de configurer encore plus finement cet objet. Cela nécessite toutefois de connaître un peu iBATIS, un outil de mappage objet/relationnel utilisé en interne dans Acegi Security, ou de construire notre propre DAO. Nous verrons cette dernière option à la section consacrée à l'étude de cas.

Authentification *via* LDAP

Les annuaires LDAP sont aujourd'hui de plus en plus répandus en entreprise. Acegi Security propose un type d'`authenticationProvider` spécifique pour accéder à un serveur LDAP. Il se configure de la façon suivante :

```
<bean id="initialDirContextFactory"
      class="org.acegisecurity.providers.ldap.
                                DefaultInitialDirContextFactory">

  <constructor-arg
        value="ldap://serveurldap:389/dc=exemple,dc=com"/>

  <property name="managerDn">
    <value>cn=manager,dc=exemple,dc=com</value>
  </property>
  <property name="managerPassword">
    <value>password</value>
  </property>
</bean>

<bean id="authenticationProvider"
      class="org.acegisecurity.providers.ldap.
                                    LdapAuthenticationProvider">

  <constructor-arg>
    <bean class="org.acegisecurity.providers.ldap.
                                authenticator.BindAuthenticator">
```

```
      <constructor-arg>
        <ref local="initialDirContextFactory"/>
      </constructor-arg>
      <property name="userDnPatterns">
        <list>
          <value>uid={0},ou=people</value>
        </list>
      </property>
    </bean>
  </constructor-arg>
  <constructor-arg>
    <bean class="org.acegisecurity.providers.ldap.
                      populator.DefaultLdapAuthoritiesPopulator">

      <constructor-arg>
        <ref local="initialDirContextFactory"/>
      </constructor-arg>
      <constructor-arg>
        <value>ou=groups</value>
      </constructor-arg>
      <property name="groupSearchFilter">
        <value>(member={0})</value>
      </property>
      <property name="groupRoleAttribute">
        <value>cn</value>
      </property>
      <property name="rolePrefix">
        <value>ROLE_</value>
      </property>
      <property name="convertToUpperCase">
        <value>true</value>
      </property>
    </bean>
  </constructor-arg>
</bean>
```

Cette configuration utilise deux Beans Spring spécifiques, qui utilisent tous deux l'injection de dépendances par constructeur, contrairement au reste d'Acegi Security, qui utilise la plus conventionnelle injection de dépendances par modificateur.

Le premier Bean, `initialDirContextFactory`, représente un gestionnaire de connexions au serveur LDAP (utilisant le design pattern fabrique), qui prend en paramètre le Distinguished Name et le mot de passe d'un administrateur. C'est grâce à ce JavaBean que nous pouvons obtenir des connexions au serveur LDAP dans le second Bean.

Ce dernier correspond à une nouvelle configuration de `authenticationProvider`, qui utilise cette fois une classe spécifique pour la connexion LDAP. Il est lui-même construit à l'aide des deux JavaBeans suivants :

- La classe `BindAuthenticator`, qui permet de lier un utilisateur au serveur LDAP. C'est grâce à elle que le serveur LDAP authentifie l'utilisateur en fonction de son Distinguished Name et de son mot de passe. Le Distinguished Name est retrouvé grâce aux

requêtes passées en paramètres dans `userDnPatterns`. Dans notre exemple, nous utilisons `{0}` pour passer en paramètre le login de l'utilisateur.

- La classe `DefaultLdapAuthoritiesPopulator`, qui permet de retrouver les autorisations de l'utilisateur. Le constructeur de cette classe prend en paramètre le Distinguished Name contenant les groupes (`ou=group` dans l'exemple). Il utilise un certain nombre de propriétés, que nous avons laissées à leurs valeurs par défaut dans l'exemple. Acegi Security recherche tous les groupes dont l'utilisateur est membre (la propriété `groupSearchFilter` prenant en paramètre le login de l'utilisateur), les met en majuscules et les préfixe de `ROLE`. Un utilisateur membre du groupe « admin » a donc un rôle `ROLE_ADMIN`.

Relativement complexe, cette configuration devrait néanmoins permettre d'intégrer Acegi Security avec tout type d'arborescence LDAP.

Gestion des autorisations

Dans Acegi Security, la permission d'accéder à une ressource est décidée par un système de vote.

Cette ressource peut être protégée de deux manières : par la sécurisation des URL utilisant cette ressource ou par programmation orientée aspect, ou POA. Nous allons détailler la configuration et le fonctionnement de ces deux systèmes.

Le système de vote

Afin de déterminer si un utilisateur a le droit d'accéder à une ressource, Acegi Security utilise un système de votes grâce à la classe essentielle `org.acegisecurity.vote.RoleVoter`. Cette classe valide l'accès à une ressource en fonction des rôles possédés par l'utilisateur. Elle se configure dans le fichier **WEB-INF/applicationContext-security.xml :**

```
<bean id="roleVoter" class="org.acegisecurity.vote.RoleVoter">
  <property name="rolePrefix">
    <value>ROLE_</value>
  </property>
</bean>
```

Notons que cette classe ne vote que pour les ressources protégées par un rôle ayant un préfixe donné (`ROLE_` par défaut).

Étant donné que plusieurs de ces Beans de vote peuvent être utilisés pour protéger une ressource, plusieurs votes sont émis pour autoriser l'accès à une ressource donnée.

Le traitement des votes est effectué par un autre Bean, qui utilise l'interface `org.acegisecurity.vote.AccessDecisionVoter`. Il en existe trois implémentations différentes, en fonction de la manière dont nous souhaitons traiter les votes. Elles sont contenues dans le package `org.acegisecurity.vote` :

- `AffirmativeBased` : donne l'autorisation d'accéder à la ressource si l'un des votants au moins a donné son accord.

- `ConsensusBased` : donne l'autorisation si la majorité des votants est d'accord.
- `UnanimousBased` : donne l'autorisation uniquement si l'ensemble des votants est d'accord.

L'implémentation sélectionnée pour gérer les votes est également configurée dans **WEB-INF/applicationContext-security.xml :**

```
<bean id="accessDecisionManager"
      class="org.acegisecurity.vote.AffirmativeBased">

  <property name="allowIfAllAbstainDecisions">
    <value>false</value>
  </property>
  <property name="decisionVoters">
    <list>
      <ref local="roleVoter" />
    </list>
</property>
</bean>
```

Comme nous pouvons le voir dans cette configuration tirée de Tudu Lists, ce Bean utilise le Bean `roleVoter`, que nous avons vu précédemment. Étant donné que, dans notre cas, il n'y a qu'un seul votant, le choix du Bean `accessDecisionManager` n'est au final pas tellement important.

Le système de vote présenté ici autorise des configurations extrêmement sophistiquées. Cependant, dans la pratique, nous n'avons souvent qu'un seul Bean votant, ce qui simplifie grandement les problèmes. C'est cette configuration, la plus courante, que nous avons présentée dans le code de nos deux Beans.

Sécurisation des URL

Nous avons déjà rencontré dans ce chapitre des exemples d'URL protégées avec Acegi Security. D'une manière similaire à la spécification J2EE, Acegi Security permet de protéger des URL en fonction de leur nom. L'utilisation d'Acegi Security offre cependant un avantage décisif : son filtre propose l'utilisation d'expressions régulières de type Ant, bien plus puissantes que leurs rivales de la spécification J2EE.

Ce type de sécurisation nécessite une planification préalable, afin de définir à l'avance quels types d'URL seront protégés et avec quels rôles. Il faut donc fournir un travail de spécification des URL lors de la phase de conception du logiciel. Les motifs d'URL généralement utilisés sont de type `/secure/administrateur/**` ou `/secure/**/responsable_commercial/**`.

Cette méthode est de loin la plus répandue et reste généralement simple d'utilisation.

Utilisation de la POA

Il existe des besoins trop complexes pour être gérés *via* une sécurisation d'URL. Imaginons, par exemple, une application de vente de produits sur Internet. Les utilisateurs finals de l'application n'ont aucun droit en écriture sur le catalogue produit. Nous voudrions être certains qu'un utilisateur ne puisse jamais accéder à une méthode de mise à jour d'un produit. Or, de manière classique, il n'est pas possible de répondre à ce besoin. Nous allons avoir que la programmation orientée aspect le permet.

Acegi Security s'intègre aux spécifications de l'AOP Alliance (implémentée en particulier par Spring AOP) et au framework AspectJ. Leur utilisation étant très similaire, nous allons nous concentrer uniquement sur l'AOP Alliance, dont la gestion de la sécurité est proche de celle des transactions. Dans les deux cas, il s'agit d'ajouter un proxy sur des Beans Spring et de modifier, par le biais de la POA, le comportement de certaines méthodes.

Voici un exemple de Bean gérant la sécurité au niveau de la couche de service à l'aide de l'API de l'AOP Alliance :

```
<bean id="secureTodoListsManager"
      class="org.acegisecurity.intercept.
                    method.aopalliance.MethodSecurityInterceptor">

  <property name="validateConfigAttributes">
    <value>true</value>
  </property>
  <property name="authenticationManager">
    <ref bean="authenticationManager"/>
  </property>
  <property name="accessDecisionManager">
    <ref bean="accessDecisionManager"/>
  </property>
  <property name="runAsManager">
    <ref bean="runAsManager"/>
  </property>
  <property name="objectDefinitionSource">
    <value>
      tudu.service.TodoListsManager.create*=ROLE_ADMIN
      tudu.service.TodoListsManager.update*=ROLE_ADMIN
      tudu.service.TodoListsManager.delete*=ROLE_ADMIN
    </value>
  </property>
</bean>
```

Il est ensuite possible d'utiliser ce Bean en lieu et place du Bean `todoListsManager`. Ce nouveau Bean ne permet l'exécution de certaines méthodes qu'aux utilisateurs possédant le rôle `ROLE_ADMIN`.

Sécurité des objets de domaine

Acegi Security propose une sécurité au niveau des instances des objets de domaine. Nous avons justifié ce besoin en début de chapitre. Si nous prenons l'exemple de Tudu Lists, nous avons jusqu'à présent sécurisé des URL ou des méthodes, mais nous n'avons pas sécurisé les todos eux-mêmes. Nous avons simplement restreint certaines fonctionnalités, par exemple, la suppression d'un todo, mais n'avons pas vérifié si une personne supprimant un todo est bien le possesseur de ce todo. Or, seul le possesseur d'un todo devrait avoir le droit de l'effacer.

Nous rencontrons ce même type de concept dans une banque (seul le possesseur d'un compte peut le voir ou effectuer des retraits) ou dans un site marchant (le panier d'achat ne peut être accédé que par l'utilisateur qui le détient).

Dans une application bien conçue, ce type de faille de sécurité apparaît peu souvent. Il faudrait en effet qu'un utilisateur malveillant puisse passer en paramètre l'identifiant d'un objet et que l'application utilise cet objet sans vérifier qu'il appartient bien à l'utilisateur. Dans Tudu Lists, la méthode recherchant un todo vérifie s'il appartient bien à l'utilisateur en cours et lève une exception si tel n'est pas le cas. Il s'agit donc d'un code spécifique, contenu dans la classe `TodosManagerImpl` :

```
public Todo findTodo(final String todoId) {
  Todo todo = todoDAO.getTodo(todoId);
  TodoList todoList = todo.getTodoList();
  User user = userManager.getCurrentUser();
  if (!user.getTodoLists().contains(todoList)) {
    throw new PermissionDeniedException(
      "Permission denied to access this Todo.");
  }
  return todo;
}
```

Acegi Security propose de résoudre ce problème de manière globale, en ajoutant des listes de contrôle d'accès par objet. Il propose pour cela de créer deux tables, `acl_object_identity` et `acl_permission`, qui sont chargées de stocker respectivement les identifiants des objets et les permissions sur ces instances. Les permissions sont en fait des masques de type UNIX, qui permettent de donner à des utilisateurs ou à des rôles des droits de lecture, création, mise à jour ou suppression par instance d'objet.

L'implémentation fournie par Acegi Security est complète et prête à l'emploi en production. Elle souffre cependant à notre sens d'un problème de performance. Comme nous l'avons vérifié avec Tudu Lists, elle impose une démultiplication des insertions et des suppressions en base à chaque modification d'un todo ou d'une liste (les listes pouvant être partagées par de multiples utilisateurs). De plus, la vérification continuelle des droits d'accès a un impact négatif lors de la simple lecture des données.

Pour remédier à cela, Acegi Security propose un cache fondé sur Ehcache. Malheureusement, cela pose des problèmes de désynchronisation entre les objets réels et le cache, le cache ne prenant pas à chaud le partage d'une liste avec un nouvel utilisateur.

En résumé

Pour pallier la problématique des failles de sécurité potentielles résultant d'un manque de validation au niveau des instances des objets métier, Acegi Security propose une solution éprouvée.

Dans la plupart des cas, il est cependant plus simple et plus performant de valider cet accès spécifiquement dans la couche de service.

Tudu Lists : utilisation d'Acegi Security

Tudu Lists utilise Acegi Security pour gérer ses utilisateurs. Il s'agit d'une authentification *via* un formulaire HTML simple, avec authentification automatique pour trente jours et des autorisations gérées au niveau des URL.

Il s'agit donc d'un modèle de sécurité simple, bien représentatif d'une application Web.

Authentification à base de formulaire HTML

Comme la très grande majorité des applications Web, Tudu Lists utilise une authentification *via* un formulaire HTML. Ce dernier permet de renseigner l'identifiant et le mot de passe de l'utilisateur et renvoie vers la page principale de l'application ou vers une page d'erreur en cas d'échec.

Reprenons plus en détail le filtre d'authentification présenté précédemment :

```
<bean id="authenticationProcessingFilter"
      class="org.acegisecurity.ui.webapp
                                  .AuthenticationProcessingFilter">

  <property name="authenticationManager">
    <ref bean="authenticationManager" />
  </property>
  <property name="authenticationFailureUrl">
    <value>/welcome.action?login_error=true</value>
  </property>
  <property name="defaultTargetUrl">
    <value>/secure/showTodos.action</value>
  </property>
  <property name="filterProcessesUrl">
    <value>/j_acegi_security_check</value>
  </property>
</bean>
```

Ce Bean permet de configurer l'URL d'erreur (`authenticationFailureUrl`), ainsi que l'URL de succès (`defaultTargetUrl`) et l'URL permettant de valider l'authentification (`filterProcessesUrl`).

Détaillons cette version simplifiée du formulaire d'identification **WEB-INF/jsp/login.jsp :**

```
<form action="<c:url value='/j_acegi_security_check'/>"
      method="POST">
```

```
<input type="text"
       name="j_username"
       size="20"
       maxlength="50"/>

<input type="password"
       name="j_password"
       size="20"
       maxlength="50"/>

<input type="submit" value="<fmt:message key="login.submit"/>"/>
<input type="reset" value="<fmt:message key="login.reset"/>"/>

</form>
```

Il s'agit d'un formulaire HTML classique, qui pointe vers le `filterProcessesUrl`. C'est tout ce qu'il suffit de coder pour obtenir un formulaire d'identification avec Acegi Security.

Afin de rendre cette page plus conviviale, nous configurons la propriété `authenticationFailureUrl` pour pointer vers cette même page (l'action Struts `WelcomeAction` redirige vers cette JSP), en prenant en paramètre `login_error=true`. Cela nous permet d'afficher un message d'erreur et de prérenseigner le champ `login` avec le dernier identifiant utilisé (nous considérons que l'utilisateur s'est trompé de mot de passe, mais pas d'identifiant) :

```
<c:if test="${not empty param.login_error}">
  <div class="error">
    <fmt:message key="login.error.title"/>
  </div>
 </c:if>

<input type="text"
       name="j_username"
       size="20"
       maxlength="50"
<c:if test="${not empty param.login_error}">
       value='<%= session
                   .getAttribute(AuthenticationProcessingFilter
                   .ACEGI_SECURITY_LAST_USERNAME_KEY) %>'
</c:if>
       />
```

Remarquons dans cet exemple qu'Acegi Security stocke lui-même en session le dernier identifiant essayé par l'utilisateur.

Authentification HTTP pour les services Web

Nous avons abordé au chapitre 14, dédié aux technologies d'intégration XML, l'utilisation des services Web. Les clients SOAP étant des applications, ils s'authentifient non pas *via* un formulaire HTML mais *via* HTTP. Nous avons choisi ici l'authentification HTTP classique, la plus simple et la plus répandue.

Remarquons qu'il est possible, avec Acegi Security, d'avoir deux types d'authen-tification : une authentification par formulaire et une authentification HTTP, ce qui est totalement impossible avec la spécification J2EE.

Nous sécurisons spécialement l'URL `/ws/**` :

```
<bean id="filterChainProxy"
      class="org.acegisecurity.util.FilterChainProxy">

  <property name="filterInvocationDefinitionSource">
    <value>
      ( ... )
      /ws/**=httpSessionContextIntegrationFilter,
             basicProcessingFilter,
             contextHolderAwareRequestFilter,
             basicSecurityEnforcementFilter
    </value>
  </property>
</bean>
```

Pour cet exemple, nous avons supprimé les autres paramètres de sécurisation et avons ajouté des retours chariot entre les filtres afin d'améliorer la lisibilité.

Notons que cette URL est sécurisée *via* un filtre spécial d'authentification, `basicProcessingFilter`, et que la sécurité est appliquée *via* un autre filtre spécifique, `basicSecurityEnforcementFilter`.

Pour l'accès à cette URL, nous demandons un rôle précis dans le Bean `filterInvocationInterceptor` :

```
<value>
  CONVERT_URL_TO_LOWERCASE_BEFORE_COMPARISON
  PATTERN_TYPE_APACHE_ANT
  /secure/admin/**=ROLE_ADMIN
  /secure/**=ROLE_USER
  /ws/**=ROLE_USER
</value>
```

Enfin, nous configurons les deux Beans `basicProcessingFilter` et `basicSecurityEnforcement-Filter` :

```
<bean id="basicProcessingFilter"
     class="org.acegisecurity.ui.basicauth.BasicProcessingFilter">

  <property name="authenticationManager">
    <ref bean="authenticationManager"/>
```

```
  </property>
  <property name="authenticationEntryPoint">
    <ref bean="basicAuthenticationEntryPoint"/>
  </property>
</bean>

<bean id="basicAuthenticationEntryPoint"
      class="org.acegisecurity.ui.basicauth.
                                BasicProcessingFilterEntryPoint">

  <property name="realmName">
    <value>Tudu Lists Realm</value>
  </property>
</bean>

<bean id="basicSecurityEnforcementFilter"
      class="org.acegisecurity.intercept.web.SecurityEnforcementFilter">

  <property name="filterSecurityInterceptor">
    <ref local="filterInvocationInterceptor" />
  </property>
  <property name="authenticationEntryPoint">
    <ref local="basicAuthenticationEntryPoint" />
  </property>
</bean>
```

Un Bean de type `SecurityEnforcementFilter`, différent de celui existant déjà, est nécessaire, car celui-ci n'a pas la même propriété `authenticationEntryPoint`. Si nous avions gardé sur cette URL le Bean `securityEnforcementFilter` utilisé dans le reste de l'application, il aurait, en cas d'erreur, redirigé l'utilisateur vers le formulaire de login, et non vers l'authentification HTTP.

Authentification automatique par cookie

Un besoin très couramment exprimé, mais totalement ignoré par la spécification J2EE, est l'authentification automatique pour une période de temps donnée. L'absence de cette fonctionnalité frustre souvent les utilisateurs, qui tendent à ne pas revenir sur un site.

Acegi Security fournit en standard un mécanisme stockant les données d'identification de l'utilisateur dans un cookie, ce qui permet de réauthentifier automatiquement ce dernier à chaque nouvelle visite.

Il s'agit d'un filtre nommé `rememberMeProcessingFilter`, que nous déjà rencontré précédemment, configuré dans le Bean `filterChainProxy` :

```
<bean id="rememberMeProcessingFilter"
      class="org.acegisecurity.ui.rememberme.
                                      RememberMeProcessingFilter">

  <property name="rememberMeServices">
    <ref local="rememberMeServices" />
```

```
  </property>
</bean>

<bean id="rememberMeServices"
      class="org.acegisecurity.ui.rememberme.
                                    TokenBasedRememberMeServices">

  <property name="userDetailsService">
    <ref local="userDetailsService" />
  </property>
  <property name="tokenValiditySeconds">
    <value>2592000</value>
  </property>
  <property name="key">
    <value>tuduRocks</value>
  </property>
</bean>
```

Ce filtre permet de configurer la durée de stockage du cookie, exprimée en seconde, sur le poste de l'utilisateur (trente jours dans notre exemple), ainsi que la clé utilisée pour le crypter. La méthode de cryptage utilisée n'est certes pas très avancée, surtout si la clé de cryptage est connue. Elle reste cependant suffisante pour un site Web classique. Si une personne malveillante surveille notre trafic réseau, elle aura de toute manière déjà récupéré notre mot de passe, qui circulait en clair lors de la phase de login.

Si nos besoins en matière de sécurité sont plus élevés, nous devons considérer le passage en HTTPS, qui permet de crypter l'ensemble des données transitant entre le serveur et les clients : cookies, mots de passe, mais aussi données métier.

Pour être actif, ce filtre doit s'intégrer dans la phase d'authentification, en injectant le Bean `rememberMeServices` dans le filtre d'authentification :

```
<bean id="authenticationProcessingFilter"
      class="org.acegisecurity.ui.webapp.
                                  AuthenticationProcessingFilter">

  <property name="authenticationManager">
    <ref bean="authenticationManager" />
  </property>
  <property name="rememberMeServices">
    <ref bean="rememberMeServices" />
  </property>

( ... )

</bean>
```

Il faut en outre configurer un nouveau fournisseur d'authentification, spécifique pour l'authentification automatique, dans le Bean `authenticationManager` :

```
<bean id="authenticationManager"
      class="org.acegisecurity.providers.ProviderManager">
```

```
  <property name="providers">
    <list>
      <ref local="daoAuthenticationProvider" />
      <ref local="rememberMeAuthenticationProvider" />
    </list>
  </property>
</bean>

<bean id="rememberMeAuthenticationProvider"
      class="org.acegisecurity.providers.rememberme.
                                RememberMeAuthenticationProvider">

  <property name="key">
    <value>tuduRocks</value>
  </property>
</bean>
```

Ce fournisseur d'authentification utilise la clé donnée en propriété pour décrypter le login et le mot de passe stockés dans le cookie de l'utilisateur.

Implémentation d'un DAO spécifique d'authentification

Dans Tudu Lists, un utilisateur est avant tout un « possesseur de todos ». Nous voulons que cet objet métier soit utilisé à la fois par l'application et par Acegi Security. Nous lui ajoutons donc un mot de passe et des rôles. De plus, les objets de Tudu Lists étant sauvegardés en base de données grâce à Hibernate, nous voulons un comportement consistant entre l'application et Acegi Security. Si nous n'avions pas un objet utilisateur unique, les caches utilisés par Hibernate et Acegi Security pourraient être désynchronisés.

Pour cette raison, nous sommes contraints de créer notre propre DAO, intégrant Acegi Security avec la couche de domaine de Tudu Lists, fondée sur Hibernate. Notre objectif final est d'utiliser ce DAO en lieu et place du DAO fourni par Acegi Security, que nous avons déjà rencontré pour l'authentification *via* la base de données :

```
<bean id="authenticationProvider"
      class="org.acegisecurity.providers.dao.DaoAuthenticationProvider">

  <property name="userDetailsService">
    <ref local="userDetailsService" />
  </property>
</bean>

<bean id="userDetailsService"
      class="tudu.security.UserDetailsServiceImpl">

  <property name="userManager">
    <ref bean="userManager" />
  </property>
</bean>
```

Pour que cette intégration fonctionne, nous codons le Bean `tudu.security.UserDetailsServiceImpl`, qui implémente l'interface `org.acegisecurity.userdetails.UserDetailsService`. En voici la méthode principale, qui permet de renseigner un objet d'Acegi Security à partir de données provenant de la couche de persistance de Tudu Lists :

```
public UserDetails loadUserByUsername(String login)
  throws UsernameNotFoundException, DataAccessException {

  login = login.toLowerCase();
  User user = userManager.findUser(login);

  Set<Role> roles = user.getRoles();
  GrantedAuthority[] arrayAuths =
    new GrantedAuthority[roles.size()];

  int index = 0;
  for (Role role : roles) {
    GrantedAuthorityImpl authority =
      new GrantedAuthorityImpl(role.getRole());

    arrayAuths[index++] = authority;
  }
  org.acegisecurity.userdetails.User acegiUser =
    new org.acegisecurity.userdetails.User(
      login, user.getPassword(), user.isEnabled(), true, true,
      true, arrayAuths);

  return acegiUser;
}
```

Cette méthode renseigne un objet `User` et des objets `GrantedAuthority`, qui sont propres à Acegi Security. Le premier stocke des données utilisateur (login, mot de passe), tandis que les autres stockent ses autorisations.

Cet exemple montre qu'il est relativement aisé de connecter Acegi Security à un système propriétaire spécifique. Nous obtenons de la sorte un système unifié, au lieu d'avoir deux systèmes dont les caches seraient probablement désynchronisés de temps à autre.

Recherche de l'utilisateur en cours

Une fonctionnalité souvent utile est la recherche de l'utilisateur en cours, un sujet également mal géré par la spécification J2EE, qui ne propose pour cela qu'une méthode `getRemoteUser()` associée à l'objet `HttpServletRequest`. Que faire si nous recherchons l'utilisateur ailleurs que dans une servlet ?

Acegi Security stocke l'utilisateur dans une variable associée au fil d'exécution en cours et peut donc le retrouver à n'importe quel endroit de l'application. Il suffit d'ajouter la méthode suivante au Bean `userManager` :

```
public User getCurrentUser() {
  SecurityContext securityContext =
    SecurityContextHolder.getContext();

  org.acegisecurity.userdetails.User acegiUser =
    (org.acegisecurity.userdetails.User)
    securityContext.getAuthentication().getPrincipal();

  tudu.domain.model.User user =
    this.findUser(acegiUser.getUsername());

  return user;
}
```

En injectant ce Bean et en appelant la méthode ci-dessus, nous pouvons retrouver l'utilisateur en cours, et ce, quelle que soit la couche de l'application dans laquelle nous nous trouvons.

Gestion des autorisations dans les JSP

Acegi Security s'intègre de deux manières au sein des JSP : *via* une bibliothèque de tags spécifique ou *via* la sécurité J2EE standard. Nous choisissons d'utiliser la sécurité J2EE standard à la fois pour minimiser les dépendances à Acegi Security et parce que l'API standard est déjà largement connue des développeurs.

Pour intégrer cette sécurité standard dans les JSP, nous utilisons le filtre `contextHolderAwareRequestFilter` dans la configuration du Bean `filterChainProxy`, que nous avons rencontré.

Ce filtre se configure très simplement :

```
<bean id="contextHolderAwareRequestFilter"
      class="org.acegisecurity.wrapper.
                        SecurityContextHolderAwareRequestFilter" />
```

Son rôle est d'enrober la requête HTTP afin de surcharger les méthodes `getRemoteUser` et `isUserInRole`, intégrant ainsi de manière transparente Acegi Security avec les méthodes standards J2EE.

Suite au passage de ce filtre, nous pouvons utiliser dans l'ensemble de l'application la sécurité standard, notamment dans la configuration de Struts (afin de limiter l'exécution de certaines Actions en fonction du rôle de l'utilisateur) et à l'intérieur des servlets. Cependant, le meilleur parti que nous pouvons tirer de ce filtre est de l'utiliser au niveau des JSP, afin de n'afficher que certaines portions de pages en fonction de l'utilisateur.

Nous retrouvons cette utilisation dans la page **WEB-INF/jspf/header.jsp,** qui affiche des menus en fonction du rôle de l'utilisateur ainsi que son identifiant s'il est authentifié. En voici une version simplifiée :

```
<%
 if (request.isUserInRole("ROLE_USER")) {
%>
<menu:useMenuDisplayer name="TabbedMenu"
                       bundle="org.apache.struts.action.MESSAGE">

  <menu:displayMenu name="Todos"/>
  <%
   if (request.isUserInRole("ROLE_ADMIN")) {
  %>
   <menu:displayMenu name="Administration"/>
  <%
   }
  %>
  <menu:displayMenu name="Logout"/>
</menu:useMenuDisplayer>
<%
 }
%>
```

Nous aurions pu également utiliser la bibliothèque de tags `logic` de Struts avec un résultat similaire, sans utiliser de code Java :

```
<logic:present role="ROLE_USER">
  Ce texte s'affiche si l'utilisateur possède le rôle demandé.
</logic:present>
```

Utilisation d'un cache

Au cours de ce chapitre, nous avons vu comment sécuriser une application Web sans trop nous soucier de la performance. Pour effectuer chaque validation, Acegi Security doit appeler des systèmes externes, notamment des bases de données et des annuaires LDAP dans nos exemples. Ces appels, multipliés par le nombre de requêtes, sont malheureusement coûteux en performances.

Pour remédier à ce problème, nous utilisons un cache, d'une manière très similaire à celle que nous avons rencontrée au chapitre 11, consacré à la persistance. Acegi Security utilise d'ailleurs le même produit que celui introduit pour Hibernate, Ehcache. Cela offre un appréciable gain de temps en terme d'installation et de configuration.

Ce cache permet d'éviter d'appeler systématiquement le DAO d'authentification. Nous le configurons au sein du `daoauthenticationProvider`, dans la propriété `userCache` :

```
<bean id="daoAuthenticationProvider"
      class="org.acegisecurity.providers.dao.
                                        DaoAuthenticationProvider">
```

```
  <property name="userDetailsService">
    <ref local="userDetailsService" />
  </property>
  <property name="userCache">
    <ref local="userCache" />
  </property>
</bean>
```

`userCache` représente un Bean spécifique, configuré de la manière suivante :

```
<bean id="userCache"
      class="org.acegisecurity.providers.dao.cache.
                                          EhCacheBasedUserCache">

  <property name="cache">
    <bean class="org.springframework.cache.ehcache.
                                             EhCacheFactoryBean">

      <property name="cacheManager">
        <bean class="org.springframework.cache.ehcache.
                                     EhCacheManagerFactoryBean" />
      </property>
      <property name="cacheName">
        <value>AcegiUserCache</value>
      </property>
    </bean>
  </property>
</bean>
```

Les données d'authentification retournées par le DAO sont ainsi stockées dans un cache Ehcache, qui se configure dans le fichier **ehcache.xml,** au côté de la configuration d'Hibernate :

```
<cache name="AcegiUserCache"
       maxElementsInMemory="5000"
       eternal="false"
       overflowToDisk="true"
       timeToIdleSeconds="300"/>
```

Dans cet exemple, les données d'authentification d'un utilisateur sont conservées pendant cinq minutes après le dernier accès.

Conclusion

La sécurité est une problématique critique pour toute application Web. Malheureusement, la spécification J2EE est particulièrement lacunaire à cet égard. De nombreux projets sont contraints de développer leur propre modèle de sécurité, alors même qu'il s'agit d'une préoccupation récurrente, qui devrait bénéficier de solutions génériques.

De plus, comme nous l'avons montré dans ce chapitre, la sécurité est un domaine complexe, qu'il ne faut pas traiter à la légère.

Acegi Security propose un modèle de sécurité éprouvé, stable et performant, à même de répondre à l'ensemble des attentes : sécurisation des URL, des méthodes et des instances d'objets, fourniture de nombreux filtres, permettant l'authentification par formulaire, authentification automatique par cookie, etc.

Il s'agit cependant d'un framework difficile à maîtriser et à mettre en place. Dans ce chapitre, nous l'avons volontairement utilisé dans une étude de cas simple, afin de servir de base à des configurations plus ambitieuses.

Le framework Acegi Security étant très peu intrusif, sa configuration est indépendante de celle des autres Beans Spring. C'est là son principal atout. La sécurité est en effet une notion transversale, qui ne doit pas avoir d'impact sur l'application en elle-même.

Partie V

Les outils connexes

Outre son conteneur léger, son support de la POA et l'intégration de frameworks tiers, Spring offre un ensemble d'outils facilitant les tests d'applications ainsi que leur supervision. Ces outils sont très utiles pour garantir la qualité de service des applications.

Le chapitre 16 décrit la technologie JMX, qui offre un cadre générique pour superviser les applications. Nous détaillons le support de cette technologie par Spring et montrons comment il facilite sa mise en œuvre en impactant au minimum le code des applications Java/J2EE.

Le chapitre 17 aborde les tests unitaires et d'intégration. Spring offre un support efficace des tests fondés sur JUnit afin de faciliter leur mise en œuvre pour vérifier le bon fonctionnement des différents composants d'une application.

16

Supervision avec JMX

La supervision offre un cadre normalisé et homogène pour visualiser des informations relatives à une application et d'interagir avec son paramétrage lors de son exécution. Ces informations peuvent être visualisées à la demande de la console de supervision ou être dispatchées par l'application elle-même. Les enjeux de la supervision sont aussi bien l'augmentation de la disponibilité des applications et de la réactivité face aux problèmes d'exécution que la facilitation du contrôle des systèmes d'information.

Java s'est très vite orienté vers la supervision par le biais de la technologie JMX (Java Management eXtensions). L'objectif de JMX est de définir une architecture ainsi qu'un ensemble d'interfaces de programmation standards afin de superviser grâce à ce langage des réseaux, des services et des équipements dont les composants peuvent être écrits dans divers langages.

Nous entendons par supervision le fait d'accéder à des informations sur l'état de composants à un moment donné de l'exécution d'une application, ce qui est particulièrement intéressant pour déterminer les causes de dysfonctionnements d'une application.

Le succès de JMX est tel dans le monde Java/J2EE que cette technologie est désormais intégrée aux serveurs d'applications J2EE ainsi qu'à certains frameworks. Cette mise en œuvre donne accès à des informations concernant les ressources utilisées et permet d'effectuer des opérations d'administration durant l'exécution des applications.

Les spécifications JMX

Cette section présente l'ensemble des spécifications relatives à JMX et donne un aperçu de son architecture générale ainsi que des concepts mis en œuvre.

La fin de la section traite des différentes implémentations de JMX ainsi que des consoles JMX disponibles.

La technologie JMX comporte plusieurs spécifications différentes, notamment les deux suivantes, qui en décrivent les fondations :

- JSR 3, en version 1.2 actuellement, qui concerne les concepts et l'architecture de JMX.
- JSR 160, qui étend la précédente et standardise la manière d'accéder aux agents JMX depuis les applications de supervision compatibles JMX, englobant les aspects d'interopérabilité, de transparence et de sécurité des accès.

Une autre spécification, la JSR 255, prend le relais des précédentes afin de décrire la version 2.0 de JMX, qui sera intégrée à la version 6.0 de Java. Le présent ouvrage ne traitant pas de cette version de Java, nous ne détaillons pas cette spécification.

L'objectif des spécifications JMX est de décrire la manière de récupérer des informations concernant des ressources applicatives, d'interagir avec ces dernières et de déclencher certaines opérations. Ces ressources peuvent correspondre à des composants techniques, tels que des pools de connexions, mais également des composants d'une application ou d'un framework.

JMX fournit un cadre robuste, qui permet de spécifier finement les propriétés et actions accessibles des composants. La technologie propose également un mécanisme pour notifier les applications de supervision suite à des événements.

Architecture de JMX

JMX a pour objet de standardiser la structuration des composants supervisables tout en les rendant accessibles aux outils de supervision.

La spécification met en œuvre une architecture à trois niveaux, comme illustré à la figure 16.1 :

- Le niveau le plus proche des ressources correspond à l'*instrumentation*, dont l'objectif est de fournir une ressource supervisable. La spécification JMX offre divers cadres techniques à cet effet, dont certains ont un impact sur les ressources.
- La couche intermédiaire entre ces ressources et les outils de supervision gère les différentes ressources supervisables et est nommée *agents*. Les entités de cette couche permettent notamment d'associer un nom à une ressource supervisable ainsi que des observateurs.
- Le niveau *services distribués* spécifie l'accès aux ressources par les outils ou applications de supervision par le biais de différents protocoles ou connecteurs.

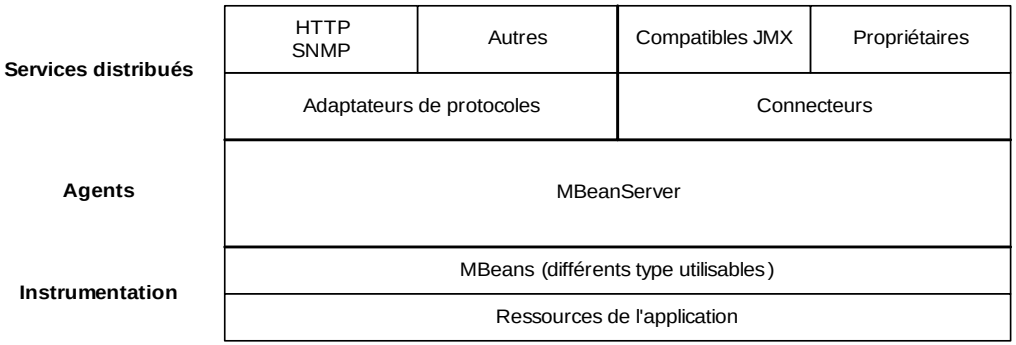


Figure 16.1
Architecture de JMX

Afin de décrire les différents MBeans JMX, nous allons commencer par introduire un exemple issu de Tudu Lists. L'objectif est de superviser le composant applicatif d'identifiant `datasource` afin de visualiser et éventuellement de modifier ses propriétés.

Ce composant est configuré dans le fichier **applicationContext-hibernate.xml** localisé dans le répertoire **WEB-INF :**

```
<bean id="dataSource" class="org.springframework
                        .jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
              value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

Comme le montre le code ci-dessus, le composant possède les propriétés suivantes :

- `driverClassName`, qui spécifie la classe du driver JDBC utilisée.
- `url`, qui définit l'adresse JDBC de la base de données utilisée.
- `username` et `password`, qui spécifient respectivement l'identifiant et le mot de passe de l'utilisateur.

Les valeurs de ces propriétés sont définies dans le fichier **hibernate.properties** localisé dans le répertoire **WEB-INF :**

```
(...)
jdbc.driverClassName=com.mysql.jdbc.Driver
jdbc.url=jdbc:mysql://localhost:3306/tudu
jdbc.username=root
jdbc.password=
(...)
```

Le niveau *instrumentation*

Le MBean (Managed Bean) est l'entité de base de JMX. Il définit un cadre de conception afin qu'un serveur de supervision compatible JMX puisse déterminer ou positionner la valeur d'une propriété ainsi qu'appeler une méthode d'un composant applicatif.

JMX décrit quatre types de MBeans, qui répondent à des règles de mise en œuvre variées, adaptées aux divers besoins des applications en terme de fonctionnalités et de complexité. Ces différentes règles doivent être strictement suivies afin que les MBeans soient compatibles JMX. Remarquons que, dans le cas contraire, ils sont considérés comme incompatibles. Dans ce cas, une exception de type `NotCompliantMBeanException` est levée lors de leur enregistrement.

La validité de ces règles est vérifiée par le serveur de MBeans au moment de l'enregistrement en utilisant les mécanismes d'introspection de Java.

Le tableau 16.1 récapitule les différents types de MBeans spécifiés par JMX. Nous les décrivons plus en détail à la section suivante.

Tableau 16.1. MBeans spécifiés par JMX

Type	Description
`StandardMBean`	Correspond au type de MBean le plus simple. Il s'appuie sur une interface utilisateur afin de déterminer les métadonnées à exposer dans JMX.
`DynamicMBean`	Étend le type de MBean précédent afin de rendre dynamique la détermination des métadonnées. Il s'appuie sur une interface définie par la spécification JMX.
`ModelMBean`	Permet de spécifier les métadonnées à exposer dans JMX par l'intermédiaire d'interfaces et de classes de JMX sans impacter le composant.
`OpenMBean`	Étend le type de MBean précédent afin d'adresser des utilisations avancées de JMX.

Les différents types de MBeans

JMX offre diverses possibilités pour mettre en œuvre des MBeans offrant différentes caractéristiques et ayant plus ou moins d'impact sur l'architecture et les composants eux-mêmes.

Le premier type de MBean, `StandardMBean`, se distingue par sa simplicité. Seule la mise en œuvre d'une interface spécifique est nécessaire pour décrire les propriétés et méthodes accessibles par JMX. Le Bean à superviser doit impérativement implémenter cette interface. De ce fait, il se trouve lié explicitement à la spécification, sauf à recourir à la POA, qui permet d'ajouter cette interface à la classe de manière transparente.

L'interface doit respecter certaines conventions décrites par la spécification JMX. De ce fait, elle doit être codée pour un composant spécifique, et son nom doit être suffixé par `MBean`. Au travers de cette interface, ce dernier définit les propriétés et méthodes exposées dans JMX. Remarquons que ces propriétés sont mises en œuvre grâce à des accesseurs et des modificateurs. Si l'accesseur est omis, la propriété est en lecture seule. Inversement, si le modificateur est omis, la propriété est uniquement disponible en écriture.

Comme la classe `DriverManagerDataSource` est non pas une classe de l'application mais une classe du support JDBC de Spring, nous ne pouvons pas la modifier. Nous devons donc créer une sous-classe afin de lui ajouter des méthodes et d'implémenter l'interface JMX.

Nous nommons cette sous-classe `JmxDriverManagerDataSource` et la configurons dans le fichier de configuration **applicationContext-hibernate.xml** de la manière suivante :

```
<bean id="dataSource" class="tudu.jmx.JmxDriverManagerDataSource">
    <property name="driverClassName"
              value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
</bean>
```

Le code suivant décrit l'interface que le composant doit implémenter pour définir les propriétés et méthodes exposées dans JMX :

```
public interface JmxDriverManagerDataSourceMBean {
    String getDriverClassName();
    String getUrl();

    String getUsername();
    void setUsername(String username);

    String getPassword();
    void setPassword(String password);

    void reset();
}
```

Le composant doit implémenter cette interface pour être utilisable par JMX, comme dans le code suivant :

```
public class JmxDriverManagerDataSource
               extends DriverManagerDataSource
               implements JmxDriverManagerDataSourceMBean {

    public void reset() { (...) }
}
```

Notons que la propriété `username` est en lecture-écriture, tandis que la propriété `driverClassName` est en lecture seule.

Ce type de MBean est contraignant, puisqu'il impose de se lier aux conventions JMX, et n'est donc pas transparent pour l'application.

Le MBean `DynamicMBean` utilise le même principe que précédemment mais permet en outre de décrire de manière dynamique les propriétés et méthodes exposées dans JMX

par le biais de l'interface `DynamicMBean`. Celle-ci est localisée dans le package `javax.management` et décrite de la façon suivante :

```
public interface DynamicMBean {
    public Object getAttribute(String attribute)
                        throws AttributeNotFoundException,
                        MBeanException, ReflectionException;
    public void setAttribute(Attribute attribute)
                        throws AttributeNotFoundException,
                        InvalidAttributeValueException,
                        MBeanException, ReflectionException;
    public AttributeList getAttributes(String[] attributes);
    public AttributeList setAttributes(AttributeList attributes);
    public Object invoke(String actionName,
                        Object params[], String signature[])
                        throws MBeanException, ReflectionException;
    public MBeanInfo getMBeanInfo();
}
```

Cette interface utilise la classe `MBeanInfo` du package `javax.management`, qui permet de décrire les métadonnées d'un MBean. Notons également dans le code ci-dessus l'utilisation de la classe `AttributeList` du même package afin d'étendre la classe `ArrayList` et d'empêcher l'ajout d'instances autres que de type `Attribute`. Cette dernière permet de stocker une clé ainsi que sa valeur associée.

Le composant doit implémenter cette interface pour renvoyer les valeurs pour des noms de propriétés ainsi que les métadonnées du MBean.

Le code suivant illustre la mise en œuvre des méthodes `getAttribute` et `invoke` dans le composant précédent afin de le convertir en `DynamicMBean` :

```
public class JmxDriverManagerDataSource
                    extends DriverManagerDataSource
                    implements DynamicMBean {

    public void reset() { (...) }

    public Object getAttribute(String attribute)
                        throws AttributeNotFoundException,
                        MBeanException, ReflectionException {
        if( "driverClassName".equals(attribute) ) {
            return getDriverClassName();
        } else if( "url".equals(attribute) ) {
            return getUrl();
        } else if( "username".equals(attribute) ) {
            return getUsername();
        } else if( "password".equals(attribute) ) {
            return getPassword();
        } else {
```

```
            throw new AttributeNotFoundException(
               "L'attribut avec le nom "+attribute+
               " n'existe pas pour le MBean");
        }
    }

    public Object invoke(String actionName,
                         Object params[], String signature[])
                         throws MBeanException, ReflectionException;
        if( "reset".equals(actionName) ) {
            reset();
            return null;
        } else {
            throw OperationsException(
                  "L'action avec le nom "+actionName+
                  "n'existe pas pour le MBean");
        }
    }
}
```

Comme pour le premier type, l'utilisation de `DynamicMBean` est contraignante. Elle impose en effet de se lier aux API JMX, ce qui n'est pas transparent pour l'application (à moins d'utiliser la POA).

Le type `ModelMBean` ouvre une perspective intéressante pour créer un MBean sans impact sur le composant lui-même en jouant un rôle de proxy. Il permet de décrire les informations concernant les propriétés et méthodes supervisées par l'intermédiaire de l'interface `ModelMBean` de JMX, localisée dans le package `javax.management`, dont le code est le suivant :

```
public interface ModelMBean extends DynamicMBean,
              PersistentMBean, ModelMBeanNotificationBroadcaster {
    public void setModelMBeanInfo(ModelMBeanInfo inModelMBeanInfo)
                throws MBeanException, RuntimeOperationsException;
    public void setManagedResource(Object mr, String mr_type)
                throws MBeanException, RuntimeOperationsException,
                       InstanceNotFoundException,
                       InvalidTargetObjectTypeException;
}
```

Le développement de ce MBean comporte les trois étapes suivantes :

1. Instanciation du MBean par le biais de l'unique implémentation `RequiredModelMBean` du package `javax.management.modelmbean`, fournie par la spécification JMX.
2. Définition des métadonnées du MBean par le biais de la classe `ModelMBeanInfo` du package `javax.management`, comme dans le code suivant :

```
//Création de la description des attributs, des opérations,
//des constructeurs et des notifications du MBean
ModelMBeanAttributeInfo[] attributes=new ModelMBeanAttributeInfo[1];
```

```
Descriptor champ1Desc = new DescriptorSupport();
champ1Desc.setField("name","username");
champ1Desc.setField("descriptorType","attribute");
champ1Desc.setField("displayName","Username");
champ1Desc.setField("getMethod","getUsername");
champ1Desc.setField("setMethod","setUsername");
champ1Desc.setField("currencyTimeLimit","20");

attributes[0]=new ModelMBeanAttributeInfo(
                  "username","java.lang.String",
                  "Description de Username.",true,
                  true,false,champ1Desc);

(...)

ModelMBeanOperationInfo[] operations
                        =new ModelMBeanOperationInfo[0];
ModelMBeanConstructorInfo[] contructors
                        =new ModelMBeanConstructorInfo[0];
ModelMBeanNotificationInfo[] notifications
                        =new ModelMBeanNotificationInfo [0];

//Création de la description globale du MBean
Descriptor descriptor=new DescriptorSupport( new String[]
                 { "name=JmxDriverManagerDataSource","descriptorType=mbean",
                   "displayName=JmxDriverManagerDataSource","log=T",
                   "logfile=jmx.log","currencyTimeLimit=5"});

// Création du ModelMBeanInfo pour l'ensemble du MBean
String className="JmxDriverManagerDataSource";
String description="Description du MBean sur MonComposant.";

ModelMBeanInfo info=new ModelMBeanInfoSupport(
                        className,description,attributes,
                        constructors,operations,notifications);

dMBeanInfo.setMBeanDescriptor(mmbDesc);
```

3. Positionnement des métadonnées et rattachement du Bean au MBean lui-même, comme dans le code suivant :

```
JmxDriverManagerDataSource jmxDataSource = createDataSource();

ModelMBean mbean = new RequiredModelMBean();
mbean.setModelMBeanInfo(mbeanInfo);
mbean.setManagedResource(jmxDataSource, "ObjectReference");
```

Notons qu'un autre type de MBean, `OpenMBean`, est réservé aux usages avancés de JMX et n'est pas détaillé dans ce chapitre.

Le niveau agent

Ce niveau spécifie les différents composants de l'infrastructure de JMX qui permettent de gérer les MBeans. Appelés agents ou serveurs JMX, ces composants offrent la possibilité d'enregistrer ou de désenregistrer les MBeans.

Un MBean est identifié de manière unique par un identifiant spécifié au moment de son enregistrement dans le serveur de MBeans. Désigné par le terme `ObjectName`, cet identifiant se présente sous la forme suivante :

```
domain-name:key1=value1[,key2=value2,...,keyN=valueN]
```

L'élément `domain-name` symbolise le domaine du MBean. Il est suivi d'une liste de clés-valeurs, qui permet de l'identifier, avec, par exemple, une clé ayant pour valeur `name`.

JMX fournit la classe `ObjectName` pour spécifier cet identifiant.

Récupération du serveur de MBeans

Un serveur de MBeans peut être embarqué dans l'application ou fourni par l'infrastructure dans laquelle fonctionne l'application, comme, par exemple, un serveur d'applications.

Dans le premier cas, il est explicitement créé au moyen de la classe JMX `MBeanServerFactory` et de sa méthode `createMBeanServer`. Le code suivant en donne un exemple d'utilisation :

```
MBeanServer server = MBeanServerFactory.createMBeanServer();
```

Dans le cas d'un serveur JMX fourni par l'infrastructure, une recherche du serveur s'impose. La classe `MBeanServerFactory` offre pour cela la méthode `findMBeanServer`, qui prend en paramètre l'identifiant de l'agent, c'est-à-dire du serveur. Ce paramètre peut prendre la valeur `null`. Dans ce cas, tous les serveurs présents sont détectés.

Le code ci-dessous en donne un exemple d'utilisation :

```
List servers = MBeanServerFactory.findMBeanServer(agentId);
```

Enregistrement de MBeans

À partir des notions que nous venons de voir, nous pouvons déduire la facilité avec laquelle il est possible d'enregistrer un MBean. Cela s'effectue en utilisant le nom de la classe à enregistrer ou une instance de celle-ci. L'interface `MBeanServer`, matérialisant le contrat du serveur JMX, fournit pour cela les méthodes `createMBean` et `registerMBean`.

Le code suivant illustre la manière d'enregistrer un MBean avec la méthode `registerMBean` :

```
MBeanServer=getMBeanServer();
ObjectName objName=new ObjectName("MonDomain:Name=Test");
Test test=new Test();
server.registerMBean(test, objName);
```

Le suivant illustre la façon de créer un MBean grâce à la méthode `createMBean` :

```
MBeanServer=getMBeanServer();
ObjectName objName=new ObjectName("MonDomain:Name=Test");
server.createMBean("package.Test", objName);
```

De même, cette interface fournit la méthode `unregisterMBean` pour désenregistrer un MBean, comme dans le code suivant :

```
MBeanServer=getMBeanServer();
ObjectName objName=new ObjectName("MonDomain:Name=Test");
server.unregisterMBean(objName);
```

Le niveau services distribués

Ce niveau spécifie la façon dont les applications clientes de supervision peuvent se connecter au serveur JMX depuis l'extérieur.

La spécification décrit les deux approches suivantes pour cela :

- **Approche par adaptateur de protocole.** Réduit l'impact de JMX sur les applications clientes en donnant accès aux composants JMX du serveur par le biais d'un protocole donné. Cette approche a l'avantage de se fonder sur des protocoles existants. La communauté J2EE possède notamment des projets d'adaptateurs pour les protocoles SNMP, HTTP et CORBA.
- **Approche par connecteur.** Contrairement aux adaptateurs, les connecteurs ont un impact sur l'application cliente puisqu'ils sont composés d'une partie cliente et d'une partie serveur, ces entités étant standardisées par le biais de la JSR 160.

Le code suivant illustre l'utilisation d'un connecteur avec les API JMX au niveau du serveur JMX afin d'autoriser son accès :

```
//Récupération du serveur JMX
MbeanServer server= getMBeanServer();

//Création de l'URL d'accès au connecteur
JMXServiceURL url=new JMXServiceURL(serviceUrl);

//Création de l'environnement
Map environment=createEnvironment();

//Création du connecteur serveur
JMXConnectorServer connectorServer
   JMXConnectorServerFactory.newJMXConnectorServer(
                                url, environment, server);

//Démarrage du connecteur
connectorServer.start();

(...)
```

```
/Arrêt du connecteur

 connectorServer.stop();
```

Une fois la partie serveur du connecteur réalisée, l'application cliente met en œuvre un connecteur client, comme dans le code suivant :

```
//Création de l'URL du connecteur à accéder
JMXServiceURL url=new JMXServiceURL(serviceUrl);

//Création de l'environnement
Map environment=createEnvironment();

//Création du connecteur client
JMXConnector connector=JMXConnectorFactory.connect(
                                    url, this.environment);

//Récupération d'un connecteur au serveur JMX
MbeanServerConnection connection=
                        connector.getMBeanServerConnection();
```

Les notifications JMX

La spécification JMX standardise un mécanisme robuste et complet permettant aux composants enregistrés ou aux agents JMX d'émettre des notifications vers les applications de supervision.

Les mécanismes de notification s'appuient sur les deux interfaces JMX décrivant l'observation ainsi que la façon de les associer ou les dissocier d'un MBean.

L'interface `NotificationListener` du package `javax.management` représente un observateur JMX qui peut être notifié par divers événements. Les observateurs JMX doivent implémenter cette interface, décrite dans le code suivant :

```
public interface NotificationListener {
    void handleNotification(Notification notification,
                                      Object handback) ;
}
```

Elle définit ensuite une fonction de rappel, qui est invoquée lorsqu'un de ces MBeans observés émet une notification. Le second paramètre de la méthode `handleNotification` correspond à des informations globales, spécifiées au moment de l'enregistrement des observateurs.

Pour finir, l'interface `NotificationBroadcaster` du package `javax.management` permet aux MBeans d'associer et de désassocier des observateurs par le biais de méthodes de cette interface appelées par le serveur de MBeans.

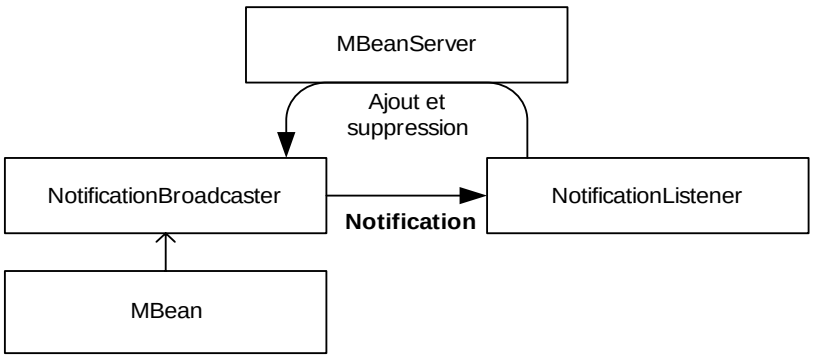
Le code suivant décrit cette interface :

```
public interface NotificationBroadcaster {
    void addNotificationListener(NotificationListener listener,
```

```
                        NotificationFilter filter, Object handback);
    MBeanNotificationInfo[] getNotificationInfo();
    void removeNotificationListener(
                        NotificationListener listener) ;
}
```

La figure 16.2 illustre l'association et la désassociation des observateurs sur un MBean ainsi que la notification.

Figure 16.2
Mécanismes de notification



Les MBeans de types simple et dynamique doivent nécessairement implémenter l'interface `NotificationBroadcaster` pour utiliser les mécanismes de notification. La spécification JMX fournit pour cela l'implémentation `NotificationBroadcasterSupport` de cette interface dans le package `javax.management`, que ces types de MBeans peuvent étendre pour bénéficier directement de ces mécanismes.

Les MBeans de type modèle fonctionnent différemment pour les notifications. L'implémentation `RequiredModelMBean` de l'interface `ModelMBean` étend indirectement `NotificationBroadcaster` par le biais de l'interface `ModelMBeanNotificationBroadcaster`. Cela permet à ce type de MBeans de bénéficier automatiquement des mécanismes de notification.

Enregistrement

Une fois les MBeans prêts à utiliser les mécanismes de notification et les observateurs implémentés, l'application doit les associer les uns aux autres par l'intermédiaire du serveur de MBeans.

Le serveur fournit les méthodes `addNotificationListener` et `removeNotificationListener` dans son interface `MBeanServer` dans le but d'associer ou de désassocier un observateur, dont les signatures sont décrites dans le code ci-dessous :

```
public interface MBeanServer extends MBeanServerConnection {
    (...)
    void addNotificationListener(ObjectName name,
                        NotificationListener listener,
                        NotificationFilter filter, Object handback);
```

```
    void addNotificationListener(
                    ObjectName name, ObjectName listener,
                    NotificationFilter filter, Object handback);
    void removeNotificationListener(
                    ObjectName name,ObjectName listener);
    void removeNotificationListener(
                    ObjectName name, ObjectName listener,
                    NotificationFilter filter, Object handback);
    (...)
}
```

L'association et la désassociation peuvent être réalisées en utilisant l'instance ou le nom JMX de l'observateur. Dans le premier cas, l'observateur est ajouté automatiquement dans JMX.

Les méthodes précédentes permettent de spécifier également les paramètres suivants :

- Un filtre de type `NotificationFilter` offrant la possibilité de filtrer les notifications envoyées à l'observateur. La spécification JMX fournit l'implémentation `NotificationFilterSupport` pour cette interface.
- Un objet `handback` permettant de spécifier des informations générales envoyées à l'observateur en même temps que les notifications.

Le code suivant illustre la façon d'utiliser le serveur de MBeans afin d'associer ou de désassocier un observateur à un MBean :

```
//Récupération du serveur utilisé
MBeanServer server=getMBeanServer();

//Création du MBean
ModelMBean modelMBean=createModelMBean();
ObjectName objectName=getObjectName();

//Enregistrement du MBean dans le serveur
server.registerMBean(modelMBean,objectName);

//Création de l'observateur
NotificationListener listener=createNotificationListener();

//Association de l'observateur pour le MBean
server.addNotificationListener(objectName,listener,null,null);

(...)

//Désenregistrement du MBean dans le serveur
server.unregisterMBean(objectName);

//Désassociation de l'observateur pour le MBean
server.removeNotificationListener(objectName,listener);
```

Implémentations de JMX

Cette section décrit les implémentations JMX les plus courantes ainsi que les consoles de supervision compatibles JMX.

Implémentations serveur

Contrairement aux versions précédentes, la version 5 de la machine virtuelle Java inclut en natif un serveur JMX, qui permet de superviser aussi bien ses constituants que les composants des applications.

Pour l'activer, le paramètre `com.sun.management.jmxremote` doit être spécifié dans la ligne de commande de lancement de la machine virtuelle, comme l'illustre le code suivant :

```
> java -Dcom.sun.management.jmxremote -classpath (...) (...)
```

Le projet MX4J, accessible à l'adresse *http://mx4j.sourceforge.net/,* propose une implémentation Open Source robuste de JMX. La spécification du paramètre `javax.management.builder.initial` dans la ligne de commande de lancement de la machine virtuelle permet de l'utiliser, comme dans le code suivant :

```
> java -Djavax.management.builder.initial
         =mx4j.server.MX4JMBeanServerBuilder -classpath (...) (...)
```

Les serveurs d'applications J2EE intègrent généralement leur propre implémentation de JMX, tel WebSphere et son implémentation développée par Tivoli.

Clients JMX

L'objectif de ces clients JMX est d'offrir une console de supervision afin d'interagir avec les MBeans aussi bien par le biais de leurs propriétés que par l'invocation d'actions.

JConsole

La version 5 de Java fournit en natif une console de supervision JMX, qui peut être démarrée par l'intermédiaire du binaire **jconsole.exe** sous Windows. Elle permet de se connecter à des processus Java locaux ou distants.

Cette console donne accès à des informations concernant le fonctionnement de la machine virtuelle aussi bien que des MBeans de l'application, comme l'illustrent les figures 16.3 et 16.4.

MC4J

Le projet MC4J, accessible à l'adresse *http://mc4j.sourceforge.net/,* fournit une implémentation d'une console de supervision compatible JMX. Elle permet de se connecter à différents serveurs JMX par le biais de connecteurs dédiés.

La figure 16.5 illustre le fonctionnement de cet outil.

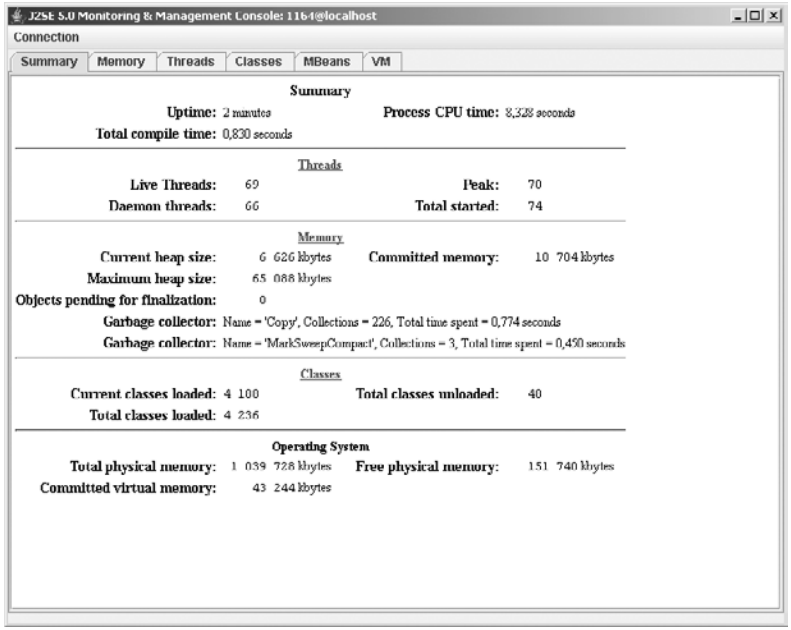


Figure 16.3

Informations concernant le processus Java

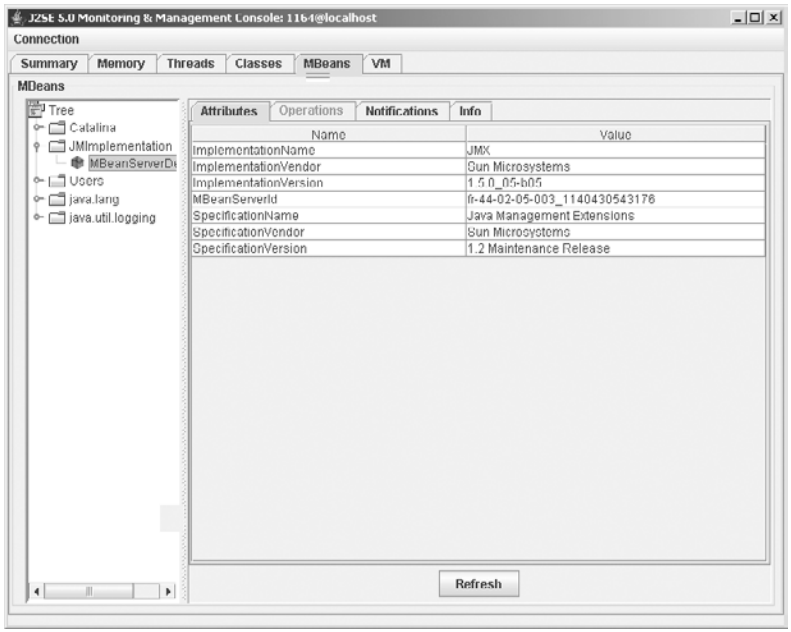


Figure 16.4

Informations concernant les MBeans enregistrés dans le serveur JMX

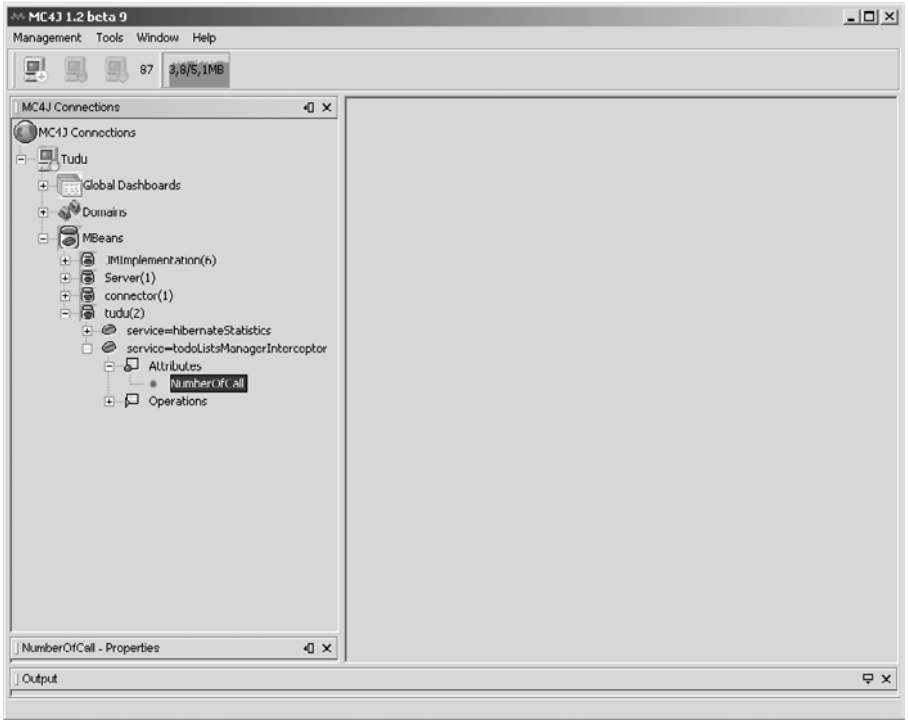


Figure 16.5
Console de supervision MC4J

En résumé

La spécification JMX fournit un cadre normalisé, robuste et éprouvé afin de rendre des applications Java/J2EE supervisables. Elle décrit les différentes ressources supervisables, les entités les gérant et la façon de les accéder depuis des applications externes.

De nombreux projets Open Source fournissent aussi bien des implémentations de serveur JMX que des consoles de supervision compatibles avec cette technologie. L'enjeu consiste à adresser de manière optimale l'intégration de cette technologie au sein de ces applications.

La section suivante détaille le support de JMX par Spring visant à intégrer cette technologie dans les applications par déclaration.

Mise en œuvre de JMX avec Spring

Spring facilite l'enregistrement et la gestion de Beans simples dans un serveur JMX. Le framework construit automatiquement les MBeans correspondant en spécifiant les propriétés et méthodes accessibles par les applications de supervision.

L'objectif du support JMX de Spring est d'intégrer par déclaration cette technologie au sein d'applications Java/J2EE utilisant ce framework. Avec Spring, l'instrumentation des composants pour la supervision ne se fait plus au sein du code mais au moment de l'assemblage de ces composants par déclaration.

Spring offre également la possibilité de spécifier des métadonnées par le biais notamment des annotations, qui permettent d'exporter aisément des informations dans JMX.

Fonctionnalités du support JMX par Spring

Grâce à de nombreuses fonctionnalités dédiées, le support JMX de Spring est très complet.

Avant de détailler ces fonctionnalités dans les sections suivantes, le tableau 16.2 en donne un bref récapitulatif.

Tableau 16.2. Support JMX de Spring

Fonctionnalité	Description
Exportation des MBeans	Permet d'enregistrer un composant ou un MBean dans un serveur JMX par déclaration.
Détermination des informations exposées dans JMX	La fonctionnalité précédente doit déterminer les informations qui seront utilisées et rendues visibles par le serveur JMX. Plusieurs stratégies sont fournies afin de déterminer les métadonnées du composant pour JMX (réflexion, annotations, interface, etc.).
Détermination du serveur JMX utilisé	L'exportation offre plusieurs stratégies (implicites ou explicites) afin de déterminer le serveur JMX à utiliser.
Gestion des noms des MBeans	Lors de l'exportation des MBeans, un nom doit être déterminé afin de les identifier dans le serveur JMX.
Configuration et utilisation des connecteurs JSR 160	Le support offre des facilités afin de mettre en œuvre des connecteurs JSR 160 permettant de rendre accessible un serveur JMX ou d'y accéder.
Configuration et utilisation des notifications	Le support offre des mécanismes afin d'associer par déclaration des observateurs à des MBeans et de déclencher des événements.

Exportation de MBeans

Spring fournit de nombreuses fonctionnalités permettant d'enregistrer des Beans en tant que MBeans dans un serveur JMX.

Le framework offre également diverses approches dans le but de contrôler les propriétés et méthodes exposées qui seront visibles et utilisables par les applications de supervision.

MBeanExporter

La classe centrale du support JMX, `MBeanExporter`, localisée dans le package `org.springframework.jmx.export`, permet de convertir un composant en un MBean suivant des critères spécifiés par déclaration et de l'enregistrer automatiquement dans le serveur JMX.

Le code suivant donne un exemple simple d'exportation d'un Bean dans JMX avec l'identifiant `bean:name=testBean1` :

```
<beans>

  <bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

</beans>
```

Des observateurs peuvent être configurés sur cette classe afin d'être avertis de l'enregistrement ou du désenregistrement d'un MBean. Remarquons que ce mécanisme est propre à Spring, et non à JMX.

Ce type d'observateur doit implémenter l'interface `MBeanExporterListener` suivante :

```
public interface MBeanExporterListener {
    void mbeanRegistered(ObjectName objectName);
    void mbeanUnregistered(ObjectName objectName);
}
```

L'enregistrement des observateurs de ce type s'effectue grâce à la propriété `listeners` de la classe `MBeanExporter`, cette dernière correspondant à un tableau d'écouteurs de type `MBeanExporterListener`.

Le code ci-dessous donne un exemple de configuration simple spécifiant des écouteurs :

```
<beans>
  <bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="listeners">
      <bean class="MonListener1"/>
      <bean class="MonListener2"/>
    </property>
  </bean>
</beans>
```

Sélection et détection du serveur JMX

Il est possible de laisser Spring détecter automatiquement le serveur JMX présent. Cette approche est idéale dans le cas où un serveur JMX est fourni par l'infrastructure technique (Java 5 ou serveurs d'applications, par exemple). Il est parfois préférable de spécifier

le serveur JMX souhaité, notamment lorsque l'application utilise sa propre instance de serveur JMX.

L'approche par détection automatique s'appuie implicitement sur les API de JMX, en particulier la méthode `findMBeanServer` de l'interface `MBeanServerFactory`. Pour sa part, l'approche par sélection de serveur consiste à injecter explicitement le serveur utilisé dans le Bean d'exportation, comme dans le code suivant :

```
<beans>

  <bean id="mbeanServer"
        class="org.springframework.jmx.support.MBeanServerFactoryBean"/>

  <bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean" value-ref="testBean"/>
      </map>
    </property>
    <property name="server" ref="mbeanServer"/>
  </bean>

  <bean id="testBean" class="...">
    (...)
  </bean>

</beans>
```

Contrôle des informations exportées

Nous allons maintenant détailler les diverses approches permettant de contrôler les informations exportées et accessibles par JMX.

La fonctionnalité de contrôle des informations exportées permet de paramétrer la construction des MBeans par rapport aux Beans de l'application. Sa configuration a l'avantage de pouvoir être spécifiée au moment de l'assemblage des composants de l'application et n'a donc aucun impact sur l'application existante.

L'interface *MBeanInfoAssembler*

L'interface `MBeanInfoAssembler` permet de définir l'interface de supervision spécifiant les informations à exposer pour chaque composant. La classe `MBeanExporter` délègue ces traitements à une implémentation de cette interface.

La classe `SimpleReflectiveMBeanInfoAssembler` est utilisée par défaut, mais il est possible d'en spécifier d'autres par l'intermédiaire de la propriété `assembler` :

```
<beans>

  <bean id="exporter"
```

```
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1">
          <ref local="testBean"/>
        </entry>
      </map>
    </property>
    <property name="assembler">
      <ref local="assembler"/>
    </property>
  </bean>

  <bean id="assembler" class="...">
    (...)
  </bean>

</beans>
```

Les implémentations de cette interface ont la responsabilité de créer les informations concernant un MBean de type modèle à partir d'une instance quelconque, comme l'illustre l'exemple suivant de description de cette interface :

```
public interface MBeanInfoAssembler {
    ModelMBeanInfo getMBeanInfo(Object managedBean,
                                String beanKey) throws JMException;
}
```

Les implémentations du contrôle des informations par le support JMX de Spring selon la stratégie désirée sont récapitulées au tableau 16.3.

Tableau 16.3. Implémentations du contrôle des informations

Implémentation	Description
`SimpleReflectiveMBeanInfoAssembler`	Permet de déterminer par introspection les informations à rendre visible dans JMX.
`MetadataMBeanInfoAssembler`	Permet de configurer les stratégies de récupération des métadonnées des composants relatives à JMX.
`MethodNameBasedMBeanInfoAssembler`	Permet de spécifier le nom des méthodes utilisables afin de déterminer les informations à rendre visibles dans JMX.
`MethodExclusionMBeanInfoAssembler`	Permet de configurer le nom des méthodes à ne pas utiliser dans le but de déterminer les informations à rendre visibles dans JMX.
`InterfaceBasedMBeanInfoAssembler`	Permet de spécifier le nom des méthodes par le biais d'interfaces afin de déterminer les informations à rendre visibles dans JMX.

Les sections qui suivent détaillent la façon d'utiliser ces implémentations.

L'approche par annotations

Cette approche s'appuie sur des annotations pour spécifier les divers attributs et méthodes accessibles par JMX. Le support utilise les informations contenues dans ces annotations dans le but de créer le MBean associé.

Le tableau 16.4 récapitule les différentes annotations fournies par le support JMX de Spring.

Tableau 16.4. Annotations fournies par le support JMX de Spring

Annotation	Description
`ManagedResource`	Permet d'exposer une classe ou interface avec JMX et de spécifier ses propriétés associées.
`ManagedOperation`	Permet d'exposer une opération avec JMX et de spécifier ses propriétés associées.
`ManagedAttribute`	Permet d'exposer un attribut avec JMX et de spécifier ses propriétés associées.
`ManagedOperationParameter`	Permet de définir les propriétés concernant les paramètres des opérations avec JMX.

Le tableau 16.5 fournit les propriétés de ces annotations.

Tableau 16.5. Propriétés des annotations

Propriété	Annotation impactée	Description
`objectName`	`ManagedResource`	Définit l'`ObjectName` de la ressource.
`description`	`ManagedResource`, `ManagedOperation`, `ManagedAttribute`, `ManagedOperationParameter`	Fournit une description de la ressource.
`currencyTimeLimit`	`ManagedResource`, `ManagedAttribute`	Définit la valeur de la propriété `currencyTimeLimit`.
`defaultValue`	`ManagedAttribute`	Définit la valeur de la propriété `defaultValue`.
`log`	`ManagedResource`	Définit la valeur de la propriété `log`.
`logFile`	`ManagedResource`	Définit la valeur de la propriété `logFile`.
`persistPolicy`	`ManagedResource`	Définit la valeur de la propriété `persistPolicy`.
`persistPeriod`	`ManagedResource`	Définit la valeur de la propriété `persistPeriod`.
`persistLocation`	`ManagedResource`	Définit la valeur de la propriété `persistLocation`.
`persistName`	`ManagedResource`	Définit la valeur de la propriété `persistName`.
`name`	`ManagedOperationParameter`	Spécifie le nom d'affichage d'un paramètre d'une opération.
`index`	`ManagedOperationParameter`	Spécifie l'index d'un paramètre d'une opération.

La configuration de cette approche se réalise en spécifiant l'implémentation `MetadataMBeanInfoAssembler` pour la classe `MBeanInfoExporter`. Cette implémentation s'appuie sur les métadonnées définies dans la classe du Bean. Puisque plusieurs types de métadonnées

sont disponibles pour cette classe, une instance de la classe `AnnotationsAttributeSource` doit être spécifiée pour l'implémentation `MetadataMBeanInfoAssembler`.

L'approche par annotations peut être configurée finement afin de spécifier les attributs et méthodes à rendre visibles dans JMX ainsi que leurs propriétés. Elle offre également la possibilité d'alléger la configuration JMX dans Spring puisque les Beans utilisant les annotations JMX de Spring peuvent être détectés et enregistrés automatiquement dans le serveur.

Le code ci-dessous donne un exemple de configuration de l'approche par annotations :

```
<beans>

  <bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="autodetect" value="true"/>
    <property name="assembler">
      <ref local="assembler"/>
    </property>
  </bean>


  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name">
      <value>TEST</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </bean>

  <bean id="attributeSource" class="org.springframework.jmx.export
                          .metadata.AnnotationsAttributeSource"/>

  <bean id="assembler" class="org.springframework.jmx
                            .assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource">
      <ref local="attributeSource"/>
    </property>
  </bean>

</beans>
```

Le code suivant décrit un exemple d'implémentation du Bean JmxTestBean, configuré ci-dessus, avec les annotations JMX de Spring :

```
@ManagedResource(objectName="bean:name=testBean1",
                 description="Bean JMX ", log=true,
                 logFile="jmx.log", currencyTimeLimit=15,
                 persistPolicy="OnUpdate", persistPeriod=200,
                 persistLocation="foo", persistName="bar")
public class JmxTestBean {
```

```
    private String name;
    private int age;

    @ManagedAttribute(description="L'attribut name ",
                      currencyTimeLimit=15)
    public String getName() {
        return name;
    }

    @ManagedOperation(description="Une opération")
    public void myOperation() { (...) }

    (...)
}
```

L'approche par autodétection

Dans cette approche, le support JMX de Spring recherche automatiquement les Beans qui possèdent des informations utilisables pour construire des MBeans JMX. Elle nécessite l'utilisation d'une entité d'assemblage implémentant l'interface `AutodetectCapableMBeanInfo`. La seule implémentation de `MBeanInfoAssembler` respectant ce critère est la classe `MetadataMBeanInfoAssembler` décrite précédemment, qui permet d'utiliser des métadonnées de type `annotation` ou `common attribute`.

Avec cette approche, Spring prend comme `objectName` l'identifiant du Bean pour l'enregistrer en tant que MBean.

Pour activer cette approche, la valeur de la propriété `autodetect` de la classe `MBeanExporter` doit être positionnée à `true`, comme dans le code suivant :

```
<beans>
  <bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="assembler" ref="assembler"/>
  <property name="autodetect" value="true"/>
  </bean>

  <bean id="bean:name=testBean1"
        class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="attributeSource" class="org.springframework
            .jmx.export.metadata.AttributesJmxAttributeSource"/>

  <bean id="assembler" class="org.springframework.jmx
                  .export.assembler.MetadataMBeanInfoAssembler">
    <property name="attributeSource" ref="attributeSource"/>
  </bean>
</beans>
```

Remarquons que la configuration de la classe `MBeanExporter` est alors beaucoup plus concise et qu'il n'est plus nécessaire de spécifier les MBeans à enregistrer au niveau de la classe précédente.

L'approche par interfaces

Les informations à exposer dans JMX peuvent être spécifiées par l'intermédiaire d'interfaces. Cette méthode s'appuie sur l'implémentation `InterfaceBasedMBeanInfoAssembler` de l'interface `MBeanInfoAssembler`, qui permet de configurer les interfaces prises en compte, comme dans le code suivant :

```
<beans>
  <bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean5">
          <ref local="testBean"/>
        </entry>
      </map>
    </property>
    <property name="assembler">,
      <bean class="org.springframework
           .jmx.export.assembler.InterfaceBasedMBeanInfoAssembler">
        <property name="managedInterfaces">
          <value>org.springframework.jmx.IJmxTestBean</value>
        </property>
      </bean>
    </property>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name">
      <value>TEST</value>
    </property>
    <property name="age">
      <value>100</value>
    </property>
  </bean>
</beans>
```

L'approche par noms de méthodes

Les méthodes et attributs à exposer dans JMX peuvent être sélectionnés en se fondant sur leur nom ainsi que sur ceux de leurs accesseurs et mutateurs, informations utilisées lors de la création des MBeans associés.

Une instance de l'implémentation `MethodNameBasedMBeanInfoAssembler` doit être spécifiée pour la propriété `assembler` de la classe `MBeanExporter`, comme dans l'exemple ci-dessous :

```
<bean id="exporter"
      class="org.springframework.jmx.export.MBeanExporter">
```

```
    <property name="beans">
      <map>
        <entry key="bean:name=testBean5">
          <ref local="testBean"/>
        </entry>
      </map>
    </property>
    <property name="assembler">
      <bean class="org.springframework.jmx.export
                   .assembler.MethodNameBasedMBeanInfoAssembler">
        <property name="managedMethods">
          <value>add,myOperation,getName,setName,getAge</value>
        </property>
      </bean>
    </property>
</bean>
```

Gestion des noms des MBeans

Le support de Spring permet de spécifier les noms des MBeans de trois manières différentes. Les deux premières consistent à les déclarer explicitement lors de la configuration de la classe `MBeanExporter` de Spring, et la troisième en s'appuyant sur des informations contenues dans les métadonnées des composants à exposer dans JMX.

La première approche consiste à définir les noms des MBeans au moment de la configuration de la classe `MBeanExporter`. Cette approche n'a aucun impact sur les Beans et ne nécessite donc aucune modification de leur code source. Les Beans ne sont pas dans l'obligation de suivre les conventions de codage imposées par certains types de MBeans.

Les Beans à exporter sont spécifiés par l'intermédiaire de la propriété `beans` de type `Map`. Les clés de cette table de hachage correspondent aux noms JMX, et les valeurs font référence aux instances des Beans.

Le code suivant donne un exemple de mise en œuvre de cette fonctionnalité :

```
<beans>
  <bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="bean:name=testBean1" value-ref="testBean"/>
      </map>
    </property>
    <property name="server" ref="mbeanServer"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>
</beans>
```

La deuxième approche consiste à externaliser la définition des noms des MBeans JMX dans un fichier de propriétés. Dans ce cas, la clé de la table de hachage de la propriété `beans` de la classe `MBeanExporter` correspond non plus à un `ObjectName`, mais à une référence qui est résolue par le biais d'une instance de l'interface `ObjectNamingStrategy`. L'implémentation `KeyNamingStrategy` de cette interface met alors en œuvre une résolution en utilisant un fichier de propriétés ou une table de hachage.

Le code suivant donne un exemple d'utilisation de la classe `KeyNamingStrategy` avec la classe `MBeanExporter` :

```
<beans>
  <bean id="exporter"
        class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
      <map>
        <entry key="testBean" value-ref="testBean"/>
      </map>
    </property>
    <property name="namingStrategy" ref="namingStrategy"/>
  </bean>

  <bean id="testBean" class="org.springframework.jmx.JmxTestBean">
    <property name="name" value="TEST"/>
    <property name="age" value="100"/>
  </bean>

  <bean id="namingStrategy" class="org.springframework.jmx.export
                                    .naming.KeyNamingStrategy">
    <property name="mappings">
      <props>
        <prop key="testBean">bean:name=testBean</prop>
      </props>
    </property>
    <property name="mappingLocations">
      <value>names.properties </value>
    </property>
  </bean>
</beans>
```

Par exemple, le code du fichier de propriétés **names.properties** est de la forme :

```
testBean1=bean:name=testBean1
```

Dans la troisième approche, le nom du MBean est spécifié dans la propriété `objectName` de l'annotation `ManagedResource`.

La section précédente relative aux annotations fournit de plus amples détails sur leur utilisation.

Les connecteurs JSR 160

Les connecteurs adressent le niveau services distribués afin de permettre l'accès à un serveur JMX distant.

La spécification JMX distingue deux types de connecteurs. Le premier est mis en œuvre conjointement avec le serveur JMX de manière à le rendre accessible. Le second doit pour sa part être utilisé dans l'application désirant se connecter au serveur distant.

Les connecteurs serveurs

Dans la plupart des cas, ces connecteurs sont déjà présents dans l'infrastructure JMX fournie. Concernant les serveurs d'applications, ces connecteurs sont automatiquement associés et démarrés pour les composants du niveau agent de JMX. Ajoutons que l'utilisation d'un serveur JMX dédié favorise la mise en pratique de connecteurs serveurs dans certaines applications.

Spring supporte cette fonctionnalité par l'intermédiaire de la classe `ConnectorServerFactoryBean`, localisée dans le package `org.springframework.jmx.support`. Cette entité offre la propriété `objectName` afin de s'enregistrer automatiquement dans le serveur JMX. Le connecteur peut également être exécuté dans un nouveau fil d'exécution grâce à la propriété `threaded`.

Le code suivant donne un exemple de configuration d'un connecteur fondé sur le protocole RMI :

```
(...)

<bean id="registry" class="org.springframework.
                           remoting.rmi.RmiRegistryFactoryBean">
    <property name="port" value="1099"/>
</bean>

<bean id="serverConnector" class="org.springframework.
                        jmx.support.ConnectorServerFactoryBean">
    <property name="server" ref="mbeanServer" />
    <property name="objectName" value="connector:name=rmi"/>
    <property name="serviceUrl"
              value="service:jmx:rmi://localhost/jndi/
                             rmi://localhost:1099/myconnector"/>
    <property name="threaded" value="true"/>
</bean>
```

Les connecteurs clients

Ce type de connecteur permet l'accès à un serveur JMX distant associé à un connecteur serveur depuis une application cliente. Spring fournit alors la classe `MBeanServerConnectionFactoryBean` dans le même package que précédemment. Sa seule propriété obligatoire est `serviceUrl`, qui permet de définir l'adresse d'accès au connecteur tout en indiquant le protocole utilisé.

Le code suivant en donne un exemple de mise en œuvre permettant d'utiliser le connecteur serveur décrit à la section précédente :

```
<bean id="clientConnector" class="org.springframework.
                  jmx.support.MBeanServerConnectionFactoryBean">
    <property name="serviceUrl"
              value="service:jmx:rmi://localhost/jndi/
                             rmi://localhost:1099/myconnector"/>
</bean>
```

Les notifications

Le support des notifications a été intégré au support JMX avec la version 2.0 de Spring. Il s'appuie sur la classe `MBeanExporter` du package `org.springframework.jmx.export`.

Cette classe fournit une propriété `notificationListeners` de type `Map` afin de configurer les différents observateurs, comme dans le code suivant :

```
    <bean id="exporter"
      class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="testBean" value-ref="testBean"/>
        </map>
    </property>
    <property name="notificationListeners">
        <map>
            <entry key="*" value-ref="jmxListener"/>
        </map>
    </property>
    <property name="server" ref="mbeanServer" />
</bean>

<bean id="jmxListener" class="MyJmxListener"/>
```

La classe `MBeanExporter` peut également être mise en œuvre exclusivement pour définir des observateurs sur des MBeans précédemment enregistrés.

Une utilisation courante correspond à la configuration d'observateurs pour la classe `MBeanServerDelegate`. Elle permet de recevoir notamment les événements correspondant aux enregistrements ou désenregistrements de MBeans, comme dans l'exemple ci-dessous :

```
<bean id="exporter"
      class="org.springframework.jmx.export.MBeanExporter">
    <property name="notificationListeners">
        <map>
            <entry key=" JMImplementation:type=MbeanServerDelegate"
                                        value-ref="jmxListener"/>
        </map>
    </property>
    <property name="server" ref="mbeanServer" />
</bean>

<bean id="jmxListener" class="MyJmxListener"/>
```

En résumé

Le support JMX de Spring offre un cadre flexible pour utiliser et configurer les éléments des différents niveaux de cette technologie. Au moment de l'assemblage des composants, le développeur choisit la façon dont sont enregistrés les Beans dans le serveur JMX. Le support peut utiliser aussi bien un serveur dédié à l'application qu'un serveur fourni par l'environnement d'exécution, tel qu'un serveur d'applications. La façon d'accéder au serveur JMX peut également être configurée lors de l'assemblage de l'application.

Les atouts de ce support sont la simplification de l'utilisation de JMX et le fait que Spring peut ainsi être utilisé simplement, aussi bien dans des applications J2EE que dans des applications Java autonomes

Tudu Lists : utilisation du support JMX de Spring

Le projet `Tudu-SpringMVC` illustre de quelle façon mettre en œuvre le support JMX de Spring afin d'utiliser les outils JMX d'Hibernate et ainsi de récupérer des informations concernant les appels aux services de l'application.

Nous séparerons la configuration des éléments relatifs à JMX dans le fichier de Spring **applicationContext-jmx.xml,** localisé dans le répertoire **WEB-INF** et utilisé par le contexte de l'application Web.

Pour la configuration du serveur JMX, nous choisissons d'embarquer dans notre application notre propre serveur JMX, de façon à démontrer la facilité de mise en œuvre d'un serveur JMX, et ce quel que soit le type d'application Java/J2EE fondé sur Spring concerné. Nous utilisons l'implémentation MX4J de JMX.

Pour déterminer l'implémentation JMX utilisée, la propriété `javax.management.builder.initial` précédemment décrite doit être spécifiée dans la ligne de commande du serveur avec la valeur `mx4j.server.MX4JMBeanServerBuilder`. Les bibliothèques JMX et MX4J suivantes doivent en outre être ajoutées dans le classpath : `jmxremote_optional.jar`, `jmxremote.jar`, `jmxri.jar` et `mx4j.jar`.

Le code suivant illustre la configuration du serveur JMX dans le fichier **applicationContext-jmx.xml** du répertoire **WEB-INF :**

```
<bean id="mbeanServer"
   class="org.springframework.jmx.support.MBeanServerFactoryBean"/>
```

Dans le but de rendre ce serveur accessible depuis des consoles de supervision telles que MC4J, notre choix se porte sur la configuration d'un connecteur JMX fondé sur le protocole RMI. Ce choix impose la mise en œuvre d'un serveur RMI au cœur de notre application en s'appuyant sur les facilités de Spring, comme dans le code suivant :

```
<bean id="registry"
   class="org.springframework.remoting.rmi.RmiRegistryFactoryBean">
    <property name="port" value="1099"/>
</bean>
```

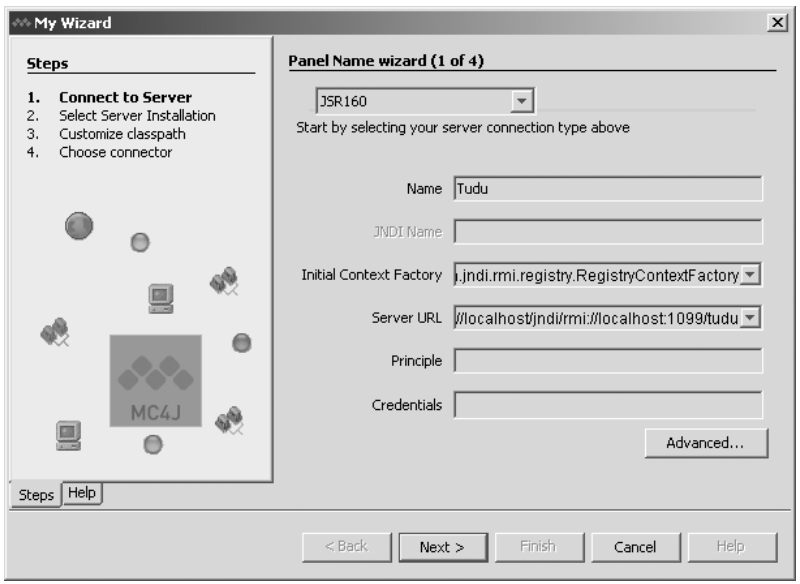
```
<bean id="serverConnector" class="org.springframework.jmx
                              .support.ConnectorServerFactoryBean">
    <property name="server" ref="mbeanServer" />
    <property name="objectName" value="connector:name=rmi"/>
    <property name="serviceUrl" value="service:jmx:rmi://localhost
                          /jndi/rmi://localhost:1099/tudu"/>
    <property name="threaded" value="true"/>
</bean>
```

Précisons que, dans le code ci-dessus, le serveur JMX est disponible par le biais du protocole RMI à l'adresse `service:jmx:rmi://localhost/jndi/rmi://localhost:1099/tudu`. De plus, le connecteur JMX est exécuté dans un fil d'exécution dédié.

Nous configurons désormais l'accès au serveur dans la console MC4J. Nous créons tout d'abord une nouvelle connexion à un serveur puis sélectionnons un type de connexion fondé sur JSR 160. L'utilisation de la classe `RegistryContextFactory` du package `com.sun.jndi.rmi.registry` pour cette connexion est nécessaire, avec en paramètre l'adresse RMI précédemment configurée.

La figure 16.6 illustre la fenêtre de configuration de MC4J.

Figure 16.6
Fenêtre de configuration des paramètres d'accès au serveur JMX



Nous mettons alors en œuvre l'adaptateur fourni par MX4J pour le protocole HTTP grâce à la classe `HttpAdapter` localisée dans le package `mx4j.tools.adaptor.http`. Cette dernière se paramètre en tant que Bean dans le contexte de Spring :

```
<bean id="httpAdaptor" class="mx4j.tools.adaptor.http.HttpAdaptor"
                        init-method="start" destroy-method="stop">
```

```
    <property name="host" value="localhost"/>
    <property name="port" value="7777"/>
</bean>
```

Enfin, nous configurons et implémentons les divers éléments relatifs à la supervision. Dans notre application, les noms des différents MBeans exposés dans JMX sont préfixés par convention par `tudu:service=`.

Utilisation d'Hibernate

Dans un premier temps, nous utilisons le MBean JMX fourni par le framework Hibernate pour la supervision, tout en obtenant des informations concernant l'utilisation des ressources JDBC, la gestion des entités, l'exécution des requêtes ainsi que les caches.

Nous implémentons ce MBean par le biais de la classe `StatisticsService` du package `org.hibernate.jmx`, qui peut être configurée dans Spring de la manière suivante :

```
<bean name="hibernateStatistics"
                      class="org.hibernate.jmx.StatisticsService">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="exporter"
             class="org.springframework.jmx.export.MBeanExporter">
    <property name="beans">
        <map>
            <entry key="tudu:service=hibernateStatistics"
                                value-ref="hibernateStatistics" />
        </map>
    </property>
    <property name="server" ref="mbeanServer" />
    </bean>
```

Tudu Lists utilise le support de la classe `MBeanExporter` de Spring décrite précédemment afin de rendre le MBean d'identifiant `hibernateStatistics` disponible dans JMX avec le nom `tudu:service:hibernateStatistics`.

Le service *todoListsManager*

Nous cherchons maintenant à instaurer un mécanisme fondé sur JMX afin d'identifier les appels au service `todoListsManager` dans Spring.

Dans cette optique, nous implémentons un intercepteur AOP qui incrémente un compteur à chaque appel des méthodes du service. Nous exposons ensuite cette entité dans JMX afin d'accéder aux valeurs du compteur et éventuellement de le réinitialiser.

L'intercepteur est implémenté grâce à la classe `TodoListsManagerInterceptor` localisée dans le package `tudu.service.impl` :

```
public class TodoListsManagerInterceptor
                        implements MethodInterceptor {
    private long numberOfCall=0;

    public Object invoke(MethodInvocation invocation)
```

```
                                        throws Throwable {
        numberOfCall++;
        return invocation.proceed();
    }

    public long getNumberOfCall() {
        return numberOfCall;
    }

    public void reset() {
        numberOfCall=0;
    }
}
```

Le Bean `todoListsManager` se situe déjà derrière un proxy de type `TransactionProxyFactoryBean` de manière à réaliser la démarcation transactionnelle. Ce type de proxy offre la possibilité d'en ajouter un autre dans la liste des intercepteurs par l'intermédiaire de la propriété `preInterceptors`.

Le code suivant illustre la configuration de notre intercepteur :

```
<bean id="todoListsManagerInterceptor"
          class="tudu.service.impl.TodoListsManagerInterceptor"/>

<bean id="todoListsManager" class="org.springframework.transaction
                        .interceptor.TransactionProxyFactoryBean">
    <property name="transactionManager">
        <ref bean="transactionManager" />
    </property>
    <property name="target">
        <ref local="todoListsManagerTarget" />
    </property>
    <property name="preInterceptors">
        <list>
            <ref local="todoListsManagerInterceptor"/>
        </list>
    </property>
    <property name="transactionAttributes">
        <props>
            <prop key="create*">PROPAGATION_REQUIRED</prop>
            <prop key="update*">PROPAGATION_REQUIRED</prop>
            <prop key="delete*">PROPAGATION_REQUIRED</prop>
            <prop key="add*">PROPAGATION_REQUIRED</prop>
            <prop key="restore*">PROPAGATION_REQUIRED</prop>
            <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
        </props>
    </property>
</bean>
```

Pour enregistrer cet intercepteur dans JMX, nous utilisons les fonctionnalités du framework Spring par l'intermédiaire de la classe `MBeanExporter` :

```
    <bean id="exporter"
             class="org.springframework.jmx.export.MBeanExporter">
```

```
<property name="beans">
    <map>
        <entry key="tudu:service=todoListsManagerInterceptor"
                    value-ref="todoListsManagerInterceptor" />
    </map>
</property>
<property name="server" ref="mbeanServer" />
</bean>
```

La supervision

Une fois notre infrastructure JMX en place et nos différents MBeans enregistrés dans le serveur JMX, nous pouvons observer l'évolution de leurs multiples propriétés grâce à la console MC4J et ses outils graphiques.

Dans un premier temps, nous démarrons Tomcat, puis nous nous connectons depuis la console MC4J sur le serveur JMX embarqué. Nous constatons l'apparition des différents MBeans du serveur, incluant ceux concernant Hibernate et les services de Tudu Lists.

La figure 16.7 illustre l'arborescence des MBeans dans la console MC4J.

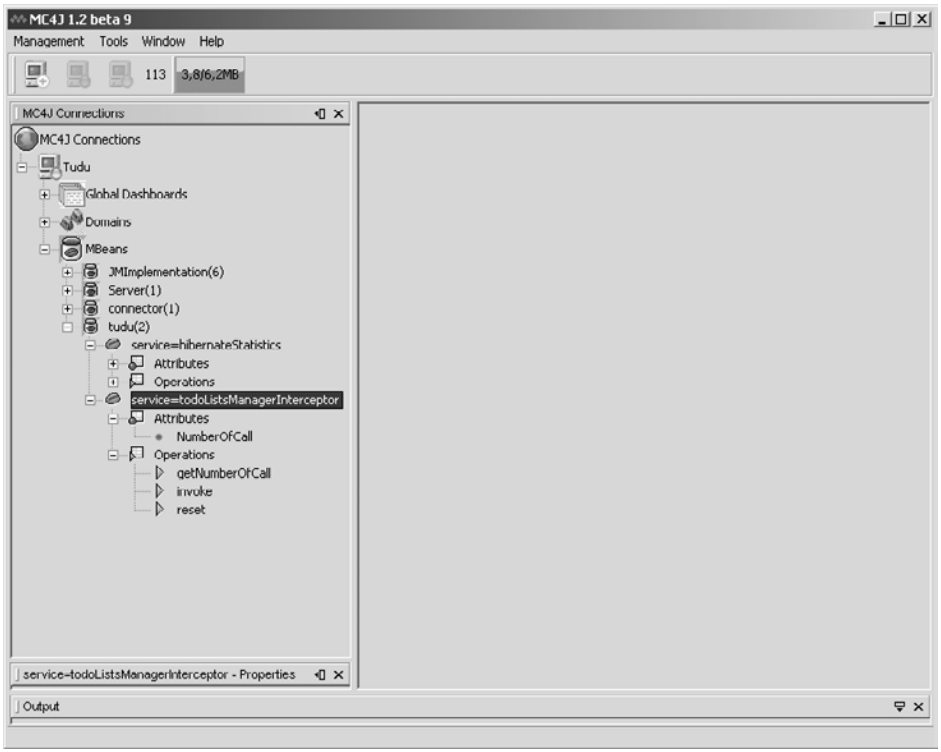


Figure 16.7
Hiérarchie des MBeans dans la console MC4J

La console permet également de visualiser en temps réel et de manière graphique la valeur de la propriété `numberOfCall` de l'intercepteur du service `todoListsManagerInterceptor`, comme l'illustre la figure 16.8.

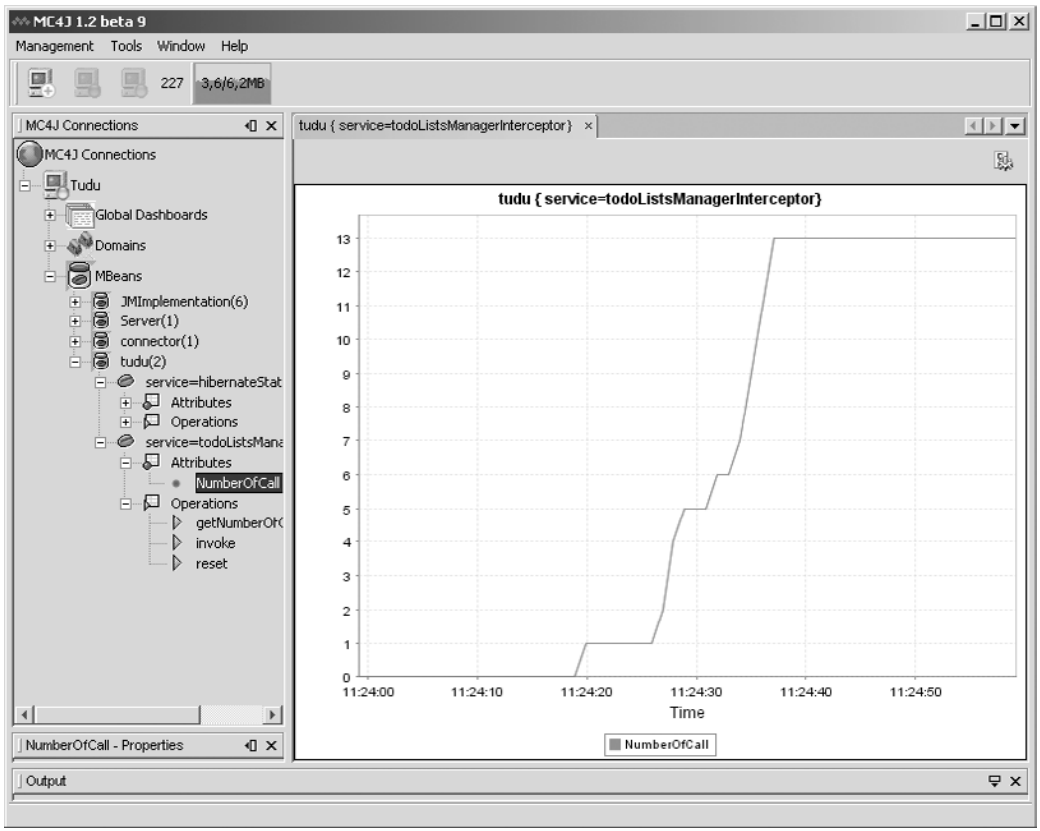


Figure 16.8

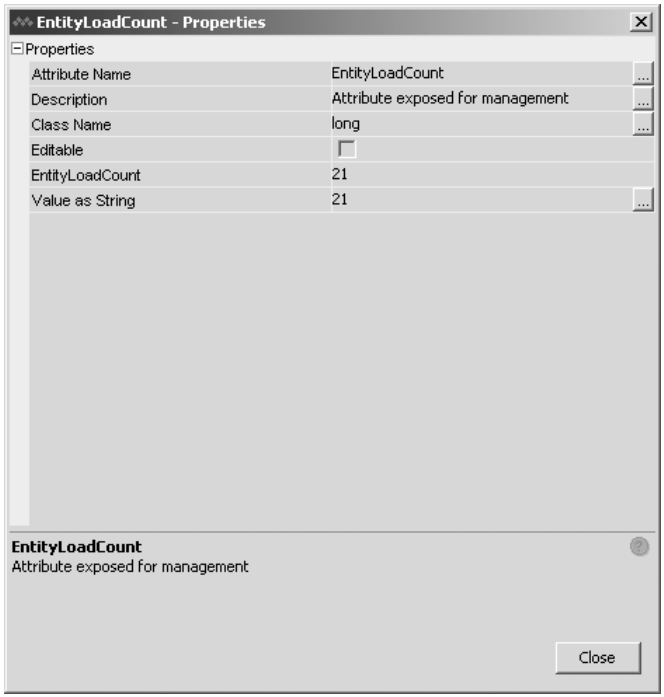
Courbe d'évolution des valeurs de la propriété numberOfCall

La console permet également de consulter des statistiques d'utilisation d'Hibernate, telles que le nombre d'entités chargées depuis le démarrage de l'application, comme l'illustre la figure 16.9.

Grâce à un adaptateur pour le protocole HTTP, nous pouvons également visualiser les différents MBeans dans un navigateur Web au format XML, le format par défaut de MX4J.

Figure 16.9

Statistiques d'utilisation d'Hibernate




```
- <Server>
    <MBean classname="mx4j.server.interceptor.ContextClassLoaderMBeanServerInterceptor" description="MBeanServer interceptor"
    objectname="JMImplementation:interceptor=contextclassloader"/>
    <MBean classname="mx4j.server.interceptor.InvokerMBeanServerInterceptor" description="The interceptor that invokes on the MBean instance"
    objectname="JMImplementation:interceptor=invoker"/>
    <MBean classname="mx4j.server.interceptor.NotificationListenerMBeanServerInterceptor" description="MBeanServer interceptor"
    objectname="JMImplementation:interceptor=notificationwrapper"/>
    <MBean classname="mx4j.server.interceptor.SecurityMBeanServerInterceptor" description="The interceptor that performs security checks for MBeanServer to
    MBean calls" objectname="JMImplementation:interceptor=security"/>
    <MBean classname="mx4j.server.MX4JMBeanServerDelegate" description="Manageable Bean"
    objectname="JMImplementation:type=MBeanServerDelegate"/>
    <MBean classname="mx4j.server.interceptor.MBeanServerInterceptorConfigurator" description="Configurator for MBeanServer to MBean interceptors"
    objectname="JMImplementation:type=MBeanServerInterceptorConfigurator"/>
    <MBean classname="mx4j.tools.adaptor.http.HttpAdaptor" description="HttpAdaptor MBean" objectname="Server:name=HttpAdaptor"/>
    <MBean classname="javax.management.remote.rmi.RMIConnectorServer" description="Manageable Bean" objectname="connector:name=rmi"/>
    <MBean classname="org.hibernate.jmx.StatisticsService" description="Manageable Bean" objectname="tudu:service=hibernateStatistics"/>
    <MBean classname="tudu.service.impl.TodoListsManagerInterceptor" description="tudu.service.impl.TodoListsManagerInterceptor"
    objectname="tudu:service=todoListsManagerInterceptor"/>
  </Server>
```


Figure 16.10

Affichage des MBeans dans un navigateur Web

Conclusion

Le support JMX de Spring offre une grande flexibilité d'utilisation, ainsi que de nombreuses fonctionnalités qui facilitent sa mise en œuvre au sein des applications d'entreprise. Les divers composants en jeu se configurent directement et simplement dans le contexte de Spring.

Différentes stratégies sont fournies pour permettre de déterminer précisément les propriétés et méthodes à rendre accessibles dans JMX. Il est également possible d'exposer de manière transparente de simples Beans. Ainsi, des composants peuvent être exposés dans JMX au moment de l'assemblage de l'application sans aucun développement supplémentaire.

Le support fournit enfin un ensemble de classes qui rendent possible d'embarquer son propre serveur JMX, ainsi que de se connecter et d'interagir avec diverses entités de JMX, comme les connecteurs et les observateurs de MBeans.

17

Test des applications Spring

Les tests sont une des activités fondamentales du développement logiciel. Ce chapitre montre comment tester une application reposant sur Spring. Les types de tests abordés sont les tests unitaires et les tests d'intégration.

Par tests unitaires, nous entendons les tests portant sur un composant unique isolé du reste de l'application et de ses composants techniques (serveur d'applications, base de données, etc.). Par tests d'intégration, nous entendons les tests portant sur un ou plusieurs composants, avec les dépendances associées. Il existe bien entendu d'autres types de tests, comme les tests de performance ou les tests fonctionnels, mais Spring ne propose pas d'outil spécifique dans ces domaines.

Nous nous arrêterons de manière synthétique sur deux outils permettant de réaliser des tests unitaires, le framework JUnit et EasyMock, ce dernier permettant de réaliser des simulacres d'objets, ou *mock objects*. Ces deux outils nous fourniront l'occasion de détailler les concepts fondamentaux des tests unitaires. Nous verrons que l'utilisation de ces outils est toute naturelle pour les applications utilisant Spring, puisque le code de ces dernières ne comporte pas de dépendance à l'égard de ce framework du fait de l'inversion de contrôle. Spring propose d'ailleurs ses propres simulacres pour émuler une partie de l'API J2EE.

Pour les tests d'intégration, nous nous intéresserons aux extensions de JUnit fournies par Spring et par le framework StrutsTestCase, spécialisé dans les tests de composants Web utilisant Struts. Là encore, les tests s'avèrent aisés à implémenter, ces extensions masquant la complexité de mise en œuvre de ces deux frameworks.

Les tests unitaires avec JUnit

JUnit est un framework Java Open Source créé par Erich Gamma et Kent Beck. Il fournit un ensemble de fonctionnalités permettant de tester unitairement les composants d'un logiciel écrit en Java. D'autres frameworks suivant la même philosophie sont disponibles pour d'autres langages ou pour des technologies spécifiques, comme HTTP. Ils constituent la famille des frameworks xUnit.

Initialement conçu pour réaliser des tests unitaires, JUnit peut aussi être utilisé pour réaliser des tests d'intégration, comme nous le verrons plus loin dans ce chapitre.

Dans les sections suivantes, nous indiquons comment manipuler les différents éléments fournis par JUnit afin de créer des tests pour Tudu Lists. Dans cette application, l'ensemble des cas de tests est regroupé dans le répertoire **Tests.** Les différents tests sont organisés selon les classes qu'ils ciblent. Ainsi, ils reproduisent la structure de packages de Tudu Lists.

Les cas de test

Les cas de test sont une des notions de base de JUnit. Il s'agit de regrouper dans une entité unique, en l'occurrence une classe Java dérivant de `junit.framework.TestCase`, un ensemble de tests portant sur une classe de l'application.

Chaque test est matérialisé sous la forme d'une méthode sans paramètre, sans valeur de retour et dont le nom est préfixé conventionnellement par `test` (par exemple, `testEquals`). Le nom de la classe regroupant les tests d'une classe est conventionnellement celui de la classe testée suffixée par `Test` (par exemple, `TodosManagerImplTest`).

Squelette d'un cas de test

Pour introduire la notion de cas de test, nous allons utiliser la classe `Todo` définie dans le package `tudu.domain.model`. Cette classe définit deux méthodes, `compareTo` (méthode de l'interface `java.lang.Comparable`) et `equals` (héritée de `java.lang.Object`).

Pour tester ces deux méthodes, nous allons créer plusieurs instances de la classe `Todo` et effectuer des comparaisons ainsi que des tests d'égalité. Nous définissons pour cela une classe `TodoECTest` ayant deux méthodes, `testCompareTo` et `testEquals` :

```
package tudu.domain.model;

import junit.framework.TestCase;

public class TodoECTest extends TestCase {

   public TodoECTest(String name) {
      super(name);
   }

   (...)

   public void testCompareTo() {
```

```
      (...)
   }

   public void testEquals() {
      (...)
   }
}
```

Notons la présence d'un constructeur faisant appel directement à celui de l'ancêtre de la classe. Ce constructeur est utile pour la méthode `addTest` de la classe `junit.framework.TestSuite`, comme nous le verrons plus loin.

La notion de fixture

Dans JUnit, la notion de *fixture*, ou contexte, correspond à un ensemble d'objets utilisés par les tests d'un cas. Typiquement, un cas est centré sur une classe précise du logiciel. Il est donc possible de définir un attribut ayant ce type et de l'utiliser dans tous les tests du cas. Il devient alors une partie du contexte. Le contexte n'est pas partagé par les tests, chacun d'eux possédant le sien, afin de leur permettre de s'exécuter indépendamment les uns des autres.

Il est possible de définir deux méthodes spécifiques pour gérer le contexte : la méthode `setUp` pour son initialisation et la méthode `tearDown` pour sa destruction. `setUp` est appelée avant l'exécution, et `tearDown` à la fin de l'exécution de chaque méthode de test.

Ces deux méthodes se présentent de la manière suivante :

```
public class TodoECTest extends TestCase {

   protected void setUp() throws Exception {
      // Création du contexte
   }

   protected void tearDown() throws Exception {
      // Destruction du contexte
   }

   (...)
}
```

Pour les tests de la classe `Todo`, nous pouvons créer un ensemble d'attributs de type `Todo` qui nous serviront de jeu d'essai pour nos méthodes de test :

```
public class TodoECTest extends TestCase {

   private Todo todo1;
   private Todo todo2;
   private Todo todo3;

   protected void setUp() throws Exception {
        todo1 = new Todo();
```

```
        todo1.setTodoId("01");
        todo1.setCompleted(false);
        todo1.setDescription("Description");
        todo1.setPriority(0);

        todo2 = new Todo();
        todo2.setTodoId("02");
        todo2.setCompleted(true);
        todo2.setDescription("Description");
        todo2.setPriority(0);

        todo3 = new Todo();
        todo3.setTodoId("01");
        todo3.setCompleted(false);
        todo3.setDescription("Description");
        todo3.setPriority(0);
    }

    (...)
}
```

Les assertions et l'échec

Dans JUnit, les assertions sont des méthodes permettant de comparer une valeur obtenue lors du test avec une valeur attendue. Si la comparaison est satisfaisante, le test peut se poursuivre. Dans le cas contraire, il échoue, et un message d'erreur s'affiche dans l'outil permettant d'exécuter les tests unitaires *(voir plus loin).*

Les assertions sont héritées de la classe `junit.framework.TestCase`. Leur nom est préfixé par `assert`.

Pour les booléens, les assertions suivantes sont disponibles :

```
assertEquals (boolean attendu,boolean obtenu);
assertFalse (boolean obtenu);
assertTrue (boolean obtenu);
```

La première assertion permet de vérifier l'égalité de la valeur obtenue par rapport à une autre variable. Les deux autres testent le booléen `obtenu` sur les deux valeurs littérales possibles, faux ou vrai.

Pour les objets, les assertions suivantes sont disponibles, quel que soit leur type :

```
assertEquals (Object attendu,Object obtenu);
assertSame (Object attendu,Object obtenu);
assertNotSame (Object attendu,Object obtenu);
assertNull (Object obtenu);
assertNotNull (Object obtenu);
```

`assertEquals` teste l'égalité de deux objets tandis qu'`assertSame` teste que `attendu` et `obtenu` font référence à un seul et même objet. Par exemple, deux objets de type `java.util.Date`

peuvent être égaux, c'est-à-dire contenir la même date, sans être pour autant un seul et même objet. `assertNotSame` vérifie que deux objets sont différents. Les deux dernières assertions testent si l'objet obtenu est nul ou non.

Pour chaque type primitif (`int`, `byte`, etc.), une méthode `assertEquals` est définie, permettant de tester l'égalité entre une valeur attendue et une valeur obtenue. Dans le cas des types primitifs correspondant à des nombres réels (`float`, `double`), un paramètre supplémentaire, le delta, est nécessaire, car les comparaisons ne peuvent être tout à fait exactes du fait des arrondis.

Il existe une variante pour chaque assertion prenant une chaîne de caractères en premier paramètre (devant les autres). Cette chaîne de caractères contient le message à afficher si le test échoue au moment de son exécution.

Les assertions ne permettent pas de capter tous les cas d'échec d'un test. Pour ces cas de figure, JUnit fournit la méthode `fail` sous deux variantes : une sans paramètre et une avec un paramètre, permettant de donner le message d'erreur à afficher sous forme de chaîne de caractères. L'appel à cette méthode entraîne l'arrêt immédiat du test en cours et l'affiche en erreur dans l'outil d'exécution des tests unitaires.

Si nous reprenons notre exemple `TodoECTest`, il se présente désormais de la manière suivante :

```
public class TodoECTest extends TestCase {

   (...)

   public void testCompareTo() {
      // Vérifie la consistance avec la méthode equals
      // Cf. JavaDoc de l'API J2SE
      assertTrue(todo1.compareTo(todo3)==0);

      // Vérifie le respect de la spec de Comparable pour null
      // Cf. JavaDoc de l'API J2SE
      try {
         todo1.compareTo(null);
         fail();
      }
      catch(NullPointerException e){
      }

      // todo1 n'est pas fermé donc < à todo2
      assertTrue(todo1.compareTo(todo2)<0);

      // Vérifie que l'inverse est vrai aussi
      assertTrue(todo2.compareTo(todo1)>0);
   }

   public void testEquals() {
      assertEquals(todo1,todo3);
      assertFalse(todo1.equals(todo2));
   }
}
```

Les suites de tests

Les cas de test sont généralement très nombreux pour un logiciel. Afin de simplifier le lancement de ces différents tests, il peut être intéressant de les regrouper dans un ou plusieurs ensembles permettant de commander leur exécution de manière collective.

Dans JUnit, de tels ensembles sont appelés des suites de tests. Une suite de tests est définie grâce à la classe `junit.framework.TestSuite` du framework. Pour cela, il suffit de créer une instance de cette classe et d'utiliser ses méthodes `addTest` et `addTestSuite`.

La méthode *addTest*

La méthode `addTest` est utilisée pour ajouter un test unique à la suite, c'est-à-dire une méthode `test` particulière. Par exemple, pour `TodoECTest`, si nous désirons créer une suite exécutant les tests des méthodes `compareTo` et `equals`, nous écrivons le code suivant :

```
TestSuite suite = new TestSuite("Test de compareTo et equals") ;
suite.addTest(new TodoECTest("testCompareTo")) ;
suite.addTest(new TodoECTest("testEquals")) ;
```

Nous constatons que nous utilisons le constructeur de `TodoECTest` pour sélectionner le test à inclure dans la suite, d'où l'importance de le définir, faute de quoi il n'est pas possible d'inclure spécifiquement une des méthodes du cas de test.

La méthode *addTestSuite*

La méthode `addTestSuite` permet d'inclure automatiquement tous les tests contenus dans un cas de test.

Supposons que nous ayons deux cas de test, `TodoECTest` et `TodoListTest`. Le code suivant montre comment créer une suite permettant de regrouper la totalité de leurs tests dans une seule et même suite :

```
TestSuite suite = new TestSuite("Tests de TodoECTest et TodoListTest");
suite.addTestSuite(TodoECTest.class);
suite.addTestSuite(TodoListTest.class);
```

Encapsulation d'une suite

Pour exécuter une suite de tests, il est nécessaire de créer une classe, qui sera utilisée par un des lanceurs de JUnit. Cette classe doit comporter une méthode statique sans paramètre, appelée `suite`, renvoyant un objet de type `junit.framework.Test` (il s'agit d'une interface implémentée notamment par `junit.framework.TestSuite`).

Pour Tudu Lists, nous avons défini la classe `AllTests` dans le package `tudu.testsuite` pour exécuter en un seul appel l'ensemble des tests unitaires du projet :

```
package tudu.testsuite;

(...)

public class AllTests {
```

```
    public static Test suite() {
       TestSuite suite = new TestSuite("Tests for tudu.*");
       suite.addTestSuite(TodoECTest.class);
       suite.addTestSuite(AuthenticationDAOImplTest.class);
       suite.addTestSuite(ConfigurationManagerImplTest.class);
       suite.addTestSuite(TodoListsManagerImplTest.class);
       (...)
       return suite;
    }
}
```

Exécution des tests

Une fois les cas de test et les suites de tests définis, il est nécessaire de les exécuter pour vérifier le logiciel.

Les lanceurs standards de JUnit

Comme expliqué précédemment, JUnit propose plusieurs lanceurs standards (TestRunners) pour exécuter les cas de test et les suites de tests :

- lanceur en mode texte ;
- lanceur en mode graphique reposant sur la bibliothèque AWT ;
- lanceur en mode graphique reposant sur la bibliothèque Swing.

Pour utiliser le lanceur en mode texte dans un cas de test (ici la classe `TodoECTest`), il suffit d'y ajouter une méthode `main` de la manière suivante :

```
public static void main(String[] args) {
    junit.textui.TestRunner.run(TodoECTest.class);
}
```

La classe `junit.textui.TestRunner` est appelée en lui passant en paramètre la classe du cas de test (et non une instance) à exécuter. Cette méthode `main` peut être écrite soit directement dans la classe du cas de test, comme dans cet exemple, soit dans une classe spécifique.

Si nous exécutons `TodoECTest`, nous obtenons le résultat suivant dans la console Java :

```
..
Time: 0,091

OK (2 tests)
```

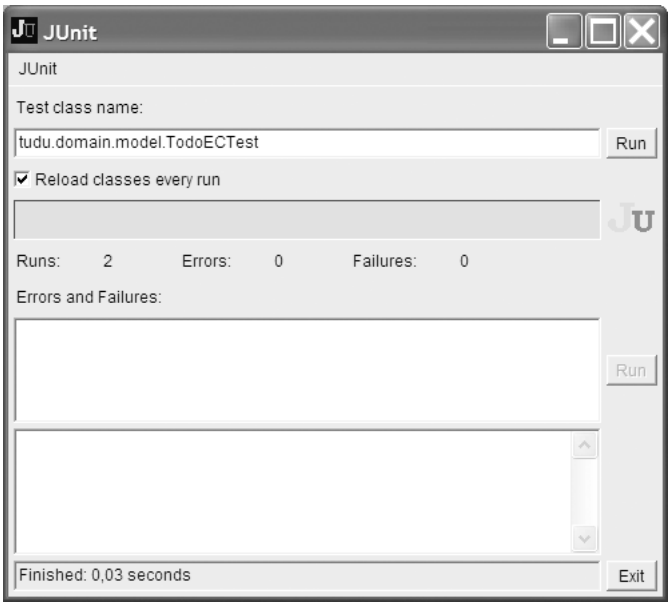
Ce résultat indique de manière laconique que les deux tests définis dans `TodoECTest` se sont bien exécutés.

Pour utiliser le lanceur fondé sur AWT, il faut coder la méthode `main` comme ceci :

```
public static void main(String[] args) {
    junit.awtui.TestRunner.run(TodoECTest.class);
}
```

Si nous exécutons `TodoECTest`, la fenêtre illustrée à la figure 17.1 s'affiche.

Figure 17.1
Le lanceur fondé sur AWT



De la même manière, pour utiliser le lanceur fondé sur Swing, il suffit de modifier légèrement le code de la méthode `main` précédent (changement en gras) :

```
public static void main(String[] args) {
    junit.swingui.TestRunner.run(TodoECTest.class);
}
```

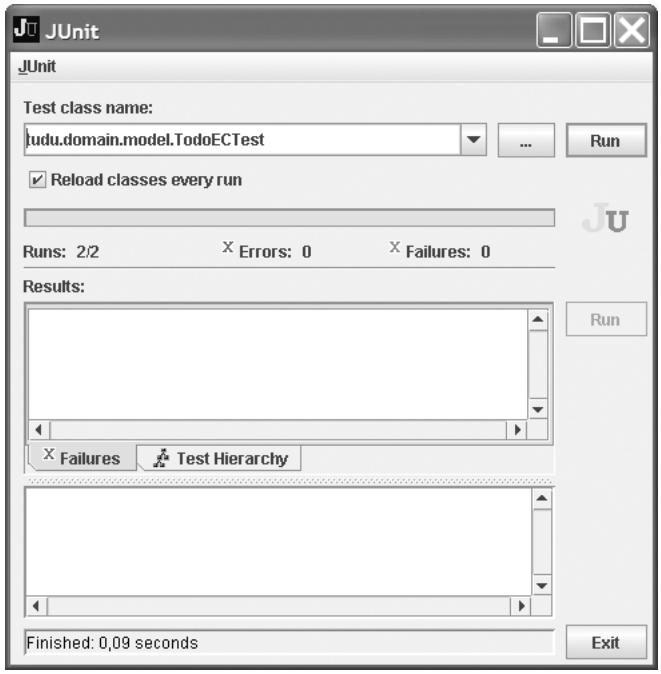
Si nous exécutons `TodoECTest`, la fenêtre illustrée à la figure 17.2 s'affiche.

Nous pouvons constater que le lanceur Swing est plus sophistiqué graphiquement que celui fondé sur AWT (présence d'onglets pour visionner les échecs et la hiérarchie des tests).

Pour exécuter une suite de tests, la démarche est similaire. Il faut créer une méthode `main` utilisant la méthode `suite`. Pour la suite `AllTests` créée précédemment, elle se présente de la manière suivante :

```
public static void main(String[] args) {
    junit.textui.TestRunner.run(AllTests.suite());
}
```

Figure 17.2
Le lanceur fondé sur Swing



Le lanceur prend comme paramètre le résultat de la méthode `suite` (en gras dans le code) pour connaître les tests à exécuter. Cette méthode `main` peut être définie soit dans la même classe que la méthode `suite`, soit dans une classe spécifique.

Le lanceur JUnit intégré à Eclipse

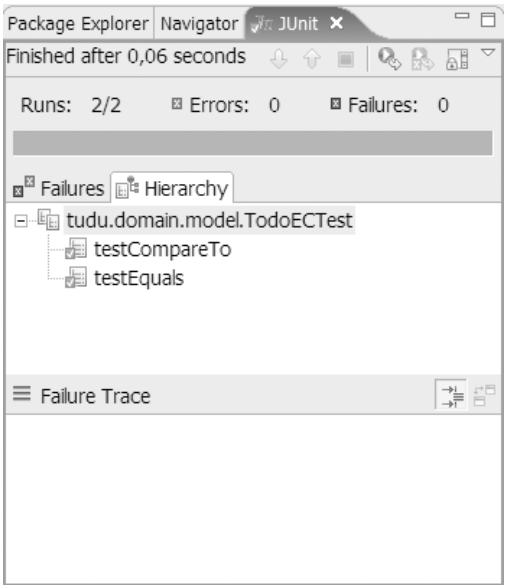
Eclipse propose son propre lanceur JUnit, parfaitement intégré à l'environnement de développement.

Pour l'utiliser, il n'est pas nécessaire de créer une méthode `main` spécifique dans les cas de test, à la différence des lanceurs standards. Il suffit de sélectionner l'explorateur de package et le fichier du cas de test ou de la suite par clic droit et de choisir Run dans le menu contextuel. Parmi les choix proposés par ce dernier, il suffit de sélectionner JUnit Test.

Une vue JUnit s'ouvre alors pour nous permettre de consulter le résultat de l'exécution des tests. Si tel n'est pas le cas, nous pouvons l'ouvrir en choisissant Window puis Show view et Other. Une liste hiérarchisée s'affiche, dans laquelle il suffit de sélectionner JUnit dans le dossier Java et de cliquer sur le bouton OK *(voir figure 17.3)*.

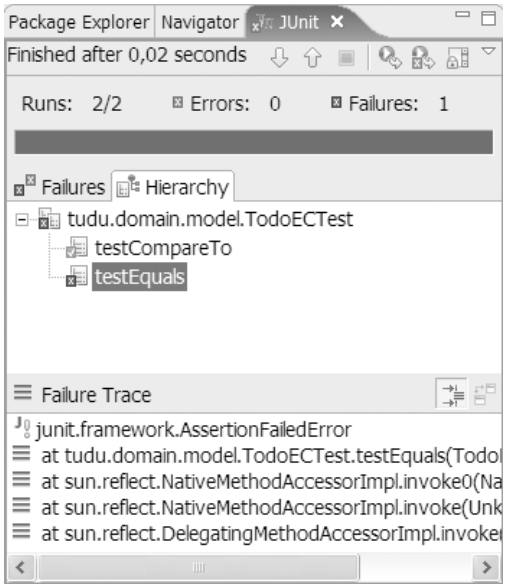
L'intérêt de ce lanceur réside dans sa gestion des échecs après l'exécution des tests.

Figure 17.3
Vue JUnit intégrée à Eclipse



Si nous modifions `TodoECTest` de manière que deux tests échouent (il suffit pour cela de mettre une valeur attendue absurde, qui ne sera pas respectée par la classe testée), nous obtenons dans le lanceur le résultat illustré à la figure 17.4.

Figure 17.4
Gestion des échecs dans la vue JUnit d'Eclipse



Les échecs *(failures)* sont indiqués par une croix bleue et les succès par une marque verte. En cliquant sur un test ayant échoué, la trace d'exécution est affichée dans la zone Failure Trace. En double-cliquant sur le test, son code source est immédiatement affiché dans l'éditeur de code d'Eclipse.

À partir des résultats des tests, le développeur peut naviguer directement dans son code et le corriger dans Eclipse, ce qui n'est pas possible avec les lanceurs standards de JUnit.

En résumé

L'écriture de tests avec JUnit n'est pas intrinsèquement compliquée. Cependant, il faut garder à l'esprit que cette simplicité d'utilisation dépend fortement de la façon dont l'application est conçue. La mise en œuvre de tests unitaires dans une modélisation naïve, à l'image de celle que nous avons présentée au chapitre 2, s'avère très difficile (d'où notre critère d'invérifiabilité).

L'utilisation de JUnit est rendue efficace grâce aux principes architecturaux que doivent appliquer les applications Spring, à savoir l'inversion de contrôle apportée par le conteneur léger et la séparation claire des couches de l'application. Grâce à ces principes, les composants se révèlent plus simples à tester, car ils se concentrent dans la mesure du possible sur une seule préoccupation.

En complément de JUnit, il est nécessaire d'utiliser des simulacres d'objets afin d'isoler des contingences extérieures le composant à tester unitairement. C'est ce que nous allons voir à la section suivante.

Les simulacres d'objets

Le framework JUnit constitue un socle pour réaliser des tests, mais il n'offre aucune fonctionnalité spécifique pour tester les relations entre les différents objets manipulés lors d'un test. Il est de ce fait difficile d'isoler les dysfonctionnements de l'objet testé de ceux qu'il manipule, ce qui n'est pas souhaitable lorsque nous réalisons des tests unitaires, par exemple.

Afin de combler ce vide, il est possible d'utiliser des simulacres d'objets *(mock objects)*. Comme leur nom l'indique, ces simulacres simulent le comportement d'objets réels. Il leur suffit pour cela d'hériter de la classe ou de l'interface de l'objet réel et de surcharger chaque méthode publique utile au test. Le comportement de ces méthodes est ainsi redéfini selon un scénario conçu spécifiquement pour le test unitaire et donc parfaitement maîtrisé.

Par exemple, si nous testons un objet faisant appel à un DAO, nous pouvons remplacer ce dernier par un simulacre. Ce simulacre ne se connectera pas à la base de données mais renverra des données statiques spécialement définies pour le test unitaire. Nous isolons de la sorte l'objet testé des contingences spécifiques au DAO (connexion à la base de données, etc.).

Les simulacres d'objets avec EasyMock

Plusieurs frameworks permettent de créer facilement ces simulacres au lieu de les développer manuellement. Pour les besoins de l'ouvrage, nous avons choisi EasyMock, qui est l'un des plus simples d'utilisation.

EasyMock est capable de créer des simulacres à partir d'interfaces. Une extension permet d'en fournir à partir de classes, mais nous ne l'abordons pas ici, puisque, dans le cas d'applications Spring, nous utilisons essentiellement des interfaces.

Cette section n'introduit que les fonctionnalités principales d'EasyMock. Nous utilisons la version 2, qui fonctionne avec Java 5. Pour plus d'informations, voir le site Web dédié au framework, à l'adresse *http://www.easymock.org.*

Les simulacres bouchons

Les simulacres les plus simples sont les bouchons, ou *stubs.* Ils consistent à définir, à partir d'une interface ou d'une classe de l'application, une implémentation simulant le comportement d'un objet réel. Cette simulation consiste généralement, pour chaque méthode implémentée, à renvoyer des valeurs prédéfinies.

Supposons que nous désirions tester de façon unitaire la classe `ConfigurationManagerImpl` en charge de la configuration des paramètres de l'application Tudu Lists. Cette classe possède une dépendance vis-à-vis de l'interface `PropertyDAO`. Pour pouvoir l'isoler des contingences liées à l'implémentation fournie par Tudu Lists de cette interface, il est nécessaire de créer un simulacre dont nous contrôlons très exactement le comportement.

Avant de programmer notre simulacre, définissons rapidement son comportement au moyen de la méthode `getProperty`, seule utile pour notre test :

- Si le paramètre de `getProperty` vaut « key », renvoyer un objet `Property` dont l'attribut `key` vaut « key » et dont l'attribut `value` vaut « value ».
- Dans tous les autres cas, renvoyer par défaut un objet `Property` dont les attributs `key` et `value` valent tous les deux « default ».

Maintenant que nous avons défini les comportements du simulacre, nous pouvons le définir au sein d'un cas de test :

```
package tudu.service.impl;

import static org.easymock.EasyMock.*;← ❶

import junit.framework.TestCase;
import tudu.domain.dao.PropertyDAO;
import tudu.domain.model.Property;

public class ConfigurationManagerImplECTest extends TestCase {

   private PropertyDAO propertyDAO = null;
   private ConfigurationManagerImpl configurationManager = null;
```

```
    protected void setUp() throws Exception {
       propertyDAO = createMock(PropertyDAO.class);← ❷
       configurationManager = new ConfigurationManagerImpl();
       configurationManager.setPropertyDAO(propertyDAO);← ❸
    }

    public void testGetProperty() {
       Property property = new Property();
       property.setKey("key");
       property.setValue("value");
       expect(propertyDAO.getProperty("key"))
          .andStubReturn(property);← ❹

       Property defaultProperty = new Property();
       defaultProperty.setKey("default");
       defaultProperty.setValue("default");
       expect(propertyDAO.getProperty((String)anyObject()))
          .andStubReturn(defaultProperty);← ❺

       replay(propertyDAO);← ❻

       Property test = configurationManager.getProperty("key");
       assertEquals("value", test.getValue());
       test = configurationManager.getProperty("anything");
       assertEquals("default",test.getValue());

       verify(propertyDAO);← ❶
    }
}
```

Pour simplifier l'écriture du code nécessaire à la définition des simulacres, des imports statiques, nouveauté introduite avec Java 5, sont utilisés (repère ❶). Les imports statiques permettent de ne plus avoir à spécifier la classe (en l'occurrence la classe `org.easymock.EasyMock`) lors des appels à ses méthodes statiques.

Nous avons redéfini la méthode `setUp`, héritée de `TestCase`, afin de créer le simulacre et de l'injecter manuellement dans le manager. La création du simulacre consiste à créer une instance d'objet implémentant l'interface `PropertyDAO` en utilisant la méthode `createMock` (repère ❷). Cette méthode prend en paramètre l'interface que le simulacre doit implémenter, en l'occurrence `PropertyDAO`.

Une fois le simulacre créé, nous l'injectons manuellement dans le manager, puisque nous sommes dans le cadre de tests unitaires, c'est-à-dire sans conteneur (repère ❸).

Pour que le simulacre fonctionne, il est nécessaire de spécifier le comportement de chacune de ses méthodes publiques utilisées dans le cadre du test. C'est ce que nous faisons au début de la méthode `testGetProperty` (repères ❹ et ❺) en utilisant la méthode statique `expect` de la classe `EasyMock`.

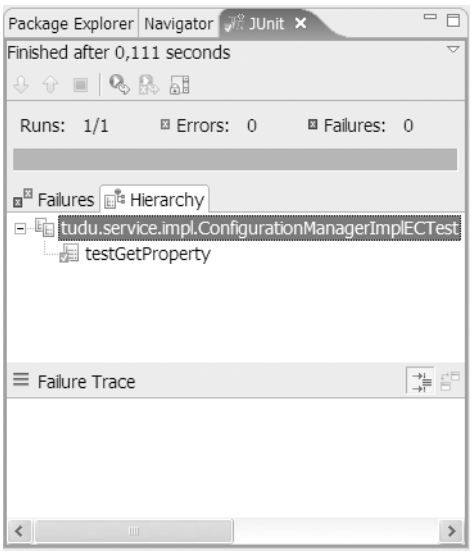
Au repère ❹, nous spécifions le comportement de la méthode `getProperty` de `propertyDAO` lorsque celle-ci a comme paramètre « key », c'est-à-dire la valeur que la méthode doit renvoyer, en l'occurrence l'objet `property`. Pour cela, nous passons en paramètre à `expect` l'appel à `getProperty` proprement dit. Nous utilisons ensuite la méthode `andStubReturn` de l'objet renvoyé par `expect` pour spécifier la valeur de retour correspondant au paramètre « key ».

Au repère ❺, nous spécifions le comportement par défaut de `getProperty`, c'est-à-dire quel que soit son paramètre. Pour indiquer que le paramètre attendu peut être n'importe lequel, nous utilisons la méthode `anyObject` d'EasyMock. La valeur de retour est, là encore, spécifiée avec la méthode `andStubReturn`.

Pour pouvoir exécuter le test unitaire proprement dit, il suffit d'appeler la méthode `replay` en lui passant en paramètre le simulacre (repère ❻). Nous indiquons ainsi à EasyMock que la phase d'enregistrement du comportement du simulacre est terminée et que le test commence. Une fois le test terminé, un appel à la méthode `verify` (repère ❼) permet de vérifier que le simulacre a été correctement utilisé (toutes les méthodes ont-elles bien été appelées, par exemple ?).

Si nous exécutons notre test unitaire, nous constatons que tout se déroule sans accroc, comme l'illustre la figure 17.5.

Figure 17.5
Exécution du cas de test avec bouchon



Notons que si une méthode dont le comportement n'est pas défini est appelée, une erreur est générée. Il est possible d'affecter un comportement par défaut à chaque méthode en utilisant la méthode `createNiceMock` au lieu de `createMock`. Son intérêt est toutefois limité, car ce comportement consiste à renvoyer `0`, `false` ou `null` comme valeur de retour.

Enfin, pour simuler une méthode sans valeur de retour (`void`), il est inutile d'utiliser `expect`, un simple appel étant suffisant.

Les simulacres avec contraintes

Le framework EasyMock permet d'aller plus loin dans les tests d'une classe en spécifiant la façon dont les méthodes du simulacre doivent être utilisées.

Au-delà de la simple définition de méthodes bouchons, il est possible de définir des contraintes sur la façon dont elles sont utilisées, à savoir le nombre de fois où elles sont appelées et l'ordre dans lequel elles le sont.

Définition du nombre d'appels

EasyMock permet de spécifier pour chaque méthode bouchon des attentes en terme de nombre d'appels. La manière la plus simple de définir cette contrainte consiste à ne pas définir de valeur par défaut, autrement dit à supprimer le code au niveau du repère ❺ et à remplacer la méthode `andStubReturn` par la méthode `andReturn`.

EasyMock s'attend alors à ce qu'une méthode soit appelée une seule fois pour une combinaison de paramètres donnée. Si, dans l'exemple précédent, après ces modifications, nous appelons la méthode `getProperty` avec le paramètre « key » deux fois dans notre test, une erreur d'exécution est générée, comme l'illustre la figure 17.6.

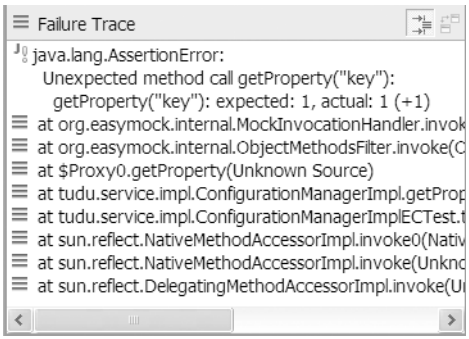


Figure 17.6
Appel inattendu à la méthode getProperty

L'erreur affichée indique que l'appel à la méthode `getProperty` est inattendu *(unexpected)*. Le nombre d'appel de ce type attendu *(expected)* et effectif *(actual)* est précisé. En l'occurrence, le nombre attendu d'appel est 1, et le nombre effectif 2 (1 + 1).

De même, si la méthode `getProperty` n'est pas appelée avec le paramètre « key », une erreur est générée puisque EasyMock s'attend à ce qu'elle soit appelée une fois de cette manière.

Pour définir le nombre de fois où une méthode doit être appelée, nous disposons des quatre méthodes suivantes, à utiliser sur le résultat produit par `andReturn` :

- `times(int nbre)` : spécifie un nombre d'appels exact.
- `times(int min,int max)` : spécifie un nombre d'appels borné.

- `atLeastOnce` : spécifie que la méthode doit être appelée au moins une fois.
- `anyTimes` : le nombre d'appels est quelconque (y compris 0).

Le code ci-dessous spécifie un nombre d'appels égal à 2 :

```
expect(propertyDAO.getProperty("key"))
   .andReturn(property).times(2);
```

Si nous exécutons notre test appelant deux fois `getProperty("key")` avec ce nouveau simulacre, nous n'obtenons plus d'erreur.

Définition de l'ordre d'appel des méthodes

L'ordre d'appel des méthodes peut être important pour tester la façon dont un objet manipule les autres. Avec EasyMock, nous pouvons définir l'ordre dans lequel les méthodes d'un simulacre doivent être appelées.

Pour tenir compte de l'ordre, il suffit de remplacer la méthode `createMock` du contrôleur par `createStrictMock`. L'ordre d'appel des méthodes est alors défini par la séquence de leur appel pour la définition du simulacre.

Pour tester l'effet de cette modification, modifions le code de notre test. Juste après le repère ❺, définissons le comportement de `getProperty` avec le paramètre « another » :

```
anotherProperty.setKey("another");
anotherProperty.setValue("something");
expect(propertyDAO.getProperty("another"))
  .andReturn(anotherProperty);
```

Introduisons ensuite le test suivant juste après le premier test de la méthode `getProperty` :

```
test = configurationManager.getProperty("another");
    assertEquals("something", test.getValue());
```

Si nous effectuons cette modification sur le code précédent sans autre modification, nous obtenons le résultat illustré à la figure 17.7.

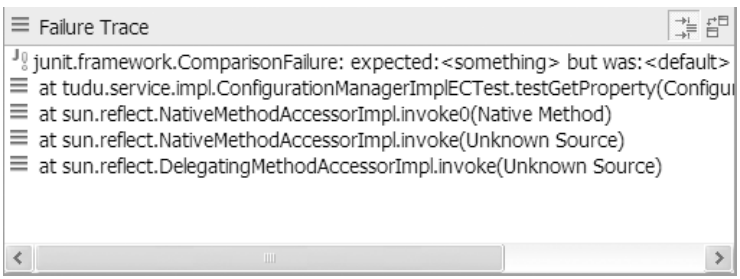


Figure 17.7
Résultat du non-respect de l'ordre d'appel

L'erreur indique que la valeur de retour attendue était celle du comportement par défaut, puisque le comportement avec le paramètre « another » a été défini après le comportement avec le paramètre « key », qui n'a pas encore été appelé.

Les simulacres d'objets de Spring

Spring n'étant pas un framework dédié aux tests unitaires, il ne fournit pas de services similaires à ceux d'EasyMock. Cependant, afin de faciliter l'écriture de tests unitaires, Spring fournit un ensemble de simulacres prêts à l'emploi. Ces simulacres simulent des composants standards de l'API J2EE souvent utilisés dans les applications.

L'ensemble de ces simulacres est défini dans des sous-packages de `org.springframework.mock`.

Les simulacres JNDI

Dans le package `org.springframework.mock.jndi`, sont définis les simulacres nécessaires à la simulation d'un annuaire JNDI. Ces simulacres sont notamment utiles lorsqu'un composant doit obtenir *via* un annuaire la référence à un autre objet, typiquement une source de données.

Le code suivant montre comme ces simulacres sont utilisés pour définir un annuaire référençant une source de données :

```
import org.springframework.jdbc.datasource.DriverManagerDataSource;
import org.springframework.mock.jndi.SimpleNamingContextBuilder;
(...)

SimpleNamingContextBuilder builder =
   new SimpleNamingContextBuilder();

DataSource ds = new DriverManagerDataSource("org.hsqldb.jdbcDriver"
   ," jdbc:hsqldb:mem:tudu ","sa","");

builder.bind("java:comp/env/jdbc/tudu", ds);

try {
   builder.activate();
}
catch(NamingException e) {
   throw new RuntimeException("Problème dans l'activation JNDI");
}
```

Dans cet exemple, nous associons une source de données (`ds`) à la clé `java:comp/env/jdbc/tudu`. Ce code est utilisé dans la méthode `launch` de la classe `HsqldbLauncher` du package `tudu.domain.dao.hibernate3`. Cette classe a la charge de créer en mémoire la base de données qui est utilisée pour nos tests.

Les simulacres Web

Dans le package `org.springframework.mock.web`, sont définis les simulacres nécessaires pour tester des composants Web. Nous y trouvons des simulacres pour les principales interfaces de l'API servlet, notamment les suivants :

- HttpServletRequest

- HttpServletResponse
- HttpSession

Dans Tudu Lists, nous disposons d'une servlet pour effectuer la sauvegarde du contenu d'une liste de todos. Le code suivant, extrait de la classe `BackupServletTest` du package `tudu.web.servlet`, montre comment les simulacres Web `MockHttpServletRequest`, `MockHttpServletResponse` et `MockHttpSession` sont utilisés pour tester cette servlet dans un cas de test JUnit classique :

```
public void testDoGet() throws Exception {
   Document doc = new Document();
   Element todoListElement = new Element("todolist");
   todoListElement.addContent(
     new   Element("title").addContent("Backup List"));
   doc.addContent(todoListElement);

   MockHttpServletRequest request = new MockHttpServletRequest();
   MockHttpSession session = new MockHttpSession();
   session.setAttribute("todoListDocument", doc);
   request.setSession(session);

   MockHttpServletResponse response =
     new MockHttpServletResponse();

   BackupServlet backupServlet = new BackupServlet();
   backupServlet.doGet(request, response);

   String xmlContent = response.getContentAsString();

   assertTrue(xmlContent.indexOf("<title>Backup List</title>")>0);
}
```

Le code en gras, qui concentre l'utilisation des simulacres, montre que l'emploi de ces derniers est très aisé. Leur création ne nécessite aucun paramétrage particulier, et toutes les méthodes pour définir leur contenu puis le récupérer sont disponibles.

Autres considérations sur les simulacres

Pour terminer cette section consacrée aux simulacres d'objets, il nous semble important d'insister sur deux points fondamentaux pour bien utiliser les simulacres :

- Un cas de test portant sur un objet utilisant des simulacres ne doit pas tester ces derniers. L'objectif des simulacres est non pas d'être testés, mais d'isoler un objet des contingences extérieures afin de faciliter sa vérification.
- Un simulacre doit être conçu de manière à produire des données susceptibles de générer des erreurs dans l'objet à tester. Le passage des tests ne démontre pas qu'un objet fonctionne correctement, mais démontre seulement qu'il a su passer les tests.

EasyMock répond à des besoins moyennement complexes en terme de simulacres. Pour des besoins plus complexes, il est conseillé de développer spécifiquement les simulacres sans l'aide d'un framework.

En résumé

Comme pour les tests avec JUnit, les principes architecturaux de Spring facilitent la création de simulacres. D'une part, la séparation des interfaces et de leurs implémentations permet d'utiliser nativement EasyMock sans passer par une extension permettant de réaliser des simulacres à partir de classes. D'autre part, l'injection des dépendances dans le composant pouvant être faite manuellement, donc sans le conteneur léger, il est aisé d'initialiser les collaborateurs du composant à tester, non pas avec des implémentations, mais avec des simulacres permettant de l'isoler des contingences extérieures.

Les simulacres prêts à l'emploi fournis par Spring sont très utiles en ce qu'ils simulent des classes standards de l'API Java, très utilisées dans les applications J2EE.

Les tests d'intégration

JUnit et EasyMock suffisent dans la plupart des cas à réaliser des tests unitaires. Les tests unitaires sont nécessaires mais pas suffisants pour tester le bon fonctionnement d'une application. Les tests d'intégration viennent en complément pour garantir que l'intégration entre les composants de l'application s'effectue correctement.

Dans cette section, nous mettrons en œuvre aussi bien le conteneur léger que la base de données, voire un simulacre de conteneur de servlets (pour tester les actions Struts, par exemple). Pour cela, nous utilisons, d'une part, les extensions de Spring à la classe `TestCase` de JUnit, afin de faire fonctionner le conteneur léger dans le cadre des tests d'intégration, et, d'autre part, le framework StrutsTestCase pour tester les actions Struts.

Les extensions de Spring pour JUnit

Comme nous l'avons déjà indiqué, JUnit peut être utilisé aussi bien pour faire des tests unitaires que pour faire des tests d'intégration. C'est donc tout à fait naturellement que Spring fournit des extensions à JUnit afin de pouvoir tester l'intégration des composants de l'application. Ces extensions ont pour vocation d'instancier le conteneur léger afin de bénéficier de l'injection de dépendances et de la POA.

Ces extensions se présentent sous la forme de classes dérivées de `junit.framework.TestCase`, contenue dans le package `org.springframework.test`. Spring définit quatre classes abstraites permettant d'utiliser le conteneur léger et la POA.

Ces classes apportent une réponse graduée aux besoins des tests d'intégration :

- `AbstractSpringContextTests`. Classe de base des trois autres, elle définit leur fonctionnement commun et ne doit normalement pas être utilisée pour implémenter un cas de test.

- `AbstractDependencyInjectionSpringContextTests`. Permet d'utiliser le conteneur léger et l'injection de dépendances. Pour cela, les fichiers de configuration sont spécifiés en surchargeant la méthode `getConfigLocations` *(voir l'exemple donné plus loin).*
- `AbstractTransactionalSpringContextTests`. Permet d'utiliser le conteneur léger dans un cadre transactionnel. Cette classe offre les mêmes services que précédente, mais à la fin des tests, un rollback global est effectué pour annuler les modifications persistantes induites par le test.
- `AbstractTransactionalDataSourceSpringContextTests`. Offre les mêmes services que la précédente et donne la possibilité de vider complètement une ou plusieurs tables de la base lors des tests.

Ces classes utilisent uniquement l'injection automatique des dépendances en se fondant sur les types. Il faut donc prendre en compte ce mode de fonctionnement dans la réalisation des tests.

Pour démontrer l'utilité de ces extensions, nous allons créer un test d'intégration pour un des DAO de Tudu Lists, en l'occurrence `UserDAOHibernate`. Il s'agit d'un test d'intégration, car la base de données est mise en œuvre.

À cette fin, nous allons créer une sous-classe de `AbstractTransactionalSpringContext-Tests`, car cette classe est en mesure de rendre la base de données dans son état initial après les tests. Le test s'exécute en effet dans un contexte transactionnel sur lequel la classe effectue un rollback à la fin.

Le code suivant montre notre cas de test `UserDAOHibernateTest`, qui utilise les services de `AbstractTransactionalSpringContextTests` :

```
package tudu.domain.dao.hibernate3;

(...)

public class UserDAOHibernateTest extends
          AbstractTransactionalSpringContextTests {



   static {←❶
      HsqldbLauncher.launch();
   }

   private User newUser = null;
   private UserDAO dao = null;

   public void setDao(UserDAO dao) {←❷
      this.dao = dao ;
   }


   protected String[] getConfigLocations() {←❸
```

```
        String webappDir = System.getProperty("TUDU_WEBAPP_DIR");
        if (webappDir == null) {
           webappDir = "WebContent";
        }

      return new String[] {
         "file:" + webappDir + "/WEB-INF/applicationContext.xml",
         "file:" + webappDir
                 + "/WEB-INF/applicationContext-hibernate.xml" };
    }

    protected void onSetUpBeforeTransaction() throws Exception {←❹
       super.onSetUpBeforeTransaction();
       newUser = new User();
       newUser.setEmail("email@test.com");
       newUser.setEnabled(true);
       newUser.setLastName("lastName");
       newUser.setLogin("testUser");
       newUser.setPassword("password");
       HashSet<Role> roles = new HashSet<Role>();
       Role role = new Role();
       role.setRole(RolesEnum.ROLE_USER.toString());
       roles.add(role);
       newUser.setRoles(roles);
    }

    protected void onSetUpInTransaction() throws Exception {←❺
       super.onSetUpInTransaction();
    }

    public void testSaveUser(){←❻
       dao.saveUser(newUser);
       User user = dao.getUser("testUser");
       assertEquals(newUser.getLastName(),user.getLastName());
    }

    public void testUpdateUser(){←❼
      dao.saveUser(newUser);
      newUser.setLastName("newLastName");
      dao.updateUser(newUser);
      User updatedUser = dao.getUser("testUser");
      assertEquals(newUser.getLastName(),updatedUser.getLastName());
    }
}
```

Ce cas de test comprend plusieurs éléments importants nécessaires à la réalisation des tests d'intégration de notre DAO. Tout d'abord, la classe comporte un bloc statique (repère ❶), qui initialise la source de données en utilisant les services de la classe `HsqldbLauncher`.

L'attribut `dao` contient l'instance de la classe à tester. Cet attribut est initialisé par le conteneur léger par le biais de sa fonctionnalité d'injection automatique *via* le modificateur `setDao` (repère ❷).

En surchargeant la méthode `getConfigLocations` (repère ❸), nous définissons les fichiers de configuration du conteneur léger dans le cadre du cas de test. Ces fichiers de configuration doivent être accessibles par défaut depuis le répertoire **WebContent\WEB-INF.** Il est possible de remplacer **WebContent** par un autre répertoire spécifié dans la propriété système `TUDU_WEBAPP_DIR`. Nous utilisons ici les mêmes fichiers de configuration que l'application, car ils n'ont pas besoin d'être adaptés spécifiquement pour les tests.

Contrairement à la classe `TestCase` définie par JUnit, `AbstractTransactionalSpringContextTests` fournit non pas une, mais deux méthodes d'initialisation des tests. La première, `onSetUpBeforeTransaction` (repère ❹), est destinée aux initialisations pouvant s'effectuer en dehors du contexte transactionnel. Il s'agit donc de traitements n'ayant pas d'impact sur la persistance des données. Dans notre exemple, il s'agit des variables nécessaires aux tests. La seconde, `onSetUpInTransaction` (repère ❺), est destinée aux initialisations devant s'effectuer dans le contexte transactionnel. Dans les deux cas, il est nécessaire d'appeler la méthode de la classe mère *via* `super`, comme avec la méthode `setUp` de la classe `TestCase`.

Grâce à l'ensemble de ces initialisations, nous pouvons tester la classe `UserDAO` en l'utilisant comme à l'intérieur des services de Tudu Lists (repères ❻ et ❼). Comme nous pouvons le constater, les tests modifient le contenu de la base de données. Fort heureusement, la classe `AbstractTransactionalSpringContextTests` effectue un rollback à la fin des tests, ce qui a pour effet de rendre à la base son état initial.

Utilisation de StrutsTestCase avec Spring

StrutsTestCase est une extension de JUnit pour tester le code utilisant le framework Struts. StrutsTestCase supporte les spécifications servlet 2.2, 2.3 et 2.4, ainsi que la version 1.2 de Struts. Ce framework est disponible à l'adresse *http://strutstestcase.sourceforge.net.*

Nous ne présentons ici de ce framework que l'essentiel pour tester Tudu Lists. Le lecteur désirant en savoir plus peut se référer à la documentation fournie sur le site Web de StrutsTestCase.

Cette extension se présente sous la forme de deux classes dérivant de `TestCase`. La première, `MockStrutsTestCase`, offre un simulacre de conteneur de servlets à même d'exécuter le code fondé sur Struts. Ce simulacre a l'avantage d'être beaucoup plus léger qu'un véritable conteneur de servlets, comme Tomcat. La contrepartie est un support partiel de l'API servlet. La seconde, `CactusStrutsTestCase`, utilise le framework Cactus de la communauté Apache Jakarta pour exécuter le code de test directement dans un véritable conteneur comme Tomcat. Avec cette deuxième classe, toutes les fonctionnalités de l'API servlet sont supportées.

Nous présentons ici la version avec simulacre, sachant que la version avec Cactus ne présente pas de différence significative pour la programmation des tests unitaires.

Pour utiliser StrutsTestCase avec Spring, le répertoire **WEB-INF** doit être accessible depuis le classpath. En effet, StrutsTestCase interprète directement les fichiers de configuration XML, principalement **struts-config.xml.** Il est aussi possible d'indiquer à StrutsTestCase l'emplacement exact des différents fichiers dont il a besoin. C'est ce que nous réalisons dans la classe `TuduBaseMockStrutsTestCase` du package `tudu.web` :

```
public class TuduBaseMockStrutsTestCase
  extends MockStrutsTestCase {

    protected void setUp() throws Exception {
        super.setUp();
        (...)

        String webappDir = System.getProperty("TUDU_WEBAPP_DIR");
        if (webappDir == null) {
            webappDir = "WebContent";
        }
        File webXml = new File(webappDir + "/WEB-INF/web.xml");
        if (!webXml.exists()) {
            fail("web.xml not found.");
        }
        File context = new File(webappDir);
        if (!context.exists()) {
            fail("context directory not found.");
        }
        File strutsXml = new File(webappDir + "/WEB-INF/struts-config.xml");
        if (!strutsXml.exists()) {
            fail("struts-config.xml not found.");
        }
        this.setServletConfigFile(webXml.getAbsolutePath());
        this.setConfigFile(strutsXml.getAbsolutePath());
        this.setContextDirectory(context);
    }
}
```

Cette classe est la mère de tous nos cas de test fondés sur StrutsTestCase.

Notons qu'une limitation de StrutsTestCase empêche d'utiliser le symbole `*` dans la définition des fichiers de configuration Spring du plug-in `org.springframework.web.struts.ContextLoaderPlugIn`. Il est donc nécessaire de lister nominativement chaque fichier de configuration à charger par Spring.

Pour Tudu Lists, le fichier **struts-config.xml** doit contenir la définition suivante :

```
<plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">

<set-property property="contextConfigLocation" value="/WEB-INF/application-
➥Context-dwr.xml,/WEB-INF/applicationContext-hibernate.xml,/WEB-INF/
➥applicationContext-security.xml,/WEB-INF/applicationContext.xml,/WEB-INF/
➥action-servlet.xml"/>

</plug-in>
```

Pour terminer, notons que le framework de sécurité Acegi Security utilisé pour Tudu Lists repose sur un filtre servlet. Or, comme nous l'avons déjà indiqué, le simulacre de StrutsTestCase ne supporte pas les filtres. Cela a pour conséquences de ne pas charger le contexte de sécurité et d'empêcher l'exécution de l'application. Il est donc nécessaire de charger manuellement un contexte de sécurité pour permettre les tests.

Le cas de test suivant montre comment tester l'action Struts `tudu.web ShowTodosAction` en dérivant la classe `MockStrutsTestCase` :

```
package tudu.web;

import org.acegisecurity.GrantedAuthority;
import org.acegisecurity.context.SecurityContextHolder;
import org.acegisecurity.context.SecurityContextImpl;
import org.acegisecurity.providers.TestingAuthenticationToken;
import org.acegisecurity.userdetails.User;

import servletunit.struts.MockStrutsTestCase;

public class ShowTodosActionTest extends TuduBaseMockStrutsTestCase {


   protected void setUp() throws Exception {← 1
      super.setUp();

      SecurityContextImpl ctx = new SecurityContextImpl();
      GrantedAuthority authority = new GrantedAuthority() {
         public String getAuthority() {
            return "ROLE_ADMIN";
         }
      };
      User user = new User("admin", "admin", true, true, true
                     ,false,new GrantedAuthority[] { authority });
      TestingAuthenticationToken token =
         new TestingAuthenticationToken(user, null
                     , new GrantedAuthority[] { authority });
      ctx.setAuthentication(token);
      SecurityContextHolder.setContext(ctx);
   }

   public void testDisplay() {← 2
      setRequestPathInfo("/secure/showTodos");
      addRequestParameter("listId"
        ,"ff808081085e18fc01085e19a3820002");
      actionPerform();
      verifyForward("show");
   }
}
```

L'initialisation du contexte de sécurité comporte trois volets (repère ❶) :

- Définition de l'utilisateur (au sens d'Acegi Security) sous lequel le code va être exécuté. Dans notre cas, il s'agit de l'utilisateur « admin », ayant pour mot de passe « admin » et rôle « ROLE_ADMIN ». Cette définition est stockée dans l'objet `user`.
- Association de l'utilisateur à un jeton d'authentification, ici `token`. Ce jeton est de type `TestingAuthenticationToken` afin de signifier à Acegi Security que l'authentification se fait dans le cadre d'un test.
- Association du jeton au contexte de sécurité `ctx` *via* la méthode `setAuthentication` puis définition de `ctx` comme étant le contexte de sécurité de l'application *via* la méthode `setContext` de la classe `SecurityContextHolder`.

Après ces initialisations nécessaires pour faire fonctionner l'application, les tests peuvent être effectués. Ces tests ne nécessitent plus que l'utilisation des fonctionnalités offertes par StrutsTestCase.

Les tests avec StrutsTestCase s'effectuent en quatre temps (repère ❷) :

1. Définition de l'action Struts à tester. Cette définition s'effectue en donnant à la méthode `setRequestPathInfo` la référence de l'action dans le fichier **struts-config.xml,** et non le nom de la classe action à tester, soit ici la classe `ShowTodosAction`. Dans notre exemple il s'agit de l'action `/secure/showTodos`.
2. Définition des paramètres attendus par l'action. Cette opération s'effectue *via* la méthode `addRequestParameter`. Dans notre exemple, nous définissons la valeur du paramètre `listId`, qui est l'identifiant de la liste de todos utilisée pour le test.
3. Exécution de l'action *via* la méthode `actionPerform`. StrutsTestCase simule alors le conteneur et exécute en son sein l'action spécifiée précédemment.
4. Vérification du résultat de l'exécution en analysant la redirection renvoyée par l'action grâce à la méthode `verifyForward`. Dans notre exemple, il s'agit de vérifier que l'action redirige vers la référence `show` définie dans le fichier **struts-config.xml.**

Outre la méthode `verifyForward`, il est possible de vérifier l'absence d'erreur (du type `ActionError`) dans l'exécution de l'action Struts grâce à la méthode `verifyNoActionErrors`.

StrutsTestCase offre en outre la possibilité de récupérer la session grâce à la méthode `getSession`. Nous pouvons ainsi accéder aux variables stockées en session pour les tester, comme le montre le code ci-dessous :

```
assertNull((String) getSession().getAttribute("todoListDocument"));
```

En résumé

Les tests d'intégration utilisant JUnit sont facilités pour les applications qui utilisent Spring. Le framework fournit des spécialisations de la classe `TestCase` de JUnit, qui permettent d'utiliser le conteneur léger en dehors de tout serveur d'applications.

Spring est en mesure de fonctionner avec le framework StrutsTestCase utilisé pour tester les développements fondés sur Struts. Le simulacre de conteneur de servlets utilisé par ce framework est capable d'instancier le conteneur léger de Spring, moyennant une légère modification du fichier de configuration de Struts.

Conclusion

Ce chapitre a montré comment tester une application Spring avec JUnit. Du point de vue des tests unitaires, le code fondé sur Spring ne nécessite pas d'extension particulière à JUnit, hormis l'utilisation le cas échéant d'un framework spécifique, comme EasyMock, pour simuler les objets dont dépend le composant à tester.

Spring fournit par ailleurs des simulacres prêts à l'emploi pour certains composants techniques de l'API J2EE. Pour les tests unitaires, seul le code applicatif est utilisé, sans l'aide du conteneur léger de Spring. L'injection des dépendances est effectuée manuellement dans les tests.

Pour les tests d'intégration, JUnit nécessite plusieurs extensions, car, dans ce cas, le conteneur léger doit être utilisé. À cette fin, Spring fournit des classes dérivant de `TestCase` de JUnit pour instancier le conteneur léger spécifiquement pour les tests. Spring peut aussi fonctionner avec StrutsTestCase pour tester l'intégration des composants utilisant les deux frameworks.

En conclusion, nous pouvons constater que l'architecture applicative qui découle de l'utilisation de Spring s'avère facilement testable, ce qui est un avantage non négligeable par rapport à d'autres choix architecturaux. Le code applicatif n'est pas dépendant de Spring, ce qui permet de le tester en dehors du conteneur léger. De plus, la séparation des interfaces et des implémentations facilite la création de simulacres, ce mode de fonctionnement étant supporté nativement par les frameworks de type EasyMock.

Spring fournit un certain nombre de simulacres très utiles pour les composants dépendants de l'API servlet ainsi que des cas de tests JUnit permettant de masquer complètement l'instanciation du framework.

Annexe

Cette annexe fournit les procédures de téléchargement et d'installation des outils nécessaires à la réalisation de l'application exemple Tudu Lists.

Ces outils étant tous issus de la communauté Open Source, ils peuvent être utilisés gratuitement dans vos projets.

Installation d'Eclipse WTP

Eclipse WTP (Web Tools Platform) peut être téléchargé gratuitement depuis la page Downloads du site Web *http://www.eclipse.org/webtools.* Cette page propose deux types d'archives compressées : All in One et Runtime. Nous conseillons d'utiliser All in One, qui contient tous les plug-in Eclipse nécessaires à WTP.

L'archive All in One peut être décompressée dans n'importe quel répertoire. La décompression crée une arborescence, dont le répertoire racine est nommé **eclipse.**

Une fois la décompression terminée, il suffit de lancer l'exécutable (**eclipse.exe** pour Windows) présent à la racine du répertoire **eclipse** pour démarrer l'environnement de développement.

Téléchargement de Tomcat

Le programme d'installation du moteur de servlets/JSP Tomcat est disponible à la sous-rubrique Binaries de la rubrique Download du site Web de Tomcat *(http://jakarta.apache.org/tomcat).*

Tomcat nécessite l'installation préalable d'un JDK, dont la version est spécifiée dans la documentation fournie sur le site Web. Pour Tomcat 5, il s'agit du JDK 1.5.

L'application Tudu Lists utilise les paramètres par défaut proposés par l'assistant d'installation :

1. Après avoir installé Tomcat, lancez-le, et vérifiez que la page affichée par l'URL *http://localhost:8080* correspond à la page d'accueil de Tomcat.

Au besoin, remplacez **8080** par le port que vous avez spécifié au moment de l'installation.

Il ne faut pas oublier d'arrêter Tomcat après ce test pour pouvoir utiliser WTP, qui lance sa propre instance de Tomcat.

Installation de MC4J

Le programme d'installation de MC4J est disponible à l'URL *http://sourceforge.net/project/showfiles.php?group_id=60228.* Il est impératif d'avoir installé au préalable un JRE version 1.4 ou 1.5 sur la machine où MC4J est installé :

1. Une fois l'installation lancée, un écran d'accueil affiche un bref descriptif du produit. Cliquez sur Next.
2. L'écran suivant permet de sélectionner le JRE à utiliser par MC4J. Une fois la sélection effectuée, cliquez sur Next.
3. Spécifiez le répertoire d'installation de MC4J, puis cliquez sur Next.
4. Indiquez au programme comment générer les icônes de lancement de MC4J. Cliquez sur Next pour l'installation des fichiers.
5. Une fois l'installation terminée, cliquez sur Done.

Références

M. FOWLER, *Inversion of Control Containers and the Dependency Injection Pattern*, *http://www.martinfowler.com*

M. FOWLER, *Patterns of Enterprise Application Architecture*, Pearson Education, 2003

E. GAMMA, R. HELM, R. JOHNSON, J. VLISSIDES, *Design Patterns : catalogue de modèles de conception réutilisables*, Vuibert, 1999

R. HARROP, J. MACHACEK, *Pro Spring*, APress, 2005

R. JOHNSON *et al.*, *Spring Reference Manual*, *http://www.springframework.org*

R. JOHNSON, J. HOELLER, A. ARENDSEN, T. RISBERG, C. SAMPALEANU, *Java Development with the Spring Framework*, Wiley, 2005

R. C. MARTIN, *The Dependency Inversion Principle*, C++ Report

R. PAWLAK, J.-Ph. RETAILLÉ, L. SEINTURIER, *Programmation orientée aspect pour Java/J2EE*, Eyrolles, 2004

M. RAIBLE, *Spring Live*, Sourcebeat, 2004

J.-Ph. RETAILLÉ, *Refactoring des applications Java/J2EE*, Eyrolles, 2005

Index

T