

Référence

Spring par l'exemple

Gary Mak

Réseaux
et télécom

Programmation

Génie logiciel

Sécurité

Système
d'exploitation



Spring

par l'exemple

Gary Mak

Traduction : Hervé Soulard
Relecture technique : Éric Hébert,
Architecte Java JEE



Pearson Education France a apporté le plus grand soin à la réalisation de ce livre afin de vous fournir une information complète et fiable. Cependant, Pearson Education France n'assume de responsabilités, ni pour son utilisation, ni pour les contrefaçons de brevets ou atteintes aux droits de tierces personnes qui pourraient résulter de cette utilisation.

Les exemples ou les programmes présents dans cet ouvrage sont fournis pour illustrer les descriptions théoriques. Ils ne sont en aucun cas destinés à une utilisation commerciale ou professionnelle.

Pearson Education France ne pourra en aucun cas être tenu pour responsable des préjudices ou dommages de quelque nature que ce soit pouvant résulter de l'utilisation de ces exemples ou programmes.

Tous les noms de produits ou marques cités dans ce livre sont des marques déposées par leurs propriétaires respectifs.

Publié par Pearson Education France
47 bis, rue des Vinaigriers
75010 PARIS
Tél. : 01 72 74 90 00
www.pearson.fr

ISBN : 978-2-7440-4107-5
Copyright © 2009 Pearson Education France
Tous droits réservés

Titre original : *Spring Recipes, a problem-solution approach*

Traduit de l'américain par Hervé Soulard
Relecture technique : Éric Hébert

ISBN original : 978-1-59059-979-2
Copyright © 2008 by Gary Mak
All rights reserved

Édition originale publiée par Apress
www.apress.com

Aucune représentation ou reproduction, même partielle, autre que celles prévues à l'article L. 122-5 2^e et 3^e a) du Code de la propriété intellectuelle ne peut être faite sans l'autorisation expresse de Pearson Education France ou, le cas échéant, sans le respect des modalités prévues à l'article L. 122-10 dudit code.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education, Inc.

Table des matières

À propos de l'auteur	VII
Introduction	IX
Public du livre	X
Organisation du livre	X
Conventions typographiques	XII
Prérequis	XII
Télécharger le code	XII
Contacter l'auteur	XII
1 Inversion de contrôle et conteneurs	1
1.1 Utiliser un conteneur pour gérer des composants	2
1.2 Utiliser un localisateur de service pour simplifier la recherche	8
1.3 Appliquer l'inversion de contrôle et l'injection de dépendance.....	10
1.4 Comprendre les différents types d'injections de dépendance	12
1.5 Configurer un conteneur à partir d'un fichier de configuration	16
1.6 En résumé	19
2 Introduction au framework Spring	21
2.1 Le framework Spring.....	22
2.2 Installer le framework Spring.....	28
2.3 Configurer un projet Spring	30
2.4 Installer Spring IDE.....	32
2.5 Utiliser les outils de gestion des beans de Spring IDE.....	34
2.6 En résumé	40
3 Configuration des beans	41
3.1 Configurer des beans dans le conteneur Spring IoC	42
3.2 Instancier le conteneur Spring IoC.....	46
3.3 Lever les ambiguïtés sur le constructeur	49
3.4 Préciser des références de beans	52
3.5 Contrôler les propriétés par vérification des dépendances.....	56
3.6 Contrôler les propriétés avec l'annotation <i>@Required</i>	59
3.7 Lier automatiquement des beans par configuration XML.....	61
3.8 Lier automatiquement des beans avec <i>@Autowired</i> et <i>@Resource</i>	66
3.9 Hériter de la configuration d'un bean.....	72

3.10	Affecter des collections aux propriétés de bean	75
3.11	Préciser le type de données des éléments d'une collection	82
3.12	Définir des collections avec des beans de fabrique et le schéma <i>util</i>	84
3.13	Rechercher les composants dans le chemin d'accès aux classes.....	87
3.14	En résumé	92
4	Fonctions élaborées du conteneur Spring IoC	95
4.1	Créer des beans en invoquant un constructeur	96
4.2	Créer des beans en invoquant une méthode statique de fabrique.....	99
4.3	Créer des beans en invoquant une méthode d'instance de fabrique.....	100
4.4	Créer des beans en utilisant un bean de fabrique Spring.....	102
4.5	Déclarer des beans correspondant à des champs statiques.....	104
4.6	Déclarer des beans correspondant aux propriétés d'un objet	106
4.7	Fixer les portées de bean	108
4.8	Modifier l'initialisation et la destruction d'un bean	110
4.9	Rendre les beans conscients de l'existence du conteneur	116
4.10	Créer des postprocesseurs de beans.....	118
4.11	Externaliser les configurations de beans	122
4.12	Obtenir des messages textuels multilingues.....	124
4.13	Communiquer à l'aide des événements d'application.....	127
4.14	Enregistrer des éditeurs de propriétés dans Spring	129
4.15	Créer des éditeurs de propriétés	133
4.16	Charger des ressources externes	134
4.17	En résumé	138
5	Proxy dynamique et Spring AOP classique	139
5.1	Problèmes associés aux préoccupations transversales non modularisées	141
5.2	Modulariser les préoccupations transversales avec un proxy dynamique.....	148
5.3	Modulariser les préoccupations transversales avec des greffons Spring classiques	155
5.4	Désigner des méthodes avec des points d'action Spring classique	164
5.5	Créer automatiquement des proxies pour les beans	167
5.6	En résumé	169
6	Spring 2.x AOP et prise en charge d'AspectJ	171
6.1	Activer la prise en charge des annotations AspectJ dans Spring.....	172
6.2	Déclarer des aspects avec des annotations AspectJ.....	175
6.3	Accéder aux informations du point de jonction	181
6.4	Préciser la précédence des aspects	182
6.5	Réutiliser des définitions de points d'action	184
6.6	Écrire des expressions AspectJ de point d'action	186
6.7	Introduire des comportements dans des beans	191
6.8	Introduire des états dans des beans.....	194
6.9	Déclarer des aspects avec des configurations XML.....	197

6.10	Tisser des aspects AspectJ au chargement dans Spring	200
6.11	Configurer des aspects AspectJ dans Spring	205
6.12	Injecter des beans Spring dans des objets de domaine	207
6.13	En résumé	210
7	Prise en charge de JDBC	213
7.1	Problèmes associés à l'utilisation directe de JDBC	214
7.2	Utiliser un template JDBC pour la mise à jour	221
7.3	Utiliser un template JDBC pour interroger une base de données.....	226
7.4	Simplifier la création d'un template JDBC	232
7.5	Utiliser le template JDBC simple avec Java 1.5	234
7.6	Utiliser des paramètres nommés dans un template JDBC.....	238
7.7	Modéliser les opérations JDBC avec des objets élémentaires	240
7.8	Gérer les exceptions dans le framework Spring JDBC	244
7.9	En résumé	250
8	Gestion des transactions	251
8.1	Problèmes associés à la gestion des transactions	252
8.2	Choisir une implémentation de gestionnaire de transactions	258
8.3	Gérer les transactions par programmation avec l'API du gestionnaire de transactions	260
8.4	Gérer les transactions par programmation avec un template de transaction	262
8.5	Gérer les transactions par déclaration avec Spring AOP classique.....	265
8.6	Gérer les transactions par déclaration avec des greffons transactionnels.....	268
8.7	Gérer les transactions par déclaration avec l'annotation <i>@Transactional</i>	270
8.8	Fixer l'attribut transactionnel de propagation	271
8.9	Fixer l'attribut transactionnel d'isolation	277
8.10	Fixer l'attribut transactionnel d'annulation.....	285
8.11	Fixer les attributs transactionnels de temporisation et de lecture seule	286
8.12	Gérer les transactions avec le tissage au chargement.....	288
8.13	En résumé	291
9	Prise en charge de l'ORM	293
9.1	Problèmes associés à l'utilisation directe des frameworks ORM	294
9.2	Configurer des fabriques de ressources ORM dans Spring.....	306
9.3	Rendre des objets persistants avec les templates ORM de Spring	312
9.4	Rendre des objets persistants avec les sessions contextuelles d'Hibernate.....	319
9.5	Rendre des objets persistants avec l'injection de contexte de JPA	322
9.6	En résumé	325
10	Framework Spring MVC	327
10.1	Développer une application web simple avec Spring MVC	328
10.2	Associer des requêtes à des gestionnaires	340
10.3	Intercepter des requêtes avec des intercepteurs de gestionnaire	344

10.4	Déterminer les paramètres régionaux de l'utilisateur.....	347
10.5	Externaliser les messages dépendant de la localisation.....	350
10.6	Déterminer les vues d'après leur nom.....	351
10.7	Associer des exceptions aux vues	355
10.8	Construire des objets <i>ModelAndView</i>	357
10.9	Créer un contrôleur avec une vue paramétrée	359
10.10	Gérer des formulaires avec des contrôleurs	361
10.11	Gérer les formulaires multipages	374
10.12	Regrouper plusieurs actions dans un contrôleur.....	383
10.13	Créer des vues Excel et PDF	389
10.14	Développer des contrôleurs avec des annotations	393
10.15	En résumé	401
11	Intégration avec d'autres frameworks web	403
11.1	Accéder à Spring depuis des applications web génériques	404
11.2	Intégrer Spring à Struts 1.x.....	409
11.3	Intégrer Spring à JSF.....	416
11.4	Intégrer Spring à DWR.....	420
11.5	En résumé	424
12	Prise en charge des tests	427
12.1	Créer des tests avec JUnit et TestNG	429
12.2	Créer des tests unitaires et des tests d'intégration.....	434
12.3	Effectuer des tests unitaires sur des contrôleurs Spring MVC	443
12.4	Gérer des contextes d'application dans les tests d'intégration.....	448
12.5	Injecter des fixtures de test dans des tests d'intégration	455
12.6	Gérer des transactions dans les tests d'intégration.....	459
12.7	Accéder à une base de données dans des tests d'intégration.....	465
12.8	Utiliser les annotations communes de Spring pour les tests	469
12.9	En résumé	472
Index	473

À propos de l'auteur

Gary Mak est architecte technique et développeur d'applications sur la plate-forme Java Entreprise depuis six ans. Au cours de sa carrière, il a participé à de nombreux projets Java, dont la plupart sont des frameworks applicatifs fondamentaux et des outils logiciels. Il préfère avant tout concevoir et implémenter les parties complexes des projets logiciels.

Gary est programmeur Java certifié Sun (SCJP, *Sun-certified Java programmer*) et titulaire d'un diplôme supérieur en informatique. Il s'intéresse à la technologie orientée objet, à la technologie orientée aspect, aux design patterns et à la réutilisation dans les logiciels. Gary est spécialisé dans la construction d'applications d'entreprise à base de frameworks, dont Spring, Hibernate, JPA, Struts, JSF et Tapestry. Il utilise le framework Spring dans ses projets depuis la sortie de la version 1.0. Il exerce également en tant que formateur sur Java Enterprise, Spring, Hibernate, les services web et le développement agile. Il a écrit plusieurs didacticiels sur Spring et Hibernate, dont certains sont disponibles publiquement et dont la popularité va croissant au sein de la communauté Java. Pendant son temps libre, il joue au tennis et assiste à des compétitions.

Introduction

C'est en 2004 que j'ai utilisé Spring pour la première fois. J'en suis devenu immédiatement un fan inconditionnel et l'emploie depuis dans pratiquement tous mes projets. Je suis particulièrement attiré par la simplicité et la sagesse de Spring. Il représente le framework applicatif Java/Java EE le plus simple et le plus puissant que j'aie jamais utilisé. Sa capacité à résoudre simplement des problèmes complexes m'a énormément impressionné. Les solutions proposées par Spring ne sont probablement pas les meilleures, mais elles sont pour le moins les plus raisonnables que je connaisse.

Spring touche à la plupart des aspects du développement d'une application Java/Java EE et propose des solutions simples. Il conduit à employer les meilleures pratiques dans la conception et la mise en œuvre des applications. Les versions 2.x de Spring ont apporté de nombreuses améliorations et nouvelles fonctionnalités par rapport aux versions 1.x. Cet ouvrage se focalise sur les fonctionnalités de Spring 2.5 pour la construction d'applications Java d'entreprise¹.

Mon expérience de formateur en technologies de programmation m'a révélé que le problème majeur des étudiants était d'arriver à des projets expérimentaux opérationnels. De nombreux ouvrages traitant de la programmation proposent des exemples de code, mais n'incluent que des fragments, non des projets complets. Il est généralement possible de télécharger le code complet depuis un site web, mais le lecteur n'a pas l'opportunité de construire les projets pas à pas. Je pense pourtant que vous avez beaucoup à apprendre du processus de construction d'un projet et que vous prendrez confiance lorsque vous verrez les projets fonctionner. Cette idée sert de fondation à ce livre.

En tant que développeurs Java en activité, nous devons souvent maîtriser de nouvelles technologies ou de nouveaux frameworks. Puisque nous sommes simplement des développeurs qui utilisent une technologie, non des étudiants ayant un examen à passer, nous pouvons nous passer de tout mémoriser. Seule une manière efficace de trouver l'information quand c'est nécessaire suffit. Pour le bénéfice du lecteur expérimenté ou novice

1. N.d.T. : au moment de l'écriture de ces lignes, la version 3.0 de Spring est en développement et devrait sortir au cours de l'année 2009. Au programme, on notera l'abandon du JDK 1.4 au profit du JDK 5, la suppression de tout ce qui était déclaré *deprecated* et l'ajout de plusieurs fonctionnalités.

qui va lire ce livre de la première page à la dernière, chaque chapitre est organisé autour de sujets présentés sous la forme problème-solution. Il lui est ainsi très facile de rechercher une solution à un problème précis.

Les thèmes abordés dans cet ouvrage sont traités sur la base d'exemples de code complets et réels que vous pouvez suivre pas à pas. À la place de descriptions abstraites de concepts complexes, ce livre propose des exemples opérationnels. Pour démarrer un nouveau projet, vous pouvez copier le code et les fichiers de configuration proposés, puis les modifier en fonction de vos besoins. En ne partant pas de zéro, vous économiserez une part importante du travail.

Public du livre

Ce livre est destiné aux développeurs Java qui souhaitent acquérir rapidement une expérience pratique du développement d'applications Java/Java EE à l'aide du framework Spring. Si vous employez déjà Spring dans vos projets, ce livre vous servira de référence dans laquelle vous trouverez des exemples de code très utiles.

Pour lire cet ouvrage, une grande expérience de Java EE n'est pas nécessaire. Cependant, il suppose que vous connaissez les bases de la programmation orientée objet en Java (par exemple créer une classe ou une interface, implémenter une interface, étendre une classe de base, exécuter une classe `Main`, configurer le chemin d'accès aux classes, etc.). Il suppose également que vous avez une certaine connaissance des concepts du Web et des bases de données, comme créer des pages web dynamiques et interroger des bases de données avec SQL.

Organisation du livre

Cet ouvrage s'articule autour de Spring 2.5 et aborde plusieurs projets Spring qui pourront vous aider dans le développement de vos applications. Il est découpé en douze chapitres. Les six premiers se focalisent sur les concepts et les mécanismes au cœur du framework Spring. Vous pourrez ainsi vous familiariser avec les bases du framework, pour ensuite aborder les autres thèmes et utiliser rapidement Spring.

- Le Chapitre 1, *Inversion de contrôle et conteneurs*, présente le concept central de Spring – la conception IoC – et l'importance des conteneurs. Si l'IoC n'a pas de secret pour vous, n'hésitez pas à sauter ce chapitre.
- Le Chapitre 2, *Introduction au framework Spring*, est un aperçu de l'architecture de Spring et des projets connexes. Il explique également comment configurer Spring dans votre environnement de développement.

- Le Chapitre 3, *Configuration des beans*, décrit les bases de la configuration des beans dans le conteneur Spring IoC. Il est indispensable de maîtriser son contenu avant d’aborder les chapitres suivants.
- Le Chapitre 4, *Fonctions élaborées du conteneur Spring IoC*, détaille les fonctionnalités élaborées et les mécanismes internes du conteneur Spring IoC. Même si ces fonctionnalités ne sont probablement pas aussi employées que celles du Chapitre 3, elles sont indispensables à tout conteneur puissant.
- Le Chapitre 5, *Proxy dynamique et Spring AOP classique*, explique pourquoi la programmation orientée aspect (POA) est nécessaire et la façon de modulariser des préoccupations transversales à l'aide de proxies dynamiques et de Spring AOP. Si vous connaissez déjà la programmation orientée aspect et souhaitez utiliser directement Spring AOP dans Spring 2.x, n'hésitez pas à sauter ce chapitre.
- Le Chapitre 6, *Spring 2.x AOP et prise en charge d'AspectJ*, se focalise sur l'utilisation de la programmation orientée aspect dans Spring 2.x et sur certains sujets avancés de la POA, notamment l'intégration du framework AspectJ dans les applications Spring.

Les six derniers chapitres concernent les sujets fondamentaux du framework Spring qui sont fréquemment employés dans le développement des applications d’entreprise.

- Le Chapitre 7, *Prise en charge de JDBC*, montre comment Spring permet de simplifier l’utilisation de JDBC. Il sert également d’introduction au module Spring pour l’accès aux données.
- Le Chapitre 8, *Gestion des transactions*, présente les différentes approches à la gestion des transactions et détaille les attributs transactionnels.
- Le Chapitre 9, *Prise en charge de l’ORM*, se focalise sur l’intégration des frameworks ORM (*Object-Relational Mapping*) répandus, notamment Hibernate et JPA, dans les applications Spring.
- Le Chapitre 10, *Framework Spring MVC*, couvre le développement d’applications web en utilisant le module Web MVC de Spring, que ce soit par l’approche traditionnelle ou par la nouvelle approche fondée sur les annotations.
- Le Chapitre 11, *Intégration avec d’autres frameworks web*, explique comment intégrer le framework Spring à d’autres frameworks d’applications web répandus, dont Struts, JSF et DWR.
- Le Chapitre 12, *Prise en charge des tests*, s’intéresse aux techniques de base pour le test des applications Java et à la prise en charge des tests dans le framework Spring.

Chaque chapitre aborde un thème Spring sous forme de multiples problèmes-solutions. Vous pouvez rechercher une solution à un problème précis et comprendre son fonctionnement dans la section consacrée. Chaque chapitre se fonde sur un exemple réel complet et cohérent. Les exemples des différents chapitres sont indépendants.

Conventions typographiques

Lorsque vous devez faire particulièrement attention à un élément du code, la partie correspondante est présentée en gras ; l'utilisation du gras ne reflète pas un changement du code par rapport à la version précédente. Lorsqu'une ligne de code est trop longue pour tenir sur la largeur de la page, elle est coupée par un caractère de prolongement du code (**→**). Pour tester le code correspondant, vous devez concaténer vous-même les parties de la ligne sans inclure d'espace.

Prérequis

Puisque le langage Java est indépendant de la plate-forme, vous pouvez choisir n'importe quel système d'exploitation pris en charge. Toutefois, sachez que ce livre se fonde sur Microsoft Windows, ce qui transparaît dans les chemins du système de fichiers. Vous devez simplement convertir ces chemins au format reconnu par votre système d'exploitation avant d'essayer le code.

Pour tirer pleinement parti de cet ouvrage, vous devez installer le JDK version 1.5 ou ultérieure. Il est également préférable de disposer d'un IDE Java qui facilitera le développement. Les projets décrits dans ce livre sont développés avec Eclipse Web Tools Platform (WTP).

Télécharger le code

Les fichiers des exemples de code sont disponibles depuis le site web Pearson (<http://www.pearson.fr>), en suivant le lien Codes sources sur la page dédiée à ce livre. Ils sont organisés selon les chapitres, avec un ou plusieurs projets Eclipse indépendants. Veuillez lire le fichier `lisezmoi.txt`, qui se trouve à la racine, pour configurer et exécuter le code source.

Contacter l'auteur

Je suis très intéressé par vos questions et vos commentaires sur le contenu de cet ouvrage. N'hésitez pas à me contacter à l'adresse springrecipes@metaarchit.com et à consulter les mises à jour sur <http://www.metaarchit.com>.

Inversion de contrôle et conteneurs

Au sommaire de ce chapitre

- ✓ Utiliser un conteneur pour gérer des composants
- ✓ Utiliser un localisateur de service pour simplifier la recherche
- ✓ Appliquer l'inversion de contrôle et l'injection de dépendance
- ✓ Comprendre les différents types d'injections de dépendance
- ✓ Configurer un conteneur à partir d'un fichier de configuration
- ✓ En résumé

Dans ce chapitre, vous allez faire connaissance avec le principe de conception *Inversion de contrôle* (IoC), que de nombreux conteneurs modernes emploient pour découpler les dépendances entre composants. Le framework Spring fournit un conteneur IoC puissant et extensible pour la gestion des composants. Il se situe au cœur du framework et s'intègre parfaitement aux autres modules de Spring. L'objectif de ce chapitre est de vous apporter les connaissances de base qui vous permettront de démarrer avec Spring.

Pour la plupart des programmeurs, dans le contexte d'une plate-forme Java EE, les composants sont des EJB (*Enterprise JavaBean*). La spécification des EJB définit clairement le contrat qui lie les composants et les conteneurs EJB. En s'exécutant dans un conteneur EJB, les composants EJB bénéficient des services de gestion du cycle de vie, de gestion des transactions et de sécurité. Toutefois, dans les versions antérieures à la version 3.0 des EJB, un seul composant EJB exige une interface Remote/Local, une interface Home et une classe d'implémentation du bean. En raison de leur complexité, ces EJB sont appelés composants lourds.

Par ailleurs, dans ces versions des EJB, un composant EJB peut s'exécuter uniquement au sein d'un conteneur EJB et doit rechercher les autres EJB à l'aide de JNDI (*Java*

Naming and Directory Interface). Les composants EJB sont dépendants de la technologie et ne peuvent pas être réutilisés ou testés en dehors d'un conteneur EJB.

De nombreux conteneurs légers sont conçus pour pallier les défauts des EJB. Ils sont légers car les objets Java simples peuvent être employés comme des composants. Toutefois, ces conteneurs posent leur propre défi : comment découpler les dépendances entre les composants. L'IoC est une solution efficace à ce problème.

Alors que l'IoC est un principe général de conception, l'injection de dépendance (DI, *Dependency Injection*) est un design pattern concret qui incarne ce principe. Puisque l'injection de dépendance constitue la mise en œuvre type de l'inversion de contrôle, si ce n'est la seule, les termes IoC et DI sont fréquemment employés de manière interchangeable.

À la fin de ce chapitre, vous serez capable d'écrire un conteneur IoC simple qui sera conceptuellement équivalent au conteneur Spring IoC. Si l'inversion de contrôle n'a pas de secret pour vous, n'hésitez pas à passer directement au Chapitre 2, qui présente l'architecture globale et la configuration du framework Spring.

1.1 Utiliser un conteneur pour gérer des composants

Problème

Derrière la conception orientée objet se cache l'idée de décomposer le système en un ensemble d'objets réutilisables. Sans un module central de gestion des objets, ces derniers doivent créer et gérer leurs propres dépendances. Par conséquent, ils sont fortement couplés.

Solution

Nous avons besoin d'un *conteneur* de gestion des objets qui composent notre système. Un conteneur centralise la création des objets et joue le rôle de registre pour les services de recherche. Un conteneur prend également en charge le cycle de vie des objets et leur fournit une plate-forme d'exécution.

Les objets qui s'exécutent à l'intérieur d'un conteneur sont appelés *composants*. Ils doivent se conformer aux spécifications définies par le conteneur.

Explications

Séparer l'interface de son implémentation

Supposons que notre objectif soit de développer un système qui génère différents types de rapports au format HTML ou PDF. Conformément au principe de "séparation de

l'interface de l'implémentation" qui prévaut dans une conception orientée objet, nous devons créer une interface commune pour la génération des rapports. Supposons que le contenu d'un rapport soit fourni par une table d'enregistrements donnée sous la forme d'un tableau de chaînes de caractères à deux dimensions.

```
package com.apress.springrecipes.report;

public interface ReportGenerator {

    public void generate(String[][] table);
}
```

Nous créons alors deux classes, `HtmlReportGenerator` et `PdfReportGenerator`, qui implémentent cette interface pour les rapports HTML et les rapports PDF. Dans le cadre de cet exemple, le squelette des méthodes suffit.

```
package com.apress.springrecipes.report;

public class HtmlReportGenerator implements ReportGenerator {

    public void generate(String[][] table) {
        System.out.println("Génération d'un rapport HTML...");
    }
}

---

package com.apress.springrecipes.report;

public class PdfReportGenerator implements ReportGenerator {

    public void generate(String[][] table) {
        System.out.println("Génération d'un rapport PDF...");
    }
}
```

Les instructions `println` présentes dans le corps des méthodes nous permettent d'être informés de l'exécution de chacune d'elles.

Les classes des générateurs de rapports étant prêtes, nous pouvons débuter la création de la classe `ReportService`, qui joue le rôle de fournisseur de service pour la génération des différents types de rapports. Ses méthodes, comme `generateAnnualReport()`, `generateMonthlyReport()` et `generateDailyReport()`, permettent de générer des rapports fondés sur les données des différentes périodes.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator = new PdfReportGenerator();

    public void generateAnnualReport(int year) {
        String[][] statistics = null;
```

```

    //
    // Collecter les statistiques pour l'année...
    //
    reportGenerator.generate(statistics);
}

public void generateMonthlyReport(int year, int month) {
    String[][] statistics = null;
    //
    // Collecter les statistiques pour le mois...
    //
    reportGenerator.generate(statistics);
}

public void generateDailyReport(int year, int month, int day) {
    String[][] statistics = null;
    //
    // Collecter les statistiques pour la journée...
    //
    reportGenerator.generate(statistics);
}
}

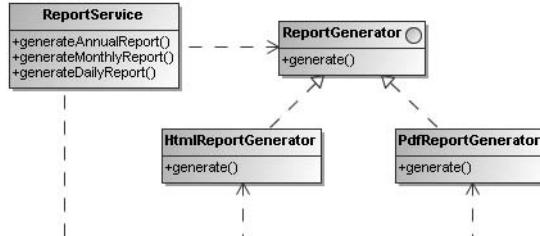
```

Puisque la logique de génération d'un rapport est déjà implémentée dans les classes des générateurs, nous pouvons créer une instance de l'une de ces classes dans une variable privée et l'invoquer dès que nous devons générer un rapport. Le format de sortie du rapport dépend de la classe instanciée.

Le diagramme de classes UML de la Figure 1.1 présente les dépendances entre ReportService et les différentes implémentations de ReportGenerator.

Figure 1.1

Dépendances entre ReportService et les différentes implémentations de ReportGenerator.



Pour le moment, ReportService crée l'instance de ReportGenerator de manière interne et doit donc savoir quelle classe concrète de ReportGenerator utiliser. Cela crée une dépendance directe entre ReportService et l'une des implémentations de ReportGenerator. Par la suite, nous supprimerons totalement les lignes de dépendance avec les implémentations de ReportGenerator.

Employer un conteneur

Supposons que notre système de génération de rapports soit destiné à plusieurs entreprises. Certains des utilisateurs préféreront les rapports au format HTML, tandis que d'autres opteront pour le format PDF. Nous devons gérer deux versions différentes de ReportService pour les deux formats de rapport. L'une crée une instance de HtmlReportGenerator, l'autre, une instance de PdfReportGenerator.

Si cette conception n'est pas vraiment souple, c'est parce que nous avons créé l'instance de ReportGenerator directement à l'intérieur de ReportService et, par conséquent, que cette classe doit savoir quelle implémentation de ReportGenerator utiliser. Rappelez-vous les lignes de dépendance partant de ReportService vers HtmlReportGenerator et PdfReportGenerator dans le diagramme de classe de la Figure 1.1. Tout changement d'implémentation du générateur de rapports conduit à une modification de ReportService.

Pour résoudre ce problème, nous avons besoin d'un conteneur qui gère les composants du système. Un conteneur complet est extrêmement complexe, mais nous pouvons commencer par en créer une version très simple :

```
package com.apress.springrecipes.report;
...
public class Container {

    // L'instance globale de cette classe Container afin que les composants
    // puissent la trouver.
    public static Container instance;

    // Un Map pour stocker les composants. L'identifiant du composant
    // sert de clé.
    private Map<String, Object> components;

    public Container() {
        components = new HashMap<String, Object>();
        instance = this;

        ReportGenerator reportGenerator = new PdfReportGenerator();
        components.put("reportGenerator", reportGenerator);

        ReportService reportService = new ReportService();
        components.put("reportService", reportService);
    }

    public Object getComponent(String id) {
        return components.get(id);
    }
}
```

Dans cet exemple de conteneur, un Map enregistre les composants, dont les identifiants servent de clés. Le constructeur du conteneur initialise les composants et les place dans

le Map. Pour le moment, il n'existe que deux composants dans notre système : ReportGenerator et ReportService. La méthode `getComponent()` retrouve un composant à partir de l'identifiant indiqué. La variable statique publique `instance` contient l'instance globale de cette classe Container. Les composants peuvent ainsi retrouver ce conteneur et rechercher d'autres composants.

Grâce au conteneur qui gère nos composants, nous pouvons remplacer la création de l'instance de ReportGenerator dans ReportService par une instruction de recherche d'un composant.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator =
        (ReportGenerator) Container.instance.getComponent("reportGenerator");

    public void generateAnnualReport(int year) {
        ...
    }

    public void generateMonthlyReport(int year, int month) {
        ...
    }

    public void generateDailyReport(int year, int month, int day) {
        ...
    }
}
```

Ainsi, ReportService ne choisit plus l'implémentation de ReportGenerator qu'elle doit employer. Il n'est donc plus nécessaire de modifier ReportService lorsque nous voulons changer d'implémentation du générateur de rapports.

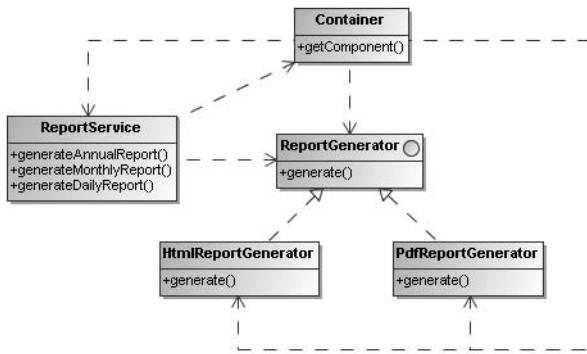
En recherchant un générateur de rapports au travers du conteneur, nous améliorons la réutilisabilité de ReportService car cette classe n'a plus de dépendance directe avec une implémentation de ReportGenerator. Nous pouvons configurer et déployer différents conteneurs pour les diverses entreprises sans modifier ReportService.

La Figure 1.2 présente le diagramme de classes UML lorsque les composants sont gérés par un conteneur.

La classe centrale Container présente des dépendances avec tous les composants sous sa responsabilité. Les dépendances entre ReportService et les deux implémentations de ReportGenerator ont été supprimées. Elles sont remplacées par une ligne de dépendance entre ReportService et Container car ReportService obtient un générateur de rapports à partir de Container.

Figure 1.2

Employer un conteneur pour gérer les composants.



Nous pouvons à présent écrire une classe `Main` pour tester notre conteneur et des composants.

```

package com.apress.springrecipes.report;

public class Main {

    public static void main(String[] args) {
        Container container = new Container();
        ReportService reportService =
            (ReportService) container.getComponent("reportService");
        reportService.generateAnnualReport(2007);
    }
}
  
```

Dans la méthode `main()`, nous commençons par créer une instance du conteneur, à partir de laquelle nous obtenons ensuite le composant `ReportService`. Puis, lorsque nous invoquons la méthode `generateAnnualReport()` sur `ReportService`, la requête de génération du rapport est prise en charge par `PdfReportGenerator`, comme cela a été fixé par le conteneur.

Pour résumer, l'emploi d'un conteneur permet de diminuer le couplage entre les différents composants d'un système et, par conséquent, d'améliorer leur indépendance et leur réutilisabilité. Cette approche permet de séparer la configuration (par exemple, le type de générateur de rapports à utiliser) de la logique de programmation (par exemple, comment générer un rapport au format PDF) et ainsi de promouvoir la réutilisabilité du système global. Pour améliorer encore notre conteneur, nous pouvons lire un fichier de configuration qui définit les composants (voir la section 1.5).

1.2 Utiliser un localisateur de service pour simplifier la recherche

Problème

Lorsque la gestion des composants est assurée par un conteneur, leurs dépendances se placent au niveau de leur interface, non de leur implémentation. Cependant, ils ne peuvent consulter le conteneur qu'en utilisant un code propriétaire complexe.

Solution

Pour simplifier la recherche de nos composants, nous pouvons appliquer le design pattern *Service Locator* (localisateur de service) proposé par Sun dans Java EE. L'idée sous-jacente à ce pattern est simple : utiliser un localisateur de service pour encapsuler la logique complexe de recherche, tout en présentant des méthodes simples pour la consultation. N'importe quel composant peut ensuite déléguer les requêtes de recherche à ce localisateur de services.

Explications

Supposons que nous devions réutiliser les composants `ReportGenerator` et `ReportService` dans d'autres conteneurs, avec des mécanismes de recherche différents, comme JNDI. `ReportGenerator` ne présente aucune difficulté. En revanche, ce n'est pas le cas pour `ReportService` car nous avons incorporé la logique de recherche dans le composant lui-même. Nous devons la modifier avant de pouvoir réutiliser cette classe.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator =
        (ReportGenerator) Container.instance.getComponent("reportGenerator");
    ...
}
```

Un localisateur de service peut être mis en œuvre par une simple classe qui encapsule la logique de recherche et présente des méthodes simples pour la consultation.

```
package com.apress.springrecipes.report;

public class ServiceLocator {

    private static Container container = Container.instance;

    public static ReportGenerator getReportGenerator() {
        return (ReportGenerator) container.getComponent("reportGenerator");
    }
}
```

Ensuite, dans ReportService, il suffit d'invoquer ServiceLocator pour obtenir un générateur de rapports, au lieu d'effectuer directement la recherche.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator =
        ServiceLocator.getReportGenerator();

    public void generateAnnualReport(int year) {
        ...
    }

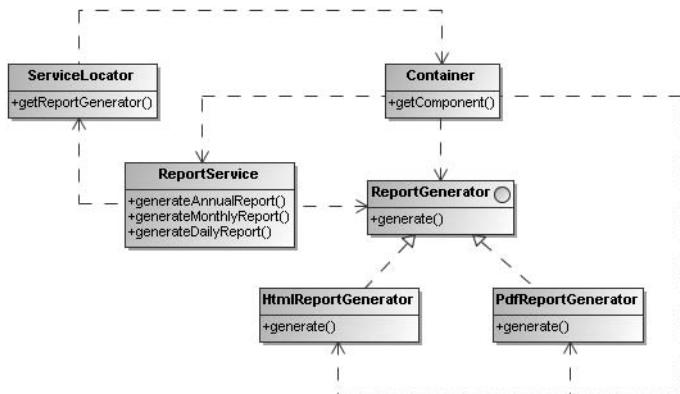
    public void generateMonthlyReport(int year, int month) {
        ...
    }

    public void generateDailyReport(int year, int month, int day) {
        ...
    }
}
```

La Figure 1.3 présente le diagramme de classes UML après application du pattern Service Locator. La ligne de dépendance qui allait initialement de ReportService à Container passe à présent par ServiceLocator.

Figure 1.3

Appliquer le pattern Service Locator de manière à réduire la complexité de la recherche.



En appliquant le pattern Service Locator, nous séparons la logique de recherche de nos composants et simplifions ainsi cette recherche. Ce pattern améliore également les possibilités de réutilisation des composants dans des environnements fondés sur des mécanismes de recherche différents. N'oubliez pas que ce design pattern est utilisé non pas seulement dans la recherche de composants, mais également dans celle de ressources.

1.3 Appliquer l'inversion de contrôle et l'injection de dépendance

Problème

Lorsqu'un composant a besoin d'une ressource externe, comme une source de données ou une référence à un autre composant, l'approche la plus directe et la plus judicieuse consiste à effectuer une recherche. Nous disons que cette opération est une recherche *active*. Elle a pour inconvénient d'obliger le composant à connaître le fonctionnement de la recherche des ressources, même si la logique est encapsulée dans un localisateur de service.

Solution

Pour la recherche de ressources, une meilleure solution consiste à appliquer l'inversion de contrôle (IoC). L'idée de ce principe est d'inverser le sens de la recherche des ressources. Dans une recherche traditionnelle, les composants dénichent les ressources en consultant un conteneur qui renvoie dûment les ressources en question. Avec l'IoC, le conteneur délivre lui-même des ressources aux composants qu'il gère. Ces derniers doivent simplement choisir une manière d'accepter les ressources. Nous qualifions cette approche de recherche *passive*.

L'IoC est un principe général, tandis que l'injection de dépendance (DI) est un design pattern concret qui incarne ce principe. Dans le pattern DI, la responsabilité d'injection des ressources appropriées dans chaque composant, en respectant un mécanisme prédéfini, par exemple au travers d'un mutateur (*setter*), est dévolue au conteneur.

Explications

Pour appliquer le pattern DI, notre `ReportService` expose un mutateur qui accepte une propriété du type `ReportGenerator`.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator; // Recherche active inutile.

    public void setReportGenerator(ReportGenerator reportGenerator) {
        this.reportGenerator = reportGenerator;
    }

    public void generateAnnualReport(int year) {
        ...
    }

    public void generateMonthlyReport(int year, int month) {
        ...
    }
}
```

```

public void generateDailyReport(int year, int month, int day) {
    ...
}
}

```

Le conteneur se charge d'injecter les ressources nécessaires dans chaque composant. Puisque la recherche active n'existe plus, nous pouvons retirer la variable statique instance dans Container et supprimer la classe ServiceLocator.

```

package com.apress.springrecipes.report;
...
public class Container {

    // Il est inutile de s'exposer pour être localisé par les composants.
    // public static Container instance;

    private Map<String, Object> components;

    public Container() {
        components = new HashMap<String, Object>();
        // Il est inutile d'exposer l'instance du conteneur.
        // instance = this;

        ReportGenerator reportGenerator = new PdfReportGenerator();
        components.put("reportGenerator", reportGenerator);

        ReportService reportService = new ReportService();
        reportService.setReportGenerator(reportGenerator);
        components.put("reportService", reportService);
    }

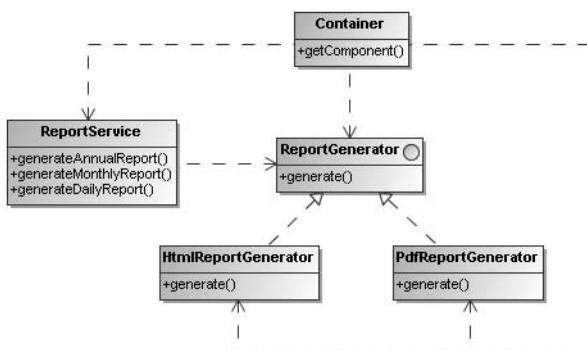
    public Object getComponent(String id) {
        return components.get(id);
    }
}

```

La Figure 1.4 présente le diagramme de classes UML après application de l'inversion de contrôle. La ligne de dépendance qui va de ReportService à Container (voir Figure 1.2) peut être retirée même sans l'aide de ServiceLocator.

Figure 1.4

Appliquer le principe IoC à l'obtention de ressources.



Le principe IoC est comparable à la devise hollywoodienne "Ne nous appelez pas, c'est nous qui vous appellerez". Il est même parfois appelé "principe de Hollywood". Par ailleurs, puisque l'injection de dépendance est l'implémentation classique de l'inversion de contrôle, ces deux termes sont souvent employés de manière interchangeable.

1.4 Comprendre les différents types d'injections de dépendance

Problème

L'utilisation d'un mutateur n'est pas la seule manière de mettre en œuvre l'injection de dépendance. En fonction des situations, nous aurons besoin de différents types de DI.

Solution

Il existe trois principaux types de DI :

- injection par interface (IoC de type 1) ;
- injection par mutateur (IoC de type 2) ;
- injection par constructeur (IoC de type 3).

L'injection par mutateur et l'injection par constructeur sont les types largement acceptés et reconnus par la plupart des conteneurs IoC.

Explications

Pour faciliter la comparaison, il est préférable de présenter les types d'injections de dépendance par ordre de popularité et d'efficacité, non par numéro de type.

Injection par mutateur (IoC de type 2)

L'injection par mutateur est le type de DI le plus répandu ; il est pris en charge par la majorité des conteneurs IoC. Le conteneur injecte une dépendance par l'intermédiaire d'un mutateur déclaré dans un composant. Par exemple, voici comment ReportService peut implémenter ce type d'injection :

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator;

    public void setReportGenerator(ReportGenerator reportGenerator) {
        this.reportGenerator = reportGenerator;
    }
    ...
}
```

Le conteneur injecte des dépendances en invoquant les mutateurs après avoir instancié chaque composant.

```
package com.apress.springrecipes.report;
...
public class Container {

    public Container() {
        ...
        ReportService reportService = new ReportService();
        reportService.setReportGenerator(reportGenerator);
        components.put("reportService", reportService);
    }
    ...
}
```

La popularité de l'injection par mutateur est due à sa simplicité et à sa facilité d'emploi. En effet, la plupart des IDE Java prennent en charge la génération automatique des mutateurs. Cependant, ce type d'injection présente quelques inconvénients mineurs. Tout d'abord, en tant que concepteurs de composants, nous ne pouvons pas être certains qu'une dépendance sera injectée *via* le mutateur. Si un utilisateur du composant oublie d'injecter une dépendance requise, une exception `NullPointerException` sera lancée et risque d'être difficile à déboguer. Toutefois, certains conteneurs IoC élaborés, comme celui de Spring, peuvent nous aider à vérifier certaines dépendances au cours de linitialisation d'un composant.

Le second inconvénient de l'injection par mutateur a trait à la sécurité du code. Après la première injection, une dépendance peut toujours être modifiée en invoquant de nouveau le mutateur, sauf si nous avons mis en œuvre nos propres mesures de sécurité pour l'empêcher. La modification imprudente des dépendances peut mener à des résultats inattendus très difficiles à déboguer.

Injection par constructeur (IoC de type 3)

L'injection par constructeur diffère de l'injection par mutateur en cela que les dépendances sont injectées depuis un constructeur non depuis un mutateur. Ce type d'injection est également pris en charge par la plupart des conteneurs IoC. Par exemple, `ReportService` pourrait accepter un générateur de rapports en argument du constructeur. Mais, si nous procédons ainsi, le compilateur Java n'ajoutera pas de constructeur par défaut pour cette classe car un constructeur aura été défini explicitement. La pratique courante consiste à définir explicitement un constructeur par défaut afin de conserver la compatibilité du code.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator;
```

```
public ReportService() {}

public ReportService(ReportGenerator reportGenerator) {
    this.reportGenerator = reportGenerator;
}
...
}
```

Le conteneur passe les dépendances en argument du constructeur au cours de l'instanciation des composants.

```
package com.apress.springrecipes.report;
...
public class Container {

    public Container() {
        ...
        ReportService reportService = new ReportService(reportGenerator);
        components.put("reportService", reportService);
    }
    ...
}
```

L'injection par constructeur permet d'éviter certains problèmes de l'injection par mutateur. Puisqu'il lui faut fournir toutes les dépendances déclarées dans la liste des arguments du constructeur, l'utilisateur ne peut pas en oublier. Par ailleurs, après qu'une dépendance a été injectée, il est impossible de la changer. Le problème de modification imprudente disparaît.

Toutefois, l'injection par constructeur présente sa propre restriction. Contrairement à l'injection par mutateur, il n'existe aucune méthode au nom significatif, comme `setSomething()`, pour indiquer l'indépendance injectée. Lors de l'invocation d'un constructeur, nous pouvons uniquement spécifier les arguments par leur emplacement. Si nous voulons en savoir plus sur les différentes versions surchargées des constructeurs et leurs arguments obligatoires, nous devons consulter la documentation JavaDoc. Par ailleurs, si le nombre de dépendances à injecter dans un composant est important, la liste des arguments du constructeur sera très longue, ce qui nuira à la lisibilité du code.

Injection par interface (IoC de type 1)

Des trois types d'injections, l'injection par interface est la moins employée. Pour l'appliquer, les composants doivent implémenter une interface spécifique définie par le conteneur, que celui-ci utilise pour injecter les dépendances. Il n'existe aucune contrainte ou caractéristique particulière pour cette interface. Il s'agit simplement d'une interface définie par le conteneur pour des questions de communication. Elle peut varier en fonction du conteneur, mais doit être implementée par les composants qu'il gère.

Dans le cas de notre conteneur simple, nous pouvons définir notre propre interface, telle que montrée dans l'exemple de code suivant. Elle déclare une seule méthode :

`inject()`. Le conteneur invoquera cette méthode sur chaque composant qui a implémenté cette interface en passant tous les composants gérés dans un Map dont les clés sont les identifiants de composants.

```
package com.apress.springrecipes.report;
...
public interface Injectable {
    public void inject(Map<String, Object> components);
}
```

Un composant doit implémenter cette interface pour que le conteneur puisse y injecter des dépendances. Les composants nécessaires sont obtenus à partir du Map en utilisant leur identifiant. Ainsi, les composants peuvent faire référence les uns aux autres sans consulter activement le conteneur.

```
package com.apress.springrecipes.report;
...
public class ReportService implements Injectable {
    private ReportGenerator reportGenerator;

    public void inject(Map<String, Object> components) {
        reportGenerator = (ReportGenerator) components.get("reportGenerator");
    }
    ...
}
```

Pour construire les dépendances, le conteneur doit injecter tous les composants, sous forme d'un Map, dans chaque composant. Cette opération doit être effectuée une fois que tous les composants ont été initialisés.

```
package com.apress.springrecipes.report;
...
public class Container {
    public Container() {
        ...
        ReportGenerator reportGenerator = new PdfReportGenerator();
        components.put("reportGenerator", reportGenerator);

        ReportService reportService = new ReportService();
        components.put("reportService", reportService);

        reportService.inject(components);
    }
    ...
}
```

Le défaut de l'injection par interface est manifeste. Elle oblige tous les composants à implémenter une interface particulière pour que le conteneur puisse injecter les dépendances. Puisque cette interface est propre au conteneur, les composants doivent s'appuyer sur ce conteneur et ne peuvent pas être réutilisés en dehors. Ce type d'inject-

tion est également dite "intrusive", car le code spécifique au conteneur "s'introduit" dans les composants. C'est pourquoi la plupart des conteneurs IoC ne prennent pas en charge ce type d'injection.

1.5 Configurer un conteneur à partir d'un fichier de configuration

Problème

Pour qu'un conteneur gère des composants et leurs dépendances, il doit au préalable être configuré avec les informations adéquates. Si le conteneur est configuré à l'aide d'un code Java, il faudra recompiler le code source après chaque modification. Pour le moins, cette approche n'est pas très efficace !

Solution

Une meilleure solution consiste à utiliser un fichier de configuration textuel lisible par un humain. Nous pouvons choisir un fichier de propriétés ou un fichier XML. Ces fichiers n'ont pas besoin d'être recompilés, ce qui facilite le fonctionnement en cas de changements fréquents.

Explications

Nous allons à présent créer un conteneur fondé sur l'injection par mutateur, dont la configuration est la plus simple. Pour les autres types d'injections, la méthode de configuration sera très proche. Tout d'abord, assurons-nous que la classe `ReportService` dispose d'un mutateur qui accepte un générateur de rapports.

```
package com.apress.springrecipes.report;

public class ReportService {

    private ReportGenerator reportGenerator;

    public void setReportGenerator(ReportGenerator reportGenerator) {
        this.reportGenerator = reportGenerator;
    }
    ...
}
```

Pour configurer un conteneur à partir d'un fichier, nous devons commencer par choisir le format de celui-ci. Pour des raisons de simplicité, nous optons pour un fichier de propriétés, même si le format XML est plus puissant et plus expressif. Un fichier de propriétés est constitué d'une liste d'entrées, chacune étant un couple clé-valeur de chaînes de caractères.

En analysant la configuration de Container, nous déterminons qu'elle concerne seulement deux aspects. Nous pouvons les exprimer sous forme de propriétés de la manière suivante :

- **Définition d'un nouveau composant.** Nous utilisons le nom du composant comme clé et le nom de classe complet comme valeur.
- **Injection de dépendance.** Nous associons le nom du composant au nom de la propriété pour former la clé, avec un point comme séparateur. Il ne faut pas oublier de définir un mutateur pour cette propriété dans la classe du composant. La valeur correspond au nom de référence de l'autre composant à injecter.

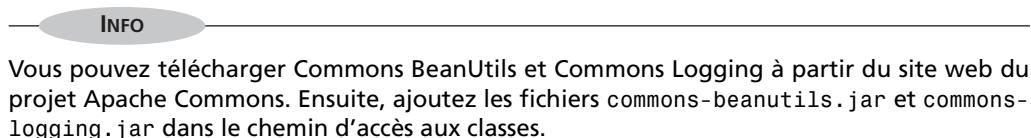
Pour remplacer la configuration par programmation par une configuration fondée sur des propriétés, nous créons le fichier `components.properties` avec le contenu suivant :

```
# Définir un nouveau composant "reportGenerator".
reportGenerator=com.apress.springrecipes.report.PdfReportGenerator

# Définir un nouveau composant "reportService".
reportService=com.apress.springrecipes.report.ReportService
# Injecter le composant "reportGenerator" dans une propriété "reportGenerator".
reportService.reportGenerator=reportGenerator
```

Notre conteneur doit lire ce fichier de configuration et interpréter son contenu comme des définitions de composants et de dépendances. Il doit également créer les instances des composants et injecter les dépendances définies dans le fichier de configuration.

Pour implémenter le conteneur, nous manipulons les propriétés de composants à l'aide du mécanisme de réflexion. Pour simplifier cette mise en œuvre, nous employons une bibliothèque tierce nommée Commons BeanUtils. Elle fait partie du projet Apache Commons (<http://commons.apache.org/>) qui fournit un ensemble d'outils pour la manipulation des propriétés d'une classe. La bibliothèque BeanUtils exige la présence d'une autre bibliothèque, appelée Commons Logging, disponible également dans ce même projet.



À présent, nous sommes prêts à implémenter Container avec cette nouvelle idée. La première étape consiste à charger le fichier de configuration dans un objet `java.util.Properties` de manière à obtenir la liste des propriétés. Ensuite, nous examinons chaque propriété constituée d'une clé et d'une valeur. Nous l'avons mentionné précédemment, il existe deux sortes de configuration.

- Si la clé ne contient pas de point, il s'agit de la définition d'un nouveau composant. Dans ce cas, nous instancions la classe indiquée grâce au mécanisme de réflexion, puis nous plaçons le composant dans le Map.
- Sinon l'entrée doit correspondre à une injection de dépendance. Nous séparons la clé en deux parties, avant et après le point. La première partie donne le nom du composant, tandis que la seconde correspond à la propriété à fixer. À l'aide de la classe `PropertyUtils` fournie par Commons BeanUtils, nous affectons à cette propriété un autre composant dont le nom est indiqué par la valeur de l'entrée.

```
package com.apress.springrecipes.report;
...
import org.apache.commons.beanutils.PropertyUtils;

public class Container {

    private Map<String, Object> components;

    public Container() {
        components = new HashMap<String, Object>();

        try {
            Properties properties = new Properties();
            properties.load(new FileInputStream("components.properties"));
            for (Map.Entry entry : properties.entrySet()) {
                String key = (String) entry.getKey();
                String value = (String) entry.getValue();
                processEntry(key, value);
            }
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }

    private void processEntry(String key, String value) throws Exception {
        String[] parts = key.split("\\.");

        if (parts.length == 1) {
            // Définition d'un nouveau composant.
            Object component = Class.forName(value).newInstance();
            components.put(parts[0], component);
        } else {
            // Injection d'une dépendance.
            Object component = components.get(parts[0]);
            Object reference = components.get(value);
            PropertyUtils.setProperty(component, parts[1], reference);
        }
    }

    public Object getComponent(String id) {
        return components.get(id);
    }
}
```

Dans la méthode `processEntry()`, nous devons découper la clé de l'entrée en fonction d'un point. Puisque dans les expressions régulières un point correspond à tout caractère, nous devons lui appliquer l'échappement à l'aide d'une barre oblique inversée. De même, en Java, l'échappement doit être appliqué à la barre oblique inversée, d'où la présence des deux barres obliques inversées.

Grâce à la lecture d'un fichier de configuration textuel pour définir les composants, notre conteneur est devenu réutilisable. À présent, aucune ligne de code Java ne doit être retouchée en cas de modification des définitions de composants. Même si ce conteneur n'est pas suffisamment élaboré pour une utilisation en production, il a atteint son objectif, c'est-à-dire illustrer le principe et le mécanisme au cœur d'un conteneur IoC.

1.6 En résumé

Dans ce chapitre, nous avons appris à utiliser un conteneur pour gérer les composants. Cela permet de réduire le couplage entre les différents composants. Un composant peut en rechercher d'autres en consultant le conteneur. Cependant, une recherche directe lie les composants au conteneur par du code complexe. Nous pouvons écrire un localisateur de service pour encapsuler la logique des recherches et lui confier le traitement de ces requêtes.

Une recherche active est une approche directe et pratique pour trouver des ressources. Cependant, elle exige que les composants connaissent le service de recherche des ressources. En utilisant le principe IoC, les composants doivent uniquement choisir une méthode pour accepter des ressources et le conteneur fournit les ressources aux composants. L'injection de dépendance est un design pattern concret conforme au principe IoC. Il existe trois principaux types de DI : injection par mutateur, injection par constructeur et injection par interface.

Nous pouvons configurer notre conteneur en utilisant du code Java ou un fichier de configuration textuel. Cette dernière solution est plus souple car il est inutile de recompiler le code source à chaque changement de la configuration.

Nous avons développé un simple conteneur IoC qui lit un fichier de propriétés. Il est conceptuellement équivalent au conteneur Spring IoC.

Le chapitre suivant donne un aperçu du framework Spring et de sa configuration.

2

Introduction au framework Spring

Au sommaire de ce chapitre

- ✓ Le framework Spring
- ✓ Installer le framework Spring
- ✓ Configurer un projet Spring
- ✓ Installer Spring IDE
- ✓ Utiliser les outils de gestion des beans de Spring IDE
- ✓ En résumé

Ce chapitre est une vue d'ensemble du framework Spring, dont l'architecture s'articule autour de multiples modules organisés de manière hiérarchique. Vous allez faire connaissance avec les principales fonctions de chaque module et aurez un aperçu des éléments les plus importants de Spring 2.0 et 2.5. Spring est non seulement un framework applicatif, mais également une plate-forme qui héberge plusieurs projets connexes regroupés sous le nom Spring Portfolio. Dans ce chapitre, vous aurez un aperçu de leurs caractéristiques.

Avant de commencer à utiliser Spring, vous devez l'installer sur votre machine de développement. L'installation du framework est très simple. Toutefois, pour tirer le meilleur parti de Spring, vous devez comprendre la structure de son répertoire d'installation et le contenu de chaque sous-répertoire.

L'équipe de développement de Spring a créé un plug-in Eclipse, appelé Spring IDE, pour faciliter le développement d'applications Spring. Vous apprendrez à installer cet IDE et à utiliser ses fonctionnalités de gestion des beans.

À la fin de ce chapitre, vous aurez acquis de solides connaissances concernant l'architecture globale et les principales fonctionnalités de Spring. Vous saurez également installer le framework Spring et le plug-in Spring IDE sur votre machine.

2.1 Le framework Spring

Spring (<http://www.springframework.org/>) est un framework complet d'applications Java/Java EE hébergé par SpringSource (<http://www.springsource.com/>), connu précédemment sous le nom Interface21. Spring prend en charge de nombreux aspects du développement d'une application Java/Java EE et peut vous aider à produire plus rapidement des applications de qualité et de performances élevées.

Le cœur du framework Spring est constitué d'un conteneur IoC léger capable d'ajouter de manière déclarative des services d'entreprise à de simples objets Java. Spring s'appuie énormément sur une excellente méthodologie de programmation, la programmation orientée aspect (POA), pour apporter ces services aux composants. Dans le contexte du conteneur Spring IoC, les composants sont également appelés beans.

Le framework Spring lui-même se fonde sur de nombreux design patterns, notamment les patterns orientés objets du GoF (*Gang of Four*, le Gang des quatre) et ceux de Core Java EE de Sun. En utilisant Spring, nous sommes conduits à employer les meilleures pratiques industrielles pour concevoir et implémenter nos applications.

Spring n'entre pas en concurrence avec les technologies existantes dans certains domaines. Bien au contraire, il s'intègre avec de nombreuses technologies majeures afin de simplifier leur utilisation. Spring représente ainsi une bonne solution parfaitement adaptée à de nombreuses situations.

Les modules de Spring

L'architecture du framework Spring repose sur des modules (voir Figure 2.1). La flexibilité d'assemblage de ces modules est telle que les applications peuvent s'appuyer sur différents sous-ensembles de manière variée.

À la Figure 2.1, les modules sont présentés sous forme hiérarchique, les modules supérieurs dépendant des modules inférieurs. Le module Core se trouve tout en bas car il constitue les fondations du framework Spring.

- **Core.** Ce module fournit les fonctionnalités de base de Spring. Il propose un conteneur IoC de base nommé BeanFactory. Les fonctionnalités de ce conteneur seront décris au Chapitre 3.
- **Context.** Ce module se fonde sur le module Core. Il en étend les fonctionnalités et propose un conteneur IoC élaboré, nommé ApplicationContext, qui ajoute ses

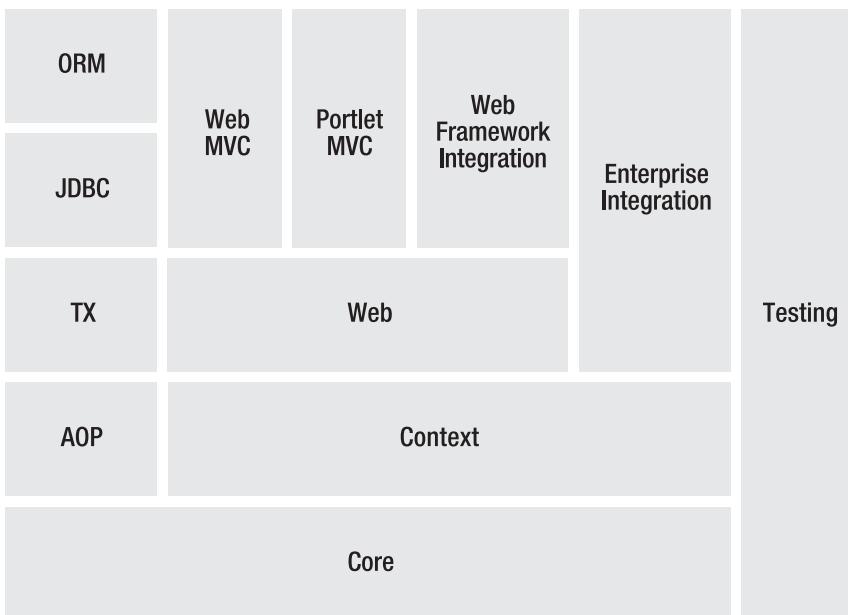


Figure 2.1

Un aperçu des modules du framework Spring.

propres fonctionnalités, comme la prise en charge de l'internationalisation (I18N), la communication à base d'événements et le chargement de ressources. Les caractéristiques de ce conteneur IoC élaboré seront examinées au Chapitre 4.

- **AOP.** Ce module définit un framework de programmation orientée aspect appelé Spring AOP. Avec l'IoC, la programmation orientée aspect est un autre concept fondamental de Spring. Les Chapitres 5 et 6 présenteront les approches POA classiques et nouvelles de Spring, ainsi que l'intégration d'AspectJ avec Spring.
- **JDBC.** Ce module définit au-dessus de l'API JDBC native une couche abstraite qui permet d'employer JDBC avec des templates et donc d'éviter la répétition du code standard (*boilerplate code*). Il convertit également les codes d'erreur des fournisseurs de bases de données en une hiérarchie d'exceptions `DataAccessException` propres à Spring. Les détails de la prise en charge de JDBC par Spring seront donnés au Chapitre 7.
- **TX.** Ce module permet de gérer les transactions par programmation ou sous forme déclarative. Ces deux approches peuvent être employées pour ajouter des possibilités transactionnelles aux objets Java simples. La gestion des transactions sera étudiée au Chapitre 8.

- **ORM.** Ce module intègre à Spring les frameworks classiques de correspondance objet-relationnel, comme Hibernate, JDO, TopLink, iBATIS et JPA. Tous les détails de l'intégration ORM de Spring se trouvent au Chapitre 9.
- **Web MVC.** Ce module définit un framework d'applications web conforme au design pattern Modèle-Vue-Contrôleur (MVC). Ce framework s'appuie sur les fonctionnalités de Spring pour qu'elles puissent servir au développement des applications web. Il sera examiné au Chapitre 10.
- **Web Framework Integration.** Ce module facilite l'utilisation de Spring en tant que support à d'autres frameworks web répandus, comme Struts, JSF, WebWork et Tapestry. Le Chapitre 11 s'intéressera à l'association de Spring avec plusieurs frameworks web.
- **Testing.** Ce module apporte la prise en charge des tests unitaires et des tests d'intégration. Il définit le framework Spring TestContext, qui rend abstraits les environnements de test sous-jacents comme JUnit 3.8, JUnit 4.4 et TestNG. Le test des applications Spring sera étudié au Chapitre 12.
- **Portlet MVC.** Ce module définit un framework de portlet, également conforme au design pattern MVC.
- **Enterprise Integration.** Ce module intègre à Spring de nombreux services d'entreprise répandus, notamment des technologies d'accès distant, les EJB, JMS, JMX, le courrier électronique et la planification, afin d'en faciliter l'usage.

Les versions de Spring

Deux ans et demi après la sortie du framework Spring 1.0 en mars 2004, la première mise à jour importante, Spring 2.0, a été publiée en octobre 2006. Voici ses principales améliorations et nouvelles fonctionnalités :

- **Configuration à base de schéma XML.** Dans Spring 1.x, les fichiers XML de configuration des beans n'acceptent que des DTD et toutes les définitions de caractéristiques doivent se faire au travers de l'élément `<bean>`. Spring 2.0 prend en charge la configuration à base de schéma XML, qui permet d'utiliser les nouvelles balises de Spring. Les fichiers de configuration des beans peuvent ainsi être plus simples et plus clairs. Dans ce livre, nous exploitons cette possibilité. Le Chapitre 3 en présentera les bases.
- **Configuration à base d'annotations.** En complément de la configuration basée sur XML, Spring 2.0 accepte dans certains modules une configuration à base d'annotations, comme `@Required`, `@Transactional`, `@PersistenceContext` et `@PersistenceUnit`. Les Chapitres 3, 8 et 9 expliqueront comment employer ces annotations.

- **Nouvelle approche de Spring AOP.** L'utilisation classique de la programmation orientée aspect dans Spring 1.x se fait au travers d'un ensemble d'API propriétaires de Spring AOP. Spring 2.0 propose une toute nouvelle approche fondée sur l'écriture de POJO avec des annotations AspectJ ou une configuration à base de schéma XML. Le Chapitre 6 en donnera tous les détails.
- **Déclaration plus aisée des transactions.** Dans Spring 2.0, il est beaucoup plus facile de déclarer des transactions. La nouvelle approche de Spring AOP permet de déclarer des greffons (*advice*s) transactionnels, mais il est également possible d'appliquer des annotations `@Transactional` avec la balise `<tx:annotation-driven>`. Le Chapitre 8 reviendra en détail sur la gestion des transactions.
- **Prise en charge de JPA.** Spring 2.0 ajoute la prise en charge de l'API de persistance Java dans son module ORM. Elle sera examinée au Chapitre 9.
- **Bibliothèque pour la balise form.** Spring 2.0 propose une nouvelle bibliothèque pour faciliter le développement de formulaires dans Spring MVC. Son utilisation sera décrite au Chapitre 10.
- **Prise en charge de JMS en mode asynchrone.** Spring 1.x n'accepte que la réception synchrone de messages JMS par l'intermédiaire de `JmsTemplate`. Spring 2.0 ajoute la réception asynchrone de messages JMS au travers de POJO orientés message.
- **Prise en charge d'un langage de script.** Spring 2.0 reconnaît l'implémentation de beans à l'aide des langages de script JRuby, Groovy et BeanShell.

Novembre 2007 a vu la sortie de Spring 2.5, qui ajoute les fonctionnalités suivantes par rapport à Spring 2.0. Au moment de l'écriture de ces lignes, Spring 2.5 est la version courante du framework.

- **Configuration à base d'annotations.** Spring 2.0 a ajouté plusieurs annotations pour simplifier la configuration des beans. Spring 2.5 en reconnaît d'autres, notamment `@Autowired` et celles de la JSR 250, `@Resource`, `@PostConstruct` et `@PreDestroy`. Les Chapitres 3 et 4 expliqueront comment les utiliser.
- **Scan de composants.** Spring 2.5 peut détecter automatiquement les composants avec des annotations particulières situés dans le chemin d'accès aux classes, sans aucune configuration manuelle. Cette possibilité sera étudiée au Chapitre 3.
- **Prise en charge du tissage AspectJ au chargement.** Spring 2.5 prend en charge le tissage des aspects AspectJ dans le conteneur Spring IoC au moment du chargement. Il est ainsi possible d'utiliser des aspects AspectJ en dehors de Spring AOP. Ce sujet sera examiné au Chapitre 6.

- **Contrôleurs web à base d'annotations.** Spring 2.5 propose une nouvelle approche basée sur les annotations pour le développement des contrôleurs web. Il peut détecter automatiquement les classes contrôleurs avec l'annotation `@Controller`, ainsi que les informations configurées à l'aide de `@RequestMapping`, `@RequestParam` et `@ModelAttribute`. Le Chapitre 10 reviendra en détail sur ce point.
- **Prise en charge améliorée des tests.** Spring 2.5 définit un nouveau framework de test appelé Spring TestContext. Il prend en charge les tests dirigés par les annotations et rend abstraits les frameworks de test sous-jacents. Ce sujet fera l'objet du Chapitre 12.

Sachez enfin que le framework Spring est conçu de manière rétrocompatible et qu'il est donc facile de passer les applications Spring 1.x à Spring 2.0, ainsi que les applications 2.0 à 2.5.

Spring 3.0 est en préparation et, au moment de l'écriture de ces lignes, la version M2 (milestone 2) vient d'être publiée. Cette nouvelle version du framework poursuit ce qui avait débuté avec Spring 2.5, c'est-à-dire l'adhésion au modèle de programmation de Java 5. La liste officielle des nouvelles fonctionnalités n'est pas établie mais devrait inclure les éléments suivants :

- Généralisation de Spring EL : il sera possible d'utiliser les Expression Language dans n'importe quel module de Spring.
- Prise en charge des Portlets 2.0.
- Validation du modèle *via* les annotations.
- Intégration de la portée conversation dans Spring Core (elle est déjà présente dans Spring WebFlow).

La rétrocompatibilité de Spring 3.0 ne devrait être assurée qu'avec Spring 2.5.

Les projets Spring

Spring n'est pas seulement un framework applicatif. Il sert également de plate-forme à plusieurs projets open-source qui se fondent sur le projet central Spring Framework. Au moment de l'écriture de ces lignes, voici les projets de Spring Portfolio :

- **Spring IDE.** Ce projet fournit des plug-in Eclipse qui facilitent l'écriture des fichiers de configuration des beans. Depuis la version 2.0, Spring IDE prend également en charge Spring AOP et Spring Web Flow. Nous verrons comment installer Spring IDE dans ce chapitre.

- **Spring Security.** Ce projet, précédemment nommé Acegi Security, définit un framework de sécurité pour les applications d'entreprise, plus particulièrement celles développées avec Spring. Il propose des options de sécurité pour l'authentification, l'autorisation et le contrôle d'accès, que nous pouvons appliquer à nos applications.
- **Spring Web Flow.** Ce projet nous permet de modéliser sous forme de flux les actions complexes des utilisateurs au sein d'une application web. Grâce à Spring Web Flow, nous pouvons développer facilement ces flux web et les réutiliser.
- **Spring Web Services.** Ce projet se concentre sur le développement de services web orientés contrat (*contract-first*) ou pilotés par les documents (*document-driven*). Il intègre de nombreuses méthodes de manipulation d'un contenu XML.
- **Spring Rich Client.** Ce projet définit un framework, construit au-dessus de Spring, pour le développement d'applications graphiques riches avec Spring.
- **Spring Batch.** Ce projet propose un framework pour le traitement par lots dans les applications d'entreprise, en se focalisant sur les grands volumes d'informations.
- **Spring Modules.** Ce projet intègre d'autres outils et projets sous forme de modules afin d'étendre le framework Spring sans développer le module Core.
- **Spring Dynamic Modules.** Ce projet prend en charge la création d'applications Spring qui s'exécutent sur la plate-forme de services OSGi (*Open Services Gateway initiative*). Elle permet d'installer, de mettre à jour et de supprimer dynamiquement des modules applicatifs.
- **Spring Integration.** Ce projet fournit une extension du framework Spring qui prend en charge l'intégration avec des systèmes externes au travers d'adaptateurs de haut niveau.
- **Spring LDAP.** Ce projet propose une bibliothèque qui simplifie les opérations LDAP et gère les exceptions LDAP par une approche fondée sur les templates.
- **Spring JavaConfig.** Ce projet apporte une alternative Java à la configuration des composants dans le conteneur Spring IoC.
- **Spring BeanDoc.** Ce projet facilite la génération d'une documentation et des diagrammes d'après le fichier de configuration des beans.
- **Spring .NET.** Comme son nom l'indique, ce projet est une version .NET du framework Spring qui facilite le développement d'applications .NET.

2.2 Installer le framework Spring

Problème

Nous souhaitons développer une application Java/Java EE en utilisant le framework Spring. Nous devons commencer par installer ce framework sur notre machine.

Solution

L'installation du framework Spring est très simple. Il suffit de télécharger sa version 2.5 au format ZIP et d'extraire le contenu de l'archive dans un répertoire de notre choix.

Explications

Installer le JDK

Avant d'installer le framework Spring, un JDK doit être présent sur notre machine. Spring 2.5 exige le JDK 1.4 ou une version ultérieure. Il est fortement conseillé d'installer le JDK 1.5 ou une version ultérieure pour utiliser certaines fonctionnalités comme les annotations, le *boxing* et l'*unboxing* automatiques, les fonctions *varargs*, les collections typées (*type-safe*) et les boucles *for-each*.

Installer un IDE Java

Bien que le développement d'applications Java puisse se faire sans IDE, il sera certainement facilité par un tel éditeur. Puisque la configuration de Spring se fait principalement avec du contenu XML, nous devons choisir un IDE qui fournit des fonctions d'édition XML, comme la validation XML et la complétion automatique pour sa syntaxe. Si vous n'avez pas d'IDE Java préféré ou si vous voulez en choisir un autre, nous vous conseillons d'installer Eclipse Web Tools Platform (WTP), disponible en téléchargement à l'adresse <http://www.eclipse.org/webtools/>. Eclipse WTP est très facile à utiliser et propose de nombreuses fonctionnalités puissantes pour Java, le Web et XML. Par ailleurs, il existe un plug-in Spring pour Eclipse qui se nomme Spring IDE.

Télécharger et installer Spring

À partir du site web de Spring, nous pouvons télécharger la version 2.5 du framework. Nous sommes alors redirigés vers SpringSource, où nous devons choisir la distribution de Spring adéquate (voir Figure 2.2).

La Figure 2.2 montre qu'il existe trois distributions de la version 2.5 de Spring. La première contient l'intégralité du framework, y compris les fichiers JAR de Spring, les bibliothèques dépendantes, la documentation, le code source et des exemples d'applications. La deuxième contient uniquement les fichiers JAR de Spring et la documentation.

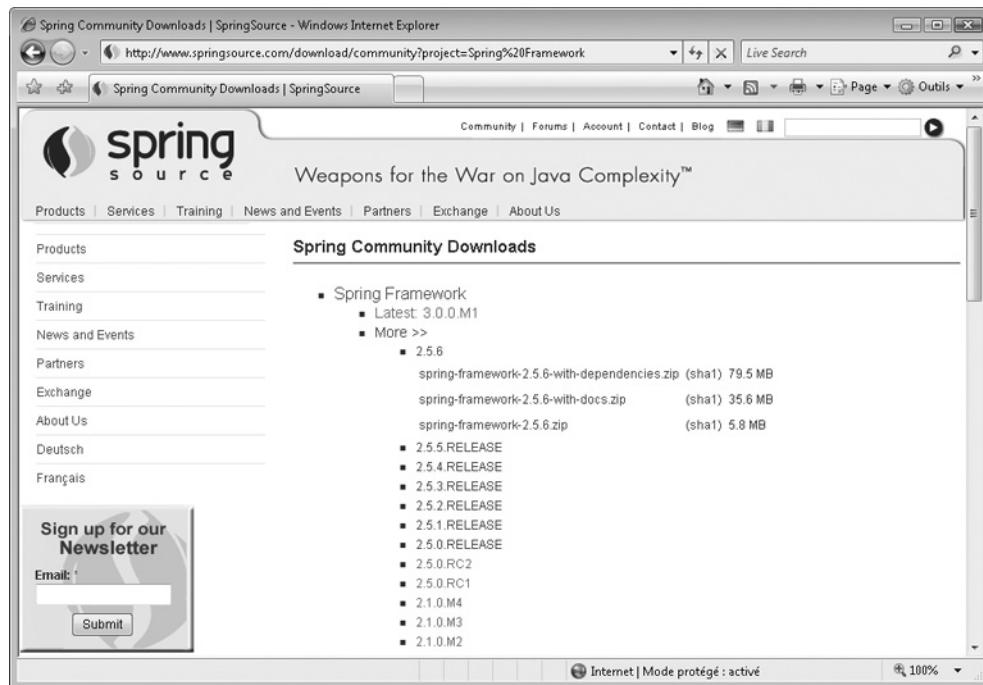


Figure 2.2

Télécharger une distribution de Spring.

La dernière contient uniquement les fichiers JAR de Spring. Il est préférable de télécharger la distribution avec toutes les dépendances, c'est-à-dire `spring-framework-2.5.6-with-dependencies.zip`. Cette archive est la plus volumineuse car elle contient la plupart des bibliothèques dont nous avons besoin pour développer des applications Spring.

Après avoir téléchargé la distribution de Spring, il suffit d'extraire le contenu de l'archive dans le répertoire de notre choix. Cet ouvrage suppose que le développement se fait sur une plate-forme Windows et que Spring est installé dans le répertoire `c:\spring-framework-2.5.6`.

Explorer le répertoire d'installation de Spring

Une fois que la distribution de Spring a été installée, le répertoire de Spring doit contenir les sous-répertoires recensés au Tableau 2.1.

Tableau 2.1 : Sous-réertoires du répertoire d'installation de Spring

Répertoire	Contenu
aspectj	Le code source et des cas de test pour les aspects Spring utilisés dans AspectJ
dist	Les fichiers JAR pour les modules Spring et la prise en charge du tissage, ainsi que les fichiers de ressources, comme les DTD et les XSD
docs	La documentation JavaDoc de l'API de Spring, la documentation de référence aux formats PDF et HTML, ainsi qu'un didacticiel pour Spring MVC
lib	Les fichiers JAR des bibliothèques utilisées par Spring, organisés par projet
mock	Le code source des objets simulacres (<i>mock object</i>) de Spring et de la prise en charge des tests
samples	Plusieurs exemples d'applications pour illustrer l'utilisation de Spring
src	Le code source de la majeure partie du framework Spring
test	Les cas de test pour tester le framework Spring
tiger	Le code source, les cas de test et les objets simulacres pour Spring spécifiques au JDK 1.5 et les versions ultérieures

2.3 Configurer un projet Spring

Problème

Nous voulons créer un projet Java en utilisant le framework Spring. Nous devons commencer par configurer ce projet.

Solution

Pour configurer un projet Spring, nous devons fixer le chemin d'accès aux classes et créer un ou plusieurs fichiers pour configurer des beans dans le conteneur Spring IoC. Ensuite, nous pouvons instancier le conteneur Spring IoC avec ce fichier de configuration des beans et lui demander des instances de beans.

Explications

Définir le chemin d'accès aux classes

Pour débuter un projet Spring, nous devons inclure les fichiers JAR des modules Spring que nous utilisons et ceux des bibliothèques de dépendance dans le chemin d'accès aux classes. Pour des raisons de commodité, nous pouvons inclure l'unique fichier `spring.jar` qui contient l'ensemble des modules standard, même s'il est relativement

volumineux (environ 2,7 Mo). Il se trouve dans le sous-répertoire `dist` du répertoire d'installation de Spring. Nous pouvons également sélectionner chaque module que nous utilisons à partir du répertoire `dist/modules`.

Un projet Spring a également besoin du fichier `commons-logging.jar`, qui se trouve dans le répertoire `lib/jakarta-commons`. Puisque le framework Spring se sert de cette bibliothèque pour produire ses messages de journalisation, son fichier JAR doit être inclus dans le chemin d'accès aux classes. La configuration de ce chemin varie en fonction des IDE.

Cet ouvrage suppose que les fichiers `spring.jar` et `commons-logging.jar` sont indiqués dans le chemin d'accès aux classes de tous les projets.

Créer le fichier de configuration des beans

Un projet Spring classique nécessite un ou plusieurs fichiers pour configurer les beans dans le conteneur Spring IoC. Nous pouvons placer ces fichiers de configuration dans le chemin d'accès aux classes ou dans un chemin du système de fichiers. Pour faciliter les tests depuis un IDE, il est préférable de le placer dans le chemin d'accès aux classes défini pour le projet. Cet ouvrage suppose que le fichier de configuration des beans est créé à la racine du chemin d'accès aux classes et qu'il se nomme `beans.xml`, sauf précision autre.

Pour illustrer la définition d'un projet Spring, créons la classe `HelloWorld` suivante. Elle accepte une propriété `message` via une injection par mutateur. Dans sa méthode `hello()`, nous construisons un message de bienvenue et l'affichons sur la console.

```
package com.apress.springrecipes.hello;

public class HelloWorld {

    private String message;

    public void setMessage(String message) {
        this.message = message;
    }

    public void hello() {
        System.out.println("Bonjour ! " + message);
    }
}
```

Pour configurer un bean de type `HelloWorld` avec un message dans le conteneur Spring IoC, nous créons le fichier suivant. Conformément aux conventions définies pour cet ouvrage, il est placé à la racine du chemin d'accès aux classes et nommé `beans.xml`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
```

```
<bean id="helloWorld" class="com.apress.springrecipes.hello.HelloWorld">
    <property name="message" value="Comment allez-vous ?" />
</bean>
</beans>
```

Utiliser les beans du conteneur Spring IoC

Nous écrivons la classe Main suivante pour instancier le conteneur Spring IoC avec le fichier de configuration des beans nommé beans.xml et situé à la racine du chemin d'accès aux classes. Nous pouvons ensuite obtenir le bean helloWorld à partir de ce conteneur et l'utiliser comme bon nous semble.

```
package com.apress.springrecipes.hello;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        HelloWorld helloWorld = (HelloWorld) context.getBean("helloWorld");
        helloWorld.hello();
    }
}
```

Si tout se passe bien, le message suivant doit apparaître sur la console, après plusieurs lignes de messages de journalisation de Spring :

```
Bonjour ! Comment allez-vous ?
```

2.4 Installer Spring IDE

Problème

Nous souhaitons installer un IDE qui nous aidera lors du développement d'applications Spring.

Solution

Si vous utilisez déjà Eclipse, vous pouvez installer le plug-in Eclipse officiel nommé Spring IDE (<http://springide.org/>). IntelliJ IDEA prend également en charge Spring, mais ses fonctionnalités ne sont pas aussi puissantes ou à jour que Spring IDE. Cet ouvrage se limite à Spring IDE. Il existe trois manières d'installer ce plug-in :

- ajouter le site de mise à jour de Spring IDE (<http://dist.springframework.org/release/IDE>) dans le gestionnaire de mise à jour d'Eclipse et procéder à l'installation en ligne ;

- télécharger le site de mise à jour archivé de Spring IDE et l'installer dans le gestionnaire de mise à jour d'Eclipse en tant que site local ;
- télécharger l'archive de Spring IDE et en extraire le contenu dans le répertoire d'installation d'Eclipse.

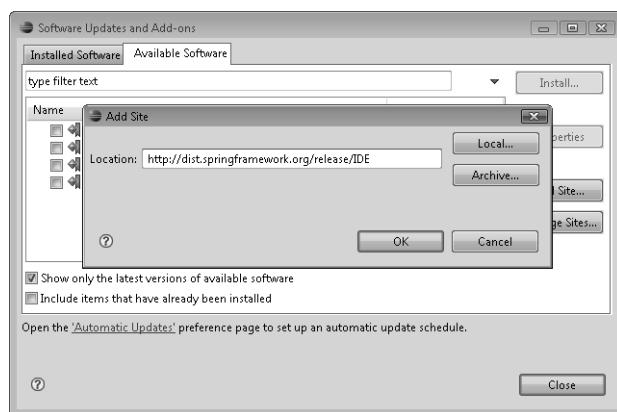
De ces trois méthodes, il est préférable de choisir la première si la machine dispose d'une connexion Internet. Dans le cas contraire, il doit être installé à partir d'un site local archivé. Ces deux solutions permettent de vérifier si tous les plug-in requis par Spring IDE sont déjà installés.

Explications

À partir du menu Help d'Eclipse, choisissez Software Updates. Dans la fenêtre Software Updates and Add-ons qui s'affiche, cliquez sur l'onglet Available Software. Vous voyez alors la liste des sites de mise à jour existants. Si celui correspondant à Spring IDE est absent, ajoutez-le en cliquant sur Add Site. Saisissez alors l'URL du site (voir Figure 2.3).

Figure 2.3

Ajouter le site de mise à jour pour Spring IDE.

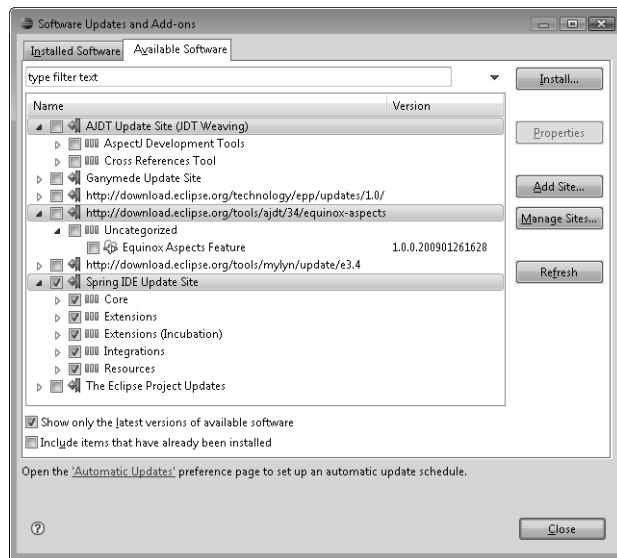


Après avoir cliqué sur OK, la liste des fonctionnalités de Spring IDE et leurs dépendances doivent s'afficher (voir Figure 2.4).

Spring IDE 2.2.1 utilise deux autres plug-in Eclipse : Eclipse Mylyn et AJDT (*AspectJ Development Tools*). Mylyn est une interface utilisateur orientée tâche qui peut intégrer de multiples tâches de développement et diminuer la surcharge d'informations. AJDT est un plug-in Eclipse pour le développement AspectJ. Spring IDE se fonde sur AJDT pour la prise en charge du développement orienté aspect de Spring 2.0. Vous pouvez également installer les fonctionnalités d'intégration de Spring IDE avec ces plug-in.

Figure 2.4

Sélectionner les fonctionnalités de Spring IDE qui seront installées.



Si vous utilisez Eclipse 3.3 ou une version ultérieure, vous ne devez pas sélectionner la catégorie Dependencies dans Spring IDE car elle est réservée à Eclipse 3.2. Pour installer Spring IDE, il suffit ensuite de suivre les instructions pas à pas.

2.5 Utiliser les outils de gestion des beans de Spring IDE

Problème

Nous voulons employer les fonctionnalités de prise en charge des beans de Spring IDE pour nous aider à développer des applications Spring.

Solution

Spring IDE propose de riches fonctions de gestion des beans qui peuvent améliorer notre productivité lors du développement d'applications Spring. En utilisant Spring IDE, nous pouvons afficher nos beans en mode explorateur ou graphique. Par ailleurs, Spring IDE peut nous assister dans l'écriture du contenu du fichier de configuration des beans et vérifier sa conformité.

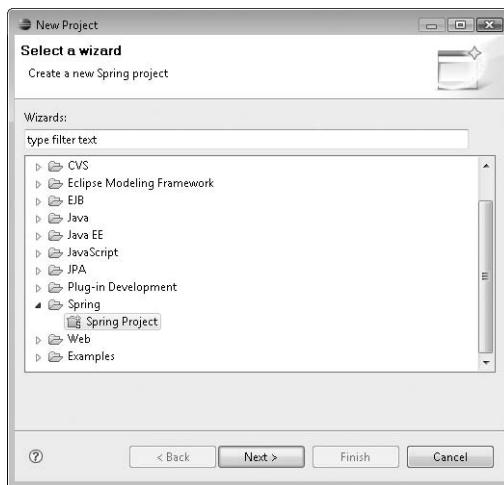
Explications

Créer un projet Spring

Pour créer un projet Spring dans Eclipse lorsque Spring IDE est installé, il suffit de choisir File > New > Project et de sélectionner Spring Project dans la rubrique Spring (voir Figure 2.5).

Figure 2.5

Créer un projet Spring.



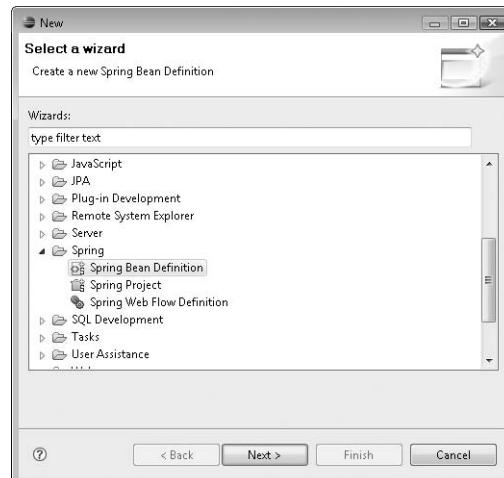
Pour convertir un projet Java existant en projet Spring, nous sélectionnons ce projet, cliquons dessus du bouton droit et choisissons Spring Tools > Add Spring Project Nature pour activer la prise en charge de Spring IDE sur ce projet. L'icône d'un projet Spring comprend un petit "S" dans le coin supérieur droit.

Créer un fichier de configuration des beans

Pour créer un fichier de configuration des beans, nous choisissons File > New > Other et sélectionnons Spring Bean Definition dans la rubrique Spring (voir Figure 2.6).

Figure 2.6

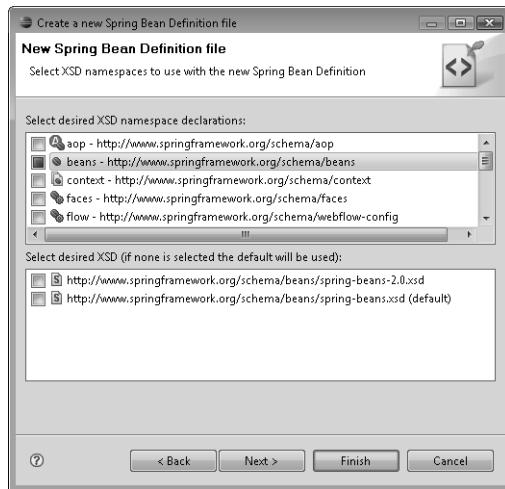
Créer un fichier de configuration des beans Spring.



Après avoir choisi l'emplacement et le nom de ce fichier de configuration des beans, nous devons indiquer les espaces de noms XSD à inclure dans ce fichier (voir Figure 2.7).

Figure 2.7

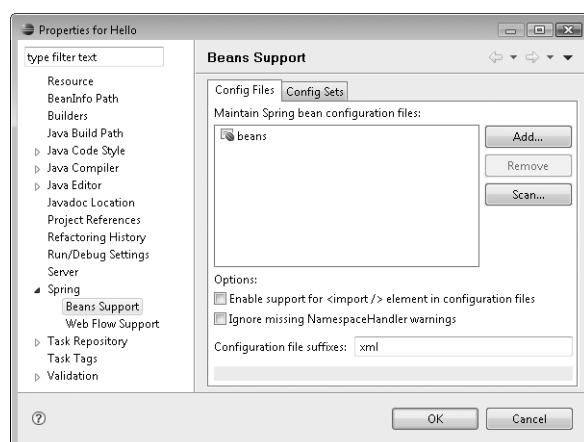
Sélectionner les espaces de noms XSD à inclure dans le fichier de configuration des beans.



Pour utiliser un fichier XML existant comme fichier de configuration des beans, il suffit de cliquer du bouton droit sur le projet et de choisir Properties. Depuis Beans Support, qui se trouve dans la rubrique Spring, nous pouvons ajouter nos fichiers XML (voir Figure 2.8).

Figure 2.8

Indiquer les fichiers de configuration des beans dans un projet Spring.



Visualiser les beans dans Spring Explorer

Pour mieux illustrer les fonctions d'exploration et les fonctions graphiques de Spring IDE, créons la classe `Holiday` suivante. Pour des questions de simplicité, les accesseurs et les mutateurs ne sont pas présentés. La commande Source > Generate Getters and Setters permet de les générer facilement.

```
package com.apress.springrecipes.hello;

public class Holiday {

    private int month;
    private int day;
    private String greeting;

    // Accesseurs et mutateurs.
    ...
}
```

Nous modifions ensuite notre classe `HelloWorld` pour lui ajouter la propriété `holidays` de type `java.util.List` et le mutateur correspondant.

```
package com.apress.springrecipes.hello;

import java.util.List;

public class HelloWorld {
    ...
    private List<Holiday> holidays;

    public void setHolidays(List<Holiday> holidays) {
        this.holidays = holidays;
    }
}
```

Dans le fichier de configuration des beans, nous déclarons les beans `christmas` et `newYear` de type `Holiday`. Ensuite, nous ajoutons des références à ces deux beans dans la propriété `holidays` du bean `helloWorld`.

```
<beans ...>
    <bean id="helloWorld" class="com.apress.springrecipes.hello.HelloWorld">
        <property name="message" value="Comment allez-vous ?" />
        <property name="holidays">
            <list>
                <ref local="christmas" />
                <ref local="newYear" />
            </list>
        </property>
    </bean>

    <bean id="christmas" class="com.apress.springrecipes.hello.Holiday">
        <property name="month" value="12" />
        <property name="day" value="25" />
        <property name="greeting" value="Joyeux Noël !" />
    </bean>
```

```

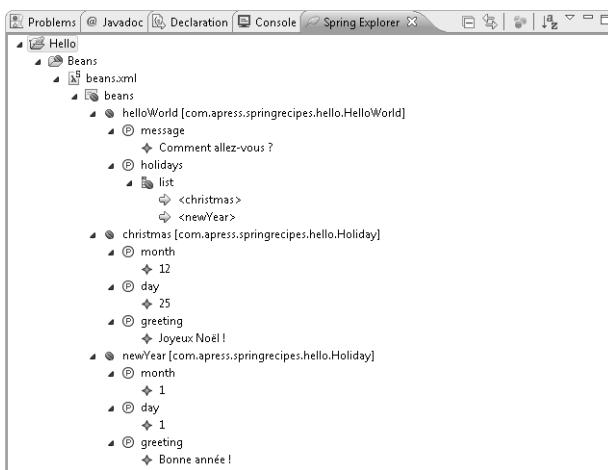
<bean id="newYear" class="com.apress.springrecipes.hello.Holiday">
    <property name="month" value="1" />
    <property name="day" value="1" />
    <property name="greeting" value="Bonne année !" />
</bean>
</beans>

```

Pour visualiser nos beans dans Spring Explorer, nous cliquons du bouton droit sur le fichier de configuration des beans et choisissons Show In > Spring Explorer (voir Figure 2.9).

Figure 2.9

Explorer les beans Spring dans Spring Explorer.



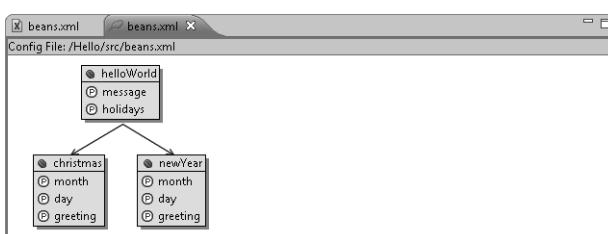
Dans Spring Explorer, nous pouvons double-cliquer sur un élément pour aller directement à sa déclaration dans le fichier de configuration des beans. Il est ainsi très commode de modifier cet élément.

Visualiser des beans Spring dans Spring Beans Graph

Nous pouvons cliquer du bouton droit sur un élément de bean (ou l'élément racine) dans Spring Explorer et choisir Open Graph pour visualiser les beans sous forme de graphe (voir Figure 2.10).

Figure 2.10

Visualiser les beans Spring dans Spring Beans Graph.

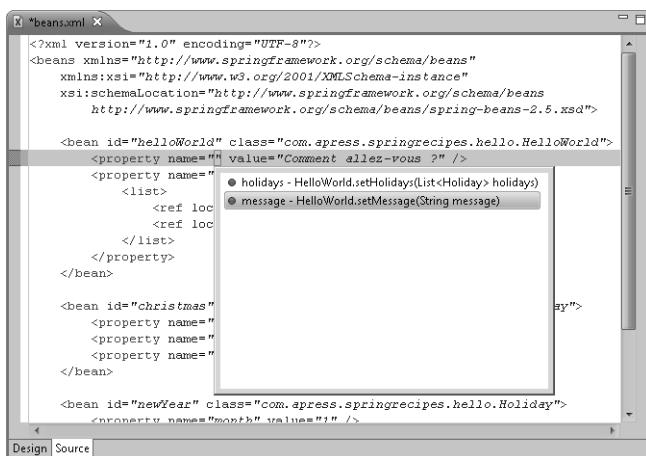


Utiliser l'assistance de contenu dans les fichiers de configuration des beans

Spring IDE prend en charge la fonctionnalité d'assistance de contenu d'Eclipse pour modifier les noms de classes, les noms de propriétés et les références aux noms de beans. Par défaut, la combinaison de touches qui active cette fonctionnalité est Ctrl+Barre d'espace. Nous pouvons par exemple l'employer pour compléter le nom d'une propriété (voir Figure 2.11).

Figure 2.11

Utiliser l'assistance de contenu dans les fichiers de configuration des beans.

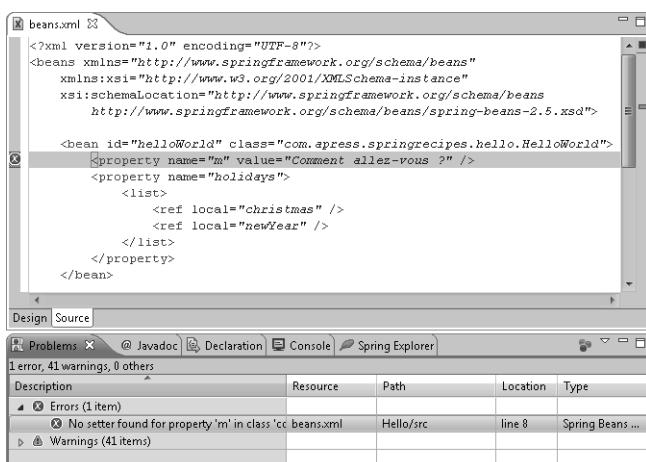


Valider les fichiers de configuration des beans

Lors de l'enregistrement du fichier de configuration des beans, Spring IDE vérifie automatiquement sa conformité par rapport aux classes des beans, aux noms des propriétés, aux références aux noms de beans, etc. Toute erreur est signalée (voir Figure 2.12).

Figure 2.12

Valider les fichiers de configuration des beans avec Spring IDE.

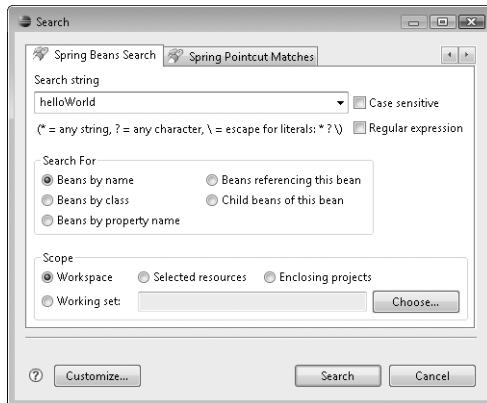


Rechercher des beans Spring

Spring IDE prend en charge la recherche de beans répondant à certains critères. Dans le menu Search, choisissez Beans pour afficher la boîte de dialogue de recherche de beans (voir Figure 2.13).

Figure 2.13

Rechercher des beans Spring.



2.6 En résumé

Dans ce chapitre, nous avons examiné l'architecture globale et les principales fonctions du framework Spring, ainsi que l'objectif général de chaque sous-projet. L'installation de Spring est très simple. Il suffit de le télécharger et d'extraire son contenu dans un répertoire. Nous avons eu un aperçu du contenu de chaque sous-répertoire du répertoire d'installation de Spring. Nous avons créé un projet Spring très simple, qui a illustré la configuration générale des projets. Nous en avons également profité pour définir les conventions qui seront employées tout au long de cet ouvrage.

L'équipe de développement de Spring a créé un plug-in Eclipse, nommé Spring IDE, pour simplifier le développement d'applications Spring. Dans ce chapitre, nous avons installé cet IDE et appris à utiliser ses fonctionnalités de base pour la gestion des beans.

Le chapitre suivant présente les bases de la configuration d'un bean dans Spring IoC.

3

Configuration des beans

Au sommaire de ce chapitre

- ✓ Configurer des beans dans le conteneur Spring IoC
- ✓ Instancier le conteneur Spring IoC
- ✓ Lever les ambiguïtés sur le constructeur
- ✓ Préciser des références de beans
- ✓ Contrôler les propriétés par vérification des dépendances
- ✓ Contrôler les propriétés avec l'annotation `@Required`
- ✓ Lier automatiquement des beans par configuration XML
- ✓ Lier automatiquement des beans avec `@Autowired` et `@Resource`
- ✓ Hériter de la configuration d'un bean
- ✓ Affecter des collections aux propriétés de bean
- ✓ Préciser le type de données des éléments d'une collection
- ✓ Définir des collections avec des beans de fabrique et le schéma `util`
- ✓ Rechercher les composants dans le chemin d'accès aux classes
- ✓ En résumé

Ce chapitre détaille les bases de la configuration d'un composant dans le conteneur Spring IoC. Placé au cœur du framework Spring, ce conteneur est conçu pour être hautement configurable et bénéficier d'une grande capacité d'adaptation. Ses fonctionnalités simplifient autant que possible la configuration des composants qui devront s'y exécuter.

Dans Spring, les composants sont également appelés "beans". Notez que ce concept est différent de celui défini par Sun dans la spécification JavaBeans. Rien n'oblige à ce que les beans déclarés dans le conteneur Spring IoC soient des JavaBeans. Ils peuvent être des objets Java tout simples (POJO, *Plain Old Java Object*). Le terme *POJO* fait référence à un objet Java ordinaire qui ne répond à aucune exigence particulière, comme

implémenter une interface spécifique ou dériver d'une certaine classe de base. Il permet de distinguer les composants Java légers des composants lourds existant dans d'autres modèles de composants complexes (par exemple les composants EJB dans les versions antérieures à la 3.0).

À la fin de ce chapitre, vous serez capable de construire une application Java complète fondée sur le conteneur Spring IoC. Par ailleurs, si vous examinez alors vos anciennes applications Java, vous pourriez constater qu'elles pourraient être grandement simplifiées et améliorées en utilisant ce conteneur.

3.1 Configurer des beans dans le conteneur Spring IoC

Problème

Spring fournit un conteneur IoC puissant pour la gestion des beans qui composent une application. Pour bénéficier des services de ce conteneur, nous devons configurer nos beans de telle sorte qu'ils s'y exécutent.

Solution

La configuration des beans dans le conteneur Spring IoC peut se faire avec des fichiers XML, des fichiers de propriétés ou même des API. En raison de sa simplicité et de sa maturité, la configuration à base de fichiers XML a été choisie dans cet ouvrage. Si vous êtes intéressé par les autres méthodes, consultez la documentation de Spring, qui détaille tous les aspects de la configuration des beans.

Spring permet de configurer les beans dans un ou plusieurs fichiers de configuration. Dans le cas d'une application simple, nous pouvons regrouper les beans dans un seul fichier. En revanche, pour une application plus complexe, avec un grand nombre de beans, il est préférable de les répartir dans plusieurs fichiers de configuration selon leurs fonctionnalités.

Explications

Supposons que nous développons une application pour la génération séquentielle de numéros. Elle doit pouvoir contenir différents ensembles de numéros en séquence pour répondre à des objectifs variés. Chaque ensemble dispose de ses propres valeurs de préfixe, de suffixe et de départ. Nous devons par conséquent créer et maintenir plusieurs instances du générateur dans notre application.

Créer la classe du bean

Conformément aux exigences, nous créons la classe SequenceGenerator avec les trois propriétés prefix, suffix et initial, qui peuvent être injectées par l’intermédiaire de mutateurs ou du constructeur. Le champ privé counter enregistre la valeur numérique actuelle du générateur. Chaque fois que nous invoquons la méthode getSequence() sur une instance du générateur, nous recevons le dernier numéro de séquence, auquel sont ajoutés le préfixe et le suffixe. Nous déclarons cette méthode synchronized pour qu’elle soit sûre vis-à-vis des threads.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {

    private String prefix;
    private String suffix;
    private int initial;
    private int counter;

    public SequenceGenerator() {}

    public SequenceGenerator(String prefix, String suffix, int initial) {
        this.prefix = prefix;
        this.suffix = suffix;
        this.initial = initial;
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }

    public void setInitial(int initial) {
        this.initial = initial;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        buffer.append(prefix);
        buffer.append(initial + counter++);
        buffer.append(suffix);
        return buffer.toString();
    }
}
```

Cette classe SequenceGenerator applique à la fois l’injection par constructeur et l’injection par mutateur pour les propriétés prefix, suffix et initial.

Créer le fichier de configuration des beans

Pour déclarer des beans dans le conteneur Spring IoC, nous devons tout d'abord créer un fichier XML de configuration des beans et le nommer de manière appropriée, par exemple `beans.xml`. Ce fichier est placé à la racine du chemin d'accès aux classes pour faciliter les tests depuis un IDE. Au début du fichier, nous indiquons la DTD Spring 2.0 de manière à importer la structure valide d'un fichier de configuration des beans pour Spring 2.x. Il est possible de définir un ou plusieurs beans sous l'élément racine `<beans>`.

```
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
  "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
  ...
</beans>
```

Spring affiche une rétrocompatibilité au niveau de sa configuration. Nous pouvons donc utiliser nos fichiers de configuration 1.0 existants (avec la DTD Spring 1.0) dans Spring 2.x. Toutefois, en procédant ainsi, nous ne pourrons pas bénéficier des nouvelles fonctions de configuration ajoutées dans Spring 2.x. L'usage des anciens fichiers de configuration doit être réservé à une phase de transition.

Spring 2.x accepte également d'utiliser XSD pour définir la structure valide du fichier XML de configuration des beans. XSD présente de nombreux avantages par rapport à une DTD classique. Dans Spring 2.x, le plus important est qu'il nous permet d'utiliser des balises personnalisées issues de différents schémas pour simplifier et clarifier la configuration. Il est donc fortement recommandé de choisir Spring XSD à la place de la DTD dès lors que c'est possible. Spring XSD est une version spécifique mais rétrocompatible. Si nous utilisons Spring 2.5, nous devons choisir Spring 2.5 XSD pour bénéficier des nouveautés de la version 2.5.

```
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
  ...
</beans>
```

Déclarer des beans dans le fichier de configuration

Pour que le conteneur Spring IoC puisse l'instancier, chaque bean doit posséder un nom unique et un nom de classe complet. Pour chaque propriété de type simple, par exemple `String` et les autres types primitifs, nous pouvons ajouter un élément `<value>`. Spring se charge de convertir la valeur dans le type déclaré de cette propriété. Pour configurer une propriété à l'aide de l'injection par mutateur, nous utilisons l'élément `<property>` et indiquons le nom de la propriété dans l'attribut `name`.

```
<bean name="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix">
      <value>30</value>
    </property>
    <property name="suffix">
      <value>A</value>
    </property>
    <property name="initial">
      <value>100000</value>
    </property>
  </bean>
```

Nous pouvons également configurer les propriétés du bean à l'aide de l'injection par constructeur. Pour cela, il suffit de les déclarer dans des éléments `<constructor-arg>`. Cet élément ne gère pas d'attribut `name` car les arguments du constructeur sont définis par leur emplacement.

```
<bean name="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg>
      <value>30</value>
    </constructor-arg>
    <constructor-arg>
      <value>A</value>
    </constructor-arg>
    <constructor-arg>
      <value>100000</value>
    </constructor-arg>
  </bean>
```

Dans le conteneur Spring IoC, le nom de chaque bean doit être unique, même s'il est possible d'employer plusieurs fois le même nom pour modifier une déclaration. Le nom d'un bean peut être donné dans l'attribut `name` de l'élément `<bean>`. Cependant, il existe une autre solution, recommandée. Elle consiste à utiliser l'attribut `id` standard, qui identifie un élément dans un document XML. Si l'éditeur de texte comprend la syntaxe XML, il peut ainsi nous aider à vérifier l'unicité de chaque bean lors de la conception.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
</bean>
```

XML impose certaines restrictions sur les caractères acceptés dans l'attribut `id`, mais, en général, personne n'utilise de caractères spéciaux dans un nom de bean. Par ailleurs, avec l'attribut `name`, il est possible de donner plusieurs noms à un bean, en les séparant par des virgules. Ce n'est pas le cas avec l'attribut `id` car les virgules n'y sont pas acceptées.

En réalité, rien ne nous oblige à nommer ou à identifier le bean. Il s'agit alors d'un *bean anonyme*.

Définir les propriétés du bean avec des raccourcis

Spring propose un raccourci pour préciser la valeur d'une propriété de type simple : définir l'attribut `value` dans l'élément `<property>` au lieu d'ajouter un sous-élément `<value>`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefix" value="30" />
    <property name="suffix" value="A" />
    <property name="initial" value="100000" />
</bean>
```

Ce raccourci fonctionne également avec les arguments du constructeur.

```
<bean name="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg value="30" />
    <constructor-arg value="A" />
    <constructor-arg value="100000" />
</bean>
```

Spring 2.x propose un autre raccourci pratique pour définir des propriétés. Il s'agit d'utiliser le schéma `p` et de définir les propriétés sous forme d'attributs dans l'élément `<bean>`. Les lignes de la configuration XML sont ainsi plus courtes.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

  <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        p:prefix="30" p:suffix="A" p:initial="100000" />
</beans>
```

3.2 Instancier le conteneur Spring IoC

Problème

Nous devons instancier le conteneur Spring IoC pour qu'il crée des instances de beans d'après leur configuration. Ensuite, nous voulons obtenir des instances de beans à partir du conteneur pour les utiliser.

Solution

Spring fournit deux implémentations du conteneur IoC. La version basique est appelée *fabrique de beans*. La version plus élaborée est appelée *contexte d'application* ; elle est compatible avec la fabrique de beans. Ces deux types de conteneurs IoC utilisent des fichiers de configuration des beans identiques.

Le contexte d'application offre des fonctions plus élaborées que la fabrique de beans, tout en conservant la compatibilité des fonctionnalités de base. Par conséquent, il est fortement conseillé d'utiliser le contexte d'application, excepté lorsque les ressources de l'application sont limitées, par exemple lorsqu'elle s'exécute dans une applet ou sur un périphérique mobile.

Les interfaces de la fabrique de beans et du contexte d'application sont respectivement `BeanFactory` et `ApplicationContext`. Cette dernière étend `BeanFactory` pour assurer la compatibilité.

Explications

Instancier une fabrique de beans

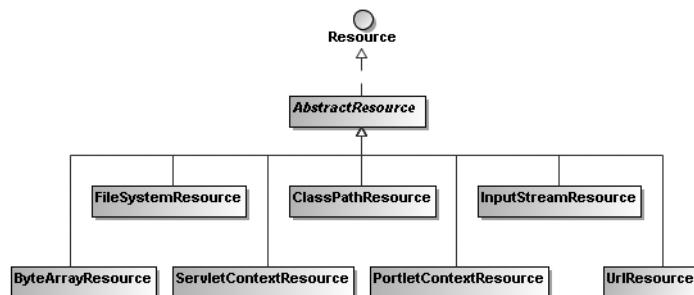
Pour instancier une fabrique de beans, nous devons commencer par charger le fichier de configuration des beans dans un objet `Resource`. Par exemple, l'instruction suivante charge le fichier de configuration à partir de la racine du chemin d'accès aux classes :

```
Resource resource = new ClassPathResource("beans.xml");
```

`Resource` n'est qu'une interface, tandis que `ClassPathResource` est l'une de ses implémentations pour charger une ressource à partir du chemin d'accès aux classes. D'autres implémentations de cette interface, comme `FileSystemResource`, `InputStreamResource` et `UrlResource`, permettent de charger une ressource à partir d'emplacements différents. La Figure 3.1 présente les implémentations de l'interface `Resource` proposées par Spring.

Figure 3.1

Implémentations communes de l'interface Resource.



Ensuite, nous utilisons l'instruction suivante pour instancier une fabrique de beans en lui passant l'objet `Resource` dans lequel a été chargé le fichier de configuration :

```
BeanFactory factory = new XmlBeanFactory(resource);
```

Nous l'avons mentionné, BeanFactory est une interface qui permet de rendre abstraites les opérations sur une fabrique de beans, tandis que XmlBeanFactory est l'implémentation qui construit une fabrique de beans à partir d'un fichier de configuration XML.

Instancier un contexte d'application

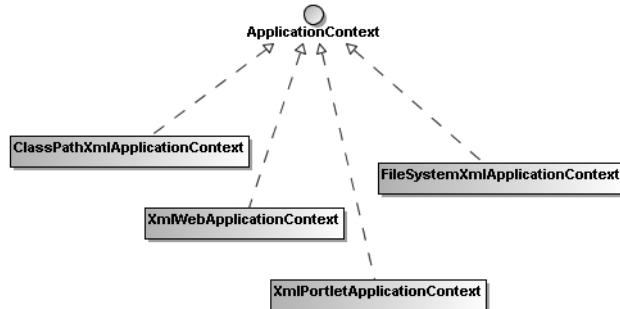
Comme BeanFactory, ApplicationContext est une interface et nous devons donc instancier l'une de ses implémentations. La classe ClassPathXmlApplicationContext construit un contexte d'application en chargeant un fichier de configuration XML à partir du chemin d'accès aux classes. Nous pouvons également lui indiquer plusieurs fichiers de configuration.

```
ApplicationContext context = new ClassPathXmlApplicationContext("beans.xml");
```

Outre ClassPathXmlApplicationContext, Spring propose d'autres implémentations d'ApplicationContext. FileSystemXmlApplicationContext charge des fichiers de configuration XML à partir du système de fichiers, tandis que XmlWebApplicationContext et XmlPortletApplicationContext peuvent uniquement être utilisées dans des applications et des portails web. La Figure 3.2 présente les implémentations de l'interface ApplicationContext disponibles dans Spring.

Figure 3.2

Implémentations communes de l'interface ApplicationContext.



Obtenir des beans depuis le conteneur IoC

Pour obtenir un bean déclaré à partir d'une fabrique de beans ou d'un contexte d'application, il suffit d'invoquer la méthode getBean() en lui passant le nom unique du bean. Cette méthode retourne un java.lang.Object, dont nous devons forcer le type avant de pouvoir l'utiliser.

```
SequenceGenerator generator =
    (SequenceGenerator) context.getBean("sequenceGenerator");
```

À partir de là, nous pouvons employer le bean comme n’importe quel autre objet créé à l’aide d’un constructeur. L’intégralité du code source pour l’exécution du générateur séquentiel de numéros est donnée dans la classe Main suivante :

```
package com.apress.springrecipes.sequence;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        SequenceGenerator generator =
            (SequenceGenerator) context.getBean("sequenceGenerator");

        System.out.println(generator.getSequence());
        System.out.println(generator.getSequence());
    }
}
```

Si tout se passe bien, nous voyons s’afficher les numéros de séquence suivants, après quelques messages de journalisation qui ne nous intéressent pas :

```
30100000A
30100001A
```

3.3 Lever les ambiguïtés sur le constructeur

Problème

Lorsque nous indiquons un ou plusieurs arguments de constructeur pour un bean, Spring tente de trouver le constructeur approprié dans la classe du bean et lui passe nos arguments lors de l’instanciation. Cependant, si la liste des arguments correspond à plusieurs constructeurs, il existe une ambiguïté sur le choix du constructeur. Dans ce cas, Spring pourrait ne pas invoquer le constructeur attendu.

Solution

Nous pouvons renseigner les attributs `type` et `index` de l’élément `<constructor-arg>` de manière à aider Spring dans sa recherche du constructeur.

Explications

Ajoutons un nouveau constructeur à la classe `SequenceGenerator` avec les arguments `prefix` et `suffix`.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }
}
```

Dans la déclaration du bean, nous pouvons préciser un ou plusieurs arguments du constructeur grâce à des éléments <constructor-arg>. Spring essaiera de trouver le constructeur pour cette classe et lui passera nos arguments lors de l'instanciation du bean. Il ne faut pas oublier que l'attribut name n'existe pas dans l'élément <constructor-arg>, car les arguments du constructeur sont fonction de leur emplacement.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg value="30" />
    <constructor-arg value="A" />
    <property name="initial" value="100000" />
</bean>
```

Spring n'a aucun mal à trouver un constructeur correspondant à ces deux arguments car il n'en existe qu'un seul. Supposons que nous ayons ajouté un autre constructeur à SequenceGenerator, avec les arguments prefix et initial.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }

    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }
}
```

Pour invoquer ce constructeur en passant un préfixe et une valeur initiale, nous déclarons le bean de la manière suivante. Le suffixe est injecté *via* le mutateur.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg value="30" />
    <constructor-arg value="100000" />
    <property name="suffix" value="A" />
</bean>
```

Cependant, si nous exécutons à présent l'application, nous obtenons le résultat suivant :

```
300A
301A
```

Ce résultat inattendu provient de l’invocation du premier constructeur, avec `prefix` et `suffix` comme arguments, au lieu du second. En effet, Spring a converti par défaut nos deux arguments en type `String` et a considéré que le premier constructeur était mieux adapté car il n’exigeait aucune conversion de type. Pour préciser le type attendu de nos arguments, nous utilisons l’attribut `type` de `<constructor-arg>`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="java.lang.String" value="30" />
    <constructor-arg type="int" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

Ajoutons un autre constructeur à `SequenceGenerator`, avec `initial` et `suffix` comme arguments, et modifions en conséquence la déclaration du bean.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    public SequenceGenerator(String prefix, String suffix) {
        this.prefix = prefix;
        this.suffix = suffix;
    }

    public SequenceGenerator(String prefix, int initial) {
        this.prefix = prefix;
        this.initial = initial;
    }

    public SequenceGenerator(int initial, String suffix) {
        this.initial = initial;
        this.suffix = suffix;
    }
}

---

<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="int" value="100000" />
    <constructor-arg type="java.lang.String" value="A" />
    <property name="prefix" value="30" />
</bean>
```

Si nous exécutons à nouveau l’application, nous pouvons obtenir le bon résultat ou le résultat inattendu suivant :

```
30100000null
30100001null
```

Cette incertitude vient du fait que Spring évalue de manière interne la compatibilité de chaque constructeur avec nos arguments. Au cours du processus d’évaluation, l’ordre d’apparition de nos arguments dans le fichier XML n’est pas pris en compte. Autrement

dit, du point de vue de Spring, le deuxième et le troisième constructeurs sont équivalents. Le constructeur choisi est celui trouvé en premier. Selon l'API Java Reflection ou, plus précisément, la méthode `Class.getDeclaredConstructors()`, les constructeurs rentrés sont dans un ordre quelconque qui peut varier de l'ordre de déclaration. En raison de tous ces éléments, il existe une ambiguïté sur la correspondance du constructeur.

Pour éviter ce problème, nous devons indiquer explicitement les indices des arguments à l'aide de l'attribut `index` de `<constructor-arg>`. Lorsque les attributs `type` et `index` sont utilisés, Spring est en mesure de trouver précisément le constructeur voulu d'un bean.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg type="int" index="0" value="100000" />
    <constructor-arg type="java.lang.String" index="1" value="A" />
    <property name="prefix" value="30" />
</bean>
```

Si nous sommes certains que nos constructeurs ne seront pas source d'ambiguïté, nous pouvons omettre les attributs `type` et `index`.

3.4 Préciser des références de beans

Problème

Les beans qui composent une application doivent souvent collaborer les uns avec les autres pour mettre en œuvre ses fonctionnalités. Pour cela, nous devons préciser les différentes références entre les beans dans le fichier de configuration des beans.

Solution

Dans le fichier de configuration, nous pouvons affecter une référence de bean à une propriété de bean ou à un argument de constructeur à l'aide de l'élément `<ref>`. La procédure n'est pas plus compliquée que la définition d'une simple valeur avec l'élément `<value>`. Nous pouvons également placer directement une déclaration de bean dans une propriété ou un argument de constructeur sous forme de bean interne.

Explications

Pour le moment, la seule valeur acceptée comme préfixe pour notre générateur séquentiel est une chaîne de caractères. Cela n'est évidemment pas suffisamment souple pour répondre aux exigences futures. Il serait préférable que le préfixe soit fourni par une forme de logique programmée. Nous créons donc l'interface `PrefixGenerator` pour définir l'opération de génération du préfixe.

```
package com.apress.springrecipes.sequence;  
  
public interface PrefixGenerator {  
    public String getPrefix();  
}
```

Une stratégie de génération consiste à mettre en forme la date système actuelle à l'aide d'un motif spécifique. Nous créons la classe `DatePrefixGenerator`, qui implémente l'interface `PrefixGenerator`.

```
package com.apress.springrecipes.sequence;  
...  
public class DatePrefixGenerator implements PrefixGenerator {  
  
    private DateFormat formatter;  
  
    public void setPattern(String pattern) {  
        this.formatter = new SimpleDateFormat(pattern);  
    }  
  
    public String getPrefix() {  
        return formatter.format(new Date());  
    }  
}
```

Le motif est injecté par l'intermédiaire du mutateur `setPattern()` et sert ensuite à créer un objet `java.text.DateFormat` de mise en forme de la date. Puisque la chaîne du motif n'est plus utilisée après avoir créé l'objet `DateFormat`, il est inutile de l'enregistrer dans un champ privé.

Nous pouvons à présent déclarer un bean de type `DatePrefixGenerator` avec une chaîne de motif quelconque pour mettre en forme la date.

```
<bean id="datePrefixGenerator"  
      class="com.apress.springrecipes.sequence.DatePrefixGenerator">  
    <property name="pattern" value="yyyyMMdd" />  
</bean>
```

Préciser les références de beans pour les mutateurs

Pour profiter de cette génération du préfixe, la classe `SequenceGenerator` doit accepter un objet de type `PrefixGenerator` à la place d'une simple chaîne de caractères. Pour cela, nous pouvons choisir l'injection par mutateur. Nous supprimons la propriété `prefix`, ainsi que ses mutateurs et constructeurs, qui sont source d'erreurs de compilation.

```
package com.apress.springrecipes.sequence;  
  
public class SequenceGenerator {  
    ...  
    private PrefixGenerator prefixGenerator;
```

```
public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
    this.prefixGenerator = prefixGenerator;
}

public synchronized String getSequence() {
    StringBuffer buffer = new StringBuffer();
    buffer.append(prefixGenerator.getPrefix());
    buffer.append(initial + counter++);
    buffer.append(suffix);
    return buffer.toString();
}
}
```

Ensuite, un bean SequenceGenerator peut faire référence au bean datePrefixGenerator dans sa propriété prefixGenerator en utilisant un élément <ref>.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator">
        <ref bean="datePrefixGenerator" />
    </property>
</bean>
```

Le nom du bean dans l'attribut bean de l'élément <ref> peut être une référence à n'importe quel bean du conteneur IoC, même s'il est défini dans un autre fichier de configuration XML. Dans le cas d'une référence à un bean présent dans le même fichier XML, nous pouvons employer l'attribut local puisqu'il s'agit d'une référence à un identifiant XML. Notre éditeur XML peut alors vérifier si un bean possédant cet identifiant existe bien dans le même fichier XML (intégrité de référence).

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="prefixGenerator">
        <ref local="datePrefixGenerator" />
    </property>
</bean>
```

Il existe également un raccourci qui permet de préciser une référence de bean dans l'attribut ref d'un élément <property>.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="prefixGenerator" ref="datePrefixGenerator" />
</bean>
```

Toutefois, cette méthode ne permet pas à notre éditeur XML de vérifier l'intégrité des références. En réalité, elle équivaut à utiliser l'attribut bean dans un élément <ref>.

Spring 2.x propose un autre raccourci pour préciser des références de beans : utiliser le schéma p et préciser les références sous forme d'attributs dans l'élément <bean>. Les lignes de la configuration XML sont ainsi plus courtes.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="sequenceGenerator"
          class="com.apress.springrecipes.sequence.SequenceGenerator"
          p:suffix="A" p:initial="1000000"
          p:prefixGenerator-ref="datePrefixGenerator" />
</beans>
```

Pour différencier une référence de bean d'une simple valeur de propriété, nous devons ajouter le suffixe `-ref` au nom de la propriété.

Preciser des références de beans pour les arguments d'un constructeur

Des références de beans peuvent également être appliquées en utilisant l'injection par constructeur. Ajoutons, par exemple, un constructeur qui accepte un objet Prefix-Generator en argument.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {
    ...
    private PrefixGenerator prefixGenerator;

    public SequenceGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

Comme nous l'avons fait dans l'élément `<property>`, nous incluons une référence de bean avec `<ref>` dans l'élément `<constructor-arg>`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg>
        <ref local="datePrefixGenerator" />
    </constructor-arg>
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

Le raccourci précédent pour indiquer une référence de bean fonctionne également avec `<constructor-arg>`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg ref="datePrefixGenerator" />
    ...
</bean>
```

Déclarer des beans internes

Dès lors qu'une instance de bean n'est utilisée que dans une seule propriété, elle peut être déclarée sous forme de bean interne. Une déclaration de bean interne est incluse directement dans un élément `<property>` ou `<constructor-arg>`, sans préciser d'attribut `id` ou `name`. De cette manière, le bean est anonyme et ne peut pas être employé ailleurs. En réalité, si nous affectons un attribut `id` ou `name` à un bean interne, il est ignoré.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
    <property name="prefixGenerator">
      <bean class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
      </bean>
    </property>
  </bean>
```

Un bean interne peut également être déclaré dans un argument de constructeur.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <constructor-arg>
      <bean class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
      </bean>
    </constructor-arg>
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
  </bean>
```

3.5 Contrôler les propriétés par vérification des dépendances

Problème

Dans une application réelle, des centaines ou des milliers de beans peuvent être déclarés dans le conteneur IoC, avec des dépendances souvent très complexes. Avec l'injection par mutateur, il est impossible de certifier qu'une propriété sera injectée. Il est donc très difficile de contrôler que toutes les propriétés requises sont définies.

Solution

Spring offre une fonction de vérification des dépendances qui peut nous aider à contrôler si toutes les propriétés d'un certain type ont été définies pour un bean. Il suffit de préciser le mode de vérification des dépendances dans l'attribut `dependency-check` de l'élément `<bean>`. Cette fonctionnalité vérifie uniquement si les propriétés ont été définies, non si leur valeur est différente de null. Le Tableau 3.1 recense tous les modes de vérification des dépendances reconnus par Spring.

Tableau 3.1 : Modes de vérification des dépendances reconnus par Spring

<i>Mode</i>	<i>Description</i>
none ¹	Aucune vérification des dépendances n'est effectuée. Des propriétés peuvent rester indéfinies.
simple	Si une propriété de type simple (types primitifs et collections) n'a pas été fixée, une exception <code>UnsatisfiedDependencyException</code> est lancée.
objects	Si une propriété de type objet (autre que les types simples) n'a pas été fixée, une exception <code>UnsatisfiedDependencyException</code> est lancée.
all	Si une propriété de n'importe quel type n'a pas été fixée, une exception <code>UnsatisfiedDependencyException</code> est lancée.

1. Le mode par défaut est `none`, mais il est possible de le changer à l'aide de l'attribut `default-dependency-check` de l'élément racine `<beans>`. Ce mode par défaut est écrasé par tout mode précis sur un bean. Cet attribut doit être employé avec prudence car il affecte le mode de vérification des dépendances par défaut pour tous les beans du conteneur IoC.

Explications

Contrôler les propriétés de type simple

Supposons que la propriété `suffix` n'ait pas été fixée par le générateur séquentiel. Dans ce cas, le générateur produit des numéros dont le suffixe est la chaîne de caractères `null`. Ce type de problème est souvent très difficile à déboguer, en particulier si le bean est complexe. Heureusement, Spring est capable de vérifier si toutes les propriétés d'un certain type ont été fixées. Pour demander à Spring de contrôler les propriétés de type simple (c'est-à-dire les types primitifs et les collections), nous affectons la valeur `simple` à l'attribut `dependency-check` de l'élément `<bean>`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator"
      dependency-check="simple">
    <property name="initial" value="100000" />
    <property name="prefixGenerator" ref="datePrefixGenerator" />
</bean>
```

Si une propriété de type simple n'a pas été fixée, une exception `UnsatisfiedDependencyException` est lancée et précise la propriété en cause.

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error
creating bean with name 'sequenceGenerator' defined in class path resource
[beans.xml]: Unsatisfied dependency expressed through bean property 'suffix':
Set this property value or disable dependency checking for this bean.
```

Contrôler les propriétés de type objet

Si le générateur du préfixe n'est pas défini, l'exception `NullPointerException` est lancée au moment de la demande de génération du préfixe. Pour activer la vérification des dépendances sur les propriétés de type objet (c'est-à-dire non de type simple), nous affectons la valeur `objects` à l'attribut `dependency-check`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator"
      dependency-check="objects">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

Dans ce cas, Spring signale, lors de l'exécution de l'application, que la propriété `prefixGenerator` n'a pas été fixée.

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error
creating bean with name 'sequenceGenerator' defined in class path resource
[beans.xml]: Unsatisfied dependency expressed through bean property
'prefixGenerator': Set this property value or disable dependency checking for
this bean.
```

Contrôler les propriétés de n'importe quel type

Pour contrôler toutes les propriétés de bean, quel que soit leur type, nous fixons l'attribut `dependency-check` à `all`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator"
      dependency-check="all">
    <property name="initial" value="100000" />
</bean>
```

Vérification des dépendances et injection par constructeur

La fonctionnalité de vérification des dépendances de Spring contrôle uniquement si une propriété a été injectée par l'intermédiaire d'un mutateur. Par conséquent, même si le générateur de préfixe a été injecté via un constructeur, une exception `UnsatisfiedDependencyException` est lancée.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator"
      dependency-check="all">
    <constructor-arg ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

3.6 Contrôler les propriétés avec l'annotation `@Required`

Problème

La fonctionnalité de vérification des dépendances de Spring ne peut contrôler que les propriétés qui sont d'un certain type. Elle n'est pas suffisamment souple pour contrôler des propriétés spécifiques. Dans la plupart des cas, nous souhaitons contrôler si certaines propriétés ont été fixées, non toutes les propriétés d'un certain type.

Solution

`RequiredAnnotationBeanPostProcessor` est un postprocesseur de beans Spring qui vérifie si toutes les propriétés de bean annotées par `@Required` ont été fixées. Un *postprocesseur de beans* est une sorte de bean Spring spécial capable d'effectuer des opérations supplémentaires sur chaque bean avant son initialisation. Pour l'activer, nous devons l'enregistrer dans le conteneur Spring IoC. Il peut uniquement contrôler si les propriétés ont été fixées, non si leur valeur est différente de null.

Explications

Supposons que les propriétés `prefixGenerator` et `suffix` soient obligatoires pour un générateur séquentiel. Nous pouvons alors annoter leur mutateur avec `@Required`.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Required;

public class SequenceGenerator {

    private PrefixGenerator prefixGenerator;
    private String suffix;
    ...
    @Required
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }

    @Required
    public void setSuffix(String suffix) {
        this.suffix = suffix;
    }
    ...
}
```

Pour demander à Spring de contrôler si ces propriétés ont été fixées pour toutes les instances du générateur séquentiel, nous devons enregistrer une instance de `RequiredAnnotationBeanPostProcessor` dans le conteneur IoC. Si nous utilisons une fabrique de beans, l'enregistrement de ce postprocesseur doit se faire au travers de l'API. Sinon il suffit de déclarer une instance de ce postprocesseur dans le contexte d'application.

```
<bean class="org.springframework.beans.factory.annotation.>
    RequiredAnnotationBeanPostProcessor" />
```

Avec Spring 2.5, nous pouvons simplement inclure l'élément `<context:annotation-config>` dans le fichier de configuration des beans pour qu'une instance de `RequiredAnnotationBeanPostProcessor` soit automatiquement enregistrée.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:annotation-config />
    ...
</beans>
```

Dans l'éventualité où des propriétés marquées par `@Required` n'ont pas été fixées, une exception `BeanInitializationException` est lancée par ce postprocesseur de beans.

```
Exception in thread "main"
org.springframework.beans.factory.BeanCreationException: Error creating bean
with name 'sequenceGenerator' defined in class path resource [beans.xml]:
Initialization of bean failed; nested exception is
org.springframework.beans.factory.BeanInitializationException: Property
'prefixGenerator' is required for bean 'sequenceGenerator'
```

Outre l'annotation `@Required`, `RequiredAnnotationBeanPostProcessor` peut également contrôler les propriétés marquées par des annotations personnalisées. Créons, par exemple, l'annotation suivante :

```
package com.apress.springrecipes.sequence;
...
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.METHOD)
public @interface Mandatory {
}
```

Nous pouvons ensuite l'appliquer aux mutateurs des propriétés obligatoires.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {

    private PrefixGenerator prefixGenerator;
    private String suffix;
    ...
    @Mandatory
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

```

@Mandatory
public void setSuffix(String suffix) {
    this.suffix = suffix;
}
...
}

```

Pour contrôler les propriétés ayant cette annotation, nous devons l'indiquer dans la propriété `requiredAnnotationType` de `RequiredAnnotationBeanPostProcessor`.

```

<bean class="org.springframework.beans.factory.annotation.<span style="color: #0070C0;">➥
      RequiredAnnotationBeanPostProcessor">
    <property name="requiredAnnotationType">
      <value>com.apress.springrecipes.sequence.Mandatory</value>
    </property>
  </bean>

```

3.7 Lier automatiquement des beans par configuration XML

Problème

Lorsqu'un bean doit accéder à un autre bean, nous pouvons les lier en précisant explicitement la référence. Toutefois, si le conteneur pouvait lier automatiquement nos beans, nous ferions l'économie de la configuration manuelle des dépendances.

Solution

Le conteneur Spring IoC peut nous aider à lier automatiquement des beans. Il suffit d'indiquer le mode de liaison automatique dans l'attribut `autowire` de l'élément `<bean>`. Le Tableau 3.2 recense les modes de liaison automatique reconnus par Spring.

Tableau 3.2 : Modes de liaison automatique reconnus par Spring

<i>Mode</i>	<i>Description</i>
no ¹	Aucune liaison automatique n'est réalisée. Les dépendances doivent être établies explicitement.
byName	Pour chaque propriété du bean, lier un bean dont le nom correspond à celui de la propriété.
byType	Pour chaque propriété du bean, lier un bean dont le type est compatible avec celui de la propriété. Si plusieurs beans sont trouvés, une exception <code>UnsatisfiedDependencyException</code> est lancée.
constructor	Pour chaque argument de chaque constructeur, commencer par trouver un bean dont le type est compatible avec celui de l'argument. Ensuite, retenir le constructeur ayant le plus grand nombre d'arguments correspondants. En cas d'ambiguïté, une exception <code>UnsatisfiedDependencyException</code> est lancée.

Tableau 3.2 : Modes de liaison automatique reconnus par Spring (suite)

<i>Mode</i>	<i>Description</i>
autodetect	Si un constructeur par défaut sans argument est trouvé, les dépendances sont injectées automatiquement en fonction du type. Sinon elles sont établies automatiquement par constructeur.

1. Le mode par défaut est no, mais il est possible de changer cette configuration en fixant l'attribut default-autowire de l'élément racine <beans>. Le mode par défaut est écrasé par le mode défini explicitement sur un bean.

Bien que la fonctionnalité de liaison automatique soit très puissante, elle a pour inconvénient de diminuer la lisibilité de la configuration des beans. Puisque la liaison automatique est effectuée par Spring au moment de l'exécution, nous ne pouvons pas déduire les liaisons des beans à partir du fichier de configuration. En pratique, il est conseillé de mettre en place la liaison automatique uniquement dans les applications dont les dépendances entre composants restent simples.

Explications

Liaison automatique par type

Nous pouvons fixer l'attribut autowire du bean sequenceGenerator à la valeur byType et laisser la propriété prefixGenerator non fixée. Spring tente alors de lier un bean dont le type est compatible avec PrefixGenerator. Dans ce cas, le bean datePrefix-Generator sera lié automatiquement.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator"
        autowire="byType">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

La liaison automatique par type présente un problème majeur : plusieurs beans du conteneur IoC peuvent être compatibles avec le type cible. Dans ce cas, Spring n'est pas en mesure de choisir le bean adapté à la propriété et ne peut donc pas effectuer de liaison automatique. Par exemple, s'il existe un autre générateur qui utilise l'année courante comme préfixe, la liaison automatique par type ne fonctionne pas.

```
<beans ...>
    <bean id="sequenceGenerator"
          class="com.apress.springrecipes.sequence.SequenceGenerator"
          autowire="byType">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
          class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>

    <bean id="yearPrefixGenerator"
          class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyy" />
    </bean>
</beans>
```

Spring lance une exception `UnsatisfiedDependencyException` lorsque plusieurs beans peuvent servir à la liaison automatique.

```
Exception in thread "main"
org.springframework.beans.factory.UnsatisfiedDependencyException: Error
creating bean with name 'sequenceGenerator' defined in class path resource
[beans.xml]: Unsatisfied dependency expressed through bean property
'prefixGenerator': No unique bean of type
[com.apress.springrecipes.sequence.PrefixGenerator] is defined: expected single
matching bean but found 2: [datePrefixGenerator, yearPrefixGenerator]
```

Liaison automatique par nom

Le mode de liaison automatique `byName` permet parfois de résoudre les problèmes de la liaison automatique `byType`. Il fonctionne de manière comparable à la liaison par type, mais, dans ce cas, Spring tente de lier un bean de même nom à la place d'un bean de type compatible. Puisque les noms des beans sont uniques au sein d'un conteneur, la liaison automatique par nom n'est pas ambiguë.

```
<beans ...>
    <bean id="sequenceGenerator"
          class="com.apress.springrecipes.sequence.SequenceGenerator"
          autowire="byName">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="prefixGenerator"
          class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

Toutefois, ce mode ne fonctionne pas dans tous les cas. Il est parfois impossible que le nom du bean cible soit le même que celui de la propriété. En pratique, il est souvent nécessaire de préciser explicitement les dépendances ambiguës, tout en établissant automatiquement les autres. Autrement dit, il faut combiner les liaisons explicites et les liaisons automatiques.

Liaison automatique par constructeur

La liaison automatique par constructeur fonctionne de manière semblable au mode `byType`, mais elle est plus complexe. Dans le cas d'un bean avec un seul constructeur, Spring tente de lier à chaque argument du constructeur un bean dont le type est compatible avec celui de cet argument. En revanche, dans le cas d'un bean avec plusieurs constructeurs, le processus est plus complexe. Spring commence par rechercher un bean dont le type est compatible avec celui de chaque argument de chaque constructeur. Ensuite, il choisit le constructeur qui présente le plus grand nombre d'arguments correspondants.

Supposons que `SequenceGenerator` ait un constructeur par défaut et un constructeur avec un argument de type `PrefixGenerator`.

```
package com.apress.springrecipes.sequence;

public class SequenceGenerator {

    public SequenceGenerator() {}

    public SequenceGenerator(PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
    ...
}
```

Dans ce cas, le deuxième constructeur est retenu car Spring peut trouver un bean dont le type est compatible avec `PrefixGenerator`.

```
<beans ...>
    <bean id="sequenceGenerator"
          class="com.apress.springrecipes.sequence.SequenceGenerator"
          autowire="constructor">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
          class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

Cependant, les multiples constructeurs d'une classe peuvent créer une ambiguïté dans la correspondance de leurs arguments. La situation devient encore plus complexe si nous demandons à Spring de déterminer un constructeur à notre place. Lorsque ce mode de liaison automatique est utilisé, il faut faire très attention à éviter les ambiguïtés.

Liaison automatique par détection automatique

Le mode autodetect demande à Spring de choisir lui-même entre les modes de liaison automatique byType et constructor. Si un constructeur par défaut sans argument est trouvé pour le bean, Spring choisit byType. Sinon il opte pour le mode constructor. Puisque la classe SequenceGenerator définit un constructeur par défaut, la liaison automatique par type est choisie. Autrement dit, le générateur de préfixe est injecté via le mutateur.

```
<beans ...>
    <bean id="sequenceGenerator"
          class="com.apress.springrecipes.sequence.SequenceGenerator"
          autowire="autodetect">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
    </bean>

    <bean id="datePrefixGenerator"
          class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

Liaison automatique et vérification des dépendances

Nous l'avons vu, lorsque Spring trouve plusieurs beans candidats pour la liaison automatique, il lance une exception UnsatisfiedDependencyException. Par ailleurs, si le mode est fixé à byName ou à byType et si Spring ne trouve pas de bean adéquat, il laisse la propriété telle quelle, ce qui peut provoquer une exception NullPointerException ou conduire à une valeur non initialisée. Si nous le souhaitons, nous pouvons être informés de l'échec de la liaison automatique. Pour cela, il suffit d'affecter la valeur objects ou all à l'attribut dependency-check. Une exception UnsatisfiedDependencyException est alors lancée si la liaison automatique échoue.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator"
      autowire="byName" dependency-check="objects">
    <property name="initial" value="100000" />
    <property name="suffix" value="A" />
</bean>
```

3.8 Lier automatiquement des beans avec `@Autowired` et `@Resource`

Problème

La liaison automatique fondée sur l'attribut `autowire` dans le fichier de configuration des beans établit des dépendances pour toutes les propriétés d'un bean. Elle ne permet pas de lier uniquement certaines propriétés. Par ailleurs, il n'est possible de lier automatiquement des beans que par le type ou le nom. Lorsque aucune de ces stratégies ne répond à nos besoins, nous devons lier les beans explicitement.

Solution

Spring 2.5 apporte de nombreuses améliorations à la fonctionnalité de liaison automatique. Nous pouvons lier automatiquement une propriété en plaçant une annotation `@Autowired` ou `@Resource` sur un mutateur, un constructeur, un champ ou même une méthode. Ces annotations sont définies dans la JSR-250, *Common Annotations for the Java Platform*. Autrement dit, nous ne sommes pas réduits à l'attribut `autowire` pour satisfaire nos exigences. Toutefois, cette solution nous oblige à utiliser Java 1.5 ou une version ultérieure.

Explications

Pour demander à Spring de lier automatiquement les propriétés de bean marquées par `@Autowired` ou `@Resource`, nous devons enregistrer une instance de `AutowiredAnnotationBeanPostProcessor` dans le conteneur IoC. Si nous utilisons une fabrique de beans, l'enregistrement de ce postprocesseur doit se faire au travers de l'API. Sinon il nous suffit d'en déclarer une instance dans le contexte d'application.

```
<bean class="org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor" />
```

Ou bien nous pouvons simplement inclure l'élément `<context:annotation-config>` dans le fichier de configuration des beans pour qu'une instance de `AutowiredAnnotationBeanPostProcessor` soit automatiquement enregistrée.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:annotation-config />
    ...
</beans>
```

Liaison automatique d'un seul bean de type compatible

L'annotation `@Autowired` appliquée à une certaine propriété permet à Spring de la lier automatiquement. Par exemple, si nous annotons le mutateur de la propriété `prefixGenerator` avec `@Autowired`, Spring tente de lier un bean dont le type est compatible avec `PrefixGenerator`.

```
package com.apress.springrecipes.sequence;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class SequenceGenerator {  
    ...  
    @Autowired  
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {  
        this.prefixGenerator = prefixGenerator;  
    }  
}
```

Si un bean dont le type est compatible avec `PrefixGenerator` est défini dans le conteur IoC, il est automatiquement affecté à la propriété `prefixGenerator`.

```
<beans ...>  
    ...  
    <bean id="sequenceGenerator"  
          class="com.apress.springrecipes.sequence.SequenceGenerator">  
        <property name="initial" value="100000" />  
        <property name="suffix" value="A" />  
    </bean>  
  
    <bean id="datePrefixGenerator"  
          class="com.apress.springrecipes.sequence.DatePrefixGenerator">  
        <property name="pattern" value="yyyyMMdd" />  
    </bean>  
</beans>
```

Par défaut, toutes les propriétés marquées par `@Autowired` sont obligatoires. Lorsque Spring n'est pas en mesure de trouver un bean adapté, il lance une exception. Pour que la liaison d'une propriété soit facultative, nous devons affecter la valeur `false` à l'attribut `required` de `@Autowired`. Dans ce cas, si Spring ne trouve aucun bean adéquat, il laisse la propriété telle quelle.

```
package com.apress.springrecipes.sequence;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class SequenceGenerator {  
    ...  
    @Autowired(required = false)  
    public void setPrefixGenerator(PrefixGenerator prefixGenerator) {  
        this.prefixGenerator = prefixGenerator;  
    }  
}
```

L'annotation `@Autowired` s'applique non seulement à un mutateur, mais également à un constructeur. Spring tente alors de trouver un bean de type compatible pour chaque argument du constructeur.

```
package com.apress.springrecipes.sequence;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class SequenceGenerator {  
    ...  
    @Autowired  
    public SequenceGenerator(PrefixGenerator prefixGenerator) {  
        this.prefixGenerator = prefixGenerator;  
    }  
}
```

L'annotation `@Autowired` peut également être appliquée à un champ, même si l'il n'est pas déclaré `public`. Nous pouvons ainsi omettre la déclaration d'un mutateur ou d'un constructeur pour ce champ. Spring injecte le bean correspondant dans ce champ par l'intermédiaire du mécanisme de réflexion. Toutefois, l'annotation d'un champ non public avec `@Autowired` diminue les possibilités de test du code car il devient réfractaire aux tests unitaires.

```
package com.apress.springrecipes.sequence;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class SequenceGenerator {  
    ...  
    @Autowired  
    private PrefixGenerator prefixGenerator;  
    ...  
}
```

Nous pouvons même appliquer l'annotation `@Autowired` à une méthode de nom quelconque et au nombre d'arguments quelconque. Spring tente alors de lier un bean de type compatible pour chacun des arguments de la méthode.

```
package com.apress.springrecipes.sequence;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class SequenceGenerator {  
    ...  
    @Autowired  
    public void inject(PrefixGenerator prefixGenerator) {  
        this.prefixGenerator = prefixGenerator;  
    }  
}
```

Liaison automatique de tous les beans de type compatible

Lorsque l'annotation `@Autowired` est appliquée à une propriété de type tableau, Spring lie automatiquement tous les beans correspondants. Par exemple, si nous marquons une propriété `PrefixGenerator[]` avec `@Autowired`, Spring lie automatiquement et en une seule fois tous les beans dont le type est compatible avec `PrefixGenerator`.

```
package com.apress.springrecipes.sequence;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class SequenceGenerator {  
  
    @Autowired  
    private PrefixGenerator[] prefixGenerators;  
    ...  
}
```

Si plusieurs beans de type compatible avec `PrefixGenerator` sont définis dans le conteneur IoC, ils sont ajoutés automatiquement au tableau `prefixGenerators`.

```
<beans ...>  
    ...  
    <bean id="datePrefixGenerator"  
          class="com.apress.springrecipes.sequence.DatePrefixGenerator">  
        <property name="pattern" value="yyyyMMdd" />  
    </bean>  
  
    <bean id="yearPrefixGenerator"  
          class="com.apress.springrecipes.sequence.DatePrefixGenerator">  
        <property name="pattern" value="yyyy" />  
    </bean>  
</beans>
```

De la même manière, nous pouvons appliquer l'annotation `@Autowired` à une collection typée. Spring est capable de lire l'information de type de cette collection et de lier automatiquement tous les beans de type compatible.

```
package com.apress.springrecipes.sequence;  
  
import org.springframework.beans.factory.annotation.Autowired;  
  
public class SequenceGenerator {  
  
    @Autowired  
    private List<PrefixGenerator> prefixGenerators;  
    ...  
}
```

Lorsque Spring constate que l'annotation `@Autowired` est appliquée à un `java.util.Map` sécurisé dont les clés sont des chaînes de caractères, il ajoute à ce `Map` tous les beans de type compatible et utilise leur nom comme clé.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    private Map<String, PrefixGenerator> prefixGenerators;
    ...
}
```

Liaison automatique par type avec qualificateur

Par défaut, la liaison automatique par type ne fonctionne pas lorsque plusieurs beans de type compatible existent dans le conteneur IoC. Toutefois, Spring nous permet de désigner un bean candidat en fournissant son nom dans l'annotation `@Qualifier`.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class SequenceGenerator {

    @Autowired
    @Qualifier("datePrefixGenerator")
    private PrefixGenerator prefixGenerator;
    ...
}
```

Spring tente alors de trouver dans le conteneur IoC un bean ayant ce nom et le lie à la propriété.

```
<bean id="datePrefixGenerator"
      class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <property name="pattern" value="yyyyMMdd" />
</bean>
```

L'annotation `@Qualifier` s'applique également à un argument de méthode.

```
package com.apress.springrecipes.sequence;ces

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;

public class SequenceGenerator {
    ...
    @Autowired
    public void inject(
        @Qualifier("datePrefixGenerator") PrefixGenerator prefixGenerator) {
        this.prefixGenerator = prefixGenerator;
    }
}
```

Nous pouvons également créer notre propre type d'annotation de qualificateur qui sera utilisée dans la liaison automatique. Ce type d'annotation doit lui-même être marqué avec `@Qualifier`.

```
package com.apress.springrecipes.sequence;
...
import org.springframework.beans.factory.annotation.Qualifier;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.FIELD, ElementType.PARAMETER })
@Qualifier
public @interface Generator {
    String value();
}
```

Ensuite, nous pouvons appliquer cette annotation à une propriété de bean marquée par `@Autowired`. Elle demande à Spring de lier automatiquement le bean ayant cette annotation de qualificateur et la valeur indiquée.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;

public class SequenceGenerator {

    @Autowired
    @Generator("prefix")
    private PrefixGenerator prefixGenerator;
    ...
}
```

Nous devons donner ce qualificateur au bean cible qui doit être lié automatiquement à la propriété précédente. Le qualificateur est ajouté par l'élément `<qualifier>` avec l'attribut `type`. La valeur du qualificateur est précisée dans l'attribut `value`.

```
<bean id="datePrefixGenerator"
      class="com.apress.springrecipes.sequence.DatePrefixGenerator">
    <qualifier type="Generator" value="prefix" />
    <property name="pattern" value="yyyyMMdd" />
</bean>
```

Liaison automatique par nom

Si nous souhaitons lier automatiquement les propriétés d'un bean par le nom, nous pouvons marquer un mutateur, un constructeur ou un champ avec l'annotation `@Resource` de la JSR-250. Par défaut, Spring tente de trouver un bean dont le nom correspond à celui de la propriété. Mais nous pouvons préciser explicitement le nom du bean avec l'attribut `name`.

INFO

Pour utiliser les annotations de la JSR-250, vous devez inclure common-annotations.jar (situé dans le répertoire lib/j2ee de l'installation de Spring) dans votre chemin d'accès aux classes. Toutefois, si votre application s'exécute sous Java SE 6 ou Java EE 5, il est inutile d'inclure ce fichier JAR.

```
package com.apress.springrecipes.sequence;  
  
import javax.annotation.Resource;  
  
public class SequenceGenerator {  
  
    @Resource(name = "datePrefixGenerator")  
    private PrefixGenerator prefixGenerator;  
    ...  
}
```

3.9 Hériter de la configuration d'un bean

Problème

Lors de la configuration des beans dans le conteneur Spring IoC, il est possible que plusieurs beans partagent certains éléments de configuration, comme des propriétés et des attributs de bean dans l'élément <bean>. Ces éléments de configuration doivent être répétés pour de multiples beans.

Solution

Spring permet d'extraire les éléments de configuration communs de manière à constituer un *bean parent*. Les beans qui héritent de ce bean parent sont appelés *beans enfants*. Les beans enfants héritent des configurations définies dans le bean parent, y compris les propriétés et les attributs de bean de l'élément <bean>. Cela permet d'éviter la redondance des configurations. Ils peuvent également écraser les configurations héritées si nécessaire.

Le bean parent peut jouer à la fois le rôle de template de configuration et d'instance de bean. Pour qu'il serve uniquement de template, sans possibilité d'en obtenir une instance, nous devons affecter la valeur true à l'attribut abstract. Spring sait alors qu'il ne doit pas instancier ce bean.

Tous les attributs définis dans l'élément <bean> parent ne sont pas hérités, par exemple les attributs autowire et dependency-check. Pour connaître les attributs hérités du parent, consultez la rubrique de documentation de Spring qui concerne l'héritage de bean.

Explications

Supposons que nous ayons besoin d'ajouter une nouvelle instance du générateur séquentiel, avec des valeurs initial et suffix identiques à celles existantes.

```
<beans ...>
    <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>

    <bean id="sequenceGenerator1"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>

    <bean id="datePrefixGenerator"
        class="com.apress.springrecipes.sequence.DatePrefixGenerator">
        <property name="pattern" value="yyyyMMdd" />
    </bean>
</beans>
```

Pour éviter de dupliquer des propriétés, nous pouvons déclarer un bean de générateur séquentiel de base qui possède ces propriétés. Ensuite, il suffit que les deux générateurs séquentiels héritent de ce générateur de base pour que leurs propriétés correspondantes soient fixées automatiquement. Il est inutile de définir les attributs class des beans enfants s'ils sont identiques à celui du parent.

```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>

    <bean id="sequenceGenerator" parent="baseSequenceGenerator" />
    <bean id="sequenceGenerator1" parent="baseSequenceGenerator" />
    ...
</beans>
```

Les propriétés héritées peuvent être modifiées par les beans enfants. Nous pouvons ajouter un générateur séquentiel enfant avec une valeur initiale différente.

```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="initial" value="100000" />
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>
```

```
<bean id="sequenceGenerator2" parent="baseSequenceGenerator">
    <property name="initial" value="200000" />
</bean>
...
</beans>
```

Avec la configuration actuelle, nous pouvons obtenir une instance du bean du générateur séquentiel de base et l'utiliser. S'il doit servir uniquement de template, nous fixons l'attribut `abstract` à `true`. Spring ne crée alors plus d'instance de ce bean.

```
<bean id="baseSequenceGenerator" abstract="true"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
</bean>
```

Nous pouvons également omettre la classe du bean parent et laisser les beans enfants la préciser. Ce cas survient lorsque le bean parent et les beans enfants ne font pas partie de la même hiérarchie de classes mais partagent des propriétés de même nom. L'attribut `abstract` du bean parent doit alors être fixé à `true` car ce bean ne peut pas être instancié. Ajoutons, par exemple, une nouvelle classe `ReverseGenerator` qui possède également une propriété `initial`.

```
package com.apress.springrecipes.sequence;

public class ReverseGenerator {

    private int initial;

    public void setInitial(int initial) {
        this.initial = initial;
    }
}
```

`SequenceGenerator` et `ReverseGenerator` ne dérivent pas de la même classe de base. Elles ne font pas partie de la même hiérarchie de classes, mais elles possèdent une propriété de même nom : `initial`. Pour extraire cette propriété commune, nous avons besoin d'un bean parent, `baseGenerator`, dans lequel aucun attribut `class` n'est défini.

```
<beans ...>
    <bean id="baseGenerator" abstract="true">
        <property name="initial" value="100000" />
    </bean>

    <bean id="baseSequenceGenerator" abstract="true" parent="baseGenerator"
          class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="suffix" value="A" />
        <property name="prefixGenerator" ref="datePrefixGenerator" />
    </bean>

    <bean id="reverseGenerator" parent="baseGenerator"
          class="com.apress.springrecipes.sequence.ReverseGenerator" />
```

```

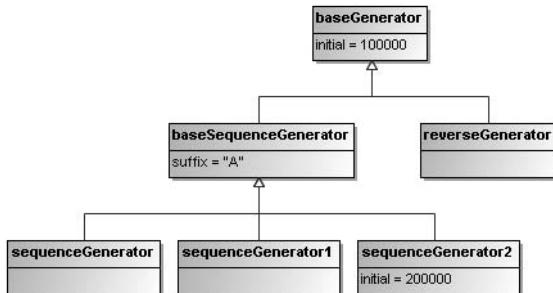
<bean id="sequenceGenerator" parent="baseSequenceGenerator" />
<bean id="sequenceGenerator1" parent="baseSequenceGenerator" />
<bean id="sequenceGenerator2" parent="baseSequenceGenerator">
    ...
</bean>
...
</beans>

```

La Figure 3.3 présente le graphe d'objets qui correspond à cette hiérarchie de beans générateurs.

Figure 3.3

Graphe d'objets pour la hiérarchie de beans générateurs.



3.10 Affecter des collections aux propriétés de bean

Problème

Les propriétés de bean sont parfois des collections qui contiennent de multiples éléments. Nous préférerions configurer ces propriétés de type collection à partir du fichier de configuration des beans plutôt que dans le code.

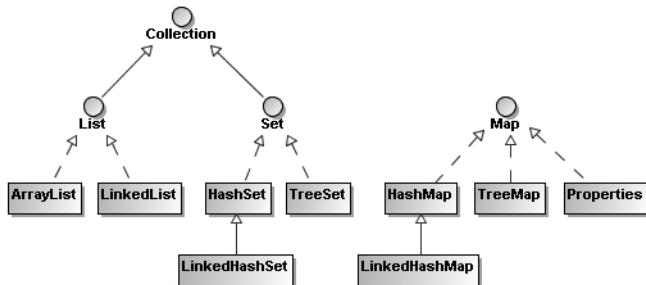
Solution

Le framework Java Collections définit un ensemble d'interfaces, d'implémentations et d'algorithmes pour différents types de collections, comme les listes (*list*), les ensembles (*set*) et les tables d'association (*map*). La Figure 3.4 montre un diagramme de classes UML simplifié qui peut faciliter la compréhension du framework Java Collections.

`List`, `Set` et `Map` sont les interfaces centrales qui représentent les trois principaux types de collections. Pour chaque type de collection, Java fournit plusieurs implémentations dont les fonctions et les caractéristiques diffèrent. Dans Spring, ces types de collections sont faciles à configurer grâce à un ensemble de balises XML intégrées, comme `<list>`, `<set>` et `<map>`.

Figure 3.4

Diagramme de classes simplifié pour le framework Java Collections.



Explications

Supposons que notre générateur séquentiel accepte plusieurs suffixes. Ils sont concaténés aux numéros de séquence en les séparant par des tirets. Nous devons accepter des suffixes ayant des types de données quelconques et les convertir en chaînes de caractères pour les ajouter aux numéros de séquence.

Listes, tableaux et ensembles

Tout d'abord, utilisons une collection `java.util.List` pour contenir nos suffixes. Une *liste* est une collection ordonnée et indexée dont les éléments sont accessibles par l'intermédiaire d'un indice ou d'une boucle `for-each`.

```

package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private List<Object> suffixes;
    public void setSuffixes(List<Object> suffixes) {
        this.suffixes = suffixes;
    }
    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        for (Object suffix : suffixes) {
            buffer.append("-");
            buffer.append(suffix);
        }
        return buffer.toString();
    }
}
  
```

Pour définir une propriété de type `java.util.List` dans le fichier de configuration des beans, nous employons une balise `<list>` qui contient les éléments de la liste. Ces éléments peuvent être des valeurs constantes simples (`<value>`), des références à des beans (`<ref>`), des définitions de beans internes (`<bean>`) ou des éléments null (`<null>`). Il est même possible d'inclure des collections dans une collection.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefixGenerator" ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffixes">
      <list>
        <value>A</value>
        <bean class="java.net.URL">
          <constructor-arg value="http" />
          <constructor-arg value="www.apress.com" />
          <constructor-arg value="/" />
        </bean>
        <null />
      </list>
    </property>
  </bean>
```

Conceptuellement, un *tableau* est très semblable à une liste en cela qu'il s'agit également d'une collection ordonnée et indexée dont les éléments sont accessibles par un indice. La principale différence vient de la taille figée du tableau, qui ne peut donc pas être étendu dynamiquement. Il est possible de convertir un tableau en liste et inversement avec les méthodes `Arrays.asList()` et `List.toArray()`. Pour notre générateur séquentiel, nous pouvons utiliser un tableau `Object[]` pour contenir les suffixes et accéder à ceux-ci par un indice ou une boucle `for-each`.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
  ...
  private Object[] suffixes;

  public void setSuffixes(Object[] suffixes) {
    this.suffixes = suffixes;
  }
  ...
}
```

Dans le fichier de configuration des beans, la définition d'un tableau est identique à celle d'une liste et se fonde sur la balise `<list>`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="suffixes">
    <list>
      <value>A</value>
      <bean class="java.net.URL">
        <constructor-arg value="http" />
        <constructor-arg value="www.apress.com" />
        <constructor-arg value="/" />
      </bean>
      <null />
    </list>
  </property>
</bean>
```

L'*ensemble* est un autre type de collection très répandu. La Figure 3.4 montre que `java.util.List` et `java.util.Set` étendent toutes deux l'interface `java.util.Collection`. Un ensemble diffère d'une liste en cela qu'il est ni ordonné ni indexé et qu'il ne peut contenir que des objets uniques. Autrement dit, il ne peut exister aucun élément en double dans un ensemble. Lorsque le même élément est ajouté une deuxième fois dans un ensemble, il remplace le premier. L'égalité de deux éléments est déterminée par la méthode `equals()`.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private Set<Object> suffixes;
    public void setSuffixes(Set<Object> suffixes) {
        this.suffixes = suffixes;
    }
    ...
}
```

Pour définir une propriété de type `java.util.Set`, nous utilisons la balise `<set>` dans laquelle les éléments sont ajoutés à la manière d'une liste.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <set>
            <value>A</value>
            <bean class="java.net.URL">
                <constructor-arg value="http" />
                <constructor-arg value="www.apress.com" />
                <constructor-arg value="/" />
            </bean>
            <null />
        </set>
    </property>
</bean>
```

Bien que le concept d'ordre n'existe pas dans les ensembles, Spring conserve l'ordre des éléments ajoutés en utilisant un `java.util.LinkedHashSet`. Cette implémentation de l'interface `java.util.Set` préserve l'ordre des éléments.

Tables d'association et propriétés

Une *table d'association* stocke ses entrées sous forme de couples clé-valeur. L'accès à une certaine valeur de la table se fait à l'aide de la clé correspondante. Nous pouvons également parcourir les entrées de la table avec une boucle `for-each`. Les clés et les valeurs d'une table d'association peuvent être de type quelconque. L'égalité des clés est déterminée par la méthode `equals()`. Par exemple, nous pouvons modifier notre générateur séquentiel pour qu'il accepte une collection `java.util.Map` qui contient des suffixes avec des clés.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private Map<Object, Object> suffixes;

    public void setSuffixes(Map<Object, Object> suffixes) {
        this.suffixes = suffixes;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        for (Map.Entry entry : suffixes.entrySet()) {
            buffer.append("-");
            buffer.append(entry.getKey());
            buffer.append("@");
            buffer.append(entry.getValue());
        }
        return buffer.toString();
    }
}
```

Dans Spring, une table d'association se définit à l'aide de la balise `<map>`, avec plusieurs balises `<entry>` pour le contenu. Chaque entrée est constituée d'une clé et d'une valeur. La clé est définie par la balise `<key>`. Puisque le type de la clé et celui de la valeur ne souffrent d'aucune restriction, nous pouvons employer des éléments `<value>`, `<ref>`, `<bean>` ou `<null>`. Spring préserve également l'ordre des entrées de la table en utilisant une collection `java.util.LinkedHashMap`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <map>
            <entry>
                <key>
                    <value>type</value>
                </key>
                <value>A</value>
            </entry>
            <entry>
                <key>
                    <value>url</value>
                </key>
                <bean class="java.net.URL">
                    <constructor-arg value="http" />
                    <constructor-arg value="www.apress.com" />
                    <constructor-arg value="/" />
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

Des raccourcis permettent de définir des clés et des valeurs sous forme d'attributs de la balise <entry>. S'il s'agit de valeurs constantes simples, nous pouvons les définir avec key et value. S'il s'agit de références de beans, nous les définissons avec key-ref et value-ref.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
      <map>
        <entry key="type" value="A" />
        <entry key="url">
          <bean class="java.net.URL">
            <constructor-arg value="http" />
            <constructor-arg value="www.apress.com" />
            <constructor-arg value="/" />
          </bean>
        </entry>
      </map>
    </property>
  </bean>
```

Une collection `java.util.Properties` ressemble fortement à une table d'association. Elle implémente également l'interface `java.util.Map` et stocke les entrées sous forme de couples clé-valeur. Toutefois, les clés et les valeurs d'une collection `Properties` sont exclusivement des chaînes de caractères.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
  ...
  private Properties suffixes;

  public void setSuffixes(Properties suffixes) {
    this.suffixes = suffixes;
  }
  ...
}
```

Pour définir une collection `java.util.Properties` dans Spring, nous employons la balise <props> avec plusieurs balises <prop> pour représenter les entrées. Chaque balise <prop> doit définir un attribut key et inclure la valeur associée.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
      <props>
        <prop key="type">A</prop>
        <prop key="url">http://www.apress.com/</prop>
      </props>
    </property>
  </bean>
```

Fusionner avec la collection du bean parent

Lorsque les beans sont définis par héritage, la collection d'un bean enfant peut fusionner avec celle de son parent si l'attribut `merge` est fixé à `true`. Pour une collection `<list>`, les éléments enfants sont ajoutés après ceux du parent de manière à conserver l'ordre. Le générateur séquentiel suivant a donc quatre suffixes : A, B, A et C.

```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="prefixGenerator" ref="datePrefixGenerator" />
        <property name="initial" value="100000" />
        <property name="suffixes">
            <list>
                <value>A</value>
                <value>B</value>
            </list>
        </property>
    </bean>

    <bean id="sequenceGenerator" parent="baseSequenceGenerator">
        <property name="suffixes">
            <list merge="true">
                <value>A</value>
                <value>C</value>
            </list>
        </property>
    </bean>
    ...
</beans>
```

Pour une collection `<set>` ou `<map>`, les éléments enfants remplacent ceux du parent s'ils ont la même valeur. Par conséquent, le générateur séquentiel suivant a trois suffixes : A, B et C.

```
<beans ...>
    <bean id="baseSequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
        <property name="prefixGenerator" ref="datePrefixGenerator" />
        <property name="initial" value="100000" />
        <property name="suffixes">
            <set>
                <value>A</value>
                <value>B</value>
            </set>
        </property>
    </bean>

    <bean id="sequenceGenerator" parent="baseSequenceGenerator">
        <property name="suffixes">
            <set merge="true">
                <value>A</value>
                <value>C</value>
            </set>
        </property>
    </bean>
```

```
</property>
</bean>
...
</beans>
```

3.11 Préciser le type de données des éléments d'une collection

Problème

Par défaut, Spring considère chaque élément d'une collection comme une chaîne de caractères. Nous devons préciser le type de données des éléments de notre collection si nous ne les utilisons pas comme des chaînes.

Solution

Nous pouvons indiquer le type de données de chaque élément d'une collection à l'aide de l'attribut `type` de la balise `<value>` ou celui de l'ensemble des éléments à l'aide de l'attribut `value-type` de la balise de la collection. Avec Java 1.5 ou une version ultérieure, nous pouvons définir une collection typée afin que Spring lise les informations de type de la collection.

Explications

Supposons à présent que nous acceptons une liste de nombres entiers comme suffixes pour notre générateur séquentiel. Chaque nombre est mis en forme sur quatre chiffres par une instance de `java.text.DecimalFormat`.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
    ...
    private List<Object> suffixes;

    public void setSuffixes(List<Object> suffixes) {
        this.suffixes = suffixes;
    }

    public synchronized String getSequence() {
        StringBuffer buffer = new StringBuffer();
        ...
        DecimalFormat formatter = new DecimalFormat("0000");
        for (Object suffix : suffixes) {
            buffer.append("-");
            buffer.append(formatter.format((Integer) suffix));
        }
        return buffer.toString();
    }
}
```

Nous définissons ensuite plusieurs suffixes dans le fichier de configuration des beans.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefixGenerator" ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffixes">
      <list>
        <value>5</value>
        <value>10</value>
        <value>20</value>
      </list>
    </property>
  </bean>
```

Toutefois, lors de l'exécution de cette application, nous recevons une exception `ClassCastException` qui indique que les suffixes ne peuvent pas être convertis en entiers car ils sont de type `String`. Par défaut, Spring considère chaque élément d'une collection comme une chaîne de caractères. Nous devons donc définir l'attribut `type` de la balise `<value>` pour préciser le type d'un élément.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="suffixes">
    <list>
      <value type="int">5</value>
      <value type="int">10</value>
      <value type="int">20</value>
    </list>
  </property>
</bean>
```

Ou bien, en utilisant l'attribut `value-type` de la balise de la collection, nous indiquons le type de tous ses éléments.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
  ...
  <property name="suffixes">
    <list value-type="int">
      <value>5</value>
      <value>10</value>
      <value>20</value>
    </list>
  </property>
</bean>
```

Dans Java 1.5 ou une version ultérieure, nous pouvons définir notre liste `suffixes` en utilisant une collection typée qui stocke des entiers.

```
package com.apress.springrecipes.sequence;
...
public class SequenceGenerator {
  ...
  private List<Integer> suffixes;
```

```
public void setSuffixes(List<Integer> suffixes) {
    this.suffixes = suffixes;
}

public synchronized String getSequence() {
    StringBuffer buffer = new StringBuffer();
    ...
    DecimalFormat formatter = new DecimalFormat("0000");
    for (int suffix : suffixes) {
        buffer.append("-");
        buffer.append(formatter.format(suffix));
    }
    return buffer.toString();
}
}
```

En ayant défini notre collection ainsi, Spring est capable de lire les informations de type de cette collection grâce au mécanisme de réflexion. Il n'est alors plus nécessaire de préciser l'attribut `value-type` de `<list>`.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
        <list>
            <value>5</value>
            <value>10</value>
            <value>20</value>
        </list>
    </property>
</bean>
```

3.12 Définir des collections avec des beans de fabrique et le schéma `util`

Problème

Lorsqu'on définit des collections à l'aide des balises des collections de base, il est impossible de préciser la classe concrète d'une collection, comme `LinkedList`, `TreeSet` ou `TreeMap`. Par ailleurs, nous ne pouvons pas partager une collection entre plusieurs beans en la définissant comme un bean autonome auquel d'autres beans font référence.

Solution

Spring propose deux solutions pour dépasser les limitations des balises des collections de base. La première consiste à utiliser les beans de fabrique des collections, comme `ListFactoryBean`, `SetFactoryBean` et `MapFactoryBean`. Un *bean de fabrique* est une sorte de bean Spring particulière servant à créer un autre bean. La seconde solution consiste à utiliser les balises des collections, comme `<util:list>`, `<util:set>` et `<util:map>`, définies dans le schéma `util` apporté par Spring 2.x.

Explications

Préciser la classe concrète des collections

Nous pouvons utiliser un bean de fabrique pour définir une collection et préciser sa classe. Par exemple, spécifions la propriété targetSetClass de SetFactoryBean. Spring instancie ensuite la classe indiquée pour cette collection.

```
<bean id="sequenceGenerator"
      class="com.apress.springrecipes.sequence.SequenceGenerator">
    <property name="prefixGenerator" ref="datePrefixGenerator" />
    <property name="initial" value="100000" />
    <property name="suffixes">
      <bean class="org.springframework.beans.factory.config.SetFactoryBean">
        <property name="targetSetClass">
          <value>java.util.TreeSet</value>
        </property>
        <property name="sourceSet">
          <set>
            <value>5</value>
            <value>10</value>
            <value>20</value>
          </set>
        </property>
      </bean>
    </property>
  </bean>
```

Nous pouvons également employer une balise de collection du schéma util pour définir une collection et fixer sa classe (par exemple avec l'attribut set-class de <util:set>). Il ne faut cependant pas oublier d'ajouter la définition du schéma dans l'élément racine <beans>.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-2.5.xsd">

  <bean id="sequenceGenerator"
        class="com.apress.springrecipes.sequence.SequenceGenerator">
    ...
    <property name="suffixes">
      <util:set set-class="java.util.TreeSet">
        <value>5</value>
        <value>10</value>
        <value>20</value>
      </util:set>
    </property>
  </bean>
  ...
</beans>
```

Définir des collections autonomes

Les beans de fabrique des collections présentent également un autre avantage. Nous pouvons définir une collection sous forme d'un bean autonome auquel d'autres beans feront référence. Par exemple, définissons un ensemble autonome avec SetFactoryBean.

```
<beans ...>
    <bean id="sequenceGenerator"
          class="com.apress.springrecipes.sequence.SequenceGenerator">
        ...
        <property name="suffixes">
            <ref local="suffixes" />
        </property>
    </bean>

    <bean id="suffixes"
          class="org.springframework.beans.factory.config.SetFactoryBean">
        <property name="sourceSet">
            <set>
                <value>5</value>
                <value>10</value>
                <value>20</value>
            </set>
        </property>
    </bean>
    ...
</beans>
```

La balise `<util:set>` du schéma util nous permet également de définir un ensemble autonome.

```
<beans ...>
    <bean id="sequenceGenerator"
          class="com.apress.springrecipes.sequence.SequenceGenerator">
        ...
        <property name="suffixes">
            <ref local="suffixes" />
        </property>
    </bean>

    <util:set id="suffixes">
        <value>5</value>
        <value>10</value>
        <value>20</value>
    </util:set>
    ...
</beans>
```

3.13 Rechercher les composants dans le chemin d'accès aux classes

Problème

Pour que le conteneur Spring IoC prenne en charge nos composants, nous les déclarons un par un dans le fichier de configuration des beans. Nous pourrions travailler plus rapidement si Spring détectait automatiquement nos composants sans passer par une configuration manuelle.

Solution

Spring 2.5 propose une fonctionnalité puissante appelée *scan de composants (component scanning)*. Elle peut rechercher, détecter et instancier automatiquement les composants marqués par des annotations stéréotypes et situés dans le chemin d'accès aux classes. `@Component` correspond à l'annotation de base qui signale un composant géré par Spring. `@Repository`, `@Service` et `@Controller` font partie des autres annotations stéréotypes. Elles dénotent, respectivement, des composants situés dans les couches de persistance, de service et de présentation.

Explications

Supposons qu'on nous demande de développer notre application de générateur séquentiel en utilisant des séquences provenant d'une base de données et d'enregistrer le préfixe et le suffixe de chaque séquence dans une table. Tout d'abord, créons la classe de domaine `Sequence` qui contient les propriétés `id`, `prefix` et `suffix`.

```
package com.apress.springrecipes.sequence;

public class Sequence {

    private String id;
    private String prefix;
    private String suffix;

    // Constructeurs, accesseurs et mutateurs.
    ...
}
```

Ensuite, créons une interface pour l'objet d'accès aux données (DAO, *Data Access Object*) qui prend en charge les accès à la base de données. La méthode `getSequence()` charge à partir de la table un objet `Sequence` correspondant à l'identifiant indiqué, tandis que la méthode `getNextValue()` obtient la valeur suivante pour une certaine séquence de la base de données.

```
package com.apress.springrecipes.sequence;

public interface SequenceDao {

    public Sequence getSequence(String sequenceId);
    public int getNextValue(String sequenceId);
}
```

Dans une application réelle, nous implémenterions cette interface DAO avec une technologie d'accès aux données, comme JDBC ou une correspondance objet-relationnel. Toutefois, pour nos tests, nous utilisons des tables d'association qui stockent les instances des séquences et les valeurs.

```
package com.apress.springrecipes.sequence;
...
public class SequenceDaoImpl implements SequenceDao {

    private Map<String, Sequence> sequences;
    private Map<String, Integer> values;

    public SequenceDaoImpl() {
        sequences = new HashMap<String, Sequence>();
        sequences.put("IT", new Sequence("IT", "30", "A"));
        values = new HashMap<String, Integer>();
        values.put("IT", 100000);
    }

    public Sequence getSequence(String sequenceId) {
        return sequences.get(sequenceId);
    }

    public synchronized int getNextValue(String sequenceId) {
        int value = values.get(sequenceId);
        values.put(sequenceId, value + 1);
        return value;
    }
}
```

Nous avons également besoin d'un objet, jouant le rôle de façade, pour offrir le service de génération des séquences. En interne, cet objet de service coopère avec le DAO pour traiter les demandes de génération des séquences. Il a donc besoin d'une référence au DAO.

```
package com.apress.springrecipes.sequence;

public class SequenceService {

    private SequenceDao sequenceDao;

    public void setSequenceDao(SequenceDao sequenceDao) {
        this.sequenceDao = sequenceDao;
    }
}
```

```
    public String generate(String sequenceId) {
        Sequence sequence = sequenceDao.getSequence(sequenceId);
        int value = sequenceDao.getNextValue(sequenceId);
        return sequence.getPrefix() + value + sequence.getSuffix();
    }
}
```

Enfin, nous devons configurer ces composants dans le fichier de configuration des beans afin que notre application fonctionne. Grâce à la liaison automatique de nos composants, nous réduisons la quantité des informations de configuration.

```
<beans ...>
    <bean id="sequenceService"
          class="com.apress.springrecipes.sequence.SequenceService"
          autowire="byType" />

    <bean id="sequenceDao"
          class="com.apress.springrecipes.sequence.SequenceDaoImpl" />
</beans>
```

Nous pouvons à présent tester ces composants à l'aide de la classe Main suivante :

```
package com.apress.springrecipes.sequence;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        SequenceService sequenceService =
            (SequenceService) context.getBean("sequenceService");

        System.out.println(sequenceService.generate("IT"));
        System.out.println(sequenceService.generate("IT"));
    }
}
```

Rechercher automatiquement des composants

Le scan de composants fourni par Spring 2.5 recherche, détecte et instancie automatiquement certains composants qui se trouvent dans le chemin d'accès aux classes. Par défaut, Spring détecte tous les composants marqués d'une annotation stéréotype. @Component correspond à l'annotation de base qui signale un composant géré par Spring. Nous pouvons l'appliquer à notre classe SequenceDaoImpl.

```
package com.apress.springrecipes.sequence;

import org.springframework.stereotype.Component;
```

```
@Component
public class SequenceDaoImpl implements SequenceDao {
    ...
}
```

Nous l'appliquons également à la classe SequenceService pour qu'elle soit détectée par Spring. Par ailleurs, en marquant le champ DAO avec l'annotation @Autowired, nous permettons à Spring de le lier automatiquement par le type.

```
package com.apress.springrecipes.sequence;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Component;

@Component
public class SequenceService {

    @Autowired
    private SequenceDao sequenceDao;
    ...
}
```

Après avoir appliqué des annotations stéréotypes aux classes de nos composants, nous pouvons demander à Spring de les rechercher en déclarant un seul élément XML, <context:component-scan>. Dans cet élément, nous précisons le paquetage où seront recherchés nos composants. Ce paquetage et tous ses sous-paquetages sont examinés. Pour indiquer plusieurs paquetages, il suffit de les séparer par des virgules.

Cet élément enregistre également une instance de AutowiredAnnotationBeanPostProcessor capable de lier automatiquement les propriétés marquées par l'annotation @Autowired.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:component-scan base-package="com.apress.springrecipes.sequence" />
</beans>
```

@Component est l'annotation stéréotype de base pour signaler des composants d'usage général. Il existe d'autres stéréotypes spécifiques pour indiquer des composants de couches différentes. Tout d'abord, l'annotation @Repository désigne un composant DAO dans la couche de persistance.

```
package com.apress.springrecipes.sequence;

import org.springframework.stereotype.Repository;
```

```
@Repository  
public class SequenceDaoImpl implements SequenceDao {  
    ...  
}
```

Quant à @Service, elle dénote un composant de la couche de service.

```
package com.apress.springrecipes.sequence;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.stereotype.Service;  
  
@Service  
public class SequenceService {  
  
    @Autowired  
    private SequenceDao sequenceDao;  
    ...  
}
```

Le stéréotype de composant @Controller désigne un composant contrôleur dans la couche de présentation. Nous y reviendrons au Chapitre 10.

Filtrer les composants à rechercher

Par défaut, Spring détecte toutes les classes annotées par @Component, @Repository, @Service, @Controller ou tout type d'annotation personnalisé lui-même marqué par @Component. Nous pouvons personnaliser la recherche en appliquant un ou plusieurs filtres d'inclusion ou d'exclusion.

Spring reconnaît quatre types d'expressions de filtrage. Les types annotation et assignable permettent de définir un filtre fondé sur un type d'annotation et une classe ou une interface. Les types regex et aspectj permettent de préciser une expression régulière et une expression de point d'action (*pointcut*) AspectJ pour la correspondance des classes.

Par exemple, le scan de composants suivant inclut toutes les classes dont le nom contient les mots Dao ou Service et exclut celles annotées avec @Controller :

```
<beans ...>  
    <context:component-scan base-package="com.apress.springrecipes.sequence">  
        <context:include-filter type="regex"  
            expression="com\.\apress\.\springrecipes\.\sequence\..*Dao.*" />  
        <context:include-filter type="regex"  
            expression="com\.\apress\.\springrecipes\.\sequence\..*Service.*" />  
        <context:exclude-filter type="annotation"  
            expression="org.springframework.stereotype.Controller" />  
    </context:component-scan>  
</beans>
```

Puisque ces filtres détectent toutes les classes dont le nom contient les mots Dao ou Service, les composants SequenceDaoImpl et SequenceService sont détectés automatiquement, même sans annotation stéréotype.

Nommer les composants détectés

Par défaut, Spring nomme les composants détectés en mettant en minuscule le premier caractère du seul nom de sa classe. Par exemple, un composant de classe SequenceService est nommé sequenceService. Nous pouvons définir explicitement le nom d'un composant en le précisant dans la valeur de l'annotation stéréotype.

```
package com.apress.springrecipes.sequence;
...
import org.springframework.stereotype.Service;

@Service("sequenceService")
public class SequenceService {
    ...
}

---

package com.apress.springrecipes.sequence;
import org.springframework.stereotype.Repository;

@Repository("sequenceDao")
public class SequenceDaoImpl implements SequenceDao {
    ...
}
```

Nous pouvons développer notre propre stratégie de nommage en implémentant l'interface BeanNameGenerator et en l'indiquant dans l'attribut name-generator de l'élément <context:component-scan>.

3.14 En résumé

Dans ce chapitre, nous avons étudié les bases de la configuration des beans dans le conteneur Spring IoC. Spring prend en charge plusieurs types de configurations. La configuration XML est la plus simple et la plus mûre. Spring propose deux implémentations du conteneur IoC. L'implémentation de base est la fabrique de beans, la plus élaborée est le contexte d'application. Il est préférable d'utiliser le contexte d'application, à moins que les ressources soient limitées. Pour définir des propriétés de bean, qui peuvent être de simples valeurs, des collections ou des références de beans, Spring propose l'injection par mutateur et l'injection par constructeur.

La vérification des dépendances et la liaison automatique sont deux fonctionnalités du conteneur fourni par Spring. Grâce à la vérification des dépendances, nous pouvons contrôler si toutes les propriétés requises sont fixées. Grâce à la liaison automatique, nos beans peuvent être liés automatiquement, en fonction du type, du nom ou de l'annotation. L'ancien style de configuration de ces deux fonctionnalités passe par des attributs XML. La nouvelle forme se fonde sur des annotations et des postprocesseurs de beans qui apportent une plus grande souplesse.

Spring prend en charge l'héritage de bean en déplaçant les configurations communes dans un bean parent. Celui-ci peut servir de template de configuration, d'instance de bean ou les deux à la fois.

Puisque les collections sont des éléments de programmation essentiels en Java, Spring fournit différentes balises pour que nous puissions configurer facilement des collections dans le fichier de configuration des beans. Nous pouvons employer les beans de fabrique de collections ou les balises des collections du schéma `util` pour préciser les détails d'une collection et pour définir des collections sous forme de beans autonomes partagés par plusieurs beans.

Enfin, Spring peut détecter automatiquement nos composants situés dans le chemin d'accès aux classes. Par défaut, il détecte tous les composants ayant des annotations stéréotypes particulières. Les filtres permettent d'inclure ou d'exclure certains composants. Le scan de composants est une fonctionnalité puissante qui réduit la quantité d'informations de configuration.

Le chapitre suivant présente les fonctionnalités élaborées du conteneur Spring IoC que nous n'avons pas étudiées dans ce chapitre.

Fonctions élaborées du conteneur Spring IoC

Au sommaire de ce chapitre

- ✓ Créer des beans en invoquant un constructeur
- ✓ Créer des beans en invoquant une méthode statique de fabrique
- ✓ Créer des beans en invoquant une méthode d'instance de fabrique
- ✓ Créer des beans en utilisant un bean de fabrique Spring
- ✓ Déclarer des beans correspondant à des champs statiques
- ✓ Déclarer des beans correspondant aux propriétés d'un objet
- ✓ Fixer les portées de bean
- ✓ Modifier l'initialisation et la destruction d'un bean
- ✓ Rendre les beans conscients de l'existence du conteneur
- ✓ Créer des postprocesseurs de beans
- ✓ Externaliser les configurations de beans
- ✓ Obtenir des messages textuels multilingues
- ✓ Communiquer à l'aide des événements d'application
- ✓ Enregistrer des éditeurs de propriétés dans Spring
- ✓ Créer des éditeurs de propriétés
- ✓ Charger des ressources externes
- ✓ En résumé

Ce chapitre présente les fonctionnalités élaborées et les mécanismes internes du conteneur Spring IoC qui permettent d'être plus efficace lors du développement des applications Spring. Bien que ces fonctionnalités ne soient pas employées très souvent, elles sont indispensables dans un conteneur complet et puissant et servent de fondation aux autres modules du framework Spring.

La conception du conteneur Spring IoC lui permet d'être étendu et personnalisé facilement. Il est possible de configurer son fonctionnement par défaut et d'étendre ses fonctionnalités en enregistrant des plug-in qui respectent ses spécifications.

À la fin de ce chapitre, la plupart des fonctionnalités du conteneur Spring IoC vous seront devenues familières. Vous disposerez ainsi des bases nécessaires pour aborder les différents thèmes Spring traités dans les chapitres ultérieurs.

4.1 Créez des beans en invoquant un constructeur

Problème

Nous souhaitons créer un bean dans le conteneur Spring IoC en invoquant son constructeur. Cette méthode, la plus commune et la plus directe pour créer des beans, équivaut à utiliser l'opérateur new pour créer des objets Java.

Solution

Lorsque nous définissons l'attribut class d'un bean, nous demandons au conteneur Spring IoC de créer une instance de ce bean en invoquant son constructeur.

Explications

Supposons que nous développons une application marchande pour vendre des produits en ligne. Nous commençons par créer la classe Product, avec ses propriétés comme le nom et le prix du produit. Puisque notre magasin propose plusieurs produits, la classe Product est abstraite et les différentes sous-classes concrètes des produits en dérivent.

```
package com.apress.springrecipes.shop;

public abstract class Product {

    private String name;
    private double price;

    public Product() {}

    public Product(String name, double price) {
        this.name = name;
        this.price = price;
    }

    // Accesseurs et mutateurs.
    ...

    public String toString() {
        return name + " " + price;
    }
}
```

Nous créons ensuite deux sous-classes pour des produits, Battery et Disc, chacune ajoutant ses propres propriétés.

```
package com.apress.springrecipes.shop;

public class Battery extends Product {

    private boolean rechargeable;

    public Battery() {
        super();
    }

    public Battery(String name, double price) {
        super(name, price);
    }

    // Accesseurs et mutateurs.
    ...
}

---

package com.apress.springrecipes.shop;

public class Disc extends Product {

    private int capacity;

    public Disc() {
        super();
    }

    public Disc(String name, double price) {
        super(name, price);
    }

    // Accesseurs et mutateurs.
    ...
}
```

Le fichier de configuration des beans suivant définit quelques produits dans le conteneur Spring IoC :

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="aaa" class="com.apress.springrecipes.shop.Battery">
        <property name="name" value="AAA" />
        <property name="price" value="2.5" />
        <property name="rechargeable" value="true" />
    </bean>
```

```
<bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
    <property name="name" value="CD-RW" />
    <property name="price" value="1.5" />
    <property name="capacity" value="700" />
</bean>
</beans>
```

Lorsque aucun élément `<constructor-arg>` n'est précisé, le constructeur par défaut sans argument est invoqué. Ensuite, pour chaque élément `<property>`, Spring injecte la valeur à l'aide du mutateur correspondant. La configuration précédente équivaut au code suivant :

```
Product aaa = new Battery();
aaa.setName("AAA");
aaa.setPrice(2.5);
aaa.setRechargeable(true);

Product cdrw = new Disc();
cdrw.setName("CD-RW");
cdrw.setPrice(1.5);
cdrw.setCapacity(700);
```

Sinon, lorsqu'un ou plusieurs éléments `<constructor-arg>` sont définis, Spring choisit le meilleur constructeur qui correspond aux arguments et l'invoque.

```
<beans ...>
    <bean id="aaa" class="com.apress.springrecipes.shop.Battery">
        <constructor-arg value="AAA" />
        <constructor-arg value="2.5" />
        <property name="rechargeable" value="true" />
    </bean>

    <bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
        <constructor-arg value="CD-RW" />
        <constructor-arg value="1.5" />
        <property name="capacity" value="700" />
    </bean>
</beans>
```

Puisqu'il n'existe aucune ambiguïté sur le constructeur de la classe `Product` et des sous-classes, la configuration suivante équivaut au code suivant :

```
Product aaa = new Battery("AAA", 2.5);
aaa.setRechargeable(true);

Product cdrw = new Disc("CD-RW", 1.5);
cdrw.setCapacity(700);
```

Nous écrivons une classe `Main` pour tester nos produits en les prenant dans le conteneur Spring IoC.

```
package com.apress.springrecipes.shop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
```

```
public class Main {  
  
    public static void main(String[] args) throws Exception {  
        ApplicationContext context =  
            new ClassPathXmlApplicationContext("beans.xml");  
  
        Product aaa = (Product) context.getBean("aaa");  
        Product cdrw = (Product) context.getBean("cdrw");  
        System.out.println(aaa);  
        System.out.println(cdrw);  
    }  
}
```

4.2 Créer des beans en invoquant une méthode statique de fabrique

Problème

Nous souhaitons créer un bean dans le conteneur Spring IoC en invoquant une *méthode statique de fabrique*, dont le rôle est d'encapsuler la procédure de création d'un objet. Le client qui demande un objet appelle simplement cette méthode sans avoir besoin de connaître les détails de la création.

Solution

Spring est capable de créer un bean en invoquant la méthode statique de fabrique précisée dans l'attribut `factory-method`.

Explications

Nous pouvons par exemple écrire la méthode statique de fabrique `createProduct()` pour créer un produit à partir d'un identifiant de produit. En fonction de cet identifiant, la méthode choisit la classe concrète à instancier. Si aucun produit ne correspond à l'identifiant, elle lance une exception `IllegalArgumentException`.

```
package com.apress.springrecipes.shop;  
  
public class ProductCreator {  
  
    public static Product createProduct(String productId) {  
        if ("aaa".equals(productId)) {  
            return new Battery("AAA", 2.5);  
        } else if ("cdrw".equals(productId)) {  
            return new Disc("CD-RW", 1.5);  
        }  
        throw new IllegalArgumentException("Produit inconnu");  
    }  
}
```

Pour déclarer un bean créé à l'aide d'une méthode statique de fabrique, nous indiquons la classe qui contient la méthode dans l'attribut `class` et le nom de la méthode dans

l'attribut `factory-method`. Les arguments de la méthode sont passés en utilisant des éléments `<constructor-arg>`.

```
<beans ...>
    <bean id="aaa" class="com.apress.springrecipes.shop.ProductCreator"
          factory-method="createProduct">
        <constructor-arg value="aaa" />
    </bean>

    <bean id="cdrw" class="com.apress.springrecipes.shop.ProductCreator"
          factory-method="createProduct">
        <constructor-arg value="cdrw" />
    </bean>
</beans>
```

Si la méthode de fabrique lance une exception, Spring l'enveloppe dans une exception `BeanCreationException`. La configuration précédente équivaut au code suivant :

```
Product aaa = ProductCreator.createProduct("aaa");
Product cdrw = ProductCreator.createProduct("cdrw");
```

4.3 Créer des beans en invoquant une méthode d'instance de fabrique

Problème

Nous souhaitons créer un bean dans le conteneur Spring IoC en invoquant une *méthode d'instance de fabrique*, dont le rôle est d'encapsuler la procédure de création d'un objet. Le client qui demande un objet appelle simplement cette méthode sans avoir besoin de connaître les détails de la création.

Solution

Spring est capable de créer un bean en invoquant une méthode d'instance de fabrique. L'instance du bean de fabrique doit être indiquée dans l'attribut `factory-bean`, tandis que la méthode de fabrique est précisée par l'attribut `factory-method`.

Explications

Nous pouvons, par exemple, écrire la classe `ProductCreator` suivante en utilisant une table d'association pour contenir les produits existants. La méthode d'instance de fabrique `createProduct()` obtient un produit en recherchant dans la table d'association l'identifiant `productId` fourni. Si aucun produit ne correspond à cet identifiant, elle lance une exception `IllegalArgumentException`.

```
package com.apress.springrecipes.shop;
...
public class ProductCreator {
```

```
private Map<String, Product> products;

public void setProducts(Map<String, Product> products) {
    this.products = products;
}

public Product createProduct(String productId) {
    Product product = products.get(productId);
    if (product != null) {
        return product;
    }
    throw new IllegalArgumentException("Produit inconnu");
}
}
```

Pour créer des produits à partir de cette classe `ProductCreator`, nous devons tout d'abord en déclarer une instance dans le conteneur IoC et configurer sa table d'association des produits. Nous déclarons les produits dans la table en utilisant des beans internes. Pour déclarer un bean créé par une méthode d'instance de fabrique, nous indiquons dans l'attribut `factory-bean` le bean qui contient la méthode de fabrique et le nom de cette méthode dans l'attribut `factory-method`. Pour finir, nous définissons les arguments de la méthode à l'aide d'éléments `<constructor-arg>`.

```
<beans ...>
    <bean id="productCreator"
          class="com.apress.springrecipes.shop.ProductCreator">
        <property name="products">
            <map>
                <entry key="aaa">
                    <bean class="com.apress.springrecipes.shop.Battery">
                        <property name="name" value="AAA" />
                        <property name="price" value="2.5" />
                    </bean>
                </entry>
                <entry key="cdrw">
                    <bean class="com.apress.springrecipes.shop.Disc">
                        <property name="name" value="CD-RW" />
                        <property name="price" value="1.5" />
                    </bean>
                </entry>
            </map>
        </property>
    </bean>

    <bean id="aaa" factory-bean="productCreator"
          factory-method="createProduct">
        <constructor-arg value="aaa" />
    </bean>

    <bean id="cdrw" factory-bean="productCreator"
          factory-method="createProduct">
        <constructor-arg value="cdrw" />
    </bean>
</beans>
```

Si la méthode de fabrique lance une exception, Spring l'enveloppe dans une exception `BeanCreationException`. La configuration précédente équivaut au code suivant :

```
ProductCreator productCreator = new ProductCreator();
productCreator.setProducts(...);

Product aaa = productCreator.createProduct("aaa");
Product cdrw = productCreator.createProduct("cdrw");
```

4.4 Créer des beans en utilisant un bean de fabrique Spring

Problème

Nous voulons créer un bean dans le conteneur Spring IoC en utilisant un bean de fabrique Spring. Un *bean de fabrique* est un bean qui joue le rôle de fabrique afin de créer d'autres beans dans le conteneur IoC. Conceptuellement, un bean de fabrique équivaut à une méthode de fabrique, mais il s'agit d'un bean Spring spécial qui peut être identifié par le conteneur Spring IoC pendant la phase de construction des beans.

Solution

Un bean de fabrique doit implémenter l'interface `FactoryBean`. Pour nous faciliter la tâche, Spring fournit une classe abstraite, `AbstractFactoryBean`, qui sert de classe de base. Les beans de fabrique sont le plus souvent utilisés pour implémenter des fonctions du framework. En voici quelques exemples :

- Pour la recherche d'un objet, comme une source de données, à partir de JNDI, nous pouvons employer `JndiObjectFactoryBean`.
- Pour créer le proxy d'un bean avec Spring AOP classique, nous pouvons employer `ProxyFactoryBean`.
- Pour créer une fabrique de session Hibernate dans le conteneur IoC, nous pouvons employer `LocalSessionFactoryBean`.

Toutefois, en tant qu'utilisateurs du framework, nous avons rarement à écrire nos propres beans de fabrique car ils sont propres au framework et ne peuvent pas être utilisés en dehors du conteneur Spring IoC. En réalité, nous pouvons toujours implémenter une méthode de fabrique pour remplacer un bean de fabrique.

Explications

Puisqu'il est bon de comprendre les mécanismes internes aux beans de fabrique, nous allons écrire un bean de fabrique qui crée un produit et lui applique une remise. Il contient une propriété `product` et une propriété `discount` pour appliquer la remise au produit, et retourne celui-ci sous forme d'un nouveau bean.

```
package com.apress.springrecipes.shop;

import org.springframework.beans.factory.config.AbstractFactoryBean;

public class DiscountFactoryBean extends AbstractFactoryBean {

    private Product product;
    private double discount;

    public void setProduct(Product product) {
        this.product = product;
    }

    public void setDiscount(double discount) {
        this.discount = discount;
    }

    public Class getObjectType() {
        return product.getClass();
    }

    protected Object createInstance() throws Exception {
        product.setPrice(product.getPrice() * (1 - discount));
        return product;
    }
}
```

Notre bean de fabrique étend la classe `AbstractFactoryBean` et redéfinit simplement la méthode `createInstance()` pour créer une instance du bean cible. Par ailleurs, sa méthode `getObjectType()` doit retourner le type du bean cible car elle sert au bon fonctionnement de la liaison automatique.

Nous déclarons ensuite nos instances de produits avec `DiscountFactoryBean`. Chaque fois que nous demandons un bean qui implémente l'interface `FactoryBean`, le conteneur Spring IoC utilise notre bean de fabrique pour créer le bean demandé et nous le retourner. Pour obtenir une instance du bean de fabrique lui-même, il suffit de faire précéder le nom de bean par le caractère &.

```
<beans ...>
    <bean id="aaa"
          class="com.apress.springrecipes.shop.DiscountFactoryBean">
        <property name="product">
            <bean class="com.apress.springrecipes.shop.Battery">
                <constructor-arg value="AAA" />
                <constructor-arg value="2.5" />
            </bean>
        </property>
        <property name="discount" value="0.2" />
    </bean>

    <bean id="cdrw"
          class="com.apress.springrecipes.shop.DiscountFactoryBean">
        <property name="product">
            <bean class="com.apress.springrecipes.shop.Disc">
```

```
<constructor-arg value="CD-RW" />
<constructor-arg value="1.5" />
</bean>
</property>
<property name="discount" value="0.1" />
</bean>
</beans>
```

La configuration précédente opère de manière semblable au pseudo-code suivant :

```
DiscountFactoryBean &aaa = new DiscountFactoryBean();
&aaa.setProduct(new Battery("AAA", 2.5));
&aaa.setDiscount(0.2);
Product aaa = (Product) &aaa.createInstance();

DiscountFactoryBean &cdrw = new DiscountFactoryBean();
&cdrw.setProduct(new Disc("CD-RW", 1.5));
&cdrw.setDiscount(0.1);
Product cdrw = (Product) &cdrw.createInstance();
```

4.5 Déclarer des beans correspondant à des champs statiques

Problème

Nous souhaitons déclarer dans le conteneur Spring IoC un bean qui correspond à un champ statique. En effet, en Java, les valeurs constantes sont généralement déclarées sous forme de champs statiques.

Solution

Pour déclarer un bean à partir d'un champ statique, nous utilisons le bean de fabrique intégré `FieldRetrievingFactoryBean` ou, dans Spring 2.x, la balise `<util:constant>`.

Explications

Tout d'abord, nous définissons dans la classe `Product` deux constantes pour des produits.

```
package com.apress.springrecipes.shop;

public abstract class Product {

    public static final Product AAA = new Battery("AAA", 2.5);
    public static final Product CDRW = new Disc("CD-RW", 1.5);
    ...
}
```

Pour déclarer un bean correspondant à un champ statique, nous pouvons employer le bean de fabrique intégré `FieldRetrievingFactoryBean` en précisant le nom complet du champ dans la propriété `staticField`.

```

<beans ...>
    <bean id="aaa" class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
        <property name="staticField">
            <value>com.apress.springrecipes.shop.Product.AAA</value>
        </property>
    </bean>

    <bean id="cdrw" class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean">
        <property name="staticField">
            <value>com.apress.springrecipes.shop.Product.CDRW</value>
        </property>
    </bean>
</beans>

```

La configuration précédente équivaut au code suivant :

```

Product aaa = com.apress.springrecipes.shop.Product.AAA;
Product cdrw = com.apress.springrecipes.shop.Product.CDRW;

```

Au lieu de préciser explicitement le nom du champ dans la propriété staticField, il est possible de l'utiliser comme nom du bean FieldRetrievingFactoryBean. Toutefois, cette solution a l'inconvénient de créer des noms assez longs et verbeux.

```

<beans ...>
    <bean id="com.apress.springrecipes.shop.Product.AAA"
          class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />

    <bean id="com.apress.springrecipes.shop.Product.CDRW"
          class="org.springframework.beans.factory.config.FieldRetrievingFactoryBean" />
</beans>

```

Spring 2.x nous permet de déclarer un bean correspondant à un champ statique à l'aide de la balise <util:constant>. Cette approche est plus simple que l'emploi de FieldRetrievingFactoryBean. Pour disposer de cette balise, nous devons ajouter la définition du schéma util dans l'élément racine <beans>.

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-2.5.xsd">

    <util:constant id="aaa"
                  static-field="com.apress.springrecipes.shop.Product.AAA" />

    <util:constant id="cdrw"
                  static-field="com.apress.springrecipes.shop.Product.CDRW" />
</beans>

```

4.6 Déclarer des beans correspondant aux propriétés d'un objet

Problème

Nous souhaitons déclarer dans le conteneur Spring IoC un bean qui correspond à une propriété d'un objet ou à une propriété imbriquée (c'est-à-dire un chemin de propriété).

Solution

Pour déclarer un bean à partir d'une propriété d'objet ou d'un chemin de propriété, nous utilisons le bean de fabrique intégré `PropertyPathFactoryBean` ou, dans Spring 2.x, la balise `<util:property-path>`.

Explications

Pour nous servir d'exemple, créons une classe `ProductRanking` qui possède une propriété `bestSeller` de type `Product`.

```
package com.apress.springrecipes.shop;

public class ProductRanking {

    private Product bestSeller;

    public Product getBestSeller() {
        return bestSeller;
    }

    public void setBestSeller(Product bestSeller) {
        this.bestSeller = bestSeller;
    }
}
```

Dans la déclaration de bean suivante, la propriété `bestSeller` est définie par un bean interne. Par définition, nous ne pouvons pas obtenir un bean interne à partir de son nom. En revanche, nous pouvons y accéder en tant que propriété du bean `productRanking`. Le bean de fabrique `PropertyPathFactoryBean` nous permet de déclarer un bean à partir d'une propriété d'objet ou d'un chemin de propriété.

```
<beans ...>
    <bean id="productRanking"
          class="com.apress.springrecipes.shop.ProductRanking">
        <property name="bestSeller">
            <bean class="com.apress.springrecipes.shop.Disc">
                <property name="name" value="CD-RW" />
                <property name="price" value="1.5" />
            </bean>
        </property>
    </bean>
```

```
<bean id="bestSeller"
      class="org.springframework.beans.factory.config.<-
            PropertyPathFactoryBean">
      <property name="targetObject" ref="productRanking" />
      <property name="propertyPath" value="bestSeller" />
  </bean>
</beans>
```

La propriété `propertyPath` de `PropertyPathFactoryBean` accepte non seulement un nom de propriété seul, mais également un chemin de propriété, avec des points comme séparateurs. La configuration précédente équivaut au code suivant :

```
Product bestSeller = productRanking.getBestSeller();
```

Au lieu de préciser explicitement les propriétés `targetObject` et `propertyPath`, nous pouvons les combiner pour former le nom du bean `PropertyPathFactoryBean`. Cette solution présente l'inconvénient d'exiger des noms assez longs et verbeux.

```
<bean id="productRanking.bestSeller"
      class="org.springframework.beans.factory.config.<-
            PropertyPathFactoryBean" />
```

Spring 2.x nous permet de déclarer un bean correspondant à une propriété d'objet ou à un chemin de propriété à l'aide de la balise `<util:property-path>`. Cette approche est plus simple que l'emploi de `PropertyPathFactoryBean`. Pour disposer de cette balise, nous devons ajouter la définition du schéma `util` dans l'élément racine `<beans>`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:util="http://www.springframework.org/schema/util"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/util
                           http://www.springframework.org/schema/util/spring-util-2.5.xsd">
    ...
    <util:property-path id="bestSeller" path="productRanking.bestSeller" />
</beans>
```

Pour tester ce chemin de propriété, retrouvons-le à partir du conteneur IoC et affichons-le sur la console.

```
package com.apress.springrecipes.shop;
...
public class Main {

    public static void main(String[] args) throws Exception {
        ...
        Product bestSeller = (Product) context.getBean("bestSeller");
        System.out.println(bestSeller);
    }
}
```

4.7 Fixer les portées de bean

Problème

Lorsque nous déclarons un bean dans le fichier de configuration, nous définissons en réalité un template pour la création du bean, non une instance réelle. Lorsqu'un bean est demandé par la méthode `getBean()` ou par le biais d'une référence depuis d'autres beans, Spring choisit l'instance de bean à retourner en fonction de la portée de bean. Nous souhaitons parfois fixer la portée appropriée à la place de la portée par défaut.

Solution

Dans Spring 2.x, la portée de bean est fixée par l'attribut `scope` de l'élément `<bean>`. Par défaut, Spring crée une seule instance de chaque bean déclaré dans le conteneur IoC et cette instance est partagée dans le contexte de ce conteneur. Cette unique instance d'un bean est retournée par tous les appels à `getBean()` et par toutes les références au bean. Cette portée se nomme `singleton` et correspond à la portée par défaut pour tous les beans. Le Tableau 4.1 recense toutes les portées de bean valides dans Spring.

Tableau 4.1 : Portées de bean reconnues par Spring

<i>Portée</i>	<i>Description</i>
<code>singleton</code>	Crée une seule instance d'un bean dans chaque conteneur Spring IoC.
<code>prototype</code>	Crée une nouvelle instance d'un bean à chaque demande.
<code>request</code>	Crée une seule instance d'un bean par requête HTTP ; valide uniquement dans le contexte d'une application web.
<code>session</code>	Crée une seule instance d'un bean par session HTTP ; valide uniquement dans le contexte d'une application web.
<code>globalSession</code>	Crée une seule instance d'un bean par session HTTP globale ; valide uniquement dans le contexte d'une application de type portail.

Dans Spring 1.x, seules les portées `singleton` et `prototype` sont reconnues. Elles sont indiquées avec l'attribut `singleton` (c'est-à-dire `singleton="true"` ou `singleton="false"`), non avec l'attribut `scope`.

Explications

Pour illustrer le concept de portée de bean, prenons l'exemple d'un chariot d'achat dans notre application marchande. Tout d'abord, nous créons la classe `ShoppingCart`.

```
package com.apress.springrecipes.shop;
...
public class ShoppingCart {

    private List<Product> items = new ArrayList<Product>();

    public void addItem(Product item) {
        items.add(item);
    }

    public List<Product> getItems() {
        return items;
    }
}
```

Nous déclarons ensuite quelques beans de produits et un bean de chariot dans le conteneur IoC.

```
<beans ...>
    <bean id="aaa" class="com.apress.springrecipes.shop.Battery">
        <property name="name" value="AAA" />
        <property name="price" value="2.5" />
    </bean>

    <bean id="cdrw" class="com.apress.springrecipes.shop.Disc">
        <property name="name" value="CD-RW" />
        <property name="price" value="1.5" />
    </bean>

    <bean id="dvdrw" class="com.apress.springrecipes.shop.Disc">
        <property name="name" value="DVD-RW" />
        <property name="price" value="3.0" />
    </bean>

    <bean id="shoppingCart"
          class="com.apress.springrecipes.shop.ShoppingCart" />
</beans>
```

Dans la classe Main suivante, nous testons notre chariot en y ajoutant plusieurs produits. Supposons que deux clients consultent simultanément notre magasin. Le premier obtient un chariot à l'aide de la méthode `getBean()` et ajoute deux produits. Ensuite, le second client obtient également un chariot à l'aide de la méthode `getBean()` et y place un autre produit.

```
package com.apress.springrecipes.shop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");
```

```
Product aaa = (Product) context.getBean("aaa");
Product cdrw = (Product) context.getBean("cdrw");
Product dvdrw = (Product) context.getBean("dvdrw");

ShoppingCart cart1 = (ShoppingCart) context.getBean("shoppingCart");
cart1.addItem(aaa);
cart1.addItem(cdrw);
System.out.println("Le chariot 1 contient " + cart1.getItems());

ShoppingCart cart2 = (ShoppingCart) context.getBean("shoppingCart");
cart2.addItem(dvdrw);
System.out.println("Le chariot 2 contient " + cart2.getItems());
}
}
```

Avec la déclaration de beans précédente, les deux clients obtiennent la même instance du chariot d'achat.

```
Le chariot 1 contient [AAA 2.5, CD-RW 1.5]
Le chariot 2 contient [AAA 2.5, CD-RW 1.5, DVD-RW 3.0]
```

Puisque Spring utilise par défaut la portée de bean singleton, il crée une seule instance du chariot d'achat par conteneur IoC.

```
<bean id="shoppingCart"
      class="com.apress.springrecipes.shop.ShoppingCart"
      scope="singleton" />
```

Dans notre application marchande, il est préférable que la méthode getBean() donne à chaque client son propre chariot. Pour cela, nous devons fixer la portée du bean shoppingCart à prototype. Spring crée alors une nouvelle instance du bean à chaque appel de la méthode getBean() et pour chaque référence depuis un autre bean.

```
<bean id="shoppingCart"
      class="com.apress.springrecipes.shop.ShoppingCart"
      scope="prototype" />
```

Si nous exécutons à nouveau la classe Main, nous constatons que les deux clients possèdent bien une instance différente du chariot d'achat.

```
Le chariot 1 contient [AAA 2.5, CD-RW 1.5]
Le chariot 2 contient [DVD-RW 3.0]
```

4.8 Modifier l'initialisation et la destruction d'un bean

Problème

Un composant réel doit souvent effectuer certaines tâches d'initialisation avant d'être prêt à l'emploi. Par exemple, il peut ouvrir un fichier, ouvrir une connexion au réseau ou une connexion à une base de données, allouer de la mémoire, etc. Il doit également effectuer les opérations de destruction correspondantes à la fin de son cycle de vie.

Nous avons donc besoin de personnaliser l'initialisation et la destruction d'un bean dans le conteneur Spring IoC.

Solution

Outre l'enregistrement des beans, le conteneur Spring IoC est également responsable de la gestion de leur cycle de vie. Il nous permet ainsi d'effectuer nos propres actions à certains moments particuliers de ce cycle de vie. Ces opérations doivent être placées dans des méthodes de rappel (*callback*) invoquées par le conteneur Spring IoC au moment opportun.

Voici les étapes du cycle de vie d'un bean tel qu'établi par le conteneur Spring IoC. Cette liste sera étendue au fur et à mesure que de nouvelles fonctionnalités du conteneur IoC seront présentées.

1. Créer l'instance du bean, en invoquant un constructeur ou une méthode de fabrique.
2. Affecter des valeurs et des références de beans aux propriétés du bean.
3. **Invoquer les méthodes de rappel pour l'initialisation.**
4. Le bean est prêt à l'emploi.
5. **Lorsque le conteneur est arrêté, invoquer les méthodes de rappel pour la destruction.**

Il existe trois façons de faire en sorte que Spring identifie les méthodes de rappel pour l'initialisation et la destruction. Premièrement, le bean peut implémenter les interfaces `InitializingBean` et `DisposableBean` du cycle de vie ; les méthodes `afterPropertiesSet()` et `destroy()` de ces interfaces correspondent à l'initialisation et à la destruction. Deuxièmement, nous pouvons définir les attributs `init-method` et `destroy-method` dans la déclaration du bean pour préciser les noms des méthodes de rappel. Troisièmement, dans Spring 2.5, nous pouvons marquer les méthodes de rappel pour l'initialisation et la destruction avec les annotations du cycle de vie `@PostConstruct` et `@PreDestroy`, définies dans la JSR-250. Ensuite, il suffit d'enregistrer une instance de `CommonAnnotationBeanPostProcessor` dans le conteneur IoC pour invoquer ces méthodes de rappel.

Explications

Pour comprendre la mise en œuvre du cycle de vie des beans dans le conteneur Spring IoC, prenons comme exemple la fonction de paiement des achats. La classe `Cashier` suivante peut servir au paiement des produits contenus dans le chariot. Elle enregistre le montant et l'heure de chaque passage en caisse dans un fichier texte.

```
package com.apress.springrecipes.shop;
...
public class Cashier {

    private String name;
    private String path;
    private BufferedWriter writer;

    public void setName(String name) {
        this.name = name;
    }

    public void setPath(String path) {
        this.path = path;
    }

    public void openFile() throws IOException {
        File logFile = new File(path, name + ".txt");
        writer = new BufferedWriter(new OutputStreamWriter(
            new FileOutputStream(logFile, true)));
    }

    public void checkout(ShoppingCart cart) throws IOException {
        double total = 0;
        for (Product product : cart.getItems()) {
            total += product.getPrice();
        }
        writer.write(new Date() + "\t" + total + "\r\n");
        writer.flush();
    }

    public void closeFile() throws IOException {
        writer.close();
    }
}
```

Dans la classe `Cashier`, la méthode `openFile()` ouvre le fichier texte avec le nom correspondant à celui de la caisse, dans le chemin indiqué. À chaque invocation de la méthode `checkout()`, un enregistrement de l'achat est ajouté au fichier texte. La méthode `closeFile()` ferme ce fichier pour libérer les ressources système.

Nous déclarons ensuite un bean de caisse nommé `cashier1` dans le conteneur IoC. L'enregistrement des achats effectués à cette caisse est placé dans le fichier `c:/cashier/cashier1.txt`. Nous devons au préalable créer ce répertoire ou préciser un autre répertoire existant.

```
<beans ...>
    ...
    <bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
        <property name="name" value="cashier1" />
        <property name="path" value="c:/cashier" />
    </bean>
</beans>
```

Cependant, si nous essayons, dans la classe Main, de payer nos achats avec cette caisse, nous recevons une exception `NullPointerException`. En effet, la méthode `openFile()` n'a jamais été invoquée pour initialiser le bean.

```
package com.apress.springrecipes.shop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new FileSystemXmlApplicationContext("beans.xml");
        ...
        Cashier cashier1 = (Cashier) context.getBean("cashier1");
        cashier1.checkout(cart1);
    }
}
```

Où doit être placé l'appel à la méthode `openFile()` afin d'initialiser le bean ? En Java, l'initialisation doit se faire dans le constructeur. Dans le cas présent, l'appel à la méthode `openFile()` depuis le constructeur par défaut de la classe `Cashier` fonctionne-t-il ? La réponse est non, car la méthode `openFile()` a besoin que le nom et le chemin aient été fixés pour déterminer le fichier à ouvrir.

```
package com.apress.springrecipes.shop;
...
public class Cashier {
    ...
    public void openFile() throws IOException {
        File logFile = new File(path, name + ".txt");
        writer = new BufferedWriter(new OutputStreamWriter(
            new FileOutputStream(logFile, true)));
    }
}
```

Au moment de l'invocation du constructeur par défaut, ces propriétés ne sont pas encore définies. Nous pourrions ajouter un constructeur qui accepte ces deux propriétés en arguments et le laisser invoquer la méthode `openFile()`. Toutefois, il n'est pas toujours possible de procéder ainsi, ou nous pourrions préférer injecter nos propriétés via un mutateur. En réalité, le meilleur moment pour invoquer la méthode `openFile()` est d'attendre que toutes les propriétés aient été fixées par le conteneur Spring IoC.

Implémenter les interfaces InitializingBean et DisposableBean

Notre bean a la possibilité d'effectuer des tâches d'initialisation et de destruction dans les méthodes de rappel `afterPropertiesSet()` et `destroy()` s'il implémente les interfaces `InitializingBean` et `DisposableBean`. Au cours de la construction du bean, Spring remarque que le bean implémente ces interfaces et invoque les méthodes de rappel au moment opportun.

```

package com.apress.springrecipes.shop;
...
import org.springframework.beans.factory.DisposableBean;
import org.springframework.beans.factory.InitializingBean;

public class Cashier implements InitializingBean, DisposableBean {
    ...
    public void afterPropertiesSet() throws Exception {
        openFile();
    }

    public void destroy() throws Exception {
        closeFile();
    }
}

```

Si nous exécutons à nouveau notre classe Main, nous constatons qu'un enregistrement d'achat est ajouté au fichier texte c:/cashier/cashier1.txt. Toutefois, en implémentant ces interfaces propriétaires, nos beans deviennent spécifiques à Spring et ne peuvent plus être employés en dehors du conteneur Spring IoC.

Définir les attributs init-method et destroy-method

Pour désigner les méthodes de rappel pour l'initialisation et la destruction, une meilleure méthode consiste à employer les attributs `init-method` et `destroy-method` dans la déclaration du bean.

```

<bean id="cashier1" class="com.apress.springrecipes.shop.Cashier"
    init-method="openFile" destroy-method="closeFile">
    <property name="name" value="cashier1" />
    <property name="path" value="c:/cashier" />
</bean>

```

Grâce à l'utilisation de ces deux attributs, la classe Cashier n'est plus obligée d'implémenter les interfaces `InitializingBean` et `DisposableBean`. Nous pouvons également supprimer ses méthodes `afterPropertiesSet()` et `destroy()`.

Utiliser les annotations @PostConstruct et @PreDestroy

Dans Spring 2.5, nous pouvons marquer les méthodes de rappel pour l'initialisation et la destruction à l'aide des annotations de cycle de vie définies dans la JSR-250, `@PostConstruct` et `@PreDestroy`.

 **INFO**

Pour utiliser les annotations de la JSR-250, vous devez inclure `common-annotations.jar` (qui se trouve dans le sous-répertoire `lib/j2ee` du répertoire d'installation de Spring) dans le chemin d'accès aux classes. Si votre application s'exécute sous Java SE 6 ou Java EE 5, il est inutile d'inclure ce fichier JAR.

```

package com.apress.springrecipes.shop;
...
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;

public class Cashier {
    ...
    @PostConstruct
    public void openFile() throws IOException {
        File logFile = new File(path, name + ".txt");
        writer = new BufferedWriter(new OutputStreamWriter(
            new FileOutputStream(logFile, true)));
    }

    @PreDestroy
    public void closeFile() throws IOException {
        writer.close();
    }
}

```

Ensuite, nous enregistrons une instance de `CommonAnnotationBeanPostProcessor` dans le conteneur IoC afin que nos méthodes de rappel pour l'initialisation et la destruction soient invoquées. Il est alors inutile de préciser les attributs `init-method` et `destroy-method` du bean.

```

<beans ...>
    ...
    <bean class="org.springframework.context.annotation.➥
        CommonAnnotationBeanPostProcessor" />

    <bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
        <property name="name" value="cashier1" />
        <property name="path" value="c:/cashier" />
    </bean>
</beans>

```

Une autre solution consiste à inclure l'élément `<context:annotation-config>` dans notre fichier de configuration des beans pour qu'une instance de `CommonAnnotationBeanPostProcessor` soit automatiquement enregistrée. Pour disposer de cette balise, il ne faut pas oublier la définition du schéma context dans l'élément racine `<beans>`.

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:annotation-config />
    ...
</beans>

```

4.9 Rendre les beans conscients de l'existence du conteneur

Problème

Lorsqu'un composant est bien conçu, il ne présente aucune dépendance directe avec son conteneur. Cependant, il est parfois nécessaire que des beans aient connaissance des ressources du conteneur.

Solution

Nos beans peuvent être informés des ressources du conteneur Spring IoC en implémentant certaines interfaces Aware recensées au Tableau 4.2. Spring injecte les ressources correspondantes dans nos beans en utilisant les mutateurs définis par ces interfaces.

Tableau 4.2 : Interfaces Aware définies dans Spring

<i>Interface</i>	<i>Ressource cible</i>
BeanNameAware	Le nom de bean de ses instances configurées dans le conteneur IoC.
BeanFactoryAware	La fabrique de beans actuelle, avec laquelle nous pouvons invoquer les services du conteneur.
ApplicationContextAware ¹	Le contexte d'application actuel, avec lequel nous pouvons invoquer les services du conteneur.
MessageSourceAware	Une source de messages, avec laquelle nous pouvons obtenir les messages textuels.
ApplicationEventPublisherAware	Un producteur d'événements applicatifs, avec lequel nous pouvons publier des événements d'application.
ResourceLoaderAware	Un chargeur de ressources, avec lequel nous pouvons charger des ressources externes.

1. En réalité, l'interface ApplicationContext étend les interfaces MessageSource, ApplicationEventPublisher et ResourceLoader. Il suffit donc de connaître le contexte d'application pour accéder à tous ces services. Cependant, il est préférable de choisir une interface Aware dont la portée est minimale et répond aux besoins.

Les mutateurs des interfaces Aware sont appelés par Spring après que les propriétés de bean ont été fixées, mais avant que les méthodes de rappel pour l'initialisation soient invoquées :

1. Créer l'instance du bean, en invoquant un constructeur ou une méthode de fabrique.
2. Affecter des valeurs et des références de beans aux propriétés du bean.

3. Invoquer les mutateurs définis dans les interfaces Aware.
4. Invoquer les méthodes de rappel pour l'initialisation.
5. Le bean est prêt à l'emploi.
6. Lorsque le conteneur est arrêté, invoquer les méthodes de rappel pour la destruction.

Lorsque des beans implémentent les interfaces Aware, il ne faut pas oublier qu'ils sont liés à Spring et qu'ils ne fonctionneront sans doute pas correctement en dehors du conteneur Spring IoC. L'utilisation de ces interfaces propriétaires doit donc se faire après mûre réflexion.

Explications

Nous pouvons, par exemple, faire en sorte que notre bean de caisse ait connaissance de son nom de bean dans le conteneur IoC en implémentant l'interface BeanNameAware. Lorsque ce nom de bean est injecté, nous l'enregistrons en tant que nom de caisse. Cela nous évite d'utiliser une autre propriété name pour la caisse.

```
package com.apress.springrecipes.shop;  
...  
import org.springframework.beans.factory.BeanNameAware;  
  
public class Cashier implements BeanNameAware {  
    ...  
    public void setBeanName(String beanName) {  
        this.name = beanName;  
    }  
}
```

Puisque nous utilisons le nom de bean comme nom de caisse, nous pouvons simplifier la déclaration de notre bean de caisse. Nous effaçons la configuration de la propriété name, voire la méthode setName().

```
<bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">  
    <property name="path" value="c:/cashier" />  
</bean>
```



INFO

Vous souvenez-vous qu'il est possible de préciser directement le nom de champ et le chemin de propriété en tant que noms de bean pour FieldRetrievingFactoryBean et PropertyPathFactoryBean ? En réalité, ces deux beans de fabrique implémentent l'interface BeanNameAware.

4.10 Crée des postprocesseurs de beans

Problème

Nous souhaitons enregistrer nos propres plug-in dans le conteneur Spring IoC de manière à manipuler les instances de beans au cours de la construction.

Solution

Un *postprocesseur de beans* autorise la mise en place de traitements supplémentaires sur le bean avant et après la méthode de rappel pour l'initialisation. Ce postprocesseur traite une par une toutes les instances de beans présentes dans le conteneur IoC, non une seule instance. De manière générale, les postprocesseurs de beans servent à vérifier la validité des propriétés de beans ou à les modifier d'après certains critères.

Un postprocesseur de beans doit implémenter l'interface `BeanPostProcessor`. Nous pouvons traiter chaque bean avant et après l'invocation de la méthode de rappel pour l'initialisation en implémentant les méthodes `postProcessBeforeInitialization()` et `postProcessAfterInitialization()`. Spring passe chaque instance de bean à ces deux méthodes avant et après l'invocation de la méthode de rappel pour l'initialisation :

1. Créer l'instance du bean, en invoquant un constructeur ou une méthode de fabrique.
2. Affecter des valeurs et des références de beans aux propriétés du bean.
3. Invoquer les mutateurs définis dans les interfaces Aware.
4. **Passer l'instance du bean à la méthode `postProcessBeforeInitialization()` de chaque postprocesseur de beans.**
5. Invoquer les méthodes de rappel pour l'initialisation.
6. **Passer l'instance du bean à la méthode `postProcessAfterInitialization()` de chaque postprocesseur de beans.**
7. Le bean est prêt à l'emploi.
8. Lorsque le conteneur est arrêté, invoquer les méthodes de rappel pour la destruction.

Lorsqu'on utilise une fabrique de beans comme conteneur IoC, les postprocesseurs de beans ne peuvent être enregistrés qu'en utilisant du code, plus précisément à l'aide de la méthode `addBeanPostProcessor()`. En revanche, lorsqu'on utilise un contexte d'application, l'enregistrement se fait simplement en déclarant une instance du postprocesseur dans le fichier de configuration des beans.

Explications

Supposons que nous voulions nous assurer que le chemin de journalisation de `Cashier` existe avant d'ouvrir le fichier journal, cela afin d'éviter une exception `FileNotFoundException`. Puisqu'il s'agit d'une contrainte commune à tous les composants qui utilisent le système de fichiers, il est préférable de la mettre en œuvre de manière générale et réutilisable. Un postprocesseur de beans est une bonne solution pour implémenter une telle fonctionnalité dans Spring.

Tout d'abord, pour que le postprocesseur de beans sache identifier les beans à vérifier, nous créons une interface de marquage, `StorageConfig`, qui sera implémentée par les beans cibles. Par ailleurs, pour que le postprocesseur puisse vérifier l'existence du chemin, il doit être en mesure d'accéder à la propriété `path`. Pour cela, il suffit d'ajouter la méthode `getPath()` à l'interface.

```
package com.apress.springrecipes.shop;

public interface StorageConfig {

    public String getPath();
}
```

Nous modifions ensuite la classe `Cashier` pour qu'elle implémente cette interface de marquage. Notre postprocesseur de beans ne vérifie que les beans qui implémentent cette interface.

```
package com.apress.springrecipes.shop;
...
public class Cashier implements BeanNameAware, StorageConfig {
    ...
    public String getPath() {
        return path;
    }
}
```

À présent, nous sommes prêts à écrire un postprocesseur de beans qui vérifie le chemin. Puisque le meilleur moment pour effectuer ce contrôle est avant l'ouverture du fichier dans la méthode d'initialisation, nous implementons la vérification dans la méthode `postProcessBeforeInitialization()`.

```
package com.apress.springrecipes.shop;
...
import org.springframework.beans.BeansException;
import org.springframework.beans.factory.config.BeanPostProcessor;

public class PathCheckingBeanPostProcessor implements BeanPostProcessor {

    public Object postProcessBeforeInitialization(Object bean, String beanName)
        throws BeansException {
        if (bean instanceof StorageConfig) {
            String path = ((StorageConfig) bean).getPath();
        }
    }
}
```

```

        File file = new File(path);
        if (!file.exists()) {
            file.mkdirs();
        }
    }
    return bean;
}

public Object postProcessAfterInitialization(Object bean, String beanName)
    throws BeansException {
    return bean;
}
}

```

Pendant la phase de construction, le conteneur Spring IoC passe une par une toutes les instances de beans à notre postprocesseur de beans. Nous devons donc filtrer les beans en vérifiant qu'ils implémentent l'interface de marquage `StorageConfig`. Si un bean implémente cette interface, nous pouvons accéder à sa propriété `path` via la méthode `getPath()` et vérifier l'existence du chemin dans le système de fichiers. Si ce chemin n'existe pas, il est créé en invoquant `File.mkdirs()`.

Les deux méthodes `postProcessBeforeInitialization()` et `postProcessAfterInitialization()` doivent retourner une instance du bean traité. Autrement dit, le postprocesseur de beans peut remplacer l'instance de bean d'origine par une toute nouvelle instance. Il ne faut pas oublier de retourner l'instance de bean d'origine même si la méthode n'effectue aucune opération.

Pour enregistrer un postprocesseur de beans dans un contexte d'application, il suffit d'en déclarer une instance dans le fichier de configuration des beans. Le contexte d'application est capable de détecter les beans qui implémentent l'interface `BeanPostProcessor` et de les enregistrer pour qu'ils traitent toutes les autres instances de beans dans le conteneur.

```

<beans ...>
    ...
    <bean class="com.apress.springrecipes.shop.<span style="color: red;">➥
          PathCheckingBeanPostProcessor" />

    <bean id="cashier1" class="com.apress.springrecipes.shop.Cashier"
          init-method="openFile" destroy-method="closeFile">
        ...
    </bean>
</beans>

```

Si nous indiquons la méthode de rappel pour l'initialisation dans l'attribut `init-method` ou si nous implementons l'interface `InitializingBean`, notre `PathCheckingBeanPostProcessor` fonctionne parfaitement car il traite le bean de caisse avant que la méthode de rappel pour l'initialisation ne soit invoquée.

En revanche, si le bean de caisse se fonde sur les annotations `@PostConstruct` et `@PreDestroy` de la JSR-250, ainsi que sur une instance de `CommonAnnotationBeanPostProcessor` pour invoquer la méthode de rappel pour l'initialisation, notre `PathCheckingBeanPostProcessor` ne peut pas fonctionner correctement. En effet, la priorité de notre postprocesseur de beans est, par défaut, inférieure à celle de `CommonAnnotationBeanPostProcessor`. Par conséquent, la méthode d'initialisation est appelée avant notre vérification du chemin.

```
<beans ...>
  ...
  <bean class="org.springframework.context.annotation.<span style="color: #0000ff;">➥
        CommonAnnotationBeanPostProcessor" />

  <bean class="com.apress.springrecipes.shop.<span style="color: #0000ff;">➥
        PathCheckingBeanPostProcessor" />

  <bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
    ...
  </bean>
</beans>
```

Pour définir l'ordre des postprocesseurs de beans, ils doivent implémenter l'interface `Ordered` ou `PriorityOrdered` et retourner leur indice dans la méthode `getOrder()`. Plus la valeur retournée par cette méthode est faible, plus la priorité est élevée. Par ailleurs, la valeur retournée depuis l'interface `PriorityOrdered` est toujours prioritaire sur celle retournée depuis l'interface `Ordered`.

Puisque `CommonAnnotationBeanPostProcessor` implémente l'interface `PriorityOrdered`, notre `PathCheckingBeanPostProcessor` doit également l'implémenter pour avoir une chance d'être invoqué avant lui.

```
package com.apress.springrecipes.shop;
...
import org.springframework.beans.factory.config.BeanPostProcessor;
import org.springframework.core.PriorityOrdered;

public class PathCheckingBeanPostProcessor implements BeanPostProcessor,
    PriorityOrdered {

    private int order;

    public int getOrder() {
        return order;
    }

    public void setOrder(int order) {
        this.order = order;
    }
    ...
}
```

Dans le fichier de configuration des beans, nous devons affecter une valeur d'ordre plus faible à notre PathCheckingBeanPostProcessor pour qu'il vérifie et crée le chemin du bean de caisse avant que sa méthode d'initialisation ne soit appelée par CommonAnnotationBeanPostProcessor. Puisque l'ordre par défaut de CommonAnnotationBeanPostProcessor est Ordered.LOWEST_PRECEDENCE, nous pouvons simplement affecter la valeur zéro à notre PathCheckingBeanPostProcessor.

```
<beans ...>
    ...
    <bean class="org.springframework.context.annotation.➥
        CommonAnnotationBeanPostProcessor" />

    <bean class="com.apress.springrecipes.shop.PathCheckingBeanPostProcessor">
        <property name="order" value="0" />
    </bean>

    <bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
        <property name="path" value="c:/cashier" />
    </bean>
</beans>
```

Puisque la valeur d'ordre par défaut de notre PathCheckingBeanPostProcessor est zéro, nous pouvons simplement omettre ce paramètre. Par ailleurs, nous pouvons toujours employer `<context:annotation-config>` pour que CommonAnnotationBeanPostProcessor soit enregistré automatiquement.

```
<beans ...>
    ...
    <context:annotation-config />

    <bean class="com.apress.springrecipes.shop.➥
        PathCheckingBeanPostProcessor" />
</beans>
```

4.11 Externaliser les configurations de beans

Problème

Lorsque les beans sont définis dans le fichier de configuration, il ne faut pas oublier que le mélange des détails de déploiement, comme le chemin de fichier, l'adresse de serveur, le nom d'utilisateur et le mot de passe, avec les configurations de beans constitue une mauvaise pratique. Habituellement, les configurations de beans sont produites par des développeurs d'applications, tandis que les détails du déploiement sont l'affaire des administrateurs système ou des déployeurs.

Solution

Spring est livré avec un postprocesseur de fabrique de beans, nommé `PropertyPlaceholderConfigurer`, pour que nous puissions externaliser une partie des configurations de beans dans un fichier de propriétés. Nous pouvons utiliser des variables de la forme `${var}` dans le fichier de configuration pour que `PropertyPlaceholderConfigurer` charge les propriétés définies dans le fichier et les utilise à la place des variables.

Un *postprocesseur de fabrique de beans* diffère d'un postprocesseur de beans en cela que sa cible est le conteneur IoC – la fabrique de beans ou le contexte d'application –, non les instances de beans. Il agit sur le conteneur IoC après le chargement des configurations de beans, mais avant la création de toute instance de bean. Ce type de postprocesseur est généralement utilisé pour modifier les configurations de beans avant l'instanciation de ces derniers. Spring est livré avec plusieurs postprocesseurs de fabrique de beans et, en pratique, nous devons rarement écrire un tel postprocesseur.

Explications

Nous avons précédemment indiqué le chemin de journalisation d'une caisse dans le fichier de configuration des beans. Il est préférable de ne pas mélanger de tels détails de déploiement avec la configuration des beans. Une meilleure approche consiste à placer les détails du déploiement dans un fichier de propriétés, par exemple `config.properties`, situé à la racine du chemin d'accès aux classes. Il suffit ensuite de définir le chemin de journalisation dans ce fichier.

```
cashier.path=c:/cashier
```

Dans le fichier de configuration des beans, nous utilisons des variables de la forme `${var}`. Pour charger des propriétés à partir d'un fichier externe et les utiliser à la place des variables, le postprocesseur de fabrique de beans `PropertyPlaceholderConfigurer` doit être enregistré dans notre contexte d'application. Nous pouvons indiquer un fichier de propriétés dans la propriété `location` ou plusieurs fichiers dans la propriété `locations`.

```
<beans ...>
    ...
    <bean class="org.springframework.beans.factory.config.<span style="color: red;">➥
          PropertyPlaceholderConfigurer">
        <property name="location">
            <value>config.properties</value>
        </property>
    </bean>

    <bean id="cashier1" class="com.apress.springrecipes.shop.Cashier">
        <property name="path" value="${cashier.path}" />
    </bean>
</beans>
```

Étant implémenté comme un postprocesseur de fabrique de beans, `PropertyPlaceholderConfigurer` remplace les variables présentes dans notre fichier de configuration des beans par les propriétés externes avant d'instancier les beans.

Dans Spring 2.5, l'enregistrement de `PropertyPlaceholderConfigurer` peut se faire simplement à l'aide de l'élément `<context:property-placeholder>`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd">

    <context:property-placeholder location="config.properties" />
    ...
</beans>
```

4.12 Obtenir des messages textuels multilingues

Problème

Pour qu'une application prenne en charge l'internationalisation (également appelée I18N, car il y a 18 caractères entre le premier, *i*, et le dernier, *n*, de ce mot), elle doit être en mesure de trouver les messages textuels qui correspondent aux paramètres régionaux¹ (*locale*).

Solution

Le contexte d'application de Spring est capable d'obtenir les messages textuels correspondant aux paramètres régionaux en utilisant leur clé. En général, les messages adaptés à une région sont enregistrés dans un fichier de propriétés séparé, appelé *bundle de ressources*.

L'interface `MessageSource` définit plusieurs méthodes permettant d'obtenir les messages. L'interface `ApplicationContext` étend cette interface afin que tous les contextes d'application puissent accéder aux messages textuels adéquats. Un contexte d'application délègue la résolution des messages à un bean nommé `messageSource`. `ResourceBundleMessageSource` est l'implémentation de `MessageSource` la plus courante et obtient les messages à partir de bundles de ressources pour différentes régions.

1. N.d.T. : les paramètres régionaux sont indiqués sous la forme d'un code de langue et d'un code de pays (facultatif). Par exemple, `fr_FR` correspond à la langue française en France et `en_US` correspond à l'anglais aux États-Unis.

Explications

Pour servir d'exemple, nous créons le bundle de ressources `messages_fr_FR.properties` pour la langue française en France. Les bundles de ressources sont chargés à partir de la racine du chemin d'accès aux classes.

`alert.checkout=Un chariot d'achat a été confirmé.2`

Pour obtenir les messages à partir des bundles de ressources, nous utilisons l'implémentation `ResourceBundleMessageSource` de `MessageSource`. Le nom de ce bean doit être `messageSource` si l'on veut que le contexte d'application le détecte. Nous devons préciser le nom de base des bundles de ressources gérés par `ResourceBundleMessageSource`.

```
<beans ...>
    ...
    <bean id="messageSource" class="org.springframework.context.support.ResourceBundleMessageSource">
        <property name="basename">
            <value>messages</value>
        </property>
    </bean>
</beans>
```

Avec cette définition de `MessageSource`, si nous recherchons un message textuel pour la France, dont la langue préférée est le français, le bundle de ressources `messages_fr_FR.properties`, qui correspond à la fois à la langue et au pays, est examiné en premier. Si ce bundle n'existe pas ou si le message ne peut pas être trouvé, le bundle `messages_fr.properties`, qui correspond uniquement à la langue, est recherché. S'il ne peut pas être trouvé, le bundle `messages.properties` par défaut, qui correspond à toutes les régions, est recherché. Pour de plus amples informations concernant le chargement d'un bundle de ressources, consultez la documentation JavaDoc de la classe `java.util.ResourceBundle`.

Nous pouvons à présent demander au contexte d'application d'obtenir un message à l'aide de la méthode `getMessage()`. Le premier argument est la clé du message et le troisième indique la région cible.

```
package com.apress.springrecipes.shop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.FileSystemXmlApplicationContext;

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new FileSystemXmlApplicationContext("beans.xml");
```

2. N.d.T. : les deux apostrophes sont obligatoires pour obtenir le caractère de l'apostrophe dans une entrée d'un bundle de ressources.

```
    ...
    String alert =
        context.getMessage("alert.checkout", null, Locale.FRANCE);
    System.out.println(alert);
}
```

Le deuxième argument de la méthode `getMessage()` est un tableau des paramètres du message. Dans le message textuel, nous pouvons définir plusieurs paramètres avec des indices.

`alert.checkout=Un chariot d'achat de {0} dollars a été confirmé le {1}.`

Nous passons un tableau dont les éléments sont convertis en chaîne de caractères avant de remplacer les paramètres du message.

```
package com.apress.springrecipes.shop;
...
public class Main {

    public static void main(String[] args) throws Exception {
        ...
        String alert = context.getMessage("alert.checkout",
            new Object[] { 4, new Date() }, Locale.FRANCE);
        System.out.println(alert);
    }
}
```

Dans la classe `Main`, nous pouvons obtenir les messages textuels car nous accédons directement au contexte d'application. En revanche, pour qu'un bean puisse obtenir les messages, il doit implémenter l'interface `ApplicationContextAware` ou l'interface `MessageSourceAware`. Ensuite, nous pouvons supprimer l'accès au message dans la classe `Main`.

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.MessageSource;
import org.springframework.context.MessageSourceAware;

public class Cashier implements BeanNameAware, MessageSourceAware,
    StorageConfig {
    ...
    private MessageSource messageSource;

    public void setMessageSource(MessageSource messageSource) {
        this.messageSource = messageSource;
    }

    public void checkout(ShoppingCart cart) throws IOException {
        ...
        String alert = messageSource.getMessage("alert.checkout",
            new Object[] { total, new Date() }, Locale.FRANCE);
        System.out.println(alert);
    }
}
```

4.13 Communiquer à l'aide des événements d'application

Problème

Dans le modèle classique de communication entre des composants, l'émetteur doit localiser le récepteur afin d'invoquer l'une de ses méthodes. Le composant émetteur doit donc connaître le composant récepteur. Ce type de communication est direct et simple, mais les deux participants sont fortement couplés.

En utilisant un conteneur IoC, les composants peuvent communiquer par une interface à la place d'une implémentation. Ce modèle de communication permet de réduire le couplage. Toutefois, il n'est efficace que si le composant émetteur communique avec un seul récepteur. Lorsque l'émetteur doit contacter plusieurs récepteurs, il doit les appeler un par un.

Solution

Le contexte d'application de Spring prend en charge les communications à base d'événements entre beans. Dans ce modèle, le composant émetteur publie simplement un événement sans connaître le récepteur. En réalité, il peut exister plusieurs composants récepteurs. Par ailleurs, le récepteur n'a pas besoin de savoir qui envoie l'événement. Il peut écouter les multiples événements provenant de différents émetteurs à la fois. Ainsi, les composants émetteurs et récepteurs sont faiblement couplés.

Dans Spring, toutes les classes d'événements doivent dériver de la classe `ApplicationEvent`. Ensuite, n'importe quel bean peut envoyer un événement en invoquant la méthode `publishEvent()` d'un producteur d'événements d'application. Pour qu'un bean écoute certains événements, il doit implémenter l'interface `ApplicationListener` et traiter les événements dans la méthode `onApplicationEvent()`. En réalité, Spring passe tous les événements à un auditeur, qui doit donc les filtrer.

Explications

Définir des événements

Pour activer la communication à base d'événements, la première étape consiste à définir l'événement. Supposons que nous voulions que notre bean de caisse publie un événement `CheckoutEvent` après que le chariot d'achat a été payé. Cet événement contient deux propriétés : le montant du chariot et la date de paiement. Dans Spring, tous les événements doivent étendre la classe abstraite `ApplicationEvent` et passer la source de l'événement en argument du constructeur.

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.ApplicationEvent;
public class CheckoutEvent extends ApplicationEvent {

    private double amount;
    private Date time;

    public CheckoutEvent(Object source, double amount, Date time) {
        super(source);
        this.amount = amount;
        this.time = time;
    }

    public double getAmount() {
        return amount;
    }

    public Date getTime() {
        return time;
    }
}
```

Publier des événements

Pour publier un événement, nous créons simplement une instance de l'événement et invoquons la méthode `publishEvent()` d'un producteur d'événements d'application, que l'on obtient en implémentant l'interface `ApplicationEventPublisherAware`.

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.ApplicationEventPublisher;
import org.springframework.context.ApplicationEventPublisherAware;

public class Cashier implements BeanNameAware, MessageSourceAware,
    ApplicationEventPublisherAware, StorageConfig {
    ...
    private ApplicationEventPublisher applicationEventPublisher;

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

    public void checkout(ShoppingCart cart) throws IOException {
        ...
        CheckoutEvent event = new CheckoutEvent(this, total, new Date());
        applicationEventPublisher.publishEvent(event);
    }
}
```

Écouter des événements

Tout bean défini dans le contexte d'application qui implémente l'interface `ApplicationListener` est averti de tous les événements. Par conséquent, dans la méthode `onApplicationEvent()`, nous devons filtrer les événements qui nous intéressent. Dans l'auditeur suivant, nous supposons que nous envoyons un courrier électronique au client afin de lui signaler l'achat.

```
package com.apress.springrecipes.shop;  
...  
import org.springframework.context.ApplicationEvent;  
import org.springframework.context.ApplicationListener;  
  
public class CheckoutListener implements ApplicationListener {  
  
    public void onApplicationEvent(ApplicationEvent event) {  
        if (event instanceof CheckoutEvent) {  
            double amount = ((CheckoutEvent) event).getAmount();  
            Date time = ((CheckoutEvent) event).getTime();  
  
            // Utiliser le montant et la date.  
            System.out.println("Événement d'achat [" +  
                amount + ", " + time + "]");  
        }  
    }  
}
```

Cet auditeur doit être enregistré dans le contexte d'application pour qu'il reçoive les événements. Pour cela, il suffit de déclarer une instance de bean de cet auditeur. Le contexte d'application reconnaît les beans qui implémentent l'interface `ApplicationListener` et leur envoie chaque événement.

```
<beans ...>  
...  
    <bean class="com.apress.springrecipes.shop.CheckoutListener" />  
</beans>
```

Le contexte d'application publie les événements du conteneur, comme `ContextClosedEvent`, `ContextRefreshedEvent` et `RequestHandledEvent`. Pour qu'un bean reçoive ces événements, il doit implémenter l'interface `ApplicationListener`.

4.14 Enregistrer des éditeurs de propriétés dans Spring

Problème

Un éditeur de propriétés est une fonctionnalité de l'API JavaBeans dont le rôle est de convertir des valeurs de propriétés vers et depuis des valeurs textuelles. Chaque éditeur de propriétés est réservé à un certain type de propriétés. Nous souhaitons employer des éditeurs de propriétés pour simplifier la configuration de nos beans.

Solution

Le conteneur Spring IoC accepte d'utiliser des éditeurs de propriétés dans la configuration des beans. Par exemple, avec un éditeur de propriétés pour le type `java.net.URL`, nous pouvons préciser une chaîne d'URL pour affecter une propriété de type `URL`. Spring convertit automatiquement la chaîne d'URL en un objet `URL` et l'injecte dans la propriété. Spring est livré avec plusieurs éditeurs de propriétés qui se chargent de la conversion des propriétés de beans de types communs.

Nous devons enregistrer un éditeur de propriétés dans le conteneur Spring IoC avant de pouvoir l'utiliser. `CustomEditorConfigurer` est implémenté comme un postprocesseur de fabrique de beans pour que nous puissions enregistrer nos propres éditeurs de propriétés avant que les beans ne soient instanciés.

Explications

Supposons que nous voulions que le classement de nos produits soit fondé sur les ventes au cours d'une certaine période. Pour cela, nous ajoutons les propriétés `fromDate` et `toDate` à la classe `ProductRanking`.

```
package com.apress.springrecipes.shop;
...
public class ProductRanking {

    private Product bestSeller;
    private Date fromDate;
    private Date toDate;

    // Accesseurs et mutateurs.
    ...
}
```

Dans un programme Java, nous pouvons fixer la valeur d'une propriété `java.util.Date` en invoquant la méthode `DateFormat.parse()` avec une chaîne de date conforme à un motif particulier.

```
DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
productRanking.setFromDate(dateFormat.parse("2007-09-01"));
productRanking.setToDate(dateFormat.parse("2007-09-30"));
```

Pour écrire la configuration de bean équivalente dans Spring, nous commençons par déclarer un bean `dateFormat` avec le motif configuré. Puisque la méthode `parse()` est appelée pour convertir les chaînes de date en objets de date, nous pouvons la considérer comme une méthode d'instance de fabrique pour créer des beans de date.

```
<beans ...>
    ...
    <bean id="dateFormat" class="java.text.SimpleDateFormat">
        <constructor-arg value="yyyy-MM-dd" />
    </bean>
```

```

<bean id="productRanking"
      class="com.apress.springrecipes.shop.ProductRanking">
    <property name="bestSeller">
      <bean class="com.apress.springrecipes.shop.Disc">
        <property name="name" value="CD-RW" />
        <property name="price" value="1.5" />
      </bean>
    </property>
    <property name="fromDate">
      <bean factory-bean="dateFormat" factory-method="parse">
        <constructor-arg value="2007-09-01" />
      </bean>
    </property>
    <property name="toDate">
      <bean factory-bean="dateFormat" factory-method="parse">
        <constructor-arg value="2007-09-30" />
      </bean>
    </property>
  </bean>
</beans>

```

La configuration précédente est un peu trop complexe pour fixer des propriétés de date. En réalité, le conteneur Spring IoC est capable de convertir les valeurs textuelles de nos propriétés en utilisant des éditeurs de propriétés. La classe `CustomDateEditor` fournie avec Spring permet de convertir des chaînes de date en propriétés de type `java.util.Date`. Pour bénéficier de cette fonction, nous devons déclarer une instance de cette classe dans le fichier de configuration des beans.

```

<beans ...>
  ...
  <bean id="dateEditor"
        class="org.springframework.beans.propertyeditors.CustomDateEditor">
    <constructor-arg>
      <bean class="java.text.SimpleDateFormat">
        <constructor-arg value="yyyy-MM-dd" />
      </bean>
    </constructor-arg>
    <constructor-arg value="true" />
  </bean>
</beans>

```

Cet éditeur exige un objet `DateFormat` en premier argument du constructeur. Le second argument indique si l'éditeur accepte les valeurs vides.

Ensuite, nous devons enregistrer cet éditeur de propriétés dans une instance de `CustomEditorConfigurer` pour que Spring puisse convertir les propriétés de type `java.util.Date`. Après cela, nous pouvons utiliser un format textuel pour affecter une date à n'importe quelle propriété `java.util.Date` :

```

<beans ...>
  ...
  <bean class="org.springframework.beans.factory.config.➥
    CustomEditorConfigurer">

```

```
<property name="customEditors">
    <map>
        <entry key="java.util.Date">
            <ref local="dateEditor" />
        </entry>
    </map>
</property>
</bean>

<bean id="productRanking"
    class="com.apress.springrecipes.shop.ProductRanking">
    <property name="bestSeller">
        <bean class="com.apress.springrecipes.shop.Disc">
            <property name="name" value="CD-RW" />
            <property name="price" value="1.5" />
        </bean>
    </property>
    <property name="fromDate" value="2007-09-01" />
    <property name="toDate" value="2007-09-30" />
</bean>
</beans>
```

Nous vérifions le fonctionnement de notre configuration de `CustomDateEditor` avec la classe Main suivante :

```
package com.apress.springrecipes.shop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) throws Exception {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");
        ...
        ProductRanking productRanking =
            (ProductRanking) context.getBean("productRanking");
        System.out.println(
            "Produit classé du " + productRanking.getFromDate() +
            " au " + productRanking.getToDate());
    }
}
```

Outre `CustomDateEditor`, Spring fournit plusieurs éditeurs de propriétés pour la conversion des types de données communs, comme `CustomNumberEditor`, `ClassEditor`, `FileEditor`, `LocaleEditor`, `StringArrayPropertyEditor` et `URLEditor`. Parmi eux, `ClassEditor`, `FileEditor`, `LocaleEditor` et `URLEditor` sont déjà enregistrés par Spring et nous n'avons donc pas à les enregistrer à nouveau. Pour de plus amples informations concernant l'utilisation de ces éditeurs, consultez la documentation JavaDoc des classes correspondantes dans le paquetage `org.springframework.beans.propertyeditors`.

4.15 Crée des éditeurs de propriétés

Problème

Nous ne voulons pas nous limiter à l'enregistrement des éditeurs de propriétés intégrés, mais également écrire nos propres éditeurs pour convertir nos types de données.

Solution

Nous pouvons développer nos propres éditeurs de propriétés en implémentant l'interface `java.beans.PropertyEditor` ou en dérivant de la classe `java.beans.PropertyEditorSupport`.

Explications

Écrivons, par exemple, un éditeur de propriétés pour la classe `Product`. La représentation textuelle d'un produit est constituée de trois parties : le nom de classe concret, le nom du produit et le prix du produit. Chaque partie est séparée par une virgule. La classe `ProductEditor` suivante assure la conversion.

```
package com.apress.springrecipes.shop;

import java.beans.PropertyEditorSupport;

public class ProductEditor extends PropertyEditorSupport {

    public String getAsText() {
        Product product = (Product) getValue();
        return product.getClass().getName() + "," + product.getName() + ","
               + product.getPrice();
    }

    public void setAsText(String text) throws IllegalArgumentException {
        String[] parts = text.split(",");
        try {
            Product product = (Product) Class.forName(parts[0]).newInstance();
            product.setName(parts[1]);
            product.setPrice(Double.parseDouble(parts[2]));
            setValue(product);
        } catch (Exception e) {
            throw new IllegalArgumentException(e);
        }
    }
}
```

La méthode `getAsText()` convertit une propriété en une chaîne de caractères, tandis que la méthode `setAsText()` convertit une chaîne de caractères en une propriété. La valeur de la propriété est obtenue et fixée en invoquant les méthodes `getValue()` et `setValue()`.

Avant qu'il puisse être utilisé, nous devons enregistrer notre éditeur personnalisé dans une instance de `CustomEditorConfigurer`. L'enregistrement se passe de la même manière que pour les éditeurs intégrés. Nous pouvons ensuite affecter à toute propriété de type `Product` un produit donné sous forme d'une chaîne de caractères.

```
<beans ...>
    ...
    <bean class="org.springframework.beans.factory.config.<span style="color: red;">➥
          CustomEditorConfigurer">
        <property name="customEditors">
            <map>
                ...
                <entry key="com.apress.springrecipes.shop.Product">
                    <bean class="com.apress.springrecipes.shop.<span style="color: red;">➥
                          ProductEditor" />
                </entry>
            </map>
        </property>
    </bean>

    <bean id="productRanking"
          class="com.apress.springrecipes.shop.ProductRanking">
        <property name="bestSeller">
            <value>com.apress.springrecipes.shop.Disc,CD-RW,1.5</value>
        </property>
        ...
    </bean>
</beans>
```

En réalité, l'API JavaBeans recherche automatiquement un éditeur de propriétés pour une classe. Pour que cette recherche puisse se faire correctement, l'éditeur doit se trouver dans le même paquetage que la classe cible et son nom doit être le nom de la classe cible suffixé par `Editor`. Si l'éditeur de propriétés respecte cette convention, comme c'est le cas de `ProductEditor`, il est inutile de l'enregistrer manuellement dans le conteneur Spring IoC.

4.16 Charger des ressources externes

Problème

Une application a parfois besoin de lire des ressources externes, tels des fichiers de texte, des fichiers XML, des fichiers de propriétés ou des fichiers d'image, à partir de différents emplacements, comme un système de fichiers, un chemin d'accès aux classes ou une URL. Normalement, nous devons employer différentes API pour charger des ressources à partir d'emplacements différents.

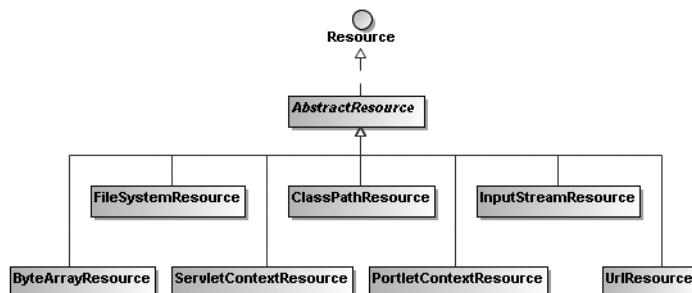
Solution

Le chargeur de ressources de Spring propose une méthode `getResource()` unifiée pour l'accès à une ressource externe à partir de son chemin. Selon le préfixe ajouté à ce chemin, le chargement des ressources se fait depuis différents emplacements. Le préfixe `file` permet ainsi de charger une ressource à partir du système de fichiers. Le chargement d'une ressource à partir du chemin d'accès aux classes se fait avec le préfixe `classpath`. Le chemin d'une ressource peut également contenir une URL.

Dans Spring, `Resource` est une interface générale qui représente une ressource externe. Spring fournit plusieurs implémentations de cette interface (voir Figure 4.1). La méthode `getResource()` du chargeur de ressources choisit l'implémentation de `Resource` à instancier en fonction du chemin de la ressource.

Figure 4.1

Implémentations communes de l'interface `Resource`.



Explications

Supposons que nous voulions afficher une bannière au démarrage de notre application marchande. Cette dernière contient le texte suivant stocké dans un fichier nommé `banner.txt`, qui peut se trouver dans le chemin de notre application.

```
*****
* Bienvenue dans Mon magasin ! *
*****
```

Nous écrivons alors la classe `BannerLoader` pour charger la bannière et l'afficher sur la console. Puisque cette classe doit accéder à un chargeur de ressources pour charger la bannière, elle doit implémenter l'interface `ApplicationContextAware` ou l'interface `ResourceLoaderAware`.

```
package com.apress.springrecipes.shop;
...
import org.springframework.context.ResourceLoaderAware;
import org.springframework.core.io.Resource;
import org.springframework.core.io.ResourceLoader;
```

```
public class BannerLoader implements ResourceLoaderAware {  
  
    private ResourceLoader resourceLoader;  
  
    public void setResourceLoader(ResourceLoader resourceLoader) {  
        this.resourceLoader = resourceLoader;  
    }  
  
    public void showBanner() throws IOException {  
        Resource banner = resourceLoader.getResource("file:banner.txt");  
        InputStream in = banner.getInputStream();  
  
        BufferedReader reader = new BufferedReader(new InputStreamReader(in));  
        while (true) {  
            String line = reader.readLine();  
            if (line == null)  
                break;  
            System.out.println(line);  
        }  
        reader.close();  
    }  
}
```

En invoquant la méthode `getResource()` du contexte d'application, nous obtenons une ressource externe désignée par un chemin. Puisque le fichier de la bannière se trouve sur le système de fichiers, le chemin de la ressource doit commencer par `file`. La méthode `getInputStream()` nous permet d'obtenir un flux d'entrée pour cette ressource. Il nous suffit ensuite de lire le contenu du fichier ligne par ligne avec un `BufferedReader` et de les afficher sur la console.

Nous déclarons une instance de `BannerLoader` dans le fichier de configuration des beans pour afficher la bannière. Puisque cet affichage doit se faire au démarrage, nous indiquons que la méthode `showBanner()` est une méthode d'initialisation.

```
<bean id="bannerLoader"  
      class="com.apress.springrecipes.shop.BannerLoader"  
      init-method="showBanner" />
```

Préfixes des ressources

Le chemin de ressource précédent désignait une ressource à partir d'un chemin relatif sur le système de fichiers. Nous pouvons également employer un chemin absolu.

`file:c:/shop/banner.txt`

Lorsque la ressource se trouve dans le chemin d'accès aux classes, nous employons le préfixe `classpath`. Si aucune information de chemin n'est présente, elle est chargée à partir de la racine du chemin d'accès aux classes.

`classpath:banner.txt`

Si la ressource est stockée dans un paquetage précis, nous indiquons le chemin absolu à partir de la racine du chemin d'accès aux classes.

```
classpath:com/apress/springrecipes/shop/banner.txt
```

Une ressource peut non seulement être désignée par un chemin du système de fichiers ou le chemin d'accès aux classes, mais également par une URL.

```
http://springrecipes.apress.com/shop/banner.txt
```

Si le chemin d'une ressource ne contient aucun préfixe, elle est chargée à partir d'un emplacement défini par le contexte d'application. Pour `FileSystemXmlApplicationContext`, la ressource est chargée depuis le système de fichiers. Pour `ClassPathXmlApplicationContext`, elle est chargée à partir du chemin d'accès aux classes.

Injecter des ressources

Nous ne sommes pas obligés d'appeler la méthode `getResource()` pour charger explicitement une ressource, mais nous pouvons l'injecter à l'aide d'un mutateur.

```
package com.apress.springrecipes.shop;
...
import org.springframework.core.io.Resource;
public class BannerLoader {
    private Resource banner;
    public void setBanner(Resource banner) {
        this.banner = banner;
    }
    public void showBanner() throws IOException {
        InputStream in = banner.getInputStream();
        ...
    }
}
```

Dans la configuration du bean, nous précisons simplement le chemin de la ressource affectée à la propriété `Resource`. Spring se sert de l'éditeur de propriétés préenregistré `ResourceEditor` pour le convertir en un objet `Resource` qui sera injecté dans notre bean.

```
<bean id="bannerLoader"
      class="com.apress.springrecipes.shop.BannerLoader"
      init-method="showBanner">
    <property name="banner">
        <value>classpath:com/apress/springrecipes/shop/banner.txt</value>
    </property>
</bean>
```

4.17 En résumé

Dans ce chapitre, nous avons découvert différentes manières de créer un bean, que ce soit en invoquant un constructeur, en invoquant une méthode statique ou d'instance de fabrique, en utilisant un bean de fabrique ou en l'obtenant à partir d'un champ statique ou d'une propriété d'objet. Le conteneur Spring IoC facilite la création des beans à l'aide de ces méthodes.

Dans Spring 2.x, nous pouvons préciser la portée de bean pour contrôler l'instanciation des beans demandés. La portée par défaut est `singleton` – pour chaque conteneur Spring IoC, Spring crée une seule instance de bean partagée. `prototype` est l'autre portée de bean souvent employée – Spring crée une nouvelle instance de bean à chaque demande.

Nous personnalisons l'initialisation et la destruction de nos beans en définissant les méthodes de rappel correspondantes. Par ailleurs, nos beans peuvent implémenter des interfaces `Aware` de manière à prendre connaissance des configurations et des infrastructures du conteneur. Le conteneur Spring IoC invoque ces méthodes à des moments particuliers du cycle de vie d'un bean.

Spring accepte l'enregistrement de postprocesseurs de beans dans le conteneur IoC pour effectuer des traitements supplémentaires sur le bean avant et après l'invocation des méthodes de rappel pour l'initialisation. Ces postprocesseurs peuvent traiter tous les beans du conteneur IoC. En général, ils servent à vérifier la validité des propriétés de beans ou à modifier des propriétés selon certains critères.

Nous avons également vu quelques fonctionnalités élaborées du conteneur IoC, comme la configuration de bean à partir des fichiers de propriétés, l'obtention de messages textuels à partir de bundles de ressources, la publication et l'écoute d'événements d'application, l'utilisation des éditeurs de propriétés pour convertir des valeurs de propriétés en valeurs textuelles, ainsi que le changement de ressources externes. Toutes ces fonctionnalités sont très utiles pour le développement d'applications avec Spring.

Le chapitre suivant se focalise sur une autre fonctionnalité centrale du framework Spring : la programmation orientée aspect.

Proxy dynamique et Spring AOP classique

Au sommaire de ce chapitre

- ✓ Problèmes associés aux préoccupations transversales non modularisées
- ✓ Modulariser les préoccupations transversales avec un proxy dynamique
- ✓ Modulariser les préoccupations transversales avec des greffons Spring classiques
- ✓ Désigner des méthodes avec des points d'action Spring classique
- ✓ Créer automatiquement des proxies pour les beans
- ✓ En résumé

Ce chapitre détaille la nature des préoccupations transversales et leur modularisation avec des proxies dynamiques. Il s'intéresse également à l'utilisation de la programmation orientée aspect dans Spring, que l'on désigne par Spring AOP classique dans Spring 1.x et qui a considérablement changé entre les versions 1.x et 2.x. La présentation de Spring AOP classique n'est là que pour des questions de compatibilité. Chaque nouveau projet créé avec Spring 2.x doit opter pour la nouvelle approche de Spring AOP, qui sera examinée en détail au chapitre suivant.

La *programmation orientée aspect* (POA) est une nouvelle méthodologie qui complète la *programmation orientée objet* (POO) traditionnelle. La POA n'a pas pour objectif de remplacer la POO et, en réalité, elles sont souvent combinées. Dans le monde de la programmation orientée objet, les applications sont organisées en classes et interfaces. Ces concepts sont bien adaptés à l'implémentation des exigences métier centrales, non à celle des préoccupations transversales, c'est-à-dire les fonctions ou les exigences qui concernent plusieurs modules d'une application. Les préoccupations transversales sont très fréquentes dans les applications d'entreprise, notamment la journalisation, la validation et la gestion des transactions.

La POA propose aux développeurs une autre manière de structurer leurs applications. À la place des classes et des interfaces de la POO, les principaux éléments de programmation de la POA sont les *aspects*. Vous pouvez considérer que, dans la POA, un aspect modularise les préoccupations transversales comme une classe modularise des états et des comportements dans la POO.

Avec le conteneur IoC, le framework de programmation orientée aspect fait partie des modules centraux de Spring. Il existe aujourd'hui sur le marché de nombreux frameworks POA servant des objectifs différents et fondés sur des technologies différentes, mais seuls les trois frameworks open-source suivants sont réellement présents :

- AspectJ, qui a fusionné avec AspectWerkz depuis la version 5 (<http://www.eclipse.org/aspectj/>) ;
- JBoss AOP, qui est un sous-projet du serveur d'applications JBoss (<http://labs.jboss.com/jbossaop/>) ;
- Spring AOP, qui fait partie du framework Spring (<http://www.springframework.org/>).

Des trois, AspectJ est le framework POA le plus complet et le plus répandu dans la communauté Java. Spring AOP n'entre pas en concurrence avec AspectJ. Son objectif est non de proposer une autre implémentation complète de la POA, mais d'apporter une solution POA qui s'intègre de manière cohérente avec le conteneur IoC. En réalité, Spring AOP s'occupe des préoccupations transversales uniquement pour les beans déclarés dans son conteneur IoC.

La mise en œuvre de Spring AOP se fonde sur la technologie des proxies dynamiques. Bien que l'emploi de Spring AOP ait beaucoup évolué entre les versions 1.x et 2.x, la technologie sous-jacente est restée la même. De plus, Spring AOP étant rétrocompatible, nous pouvons utiliser nos éléments de Spring 1.x AOP dans Spring 2.x.

- Dans Spring version 1.x, l'utilisation de la POA se fait au travers d'un ensemble d'API Spring AOP propriétaires.
- Dans Spring version 2.0, l'utilisation de la POA se fait en écrivant des POJO avec des annotations AspectJ ou des configurations XML dans le fichier de configuration des beans.

À la fin de ce chapitre, vous serez en mesure d'identifier les préoccupations transversales d'une application et de les modulariser avec Spring AOP classique. Vous aurez également acquis des connaissances de base sur la technologie d'implémentation de Spring AOP, les proxies dynamiques. Elles vous permettront de mieux comprendre les mécanismes de Spring AOP.

5.1 Problèmes associés aux préoccupations transversales non modularisées

Par définition, une préoccupation transversale est une fonctionnalité qui concerne plusieurs modules d'une application. En général, il est difficile de modulariser ce type de préoccupation avec une approche orientée objet classique. Pour comprendre les problèmes liés aux préoccupations transversales, prenons l'exemple d'une calculatrice. Tout d'abord, nous créons deux interfaces, `ArithmeticCalculator` et `UnitCalculator`, pour le calcul arithmétique et la conversion des unités de mesure.

```
package com.apress.springrecipes.calculator;

public interface ArithmeticCalculator {

    public double add(double a, double b);
    public double sub(double a, double b);
    public double mul(double a, double b);
    public double div(double a, double b);
}

---

package com.apress.springrecipes.calculator;

public interface UnitCalculator {

    public double kilogramToPound(double kilogram);
    public double kilometerToMile(double kilometer);
}
```

Nous proposons ensuite une implémentation simple de chaque interface. Les instructions `println` nous informent de l'exécution des méthodes.

```
package com.apress.springrecipes.calculator;

public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    public double add(double a, double b) {
        double result = a + b;
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }

    public double sub(double a, double b) {
        double result = a - b;
        System.out.println(a + " - " + b + " = " + result);
        return result;
    }

    public double mul(double a, double b) {
        double result = a * b;
        System.out.println(a + " * " + b + " = " + result);
        return result;
    }
}
```

```

        public double div(double a, double b) {
            double result = a / b;
            System.out.println(a + " / " + b + " = " + result);
            return result;
        }
    }

---

package com.apress.springrecipes.calculator;

public class UnitCalculatorImpl implements UnitCalculator {

    public double kilogramToPound(double kilogram) {
        double pound = kilogram * 2.2;
        System.out.println(kilogram + " kilogrammes = " + pound + " livres");
        return pound;
    }

    public double kilometerToMile(double kilometer) {
        double mile = kilometer * 0.62;
        System.out.println(kilometer + " kilomètres = " + mile + " miles");
        return mile;
    }
}

```

Tracer les méthodes

Dans la plupart des applications, nous souhaitons tracer les activités qui ont lieu au cours de l'exécution du programme. Pour la plate-forme Java, il existe plusieurs implémentations d'un tel mécanisme de suivi. Toutefois, si nous souhaitons que l'application reste indépendante du système de journalisation, nous pouvons choisir la bibliothèque Apache Commons Logging. Elle fournit des API abstraites indépendantes de l'implémentation et nous permet de changer d'implémentation sans modifier notre code.


INFO

Pour utiliser la bibliothèque Apache Commons Logging, vous devez inclure le fichier commons-logging.jar (situé dans le répertoire lib/jakarta-commons de l'installation de Spring) dans le chemin d'accès aux classes.

Dans le cas de nos calculatrices, nous pouvons consigner le début et la fin de chaque méthode, ainsi que les arguments passés et la valeur de retour.

```

package com.apress.springrecipes.calculator;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

```

```
private Log log = LogFactory.getLog(this.getClass());  
  
public double add(double a, double b) {  
    log.info("Méthode add() invoquée avec " + a + ", " + b);  
    double result = a + b;  
    System.out.println(a + " + " + b + " = " + result);  
    log.info("Méthode add() terminée par " + result);  
    return result;  
}  
  
public double sub(double a, double b) {  
    log.info("Méthode sub() invoquée avec " + a + ", " + b);  
    double result = a - b;  
    System.out.println(a + " - " + b + " = " + result);  
    log.info("Méthode sub() terminée par " + result);  
    return result;  
}  
  
public double mul(double a, double b) {  
    log.info("Méthode mul() invoquée avec " + a + ", " + b);  
    double result = a * b;  
    System.out.println(a + " * " + b + " = " + result);  
    log.info("Méthode mul() terminée par " + result);  
    return result;  
}  
  
public double div(double a, double b) {  
    log.info("Méthode div() invoquée avec " + a + ", " + b);  
    double result = a / b;  
    System.out.println(a + " / " + b + " = " + result);  
    log.info("Méthode div() terminée par " + result);  
    return result;  
}  
}  
  
---  
  
package com.apress.springrecipes.calculator;  
  
import org.apache.commons.logging.Log;  
import org.apache.commons.logging.LogFactory;  
  
public class UnitCalculatorImpl implements UnitCalculator {  
  
    private Log log = LogFactory.getLog(this.getClass());  
  
    public double kilogramToPound(double kilogram) {  
        log.info("Méthode kilogramToPound() invoquée avec " + kilogram);  
        double pound = kilogram * 2.2;  
        System.out.println(kilogram + " kilogram = " + pound + " pound");  
        log.info("Méthode kilogramToPound() terminée par " + pound);  
        return pound;  
    }  
  
    public double kilometerToMile(double kilometer) {  
        log.info("Méthode kilometerToMile() invoquée avec " + kilometer);  
    }  
}
```

```
        double mile = kilometer * 0.62;
        System.out.println(kilometer + " kilometer = " + mile + " mile");
        log.info("Méthode kilometerToMile() terminée par " + mile);
        return mile;
    }
}
```

Nous pouvons à présent choisir une implémentation de la journalisation prise en charge par la bibliothèque Commons Logging. Aujourd'hui, elle reconnaît principalement la bibliothèque Log4J d'Apache et l'API JDK Logging (disponible pour le JDK 1.4 et les versions ultérieures). Nous préconisons Log4J car elle est plus puissante et plus facile à configurer.

INFO

Pour utiliser la bibliothèque Log4J, vous devez inclure le fichier log4j-1.2.15.jar (situé dans le répertoire lib/log4j de l'installation de Spring) dans le chemin d'accès aux classes. Lorsque la bibliothèque Log4J est détectée dans le chemin d'accès aux classes, Commons Logging l'utilise comme implémentation de la journalisation.

La configuration de la bibliothèque Log4J peut se faire par l'intermédiaire d'un fichier de propriétés nommé log4j.properties et placé à la racine du chemin d'accès aux classes. L'exemple de configuration suivant définit un Appender nommé stdout qui affiche les messages de journalisation sur la console en utilisant le motif indiqué. Pour de plus amples informations concernant les formats de journalisation de Log4J, consultez la documentation de cette bibliothèque.

```
### Envoyer les messages de journalisation sur stdout. ###
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=➥
%d{yyyy-MM-dd HH:mm:ss} %5p %c{1}:%L - %m%n

### Fixer le niveau de journalisation racine. ###
log4j.rootLogger=error, stdout

### Fixer le niveau de journalisation de l'application. ###
log4j.logger.com.apress.springrecipes.calculator=info
```

Log4J reconnaît six niveaux de journalisation qui fixent l'urgence des messages. Les voici par ordre décroissant de priorité : fatal, error, warn, info, debug et trace. Dans le fichier de configuration de Log4J précédent, le niveau de journalisation racine (par défaut) de l'application est fixé à error. Autrement dit, seuls les messages de niveau error et fatal sont consignés par défaut. En revanche, pour le paquetage com.apress.springrecipes.calculator et ses sous-paquetages, les messages de niveau info et supérieurs sont affichés.

Pour tester les fonctions de base de ces deux calculatrices et la configuration de la journalisation, nous écrivons la classe Main suivante :

```
package com.apress.springrecipes.calculator;

public class Main {

    public static void main(String[] args) {
        ArithmeticCalculator arithmeticCalculator =
            new ArithmeticCalculatorImpl();
        arithmeticCalculator.add(1, 2);
        arithmeticCalculator.sub(4, 3);
        arithmeticCalculator.mul(2, 3);
        arithmeticCalculator.div(4, 2);

        UnitCalculator unitCalculator = new UnitCalculatorImpl();
        unitCalculator.kilogramToPound(10);
        unitCalculator.kilometerToMile(5);
    }
}
```

Valider les arguments

Nous allons à présent ajouter une restriction à nos calculatrices : seuls les nombres positifs sont acceptés. Au début de chaque méthode, nous invoquons validate() pour vérifier que tous les arguments sont des nombres positifs. Si un argument est négatif, nous lançons une exception `IllegalArgumentException`.

```
package com.apress.springrecipes.calculator;
...
public class ArithmeticCalculatorImpl implements ArithmeticCalculator {
    ...
    public double add(double a, double b) {
        validate(a);
        validate(b);
        ...
    }

    public double sub(double a, double b) {
        validate(a);
        validate(b);
        ...
    }

    public double mul(double a, double b) {
        validate(a);
        validate(b);
        ...
    }

    public double div(double a, double b) {
        validate(a);
        validate(b);
        ...
    }
}
```

```

private void validate(double a) {
    if (a < 0) {
        throw new IllegalArgumentException("Nombres positifs attendus");
    }
}
---

package com.apress.springrecipes.calculator;
...
public class UnitCalculatorImpl implements UnitCalculator {
    ...
    public double kilogramToPound(double kilogram) {
        validate(kilogram);
        ...
    }

    public double kilometerToMile(double kilometer) {
        validate(kilometer);
        ...
    }

    private void validate(double a) {
        if (a < 0) {
            throw new IllegalArgumentException("Nombres positifs attendus");
        }
    }
}

```

Identifier les problèmes

Plus nous ajoutons d'exigences sans rapport avec l'objectif principal de l'application, comme la journalisation et la validation, plus les méthodes des calculatrices s'allongent. Ces exigences de niveau système concernent de nombreux modules et sont appelées *préoccupations transversales* pour les différencier des exigences métier, qui constituent les *préoccupations centrales* d'un système. Les préoccupations transversales d'une application d'entreprise sont notamment la journalisation, la validation, le pooling, la mise en cache, l'authentification et les transactions. La Figure 5.1 illustre les préoccupations transversales dans notre application de calculatrice.

Cependant, avec les classes et les interfaces comme seuls éléments de programmation, l'approche orientée objet traditionnelle ne permet pas de modulariser correctement les préoccupations transversales. Les développeurs doivent souvent les mélanger aux préoccupations centrales au sein des mêmes modules. Par conséquent, les préoccupations transversales se retrouvent dans différents modules d'une application et ne peuvent pas être modularisées.

L'absence de modularisation des préoccupations transversales est la source de deux principaux problèmes. Le premier se nomme *mélange de code*. Dans l'exemple des cal-

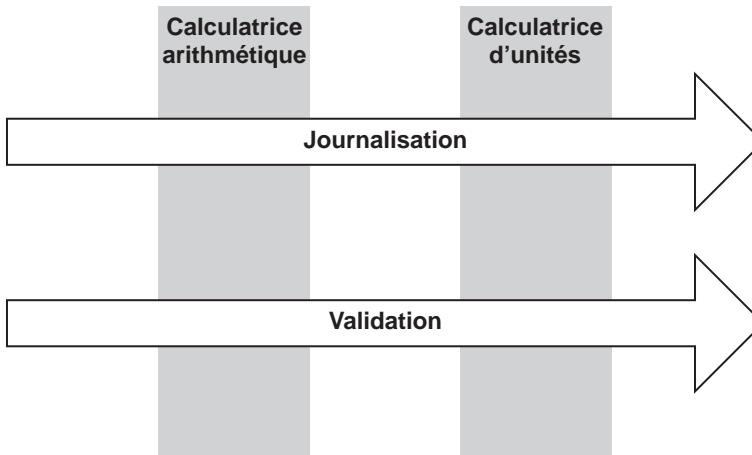


Figure 5.1

Préoccupations transversales dans l'application de calculatrice.

culatrices, chaque méthode doit prendre en charge à la fois plusieurs préoccupations et la logique de calcul. Cela ne facilite pas la maintenance et la réutilisabilité du code. Ainsi, les implémentations des méthodes de calcul précédentes sont difficiles à réutiliser dans une autre application qui ne présente aucune exigence de journalisation et accepte les opérands négatifs.

Le second problème provoqué par l'absence de modularisation des préoccupations transversales se nomme *dispersion de code*. Pour l'exigence de journalisation, nous devons répéter plusieurs fois les instructions de journalisation dans de multiples modules, uniquement pour satisfaire une seule contrainte. Si, par la suite, les critères de journalisation changent, nous devons modifier tous les modules. Par ailleurs, il est difficile de certifier que l'exigence de journalisation est implémentée de manière cohérente. S'il manque une instruction quelque part, le système global de journalisation est incohérent.

Pour toutes ces raisons, les calculatrices doivent se focaliser uniquement sur la logique de calcul. Nous devons retirer les aspects journalisation et validation.

```
package com.apress.springrecipes.calculator;

public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    public double add(double a, double b) {
        double result = a + b;
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }
}
```

```
public double sub(double a, double b) {
    double result = a - b;
    System.out.println(a + " - " + b + " = " + result);
    return result;
}

public double mul(double a, double b) {
    double result = a * b;
    System.out.println(a + " * " + b + " = " + result);
    return result;
}

public double div(double a, double b) {
    double result = a / b;
    System.out.println(a + " / " + b + " = " + result);
    return result;
}
}

---
package com.apress.springrecipes.calculator;

public class UnitCalculatorImpl implements UnitCalculator {

    public double KilogramToPound(double kilogram) {
        double pound = kilogram * 2.2;
        System.out.println(kilogram + " kilogrammes = " + pound + " livres");
        return pound;
    }

    public double KilometerToMile(double kilometer) {
        double mile = kilometer * 0.62;
        System.out.println(kilometer + " kilomètres = " + mile + " miles");
        return mile;
    }
}
```

5.2 Modulariser les préoccupations transversales avec un proxy dynamique

Problème

Puisque l'absence de modularisation des préoccupations transversales conduit aux problèmes de mélange et de dispersion de code, nous souhaitons trouver une méthode pour les modulariser. L'approche orientée objet traditionnelle ne facilite pas cette modularisation car les préoccupations transversales recouvrent plusieurs modules d'une application.

Solution

Nous pouvons appliquer le design pattern *Proxy* pour séparer les préoccupations transversales des préoccupations centrales. *Proxy* (ou *Procuration*) fait partie des vingt-trois design patterns orientés objet du GoF (*Gang of Four*) ; il appartient à la catégorie "patterns structuraux".

Le principe du design pattern *Proxy* est d'envelopper un objet dans un mandataire et d'utiliser celui-ci à la place de l'objet original. Tous les appels effectués sur l'objet original passent tout d'abord au travers du proxy. La Figure 5.2 illustre l'idée générale.

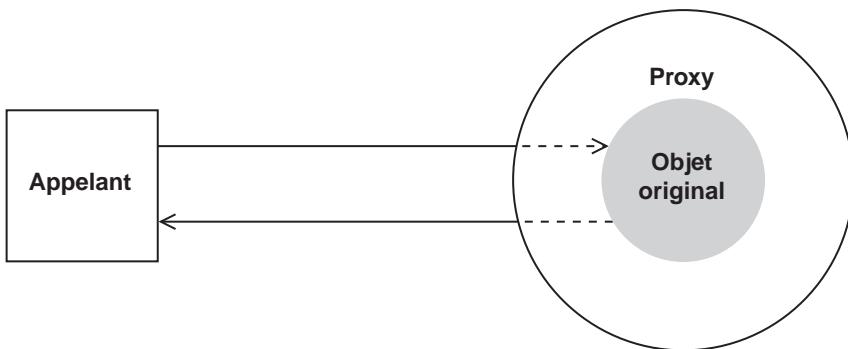


Figure 5.2

Principe général du design pattern *Proxy*.

Le proxy décide si les appels de méthodes doivent être transmis à l'objet original, ainsi que du moment où cela se produit. Dans le même temps, le proxy peut également effectuer des opérations supplémentaires avant et après chaque appel de méthode. C'est pourquoi le proxy représente un bon endroit où mettre en œuvre les préoccupations transversales.

En Java, il existe deux manières d'implémenter le design pattern *Proxy*. La solution traditionnelle consiste à écrire un proxy statique dans un style orienté objet pur. Dans ce cas, un proxy dédié enveloppe l'objet et réalise des tâches avant et après chaque appel de méthode. Puisqu'il est dédié, nous devons écrire, pour chaque interface, une classe de proxy capable de remplacer l'implémentation d'origine. Cette approche est peu efficace dans les grandes applications constituées de centaines ou de milliers de composants.

Une autre solution se fonde sur la notion de proxy dynamique disponible dans le JDK version 1.3 et ultérieure. Ce JDK est capable de créer dynamiquement un proxy pour n'importe quel objet. La seule restriction est que l'objet doit implémenter au moins une

interface, et seuls les appels aux méthodes déclarées dans les interfaces passeront par le proxy. Il existe cependant une autre sorte de proxy, le proxy CGLIB, qui ne souffre pas de cette restriction. Il peut traiter toutes les méthodes déclarées dans une classe, même si elle n'implémente aucune interface.

Les proxies dynamiques sont implémentés à l'aide de l'API Java Reflection. Ils peuvent donc être utilisés de manière plus générale que les proxies statiques. C'est pour cette raison que Spring les emploie au cœur de son implémentation de la POA.

Expllications

Un proxy dynamique du JDK a besoin d'un gestionnaire pour prendre en charge les invocations des méthodes. Un gestionnaire d'invocation n'est rien d'autre qu'une classe qui implémente l'interface `InvocationHandler` suivante :

```
package java.lang.reflect;

public interface InvocationHandler {

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

La seule méthode déclarée par cette interface est `invoke()`. Elle nous permet de contrôler le processus global d'invocation. Son premier argument correspond à l'instance du proxy dont une méthode a été invoquée. Le deuxième argument est l'objet représentant la méthode invoquée ; il est du type `java.lang.reflect.Method`. Le troisième argument est un tableau des arguments d'invocation de la méthode cible. La valeur de retour correspond au résultat de l'invocation de la méthode cible.

Créer le proxy pour la journalisation

En implémentant l'interface `InvocationHandler`, nous pouvons écrire un gestionnaire d'invocation qui consigne l'entrée et la sortie dans une méthode. Nous avons besoin d'un objet de calculatrice qui se charge des calculs ; il est passé en argument du constructeur.

```
package com.apress.springrecipes.calculator;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.util.Arrays;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class CalculatorLoggingHandler implements InvocationHandler {
```

```
private Log log = LogFactory.getLog(this.getClass());  
  
private Object target;  
  
public CalculatorLoggingHandler(Object target) {  
    this.target = target;  
}  
  
public Object invoke(Object proxy, Method method, Object[] args)  
    throws Throwable {  
    // Consigner le début de la méthode, avec le nom de la méthode et  
    // ses arguments.  
    log.info("Méthode " + method.getName() + "() invoquée avec "  
        + Arrays.toString(args));  
  
    // Effectuer les calculs réels avec l'objet calculatrice cible en  
    // appelant Method.invoke() et en lui passant l'objet cible ainsi  
    // que les arguments de la méthode.  
    Object result = method.invoke(target, args);  
  
    // Consigner la fin de la méthode, avec la valeur de retour.  
    log.info("Méthode " + method.getName() + "() terminée par " + result);  
    return result;  
}  
}
```

Grâce au mécanisme de réflexion, la méthode `invoke()` est suffisamment générale pour prendre en charge tous les appels aux méthodes des deux calculatrices. Nous accédons au nom de la méthode en invoquant `Method.getName()` et aux arguments à partir d'un tableau d'objets. Pour effectuer les calculs réels, nous appelons `invoke()` sur l'objet de méthode et lui passons l'objet de la calculatrice cible ainsi que les arguments de la méthode.

Pour créer une instance d'un proxy dynamique du JDK avec un gestionnaire d'invocation, il suffit d'appeler la méthode statique `Proxy.newProxyInstance()`.

```
package com.apress.springrecipes.calculator;  
  
import java.lang.reflect.Proxy;  
  
public class Main {  
  
    public static void main(String[] args) {  
        ArithmeticCalculator arithmeticCalculatorImpl =  
            new ArithmeticCalculatorImpl();  
  
        ArithmeticCalculator arithmeticCalculator =  
            (ArithmeticCalculator) Proxy.newProxyInstance(  
                arithmeticCalculatorImpl.getClass().getClassLoader(),  
                arithmeticCalculatorImpl.getClass().getInterfaces(),  
                new CalculatorLoggingHandler(arithmeticCalculatorImpl));  
        ...  
    }  
}
```

Le premier argument de cette méthode désigne le chargeur de classes pour enregistrer le proxy. Dans la plupart des cas, le proxy doit être défini dans le même chargeur de classes que la classe d'origine. Le deuxième argument correspond aux interfaces implémentées par le proxy. Seuls les appels aux méthodes déclarées dans ces interfaces se feront au travers du proxy. En général, on souhaite que toutes les interfaces de la classe cible soient prises en charge par le proxy. Le dernier argument est notre gestionnaire d'invocation, qui traite les invocations de méthodes. Suite à l'appel de cette méthode, nous obtenons une instance de proxy créée dynamiquement par le JDK. Nous pouvons nous en servir pour que tous les appels aux méthodes de calcul passent par le gestionnaire de journalisation.

Pour faciliter la réutilisation, nous pouvons encapsuler le code de création du proxy dans une méthode statique de la classe du gestionnaire.

```
package com.apress.springrecipes.calculator;
...
import java.lang.reflect.Proxy;

public class CalculatorLoggingHandler implements InvocationHandler {
    ...
    public static Object createProxy(Object target) {
        return Proxy.newProxyInstance(
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            new CalculatorLoggingHandler(target));
    }
}
```

Dans la classe Main, il suffit alors d'appeler la méthode statique pour créer le proxy.

```
package com.apress.springrecipes.calculator;

public class Main {
    public static void main(String[] args) {
        ArithmeticCalculator arithmeticCalculatorImpl =
            new ArithmeticCalculatorImpl();

        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) CalculatorLoggingHandler.createProxy(
                arithmeticCalculatorImpl);
        ...
    }
}
```

Avec ce gestionnaire d'invocation général, nous pouvons également créer dynamiquement un proxy pour UnitCalculator.

```
package com.apress.springrecipes.calculator;

public class Main {
```

```

public static void main(String[] args) {
    ...
    UnitCalculator unitCalculatorImpl = new UnitCalculatorImpl();
    UnitCalculator unitCalculator =
        (UnitCalculator) CalculatorLoggingHandler.createProxy(
            unitCalculatorImpl);
    ...
}

```

La Figure 5.3 illustre la mise en œuvre de la fonctionnalité de journalisation fondée sur le design pattern Proxy.

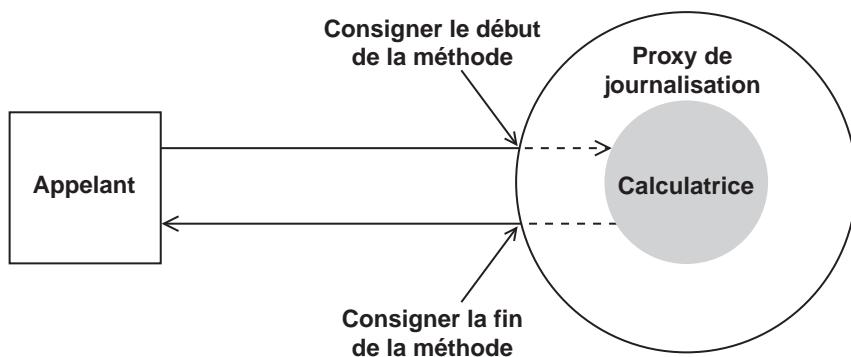


Figure 5.3

Implémenter la fonctionnalité de journalisation avec un proxy.

Créer le proxy pour la validation

De manière similaire, nous pouvons écrire un gestionnaire de validation. Puisque le nombre d'arguments des différentes méthodes peut varier, nous devons parcourir le tableau des arguments pour valider chacun d'eux.

```

package com.apress.springrecipes.calculator;

import java.lang.reflect.InvocationHandler;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class CalculatorValidationHandler implements InvocationHandler {

    public static Object createProxy(Object target) {
        return Proxy.newProxyInstance(
            target.getClass().getClassLoader(),
            target.getClass().getInterfaces(),
            new CalculatorValidationHandler(target));
    }
}

```

```

private Object target;

public CalculatorValidationHandler(Object target) {
    this.target = target;
}

public Object invoke(Object proxy, Method method, Object[] args)
    throws Throwable {
    for (Object arg : args) {
        validate((Double) arg);
    }
    Object result = method.invoke(target, args);
    return result;
}

private void validate(double a) {
    if (a < 0) {
        throw new IllegalArgumentException("Nombres positifs attendus");
    }
}
}

```

Dans la classe Main, nous enveloppons le proxy de journalisation par un proxy de validation afin de constituer une chaîne de proxies. Un appel à une méthode de la calculatrice passe tout d'abord par le proxy de validation, puis par le proxy de journalisation. Autrement dit, la validation se fait avant la journalisation. Si nous préférions inverser l'ordre, il suffit d'envelopper le proxy de validation à l'aide du proxy de journalisation.

```

package com.apress.springrecipes.calculator;

public class Main {

    public static void main(String[] args) {
        ArithmeticCalculator arithmeticCalculatorImpl =
            new ArithmeticCalculatorImpl();

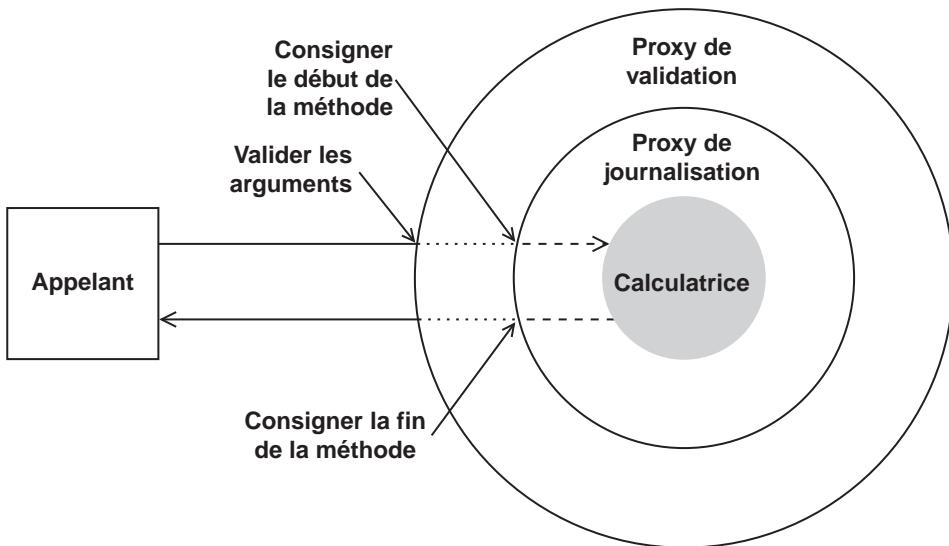
        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) CalculatorValidationHandler.createProxy(
                CalculatorLoggingHandler.createProxy(
                    arithmeticCalculatorImpl));
        ...

        UnitCalculator unitCalculatorImpl = new UnitCalculatorImpl();

        UnitCalculator unitCalculator =
            (UnitCalculator) CalculatorValidationHandler.createProxy(
                CalculatorLoggingHandler.createProxy(
                    unitCalculatorImpl));
        ...
    }
}

```

La Figure 5.4 illustre la mise en œuvre des fonctionnalités de validation et de journalisation à l'aide du design pattern Proxy.

**Figure 5.4**

Implémenter les fonctionnalités de validation et de journalisation avec un proxy.

5.3 Modulariser les préoccupations transversales avec des greffons Spring classiques

Problème

Puisque les préoccupations transversales sont souvent difficiles à modulariser avec une approche orientée objet traditionnelle, nous souhaitons trouver une autre méthode pour effectuer cette modularisation. Les proxies dynamiques constituent une solution, mais ils imposent un travail trop important aux développeurs d'applications, qui doivent écrire le code de bas niveau des proxies.

Solution

La POA définit des concepts de haut niveau pour que les développeurs d'applications puissent exprimer leurs préoccupations transversales. Tout d'abord, l'opération transversale réalisée à un point précis de l'exécution est encapsulée dans un *greffon* (*advice*). Par exemple, nous pouvons encapsuler la journalisation et la validation dans un ou plusieurs greffons.

Spring AOP classique prend en charge quatre types de greffons, chacun intervenant à différents points de l'exécution. Dans la définition formelle de la POA, il existe plusieurs types de points d'exécution, notamment les exécutions de méthodes, les exécu-

tions de constructeurs et les accès aux champs. Toutefois, Spring AOP ne reconnaît que les exécutions de méthodes. Par conséquent, nous pouvons réduire les quatre types de greffons classiques à la liste suivante :

- **Greffon Before.** Intervient avant une exécution de méthode.
- **Greffon After returning.** Intervient après que la méthode a retourné un résultat.
- **Greffon After throwing.** Intervient après que la méthode a lancé une exception.
- **Greffon Around.** Intervient autour d'une exécution de méthode.

Dans l'approche Spring AOP classique, les greffons implémentent l'une des interfaces de greffons propriétaires.

Explications

Dans Spring AOP, la prise en compte des préoccupations transversales se fait uniquement sur les beans déclarés dans le conteneur IoC. Par conséquent, avant de modulariser les préoccupations transversales avec Spring AOP, nous devons adapter notre application de calculatrice au conteneur Spring IoC. Pour cela, il suffit de déclarer les deux calculatrices dans le fichier de configuration des beans.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="arithmeticCalculator"
          class="com.apress.springrecipes.calculator.ArithmeticCalculatorImpl" />

    <bean id="unitCalculator"
          class="com.apress.springrecipes.calculator.UnitCalculatorImpl" />
</beans>
```

Nous modifions ensuite la classe Main pour obtenir les instances des calculatrices à partir du conteneur IoC.

```
package com.apress.springrecipes.calculator;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        ...
    }
}
```

```
    UnitCalculator unitCalculator =
        (UnitCalculator) context.getBean("unitCalculator");
    ...
}
```

Greffon Before

Un greffon *Before* intervient avant l'exécution d'une méthode. Il est créé en implémentant l'interface `MethodBeforeAdvice`. Depuis la méthode `before()`, nous avons accès aux détails de la méthode ainsi qu'à ses arguments.

```
package com.apress.springrecipes.calculator;

import java.lang.reflect.Method;
import java.util.Arrays;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.aop.MethodBeforeAdvice;

public class LoggingBeforeAdvice implements MethodBeforeAdvice {

    private Log log = LogFactory.getLog(this.getClass());

    public void before(Method method, Object[] args, Object target)
        throws Throwable {
        log.info("Méthode " + method.getName() + "() invoquée avec "
            + Arrays.toString(args));
    }
}
```

ASTUCE

Le concept de greffon est comparable à celui de gestionnaire d'invocation lorsqu'on utilise un proxy dynamique.

Le greffon étant prêt, l'étape suivante consiste à l'appliquer à nos beans de calculatrice. Tout d'abord, nous devons déclarer une instance de ce greffon dans le conteneur IoC. Puis l'étape la plus importante est de créer un proxy pour chacun de nos beans de calculatrice afin d'appliquer le greffon. Dans Spring AOP, la création d'un proxy se fait *via* un bean de fabrique nommé `ProxyFactoryBean`.

```
<beans ...>
    ...
    <bean id="loggingBeforeAdvice"
          class="com.apress.springrecipes.calculator.LoggingBeforeAdvice" />

    <bean id="arithmeticCalculatorProxy"
          class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
```

```
<list>
    <value>
        com.apress.springrecipes.calculator.ArithmetricCalculator
    </value>
</list>
</property>
<property name="target" ref="arithmeticCalculator" />
<property name="interceptorNames">
    <list>
        <value>loggingBeforeAdvice</value>
    </list>
</property>
</bean>

<bean id="unitCalculatorProxy"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="proxyInterfaces">
        <list>
            <value>
                com.apress.springrecipes.calculator.UnitCalculator
            </value>
        </list>
    </property>
    <property name="target" ref="unitCalculator" />
    <property name="interceptorNames">
        <list>
            <value>loggingBeforeAdvice</value>
        </list>
    </property>
</bean>
</beans>
```

ASTUCE

La création d'un proxy avec ProxyFactoryBean est comparable à l'écriture du code de création du proxy lorsqu'on utilise un proxy dynamique.

Dans la configuration précédente, nous avons indiqué les proxies avec les interfaces qu'ils implémentent. Seuls les appels aux méthodes déclarées dans ces interfaces passeront par le proxy. L'objet cible (*target*) est celui qui traite les appels aux méthodes réelles. Dans la propriété *interceptorNames*, nous indiquons les noms de greffons pour ce proxy. Les greffons reçoivent une priorité correspondant à leur ordre de définition. Grâce à toutes ces informations, ProxyFactoryBean est en mesure de créer un proxy pour l'objet cible et de l'enregistrer dans le conteneur IoC.

Par défaut, ProxyFactoryBean détecte automatiquement les interfaces implementées par le bean cible et se charge de les faire passer par le proxy. Par conséquent, pour diriger toutes les interfaces d'un bean vers un proxy, nous n'avons rien à faire de particulier.

```
<beans ...>
    ...
    <bean id="arithmeticCalculatorProxy"
          class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="arithmeticCalculator" />
        <property name="interceptorNames">
            <list>
                <value>loggingBeforeAdvice</value>
            </list>
        </property>
    </bean>

    <bean id="unitCalculatorProxy"
          class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="unitCalculator" />
        <property name="interceptorNames">
            <list>
                <value>loggingBeforeAdvice</value>
            </list>
        </property>
    </bean>
</beans>
```

ProxyFactoryBean crée un proxy du JDK si notre bean cible implémente une interface. Dans le cas contraire, il crée un proxy CGLIB qui peut prendre en charge toutes les méthodes déclarées dans la classe du bean.

Dans la classe Main, nous devons obtenir les beans de proxies à partir du conteneur IoC pour que notre greffon de journalisation soit appliqué.

```
package com.apress.springrecipes.calculator;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator)context.getBean("arithmeticCalculatorProxy");
        ...

        UnitCalculator unitCalculator =
            (UnitCalculator) context.getBean("unitCalculatorProxy");
        ...
    }
}
```

Greffon After returning

Outre un greffon Before, nous pouvons également écrire un greffon *After returning* pour journaliser la fin d'une méthode et sa valeur de retour.

```
package com.apress.springrecipes.calculator;

import java.lang.reflect.Method;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.aop.AfterReturningAdvice;

public class LoggingAfterAdvice implements AfterReturningAdvice {

    private Log log = LogFactory.getLog(this.getClass());

    public void afterReturning(Object returnValue, Method method,
        Object[] args, Object target) throws Throwable {
        log.info("Méthode " + method.getName() + "() terminée par "
            + returnValue);
    }
}
```

Pour que ce greffon entre en scène, nous en déclarons une instance dans le conteneur IoC et ajoutons une entrée dans la propriété `interceptorNames` :

```
<beans ...>
    ...
<bean id="loggingAfterAdvice"
      class="com.apress.springrecipes.calculator.LoggingAfterAdvice" />

<bean id="arithmeticCalculatorProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="arithmeticCalculator" />
    <property name="interceptorNames">
        <list>
            <value>loggingBeforeAdvice</value>
            <value>loggingAfterAdvice</value>
        </list>
    </property>
</bean>

<bean id="unitCalculatorProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="unitCalculator" />
    <property name="interceptorNames">
        <list>
            <value>loggingBeforeAdvice</value>
            <value>loggingAfterAdvice</value>
        </list>
    </property>
</bean>
</beans>
```

Greffon After throwing

Le greffon *After throwing* correspond au troisième type de greffon. Tout d'abord, pour générer une exception, nous ajoutons une vérification dans la méthode `div()` de la calculatrice arithmétique. En cas de division par zéro, nous lançons une exception `IllegalArgumentException`.

```
package com.apress.springrecipes.calculator;

public class ArithmeticCalculatorImpl implements ArithmeticCalculator {
    ...
    public double div(double a, double b) {
        if (b == 0) {
            throw new IllegalArgumentException("Division par zéro");
        }
        ...
    }
}
```

Pour ce greffon, nous devons implémenter l'interface `ThrowsAdvice`, qui ne déclare aucune méthode. Ainsi, nous pouvons traiter différents types d'exceptions dans différentes méthodes. Cependant, le nom de chaque méthode doit être `afterThrowing` pour traiter un type particulier d'exception, qui est indiqué par le type de l'argument de la méthode. Par exemple, pour traiter une exception `IllegalArgumentException`, nous écrivons la méthode suivante. À l'exécution, toute exception compatible avec le type indiqué (c'est-à-dire le type et ses sous-types) est prise en charge par cette méthode.

```
package com.apress.springrecipes.calculator;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.springframework.aop.ThrowsAdvice;

public class LoggingThrowsAdvice implements ThrowsAdvice {

    private Log log = LogFactory.getLog(this.getClass());

    public void afterThrowing(IllegalArgumentException e) throws Throwable {
        log.error("Argument invalide");
    }
}
```

Nous devons également déclarer une instance de ce greffon dans le conteneur IoC et ajouter une entrée dans les propriétés `interceptorNames`.

```
<beans ...>
    ...
    <bean id="loggingThrowsAdvice"
          class="com.apress.springrecipes.calculator.LoggingThrowsAdvice" />

    <bean id="arithmeticCalculatorProxy"
          class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="target" ref="arithmeticCalculator" />
```

```

<property name="interceptorNames">
    <list>
        <value>loggingBeforeAdvice</value>
        <value>loggingAfterAdvice</value>
        <value>loggingThrowsAdvice</value>
    </list>
</property>
</bean>

<bean id="unitCalculatorProxy"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="unitCalculator" />
    <property name="interceptorNames">
        <list>
            <value>loggingBeforeAdvice</value>
            <value>loggingAfterAdvice</value>
            <value>loggingThrowsAdvice</value>
        </list>
    </property>
</bean>
</beans>

```

Si nous voulons accéder aux détails de la méthode et à la valeur des arguments, nous pouvons étendre la liste des arguments de la méthode `afterThrowing()`.

```

package com.apress.springrecipes.calculator;

import java.lang.reflect.Method;
import java.util.Arrays;
...
public class LoggingThrowsAdvice implements ThrowsAdvice {
    ...
    public void afterThrowing(Method method, Object[] args, Object target,
        IllegalArgumentException e) throws Throwable {
        log.error("Argument invalide " + Arrays.toString(args)
            + " pour la méthode " + method.getName() + "()");
    }
}

```

Les méthodes d'un greffon After throwing doivent avoir la signature suivante. Les trois premiers arguments sont facultatifs, mais ils doivent être tous déclarés ou aucun.

```
afterThrowing([method, args, target], Throwable subclass)
```

Greffon Around

Le greffon *Around* correspond au dernier type, le plus puissant de tous. Il a un contrôle total sur l'exécution de la méthode et permet de combiner toutes les actions des greffons précédents en un seul. Nous pouvons même contrôler le moment d'exécution de la méthode d'origine et si elle est invoquée.

Dans Spring AOP classique, un greffon Around doit implémenter l'interface `MethodInterceptor`. Elle est définie par l'AOP Alliance pour des raisons de compatibilité entre les différents frameworks POA. Lorsqu'on écrit un greffon Around, il ne faut sur-

tout pas oublier d'invoquer `methodInvocation.proceed()` pour exécuter la méthode d'origine. Le greffon Around suivant combine les greffons Before, After returning et After throwing développés précédemment.

```
package com.apress.springrecipes.calculator;

import java.util.Arrays;

import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;

public class LoggingAroundAdvice implements MethodInterceptor {

    private Log log = LogFactory.getLog(this.getClass());

    public Object invoke(MethodInvocation methodInvocation) throws Throwable {
        log.info("Méthode " + methodInvocation.getMethod().getName()
            + "() invoquée avec "
            + Arrays.toString(methodInvocation.getArguments()));

        try {
            Object result = methodInvocation.proceed();
            log.info("Méthode " + methodInvocation.getMethod().getName()
                + "() terminée par " + result);
            return result;
        } catch (IllegalArgumentException e) {
            log.error("Argument invalide "
                + Arrays.toString(methodInvocation.getArguments())
                + " pour la méthode "
                + methodInvocation.getMethod().getName()
                + "()");
            throw e;
        }
    }
}
```

Le greffon de type Around est très puissant et très souple car nous pouvons même modifier les valeurs initiales des arguments et la valeur de retour. L'utilisation d'un tel greffon demande une grande prudence, car il est très facile d'oublier d'exécuter la méthode d'origine.

ASTUCE

Pour choisir le type de greffon, la règle classique consiste à utiliser la version la moins puissante qui répond aux exigences.

Puisque ce greffon combine tous les autres, il doit être le seul associé aux proxies.

```
<beans ...>
    ...
<bean id="loggingAroundAdvice"
      class="com.apress.springrecipes.calculator.LoggingAroundAdvice" />
```

```
<bean id="arithmeticCalculatorProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="arithmeticCalculator" />
    <property name="interceptorNames">
      <list>
        <value>loggingAroundAdvice</value>
      </list>
    </property>
  </bean>

<bean id="unitCalculatorProxy"
      class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="unitCalculator" />
    <property name="interceptorNames">
      <list>
        <value>loggingAroundAdvice</value>
      </list>
    </property>
  </bean>
</beans>
```

5.4 Désigner des méthodes avec des points d'action Spring classique

Problème

Lorsque nous précisons un greffon pour un proxy AOP, toutes les méthodes déclarées dans la classe cible ou les interfaces du proxy sont interceptées. Dans la plupart des cas, nous souhaitons uniquement intercepter certaines d'entre elles.

Solution

Le *point d'action (pointcut)* est un autre concept POA essentiel, généralement donné sous forme d'une expression, qui nous permet de désigner certains points d'exécution du programme pour appliquer un greffon. Dans Spring AOP classique, les points d'action sont également déclarés comme des beans Spring en utilisant des classes de point d'action.

Spring propose plusieurs classes de point d'action pour désigner les points d'exécution d'un programme. Il nous suffit de déclarer des beans de ces types dans le fichier de configuration des beans pour définir des points d'action. Cependant, si les classes intégrées ne répondent pas à nos besoins, nous pouvons écrire les nôtres en étendant `StaticMethodMatcherPointcut` ou `DynamicMethodMatcherPointcut`. La première désigne des points d'exécution d'après les informations statiques de classe et de méthode, tandis que la seconde les trouve en utilisant également des valeurs dynamiques des arguments.

Explications

Point d'action avec nom de méthode

Pour intercepter une seule méthode, nous pouvons employer NameMatchMethodPointcut de manière à définir statiquement la méthode par son nom. Dans la propriété mappedName, nous donnons un nom de méthode précis ou utilisons une expression avec des caractères génériques.

```
<bean id="methodNamePointcut"
      class="org.springframework.aop.support.NameMatchMethodPointcut">
    <property name="mappedName" value="add" />
</bean>
```

Un point d'action doit être associé à un greffon pour indiquer où celui-ci doit s'appliquer. Cette association se nomme *conseiller (advisor)* dans Spring AOP classique. La classe DefaultPointcutAdvisor sert uniquement à associer un point d'action et un greffon. Pour appliquer un conseiller à un proxy, nous procédons comme pour un greffon.

```
<beans ...>
  ...
  <bean id="methodNameAdvisor"
        class="org.springframework.aop.support.DefaultPointcutAdvisor">
    <property name="pointcut" ref="methodNamePointcut" />
    <property name="advice" ref="loggingAroundAdvice" />
  </bean>

  <bean id="arithmeticCalculatorProxy"
        class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="arithmeticCalculator" />
    <property name="interceptorNames">
      <list>
        <value>methodNameAdvisor</value>
      </list>
    </property>
  </bean>
</beans>
```

Si nous souhaitons désigner plusieurs méthodes en utilisant un point d'action avec nom de méthode, nous devons les indiquer dans la propriété mappedNames de type java.util.List.

```
<bean id="methodNamePointcut"
      class="org.springframework.aop.support.NameMatchMethodPointcut">
    <property name="mappedNames">
      <list>
        <value>add</value>
        <value>sub</value>
      </list>
    </property>
  </bean>
```

Pour chaque type de point d'action commun, Spring fournit également une classe de conseiller qui permet de déclarer un conseiller en une opération. Pour un point d'action `NameMatchMethodPointcut`, la classe de conseiller se nomme `NameMatchMethodPointcutAdvisor`.

```
<bean id="methodNameAdvisor"
      class="org.springframework.aop.support.NameMatchMethodPointcutAdvisor">
    <property name="mappedNames">
      <list>
        <value>add</value>
        <value>sub</value>
      </list>
    </property>
    <property name="advice" ref="loggingAroundAdvice" />
  </bean>
```

Point d'action avec expression régulière

Outre la désignation des méthodes fondée sur leur nom, nous pouvons également les cibler par une expression régulière. La classe `RegexpMethodPointcutAdvisor` nous permet d'indiquer une ou plusieurs expressions régulières. Par exemple, l'expression suivante correspond aux méthodes dont le nom contient le terme `mul` ou `div` :

```
<beans ...>
  ...
  <bean id="regexpAdvisor"
        class="org.springframework.aop.support.RegexpMethodPointcutAdvisor">
    <property name="patterns">
      <list>
        <value>.*mul.*</value>
        <value>.*div.*</value>
      </list>
    </property>
    <property name="advice" ref="loggingAroundAdvice" />
  </bean>

  <bean id="arithmeticCalculatorProxy"
        class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="arithmeticCalculator" />
    <property name="interceptorNames">
      <list>
        <value>methodNameAdvisor</value>
        <value>regexpAdvisor</value>
      </list>
    </property>
  </bean>
</beans>
```

Point d'action avec expression AspectJ

Le framework AspectJ définit un puissant langage d'expression des points d'action. Nous pouvons employer un `AspectJExpressionPointcutAdvisor` pour désigner une méthode à l'aide d'une expression AspectJ de point d'action. Par exemple, l'expression

suivante correspond à toutes les méthodes dont le nom contient le terme To. Le langage d'AspectJ pour les points d'action sera présenté au Chapitre 6, qui traite de Spring 2.x AOP. Pour une description plus précise de ce langage, consultez la documentation d'AspectJ.

INFO

Pour définir des points d'action avec des expressions AspectJ, vous devez inclure le fichier aspectjweaver.jar (situé dans le répertoire lib/aspectj de l'installation de Spring) dans le chemin d'accès aux classes.

```
<beans ...>
  ...
  <bean id="aspectjAdvisor" class="org.springframework.aop.aspectj.<span style="color: #0000ff;">➥
        AspectJExpressionPointcutAdvisor">
    <property name="expression">
        <value>execution(* *.*To*(..))</value>
    </property>
    <property name="advice">
        <ref bean="loggingAroundAdvice" />
    </property>
  </bean>

  <bean id="unitCalculatorProxy"
        class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="unitCalculator" />
    <property name="interceptorNames">
        <list>
            <value>aspectjAdvisor</value>
        </list>
    </property>
  </bean>
</beans>
```

5.5 Créer automatiquement des proxies pour les beans

Problème

Avec Spring AOP classique, nous devons créer un proxy pour chaque bean intercepté et le lier au bean cible. Notre fichier de configuration des beans déclare donc un grand nombre de beans de proxies.

Solution

Spring propose une fonction de *création automatique des proxies* pour créer automatiquement les proxies de nos beans. Elle est implémentée sous forme d'un postprocessseur de beans qui remplace les beans cibles par les proxies créés. Grâce à cette fonctionnalité, il devient inutile de créer manuellement les proxies avec ProxyFactoryBean.

Explications

Spring fournit deux implémentations pour la création automatique des proxies. La première, `BeanNameAutoProxyCreator`, demande la configuration d'une liste d'expressions de noms de beans. Dans chaque expression de nom de bean, nous pouvons placer des caractères génériques pour désigner un groupe de beans. Par exemple, le créateur suivant crée des proxies pour les beans dont le nom se termine par `Calculator`. Chaque proxy créé est visé par les conseillers indiqués dans le créateur automatique de proxies.

```
<beans ...>
    ...
    <bean id="arithmeticCalculator"
          class="com.apress.springrecipes.calculator.ArithmeticCalculatorImpl" />

    <bean id="unitCalculator"
          class="com.apress.springrecipes.calculator.UnitCalculatorImpl" />

    <bean class="org.springframework.aop.framework.autoproxy.<span style="color: red;">➥
          BeanNameAutoProxyCreator">
        <property name="beanNames">
            <list>
                <value>*Calculator</value>
            </list>
        </property>
        <property name="interceptorNames">
            <list>
                <value>methodNameAdvisor</value>
                <value>regexpAdvisor</value>
                <value>aspectjAdvisor</value>
            </list>
        </property>
    </bean>
</beans>
```

Dans la classe `Main`, nous pouvons simplement obtenir les beans d'après leur nom original même sans savoir qu'ils se trouvent derrière un proxy.

```
package com.apress.springrecipes.calculator;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        ArithmeticCalculator calculator1 =
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        ...
    }
}
```

```
    UnitCalculator calculator2 =  
        (UnitCalculator) context.getBean("unitCalculator");  
    ...  
}
```

DefaultAdvisorAutoProxyCreator est la seconde implémentation de la création automatique des proxies. Elle ne demande aucune configuration particulière. Elle vérifie automatiquement chaque bean avec chaque conseiller déclaré dans le conteneur IoC. Si l'un des beans est désigné par un point d'action du conseiller, DefaultAdvisorAutoProxyCreator crée automatiquement un proxy pour lui.

```
<bean class="org.springframework.aop.framework.autoproxy.DefaultAdvisorAutoProxyCreator" />
```

Toutefois, ce créateur automatique de proxies doit être employé avec prudence, car il peut viser des beans qui ne sont pas censés l'être.

5.6 En résumé

Dans ce chapitre, nous avons vu l'importance de la modularisation des préoccupations transversales dans une application. Ce type de fonctionnalité est souvent difficile à modulariser avec l'approche orientée objet classique. L'absence de modularisation des préoccupations transversales conduit aux problèmes de mélange de code et de dispersion de code.

Un proxy dynamique facilite la modularisation des préoccupations transversales, mais il demande un travail trop important de la part du développeur d'applications. Toutefois, il constitue l'une des technologies d'implémentation au cœur de Spring AOP.

La POA complète la POO en intervenant expressément dans la modularisation des préoccupations transversales. Elle définit des concepts de haut niveau, comme le greffon et le point d'action, pour permettre aux développeurs d'applications d'exprimer leurs préoccupations transversales.

Un greffon encapsule l'action qui doit être réalisée à un point d'exécution précis du programme. Spring AOP classique reconnaît quatre types de greffons : Before, After returning, After throwing et Around. Pour appliquer des greffons à un bean du conteneur Spring IoC, nous devons créer un proxy à l'aide de ProxyFactoryBean.

Un point d'action apparaît généralement sous forme d'une expression qui désigne certains points d'exécution du programme où sont appliqués des greffons. Dans Spring AOP classique, un point d'action est déclaré sous forme d'un bean Spring et associé à un greffon par un conseiller.

Lorsqu'on utilise Spring AOP classique, nous devons créer un proxy pour chaque bean contrôlé. Spring fournit des créateurs automatiques de proxies qui se chargent de créer automatiquement des proxies pour nos beans. Grâce à ces créateurs, il devient inutile de créer manuellement des proxies avec `ProxyFactoryBean`.

Le chapitre suivant détaille la nouvelle approche de Spring 2.x AOP et sa prise en charge du framework AspectJ.

6

Spring 2.x AOP et prise en charge d'AspectJ

Au sommaire de ce chapitre

- ✓ Activer la prise en charge des annotations AspectJ dans Spring
- ✓ Déclarer des aspects avec des annotations AspectJ
- ✓ Accéder aux informations du point de jonction
- ✓ Préciser la précédence des aspects
- ✓ Réutiliser des définitions de points d'action
- ✓ Écrire des expressions AspectJ de point d'action
- ✓ Introduire des comportements dans des beans
- ✓ Introduire des états dans des beans
- ✓ Déclarer des aspects avec des configurations XML
- ✓ Tisser des aspects AspectJ au chargement dans Spring
- ✓ Configurer des aspects AspectJ dans Spring
- ✓ Injecter des beans Spring dans des objets de domaine
- ✓ En résumé

Ce chapitre détaille l'utilisation de Spring 2.x AOP, ainsi que certains sujets avancés de la programmation orientée aspect, comme la priorité et l'introduction des greffons. L'emploi du framework Spring AOP a beaucoup évolué entre les versions 1.x et 2.x. Ce chapitre se focalise sur la nouvelle approche de Spring AOP, qui permet d'écrire des aspects plus puissants et offrant une meilleure compatibilité. Par ailleurs, il explique comment bénéficier du framework AspectJ dans les applications Spring.

Au chapitre précédent, nous avons étudié Spring AOP classique, qui se fonde sur un ensemble d'API propriétaires. Dans Spring version 2.x, il est possible d'écrire les aspects sous forme de POJO, que ce soit avec des annotations AspectJ ou des configu-

rations XML dans le fichier de configuration des beans. Puisque ces deux solutions conduisent au même résultat, la majeure partie de ce chapitre se focalise sur les annotations AspectJ, tout en proposant des configurations XML à titre de comparaison.

Même si l'utilisation de Spring AOP a changé, la technologie au cœur de l'implémentation reste les proxies dynamiques. De plus, Spring AOP garantissant la rétrocompatibilité, nous pouvons utiliser dans Spring 2.x AOP les greffons, les points d'action et les créateurs automatiques de proxies développés pour Spring classique.

AspectJ étant aujourd'hui un framework AOP complet et répandu, Spring 2.x AOP prend en charge les aspects POJO écrits avec des annotations AspectJ. Puisque, dans le futur, les frameworks AOP seront de plus en plus nombreux à prendre en charge ces annotations, nos aspects AspectJ ont de fortes chances de pouvoir être réutilisés dans ces autres frameworks compatibles AspectJ.

Toutefois, l'utilisation des aspects AspectJ dans Spring 2.x AOP n'est pas totalement équivalente à utiliser le framework AspectJ. En effet, Spring AOP présente quelques limitations sur l'emploi des aspects AspectJ. Par exemple, les aspects ne peuvent être appliqués qu'aux beans déclarés dans le conteneur Spring IoC. Pour appliquer des aspects en dehors de ce contexte, il faut opter pour le framework AspectJ, qui sera présenté à la fin de ce chapitre.

Lorsque vous aurez lu ce chapitre, vous serez en mesure d'écrire des aspects POJO utilisables dans le framework Spring 2.x AOP. Vous pourrez également utiliser le framework AspectJ dans vos applications Spring.

6.1 Activer la prise en charge des annotations AspectJ dans Spring

Problème

Le framework AOP de Spring version 2.x prend en charge l'utilisation des aspects POJO fondés sur les annotations AspectJ. Mais, avant tout, il est nécessaire d'activer la prise en charge des annotations AspectJ dans le conteneur Spring IoC.

Solution

Pour activer la prise en charge des annotations AspectJ dans le conteneur Spring IoC, il suffit d'ajouter un élément `<aop:aspectj-autoproxy>` vide dans le fichier de configuration des beans. Ensuite, Spring crée automatiquement des proxies pour les beans qui correspondent aux aspects AspectJ.

Explications

Pour que la comparaison avec Spring AOP classique soit plus facile, nous réutilisons l'exemple des calculatrices du chapitre précédent. Nous avions alors défini les interfaces suivantes :

```
package com.apress.springrecipes.calculator;

public interface ArithmeticCalculator {

    public double add(double a, double b);
    public double sub(double a, double b);
    public double mul(double a, double b);
    public double div(double a, double b);
}

---

package com.apress.springrecipes.calculator;

public interface UnitCalculator {

    public double kilogramToPound(double kilogram);
    public double kilometerToMile(double kilometer);
}
```

Nous avions également fourni une implémentation de chaque interface, avec des instructions `println` pour signaler l'exécution des méthodes.

```
package com.apress.springrecipes.calculator;

public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    public double add(double a, double b) {
        double result = a + b;
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }

    public double sub(double a, double b) {
        double result = a - b;
        System.out.println(a + " - " + b + " = " + result);
        return result;
    }

    public double mul(double a, double b) {
        double result = a * b;
        System.out.println(a + " * " + b + " = " + result);
        return result;
    }

    public double div(double a, double b) {
        if (b == 0) {
            throw new IllegalArgumentException("Division par zéro");
        }
    }
}
```

```

        double result = a / b;
        System.out.println(a + " / " + b + " = " + result);
        return result;
    }
}

---

package com.apress.springrecipes.calculator;

public class UnitCalculatorImpl implements UnitCalculator {

    public double kilogramToPound(double kilogram) {
        double pound = kilogram * 2.2;
        System.out.println(kilogram + " kilogrammes = " + pound + " livres");
        return pound;
    }

    public double kilometerToMile(double kilometer) {
        double mile = kilometer * 0.62;
        System.out.println(kilometer + " kilomètres = " + mile + " miles");
        return mile;
    }
}

```

Pour activer la prise en charge des annotations AspectJ dans cette application, nous ajoutons simplement un élément XML vide, `<aop:aspectj-autoproxy>`, dans le fichier de configuration des beans, sans oublier la définition du schéma `aop` dans l'élément racine `<beans>`. Lorsque le conteneur Spring IoC détecte l'élément `<aop:aspectj-autoproxy>` dans le fichier de configuration, il crée automatiquement des proxies pour les beans qui correspondent à nos aspects AspectJ.

INFO

Pour utiliser des annotations AspectJ dans votre application Spring, vous devez inclure les bibliothèques AspectJ (`aspectjrt.jar` et `aspectjweaver.jar` situés dans le répertoire `lib/aspectj` de l'installation de Spring) et leurs dépendances (`asm-2.2.3.jar`, `asm-commons-2.2.3.jar` et `asm-util-2.2.3.jar` situés dans le répertoire `lib/asm`) dans le chemin d'accès aux classes.

```

<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-2.5.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <aop:aspectj-autoproxy />

    <bean id="arithmeticCalculator"
          class="com.apress.springrecipes.calculator.ArithmeticCalculatorImpl" />

```

```
<bean id="unitCalculator"
      class="com.apress.springrecipes.calculator.UnitCalculatorImpl" />
</beans>
```

Cet élément enregistre en réalité un `AnnotationAwareAspectJAutoProxyCreator`, qui fonctionne à la manière des créateurs automatiques de proxies décrits au Chapitre 5, comme `BeanNameAutoProxyCreator` et `DefaultAdvisorAutoProxyCreator`.

6.2 Déclarer des aspects avec des annotations AspectJ

Problème

Depuis sa fusion avec AspectWerkz à partir de la version 5, AspectJ accepte que les aspects soient écrits sous forme de POJO marqués avec un ensemble d'annotations AspectJ. Si ces aspects sont pris en charge par le framework Spring 2.x AOP, ils doivent être enregistrés dans le conteneur Spring IoC pour être actifs.

Solution

Pour enregistrer des aspects AspectJ dans Spring, nous les déclarons simplement comme des instances de beans dans le conteneur IoC. Dès lors qu'AspectJ a été activé dans le conteneur Spring IoC, celui-ci crée des proxies pour les beans qui correspondent aux aspects AspectJ déclarés.

Avec les annotations AspectJ, un aspect n'est rien d'autre qu'une classe Java marquée par `@Aspect`. Un greffon est une simple méthode Java marquée par l'une des annotations de greffons : `@Before`, `@After`, `@AfterReturning`, `@AfterThrowing` et `@Around`.

Explications

Greffon Before

Pour créer un greffon *Before* qui traite les préoccupations transversales avant certains points d'exécution d'un programme, nous utilisons l'annotation `@Before` et incluons l'expression de point d'action dans la valeur de l'annotation.

```
package com.apress.springrecipes.calculator;

import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {

    private Log log = LogFactory.getLog(this.getClass());
```

```

@Before("execution(* ArithmeticCalculator.add(..))")
public void logBefore() {
    log.info("Début de la méthode add()");
}
}

```

Cette expression de point d'action représente l'exécution de la méthode add() de l'interface ArithmeticCalculator. L'astérisque qui précède l'expression correspond à n'importe quel modificateur (public, protected et private) et à n'importe quel type de valeur de retour. Les deux points dans la liste des arguments indiquent un nombre quelconque d'arguments.

Pour enregistrer cet aspect, nous en déclarons une instance de bean dans le conteneur IoC. Le bean de l'aspect peut être anonyme s'il n'est pas référencé par d'autres beans.

```

<beans ...>
    ...
    <bean class="com.apress.springrecipes.calculator.➥
        CalculatorLoggingAspect" />
</beans>

```

La classe Main suivante nous permet de tester notre aspect :

```

package com.apress.springrecipes.calculator;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        arithmeticCalculator.add(1, 2);
        arithmeticCalculator.sub(4, 3);
        arithmeticCalculator.mul(2, 3);
        arithmeticCalculator.div(4, 2);

        UnitCalculator unitCalculator =
            (UnitCalculator) context.getBean("unitCalculator");
        unitCalculator.kilogramToPound(10);
        unitCalculator.kilometerToMile(5);
    }
}

```

Les points d'exécution qui correspondent à un point d'action sont appelés *points de jonction*. Un point d'action est donc une expression qui permet de désigner un ensemble de points de jonction, tandis qu'un greffon représente l'action effectuée à un point de jonction précis.

Pour que notre greffon accède aux détails du point de jonction courant, comme le nom de la méthode et la valeur des arguments, nous déclarons un argument de type `JoinPoint` dans la méthode de greffon. Ainsi, nous pouvons écrire un point d'action qui désigne toutes les méthodes, simplement en remplaçant les noms de classe et de méthode par des caractères génériques.

```
package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Before("execution(* *.*(..))")
    public void logBefore(JoinPoint joinPoint) {
        log.info("Méthode " + joinPoint.getSignature().getName()
            + "() invoquée avec " + Arrays.toString(joinPoint.getArgs()));
    }
}
```

Greffon After

Un greffon *After* est exécuté après la fin d'un point de jonction, dès que celui-ci retourne un résultat ou lance une exception. Le greffon *After* suivant consigne la fin d'une méthode de la calculatrice. Un aspect peut inclure un ou plusieurs greffons.

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @After("execution(* *.*(..))")
    public void logAfter(JoinPoint joinPoint) {
        log.info("Méthode " + joinPoint.getSignature().getName()
            + "() terminée");
    }
}
```

Greffon After returning

Un greffon *After* est exécuté que le point de jonction retourne de manière normale ou lance une exception. Pour que la journalisation se fasse uniquement lors d'un retour normal, nous remplaçons le greffon *After* par un greffon *After returning*.

```

package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterReturning("execution(* *.*(..))")
    public void logAfterReturning(JoinPoint joinPoint) {
        log.info("Méthode " + joinPoint.getSignature().getName()
            + "() terminée");
    }
}

```

Dans un greffon After returning, nous pouvons accéder à la valeur de retour d'un point de jonction en ajoutant un attribut `returning` à l'annotation `@AfterReturning`. La valeur de cet attribut doit être le nom de l'argument de la méthode de greffon dans lequel la valeur de retour sera placée par Spring AOP au moment de l'exécution. Il ne faut pas oublier d'ajouter un argument de ce nom à la méthode de greffon. Par ailleurs, l'expression du point d'action doit alors être donnée dans l'attribut `pointcut`.

```

package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterReturning;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterReturning(
        pointcut = "execution(* *.*(..))",
        returning = "result")
    public void logAfterReturning(JoinPoint joinPoint, Object result) {
        log.info("Méthode " + joinPoint.getSignature().getName()
            + "() terminée par " + result);
    }
}

```

Greffon After throwing

Un greffon *After throwing* est exécuté uniquement lorsqu'une exception est lancée par un point de jonction.

```

package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {

```

```
...
@AfterThrowing("execution(* *.*(..))")
public void logAfterThrowing(JoinPoint joinPoint) {
    log.error("Exception lancée dans "
              + joinPoint.getSignature().getName() + "()");
}
}
```

Il est possible d'accéder à l'exception lancée par le point de jonction en ajoutant un attribut throwing à l'annotation @AfterThrowing. Dans le langage Java, Throwable est la superclasse de toutes les erreurs et exceptions. Par conséquent, le greffon suivant intercepte toutes les erreurs et exceptions générées par les points de jonction :

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterThrowing(
        pointcut = "execution(* *.*(..))",
        throwing = "e")
    public void logAfterThrowing(JoinPoint joinPoint, Throwable e) {
        log.error("Exception " + e + " lancée dans "
                  + joinPoint.getSignature().getName() + "()");
    }
}
```

Si nous ne nous intéressons qu'à un type d'exception précis, nous pouvons le préciser dans l'argument. Le greffon est alors exécuté uniquement lorsque des exceptions de type compatible (c'est-à-dire ce type et ses sous-types) sont lancées.

```
package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.AfterThrowing;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @AfterThrowing(
        pointcut = "execution(* *.*(..))",
        throwing = "e")
    public void logAfterThrowing(JoinPoint joinPoint,
                                IllegalArgumentException e) {
        log.error("Argument invalide " + Arrays.toString(joinPoint.getArgs())
                  + " dans " + joinPoint.getSignature().getName() + "()");
    }
}
```

Greffon Around

Le greffon *Around* est le dernier type de greffon. Il est le plus puissant de tous. Il a un contrôle total sur l'exécution de la méthode et permet de combiner toutes les actions des greffons précédents en un seul. Nous pouvons même contrôler le moment d'exécution du point de jonction d'origine et s'il est invoqué.

Le greffon *Around* suivant combine les greffons *Before*, *After returning* et *After throwing* développés précédemment. Pour ce type de greffon, l'argument du point de jonction doit être de type *ProceedingJoinPoint*. Il s'agit d'une sous-interface de *JoinPoint* qui nous permet de contrôler la poursuite de l'exécution dans le point de jonction d'origine.

```
package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Around("execution(* *.*(..))")
    public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
        log.info("Méthode " + joinPoint.getSignature().getName()
            + "() invoquée avec " + Arrays.toString(joinPoint.getArgs()));
        try {
            Object result = joinPoint.proceed();
            log.info("Méthode " + joinPoint.getSignature().getName()
                + "() terminée par " + result);
            return result;
        } catch (IllegalArgumentException e) {
            log.error("Argument invalide "
                + Arrays.toString(joinPoint.getArgs()) + " dans "
                + joinPoint.getSignature().getName() + "()");
            throw e;
        }
    }
}
```

Le greffon de type *Around* est très puissant et très souple puisque nous pouvons même modifier les valeurs initiales des arguments et la valeur de retour. L'utilisation d'un tel greffon demande une grande prudence, car il est très facile d'oublier d'exécuter le point de jonction d'origine.

ASTUCE

Pour choisir le type de greffon, la règle classique consiste à utiliser la version la moins puissante qui répond aux exigences.

6.3 Accéder aux informations du point de jonction

Problème

Dans la POA, un greffon est appliqué aux différents points d'exécution d'un programme, également appelés points de jonction. Pour qu'un greffon effectue l'opération adéquate, il a souvent besoin d'informations détaillées sur les points de jonction.

Solution

Un greffon peut accéder aux informations du point de jonction courant en déclarant un argument de type `org.aspectj.lang.JoinPoint` dans la méthode de greffon.

Explications

Par exemple, nous pouvons accéder aux informations du point de jonction à l'aide du greffon suivant. Parmi ces informations, nous trouvons le type du point de jonction (uniquement `method-execution` dans Spring AOP), la signature de la méthode (type déclarant et nom) et les valeurs des arguments, ainsi que l'objet cible et l'objet proxy.

```
package com.apress.springrecipes.calculator;
...
import java.util.Arrays;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Before("execution(* *.*(..))")
    public void logJoinPoint(JoinPoint joinPoint) {
        log.info("Type du point de jonction : "
            + joinPoint.getKind());
        log.info("Type déclarant : "
            + joinPoint.getSignature().getDeclaringTypeName());
        log.info("Nom de la méthode : "
            + joinPoint.getSignature().getName());
        log.info("Arguments : "
            + Arrays.toString(joinPoint.getArgs()));
        log.info("Classe cible : "
            + joinPoint.getTarget().getClass().getName());
        log.info("Classe this : "
            + joinPoint.getThis().getClass().getName());
    }
}
```

Le bean original enveloppé par un proxy est appelé *objet cible*, tandis que l'objet proxy est appelé *objet this*. Nous pouvons y accéder à l'aide des méthodes `getTarget()` et `getThis()` du point de jonction. La sortie suivante montre que les classes de ces deux objets sont différentes :

```
Type du point de jonction : method-execution
Type déclarant : com.apress.springrecipes.calculator.ArithmeticCalculator
Nom de la méthode : add
Arguments : [1.0, 2.0]
Classe cible : com.apress.springrecipes.calculator.ArithmeticCalculatorImpl
Classe this : $Proxy6
```

6.4 Préciser la précédence des aspects

Problème

Lorsque plusieurs aspects sont appliqués au même point de jonction, leur précédence est indéfinie, sauf si elle a été fixée de manière explicite.

Solution

La précédence des aspects peut être définie, soit en implémentant l'interface Ordered, soit en utilisant l'annotation @Order.

Explications

Supposons que nous ayons écrit un autre aspect pour valider les arguments de la calculatrice. Cet aspect ne contient qu'un seul seul greffon Before.

```
package com.apress.springrecipes.calculator;

import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorValidationAspect {

    @Before("execution(* *.*(double, double))")
    public void validateBefore(JoinPoint joinPoint) {
        for (Object arg : joinPoint.getArgs()) {
            validate((Double) arg);
        }
    }

    private void validate(double a) {
        if (a < 0) {
            throw new IllegalArgumentException("Nombres positifs attendus");
        }
    }
}
```

Pour enregistrer cet aspect dans Spring, nous en déclarons simplement une instance dans le fichier de configuration des beans.

```
<beans ...>
    ...
    <bean class="com.apress.springrecipes.calculator.➥
        CalculatorLoggingAspect" />
```

```
<bean class="com.apress.springrecipes.calculator.&gt;
    CalculatorValidationAspect" />
</beans>
```

À présent, la précédence des aspects est indéfinie. Il faut savoir qu'elle ne dépend pas de l'ordre de déclaration des beans. Par conséquent, pour qu'elle soit explicite, les deux aspects doivent implémenter l'interface Ordered. Plus la valeur renournée par la méthode getOrder() est faible, plus la priorité est élevée. Si nous préférons que l'aspect de validation soit appliqué en premier, sa méthode getOrder() doit donc retourner une valeur plus faible que celle de l'aspect de journalisation.

```
package com.apress.springrecipes.calculator;
...
import org.springframework.core.Ordered;

@Aspect
public class CalculatorValidationAspect implements Ordered {
    ...
    public int getOrder() {
        return 0;
    }
}

---

package com.apress.springrecipes.calculator;
...
import org.springframework.core.Ordered;

@Aspect
public class CalculatorLoggingAspect implements Ordered {
    ...
    public int getOrder() {
        return 1;
    }
}
```

Pour préciser la précédence, nous pouvons également utiliser l'annotation @Order. Le numéro d'ordre doit être indiqué dans la valeur de l'annotation.

```
package com.apress.springrecipes.calculator;
...
import org.springframework.core.annotation.Order;

@Aspect
@Order(0)
public class CalculatorValidationAspect {
    ...
}

---

package com.apress.springrecipes.calculator;
...
import org.springframework.core.annotation.Order;
```

```
@Aspect  
 @Order(1)  
 public class CalculatorLoggingAspect {  
     ...  
 }
```

6.5 Réutiliser des définitions de points d'action

Problème

Lorsque nous écrivons des aspects AspectJ, nous pouvons directement incorporer une expression de point d'action dans une annotation de greffon. Toutefois, la même expression peut se retrouver dans de multiples greffons.

Solution

Comme bien d'autres mises en œuvre de la POA, AspectJ nous permet de définir un point d'action de manière indépendante et de le réutiliser dans plusieurs greffons.

Explications

Dans un aspect AspectJ, un point d'action peut être déclaré comme une simple méthode avec l'annotation `@Pointcut`. Le corps de cette méthode est généralement vide, car il est peu raisonnable de mélanger la définition d'un point d'action et la logique applicative. Le modificateur d'accès d'une méthode de point d'action fixe également la visibilité de celui-ci. D'autres greffons peuvent faire référence à ce point d'action en utilisant le nom de la méthode.

```
package com.apress.springrecipes.calculator;  
...  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class CalculatorLoggingAspect {  
    ...  
    @Pointcut("execution(* *.*(..))")  
    private void loggingOperation() {}  
  
    @Before("loggingOperation()")  
    public void logBefore(JoinPoint joinPoint) {  
        ...  
    }  
  
    @AfterReturning(  
        pointcut = "loggingOperation()",  
        returning = "result")  
    public void logAfterReturning(JoinPoint joinPoint, Object result) {  
        ...  
    }
```

```
@AfterThrowing(  
    pointcut = "loggingOperation()",  
    throwing = "e")  
public void logAfterThrowing(JoinPoint joinPoint,  
    IllegalArgumentException e) {  
    ...  
}  
  
@Around("loggingOperation()")  
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {  
    ...  
}
```

Si des points d'action sont partagés entre plusieurs aspects, il est préférable de les centraliser dans une classe commune. Ils doivent alors être déclarés `public`.

```
package com.apress.springrecipes.calculator;  
  
import org.aspectj.lang.annotation.Aspect;  
import org.aspectj.lang.annotation.Pointcut;  
  
@Aspect  
public class CalculatorPointcuts {  
  
    @Pointcut("execution(* *.*(..))")  
    public void loggingOperation() {}  
}
```

Pour faire référence à ce point d'action, nous devons préciser le nom de classe. Si elle ne se trouve pas dans le même paquetage que l'aspect, il faut également ajouter le nom du paquetage.

```
package com.apress.springrecipes.calculator;  
...  
@Aspect  
public class CalculatorLoggingAspect {  
    ...  
    @Before("CalculatorPointcuts.loggingOperation()")  
    public void logBefore(JoinPoint joinPoint) {  
        ...  
    }  
  
    @AfterReturning(  
        pointcut = "CalculatorPointcuts.loggingOperation()",  
        returning = "result")  
    public void logAfterReturning(JoinPoint joinPoint, Object result) {  
        ...  
    }  
  
    @AfterThrowing(  
        pointcut = "CalculatorPointcuts.loggingOperation()",  
        throwing = "e")  
    public void logAfterThrowing(JoinPoint joinPoint,  
        IllegalArgumentException e) {  
        ...  
    }
```

```
@Around("CalculatorPointcuts.loggingOperation()")
public Object logAround(ProceedingJoinPoint joinPoint) throws Throwable {
    ...
}
```

6.6 Écrire des expressions AspectJ de point d'action

Problème

Les préoccupations transversales peuvent se placer à différents points d'exécution du programme, appelés points de jonction. En raison de la diversité des points de jonction, nous avons besoin d'un puissant langage d'expression qui nous permet de les désigner.

Solution

Le langage de point d'action d'AspectJ est un langage puissant qui permet de désigner différentes sortes de points de jonction. Toutefois, Spring AOP ne reconnaît que les points de jonction de type exécution de méthode pour les beans déclarés dans son conteneur IoC. C'est pourquoi nous présentons ici uniquement les expressions de point d'action prises en charge par Spring AOP. Pour une description complète du langage de point d'action d'AspectJ, consultez le guide de programmation AspectJ disponible sur le site <http://www.eclipse.org/aspectj/>.

Spring 2.x AOP utilise ce langage pour la définition des points d'action. En réalité, il interprète les expressions de point d'action au moment de l'exécution en utilisant une bibliothèque d'AspectJ.

Lors de l'écriture des expressions AspectJ de point d'action pour Spring AOP, il ne faut pas oublier que ce dernier prend uniquement en charge les points de jonction de type exécution de méthode pour les beans enregistrés dans son conteneur IoC. Si nous employons une expression de point d'action en dehors de ce contexte, une exception `IllegalArgumentException` est lancée.

Explications

Motifs de signature de méthode

Les expressions de point d'action les plus utilisées sont celles qui désignent des méthodes par leur signature. Par exemple, l'expression de point d'action suivante correspond à toutes les méthodes déclarées dans l'interface `ArithmeticCalculator`. L'astérisque de début correspond à n'importe quel modificateur (`public`, `protected` et `private`) et à n'importe quel type de valeur de retour. Les deux points dans la liste des arguments correspondent à un nombre quelconque d'arguments.

```
execution(* com.apress.springrecipes.calculator.ArithmeticCalculator.*(..))
```

Le nom du paquetage est facultatif si la classe ou l'interface cible se trouve dans le même paquetage que l'aspect.

```
execution(* ArithmeticCalculator.*(..))
```

L'expression de point d'action suivante désigne toutes les méthodes publiques déclarées dans l'interface `ArithmeticCalculator`.

```
execution(public * ArithmeticCalculator.*(..))
```

Nous pouvons également appliquer une restriction sur le type de la valeur de retour. Par exemple, le point d'action suivant correspond à toutes les méthodes qui retournent une valeur de type `double`.

```
execution(public double ArithmeticCalculator.*(..))
```

La liste des arguments des méthodes peut également servir de contrainte. Par exemple, le point d'action suivant correspond à toutes les méthodes dont le premier argument est de type `double`. Les deux points indiquent que le nombre d'arguments restants est variable.

```
execution(public double ArithmeticCalculator.*(double, ..))
```

Nous pouvons également préciser le type de tous les arguments de la méthode.

```
execution(public double ArithmeticCalculator.*(double, double))
```

Bien que le langage de point d'action d'AspectJ soit performant pour la correspondance des points de jonction, il est possible que nous ne soyons pas en mesure de trouver des caractéristiques communes (par exemple modificateur, type de la valeur de retour, motif pour les noms de méthodes ou arguments) aux méthodes que nous souhaitons désigner. Dans ce cas, nous pouvons envisager de les marquer par notre propre annotation. Par exemple, nous pouvons définir l'annotation suivante qui s'applique au niveau de la méthode et du type.

```
package com.apress.springrecipes.calculator;

import java.lang.annotation.Documented;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Target( { ElementType.METHOD, ElementType.TYPE } )
@Retention(RetentionPolicy.RUNTIME)
@Documented
public @interface LoggingRequired {
}
```

Ensuite, nous marquons avec cette annotation toutes les méthodes qui ont besoin d'une journalisation. Les annotations doivent être ajoutées à la classe d'implémentation, non à l'interface, car elles ne sont pas héritées.

```
package com.apress.springrecipes.calculator;

public class ArithmeticCalculatorImpl implements ArithmeticCalculator {

    @LoggingRequired
    public double add(double a, double b) {
        ...
    }

    @LoggingRequired
    public double sub(double a, double b) {
        ...
    }

    @LoggingRequired
    public double mul(double a, double b) {
        ...
    }

    @LoggingRequired
    public double div(double a, double b) {
        ...
    }
}
```

À présent, nous pouvons écrire une expression de point d'action qui correspond à toutes les méthodes marquées par cette annotation `@LoggingRequired`.

```
@annotation(com.apress.springrecipes.calculator.LoggingRequired)
```

Motifs de signature de type

Une autre forme de point d'action désigne tous les points de jonction ayant certains types. Lorsqu'ils sont appliqués à Spring AOP, la portée de ces points d'action est réduite afin de correspondre à toutes les exécutions de méthode au sein des types indiqués. Par exemple, le point d'action suivant correspond à tous les points de jonction de type exécution de méthode au sein du paquetage `com.apress.springrecipes.calculator`.

```
within(com.apress.springrecipes.calculator.*)
```

Pour désigner les points de jonction au sein d'un paquetage et ses sous-paquetages, nous ajoutons un point supplémentaire avant le caractère générique.

```
within(com.apress.springrecipes.calculator..*)
```

L'expression de point d'action suivante correspond à tous les points de jonction de type exécution de méthode au sein d'une classe précise.

```
within(com.apress.springrecipes.calculator.ArithmeticCalculatorImpl)
```

À nouveau, si la classe cible se trouve dans le paquetage de l’aspect, le nom de paquetage peut être omis.

```
within(ArithmetricCalculatorImpl)
```

En ajoutant un signe plus, nous pouvons désigner les points de jonction de type exécution de méthode à l’intérieur de toutes les classes qui implémentent l’interface ArithmetricCalculator.

```
within(ArithmetricCalculator+)
```

Notre annotation personnalisée @LoggingRequired peut également être appliquée au niveau de la classe.

```
package com.apress.springrecipes.calculator;  
  
@LoggingRequired  
public class ArithmetricCalculatorImpl implements ArithmetricCalculator {  
    ...  
}
```

Nous pouvons alors désigner les points de jonction qui ont été annotés avec @LoggingRequired à l’intérieur des classes.

```
@within(com.apress.springrecipes.calculator.LoggingRequired)
```

Motifs de nom de bean

Spring 2.5 prend en charge un nouveau type de point d’action utilisé pour la correspondance avec des noms de beans. Par exemple, l’expression de point d’action suivante désigne tous les beans dont le nom se termine par Calculator.

```
bean(*Calculator)
```

ATTENTION

Ce type de point d’action n’est reconnu que dans les configurations XML de Spring AOP, non dans les annotations AspectJ.

Combiner des expressions de point d’action

Dans AspectJ, il est possible de combiner des expressions de point d’action à l’aide des opérateurs && (et), || (ou) et ! (non). Par exemple, le point d’action suivant désigne les points de jonction au sein des classes qui implémentent l’interface ArithmetricCalculator ou UnitCalculator :

```
within(ArithmetricCalculator+) || within(UnitCalculator+)
```

Les opérandes de ces opérateurs sont des expressions de point d’action ou des références à d’autres points d’action.

```

package com.apress.springrecipes.calculator;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

@Aspect
public class CalculatorPointcuts {

    @Pointcut("within(ArithmeticCalculator+)")
    public void arithmeticOperation() {}

    @Pointcut("within(UnitCalculator+)")
    public void unitOperation() {}

    @Pointcut("arithmeticOperation() || unitOperation()")
    public void loggingOperation() {}
}

```

Déclarer des paramètres de point d'action

Le mécanisme de réflexion est une solution pour accéder aux informations d'un point de jonction (c'est-à-dire *via* un argument de type `org.aspectj.lang.JoinPoint` dans la méthode de greffon). Il est également possible d'accéder à ces informations de manière déclarative en employant certaines formes d'expressions de point d'action. Par exemple, les expressions `target()` et `args()` représentent l'objet cible et les valeurs des arguments du point de jonction courant et les exposent sous forme de paramètres de point d'action. Ces paramètres sont passés à la méthode de greffon au travers d'arguments de mêmes noms.

```

package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Before("execution(* .*(..)) && target(target) && args(a,b)")
    public void logParameter(Object target, double a, double b) {
        log.info("Classe cible : " + target.getClass().getName());
        log.info("Arguments : " + a + ", " + b);
    }
}

```

Lorsqu'on déclare un point d'action indépendant qui expose des paramètres, il faut également les inclure dans la liste des arguments de la méthode de point d'action.

```

package com.apress.springrecipes.calculator;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Pointcut;

```

```
@Aspect
public class CalculatorPointcuts {
    ...
    @Pointcut("execution(* *.*(..)) && target(target) && args(a,b)")
    public void parameterPointcut(Object target, double a, double b) {}
}
```

Tout greffon qui fait référence à ce point d'action paramétré peut accéder aux paramètres par l'intermédiaire des arguments de méthode éponymes.

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
public class CalculatorLoggingAspect {
    ...
    @Before("CalculatorPointcuts.parameterPointcut(target, a, b)")
    public void logParameter(Object target, double a, double b) {
        log.info("Classe cible : " + target.getClass().getName());
        log.info("Arguments : " + a + ", " + b);
    }
}
```

6.7 Introduire des comportements dans des beans

Problème

Il arrive parfois qu'un ensemble de classes partagent un comportement commun. En programmation orientée objet, elles étendent alors la même classe de base ou implémentent la même interface. Cette question est en réalité une préoccupation transversale qui peut être modularisée avec la programmation orientée aspect.

Par ailleurs, en raison du mécanisme d'héritage simple de Java, une classe ne peut dériver que d'une seule classe de base. Il est donc impossible d'hériter de comportements provenant de multiples classes d'implémentation en même temps.

Solution

Dans la POA, l'*introduction* est un type de greffon particulier. Il permet à nos objets d'implémenter dynamiquement une interface en fournissant une classe d'implémentation de cette interface. Cela revient à étendre la classe d'implémentation d'un objet à l'exécution.

Par ailleurs, nous pouvons introduire simultanément dans nos objets plusieurs interfaces avec plusieurs classes d'implémentation. Nous obtenons ainsi l'équivalent d'un héritage multiple.

Explications

Supposons que deux interfaces, MaxCalculator et MinCalculator, définissent les opérations `max()` et `min()`.

```
package com.apress.springrecipes.calculator;

public interface MaxCalculator {

    public double max(double a, double b);
}

---

package com.apress.springrecipes.calculator;

public interface MinCalculator {

    public double min(double a, double b);
}
```

L'implémentation de chaque interface utilise des instructions `println` pour signaler l'exécution des méthodes.

```
package com.apress.springrecipes.calculator;

public class MaxCalculatorImpl implements MaxCalculator {

    public double max(double a, double b) {
        double result = (a >= b) ? a : b;
        System.out.println("max(" + a + ", " + b + ") = " + result);
        return result;
    }
}

---

package com.apress.springrecipes.calculator;

public class MinCalculatorImpl implements MinCalculator {

    public double min(double a, double b) {
        double result = (a <= b) ? a : b;
        System.out.println("min(" + a + ", " + b + ") = " + result);
        return result;
    }
}
```

Supposons à présent que nous voulions qu'`ArithmeticCalculatorImpl` effectue également les opérations `max()` et `min()`. Puisque le langage Java ne reconnaît que l'héritage simple, la classe `ArithmeticCalculatorImpl` ne peut pas dériver à la fois de `MaxCalculatorImpl` et de `MinCalculatorImpl`. La seule solution est d'étendre l'une des classes (par exemple `MaxCalculatorImpl`) et d'implémenter une autre interface (par exemple `MinCalculator`), que ce soit en copiant le code d'implémentation ou en déléguant le

traitement à la classe d'implémentation réelle. Dans tous les cas, nous devons répéter les déclarations de méthodes.

Grâce à l'introduction, nous pouvons faire en sorte qu'`ArithmeticCalculatorImpl` implémente dynamiquement les deux interfaces `MaxCalculator` et `MinCalculator` en s'appuyant sur les classes d'implémentation `MaxCalculatorImpl` et `MinCalculatorImpl`. Cela revient à dériver à la fois de `MaxCalculatorImpl` et de `MinCalculatorImpl`. Le grand intérêt de l'introduction est qu'il est inutile de modifier la classe `ArithmeticCalculatorImpl` pour ajouter de nouvelles méthodes. Autrement dit, nous pouvons introduire des méthodes dans nos classes existantes sans même disposer du code source.

ASTUCE

Vous vous demandez peut-être comment une introduction peut parvenir à cela dans Spring AOP. La solution est apportée par un *proxy dynamique*. Vous le savez, il est possible de préciser les interfaces implémentées par un proxy dynamique. L'introduction fonctionne en ajoutant une interface (par exemple `MaxCalculator`) au proxy dynamique. Lorsque les méthodes déclarées dans cette interface sont invoquées sur l'objet proxy, celui-ci délègue les appels à la classe d'implémentation sous-jacente (par exemple `MaxCalculatorImpl`).

Tout comme les greffons, les introductions doivent être déclarées dans un aspect. Pour cela, nous pouvons créer un nouvel aspect ou réutiliser un aspect existant. Dans l'aspect suivant, nous déclarons une introduction en marquant un champ quelconque avec l'annotation `@DeclareParents`.

```
package com.apress.springrecipes.calculator;

import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class CalculatorIntroduction {

    @DeclareParents(
        value = "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl",
        defaultImpl = MaxCalculatorImpl.class)
    public MaxCalculator maxCalculator;

    @DeclareParents(
        value = "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl",
        defaultImpl = MinCalculatorImpl.class)
    public MinCalculator minCalculator;
}
```

L'attribut `value` de l'annotation `@DeclareParents` précise les classes cibles pour l'introduction. L'interface à introduire est déterminée par le type du champ annoté.

Enfin, la classe d'implémentation utilisée pour cette nouvelle interface est indiquée par l'attribut `defaultImpl`.

Au travers de ces deux introductions, nous introduisons dynamiquement deux interfaces dans la classe `ArithmetricCalculatorImpl`. Nous pouvons également employer une expression AspectJ de correspondance de type dans l'attribut `value` de l'annotation `@DeclareParents` pour introduire une interface dans plusieurs classes. Pour finir, il ne faut pas oublier de déclarer une instance de l'aspect dans le fichier de configuration des beans.

```
<beans ...>
    ...
    <bean class="com.apress.springrecipes.calculator.CalculatorIntroduction" />
</beans>
```

Puisque nous avons introduit les deux interfaces `MaxCalculator` et `MinCalculator` dans notre calculatrice arithmétique, nous pouvons forcer son type dans l'interface correspondante pour effectuer les opérations `max()` et `min()`.

```
package com.apress.springrecipes.calculator;

public class Main {

    public static void main(String[] args) {
        ...
        ArithmetricCalculator arithmeticCalculator =
            (ArithmetricCalculator) context.getBean("arithmetricCalculator");
        ...
        MaxCalculator maxCalculator = (MaxCalculator) arithmeticCalculator;
        maxCalculator.max(1, 2);

        MinCalculator minCalculator = (MinCalculator) arithmeticCalculator;
        minCalculator.min(1, 2);
    }
}
```

6.8 Introduire des états dans des beans

Problème

Il arrive parfois que nous voulions ajouter de nouveaux états dans un groupe d'objets existants afin de suivre leur utilisation, comme le nombre d'appels, la date de dernière modification, etc. Cela ne poserait aucune difficulté si tous les objets avaient la même classe de base. En revanche, il est difficile d'ajouter de tels états dans des classes différentes qui ne font pas partie de la même hiérarchie.

Solution

Nous pouvons introduire une nouvelle interface dans nos objets avec une classe d'implémentation qui contient les champs d'état. Ensuite, nous pouvons écrire un autre greffon pour modifier l'état en fonction d'une condition particulière.

Explications

Supposons que nous voulions suivre le nombre d'invocations de chaque objet de calculatrice. Puisque les classes d'origine des calculatrices ne disposent d'aucun champ pour enregistrer ce compteur, nous l'introduisons avec Spring AOP. Tout d'abord, créons l'interface des opérations de comptage.

```
package com.apress.springrecipes.calculator;

public interface Counter {

    public void increase();
    public int getCount();
}
```

Nous écrivons ensuite une implémentation simple de cette interface. Le champ count de cette classe stocke la valeur du compteur.

```
package com.apress.springrecipes.calculator;

public class CounterImpl implements Counter {

    private int count;

    public void increase() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

Pour introduire l'interface Counter dans tous nos objets de calculatrice, en utilisant CounterImpl comme implémentation, nous écrivons l'introduction suivante avec une expression de correspondance de type qui désigne toutes les implementations de la calculatrice :

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.DeclareParents;

@Aspect
public class CalculatorIntroduction {
    ...
}
```

```
    @DeclareParents(
        value = "com.apress.springrecipes.calculator.*CalculatorImpl",
        defaultImpl = CounterImpl.class)
    public Counter counter;
}
```

Cette introduction introduit CounterImpl dans chacun de nos objets de calculatrice. Cependant, cela ne suffit pas pour suivre le comptage des invocations. Nous devons augmenter la valeur du compteur à chaque appel d'une méthode de calculatrice. Pour cela, nous écrivons un greffon After. Attention, nous devons obtenir l'objet *this*, non l'objet *cible*, car seul l'objet proxy implémente l'interface Counter !

```
package com.apress.springrecipes.calculator;
...
import org.aspectj.lang.annotation.After;
import org.aspectj.lang.annotation.Aspect;

@Aspect
public class CalculatorIntroduction {
    ...
    @After("execution(* com.apress.springrecipes.calculator.*Calculator.*(..))"
           + " && this(counter)")
    public void increaseCount(Counter counter) {
        counter.increase();
    }
}
```

Dans la classe Main, nous affichons la valeur du compteur pour chacun des objets de calculatrice en forçant leur type à Counter.

```
package com.apress.springrecipes.calculator;

public class Main {

    public static void main(String[] args) {
        ...
        ArithmeticCalculator arithmeticCalculator =
            (ArithmeticCalculator) context.getBean("arithmeticCalculator");
        ...

        UnitCalculator unitCalculator =
            (UnitCalculator) context.getBean("unitCalculator");
        ...

        Counter arithmeticCounter = (Counter) arithmeticCalculator;
        System.out.println(arithmeticCounter.getCount());

        Counter unitCounter = (Counter) unitCalculator;
        System.out.println(unitCounter.getCount());
    }
}
```

6.9 Déclarer des aspects avec des configurations XML

Problème

La déclaration des aspects avec des annotations AspectJ convient parfaitement dans la majorité des cas. Toutefois, si nous utilisons une JVM version 1.4 ou antérieure, donc sans la prise en charge des annotations, ou si nous ne souhaitons pas que notre application dépende d'AspectJ, nous ne devons pas déclarer des aspects avec des annotations AspectJ.

Solution

Spring accepte la déclaration des aspects non seulement à l'aide d'annotations AspectJ, mais également à partir du fichier de configuration des beans. Ces déclarations se font en utilisant les éléments XML du schéma aop.

Dans une situation normale, une déclaration par annotation est préférable à une déclaration XML. Avec les annotations AspectJ, les aspects sont compatibles avec AspectJ, tandis que les configurations XML sont propres à Spring. Puisque de plus en plus de frameworks AOP sont compatibles avec AspectJ, les aspects écrits avec des annotations ont de plus fortes chances d'être réutilisés.

Explications

Pour activer la prise en charge des annotations AspectJ dans Spring, nous avons déjà ajouté un élément XML vide, `<aop:aspectj-autoproxy>`, dans notre fichier de configuration des beans. Pour déclarer des aspects en XML, cet élément est inutile et doit être supprimé afin que Spring AOP ignore les annotations AspectJ. En revanche, la définition du schéma aop doit être conservée dans l'élément racine `<beans>` car c'est dans ce schéma que sont définis tous les éléments XML pour la configuration POA.

```
<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-2.5.xsd
        http://www.springframework.org/schema/aop
        http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

    <!--
    <aop:aspectj-autoproxy />
    -->
    ...
</beans>
```

Déclarer des aspects

Dans le fichier de configuration des beans, toutes les configurations de Spring AOP doivent se trouver dans l'élément `<aop:config>`. Pour chaque aspect, nous ajoutons un élément `<aop:aspect>` pour faire référence à l'instance du bean qui implémente l'aspect. Par conséquent, nos beans d'aspect doivent avoir un identifiant, que l'on retrouve dans les éléments `<aop:aspect>`.

```

<beans ...>
    <aop:config>
        <aop:aspect id="loggingAspect" ref="calculatorLoggingAspect">
            </aop:aspect>

        <aop:aspect id="validationAspect" ref="calculatorValidationAspect">
            </aop:aspect>

        <aop:aspect id="introduction" ref="calculatorIntroduction">
            </aop:aspect>
    </aop:config>

    <bean id="calculatorLoggingAspect"
          class="com.apress.springrecipes.calculator.CalculatorLoggingAspect" />

    <bean id="calculatorValidationAspect"
          class="com.apress.springrecipes.calculator.➥
          CalculatorValidationAspect" />

    <bean id="calculatorIntroduction"
          class="com.apress.springrecipes.calculator.CalculatorIntroduction" />
    ...
</beans>

```

Déclarer des points d'action

Un point d'action se définit soit dans l'élément `<aop:aspect>`, soit directement sous l'élément `<aop:config>`. Dans le premier cas, le point d'action est visible uniquement par l'aspect déclarant. Dans le second cas, il s'agit d'une définition globale de point d'action, visible par tous les aspects.

Il ne faut pas oublier que, contrairement aux annotations AspectJ, les configurations POA de type XML ne permettent pas de faire référence à d'autres points d'action par le nom au sein d'une expression de point d'action. Autrement dit, nous devons copier l'expression du point d'action et l'incorporer directement.

```

<aop:config>
    <aop:pointcut id="loggingOperation" expression=
        "within(com.apress.springrecipes.calculator.ArithmeticCalculator+) || ➥
        within(com.apress.springrecipes.calculator.UnitCalculator+)" />

    <aop:pointcut id="validationOperation" expression=
        "within(com.apress.springrecipes.calculator.ArithmeticCalculator+) || ➥
        within(com.apress.springrecipes.calculator.UnitCalculator+)" />

```

```
...
```

```
</aop:config>
```

Avec les annotations AspectJ, il est possible de combiner deux expressions de point d'action à l'aide de l'opérateur `&&`. Malheureusement, le caractère `&` est utilisé pour les entités en XML. L'opérateur de point d'action `&&` est donc invalide dans un document XML et doit être remplacé par le mot-clé `and`.

Déclarer des greffons

Dans le schéma aop, il existe un élément XML pour chaque type de greffon. Un élément de greffon doit préciser un attribut `pointcut-ref`, pour faire référence à un point d'action, ou un attribut `pointcut`, pour incorporer directement une expression de point d'action. L'attribut `method` donne le nom de la méthode de greffon dans la classe d'aspect.

```
<aop:config>
...
<aop:aspect id="loggingAspect" ref="calculatorLoggingAspect">
    <aop:before pointcut-ref="loggingOperation"
        method="logBefore" />

    <aop:after-returning pointcut-ref="loggingOperation"
        returning="result" method="logAfterReturning" />

    <aop:after-throwing pointcut-ref="loggingOperation"
        throwing="e" method="logAfterThrowing" />

    <aop:around pointcut-ref="loggingOperation"
        method="logAround" />
</aop:aspect>

<aop:aspect id="validationAspect" ref="calculatorValidationAspect">
    <aop:before pointcut-ref="validationOperation"
        method="validateBefore" />
</aop:aspect>
</aop:config>
```

Déclarer des introductions

La déclaration d'une introduction dans un aspect se fait avec l'élément `<aop:declare-parents>`.

```
<aop:config>
...
<aop:aspect id="introduction" ref="calculatorIntroduction">
    <aop:declare-parents
        types-matching=
            "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl"
        implement-interface=
            "com.apress.springrecipes.calculator.MaxCalculator"
        default-impl=
            "com.apress.springrecipes.calculator.MaxCalculatorImpl" />
```

```
<aop:declare-parents
    types-matching=
        "com.apress.springrecipes.calculator.ArithmeticCalculatorImpl"
    implement-interface=
        "com.apress.springrecipes.calculator.MinCalculator"
    default-impl=
        "com.apress.springrecipes.calculator.MinCalculatorImpl" />

<aop:declare-parents
    types-matching=
        "com.apress.springrecipes.calculator.*CalculatorImpl"
    implement-interface=
        "com.apress.springrecipes.calculator.Counter"
    default-impl= "com.apress.springrecipes.calculator.CounterImpl" />

<aop:after pointcut=
    "execution(* com.apress.springrecipes.calculator.*Calculator.*(..)) &
     and this(counter)"
    method="increaseCount" />
</aop:aspect>
</aop:config>
```

6.10 Tisser des aspects AspectJ au chargement dans Spring

Problème

Le framework Spring AOP ne prend en charge que certains types de points d'action AspectJ et n'applique des aspects qu'aux beans déclarés dans le conteneur IoC. Si nous souhaitons employer d'autres types de points d'action ou appliquer nos aspects à des objets créés en dehors du conteneur Spring IoC, nous devons utiliser le framework AspectJ dans notre application Spring.

Solution

Le *tissage* (*weaving*) désigne le processus d'application des aspects aux objets cibles. Dans Spring AOP, le tissage se produit à l'exécution par l'intermédiaire des proxies dynamiques. En revanche, le framework AspectJ prend en charge le tissage au moment de la compilation et au moment du chargement.

Le tissage à la compilation d'AspectJ passe par un compilateur particulier nommé ajc. Il permet de tisser des aspects dans nos fichiers sources Java et de produire des fichiers binaires tissés. Il peut également tisser des aspects dans les fichiers de classe compilés ou dans les fichiers JAR. Ce processus est appelé tissage postcompilation. Nous pouvons effectuer le tissage à la compilation et le tissage postcompilation de nos classes avant de les déclarer dans le conteneur Spring IoC. Spring n'est alors pas impliqué dans le processus de tissage. Pour de plus amples informations concernant ces tissages, consultez la documentation d'AspectJ.

Le tissage *au chargement* (LTW, *Load-Time Weaving*) d'AspectJ se produit pendant la phase de chargement des classes cibles dans la JVM par un chargeur de classes. Pour qu'une classe soit tissée, il faut un chargeur de classe spécial capable d'intervenir au niveau de son bytecode. AspectJ et Spring fournissent des tisseurs au chargement qui ajoutent cette possibilité au chargeur de classes. Nous avons simplement besoin de configurations simples pour activer ces tisseurs au chargement.

Explications

Pour bien comprendre le processus de tissage au chargement d'AspectJ dans une application Spring, prenons comme exemple une calculatrice qui manipule des nombres complexes. Tout d'abord, créons la classe `Complex` pour représenter des nombres complexes. Nous définissons la méthode `toString()`, qui convertit un nombre complexe en une représentation textuelle ($a + bi$).

```
package com.apress.springrecipes.calculator;

public class Complex {

    private int real;
    private int imaginary;

    public Complex(int real, int imaginary) {
        this.real = real;
        this.imaginary = imaginary;
    }

    // Accesseurs et mutateurs.
    ...

    public String toString() {
        return "(" + real + " + " + imaginary + "i)";
    }
}
```

Ensuite, nous définissons une interface pour les opérations sur les nombres complexes. Pour des questions de simplicité, seules `add()` et `sub()` sont prises en charge.

```
package com.apress.springrecipes.calculator;

public interface ComplexCalculator {

    public Complex add(Complex a, Complex b);
    public Complex sub(Complex a, Complex b);
}
```

Voici le code d'implémentation de cette interface. À chaque fois, un nouvel objet complexe est retourné en résultat.

```

package com.apress.springrecipes.calculator;

public class ComplexCalculatorImpl implements ComplexCalculator {

    public Complex add(Complex a, Complex b) {
        Complex result = new Complex(a.getReal() + b.getReal(),
                                      a.getImaginary() + b.getImaginary());
        System.out.println(a + " + " + b + " = " + result);
        return result;
    }

    public Complex sub(Complex a, Complex b) {
        Complex result = new Complex(a.getReal() - b.getReal(),
                                      a.getImaginary() - b.getImaginary());
        System.out.println(a + " - " + b + " = " + result);
        return result;
    }
}

```

Avant de pouvoir utiliser cette calculatrice, nous devons la déclarer en tant que bean dans le conteneur Spring IoC.

```
<bean id="complexCalculator"
      class="com.apress.springrecipes.calculator.ComplexCalculatorImpl" />
```

Nous pouvons alors tester notre nouvelle calculatrice avec la classe Main suivante :

```

package com.apress.springrecipes.calculator;
...
public class Main {

    public static void main(String[] args) {
        ...
        ComplexCalculator complexCalculator =
            (ComplexCalculator) context.getBean("complexCalculator");
        complexCalculator.add(new Complex(1, 2), new Complex(2, 3));
        complexCalculator.sub(new Complex(5, 8), new Complex(2, 3));
    }
}

```

Notre calculatrice complexe fonctionne parfaitement. Cependant, nous souhaitons en améliorer les performances en plaçant dans un cache les objets de nombres complexes. Puisque l'utilisation d'un cache est une préoccupation transversale bien connue, nous la modularisons avec un aspect.

```

package com.apress.springrecipes.calculator;

import java.util.Collections;
import java.util.HashMap;
import java.util.Map;

import org.aspectj.lang.ProceedingJoinPoint;
import org.aspectj.lang.annotation.Around;
import org.aspectj.lang.annotation.Aspect;

```

```
@Aspect
public class ComplexCachingAspect {

    private Map<String, Complex> cache;

    public ComplexCachingAspect() {
        cache = Collections.synchronizedMap(new HashMap<String, Complex>());
    }

    @Around("call(public Complex.new(int, int)) && args(a,b)")
    public Object cacheAround(ProceedingJoinPoint joinPoint, int a, int b)
        throws Throwable {
        String key = a + "," + b;
        Complex complex = cache.get(key);
        if (complex == null) {
            System.out.println("ABSENCE de (" + key + ") dans le cache");
            complex = (Complex) joinPoint.proceed();
            cache.put(key, complex);
        }
        else {
            System.out.println("PRÉSENCE de (" + key + ") dans le cache");
        }
        return complex;
    }
}
```

Dans cet aspect, les objets de nombres complexes sont mis en cache à l'aide d'une table d'association, dont la clé est représentée par les valeurs réelle et imaginaire. Pour que cette table d'association soit sûre vis-à-vis des threads, nous l'enveloppons dans un Map synchronisé. Le meilleur moment pour effectuer une recherche dans le cache est lors de la création d'un objet de nombre complexe par invocation de son constructeur. Nous utilisons une expression AspectJ de point d'action call pour intercepter les points de jonction des appels au constructeur Complex(int, int). Ce point d'action n'étant pas reconnu par Spring AOP, nous ne l'avons pas encore présenté dans ce chapitre.

Nous avons ensuite besoin d'un greffon Around pour modifier la valeur de retour. Si nous trouvons un objet de nombre complexe de la même valeur dans le cache, nous le retournons directement à l'appelant. Dans le cas contraire, nous poursuivons avec l'invocation initiale du constructeur de manière à créer un nouvel objet de nombre complexe. Avant de le retourner à l'appelant, nous l'ajoutons dans le cache.

Puisque le point d'action call n'est pas pris en charge par Spring AOP, nous devons nous servir du framework AspectJ pour appliquer notre aspect. La configuration du framework AspectJ se fait au travers d'un fichier nommé aop.xml et placé dans le répertoire META-INF à la racine du chemin d'accès aux classes.

```
<!DOCTYPE aspectj PUBLIC "-//AspectJ//DTD//EN"
"http://www.eclipse.org/aspectj/dtd/aspectj.dtd">
```

```

<aspectj>
    <weaver>
        <include within="com.apress.springrecipes.calculator.*" />
    </weaver>

    <aspects>
        <aspect
            name="com.apress.springrecipes.calculator.ComplexCachingAspect" />
        </aspects>
    </aspectj>

```

Dans ce fichier de configuration d'AspectJ, nous devons préciser les aspects et les classes dans lesquelles ces aspects doivent être tissés. Dans notre exemple, nous tissons ComplexCachingAspect dans toutes les classes du paquetage com.apress.springrecipes.calculator.

Tissage au chargement par le tisseur d'AspectJ

AspectJ fournit un agent de tissage au chargement. Il suffit d'ajouter des arguments de machine virtuelle à la commande d'exécution de l'application. Nos classes sont alors tissées lorsqu'elles sont chargées dans la JVM.

```

java -javaagent:c:/spring-framework-2.5.6/lib/aspectj/aspectjweaver.jar
com.apress.springrecipes.calculator.Main

```

En exécutant notre application avec l'argument précédent, nous obtenons la sortie suivante. L'agent AspectJ intercepte tous les appels au constructeur Complex(int, int).

```

ABSENCE de (1,2) dans le cache
ABSENCE de (2,3) dans le cache
ABSENCE de (3,5) dans le cache
(1 + 2i) + (2 + 3i) = (3 + 5i)
ABSENCE de (5,8) dans le cache
PRÉSENCE de (2,3) dans le cache
PRÉSENCE de (3,5) dans le cache
(5 + 8i) - (2 + 3i) = (3 + 5i)

```

Tissage au chargement par le tisseur de Spring 2.5

Spring 2.5 fournit plusieurs tisseurs au chargement pour différents environnements d'exécution. Pour activer un tisseur adapté à notre application Spring, nous devons simplement ajouter l'élément XML vide `<context:load-time-weaver>`. Il est défini dans le schéma context et n'est disponible que dans Spring version 2.5 et ultérieures.

```

<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:context="http://www.springframework.org/schema/context"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/aop
                           http://www.springframework.org/schema/aop/spring-aop-2.5.xsd"

```

```
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:load-time-weaver />
...
</beans>
```

Spring est capable de détecter le tisseur au chargement adapté à notre environnement d'exécution. Certains serveurs d'applications Java EE possèdent des chargeurs de classes qui prennent en charge le mécanisme de tissage au chargement de Spring. Dans ce cas, il est inutile de préciser un agent Java dans la commande de lancement.

En revanche, pour une simple application Java, nous devons indiquer un agent de tissage au chargement fourni par Spring. Cet agent Spring est précisé dans l'argument de machine virtuelle de la commande d'exécution.

```
java -javaagent:c:/spring-framework-2.5.6/dist/weaving/spring-agent.jar=
com.apress.springrecipes.calculator.Main
```

Cependant, si nous exécutons notre application, nous obtenons la sortie suivante.

```
ABSENCE de (3,5) dans le cache
(1 + 2i) + (2 + 3i) = (3 + 5i)
PRÉSENCE de (3,5) dans le cache
(5 + 8i) - (2 + 3i) = (3 + 5i)
```

En effet, l'agent Spring intercepte uniquement les appels au constructeur `Complex(int, int)` effectués par les beans déclarés dans le conteneur Spring IoC. Puisque les opérations complexes sont créées dans la classe `Main`, l'agent Spring n'intercepte pas les appels à leur constructeur.

6.11 Configurer des aspects AspectJ dans Spring

Problème

Les aspects Spring AOP sont déclarés dans le fichier de configuration des beans afin que nous puissions les configurer facilement. Cependant, les aspects utilisés dans le framework AspectJ sont instanciés par ce framework lui-même. Nous devons obtenir les instances des aspects à partir du framework AspectJ pour les configurer.

Solution

Chaque aspect AspectJ offre une méthode statique de fabrique, nommée `aspectOf()`, qui nous permet d'accéder à l'instance courante de l'aspect. Dans le conteneur Spring IoC, nous pouvons déclarer un bean créé par une méthode de fabrique en définissant l'attribut `factory-method`.

Explications

Nous pouvons faire en sorte que la table d'association du cache de ComplexCachingAspect soit configurée par un mutateur et ainsi supprimer son instantiation dans le constructeur.

```
package com.apress.springrecipes.calculator;
...
import java.util.Collections;
import java.util.Map;

import org.aspectj.lang.annotation.Aspect;

@Aspect
public class ComplexCachingAspect {

    private Map<String, Complex> cache;

    public void setCache(Map<String, Complex> cache) {
        this.cache = Collections.synchronizedMap(cache);
    }
    ...
}
```

Pour configurer cette propriété dans le conteneur Spring IoC, nous déclarons un bean créé par la méthode de fabrique aspectOf().

```
<bean class="com.apress.springrecipes.calculator.ComplexCachingAspect"
      factory-method="aspectOf">
    <property name="cache">
        <map>
            <entry key="2,3">
                <bean class="com.apress.springrecipes.calculator.Complex">
                    <constructor-arg value="2" />
                    <constructor-arg value="3" />
                </bean>
            </entry>
            <entry key="3,5">
                <bean class="com.apress.springrecipes.calculator.Complex">
                    <constructor-arg value="3" />
                    <constructor-arg value="5" />
                </bean>
            </entry>
        </map>
    </property>
</bean>
```

ASTUCE

Vous vous demandez peut-être pourquoi la classe ComplexCachingAspect possède une méthode statique de fabrique aspectOf() que nous n'avons pas déclarée. Cette méthode est ajoutée par AspectJ au chargement afin que nous puissions accéder à l'instance de l'aspect courant. Si vous utilisez Spring IDE, il risque donc d'afficher un avertissement indiquant qu'il ne trouve pas cette méthode dans la classe.

6.12 Injecter des beans Spring dans des objets de domaine

Problème

Les beans déclarés dans le conteneur Spring IoC peuvent être liés les uns aux autres grâce à l'injection de dépendance de Spring. En revanche, les objets créés hors du conteneur Spring IoC ne peuvent pas être liés à des beans par configuration. Nous devons procéder à une liaison manuelle dans le code.

Solution

Les objets créés en dehors du conteneur Spring IoC sont habituellement des objets de domaine. Ils sont instanciés à l'aide de l'opérateur `new` ou sont le résultat de requêtes sur une base de données.

Pour injecter un bean Spring dans les objets de domaine créés hors de Spring, nous devons nous aider de la programmation orientée aspect. En réalité, l'injection de beans Spring est également une sorte de préoccupation transversale. Puisque les objets de domaine ne sont pas créés par Spring, nous ne pouvons pas employer Spring AOP pour l'injection. Spring fournit donc un aspect AspectJ dédié à cette fonction. Nous pouvons l'activer dans le framework Aspect.

Explications

Supposons qu'il existe un formateur global des nombres complexes. Il accepte un motif de mise en forme.

```
package com.apress.springrecipes.calculator;

public class ComplexFormatter {

    private String pattern;

    public void setPattern(String pattern) {
        this.pattern = pattern;
    }

    public String format(Complex complex) {
        return pattern.replaceAll("a", Integer.toString(complex.getReal()))
                      .replaceAll("b", Integer.toString(complex.getImaginary()));
    }
}
```

Nous configurons ce formateur dans le conteneur Spring IoC en lui indiquant le motif.

```
<bean id="complexFormatter"
      class="com.apress.springrecipes.calculator.ComplexFormatter">
    <property name="pattern" value="(a + bi)" />
</bean>
```

Nous souhaitons utiliser ce formateur dans la méthode `toString()` de la classe `Complex` pour convertir un nombre complexe en une chaîne de caractères. Elle expose un mutateur pour la classe `ComplexFormatter`.

```
package com.apress.springrecipes.calculator;

public class Complex {

    private int real;
    private int imaginary;
    ...
    private ComplexFormatter formatter;

    public void setFormatter(ComplexFormatter formatter) {
        this.formatter = formatter;
    }

    public String toString() {
        return formatter.format(this);
    }
}
```

Toutefois, puisque les objets de nombres complexes ne sont pas créés dans le conteneur Spring IoC, ils ne peuvent pas être configurés pour l'injection de dépendance. Nous devons écrire du code qui injecte une instance de `ComplexFormatter` dans chaque objet de nombre complexe.

Dans la bibliothèque d'aspects de Spring, nous trouvons `AnnotationBeanConfigurerAspect`, qui permet de configurer les dépendances de n'importe quel objet même s'il n'est pas créé par le conteneur Spring IoC. En premier lieu, nous devons marquer notre type d'objet avec l'annotation `@Configurable`. Elle indique que ce type d'objet est configurable.

```
package com.apress.springrecipes.calculator;

import org.springframework.beans.factory.annotation.Configurable;

@Configuration
public class Complex {
    ...
}
```

Pour activer l'aspect mentionné, Spring définit l'élément XML `<context:spring-configured>`. Dans Spring 2.0, cet élément se trouve dans le schéma `aop`. Il a été déplacé dans le schéma `context` depuis Spring 2.5.

INFO

Pour utiliser la bibliothèque d'aspects de Spring pour AspectJ, vous devez inclure le fichier `spring-aspects.jar` (situé dans le répertoire `dist/weaving` de l'installation de Spring) dans le chemin d'accès aux classes.

```
<beans ...>
  ...
  <context:load-time-weaver />
  <context:spring-configured />

  <bean class="com.apress.springrecipes.calculator.Complex"
    scope="prototype">
    <property name="formatter" ref="complexFormatter" />
  </bean>
</beans>
```

Lorsqu'une classe annotée par `@Configurable` est instanciée, l'aspect recherche une définition de bean de portée prototype dont le type correspond à celui de la classe. Il configure ensuite les nouvelles instances en fonction de cette définition de bean. Si des propriétés sont définies dans la définition du bean, elles sont fixées par l'aspect dans les nouvelles instances.

Enfin, l'aspect doit être activé par le framework AspectJ. Nous pouvons le tisser au chargement dans nos classes à l'aide de l'agent Spring.

```
java -javaagent:c:/spring-framework-2.5.6/dist/weaving/spring-agent.jar=
  com.apress.springrecipes.calculator.Main
```

Nous pouvons également relier une classe configurable à une définition de bean en utilisant l'identifiant du bean. Celui-ci est indiqué dans la valeur de l'annotation `@Configurable`.

```
package com.apress.springrecipes.calculator;

import org.springframework.beans.factory.annotation.Configurable;

@Configurable("complex")
public class Complex {
  ...
}
```

Nous devons ensuite ajouter l'attribut `id` dans la définition de bean correspondant pour établir la liaison avec une classe configurable.

```
<bean id="complex" class="com.apress.springrecipes.calculator.Complex"
  scope="prototype">
  <property name="formatter" ref="complexFormatter" />
</bean>
```

Comme les beans Spring normaux, les beans configurables prennent en charge la liaison automatique et la vérification des dépendances.

```
package com.apress.springrecipes.calculator;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Configurable;
```

```
@Configurable(  
    value = "complex",  
    autowire = Autowire.BY_TYPE,  
    dependencyCheck = true)  
public class Complex {  
    ...  
}
```

L'attribut `dependencyCheck` est de type `Boolean`, non une énumération. Lorsqu'il est fixé à `true`, il a le même effet que `dependency-check="objects"`, c'est-à-dire vérifier les types autres que les types primitifs et les collections. Lorsque la liaison automatique est activée, il devient inutile de fixer explicitement la propriété `formatter`.

```
<bean id="complex" class="com.apress.springrecipes.calculator.Complex"  
    scope="prototype" />
```

Dans Spring 2.5, nous n'avons plus besoin de configurer la liaison automatique et la vérification des dépendances au niveau de la classe pour `@Configurable`. À la place, nous pouvons marquer le mutateur du formateur avec l'annotation `@Autowired`.

```
package com.apress.springrecipes.calculator;  
  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.beans.factory.annotation.Configurable;  
  
@Configurable("complex")  
public class Complex {  
    ...  
    private ComplexFormatter formatter;  
  
    @Autowired  
    public void setFormatter(ComplexFormatter formatter) {  
        this.formatter = formatter;  
    }  
}
```

Ensuite, nous ajoutons l'élément `<context:annotation-config>` dans le fichier de configuration des beans pour traiter les méthodes marquées par ces annotations.

```
<beans ...>  
    <context:annotation-config />  
    ...  
</beans>
```

6.13 En résumé

Dans ce chapitre, nous avons vu comment écrire des aspects avec des annotations AspectJ ou des configurations XML dans le fichier de configuration des beans, et comment les enregistrer dans le conteneur Spring IoC. Spring 2.x AOP prend en charge cinq types de greffons : Before, After, After returning, After throwing et Around.

Nous avons également examiné les différents types de points d'action pour désigner des points de jonction par signature de méthode, signature de type et nom de bean. Toutefois, Spring AOP prend uniquement en charge les points de jonction de type exécution de méthode pour les beans déclarés dans son conteneur IoC. Si nous utilisons une expression de point d'action en dehors de ce contexte, une exception est lancée.

L'introduction est un type de greffon POA particulier. Il permet à nos objets d'implémenter dynamiquement une interface en fournissant une classe d'implémentation. Nous disposons ainsi d'un équivalent à l'héritage multiple. Les introductions servent souvent à ajouter des comportements et des états à un ensemble d'objets existants.

Pour utiliser des types de points d'action non reconnus par Spring AOP ou appliquer des aspects à des objets créés en dehors du conteneur Spring IoC, nous devons utiliser le framework AspectJ dans notre application Spring. Des aspects peuvent être tissés dans nos classes par un tisseur au chargement. La bibliothèque d'aspects de Spring fournit également plusieurs aspects AspectJ utiles. L'un d'eux injecte des beans Spring dans des objets de domaine créés en dehors de Spring.

Le chapitre suivant explique comment le framework Spring JDBC permet de simplifier la gestion des accès à une base de données relationnelle.

Prise en charge de JDBC

Au sommaire de ce chapitre

- ✓ Problèmes associés à l'utilisation directe de JDBC
- ✓ Utiliser un template JDBC pour la mise à jour
- ✓ Utiliser un template JDBC pour interroger une base de données
- ✓ Simplifier la création d'un template JDBC
- ✓ Utiliser le template JDBC simple avec Java 1.5
- ✓ Utiliser des paramètres nommés dans un template JDBC
- ✓ Modéliser les opérations JDBC avec des objets élémentaires
- ✓ Gérer les exceptions dans le framework Spring JDBC
- ✓ En résumé

Ce chapitre explique comment Spring permet de simplifier les opérations d'accès à une base de données. L'accès aux données est une exigence fréquente pour les applications d'entreprise, qui s'appuient souvent sur des données enregistrées dans des bases de données relationnelles. JDBC (*Java Database Connectivity*), une composante essentielle de Java SE, définit un ensemble d'API standard pour accéder aux bases de données relationnelles de manière indépendante du fournisseur.

L'objectif de JDBC est d'offrir des API au travers desquelles nous pouvons exécuter des requêtes SQL sur une base de données. Cependant, avec JDBC, nous devons gérer nous-mêmes les ressources liées à la base de données et traiter explicitement les exceptions. Pour faciliter l'utilisation de JDBC, Spring propose un framework d'accès JDBC en définissant une couche abstraite au-dessus de ses API.

Au cœur du framework Spring JDBC, des templates JDBC offrent des méthodes templates pour les différents types d'opérations JDBC. Chaque méthode template est en charge de la procédure globale, mais permet d'en remplacer les phases. Nous pouvons ainsi minimiser le travail lié aux accès à la base de données, tout en conservant une grande souplesse.

Outre les templates JDBC, le framework Spring JDBC apporte une autre approche, plus orientée objet, pour organiser la logique d'accès aux données. Chaque opération de base de données peut être modélisée sous forme d'un objet d'opération léger. Par rapport aux templates JDBC, les objets d'opération ne sont qu'une solution pour organiser la logique d'accès aux données. Certains développeurs peuvent préférer la première approche, tandis que d'autres préféreront la seconde.

À la fin de ce chapitre, vous serez capable d'utiliser le framework Spring JDBC pour l'accès aux bases de données relationnelles. Puisqu'il fait partie du module Spring d'accès aux données, le framework Spring JDBC reste dans la droite ligne des autres éléments de ce module. Par conséquent, l'apprentissage du framework JDBC est une bonne manière d'appréhender l'ensemble du module d'accès aux données.

7.1 Problèmes associés à l'utilisation directe de JDBC

Supposons que nous développons une application d'enregistrement de véhicules. Ses principales fonctions sont les opérations CRUD (*Create, Read, Update et Delete*) de base sur les enregistrements de véhicules. Ces enregistrements sont stockés dans une base de données relationnelle, à laquelle on accède via JDBC. Tout d'abord, nous écrivons la classe `Vehicle` pour représenter un véhicule en Java.

```
package com.apress.springrecipes.vehicle;

public class Vehicle {

    private String vehicleNo;
    private String color;
    private int wheel;
    private int seat;

    // Constructeurs, accesseurs et mutateurs.
    ...
}
```

Configurer la base de données de l'application

Avant de développer notre application d'enregistrement de véhicules, nous devons configurer la base de données sous-jacente. Pour des questions de faible consommation mémoire et de facilité de configuration, nous choisissons Apache Derby (<http://db.apache.org/derby/>). Derby est un moteur de bases de données relationnelle open-source disponible sous licence Apache et implémenté exclusivement en Java.

INFO

Vous pouvez télécharger la distribution binaire d'Apache Derby (par exemple la version v10.4) depuis son site web et extraire son contenu dans le répertoire de votre choix pour terminer l'installation.

Derby peut fonctionner en mode embarqué ou en mode client-serveur. Pour les tests, le mode client-serveur est le mieux adapté car il nous permet d'inspecter et de modifier des données avec n'importe quel outil graphique de bases de données qui prend en charge JDBC, par exemple Eclipse Data Tools Platform (DTP).

INFO

Pour démarrer le serveur Derby en mode client-serveur, il suffit d'exécuter le script `start- NetworkServer` correspondant à votre plate-forme (il se trouve dans le répertoire `bin` de l'installation de Derby).

Après avoir démarré le serveur réseau de Derby sur la machine locale, nous pouvons nous y connecter en utilisant les propriétés JDBC données au Tableau 7.1.

INFO

Pour établir une connexion avec le serveur Derby, vous avez besoin du pilote JDBC `derby-client.jar` (situé dans le répertoire `lib` de l'installation de Derby).

Tableau 7.1 : Propriétés JDBC pour la connexion à la base de données de l'application

<i>Propriété</i>	<i>Valeur</i>
Classe du pilote	<code>org.apache.derby.jdbc.ClientDriver</code>
URL	<code>jdbc:derby://localhost:1527/vehicle;create=true</code>
Nom d'utilisateur	<code>app</code>
Mot de passe	<code>app</code>

Lors de la première connexion à cette base de données, l'instance de base de données `vehicle` est créée si elle n'existe pas déjà. En effet, l'URL de connexion précise `create=true`. Le nom d'utilisateur et le mot de passe sont arbitraires car Derby désactive par défaut l'authentification. Ensuite, nous devons créer la table `VEHICLE` pour stocker les enregistrements de véhicules. Pour cela, nous exécutons la requête SQL suivante, par exemple dans l'outil `ij` fourni avec Derby. Par défaut, la table est créée dans le schéma de bases de données `APP`.

```
CREATE TABLE VEHICLE (
    VEHICLE_NO      VARCHAR(10)      NOT NULL,
    COLOR           VARCHAR(10),
    WHEEL           INT,
    SEAT            INT,
    PRIMARY KEY (VEHICLE_NO)
);
```

Comprendre le design pattern DAO

Très souvent, les développeurs expérimentés font l'erreur de mélanger différents types de logiques (par exemple, logique de présentation, logique métier et logique d'accès aux données) au sein d'un grand module unique. Cette approche réduit la possibilité de réutiliser les modules et complique leur maintenance en raison du couplage étroit introduit. Le design pattern Objet d'accès aux données (DAO, *Data Access Object*) a pour objectif général d'éviter ces problèmes en séparant la logique d'accès aux données de la logique métier et de la logique de présentation. Ce pattern recommande d'encapsuler la logique d'accès aux données dans des modules indépendants appelés objets d'accès aux données.

Dans le cas de notre application, nous pouvons rendre abstraites les opérations d'accès aux données pour l'insertion, la mise à jour, la suppression et la recherche d'un véhicule. Ces opérations sont déclarées dans une interface de DAO afin de ne pas figer la technologie d'implémentation de l'accès aux données.

```
package com.apress.springrecipes.vehicle;

public interface VehicleDao {

    public void insert(Vehicle vehicle);
    public void update(Vehicle vehicle);
    public void delete(Vehicle vehicle);
    public Vehicle findByVehicleNo(String vehicleNo);
}
```

La plupart des méthodes de l'API JDBC déclarent lancer une exception `java.sql.SQLException`. Toutefois, puisque le rôle de cette interface est de rendre abstraites les opérations d'accès aux données, elle ne doit pas dépendre d'une technologie d'implémentation. Par conséquent, il est peu judicieux qu'elle déclare lancer l'exception `SQLException` caractéristique de JDBC. Lorsqu'on implémente une interface de DAO, une pratique courante consiste à envelopper ce type d'exception dans une exception d'exécution.

Implémenter l'objet d'accès aux données avec JDBC

Pour accéder à la base de données au travers de JDBC, nous créons une implémentation de cette interface de DAO (par exemple `JdbcVehicleDao`). Puisque notre implémentation doit se connecter à la base de données pour exécuter des requêtes SQL, nous établissons des connexions en précisant la classe du pilote, l'URL de la base de données, un nom d'utilisateur et un mot de passe. Toutefois, dans JDBC 2.0 et les versions ultérieures, nous pouvons obtenir des connexions de base de données à partir d'un objet `javax.sql.DataSource` préconfiguré, sans connaître les détails de connexion.

```
package com.apress.springrecipes.vehicle;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

public class JdbcVehicleDao implements VehicleDao {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        Connection conn = null;
        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setString(1, vehicle.getVehicleNo());
            ps.setString(2, vehicle.getColor());
            ps.setInt(3, vehicle.getWheel());
            ps.setInt(4, vehicle.getSeat());
            ps.executeUpdate();
            ps.close();
        } catch (SQLException e) {
            throw new RuntimeException(e);
        } finally {
            if (conn != null) {
                try {
                    conn.close();
                } catch (SQLException e) {}
            }
        }
    }

    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        Connection conn = null;
        try {
            conn = dataSource.getConnection();
            PreparedStatement ps = conn.prepareStatement(sql);
            ps.setString(1, vehicleNo);

            Vehicle vehicle = null;
            ResultSet rs = ps.executeQuery();
            if (rs.next()) {
                vehicle = new Vehicle(rs.getString("VEHICLE_NO"),
                    rs.getString("COLOR"), rs.getInt("WHEEL"),
                    rs.getInt("SEAT"));
            }
        }
```

```
        rs.close();
        ps.close();
        return vehicle;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        if (conn != null) {
            try {
                conn.close();
            } catch (SQLException e) {}
        }
    }
}

public void update(Vehicle vehicle) {}

public void delete(Vehicle vehicle)
}
```

L'opération d'insertion d'un véhicule suit un scénario classique de mise à jour avec JDBC. À chaque invocation de cette méthode, nous obtenons une connexion à partir de la source de données et exécutons la requête SQL sur cette connexion. Puisque notre interface de DAO ne déclare pas le lancement d'une exception vérifiée, nous devons envelopper toute exception `SQLException` dans une exception non vérifiée de type `RuntimeException`. Enfin, il ne faut pas oublier de libérer la connexion dans le bloc `finally` car, dans le cas contraire, l'application pourrait rapidement arriver à court de connexions.

Dans l'exemple de code, nous avons omis les opérations de mise à jour et de suppression car, d'un point de vue technique, elles ressemblent fortement à l'opération d'insertion. Pour l'opération d'interrogation de la base, outre l'exécution de la requête SQL, nous devons extraire les données à partir de l'ensemble de résultat obtenu et nous en servir pour construire un objet de véhicule.

Configurer une source de données dans Spring

L'interface standard `javax.sql.DataSource` est définie par les spécifications de JDBC. Il existe de nombreuses implémentations d'une source de données, proposées par différents fournisseurs et projets. Il est très facile de passer d'une implémentation à l'autre, car elles implémentent toute l'interface commune `DataSource`. En tant que framework applicatif Java, Spring fournit également plusieurs implémentations pratiques, quoique moins puissantes, d'une source de données. La plus simple, nommée `DriverManagerDataSource`, ouvre une nouvelle connexion à chaque demande.

INFO

Pour accéder à une instance de base de données s'exécutant sur le serveur Derby, vous devez inclure le fichier `derbyclient.jar` (situé dans le répertoire `lib` de l'installation de Derby) dans le chemin d'accès aux classes.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
                  value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
                  value="jdbc:derby://localhost:1527/vehicle;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

    <bean id="vehicleDao"
          class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

`DriverManagerDataSource` n'est pas une implémentation très efficace car une nouvelle connexion est ouverte pour le client à chaque requête. Spring propose une autre implémentation d'une source de données, `SingleConnectionDataSource`. Comme son nom l'indique, elle maintient une seule connexion réutilisée à chaque requête et jamais fermée. Elle n'est donc pas adaptée à un environnement multithread.

Les implémentations fournies par Spring sont principalement destinées aux tests. En revanche, de nombreuses implémentations professionnelles d'une source de données prennent en charge la mutualisation des connexions. Par exemple, le module DBCP (*Database Connection Pooling Services*) de la bibliothèque Apache Commons dispose de plusieurs implémentations d'une source de données qui prennent en charge les pools de connexions. En particulier, `BasicDataSource` accepte les mêmes propriétés de connexion que `DriverManagerDataSource`, mais nous permet de préciser le nombre initial de connexions du pool et le nombre maximal de connexions actives.

INFO

Pour utiliser les implémentations d'une source de données du module DBCP, vous devez inclure les fichiers `commons-dbc.jar` et `commons-pool.jar` (situés dans le répertoire `lib/jakarta-commons` de l'installation de Spring) dans le chemin d'accès aux classes.

```
<bean id="dataSource"
      class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName"
              value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url"
              value="jdbc:derby://localhost:1527/vehicle;create=true" />
    <property name="username" value="app" />
    <property name="password" value="app" />
    <property name="initialSize" value="2" />
    <property name="maxActive" value="5" />
</bean>
```

Plusieurs serveurs d'applications Java EE incluent des implémentations d'une source de données configurables depuis la console du serveur. Si une source de données a été configurée dans un serveur d'applications et si elle est exposée au travers de JNDI, nous pouvons utiliser JndiObjectFactoryBean pour l'obtenir.

```
<bean id="dataSource"
      class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiName" value="jdbc/VehicleDS" />
</bean>
```

Dans Spring 2.x, l'élément jndi-lookup, défini dans le schéma jee, simplifie les recherches JNDI.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:jee="http://www.springframework.org/schema/jee"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.0.xsd
                           http://www.springframework.org/schema/jee
                           http://www.springframework.org/schema/jee/spring-jee-2.0.xsd">

    <jee:jndi-lookup id="dataSource" jndi-name="jdbc/VehicleDS" />
    ...
</beans>
```

Exécuter l'objet d'accès aux données

La classe Main suivante teste notre objet d'accès aux données en procédant à l'insertion d'un nouveau véhicule dans la base. En cas de succès, nous pouvons immédiatement retrouver le véhicule depuis la base de données.

```
package com.apress.springrecipes.vehicle;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");
```

```
VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
Vehicle vehicle = new Vehicle("TEM0001", "Rouge", 4, 4);
vehicleDao.insert(vehicle);

vehicle = vehicleDao.findByVehicleNo("TEM0001");
System.out.println("N° véhicule : " + vehicle.getVehicleNo());
System.out.println("Couleur : " + vehicle.getColor());
System.out.println("Roues motrices : " + vehicle.getWheel());
System.out.println("Places : " + vehicle.getSeat());
}
}
```

Nous venons d'implémenter un objet d'accès aux données en utilisant directement JDBC. Toutefois, il est facile de constater qu'une grande partie du code JDBC est répétée pour chaque opération sur la base de données. Un tel code redondant augmente la taille des méthodes d'accès aux données et diminue leur lisibilité.

7.2 Utiliser un template JDBC pour la mise à jour

Problème

Pour implémenter une opération JDBC de mise à jour, nous devons effectuer les tâches suivantes, dont plusieurs sont redondantes :

1. Obtenir une connexion à la base de données à partir de la source de données.
2. Créer un objet `PreparedStatement` à partir de la connexion.
3. Lier les paramètres à l'objet `PreparedStatement`.
4. Exécuter l'objet `PreparedStatement`.
5. Gérer l'exception `SQLException`.
6. Nettoyer l'objet de requête et la connexion.

Solution

La classe `JdbcTemplate` déclare plusieurs méthodes `update()` surchargées qui contrôlent le déroulement global d'une mise à jour. Les différentes versions de cette méthode nous permettent de remplacer des sous-tâches du processus par défaut. Le framework Spring JDBC définit plusieurs interfaces de rappel pour l'encapsulation des différents sous-ensembles de tâches. Nous pouvons implémenter l'une de ces interfaces de rappel et passer son instance à la méthode `update()` correspondante afin d'intervenir sur le processus.

Explications

Mettre à jour une base de données avec un créateur de requête

La première interface de rappel examinée se nomme `PreparedStatementCreator`. Nous implémentons cette interface pour remplacer la création de la requête (tâche 2) et la liaison des paramètres (tâche 3) de la procédure globale de mise à jour. Voici notre implémentation de l'interface `PreparedStatementCreator` pour l'insertion d'un véhicule dans la base de données :

```
package com.apress.springrecipes.vehicle;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

import org.springframework.jdbc.core.PreparedStatementCreator;

public class InsertVehicleStatementCreator
    implements PreparedStatementCreator {

    private Vehicle vehicle;

    public InsertVehicleStatementCreator(Vehicle vehicle) {
        this.vehicle = vehicle;
    }

    public PreparedStatement createPreparedStatement(Connection conn)
        throws SQLException {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        PreparedStatement ps = conn.prepareStatement(sql);
        ps.setString(1, vehicle.getVehicleNo());
        ps.setString(2, vehicle.getColor());
        ps.setInt(3, vehicle.getWheel());
        ps.setInt(4, vehicle.getSeat());
        return ps;
    }
}
```

Lorsqu'on implémente l'interface `PreparedStatementCreator`, la connexion à la base de données est passée en argument de la méthode `createPreparedStatement()`. Dans cette méthode, il suffit ensuite de créer un objet `PreparedStatement` sur la connexion fournie et de lier nos paramètres à cet objet. Pour finir, nous devons retourner l'objet `PreparedStatement` dans la valeur de retour de la méthode. Puisque la signature de cette méthode déclare lancer l'exception `SQLException`, nous n'avons pas besoin de la traiter.

Grâce à ce créateur de requête, nous pouvons simplifier l'opération d'ajout d'un véhicule. Tout d'abord, nous créons une instance de la classe `JdbcTemplate` et lui passons la source de données de manière à obtenir une connexion. Ensuite, nous invoquons sim-

plement la méthode `update()` en lui fournissant notre créateur de requête pour que le template termine la procédure de mise à jour.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(Vehicle vehicle) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
        jdbcTemplate.update(new InsertVehicleStatementCreator(vehicle));
    }
}
```

Si elles ne sont utilisées que dans une seule méthode, il est généralement préférable d'implémenter l'interface `PreparedStatementCreator` et les autres interfaces de rappel sous forme de classes internes. En effet, cela nous permet d'accéder directement aux variables locales et aux arguments des méthodes depuis la classe interne, au lieu de les passer en arguments du constructeur. Toutefois, ces variables et arguments doivent être déclarés `final`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementCreator;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(new PreparedStatementCreator() {
            ...
            public PreparedStatement createPreparedStatement(Connection conn)
                throws SQLException {
                String sql = "INSERT INTO VEHICLE "
                    + "(VEHICLE_NO, COLOR, WHEEL, SEAT) "
                    + "VALUES (?, ?, ?, ?)";
                PreparedStatement ps = conn.prepareStatement(sql);
                ps.setString(1, vehicle.getVehicleNo());
                ps.setString(2, vehicle.getColor());
                ps.setInt(3, vehicle.getWheel());
                ps.setInt(4, vehicle.getSeat());
                return ps;
            }
        });
    }
}
```

Puisqu'elle n'est plus utilisée, nous pouvons supprimer la classe `InsertVehicleStatementCreator`.

Mettre à jour une base de données avec un configurateur de requête

La seconde interface de rappel, `PreparedStatementSetter`, concerne uniquement l'opération de liaison des paramètres (tâche 3) de la procédure globale de mise à jour.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.PreparedStatementSetter;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.update(sql, new PreparedStatementSetter() {

            public void setValues(PreparedStatement ps)
                throws SQLException {
                    ps.setString(1, vehicle.getVehicleNo());
                    ps.setString(2, vehicle.getColor());
                    ps.setInt(3, vehicle.getWheel());
                    ps.setInt(4, vehicle.getSeat());
                }
        });
    }
}
```

Une autre version de la méthode template `update()` accepte en arguments une requête SQL et un objet `PreparedStatementSetter`. À partir de la requête SQL, elle crée un objet `PreparedStatement` à notre place. Dans ce cas, il nous reste simplement à lier nos paramètres à l'objet `PreparedStatement`.

Mettre à jour une base de données avec une requête SQL et des valeurs de paramètres

Enfin, dans sa variante la plus simple, la méthode `update()` accepte une requête SQL et un tableau d'objets définissant ses paramètres. Elle se charge de créer un objet `PreparedStatement` à partir de la requête SQL et de lier les paramètres. Par conséquent, il devient inutile de redéfinir les tâches de la mise à jour.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);
```

```
        jdbcTemplate.update(sql, new Object[] { vehicle.getVehicleNo(),
                                              vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
    }
}
```

Des trois versions de la méthode `update()` décrites, la dernière est la plus simple car il est inutile d'implémenter une interface de rappel. *A contrario*, la première est la plus souple car nous pouvons intervenir sur l'objet `PreparedStatement` avant son exécution. En pratique, il faut toujours choisir la version la plus simple qui répond aux besoins.

La classe `JdbcTemplate` propose d'autres méthodes `update()` surchargées. Pour de plus amples informations, consultez sa documentation JavaDoc.

Mettre à jour en série une base de données

Supposons que nous voulions insérer tout un lot de véhicules dans la base de données. Si nous invoquons la méthode `insert()` à plusieurs reprises, l'insertion est très lente car la requête SQL est compilée à chaque fois. Il est préférable d'ajouter une nouvelle méthode à l'interface de DAO de manière à insérer un ensemble de véhicules.

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles);
}
```

La classe `JdbcTemplate` fournit la méthode `template.batchUpdate()` pour procéder à des opérations de mise à jour en série. Elle prend en arguments une requête SQL et un objet `BatchPreparedStatementSetter`. Dans cette méthode, la requête est exécutée plusieurs fois, mais compilée une seule fois. Si notre pilote de base de données est compatible avec JDBC 2.0, cette méthode utilise automatiquement la fonctionnalité de mise à jour en série de manière à améliorer les performances.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BatchPreparedStatementSetter;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public void insertBatch(final List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
                    + "VALUES (?, ?, ?, ?)";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        jdbcTemplate.batchUpdate(sql, new BatchPreparedStatementSetter() {

            public int getBatchSize() {
                return vehicles.size();
            }
        }
```

```
        public void setValues(PreparedStatement ps, int i)
            throws SQLException {
            Vehicle vehicle = vehicles.get(i);
            ps.setString(1, vehicle.getVehicleNo());
            ps.setString(2, vehicle.getColor());
            ps.setInt(3, vehicle.getWheel());
            ps.setInt(4, vehicle.getSeat());
        }
    });
}
}
```

Pour tester l'opération d'insertion en série, nous ajoutons le code suivant dans la classe Main :

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
    ...
    VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
    Vehicle vehicle1 = new Vehicle("TEM0002", "Bleu", 4, 4);
    Vehicle vehicle2 = new Vehicle("TEM0003", "Noir", 4, 6);
    vehicleDao.insertBatch(
        Arrays.asList(new Vehicle[] { vehicle1, vehicle2 }));
    }
}
```

7.3 Utiliser un template JDBC pour interroger une base de données

Problème

Pour mettre en œuvre une interrogation JDBC, nous devons effectuer les tâches suivantes, dont deux (les tâches 5 et 6) n'existent pas dans une opération de mise à jour :

1. Obtenir une connexion à la base de données à partir de la source de données.
2. Créer un objet PreparedStatement à partir de la connexion.
3. Lier les paramètres à l'objet PreparedStatement.
4. Exécuter l'objet PreparedStatement.
5. Parcourir l'ensemble de résultat obtenu.
6. Extraire les données de l'ensemble de résultat.
7. Gérer l'exception SQLException.
8. Nettoyer l'objet de requête et la connexion.

Solution

La classe `JdbcTemplate` déclare plusieurs méthodes `query()` surchargées qui contrôlent le déroulement global d'une interrogation. Nous pouvons remplacer la création de la requête (tâche 2) et la liaison des paramètres (tâche 3) en implémentant les interfaces `PreparedStatementCreator` et `PreparedStatementSetter`, comme nous l'avons fait pour les opérations de mise à jour. Par ailleurs, le framework Spring JDBC propose plusieurs manières de remplacer l'extraction des données (tâche 6).

Explications

Extraire des données avec un gestionnaire de rappel pour les lignes

`RowCallbackHandler` est la principale interface permettant de traiter la ligne courante de l'ensemble de résultat. L'une de ses méthodes `query()` parcourt le contenu de l'ensemble de résultat à notre place et invoque `RowCallbackHandler` pour chaque ligne. La méthode `processRow()` est ainsi invoquée une fois pour chaque ligne de l'ensemble de résultat.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowCallbackHandler;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        final Vehicle vehicle = new Vehicle();
        jdbcTemplate.query(sql, new Object[] { vehicleNo },
            new RowCallbackHandler() {
                public void processRow(ResultSet rs) throws SQLException {
                    vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
                    vehicle.setColor(rs.getString("COLOR"));
                    vehicle.setWheel(rs.getInt("WHEEL"));
                    vehicle.setSeat(rs.getInt("SEAT"));
                }
            });
        return vehicle;
    }
}
```

Puisque l'interrogation SQL retourne une seule ligne au maximum, nous pouvons créer un objet de véhicule sous forme de variable locale et fixer ses propriétés en extrayant les données à partir de l'ensemble de résultat. Lorsque l'ensemble de résultat peut contenir plusieurs lignes, les objets doivent être collectés sous forme d'une liste.

Extraire des données par correspondance de ligne

L'interface RowMapper est plus générale que l'interface RowCallbackHandler. Son rôle est de créer une correspondance entre une seule ligne de l'ensemble de résultat et un objet personnalisé. Elle peut donc être appliquée à un ensemble de résultat contenant une ou plusieurs lignes. Pour faciliter la réutilisation, il est préférable d'implémenter l'interface RowMapper sous forme d'une classe normale plutôt que sous forme d'une classe interne. Dans la méthode mapRow() de cette interface, nous construisons l'objet qui représente une ligne et l'utilisons comme valeur de retour.

```
package com.apress.springrecipes.vehicle;

import java.sql.ResultSet;
import java.sql.SQLException;

import org.springframework.jdbc.core.RowMapper;

public class VehicleRowMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Vehicle vehicle = new Vehicle();
        vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
        vehicle.setColor(rs.getString("COLOR"));
        vehicle.setWheel(rs.getInt("WHEEL"));
        vehicle.setSeat(rs.getInt("SEAT"));
        return vehicle;
    }
}
```

Nous l'avons mentionné, RowMapper peut être utilisée avec un ensemble de résultat contenant une ou plusieurs lignes. Lorsque l'interrogation de la base de données retourne un seul objet, comme c'est le cas pour findByVehicleNo(), nous devons invoquer la méthode queryForObject() de JdbcTemplate.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        Vehicle vehicle = (Vehicle) jdbcTemplate.queryForObject(sql,
            new Object[] { vehicleNo }, new VehicleRowMapper());
        return vehicle;
    }
}
```

Spring 2.5 fournit une implémentation pratique de RowMapper, BeanPropertyRowMapper, qui peut associer automatiquement une ligne à une nouvelle instance de la classe indiquée. Elle commence par instancier cette classe, puis elle associe chaque

valeur de colonne à une propriété en mettant en correspondance leur nom. Elle est capable d'établir une correspondance entre un nom de propriété (par exemple vehicleNo) et une colonne de même nom ou une colonne de nom contenant un caractère souligné (par exemple VEHICLE_NO).

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        Vehicle vehicle = (Vehicle) jdbcTemplate.queryForObject(sql,
            new Object[] { vehicleNo } ,
            BeanPropertyRowMapper.newInstance(Vehicle.class));
        return vehicle;
    }
}
```

Demander plusieurs lignes

Voyons à présent comment effectuer une interrogation dont l'ensemble de résultat contient plusieurs lignes. Par exemple, supposons que nous ayons besoin d'une méthode findAll() dans l'interface de DAO pour obtenir tous les véhicules.

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public List<Vehicle> findAll();
}
```

Sans l'aide de RowMapper, nous pouvons toujours invoquer la méthode queryForList() et lui passer une requête SQL. Le résultat retourné est une liste de tables d'association. Chaque table correspond à une ligne de l'ensemble de résultats, les noms de colonnes servant de clés.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        List<Vehicle> vehicles = new ArrayList<Vehicle>();
        List<Map> rows = jdbcTemplate.queryForList(sql);
```

```

        for (Map row : rows) {
            Vehicle vehicle = new Vehicle();
            vehicle.setVehicleNo((String) row.get("VEHICLE_NO"));
            vehicle.setColor((String) row.get("COLOR"));
            vehicle.setWheel((Integer) row.get("WHEEL"));
            vehicle.setSeat((Integer) row.get("SEAT"));
            vehicles.add(vehicle);
        }
        return vehicles;
    }
}

```

Dans la classe Main, le code suivant permet de tester notre méthode findAll() :

```

package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        List<Vehicle> vehicles = vehicleDao.findAll();
        for (Vehicle vehicle : vehicles) {
            System.out.println("N° véhicule : " + vehicle.getVehicleNo());
            System.out.println("Couleur : " + vehicle.getColor());
            System.out.println("Roues motrices : " + vehicle.getWheel());
            System.out.println("Places : " + vehicle.getSeat());
        }
    }
}

```

Si nous utilisons un objet RowMapper pour la correspondance des lignes d'un ensemble de résultat, la méthode query() retourne une liste des objets associés.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.BeanPropertyRowMapper;
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        List<Vehicle> vehicles = jdbcTemplate.query(sql,
            BeanPropertyRowMapper.newInstance(Vehicle.class));
        return vehicles;
    }
}

```

Demandeur une seule valeur

Enfin, voyons comment effectuer une interrogation pour obtenir un ensemble de résultat d'une seule ligne et d'une seule colonne. Pour l'exemple, nous ajoutons les opérations suivantes à l'interface de DAO :

```
package com.apress.springrecipes.vehicle;
...
public interface VehicleDao {
    ...
    public String getColor(String vehicleNo);
    public int countAll();
}
```

Pour demander une seule valeur de type chaîne de caractères, nous invoquons la méthode surchargée `queryForObject()`, dont l'argument est de type `java.lang.Class`. Cette méthode permet d'associer le type indiqué à la valeur résultante. Pour les valeurs entières, nous appelons la méthode `queryForInt()`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {
    ...
    public String getColor(String vehicleNo) {
        String sql = "SELECT COLOR FROM VEHICLE WHERE VEHICLE_NO = ?";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        String color = (String) jdbcTemplate.queryForObject(sql,
            new Object[] { vehicleNo }, String.class);
        return color;
    }

    public int countAll() {
        String sql = "SELECT COUNT(*) FROM VEHICLE";
        JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

        int count = jdbcTemplate.queryForInt(sql);
        return count;
    }
}
```

Pour tester ces deux méthodes, nous utilisons le code suivant dans la classe `Main` :

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        int count = vehicleDao.countAll();
        System.out.println("Nb véhicules : " + count);
        String color = vehicleDao.getColor("TEM0001");
        System.out.println("Couleur de [TEM0001] : " + color);
    }
}
```

7.4 Simplifier la création d'un template JDBC

Problème

La création d'une nouvelle instance de `JdbcTemplate` à chaque utilisation est peu efficace, car nous devons répéter l'instruction de création et payer le prix de création d'un nouvel objet.

Solution

La classe `JdbcTemplate` est conçue pour être sûre vis-à-vis des threads. Nous pouvons donc en déclarer une seule instance dans le conteneur IoC et l'injecter dans toutes nos instances d'objets d'accès aux données. Par ailleurs, Spring JDBC offre une classe pratique, `JdbcDaoSupport`, qui simplifie l'implémentation d'un DAO. Elle déclare une propriété `jdbcTemplate` qui peut être injectée depuis le conteneur IoC ou créée automatiquement à partir d'une source de données. Notre DAO peut étendre cette classe pour hériter de cette propriété.

Explications

Injecter un template JDBC

Jusqu'à présent, nous avons créé une nouvelle instance de `JdbcTemplate` dans chaque méthode du DAO. En réalité, il est possible d'injecter cette instance au niveau de la classe et de l'utiliser dans toutes les méthodes du DAO. Pour des raisons de simplicité, le code suivant montre uniquement les modifications de la méthode `insert()`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.JdbcTemplate;

public class JdbcVehicleDao implements VehicleDao {

    private JdbcTemplate jdbcTemplate;

    public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
    }

    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";

        jdbcTemplate.update(sql, new Object[] { vehicle.getVehicleNo(),
            vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
    }
}
```

Un template JDBC a besoin qu'une source de données soit définie. Nous pouvons injecter cette propriété par un mutateur ou un argument du constructeur. Ensuite, nous injectons le template JDBC dans notre objet d'accès aux données.

```
<beans ...>
    ...
    <bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="vehicleDao"
        class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
        <property name="jdbcTemplate" ref="jdbcTemplate" />
    </bean>
</beans>
```

Étendre la classe *JdbcDaoSupport*

En étendant la classe *JdbcDaoSupport*, qui fournit les méthodes *setDataSource()* et *setJdbcTemplate()*, notre classe de DAO hérite de ces deux méthodes. Ensuite, nous pouvons injecter directement un template JDBC ou injecter une source de données pour créer un template JDBC. Le morceau de code suivant est extrait de la classe *JdbcDaoSupport* de Spring :

```
package org.springframework.jdbc.core.support;
...
public abstract class JdbcDaoSupport extends DaoSupport {

    private JdbcTemplate jdbcTemplate;

    public final void setDataSource(DataSource dataSource) {
        this.jdbcTemplate = createJdbcTemplate(dataSource);
        initTemplateConfig();
    }

    public final void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
        this.jdbcTemplate = jdbcTemplate;
        initTemplateConfig();
    }

    public final JdbcTemplate getJdbcTemplate() {
        return jdbcTemplate;
    }
}
```

Dans les méthodes de notre DAO, nous invoquons simplement *getJdbcTemplate()* pour obtenir le template JDBC. Nous supprimons également les propriétés *dataSource* et *jdbcTemplate*, ainsi que les mutateurs correspondants, car elles sont héritées. À nouveau, pour des raisons de simplicité, seules les modifications de la méthode *insert()* sont présentées.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class JdbcVehicleDao extends JdbcDaoSupport implements VehicleDao {

    public void insert(final Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";

        getJdbcTemplate().update(sql, new Object[] { vehicle.getVehicleNo(),
            vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat() });
    }
    ...
}

```

Puisqu'elle étend `JdbcDaoSupport`, la classe de notre DAO hérite de la méthode `setDataSource()`. Nous pouvons injecter une source de données dans notre instance de DAO pour qu'elle crée un template JDBC.

```

<beans ...>
    ...
    <bean id="vehicleDao"
        class="com.apress.springrecipes.vehicle.JdbcVehicleDao">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>

```

7.5 Utiliser le template JDBC simple avec Java 1.5

Problème

La classe `JdbcTemplate` est parfaitement adaptée à de nombreux cas, mais il est possible de l'améliorer pour profiter des fonctionnalités offertes par Java 1.5.

Solution

`SimpleJdbcTemplate` est une évolution de `JdbcTemplate` qui tire profit des fonctionnalités de Java 1.5, comme le boxing automatique, les génériques et le nombre d'arguments variable, pour simplifier son utilisation.

Explications

Mettre à jour une base de données avec un template JDBC simple

Plusieurs méthodes de la classe `JdbcTemplate` attendent que les paramètres de requête soient passés sous forme d'un tableau d'objets. Dans `SimpleJdbcTemplate`, ils peuvent être fournis sous forme d'une liste variable d'arguments, ce qui nous évite de les placer dans un tableau. Pour utiliser `SimpleJdbcTemplate`, nous pouvons l'instancier directement ou en obtenir une instance en étendant la classe `SimpleJdbcDaoSupport`.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport
    implements VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";

        getSimpleJdbcTemplate().update(sql, vehicle.getVehicleNo(),
            vehicle.getColor(), vehicle.getWheel(), vehicle.getSeat());
    }
    ...
}

```

SimpleJdbcTemplate fournit une méthode de mise à jour en série qui prend en arguments une requête SQL et un ensemble de paramètres sous la forme `List<Object[]>`, ce qui nous évite d'implémenter l'interface `BatchPreparedStatementSetter`.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport
    implements VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)";

        List<Object[]> parameters = new ArrayList<Object[]>();
        for (Vehicle vehicle : vehicles) {
            parameters.add(new Object[] { vehicle.getVehicleNo(),
                vehicle.getColor(), vehicle.getWheel(),
                vehicle.getSeat() });
        }
        getSimpleJdbcTemplate().batchUpdate(sql, parameters);
    }
}

```

Interroger une base de données avec un template JDBC simple

Lorsqu'on implémente l'interface `RowMapper`, la valeur de retour de la méthode `mapRow()` est de type `java.lang.Object`. Le paramètre de type de la sous-interface `ParameterizedRowMapper` indique le type de la valeur de retour de `mapRow()`.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedRowMapper;

public class VehicleRowMapper implements ParameterizedRowMapper<Vehicle> {

```

```
public Vehicle mapRow(ResultSet rs, int rowNum) throws SQLException {
    Vehicle vehicle = new Vehicle();
    vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
    vehicle.setColor(rs.getString("COLOR"));
    vehicle.setWheel(rs.getInt("WHEEL"));
    vehicle.setSeat(rs.getInt("SEAT"));
    return vehicle;
}
```

En utilisant `SimpleJdbcTemplate` avec `ParameterizedRowMapper`, il devient inutile de forcer le type du résultat retourné. Pour la méthode `queryForObject()`, le type de retour est déterminé par le paramètre de type de l'objet `ParameterizedRowMapper`, c'est-à-dire `Vehicle` dans notre cas. Les paramètres de la requête doivent être fournis à la fin de la liste des arguments car leur nombre est variable.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport
    implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";

        // Il est dorénavant inutile de forcer le type à Vehicle.
        Vehicle vehicle = getSimpleJdbcTemplate().queryForObject(sql,
            new VehicleRowMapper(), vehicleNo);
        return vehicle;
    }
}
```

Spring 2.5 propose également une implémentation commode de `ParameterizedRowMapper`, `ParameterizedBeanPropertyRowMapper`, qui peut associer automatiquement une ligne à une nouvelle instance de la classe indiquée.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport
    implements VehicleDao {
    ...
    public Vehicle findByVehicleNo(String vehicleNo) {
        String sql = "SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?";

        Vehicle vehicle = getSimpleJdbcTemplate().queryForObject(sql,
            ParameterizedBeanPropertyRowMapper.newInstance(Vehicle.class),
            vehicleNo);
        return vehicle;
    }
}
```

Lorsqu'on utilise la version classique de `JdbcTemplate`, la méthode `findAll()` provoque un avertissement du compilateur Java en raison d'une conversion non vérifiée de `List` en `List<Vehicle>`. En effet, la valeur de retour de la méthode `query()` est de type `List` à la place de la collection typée `List<Vehicle>`. En passant de `SimpleJdbcTemplate` à `ParameterizedBeanPropertyRowMapper`, l'avertissement disparaît immédiatement, car le paramètre de type de la classe `List` est également celui de la classe `ParameterizedRowMapper`.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.ParameterizedBeanPropertyRowMapper;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport
    implements VehicleDao {
    ...
    public List<Vehicle> findAll() {
        String sql = "SELECT * FROM VEHICLE";

        List<Vehicle> vehicles = getSimpleJdbcTemplate().query(sql,
            ParameterizedBeanPropertyRowMapper.newInstance(Vehicle.class));
        return vehicles;
    }
}
```

Lorsque l'interrogation avec `SimpleJdbcTemplate` concerne une seule valeur, le type de la valeur de retour de la méthode `queryForObject()` est déterminé par l'argument `class` (par exemple `String.class`). Il devient donc inutile de forcer manuellement le type. Le nombre de paramètres de la requête étant variable, ils doivent être fournis à la fin de la liste des arguments.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport
    implements VehicleDao {
    ...
    public String getColor(String vehicleNo) {
        String sql = "SELECT COLOR FROM VEHICLE WHERE VEHICLE_NO = ?";

        // Il est dorénavant inutile de forcer le type à String.
        String color = getSimpleJdbcTemplate().queryForObject(sql,
            String.class, vehicleNo);
        return color;
    }
}
```

7.6 Utiliser des paramètres nommés dans un template JDBC

Problème

Dans une utilisation classique de JDBC, les paramètres SQL sont représentés par le caractère ? et leur liaison se fait en fonction des emplacements. Ces paramètres positionnels posent un problème car la modification de leur ordre demande également de changer les liaisons. Lorsqu'une requête SQL contient de nombreux paramètres, la correspondance des paramètres en fonction de leur emplacement devient vite pénible.

Solution

Le framework Spring JDBC propose une autre méthode pour lier les paramètres SQL : utiliser des paramètres nommés. Dans ce cas, les paramètres SQL sont précisés non plus par un emplacement mais par un nom qui commence par des deux-points. De cette manière, la maintenance des paramètres est facilitée et la lisibilité du code, améliorée. À l'exécution, le framework remplace les paramètres nommés par des caractères ?. Seules les classes `SimpleJdbcTemplate` et `NamedParameterJdbcTemplate` reconnaissent les paramètres nommés.

Explications

Lorsque nous utilisons des paramètres nommés dans une requête SQL, nous pouvons donner leur valeur dans une table d'association dont les clés correspondent aux noms des paramètres.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport
    implements VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        Map<String, Object> parameters = new HashMap<String, Object>();
        parameters.put("vehicleNo", vehicle.getVehicleNo());
        parameters.put("color", vehicle.getColor());
        parameters.put("wheel", vehicle.getWheel());
        parameters.put("seat", vehicle.getSeat());

        getSimpleJdbcTemplate().update(sql, parameters);
    }
    ...
}
```

Nous pouvons également indiquer une source de paramètres SQL, dont la fonction est de fournir les valeurs des paramètres nommés. Il existe deux implémentations de l'interface `SqlParameterSource`. La version de base, `MapSqlParameterSource`, représente la source de paramètres avec une table d'association.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.MapSqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport
    implements VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        Map<String, Object> parameters = new HashMap<String, Object>();
        ...
        SqlParameterSource parameterSource =
            new MapSqlParameterSource(parameters);

        getSimpleJdbcTemplate().update(sql, parameterSource);
    }
    ...
}
```

La seconde implémentation de `SqlParameterSource` se nomme `BeanPropertySqlParameterSource`. Elle utilise un objet Java normal comme source de paramètres SQL. La valeur de chaque paramètre nommé est fournie par une propriété de même nom.

```
package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport
    implements VehicleDao {

    public void insert(Vehicle vehicle) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        SqlParameterSource parameterSource =
            new BeanPropertySqlParameterSource(vehicle);

        getSimpleJdbcTemplate().update(sql, parameterSource);
    }
    ...
}
```

Les paramètres nommés sont également utilisables dans une mise à jour en série. Les valeurs des paramètres sont fournies par un tableau de Map ou un tableau de SqlParameterSource.

```

package com.apress.springrecipes.vehicle;
...
import org.springframework.jdbc.core.namedparam.BeanPropertySqlParameterSource;
import org.springframework.jdbc.core.namedparam.SqlParameterSource;
import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcVehicleDao extends SimpleJdbcDaoSupport
    implements VehicleDao {
    ...
    public void insertBatch(List<Vehicle> vehicles) {
        String sql = "INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (:vehicleNo, :color, :wheel, :seat)";

        List<SqlParameterSource> parameters =
            new ArrayList<SqlParameterSource>();
        for (Vehicle vehicle : vehicles) {
            parameters.add(new BeanPropertySqlParameterSource(vehicle));
        }

        getSimpleJdbcTemplate().batchUpdate(sql,
            parameters.toArray(new SqlParameterSource[0]));
    }
}

```

7.7 Modéliser les opérations JDBC avec des objets élémentaires

Problème

Nous souhaitons organiser la logique d'accès aux données sous une forme plus orientée objet en modélisant chaque opération sur la base de données avec un objet élémentaire.

Solution

Outre l'approche template JDBC, le framework Spring JDBC prend en charge la modélisation de chaque opération de base de données par un objet d'opération élémentaire qui étend l'une des classes d'opération JDBC prédéfinies (par exemple SqlUpdate, MappingSqlQuery ou SqlFunction). Avant de pouvoir utiliser un objet d'opération, nous devons réaliser les tâches suivantes :

- préciser la source de données pour cet objet d'opération ;
- préciser la requête SQL pour cet objet d'opération ;
- déclarer les types des paramètres dans l'ordre adéquat pour cet objet d'opération ;
- invoquer la méthode `compile()` sur cet objet d'opération de manière à compiler la requête SQL.

Le meilleur endroit pour effectuer ces tâches reste le constructeur de l'objet d'opération. Dès qu'un objet d'opération a été initialisé en effectuant ces tâches, il est sûr vis-à-vis des threads et nous pouvons donc en déclarer une seule instance dans le conteneur Spring IoC pour l'utiliser ultérieurement.

Explications

Créer un objet d'opération pour une mise à jour

Voyons tout d'abord comment modéliser l'opération d'insertion d'un véhicule sous forme d'un objet d'opération. Pour une opération de mise à jour, nous créons la classe `VehicleInsertOperation` en étendant `SqlUpdate`.

```
package com.apress.springrecipes.vehicle;

import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class VehicleInsertOperation extends SqlUpdate {

    public VehicleInsertOperation(DataSource dataSource) {
        setDataSource(dataSource);
        setSql("INSERT INTO VEHICLE (VEHICLE_NO, COLOR, WHEEL, SEAT) "
            + "VALUES (?, ?, ?, ?)");
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.VARCHAR));
        declareParameter(new SqlParameter(Types.INTEGER));
        declareParameter(new SqlParameter(Types.INTEGER));
        compile();
    }

    public void perform(Vehicle vehicle) {
        update(new Object[] { vehicle.getVehicleNo(), vehicle.getColor(),
            vehicle.getWheel(), vehicle.getSeat() });
    }
}
```

Dès lors que l'objet d'opération de mise à jour est correctement initialisé, nous pouvons invoquer sa méthode `update()` en lui passant les valeurs des paramètres dans un tableau d'objets. Pour une meilleure encapsulation et éviter les paramètres invalides, nous pouvons fournir notre propre méthode `perform()` qui extrait les paramètres depuis un objet `Vehicle`. Nous déclarons ensuite son instance dans le conteneur IoC.

```
<bean id="vehicleInsertOperation"
    class="com.apress.springrecipes.vehicle.VehicleInsertOperation">
    <constructor-arg ref="dataSource" />
</bean>
```

Le code suivant de la classe `Main` illustre l'emploi de cet objet d'opération pour ajouter un nouveau véhicule :

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleInsertOperation operation =
            (VehicleInsertOperation) context.getBean("vehicleInsertOperation");

        Vehicle vehicle = new Vehicle("OBJ0001", "Rouge", 4, 4);
        operation.perform(vehicle);
    }
}
```

Créer un objet d'opération pour une interrogation

Pour un objet d'opération d'interrogation, nous étendons la classe `MappingSqlQuery` de manière à centraliser la logique d'interrogation et de correspondance dans une même classe. En écrivant notre propre méthode `perform()` qui regroupe les paramètres d'interrogation dans un tableau d'objets et force le type de retour, nous améliorons l'encapsulation.

```
package com.apress.springrecipes.vehicle;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.MappingSqlQuery;

public class VehicleQueryOperation extends MappingSqlQuery {

    public VehicleQueryOperation(DataSource dataSource) {
        setDataSource(dataSource);
        setSql("SELECT * FROM VEHICLE WHERE VEHICLE_NO = ?");
        declareParameter(new SqlParameter(Types.VARCHAR));
        compile();
    }

    protected Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Vehicle vehicle = new Vehicle();
        vehicle.setVehicleNo(rs.getString("VEHICLE_NO"));
        vehicle.setColor(rs.getString("COLOR"));
        vehicle.setWheel(rs.getInt("WHEEL"));
        vehicle.setSeat(rs.getInt("SEAT"));
        return vehicle;
    }
}
```

```
    public Vehicle perform(String vehicleNo) {
        return (Vehicle) findObject(new Object[] { vehicleNo });
    }
}
```

Ensuite, il suffit de déclarer une instance de cet objet d'opération dans le conteneur Spring IoC.

```
<bean id="vehicleQueryOperation"
      class="com.apress.springrecipes.vehicle.VehicleQueryOperation">
    <constructor-arg ref="dataSource" />
</bean>
```

Dans la classe Main, nous utilisons cet objet d'opération pour obtenir le véhicule correspondant à un numéro.

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleQueryOperation operation =
            (VehicleQueryOperation) context.getBean("vehicleQueryOperation");
        Vehicle vehicle = operation.perform("OBJ0001");
        System.out.println("N° véhicule : " + vehicle.getVehicleNo());
        System.out.println("Couleur : " + vehicle.getColor());
        System.out.println("Roues motrices : " + vehicle.getWheel());
        System.out.println("Places : " + vehicle.getSeat());
    }
}
```

Créer un objet d'opération pour une fonction

Un objet d'opération de type SqlFunction est utilisé pour obtenir une seule valeur. Par exemple, l'opération permettant d'obtenir le nombre de véhicules peut être modélisée avec SqlFunction. Sa méthode run() retourne le résultat sous forme d'un entier, ce qui convient parfaitement dans notre cas. Si nous souhaitons retourner un résultat d'un autre type, nous devons invoquer la méthode runGeneric().

```
package com.apress.springrecipes.vehicle;

import javax.sql.DataSource;
import org.springframework.jdbc.object.SqlFunction;

public class VehicleCountOperation extends SqlFunction {

    public VehicleCountOperation(DataSource dataSource) {
        setDataSource(dataSource);
        setSql("SELECT COUNT(*) FROM VEHICLE");
        compile();
    }
}
```

```
    public int perform() {
        return run();
    }
}
```

Nous déclarons une instance de cet objet d'opération dans le conteneur Spring IoC.

```
<bean id="vehicleCountOperation"
      class="com.apress.springrecipes.vehicle.VehicleCountOperation">
    <constructor-arg ref="dataSource" />
</bean>
```

Le code suivant de la classe Main teste l'objet d'opération qui interroge la base de données pour obtenir le nombre total de véhicules :

```
package com.apress.springrecipes.vehicle;
...
public class Main {

    public static void main(String[] args) {
        ...
        VehicleCountOperation operation =
            (VehicleCountOperation) context.getBean("vehicleCountOperation");
        int count = operation.perform();
        System.out.println("Nb véhicules : " + count);
    }
}
```

7.8 Gérer les exceptions dans le framework Spring JDBC

Problème

Plusieurs API de JDBC déclarent lancer `java.sql.SQLException`, une exception vérifiée qui doit être interceptée. Il est très pénible de traiter ce type d'exception chaque fois que nous effectuons une opération de base de données. Nous devons souvent définir notre propre stratégie pour gérer ces exceptions, ou nous risquons d'arriver à un traitement incohérent des exceptions.

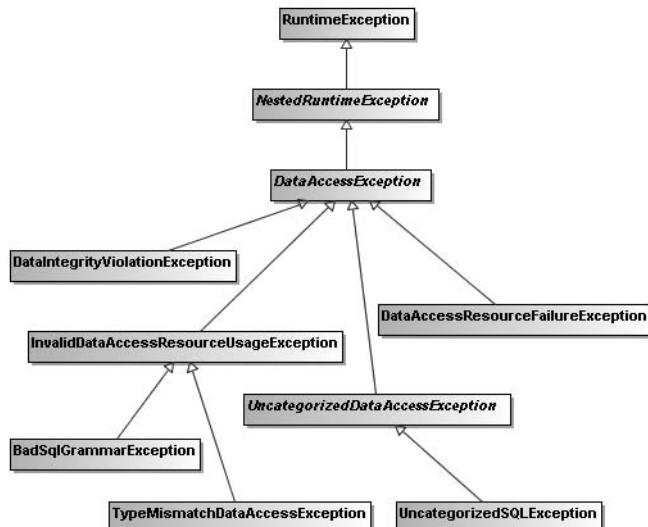
Solution

Le framework Spring propose, pour son module d'accès aux données et notamment le framework JDBC, un mécanisme permettant de traiter de manière cohérente les exceptions d'accès aux données. En général, toutes les exceptions lancées par le framework Spring JDBC sont des sous-classes de `DataAccessException`, une exception de type `RuntimeException` que nous ne sommes pas obligés d'intercepter. Il s'agit de la classe racine de toutes les exceptions dans le module Spring d'accès aux données.

La Figure 7.1 présente une partie de la hiérarchie de `DataAccessException` dans le module d'accès aux données de Spring. Au total, elle contient plus de trente classes définies pour différentes catégories d'exceptions d'accès aux données.

Figure 7.1

Classes d'exception courantes dans la hiérarchie de `DataAccessException`.



Explications

Comprendre la gestion des exceptions dans le framework Spring JDBC

Jusqu'à présent, nous n'avons pas traité explicitement les exceptions JDBC lors de nos utilisations d'un template JDBC ou d'un objet d'opération JDBC. Pour mieux comprendre le mécanisme de gestion des exceptions dans le framework Spring JDBC, examinons le code suivant, extrait de la classe `Main`, qui ajoute un véhicule. Que se passe-t-il si nous insérons un véhicule dont le numéro correspond à un véhicule existant ?

```

package com.apress.springrecipes.vehicle;
...
public class Main {
    ...
    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle = new Vehicle("EX0001", "Vert", 4, 4);
        vehicleDao.insert(vehicle);
    }
}
    
```

Si nous exécutons deux fois la méthode ou si le véhicule est déjà présent dans la base de données, une exception `DataIntegrityViolationException`, qui dérive de `DataAccessException`, est lancée. Dans les méthodes de notre DAO, il est inutile d'entourer le code par un bloc `try/catch` ou de déclarer le lancement d'une exception dans les signatures. En effet, `DataAccessException`, et donc ses sous-classes comme `DataIntegrityViolationException`, est une exception non vérifiée que nous ne sommes pas obligés d'intercepter. La classe parente directe de `DataAccessException` est `NestedRuntimeException`, une classe d'exception au cœur de Spring qui enveloppe une autre exception dans `RuntimeException`.

Lorsque nous employons les classes du framework Spring JDBC, elles interceptent l'exception `SQLException` à notre place et l'enveloppent dans l'une des sous-classes de `DataAccessException`. Puisqu'il s'agit d'une exception `RuntimeException`, nous ne sommes pas obligés de l'intercepter.

Toutefois, comment le framework Spring JDBC peut-il savoir quelle exception concrète de la hiérarchie de `DataAccessException` doit être lancée ? Il examine les propriétés `errorCode` et `SQLState` de l'exception `SQLException` interceptée. Puisqu'une exception `DataAccessException` enveloppe l'exception `SQLException` sous-jacente qui représente la cause initiale, nous pouvons inspecter les propriétés `errorCode` et `SQLState` dans un bloc `catch` :

```
package com.apress.springrecipes.vehicle;
...
import java.sql.SQLException;
import org.springframework.dao.DataAccessException;
public class Main {
    public static void main(String[] args) {
        ...
        VehicleDao vehicleDao = (VehicleDao) context.getBean("vehicleDao");
        Vehicle vehicle = new Vehicle("EX0001", "Vert", 4, 4);
        try {
            vehicleDao.insert(vehicle);
        } catch (DataAccessException e) {
            SQLException sqle = (SQLException) e.getCause();
            System.out.println("Code d'erreur : " + sqle.getErrorCode());
            System.out.println("État SQL : " + sqle.getSQLState());
        }
    }
}
```

Si nous insérons deux fois le même véhicule, Apache Derby retourne le code d'erreur et l'état SQL suivants :

```
Code d'erreur : -1
État SQL : 23505
```

En consultant le manuel de référence d'Apache Derby, nous prenons connaissance de la description de l'erreur, retranscrite dans le Tableau 7.2.

Tableau 7.2 : Description d'une erreur générée par Apache Derby

État SQL	Message
23505	La requête a été annulée car elle conduirait à une valeur de clé dupliquée dans une contrainte de clé unique ou primaire ou un index unique identifié par ' <i>valeur</i> ' défini sur ' <i>valeur</i> '.

Comment le framework Spring JDBC sait-il que l'état 23505 doit correspondre à une exception `DataIntegrityViolationException` ? Le code d'erreur et l'état SQL étant spécifiques à la base de données, chaque système de gestion de bases de données peut retourner des codes différents pour le même type d'erreur. Par ailleurs, certains systèmes indiquent l'erreur dans la propriété `errorCode`, tandis que d'autres, comme Derby, utilisent la propriété `SQLState`.

En tant que framework d'applications Java ouvert, Spring reconnaît les codes d'erreur de la plupart des bases de données répandues. Cependant, en raison du grand nombre de codes d'erreur, il gère la correspondance uniquement pour les erreurs les plus fréquentes. Cette correspondance est définie dans le fichier `sql-error-codes.xml` situé dans le paquetage `org.springframework.jdbc.support` et dont voici un extrait pour Apache Derby :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
 "http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
  ...
  <bean id="Derby"
        class="org.springframework.jdbc.support.SQLExceptionCodes">
    <property name="databaseProductName">
      <value>Apache Derby</value>
    </property>
    <property name="useSqlStateForTranslation">
      <value>true</value>
    </property>
    <property name="badSqlGrammarCodes">
      <value>
        42802,42821,42X01,42X02,42X03,42X04,42X05,42X06,42X07,42X08
      </value>
    </property>
    <property name="dataAccessResourceFailureCodes">
      <value>04501,08004,42Y07</value>
    </property>
  </bean>
</beans>
```

```
<property name="dataIntegrityViolationCodes">
    <value>22001,22005,23502,23503,23505,23513,X0Y32</value>
</property>
<property name="cannotAcquireLockCodes">
    <value>40XL1</value>
</property>
<property name="deadlockLoserCodes">
    <value>40001</value>
</property>
</bean>
</beans>
```

La propriété `databaseProductName` est comparée au nom du système de gestion de bases de données retourné par la méthode `Connection.getMetaData().getDatabaseProductName()`. Cela permet à Spring d'identifier la base de données à laquelle il est connecté. La propriété `useSqlStateForTranslation` signifie que la propriété `SQLState` doit être utilisée à la place d'`errorCode` pour connaître le code d'erreur. Enfin, la classe `SQLExceptionCodes` définit plusieurs catégories pour la correspondance des codes d'erreur. Le code `23505` entre dans la catégorie `dataIntegrityViolationCodes`.

Personnaliser le traitement des exceptions d'accès aux données

Le framework Spring JDBC ne traduit que les codes d'erreur répandus. Parfois, nous devons donc personnaliser nous-mêmes la correspondance. Par exemple, nous pouvons décider d'ajouter d'autres codes à une catégorie existante ou définir une nouvelle exception pour certains codes d'erreur.

Le Tableau 7.2 explique que, dans Apache Derby, le code d'erreur `23505` signale une clé dupliquée. Par défaut, ce code est associé à une exception `DataIntegrityViolationException`. Supposons que nous voulions créer une nouvelle exception pour ces erreurs, `DuplicateKeyException`. Puisqu'il s'agit d'une forme d'erreur de violation de l'intégrité des données, elle doit étendre `DataIntegrityViolationException`. Pour que l'exception puisse être lancée par le framework Spring JDBC, il faut qu'elle soit compatible avec la classe racine des exceptions, `DataAccessException`.

```
package com.apress.springrecipes.vehicle;

import org.springframework.dao.DataIntegrityViolationException;

public class DuplicateKeyException extends DataIntegrityViolationException {

    public DuplicateKeyException(String msg) {
        super(msg);
    }

    public DuplicateKeyException(String msg, Throwable cause) {
        super(msg, cause);
    }
}
```

Par défaut, Spring recherche l'exception dans le fichier `sql-error-codes.xml`, qui se trouve dans le paquetage `org.springframework.jdbc.support`. Cependant, nous pouvons redéfinir certaines correspondances en plaçant un fichier de même nom à la racine du chemin d'accès aux classes. Si Spring trouve notre fichier personnalisé, il commence par y rechercher une correspondance pour l'exception. S'il ne trouve rien d'adéquat, il la recherche dans le fichier par défaut.

Par exemple, supposons que nous voulions que `DuplicateKeyException` corresponde au code d'erreur `23505`. Nous devons établir ce lien à l'aide d'un bean `CustomSQLExceptionCodesTranslation`, que nous ajoutons dans la catégorie `customTranslations`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN 2.0//EN"
http://www.springframework.org/dtd/spring-beans-2.0.dtd">

<beans>
    <bean id="Derby"
          class="org.springframework.jdbc.support.SQLExceptionCodes">
        <property name="databaseProductName">
            <value>Apache Derby</value>
        </property>
        <property name="useSqlStateForTranslation">
            <value>true</value>
        </property>
        <property name="customTranslations">
            <list>
                <ref local="duplicateKeyTranslation" />
            </list>
        </property>
    </bean>

    <bean id="duplicateKeyTranslation"
          class="org.springframework.jdbc.support.➥
CustomSQLExceptionCodesTranslation">
        <property name="errorCodes">
            <value>23505</value>
        </property>
        <property name="exceptionClass">
            <value>
                com.apress.springrecipes.vehicle.DuplicateKeyException
            </value>
        </property>
    </bean>
</beans>
```

À présent, si nous retirons le bloc `try/catch` qui entoure l'opération d'insertion du véhicule et insérons deux fois le même véhicule, le framework Spring JDBC lance une exception `DuplicateKeyException`.

Si le principe de correspondance d'un code à une exception employé par la classe `SQLExceptionCodes` ne convient pas, nous pouvons implémenter l'interface `SQLEceptionTranslator` et injecter son instance dans un template JDBC à l'aide de la méthode `setExceptionTranslator()`.

7.9 En résumé

Dans ce chapitre, nous avons vu comment le framework Spring JDBC simplifie les opérations d'accès à une base de données. Lorsqu'on utilise directement JDBC, sans l'aide de Spring, une grande partie du code JDBC se répète pour chaque opération. Un tel code redondant allonge les méthodes des DAO et les rend moins lisibles.

Au cœur du framework Spring JDBC, le template JDBC fournit des méthodes template pour différents types d'opérations JDBC. Chaque méthode template contrôle la procédure globale, mais nous permet de redéfinir certaines étapes précises en implémentant une interface de rappel prédéfinie.

La classe `JdbcTemplate` correspond au template Spring JDBC classique. Pour bénéficier des fonctionnalités de Java 1.5, Spring fournit `SimpleJdbcTemplate`, qui utilise le boxing automatique, les génériques et le nombre d'arguments variable pour simplifier son utilisation. Cette classe reconnaît également les paramètres nommés dans les requêtes SQL.

Outre l'approche par template JDBC, le framework Spring JDBC prend en charge une approche plus orientée objet qui consiste à modéliser chaque opération de base de données avec un objet d'opération élémentaire. Le choix entre ces deux approches se fonde sur les préférences du développeur et les besoins.

Pour son module d'accès aux données, et notamment le framework JDBC, Spring fournit un mécanisme cohérent de gestion des exceptions d'accès aux données. Lorsqu'une exception `SQLException` est lancée, Spring l'enveloppe dans l'exception appropriée de la hiérarchie de `DataAccessException`. Puisque cette classe correspond à une exception non vérifiée, nous ne sommes pas obligés d'intercepter et de traiter ses instances.

Le chapitre suivant explique comment utiliser les puissantes fonctionnalités de gestion des transactions dans les applications Spring.

8

Gestion des transactions

Au sommaire de ce chapitre

- ✓ Problèmes associés à la gestion des transactions
- ✓ Choisir une implémentation de gestionnaire de transactions
- ✓ Gérer les transactions par programmation avec l'API du gestionnaire de transactions
- ✓ Gérer les transactions par programmation avec un template de transaction
- ✓ Gérer les transactions par déclaration avec Spring AOP classique
- ✓ Gérer les transactions par déclaration avec des greffons transactionnels
- ✓ Gérer les transactions par déclaration avec l'annotation `@Transactional`
- ✓ Fixer l'attribut transactionnel de propagation
- ✓ Fixer l'attribut transactionnel d'isolation
- ✓ Fixer l'attribut transactionnel d'annulation
- ✓ Fixer les attributs transactionnels de temporisation et de lecture seule
- ✓ Gérer les transactions avec le tissage au chargement
- ✓ En résumé

Ce chapitre présente les concepts de base des transactions et les possibilités de Spring dans ce domaine. La gestion des transactions est un élément essentiel dans les applications d'entreprise car elle permet de garantir l'intégrité et la cohérence des données. Spring, en tant que framework d'applications d'entreprise, définit une couche abstraite au-dessus des différentes API de gestion des transactions. En tant que développeurs d'applications, nous pouvons nous servir des outils de Spring pour la gestion des transactions sans vraiment connaître ces API.

À l'instar des approches BMT (*Bean-Managed Transaction*) et CMT (*Container-Managed Transaction*) dans les EJB, la gestion des transactions dans Spring peut se faire par programmation ou par déclaration. Les fonctions de prise en charge des transactions dans Spring offrent une alternative aux transactions des EJB en apportant des possibilités transactionnelles aux POJO.

Dans la *gestion des transactions par programmation*, le code de gestion des transactions est incorporé dans les méthodes métier de manière à contrôler la validation (*commit*) et l'annulation (*rollback*) des transactions. En général, une transaction est validée lorsqu'une méthode se termine de manière normale, elle est annulée lorsqu'une méthode lance certains types d'exceptions. Avec cette gestion des transactions, nous pouvons définir nos propres règles de validation et d'annulation.

Toutefois, cette approche nous oblige à écrire du code de gestion supplémentaire pour chaque opération transactionnelle. Un code transactionnel standard est ainsi reproduit pour chacune des opérations. Par ailleurs, il nous est difficile d'activer et de désactiver la gestion des transactions pour différentes applications. Avec nos connaissances en POA, nous comprenons sans peine que la gestion des transactions est une forme de préoccupation transversale.

Dans la plupart des cas, la *gestion des transactions par déclaration* doit être préférée à la gestion par programmation. Le code de gestion des transactions est séparé des méthodes métier *via* des déclarations. En tant que préoccupation transversale, la gestion des transactions peut être modularisée par une approche POA. Le framework Spring AOP sert de fondation à la gestion déclarative des transactions dans Spring. Nous pouvons ainsi activer plus facilement les transactions dans nos applications et définir une politique transactionnelle cohérente. En revanche, cette forme de gestion est moins souple, car nous ne pouvons pas contrôler précisément les transactions à partir du code.

Grâce à un ensemble d'attributs transactionnels, nous pouvons configurer de manière très fine les transactions. Les attributs reconnus par Spring concernent la propagation, le niveau d'isolation, les règles d'annulation, les temporisations et le fonctionnement en lecture seule. Ils nous permettent ainsi d'adapter le comportement de nos transactions.

À la fin de ce chapitre, vous serez capable d'appliquer différentes stratégies de gestion des transactions dans vos applications. Par ailleurs, les attributs transactionnels vous seront devenus suffisamment familiers pour définir précisément des transactions.

8.1 Problèmes associés à la gestion des transactions

La gestion des transactions est un élément essentiel dans les applications d'entreprise car elle permet de garantir l'intégrité et la cohérence des données. Sans les transactions, les données et les ressources pourraient être endommagées et laissées dans un état incohérent. Cette gestion est extrêmement importante dans les environnements concurrents et répartis lorsque le traitement doit se poursuivre après des erreurs inattendues.

En première définition, une transaction est une suite d'actions qui forment une seule unité de travail. Ces actions doivent toutes réussir ou n'avoir aucun effet. Si toutes les actions réussissent, la transaction est validée de manière permanente. En revanche, si

l'une des actions se passe mal, la transaction est annulée de manière à revenir dans l'état initial, comme si rien ne s'était passé.

Le concept de transactions peut être décrit par les quatre propriétés ACID :

- **Atomicité.** Une transaction est une opération atomique constituée d'une suite d'opérations. L'atomicité d'une transaction garantit que toutes les actions sont entièrement exécutées ou qu'elles n'ont aucun effet.
- **Cohérence.** Dès lors que toutes les actions d'une transaction se sont exécutées, la transaction est validée. Les données et les ressources sont alors dans un état cohérent qui respecte les règles métier.
- **Isolation.** Puisque plusieurs transactions peuvent manipuler le même jeu de données au même moment, chaque transaction doit être isolée des autres afin d'éviter la corruption des données.
- **Durabilité.** Dès lors qu'une transaction est terminée, les résultats doivent survivre à toute panne du système. En général, les résultats d'une transaction sont écrits dans une zone de stockage persistant.

Pour comprendre l'importance de la gestion des transactions, prenons pour exemple l'achat de livres auprès d'une librairie en ligne. Tout d'abord, nous devons créer un nouveau schéma pour cette application dans notre base de données. Pour le moteur de base de données Apache Derby, la connexion se fait à l'aide des propriétés JDBC indiquées dans le Tableau 8.1.

Tableau 8.1 : Propriétés JDBC pour la connexion à la base de données de l'application

Propriété	Valeur
Classe du pilote	org.apache.derby.jdbc.ClientDriver
URL	jdbc:derby://localhost:1527/bookshop;create=true
Nom d'utilisateur	app
Mot de passe	app

Pour l'enregistrement des données de notre application de librairie, nous créons plusieurs tables.

```
CREATE TABLE BOOK (
    ISBN      VARCHAR(50)    NOT NULL,
    BOOK_NAME VARCHAR(100)   NOT NULL,
    PRICE     INT,
    PRIMARY KEY (ISBN)
);
```

```
CREATE TABLE BOOK_STOCK (
    ISBN      VARCHAR(50)  NOT NULL,
    STOCK     INT          NOT NULL,
    PRIMARY KEY (ISBN),
    CHECK (STOCK >= 0)
);

CREATE TABLE ACCOUNT (
    USERNAME  VARCHAR(50)  NOT NULL,
    BALANCE   INT          NOT NULL,
    PRIMARY KEY (USERNAME),
    CHECK (BALANCE >= 0)
);
```

La table `BOOK` contient les informations de base sur un livre, comme son titre et son prix. L'`ISBN` du livre en est la clé primaire. La table `BOOK_STOCK` indique le stock de chaque livre. La quantité en stock est affectée d'une contrainte `CHECK` de manière à rester positive. Bien que la contrainte `CHECK` soit définie dans SQL-99, elle n'est pas reconnue par tous les moteurs de base de données. Si c'est le cas du vôtre, consultez sa documentation pour connaître la contrainte équivalente. Enfin, la table `ACCOUNT` enregistre les comptes des clients et leur solde, qui est garanti positif par une contrainte `CHECK`.

L'interface `BookShop` définit les opérations de notre librairie. Pour le moment, elle ne contient que l'opération `purchase()`.

```
package com.apress.springrecipes.bookshop;

public interface BookShop {

    public void purchase(String isbn, String username);
}
```

Puisque nous implémentons cette interface avec JDBC, nous créons la classe `JdbcBookShop` suivante. Pour mieux comprendre la nature des transactions, procédons sans l'aide de Spring JDBC.

```
package com.apress.springrecipes.bookshop;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

import javax.sql.DataSource;

public class JdbcBookShop implements BookShop {

    private DataSource dataSource;

    public void setDataSource(DataSource dataSource) {
        this.dataSource = dataSource;
    }
```

```
public void purchase(String isbn, String username) {  
    Connection conn = null;  
    try {  
        conn = dataSource.getConnection();  
  
        PreparedStatement stmt1 = conn.prepareStatement(  
            "SELECT PRICE FROM BOOK WHERE ISBN = ?");  
        stmt1.setString(1, isbn);  
        ResultSet rs = stmt1.executeQuery();  
        rs.next();  
        int price = rs.getInt("PRICE");  
        stmt1.close();  
  
        PreparedStatement stmt2 = conn.prepareStatement(  
            "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 " +  
            "WHERE ISBN = ?");  
        stmt2.setString(1, isbn);  
        stmt2.executeUpdate();  
        stmt2.close();  
  
        PreparedStatement stmt3 = conn.prepareStatement(  
            "UPDATE ACCOUNT SET BALANCE = BALANCE - ? " +  
            "WHERE USERNAME = ?");  
        stmt3.setInt(1, price);  
        stmt3.setString(2, username);  
        stmt3.executeUpdate();  
        stmt3.close();  
    } catch (SQLException e) {  
        throw new RuntimeException(e);  
    } finally {  
        if (conn != null) {  
            try {  
                conn.close();  
            } catch (SQLException e) {}  
        }  
    }  
}
```

Pour l'opération `purchase()`, nous exécutons trois requêtes SQL. La première obtient le prix du livre, tandis que la deuxième et la troisième mettent à jour le stock du livre et ajustent le solde du compte en conséquence.

Nous déclarons ensuite une instance de la librairie dans le conteneur Spring IoC de manière à offrir les services d'achat. Pour simplifier, nous utilisons un `DriverManagerDataSource` qui ouvre une nouvelle connexion à la base de données à chaque requête.

INFO

Pour accéder à une base de données du serveur Derby, vous devez inclure le fichier `derby-client.jar` (situé dans le répertoire `lib` de l'installation de Derby) dans le chemin d'accès aux classes.

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
                  value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
                  value="jdbc:derby://localhost:1527/bookshop;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

    <bean id="bookShop" class="com.apress.springrecipes.bookshop.JdbcBookShop">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>

```

Afin d'illustrer les problèmes qui peuvent survenir sans la gestion des transactions, supposons que la base de données de notre librairie contienne les données présentées dans les Tableaux 8.2 à 8.4.

Tableau 8.2 : Données d'exemple de la table BOOK pour le test des transactions

<i>ISBN</i>	<i>BOOK_NAME</i>	<i>PRICE</i>
0001	Le premier livre	30

Tableau 8.3 : Données d'exemple de la table BOOK_STOCK pour le test des transactions

<i>ISBN</i>	<i>STOCK</i>
0001	10

Tableau 8.4 : Données d'exemple de la table ACCOUNT pour le test des transactions

<i>USERNAME</i>	<i>BALANCE</i>
utilisateur1	20

Dans la classe Main suivante, le client utilisateur1 achète le livre dont l'ISBN est 0001. Puisque ce client ne dispose que de 20 € sur son compte, il ne peut pas acheter le livre.

```

package com.apress.springrecipes.bookshop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

```

```
public class Main {  
  
    public static void main(String[] args) {  
        ApplicationContext context =  
            new ClassPathXmlApplicationContext("beans.xml");  
  
        BookShop bookShop = (BookShop) context.getBean("bookShop");  
        bookShop.purchase("0001", "utilisateur1");  
    }  
}
```

Si nous exécutons cette application, nous recevons une exception `SQLException` car la contrainte `CHECK` sur la table `ACCOUNT` n'est pas respectée. Ce résultat était attendu, puisque nous avons tenté un débit supérieur au solde du compte. Toutefois, si nous examinons le stock de ce livre dans la table `BOOK_STOCK`, nous constatons qu'il a été diminué par cette opération ratée ! En effet, nous avons exécuté la deuxième requête SQL pour diminuer le stock avant de recevoir l'exception générée par la troisième.

L'absence d'une gestion des transactions a donc placé nos données dans un état incohérent. Pour éviter cette situation, nos trois requêtes SQL de l'opération `purchase()` doivent être exécutées au sein d'une même transaction. Si l'une des actions de la transaction échoue, l'intégralité de la transaction est annulée pour défaire les changements apportés par les actions exécutées.

Gérer les transactions avec la validation et l'annulation de JDBC

Lorsque la mise à jour d'une base de données se fait avec JDBC, chaque requête SQL est, par défaut, validée immédiatement après son exécution. Ce comportement se nomme *validation automatique*. Il ne nous permet pas de gérer des transactions dans nos opérations.

JDBC prend en charge une stratégie élémentaire de gestion des transactions qui consiste à invoquer explicitement les méthodes `commit()` et `rollback()` sur une connexion. Cependant, nous devons commencer par désactiver la validation automatique, qui est active par défaut.

```
package com.apress.springrecipes.bookshop;  
...  
public class JdbcBookShop implements BookShop {  
    ...  
    public void purchase(String isbn, String username) {  
        Connection conn = null;  
        try {  
            conn = dataSource.getConnection();  
            conn.setAutoCommit(false);  
            ...  
            conn.commit();  
        } catch (SQLException e) {  
            if (conn != null) {
```

```
        try {
            conn.rollback();
        } catch (SQLException e1) {}
    }
    throw new RuntimeException(e);
} finally {
    if (conn != null) {
        try {
            conn.close();
        } catch (SQLException e) {}
    }
}
```

La méthode `setAutoCommit()` permet de configurer le fonctionnement en validation automatique d'une connexion à une base de données. Par défaut, la validation automatique est active de manière à valider chaque requête SQL immédiatement après son exécution. Pour gérer les transactions, nous devons désactiver ce comportement par défaut et valider la connexion uniquement lorsque toutes les requêtes SQL ont été exécutées avec succès. Si l'une des requêtes échoue, nous devons annuler toutes les modifications apportées par cette connexion.

À présent, si nous exécutons à nouveau notre application, le stock du livre n'est pas affecté lorsque le solde d'un utilisateur est insuffisant pour acheter l'ouvrage.

Bien que nous puissions gérer les transactions en validant et en annulant explicitement les connexions JDBC, le code correspondant est un code standard que nous devons répéter dans différentes méthodes. Par ailleurs, ce code est propre à JDBC et doit être changé si nous décidons d'employer une autre technologie d'accès aux données. Pour simplifier la gestion des transactions, Spring propose donc un ensemble d'outils transactionnels indépendants de la technologie, en particulier des gestionnaires de transactions, un template de transaction et la gestion des transactions par déclaration.

8.2 Choisir une implémentation de gestionnaire de transactions

Problème

De manière générale, si notre application n'emploie qu'une seule source de données, nous pouvons simplement gérer les transactions en invoquant les méthodes `commit()` et `rollback()` sur la connexion à la base de données. En revanche, si les transactions concernent plusieurs sources de données ou si nous préférons utiliser les possibilités de gestion des transactions apportées par le serveur d'applications Java EE, nous pouvons opter pour JTA (*Java Transaction API*). Par ailleurs, nous risquons d'avoir à invoquer différentes API transactionnelles propriétaires en fonction des différents frameworks de correspondance objet-relationnel, comme Hibernate et JPA.

En conséquence, nous devons manipuler différentes API transactionnelles pour différentes technologies. Il est alors difficile de passer d'un jeu d'API à l'autre.

Solution

Spring propose un ensemble d'outils transactionnels généraux pour différentes API de gestion des transactions. En tant que développeurs d'applications, nous pouvons simplement employer ces fonctions sans vraiment connaître les API transactionnelles sous-jacentes. Ainsi, notre code de gestion des transactions devient indépendant de la technologie transactionnelle employée.

L'abstraction au cœur de la gestion des transactions dans Spring se nomme `PlatformTransactionManager`. Cette interface encapsule un ensemble de méthodes indépendantes de la technologie pour la gestion des transactions. Il ne faut pas oublier qu'un gestionnaire de transactions est nécessaire quelle que soit la stratégie de gestion des transactions (par programmation ou par déclaration) choisie dans Spring.

Explications

`PlatformTransactionManager` est une interface générale pour tous les gestionnaires de transactions de Spring. Spring propose plusieurs implémentations de cette interface qui sont utilisées avec différentes API de gestion des transactions :

- Si, dans notre application, nous accédons à une seule source de données au travers de JDBC, `DataSourceTransactionManager` doit répondre à nos besoins.
- Si nous employons JTA pour la gestion des transactions sur un serveur d'applications Java EE, nous devons choisir `JtaTransactionManager` pour obtenir une transaction à partir du serveur d'applications.
- Si nous utilisons un framework de correspondance objet-relationnel pour accéder à une base de données, nous devons choisir le gestionnaire transactionnel adapté à ce framework, comme `HibernateTransactionManager` ou `JpaTransactionManager`.

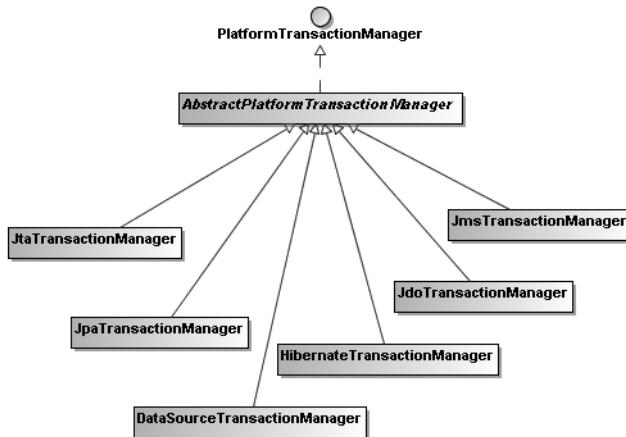
La Figure 8.1 présente les implémentations communes de l'interface `PlatformTransactionManager` dans Spring.

Pour déclarer un gestionnaire de transactions dans le conteneur Spring IoC, nous procérons comme pour un bean normal. Par exemple, la configuration de bean suivante déclare une instance de `DataSourceTransactionManager`. Il est indispensable de fixer la propriété `dataSource` pour qu'elle puisse gérer les transactions sur cette source de données.

```
<bean id="transactionManager"
      class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Figure 8.1

Implémentations communes de l'interface PlatformTransactionManager.



8.3 Gérer les transactions par programmation avec l'API du gestionnaire de transactions

Problème

Nous avons besoin d'un contrôle précis sur la validation et l'annulation des transactions dans nos méthodes métier, mais nous ne souhaitons pas accéder directement à l'API transactionnelle sous-jacente.

Solution

Dans Spring, un gestionnaire de transactions offre une API indépendante de la technologie qui nous permet de débuter une nouvelle transaction en invoquant la méthode `getTransaction()` et de la gérer par l'intermédiaire des méthodes `commit()` et `rollback()`. Puisque `PlatformTransactionManager` est une entité abstraite pour la gestion des transactions, les méthodes appelées sont indépendantes de la technologie.

Explications

Pour illustrer l'utilisation de l'API de gestionnaire de transactions, nous créons une nouvelle classe, `TransactionalJdbcBookShop`, qui se fonde sur le template Spring JDBC. Puisqu'elle doit travailler avec un gestionnaire de transactions, nous ajoutons une propriété de type `PlatformTransactionManager`, dont l'injection peut se faire par un mutateur.

```

package com.apress.springrecipes.bookshop;

import org.springframework.dao.DataAccessViolationException;
import org.springframework.jdbc.core.support.JdbcDaoSupport;
import org.springframework.transaction.PlatformTransactionManager;
  
```

```
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;

public class TransactionalJdbcBookShop extends JdbcDaoSupport
    implements BookShop {

    private PlatformTransactionManager transactionManager;

    public void setTransactionManager(
        PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void purchase(String isbn, String username) {
        TransactionDefinition def = new DefaultTransactionDefinition();
        TransactionStatus status = transactionManager.getTransaction(def);

        try {
            int price = getJdbcTemplate().queryForInt(
                "SELECT PRICE FROM BOOK WHERE ISBN = ?",
                new Object[] { isbn });

            getJdbcTemplate().update(
                "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 " +
                "WHERE ISBN = ?", new Object[] { isbn });

            getJdbcTemplate().update(
                "UPDATE ACCOUNT SET BALANCE = BALANCE - ? " +
                "WHERE USERNAME = ?",
                new Object[] { price, username });

            transactionManager.commit(status);
        } catch (DataAccessException e) {
            transactionManager.rollback(status);
            throw e;
        }
    }
}
```

Avant de débuter une nouvelle transaction, nous devons préciser les attributs transactionnels dans un objet de définition de transaction de type `TransactionDefinition`. Dans notre exemple, nous créons simplement une instance de `DefaultTransactionDefinition` pour utiliser les attributs transactionnels par défaut.

Après avoir défini une transaction, nous demandons au gestionnaire de transactions de démarrer une nouvelle transaction correspondant à cette définition en invoquant la méthode `getTransaction()`. Elle retourne un objet `TransactionStatus` qui nous permet de suivre l'état de la transaction. Si toutes les requêtes s'exécutent avec succès, nous demandons au gestionnaire de valider la transaction en lui passant l'objet d'état. Puisque toutes les exceptions lancées par le template Spring JDBC dérivent de la classe `DataAccessException`, nous demandons au gestionnaire d'annuler la transaction dès que nous recevons ce type d'exception.

Dans cette classe, la propriété pour le gestionnaire de transactions est du type général `PlatformTransactionManager`. Nous devons cependant injecter une implémentation de gestionnaire adaptée. Puisque nous manipulons une seule source de données et y accédons au travers de JDBC, nous choisissons `DataSourceTransactionManager`.

```
<beans ...>
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.TransactionalJdbcBookShop">
        <property name="dataSource" ref="dataSource" />
        <property name="transactionManager" ref="transactionManager" />
    </bean>
</beans>
```

8.4 Gérer les transactions par programmation avec un template de transaction

Problème

Supposons qu'un bloc de code d'une méthode de métier, non l'intégralité du corps de cette méthode, ait le comportement transactionnel suivant :

- démarrer une nouvelle transaction au début du bloc ;
- valider la transaction après exécution réussie du bloc ;
- annuler la transaction en cas de réception d'une exception dans le bloc.

Si nous invoquons directement l'API d'un gestionnaire de transactions de Spring, le code de gestion des transactions peut être généralisé de manière indépendante de la technologie. Cependant, nous ne souhaitons pas répéter ce code standard pour chaque bloc de code semblable.

Solution

Outre le template JDBC, Spring fournit un template de transaction pour nous aider à contrôler le processus global de gestion des transactions. Nous devons simplement encapsuler le bloc de code concerné dans une classe de rappel qui implémente l'interface `TransactionCallback` et la passer au template de transaction pour son exécution. De cette manière, nous n'avons pas à reproduire le code standard de gestion des transactions pour ce bloc.

Explications

Un template de transaction est créé pour un gestionnaire de transactions, tout comme un template JDBC est créé pour une source de données. Un template de transaction exécute un objet de rappel qui encapsule un bloc de code transactionnel. Nous pouvons implémenter l'interface de rappel par un classe séparée ou une classe interne. Dans ce dernier cas, les arguments de la méthode doivent être déclarés `final` pour que la classe puisse y accéder.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.*;
    TransactionCallbackWithoutResult;
import org.springframework.transaction.support.TransactionTemplate;

public class TransactionalJdbcBookShop extends JdbcDaoSupport
    implements BookShop {

    private PlatformTransactionManager transactionManager;

    public void setTransactionManager(
        PlatformTransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    public void purchase(final String isbn, final String username) {
        TransactionTemplate transactionTemplate =
            new TransactionTemplate(transactionManager);

        transactionTemplate.execute(new TransactionCallbackWithoutResult() {

            protected void doInTransactionWithoutResult(
                TransactionStatus status) {

                int price = getJdbcTemplate().queryForInt(
                    "SELECT PRICE FROM BOOK WHERE ISBN = ?",
                    new Object[] { isbn });

                getJdbcTemplate().update(
                    "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 " +
                    "WHERE ISBN = ?", new Object[] { isbn });

                getJdbcTemplate().update(
                    "UPDATE ACCOUNT SET BALANCE = BALANCE - ? " +
                    "WHERE USERNAME = ?",
                    new Object[] { price, username });
            }
        });
    }
}
```

Un template de transaction accepte un objet de rappel qui implémente soit l'interface `TransactionCallback`, soit l'interface `TransactionCallbackWithoutResult`. Puisque le bloc de code qui ajuste le stock du livre et le solde du compte dans `purchase()` n'a aucun résultat à retourner, `TransactionCallbackWithoutResult` convient parfaitement. Lorsque le bloc de code doit retourner une valeur, nous devons implémenter à la place l'interface `TransactionCallback`. La valeur de retour de l'objet de rappel est renvoyée à la méthode `execute()` du template.

Pendant l'exécution de l'objet de rappel, si une exception non vérifiée est lancée (catégorie dans laquelle entrent `RuntimeException` et `DataAccessException`) ou si nous invoquons `setRollbackOnly()` sur l'argument `TransactionStatus`, la transaction est annulée. Sinon elle est validée dès la fin de l'exécution de l'objet de rappel.

Dans le fichier de configuration des beans, le bean de la librairie a toujours besoin d'un gestionnaire de transactions pour créer un template de transaction.

```
<beans ...>
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.TransactionalJdbcBookShop">
        <property name="dataSource" ref="dataSource" />
        <property name="transactionManager" ref="transactionManager" />
    </bean>
</beans>
```

Nous pouvons également demander au conteneur IoC d'injecter le template de transaction au lieu de le créer directement. Puisque le template suffit à la gestion des transactions, notre classe n'a plus besoin de faire référence au gestionnaire de transactions.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.support.TransactionTemplate;

public class TransactionalJdbcBookShop extends JdbcDaoSupport
    implements BookShop {

    private TransactionTemplate transactionTemplate;

    public void setTransactionTemplate(
        TransactionTemplate transactionTemplate) {
        this.transactionTemplate = transactionTemplate;
    }

    public void purchase(final String isbn, final String username) {
        transactionTemplate.execute(new TransactionCallbackWithoutResult() {
```

```
protected void doInTransactionWithoutResult(
    TransactionStatus status) {
    ...
}
```

Nous définissons ensuite un template de transaction dans le fichier de configuration des beans et l'injectons à la place du gestionnaire de transactions dans notre bean de librairie. L'instance du template de transaction peut être utilisée par plusieurs beans transactionnels car elle est sûre vis-à-vis des threads. Enfin, il ne faut pas oublier de définir la propriété du gestionnaire de transactions dans le template de transaction.

```
<beans ...>
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="transactionTemplate"
        class="org.springframework.transaction.support.TransactionTemplate">
        <property name="transactionManager" ref="transactionManager" />
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.TransactionalJdbcBookShop">
        <property name="dataSource" ref="dataSource" />
        <property name="transactionTemplate" ref="transactionTemplate" />
    </bean>
</beans>
```

8.5 Gérer les transactions par déclaration avec Spring AOP classique

Problème

Puisque la gestion des transactions est une forme de préoccupation transversale, la programmation orientée aspect constitue une bonne approche pour sa mise en œuvre. Toutefois, si nous voulons gérer les transactions de manière déclarative dans Spring 1.x, nous devons choisir Spring AOP classique.

Solution

Spring fournit un greffon Around classique nommé `TransactionInterceptor` pour la gestion des transactions. À l'instar d'un template de transaction, il permet de contrôler le processus de gestion des transactions, mais il s'applique à l'intégralité du corps de la méthode, non à un bloc de code arbitraire. Par défaut, avant que l'exécution de la

méthode ne commence, ce greffon démarre une nouvelle transaction. Si la méthode lance une exception non vérifiée, la transaction est annulée. Sinon elle est validée dès la méthode terminée.

Dans Spring AOP classique, pour appliquer le greffon `TransactionInterceptor`, nous devons créer un proxy de bean en utilisant `ProxyFactoryBean`. Puisque la gestion des transactions est une constante dans les applications d'entreprise, Spring fournit la classe `TransactionProxyFactoryBean` spécialisée dans la création de proxies transactionnels.

Explications

Tout d'abord, voyons comment implémenter notre librairie à l'aide de Spring JDBC, mais sans le code de gestion des transactions.

```
package com.apress.springrecipes.bookshop;

import org.springframework.jdbc.core.support.JdbcDaoSupport;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {

    public void purchase(String isbn, String username) {
        int price = getJdbcTemplate().queryForInt(
            "SELECT PRICE FROM BOOK WHERE ISBN = ?",
            new Object[] { isbn });

        getJdbcTemplate().update(
            "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 " +
            "WHERE ISBN = ?",
            new Object[] { isbn });

        getJdbcTemplate().update(
            "UPDATE ACCOUNT SET BALANCE = BALANCE - ? " +
            "WHERE USERNAME = ?",
            new Object[] { price, username });
    }
}
```

Dans le fichier de configuration des beans, nous déclarons un greffon `TransactionInterceptor` qui fait référence au gestionnaire de transactions. Nous créons ensuite un proxy avec `ProxyFactoryBean` pour appliquer ce greffon au bean cible `bookShop`.

```
<beans ...>
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.JdbcBookShop">
        <property name="dataSource" ref="dataSource" />
    </bean>
```

```
<bean id="transactionInterceptor" class="org.springframework.transaction➥
interceptor.TransactionInterceptor">
    <property name="transactionManager" ref="transactionManager" />
    <property name="transactionAttributes">
        <props>
            <prop key="purchase">PROPAGATION_REQUIRED</prop>
        </props>
    </property>
</bean>

<bean id="bookShopProxy"
    class="org.springframework.aop.framework.ProxyFactoryBean">
    <property name="target" ref="bookShop" />
    <property name="interceptorNames">
        <list>
            <value>transactionInterceptor</value>
        </list>
    </property>
</bean>
</beans>
```

Les méthodes qui requièrent une gestion des transactions sont indiquées dans la propriété `transactionAttributes` du greffon `TransactionInterceptor`. Cette propriété est de type `Properties`, avec les noms des méthodes transactionnelles servant de clés et les attributs transactionnels servant de valeurs. Elle reconnaît les caractères génériques dans les clés afin que nous puissions donner les mêmes attributs transactionnels à plusieurs méthodes. Pour chaque méthode transactionnelle, nous devons préciser au moins l'attribut de propagation. Pour des raisons de simplicité, nous choisissons la valeur par défaut `PROPAGATION_REQUIRED`.

Nos méthodes doivent être invoquées sur le proxy pour que la gestion des transactions puisse avoir lieu. Par conséquent, dans la classe `Main`, nous demandons le bean `bookShopProxy` à la place du bean `bookShop`.

```
package com.apress.springrecipes.bookshop;
...
public class Main {

    public static void main(String[] args) {
        ...
        BookShop bookShop = (BookShop) context.getBean("bookShopProxy");
        bookShop.purchase("0001", "utilisateur1");
    }
}
```

Une autre solution se fonde sur la classe `TransactionProxyFactoryBean`, spécialisée dans la création de proxies transactionnels. Nous pouvons alors combiner les informations nécessaires au greffon `TransactionInterceptor` et à la déclaration de `ProxyFactoryBean` dans une seule déclaration de `TransactionProxyFactoryBean`.

```
<bean id="bookShopProxy" class="org.springframework.transaction➥
interceptor.TransactionProxyFactoryBean">
```

```

<property name="target" ref="bookShop" />
<property name="transactionManager" ref="transactionManager" />
<property name="transactionAttributes">
    <props>
        <prop key="purchase">PROPAGATION_REQUIRED</prop>
    </props>
</property>
</bean>

```

8.6 Gérer les transactions par déclaration avec des greffons transactionnels

Problème

Puisque la gestion des transactions est une forme de préoccupation transversale, nous souhaitons effectuer cette gestion de manière déclarative avec la nouvelle approche POA de Spring 2.x.

Solution

Spring 2.x fournit un greffon transactionnel facile à configurer à l'aide de l'élément `<tx:advice>` défini dans le schéma `tx`. Ce greffon peut être activé avec les fonctions de configuration de la POA définies dans le schéma `aop`.

Explications

Pour activer la gestion déclarative des transactions dans Spring 2.x, nous pouvons déclarer un greffon transactionnel via l'élément `<tx:advice>` du schéma `tx`. Par conséquent, nous devons au préalable ajouter la définition de ce schéma dans l'élément racine `<beans>`. Après avoir déclaré ce greffon, nous devons l'associer à un point d'action. Puisqu'un greffon transactionnel est déclaré en dehors de l'élément `<aop:config>`, il ne peut pas être lié directement à un point d'action. Nous devons déclarer un conseiller dans l'élément `<aop:config>` pour associer un greffon à un point d'action.


INFO

Puisque Spring 2.x AOP utilise des expressions AspectJ pour définir des points d'action, vous devez inclure le fichier `aspectjweaver.jar` (situé dans le répertoire `lib/aspectj` de l'installation de Spring) dans le chemin d'accès aux classes.

```

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

```

```
xsi:schemaLocation="http://www.springframework.org/schema/beans
    http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
    http://www.springframework.org/schema/tx
    http://www.springframework.org/schema/tx/spring-tx-2.5.xsd
    http://www.springframework.org/schema/aop
    http://www.springframework.org/schema/aop/spring-aop-2.5.xsd">

<tx:advice id="bookShopTxAdvice"
    transaction-manager="transactionManager">
    <tx:attributes>
        <tx:method name="purchase" />
    </tx:attributes>
</tx:advice>

<aop:config>
    <aop:pointcut id="bookShopOperation" expression=
        "execution(* com.apress.springrecipes.bookshop.BookShop.*(..))" />
    <aop:advisor advice-ref="bookShopTxAdvice"
        pointcut-ref="bookShopOperation" />
</aop:config>
...
<bean id="transactionManager"
    class="org.springframework.jdbc.➥
        datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>

<bean id="bookShop"
    class="com.apress.springrecipes.bookshop.JdbcBookShop">
    <property name="dataSource" ref="dataSource" />
</bean>
</beans>
```

L’expression de point d’action précédente désigne toutes les méthodes déclarées dans l’interface BookShop. Toutefois, en raison de son approche fondée sur un proxy, Spring AOP ne prend en charge que les méthodes publiques. Autrement dit, avec Spring AOP, seules les méthodes publiques peuvent devenir transactionnelles.

Chaque greffon transactionnel a besoin d’un identifiant et d’une référence à un gestionnaire de transactions dans le conteneur IoC. Les méthodes qui requièrent une gestion des transactions sont précisées par de multiples éléments `<tx:method>` dans l’élément `<tx:attributes>`. Les caractères génériques sont acceptés dans le nom de méthode afin de désigner un groupe de méthodes. Nous pouvons également définir des attributs transactionnels pour chaque groupe de méthodes, mais, dans notre exemple, nous conservons les attributs par défaut.

Nous pouvons à présent obtenir le bean bookShop à partir du conteneur Spring IoC. Puisque les méthodes de ce bean sont désignées par le point d’action, Spring retourne un proxy pour lequel la gestion des transactions est activée.

```
package com.apress.springrecipes.bookshop;
...
public class Main {
    public static void main(String[] args) {
        ...
        BookShop bookShop = (BookShop) context.getBean("bookShop");
        bookShop.purchase("0001", "utilisateur1");
    }
}
```

8.7 Gérer les transactions par déclaration avec l'annotation *@Transactional*

Problème

Pour déclarer des transactions dans le fichier de configuration des beans, il faut comprendre les concepts de la POA, comme les points d'action, les greffons et les conseillers. Les développeurs qui ne possèdent pas ces connaissances ont du mal à mettre en place la gestion déclarative des transactions.

Solution

Outre la déclaration des transactions dans le fichier des configurations des beans avec des points d'action, des greffons et des conseillers, Spring nous permet de déclarer des transactions en marquant simplement les méthodes transactionnelles avec l'annotation `@Transactional` et en activant l'élément `<tx:annotation-driven>`. Toutefois, cette approche nécessite l'emploi de Java 1.5 ou une version ultérieure.

Explications

Pour indiquer le caractère transactionnel d'une méthode, nous la marquons simplement avec l'annotation `@Transactional`. Puisque Spring AOP se fonde sur les proxies, seules les méthodes publiques doivent être annotées.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {

    @Transactional
    public void purchase(String isbn, String username) {
        ...
    }
}
```

L'annotation `@Transactional` peut être appliquée au niveau de la méthode ou de la classe. Lorsqu'elle concerne une classe, toutes ses méthodes publiques sont considérées comme transactionnelles. Bien qu'il soit possible d'appliquer `@Transactional` à des interfaces ou à des déclarations de méthodes dans une interface, cette pratique n'est pas conseillée car elle peut ne pas fonctionner correctement avec les proxies basés sur les classes (c'est-à-dire les proxies CGLIB).

Dans le fichier de configuration des beans, nous ajoutons simplement un élément `<tx:annotation-driven>` en lui indiquant un gestionnaire de transactions. Spring greffe les méthodes marquées par `@Transactional` ou les méthodes d'une classe marquée par `@Transactional`, pour les beans déclarés dans le conteneur IoC. Spring est ainsi capable de gérer les transactions pour ces méthodes.

```
<beans ...>
    <tx:annotation-driven transaction-manager="transactionManager" />
    ...
    <bean id="transactionManager"
        class="org.springframework.jdbc.datasource.➥
        DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>

    <bean id="bookShop"
        class="com.apress.springrecipes.bookshop.JdbcBookShop">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

Si le gestionnaire de transactions se nomme `transactionManager`, nous pouvons omettre l'attribut `transaction-manager` dans l'élément `<tx:annotation-driven>` car celui-ci détecte automatiquement tout gestionnaire de transactions de ce nom. Nous devons préciser un gestionnaire de transactions uniquement si son nom est différent.

```
<beans ...>
    <tx:annotation-driven />
    ...
</beans>
```

8.8 Fixer l'attribut transactionnel de propagation

Problème

Lorsqu'une méthode transactionnelle est invoquée par une autre méthode, il est nécessaire de préciser la manière dont la transaction est propagée. Par exemple, la méthode peut continuer à s'exécuter au sein de la transaction existante ou elle peut démarrer une nouvelle transaction et s'exécuter à l'intérieur de celle-ci.

Solution

L'attribut transactionnel de *propagation* nous permet de préciser la manière dont se fait la propagation des transactions. Spring définit sept comportements de propagation (voir Tableau 8.5). Ils sont définis dans l'interface `org.springframework.transaction.TransactionDefinition`. Certains gestionnaires de transactions peuvent ne pas prendre en charge l'ensemble de ces comportements de propagation.

Tableau 8.5 : Comportements de la propagation reconnus par Spring

<i>Propagation</i>	<i>Description</i>
REQUIRED	S'il existe une transaction en cours, la méthode actuelle doit s'exécuter au sein de cette transaction. Sinon elle doit démarrer une nouvelle transaction et s'exécuter dans celle-ci.
REQUIRES_NEW	La méthode actuelle doit démarrer une nouvelle transaction et s'exécuter dans celle-ci. S'il existe une transaction en cours, elle doit être suspendue.
SUPPORTS	S'il existe une transaction en cours, la méthode actuelle peut s'exécuter au sein de cette transaction. Sinon elle n'est pas obligée de s'exécuter dans une transaction.
NOT_SUPPORTED	La méthode actuelle ne doit pas s'exécuter au sein d'une transaction. S'il existe une transaction en cours, elle doit être suspendue.
MANDATORY	La méthode actuelle doit s'exécuter au sein d'une transaction. S'il n'existe aucune transaction en cours, une exception est lancée.
NEVER	La méthode actuelle ne doit pas s'exécuter au sein d'une transaction. S'il existe une transaction en cours, une exception est lancée.
NESTED	S'il existe une transaction en cours, la méthode actuelle doit s'exécuter au sein d'une transaction imbriquée (prise en charge par la fonctionnalité de point de sauvegarde de JDBC 3.0) dans cette transaction. Sinon elle doit démarrer une nouvelle transaction et s'exécuter dans celle-ci.

Explications

La propagation de transaction se produit lorsqu'une méthode transactionnelle est invoquée par une autre méthode. Par exemple, supposons qu'un client souhaite payer tous les livres achetés sur la librairie en ligne. Pour mettre en œuvre cette opération, nous définissons l'interface `Cashier` suivante :

```
package com.apress.springrecipes.bookshop;
...
public interface Cashier {
    public void checkout(List<String> isbn, String username);
}
```

Nous pouvons implémenter cette interface en déléguant les achats à un bean de librairie dont la méthode `purchase()` sera invoquée plusieurs fois. Il est bon de remarquer que la méthode `checkout()` est rendue transactionnelle en la marquant avec `@Transactional`.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Transactional;

public class BookShopCashier implements Cashier {

    private BookShop bookShop;

    public void setBookShop(BookShop bookShop) {
        this.bookShop = bookShop;
    }

    @Transactional
    public void checkout(List<String> isbn, String username) {
        for (String isbn : isbn) {
            bookShop.purchase(isbn, username);
        }
    }
}
```

Ensuite, nous définissons un bean de caisse dans notre fichier de configuration et faisons référence au bean de librairie pour l'achat des livres.

```
<bean id="cashier"
      class="com.apress.springrecipes.bookshop.BookShopCashier">
    <property name="bookShop" ref="bookShop" />
</bean>
```

Pour illustrer la propagation d'une transaction, nous remplissons la base avec les données présentées dans les Tableaux 8.6 à 8.8.

Tableau 8.6 : Données d'exemple de la table BOOK pour le test de la propagation

ISBN	BOOK_NAME	PRICE
0001	Le premier livre	30
0002	Le deuxième livre	50

Tableau 8.7 : Données d'exemple de la table BOOK_STOCK pour le test de la propagation

ISBN	STOCK
0001	10
0002	10

Tableau 8.8 : Données d'exemple de la table ACCOUNT pour le test de la propagation

USERNAME	BALANCE
utilisateur1	40

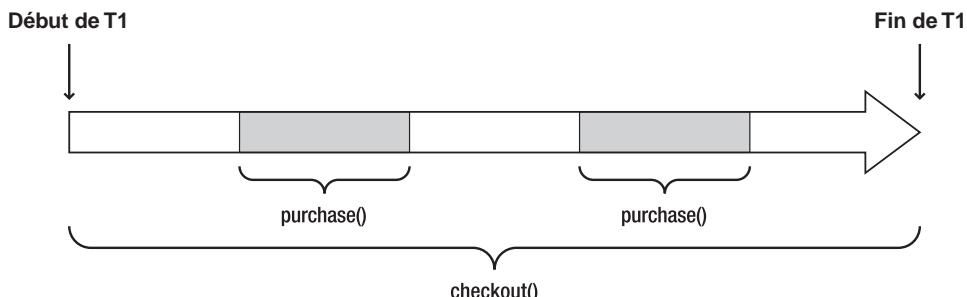
La propagation REQUIRED

Lorsque le client utilisateur1 passe à la caisse avec ses deux livres, le solde est suffisant pour acheter le premier mais pas le second.

```
package com.apress.springrecipes.bookshop;
...
public class Main {

    public static void main(String[] args) {
        ...
        Cashier cashier = (Cashier) context.getBean("cashier");
        List<String> isbnList =
            Arrays.asList(new String[] { "0001", "0002" });
        cashier.checkout(isbnList, "utilisateur1");
    }
}
```

Si la méthode `purchase()` de la librairie est invoquée par une autre méthode transactionnelle, comme `checkout()`, elle s'exécute par défaut dans la transaction existante. Ce comportement par défaut de la propagation se nomme REQUIRED. Dans ce cas, il existe une seule transaction dont les limites sont fixées par le début et la fin de la méthode `checkout()`. Cette transaction ne sera pas validée avant la fin de la méthode `checkout()`. Par conséquent, l'utilisateur ne peut acheter aucun des livres. La Figure 8.2 illustre le comportement REQUIRED de la propagation.

**Figure 8.2**

Propagation des transactions selon le comportement REQUIRED.

En revanche, si la méthode `purchase()` est appelée par une méthode non transactionnelle et si aucune transaction n'est en cours, elle démarre une nouvelle transaction et s'exécute dans sa propre transaction.

L'attribut de propagation peut être défini dans l'annotation `@Transactional`. Par exemple, voici comment fixer le comportement REQUIRED avec cet attribut. En réalité, cette précision est inutile car il s'agit du comportement par défaut.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(propagation = Propagation.REQUIRED)
    public void purchase(String isbn, String username) {
        ...
    }
}

---

package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class BookShopCashier implements Cashier {
    ...
    @Transactional(propagation = Propagation.REQUIRED)
    public void checkout(List<String> isbns, String username) {
        ...
    }
}
```

La propagation REQUIRED_NEW

Le comportement `REQUIRED_NEW` de la programmation est également très utilisé. Il indique que la méthode doit démarrer une nouvelle transaction et s'exécuter dans celle-ci. Si une transaction est déjà en cours, elle est suspendue.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(propagation = Propagation.REQUIRED_NEW)
    public void purchase(String isbn, String username) {
        ...
    }
}
```

Cet exemple met en scène trois transactions. La première est démarrée par la méthode `checkout()`, mais, lorsque la première méthode `purchase()` est invoquée, elle est suspendue et une nouvelle transaction est démarrée. À la fin de la première méthode `purchase()`, la nouvelle transaction se termine et est validée. Lorsque la seconde méthode `purchase()` est invoquée, une autre nouvelle transaction est démarrée. Cependant, elle échoue et est annulée. Par conséquent, le premier livre est bien acheté, contrairement au second.

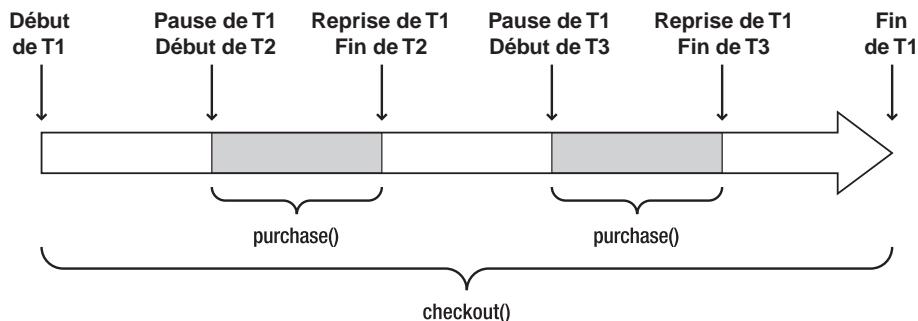


Figure 8.3

Propagation des transactions selon le comportement `REQUIRES_NEW`.

Fixer l'attribut de propagation dans les greffons transactionnels, les proxies et les API

Dans un greffon transactionnel Spring 2.x, nous définissons l'attribut transactionnel de propagation dans l'élément `<tx:method>`.

```
<tx:advice ...>
  <tx:attributes>
    <tx:method name="..." propagation="REQUIRES_NEW" />
  </tx:attributes>
</tx:advice>
```

Dans Spring AOP classique, cet attribut peut être précisé dans les attributs transactionnels du `TransactionInterceptor` et du `TransactionProxyFactoryBean`.

```
<property name="transactionAttributes">
  <props>
    <prop key="... ">PROPAGATION_REQUIRES_NEW</prop>
  </props>
</property>
```

Dans l'API de gestion des transactions de Spring, l'attribut transactionnel de propagation peut être fixé dans un objet `DefaultTransactionDefinition` et passé ensuite à la

méthode `getTransaction()` d'un gestionnaire de transactions ou à un constructeur de template de transaction.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRES_NEW);
```

8.9 Fixer l'attribut transactionnel d'isolation

Problème

Lorsque plusieurs transactions de la même application ou d'applications différentes opèrent de manière concurrente sur le même jeu de données, des problèmes inattendus peuvent survenir. Nous devons préciser l'isolation de nos transactions l'une par rapport à l'autre.

Solution

Il existe trois catégories de problèmes provoqués par des transactions concurrentes :

- **Lecture sale.** Pour deux transactions T1 et T2, T1 lit un champ qui a été mis à jour par T2 mais pas encore validé. Plus tard, si T2 est annulée, la valeur du champ lu par T1 était en réalité une valeur temporaire désormais invalide.
- **Lecture non reproductible.** Pour deux transactions T1 et T2, T1 lit un champ et T2 met ensuite ce champ à jour. Plus tard, si T1 lit à nouveau le même champ, la valeur est différente.
- **Lecture fantôme.** Pour deux transactions T1 et T2, T1 lit quelques lignes d'une table, puis T2 insère de nouvelles lignes dans la table. Plus tard, si T1 lit à nouveau la même table, il existe des lignes supplémentaires.

En théorie, les transactions doivent être totalement isolées les unes des autres (c'est-à-dire sérialisables) pour éviter les problèmes précédents. Cependant, un tel niveau d'isolation a un impact important sur les performances car les transactions s'exécutent alors l'une après l'autre. Dans la pratique, les transactions peuvent s'exécuter à des niveaux d'isolation plus faibles de manière à améliorer les performances.

Le niveau d'isolation d'une transaction peut être fixé par l'attribut transactionnel *d'isolation*. Spring reconnaît cinq niveaux d'isolation (voir Tableau 8.9), définis dans l'interface `org.springframework.transaction.TransactionDefinition`.

L'isolation des transactions est prise en charge par le moteur de base de données sous-jacent, non par l'application ou le framework. Toutefois, tous les niveaux d'isolation ne sont pas forcément reconnus par tous les moteurs de bases de données. Le niveau d'isolation d'une connexion JDBC est défini en appelant la méthode `setTransactionIsolation()`.

Tableau 8.9 : Niveaux d'isolation reconnus par Spring

<i>Isolation</i>	<i>Description</i>
DEFAULT	Le niveau d'isolation par défaut de la base de données sous-jacente est utilisé. Pour la plupart des bases de données, le niveau d'isolation par défaut est READ_COMMITTED.
READ_UNCOMMITTED	Une transaction est autorisée à lire les modifications non validées effectuées par d'autres transactions. Des problèmes de lecture sale, de lecture non reproductible et de lecture fantôme peuvent survenir.
READ_COMMITTED	Une transaction est autorisée à lire uniquement les modifications validées effectuées par d'autres transactions. Le problème de lecture sale peut être évité, mais les lectures non reproductibles et fantômes sont toujours possibles.
REPEATABLE_READ	Une transaction qui lit plusieurs fois un champ obtient des valeurs identiques. Pour la durée de cette transaction, les mises à jour du champ par d'autres transactions sont interdites. Les problèmes de lecture sale et de lecture non reproductible peuvent être évités, mais les lectures fantômes sont toujours possibles.
SERIALIZABLE	Une transaction qui lit plusieurs fois une table obtient des lignes identiques. Pour la durée de cette transaction, les insertions, les mises à jour et les suppressions sur la table sont interdites depuis d'autres transactions. Tous les problèmes de concurrence peuvent être évités, mais les performances sont ralenties.

Explications

Pour illustrer les problèmes provoqués par des transactions concurrentes, ajoutons deux nouvelles opérations à notre librairie pour augmenter et vérifier le stock d'un livre.

```
package com.apress.springrecipes.bookshop;

public interface BookShop {
    ...
    public void increaseStock(String isbn, int stock);
    public int checkStock(String isbn);
}
```

Nous implémentons ces opérations de la manière suivante, sans oublier de les déclarer transactionnelles.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
}
```

```
@Transactional
public void increaseStock(String isbn, int stock) {
    String threadName = Thread.currentThread().getName();
    System.out.println(threadName
        + " - Prêt à augmenter le stock du livre");

    getJdbcTemplate().update(
        "UPDATE BOOK_STOCK SET STOCK = STOCK + ? " +
        "WHERE ISBN = ?",
        new Object[] { stock, isbn });

    System.out.println(threadName + " - Stock du livre augmenté de "
        + stock);
    sleep(threadName);

    System.out.println(threadName + " - Augmentation du stock annulée");
    throw new RuntimeException("Augmentation par erreur");
}

@Transactional
public int checkStock(String isbn) {
    String threadName = Thread.currentThread().getName();
    System.out.println(threadName
        + " - Prêt à vérifier le stock du livre");

    int stock = getJdbcTemplate().queryForInt(
        "SELECT STOCK FROM BOOK_STOCK WHERE ISBN = ?",
        new Object[] { isbn });

    System.out.println(threadName + " - Le stock du livre est " + stock);
    sleep(threadName);

    return stock;
}

private void sleep(String threadName) {
    System.out.println(threadName + " - En sommeil");
    try {
        Thread.sleep(10000);
    } catch (InterruptedException e) {}
    System.out.println(threadName + " - Réveillé");
}
```

Pour simuler la concurrence, nos opérations sont exécutées par plusieurs threads. Nous sommes informés de l'état courant des opérations par l'intermédiaire des instructions `println`. Pour chaque opération, nous affichons des messages sur la console, avant et après l'exécution des requêtes SQL. Ces messages incluent le nom du thread pour que nous sachions lequel effectue l'opération.

Après que chaque opération a exécuté la requête SQL, nous demandons au thread de s'endormir pendant dix secondes. Nous l'avons expliqué précédemment, la transaction est validée ou annulée immédiatement après la fin de l'opération. En insérant une instruction `sleep`, nous retardons la validation ou l'annulation.

Pour l'opération `increase()`, nous lançons une exception `RuntimeException` afin d'annuler la transaction.

Avant de tester ces exemples de niveaux d'isolation, nous saisissons les données des Tableaux 8.10 et 8.11 dans la base.

Tableau 8.10 : Données d'exemple de la table BOOK pour le test des niveaux d'isolation

ISBN	BOOK_NAME	PRICE
0001	Le premier livre	30

Tableau 8.11 : Données d'exemple de la table BOOK_STOCK pour le test des niveaux d'isolation

ISBN	STOCK
0001	10

Les niveaux d'isolation READ_UNCOMMITTED et READ_COMMITTED

`READ_UNCOMMITTED` correspond au niveau d'isolation le plus faible. Il autorise une transaction à lire des modifications non validées effectuées par d'autres transactions. Nous fixons ce niveau d'isolation dans l'annotation `@Transactional` de la méthode `checkStock()`.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.READ_UNCOMMITTED)
    public int checkStock(String isbn) {
        ...
    }
}
```

Nous pouvons créer plusieurs threads pour mener des tests sur ce niveau d'isolation. La classe `Main` suivante crée deux threads. Le thread 1 augmente le stock du livre, tandis que le thread 2 vérifie ce stock. Le thread 2 démarre cinq secondes après le thread 1.

```
package com.apress.springrecipes.bookshop;
...
public class Main {

    public static void main(String[] args) {
        ...
        final BookShop bookShop = (BookShop) context.getBean("bookShop");
    }
}
```

```
Thread thread1 = new Thread(new Runnable() {
    public void run() {
        try {
            bookShop.increaseStock("0001", 5);
        } catch (RuntimeException e) {}
    }
}, "Thread 1");

Thread thread2 = new Thread(new Runnable() {
    public void run() {
        bookShop.checkStock("0001");
    }
}, "Thread 2");

thread1.start();
try {
    Thread.sleep(5000);
} catch (InterruptedException e) {}
thread2.start();
}
```

L'exécution de l'application produit le résultat suivant :

```
Thread 1 - Prêt à augmenter le stock du livre
Thread 1 - Stock du livre augmenté de 5
Thread 1 - En sommeil
Thread 2 - Prêt à vérifier le stock du livre
Thread 2 - Le stock du livre est 15
Thread 2 - En sommeil
Thread 1 - Réveillé
Thread 1 - Augmentation du stock annulée
Thread 2 - Réveillé
```

Le thread 1 a tout d'abord augmenté le stock du livre, puis s'est endormi. À ce moment-là, la transaction du thread 1 n'a pas encore été annulée. Pendant que le thread 1 était endormi, le thread 2 a démarré et a lu le stock du livre. Avec le niveau d'isolation **READ_UNCOMMITTED**, le thread 2 est autorisé à lire la valeur de stock qui a été mise à jour par une transaction non validée.

Cependant, lorsque le thread 1 s'est réveillé, sa transaction a été annulée en raison d'une exception `RuntimeException`. Par conséquent, la valeur lue par le thread 2 n'était que temporaire et est à présent invalide. Ce problème est dit de *lecture sale*, car une transaction peut lire des valeurs qui sont "sales".

Pour éviter ce problème, nous devons augmenter le niveau d'isolation de `checkStock()` à `READ_COMMITTED`.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;
```

```

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.READ_COMMITTED)
    public int checkStock(String isbn) {
        ...
    }
}

```

Si nous exécutons à nouveau l'application, le thread 2 n'est pas en mesure de lire le stock du livre tant que le thread 1 n'a pas annulé la transaction. Ainsi, le problème de lecture sale est évité en empêchant qu'une transaction ne lise un champ dont la valeur a été mise à jour par une autre transaction non validée.

```

Thread 1 - Prêt à augmenter le stock du livre
Thread 1 - Stock du livre augmenté de 5
Thread 1 - En sommeil
Thread 2 - Prêt à vérifier le stock du livre
Thread 1 - Réveillé
Thread 1 - Augmentation du stock annulée
Thread 2 - Le stock du livre est 10
Thread 2 - En sommeil
Thread 2 - Réveillé

```

Pour que la base de données sous-jacente prenne en charge le niveau d'isolation READ_COMMITTED, elle doit acquérir un *verrou de mise à jour* sur une ligne qui a été modifiée mais non encore validée. Ensuite, les autres transactions qui veulent lire cette ligne doivent patienter jusqu'à ce que le verrou de mise à jour soit libéré, ce qui se produit lorsque la première transaction est validée ou annulée.

Le niveau d'isolation REPEATABLE_READ

Restructurons à présent les threads de manière à illustrer un autre problème de concurrence. Nous échangeons le rôle des deux threads : le thread 1 vérifie le stock du livre, avant que le thread 2 ne l'augmente.

```

package com.apress.springrecipes.bookshop;
...
public class Main {

    public static void main(String[] args) {
        ...
        final BookShop bookShop = (BookShop) context.getBean("bookShop");

        Thread thread1 = new Thread(new Runnable() {
            public void run() {
                bookShop.checkStock("0001");
            }
        }, "Thread 1");

        Thread thread2 = new Thread(new Runnable() {
            public void run() {
                try {
                    bookShop.increaseStock("0001", 5);
                }
            }
        }, "Thread 2");
    }
}

```

```
        } catch (RuntimeException e) {}
    }
}, "Thread 2");

thread1.start();
try {
    Thread.sleep(5000);
} catch (InterruptedException e) {}
thread2.start();
}
}
```

L'exécution de l'application produit le résultat suivant :

```
Thread 1 - Prêt à vérifier le stock du livre
Thread 1 - Le stock du livre est 10
Thread 1 - En sommeil
Thread 2 - Prêt à augmenter le stock du livre
Thread 2 - Stock du livre augmenté de 5
Thread 2 - En sommeil
Thread 1 - Réveillé
Thread 2 - Réveillé
Thread 2 - Augmentation du stock annulée
```

Le thread 1 a tout d'abord lu le stock du livre, puis s'est endormi. À ce moment-là, la transaction du thread 1 n'a pas encore été validée. Pendant que le thread 1 est endormi, le thread 2 a démarré et a augmenté le stock du livre. Avec le niveau d'isolation **READ_COMMITTED**, le thread 2 est capable de mettre à jour la valeur du stock qui a été lue par une transaction non validée.

Cependant, si le thread 1 lit à nouveau le stock, la valeur est différente de celle obtenue lors de la première lecture. Ce problème est dit de *lecture non reproductible*, car une transaction peut obtenir des valeurs différentes pour le même champ.

Pour éviter ce problème, nous augmentons le niveau d'isolation de `checkStock()` à **REPEATABLE_READ**.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(isolation = Isolation.REPEATABLE_READ)
    public int checkStock(String isbn) {
        ...
    }
}
```

Si nous exécutons à nouveau l'application, le thread 2 n'est pas en mesure d'actualiser le stock du livre tant que le thread 1 n'a pas validé la transaction. Ainsi, le problème de lecture non reproductible peut être évité en empêchant qu'une transaction ne mette à jour une valeur qui a été lue par une autre transaction non validée.

```
Thread 1 - Prêt à vérifier le stock du livre
Thread 1 - Le stock du livre est 10
Thread 1 - En sommeil
Thread 2 - Prêt à augmenter le stock du livre
Thread 1 - Réveillé
Thread 2 - Stock du livre augmenté de 5
Thread 2 - En sommeil
Thread 2 - Réveillé
Thread 2 - Augmentation du stock annulée
```

Pour que la base de données sous-jacente prenne en charge le niveau d'isolation REPEATABLE_READ, elle doit acquérir un *verrou de lecture* sur une ligne qui a été lue mais non encore validée. Ensuite, les autres transactions qui veulent mettre à jour cette ligne doivent patienter jusqu'à ce que le verrou de lecture soit libéré, ce qui se produit lorsque la première transaction est validée ou annulée.

Le niveau d'isolation SERIALIZABLE

Après qu'une transaction a lu plusieurs lignes d'une table, une autre transaction insère de nouvelles lignes dans la même table. Si la première transaction lit à nouveau cette table, elle obtient des lignes supplémentaires différentes de la première lecture. Ce problème est dit de *lecture fantôme*. En réalité, une lecture fantôme s'apparente à une lecture non reproductible, mais elle concerne plusieurs lignes.

Pour éviter ce problème, nous devons choisir le niveau d'isolation le plus élevé : SERIALIZABLE. Il s'agit également du niveau d'isolation le plus lent car il peut être nécessaire d'acquérir un verrou de lecture sur l'intégralité de la table. Dans la pratique, il faut toujours choisir le niveau d'isolation le plus faible qui répond aux exigences.

Fixer l'attribut d'isolation dans les greffons transactionnels, les proxies et les API

Dans un greffon transactionnel Spring 2.x, nous définissons le niveau d'isolation dans l'élément <tx:method>.

```
<tx:advice ...>
    <tx:attributes>
        <tx:method name="..." 
            isolation="REPEATABLE_READ" />
    </tx:attributes>
</tx:advice>
```

Dans Spring AOP classique, ce niveau d'isolation peut être précisé dans les attributs transactionnels de TransactionInterceptor et de TransactionProxyFactoryBean.

```
<property name="transactionAttributes">
    <props>
        <prop key="...">
            PROPAGATION_REQUIRED, ISOLATION_REPEATABLE_READ
        </prop>
    </props>
</property>
```

Dans l'API de gestion des transactions de Spring, le niveau d'isolation peut être fixé dans un objet `DefaultTransactionDefinition` et passé ensuite à la méthode `getTransaction()` d'un gestionnaire de transactions ou à un constructeur de template de transaction.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setIsolationLevel(TransactionDefinition.ISOLATION_REPEATABLE_READ);
```

8.10 Fixer l'attribut transactionnel d'annulation

Problème

Par défaut, seules les exceptions non vérifiées, c'est-à-dire de type `RuntimeException` et `Error`, provoquent l'annulation d'une transaction. Parfois, nous souhaitons transgresser cette règle et faire en sorte que nos propres exceptions conduisent à l'annulation des transactions.

Solution

L'attribut transactionnel d'*annulation* permet d'indiquer les exceptions qui déclenchent l'annulation d'une transaction. Toutes les exceptions qui ne sont pas explicitement précisées dans cet attribut sont soumises à la règle d'annulation par défaut : annulation uniquement pour les exceptions non vérifiées.

Explications

Les attributs `rollbackFor` et `noRollbackFor` de l'annotation `@Transactional` permettent de fixer les règles d'annulation d'une transaction. Ces deux attributs étant de type `Class[]`, nous pouvons préciser plusieurs exceptions dans chacun d'eux.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Propagation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(
        propagation = Propagation.REQUIRES_NEW,
        rollbackFor = IOException.class,
        noRollbackFor = ArithmeticException.class)
    public void purchase(String isbn, String username) {
        ...
    }
}
```

Dans un greffon transactionnel Spring 2.x, la règle d'annulation est définie dans l'élément `<tx:method>`. Pour préciser plusieurs exceptions, elles doivent être séparées par des virgules.

```
<tx:advice ...>
    <tx:attributes>
        <tx:method name="..." ...
            rollback-for="java.io.IOException"
            no-rollback-for="java.lang.ArithmetricException" />
        ...
    </tx:attributes>
</tx:advice>
```

Dans Spring AOP, la règle d'annulation est établie dans les attributs transactionnels de `TransactionInterceptor` et de `TransactionProxyFactoryBean`. Le signe moins indique une exception qui provoque l'annulation de la transaction, tandis que le signe plus indique une exception qui provoque la validation de la transaction.

```
<property name="transactionAttributes">
    <props>
        <prop key="...">
            PROPAGATION_REQUIRED, -java.io.IOException,
            +java.lang.ArithmetricException
        </prop>
    </props>
</property>
```

Dans l'API de gestion des transactions de Spring, la règle d'annulation peut être fixée dans un objet `RuleBasedTransactionAttribute`. Puisqu'il implémente l'interface `TransactionDefinition`, il peut ensuite être passé à la méthode `getTransaction()` d'un gestionnaire de transactions ou à un constructeur de template de transaction.

```
RuleBasedTransactionAttribute attr = new RuleBasedTransactionAttribute();
attr.getRollbackRules().add(
    new RollbackRuleAttribute(IOException.class));
attr.getRollbackRules().add(
    new NoRollbackRuleAttribute(SendFailedException.class));
```

8.11 Fixer les attributs transactionnels de temporisation et de lecture seule

Problème

Puisqu'une transaction peut détenir des verrous sur des lignes et des tables, une longue transaction immobilisera des ressources et aura un impact sur les performances globales. Par ailleurs, si une transaction lit uniquement des données, sans procéder à des mises à jour, le moteur de base de données peut optimiser cette transaction. Nous pouvons définir les attributs correspondants de manière à améliorer les performances de l'application.

Solution

L’attribut transactionnel de *temporisation* fixe la durée de vie d’une transaction avant qu’elle ne subisse une annulation forcée. Cela permet d’éviter qu’une longue transaction n’immobilise des ressources. L’attribut de *lecture seule* indique que la transaction se contente de lire des données, sans effectuer de mises à jour. Le moteur de base de données peut ainsi optimiser la transaction.

Explications

Les attributs de temporisation et de lecture seule peuvent être définis dans l’annotation `@Transactional`. La temporisation est indiquée en secondes.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.transaction.annotation.Isolation;
import org.springframework.transaction.annotation.Transactional;

public class JdbcBookShop extends JdbcDaoSupport implements BookShop {
    ...
    @Transactional(
        isolation = Isolation.REPEATABLE_READ,
        timeout = 30,
        readOnly = true)
    public int checkStock(String isbn) {
        ...
    }
}
```

Dans un greffon transactionnel Spring 2.x, nous définissons les attributs de temporisation et de lecture seule dans l’élément `<tx:method>`.

```
<tx:advice ...>
    <tx:attributes>
        <tx:method name="..." 
            timeout="30"
            read-only="true" />
    </tx:attributes>
</tx:advice>
```

Dans Spring AOP classique, ces attributs peuvent être précisés dans les attributs transactionnels de `TransactionInterceptor` et de `TransactionProxyFactoryBean`.

```
<property name="transactionAttributes">
    <props>
        <prop key="...">
            PROPAGATION_REQUIRED, timeout_30, readOnly
        </prop>
    </props>
</property>
```

Dans l'API de gestion des transactions de Spring, les attributs de temporisation et de lecture seule peuvent être fixés dans un objet `DefaultTransactionDefinition` passé ensuite à la méthode `getTransaction()` d'un gestionnaire de transactions ou à un constructeur de template de transaction.

```
DefaultTransactionDefinition def = new DefaultTransactionDefinition();
def.setTimeout(30);
def.setReadOnly(true);
```

8.12 Gérer les transactions avec le tissage au chargement

Problème

Par défaut, la gestion déclarative des transactions dans Spring est activée par son framework POA. Puisque Spring AOP ne peut greffer que les méthodes publiques des beans déclarés dans le conteneur IoC, la gestion des transactions se limite à cette portée. Parfois, nous souhaitons gérer des transactions pour des méthodes non publiques ou des méthodes d'objets créés hors du conteneur Spring IoC, comme des objets de domaine.

Solution

Spring 2.5 fournit un aspect AspectJ nommé `AnnotationTransactionAspect` qui peut gérer des transactions pour n'importe quelle méthode de n'importe quel objet, même dans le cas de méthodes non publiques ou d'objets créés hors du conteneur Spring IoC. Cet aspect se charge des transactions pour les méthodes marquées par `@Transactional`. Son activation peut se faire par un tissage à la compilation ou au chargement.

Explications

Tout d'abord, créons la classe de domaine `Book`, dont les instances (des objets de domaine) peuvent être créées hors du conteneur Spring IoC.

```
package com.apress.springrecipes.bookshop;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Configurable;
import org.springframework.jdbc.core.JdbcTemplate;

@Configuration
public class Book {

    private String isbn;
    private String name;
    private int price;

    // Constructeurs, accesseurs et mutateurs.
    ...

    private JdbcTemplate jdbcTemplate;
```

```
@Autowired
public void setJdbcTemplate(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
}

public void purchase(String username) {
    jdbcTemplate.update(
        "UPDATE BOOK_STOCK SET STOCK = STOCK - 1 " +
        "WHERE ISBN = ?",
        new Object[] { isbn });

    jdbcTemplate.update(
        "UPDATE ACCOUNT SET BALANCE = BALANCE - ? " +
        "WHERE USERNAME = ?",
        new Object[] { price, username });
}
```

Cette classe de domaine déclare une méthode `purchase()` qui réduit le stock de l'instance de livre courante et le solde du compte du client dans la base de données. Pour exploiter les fonctionnalités JDBC de Spring, nous injectons le template JDBC par un mutateur.

Nous pouvons exploiter le tissage au chargement de Spring pour injecter un template JDBC dans des objets de domaine de type `Book`. Nous devons annoter cette classe avec `@Configurable` pour déclarer que ce type d'objet est configurable dans le conteneur Spring IoC. Par ailleurs, nous annotons le mutateur du template JDBC avec `@Autowired` pour bénéficier de la liaison automatique.

La bibliothèque des aspects de Spring propose un aspect `AspectJ`, `AnnotationBeanConfiguratorAspect`, pour la configuration des dépendances d'objets, même si ces objets sont créés hors du conteneur IoC. Pour activer cet aspect, nous ajoutons simplement l'élément `<context:spring-configured>` dans le fichier de configuration des beans. Pour tisser cet aspect dans nos classes de domaine au chargement, nous ajoutons également l'élément `<context:load-time-weaver>`. Enfin, pour lier automatiquement le template JDBC dans des objets de domaine de type `Book` avec `@Autowired`, nous avons également besoin de `<context:annotation-config>`.

INFO

Pour utiliser la bibliothèque des aspects de Spring, vous devez inclure le fichier `spring-aspects.jar` (situé dans le répertoire `dist/weaving` de l'installation de Spring) dans le chemin d'accès aux classes.

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

```

http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:load-time-weaver />

<context:annotation-config />

<context:spring-configured />

<bean id="dataSource"
      class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
              value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url"
              value="jdbc:derby://localhost:1527/bookshop;create=true" />
    <property name="username" value="app" />
    <property name="password" value="app" />
  </bean>

  <bean id="jdbcTemplate"
        class="org.springframework.jdbc.core.JdbcTemplate">
    <property name="dataSource" ref="dataSource" />
  </bean>
</beans>

```

Dans ce fichier de configuration des beans, nous définissons un template JDBC sur une source de données. Il est lié automatiquement aux objets de domaine Book pour qu'ils puissent accéder à la base de données. Nous créons la classe Main suivante pour tester cette classe de domaine. Les transactions ne sont pas encore prises en charge.

```

package com.apress.springrecipes.bookshop;

import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans.xml");

        Book book = new Book("0001", "Le premier livre", 30);
        book.purchase("utilisateur1");
    }
}

```

Dans le cas d'une application Java simple, nous pouvons tisser cet aspect dans nos classes au chargement en passant l'agent Spring en argument de la machine virtuelle.

```

java -javaagent:c:/spring-framework-2.5.6/dist/weaving/spring-agent.jar
com.apress.springrecipes.bookshop.Main

```

Pour activer la gestion des transactions sur une méthode d'un objet de domaine, nous la marquons simplement avec @Transactional, comme c'était le cas pour les méthodes des beans Spring.

```
package com.apress.springrecipes.bookshop;
...
import org.springframework.beans.factory.annotation.Configurable;
import org.springframework.transaction.annotation.Transactional;

@Configurable
public class Book {
    ...
    @Transactional
    public void purchase(String username) {
        ...
    }
}
```

Enfin, pour que `AnnotationTransactionAspect` intervienne dans la gestion des transactions, nous définissons simplement l'élément `<tx:annotation-driven>` en fixant son mode à `aspectj`. Cela active automatiquement l'aspect transactionnel. Nous devons également préciser un gestionnaire de transactions pour cet aspect. Par défaut, il recherche un gestionnaire de transactions nommé `transactionManager`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    ...
    <tx:annotation-driven mode="aspectj" />
    ...
    <bean id="transactionManager"
          class="org.springframework.jdbc.datasource.➥
          DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
    </bean>
</beans>
```

8.13 En résumé

Ce chapitre a montré l'importance de la gestion des transactions pour l'intégrité et la cohérence des données. Sans cette gestion, nos données et nos ressources peuvent être endommagées et laissées dans un état incohérent.

L'abstraction au cœur de la gestion des transactions dans Spring se nomme `Platform-TransactionManager`. Cette interface encapsule un ensemble de méthodes indépendantes de la technologie pour la gestion des transactions, sans que nous ayons à connaître les API de transaction sous-jacentes. Spring fournit plusieurs implémentations d'un gestionnaire de transactions pour les différentes API de gestion des transactions.

Dans Spring, la gestion des transactions par programmation se fait en invoquant les méthodes `commit()` et `rollback()` sur un gestionnaire de transactions. Conceptuellement, cela revient à invoquer les mêmes méthodes sur une connexion JDBC, mais les méthodes déclarées par le gestionnaire de transactions de Spring sont indépendantes de la technologie.

Il est également possible de gérer les transactions par programmation en utilisant un template de transaction. Dans ce cas, nous encapsulons simplement l'opération dans un objet de rappel, que nous passons à un template de transaction pour son exécution. De cette manière, les opérations n'ont pas à se préoccuper des détails de bas niveau concernant la transaction.

Par l'intermédiaire de son framework POA, Spring prend en charge la gestion des transactions par déclaration. Puisque l'utilisation de la POA dans Spring a beaucoup évolué entre les versions 1.x et 2.x, il existe différentes manières de déclarer des transactions. Dans Spring 2.x, nous déclarons simplement un greffon transactionnel et l'appliquons aux beans en définissant un point d'action et un conseiller. Dans Spring AOP classique, nous créons un proxy de transaction en utilisant `TransactionInterceptor` ou `TransactionProxyFactoryBean`.

Outre la déclaration des transactions avec les fonctions de la POA, il existe une solution plus pratique pour la gestion déclarative des transactions. Il suffit d'appliquer l'annotation `@Transactional` aux méthodes qui ont besoin de la gestion des transactions et d'ajouter l'élément `<tx:annotation-driven>` dans le fichier de configuration des beans. Spring 2.5 fournit un aspect AspectJ capable de gérer les transactions pour toute méthode marquée par `@Transactional`, même si elle n'est pas publique ou si elle vient d'un objet créé en dehors de Spring. Pour activer cet aspect, nous pouvons choisir un tissage AspectJ à la compilation ou au chargement.

Enfin, nous pouvons définir des attributs transactionnels pour nos transactions. Spring reconnaît cinq catégories d'attributs transactionnels. L'attribut de propagation précise comment se passe la propagation d'une transaction lorsqu'une méthode transactionnelle est invoquée par une autre méthode. Le niveau d'isolation indique comment une transaction est isolée des autres lorsque plusieurs transactions ont lieu simultanément. La règle d'annulation établit les exceptions qui provoquent ou non l'annulation d'une transaction. La temporisation fixe la durée de vie d'une transaction avant qu'elle ne fasse l'objet d'une annulation forcée. L'attribut de lecture seule signale qu'une transaction se contente de lire des données, sans effectuer de mises à jour.

Le chapitre suivant détaille la prise en charge des différents frameworks de correspondance objet-relationnel dans Spring pour la mise en œuvre des services d'accès aux données.

Prise en charge de l'ORM

Au sommaire de ce chapitre

- ✓ Problèmes associés à l'utilisation directe des frameworks ORM
- ✓ Configurer des fabriques de ressources ORM dans Spring
- ✓ Rendre des objets persistants avec les templates ORM de Spring
- ✓ Rendre des objets persistants avec les sessions contextuelles d'Hibernate
- ✓ Rendre des objets persistants avec l'injection de contexte de JPA
- ✓ En résumé

Ce chapitre détaille l'intégration des frameworks de correspondance objet-relationnel (ORM, *Object/Relational Mapping*) dans les applications Spring. La plupart des frameworks ORM répandus, dont Hibernate, JDO, TopLink, iBATIS et JPA, sont reconnus par Spring. Bien que ce chapitre se focalise sur Hibernate et JPA (*Java Persistence API*), la prise en charge de ces frameworks ORM reste cohérente et les techniques décrites peuvent donc facilement s'appliquer à tous.

La *correspondance objet-relationnel* est une technologie moderne permettant d'assurer la persistance des objets dans une base de données relationnelle. Un framework ORM enregistre les objets en fonction des métadonnées de correspondance fournies, comme la correspondance entre les classes et les tables, les propriétés et les colonnes, etc. À l'exécution, il génère les requêtes SQL qui assurent la persistance des objets. Il est donc inutile d'écrire des requêtes SQL propres à une base de données, excepté pour tirer profit de fonctionnalités particulières ou pour optimiser les requêtes. En conséquence, l'application est indépendante de la base de données et peut facilement en changer dans le futur. Par rapport à une utilisation directe de JDBC, un framework ORM permet de réduire de manière importante le travail d'accès aux données dans les applications.

Hibernate est un framework ORM open-source à hautes performances très répandu dans la communauté Java. Il prend en charge la plupart des bases de données compati-

bles JDBC et permet d'utiliser des dialectes spécifiques pour accéder à certaines bases. Outre ses fonctions ORM de base, Hibernate dispose de fonctionnalités élaborées, comme la mise en cache, les opérations en cascade et le chargement paresseux. Il définit également un langage d'interrogation nommé HQL (*Hibernate Query Language*) pour que nous puissions écrire des requêtes d'objets simples mais puissantes.

JPA propose un ensemble d'annotations et d'API standard pour la persistance des objets dans les plates-formes Java SE et Java EE. JPA est défini par la spécification EJB 3.0, dans la JSR-220. JPA n'est qu'un ensemble d'API standard qui nécessite la présence d'un moteur compatible JPA pour mettre en œuvre la persistance. Nous pouvons comparer JPA à l'API JDBC, et un moteur JPA à un pilote JDBC. Grâce à son module *Hibernate Entity-Manager*, Hibernate peut être configuré en moteur compatible JPA. Ce chapitre décrit JPA en supposant que le moteur sous-jacent est Hibernate.

Au moment de l'écriture de ces lignes, Hibernate est en version 3.3. Spring 2.0 est compatible avec Hibernate 2.x et 3.x. La prise en charge d'Hibernate 2.x se fait au travers des classes et des interfaces définies dans `org.springframework.orm.hibernate`, tandis que celle de la version 3.x se fonde sur le paquetage `org.springframework.orm.hibernate3`. Il faut donc faire attention lors de l'importation des classes et des interfaces dans une application. Spring 2.5 est compatible uniquement avec Hibernate 3.1 et les versions ultérieures. Autrement dit, Hibernate 2.1 et Hibernate 3.0 ne sont plus officiellement pris en charge.

À la fin de ce chapitre, vous serez en mesure d'exploiter Hibernate et JPA pour l'accès aux données dans vos applications Spring. Vous aurez également acquis une connaissance approfondie du module d'accès aux données de Spring.

9.1 Problèmes associés à l'utilisation directe des frameworks ORM

Supposons que nous développions un système de gestion des cours pour un centre de formation. La première classe que nous créons se nomme `Course`. Il s'agit d'une *classe d'entité* ou *classe persistante*, car elle représente une entité du monde réel et ses instances vont persister dans une base de données. Pour que la persistance d'une classe d'entité puisse être prise en charge par un framework ORM, elle doit disposer d'un constructeur sans argument.

```
package com.apress.springrecipes.course;  
...  
public class Course {  
  
    private Long id;  
    private String title;  
    private Date beginDate;  
    private Date endDate;  
    private int fee;
```

```
// Constructeurs, accesseurs et mutateurs.  
...  
}
```

Pour chaque classe d’entité, nous devons définir une propriété qui identifie l’entité de manière unique. Il est conseillé de choisir un identifiant généré automatiquement, car il n’a aucune signification métier et ne sera jamais modifié. Par ailleurs, cet identifiant est utilisé par le framework ORM pour déterminer l’état de l’entité. Si sa valeur est `null`, l’entité est considérée comme une entité nouvelle non sauvegardée. La requête SQL générée pour la persistance d’une entité nouvelle est une insertion, sinon il s’agit d’une mise à jour. Pour que l’identifiant puisse être `null`, nous devons lui attribuer un type enveloppe (*wrapper*), comme `java.lang.Integer` ou `java.lang.Long`.

Dans le système de gestion des cours, nous avons besoin d’une interface de DAO pour encapsuler la logique d’accès aux données. Nous définissons les opérations suivantes dans l’interface `CourseDao` :

```
package com.apress.springrecipes.course;  
...  
public interface CourseDao {  
  
    public void store(Course course);  
    public void delete(Long courseId);  
    public Course findById(Long courseId);  
    public List<Course> findAll();  
}
```

En général, lorsqu’on utilise la correspondance objet-relationnel pour la persistance des objets, les opérations d’insertion et de mise à jour sont combinées en une seule opération, par exemple `store`. Ainsi, c’est le framework ORM, non le développeur, qui décide si l’objet doit être inséré ou mis à jour.

Pour qu’un framework ORM puisse enregistrer les objets dans une base de données, il doit connaître les métadonnées de correspondance pour les classes d’entité. Nous devons les lui fournir dans un format qu’il reconnaît. Le format natif d’Hibernate est XML. Cependant, puisque chaque framework ORM peut employer son propre format, JPA définit un ensemble d’annotations de persistance. Elles permettent de préciser les métadonnées de correspondance en utilisant un format standard qui a de fortes chances d’être réutilisé dans d’autres frameworks ORM.

Puisque Hibernate reconnaît les annotations JPA, il existe trois stratégies différentes pour la correspondance et la persistance de nos objets avec Hibernate et JPA :

- Utiliser l’API d’Hibernate pour la persistance des objets avec des correspondances Hibernate définies en XML.
- Utiliser l’API d’Hibernate pour la persistance des objets avec des annotations JPA.
- Utiliser JPA pour la persistance des objets avec des annotations JPA.

Les éléments de programmation d’Hibernate, de JPA et d’autres frameworks ORM ressemblent à ceux de JDBC. Ils sont recensés au Tableau 9.1.

Tableau 9.1 : Éléments de programmation fondamentaux pour différentes stratégies d'accès aux données

Concept	JDBC	Hibernate	JPA
Ressource	Connection	Session	EntityManager
Fabrique de ressources	DataSource	SessionFactory	EntityManagerFactory
Exception	SQLException	HibernateException	PersistenceException

Dans Hibernate, l’interface principale de persistance d’un objet se nomme `Session`. Nous pouvons en obtenir des instances depuis une instance de `SessionFactory`. Dans JPA, l’interface correspondante se nomme `EntityManager`, dont les instances sont obtenues à partir d’une instance de `EntityManagerFactory`. Les exceptions lancées par Hibernate sont de type `HibernateException`, tandis que celles lancées par JPA peuvent être de type `PersistenceException` ou toute exception Java SE, comme `IllegalArgumentException` et `IllegalStateException`. Ces exceptions étant des sous-classes de `RuntimeException`, nous ne sommes pas obligés de les intercepter.

Persistance des objets avec l’API Hibernate et les correspondances en XML

Pour la correspondance des classes d’entité avec le format XML d’Hibernate, nous pouvons fournir un seul fichier de correspondance pour chaque classe ou un grand fichier pour plusieurs classes. Dans la pratique, il est préférable d’en créer un pour chaque classe, dont le nom est celui de la classe auquel on ajoute l’extension de fichier `.hbm.xml` ; `hbm` signifie *Hibernate metadata*. Cette solution facilite la maintenance.

Le fichier de correspondance pour la classe `Course` se nomme donc `Course.hbm.xml`. Nous le plaçons dans le même paquetage que la classe d’entité.

```
<!DOCTYPE hibernate-mapping
PUBLIC "-//Hibernate/Hibernate Mapping DTD 3.0//EN"
"http://hibernate.sourceforge.net/hibernate-mapping-3.0.dtd">

<hibernate-mapping package="com.apress.springrecipes.course">
    <class name="Course" table="COURSE">
        <id name="id" type="long" column="ID">
            <generator class="identity" />
        </id>
        <property name="title" type="string">
            <column name="TITLE" length="100" not-null="true" />
        </property>
        <property name="beginDate" type="date" column="BEGIN_DATE" />
        <property name="endDate" type="date" column="END_DATE" />
    </class>
</hibernate-mapping>
```

```
<property name="fee" type="int" column="FEE" />
</class>
</hibernate-mapping>
```

Dans le fichier de correspondance, nous indiquons un nom de table pour la classe d'entité et une colonne de table pour chaque propriété simple. Nous pouvons également détailler la colonne, comme sa longueur, ses contraintes `not-null` et ses contraintes d'unicité. Par ailleurs, chaque entité doit avoir un identifiant, qui peut être généré automatiquement ou attribué manuellement. Dans notre exemple, l'identifiant est généré en utilisant une colonne d'identité de la table.

Chaque application qui utilise Hibernate a besoin d'un fichier global pour configurer certaines propriétés, comme les paramètres de la base de données (les propriétés de connexion JDBC ou le nom JNDI d'une source de données), le dialecte de la base de données, l'emplacement des métadonnées de correspondance, etc. Lorsque les métadonnées de correspondance sont définies dans des fichiers XML, nous devons indiquer les emplacements de ces fichiers. Par défaut, Hibernate examine le fichier `hibernate.cfg.xml` à la racine du chemin d'accès aux classes ; `cfg` signifie *configuration*.

```
<!DOCTYPE hibernate-configuration PUBLIC
  "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
  "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>
    <property name="connection.driver_class">
      org.apache.derby.jdbc.ClientDriver
    </property>
    <property name="connection.url">
      jdbc:derby://localhost:1527/course;create=true
    </property>
    <property name="connection.username">app</property>
    <property name="connection.password">app</property>
    <property name="dialect">org.hibernate.dialect.DerbyDialect</property>
    <property name="show_sql">true</property>
    <property name="hbm2ddl.auto">update</property>

    <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
  </session-factory>
</hibernate-configuration>
```

Avant que les objets ne puissent persister, nous devons créer les tables qui contiendront leurs données. Lorsqu'on utilise un framework ORM comme Hibernate, la création des tables n'est, en général, pas de notre ressort. En fixant la propriété `hbm2ddl.auto` à `update`, Hibernate peut nous aider à mettre à jour le schéma de base de données et à créer les tables lorsque c'est nécessaire.

Implémentons à présent l'interface de DAO dans le sous-paquetage `hibernate` en utilisant l'API Hibernate. Avant de pouvoir invoquer cette API pour la persistance d'un objet, nous devons initialiser une fabrique de sessions Hibernate, par exemple dans le constructeur.

INFO

Pour que la persistance des objets soit prise en charge par Hibernate, vous devez inclure les fichiers hibernate3.jar (situé dans le répertoire lib/hibernate de l'installation de Spring), antlr-2.7.6.jar (situé dans lib/antlr), asm-2.2.3.jar (situé dans lib/asm), cglib-nodep-2.1_3.jar (situé dans lib/cglib), dom4j-1.6.1.jar (situé dans lib/dom4j), ehcache-1.5.0.jar (situé dans lib/ehcache), jta.jar (situé dans lib/j2ee), commons-collections.jar et commons-logging.jar (situé dans lib/jakarta-commons) et log4j-1.2.15.jar (situé dans lib/log4j) dans le chemin d'accès aux classes. Pour utiliser Apache Derby comme moteur de base de données, vous devez également ajouter le fichier derby-client.jar (situé dans le répertoire lib de l'installation de Derby).

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.Query;
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public HibernateCourseDao() {
        Configuration configuration = new Configuration().configure();
        sessionFactory = configuration.buildSessionFactory();
    }

    public void store(Course course) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();
        try {
            tx.begin();
            session.saveOrUpdate(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            session.close();
        }
    }

    public void delete(Long courseId) {
        Session session = sessionFactory.openSession();
        Transaction tx = session.beginTransaction();
        try {
            tx.begin();
            Course course = (Course) session.get(Course.class, courseId);
            session.delete(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
        }
    }
}
```

```
        throw e;
    } finally {
        session.close();
    }
}

public Course findById(Long courseId) {
    Session session = sessionFactory.openSession();
    try {
        return (Course) session.get(Course.class, courseId);
    } finally {
        session.close();
    }
}

public List<Course> findAll() {
    Session session = sessionFactory.openSession();
    try {
        Query query = session.createQuery("from Course");
        return query.list();
    } finally {
        session.close();
    }
}
}
```

La première étape consiste à créer un objet Configuration et à lui demander de charger le fichier de configuration d’Hibernate. L’appel à la méthode configure() recherche par défaut le fichier hibernate.cfg.xml à la racine du chemin d’accès aux classes. Nous construisons ensuite une fabrique de sessions Hibernate en invoquant cet objet Configuration. Le rôle de la fabrique de sessions est de produire des sessions pour que nous puissions enregistrer nos objets.

Dans les méthodes du DAO précédent, nous commençons par ouvrir une session à partir de la fabrique. Dans toute opération qui implique une mise à jour de la base de données, comme saveOrUpdate() et delete(), nous initions une transaction Hibernate sur cette session. Si l’opération se termine avec succès, nous validons la transaction. Dans le cas contraire, toute exception RuntimeException annule la transaction. Pour les opérations en lecture seule, comme get() et les interrogations HQL, il est inutile de démarrer une transaction. Enfin, nous ne devons pas oublier de fermer la session afin de libérer les ressources qu’elle détient.

Nous créons la classe Main suivante pour tester les méthodes du DAO. Elle illustre également le cycle de vie particulier d’une entité.

```
package com.apress.springrecipes.course;
...
public class Main {

    public static void main(String[] args) {
        CourseDao courseDao = new HibernateCourseDao();
```

```
Course course = new Course();
course.setTitle("Les fondamentaux de Spring");
course.setBeginDate(new GregorianCalendar(2007, 8, 1).getTime());
course.setEndDate(new GregorianCalendar(2007, 9, 1).getTime());
course.setFee(1000);
courseDao.store(course);

List<Course> courses = courseDao.findAll();
Long courseId = courses.get(0).getId();

course = courseDao.findById(courseId);
System.out.println("Intitulé de la formation : " + course.getTitle());
System.out.println("Date de début : " + course.getBeginDate());
System.out.println("Date de fin : " + course.getEndDate());
System.out.println("Prix : " + course.getFee());

courseDao.delete(courseId);
}

}
```

Persistance des objets avec l'API Hibernate et les annotations JPA

Les annotations JPA ont été standardisées dans la spécification JSR-220. Elles sont donc reconnues par tous les frameworks ORM compatibles JPA, notamment Hibernate. En outre, grâce aux annotations, il est plus pratique de manipuler les métadonnées de correspondance dans le même fichier source.

La classe `Course` suivante illustre l'emploi des annotations JPA pour la définition des métadonnées de correspondance.

INFO

Pour utiliser les annotations JPA dans Hibernate, vous devez inclure les fichiers `persistence.jar` (situé dans le répertoire `lib/j2ee` de l'installation de Spring), `hibernate-annotations.jar` et `hibernate-commons-annotations.jar` (situé dans `lib/hibernate`) dans le chemin d'accès aux classes.

```
package com.apress.springrecipes.course;
...
import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

@Entity
@Table(name = "COURSE")
public class Course {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
@Column(name = "ID")
private Long id;

@Column(name = "TITLE", length = 100, nullable = false)
private String title;

@Column(name = "BEGIN_DATE")
private Date beginDate;

@Column(name = "END_DATE")
private Date endDate;

@Column(name = "FEE")
private int fee;

// Constructeurs, accesseurs et mutateurs.
...
}
```

Chaque classe d’entité doit être marquée avec l’annotation `@Entity`, dans laquelle nous pouvons également préciser le nom de la table associée. Pour chaque propriété, nous pouvons indiquer un nom de colonne et les détails de la colonne en utilisant une annotation `@Column`. Un identifiant est attribué à chaque classe d’entité avec l’annotation `@Id`. Nous choisissons la stratégie de génération de l’identifiant via l’annotation `@GeneratedValue`. Dans notre cas, il est généré par une colonne d’identité de table.

Dans Hibernate, la définition des métadonnées de correspondance peut se faire avec des fichiers XML de correspondance ou avec des annotations JPA. Si l’on utilise cette dernière solution, nous devons préciser les noms complets des classes d’entité dans le fichier `hibernate.cfg.xml` pour qu’Hibernate lise les annotations.

```
<hibernate-configuration>
  <session-factory>
    ...
    <!--Pour les correspondances XML d'Hibernate. -->
    <!--
    <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
    -->

    <!--Pour les annotations JPA. -->
    <mapping class="com.apress.springrecipes.course.Course" />
  </session-factory>
</hibernate-configuration>
```

Dans l’implémentation DAO Hibernate, la classe `Configuration` employée sert à lire les correspondances XML. Si nous définissons les métadonnées de correspondance Hibernate avec des annotations JPA, nous utilisons à la place sa sous-classe `AnnotationConfiguration`.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;
```

```
import org.hibernate.cfg.AnnotationConfiguration;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public HibernateCourseDao() {
        // Pour les correspondances XML d'Hibernate.
        // Configuration configuration = new Configuration().configure();

        // Pour les annotations JPA.
        Configuration configuration =
            new AnnotationConfiguration().configure();

        sessionFactory = configuration.buildSessionFactory();
    }
    ...
}
```

Persistante des objets avec JPA et le moteur Hibernate

Outre les annotations de persistante, JPA définit un ensemble d'interfaces de programmation pour la persistante des objets. Toutefois, JPA n'implémente pas lui-même la persistante mais s'appuie sur un moteur compatible JPA. Grâce au module Hibernate EntityManager, Hibernate est compatible JPA et peut alors servir de moteur JPA pour enregistrer les objets.

Dans un environnement Java EE, nous pouvons configurer le moteur JPA dans un conteneur Java EE. En revanche, dans une application Java SE, nous devons le configurer localement. La configuration de JPA se fait au travers du fichier XML central persistence.xml, qui se trouve dans le répertoire META-INF à la racine du chemin d'accès aux classes. Dans ce fichier, nous définissons les propriétés spécifiques pour la configuration du moteur sous-jacent.

À présent, créons le fichier persistence.xml dans le répertoire META-INF du chemin d'accès aux classes pour la configuration de JPA. Chaque fichier de configuration contient un ou plusieurs éléments <persistence-unit>. Une *unité de persistante* définit un ensemble de classes persistantes et la manière dont leur persistante est assurée. Chaque unité de persistante exige un nom d'identification. Nous la nommons course.

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/persistence
        http://java.sun.com/xml/ns/persistence/persistence_1_0.xsd"
    version="1.0">

    <persistence-unit name="course">
        <properties>
            <property name="hibernate.ejb.cfgfile"
                value="/hibernate.cfg.xml" />
```

```
</properties>
</persistence-unit>
</persistence>
```

Dans ce fichier de configuration, nous précisons qu'Hibernate est le moteur JPA sous-jacent en faisant référence au fichier de configuration d'Hibernate situé à la racine du chemin d'accès aux classes. Cependant, puisque le module Hibernate EntityManager détecte automatiquement que les fichiers XML de correspondance et les annotations JPA sont des métadonnées de correspondance, il est inutile de les préciser explicitement. Dans le cas contraire, nous recevons une exception `org.hibernate.DuplicateMappingException`.

```
<hibernate-configuration>
    <session-factory>
        ...
        <!-- Inutile de préciser les fichiers de correspondance et
            les classes annotées -->
        <!--
            <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
            <mapping class="com.apress.springrecipes.course.Course" />
        -->
    </session-factory>
</hibernate-configuration>
```

Au lieu de faire référence au fichier de configuration d'Hibernate, nous pouvons également réunir tous les éléments de configuration dans `persistence.xml`.

```
<persistence ...>
    <persistence-unit name="course">
        <properties>
            <property name="hibernate.connection.driver_class"
                value="org.apache.derby.jdbc.ClientDriver" />
            <property name="hibernate.connection.url"
                value="jdbc:derby://localhost:1527/course;create=true" />
            <property name="hibernate.connection.username" value="app" />
            <property name="hibernate.connection.password" value="app" />
            <property name="hibernate.dialect"
                value="org.hibernate.dialect.DerbyDialect" />
            <property name="hibernate.show_sql" value="true" />
            <property name="hibernate.hbm2ddl.auto" value="update" />
        </properties>
    </persistence-unit>
</persistence>
```

Un conteneur Java EE peut prendre en charge le gestionnaire d'entités à notre place et l'injecter directement dans les composants EJB. Mais, lorsque nous utilisons JPA en dehors d'un environnement Java EE, par exemple dans des applications Java SE, nous devons créer et maintenir le gestionnaire d'entités nous-mêmes.

Implémentons à présent l'interface `CourseDao` dans le sous-paquetage `jpa` en utilisant JPA dans une application Java SE. Avant d'invoquer JPA pour la persistance des objets,

nous devons initialiser une fabrique de gestionnaires d'entités. Le rôle de cette fabrique est de produire des gestionnaires d'entités pour que nous puissions rendre nos objets persistants.

INFO

Pour utiliser Hibernate comme moteur JPA sous-jacent, vous devez inclure les fichiers hibernate-entitymanager.jar et jboss-archive-browsing.jar (situés dans le répertoire lib/hibernate de l'installation de Spring) dans le chemin d'accès aux classes. Puisque le module EntityManager dépend de Javassist (<http://www.jboss.org/javassist/>), vous devez également inclure javassist.jar dans le chemin d'accès aux classes. Si vous avez installé Hibernate, vous le trouverez dans le répertoire lib. Sinon vous pouvez le télécharger à partir du site web.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManager;
import javax.persistence.EntityManagerFactory;
import javax.persistence.EntityTransaction;
import javax.persistence.Persistence;
import javax.persistence.Query;

public class JpaCourseDao implements CourseDao {

    private EntityManagerFactory entityManagerFactory;

    public JpaCourseDao() {
        entityManagerFactory =
            Persistence.createEntityManagerFactory("course");
    }

    public void store(Course course) {
        EntityManager manager = entityManagerFactory.createEntityManager();
        EntityTransaction tx = manager.getTransaction();
        try {
            tx.begin();
            manager.merge(course);
            tx.commit();
        } catch (RuntimeException e) {
            tx.rollback();
            throw e;
        } finally {
            manager.close();
        }
    }

    public void delete(Long courseId) {
        EntityManager manager = entityManagerFactory.createEntityManager();
        EntityTransaction tx = manager.getTransaction();
        try {
            tx.begin();
            Course course = manager.find(Course.class, courseId);
            manager.remove(course);
        } catch (Exception e) {
            tx.rollback();
        } finally {
            manager.close();
        }
    }
}
```

```
        tx.commit();
    } catch (RuntimeException e) {
        tx.rollback();
        throw e;
    } finally {
        manager.close();
    }
}

public Course findById(Long courseId) {
    EntityManager manager = entityManagerFactory.createEntityManager();
    try {

        return manager.find(Course.class, courseId);
    } finally {
        manager.close();
    }
}

public List<Course> findAll() {
    EntityManager manager = entityManagerFactory.createEntityManager();
    try {
        Query query = manager.createQuery(
            "select course from Course course");
        return query.getResultList();
    } finally {
        manager.close();
    }
}
}
```

La fabrique de gestionnaires d’entités est créée en appelant la méthode statique `createEntityManagerFactory()` de la classe `javax.persistence.Persistence`. Elle attend en argument le nom d’une unité de persistance définie dans `persistence.xml`.

Dans les méthodes du DAO précédent, nous commençons par créer un gestionnaire d’entités à l’aide de la fabrique. Pour chaque opération qui implique une mise à jour de la base de données, comme `merge()` et `remove()`, nous initions une transaction JPA sur le gestionnaire d’entités. Pour les opérations en lecture seule, comme `find()` et les interrogations JPA, il est inutile de démarrer une transaction. Enfin, nous fermons le gestionnaire d’entités pour libérer ses ressources.

Nous testons les méthodes du DAO à l’aide de la classe `Main` suivante, dans laquelle nous choisissons l’implémentation JPA.

```
package com.apress.springrecipes.course;
...
public class Main {

    public static void main(String[] args) {
        CourseDao courseDao = new JpaCourseDao();
        ...
    }
}
```

Dans les implémentations précédentes pour Hibernate et JPA, chaque méthode du DAO diffère seulement sur une ou deux lignes. Le reste du code est standard et nous devons le reproduire de multiples fois. Par ailleurs, chaque framework ORM possède sa propre API pour la gestion locale des transactions.

9.2 Configurer des fabriques de ressources ORM dans Spring

Problème

Lorsqu'on utilise un framework ORM seul, nous devons configurer sa fabrique de ressources au travers de son API. Pour Hibernate et JPA, nous construisons une fabrique de sessions et une fabrique de gestionnaires d'entités à partir de l'API native d'Hibernate et de JPA. Avec cette approche, nous devons assurer nous-mêmes la gestion de ces fabriques. Par ailleurs, notre application ne peut pas bénéficier des fonctions d'accès aux données apportées par Spring.

Solution

Spring fournit plusieurs beans de fabrique pour que nous puissions créer une fabrique de sessions Hibernate ou une fabrique de gestionnaires d'entités JPA sous forme d'un bean unique dans le conteneur IoC. Ces fabriques peuvent être partagées par plusieurs beans grâce à l'injection de dépendance. Par ailleurs, elles peuvent être associées aux autres fonctionnalités d'accès aux données de Spring, comme les sources de données et les gestionnaires de transactions.

Explications

Configurer une fabrique de sessions Hibernate dans Spring

Commençons par modifier la classe `HibernateCourseDao` pour qu'elle accepte une fabrique de sessions par injection de dépendance au lieu de la créer directement dans le constructeur avec l'API native d'Hibernate.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;

public class HibernateCourseDao implements CourseDao {
    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    ...
}
```

Voyons à présent comment déclarer dans Spring une fabrique de sessions qui utilise des fichiers XML de correspondance. Pour cela, nous devons activer à nouveau la définition des correspondances XML dans `hibernate.cfg.xml`.

```
<hibernate-configuration>
    <session-factory>
        ...
        <!--Pour les correspondances XML d'Hibernate. -->
        <mapping resource="com/apress/springrecipes/course/Course.hbm.xml" />
    </session-factory>
</hibernate-configuration>
```

Nous créons ensuite un fichier de configuration des beans pour utiliser Hibernate comme framework ORM (par exemple `beans-hibernate.xml` à la racine du chemin d'accès aux classes). Nous déclarons une fabrique de sessions qui utilise des fichiers XML de correspondance à l'aide du bean de fabrique `LocalSessionFactoryBean`. Nous déclarons également une instance de `HibernateCourseDao` sous le contrôle de Spring.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

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

    <bean id="courseDao"
        class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>
```

La propriété `configLocation` nous permet d'indiquer au bean de fabrique qu'il doit charger le fichier de configuration d'Hibernate. Elle est de type `Resource`, mais nous pouvons lui affecter une chaîne de caractères. L'éditeur de propriétés intégré `Resource-Editor` se charge de convertir cette chaîne en un objet `Resource`. Le fichier de configuration est chargé depuis la racine du chemin d'accès aux classes.

Nous modifions à présent la classe `Main` pour qu'elle obtienne l'instance de `HibernateCourseDao` à partir du conteneur Spring IoC.

```
package com.apress.springrecipes.course;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {
    public static void main(String[] args) {
```

```

        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-hibernate.xml");

        CourseDao courseDao = (CourseDao) context.getBean("courseDao");
        ...
    }
}

```

Le bean de fabrique précédent crée une fabrique de sessions en chargeant le fichier de configuration d’Hibernate, qui contient les paramètres de la base de données (les propriétés de connexion JDBC ou le nom JNDI d’une source de données). Supposons à présent que nous ayons défini une source de données dans le conteneur Spring IoC. Si nous voulons l’utiliser pour la fabrique de sessions, nous pouvons l’injecter dans la propriété `dataSource` de `LocalSessionFactoryBean`. La source de données affectée à cette propriété remplace les paramètres de base de données définis dans le fichier de configuration d’Hibernate.

```

<beans ...>
    ...
    <bean id="dataSource"
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">
        <property name="driverClassName"
                  value="org.apache.derby.jdbc.ClientDriver" />
        <property name="url"
                  value="jdbc:derby://localhost:1527/course;create=true" />
        <property name="username" value="app" />
        <property name="password" value="app" />
    </bean>

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

```

Nous pouvons même ignorer le fichier de configuration d’Hibernate en regroupant tous les éléments de configuration dans `LocalSessionFactoryBean`. Par exemple, nous précisons l’emplacement des fichiers XML de correspondance dans la propriété `mappingResources` et les autres propriétés d’Hibernate, comme le dialecte de la base de données, dans `hibernateProperties`.

```

<bean id="sessionFactory"
      class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingResources">
        <list>
            <value>com/apress/springrecipes/course/Course.hbm.xml</value>
        </list>
    </property>
    <property name="hibernateProperties">
        <props>

```

```
<prop key="hibernate.dialect">
    org.hibernate.dialect.DerbyDialect
</prop>
<prop key="hibernate.show_sql">true</prop>
<prop key="hibernate.hbm2ddl.auto">update</prop>
</props>
</property>
</bean>
```

La propriété `mappingResources` étant de type `String[]`, nous pouvons indiquer un ensemble de fichiers de correspondance dans le chemin d'accès aux classes. `LocalSessionFactoryBean` nous permet également de profiter du chargement de ressources de Spring pour charger les fichiers de correspondance à partir de différents types d'emplacements. Nous indiquons les chemins des fichiers de correspondance dans la propriété `mappingLocations`, qui est de type `Resource[]`.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    ...
    <property name="mappingLocations">
        <list>
            <value>
                classpath:com/apress/springrecipes/course/Course.hbm.xml
            </value>
        </list>
    </property>
    ...
</bean>
```

Grâce au mécanisme de chargement des ressources de Spring, nous pouvons également inclure des caractères génériques dans le chemin de ressource pour désigner plusieurs fichiers de correspondance. Cela nous évite d'ajuster les emplacements chaque fois que nous ajoutons une nouvelle classe d'entité. L'éditeur `ResourceArrayPropertyEditor` préenregistré de Spring se charge de convertir ce chemin en un tableau de `Resource`.

```
<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    ...
    <property name="mappingLocations"
        value="classpath:com/apress/springrecipes/course/*.hbm.xml" />
    ...
</bean>
```

Si les métadonnées de correspondance sont fournies par des annotations JPA, nous devons nous tourner vers `AnnotationSessionFactoryBean`. Nous indiquons les classes persistantes dans la propriété `annotatedClasses` de `AnnotationSessionFactoryBean`.

```
<bean id="sessionFactory" class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="annotatedClasses">
        <list>
            <value>com.apress.springrecipes.course.Course</value>
        </list>
    </property>

```

```
</property>
<property name="hibernateProperties">
    <props>
        <prop key="hibernate.dialect">
            org.hibernate.dialect.DerbyDialect
        </prop>
        <prop key="hibernate.show_sql">true</prop>
        <prop key="hibernate.hbm2ddl.auto">update</prop>
    </props>
</property>
</bean>
```

Nous supprimons le fichier de configuration d’Hibernate, `hibernate.cfg.xml`, car les éléments de la configuration ont été transférés sur Spring.

Configurer une fabrique de gestionnaires d’entités JPA dans Spring

Avant tout, modifions `JpaCourseDao` pour qu’elle accepte une fabrique de gestionnaires d’entités *via* l’injection de dépendance au lieu de la créer dans le constructeur.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManagerFactory;

public class JpaCourseDao implements CourseDao {

    private EntityManagerFactory entityManagerFactory;

    public void setEntityManagerFactory(
        EntityManagerFactory entityManagerFactory) {
        this.entityManagerFactory = entityManagerFactory;
    }
    ...
}
```

Les spécifications de JPA expliquent comment procéder pour obtenir une fabrique de gestionnaires d’entités dans les environnements Java SE et Java EE. Dans un environnement Java SE, une fabrique de gestionnaires d’entités est créée manuellement en invoquant la méthode statique `createEntityManagerFactory()` de la classe `Persistence`.

Créons à présent un fichier de configuration des beans pour utiliser JPA (par exemple `beans-jpa.xml` à la racine du chemin d’accès aux classes). Spring fournit un bean de fabrique, `LocalEntityManagerFactoryBean`, pour que nous puissions créer une fabrique de gestionnaires d’entités dans le conteneur IoC. Nous indiquons le nom de l’unité de persistance définie dans le fichier de configuration JPA et déclarons une instance de `JpaCourseDao` sous le contrôle de Spring.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
```

```
<bean id="entityManagerFactory"
      class="org.springframework.orm.jpa.LocalEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="course" />
</bean>

<bean id="courseDao"
      class="com.apress.springrecipes.course.jpa.JpaCourseDao">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
</beans>
```

Pour tester cette instance de JpaCourseDao dans la classe Main, nous la retrouvons à partir du conteneur Spring IoC.

```
package com.apress.springrecipes.course;
...
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

public class Main {

    public static void main(String[] args) {
        ApplicationContext context =
            new ClassPathXmlApplicationContext("beans-jpa.xml");

        CourseDao courseDao = (CourseDao) context.getBean("courseDao");
        ...
    }
}
```

Dans un environnement Java EE, nous pouvons utiliser JNDI pour retrouver une fabrique de gestionnaires d'entités à partir d'un conteneur Java EE. Dans Spring 2.x, une recherche JNDI se fait à l'aide de l'élément `<jee:jndi-lookup>`.

```
<jee:jndi-lookup id="entityManagerFactory" jndi-name="jpa/coursePU" />
```

LocalEntityManagerFactoryBean crée une fabrique de gestionnaires d'entités en chargeant le fichier de configuration de JPA (c'est-à-dire `persistence.xml`). Une autre solution, plus souple, consiste à créer une fabrique de gestionnaires d'entités à l'aide d'un autre bean de fabrique, LocalContainerEntityManagerFactoryBean. Nous pouvons ainsi écraser certains éléments du fichier de configuration de JPA, comme la source de données et le dialecte de la base de données. Par conséquent, nous pouvons profiter des fonctions d'accès aux données de Spring pour configurer la fabrique de gestionnaires d'entités.

```
<beans ...>
  ...
  <bean id="dataSource"
        class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName"
              value="org.apache.derby.jdbc.ClientDriver" />
    <property name="url"
              value="jdbc:derby://localhost:1527/course;create=true" />
```

```

<property name="username" value="app" />
<property name="password" value="app" />
</bean>

<bean id="entityManagerFactory" class="org.springframework.orm.jpa.LocalContainerEntityManagerFactoryBean">
    <property name="persistenceUnitName" value="course" />
    <property name="dataSource" ref="dataSource" />
    <property name="jpaVendorAdapter">
        <bean class="org.springframework.orm.jpa.vendor.HibernateJpaVendorAdapter">
            <property name="databasePlatform" value="org.hibernate.dialect.DerbyDialect" />
            <property name="showSql" value="true" />
            <property name="generateDdl" value="true" />
        </bean>
    </property>
</bean>
</beans>
```

Dans les configurations de beans précédentes, nous injectons une source de données dans la fabrique de gestionnaires d'entités. Elle remplace les paramètres de base de données présents dans le fichier de configuration de JPA. Nous fixons l'adaptateur de fournisseur JPA à `LocalContainerEntityManagerFactoryBean` de manière à définir les propriétés spécifiques au moteur JPA. Puisque ce moteur est Hibernate, nous choisissons `HibernateJpaVendorAdapter`. Les autres propriétés non reconnues par cet adaptateur peuvent être indiquées dans `jpaProperties`.

Nous simplifions à présent le fichier de configuration de JPA, `persistence.xml`, car ses éléments de configuration ont été transférés dans Spring.

```

<persistence ...>
    <persistence-unit name="course" />
</persistence>
```

9.3 Rendre des objets persistants avec les templates ORM de Spring

Problème

Lorsque nous utilisons un framework seul, nous devons répéter certaines tâches standard pour chaque opération de DAO. Par exemple, dans une opération de DAO mise en œuvre avec Hibernate ou JPA, nous devons ouvrir et fermer une session ou un gestionnaire d'entités, ainsi que démarrer, valider et annuler une transaction avec l'API native.

Solution

Pour simplifier l'utilisation d'un framework ORM, Spring prend une approche comparable à celle employée avec JDBC : il fournit des classes templates et des classes de support des DAO. Par ailleurs, Spring définit une couche abstraite au-dessus des diffé-

rentes API de gestion de transactions. En fonction du framework ORM, nous devons simplement choisir l'implémentation correspondante du gestionnaire de transactions. Ensuite, la gestion des transactions se fait de manière semblable.

Dans le module d'accès aux données de Spring, la prise en charge des différentes stratégies d'accès aux données reste cohérente. Le Tableau 9.2 compare les classes de support pour JDBC, Hibernate et JPA.

Tableau 9.2 : Classes de support fournies par Spring pour les différentes stratégies d'accès aux données

<i>Classe de support</i>	<i>JDBC</i>	<i>Hibernate</i>	<i>JPA</i>
Classe template	JdbcTemplate	HibernateTemplate	JpaTemplate
Classe de support du DAO	JdbcDaoSupport	HibernateDaoSupport	JpaDaoSupport
Gestionnaire de transactions	DataSourceTransactionManager	HibernateTransactionManager	JpaTransactionManager

Spring définit les classes `HibernateTemplate` et `JpaTemplate` pour offrir les méthodes templates qui correspondent aux différents types d'opérations Hibernate et JPA et faciliter leur utilisation. Ces méthodes templates garantissent que les sessions Hibernate et les gestionnaires d'entités JPA sont ouverts et fermés correctement. Des transactions Hibernate et JPA natives sont également impliquées dans les transactions gérées par Spring. Par conséquent, nous pouvons gérer les transactions par déclaration pour nos DAO Hibernate et JPA, sans avoir à répéter du code transactionnel standard.

Explications

Utiliser des templates Hibernate et JPA

Tout d'abord, voici comment simplifier la classe `HibernateCourseDao` en nous aidant de `HibernateTemplate`.

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.orm.hibernate3.HibernateTemplate;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao implements CourseDao {

    private HibernateTemplate hibernateTemplate;

    public void setHibernateTemplate(HibernateTemplate hibernateTemplate) {
        this.hibernateTemplate = hibernateTemplate;
    }
}
```

```

@Transactional
public void store(Course course) {
    hibernateTemplate.saveOrUpdate(course);
}

@Transactional
public void delete(Long courseId) {
    Course course = (Course) hibernateTemplate.get(Course.class, courseId);
    hibernateTemplate.delete(course);
}

@Transactional(readOnly = true)
public Course findById(Long courseId) {
    return (Course) hibernateTemplate.get(Course.class, courseId);
}

@Transactional(readOnly = true)
public List<Course> findAll() {
    return hibernateTemplate.find("from Course");
}
}

```

Dans cette implémentation, nous déclarons que toutes les méthodes du DAO sont transactionnelles en les marquant avec `@Transactional`. Parmi ces méthodes, `findById()` et `findAll()` sont en lecture seule. Les méthodes templates de `HibernateTemplate` sont responsables de la gestion des sessions et des transactions. Si plusieurs opérations Hibernate ont lieu dans une méthode transactionnelle du DAO, les méthodes templates s'assurent qu'elles s'exécutent au sein de la même session et transaction. Par conséquent, nous n'avons pas besoin d'utiliser l'API d'Hibernate pour gérer les sessions et les transactions.

La classe `HibernateTemplate` étant sûre vis-à-vis des threads, nous pouvons en déclarer une seule instance dans le fichier de configuration des beans pour Hibernate (c'est-à-dire `beans-hibernate.xml`) et l'injecter dans tous les DAO Hibernate. La propriété `sessionFactory` d'une instance de `HibernateTemplate` doit obligatoirement être définie. Nous pouvons l'injecter par un mutateur ou un argument de constructeur.

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    ...
    <tx:annotation-driven />

    <bean id="transactionManager"
          class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

```

```
<bean id="hibernateTemplate"
      class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

<bean name="courseDao"
      class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
    <property name="hibernateTemplate" ref="hibernateTemplate" />
</bean>
</beans>
```

Pour activer la gestion déclarative des transactions sur les méthodes marquées par `@Transactional`, nous ajoutons l'élément `<tx:annotation-driven>` dans le fichier de configuration des beans. Puisqu'il recherche par défaut un gestionnaire de transactions nommé `transactionManager`, nous déclarons une instance de `HibernateTransactionManager` avec ce nom. Elle prend en charge les transactions pour les sessions ouvertes par l'intermédiaire de sa fabrique de sessions, dont la propriété doit donc être définie.

De la même manière, nous pouvons simplifier la classe `JpaCourseDao` en nous aidant de `JpaTemplate`. Toutes les méthodes du DAO sont également déclarées transactionnelles.

```
package com.apress.springrecipes.course.jpa;
...
import org.springframework.orm.jpa.JpaTemplate;
import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao implements CourseDao {

    private JpaTemplate jpaTemplate;

    public void setJpaTemplate(JpaTemplate jpaTemplate) {
        this.jpaTemplate = jpaTemplate;
    }

    @Transactional
    public void store(Course course) {
        jpaTemplate.merge(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = jpaTemplate.find(Course.class, courseId);
        jpaTemplate.remove(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return jpaTemplate.find(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return jpaTemplate.find("from Course");
    }
}
```

Dans le fichier de configuration des beans pour JPA (c'est-à-dire beans-jpa.xml), nous déclarons une instance de JpaTemplate et l'injectons dans tous les DAO JPA. De plus, nous devons déclarer une instance de JpaTransactionManager pour la gestion des transactions JPA.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    ...
    <tx:annotation-driven />

    <bean id="transactionManager"
        class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean id="jpaTemplate"
        class="org.springframework.orm.jpa.JpaTemplate">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean name="courseDao"
        class="com.apress.springrecipes.course.jpa.JpaCourseDao">
        <property name="jpaTemplate" ref="jpaTemplate" />
    </bean>
</beans>
```

HibernateTemplate et JpaTemplate présentent également l'avantage de convertir les exceptions natives d'Hibernate et de JPA en exceptions de la hiérarchie DataAccessException de Spring. Nous pouvons ainsi mettre en place une gestion cohérente des exceptions pour toutes les stratégies d'accès aux données. Par exemple, si une contrainte de base de données n'est pas respectée lors de la persistance d'un objet, Hibernate lance une exception `org.hibernate.exception.ConstraintViolationException`, tandis que JPA lance une exception `javax.persistence.EntityExistsException`. Ces exceptions sont converties par HibernateTemplate et par JpaTemplate en une exception `DataIntegrityViolationException`, qui est une sous-classe de l'exception DataAccessException de Spring.

Si nous souhaitons accéder à la session Hibernate ou au gestionnaire d'entités JPA sous-jacent dans HibernateTemplate ou JpaTemplate, par exemple pour effectuer des opérations Hibernate ou JPA natives, nous pouvons implémenter l'interface `HibernateCallback` ou `JpaCallback` et passer une instance de cette implémentation à la méthode `execute()` du template.

```
hibernateTemplate.execute(new HibernateCallback() {
    public Object doInHibernate(Session session) throws HibernateException,
        SQLException {
        ...
    }
};

jpaTemplate.execute(new JpaCallback() {
    public Object doInJpa(EntityManager em) throws PersistenceException {
        ...
    }
};
```

Étendre les classes de support des DAO d'Hibernate et de JPA

En dérivant de `HibernateDaoSupport`, notre DAO Hibernate hérite des méthodes `setSessionFactory()` et `setHibernateTemplate()`. Ensuite, dans les méthodes du DAO, nous pouvons simplement invoquer `getHibernateTemplate()` pour obtenir l'instance du template.

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao extends HibernateDaoSupport
    implements CourseDao {

    @Transactional
    public void store(Course course) {
        getHibernateTemplate().saveOrUpdate(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = (Course) getHibernateTemplate().get(Course.class,
            courseId);
        getHibernateTemplate().delete(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return (Course) getHibernateTemplate().get(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return getHibernateTemplate().find("from Course");
    }
}
```

Puisque `HibernateCourseDao` hérite des méthodes `setSessionFactory()` et `setHibernateTemplate()`, nous pouvons injecter l'une ou l'autre dans notre DAO de manière à

retrouver l'instance de `HibernateTemplate`. Si nous injectons une fabrique de sessions, nous supprimons la déclaration de `HibernateTemplate`.

```
<bean name="courseDao"
      class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>
```

De la même manière, notre DAO JPA peut étendre `JpaDaoSupport` pour hériter des méthodes `setEntityManagerFactory()` et `setJpaTemplate()`. Dans les méthodes du DAO, nous invoquons simplement `getJpaTemplate()` pour obtenir l'instance du template.

```
package com.apress.springrecipes.course.jpa;
...
import org.springframework.orm.jpa.support.JpaDaoSupport;
import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao extends JpaDaoSupport implements CourseDao {

    @Transactional
    public void store(Course course) {
        getJpaTemplate().merge(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = getJpaTemplate().find(Course.class, courseId);
        getJpaTemplate().remove(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return getJpaTemplate().find(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        return getJpaTemplate().find("from Course");
    }
}
```

Puisque `JpaCourseDao` hérite des méthodes `setEntityManagerFactory()` et `setJpaTemplate()`, nous pouvons injecter l'une ou l'autre dans notre DAO. Si nous injectons une fabrique de gestionnaires d'entités, nous pouvons supprimer la déclaration de `JpaTemplate`.

```
<bean name="courseDao"
      class="com.apress.springrecipes.course.jpa.JpaCourseDao">
    <property name="entityManagerFactory" ref="entityManagerFactory" />
</bean>
```

9.4 Rendre des objets persistants avec les sessions contextuelles d'Hibernate

Problème

La classe `HibernateTemplate` de Spring permet de simplifier l'implémentation d'un DAO en gérant à notre place les sessions et les transactions. Cependant, l'utilisation de cette classe signifie que le DAO dépend de l'API de Spring.

Solution

À la place de `HibernateTemplate`, nous pouvons employer les sessions contextuelles d'Hibernate. Dans Hibernate 3, une fabrique de sessions est capable de gérer des sessions contextuelles pour notre compte et nous pouvons les obtenir avec la méthode `getCurrentSession()`. Au sein d'une même transaction, nous obtenons la même session pour chaque appel à la méthode `getCurrentSession()`. Il n'y a ainsi qu'une seule session Hibernate par transaction, ce qui fonctionne parfaitement avec la gestion des transactions de Spring.

Explications

Pour mettre en place la solution fondée sur les sessions contextuelles, les méthodes du DAO doivent avoir accès à la fabrique de sessions, qui peut être injectée par un mutateur ou un argument du constructeur. Ensuite, dans chaque méthode du DAO, nous obtenons la session contextuelle à partir de la fabrique de sessions et l'utilisons pour la persistance de l'objet.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.Query;
import org.hibernate.SessionFactory;
import org.springframework.transaction.annotation.Transactional;

public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }

    @Transactional
    public void store(Course course) {
        sessionFactory.getCurrentSession().saveOrUpdate(course);
    }
}
```

```

@Transactional
public void delete(Long courseId) {
    Course course = (Course) sessionFactory.getCurrentSession().get(
        Course.class, courseId);
    sessionFactory.getCurrentSession().delete(course);
}

@Transactional(readOnly = true)
public Course findById(Long courseId) {
    return (Course) sessionFactory.getCurrentSession().get(
        Course.class, courseId);
}

@Transactional(readOnly = true)
public List<Course> findAll() {
    Query query = sessionFactory.getCurrentSession().createQuery(
        "from Course");
    return query.list();
}
}

```

Toutes les méthodes du DAO doivent être transactionnelles. Pour cela, nous pouvons annoter chacune d'elles avec `@Transactional`, ou directement la classe. Ainsi, les opérations de persistance au sein d'une méthode du DAO sont exécutées dans la même transaction et, par conséquent, la même session. Par ailleurs, si une méthode d'un composant de la couche de service appelle plusieurs méthodes du DAO et si elle propage sa propre transaction à ces méthodes, alors, ces méthodes du DAO s'exécutent également au sein de la même session.

Dans le fichier de configuration des beans pour Hibernate, `beans-hibernate.xml`, nous déclarons une instance de `HibernateTransactionManager` pour cette application et activons la gestion déclarative des transactions *via* `<tx:annotation-driven>`.

```

<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    ...
    <tx:annotation-driven />

    <bean id="transactionManager"
        class="org.springframework.orm.hibernate3.HibernateTransactionManager">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>

    <bean name="courseDao"
        class="com.apress.springrecipes.course.hibernate.HibernateCourseDao">
        <property name="sessionFactory" ref="sessionFactory" />
    </bean>
</beans>

```

HibernateTemplate convertit les exceptions natives d'Hibernate en exceptions de la hiérarchie `DataAccessException` de Spring. Nous pouvons ainsi mettre en place une gestion cohérente des exceptions pour toutes les stratégies d'accès aux données. Cependant, lorsqu'on appelle des méthodes natives sur une session Hibernate, les exceptions lancées sont du type `HibernateException`. Si nous voulons qu'elles soient converties en exceptions `DataAccessException` de Spring afin d'avoir une gestion cohérente des exceptions, nous appliquons l'annotation `@Repository` à la classe du DAO qui doit bénéficier de la conversion des exceptions.

```
package com.apress.springrecipes.course.hibernate;
...
import org.springframework.stereotype.Repository;

@Repository
public class HibernateCourseDao implements CourseDao {
    ...
}
```

Nous enregistrons ensuite une instance de `PersistenceExceptionTranslationPostProcessor` pour convertir les exceptions natives d'Hibernate en exceptions d'accès aux données de la hiérarchie `DataAccessException` de Spring. Ce postprocesseur de beans traduit les exceptions uniquement pour les beans marqués par `@Repository`.

```
<beans ...>
    ...
    <bean class="org.springframework.dao.annotation.PersistenceExceptionTranslationPostProcessor" />
</beans>
```

Dans Spring 2.5, `@Repository` est une annotation stéréotype. En l'utilisant, une classe de composant peut être détectée automatiquement par la fonction de scan des composants. Nous précisons un nom de composant dans cette annotation et appliquons la liaison automatique par le conteneur Spring IoC à la fabrique de sessions grâce à `@Autowired`.

```
package com.apress.springrecipes.course.hibernate;
...
import org.hibernate.SessionFactory;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Repository;

@Repository("courseDao")
public class HibernateCourseDao implements CourseDao {

    private SessionFactory sessionFactory;

    @Autowired
    public void setSessionFactory(SessionFactory sessionFactory) {
        this.sessionFactory = sessionFactory;
    }
    ...
}
```

Ensuite, il suffit simplement d'ajouter l'élément <context:component-scan> et de supprimer la déclaration originale du bean `HibernateCourseDao`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:context="http://www.springframework.org/schema/context"
    xmlns:tx="http://www.springframework.org/schema/tx"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
        http://www.springframework.org/schema/context
        http://www.springframework.org/schema/context/spring-context-2.5.xsd
        http://www.springframework.org/schema/tx
        http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <context:component-scan
        base-package="com.apress.springrecipes.course.hibernate" />
    ...
</beans>
```

9.5 Rendre des objets persistants avec l'injection de contexte de JPA

Problème

Dans un environnement Java EE, un conteneur Java EE peut s'occuper des gestionnaires d'entités à notre place et les injecter directement dans nos composants EJB. Un composant EJB effectue simplement des opérations de persistance sur le gestionnaire d'entités injecté sans se préoccuper de sa création et de la gestion des transactions.

De la même manière, Spring fournit la classe `JpaTemplate` pour simplifier l'implémentation d'un DAO en prenant en charge les gestionnaires d'entités et les transactions à notre place. Cependant, l'utilisation de `JpaTemplate` rend le DAO dépendant de l'API de Spring.

Solution

À la place de la classe Spring `JpaTemplate`, nous pouvons utiliser l'injection de contexte de JPA. À l'origine, l'annotation `@PersistenceContext` est destinée à l'injection de gestionnaires d'entités dans des composants EJB. Spring voit également cette annotation comme un postprocesseur de beans. Il injecte un gestionnaire d'entités dans toute propriété marquée par cette annotation. Il s'assure que toutes les opérations de persistance au sein d'une même transaction sont prises en charge par le même gestionnaire d'entités.

Explications

Pour employer l'injection de contexte, nous déclarons dans notre DAO un champ de gestionnaire d'entités et l'annotons avec `@PersistenceContext`. Spring injectera un gestionnaire d'entités dans ce champ pour la persistance de nos objets.

```
package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
import javax.persistence.Query;

import org.springframework.transaction.annotation.Transactional;

public class JpaCourseDao implements CourseDao {

    @PersistenceContext
    private EntityManager entityManager;

    @Transactional
    public void store(Course course) {
        entityManager.merge(course);
    }

    @Transactional
    public void delete(Long courseId) {
        Course course = entityManager.find(Course.class, courseId);
        entityManager.remove(course);
    }

    @Transactional(readOnly = true)
    public Course findById(Long courseId) {
        return entityManager.find(Course.class, courseId);
    }

    @Transactional(readOnly = true)
    public List<Course> findAll() {
        Query query = entityManager.createQuery("from Course");
        return query.getResultList();
    }
}
```

Pour que toutes les méthodes du DAO soient transactionnelles, nous pouvons les annoter avec `@Transactional`, ou directement la classe. Les opérations de persistance au sein d'une méthode du DAO sont alors exécutées dans la même transaction et, par conséquent, la même session.

Dans le fichier de configuration des beans pour JPA, `beans-jpa.xml`, nous déclarons une instance de `JpaTransactionManager` et activons la gestion déclarative des transactions *via* `<tx:annotation-driven>`. Nous enregistrons une instance de `PersistenceAnnotationBeanPostProcessor` pour injecter des gestionnaires d'entités dans les propriétés marquées par `@PersistenceContext`.

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">
    ...
    <tx:annotation-driven />

    <bean id="transactionManager"
          class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name="entityManagerFactory" ref="entityManagerFactory" />
    </bean>

    <bean name="courseDao"
          class="com.apress.springrecipes.course.jpa.JpaCourseDao" />

    <bean class="org.springframework.orm.jpa.support.➥
               PersistenceAnnotationBeanPostProcessor" />
</beans>
```

Dans Spring 2.5, une instance de `PersistenceAnnotationBeanPostProcessor` est enregistrée automatiquement lorsque l'élément `<context:annotation-config>` est présent. Nous pouvons alors supprimer la déclaration explicite du bean.

```

<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.springframework.org/schema/context
                           http://www.springframework.org/schema/context/spring-context-2.5.xsd
                           http://www.springframework.org/schema/tx
                           http://www.springframework.org/schema/tx/spring-tx-2.5.xsd">

    <context:annotation-config />
    ...
</beans>
```

Ce postprocesseur de beans peut également injecter la fabrique de gestionnaires d'entités dans une propriété marquée par `@PersistenceUnit`. Cela nous permet de créer des gestionnaires d'entités et de gérer les transactions nous-mêmes. Cela revient à injecter la fabrique de gestionnaires d'entités par un mutateur.

```

package com.apress.springrecipes.course.jpa;
...
import javax.persistence.EntityManagerFactory;
import javax.persistence.PersistenceUnit;
public class JpaCourseDao implements CourseDao {
    @PersistenceUnit
    private EntityManagerFactory entityManagerFactory;
    ...
}
```

JpaTemplate convertit les exceptions natives de JPA en exceptions de la hiérarchie `DataAccessException` de Spring. Cependant, lorsqu'on appelle des méthodes natives sur un gestionnaire d'entités JPA, les exceptions lancées sont de type `PersistenceException`, ou toute autre exception Java SE comme `IllegalArgumentException` et `IllegalStateException`. Si nous souhaitons que les exceptions de JPA soient converties en exceptions `DataAccessException` de Spring, nous devons appliquer l'annotation `@Repository` à la classe du DAO.

```
package com.apress.springrecipes.course.jpa;
...
import org.springframework.stereotype.Repository;

@Repository("courseDao")
public class JpaCourseDao implements CourseDao {
    ...
}
```

Nous enregistrons ensuite une instance de `PersistenceExceptionTranslationPostProcessor` pour convertir les exceptions natives de JPA en exceptions de la hiérarchie `DataAccessException` de Spring. Nous ajoutons également l'élément `<context:component-scan>` afin de supprimer la déclaration originale du bean `JpaCourseDao`, car `@Repository` est une annotation stéréotype dans Spring 2.5.

```
<beans ...>
    ...
    <context:component-scan
        base-package="com.apress.springrecipes.course.jpa" />
    <bean class="org.springframework.dao.annotation.➥
        PersistenceExceptionTranslationPostProcessor" />
</beans>
```

9.6 En résumé

Dans ce chapitre, nous avons vu comment intégrer deux frameworks ORM répandus, Hibernate et JPA, dans nos applications Spring. Puisque la prise en charge des différents frameworks ORM dans Spring est cohérente, nous pouvons facilement appliquer les techniques décrites à d'autres frameworks ORM.

Lorsqu'on utilise un framework ORM, il faut configurer sa fabrique de ressources (par exemple une fabrique de sessions dans Hibernate ou une fabrique de gestionnaires d'entités dans JPA) à l'aide de son API. Spring fournit plusieurs beans de fabrique pour que nous puissions créer une fabrique de ressources sous forme d'un bean unique dans le conteneur IoC et qu'elle puisse être partagée par plusieurs beans grâce à l'injection de dépendance. Cette fabrique de ressources peut également bénéficier des autres fonctionnalités d'accès aux données de Spring, comme les sources de données et les gestionnaires de transactions.

Lorsqu'on utilise un framework ORM seul, nous devons répéter certaines tâches standard dans chaque opération du DAO. Spring simplifie l'utilisation d'un framework ORM en fournissant des classes templates, des classes de support des DAO et des implémentations de gestionnaires de transactions. Nous pouvons les utiliser de manière similaire dans les différents frameworks ORM, ainsi que pour JDBC.

Nous pouvons également utiliser les API natives d'Hibernate et de JPA pour implémenter des DAO compatibles avec la prise en charge de l'ORM dans Spring. De cette manière, ces DAO ne dépendent plus d'une API de Spring et peuvent présenter les mêmes avantages que l'utilisation des classes templates.

Le chapitre suivant explique comment construire des applications web avec le framework Spring MVC.

10

Framework Spring MVC

Au sommaire de ce chapitre

- ✓ Développer une application web simple avec Spring MVC
- ✓ Associer des requêtes à des gestionnaires
- ✓ Intercepter des requêtes avec des intercepteurs de gestionnaire
- ✓ Déterminer les paramètres régionaux de l'utilisateur
- ✓ Externaliser les messages dépendant de la localisation
- ✓ Déterminer les vues d'après leur nom
- ✓ Associer des exceptions aux vues
- ✓ Construire des objets *ModelAndView*
- ✓ Créer un contrôleur avec une vue paramétrée
- ✓ Gérer des formulaires avec des contrôleurs
- ✓ Gérer les formulaires multipages
- ✓ Regrouper plusieurs actions dans un contrôleur
- ✓ Créer des vues Excel et PDF
- ✓ Développer des contrôleurs avec des annotations
- ✓ En résumé

Ce chapitre détaille le développement d'une application web à l'aide du framework Spring MVC. Ce module est l'un des plus importants de Spring. Il se fonde sur le conteneur Spring IoC, dont il exploite les fonctionnalités pour simplifier sa configuration. La plupart des éléments de configuration de Spring MVC sont placés dans les fichiers de configuration des beans.

Le design pattern MVC (*Modèle-Vue-Contrôleur*) est très répandu dans la conception d'interfaces utilisateur. Il permet de découpler la logique métier de l'interface graphique, en séparant les rôles de modèle, de vue et de contrôleur dans une application. Les *modèles* encapsulent les données de l'application, qui sont affichées par les vues. Les *vues* se chargent uniquement de l'affichage des données, sans comprendre un seul élément de la

logique métier. Les *contrôleurs* reçoivent les demandes des utilisateurs et invoquent des services dorsaux (*back-end*) pour les traitements métier. Une fois ces traitements réalisés, les services dorsaux retournent les données qui devront être affichées par les vues. Les contrôleurs collectent ces données et préparent des modèles passés aux vues. L'idée au cœur du pattern MVC est de séparer la logique métier de l'interface utilisateur afin que la modification de l'une puisse se faire sans affecter l'autre.

Dans une application Spring MVC, les modèles sont généralement constitués d'objets de domaine manipulés par la couche de service et conservés par la couche de persistance. Les vues sont des templates JSP écrits avec la bibliothèque de balises standard de Java (JSTL, *Java Standard Tag Library*). Dans Spring 2.0 et les versions antérieures, les contrôleurs doivent étendre l'une des classes de contrôleurs de Spring. Dans Spring 2.5, les contrôleurs peuvent être des objets Java quelconques auxquels sont appliquées les annotations de contrôleurs de Spring. Les contrôleurs interagissent avec les composants de la couche de service pour les traitements métier, pour lesquels la gestion des transactions est souvent activée.

À la fin de ce chapitre, vous serez en mesure de développer des applications web Java en utilisant Spring MVC. Vous comprendrez également le fonctionnement des types de contrôleurs et de vues communs de Spring MVC et saurez dans quelles circonstances ils doivent être utilisés. Par ailleurs, vous serez capable de développer des applications web en utilisant l'approche Spring 2.5 fondée sur les annotations.

10.1 Développer une application web simple avec Spring MVC

Problème

Nous souhaitons développer une application web simple avec Spring MVC de manière à comprendre les concepts de base et la configuration de ce framework.

Solution

`DispatcherServlet` est le composant central de Spring MVC. Comme son nom l'indique, il distribue les requêtes vers les gestionnaires appropriés pour qu'ils prennent en charge ces requêtes. Il s'agit de la seule servlet que nous devons configurer dans le descripteur de déploiement web. `DispatcherServlet` implémente l'un des design patterns proposés par Sun dans Java EE et appelé *Contrôleur frontal (Front Controller)*. Il joue le rôle de contrôleur placé devant le framework Spring MVC et chaque requête web passe par lui pour le déroulement du processus de traitement des requêtes.

Lorsqu'une requête web est envoyée à une application Spring MVC, elle est tout d'abord reçue par `DispatcherServlet`. Celui-ci planifie ensuite les différents composants confi-

gurés dans le contexte d'application web de Spring pour la prise en charge de cette requête. La Figure 10.1 illustre le traitement d'une requête dans Spring MVC.

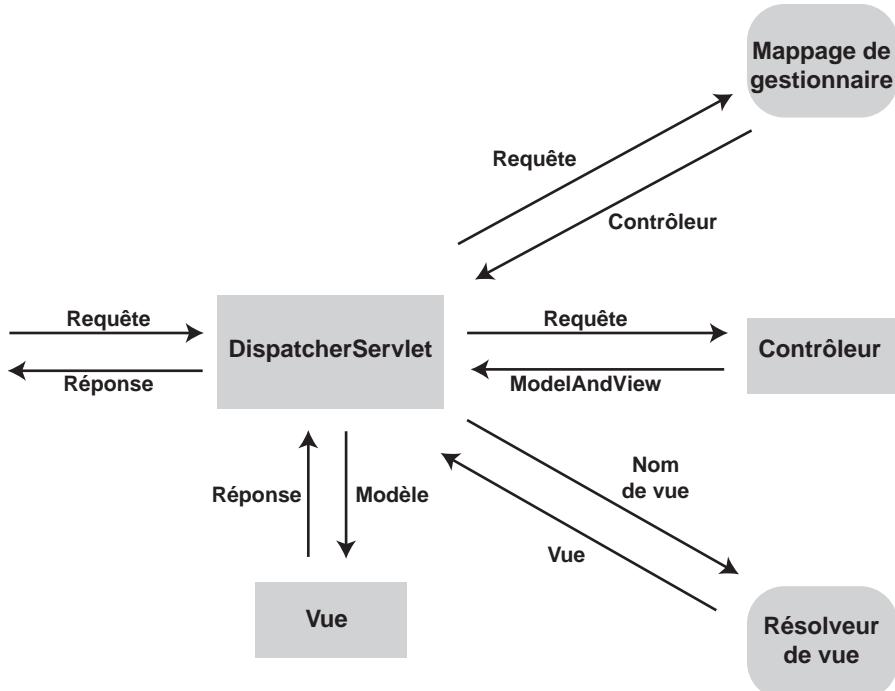


Figure 10.1

Flux principal du traitement d'une requête dans Spring MVC.

Lorsque DispatcherServlet reçoit une requête, il commence par rechercher un gestionnaire adapté pour son traitement. Il associe chaque requête à un gestionnaire en utilisant un ou plusieurs mappages de gestionnaire. Un mappage de gestionnaire est un bean configuré dans le contexte de l'application web qui implémente l'interface HandlerMapping. Son rôle est de retourner le gestionnaire adapté à une requête. En général, les mappages de gestionnaire se fondent sur l'URL de la requête pour trouver le gestionnaire adéquat.

Après que DispatcherServlet a obtenu le gestionnaire approprié, il l'invoque pour traiter la requête. Un gestionnaire est un objet Java quelconque capable de prendre en charge des requêtes web. Dans Spring MVC, un tel gestionnaire est le plus souvent mis en œuvre par un *contrôleur*. Celui-ci invoque généralement les services dorsaux pour réaliser les traitements.

Lorsque le contrôleur a terminé son travail sur la requête, il retourne à DispatcherServlet un nom de modèle et de vue ou, parfois, un objet de vue. Le modèle contient les attributs que le contrôleur souhaite passer à la vue pour qu'elle les affiche. Si le nom d'une vue est retourné, il est converti en un objet de vue pour l'affichage. La classe de base qui lie un modèle et une vue se nomme `ModelAndView`.

Lorsque DispatcherServlet reçoit un nom de modèle et de vue, il convertit le nom logique de la vue en un objet de vue qui se charge de l'affichage. Pour effectuer cette conversion, DispatcherServlet contacte un ou plusieurs résolveurs de vue. Un *résolveur de vue* est un bean configuré dans le contexte d'application web et qui implémente l'interface `ViewResolver`. Son rôle est de retourner un objet de vue qui correspond à un nom logique de vue.

Après que DispatcherServlet a converti un nom de vue en un objet de vue, il affiche celui-ci et lui passe le modèle retourné par le contrôleur. La vue se charge de présenter à l'utilisateur les attributs contenus dans le modèle.

Explications

Supposons que nous développons un système de réservation de terrains pour un centre sportif. Cette application web permet aux utilisateurs d'effectuer des réservations en ligne *via* Internet. Nous utilisons Spring MVC pour réaliser cette application. Tout d'abord, nous créons les classes de domaine suivantes dans le paquetage `domain`.

```
package com.apress.springrecipes.court.domain;
...
public class Reservation {

    private String courtName;
    private Date date;
    private int hour;
    private Player player;
    private SportType sportType;

    // Constructeurs, accesseurs et mutateurs.
    ...
}

---

package com.apress.springrecipes.court.domain;

public class Player {

    private String name;
    private String phone;

    // Constructeurs, accesseurs et mutateurs.
    ...
}
```

```
---  
package com.apress.springrecipes.court.domain;  
  
public class SportType {  
  
    private int id;  
    private String name;  
  
    // Constructeurs, accesseurs et mutateurs.  
    ...  
}
```

Ensuite, nous définissons l’interface suivante dans le paquetage service de manière à offrir un service de réservation à la couche de présentation.

```
package com.apress.springrecipes.court.service;  
...  
public interface ReservationService {  
  
    public List<Reservation> query(String courtName);  
}
```

Dans une application réelle, nous implémenterions cette interface en assurant la persistance dans une base de données. Pour des raisons de simplicité, nous enregistrons les réservations dans une liste et créons quelques réservations pour les tests.

```
package com.apress.springrecipes.court.service;  
...  
public class ReservationServiceImpl implements ReservationService {  
  
    public static final SportType TENNIS = new SportType(1, "Tennis");  
    public static final SportType SOCCER = new SportType(2, "Football");  
  
    private List<Reservation> reservations;  
  
    public ReservationServiceImpl() {  
        reservations = new ArrayList<Reservation>();  
        reservations.add(new Reservation("Tennis #1",  
            new GregorianCalendar(2008, 0, 14).getTime(), 16,  
            new Player("Roger", "N/A"), TENNIS));  
        reservations.add(new Reservation("Tennis #2",  
            new GregorianCalendar(2008, 0, 14).getTime(), 20,  
            new Player("Paul", "N/A"), TENNIS));  
    }  
  
    public List<Reservation> query(String courtName) {  
        List<Reservation> result = new ArrayList<Reservation>();  
        for (Reservation reservation : reservations) {  
            if (reservation.getCourtName().equals(courtName)) {  
                result.add(reservation);  
            }  
        }  
        return result;  
    }  
}
```

Configurer une application Spring MVC

Avant de pouvoir développer une application web avec Spring MVC, nous devons la configurer correctement. En général, cette configuration se fait de la même manière que pour une application web Java standard, excepté l'ajout de deux fichiers de configuration et des bibliothèques requises par Spring MVC.

Les spécifications de Java EE définissent la structure valide des répertoires pour une application web Java. Par exemple, nous devons placer un descripteur de déploiement `web.xml`, à la racine du répertoire `WEB-INF`. Les fichiers de classes et les fichiers JAR de cette application doivent se trouver, respectivement, dans les répertoires `WEB-INF/classes` et `WEB-INF/lib`.

Dans le cas de notre système de réservation de terrains, nous créons la structure de répertoires suivante. Les fichiers présentés en gras sont les fichiers de configuration propres à Spring.

INFO

Pour développer une application web avec Spring MVC, vous devez copier le fichier `spring-webmvc.jar` (situé dans le répertoire `dist/modules` de l'installation de Spring) dans le répertoire `WEB-INF/lib`. Si vous utilisez JSTL dans les pages JSP, vous devez également y copier `standard.jar` (situé dans `lib/jakarta-taglibs`) et `jstl.jar` (situé dans `lib/j2ee`).

```
court/
  css/
  images/
  WEB-INF/
    classes/
    lib/
      commons-logging.jar
      jstl.jar
      spring-webmvc.jar
      spring.jar
      standard.jar
  jsp/
    welcome.jsp
    reservationQuery.jsp
court-service.xml
court-servlet.xml
  web.xml
```

Les fichiers situés hors du répertoire `WEB-INF` sont directement accessibles par les utilisateurs au travers d'URL. Par conséquent, les fichiers CSS et les fichiers d'images doivent être placés à cet endroit. Avec Spring MVC, les fichiers JSP servent uniquement de templates. Ils sont lus par le framework pour la génération du contenu dynamique, non par les utilisateurs. Pour empêcher un accès direct à ces fichiers, ils sont donc placés

dans le répertoire WEB-INF. Toutefois, certains serveurs d'applications n'autorisent pas les applications web à lire en interne les fichiers qui se trouvent dans la hiérarchie WEB-INF. Dans ce cas, il faut les placer hors de ce répertoire.

Créer les fichiers de configuration

Le descripteur de déploiement web web.xml est le principal fichier de configuration d'une application web Java. Nous y définissons les servlets de l'application et leur correspondance avec les requêtes web. Pour une application Spring MVC, nous définissons uniquement une seule instance de DispatcherServlet qui joue le rôle de contrôleur frontal pour Spring MVC. Toutefois, rien ne nous interdit de définir d'autres servlets.

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <display-name>Court Reservation System</display-name>

    <servlet>
        <servlet-name>court</servlet-name>
        <servlet-class>
            org.springframework.web.servlet.DispatcherServlet
        </servlet-class>
        <load-on-startup>1</load-on-startup>
    </servlet>

    <servlet-mapping>
        <servlet-name>court</servlet-name>
        <url-pattern>*.htm</url-pattern>
    </servlet-mapping>
</web-app>
```

Dans ce descripteur de déploiement, nous définissons une servlet de type DispatcherServlet. Il s'agit de la classe de servlet principale de Spring MVC, qui reçoit les requêtes web et les distribue aux gestionnaires appropriés. Nous la nommons court et lui associons les URL dont l'extension est .htm. Cette extension est arbitraire, mais, pour éviter de révéler la technologie d'implémentation aux utilisateurs de l'application web, il est généralement conseillé de choisir .htm, .html ou comparable.

Le nom de la servlet est également utilisé par DispatcherServlet pour trouver le fichier qui contient les éléments de configuration de Spring MVC. Par défaut, le nom de ce fichier est celui de la servlet auquel est adjoint -servlet.xml. Nous pouvons indiquer explicitement le fichier de configuration dans le paramètre contextConfigLocation de la servlet. Dans notre exemple, la servlet court charge le fichier de configuration par défaut nommé court-servlet.xml. Il s'agit d'un fichier standard de configuration des beans Spring :

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">
    ...
</beans>
```

Plus loin, nous configurerons Spring MVC en déclarant plusieurs beans Spring dans ce fichier. Nous pouvons également y déclarer d'autres composants applicatifs, comme des objets d'accès aux données et des objets de service. Toutefois, il est fortement déconseillé de mélanger des beans de couches différentes dans un même fichier de configuration. À la place, il est préférable de déclarer un fichier de configuration des beans pour chaque couche, comme `court-persistence.xml` pour la couche de persistance et `court-service.xml` pour la couche de service. Le fichier `court-service.xml` contient, par exemple, l'objet de service suivant :

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="reservationService"
          class="com.apress.springrecipes.court.service.➥
          ReservationServiceImpl" />
</beans>
```

Pour que Spring charge nos fichiers de configuration, en plus de `court-servlet.xml`, nous devons définir l'écouteur de servlet `ContextLoaderListener` dans `web.xml`. Par défaut, il charge le fichier de configuration des beans `/WEB-INF/application-Context.xml`, mais nous pouvons en indiquer un autre dans le paramètre de contexte `contextConfigLocation`. Pour préciser plusieurs fichiers de configuration, il suffit de les séparer par des virgules ou des espaces.

```
<web-app ...>
    <context-param>
        <param-name>contextConfigLocation</param-name>
        <param-value>/WEB-INF/court-service.xml</param-value>
    </context-param>

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

`ContextLoaderListener` charge les fichiers de configuration des beans indiqués dans le contexte d'application racine. En revanche, chaque instance de `DispatcherServlet` charge son fichier de configuration dans son propre contexte d'application, le contexte d'application racine étant son parent. Par conséquent, le contexte chargé par chaque ins-

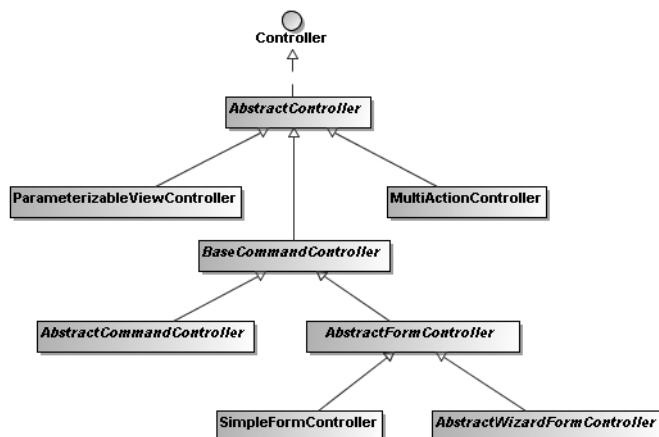
tance de DispatcherServlet peut accéder aux beans déclarés dans le contexte d'application racine, mais pas l'inverse, et même les redéfinir. Toutefois, les contextes chargés par les différentes instances de DispatcherServlet ne peuvent pas accéder l'un à l'autre.

Créer des contrôleurs Spring MVC

Spring MVC fournit plusieurs types de contrôleurs pour différents scénarios d'utilisation. La Figure 10.2 illustre les types de contrôleurs communs dans Spring MVC.

Figure 10.2

Types de contrôleurs communs dans Spring MVC.



`Controller` est l'interface de base de toutes les classes de contrôleurs dans Spring MVC. Nous pouvons créer notre propre contrôleur en implémentant cette interface. Dans la méthode `handleRequest()`, le traitement d'une requête web peut se faire comme dans une servlet. Nous devons retourner un objet de type `ModelAndView` qui contient un nom de vue ou un objet de vue, ainsi que les attributs du modèle. Par exemple, voici le contrôleur d'accueil créé pour notre système de réservation de terrains.

INFO

Pour développer une application web qui implique l'API des servlets, vous devez inclure le fichier `servlet-api.jar` (situé dans le répertoire `lib/j2ee` de l'installation de Spring) dans le chemin d'accès aux classes.

```
package com.apress.springrecipes.court.web;
...
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
```

```

public class WelcomeController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        Date today = new Date();
        return new ModelAndView("welcome", "today", today);
    }
}

```

Ce contrôleur crée un objet `java.util.Date` pour lire la date courante. Il retourne un objet `ModelAndView` qui contient le nom de vue `welcome` et la date courante en tant qu'attribut du modèle qui sera affiché par la vue indiquée.

Après avoir créé la classe du contrôleur, nous devons déclarer son instance dans le contexte d'application web. Puisqu'il s'agit d'un composant de la couche de présentation, nous plaçons cette déclaration dans `court-servlet.xml`. Par défaut, `DispatcherServlet` utilise un `BeanNameUrlHandlerMapping` pour le mappage des gestionnaires, mais nous pouvons en changer explicitement. Cette classe associe les requêtes aux gestionnaires en fonction des motifs d'URL définis dans les noms donnés aux beans des gestionnaires. Puisque l'attribut `id` d'un élément `<bean>` ne peut pas contenir de caractère `/`, nous devons utiliser l'attribut `name` à la place.

```

<bean name="/welcome.htm"
      class="com.apress.springrecipes.court.web.WelcomeController" />

```

Si nous voulons que notre contrôleur dispose de certaines fonctionnalités de base d'un contrôleur, comme le filtrage des méthodes HTTP (GET, POST et HEAD) et la génération des en-têtes de contrôle du cache dans les réponses HTTP, il doit dériver de la classe `AbstractController`, qui implémente l'interface `Controller`. La méthode de cette classe de contrôleur à redéfinir se nomme `handleRequestInternal()`.

```

package com.apress.springrecipes.court.web;
...
import org.springframework.web.servlet.mvc.AbstractController;

public class WelcomeController extends AbstractController {

    public ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        Date today = new Date();
        return new ModelAndView("welcome", "today", today);
    }
}

```

L'exploitation des fonctionnalités mentionnées précédemment se fait en fixant plusieurs propriétés héritées de la classe `AbstractController` :

```

<bean name="/welcome.htm"
      class="com.apress.springrecipes.court.web.WelcomeController">
    <property name="supportedMethods" value="GET" />
    <property name="cacheSeconds" value="60" />
</bean>

```

Par exemple, les méthodes HTTP prises en charge par le contrôleur sont indiquées à l'aide de la propriété `supportedMethods`, en utilisant la virgule comme séparateur. Si la méthode d'une requête HTTP n'est pas présente dans cette liste, une exception `ServletException` est lancée. Le contrôleur peut également définir la durée de vie d'un document dans le cache, qui sera indiquée dans l'en-tête de réponse HTTP.

Un contrôleur classique accepte des paramètres de requête HTTP et invoque des services dorsaux pour les traitements métier. Par exemple, voici un contrôleur qui prend en charge les requêtes de réservation d'un terrain :

```
package com.apress.springrecipes.court.web;
...
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.bind.ServletRequestUtils;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class ReservationQueryController extends AbstractController {

    private ReservationService reservationService;

    public void setReservationService(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    public ModelAndView handleRequestInternal(HttpServletRequest request,
                                              HttpServletResponse response) throws Exception {
        String courtName =
            ServletRequestUtils.getStringParameter(request, "courtName");

        Map<String, Object> model = new HashMap<String, Object>();
        if (courtName != null) {
            model.put("courtName", courtName);
            model.put("reservations", reservationService.query(courtName));
        }
        return new ModelAndView("reservationQuery", model);
    }
}
```

Nous commençons par retrouver le paramètre `courtName` passé avec la requête. Pour cela, nous pouvons utiliser `HttpServletRequest.getParameter()`, disponible dans l'API standard des servlets, mais Spring fournit des méthodes statiques plus pratiques dans la classe `ServletRequestUtils`. Ces méthodes nous permettent d'obtenir les paramètres de requête dans le bon type, ainsi que les paramètres obligatoires. Cela nous évite les conversions et les vérifications. Lorsque le modèle doit passer plusieurs attributs à la vue, nous les plaçons dans une table d'association donnée au constructeur de `ModelAndView`.

Nous devons à présent déclarer ce contrôleur dans court-servlet.xml, en faisant référence au bean de service pour la réservation, qui est déclaré dans le fichier de configuration de la couche de service, court-service.xml.

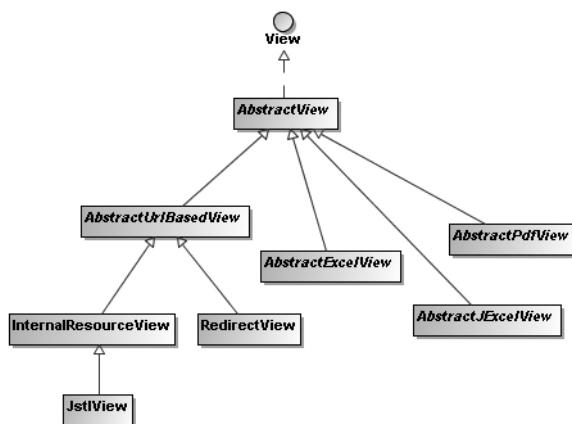
```
<bean name="/reservationQuery.htm"
      class="com.apress.springrecipes.court.web.ReservationQueryController">
    <property name="reservationService" ref="reservationService" />
</bean>
```

Créer des vues JSP

Spring MVC prend en charge plusieurs types de vues pour les différentes technologies de présentation. La Figure 10.3 présente les types de vues communs dans Spring MVC.

Figure 10.3

Types de vues communs dans Spring MVC.



Dans une application Spring MVC, les vues correspondent généralement à des templates JSP écrits à l'aide de la bibliothèque JSTL. Lorsque DispatcherServlet reçoit un nom de vue de la part d'un gestionnaire, il le convertit en un objet de vue pour l'affichage. Par exemple, nous configurons un bean `InternalResourceViewResolver` dans le contexte d'application web de manière à convertir les noms de vues en fichiers JSP stockés dans le répertoire `/WEB-INF/jsp/`.

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

Ensuite, nous créons le template JSP suivant pour le contrôleur d'accueil. Nous le nommons `welcome.jsp` et le plaçons dans le répertoire `/WEB-INF/jsp/`.

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

<html>
<head>
<title>Bienvenue</title>
</head>

<body>
<h2>Bienvenue sur le système de réservation des terrains</h2>
Nous sommes le <fmt:formatDate value="${today}" pattern="yyyy-MM-dd" />.
</body>
</html>
```

Dans ce template JSP, nous utilisons la bibliothèque de balises `fmt` apportée par JSTL pour mettre en forme l'attribut `today` du modèle conformément au motif `yyyy-MM-dd`. Il ne faut pas oublier d'inclure la définition de la bibliothèque `fmt` au début du template.

Nous créons également un autre template JSP pour le contrôleur de demande de réservation. Nous le nommons `reservationQuery.jsp` pour qu'il corresponde au nom de la vue.

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

<html>
<head>
<title>Demande de réservation</title>
</head>

<body>
<form method="POST">
Nom du terrain
<input type="text" name="courtName" value="${courtName}" />
<input type="submit" value="Query" />
</form>

<table border="1">
<tr>
<th>Nom du terrain</th>
<th>Date</th>
<th>Horaire</th>
<th>Joueur</th>
</tr>
<c:forEach items="${reservations}" var="reservation">
<tr>
<td>${reservation.courtName}</td>
<td>
<fmt:formatDate value="${reservation.date}" pattern="yyyy-MM-dd" />
</td>
<td>${reservation.hour}</td>
<td>${reservation.player.name}</td>
</tr>
</c:forEach>
</table>
</body>
</html>
```

Dans ce template JSP, nous utilisons un formulaire pour que les utilisateurs saisissent le nom du terrain qu'ils souhaitent réserver et nous parcourons les attributs du modèle de réservation avec une balise <c:forEach> pour générer le tableau correspondant.

Déployer l'application web

Pour le développement des applications web, il est fortement conseillé d'installer localement un serveur d'applications Java EE qui propose un conteneur web permettant de procéder aux tests et au débogage. Pour simplifier la configuration et le déploiement, nous proposons d'utiliser Apache Tomcat 6.0 comme conteneur web. Nous créons le descripteur de contexte court.xml dans le répertoire conf/Catalina/localhost, en faisant référence au répertoire racine de notre application web.

```
<Context docBase="c:/eclipse/workspace/Court/court" />
```

Par défaut, Tomcat écoute sur le port 8080 et utilise le nom du descripteur de contexte (court dans ce cas) comme chemin de contexte. Par conséquent, le contrôleur d'accueil et celui de demande de réservation sont accessibles au travers des URL suivantes :

```
http://localhost:8080/court/welcome.htm  
http://localhost:8080/court/reservationQuery.htm
```

10.2 Associer des requêtes à des gestionnaires

Problème

Lorsque DispatcherServlet reçoit une requête web, il la transfère simplement au gestionnaire approprié pour son traitement. Nous souhaitons définir notre propre stratégie d'association des requêtes à des gestionnaires.

Solution

Dans une application Spring MVC, les requêtes web sont associées aux gestionnaires par un ou plusieurs beans de mappage de gestionnaires déclarés dans le contexte d'application web. Ces beans doivent implémenter l'interface HandlerMapping pour que DispatcherServlet les détecte automatiquement. Spring MVC fournit plusieurs implémentations de HandlerMapping pour mettre en œuvre différentes stratégies d'association. Par défaut, si nous ne configurons explicitement aucun bean de mappage de gestionnaires, DispatcherServlet utilise la classe BeanNameUrlHandlerMapping, qui associe les requêtes aux gestionnaires en fonction des motifs d'URL indiqués dans les noms des beans. Par conséquent, si cette stratégie convient, il est inutile de définir notre propre mappage des gestionnaires.

Les mappages de gestionnaires associent les URL en fonction de leur chemin relativement au chemin du contexte (le chemin où est déployé le contexte d'application web) et au chemin de servlet (le chemin associé à DispatcherServlet). Ainsi, pour l'URL `http://localhost:8080/court/welcome.htm`, le chemin utilisé pour la correspondance est `/welcome.htm`, car le chemin du contexte est `/court` et le chemin de servlet n'existe pas.

Expllications

Associer les requêtes d'après des noms de beans

La stratégie par défaut, et la plus simple, pour associer des requêtes à des gestionnaires se fonde sur les noms de beans des gestionnaires. Pour la mettre en œuvre, nous devons déclarer le nom de bean de chaque gestionnaire sous forme d'un motif d'URL. Les caractères génériques sont acceptés, ce qui permet à un gestionnaire de prendre en charge plusieurs URL. Lorsque le nom d'un bean correspond à l'URL d'une requête, DispatcherServlet transmet la requête à ce gestionnaire.

```
<beans ...>
    ...
    <bean name="/welcome.htm"
          class="com.apress.springrecipes.court.web.WelcomeController">
        ...
    </bean>

    <bean name="/reservationQuery.htm"
          class="com.apress.springrecipes.court.web.ReservationQueryController">
        ...
    </bean>
</beans>
```

Avec cette stratégie de correspondance, nous fixons le nom d'un gestionnaire dans l'attribut name car le caractère / n'est pas accepté dans l'attribut id.

Associer les requêtes d'après des noms de classes de contrôleurs

Spring fournit également la classe `ControllerClassNameHandlerMapping` pour le mapping des gestionnaires, fondée sur l'implémentation de l'interface `Controller`. Elle génère automatiquement des mappages d'après les noms de classes des contrôleurs déclarés dans le contexte d'application web.

```
<beans ...>
    ...
    <bean class="org.springframework.web.servlet.mvc.support.<span style="color: #0000ff; font-weight: bold;">➥
          ControllerClassNameHandlerMapping" />

    <bean class="com.apress.springrecipes.court.web.WelcomeController">
        ...
    </bean>
```

```

<bean class="com.apress.springrecipes.court.web.➥
    ReservationQueryController">
    ...
</bean>
</beans>

```

Pour ces déclarations de beans, ControllerClassNameHandlerMapping génère automatiquement les mappages de gestionnaires en retirant le suffixe Controller du nom de la classe et en convertissant la partie restante en minuscules.

```

WelcomeController → /welcome*
ReservationQueryController → /reservationquery*

```

Toutefois, si nous souhaitons que nos motifs d'URL respectent la convention de nommage des variables en Java, par exemple pour générer /reservationQuery* à la place de /reservationquery*, nous fixons la propriété caseSensitive à true. Par ailleurs, nous pouvons donner dans la propriété pathPrefix un préfixe pour les motifs d'URL générés. Si nous indiquons également le paquetage de base dans la propriété basePackage, le sous-paquetage relatif à ce paquetage de base est inclus dans le mappage.

```

<bean class="org.springframework.web.servlet.mvc.support.➥
    ControllerClassNameHandlerMapping">
    <property name="caseSensitive" value="true" />
    <property name="pathPrefix" value="/reservation" />
    <property name="basePackage" value="com.apress.springrecipes.court" />
</bean>

```

Cette définition de ControllerClassNameHandlerMapping génère les mappages de gestionnaires suivants :

```

WelcomeController → /reservation/web/welcome*
ReservationQueryController → /reservation/web/reservationQuery*

```

Associer des requêtes d'après des définitions de mappage personnalisées

La stratégie la plus directe et la plus souple pour associer les requêtes à des gestionnaires consiste à définir explicitement les mappages entre des motifs d'URL et des gestionnaires. Pour cela, nous utilisons SimpleUrlHandlerMapping.

```

<beans ...>
    ...
<bean class="org.springframework.web.servlet.handler.➥
    SimpleUrlHandlerMapping">
    <property name="mappings">
        <props>
            <prop key="/welcome.htm">welcomeController</prop>
            <prop key="/reservationQuery.htm">
                reservationQueryController
            </prop>
        </props>
    </property>
</bean>

```

```
<bean id="welcomeController"
      class="com.apress.springrecipes.court.web.WelcomeController">
    ...
</bean>

<bean id="reservationQueryController"
      class="com.apress.springrecipes.court.web.ReservationQueryController">
    ...
</bean>
</beans>
```

SimpleUrlHandlerMapping accepte la définition d'un mappage sous forme d'un objet Properties. Les clés des propriétés sont les motifs d'URL, tandis que les valeurs sont les identifiants ou les noms des gestionnaires. Les motifs d'URL peuvent également contenir des caractères génériques pour qu'un gestionnaire puisse correspondre à plusieurs URL.

Associer des requêtes avec plusieurs stratégies

Dans une application web comprenant un grand nombre de gestionnaires, le choix d'une seule stratégie de mappage des gestionnaires ne suffit pas toujours. En général, ControllerClassNameHandlerMapping permet de satisfaire la majorité des besoins de mappage. Cependant, certains doivent se faire explicitement avec SimpleUrlHandlerMapping. Dans ce cas, nous combinons les deux stratégies.

```
<beans ...>
  ...
  <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="mappings">
      <props>
        <prop key="/index.htm">welcomeController</prop>
        <prop key="/main.htm">welcomeController</prop>
      </props>
    </property>
    <property name="order" value="0" />
  </bean>

  <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping">
    <property name="caseSensitive" value="true" />
    <property name="order" value="1" />
  </bean>
</beans>
```

Lorsque plusieurs stratégies sont employées en même temps, il est important de préciser la priorité des mappages. Pour cela, nous fixons les propriétés order des beans pour le mappage des gestionnaires. Plus la valeur est faible, plus la priorité est élevée. Dans l'exemple précédent, les requêtes web sont associées aux gestionnaires de la manière suivante :

```
/index.htm → WelcomeController  
/main.htm → WelcomeController  
/welcome* → WelcomeController  
/reservationQuery* → ReservationQueryController
```

10.3 Intercepter des requêtes avec des intercepteurs de gestionnaire

Problème

Les filtres définis dans l'API des servlets permettent de traiter chaque requête web avant et après sa prise en charge par une servlet. Nous souhaitons obtenir un fonctionnement comparable dans un contexte d'application web de Spring afin de profiter des fonctionnalités du conteneur.

Par ailleurs, nous souhaitons parfois pré- et posttraiter uniquement les requêtes web prises en charge par certains gestionnaires Spring MVC, ainsi que manipuler les attributs du modèle retourné par ces gestionnaires avant qu'il ne soit passé aux vues.

Solution

Par l'intermédiaire des *intercepteurs de gestionnaire*, Spring MVC nous permet d'intercepter des requêtes web afin de leur appliquer un pré- et un posttraitement. Les intercepteurs de gestionnaire sont configurés dans le contexte d'application web de Spring. Ils peuvent donc bénéficier des fonctionnalités du conteneur et faire référence aux beans qui y sont déclarés. Puisqu'un intercepteur de gestionnaire est enregistré pour des mappages de gestionnaires précis, il n'intercepte que les requêtes sélectionnées par ces mappages.

Chaque intercepteur de gestionnaire implémente l'interface `HandlerInterceptor`, qui contient trois méthodes de rappel que nous devons mettre en œuvre : `preHandle()`, `postHandle()` et `afterCompletion()`. Les deux premières sont invoquées avant et après le traitement d'une requête par un gestionnaire. La deuxième nous donne également accès à l'objet `ModelAndView` retourné, ce qui nous permet de manipuler les attributs du modèle. La dernière méthode est appelée une fois l'intégralité du traitement de la requête terminée, c'est-à-dire après l'affichage de la vue.

Explications

Supposons que nous voulions mesurer le temps de traitement de chaque requête web par chaque gestionnaire et présenter les résultats à l'utilisateur. Pour cela, nous créons notre propre intercepteur de gestionnaire.

```
package com.apress.springrecipes.court.web;  
...  
import org.springframework.web.servlet.HandlerInterceptor;  
import org.springframework.web.servlet.ModelAndView;
```

```
public class MeasurementInterceptor implements HandlerInterceptor {  
  
    public boolean preHandle(HttpServletRequest request,  
                             HttpServletResponse response, Object handler) throws Exception {  
        long startTime = System.currentTimeMillis();  
        request.setAttribute("startTime", startTime);  
        return true;  
    }  
  
    public void postHandle(HttpServletRequest request,  
                           HttpServletResponse response, Object handler,  
                           ModelAndView modelAndView) throws Exception {  
        long startTime = (Long) request.getAttribute("startTime");  
        request.removeAttribute("startTime");  
  
        long endTime = System.currentTimeMillis();  
        modelAndView.addObject("handlingTime", endTime - startTime);  
    }  
  
    public void afterCompletion(HttpServletRequest request,  
                               HttpServletResponse response, Object handler, Exception ex)  
        throws Exception {  
    }  
}
```

Dans la méthode `preHandle()` de cet intercepteur, nous enregistrons l'heure de début dans un attribut de la requête. Cette méthode doit retourner `true` pour que `DispatcherServlet` poursuive le traitement de la requête. Sinon `DispatcherServlet` suppose que la méthode a déjà traité la requête et retourne directement la réponse à l'utilisateur. Ensuite, dans la méthode `postHandle()`, nous récupérons l'heure de début à partir de l'attribut de la requête et la comparons à l'heure actuelle. Nous calculons la durée totale et ajoutons le résultat dans le modèle afin de le passer à la vue. Enfin, puisque la méthode `afterCompletion()` n'a rien à faire, elle reste vide.

Lorsque l'on implémente une interface, il faut mettre en œuvre toutes les méthodes même si certaines ne sont pas nécessaires. Une meilleure solution consiste à étendre la classe adaptateur de l'intercepteur, qui fournit une implémentation par défaut de toutes les méthodes. Il nous suffit ensuite de redéfinir celles qui nous intéressent.

```
package com.apress.springrecipes.court.web;  
...  
import org.springframework.web.servlet.ModelAndView;  
import org.springframework.web.handler.HandlerInterceptorAdapter;  
  
public class MeasurementInterceptor extends HandlerInterceptorAdapter {  
  
    public boolean preHandle(HttpServletRequest request,  
                             HttpServletResponse response, Object handler) throws Exception {  
        ...  
    }  
}
```

```

public void postHandle(HttpServletRequest request,
    HttpServletResponse response, Object handler,
    ModelAndView modelAndView) throws Exception {
    ...
}
}

```

Un intercepteur de gestionnaire est enregistré avec un bean de mappage des gestionnaires pour qu'il puisse intercepter les requêtes web désignées par ce bean. Dans la propriété `interceptors`, de type tableau, nous pouvons indiquer plusieurs intercepteurs pour un mappage de gestionnaires. Si plusieurs mappages de gestionnaires sont configurés dans le contexte d'application web et si nous souhaitons intercepter toutes les requêtes auxquelles ils correspondent, nous devons enregistrer l'intercepteur pour chacun d'eux.

```

<beans ...>
    ...
    <bean id="measurementInterceptor"
        class="com.apress.springrecipes.court.web.MeasurementInterceptor" />

    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="interceptors">
            <list>
                <ref bean="measurementInterceptor" />
            </list>
        </property>
    ...
</bean>

<bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping">
    <property name="interceptors">
        <list>
            <ref bean="measurementInterceptor" />
        </list>
    </property>
    ...
</bean>
</beans>

```

Nous affichons la durée dans `welcome.jsp` de manière à vérifier le fonctionnement de l'intercepteur. Puisque le travail du contrôleur `WelcomeController` est réduit, il est probable que le temps de traitement soit égal à 0 milliseconde. Dans ce cas, nous pouvons ajouter une instruction `sleep` dans la classe pour allonger le temps de traitement.

```

<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>

<html>
<head>
<title>Bienvenue</title>
</head>

```

```
<body>
...
<hr />
Temps de traitement : ${handlingTime} ms
</body>
</html>
```

10.4 Déterminer les paramètres régionaux de l'utilisateur

Problème

Pour qu'une application web prenne en charge l'internationalisation, nous devons identifier les paramètres régionaux de chaque utilisateur et afficher le contenu conformément à ces paramètres.

Solution

Dans une application Spring MVC, les paramètres régionaux d'un utilisateur sont déterminés par un résolveur de localisation qui implémente l'interface `LocaleResolver`. Spring MVC fournit plusieurs implémentations de `LocaleResolver` pour que nous puissions obtenir les paramètres régionaux en fonction de différents critères. Il est même possible de créer son propre résolveur de localisation en implémentant cette interface.

Pour définir un résolveur de localisation, il suffit d'enregistrer un bean de type `LocaleResolver` dans le contexte d'application web. Pour que `DispatcherServlet` détecte automatiquement ce bean, son nom doit être `localeResolver`. Nous ne pouvons enregistrer qu'un seul résolveur de localisation par `DispatcherServlet`.

Explications

Obtenir les paramètres régionaux à partir d'un en-tête de requête HTTP

Par défaut, Spring utilise le résolveur de localisation `AcceptHeaderLocaleResolver`. Celui-ci détermine les paramètres régionaux en examinant l'en-tête `accept-language` de la requête HTTP. Cet en-tête est fixé par le navigateur web de l'utilisateur en fonction des paramètres régionaux définis dans le système d'exploitation sous-jacent. Ce résolveur de localisation ne peut pas modifier les paramètres régionaux de l'utilisateur car il ne peut pas intervenir sur la configuration de son système d'exploitation.

Déterminer les paramètres régionaux à partir d'un attribut de session

SessionLocaleResolver met en œuvre une autre méthode. Ce résolveur détermine les paramètres régionaux en examinant un attribut prédéfini de la session d'un utilisateur. Si cet attribut n'existe pas, il examine l'en-tête HTTP accept-language.

```
<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
    <property name="defaultLocale" value="fr" />
</bean>
```

La valeur de la propriété defaultLocale est utilisée lorsque l'attribut de session n'existe pas. Ce résolveur de localisation est capable de changer les paramètres régionaux d'un utilisateur par modification de l'attribut de session qui les définit.

Déterminer les paramètres régionaux à partir d'un cookie

CookieLocaleResolver détermine les paramètres régionaux en examinant un cookie du navigateur de l'utilisateur. Si le cookie n'existe pas, il examine l'en-tête HTTP accept-language.

```
<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver" />
```

Il est possible de changer le cookie utilisé par ce résolveur de localisation à l'aide des propriétés cookieName et cookieMaxAge. Cette dernière indique la durée de vie du cookie en secondes. La valeur -1 invalide le cookie après la fermeture du navigateur.

```
<bean id="localeResolver"
      class="org.springframework.web.servlet.i18n.CookieLocaleResolver">
    <property name="cookieName" value="language" />
    <property name="cookieMaxAge" value="3600" />
    <property name="defaultLocale" value="fr" />
</bean>
```

La propriété defaultLocale est utilisée lorsque le cookie n'existe pas dans le navigateur de l'utilisateur. Ce résolveur de localisation est capable de changer les paramètres régionaux d'un utilisateur en modifiant le cookie qui les définit.

Modifier les paramètres régionaux d'un utilisateur

Outre la modification des paramètres régionaux d'un utilisateur en invoquant explicitement LocaleResolver.setLocale(), nous pouvons y procéder en appliquant un intercepteur LocaleChangeInterceptor aux mappages de gestionnaires. Cet intercepteur détecte dans la requête HTTP la présence d'un paramètre particulier, dont le nom est indiqué par la propriété paramName de l'intercepteur. Lorsque ce paramètre est défini dans la requête en cours, l'intercepteur l'utilise pour modifier les paramètres régionaux de l'utilisateur.

```
<beans ...>
    ...
    <bean id="localeChangeInterceptor"
          class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
        <property name="paramName" value="language" />
    </bean>

    <bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="interceptors">
            <list>
                ...
                <ref bean="localeChangeInterceptor" />
            </list>
        </property>
    ...
    </bean>

    <bean class="org.springframework.web.servlet.mvc.support.ControllerClassNameHandlerMapping">
        <property name="interceptors">
            <list>
                ...
                <ref bean="localeChangeInterceptor" />
            </list>
        </property>
    ...
    </bean>
</beans>
```

LocaleChangeInterceptor détecte le paramètre uniquement dans les mappages qui l'ont activé. Par conséquent, si plusieurs mappages de gestionnaires sont configurés dans le contexte d'application web, nous devons enregistrer l'intercepteur dans chacun d'eux afin que les utilisateurs puissent modifier les paramètres régionaux dans toutes les URL.

Le paramètre language permet ensuite à l'utilisateur de changer ces paramètres régionaux dans n'importe quelle URL. Par exemple, la première URL suivante choisit la langue française, tandis que la seconde opte pour l'anglais aux États-Unis :

```
http://localhost:8080/court/welcome.htm?language=fr
http://localhost:8080/court/welcome.htm?language=en_US
```

Nous pouvons ensuite afficher les paramètres régionaux de l'objet de réponse HTTP dans welcome.jsp de manière à vérifier la configuration de l'intercepteur :

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

<html>
<head>
<title>Bienvenue</title>
</head>
```

```
<body>
...
<br />
Paramètres régionaux : ${pageContext.response.locale}
</body>
</html>
```

10.5 Externaliser les messages dépendant de la localisation

Problème

Lors du développement d'une application web internationalisée, nous devons afficher les pages web en tenant compte des paramètres régionaux de l'utilisateur. Mais nous ne voulons pas créer différentes versions des mêmes pages.

Solution

Pour ne pas avoir à créer différentes versions d'une même page pour les différents paramètres régionaux, nous pouvons écrire une page web indépendante de la localisation en externalisant les messages textuels qui en dépendent. Spring retrouve des messages textuels localisés en utilisant une source de messages qui implémente l'interface `MessageSource`. Dans les fichiers JSP, nous utilisons la balise `<spring:message>`, fournie par la bibliothèque de balises de Spring, afin d'obtenir le message qui correspond à un code.

Explications

Pour définir une source de messages, il suffit d'enregistrer un bean de type `MessageSource` dans le contexte d'application web. Pour que `DispatcherServlet` détecte automatiquement cette source de messages, le bean doit se nommer `messageSource`. Nous ne pouvons enregistrer qu'une seule source de messages par `DispatcherServlet`.

L'implémentation `ResourceBundleMessageSource` obtient les messages à partir de différents bundles de ressources pour les différentes localisations. Nous pouvons, par exemple, l'utiliser dans `court-servlet.xml` de manière à charger les bundles de ressources dont le nom de base est `messages`.

```
<bean id="messageSource"
      class="org.springframework.context.support.ResourceBundleMessageSource">
    <property name="basename" value="messages" />
</bean>
```

Ensuite, nous créons deux bundles de ressources, les fichiers `messages.properties` et `messages_en_US.properties`, pour enregistrer les messages correspondant à la localisation par défaut et à l'anglais des États-Unis. Ces bundles de ressources doivent être placés à la racine du chemin d'accès aux classes.

```
welcome.title=Bienvenue
welcome.message=Bienvenue sur le système de réservation des terrains

---
welcome.title=Welcome
welcome.message=Welcome to Court Reservation System
```

Ensuite, dans les fichiers JSP, comme `welcome.jsp`, nous utilisons la balise `<spring:message>` pour obtenir le message correspondant au code indiqué et aux paramètres régionaux de l'utilisateur. Puisque cette balise est définie dans la bibliothèque de balises de Spring, il ne faut pas oublier de déclarer celle-ci au début du fichier JSP.

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

<html>
<head>
<title><spring:message code="welcome.title" text="Bienvenue" /></title>
</head>

<body>
<h2><spring:message code="welcome.message"
    text="Bienvenue sur le système de réservation des terrains" /></h2>
...
</body>
</html>
```

Dans `<spring:message>`, nous indiquons le texte qui est affiché par défaut lorsque aucun message ne correspond au code indiqué.

10.6 Déterminer les vues d'après leur nom

Problème

Lorsqu'un gestionnaire a terminé le traitement d'une requête, il peut retourner un objet de vue ou le nom logique de celui-ci. S'il retourne un nom de vue, `DispatcherServlet` doit créer l'objet de vue à partir de ce nom et l'afficher à l'utilisateur. Nous souhaitons définir la stratégie employée par `DispatcherServlet` pour convertir des noms en vues.

Solution

Dans une application Spring MVC, les vues sont déterminées par un ou plusieurs beans de résolution de vue déclarés dans le contexte d'application web. Ces beans doivent implémenter l'interface `ViewResolver` pour que `DispatcherServlet` les détecte automatiquement. Spring MVC propose plusieurs implémentations de `ViewResolver` qui mettent en œuvre différentes stratégies de résolution.

Explications

Déterminer des vues à partir d'URL

Pour déterminer des vues, la stratégie de base consiste à les associer directement à des URL. Le résolveur de vue `InternalResourceViewResolver` associe chaque nom de vue à une URL en lui ajoutant un préfixe et un suffixe. Pour enregistrer `InternalResourceViewResolver`, nous déclarons un bean de ce type dans le contexte d'application web.

```
<bean 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>
```

Voici, par exemple, comment `InternalResourceViewResolver` convertit les noms de vues `welcome` et `reservationQuery` :

```
welcome → /WEB-INF/jsp/welcome.jsp
reservationQuery → /WEB-INF/jsp/reservationQuery.jsp
```

Le type des vues résolues peut être indiqué dans la propriété `viewClass`. Par défaut, si la bibliothèque JSTL (c'est-à-dire `jstl.jar`) est présente dans le chemin d'accès aux classes, `InternalResourceViewResolver` convertit les noms de vues en objets de vues de type `JstlView`. Par conséquent, nous pouvons omettre la propriété `viewClass` lorsque nos vues sont des templates JSP avec des balises JSTL.

`InternalResourceViewResolver` a l'avantage d'être simple, mais il ne peut résoudre que les vues qui correspondent à des ressources internes (par exemple un fichier JSP ou une servlet interne) transmises par `RequestDispatcher` de l'API des servlets. Pour les autres types de vues reconnus par Spring MVC, d'autres stratégies de résolutions doivent être mises en œuvre.

Déterminer des vues à partir d'un fichier de configuration XML

Pour déterminer des vues, une autre stratégie consiste à les déclarer sous forme de beans Spring et à utiliser les noms de beans pour la résolution. Nous pouvons déclarer les beans de vues dans le même fichier de configuration que le contexte d'application web, mais il est préférable de les isoler dans un fichier de configuration distinct. Par défaut, `XmlViewResolver` charge les beans de vues à partir de `/WEB-INF/views.xml`, mais la propriété `location` permet de modifier ce chemin.

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

Dans le fichier de configuration `court-views.xml`, chaque vue est déclarée sous forme d'un bean Spring normal en précisant le nom de classe et les propriétés. Nous pouvons ainsi déclarer tout type de vue (par exemple `RedirectView` et même nos propres types de vues).

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="welcome"
          class="org.springframework.web.servlet.view.JstlView">
        <property name="url" value="/WEB-INF/jsp/welcome.jsp" />
    </bean>

    <bean id="reservationQuery"
          class="org.springframework.web.servlet.view.JstlView">
        <property name="url" value="/WEB-INF/jsp/reservationQuery.jsp" />
    </bean>

    <bean id="welcomeRedirect"
          class="org.springframework.web.servlet.view.RedirectView">
        <property name="url" value="welcome.htm" />
    </bean>
</beans>
```

Déterminer des vues à partir d'un bundle de ressources

À la place du fichier de configuration XML, nous pouvons déclarer les beans de vues dans un bundle de ressources. `ResourceBundleViewResolver` utilise le bundle de ressources situé à la racine du chemin d'accès aux classes. Il peut également profiter de la gestion des paramètres régionaux pour charger les beans de vues à partir de différents bundles de ressources.

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

Puisque le nom de base de `ResourceBundleViewResolver` est `views`, le bundle de ressources par défaut se nomme `views.properties`. Dans ce fichier, nous déclarons des beans de vues sous forme de propriétés. Ce type de déclaration équivaut à la déclaration XML, mais il est moins puissant.

```
welcome.(class)=org.springframework.web.servlet.view.JstlView
welcome.url=/WEB-INF/jsp/welcome.jsp

reservationQuery.(class)=org.springframework.web.servlet.view.JstlView
reservationQuery.url=/WEB-INF/jsp/reservationQuery.jsp

welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView
welcomeRedirect.url=welcome.htm
```

Déterminer des vues avec plusieurs résolveurs

Lorsque l'application web utilise un grand nombre de vues, le choix d'une seule stratégie de résolution est bien souvent insuffisant. `InternalResourceViewResolver` peut généralement résoudre la plupart des vues JSP internes, mais la résolution des autres types de vues doit se faire avec `ResourceBundleViewResolver`. Dans ce cas, nous devons combiner les deux stratégies.

```
<beans ...>
    ...
    <bean class="org.springframework.web.servlet.view.<span style="color: #0070C0;">➥
          ResourceBundleViewResolver">
        <property name="basename" value="views" />
        <property name="order" value="0" />
    </bean>

    <bean class="org.springframework.web.servlet.view.<span style="color: #0070C0;">➥
          InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/jsp/" />
        <property name="suffix" value=".jsp" />
        <property name="order" value="1" />
    </bean>
</beans>
```

Lorsque plusieurs stratégies sont retenues, il est important d'en indiquer la priorité. Pour cela, nous fixons l'ordre des beans de résolution. Plus la valeur est faible, plus la priorité est élevée. Nous pouvons donner la priorité la plus faible à `InternalResourceViewResolver` car il est toujours en mesure de déterminer une vue, qu'elle existe ou non. Les autres résolveurs n'ont aucune chance d'intervenir s'ils ont une priorité plus faible.

Le bundle de ressources `views.properties` doit à présent contenir uniquement les vues qui ne peuvent pas être déterminées par `InternalResourceViewResolver` (par exemple les vues de redirection).

```
welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView
welcomeRedirect.url=welcome.htm
```

Le préfixe de redirection

Si `InternalResourceViewResolver` est configuré dans le contexte d'application web, il peut déterminer des vues de redirection lorsque le nom de vue est préfixé par `redirect`. La suite du nom est interprétée comme l'URL de redirection. Par exemple, le nom `redirect:welcome.htm` déclenche une redirection vers l'URL relative `welcome.htm`. Nous pouvons également utiliser une URL absolue dans le nom de vue.

10.7 Associer des exceptions aux vues

Problème

Lorsqu'une exception inconnue survient, le serveur d'applications affiche généralement la trace de la pile des exceptions à l'utilisateur. Ces informations n'ont aucun intérêt pour les utilisateurs, qui verront alors une application peu conviviale. Par ailleurs, elles représentent un risque potentiel pour la sécurité car elles détaillent la hiérarchie des appels de méthodes internes.

Solution

Dans une application Spring MVC, nous pouvons enregistrer dans le contexte d'application web un ou plusieurs beans de résolution des exceptions non interceptées. Ils doivent implémenter l'interface `HandlerExceptionResolver` pour être détectés automatiquement par `DispatcherServlet`. Spring MVC fournit un résolveur d'exception simple qui permet d'associer chaque catégorie d'exceptions à une vue.

Explications

Supposons que notre service de réservation lance l'exception suivante lorsqu'une réservation n'est pas disponible :

```
package com.apress.springrecipes.court.service;
...
public class ReservationNotAvailableException extends RuntimeException {

    private String courtName;
    private Date date;
    private int hour;

    // Constructeur et accesseurs.
    ...
}
```

Pour prendre en charge les exceptions non interceptées, nous écrivons notre propre résolveur d'exceptions en implementant l'interface `HandlerExceptionResolver`. Normalement, nous associons les différentes catégories d'exceptions à différentes pages d'erreur. Spring MVC fournit `SimpleMappingExceptionResolver`, qui nous permet de configurer les correspondances d'exceptions dans le contexte d'application web. Nous enregistrons, par exemple, le résolveur d'exceptions suivant dans `court-servlet.xml` :

```
<bean class="org.springframework.web.servlet.handler.<-
      SimpleMappingExceptionResolver">
    <property name="exceptionMappings">
        <props>
            <prop key="com.apress.springrecipes.court.service.<-
                  ReservationNotAvailableException">
                reservationNotAvailable
            </prop>
        </props>
    </property>
</bean>
```

```
<prop key="java.lang.Exception">error</prop>
</props>
</property>
</bean>
```

Nous associons le nom de vue logique `reservationNotAvailable` à `ReservationNotAvailableException`. Si `InternalResourceViewResolver` est configuré dans le contexte d'application web, la page `reservationNotAvailable.jsp` suivante est affichée lorsqu'une réservation n'est pas disponible :

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

<html>
<head>
<title>Réservation non disponible</title>
</head>

<body>
Le terrain ${exception.courtName} ne peut pas être réservé pour
le <fmt:formatDate value="${exception.date}" pattern="yyyy-MM-dd" />
à ${exception.hour}:00.
</body>
</html>
```

Dans une page d'erreur, nous pouvons accéder à l'instance de l'exception au travers de la variable `${exception}`. Cela nous permet de présenter à l'utilisateur quelques détails sur cette exception.

Il est conseillé de définir une page d'erreur par défaut pour toutes les exceptions inconnues. Elle doit être associée à la clé `java.lang.Exception` en dernière position afin qu'elle soit affichée si aucune autre entrée adéquate n'a été trouvée. Voici notre page `error.jsp` :

```
<html>
<head>
<title>Erreur</title>
</head>

<body>
Une erreur est survenue. Pour de plus amples informations, veuillez
contacter notre administrateur.
</body>
</html>
```

10.8 Construire des objets *ModelAndView*

Problème

Lorsqu'un contrôleur a terminé le traitement d'une requête, il retourne habituellement à DispatcherServlet un objet ModelAndView qui contient un nom de vue ou un objet de vue, ainsi que des attributs du modèle. Nous devons donc construire des objets ModelAndView dans les contrôleurs.

Solution

La classe ModelAndView déclare plusieurs constructeurs surchargés et quelques méthodes pratiques pour faciliter la construction d'un objet ModelAndView. Ces constructeurs et méthodes gèrent un nom de vue et un objet de manière comparable.

Explications

Lorsque nous devons retourner un seul attribut du modèle, nous pouvons construire un objet ModelAndView en passant cet attribut au constructeur.

```
package com.apress.springrecipes.court.web;
...
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class WelcomeController extends AbstractController {

    public ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        Date today = new Date();
        return new ModelAndView("welcome", "today", today);
    }
}
```

Si nous devons retourner plusieurs attributs, nous créons un objet ModelAndView en lui passant une table d'association.

```
package com.apress.springrecipes.court.web;
...
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class ReservationQueryController extends AbstractController {

    ...
    public ModelAndView handleRequestInternal(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        ...
        Map<String, Object> model = new HashMap<String, Object>();
        if (courtName != null) {
            model.put("courtName", courtName);
        }
    }
}
```

```

        model.put("reservations", reservationService.query(courtName));
    }
    return new ModelAndView("reservationQuery", model);
}
}

```

Spring fournit également ModelMap, une implémentation de java.util.Map capable de générer automatiquement les noms des attributs du modèle d'après leur type concret.

```

package com.apress.springrecipes.court.web;
...
import org.springframework.ui.ModelMap;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class ReservationQueryController extends AbstractController {
    ...
    public ModelAndView handleRequestInternal(HttpServletRequest request,
                                              HttpServletResponse response) throws Exception {
        ...
        ModelMap model = new ModelMap();
        if (courtName != null) {
            model.addAttribute("courtName", courtName);
            model.addAttribute("reservations",
                               reservationService.query(courtName));
        }
        return new ModelAndView("reservationQuery", model);
    }
}

```

Puisque les attributs du modèle sont de type String et List<Reservation>, ModelMap génère les noms string et reservationList. Si ces noms ne conviennent pas, nous pouvons les préciser explicitement.

Après avoir créé un objet ModelAndView, nous pouvons continuer à lui ajouter des attributs du modèle en invoquant la méthode addObject(). Elle retourne l'objet ModelAndView lui-même, ce qui nous permet de construire un objet ModelAndView en une seule instruction. Nous pouvons également omettre le nom de l'attribut lors de l'appel à addObject(). Dans ce cas, la méthode génère le nom comme le ferait ModelMap.

```

package com.apress.springrecipes.court.web;
...
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class ReservationQueryController extends AbstractController {
    ...
    public ModelAndView handleRequestInternal(HttpServletRequest request,
                                              HttpServletResponse response) throws Exception {
        ...
        List<Reservation> reservations = null;
        if (courtName != null) {
            reservations = reservationService.query(courtName);
        }
    }
}

```

```
        return new ModelAndView("reservationQuery", "courtName", courtName)
               . addObject("reservations", reservations);
    }
}
```

En réalité, nous ne sommes pas obligés de retourner un modèle et une vue. Dans certains cas, nous voulons uniquement retourner la vue, sans aucun attribut dans le modèle. Ou bien nous voulons retourner le modèle sans la vue et laisser Spring MVC déterminer celle-ci en fonction de l'URL de la requête. Parfois, nous voulons même retourner null si le contrôleur prend directement en charge l'objet `HttpServletResponse`, notamment lorsqu'il renvoie à l'utilisateur un fichier sous forme de flux continu.

10.9 Créer un contrôleur avec une vue paramétrée

Problème

Nous ne souhaitons pas figer le nom de vue dans la classe d'un contrôleur. À la place, nous voulons qu'il soit paramétré pour que nous puissions le définir dans le fichier de configuration des beans.

Solution

`ParameterizableViewController` est une sous-classe de `AbstractController` qui définit une propriété `viewName` avec ses accesseurs et mutateurs. Nous pouvons employer directement cette classe pour un contrôleur qui affiche simplement une vue à l'utilisateur sans inclure une logique de traitement. Mais nous pouvons également l'étendre pour hériter de la propriété `viewName`.

Explications

Par exemple, supposons que nous ayons un contrôleur très simple dont le seul rôle est de produire une vue des informations concernant l'application. Nous déclarons un contrôleur de type `ParameterizableViewController` et affectons la valeur `about` à la propriété `viewName`.

```
<bean id="aboutController"
      class="org.springframework.web.servlet.mvc.ParameterizableViewController">
  <property name="viewName" value="about" />
</bean>
```

Nous créons également la page `about.jsp` dans le répertoire `/WEB-INF/jsp/`.

```
<html>
<head>
<title>À propos</title>
</head>
```

```
<body>
<h2>Système de réservation des terrains</h2>
<table>
<tr>
<td>Version :</td>
<td>1.0</td>
</tr>
</table>
</body>
</html>
```

Si nous souhaitons ajouter une logique de traitement au contrôleur tout en conservant la vue paramétrée, sa classe doit étendre `ParameterizableViewController`. Le contrôleur `AboutController` suivant accepte une propriété `email` et l'inclut dans le modèle.

```
package com.apress.springrecipes.court.web;
...
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.ParameterizableViewController;

public class AboutController extends ParameterizableViewController {

    private String email;

    public void setEmail(String email) {
        this.email = email;
    }

    protected ModelAndView handleRequestInternal(HttpServletRequest request,
                                                HttpServletResponse response) throws Exception {
        return new ModelAndView(getViewName(), "email", email);
    }
}
```

Puisque `AboutController` dérive de `ParameterizableViewController`, il possède également la propriété `viewName`, qui peut être injectée.

```
<bean id="aboutController"
      class="com.apress.springrecipes.court.web.AboutController">
    <property name="viewName" value="about" />
    <property name="email" value="reservation@court.com" />
</bean>
```

Dans `about.jsp`, nous affichons l'attribut `email` extrait du modèle.

```
<html>
<head>
<title>À propos</title>
</head>

<body>
<h2>Système de réservation des terrains</h2>
<table>
<tr>
<td>Email :</td>
```

```
<td><a href="mailto:${email}">${email}</a></td>
</tr>
</table>
</body>
</html>
```

Puisque `ControllerClassNameHandlerMapping` est configuré dans notre contexte d'application web, nous pouvons accéder à ce contrôleur avec l'URL suivante :

`http://localhost:8080/court/about.htm`

10.10 Gérer des formulaires avec des contrôleurs

Problème

Dans une application web, il est souvent nécessaire de traiter des formulaires. Un contrôleur de formulaire doit afficher un formulaire à l'utilisateur et prendre en charge sa soumission. Cette tâche peut se révéler très complexe et très variée. Si nous construisons un contrôleur de formulaire à partir de zéro, nous avons de nombreux détails à prendre en considération.

Solution

La classe `SimpleFormController` fournie par Spring MVC définit le flux de base du traitement d'un formulaire. Elle prend en charge le concept d'objet de commande et permet de lier les champs d'un formulaire aux propriétés éponymes d'un objet de commande. En dérivant de la classe `SimpleFormController`, notre contrôleur hérite des possibilités de gestion des formulaires.

Lorsque `SimpleFormController` doit présenter un formulaire suite à une requête HTTP GET, il affiche la vue de formulaire à l'utilisateur. Ensuite, lorsque le formulaire est soumis par une requête HTTP POST, `SimpleFormController` gère cette opération en liant les champs du formulaire à un objet de commande et en invoquant la méthode `onSubmit()`. Si le traitement du formulaire se passe correctement, la vue prévue en cas de réussite est affichée à l'utilisateur. Sinon la vue de formulaire, avec les erreurs, est à nouveau présentée.

Pour tenir compte des différentes contraintes, `SimpleFormController` nous permet d'adapter le flux de traitement du formulaire en redéfinissant plusieurs méthodes du cycle de vie.

Explications

Supposons que l'utilisateur effectue sa réservation en remplissant un formulaire. Nous commençons par définir une méthode `make()` dans l'interface `ReservationService`.

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public void make(Reservation reservation)
        throws ReservationNotAvailableException;
}
```

Nous implémentons ensuite cette méthode en ajoutant un élément `Reservation` à la liste qui contient les réservations. Si la nouvelle réservation entre en conflit avec une autre réservation, nous lançons une exception `ReservationNotAvailableException`.

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public void make(Reservation reservation)
        throws ReservationNotAvailableException {
        for (Reservation made : reservations) {
            if (made.getCourtName().equals(reservation.getCourtName())
                && made.getDate().equals(reservation.getDate())
                && made.getHour() == reservation.getHour()) {
                throw new ReservationNotAvailableException(
                    reservation.getCourtName(), reservation.getDate(),
                    reservation.getHour());
            }
        }
        reservations.add(reservation);
    }
}
```

Créer un contrôleur de formulaire

Créons à présent le contrôleur qui prend en charge le formulaire de réservation des terrains. En dérivant de la classe `SimpleFormController`, nous indiquons simplement la classe de commande (`Reservation` dans ce cas) associée au contrôleur, puis les champs du formulaire sont liés aux propriétés éponymes de l'objet de commande. Nous précisons également le nom de l'objet de commande (`reservation` dans ce cas) employé par la vue pour y accéder, mais cela reste facultatif et le nom par défaut est `command`.

```
package com.apress.springrecipes.court.web;
...
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.web.bind.ServletRequestDataBinder;
import org.springframework.web.servlet.mvc.SimpleFormController;

public class ReservationFormController extends SimpleFormController {

    private ReservationService reservationService;

    public ReservationFormController() {
        setCommandClass(Reservation.class);
        setCommandName("reservation");
    }
}
```

```
public void setReservationService(ReservationService reservationService) {  
    this.reservationService = reservationService;  
}  
  
protected void initBinder(HttpServletRequest request,  
    ServletRequestDataBinder binder) throws Exception {  
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");  
    dateFormat.setLenient(false);  
    binder.registerCustomEditor(Date.class, new CustomDateEditor(  
        dateFormat, true));  
}  
  
protected void doSubmitAction(Object command) throws Exception {  
    Reservation reservation = (Reservation) command;  
    reservationService.make(reservation);  
}  
}
```

Pour lier les champs du formulaire à un objet de commande, le contrôleur doit réaliser des conversions de type sur les valeurs des champs car elles sont envoyées sous forme de chaînes de caractères. Les conversions de type sont effectuées par des éditeurs de propriétés enregistrés dans le contrôleur. Spring enregistre lui-même plusieurs éditeurs de propriétés pour les types de données répandus, comme les nombres et les booléens. Nous devons enregistrer nos propres éditeurs pour les autres types de données, comme `java.util.Date`. Pour cela, nous invoquons l'argument de type `ServletRequestDataBinder` dans la méthode `initBinder()`.

Si la liaison des champs du formulaire se passe mal, `SimpleFormController` présente à nouveau, sans autre action, la vue de formulaire avec les erreurs. Sinon il invoque la méthode `onSubmit()` pour traiter la soumission du formulaire. Dans un argument de cette méthode, nous retrouvons l'objet de commande créé et lié. Pour notre contrôleur, il s'agit d'un objet `Reservation` puisque la classe de commande que nous avons indiquée est `Reservation`. Il existe trois variantes de la méthode `onSubmit()`. Nous devons redéfinir la plus simple, qui donne accès aux arguments dont nous avons besoin.

```
protected ModelAndView onSubmit(Object command) throws Exception;  
  
protected ModelAndView onSubmit(Object command, BindException errors)  
    throws Exception;  
  
protected ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse  
    response, Object command, BindException errors) throws Exception;
```

Si nous redéfinissons une méthode `onSubmit()`, nous devons retourner un objet `ModelAndView`. Si nous devons simplement effectuer une opération sur l'objet de commande et retourner la vue correspondant à une réussite une fois cette opération terminée, nous pouvons à la place redéfinir la méthode `doSubmitAction()`, dont le type de retour est `void` et qui, par défaut, présente cette vue. Dans la méthode `doSubmitAction()` précédente, nous passons l'objet de commande à `ReservationService` pour effectuer la réservation.

Dans la déclaration de ce contrôleur, nous ajoutons la référence au bean `reservationService` de la couche de service pour qu'il se charge des réservations. Par ailleurs, nous définissons la vue de formulaire et la vue de réussite associées à ce contrôleur de formulaire.

```
<bean id="reservationFormController"
    class="com.apress.springrecipes.court.web.ReservationFormController">
    <property name="reservationService" ref="reservationService" />
    <property name="formView" value="reservationForm" />
    <property name="successView" value="reservationSuccess" />
</bean>
```

Commençons par créer la vue de formulaire `reservationForm.jsp`. Elle est constituée d'un formulaire HTML contenant plusieurs champs. En cas d'erreur de liaison ou de validation, `SimpleFormController` affiche à nouveau cette vue à l'utilisateur et inclut les erreurs dans le modèle pour que la vue puisse y accéder. Nous affichons le message d'erreur et les valeurs saisies initialement par l'utilisateur.

Dans Spring version 1.x, nous devons employer la balise `<spring:bind>` pour lier chaque champ du formulaire à une propriété de l'objet de commande, puis accéder au message d'erreur et aux valeurs d'origine à l'aide des expressions `${status.errorMessage}` et `${status.value}`. Dans Spring 2.x, nous pouvons utiliser les puissantes balises de formulaire définies par la bibliothèque `form` de Spring.

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

<html>
<head>
<title>Formulaire de réservation</title>
<style>
.error {
    color: #ff0000;
    font-weight: bold; }
</style>
</head>

<body>
<form:form method="POST" commandName="reservation">
<form:errors path="*" cssClass="error" />
<table>
<tr>
    <td>Nom du terrain</td>
    <td><form:input path="courtName" /></td>
    <td><form:errors path="courtName" cssClass="error" /></td>
</tr>
<tr>
    <td>Date</td>
    <td><form:input path="date" /></td>
    <td><form:errors path="date" cssClass="error" /></td>
</tr>
<tr>
    <td>Horaire</td>
```

```
<td><form:input path="hour" /></td>
<td><form:errors path="hour" cssClass="error" /></td>
</tr>
<tr>
    <td colspan="3"><input type="submit" /></td>
</tr>
</table>
</form:form>
</body>
</html>
```

L'attribut path des balises `<form:input>` et `<form:select>` permet d'établir une liaison avec un chemin de propriété de l'objet de commande. Ces balises présentent à l'utilisateur la valeur d'origine du champ, qui est la valeur de la propriété liée ou la valeur rejetée en raison d'une erreur de liaison. Elles doivent être utilisées à l'intérieur de la balise `<form:form>`, qui définit un formulaire et lie l'objet de commande à partir de son nom. Si nous ne précisons pas le nom de l'objet de commande, le nom par défaut `command` est utilisé. La balise `<form:errors>` affiche tous les messages d'erreur associés au chemin de propriété indiqué. Si le chemin contient un astérisque, toutes les erreurs sont présentées.

Nous créons ensuite la vue de réussite, `reservationSuccess.jsp`, qui signale à l'utilisateur le bon déroulement de la réservation.

```
<html>
<head>
<title>Réservation effectuée</title>
</head>

<body>
Votre réservation a été prise en compte.
</body>
</html>
```

Au cours du processus de liaison des champs du formulaire, des erreurs peuvent survenir en raison de valeurs invalides. Par exemple, si la date n'est pas dans un format reconnu par `CustomDateEditor` ou si l'horaire contient un caractère alphabétique, le contrôleur de formulaire précédent est incapable de convertir ces champs. Il génère une liste de codes correspondant à chaque erreur. Pour la saisie d'une valeur invalide dans le champ de date, les codes d'erreur suivants sont générés :

```
typeMismatch.command.date
typeMismatch.date
typeMismatch.java.util.Date
typeMismatch
```

Si nous avons défini `ResourceBundleMessageSource`, nous pouvons inclure les messages d'erreur suivants dans le bundle de ressources correspondant aux paramètres régionaux adéquats (par exemple `messages.properties` pour la localisation par défaut) :

```
typeMismatch.date=Format de date invalide
typeMismatch.hour=Format d'horaire invalide
```

Puisque `ControllerClassNameHandlerMapping` est configuré dans notre contexte d'application web, nous pouvons accéder à ce contrôleur via l'URL suivante :

`http://localhost:8080/court/reservationForm.htm`

En saisissant cette URL dans le navigateur, il envoie une requête HTTP GET à notre application web. Le contrôleur de formulaire affiche la vue de formulaire qui correspond à cette requête. Une fois que nous avons rempli les différents champs, nous soumettons le formulaire avec une requête HTTP POST. Le contrôleur de ce formulaire prend alors en charge cette soumission. Si le traitement se passe correctement, la vue de réussite est affichée. Sinon la vue de formulaire est à nouveau présentée, avec les erreurs.

Appliquer le design pattern Post-Redirect-Get

Si nous actualisons la page web affichée par la vue de réussite, le formulaire est soumis une nouvelle fois. Pour éviter ce problème de *soumission redondante d'un formulaire*, nous pouvons appliquer le design pattern *Post-Redirect-Get*. Il recommande une redirection vers une autre URL après le traitement réussi du formulaire, au lieu de retourner directement à une page HTML.

Tout d'abord, nous utilisons `ParameterizableViewController` pour définir un contrôleur qui affiche uniquement la vue `reservationSuccess`, associée à `reservationSuccess.jsp`.

```
<bean id="reservationSuccessController"
      class="org.springframework.web.servlet.mvc.ParameterizableViewController">
    <property name="viewName" value="reservationSuccess" />
</bean>
```

Ensuite, puisque `ControllerClassNameHandlerMapping` ne génère pas de correspondance pour un contrôleur Spring MVC intégré, nous définissons un mappage explicite pour ce contrôleur dans `SimpleUrlHandlerMapping`.

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
  ...
  <property name="mappings">
    <props>
      ...
      <prop key="/reservationSuccess.htm">
        reservationSuccessController
      </prop>
    </props>
  </property>
</bean>
```

`ResourceBundleViewResolver` étant configuré dans notre contexte d'application web, nous pouvons définir la vue de redirection suivante dans le fichier `views.properties` situé à la racine du chemin d'accès aux classes :

```
reservationSuccessRedirect.(class)=>
    org.springframework.web.servlet.view.RedirectView
reservationSuccessRedirect.url=reservationSuccess.htm
```

Enfin, nous utilisons cette vue de redirection comme vue de réussite pour `ReservationFormController`. À présent, lorsque le traitement du formulaire soumis se passe sans encombre, l'utilisateur est redirigé vers une autre URL. S'il actualise cette page, le problème de soumission de formulaire redondante n'apparaît plus.

```
<bean id="reservationFormController"
      class="com.apress.springrecipes.court.web.ReservationFormController">
    <property name="reservationService" ref="reservationService" />
    <property name="formView" value="reservationForm" />
    <property name="successView" value="reservationSuccessRedirect" />
</bean>
```

Initialiser l'objet de commande

Lorsque nous indiquons une classe de commande à `SimpleFormController`, elle est instanciée et s'occupe de la liaison des champs du formulaire. Cependant, dans certains cas, nous préférerons initialiser nous-mêmes l'objet de commande. Par exemple, supposons que nous voulions lier le nom et le numéro de téléphone du joueur à la propriété `player` d'un objet `Reservation`.

```
<html>
<head>
<title>Formulaire de réservation</title>
</head>

<body>
<form method="POST">
<table>
  ...
  <tr>
    <td>Nom du joueur</td>
    <td><form:input path="player.name" /></td>
    <td><form:errors path="player.name" cssClass="error" /></td>
  </tr>
  <tr>
    <td>N° de téléphone du joueur</td>
    <td><form:input path="player.phone" /></td>
    <td><form:errors path="player.phone" cssClass="error" /></td>
  </tr>
  <tr>
    <td colspan="3"><input type="submit" /></td>
  </tr>
</table>
</form>
</body>
</html>
```

Lorsque la classe de commande `Reservation` est instanciée, la propriété `player` vaut null. Par conséquent, l'affichage de ce formulaire produit une exception.

Afin d'éviter ce problème, nous devons initialiser nous-mêmes l'objet de commande. Pour cela, nous redéfinissons la méthode `formBackingObject()` de `SimpleFormController`. L'implémentation par défaut de cette méthode instancie simplement la classe de commande. Lorsque nous la redéfinissons, il est inutile de préciser la classe de commande car `SimpleFormController` ne se charge plus de son instantiation.

```
package com.apress.springrecipes.court.web;
...
public class ReservationFormController extends SimpleFormController {
    ...
    public ReservationFormController() {
        // Inutile de préciser la classe de commande.
        // setCommandClass(Reservation.class);
        setCommandName("reservation");
    }

    protected Object formBackingObject(HttpServletRequest request)
        throws Exception {
        Reservation reservation = new Reservation();
        reservation.setPlayer(new Player());
        return reservation;
    }
}
```

Cette méthode est également souvent employée pour accéder à certains paramètres de requête de manière à initialiser l'objet de commande (par exemple pour retrouver une entité à partir de son identifiant depuis un framework ORM). Nous pouvons par exemple lire la valeur du paramètre de requête `username` et initialiser le joueur avec le nom obtenu.

```
package com.apress.springrecipes.court.web;
...
import org.springframework.web.bind.ServletRequestUtils;

public class ReservationFormController extends SimpleFormController {
    ...
    protected Object formBackingObject(HttpServletRequest request)
        throws Exception {
        Reservation reservation = new Reservation();
        String username =
            ServletRequestUtils.getStringParameter(request, "username");
        reservation.setPlayer(new Player(username, null));
        return reservation;
    }
}
```

À présent, lorsque le formulaire est affiché, la valeur du paramètre `username` est automatiquement placée dans le champ du nom du joueur. L'URL suivante permet de vérifier ce comportement :

<http://localhost:8080/court/reservationForm.htm?username=Roger>

L'objet de commande possède deux autres propriétés que nous pouvons configurer. Tout d'abord, `bindOnNewForm` précise si les paramètres de requête doivent être liés à l'objet de commande lors de la création d'un nouveau formulaire. Ce processus s'apparente à la liaison du paramètre `username` au champ du nom du joueur. La seule différence est que les paramètres de requête sont liés aux propriétés éponymes. Nous pouvons activer cette propriété dans le constructeur.

```
package com.apress.springrecipes.court.web;
...
public class ReservationFormController extends SimpleFormController {
    ...
    public ReservationFormController() {
        ...
        setBindOnNewForm(true);
    }
}
```

Dorénavant, lorsque le formulaire est affiché, les paramètres de requête sont liés aux propriétés de même nom. Nous testons ce fonctionnement avec l'URL suivante :

```
http://localhost:8080/court/reservationForm.htm?date=2008-01-14
```

La seconde propriété, `sessionForm`, précise si l'objet de commande doit être enregistré dans la session. Par défaut, elle vaut `false` pour qu'un nouvel objet de commande soit créé à chaque requête, même lors du réaffichage du formulaire suite à des erreurs de liaison. Lorsque cette propriété est fixée à `true`, l'objet de commande est enregistré dans la session et réutilisé ensuite, jusqu'à ce que le traitement du formulaire réussisse. À ce moment-là, l'objet de commande est effacé de la session. Ce fonctionnement est habituellement mis en œuvre lorsque l'objet de commande est un objet persistant qui doit rester identique entre différentes requêtes de manière à effectuer un suivi.

```
package com.apress.springrecipes.court.web;
...
public class ReservationFormController extends SimpleFormController {
    ...
    public ReservationFormController() {
        ...
        setSessionForm(true);
    }
}
```

Fournir des données de référence au formulaire

Lorsqu'un contrôleur de formulaire doit afficher la vue de formulaire, il peut avoir à passer certaines données de référence au formulaire, par exemple les éléments d'une sélection HTML. Supposons à présent que nous voulions que l'utilisateur sélectionne le type de sport au moment de la réservation d'un terrain. Nous définissons la méthode `getAllSportTypes()` dans l'interface `ReservationService` pour obtenir la liste de tous les sports disponibles.

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public List<SportType> getAllSportTypes();
}
```

Nous implémentons cette méthode en retournant une liste figée.

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public static final SportType TENNIS = new SportType(1, "Tennis");
    public static final SportType SOCCER = new SportType(2, "Football");

    public List<SportType> getAllSportTypes() {
        return Arrays.asList(new SportType[] { TENNIS, SOCCER });
    }
}
```

Lorsque `ReservationFormController` doit afficher le formulaire de réservation, nous devons inclure tous les sports disponibles dans le modèle afin que ce formulaire puisse les présenter dans une sélection HTML. Pour cela, nous redéfinissons la méthode `referenceData()` de `SimpleFormController`. Nous plaçons les données de référence dans une table d'association retournée par cette méthode. Elle est ajoutée au modèle et passée automatiquement à la vue de formulaire. L'implémentation par défaut de la méthode `referenceData()` retourne null.

```
package com.apress.springrecipes.court.web;
...
public class ReservationFormController extends SimpleFormController {
    ...
    protected Map referenceData(HttpServletRequest request) throws Exception {
        Map referenceData = new HashMap();
        List<SportType> sportTypes = reservationService.getAllSportTypes();
        referenceData.put("sportTypes", sportTypes);
        return referenceData;
    }
}
```

Dans la vue de formulaire, nous utilisons la balise `<form:select>` de Spring pour définir la sélection HTML. Il suffit de préciser la liste de l'élément, la valeur de l'élément et le libellé de l'élément pour que la balise génère automatiquement les options de la sélection.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
<title>Formulaire de réservation</title>
</head>
```

```
<body>
<form method="POST">
<table>
...
<tr>
    <td>Type de sport</td>
    <td>
        <form:select path="sportType" items="${sportTypes}"
                      itemValue="id" itemLabel="name" />
    </td>
    <td><form:errors path="sportType" cssClass="error" /></td>
</tr>
<tr>
    <td colspan="3"><input type="submit" /></td>
</tr>
</table>
</form>
</body>
</html>
```

Lier des propriétés de types personnalisés

Lorsqu'un formulaire est soumis, `SimpleFormController` prend en charge la liaison des champs du formulaire aux propriétés éponymes d'un objet de commande. Cependant, `SimpleFormController` est incapable de convertir les propriétés qui ne sont pas d'un type prédéfini, à moins que nous ne précisions les éditeurs de propriétés adéquats. Par exemple, le champ de sélection du type de sport fournit uniquement l'identifiant du type sélectionné. Nous devons le convertir en un objet `SportType` à l'aide d'un éditeur de propriétés. Avant tout, la méthode `getSportType()` de `ReservationService` doit obtenir un objet `SportType` à partir de son identifiant.

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public SportType getSportType(int sportTypeId);
}
```

Pour les tests, nous implémentons cette méthode avec une instruction `switch/case`.

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public SportType getSportType(int sportTypeId) {
        switch (sportTypeId) {
            case 1:
                return TENNIS;
            case 2:
                return SOCCER;
            default:
                return null;
        }
    }
}
```

Ensuite, nous créons la classe `SportTypeEditor`, qui convertit un identifiant de type de sport en un objet `SportType`. Cet éditeur de propriétés a besoin de `ReservationService` pour effectuer la recherche.

```
package com.apress.springrecipes.court.domain;
...
import java.beans.PropertyEditorSupport;

public class SportTypeEditor extends PropertyEditorSupport {

    private ReservationService reservationService;

    public SportTypeEditor(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    public void setAsText(String text) throws IllegalArgumentException {
        int sportTypeId = Integer.parseInt(text);
        SportType sportType = reservationService.getSportType(sportTypeId);
        setValue(sportType);
    }
}
```

La dernière étape consiste à enregistrer cet éditeur de propriétés dans notre contrôleur. Cela se passe dans la méthode `initBinder()` de l'objet `ServletRequestDataBinder`.

```
package com.apress.springrecipes.court.web;
...
public class ReservationFormController extends SimpleFormController {
    ...
    protected void initBinder(HttpServletRequest request,
        ServletRequestDataBinder binder) throws Exception {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(
            dateFormat, true));
        binder.registerCustomEditor(SportType.class, new SportTypeEditor(
            reservationService));
    }
}
```

Valider les données d'un formulaire

`SimpleFormController` peut nous aider à valider l'objet de commande avant qu'il ne procède à la liaison des champs du formulaire. La validation est opérée par un objet qui implémente l'interface `Validator`. Le validateur suivant vérifie que les champs obligatoires sont remplis et que l'horaire de réservation est valide pour les jours de semaine et les jours de vacances.

```
package com.apress.springrecipes.court.domain;
...
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
```

```
import org.springframework.validation.Validator;

public class ReservationValidator implements Validator {

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

    public void validate(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "courtName",
            "required.courtName", "Nom du terrain obligatoire.");
        ValidationUtils.rejectIfEmpty(errors, "date",
            "required.date", "Date obligatoire.");
        ValidationUtils.rejectIfEmpty(errors, "hour",
            "required.hour", "Horaire obligatoire.");
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "player.name",
            "required.playerName", "Nom du joueur obligatoire.");
        ValidationUtils.rejectIfEmpty(errors, "sportType",
            "required.sportType", "Type de sport obligatoire.");

        Reservation reservation = (Reservation) target;
        Date date = reservation.getDate();
        int hour = reservation.getHour();
        if (date != null) {
            Calendar calendar = Calendar.getInstance();
            calendar.setTime(date);
            if (calendar.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY) {
                if (hour < 8 || hour > 22) {
                    errors.reject("invalid.holidayHour",
                        "Horaire invalide pendant les vacances.");
                }
            } else {
                if (hour < 9 || hour > 21) {
                    errors.reject("invalid.weekdayHour",
                        "Horaire invalide en semaine.");
                }
            }
        }
    }
}
```

Dans la classe `ValidationUtils`, nous nous servons de méthodes utilitaires, comme `rejectIfEmptyOrWhitespace()` et `rejectIfEmpty()`, pour valider les champs de formulaire obligatoires. Si l'un de ces champs est vide, ces méthodes créent une *erreur de champ* liée au champ. Le deuxième argument de ces méthodes indique le nom de la propriété, tandis que le troisième et le quatrième correspondent au code d'erreur et au message d'erreur par défaut.

Nous vérifions également que l'horaire de réservation est valide pendant les vacances et les jours de la semaine. Si ce n'est pas le cas, nous utilisons la méthode `reject()` pour créer une *erreur d'objet*, qui est liée à l'objet de réservation, non à un champ.

Pour appliquer ce validateur à notre contrôleur, nous en configurons simplement une instance, que ce soit sous forme de bean interne ou de référence de bean, dans la propriété `validator`. Si plusieurs validateurs doivent être appliqués à un contrôleur, ils sont indiqués dans la propriété `validators` de type tableau.

```
<bean id="reservationFormController"
      class="com.apress.springrecipes.court.web.ReservationFormController">
    ...
    <property name="validator">
      <bean class="com.apress.springrecipes.court.domain.➥
                  ReservationValidator" />
    </property>
  </bean>
```

Puisque les validateurs peuvent créer des erreurs au cours de la validation, nous devons définir les messages qui sont affichés à l'utilisateur. Si nous avons défini un `ResourceBundleMessageSource`, nous pouvons inclure les messages d'erreur suivants dans le bundle de ressources pour les paramètres régionaux adéquats (par exemple `messages.properties` pour la localisation par défaut) :

```
required.courtName=Nom du terrain obligatoire
required.date=Date obligatoire
required.hour=Horaire obligatoire
required.playerName=Nom du joueur obligatoire
required.sportType=Type de sport obligatoire
invalid.holidayHour=Horaire invalide pendant les vacances
invalid.weekdayHour=Horaire invalide en semaine
```

10.11 Gérer les formulaires multipages

Problème

Dans une application web, il arrive que les formulaires soient très complexes et occupent plusieurs pages. Ces formulaires sont parfois appelés *formulaires assistants* car l'utilisateur doit les remplir page par page, comme dans un assistant logiciel. Indubitablement, pour gérer ces formulaires multipages, nous pouvons créer un ou plusieurs contrôleurs de formulaires en étendant `SimpleFormController`. Cependant, le travail devient assez complexe si nous devons conserver l'état du formulaire sur l'ensemble des pages.

Solution

La classe `AbstractWizardFormController` de Spring MVC définit les tâches élémentaires de la gestion des formulaires de type assistants. Elle prend également en charge le concept d'objet de commande et permet de lier les champs des différentes pages du formulaire aux propriétés éponymes du même objet de commande. En dérivant de la classe `AbstractWizardFormController`, notre contrôleur de formulaire devient capable de gérer les formulaires assistants.

Puisqu'un formulaire assistant est composé de plusieurs pages, nous devons définir plusieurs vues de page pour le contrôleur. Celui-ci gère l'état du formulaire sur l'ensemble des pages. Un formulaire assistant doit proposer plusieurs actions, pas uniquement la soumission comme dans `SimpleFormController`. La classe `AbstractWizardFormController` détermine l'action de l'utilisateur par l'intermédiaire d'un paramètre de requête particulier, généralement indiqué dans le nom d'un bouton de soumission :

- `_finish`. Terminer le formulaire assistant.
- `_cancel`. Annuler le formulaire assistant.
- `_targetx`. Aller à la page indiquée par *x*, l'indice de la première page étant zéro.

La classe `AbstractWizardFormController` offre la plupart des méthodes de `SimpleFormController` pour la gestion du cycle de vie afin que nous puissions personnaliser le flux de traitement du formulaire. En réalité, elles dérivent toutes deux de la classe de base `AbstractFormController`.

Explications

Nous voulons offrir un service qui permet à l'utilisateur de réserver un terrain aux mêmes heures pendant une certaine période. Nous définissons tout d'abord la classe `PeriodicReservation` dans le sous-paquetage `domain`.

```
package com.apress.springrecipes.court.domain;
...
public class PeriodicReservation {

    private String courtName;
    private Date fromDate;
    private Date toDate;
    private int period;
    private int hour;
    private Player player;

    // Accesseurs et mutateurs.
    ...
}
```

Nous ajoutons ensuite la méthode `makePeriodic()` à l'interface `ReservationService` pour effectuer une réservation périodique.

```
package com.apress.springrecipes.court.service;
...
public interface ReservationService {
    ...
    public void makePeriodic(PeriodicReservation periodicReservation)
        throws ReservationNotAvailableException;
}
```

Dans l'implémentation de cette méthode, nous générerons différents objets Reservation à partir de PeriodicReservation et passons chaque réservation à la méthode make(). Pour cette application simple, nous choisissons de ne pas utiliser les transactions.

```
package com.apress.springrecipes.court.service;
...
public class ReservationServiceImpl implements ReservationService {
    ...
    public void makePeriodic(PeriodicReservation periodicReservation)
        throws ReservationNotAvailableException {
        Calendar fromCalendar = Calendar.getInstance();
        fromCalendar.setTime(periodicReservation.getFromDate());

        Calendar toCalendar = Calendar.getInstance();
        toCalendar.setTime(periodicReservation.getToDate());

        while (fromCalendar.before(toCalendar)) {
            Reservation reservation = new Reservation();
            reservation.setCourtName(periodicReservation.getCourtName());
            reservation.setDate(fromCalendar.getTime());
            reservation.setHour(periodicReservation.getHour());
            reservation.setPlayer(periodicReservation.getPlayer());
            make(reservation);

            fromCalendar.add(Calendar.DATE, periodicReservation.getPeriod());
        }
    }
}
```

Créer les pages du formulaire

Supposons que la réservation périodique soit décomposée en trois pages, chacune présentant une partie des champs du formulaire. La première page, reservationCourtForm.jsp, demande uniquement le nom du terrain à réserver.

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

<html>
<head>
<title>Formulaire de réservation du terrain</title>
<style>
.error {
    color: #ff0000;
    font-weight: bold;
}
</style>
</head>

<body>
<form:form method="POST" commandName="reservation">
<table>
<tr>
    <td>Nom du terrain</td>
    <td><form:input path="courtName" /></td>
    <td><form:errors path="courtName" cssClass="error" /></td>
</tr>
</table>
</form:form>
</body>

```

```
<tr>
    <td colspan="3">
        <input type="submit" value="Suivant" name="_target1" />
        <input type="submit" value="Annuler" name="_cancel" />
    </td>
</tr>
</table>
</form:form>
</body>
</html>
```

Le formulaire et les champs de saisie de cette page sont définis à l'aide des balises `<form:form>` et `<form:input>` de Spring. Ils sont liés à l'objet de commande et à ses propriétés. Une balise d'erreur permet d'afficher un message d'erreur à l'utilisateur. Cette page contient deux boutons de soumission. Le nom du bouton Suivant est `_target1`. Il demande au contrôleur de formulaire assistant de passer à la deuxième page, dont l'indice est 1 (les numéros de pages commencent à zéro). Le nom du bouton Annuler est `_cancel`, car il demande au contrôleur d'annuler ce formulaire.

La deuxième page, `reservationTimeForm.jsp`, demande la date et l'horaire de la réservation périodique.

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

<html>
<head>
<title>Formulaire de réservation de l'horaire</title>
<style>
.error {
    color: #ff0000;
    font-weight: bold;
}
</style>
</head>

<body>
<form:form method="POST" commandName="reservation">
<table>
    <tr>
        <td>Date de début</td>
        <td><form:input path="fromDate" /></td>
        <td><form:errors path="fromDate" cssClass="error" /></td>
    </tr>
    <tr>
        <td>Date de fin</td>
        <td><form:input path="toDate" /></td>
        <td><form:errors path="toDate" cssClass="error" /></td>
    </tr>
    <tr>
        <td>Période</td>
        <td><form:select path="period" items="${periods}" /></td>
        <td><form:errors path="period" cssClass="error" /></td>
    </tr>
</table>
</form:form>
</body>
```

```
<tr>
    <td>Horaire</td>
    <td><form:input path="hour" /></td>
    <td><form:errors path="hour" cssClass="error" /></td>
</tr>
<tr>
    <td colspan="3">
        <input type="submit" value="Précédent" name="_target0" />
        <input type="submit" value="Suivant" name="_target2" />
        <input type="submit" value="Annuler" name="_cancel" />
    </td>
</tr>
</table>
</form:form>
</body>
</html>
```

Ce formulaire contient trois boutons de soumission. Les noms des boutons Précédent et Suivant sont respectivement _target0 et _target2. Ils demandent au contrôleur de formulaire assistant d'aller à la première et à la troisième page. Le bouton Annuler demande au contrôleur d'annuler ce formulaire.

La troisième page, `reservationPlayerForm.jsp`, demande les informations concernant le joueur.

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

<html>
<head>
<title>Formulaire de réservation du joueur</title>
<style>
.error {
    color: #ff0000;
    font-weight: bold;
}
</style>
</head>

<body>
<form:form method="POST" commandName="reservation">
<table>
<tr>
    <td>Nom du joueur</td>
    <td><form:input path="player.name" /></td>
    <td><form:errors path="player.name" cssClass="error" /></td>
</tr>
<tr>
    <td>N° de téléphone du joueur</td>
    <td><form:input path="player.phone" /></td>
    <td><form:errors path="player.phone" cssClass="error" /></td>
</tr>
<tr>
    <td colspan="3">
        <input type="submit" value="Précédent" name="_target1" />
        <input type="submit" value="Terminer" name="_finish" />
    </td>
</tr>
</table>
</form:form>
</body>
</html>
```

```
<input type="submit" value="Annuler" name="_cancel" />
</td>
</tr>
</table>
</form:form>
</body>
</html>
```

Ce formulaire contient trois boutons de soumission. Le bouton Précédent demande au contrôleur de formulaire assistant de revenir à la deuxième page. Le nom du bouton Terminer est `_finish` afin que le contrôleur termine ce formulaire. Le bouton Annuler demande au contrôleur d'annuler ce formulaire.

Créer un contrôleur de formulaire assistant

Créons à présent le contrôleur qui prend en charge ce formulaire de réservation périodique. Comme `SimpleFormController`, `AbstractWizardFormController` reconnaît le concept d'objet de commande. Soit nous indiquons la classe de commande qu'il doit instancier, soit nous initialisons l'objet de commande dans la méthode `formBackingObject()`. Les valeurs des champs du formulaire sont ensuite liées aux propriétés épynomiques de l'objet de commande. Pour un contrôleur de formulaire assistant, tous les champs du formulaire des différentes pages sont liés au même objet de commande. Il est enregistré dans la session afin de persister tout au long des multiples requêtes.

```
package com.apress.springrecipes.court.web;
...
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.validation.BindException;
import org.springframework.web.bind.ServletRequestDataBinder;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractWizardFormController;

public class PeriodicReservationController
    extends AbstractWizardFormController {

    private ReservationService reservationService;

    public PeriodicReservationController() {
        setCommandName("reservation");
    }

    public void setReservationService(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    protected Object formBackingObject(HttpServletRequest request)
        throws Exception {
        PeriodicReservation reservation = new PeriodicReservation();
        reservation.setPlayer(new Player());
        return reservation;
    }
}
```

```
protected void initBinder(HttpServletRequest request,
    ServletRequestDataBinder binder) throws Exception {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    dateFormat.setLenient(false);
    binder.registerCustomEditor(Date.class, new CustomDateEditor(
        dateFormat, true));
}

protected Map referenceData(HttpServletRequest request, int page)
    throws Exception {
    Map referenceData = new HashMap();
    if (page == 1) {
        Map<Integer, String> periods = new HashMap<Integer, String>();
        periods.put(1, "Quotidienne");
        periods.put(7, "Hebdomadaire");
        referenceData.put("periods", periods);
    }
    return referenceData;
}

protected ModelAndView processFinish(HttpServletRequest request,
    HttpServletResponse response, Object command, BindException errors)
    throws Exception {
    PeriodicReservation reservation = (PeriodicReservation) command;
    reservationService.makePeriodic(reservation);
    return new ModelAndView("reservationSuccessRedirect");
}

protected ModelAndView processCancel(HttpServletRequest request,
    HttpServletResponse response, Object command, BindException errors)
    throws Exception {
    return new ModelAndView("welcomeRedirect");
}
}
```

Pour que le contrôleur de formulaire assistant puisse convertir les valeurs des champs liés, nous devons enregistrer nos éditeurs de propriétés auprès de l'objet `ServletRequestDataBinder` dans la méthode `initBinder()`.

Nous devons afficher les périodes disponibles dans une sélection HTML sur la deuxième page. Nous redéfinissons donc la méthode `referenceData()` pour créer une table des périodes existantes, que nous plaçons dans le modèle. Il ne faut pas oublier de vérifier l'indice de la page affichée avant de créer la table des périodes. En effet, un contrôleur de formulaire assistant invoque la méthode `referenceData()` avant la présentation de chaque page.

Un argument des méthodes `processFinish()` et `processCancel()` nous permet d'accéder à l'objet de commande lié aux champs du formulaire. Pour terminer l'assistant, nous passons cet objet à `ReservationService`, qui effectue la réservation périodique, puis nous allons à la page `reservationSuccess`. Si l'utilisateur annule l'assistant, nous le dirigeons simplement vers la page d'accueil.

Dans la déclaration de ce contrôleur, nous devons fournir une référence au bean reservationService de la couche de service pour effectuer les réservations périodiques. Par ailleurs, nous devons indiquer l'ordre des pages du contrôleur.

```
<bean id="periodicReservationController"
      class="com.apress.springrecipes.court.web.PeriodicReservationController">
    <property name="reservationService" ref="reservationService" />
    <property name="pages">
      <list>
        <value>reservationCourtForm</value>
        <value>reservationTimeForm</value>
        <value>reservationPlayerForm</value>
      </list>
    </property>
  </bean>
```

Puisque ControllerClassNameHandlerMapping est configuré dans le contexte d'application web, l'URL suivante permet d'accéder à notre contrôleur :

<http://localhost:8080/court/periodicReservation.htm>

Valider les données du formulaire

Dans un contrôleur de formulaire simple, nous validons l'intégralité de l'objet de commande en une seule opération au moment où le formulaire est soumis. En revanche, lorsque le formulaire contient plusieurs pages, il est nécessaire d'effectuer une validation lors de la soumission de chaque page. C'est pourquoi nous créons le validateur suivant, qui décompose la méthode validate() en plusieurs méthodes de validation élémentaires, chacune validant les champs d'une page précise.

```
package com.apress.springrecipes.court.domain;

import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;

public class PeriodicReservationValidator implements Validator {

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

    public void validate(Object target, Errors errors) {
        validateCourt(target, errors);
        validateTime(target, errors);
        validatePlayer(target, errors);
    }

    public void validateCourt(Object target, Errors errors) {
        ValidationUtils.rejectIfEmptyOrWhitespace(errors, "courtName",
            "required.courtName", "Nom du terrain obligatoire.");
    }
}
```

```

public void validateTime(Object target, Errors errors) {
    ValidationUtils.rejectIfEmpty(errors, "fromDate",
        "required.fromDate", "Date de début obligatoire.");
    ValidationUtils.rejectIfEmpty(errors, "toDate", "required.toDate",
        "Date de fin obligatoire.");
    ValidationUtils.rejectIfEmpty(errors, "period",
        "required.period", "Période obligatoire.");
    ValidationUtils.rejectIfEmpty(errors, "hour", "required.hour",
        "Période obligatoire.");
}

public void validatePlayer(Object target, Errors errors) {
    ValidationUtils.rejectIfEmptyOrWhitespace(errors, "player.name",
        "required.playerName", "Nom du joueur obligatoire.");
}
}

```

Dans un contrôleur de formulaire simple, la validation se fait automatiquement par invocation des validateurs enregistrés. En revanche, dans un contrôleur de formulaire assistant, les validateurs enregistrés ne sont pas invoqués automatiquement. Nous devons procéder à la validation en redéfinissant `validatePage()` et en y invoquant manuellement les validateurs.

```

package com.apress.springrecipes.court.web;
...
import org.springframework.validation.Errors;
import org.springframework.validation.ValidationUtils;
import org.springframework.web.servlet.mvc.AbstractWizardFormController;

public class PeriodicReservationController
    extends AbstractWizardFormController {
    ...
    protected void validatePage(Object command, Errors errors, int page) {
        PeriodicReservationValidator validator =
            (PeriodicReservationValidator) getValidator();
        switch (page) {
            case 0:
                validator.validateCourt(command, errors);
                break;
            case 1:
                validator.validateTime(command, errors);
                break;
            case 2:
                validator.validatePlayer(command, errors);
                break;
        }
    }
}

```

Puisque cette méthode est appelée à chaque soumission de page, nous devons valider individuellement chaque page en examinant son indice. Une instruction `switch` simplifie cette opération. Toutefois, pour que cela fonctionne, nous devons configurer une instance du validateur précédent dans la propriété `validator`.

```
<bean id="periodicReservationController"
      class="com.apress.springrecipes.court.web.PeriodicReservationController">
    ...
    <property name="validator">
      <bean class="com.apress.springrecipes.court.domain.➥
                  PeriodicReservationValidator" />
    </property>
  </bean>
```

En cas d'erreur de liaison ou de validation dans une page du formulaire, le contrôleur de formulaire assistant affiche à nouveau cette page avec les messages d'erreur.

10.12 Regrouper plusieurs actions dans un contrôleur

Problème

Avec l'approche un contrôleur par action, l'ajout d'une action dans notre application nous oblige à configurer un contrôleur supplémentaire dans le contexte d'application web. Pour simplifier la configuration, nous souhaitons minimiser le nombre de contrôleurs dans notre application Spring MVC.

Solution

La classe `MultiActionController` fournie par Spring MVC permet de regrouper plusieurs actions connexes dans un même contrôleur. En dérivant notre contrôleur de cette classe, il peut offrir plusieurs méthodes gestionnaires pour traiter de multiples actions.

Dans un contrôleur multiaction, voici comment définir des méthodes gestionnaires :

```
public (ModelAndView | Map | String | void) actionPerformed(
    HttpServletRequest, HttpServletResponse [,HttpSession] [,CommandObject]);
```

La valeur de retour d'une méthode gestionnaire peut être de type `ModelAndView` (un modèle et un nom de vue ou un objet de vue), `Map` (uniquement un modèle), `String` (uniquement un nom de vue) ou `void` (la méthode gère elle-même la réponse HTTP).

Lorsqu'une requête est associée à un contrôleur multiaction par les mappages de gestionnaires, la correspondance doit être affinée de manière à viser une méthode gestionnaire précise du contrôleur. `MultiActionController` nous permet de configurer des mappages de méthodes en utilisant un objet `MethodNameResolver`.

Explications

Supposons que nous devions développer une fonction de suivi des membres du centre sportif. Pour que les administrateurs puissent ajouter, supprimer et afficher les membres, nous commençons par définir la classe de domaine `Member` et l'interface `MemberService`.

```
package com.apress.springrecipes.court.domain;

public class Member {

    private String name;
    private String phone;
    private String email;

    // Accesseurs et mutateurs.
    ...
}

---

package com.apress.springrecipes.court.service;
...
public interface MemberService {

    public void add(Member member);
    public void remove(String memberName);
    public List<Member> list();
}
```

Pour les tests, l'implémentation de cette interface enregistre les membres dans une table d'association dont les clés sont les noms des membres. Dans une application de production, ils seraient enregistrés dans une base de données.

```
package com.apress.springrecipes.court.service;
...
public class MemberServiceImpl implements MemberService {

    private Map<String, Member> members = new TreeMap<String, Member>();

    public void add(Member member) {
        members.put(member.getName(), member);
    }

    public void remove(String memberName) {
        members.remove(memberName);
    }

    public List<Member> list() {
        return new ArrayList<Member>(members.values());
    }
}
```

Nous déclarons ensuite un bean de ce type dans le fichier de configuration de la couche de service (c'est-à-dire court-service.xml).

```
<bean id="memberService"
      class="com.apress.springrecipes.court.service.MemberServiceImpl" />
```

Créer un contrôleur multiaction

Nous pourrions écrire un contrôleur simple pour traiter chaque action, mais cela nous obligerait à configurer trois contrôleurs supplémentaires dans le contexte d'application web. Puisque ces contrôleurs sont très simples et assurent des actions connexes, il est préférable de les regrouper dans un seul contrôleur qui étend la classe `MultiActionController`. Ensuite, nous pouvons déclarer plusieurs méthodes gestionnaires au sein du contrôleur.

```
package com.apress.springrecipes.court.web;
...
import org.springframework.web.bind.ServletRequestUtils;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.multiaction.MultiActionController;

public class MemberController extends MultiActionController {

    private MemberService memberService;

    public void setMemberService(MemberService memberService) {
        this.memberService = memberService;
    }

    public ModelAndView add(HttpServletRequest request,
                           HttpServletResponse response, Member member) throws Exception {
        memberService.add(member);
        return new ModelAndView("redirect:list.htm");
    }

    public ModelAndView remove(HttpServletRequest request,
                           HttpServletResponse response) throws Exception {
        String memberName = ServletRequestUtils.getRequiredStringParameter(
            request, "memberName");
        memberService.remove(memberName);
        return new ModelAndView("redirect:list.htm");
    }

    public ModelAndView list(HttpServletRequest request,
                           HttpServletResponse response) throws Exception {
        List<Member> members = memberService.list();
        return new ModelAndView("memberList", "members", members);
    }
}
```

Pour la méthode `add()`, nous voulons que le contrôleur lie les paramètres de requête à un objet de commande de type `Member`. Par conséquent, le troisième argument de la méthode est de type `Member`. Pour les méthodes `remove()` et `list()`, nous déclarons uniquement les arguments de requête et de réponse. À la fin des méthodes `add()` et `remove()`, nous redirigeons l'utilisateur vers la page qui affiche la liste des membres.

Dans la déclaration de ce contrôleur, nous devons fournir une référence au bean `memberService` de la couche de service pour la gestion de la liste des membres.

```
<bean id="memberController"
      class="com.apress.springrecipes.court.web.MemberController">
    <property name="memberService" ref="memberService" />
</bean>
```

Puisque ControllerClassNameHandlerMapping est configuré dans le contexte d'application web, il remarque que ce contrôleur est de type MultiActionController et génère la correspondance suivante :

MemberController → /member/*

Par défaut, MultiActionController associe les URL aux méthodes gestionnaires en utilisant les noms de méthodes. Pour notre contrôleur, les URL sont associées aux méthodes suivantes :

```
/member/add.htm → add()
/member/remove.htm → remove()
/member/list.htm → list()
```

Créons à présent la vue memberList.jsp pour ce contrôleur. Dans cette page, un formulaire permet d'ajouter un nouveau membre. Il est suivi d'une liste qui affiche tous les membres. Dans la dernière colonne de la liste, un lien permet de supprimer un membre.

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>

<html>
<head>
<title>Liste des membres</title>
</head>

<body>
<form action="add.htm">
  Nom <input type="text" name="name" />
  Téléphone <input type="text" name="phone" />
  Email <input type="text" name="email" />
  <input type="submit" />
</form>
<table border="1">
  <tr>
    <th>Nom</th>
    <th>Téléphone</th>
    <th>Email</th>
    <th></th>
  </tr>
  <c:forEach items="${members}" var="member">
    <tr>
      <td>${member.name}</td>
      <td>${member.phone}</td>
      <td>${member.email}</td>
      <td><a href="remove.htm?memberName=${member.name}">Supprimer</a></td>
    </tr>
  </c:forEach>
</table>
</body>
</html>
```

Puisque l'action list qui affiche cette page est accessible par /member/list.htm, le formulaire d'ajout d'un membre doit être soumis à l'URL relative add.htm, qui correspond à /member/add.htm. Pour la même raison, le lien de suppression d'un membre doit cibler l'URL relative remove.htm, qui correspond à /member/remove.htm.

Lorsque l'action d'ajout ou de suppression est terminée, nous redirigeons le navigateur de l'utilisateur vers l'action d'affichage de la liste des membres. Nous pouvons afficher tous les membres à l'aide de l'URL suivante :

```
http://localhost:8080/court/member/list.htm
```

Associer des URL à des méthodes gestionnaires

Par défaut, MultiActionController utilise InternalPathMethodNameResolver pour la correspondance des URL et des méthodes gestionnaires en se fondant sur les noms de méthodes. Toutefois, si nous souhaitons ajouter un préfixe ou un suffixe aux noms des méthodes, nous devons configurer explicitement ce résolveur.

```
<bean id="memberController"
      class="com.apress.springrecipes.court.web.MemberController">
    ...
    <property name="methodNameResolver">
      <bean class="org.springframework.web.servlet.mvc.multiaction.<span style="color: #0000ff;">➥
            InternalPathMethodNameResolver">
        <property name="suffix" value="Member" />
      </bean>
    </property>
  </bean>
```

Ainsi, le chemin final d'une URL, avant l'extension, est associé à une méthode gestionnaire en ajoutant le suffixe Member.

```
/member/add.htm → addMember()
/member/remove.htm → removeMember()
/member/list.htm → listMember()
```

Pour tester ce résolveur, nous renommons les méthodes de MemberController.

```
package com.apress.springrecipes.court.web;
...
public class MemberController extends MultiActionController {
  ...
  public ModelAndView addMember(HttpServletRequest request,
                                HttpServletResponse response, Member member) throws Exception {
  ...
}

  public ModelAndView removeMember(HttpServletRequest request,
                                HttpServletResponse response) throws Exception {
  ...
}
```

```

public ModelAndView listMember(HttpServletRequest request,
    HttpServletResponse response) throws Exception {
    ...
}
}

```

Une autre solution consiste à configurer un `PropertiesMethodNameResolver` pour associer des URL à des méthodes gestionnaires en précisant explicitement les correspondances.

```

<bean id="memberController"
    class="com.apress.springrecipes.court.web.MemberController">
    ...
    <property name="methodNameResolver">
        <bean class="org.springframework.web.servlet.mvc.multiaction.>
            PropertiesMethodNameResolver">
            <property name="mappings">
                <props>
                    <prop key="/member/add.htm">addMember</prop>
                    <prop key="/member/remove.htm">removeMember</prop>
                    <prop key="/member/list.htm">listMember</prop>
                </props>
            </property>
        </bean>
    </property>
</bean>

```

Dans ce cas, une URL est associée à la méthode gestionnaire dont le nom est indiqué dans la définition des correspondances.

```

/member/add.htm → addMember()
/member/remove.htm → removeMember()
/member/list.htm → listMember()

```

Enfin, nous pouvons configurer un `ParameterMethodNameResolver` pour associer les URL aux méthodes gestionnaires en fonction d'un paramètre de requête. Le nom du paramètre pour ce résolveur peut être précisé dans la propriété `paramName`. Si nous le fixons à `method`, une URL est associée à la méthode gestionnaire dont le nom correspond à la valeur du paramètre de requête `method`. Dans les URL suivantes, les caractères génériques correspondent à n'importe quelle chaîne de caractères.

```

/member/*.htm?method=addMember → addMember()
/member/*.htm?method=removeMember → removeMember()
/member/*.htm?method=listMember → listMember()

```

10.13 Créer des vues Excel et PDF

Problème

Bien que HTML reste la solution la plus répandue pour afficher du contenu web, les utilisateurs souhaitent parfois exporter le contenu d'une application web au format Excel ou PDF. Il existe plusieurs bibliothèques Java pour générer des fichiers Excel ou PDF. Cependant, lorsque ces bibliothèques sont employées directement dans une application web, il est nécessaire de générer des fichiers et de les retourner aux utilisateurs sous forme de pièces jointes binaires. Pour cela, nous devons manipuler les en-têtes de réponse HTTP et les flux de sortie.

Solution

Dans son framework MVC, Spring propose une fonction de génération de fichiers Excel et PDF. Si l'on considère que les fichiers Excel et PDF constituent des sortes de vues particulières, nous pouvons traiter une requête web normalement dans un contrôleur et ajouter des données au modèle pour les passer aux vues Excel et PDF. De cette manière, nous n'avons aucunement besoin de manipuler des en-têtes de réponse HTTP et des flux de sortie.

La génération de fichiers Excel dans Spring MVC se fait avec la bibliothèque Apache POI (<http://poi.apache.org/>) ou la bibliothèque JExcelAPI (<http://jexcelapi.sourceforge.net/>). Les classes correspondantes pour les vues se nomment `AbstractExcelView` et `AbstractJExcelView`. Les fichiers PDF sont générés à l'aide de la bibliothèque iText (<http://www.lowagie.com/iText/>), et la classe de vue se nomme `AbstractPdfView`.

Explications

Supposons que nos utilisateurs souhaitent générer un rapport récapitulant les réservations d'une journée. Ils veulent ce rapport au format Excel ou PDF. Pour cette fonction de génération de rapport, nous devons déclarer une méthode dans la couche de service qui retourne toutes les réservations pour une journée.

```
package com.apress.springrecipes.court.service;  
...  
public interface ReservationService {  
    ...  
    public List<Reservation> findByDate(Date date);  
}
```

Dans cette méthode, nous parcourons toutes les réservations effectuées.

```
package com.apress.springrecipes.court.service;  
...  
public class ReservationServiceImpl implements ReservationService {  
    ...
```

```

        public List<Reservation> findByDate(Date date) {
            List<Reservation> result = new ArrayList<Reservation>();
            for (Reservation reservation : reservations) {
                if (reservation.getDate().equals(date)) {
                    result.add(reservation);
                }
            }
            return result;
        }
    }
}

```

Nous pouvons à présent écrire un contrôleur simple qui récupère des paramètres de date et de format dans l'URL. Le paramètre date est converti en un objet de date et passé à la couche de service pour obtenir les réservations. Le paramètre format indique le format, Excel ou PDF, dans lequel le rapport doit être généré.

```

package com.apress.springrecipes.court.web;
...
import org.springframework.web.bind.ServletRequestUtils;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class ReservationSummaryController extends AbstractController {

    private ReservationService reservationService;

    public void setReservationService(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    protected ModelAndView handleRequestInternal(HttpServletRequest request,
                                                HttpServletResponse response) throws Exception {
        String date =
            ServletRequestUtils.getRequiredStringParameter(request, "date");
        String format =
            ServletRequestUtils.getRequiredStringParameter(request, "format");

        Date summaryDate = new SimpleDateFormat("yyyy-MM-dd").parse(date);
        List<Reservation> reservations =
            reservationService.findByDate(summaryDate);
        return new ModelAndView(format + "Summary", "reservations",
                               reservations);
    }
}

```

Ce contrôleur retourne une vue dont le nom est excelSummary lorsque le paramètre de format vaut excel, et pdfSummary lorsqu'il vaut pdf. Pour déclarer ce contrôleur, nous devons fournir une référence au bean reservationService de la couche de service de manière à consulter les réservations d'une certaine journée.

```

<bean id="reservationSummaryController"
      class="com.apress.springrecipes.court.web.ReservationSummaryController">
    <property name="reservationService" ref="reservationService" />
</bean>

```

Créer des vues Excel

Pour créer une vue Excel, nous étendons la classe `AbstractExcelView` (pour la bibliothèque Apache POI) ou la classe `AbstractJExcelView` (pour JExcelAPI) ; nous prenons comme exemple `AbstractExcelView`. Dans la méthode `buildExcelDocument()`, nous accédons au modèle passé par le contrôleur, ainsi qu'à un classeur Excel déjà créé. Notre travail consiste simplement à remplir le classeur avec les données du modèle.

INFO

Pour générer des fichiers Excel avec la bibliothèque Apache POI dans une application web, vous devez copier le fichier `poi-3.0.1.jar` (situé dans le répertoire `lib/poi` de l'installation de Spring) dans le répertoire `WEB-INF/lib`.

```
package com.apress.springrecipes.court.web.view;
...
import org.apache.poi.hssf.usermodel.HSSFRow;
import org.apache.poi.hssf.usermodel.HSSFSheet;
import org.apache.poi.hssf.usermodel.HSSFWorkbook;

import org.springframework.web.servlet.view.document.AbstractExcelView;

public class ExcelReservationSummary extends AbstractExcelView {

    protected void buildExcelDocument(Map model, HSSFWorkbook workbook,
        HttpServletRequest request, HttpServletResponse response)
        throws Exception {
        List<Reservation> reservations = (List) model.get("reservations");
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        HSSFSheet sheet = workbook.createSheet();

        HSSFRow header = sheet.createRow(0);
        header.createCell((short) 0).setCellValue("Nom du terrain");
        header.createCell((short) 1).setCellValue("Date");
        header.createCell((short) 2).setCellValue("Horaire");
        header.createCell((short) 3).setCellValue("Nom du terrain");
        header.createCell((short) 4).setCellValue("N° de téléphone du joueur");

        int rowNum = 1;
        for (Reservation reservation : reservations) {
            HSSFRow row = sheet.createRow(rowNum++);
            row.createCell((short) 0).setCellValue(reservation.getCourtName());
            row.createCell((short) 1).setCellValue(
                dateFormat.format(reservation.getDate()));
            row.createCell((short) 2).setCellValue(reservation.getHour());
            row.createCell((short) 3).setCellValue(
                reservation.getPlayer().getName());
            row.createCell((short) 4).setCellValue(
                reservation.getPlayer().getPhone());
        }
    }
}
```

Dans la vue Excel précédente, nous commençons par créer une feuille de calcul dans le classeur. Dans cette feuille, nous plaçons les intitulés du rapport sur la première ligne. Ensuite, nous parcourons la liste des réservations afin de créer une ligne pour chacune d'elles.

Puisque `ResourceBundleViewResolver` est configuré dans le contexte d'application web, nous pouvons définir cette vue en ajoutant l'entrée suivante dans `views.properties`, situé à la racine du chemin d'accès aux classes :

```
excelSummary.(class)=>
    com.apress.springrecipes.court.web.view.ExcelReservationSummary
```

Puisque `ControllerClassNameHandlerMapping` est configuré dans le contexte d'application web, nous pouvons accéder à ce contrôleur via l'URL suivante, qui n'oublie pas de préciser les paramètres date et format :

```
http://localhost:8080/court/reservationSummary.htm?date=2008-01-14&format=excel
```

Créer des vues PDF

Pour créer une vue PDF, il faut étendre la classe `AbstractPdfView`. Dans la méthode `buildPdfDocument()`, nous accédons au modèle passé par le contrôleur, ainsi qu'à un document PDF déjà créé. Notre travail consiste simplement à remplir le document à l'aide des données du modèle.

INFO

Pour générer des fichiers PDF avec iText dans une application web, vous devez copier le fichier `iText-2.1.3.jar` (situé dans le répertoire `lib/iText` de l'installation de Spring) dans le répertoire `WEB-INF/lib`.

```
package com.apress.springrecipes.court.web.view;
...
import org.springframework.web.servlet.view.document.AbstractPdfView;
import com.lowagie.text.Document; import com.lowagie.text.Table;
import com.lowagie.text.pdf.PdfWriter;

public class PdfReservationSummary extends AbstractPdfView {

    protected void buildPdfDocument(Map model, Document document,
        PdfWriter writer, HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        List<Reservation> reservations = (List) model.get("reservations");
        DateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        Table table = new Table(5);
        table.addCell("Nom du terrain");
        table.addCell("Date");
        table.addCell("Horaire");
        table.addCell("Nom du joueur");
        table.addCell("N° de téléphone du joueur");
    }
}
```

```
        for (Reservation reservation : reservations) {
            table.addCell(reservation.getCourtName());
            table.addCell(dateFormat.format(reservation.getDate()));
            table.addCell(Integer.toString(reservation.getHour()));
            table.addCell(reservation.getPlayer().getName());
            table.addCell(reservation.getPlayer().getPhone());
        }
        document.add(table);
    }
}
```

Dans la vue PDF précédente, nous créons une table et affichons les intitulés du rapport sur la première ligne. Ensuite, nous parcourons la liste des réservations de manière à créer une ligne pour chacune d'elles. Enfin, nous ajoutons la table au document PDF.

Nous ajoutons l'entrée suivante dans `views.properties` pour définir cette vue :

```
pdfSummary.(class)=com.apress.springrecipes.court.web.view.➥
PdfReservationSummary
```

10.14 Développer des contrôleurs avec des annotations

Problème

Avec l'approche Spring MVC classique, nous devons définir dans le fichier de configuration des beans une instance et une correspondance de requête pour chaque classe de contrôleur. En permettant à Spring MVC de détecter automatiquement les classes des contrôleurs et les correspondances des requêtes, nous pouvons réduire le travail de configuration. Par ailleurs, chaque classe de contrôleur doit implémenter une interface ou étendre une classe de base propres au framework, ce qui n'est pas vraiment souple.

Solution

Dans Spring 2.5, nous pouvons développer des contrôleurs en utilisant des annotations. Grâce à l'annotation `@Controller`, Spring est capable de détecter automatiquement les classes de nos contrôleurs. Quant aux annotations `@RequestMapping`, elles permettent la détection des correspondances de requêtes. Nous échappons ainsi à la configuration des contrôleurs dans le fichier de configuration des beans. Par ailleurs, en utilisant les annotations, les classes de contrôleurs et les méthodes gestionnaires bénéficient d'une plus grande souplesse pour l'accès aux ressources du contexte, par exemple aux paramètres de la requête, aux attributs du modèle et aux attributs de la session.

L'annotation `@Controller` permet de désigner une classe quelconque comme une classe de contrôleur. A contrario des contrôleurs traditionnels, une classe de contrôleur défini par une annotation n'a pas besoin d'implémenter une interface ou d'étendre une classe de base du framework. À l'intérieur de cette classe, une ou plusieurs méthodes gestionnaires peuvent être marquées par `@RequestMapping`.

La signature des méthodes gestionnaires est très souple. Nous pouvons choisir un nom de méthode quelconque et définir des arguments dont les types sont pris dans la liste suivante. Cette liste se limite aux types les plus répandus, mais la liste complète est disponible dans la documentation de Spring qui traite de la configuration des contrôleurs avec des annotations et dans la documentation JavaDoc de `@RequestMapping`.

- `HttpServletRequest`, `HttpServletResponse` ou `HttpSession` ;
- des paramètres de requête de type quelconque, annotés par `@RequestParam` ;
- des attributs du modèle de type quelconque, annotés par `@ModelAttribute` ;
- un objet de commande de type quelconque, pour la liaison des paramètres de requêtes par Spring ;
- `Map` ou `ModelMap`, pour que la méthode gestionnaire ajoute des attributs au modèle ;
- `Errors` ou `BindingResult`, pour que la méthode de gestionnaire accède au résultat de la liaison et de la validation pour l'objet de commande ;
- `SessionStatus`, pour que la méthode gestionnaire signale la fin du traitement de la session.

La valeur de retour de la méthode gestionnaire peut être de type `ModelAndView` (un modèle et un nom de vue ou un objet de vue), `Map` (uniquement un modèle), `String` (uniquement un nom de vue) ou `void` (la méthode gère elle-même la réponse HTTP).

Expliations

Avant de pouvoir créer des contrôleurs avec des annotations, nous devons configurer leur prise en charge dans le contexte d'application web. Tout d'abord, pour que Spring détecte automatiquement nos contrôleurs marqués par `@Controller`, nous activons la fonctionnalité de scan des composants avec l'élément `<context:component-scan>`.

Pour que Spring MVC puisse associer des requêtes à des classes de contrôleurs et à des méthodes gestionnaires marquées par `@RequestMapping`, nous enregistrons une instance de `DefaultAnnotationHandlerMapping` et une instance de `AnnotationMethodHandlerAdapter` dans le contexte d'application web. Elles permettent, respectivement, de traiter les annotations `@RequestMapping` au niveau de la classe et au niveau de la méthode.

Pour nous limiter à cette nouvelle approche, c'est-à-dire des contrôleurs fondés sur des annotations, nous incluons uniquement les configurations Spring MVC requises dans `court-servlet.xml`.

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

```
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
                     http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                     http://www.springframework.org/schema/context
                     http://www.springframework.org/schema/context/spring-context-2.5.xsd">

<context:component-scan
    base-package="com.apress.springrecipes.court.web" />

<bean class="org.springframework.web.servlet.mvc.annotation.➥
    DefaultAnnotationHandlerMapping" />

<bean class="org.springframework.web.servlet.mvc.annotation.➥
    AnnotationMethodHandlerAdapter" />

<bean class="org.springframework.web.servlet.view.➥
    InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
</bean>

<bean id="messageSource" class="org.springframework.context.support.➥
    ResourceBundleMessageSource">
    <property name="basename" value="messages" />
</bean>
</beans>
```

Les instances de `DefaultAnnotationHandlerMapping` et de `AnnotationMethodHandlerAdapter` sont déjà enregistrées dans le contexte d'application web par défaut. Toutefois, si nous enregistrons explicitement d'autres mappages de gestionnaires ou des adaptateurs de gestionnaires, elles ne le sont plus et nous devons les enregistrer nous-mêmes.

Développer des contrôleurs mono et multiactions

Une classe de contrôleur définie par une annotation est une classe quelconque qui n'implémente aucune interface ni n'étend une classe de base particulière. Il suffit de la marquer avec l'annotation `@Controller`. Dans le contrôleur, nous pouvons définir une ou plusieurs méthodes gestionnaires pour effectuer une ou plusieurs actions. La signature des méthodes gestionnaires est suffisamment souple pour accepter toute une gamme d'arguments.

L'annotation `@RequestMapping` s'applique au niveau de la classe ou niveau de la méthode. La première stratégie consiste à associer un motif d'URL à une classe de contrôleur, puis une méthode HTTP précise à chaque méthode gestionnaire.

```
package com.apress.springrecipes.court.web;
...
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.servlet.ModelAndView;
```

```

@Controller
@RequestMapping("/welcome.htm")
public class WelcomeController {

    @RequestMapping(method = RequestMethod.GET)
    public ModelAndView welcome() {
        Date today = new Date();
        return new ModelAndView("welcome", "today", today);
    }
}

```

La seconde stratégie consiste à associer un motif d'URL directement à chaque méthode gestionnaire, sans définir de correspondance pour la classe de contrôleur.

```

package com.apress.springrecipes.court.web;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.servlet.ModelAndView;

@Controller
public class MemberController {

    private MemberService memberService;

    @Autowired
    public MemberController(MemberService memberService) {
        this.memberService = memberService;
    }

    @RequestMapping("/member/add.htm")
    public String addMember(Member member) {
        memberService.add(member);
        return "redirect:list.htm";
    }

    @RequestMapping("/member/remove.htm")
    public String removeMember(
        @RequestParam("memberName") String memberName) {
        memberService.remove(memberName);
        return "redirect:list.htm";
    }

    @RequestMapping("/member/list.htm")
    public ModelAndView listMember() {
        List<Member> members = memberService.list();
        return new ModelAndView("memberList", "members", members);
    }
}

```

Nous pouvons attribuer différents arguments à une méthode gestionnaire. Par exemple, l'argument de la méthode `addMember()` est un objet de commande de type `Member`. Les paramètres de requête sont ensuite liés aux propriétés éponymes de cet objet. Pour la

méthode `removeMember()`, nous définissons un argument lié à un paramètre de requête via l'annotation `@RequestParam()`. Par défaut, les paramètres de requête liés avec une annotation `@RequestParam()` sont obligatoires. Nous pouvons fixer l'attribut `required` à `false` pour les paramètres de requête facultatifs.

Développer un contrôleur de formulaire

Avec l'approche Spring MVC classique, nous créons un contrôleur de formulaire simple en dérivant de la classe `SimpleFormController`. Elle définit le déroulement type du traitement d'un formulaire et nous permet de l'adapter en redéfinissant les méthodes du cycle de vie. Avec l'approche Spring MVC fondée sur les annotations, nous reproduisons le flux de traitement d'un formulaire avec des annotations.

Dans le cadre de l'approche fondée sur les annotations, une classe de contrôleur marquée par `@Controller` est également capable de traiter les formulaires. La première étape consiste à associer un motif d'URL à la classe de contrôleur avec une annotation `@RequestMapping`. Pour qu'un contrôleur puisse gérer des formulaires, il doit fournir deux méthodes importantes. La première sert à afficher le formulaire en réponse à une requête HTTP `GET`. La seconde prend en charge la soumission du formulaire par une requête HTTP `POST`. Les noms de ces méthodes ne sont pas figés, mais elles doivent être associées à une méthode HTTP à l'aide de l'annotation `@RequestMapping`.

```
package com.apress.springrecipes.court.web;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.propertyeditors.CustomDateEditor;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.annotation.InitBinder;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

@Controller
@RequestMapping("/reservationForm.htm")
@SessionAttributes("reservation")
public class ReservationFormController {

    private ReservationService reservationService;
    private ReservationValidator validator;

    @Autowired
    public ReservationFormController(ReservationService reservationService,
                                    ReservationValidator validator) {
        this.reservationService = reservationService;
    }
}
```

```

        this.validator = validator;
    }

@InitBinder
public void initBinder(WebDataBinder binder) {
    SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
    dateFormat.setLenient(false);
    binder.registerCustomEditor(Date.class, new CustomDateEditor(
        dateFormat, true));
    binder.registerCustomEditor(SportType.class, new SportTypeEditor(
        reservationService));
}

@ModelAttribute("sportTypes")
public List<SportType> populateSportTypes() {
    return reservationService.getAllSportTypes();
}

@RequestMapping(method = RequestMethod.GET)
public String setupForm(
    @RequestParam(required = false,
        value = "username") String username, ModelMap model) {
    Reservation reservation = new Reservation();
    reservation.setPlayer(new Player(username, null));
    model.addAttribute("reservation", reservation);
    return "reservationForm";
}

@RequestMapping(method = RequestMethod.POST)
public String processSubmit(
    @ModelAttribute("reservation") Reservation reservation,
    BindingResult result, SessionStatus status) {
    validator.validate(reservation, result);
    if (result.hasErrors()) {
        return "reservationForm";
    } else {
        reservationService.make(reservation);
        status.setComplete();
        return "redirect:reservationSuccess.htm";
    }
}
}

```

La méthode `setupForm()`, associée à la méthode HTTP GET, correspond à la méthode `formBackingObject()` de `SimpleFormController`, qui initialise l'objet de commande pour les besoins de la liaison. Dans cette méthode, nous pouvons employer l'annotation `@RequestParam` pour accéder aux paramètres de requête de façon à initialiser l'objet de commande. Nous créons l'objet de commande nous-mêmes et l'enregistrons dans un attribut du modèle. Le nom de l'attribut est celui de l'objet de commande, auquel nous pouvons accéder dans les fichiers JSP. Pour enregistrer l'objet de commande dans la session, comme par simple activation de la propriété `sessionForm` de `SimpleFormController`, il suffit d'appliquer l'annotation `@SessionAttributes` à la classe de contrôleur et de préciser le nom de l'attribut du modèle à enregistrer dans la session.

La méthode `processSubmit()`, associée à la méthode HTTP POST, correspond à la méthode `onSubmit()` de `SimpleFormController`, qui s'occupe de la soumission du formulaire. Dans cette méthode, l'annotation `@ModelAttribute` nous permet d'accéder à l'objet de commande à partir du modèle. Appliquée à un argument de méthode, l'annotation `@ModelAttribute` sert à lier un attribut du modèle à cet argument. Contrairement à la méthode `onSubmit()` de `SimpleFormController`, nous devons procéder nous-mêmes à la validation du formulaire et décider de présenter la vue de formulaire ou la vue de réussite. Dès le traitement du formulaire terminé avec succès, nous effaçons l'objet de commande de la session en invoquant la méthode `setComplete()` de `SessionStatus`.

Appliquée à une méthode comme `populateSportTypes()`, l'annotation `@ModelAttribute` sert à fournir les données de référence du modèle. Le résultat est identique à la redéfinition de la méthode `referenceData()` de `SimpleFormController`.

La méthode `initBinder()` marquée par `@InitBinder` enregistre nos propres éditeurs de propriétés auprès de l'objet de liaison. Elle correspond à la méthode `initBinder()` de `SimpleFormController`.

En cas de réussite du traitement, la vue choisie par ce contrôleur est une vue de redirection vers la page de réservation confirmée. Nous pouvons créer un autre contrôleur à base d'annotation pour cette vue.

```
package com.apress.springrecipes.court.web;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;

@Controller
public class ReservationSuccessController {

    @RequestMapping("/reservationSuccess.htm")
    public String reservationSuccess() {
        return "reservationSuccess";
    }
}
```

Enfin, avant que ce contrôleur de formulaire ne soit opérationnel, nous devons définir une instance de validateur pour que Spring procède à la liaison automatique.

```
<bean id="reservationValidator"
      class="com.apress.springrecipes.court.domain.ReservationValidator" />
```

À des fins de réutilisation, nous pouvons extraire la tâche d'initialisation de la liaison de chaque classe de contrôleur pour former un initialiseur de liaison.

```
package com.apress.springrecipes.court.web;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.propertyeditors.CustomDateEditor;
```

```
import org.springframework.web.bind.WebDataBinder;
import org.springframework.web.bind.support.WebBindingInitializer;
import org.springframework.web.context.request.WebRequest;

public class ReservationBindingInitializer implements WebBindingInitializer {

    private ReservationService reservationService;

    @Autowired
    public ReservationBindingInitializer(
        ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    public void initBinder(WebDataBinder binder, WebRequest request) {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);
        binder.registerCustomEditor(Date.class, new CustomDateEditor(
            dateFormat, true));
        binder.registerCustomEditor(SportType.class, new SportTypeEditor(
            reservationService));
    }
}
```

Ensuite, nous signalons cet initialiseur de liaison à `AnnotationMethodHandlerAdapter` pour que toutes les méthodes gestionnaires partagent les mêmes éditeurs de propriétés.

```
<bean class="org.springframework.web.servlet.mvc.annotation.➥
    AnnotationMethodHandlerAdapter">
    <property name="webBindingInitializer">
        <bean class="com.apress.springrecipes.court.web.➥
            ReservationBindingInitializer" />
    </property>
</bean>
```

Nous pouvons à présent supprimer la méthode `initBinder()` de `ReservationFormController`, puisque le système de liaison est partagé entre tous les contrôleurs définis par des annotations.

Comparaison avec l'approche classique

L'approche fondée sur les annotations apporte une plus grande souplesse dans l'écriture des classes de contrôleurs et des méthodes gestionnaires, car nous n'avons pas besoin d'implémenter une interface ou d'étendre une classe de base particulières. Toutefois, pour qu'un tel contrôleur puisse gérer des formulaires, ses méthodes gestionnaires doivent effectuer des tâches qui, à l'origine, sont du ressort de `SimpleFormController`, comme invoquer le validateur, revenir à la vue de formulaire en cas d'erreur et aller à la vue de réussite en cas de succès du traitement. Par conséquent, nous devons plonger plus profond dans les détails de la gestion des formulaires.

10.15 En résumé

Dans ce chapitre, nous avons appris à développer une application web Java en utilisant le framework Spring MVC. L'élément central de Spring MVC est mis en œuvre par `DispatcherServlet`, dont le rôle de contrôleur frontal est de distribuer les requêtes aux gestionnaires appropriés en vue de leur traitement. Lorsque `DispatcherServlet` reçoit une requête, il commence à rechercher un gestionnaire à partir d'un ou de plusieurs mappages de gestionnaires. Dès qu'un gestionnaire a été trouvé, `DispatcherServlet` invoque celui-ci pour traiter la requête. À la fin du traitement de la requête, le gestionnaire retourne un modèle et un nom ou un objet de vue. Si un nom de vue est retourné, `DispatcherServlet` demande aux résolveurs de vue de le convertir en un objet de vue, auquel il passe ensuite le modèle pour son affichage.

`DispatcherServlet` a besoin d'un ensemble d'outils pour simplifier le traitement des requêtes. Il s'agit des mappages de gestionnaires, des résolveurs de localisation, des résolveurs de vue, des sources de messages et des résolveurs d'exceptions. Il est très facile de configurer et de personnaliser ces outils dans le contexte d'application web.

Spring MVC fournit plusieurs classes de contrôleurs que nous devons étendre en fonction du scénario d'utilisation, notamment `AbstractController`, `ParameterizableViewController`, `SimpleFormController`, `AbstractWizardFormController` et `MultiActionController`. Elles encapsulent la plupart des fonctionnalités, dont nos contrôleurs héritent en étendant simplement ces classes.

Spring MVC prend en charge différents types de vues pour différentes technologies de présentation. Par exemple, puisqu'il considère que les fichiers Excel et PDF sont des sortes de vues, nous pouvons traiter les requêtes web de manière normale dans un contrôleur et ajouter des données au modèle passé à des vues Excel et PDF.

Spring 2.5 nous permet de développer des contrôleurs en utilisant des annotations. Ils ont alors besoin non pas d'étendre les classes de contrôleurs classiques, mais simplement d'inclure des annotations. Avec cette approche, Spring peut détecter automatiquement les classes de contrôleurs et les mappages de gestionnaires, ce qui nous évite de les configurer manuellement.

Le chapitre suivant détaille l'intégration de Spring avec d'autres frameworks d'applications web répandus, notamment Struts, JSF et DWR.

Intégration avec d'autres frameworks web

Au sommaire de ce chapitre

- ✓ Accéder à Spring depuis des applications web génériques
- ✓ Intégrer Spring à Struts 1.x
- ✓ Intégrer Spring à JSF
- ✓ Intégrer Spring à DWR
- ✓ En résumé

Ce chapitre explique comment intégrer le framework Spring avec plusieurs autres frameworks d'applications web répandus, notamment Struts, JSF et DWR. La puissance du conteneur Spring IoC et ses fonctionnalités pour les applications d'entreprise font de Spring un outil bien adapté à la mise en œuvre des couches de service et de persistance dans les applications Java EE. En revanche, pour la couche de présentation, nous avons le choix entre plusieurs frameworks web différents. Par conséquent, nous devons souvent intégrer Spring avec un autre framework d'applications web. Cette intégration consiste principalement à permettre l'accès aux beans du conteneur Spring IoC depuis cet autre framework.

Apache Struts (<http://struts.apache.org/>) est un framework d'applications web open-source très connu, fondé sur le design pattern MVC. Struts a été utilisé par la communauté Java dans de nombreux projets web et dispose ainsi d'une grande base d'utilisateurs. Dans Spring, la prise en charge ne concerne que Struts 1.x. En effet, puisque Struts a fusionné avec WebWork dans sa version 2, il est très facile de configurer des actions Struts dans Spring en utilisant le conteneur Spring IoC comme fabrique d'objets pour Struts 2.

JSF (*JavaServer Faces*, <http://java.sun.com/javaee/javaserverfaces/>) est un excellent framework d'applications web basé sur les composants et orienté événement qui fait partie de la spécification Java EE. Nous pouvons non seulement utiliser les nombreux composants JSF standard, mais également développer nos propres composants pour les réutiliser. JSF permet une séparation propre entre la logique de présentation et l'interface utilisateur, en l'encapsulant dans un ou plusieurs beans gérés. En raison de son approche fondée sur les composants, JSF est pris en charge par un grand nombre d'IDE pour du développement visuel.

DWR (*Direct Web Remoting*, <http://getahead.org/dwr>) est une bibliothèque qui apporte des fonctionnalités Ajax (*Asynchronous JavaScript and XML*) aux applications web. Elle nous permet d'invoquer des objets Java présents sur le serveur en utilisant du code JavaScript dans un navigateur web. Nous pouvons également mettre à jour dynamiquement des parties d'une page web, sans avoir à actualiser l'intégralité de la page.

À la fin de ce chapitre, vous serez en mesure d'intégrer Spring dans des applications web réalisées avec le couple servlets/JSP et des frameworks d'applications web répandus, comme Struts, JSF et DWR.

11.1 Accéder à Spring depuis des applications web génériques

Problème

Nous voulons accéder aux beans déclarés dans le conteneur Spring IoC depuis une application web, quel que soit le framework qu'elle utilise.

Solution

Une application web peut charger un contexte d'application Spring en enregistrant l'écouteur de servlet `ContextLoaderListener`. Il enregistre le contexte d'application chargé dans le contexte de servlet de l'application web. Par la suite, une servlet, ou n'importe quel objet capable d'accéder au contexte de servlet, peut également accéder au contexte d'application Spring grâce à une méthode utilitaire.

Explications

Supposons que nous développons une application web qui permet aux utilisateurs de déterminer la distance entre deux villes, en kilomètres. Nous définissons tout d'abord l'interface de service suivante :

```
package com.apress.springrecipes.city;  
  
public interface CityService {  
  
    public double findDistance(String srcCity, String destCity);  
}
```

Pour des questions de simplicité, l'implémentation de cette interface enregistre les données de distance dans une table d'association Java. Les clés de cette table sont les villes de départ, tandis que les valeurs sont des tables d'associations imbriquées qui contiennent des villes d'arrivée et la distance qui les sépare de la ville de départ.

```
package com.apress.springrecipes.city;
...
public class CityServiceImpl implements CityService {

    private Map<String, Map<String, Double>> distanceMap;

    public void setDistanceMap(Map<String, Map<String, Double>> distanceMap) {
        this.distanceMap = distanceMap;
    }

    public double findDistance(String srcCity, String destCity) {
        Map<String, Double> destinationMap = distanceMap.get(srcCity);
        if (destinationMap == null) {
            throw new IllegalArgumentException("Ville de départ inconnue");
        }
        Double distance = destinationMap.get(destCity)
        if (distance == null) {
            throw new IllegalArgumentException("Ville d'arrivée inconnue");
        }
        return distance;
    }
}
```

Nous créons ensuite la structure de répertoires suivante pour l'application web. Puisqu'elle doit accéder au conteneur Spring IoC, nous devons placer deux fichiers JAR requis par Spring dans le répertoire WEB-INF/lib.

```
city/
WEB-INF/
    classes/
    lib/
        commons-logging.jar
        spring.jar
jsp/
    distance.jsp
applicationContext.xml
web.xml
```

Dans le fichier de configuration des beans de Spring, nous figeons quelques données de distance pour plusieurs villes en utilisant l'élément <map>. Nous nommons ce fichier applicationContext.xml et le plaçons à la racine de WEB-INF.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="cityService"
        class="com.apress.springrecipes.city.CityServiceImpl">
```

```

<property name="distanceMap">
    <map>
        <entry key="Paris">
            <map>
                <entry key="New York" value="5842" />
                <entry key="Pékin" value="8230" />
            </map>
        </entry>
    </map>
</property>
</bean>
</beans>

```

Dans le descripteur de déploiement web, `web.xml`, nous enregistrons `ContextLoaderListener`, l'écouteur de servlet fourni par Spring, pour charger au démarrage le contexte d'application de Spring dans le contexte de servlet. Cet écouteur examine le paramètre de contexte `contextConfigLocation` pour connaître l'emplacement du fichier de configuration des beans. Nous pouvons indiquer plusieurs fichiers de configuration en les séparant par des virgules ou des espaces.

```

<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

    <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>

```

Par défaut, cet écouteur recherche le fichier de configuration des beans exactement à l'endroit que nous avons indiqué, c'est-à-dire `/WEB-INF/applicationContext.xml`. Par conséquent, nous pouvons omettre ce paramètre de contexte.

Pour que les utilisateurs puissent obtenir la distance entre deux villes, nous créons un fichier JSP qui contient un formulaire. Nous le nommons `distance.jsp` et le plaçons dans le répertoire `WEB-INF/jsp` de manière à empêcher tout accès direct à son contenu. Deux champs de ce formulaire permettent aux utilisateurs de saisir les villes de départ et d'arrivée. Il contient également une table qui affiche la distance.

```

<html>
<head>
<title>Distance entre deux villes</title>
</head>

```

```
<body>
<form method="POST">
<table>
<tr>
    <td>Ville de départ</td>
    <td><input type="text" name="srcCity" value="${param.srcCity}" /></td>
</tr>
<tr>
    <td>Ville d'arrivée</td>
    <td><input type="text" name="destCity" value="${param.destCity}" /></td>
</tr>
<tr>
    <td>Distance</td>
    <td>${distance}</td>
</tr>
<tr>
    <td colspan="2"><input type="submit" value="Chercher" /></td>
</tr>
</table>
</form>
</body>
</html>
```

Nous avons besoin d'une servlet pour traiter les demandes de distance. Lorsqu'on accède à cette servlet au travers d'une méthode HTTP GET, elle affiche simplement le formulaire. Par la suite, lors de la soumission du formulaire à l'aide de la méthode POST, la servlet recherche la distance entre les deux villes saisies et l'affiche dans le formulaire.

INFO

Pour développer des applications web qui utilisent l'API des servlets, vous devez inclure le fichier `servlet-api.jar` (situé dans le répertoire `lib/j2ee` de l'installation de Spring) dans le chemin d'accès aux classes.

```
package com.apress.springrecipes.city.servlet;
...
import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.web.context.WebApplicationContext;
import org.springframework.web.context.support.WebApplicationContextUtils;

public class DistanceServlet extends HttpServlet {

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws ServletException, IOException {
        forward(request, response);
    }
}
```

```

protected void doPost(HttpServletRequest request,
                      HttpServletResponse response)
                      throws ServletException, IOException {
    String srcCity = request.getParameter("srcCity");
    String destCity = request.getParameter("destCity");

    WebApplicationContext context =
        WebApplicationContextUtils.getRequiredWebApplicationContext(
            getServletContext());
    CityService cityService = (CityService) context.getBean("cityService");
    double distance = cityService.findDistance(srcCity, destCity);
    request.setAttribute("distance", distance);
    forward(request, response);
}

private void forward(HttpServletRequest request,
                     HttpServletResponse response)
                     throws ServletException, IOException {
    RequestDispatcher dispatcher =
        request.getRequestDispatcher("WEB-INF/jsp/distance.jsp");
    dispatcher.forward(request, response);
}
}

```

Pour déterminer les distances, la servlet doit accéder au bean `cityService` déclaré dans le conteneur Spring IoC. Puisque le contexte d'application Spring est enregistré dans le contexte de servlet, nous pouvons l'obtenir en invoquant la méthode `WebApplicationContextUtils.getRequiredWebApplicationContext()`, qui attend le contexte de servlet en argument.

Enfin, nous ajoutons la déclaration de cette servlet dans `web.xml` et l'associons au motif d'URL `/distance`.

```

<web-app ...>
    ...
    <servlet>
        <servlet-name>distance</servlet-name>
        <servlet-class>
            com.apress.springrecipes.city.servlet.DistanceServlet
        </servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>distance</servlet-name>
        <url-pattern>/distance</url-pattern>
    </servlet-mapping>
</web-app>

```

Nous pouvons à présent déployer cette application web dans un conteneur web, comme Apache Tomcat 6.0. Par défaut, Tomcat écoute sur le port 8080. Par conséquent, si nous déployons notre application dans le chemin de contexte `city`, nous pouvons y accéder au travers de l'URL suivante :

`http://localhost:8080/city/distance`

11.2 Intégrer Spring à Struts 1.x

Problème

Nous souhaitons accéder aux beans déclarés dans le conteneur Spring IoC à partir d'une application web développée avec Apache Struts 1.x.

Solution

Une application Struts est en mesure de charger un contexte d'application Spring en enregistrant l'écouteur de servlet `ContextLoaderListener` et en y accédant à partir du contexte de servlet, comme le fait une application web générique. Cependant, Spring propose de meilleures solutions propres à Struts pour accéder à son contexte d'application.

Tout d'abord, Spring nous permet de charger un contexte d'application en enregistrant un plug-in Struts dans le fichier de configuration de Struts. Ce contexte d'application considère automatiquement le contexte d'application chargé par l'écouteur de servlet comme son parent et peut donc faire référence aux beans qui y sont déclarés.

Ensuite, Spring fournit la classe `ActionSupport`, qui dérive de la classe de base `Action` dont la méthode `getWebApplicationContext()` nous permet d'accéder à un contexte d'application Spring.

Enfin, il est possible d'injecter des beans Spring dans des actions Struts en utilisant l'injection de dépendance. Pour cela, il faut qu'elles soient déclarées dans le contexte d'application Spring et indiquer à Struts de les obtenir à partir de Spring.

Explications

Nous utilisons à présent Apache Struts pour implémenter notre application web de recherche des distances entre des villes. Tout d'abord, nous créons la structure de répertoires suivante pour l'application.

INFO

Pour une application web développée avec Struts 1.2, vous devez copier les fichiers `struts.jar` (situé dans le répertoire `lib/struts` de l'installation de Spring), `commons-beanutils.jar`, `commons-digester.jar` et `commons-logging.jar` (situés dans `lib/jakarta-commons`) dans le répertoire `WEB-INF/lib`. Si vous souhaitez bénéficier de la prise en charge de Struts par Spring, vous devez également copier le fichier `spring-webmvc-struts.jar` (situé dans `dist/modules`).

```
city/
  WEB-INF/
    classes/
      lib/
        commons-beanutils.jar
        commons-digester.jar
        commons-logging.jar
        spring.jar
        spring-webmvc-struts.jar
        struts.jar
    jsp/
      distance.jsp
    applicationContext.xml
    struts-config.xml
    web.xml
```

Dans le descripteur de déploiement web d'une application Struts, `web.xml`, nous devons enregistrer la servlet Struts `ActionServlet` pour le traitement des requêtes web. Nous pouvons associer cette servlet au motif d'URL `*.do`.

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

  <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>
```

Charger un contexte d'application Spring dans une application Struts

Il existe deux manières de charger un contexte d'application Spring dans une application Struts. La première consiste à enregistrer l'écouteur de servlet `ContextLoaderListener` dans `web.xml`. Puisqu'il considère par défaut que les beans Spring sont configurés dans `/WEB-INF/applicationContext.xml`, nous n'avons pas à préciser explicitement cet emplacement.

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

La seconde solution consiste à enregistrer le plug-in Struts ContextLoaderPlugin dans le fichier de configuration de Struts, struts-config.xml. Par défaut, ce plug-in charge le fichier de configuration des beans dont le nom est indiqué par l'instance de ActionServlet enregistrée dans web.xml et en lui ajoutant le suffixe -servlet.xml (action-servlet.xml dans notre cas). Pour charger un autre fichier de configuration des beans, nous précisons son nom dans la propriété contextConfigLocation.

```
<!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>
  ...
  <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn">
    <set-property property="contextConfigLocation"
      value="/WEB-INF/applicationContext.xml" />
  </plug-in>
</struts-config>
```

Si les deux configurations sont présentes, le contexte d'application Spring chargé par le plug-in Struts considère automatiquement le contexte d'application chargé par l'écouteur de servlet comme son parent. De manière générale, les services métier doivent être déclarés dans le contexte d'application chargé par l'écouteur de servlet, tandis que les composants web doivent être séparés dans un autre contexte d'application chargé par le plug-in Struts. Par conséquent, pour le moment nous omettons la configuration du plug-in Struts.

Accéder à un contexte d'application Spring depuis des actions Struts

Struts facilite la liaison des champs d'un formulaire HTML aux propriétés des beans de formulaire au moment de la soumission du formulaire. Tout d'abord, nous créons une classe de formulaire qui dérive de ActionForm et qui comprend deux propriétés pour les villes de départ et d'arrivée.

```
package com.apress.springrecipes.city.struts;

import org.apache.struts.action.ActionForm;

public class DistanceForm extends ActionForm {

    private String srcCity;
    private String destCity;

    // Accesseurs et mutateurs.
    ...
}
```

Ensuite, nous créons un fichier JSP dans lequel un formulaire permet aux utilisateurs de saisir une ville de départ et une ville d'arrivée. Nous devons définir ce formulaire et ses

champs en utilisant la bibliothèque de balises fournie par Struts. En effet, nous voulons que les champs soient liés automatiquement aux propriétés du bean de formulaire. Nous nommons ce fichier JSP `distance.jsp` et le plaçons dans le répertoire `WEB-INF/jsp` de manière à interdire tout accès direct.

```
<%@ taglib prefix="html" uri="http://struts.apache.org/tags-html" %>

<html>
<head>
<title>Distance entre deux villes</title>
</head>

<body>
<html:form method="POST" action="/distance.do">
<table>
<tr>
    <td>Ville de départ</td>
    <td><html:text property="srcCity" /></td>
</tr>
<tr>
    <td>Ville d'arrivée</td>
    <td><html:text property="destCity" /></td>
</tr>
<tr>
    <td>Distance</td>
    <td>${distance}</td>
</tr>
<tr>
    <td colspan="2"><input type="submit" value="Chercher" /></td>
</tr>
</table>
</html:form>
</body>
</html>
```

Dans Struts, chaque requête web est traitée par une action qui étend la classe `Action`. Nos actions Struts doivent parfois accéder à des beans Spring. Pour accéder au contexte d'application chargé par l'écouteur de servlet `ContextLoaderListener`, nous pouvons invoquer la méthode statique `WebApplicationContextUtils.getRequiredWebApplicationContext()`.

Cependant, il existe une meilleure solution pour accéder à un contexte d'application Spring depuis une action Struts : étendre la classe `ActionSupport`. Cette classe dérive de la classe `Action`, qui offre une méthode pratique, `getWebApplicationContext()`, pour accéder à un contexte d'application Spring. Cette méthode tente tout d'abord de retrouver le contexte d'application chargé par `ContextLoaderPlugin`. S'il n'existe pas, elle tente de retourner son parent, c'est-à-dire le contexte d'application chargé par `ContextLoaderListener`.

```
package com.apress.springrecipes.city.struts;
...
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;
import org.springframework.web.struts.ActionSupport;

public class DistanceAction extends ActionSupport {

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) {
        if (request.getMethod().equals("POST")) {
            DistanceForm distanceForm = (DistanceForm) form;
            String srcCity = distanceForm.getSrcCity();
            String destCity = distanceForm.getDestCity();

            CityService cityService =
                (CityService) getWebApplicationContext().getBean("cityService");
            double distance = cityService.findDistance(srcCity, destCity);
            request.setAttribute("distance", distance);
        }
        return mapping.findForward("success");
    }
}
```

Dans le fichier de configuration de Struts, struts-config.xml, nous déclarons les beans de formulaire, ainsi que les actions et leur correspondance pour notre application.

```
<struts-config>
    <form-beans>
        <form-bean name="distanceForm"
            type="com.apress.springrecipes.city.struts.DistanceForm" />
    </form-beans>

    <action-mappings>
        <action path="/distance"
            type="com.apress.springrecipes.city.struts.DistanceAction"
            name="distanceForm" validate="false">
            <forward name="success"
                path="/WEB-INF/jsp/distance.jsp" />
        </action>
    </action-mappings>
</struts-config>
```

Nous pouvons alors déployer cette application dans notre conteneur web et y accéder par l'intermédiaire de l'URL suivante :

<http://localhost:8080/city/distance.do>

Déclarer des actions Struts dans le fichier de configuration des beans de Spring

Outre la recherche active de beans Spring depuis une action Struts *via* un contexte d'application Spring, nous pouvons utiliser l'injection de dépendance pour affecter des beans Spring à une action Struts. Dans ce cas, l'action Struts n'est plus obligée d'éten-dre la classe `ActionSupport`, mais peut simplement dériver de la classe `Action`.

```
package com.apress.springrecipes.city.struts;
...
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.apache.struts.action.Action;
import org.apache.struts.action.ActionForm;
import org.apache.struts.action.ActionForward;
import org.apache.struts.action.ActionMapping;

public class DistanceAction extends Action {
    private CityService cityService;

    public void setCityService(CityService cityService) {
        this.cityService = cityService;
    }

    public ActionForward execute(ActionMapping mapping, ActionForm form,
        HttpServletRequest request, HttpServletResponse response) {
        if (request.getMethod().equals("POST")) {
            ...
            double distance = cityService.findDistance(srcCity, destCity);
            request.setAttribute("distance", distance);
        }
        return mapping.findForward("success");
    }
}
```

Toutefois, cette action doit être sous le contrôle de Spring pour que ses dépendances puissent être injectées. Nous pouvons choisir de la déclarer dans le fichier `applicationContext.xml` ou un autre fichier de configuration des beans chargé par `ContextLoaderPlugin`. Afin de mieux séparer les services métier et les composants web, il est préférable de la déclarer dans `action-servlet.xml` à la racine du répertoire `WEB-INF`, un fichier qui sera chargé par défaut par `ContextLoaderPlugin` puisque l'instance de `ActionServlet` se nomme `action`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean name="/distance"
        class="com.apress.springrecipes.city.struts.DistanceAction">
        <property name="cityService" ref="cityService" />
    </bean>
</beans>
```

Le nom de bean de cette action doit être identique à son chemin d'action dans struts-config.xml. Puisque le caractère / n'est pas autorisé dans l'attribut id d'un élément <bean>, nous devons employer l'attribut name à la place. Dans ce fichier de configuration, nous pouvons faire référence aux beans déclarés dans le contexte d'application parent, qui est chargé par ContextLoaderListener.

Nous enregistrons ContextLoaderPlugin dans struts-config.xml pour charger le fichier de configuration des beans précédent.

```
<struts-config>
    ...
    <action-mappings>
        <action path="/distance"
            name="distanceForm" validate="false">
            <forward name="success"
                path="/WEB-INF/jsp/distance.jsp" />
        </action>
    </action-mappings>

    <controller processorClass="org.springframework.web.struts.<span style="color: red;">*
        DelegatingRequestProcessor" />

    <plug-in className="org.springframework.web.struts.ContextLoaderPlugIn" />
</struts-config>
```

Nous devons également enregistrer le processeur de requêtes Struts, DelegatingRequestProcessor, pour indiquer à Struts de rechercher ses actions à partir du contexte d'application Spring en utilisant une correspondance entre le chemin d'action et le nom de bean. Lorsque ce processeur est enregistré, nous n'avons plus besoin de préciser l'attribut type d'une action.

Si un autre processeur de requêtes est déjà présent, nous ne pouvons pas enregistrer DelegatingRequestProcessor. Dans ce cas, nous précisons que l'action est de type DelegatingActionProxy afin d'obtenir le même effet.

```
<action path="/distance"
    type="org.springframework.web.struts.DelegatingActionProxy"
    name="distanceForm" validate="false">
    <forward name="success"
        path="/WEB-INF/jsp/distance.jsp" />
</action>
```

11.3 Intégrer Spring à JSF

Problème

Nous souhaitons accéder aux beans déclarés dans le conteneur Spring IoC à partir d'une application web développée avec JSF.

Solution

Une application JSF est capable d'accéder au contexte d'application Spring comme le fait une application web générique, c'est-à-dire en enregistrant l'écouteur de servlet `ContextLoaderListener` et en y accédant à partir du contexte de servlet. Toutefois, en raison de la similitude entre les modèles de beans de Spring et de JSF, il est très facile d'intégrer ces deux frameworks en enregistrant le résolveur de variables JSF fourni par Spring, `DelegatingVariableResolver`. Il permet de résoudre des variables JSF en beans Spring. Nous pouvons même regrouper des beans JSF gérés et nos beans Spring en les déclarant dans un fichier de configuration des beans de Spring.

Explications

Supposons que nous utilisions JSP pour implémenter notre application web de recherche de la distance entre deux villes. Tout d'abord, nous créons la structure de répertoires suivante pour cette application.

INFO

Avant de commencer à développer une application web avec JSF, vous avez besoin d'une bibliothèque d'implémentation de JSF. Vous pouvez télécharger l'implémentation de référence de JSF (JSF-RI, *JSF Reference Implementation*) en version 1.2 à l'adresse <https://java-serverfaces.dev.java.net/>. Ensuite, extrayez le contenu du fichier ZIP dans le répertoire de votre choix et copiez les fichiers `jsf-api.jar` et `jsf-impl.jar` (situés dans le répertoire `lib` de l'installation de JSF-RI) dans le répertoire `WEB-INF/lib`. Puisque JSF-RI dépend de JSTL, vous devez également copier le fichier `jstl.jar` (situé dans le répertoire `lib/j2ee` de l'installation de Spring).

```
city/
  WEB-INF/
    classes/
    lib/
      commons-logging.jar
      jsf-api.jar
      jsf-impl.jar
      jstl.jar
      spring.jar
    applicationContext.xml
    faces-config.xml
    web.xml
  distance.jsp
```

Dans le descripteur de déploiement web d'une application JSF, `web.xml`, nous enregistrons la servlet JSF `FacesServlet` pour le traitement des requêtes web et nous l'associons au motif d'URL `*.faces`. Pour charger un contexte d'application Spring au démarrage, nous devons également enregistrer l'écouteur de servlet `ContextLoaderListener`.

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
        http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

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

    <servlet>
        <servlet-name>faces</servlet-name>
        <servlet-class>javax.faces.webapp.FacesServlet</servlet-class>
    </servlet>

    <servlet-mapping>
        <servlet-name>faces</servlet-name>
        <url-pattern>*.faces</url-pattern>
    </servlet-mapping>
</web-app>
```

Dans JSF, l'idée de base est de séparer la logique de présentation de l'interface utilisateur en l'encapsulant dans un ou plusieurs beans JSF générés. Pour notre fonction de recherche de la distance, nous créons la classe `DistanceBean` pour un bean JSF géré.

```
package com.apress.springrecipes.city.jsf;
...
public class DistanceBean {

    private String srcCity;
    private String destCity;
    private double distance;
    private CityService cityService;

    public String getSrcCity() {
        return srcCity;
    }

    public String getDestCity() {
        return destCity;
    }

    public double getDistance() {
        return distance;
    }
}
```

```
public void setSrcCity(String srcCity) {
    this.srcCity = srcCity;
}

public void setDestCity(String destCity) {
    this.destCity = destCity;
}

public void setCityService(CityService cityService) {
    this.cityService = cityService;
}

public void find() {
    distance = cityService.findDistance(srcCity, destCity);
}
}
```

Ce bean définit quatre propriétés. Puisque la page doit afficher les propriétés `srcCity`, `destCity` et `distance`, nous définissons un accesseur pour chacune d'elles. Les utilisateurs ne peuvent modifier que les propriétés `srcCity` et `destCity`, pour lesquelles nous définissons donc également un mutateur. Le bean `CityService` est injecté par mutateur. Lorsque la méthode `find()` est appelée sur ce bean, elle invoque le service métier de manière à obtenir la distance qui sépare les deux villes et stocke ce résultat dans la propriété `distance` en vue de son affichage ultérieur.

Nous créons ensuite le fichier `distance.jsp` à la racine du contexte d'application web. Nous devons le placer à cet endroit car, lorsque `FacesServlet` reçoit une requête, elle l'associe à un fichier JSP du même nom. Par exemple, si l'URL est `/distance.faces`, `FacesServlet` charge le fichier `/distance.jsp`.

```
<%@ taglib prefix="f" uri="http://java.sun.com/jsf/core" %>
<%@ taglib prefix="h" uri="http://java.sun.com/jsf/html" %>

<html>
<head>
<title>Distance entre deux villes</title>
</head>

<body>
<f:view>
<h:form>
    <h:panelGrid columns="2">
        <h:outputLabel for="srcCity">Ville de départ</h:outputLabel>
        <h:inputText id="srcCity" value="#{distanceBean.srcCity}" />
        <h:outputLabel for="destCity">Ville d'arrivée</h:outputLabel>
        <h:inputText id="destCity" value="#{distanceBean.destCity}" />
        <h:outputLabel>Distance</h:outputLabel>
        <h:outputText value="#{distanceBean.distance}" />
        <h:commandButton value="Chercher" action="#{distanceBean.find}" />
    </h:panelGrid>
</h:form>
</f:view>
</body>
</html>
```

Ce fichier JSP contient un composant <h:form> pour que les utilisateurs puissent saisir les villes de départ et d'arrivée. Les deux champs correspondants sont définis avec des composants <h:inputText> liés aux propriétés du bean JSF géré. La distance trouvée est définie avec un composant <h:outputText> car sa valeur est en lecture seule. Enfin, nous ajoutons un composant <h:commandButton> dont l'action est déclenchée côté serveur lorsque l'utilisateur clique sur le bouton.

Résoudre des beans Spring dans JSF

Dans le fichier de configuration de JSF, faces-config.xml, placé à la racine de WEB-INF, nous configurons les règles de navigation et les beans JSF gérés. Pour cette application simple, avec un seul écran, il n'y a aucune règle de navigation. Nous configurons simplement le bean DistanceBean.

```
<faces-config xmlns="http://java.sun.com/xml/ns/javaee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/javaee/web-facesconfig_1_2.xsd"
    version="1.2">

    <application>
        <variable-resolver>
            org.springframework.web.jsf.DelegatingVariableResolver
        </variable-resolver>
    </application>

    <managed-bean>
        <managed-bean-name>distanceBean</managed-bean-name>
        <managed-bean-class>
            com.apress.springrecipes.city.jsf.DistanceBean
        </managed-bean-class>
        <managed-bean-scope>request</managed-bean-scope>
        <managed-property>
            <property-name>cityService</property-name>
            <value>#{cityService}</value>
        </managed-property>
    </managed-bean>
</faces-config>
```

La portée de DistanceBean est fixée à request. Autrement dit, une nouvelle instance de bean est créée à chaque requête. En enregistrant le résolveur de variables DelegatingVariableResolver, nous pouvons facilement faire référence à un bean déclaré dans le contexte d'application Spring en utilisant une variable JSF de la forme #{nomDuBean}. Ce résolveur de variables tente tout d'abord de convertir les variables en utilisant le résolveur de variables JSF d'origine. S'il échoue, il recherche dans le contexte d'application Spring un bean de même nom.

Nous pouvons à présent déployer cette application dans notre conteneur web et y accéder par l'intermédiaire de l'URL suivante :

<http://localhost:8080/city/distance.faces>

Déclarer des beans JSF gérés dans un fichier de configuration des beans de Spring

En enregistrant DelegatingVariableResolver, nous pouvons faire référence aux beans déclarés dans Spring à partir des beans JSF gérés. Toutefois, ils sont alors gérés par deux conteneurs différents : celui de JSF et celui de Spring. Une meilleure solution consiste à les placer sous le contrôle du conteneur Spring IoC. Nous retirons la déclaration du bean géré du fichier de configuration de JSF et ajoutons la déclaration de bean Spring suivante dans applicationContext.xml :

```
<bean id="distanceBean"
      class="com.apress.springrecipes.city.jsf.DistanceBean"
      scope="request">
    <property name="cityService" ref="cityService" />
</bean>
```

Pour activer la portée de bean request dans un contexte d'application Spring, nous devons enregistrer RequestContextListener dans le descripteur de déploiement web.

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

  <listener>
    <listener-class>
      org.springframework.web.context.request.RequestContextListener
    </listener-class>
  </listener>
  ...
</web-app>
```

11.4 Intégrer Spring à DWR

Problème

Nous souhaitons accéder aux beans déclarés dans le conteneur Spring IoC à partir d'une application web développée avec DWR.

Solution

Au travers du créateur spring, DWR nous permet d'exposer des beans Spring en vue de leur invocation à distance. De plus, DWR 2.0 fournit un schéma XML pour Spring qui nous permet de configurer DWR dans le fichier de configuration des beans de Spring. Pour configurer des beans accessibles à distance, il nous suffit d'inclure la balise `<dwr:remote>`, ce qui n'implique aucunement le fichier de configuration de DWR.

Explications

Supposons que nous utilisions DWR pour implémenter notre application web de recherche de la distance entre deux villes et la rendre compatible avec Ajax. Tout d'abord, nous créons la structure de répertoires suivante pour cette application.

INFO

Pour développer une application web avec DWR, vous devez télécharger dwr.jar à partir du site <http://getahead.org/dwr/> et copier ce fichier dans le répertoire WEB-INF/lib.

```
city/
  WEB-INF/
    classes/
      lib/
        commons-logging.jar
        dwr.jar
        spring.jar
      applicationContext.xml
      dwr.xml
      web.xml
    distance.html
```

Dans le descripteur de déploiement web d'une application DWR, web.xml, nous devons enregistrer la servlet DWR DwrServlet pour le traitement des requêtes web Ajax. Nous associons cette servlet au motif d'URL /dwr/*. Pour charger un contexte d'application Spring au démarrage, nous devons également enregistrer l'écouteur de servlet Context-LoaderListener.

```
<web-app version="2.4" xmlns="http://java.sun.com/xml/ns/j2ee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://java.sun.com/xml/ns/j2ee/web-app_2_4.xsd">

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

  <servlet>
    <servlet-name>dwr</servlet-name>
    <servlet-class>
      org.directwebremoting.servlet.DwrServlet
    </servlet-class>
  </servlet>

  <servlet-mapping>
    <servlet-name>dwr</servlet-name>
    <url-pattern>/dwr/*</url-pattern>
  </servlet-mapping>
</web-app>
```

Toute application DWR a besoin d'un fichier de configuration pour définir les objets qui sont exposés à l'invocation à distance depuis du code JavaScript. Par défaut, Dwr-Servlet charge le fichier `dwr.xml` à partir de la racine du répertoire WEB-INF.

```
<!DOCTYPE dwr PUBLIC
  "-//GetAhead Limited//DTD Direct Web Remoting 2.0//EN"
  "http://getahead.org/dwr/dwr20.dtd">

<dwr>
  <allow>
    <create creator="new" javascript="CityService">
      <param name="class"
            value="com.apress.springrecipes.city.CityServiceImpl" />
      <include method="findDistance" />
    </create>
  </allow>
</dwr>
```

Ce fichier de configuration de DWR expose la classe `CityServiceImpl` à une invocation à distance depuis du code JavaScript. Le source de cette classe est généré dynamiquement dans `CityService.js`. Le créateur `new` de DWR est le plus employé. Il crée une nouvelle instance de la classe à chaque invocation. Nous indiquons que seule la méthode `findDistance()` de cette classe peut être invoquée à distance.

Exposer des beans Spring à une invocation à distance

Le créateur `new` de DWR crée une nouvelle instance d'objet à chaque invocation. Si nous voulons qu'un bean du contexte d'application Spring puisse être invoqué à distance, nous pouvons employer le créateur `spring` en précisant le nom de ce bean.

```
<dwr>
  <allow>
    <create creator="spring" javascript="CityService">
      <param name="beanName" value="cityService" />
      <include method="findDistance" />
    </create>
  </allow>
</dwr>
```

Nous écrivons ensuite la page web qui permet aux utilisateurs de déterminer la distance entre deux villes. Lorsqu'on utilise Ajax, il est inutile d'actualiser la page web comme dans le cas d'une page web classique. Par conséquent, nous créons une page HTML statique, par exemple `distance.html`, située à la racine du contexte d'application web.

```
<html>
<head>
<title>Distance entre deux villes</title>
<script src='dwr/interface/CityService.js'></script>
<script src='dwr/engine.js'></script>
<script src='dwr/util.js'></script>
<script type="text/javascript">
```

```
function find() {
    var srcCity = dwr.util.getValue("srcCity");
    var destCity = dwr.util.getValue("destCity");
    CityService.findDistance(srcCity, destCity, function(data) {
        dwr.util.setValue("distance", data);
    });
}
</script>
</head>

<body>
<form>
<table>
<tr>
    <td>Ville de départ</td>
    <td><input type="text" id="srcCity" /></td>
</tr>
<tr>
    <td>Ville d'arrivée</td>
    <td><input type="text" id="destCity" /></td>
</tr>
<tr>
    <td>Distance</td>
    <td><span id="distance" /></td>
</tr>
<tr>
    <td colspan="2">
        <input type="button" value="Chercher" onclick="find()" />
    </td>
</tr>
</table>
</form>
</body>
</html>
```

Lorsque l'utilisateur clique sur le bouton Chercher, la fonction JavaScript `find()` est invoquée. Elle effectue une requête Ajax sur la méthode `CityService.findDistance()` en passant les valeurs des champs correspondants aux villes de départ et d'arrivée. Lorsque la réponse Ajax arrive, elle affiche la valeur reçue dans l'élément `distance`. Pour que cela fonctionne, nous devons inclure les bibliothèques JavaScript générées dynamiquement par DWR.

Nous pouvons à présent déployer cette application dans notre conteneur web et y accéder par l'intermédiaire de l'URL suivante :

<http://localhost:8080/city/distance.html>

Configurer DWR dans le fichier de configuration des beans de Spring

La configuration de DWR 2.0 peut se faire directement dans le fichier de configuration des beans de Spring. Toutefois, pour que cela soit possible, nous devons remplacer le `DwrServlet` enregistré précédemment par `DwrSpringServlet` dans le descripteur de déploiement web.

```
<web-app ...>
...
<servlet>
    <servlet-name>dwr</servlet-name>
    <servlet-class>
        org.directwebremoting.spring.DwrSpringServlet
    </servlet-class>
</servlet>
</web-app>
```

Dans le fichier de configuration des beans de Spring `applicationContext.xml`, nous configurons DWR en utilisant les éléments XML définis par le schéma de DWR. Tout d'abord, nous ajoutons l'élément `<dwr:configuration>` pour activer DWR dans Spring. Ensuite, pour chaque bean que nous voulons accessible à distance, nous ajoutons un élément `<dwr:remote>` avec les informations de configuration équivalentes à celles de `dwr.xml`.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:dwr="http://www.directwebremoting.org/schema/spring-dwr"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd
                           http://www.directwebremoting.org/schema/spring-dwr
                           http://www.directwebremoting.org/schema/spring-dwr-2.0.xsd">

    <dwr:configuration />

    <bean id="cityService"
          class="com.apress.springrecipes.city.CityServiceImpl">
        <dwr:remote javascript="CityService">
            <dwr:include method="findDistance" />
        </dwr:remote>
        ...
    </bean>
</beans>
```

Nous pouvons alors supprimer `dwr.xml`, car les informations de configuration qu'il contient ont été transférées dans le fichier de configuration des beans de Spring.

11.5 En résumé

Dans ce chapitre, nous avons vu comment intégrer Spring dans des applications web développées avec le couple servlets/JSP et des frameworks d'applications web répandus, comme Struts, JSF et DWR. Cette intégration consiste principalement à permettre l'accès aux beans du conteneur Spring IoC depuis ces autres frameworks.

Dans une application web générique, quel que soit le framework utilisé, nous pouvons enregistrer l'écouteur de servlet `ContextLoaderListener` fourni par Spring pour charger un contexte d'application Spring dans le contexte de servlet de l'application web.

Par la suite, une servlet, ou n'importe quel objet capable d'accéder au contexte de servlet, peut également accéder au contexte d'application Spring grâce à une méthode utilitaire.

Dans une application web développée avec Struts, nous pouvons charger un contexte d'application Spring en enregistrant un écouteur de servlet, mais également en enregistrant un plug-in Struts. Ce contexte d'application considère automatiquement le contexte d'application chargé par l'écouteur de servlet comme son parent. Spring fournit la classe `ActionSupport`, dont une méthode pratique nous permet d'accéder à un contexte d'application Spring. Nous pouvons également déclarer des actions Struts dans le contexte d'application Spring pour que des beans Spring soient injectés.

Pour JSF, nous pouvons enregistrer le résolveur de variables `DelegatingVariableResolver`, qui permet de résoudre des variables JSF en beans Spring. Nous pouvons même regrouper des beans JSF gérés et nos beans Spring en les déclarant dans un fichier de configuration des beans de Spring.

Par l'intermédiaire de son créateur `spring`, DWR nous permet d'exposer des beans Spring en vue de leur invocation à distance. DWR 2.0 fournit un schéma XML pour Spring afin de configurer DWR dans le fichier de configuration des beans de Spring.

Le chapitre suivant détaille les fonctionnalités Spring de prise en charge des tests et la manière de tester des applications fondées sur Spring.

Prise en charge des tests

Au sommaire de ce chapitre

- ✓ Créer des tests avec JUnit et TestNG
- ✓ Créer des tests unitaires et des tests d'intégration
- ✓ Effectuer des tests unitaires sur des contrôleurs Spring MVC
- ✓ Gérer des contextes d'application dans les tests d'intégration
- ✓ Injecter des fixtures de test dans des tests d'intégration
- ✓ Gérer des transactions dans les tests d'intégration
- ✓ Accéder à une base de données dans des tests d'intégration
- ✓ Utiliser les annotations communes de Spring pour les tests
- ✓ En résumé

Ce chapitre présente les techniques de base pour le test des applications Java, ainsi que les fonctionnalités Spring de prise en charge des tests. Ces fonctionnalités facilitent les tests et conduisent à une meilleure conception des applications. De manière générale, les applications développées avec le framework Spring et l'injection de dépendance sont faciles à tester.

Les tests représentent une activité essentielle du développement des logiciels car ils permettent de garantir leur qualité. Il existe plusieurs sortes de tests, notamment les tests unitaires, les tests d'intégration, les tests fonctionnels, les tests système, les tests de performances et les tests de recette. Spring se focalise sur les tests unitaires et les tests d'intégration, mais ses fonctionnalités peuvent également constituer une aide pour les autres types de tests. Les tests sont manuels ou automatiques. Toutefois, puisque les tests automatiques peuvent être exécutés de manière répétée et continue à différentes phases du processus de développement, ils sont fortement recommandés, en particulier dans le cadre du développement agile. Spring est un framework agile parfaitement adapté à cette forme de développement.

Il existe plusieurs frameworks de test pour la plate-forme Java. JUnit et TestNG sont aujourd'hui les plus répandus. JUnit existe depuis longtemps et ses utilisateurs sont nombreux au sein de la communauté Java. Les améliorations apportées par sa version 4.0 sont nombreuses, notamment la prise en charge des annotations. TestNG est un autre framework de test Java répandu, qui se fonde énormément sur les annotations. Par rapport à JUnit, TestNG propose de puissantes fonctions supplémentaires, comme le regroupement de tests, les méthodes de test dépendantes et les tests orientés données.

Avant la version 2.5, la prise en charge des tests dans Spring était spécifique à JUnit 3.8. Pour bénéficier de cette prise en charge ancienne de JUnit 3.8, nous devons étendre les classes de base fournies pour les tests. Spring 2.5 a renouvelé ses fonctionnalités de prise en charge des tests au travers du framework *Spring TestContext*, qui exige Java 1.5 ou une version ultérieure. Grâce aux concepts suivants, ce framework rend abstrait le framework de test sous-jacent :

- **Contexte de test.** Il représente le contexte d'exécution des tests, notamment le contexte d'application, ainsi que la classe, l'instance, la méthode et l'exception de test.
- **Gestionnaire de contexte de test.** Il gère un contexte de test pour un test et déclenche des écouteurs d'exécution du test à des points d'exécution prédéfinis, notamment lors la préparation d'une instance de test, avant d'exécuter une méthode de test (avant toute méthode d'initialisation propre au framework) et après l'exécution d'une méthode de test (après toute méthode de nettoyage propre au framework).
- **Écouteur d'exécution du test.** Il définit une interface dont l'implémentation nous permet d'écouter les événements d'exécution du test. Le framework TestContext fournit plusieurs écouteurs d'exécution du test pour les fonctions communes, mais nous pouvons créer nos propres écouteurs.

Spring 2.5 fournit des classes TestContext pour JUnit 3.8, JUnit 4.4 et TestNG 5.8, avec des écouteurs d'exécution du test préenregistrés. Nous pouvons simplement étendre ces classes pour utiliser le framework TestContext sans vraiment en connaître les détails.

À la fin de ce chapitre, vous comprendrez les concepts et les techniques de base des tests et aurez acquis une bonne connaissance des frameworks de test Java répandus, JUnit et TestNG. Vous serez également en mesure de créer des tests unitaires et des tests d'intégration à partir de la prise en charge ancienne de JUnit 3.8 et du framework Spring TestContext.

12.1 Créer des tests avec JUnit et TestNG

Problème

Nous souhaitons créer des tests automatiques afin qu'ils puissent être lancés de manière répétée pour garantir la conformité de notre application Java.

Solution

Sur la plate-forme Java, les frameworks de test les plus connus sont JUnit et TestNG. JUnit 4 bénéficie d'améliorations majeures par rapport à JUnit 3.8, qui identifie un cas de test à partir d'une classe de base (`TestCase`) et de la signature des méthodes (celles dont le nom commence par `test`) – cette solution manque cruellement de flexibilité. JUnit 4 nous permet d'annoter les méthodes de test avec `@Test`. Ainsi, n'importe quelle méthode publique peut être exécutée comme un cas de test. TestNG est un autre framework de test puissant qui utilise des annotations, dont `@Test` pour identifier les cas de test.

Explications

Supposons que nous développons un système bancaire. Pour garantir la qualité de ce système, nous devons en tester chaque partie. Tout d'abord, examinons une calculatrice des intérêts, dont voici l'interface :

```
package com.apress.springrecipes.bank;

public interface InterestCalculator {

    public void setRate(double rate);
    public double calculate(double amount, double year);
}
```

À chaque instance de la calculatrice des intérêts, nous devons affecter un taux d'intérêt fixe. Notre implémentation de cette calculatrice utilise une formule de calcul simple.

```
package com.apress.springrecipes.bank;

public class SimpleInterestCalculator implements InterestCalculator {

    private double rate;

    public void setRate(double rate) {
        this.rate = rate;
    }

    public double calculate(double amount, double year) {
        if (amount < 0 || year < 0) {
            throw new IllegalArgumentException(
                "Le montant et la durée doivent être positifs");
        }
        return amount * year * rate;
    }
}
```

Nous allons tester cette calculatrice des intérêts avec les frameworks de test JUnit (versions 3.8 et 4) et TestNG (version 5).

ASTUCE

En général, un test et sa classe cible se trouvent dans le même paquetage, mais les fichiers source des tests sont placés dans un répertoire distinct (par exemple `test`) de celui des fichiers source des autres classes (par exemple `src`).

Tester avec JUnit 3.8

Dans JUnit 3.8, une classe qui contient des cas de test doit dériver de la classe `TestCase` fournie par le framework. Chaque cas de test doit être une méthode publique dont le nom commence par `test`. Un cas de test doit s'exécuter dans un environnement défini, constitué d'un ensemble d'objets appelés *fixtures de test*. Dans JUnit 3.8, nous pouvons initialiser les fixtures de test en redéfinissant la méthode `setUp()` de la classe `TestCase`. Cette méthode est invoquée par JUnit avant chaque exécution d'un cas de test. De même, nous pouvons redéfinir la méthode `tearDown()` pour procéder à des tâches de nettoyage, comme la libération des ressources. Cette méthode est invoquée par JUnit après chaque exécution d'un cas de test. Pour tester notre calculatrice des intérêts, nous écrivons les cas de test JUnit 3.8 suivants.

INFO

Pour compiler et exécuter des cas de test créés pour JUnit 3.8, vous devez inclure le fichier `junit-3.8.2.jar` ou `junit-4.4.jar` (situés dans le répertoire `lib/junit` de l'installation de Spring) dans le chemin d'accès aux classes.

```
package com.apress.springrecipes.bank;

import junit.framework.TestCase;

public class SimpleInterestCalculatorJUnit38Tests extends TestCase {

    private InterestCalculator interestCalculator;

    protected void setUp() throws Exception {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }

    public void testCalculate() {
        double interest = interestCalculator.calculate(10000, 2);
        assertEquals(interest, 1000.0);
    }
}
```

```
public void testIllegalCalculate() {  
    try {  
        interestCalculator.calculate(-10000, 2);  
        fail("Pas d'exception en cas d'argument invalide");  
    } catch (IllegalArgumentException e) {}  
}
```

Outre les cas normaux, un test doit inclure des cas exceptionnels qui sont généralement supposés lancer une exception. Dans JUnit 3.8, pour tester si une méthode lance une exception, nous pouvons placer son invocation dans un bloc `try-catch`. Ensuite, sur la ligne qui suit l’invocation de la méthode dans le bloc `try`, nous devons faire échouer le test si aucune exception n’a été lancée.

ATTENTION

La plupart des IDE Java, y compris Eclipse, disposent d’un outil d’exécution des cas de test JUnit. Lorsque les tests réussissent, vous voyez une barre verte. S’ils échouent, la barre est rouge. En revanche, si vous n’utilisez pas un IDE compatible avec JUnit, vous pouvez exécuter des tests JUnit depuis la ligne de commande. JUnit 3.8 fournit des outils d’exécution graphiques et textuels, mais JUnit 4 ne propose plus d’outil graphique.

Tester avec JUnit 4

Dans JUnit 4, une classe qui contient des cas de test n’a plus besoin de dériver de la classe `TestCase`. Il peut s’agir d’une classe quelconque. Un cas de test n’est rien d’autre qu’une méthode publique marquée avec l’annotation `@Test`. De même, il est nécessaire non plus de redéfinir les méthodes `setUp()` et `tearDown()`, mais simplement d’annoter une méthode publique avec `@Before` ou `@After`. Nous pouvons également annoter une méthode statique publique avec `@BeforeClass` ou `@AfterClass` pour qu’elle soit exécutée avant ou après tous les cas de test de la classe.

Puisque la classe n’étend pas `TestCase`, elle n’hérite pas des méthodes `assert`. Par conséquent, nous devons invoquer directement les méthodes statiques d’assertion déclarées dans la classe `Assert`. Pour simplifier, nous pouvons importer toutes les méthodes d’assertion en utilisant une instruction d’importation statique dans Java 1.5. Nous créons les cas de test JUnit 4 suivants pour notre calculatrice des intérêts.

INFO

Pour compiler et exécuter des cas de test créés pour JUnit 4.4, vous devez inclure le fichier `junit-4.4.jar` (situé dans le répertoire `lib/junit` de l’installation de Spring) dans le chemin d’accès aux classes.

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class SimpleInterestCalculatorJUnit4Tests {

    private InterestCalculator interestCalculator;

    @Before
    public void init() {
        interestCalculator = new SimpleInterestCalculator();
        interestCalculator.setRate(0.05);
    }

    @Test
    public void calculate() {
        double interest = interestCalculator.calculate(10000, 2);
        assertEquals(interest, 1000.0, 0);
    }

    @Test(expected = IllegalArgumentException.class)
    public void illegalCalculate() {
        interestCalculator.calculate(-10000, 2);
    }
}
```

Grâce à une fonction puissante de JUnit 4, nous pouvons indiquer que le lancement d'une exception est attendu dans un cas de test. Il suffit simplement de préciser le type de l'exception dans l'attribut `expected` de l'annotation `@Test`.

Tester avec TestNG

Un test TestNG ressemble fortement à un test JUnit 4, mais il utilise les classes et les annotations définies dans le framework TestNG.

INFO

Pour compiler et exécuter des cas de test créés pour TestNG 5, vous devez inclure le fichier `testng-5.8-jdk15.jar` (situé dans le répertoire `lib/testng` de l'installation de Spring) dans le chemin d'accès aux classes.

```
package com.apress.springrecipes.bank;

import static org.testng.Assert.*;

import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;
```

```
public class SimpleInterestCalculatorTestNG5Tests {  
  
    private InterestCalculator interestCalculator;  
  
    @BeforeMethod  
    public void init() {  
        interestCalculator = new SimpleInterestCalculator();  
        interestCalculator.setRate(0.05);  
    }  
  
    @Test  
    public void calculate() {  
        double interest = interestCalculator.calculate(10000, 2);  
        assertEquals(interest, 1000.0);  
    }  
  
    @Test(expectedExceptions = IllegalArgumentException.class)  
    public void illegalCalculate() {  
        interestCalculator.calculate(-10000, 2);  
    }  
}
```

INFO

Si vous utilisez Eclipse pour le développement, vous pouvez télécharger et installer le plugin TestNG disponible à l'adresse <http://testng.org/doc/eclipse.html> de manière à exécuter les tests TestNG depuis Eclipse. À nouveau, vous verrez une barre verte lorsque tous les tests réussissent, sinon la barre est rouge.

TestNG présente l'avantage de prendre en charge les tests orientés données. Il sépare clairement les données de test de la logique de test. Nous pouvons ainsi exécuter plusieurs fois une méthode de test avec des jeux de données différents. Dans TestNG, ces jeux de données sont apportés par des fournisseurs de données, c'est-à-dire les méthodes marquées par l'annotation `@DataProvider`.

```
package com.apress.springrecipes.bank;  
  
import static org.testng.Assert.*;  
  
import org.testng.annotations.BeforeMethod;  
import org.testng.annotations.DataProvider;  
import org.testng.annotations.Test;  
  
public class SimpleInterestCalculatorTestNG5Tests {  
  
    private InterestCalculator interestCalculator;  
  
    @BeforeMethod  
    public void init() {  
        interestCalculator = new SimpleInterestCalculator();  
        interestCalculator.setRate(0.05);  
    }
```

```
@DataProvider(name = "legal")
public Object[][] createLegalInterestParameters() {
    return new Object[][] { new Object[] { 10000, 2, 1000.0 } };
}

@DataProvider(name = "illegal")
public Object[][] createIllegalInterestParameters() {
    return new Object[][] {new Object[] { -10000, 2 },
                         new Object[] { 10000, -2 }, new Object[] { -10000, -2 }};
}

@Test(dataProvider = "legal")
public void calculate(double amount, double year, double result) {
    double interest = interestCalculator.calculate(amount, year);
    assertEquals(interest, result);
}

@Test(
    dataProvider = "illegal",
    expectedExceptions = IllegalArgumentException.class)
public void illegalCalculate(double amount, double year) {
    interestCalculator.calculate(amount, year);
}
```

Si nous lançons le test précédent dans TestNG, la méthode `calculate()` est exécutée une fois, tandis que la méthode `illegalCalculate()` est exécutée trois fois. En effet, trois jeux de données sont retournés par le fournisseur de données `illegal`.

12.2 Créer des tests unitaires et des tests d'intégration

Problème

La méthode classique pour mener les tests consiste à tester indépendamment chaque module de l'application, puis à les tester tous ensemble. Nous souhaitons appliquer cette méthode pour le test de nos applications Java.

Solution

Les *tests unitaires* sont employés pour tester une seule unité de programmation. Dans les langages orientés objet, une unité correspond généralement à une classe ou à une méthode. Un test unitaire porte sur une seule unité, mais, dans le monde réel, la plupart des unités ne fonctionnent pas indépendamment les unes des autres. Elles ont souvent besoin de coopérer pour mener à bien leurs tâches. Pour tester une unité qui dépend d'autres unités, la technique classique consiste à simuler les dépendances entre les unités avec des bouchons et des objets simulacres. Cela permet de réduire la complexité apportée par les dépendances.

Un *bouchon* (*stub*) est un objet qui simule un objet dépendant en offrant uniquement les méthodes nécessaires au test. Ces méthodes sont implémentées de manière prédefinie, généralement avec des données figées. Un bouchon expose également des méthodes permettant de vérifier son état interne. À l'opposé, un *objet simulacre* (*mock object*) sait généralement comment ses méthodes sont appelées lors du test. Il peut ensuite comparer les méthodes réellement invoquées à celles attendues. En Java, plusieurs bibliothèques facilitent la création d'objets simulacres, dont EasyMock et jMock. La principale différence entre un bouchon et un objet simulacre se situe au niveau de leur utilisation. Un bouchon sert généralement à vérifier un état, tandis qu'un objet simulacre vérifie un comportement.

Les *tests d'intégration* sont employés pour tester plusieurs unités comme un tout. Ils vérifient si l'intégration entre les unités et leurs interactions sont correctes. Chacune des unités doit avoir été préalablement soumise à des tests unitaires. Les tests d'intégration sont donc généralement menés après les tests unitaires.

Enfin, les applications développées sur le principe de séparation de l'interface de l'implémentation et sur le pattern d'injection de dépendance sont faciles à tester, tant avec des tests unitaires qu'avec des tests d'intégration. En effet, ce principe et ce pattern permettent de réduire le couplage entre les différentes unités de l'application.

Explications

Créer des tests unitaires pour des classes isolées

Les fonctions centrales de notre système bancaire s'articulent autour des comptes des clients. Tout d'abord, nous créons la classe de domaine Account, qui fournit sa propre méthode equals().

```
package com.apress.springrecipes.bank;

public class Account {

    private String accountNo;
    private double balance;

    // Constructeurs, accesseurs et mutateurs.
    ...

    public boolean equals(Object obj) {
        if (!(obj instanceof Account)) {
            return false;
        }
        Account account = (Account) obj;
        return account.accountNo.equals(accountNo)
            && account.balance == balance;
    }
}
```

Nous définissons ensuite l'interface de DAO suivante pour la persistance des comptes :

```
package com.apress.springrecipes.bank;

public interface AccountDao {

    public void createAccount(Account account);
    public void updateAccount(Account account);
    public void removeAccount(Account account);
    public Account findAccount(String accountNo);
}
```

Afin d'illustrer le concept de tests unitaires, nous implémentons cette interface en enrégistrant les objets de comptes dans une table d'association. Les classes `AccountNotFoundException` et `DuplicateAccountException` dérivent de `RuntimeException` et vous ne devriez pas avoir de difficulté à les écrire.

```
package com.apress.springrecipes.bank;
...
public class InMemoryAccountDao implements AccountDao {

    private Map<String, Account> accounts;

    public InMemoryAccountDao() {
        accounts = Collections.synchronizedMap(new HashMap<String, Account>());
    }

    public boolean accountExists(String accountNo) {
        return accounts.containsKey(accountNo);
    }

    public void createAccount(Account account) {
        if (accountExists(account.getAccountNo())) {
            throw new DuplicateAccountException();
        }
        accounts.put(account.getAccountNo(), account);
    }

    public void updateAccount(Account account) {
        if (!accountExists(account.getAccountNo())) {
            throw new AccountNotFoundException();
        }
        accounts.put(account.getAccountNo(), account);
    }

    public void removeAccount(Account account) {
        if (!accountExists(account.getAccountNo())) {
            throw new AccountNotFoundException();
        }
        accounts.remove(account.getAccountNo());
    }

    public Account findAccount(String accountNo) {
        Account account = accounts.get(accountNo);
        ...
}
```

```
        if (account == null) {
            throw new AccountNotFoundException();
        }
        return account;
    }
}
```

Cette implémentation simple du DAO ne prend pas en charge les transactions. Toutefois, pour qu'elle soit sûre vis-à-vis des threads, nous pouvons envelopper la table d'enregistrement des comptes dans une table synchronisée qui sérialise les accès.

Créons à présent des tests unitaires JUnit 4 pour cette implémentation. Puisque la classe ne dépend d'aucune autre, elle est facile à tester. Pour nous assurer qu'elle fonctionne aussi bien dans les cas exceptionnels que dans les cas normaux, nous devons également créer des cas de test exceptionnels. En général, ces cas de test s'attendent à ce qu'une exception soit lancée.

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;
import org.junit.Before;
import org.junit.Test;

public class InMemoryAccountDaoTests {

    private static final String EXISTING_ACCOUNT_NO = "1234";
    private static final String NEW_ACCOUNT_NO = "5678";

    private Account existingAccount;
    private Account newAccount;
    private InMemoryAccountDao accountDao;

    @Before
    public void init() {
        existingAccount = new Account(EXISTING_ACCOUNT_NO, 100);
        newAccount = new Account(NEW_ACCOUNT_NO, 200);
        accountDao = new InMemoryAccountDao();
        accountDao.createAccount(existingAccount);
    }

    @Test
    public void accountExists() {
        assertTrue(accountDao.accountExists(EXISTING_ACCOUNT_NO));
        assertFalse(accountDao.accountExists(NEW_ACCOUNT_NO));
    }

    @Test
    public void createNewAccount() {
        accountDao.createAccount(newAccount);
        assertEquals(accountDao.findAccount(NEW_ACCOUNT_NO), newAccount);
    }
}
```

```
@Test(expected = DuplicateAccountException.class)
public void createDuplicateAccount() {
    accountDao.createAccount(existingAccount);
}

@Test
public void updateExistedAccount() {
    existingAccount.setBalance(150);
    accountDao.updateAccount(existingAccount);
    assertEquals(accountDao.findAccount(EXISTING_ACCOUNT_NO),
                existingAccount);
}

@Test(expected = AccountNotFoundException.class)
public void updateNotExistedAccount() {
    accountDao.updateAccount(newAccount);
}

@Test
public void removeExistedAccount() {
    accountDao.removeAccount(existingAccount);
    assertFalse(accountDao.accountExists(EXISTING_ACCOUNT_NO));
}

@Test(expected = AccountNotFoundException.class)
public void removeNotExistedAccount() {
    accountDao.removeAccount(newAccount);
}

@Test
public void findExistedAccount() {
    Account account = accountDao.findAccount(EXISTING_ACCOUNT_NO);
    assertEquals(account, existingAccount);
}

@Test(expected = AccountNotFoundException.class)
public void findNotExistedAccount() {
    accountDao.findAccount(NEW_ACCOUNT_NO);
}
```

Créer des tests unitaires pour des classes dépendantes en utilisant des bouchons et des objets simulacres

Le test d'une classe indépendante reste facile car nous n'avons aucune dépendance à comprendre et à configurer. En revanche, le test d'une classe qui dépend des résultats d'autres classes ou services, par exemple des services de base de données et des services réseau, devient plus complexe. Supposons que l'interface `AccountService` suivante soit définie dans la couche de service :

```
package com.apress.springrecipes.bank;

public interface AccountService {

    public void createAccount(String accountNo);
```

```
    public void removeAccount(String accountNo);
    public void deposit(String accountNo, double amount);
    public void withdraw(String accountNo, double amount);
    public double getBalance(String accountNo);
}
```

Son implémentation présente une dépendance avec un objet AccountDao de la couche de persistance pour la persistance des objets de comptes. InsufficientBalanceException est une sous-classe de RuntimeException que vous devez écrire.

```
package com.apress.springrecipes.bank;

public class AccountServiceImpl implements AccountService {

    private AccountDao accountDao;

    public AccountServiceImpl(AccountDao accountDao) {
        this.accountDao = accountDao;
    }

    public void createAccount(String accountNo) {
        accountDao.createAccount(new Account(accountNo, 0));
    }

    public void removeAccount(String accountNo) {
        Account account = accountDao.findAccount(accountNo);
        accountDao.removeAccount(account);
    }

    public void deposit(String accountNo, double amount) {
        Account account = accountDao.findAccount(accountNo);
        account.setBalance(account.getBalance() + amount);
        accountDao.updateAccount(account);
    }

    public void withdraw(String accountNo, double amount) {
        Account account = accountDao.findAccount(accountNo);
        if (account.getBalance() < amount) {
            throw new InsufficientBalanceException();
        }
        account.setBalance(account.getBalance() - amount);
        accountDao.updateAccount(account);
    }

    public double getBalance(String accountNo) {
        return accountDao.findAccount(accountNo).getBalance();
    }
}
```

Pour réduire la complexité apportée par les dépendances, la technique employée couramment dans les tests unitaires consiste à utiliser des bouchons. Un bouchon implémente la même interface que l'objet cible de manière à le remplacer. Par exemple, nous créons un bouchon pour AccountDao qui enregistre un seul compte de client et implémente uniquement les méthodes `findAccount()` et `updateAccount()`, car elles sont invoquées depuis `deposit()` et `withdraw()`.

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;

import org.junit.Before;
import org.junit.Test;

public class AccountServiceImplStubTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountDaoStub accountDaoStub;
    private AccountService accountService;

    private class AccountDaoStub implements AccountDao {
        private String accountNo;
        private double balance;

        public void createAccount(Account account) {}
        public void removeAccount(Account account) {}

        public Account findAccount(String accountNo) {
            return new Account(this.accountNo, this.balance);
        }

        public void updateAccount(Account account) {
            this.accountNo = account.getAccountNo();
            this.balance = account.getBalance();
        }
    }

    @Before
    public void init() {
        accountDaoStub = new AccountDaoStub();
        accountDaoStub.accountNo = TEST_ACCOUNT_NO;
        accountDaoStub.balance = 100;
        accountService = new AccountServiceImpl(accountDaoStub);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        assertEquals(accountDaoStub.accountNo, TEST_ACCOUNT_NO);
        assertEquals(accountDaoStub.balance, 150, 0);
    }

    @Test
    public void withdrawWithSufficientBalance() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        assertEquals(accountDaoStub.accountNo, TEST_ACCOUNT_NO);
        assertEquals(accountDaoStub.balance, 50, 0);
    }

    @Test(expected = InsufficientBalanceException.class)
    public void withdrawWithInsufficientBalance() {
        accountService.withdraw(TEST_ACCOUNT_NO, 150);
    }
}
```

Toutefois, l'écriture des bouchons implique un développement important. Une technique plus efficace consiste à utiliser des objets simulacres. La bibliothèque EasyMock est capable de créer dynamiquement des objets simulacres qui fonctionnent sur un mécanisme d'enregistrement/lecture.

INFO

Pour utiliser EasyMock dans les tests, vous devez inclure le fichier easymock.jar (situé dans le répertoire lib/easymock de l'installation de Spring) dans le chemin d'accès aux classes.

```
package com.apress.springrecipes.bank;

import org.easymock.MockControl;
import org.junit.Before;
import org.junit.Test;

public class AccountServiceImplMockTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private MockControl mockControl;

    private AccountDao accountDao;
    private AccountService accountService;

    @Before
    public void init() {
        mockControl = MockControl.createControl(AccountDao.class);
        accountDao = (AccountDao) mockControl.getMock();
        accountService = new AccountServiceImpl(accountDao);
    }

    @Test
    public void deposit() {
        Account account = new Account(TEST_ACCOUNT_NO, 100);
        accountDao.findAccount(TEST_ACCOUNT_NO);
        mockControl.setReturnValue(account);
        account.setBalance(150);
        accountDao.updateAccount(account);
        mockControl.replay();

        accountService.deposit(TEST_ACCOUNT_NO, 50);
        mockControl.verify();
    }

    @Test
    public void withdrawWithSufficientBalance() {
        Account account = new Account(TEST_ACCOUNT_NO, 100);
        accountDao.findAccount(TEST_ACCOUNT_NO);
        mockControl.setReturnValue(account);
        account.setBalance(50);
        accountDao.updateAccount(account);
        mockControl.replay();
```

```
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        mockControl.verify();
    }

    @Test(expected = InsufficientBalanceException.class)
    public void testWithdrawWithInsufficientBalance() {
        Account account = new Account(TEST_ACCOUNT_NO, 100);
        accountDao.findAccount(TEST_ACCOUNT_NO);
        mockControl.setReturnValue(account);
        mockControl.replay();

        accountService.withdraw(TEST_ACCOUNT_NO, 150);
        mockControl.verify();
    }
}
```

Avec EasyMock, nous pouvons créer dynamiquement un objet simulacre pour n'importe quelle interface ou classe. Après qu'il a été créé par EasyMock, un objet simulacre se trouve dans l'état *enregistrement*. Tout appel de l'une de ses méthodes est enregistré pour vérification ultérieure. Au cours de l'enregistrement, nous pouvons également préciser la valeur qu'une méthode doit retourner. Après avoir invoqué sa méthode `replay()`, l'objet simulacre se trouve dans l'état *lecture*. Tout appel effectué par la suite est comparé aux appels enregistrés. Nous pouvons invoquer la méthode `verify()` pour vérifier si tous les appels de méthodes enregistrés ont été intégralement effectués. Enfin, nous pouvons appeler la méthode `reset()` pour réinitialiser un objet simulacre et pouvoir le réutiliser. Cependant, puisque nous créons un nouvel objet simulacre dans la méthode marquée par `@Before`, qui est invoquée avant chaque méthode de test, nous ne réutilisons pas l'objet simulacre.

Créer des tests d'intégration

Les tests d'intégration sont employés pour tester plusieurs unités comme un tout de manière à vérifier si l'intégration entre les unités et leurs interactions sont correctes. Par exemple, nous créons un test d'intégration pour tester l'utilisation de `InMemoryAccountDao` par `AccountServiceImpl` comme implémentation du DAO.

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;
import org.junit.After;
import org.junit.Before;
import org.junit.Test;

public class AccountServiceTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;
```

```
@Before  
public void init() {  
    accountService = new AccountServiceImpl(new InMemoryAccountDao());  
    accountService.createAccount(TEST_ACCOUNT_NO);  
    accountService.deposit(TEST_ACCOUNT_NO, 100);  
}  
  
@Test  
public void deposit() {  
    accountService.deposit(TEST_ACCOUNT_NO, 50);  
    assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150, 0);  
}  
  
@Test  
public void withdraw() {  
    accountService.withdraw(TEST_ACCOUNT_NO, 50);  
    assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50, 0);  
}  
  
@After public void cleanup() {  
    accountService.removeAccount(TEST_ACCOUNT_NO);  
}  
}
```

12.3 Effectuer des tests unitaires sur des contrôleurs Spring MVC

Problème

Dans une application web, nous souhaitons tester les contrôleurs web développés à l'aide du framework Spring MVC.

Solution

Un contrôleur Spring MVC est invoqué par `DispatcherServlet`, avec un objet de requête HTTP et un objet de réponse HTTP en arguments. Après avoir traité la requête, le contrôleur crée un objet `ModelAndView` et le retourne à `DispatcherServlet` pour l'affichage de la vue. Pour tester des contrôleurs Spring MVC, ainsi que des contrôleurs web développés avec d'autres frameworks d'application web, le principal problème est de simuler des objets de requête et de réponse HTTP dans un environnement de tests unitaires. Heureusement, Spring facilite le test des contrôleurs web en fournissant un ensemble d'objets simulacres pour l'API des servlets, notamment `MockHttpServletRequest`, `MockHttpServletResponse` et `MockHttpSession`.

Pour tester le résultat d'un contrôleur Spring MVC, nous devons vérifier si l'objet `ModelAndView` retourné à `DispatcherServlet` est correct. Spring fournit également un ensemble d'outils d'assertion pour vérifier le contenu d'un objet `ModelAndView`.

Spring 2.5 prend en charge les contrôleurs basés sur les annotations, qui ne sont pas obligés d'étendre une classe de contrôleur Spring MVC ou de manipuler directement les requêtes et les réponses HTTP. Par conséquent, ces contrôleurs peuvent être testés comme de simples classes Java.

Explications

Créer un test unitaire pour un contrôleur Spring MVC classique

Dans notre système bancaire, supposons que nous développons une interface web avec Spring MVC pour que le personnel saisisse le numéro de compte et le montant d'un dépôt. Nous créons un contrôleur simple en étendant la classe `AbstractController` de Spring MVC.

ATTENTION

Pour développer un contrôleur web avec Spring MVC, vous devez inclure les fichiers `spring-webmvc.jar` (situé dans le répertoire `dist/modules` de l'installation de Spring) et `servlet-api.jar` (situé dans `lib/j2ee`) dans le chemin d'accès aux classes.

```
package com.apress.springrecipes.bank;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import org.springframework.web.bind.ServletRequestUtils;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.AbstractController;

public class DepositController extends AbstractController {

    private AccountService accountService;

    public DepositController(AccountService accountService) {
        this.accountService = accountService;
    }

    protected ModelAndView handleRequestInternal(HttpServletRequest request,
                                                HttpServletResponse response) throws Exception {
        String accountNo = ServletRequestUtils.getRequiredStringParameter(
            request, "accountNo");
        double amount = ServletRequestUtils.getRequiredDoubleParameter(
            request, "amount");

        accountService.deposit(accountNo, amount);

        double balance = accountService.getBalance(accountNo);
        return new ModelAndView("success", "accountNo", accountNo)
            .addObject("balance", balance);
    }
}
```

Dans la méthode de gestion des requêtes de ce contrôleur, nous obtenons simplement les deux paramètres requis, accountNo et amount, à partir de l'objet de requête HTTP, puis nous les passons à l'objet de service AccountService pour effectuer un dépôt. Ensuite, nous consultons le solde courant du compte et le stockons avec le numéro du compte dans l'objet ModelAndView, dont le nom de vue est fixé à success.

À l'aide des outils de test des contrôleurs fournis par Spring, nous pouvons créer un test unitaire pour ce contrôleur. Afin d'être certains que l'objet de service AccountService soit invoqué correctement, nous utilisons EasyMock.

INFO

Pour bénéficier des outils de test de Spring, vous devez inclure le fichier spring-test.jar (situé dans le répertoire dist/modules de l'installation de Spring) dans le chemin d'accès aux classes.

```
package com.apress.springrecipes.bank;

import static org.springframework.test.web.ModelAndViewAssert.*;
import javax.servlet.http.HttpServletResponse;
import org.easymock.MockControl;
import org.junit.Before;
import org.junit.Test;
import org.springframework.mock.web.MockHttpServletRequest;
import org.springframework.mock.web.MockHttpServletResponse;
import org.springframework.web.servlet.ModelAndView;

public class DepositControllerTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private static final double TEST_AMOUNT = 50;
    private MockControl mockControl;
    private AccountService accountService;
    private DepositController depositController;

    @Before
    public void init() {
        mockControl = MockControl.createControl(AccountService.class);
        accountService = (AccountService) mockControl.getMock();
        depositController = new DepositController(accountService);
    }

    @Test
    public void deposit() throws Exception {
        MockHttpServletRequest request = new MockHttpServletRequest();
        request.setMethod("POST");
        request.addParameter("accountNo", TEST_ACCOUNT_NO);
        request.addParameter("amount", String.valueOf(TEST_AMOUNT));
        HttpServletResponse response = new MockHttpServletResponse();
    }
}
```

```

        accountService.deposit(TEST_ACCOUNT_NO, 50);
        accountService.getBalance(TEST_ACCOUNT_NO);
        mockControl.setReturnValue(150.0);
        mockControl.replay();

        ModelAndView modelAndView =
            depositController.handleRequest(request, response);
        mockControl.verify();

        assertEquals(modelAndView, "success");
        assertEquals(modelAndView.getAttribute("accountNo"), TEST_ACCOUNT_NO);
        assertEquals(modelAndView.getAttribute("balance"), 150.0);
    }
}

```

Avec `MockHttpServletRequest`, nous pouvons simuler une méthode de requête HTTP (par exemple GET ou POST), des paramètres, des en-têtes, des cookies, etc. Pour invoquer la méthode `handleRequest()` d'un contrôleur, nous devons également créer un objet `MockHttpServletResponse`.

Si `DepositController` fonctionne correctement, il doit appeler l'objet `AccountService` avec le numéro de compte et le montant indiqués dans les paramètres de requête. Pour vérifier ce comportement, nous utilisons EasyMock.

Enfin, nous devons vérifier le contenu de l'objet `ModelAndView`. Pour JUnit 3.8, notre classe de test peut étendre `AbstractModelAndViewTests` de manière à hériter de la méthode d'assertion spécifique à `ModelAndView`. Avec JUnit 4, puisque nous ne devons pas dériver d'une classe de test de base, nous utilisons une importation statique pour importer toutes les méthodes statiques d'assertion de la classe `ModelAndViewAssert`.

Créer un test unitaire pour un contrôleur Spring MVC défini par des annotations

Implémentons à présent `DepositController` en utilisant l'approche Spring 2.5 fondée sur les annotations.

```

package com.apress.springrecipes.bank;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.ModelMap;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;

@Controller
public class DepositController {

    private AccountService accountService;

    @Autowired
    public DepositController(AccountService accountService) {
        this.accountService = accountService;
    }
}

```

```
@RequestMapping("/deposit.do")
protected String deposit(
    @RequestParam("accountNo") String accountNo,
    @RequestParam("amount") double amount,
    ModelMap model) {
    accountService.deposit(accountNo, amount);
    model.addAttribute("accountNo", accountNo);
    model.addAttribute("balance", accountService.getBalance(accountNo));
    return "success";
}
```

Puisque ce contrôleur ne manipule pas directement l'API des servlets et ne retourne pas d'objet `ModelAndView` spécifique à Spring, il est très facile à tester. Nous procérons comme pour une simple classe Java.

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;

import org.easymock.MockControl;
import org.junit.Before;
import org.junit.Test;
import org.springframework.ui.ModelMap;

public class DepositControllerTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private static final double TEST_AMOUNT = 50;
    private MockControl mockControl;
    private AccountService accountService;
    private DepositController depositController;

    @Before
    public void init() {
        mockControl = MockControl.createControl(AccountService.class);
        accountService = (AccountService) mockControl.getMock();
        depositController = new DepositController(accountService);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        accountService.getBalance(TEST_ACCOUNT_NO);
        mockControl.setReturnValue(150.0);
        mockControl.replay();

        ModelMap model = new ModelMap();
        String viewName =
            depositController.deposit(TEST_ACCOUNT_NO, TEST_AMOUNT, model);
        mockControl.verify();

        assertEquals(viewName, "success");
        assertEquals(model.get("accountNo"), TEST_ACCOUNT_NO);
        assertEquals(model.get("balance"), 150.0);
    }
}
```

12.4 Gérer des contextes d'application dans les tests d'intégration

Problème

Lorsque nous créons des tests d'intégration pour une application Spring, nous devons accéder aux beans déclarés dans le contexte d'application. Sans le soutien de Spring pour les tests, nous devons charger manuellement le contexte d'application dans une méthode d'initialisation de nos tests, comme `setUp()` avec JUnit 3.8, ou dans une méthode marquée par `@Before` ou `@BeforeClass` avec JUnit 4. Cependant, puisqu'une méthode d'initialisation est appelée avant chaque méthode ou classe de test, le même contexte d'application peut être chargé plusieurs fois. Dans une grande application constituée de nombreux beans, le chargement d'un contexte d'application peut demander beaucoup de temps, ce qui ralentit l'exécution des tests.

Solution

Les outils de prise en charge des tests fournis par Spring peuvent nous aider à gérer les contextes d'application dans nos tests, par exemple en les chargeant depuis un ou plusieurs fichiers de configuration des beans et en les plaçant dans un cache pour accélérer les multiples exécutions d'un test. Un contexte d'application est placé dans un cache pour tous les tests d'une même JVM, en utilisant les emplacements du fichier de configuration comme clés. Puisqu'un contexte d'application n'est plus chargé à de multiples reprises, les tests s'exécutent plus rapidement.

Avec la prise en charge ancienne de JUnit 3.8 dans Spring, c'est-à-dire dans les versions antérieures à la 2.5, notre classe de test peut étendre la classe de base `AbstractSingleSpringContextTests` pour accéder au contexte d'application géré au travers de la méthode héritée `getApplicationContext()`.

Dans Spring 2.5, le framework `TestContext` fournit deux écouteurs d'exécution du test en lien avec la gestion des contextes. Ils peuvent être enregistrés avec un gestionnaire de contexte de test par défaut si aucun n'est indiqué explicitement.

- **DependencyInjectionTestExecutionListener.** Il injecte des dépendances dans nos tests, notamment le contexte d'application géré.
- **DirtiesContextTestExecutionListener.** Il traite l'annotation `@DirtiesContext` et recharge le contexte d'application lorsque c'est nécessaire.

Pour que le framework `TestContext` gère le contexte d'application, notre classe de test doit être intégrée en interne à un gestionnaire de contexte de test. Dans ce but, et afin de simplifier notre travail, le framework `TestContext` fournit des classes de support (voir Tableau 12.1). Ces classes s'intègrent à un gestionnaire de contexte de test et implémentent l'interface `ApplicationContextAware` pour fournir un accès au contexte

d'application géré par l'intermédiaire du champ protégé `ApplicationContext`. Notre classe de test doit simplement étendre la classe de `TestContext` qui correspond au framework de test.

Tableau 12.1 : Classes de `TestContext` pour la gestion du contexte

<i>Framework de test</i>	<i>Classe de support de <code>TestContext</code>¹</i>
JUnit 3.8	<code>AbstractJUnit38SpringContextTests</code>
JUnit 4.4	<code>AbstractJUnit4SpringContextTests</code>
TestNG	<code>AbstractTestNGSpringContextTests</code>

1. Pour ces trois classes de `TestContext`, seuls `DependencyInjectionTestExecutionListener` et `DirtiesContextTestExecutionListener` sont activés.

Si nous utilisons JUnit 4.4 ou TestNG, nous pouvons intégrer nous-mêmes notre classe de test à un gestionnaire de contexte de test et implémenter directement l'interface `ApplicationContextAware`, sans étendre l'une des classes de `TestContext`. En procédant ainsi, la classe de test n'est pas liée à la hiérarchie de classes du framework `TestContext` et nous pouvons donc dériver de notre propre classe de base. Dans JUnit 4.4, nous pouvons simplement exécuter notre test avec `SpringJUnit4ClassRunner` pour qu'un gestionnaire de contexte de test soit intégré. En revanche, dans TestNG, l'intégration à un gestionnaire de contexte de test doit se faire manuellement.

Explications

Tout d'abord, nous déclarons une instance de `AccountService` et une instance de `AccountDao` dans le fichier de configuration des beans. Par la suite, nous créerons des tests d'intégration pour ces instances.

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
                           http://www.springframework.org/schema/beans/spring-beans-2.5.xsd">

    <bean id="accountDao"
          class="com.apress.springrecipes.bank.InMemoryAccountDao" />

    <bean id="accountService"
          class="com.apress.springrecipes.bank.AccountServiceImpl">
        <constructor-arg ref="accountDao" />
    </bean>
</beans>
```

Accéder au contexte avec la prise en charge ancienne de JUnit 3.8

Lorsque les tests sont créés en utilisant la prise en charge ancienne de JUnit 3.8, notre classe de test peut étendre `AbstractSingleSpringContextTests` pour accéder au contexte d'application géré.

```
package com.apress.springrecipes.bank;

import org.springframework.test.AbstractSingleSpringContextTests;

public class AccountServiceJUnit38LegacyTests
    extends AbstractSingleSpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;

    protected String[] getConfigLocations() {
        return new String[] { "beans.xml" };
    }

    protected void onSetUp() throws Exception {
        accountService =
            (AccountService) getApplicationContext().getBean("accountService");
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    public void testDeposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150.0);
    }

    public void testWithDraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50.0);
    }

    protected void onTearDown() throws Exception {
        accountService.removeAccount(TEST_ACCOUNT_NO);
    }
}
```

Dans cette classe, nous redéfinissons la méthode `getConfigLocations()` pour qu'elle retourne la liste des emplacements des fichiers de configuration des beans. Par défaut, il s'agit d'emplacements du chemin d'accès aux classes relativement à la racine, mais les préfixes de ressources de Spring sont acceptés (par exemple `file` et `classpath`). Une autre solution consiste à redéfinir la méthode `getConfigPath()` ou la méthode `getConfigPaths()` pour retourner un ou plusieurs chemins de fichiers de configuration, qui peuvent être des emplacements absous du chemin d'accès aux classes commençant par une barre oblique, ou des chemins relatifs au paquetage de la classe de test courante.

Par défaut, le contexte d'application est chargé dans un cache et réutilisé ensuite pour chaque méthode de test. Cependant, dans certains cas, par exemple si nous modifions

les configurations de beans ou changeons l'état d'un bean dans une méthode de test, nous devons recharger le contexte d'application. En invoquant la méthode `setDirty()`, nous indiquons que le contexte d'application n'est plus valide et qu'il doit être rechargeé automatiquement lors du prochain appel à une méthode de test.

Enfin, puisqu'elles sont déclarées `final`, il est impossible de redéfinir les méthodes `setUp()` et `tearDown()` de la classe de base. Pour procéder aux tâches d'initialisation et de nettoyage, nous devons à la place redéfinir les méthodes `onSetUp()` et `onTearDown()`. Elles sont invoquées par les méthodes `setUp()` et `tearDown()` de la classe parent.

Accéder au contexte avec le framework TestContext dans JUnit 4.4

Si nous utilisons JUnit 4.4 pour créer des tests avec le framework `TestContext`, deux solutions s'offrent à nous pour accéder au contexte d'application géré. La première consiste à implémenter l'interface `ApplicationContextAware`, mais nous devons alors préciser explicitement un exécuteur de test Spring, `SpringJUnit4ClassRunner`, avec l'annotation `@RunWith` au niveau de la classe.

```
package com.apress.springrecipes.bank;

import static org.junit.Assert.*;

import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import org.junit.runner.RunWith;
import org.springframework.context.ApplicationContext;
import org.springframework.context.ApplicationContextAware;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests
    implements ApplicationContextAware {

    private static final String TEST_ACCOUNT_NO = "1234";
    private ApplicationContext applicationContext;
    private AccountService accountService;

    public void setApplicationContext(ApplicationContext applicationContext) {
        this.applicationContext = applicationContext;
    }

    @Before
    public void init() {
        accountService =
            (AccountService) applicationContext.getBean("accountService");
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
}
```

```

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150, 0);
    }

    @Test
    public void withdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50, 0);
    }

    @After public void cleanup() {
        accountService.removeAccount(TEST_ACCOUNT_NO);
    }
}

```

Les emplacements des fichiers de configuration de beans sont indiqués dans l'attribut `locations` de l'annotation `@ContextConfiguration` au niveau de la classe. Par défaut, il s'agit d'emplacements du chemin d'accès aux classes relatifs à la classe de test, mais les préfixes de ressources Spring sont reconnus. Si cet attribut n'est pas défini explicitement, le framework `TestContext` charge le fichier dont le nom correspond au nom de la classe de test suffixé par `-context.xml` (c'est-à-dire `AccountServiceJUnit4Tests-context.xml`).

Par défaut, le contexte d'application est placé dans un cache et réutilisé pour chaque méthode de test. Toutefois, si nous souhaitons le recharger après l'exécution d'une certaine méthode de test, nous pouvons annoter celle-ci avec `@DirtiesContext`. Le contexte d'application est rechargeé lors du prochain appel à une méthode de test.

La seconde solution pour accéder au contexte d'application géré consiste à étendre la classe de `TestContext` spécifique à JUnit 4.4 : `AbstractJUnit4SpringContextTests`. Puisqu'elle implémente l'interface `ApplicationContextAware`, nous pouvons l'étendre pour accéder au contexte d'application géré au travers du champ protégé `application-Context`. Toutefois, nous devons commencer par retirer le champ privé `application-Context` et son mutateur. Si nous dérivons de cette classe de support, il est inutile de préciser `SpringJUnit4ClassRunner` dans l'annotation `@RunWith` car elle est héritée du parent.

```

package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.RunWith
    AbstractJUnit4SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests
    extends AbstractJUnit4SpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;
}

```

```
@Before  
public void init() {  
    accountService =  
        (AccountService) applicationContext.getBean("accountService");  
    accountService.createAccount(TEST_ACCOUNT_NO);  
    accountService.deposit(TEST_ACCOUNT_NO, 100);  
}  
...  
}
```

Accéder au contexte avec le framework TestContext dans JUnit 3.8

Si nous voulons accéder au contexte d'application géré avec le framework TestContext dans JUnit 3.8, nous devons étendre la classe `AbstractJUnit38SpringContextTests` fournie par `TestContext`. Puisqu'elle implémente l'interface `ApplicationContext-Aware`, nous pouvons accéder au contexte d'application au travers du champ protégé `applicationContext`.

```
package com.apress.springrecipes.bank;  
  
import org.springframework.test.context.ContextConfiguration;  
import org.springframework.test.context.junit38.  
    AbstractJUnit38SpringContextTests;  
  
@ContextConfiguration(locations = "/beans.xml")  
public class AccountServiceJunit38ContextTests  
    extends AbstractJUnit38SpringContextTests {  
  
    private static final String TEST_ACCOUNT_NO = "1234";  
    private AccountService accountService;  
  
    protected void setUp() throws Exception {  
        accountService =  
            (AccountService) applicationContext.getBean("accountService");  
        accountService.createAccount(TEST_ACCOUNT_NO);  
        accountService.deposit(TEST_ACCOUNT_NO, 100);  
    }  
  
    public void testDeposit() {  
        accountService.deposit(TEST_ACCOUNT_NO, 50);  
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150.0);  
    }  
  
    public void testWithDraw() {  
        accountService.withdraw(TEST_ACCOUNT_NO, 50);  
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50.0);  
    }  
  
    protected void tearDown() throws Exception {  
        accountService.removeAccount(TEST_ACCOUNT_NO);  
    }  
}
```

Accéder au contexte avec le framework TestContext dans TestNG

Pour accéder au contexte d'application géré avec le framework TestContext dans TestNG, nous pouvons étendre la classe `AbstractTestNGSpringContextTests` de `TestContext`. Elle implémente également l'interface `ApplicationContextAware`.

```
package com.apress.springrecipes.bank;

import static org.testng.Assert.*;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.*
    AbstractTestNGSpringContextTests;
import org.testng.annotations.AfterMethod;
import org.testng.annotations.BeforeMethod;
import org.testng.annotations.Test;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceTestNGContextTests
    extends AbstractTestNGSpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";
    private AccountService accountService;

    @BeforeMethod
    public void init() {
        accountService =
            (AccountService) applicationContext.getBean("accountService");
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    @Test
    public void deposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 150, 0);
    }

    @Test
    public void withdraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        assertEquals(accountService.getBalance(TEST_ACCOUNT_NO), 50, 0);
    }

    @AfterMethod public void cleanup() {
        accountService.removeAccount(TEST_ACCOUNT_NO);
    }
}
```

Si nous ne voulons pas que notre classe de test TestNG dérive d'une classe de support de `TestContext`, nous pouvons implémenter l'interface `ApplicationContextAware` comme cela a été fait pour JUnit 4.4. Cependant, nous devons procéder nous-mêmes à l'intégration à un gestionnaire de contexte de test. Pour plus de détails, consultez le code source de `AbstractTestNGSpringContextTests`.

12.5 Injecter des fixtures de test dans des tests d'intégration

Problème

Dans un test d'intégration pour une application Spring, les fixtures de test sont principalement des beans déclarés dans le contexte d'application. Nous souhaitons qu'elles soient injectées automatiquement par Spring *via* une injection de dépendance afin que nous n'ayons plus besoin de les obtenir manuellement à partir du contexte d'application.

Solution

Les fonctions Spring de prise en charge des tests permettent d'injecter automatiquement dans nos tests des beans du contexte d'application géré, sous forme de fixtures de test.

Avec la prise en charge ancienne de JUnit 3.8 dans les versions de Spring antérieures à la version 2.5, notre classe de test doit étendre la classe de base `AbstractDependencyInjectionSpringContextTests`, qui est une sous-classe de `AbstractSingleSpringContextTests`, pour que ses fixtures de test soient injectées automatiquement. Cette classe reconnaît deux manières de procéder à l'injection de dépendance. La première lie automatiquement des beans selon le type en utilisant des mutateurs. La seconde lie automatiquement des beans selon le nom en utilisant des champs protégés.

Dans le framework `TestContext` de Spring 2.5, `DependencyInjectionTestExecutionListener` est en mesure d'injecter automatiquement des dépendances dans nos tests. Si cet écouteur est enregistré, il suffit de marquer un mutateur ou un champ du test avec l'annotation `@Autowired` de Spring ou l'annotation `@Resource` de la JSR-250 pour qu'une fixture soit injectée automatiquement. Avec `@Autowired`, la fixture est injectée selon le type, tandis qu'avec `@Resource` elle est injectée selon le nom. Cette fonctionnalité reste cohérente avec la prise en charge des annotations dans Spring 2.5.

Explications

Injecter des fixtures de test avec la prise en charge ancienne de JUnit 3.8

Lorsque les tests sont créés avec la prise en charge ancienne de JUnit 3.8, la classe de test peut étendre `AbstractDependencyInjectionSpringContextTests` pour que ses fixtures de test soient injectées à partir des beans du contexte d'application géré. Nous définissons un mutateur pour la fixture à injecter.

```
package com.apress.springrecipes.bank;  
  
import org.springframework.test.AbstractDependencyInjectionSpringContextTests;  
  
public class AccountServiceJUnit38LegacyTests  
    extends AbstractDependencyInjectionSpringContextTests {  
  
    private AccountService accountService;
```

```
public void setAccountService(AccountService accountService) {
    this.accountService = accountService;
}

protected void setUp() throws Exception {
    accountService.createAccount(TEST_ACCOUNT_NO);
    accountService.deposit(TEST_ACCOUNT_NO, 100);
}
...
}
```

Puisque `AbstractDependencyInjectionSpringContextTests` est une sous-classe de `AbstractSingleSpringContextTests`, elle gère également un contexte d'application chargé à partir des fichiers de configuration des beans indiqués par la méthode `getConfigLocations()`. Par défaut, cette classe utilise la liaison automatique par type pour injecter des beans issus du contexte d'application. Cependant, la liaison automatique ne fonctionne pas lorsque plusieurs beans du contexte d'application sont du même type. Dans ce cas, nous devons rechercher explicitement le bean dans le contexte d'application fourni par `getApplicationContext()` et retirer le mutateur à l'origine de l'ambiguïté.

Pour injecter des fixtures de test avec la prise en charge ancienne de JUnit 3.8, nous pouvons également utiliser des champs protégés. Pour que cela fonctionne, nous devons activer la propriété `populateProtectedVariables` dans un constructeur. Dans ce cas, nous n'avons pas à fournir de mutateur pour chaque champ à injecter.

```
package com.apress.springrecipes.bank;

import org.springframework.test.AbstractDependencyInjectionSpringContextTests;

public class AccountServiceJUnit38LegacyTests
    extends AbstractDependencyInjectionSpringContextTests {

    protected AccountService accountService;

    public AccountServiceJUnit38LegacyTests() {
        setPopulateProtectedVariables(true);
    }
    ...
}
```

Le nom du champ protégé est utilisé pour rechercher un bean éponyme dans le contexte d'application géré.

Injecter des fixtures de test avec le framework TestContext dans JUnit 4.4

Lorsque les tests sont créés avec le framework `TestContext`, les fixtures de test peuvent être injectées à partir du contexte d'application géré en marquant un champ ou un mutateur avec l'annotation `@Autowired` ou `@Resource`. Dans JUnit 4.4, nous pouvons opter pour l'exécuteur de tests `SpringJUnit4ClassRunner` sans étendre une classe de support.

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @Before
    public void init() {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
    ...
}
```

Si nous annotons un champ ou un mutateur du test avec `@Autowired`, l'injection se fait par liaison automatique selon le type. Nous pouvons indiquer un bean candidat à la liaison automatique en précisant son nom dans l'annotation `@Qualifier`. Si nous souhaitons que la liaison automatique d'un champ ou d'un mutateur se fasse par le nom, nous utilisons l'annotation `@Resource`.

Nous pouvons également dériver de la classe `AbstractJUnit4SpringContextTests` pour que les fixtures de test soient injectées à partir du contexte d'application géré. Dans ce cas, il est inutile de préciser `SpringJUnit4ClassRunner` car il est hérité du parent.

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.extends
    AbstractJUnit4SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests
    extends AbstractJUnit4SpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;
    ...
}
```

Injecter des fixtures de test avec le framework TestContext dans JUnit 3.8

Dans JUnit 3.8, nous pouvons employer le framework TestContext pour créer des tests qui utilisent la même approche à l'injection des fixtures de test. Toutefois, notre classe de test doit alors dériver de la classe `AbstractJUnit38SpringContextTests` fournie par `TestContext`.

```
package com.apress.springrecipes.bank;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit38.AbstractJUnit38SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceUnit38ContextTests
    extends AbstractJUnit38SpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    protected void setUp() throws Exception {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }
    ...
}
```

Injecter des fixtures de test avec le framework TestContext dans TestNG

Dans TestNG, nous étendons la classe `AbstractTestNGSpringContextTests` de `TestContext` pour que les fixtures de test soient injectées à partir du contexte d'application géré.

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.AbstractTestNGSpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceTestNGContextTests
    extends AbstractTestNGSpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @BeforeMethod
    public void init() {
```

```
    accountService.createAccount(TEST_ACCOUNT_NO);
    accountService.deposit(TEST_ACCOUNT_NO, 100);
}
...
}
```

12.6 Gérer des transactions dans les tests d'intégration

Problème

Lorsque l'on crée des tests d'intégration pour une application qui accède à une base de données, les données de test sont généralement préparées dans la méthode d'initialisation. L'exécution de chaque méthode de test peut modifier la base de données. Nous devons donc nettoyer la base pour que l'exécution de la méthode de test suivante se fasse sur un état cohérent. Par conséquent, nous devons écrire de nombreuses tâches de nettoyage de la base de données.

Solution

Les fonctions Spring de prise en charge des tests permettent de créer et d'annuler une transaction pour chaque méthode de test. Ainsi, les modifications apportées à la base de données par une méthode de test n'ont pas d'effet sur la méthode suivante. Cela nous évite également d'écrire des opérations de nettoyage de la base de données.

Avec la prise en charge ancienne de JUnit 3.8 dans les versions de Spring antérieure à la version 2.5, notre classe de test peut étendre la classe de base `AbstractTransactionalSpringContextTests`, qui est une sous-classe de `AbstractDependencyInjectionSpringContextTests`, pour créer et annuler une transaction pour chaque méthode de test. Un gestionnaire de transactions doit être configuré correctement dans le fichier de configuration des beans.

Dans Spring 2.5, le framework `TestContext` fournit un écouteur d'exécution du test en rapport avec la gestion des transactions. Il est enregistré avec un gestionnaire de contexte de test par défaut si aucun n'est précisé explicitement. `TransactionalTestExecutionListener` traite l'annotation `@Transactional` au niveau de la classe ou de la méthode, et les méthodes sont exécutées automatiquement dans des transactions.

Pour que ses méthodes s'exécutent dans des transactions, notre classe de test étend la classe de `TestContext` qui correspond à notre framework de test (voir Tableau 12.2). Ces classes s'intègrent à un gestionnaire de contexte de test et sont marquées par `@Transactional`. Un gestionnaire de transactions doit également être déclaré dans le fichier de configuration des beans.

Tableau 12.2 : Classes de TestContext pour la gestion des transactions

<i>Framework de test</i>	<i>Classe de support de TestContext¹</i>
JUnit 3.8	AbstractTransactionalJUnit38SpringContextTests
JUnit 4.4	AbstractTransactionalJUnit4SpringContextTests
TestNG	AbstractTransactionalTestNGSpringContextTests

1. Pour ces trois classes de TestContext, `TransactionalTestExecutionListener` est activé, en plus de `DependencyInjectionTestExecutionListener` et `DirtiesContextTestExecutionListener`.

Dans JUnit 4.4 ou TestNG, nous pouvons simplement utiliser `@Transactional` au niveau de la classe ou de la méthode pour que les méthodes de test s'exécutent dans des transactions, sans avoir à étendre l'une des classes de TestContext. Cependant, pour l'intégration à un gestionnaire de contexte de test, nous devons exécuter le test JUnit 4.4 avec `SpringJUnit4ClassRunner` et réaliser l'intégration manuellement pour un test TestNG.

Explications

Voyons à présent comment enregistrer les comptes de notre système bancaire dans une base de données relationnelle. Nous pouvons choisir tout moteur de bases de données compatible JDBC qui prend en charge les transactions, puis exécuter la requête SQL suivante pour créer la table ACCOUNT. Dans notre exemple, nous optons pour le moteur Apache Derby et créons la table dans l'instance bank.

```
CREATE TABLE ACCOUNT (
    ACCOUNT_NO      VARCHAR(10)      NOT NULL,
    BALANCE         DOUBLE           NOT NULL,
    PRIMARY KEY (ACCOUNT_NO)
);
```

Nous créons ensuite une nouvelle implémentation du DAO qui accède à la base de données via JDBC. Pour simplifier les opérations, nous tirons parti de `SimpleJdbcTemplate`.

```
package com.apress.springrecipes.bank;

import org.springframework.jdbc.core.simple.SimpleJdbcDaoSupport;

public class JdbcAccountDao extends SimpleJdbcDaoSupport
    implements AccountDao {

    public void createAccount(Account account) {
        String sql = "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)";
        getSimpleJdbcTemplate().update(
            sql, account.getAccountNo(), account.getBalance());
    }
}
```

```
public void updateAccount(Account account) {  
    String sql = "UPDATE ACCOUNT SET BALANCE = ? WHERE ACCOUNT_NO = ?";  
    getSimpleJdbcTemplate().update(  
        sql, account.getBalance(), account.getAccountNo());  
}  
  
public void removeAccount(Account account) {  
    String sql = "DELETE FROM ACCOUNT WHERE ACCOUNT_NO = ?";  
    getSimpleJdbcTemplate().update(sql, account.getAccountNo());  
}  
  
public Account findAccount(String accountNo) {  
    String sql = "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?";  
    double balance = getSimpleJdbcTemplate().queryForObject(  
        sql, Double.class, accountNo);  
    return new Account(accountNo, balance);  
}  
}
```

Avant de créer des tests d'intégration pour tester l'instance de `AccountService` qui utilise ce DAO pour la persistance des comptes, nous devons remplacer `InMemoryAccountDao` par ce DAO dans le fichier de configuration des beans et configurer également la source de données.

INFO

Pour accéder à une base de données hébergée sur le serveur Derby, vous devez inclure le fichier `derbyclient.jar` (situé dans le répertoire `lib` de l'installation de Derby) dans le chemin d'accès aux classes.

```
<beans ...>  
    <bean id="dataSource"  
          class="org.springframework.jdbc.datasource.DriverManagerDataSource">  
        <property name="driverClassName"  
                 value="org.apache.derby.jdbc.ClientDriver" />  
        <property name="url"  
                 value="jdbc:derby://localhost:1527/bank;create=true" />  
        <property name="username" value="app" />  
        <property name="password" value="app" />  
    </bean>  
  
    <bean id="accountDao"  
          class="com.apress.springrecipes.bank.JdbcAccountDao">  
        <property name="dataSource" ref="dataSource" />  
    </bean>  
    ...  
</beans>
```

Gérer les transactions avec la prise en charge ancienne de JUnit 3.8

Lorsque les tests sont créés avec la prise en charge ancienne de JUnit 3.8, la classe de test peut étendre `AbstractTransactionalSpringContextTests` pour que ses méthodes de test s'exécutent dans des transactions.

```
package com.apress.springrecipes.bank;

import org.springframework.test.AbstractTransactionalSpringContextTests;

public class AccountServiceJUnit38LegacyTests
    extends AbstractTransactionalSpringContextTests {

    protected void onSetUpInTransaction() throws Exception {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    // onTearDown() n'est plus utile.
    ...
}
```

Par défaut, chaque méthode de test s'exécute au sein d'une transaction, qui est annulée à la fin de la méthode. Ainsi, nous n'avons aucunement besoin de nettoyer la base de données dans la méthode `onTearDown()`, que nous retirons simplement. Les tâches de préparation des données doivent se faire dans la méthode `onSetUpInTransaction()`, non dans `onSetUp()`, pour qu'elles s'exécutent au sein des mêmes transactions que les méthodes de test, qui sont annulées.

En invoquant explicitement `setComplete()`, nous pouvons faire en sorte qu'une transaction soit validée à la fin d'une méthode de test au lieu d'être annulée. Par ailleurs, nous pouvons terminer une transaction au cours d'une méthode de test en appelant `endTransaction()`, qui annule la transaction normalement ou la valide si `setComplete()` avait été invoquée auparavant.

La classe exige qu'un gestionnaire de transactions soit déclaré dans le fichier de configuration des beans. Par défaut, elle recherche un bean de type `PlatformTransactionManager` et l'utilise pour gérer les transactions des méthodes de test.

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Gérer des transactions avec le framework TestContext dans JUnit 4.4

Lorsque les tests sont créés avec le framework `TestContext`, nous pouvons utiliser l'annotation `@Transactional` au niveau de la classe ou de la méthode pour que les méthodes de test s'exécutent dans des transactions. Avec JUnit 4.4, nous pouvons choi-

sir une exécution de la classe de test avec `SpringJUnit4ClassRunner` afin qu'elle ne soit pas obligée d'étendre une classe de support.

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.SpringJUnit4ClassRunner;
import org.springframework.transaction.annotation.Transactional;

@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(locations = "/beans.xml")
@Transactional
public class AccountServiceJUnit4ContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @Before
    public void init() {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    // cleanup() n'est plus utile.
    ...
}
```

Lorsque la classe de test est marquée par `@Transactional`, toutes ses méthodes de test s'exécutent au sein de transactions. Si nous voulons qu'une méthode ne s'exécute pas dans une transaction, nous la marquons avec `@NotTransactional`. Une autre solution consiste à marquer chaque méthode individuelle avec `@Transactional` à la place de la classe.

Par défaut, les transactions des méthodes de test sont annulées à la fin de l'exécution des méthodes. Nous pouvons modifier ce comportement en désactivant l'attribut `defaultRollback` de l'annotation `@TransactionConfiguration`, qui doit être appliquée au niveau de la classe. Par ailleurs, nous pouvons redéfinir au niveau de la méthode ce comportement d'annulation au niveau de la classe en utilisant l'annotation `@Rollback`, qui prend une valeur booléenne en argument.

Les méthodes marquées par `@Before` ou `@After` sont exécutées au sein des mêmes transactions que les méthodes de test. Si nous voulons que des méthodes réalisent des opérations d'initialisation ou de nettoyage avant ou après une transaction, nous devons les annoter avec `@BeforeTransaction` ou `@AfterTransaction`. Ces méthodes ne sont pas exécutées pour les méthodes de test marquées par `@NotTransactional`.

Enfin, un gestionnaire de transactions doit être déclaré dans le fichier de configuration des beans. Un bean de type `PlatformTransactionManager` est utilisé par défaut, mais l'attribut `transactionManager` de l'annotation `@TransactionConfiguration` permet de le préciser explicitement en donnant son nom.

```
<bean id="transactionManager"
    class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
    <property name="dataSource" ref="dataSource" />
</bean>
```

Dans JUnit 4.4, la gestion des transactions pour les méthodes de test peut également se faire en dérivant de la classe transactionnelle de support `AbstractTransactionalJUnit4SpringContextTests` fournie par `TestContext`. Puisque `@Transactional` est déjà appliquée au niveau de cette classe, nous n'avons plus besoin de l'activer à nouveau. De même, il n'est plus utile de préciser `SpringJUnit4ClassRunner` pour le test, car il est hérité du parent.

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.→
AbstractTransactionalJUnit4SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests
    extends AbstractTransactionalJUnit4SpringContextTests {
    ...
}
```

Gérer des transactions avec le framework `TestContext` dans JUnit 3.8

Dans JUnit 3.8, nous pouvons également utiliser le framework `TestContext` pour créer des tests qui s'exécutent au sein de transactions. Toutefois, la classe de test doit dériver de la classe `AbstractTransactionalJUnit38SpringContextTests` fournie par ce framework.

```
package com.apress.springrecipes.bank;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit38.→
AbstractTransactionalJUnit38SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit38ContextTests
    extends AbstractTransactionalJUnit38SpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;
```

```
protected void setUp() throws Exception {
    accountService.createAccount(TEST_ACCOUNT_NO);
    accountService.deposit(TEST_ACCOUNT_NO, 100);
}

// tearDown() n'est plus utile.
...
}
```

Gérer des transactions avec le framework TestContext dans TestNG

Pour créer des tests TestNG au sein de transactions, la classe de test doit dériver de la classe `AbstractTransactionalTestNGSpringContextTests` fournie par `TestContext`.

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.Context
    AbstractTransactionalTestNGSpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceTestNGContextTests
    extends AbstractTransactionalTestNGSpringContextTests {

    private static final String TEST_ACCOUNT_NO = "1234";

    @Autowired
    private AccountService accountService;

    @BeforeMethod
    public void init() {
        accountService.createAccount(TEST_ACCOUNT_NO);
        accountService.deposit(TEST_ACCOUNT_NO, 100);
    }

    // cleanup() n'est plus utile.
    ...
}
```

12.7 Accéder à une base de données dans des tests d'intégration

Problème

Lorsque nous créons des tests d'intégration pour une application qui accède à une base de données, en particulier si elle est développée avec un framework ORM, nous souhaitons accéder directement à la base de manière à préparer les données de test et à valider les données après l'exécution d'une méthode de test.

Solution

Les fonctions Spring de prise en charge des tests permettent de créer et de fournir un template JDBC pour que nous puissions effectuer des opérations de bases de données dans nos tests.

Avec la prise en charge ancienne de JUnit 3.8 dans les versions de Spring antérieures à la version 2.5, notre classe de test peut étendre la classe de base `AbstractTransactionalDataSourceSpringContextTests`, qui est une sous-classe de `AbstractTransactionalSpringContextTests`, pour accéder au travers de la méthode `getJdbcTemplate()` à l'instance de `JdbcTemplate` déjà créée. Une source de données et un gestionnaire de transactions doivent être déclarés correctement dans le fichier de configuration des beans.

Dans le framework `TestContext` de Spring 2.5, la classe de test peut étendre l'une des classes transactionnelles fournies par `TestContext` pour accéder à l'instance de `SimpleJdbcTemplate` existante. Ces classes ont également besoin d'une source de données et d'un gestionnaire de transactions dans le fichier de configuration des beans.

Explications

Accéder à une base de données avec la prise en charge ancienne de JUnit 3.8

Lorsque les tests sont créés avec la prise en charge ancienne de JUnit 3.8, la classe de test doit dériver de `AbstractTransactionalDataSourceSpringContextTests` pour obtenir une instance de `JdbcTemplate` via la méthode `getJdbcTemplate()`, dans le but de préparer et de valider les données de test.

```
package com.apress.springrecipes.bank;

import org.springframework.test.base
                           AbstractTransactionalDataSourceSpringContextTests;

public class AccountServiceJUnit38LegacyTests
    extends AbstractTransactionalDataSourceSpringContextTests {
    ...
    protected void setUpInTransaction() throws Exception {
        getJdbcTemplate().update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            new Object[] { TEST_ACCOUNT_NO, 100 });
    }

    public void testDeposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        double balance = (Double) getJdbcTemplate().queryForObject(
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
            new Object[] { TEST_ACCOUNT_NO }, Double.class);
        assertEquals(balance, 150.0);
    }
}
```

```
public void testWithDraw() {
    accountService.withdraw(TEST_ACCOUNT_NO, 50);
    double balance = (Double) getJdbcTemplate().queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
        new Object[] { TEST_ACCOUNT_NO }, Double.class);
    assertEquals(balance, 50.0);
}
}
```

Outre `getJdbcTemplate()`, cette classe fournit des méthodes pour dénombrer ou supprimer des lignes d'une table et exécuter un script SQL. Pour plus de détails, consultez la documentation JavaDoc de cette classe.

Accéder à une base de données avec le framework TestContext

Lorsque les tests sont créés avec le framework `TestContext`, nous étendons la classe adéquate de `TestContext` de manière à utiliser une instance de `SimpleJdbcTemplate` via un champ protégé. Pour JUnit 4.4, il s'agit de la classe `AbstractTransactionalJUnit4SpringContextTests`, qui fournit également des méthodes pour dénombrer ou supprimer des lignes d'une table et exécuter un script SQL.

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.→
AbstractTransactionalJUnit4SpringContextTests;
@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests
    extends AbstractTransactionalJUnit4SpringContextTests {
...
@Before
public void init() {
    simpleJdbcTemplate.update(
        "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
        TEST_ACCOUNT_NO, 100);
}
@Test
public void deposit() {
    accountService.deposit(TEST_ACCOUNT_NO, 50);
    double balance = simpleJdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
        Double.class, TEST_ACCOUNT_NO);
    assertEquals(balance, 150.0, 0);
}
@Test
public void withDraw() {
    accountService.withdraw(TEST_ACCOUNT_NO, 50);
    double balance = simpleJdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
        Double.class, TEST_ACCOUNT_NO);
    assertEquals(balance, 50.0, 0);
}
}
```

Dans JUnit 3.8, nous étendons `AbstractTransactionalJUnit38SpringContextTests` pour utiliser une instance de `SimpleJdbcTemplate` par le biais d'un champ protégé.

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit38.*;
import org.springframework.test.context.junit38.*;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit38ContextTests
    extends AbstractTransactionalJUnit38SpringContextTests {
    ...
    protected void setUp() throws Exception {
        simpleJdbcTemplate.update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            TEST_ACCOUNT_NO, 100);
    }

    public void testDeposit() {
        accountService.deposit(TEST_ACCOUNT_NO, 50);
        double balance = simpleJdbcTemplate.queryForObject(
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
            Double.class, TEST_ACCOUNT_NO);
        assertEquals(balance, 150.0);
    }

    public void testWithDraw() {
        accountService.withdraw(TEST_ACCOUNT_NO, 50);
        double balance = simpleJdbcTemplate.queryForObject(
            "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
            Double.class, TEST_ACCOUNT_NO);
        assertEquals(balance, 50.0);
    }
}
```

Dans TestNG, nous étendons `AbstractTransactionalTestNGSpringContextTests` pour utiliser une instance de `SimpleJdbcTemplate`.

```
package com.apress.springrecipes.bank;
...
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.testng.*;
import org.springframework.test.context.testng.*;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceTestNGContextTests
    extends AbstractTransactionalTestNGSpringContextTests {
    ...
    @BeforeMethod
    public void init() {
        simpleJdbcTemplate.update(
            "INSERT INTO ACCOUNT (ACCOUNT_NO, BALANCE) VALUES (?, ?)",
            TEST_ACCOUNT_NO, 100);
    }
}
```

```
@Test
public void deposit() {
    accountService.deposit(TEST_ACCOUNT_NO, 50);
    double balance = simpleJdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
        Double.class, TEST_ACCOUNT_NO);
    assertEquals(balance, 150, 0);
}

@Test
public void withdraw() {
    accountService.withdraw(TEST_ACCOUNT_NO, 50);
    double balance = simpleJdbcTemplate.queryForObject(
        "SELECT BALANCE FROM ACCOUNT WHERE ACCOUNT_NO = ?",
        Double.class, TEST_ACCOUNT_NO);
    assertEquals(balance, 50, 0);
}
```

12.8 Utiliser les annotations communes de Spring pour les tests

Problème

Puisque JUnit 3.8, contrairement à JUnit 4, ne prend pas en charge les annotations de test, nous devons souvent implémenter manuellement les tâches de test communes, comme s'attendre au lancement d'une exception, répéter plusieurs fois une méthode de test, s'assurer qu'une méthode de test se termine en un temps précis, etc.

Solution

Spring fournit un ensemble d'annotations communes qui simplifient la création d'un test. Elles sont propres à Spring, mais indépendantes du framework de test sous-jacent. Parmi ces annotations, les suivantes se révèlent très utiles pour les opérations de test communes, mais elles ne sont acceptées qu'avec JUnit (3.8 et 4.4) :

- **@Repeat.** Elle indique qu'une méthode de test doit être exécutée à plusieurs reprises. Le nombre est précisé dans la valeur de l'annotation.
- **@Timed.** Elle indique qu'une méthode de test doit se terminer en un temps déterminé, donné en millisecondes. Dans le cas contraire, le test échoue. La période de temps inclut les répétitions de la méthode de test, ainsi que toute méthode d'initialisation et de nettoyage.
- **@IfProfileValue.** Elle indique qu'une méthode de test doit s'exécuter uniquement dans un environnement de test particulier. Cette méthode de test ne s'exécute que si la valeur de profil actuelle correspond à celle indiquée. Nous pouvons préciser plusieurs valeurs afin que la méthode de test s'exécute dans plusieurs environne-

ments. Par défaut, la classe `SystemProfileValueSource` est utilisée pour retrouver des propriétés système sous forme de valeurs de profil, mais nous pouvons créer notre propre implémentation de `ProfileValueSource` et l'indiquer dans l'annotation `@ProfileValueSourceConfiguration`.

- **@ExpectedException.** Elle a un effet équivalent à la prise en charge des exceptions attendues dans JUnit 4 et TestNG. Toutefois, puisque JUnit 3.8 ne dispose pas de cette prise en charge, elle représente un bon complément au test des exceptions dans JUnit 3.8.

Avec la prise en charge ancienne de JUnit 3.8 dans les versions de Spring antérieures à la version 2.5, notre classe de test peut étendre la classe de base `AbstractAnnotationAwareTransactionalTests`, qui est une sous-classe de `AbstractTransactionalDataSourceSpringContextTests`, de manière à utiliser les annotations communes de Spring pour les tests.

Dans le framework `TestContext` de Spring 2.5, nous pouvons employer des annotations de test de Spring en dérivant de l'une des classes de support de `TestContext`. Si nous n'étendons pas une telle classe, nous pouvons néanmoins utiliser ces annotations à condition d'exécuter le test JUnit 4.4 avec `SpringJUnit4ClassRunner`.

Expllications

Utiliser les annotations communes de test avec la prise en charge ancienne de JUnit 3.8

Lorsque les tests sont créés avec la prise en charge ancienne de JUnit 3.8, la classe de test doit dériver de `AbstractAnnotationAwareTransactionalTests` pour utiliser les annotations communes de test de Spring.

```
package com.apress.springrecipes.bank;

import org.springframework.test.annotation.➥
    AbstractAnnotationAwareTransactionalTests;
import org.springframework.test.annotation.Repeat;
import org.springframework.test.annotation.Timed;

public class AccountServiceJUnit38LegacyTests
    extends AbstractAnnotationAwareTransactionalTests {
    ...
    @Timed(milliseconds = 1000)
    public void testDeposit() {
        ...
    }
    @Repeat(5)
    public void testWithdraw() {
        ...
    }
}
```

Utiliser les annotations communes de test avec le framework TestContext

Lorsque les tests JUnit 4.4 sont créés avec le framework TestContext, nous pouvons utiliser des annotations de test de Spring, lorsque le test est exécuté avec `SpringJUnit4ClassRunner`, ou étendre une classe de `TestContext` pour JUnit 4.4.

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.annotation.Repeat;
import org.springframework.test.annotation.Timed;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit4.➥
    AbstractTransactionalJUnit4SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit4ContextTests
    extends AbstractTransactionalJUnit4SpringContextTests {
    ...
    @Test
    @Timed(milliseconds = 1000)
    public void deposit() {
        ...
    }

    @Test
    @Repeat(5)
    public void withdraw() {
        ...
    }
}
```

Pour créer des tests JUnit 3.8 avec le framework TestContext en utilisant les annotations de test de Spring, nous devons étendre une classe de `TestContext`.

```
package com.apress.springrecipes.bank;
...
import org.springframework.test.annotation.Repeat;
import org.springframework.test.annotation.Timed;
import org.springframework.test.context.ContextConfiguration;
import org.springframework.test.context.junit38.➥
    AbstractTransactionalJUnit38SpringContextTests;

@ContextConfiguration(locations = "/beans.xml")
public class AccountServiceJUnit38ContextTests
    extends AbstractTransactionalJUnit38SpringContextTests {
    ...
    @Timed(milliseconds = 1000)
    public void testDeposit() {
        ...
    }

    @Repeat(5)
    public void testWithdraw() {
        ...
    }
}
```

12.9 En résumé

Dans ce chapitre, nous avons pris connaissance des concepts et des techniques de base employés dans le test des applications Java. JUnit et TestNG sont les frameworks de test les plus utilisés sur la plate-forme Java. JUnit 4 apporte plusieurs améliorations majeures par rapport à JUnit 3.8, dont la plus importante est la prise en charge des annotations. TestNG est également un puissant framework de test à base d'annotations.

Les tests unitaires sont employés pour tester une seule unité de programmation, qui correspond généralement à une classe ou à une méthode dans les langages orientés objet. Pour tester une unité qui dépend d'autres unités, nous utilisons des bouchons et des objets simulacres pour simuler les dépendances. Cela permet de simplifier les tests. À l'opposé, les tests d'intégration sont employés pour tester plusieurs unités comme un tout.

Les contrôleurs de la couche web sont généralement complexes à tester. Spring fournit des objets simulacres pour l'API des servlets afin que nous puissions facilement simuler des objets de requête et de réponse web dans le test d'un contrôleur web. Spring offre également un ensemble d'assertions pour vérifier l'objet `ModelAndView` retourné par un contrôleur. Dans Spring 2.5, puisque les contrôleurs ne dépendent pas de l'API Spring MVC ni de l'API des servlets, ils sont plus faciles à tester.

Les fonctions Spring de prise en charge des tests peuvent gérer des contextes d'application pour nos tests en les chargeant à partir des fichiers de configuration des beans et en les mettant dans un cache pour les réutiliser entre plusieurs exécutions d'un test. Avant sa version 2.5, Spring fournissait une prise en charge des tests spécifique à JUnit 3.8. Par l'intermédiaire du framework `TestContext`, Spring 2.5 apporte une prise en charge équivalente.

Nous pouvons accéder au contexte d'application géré dans nos tests et injecter automatiquement des fixtures de test dans le contexte d'application. Par ailleurs, si nos tests mettent à jour une base de données, Spring peut les exécuter dans des transactions afin que les modifications apportées par une méthode de test soient annulées et n'affectent donc pas la méthode de test suivante. Spring peut également créer un template JDBC pour que nous puissions préparer et valider les données de test dans la base de données.

Spring fournit un ensemble d'annotations de test pour simplifier la création d'un test. Elles sont spécifiques à Spring, mais indépendantes du framework de test sous-jacent. Toutefois, elles ne peuvent être employées qu'avec JUnit.

Index

Symboles

- &&, opérateur [199](#)
- .NET, projet (Spring Portfolio) [27](#)
- ? (point d'interrogation) pour les paramètres SQL [238](#)

A

- AbstractController**, classe [336](#)
- AbstractDependencyInjectionSpring-ContextTests**, classe [455](#)
- AbstractFactoryBean**, classe [103](#)
- AbstractJExcelView**, classe [389](#)
- AbstractJUnit4SpringContextTests**, classe [452](#)
- AbstractPdfView**, classe [389](#)
- AbstractSingleSpringContextTests**, classe [448](#)
- AbstractTestNGSpringContextTests support**, classe [458](#)
- AbstractTransactionalDataSource-SpringContextTests**, classe [466](#)
- AbstractTransactionalJUnit38Spring-ContextTests**, classe [468](#)
- AbstractTransactionalJUnit4Spring-ContextTests**, classe [464, 467](#)
- AbstractTransactionalSpringContext-Tests**, classe [459, 462](#)
- AbstractTransactionalTestNGSpring-ContextTests**, classe [465, 468](#)
- AbstractWizardFormController**, classe [374](#)
- AcceptHeaderLocaleResolver**, classe [347](#)

accept-language, en-tête [347](#)

Accès aux EJB 3.0 avec Spring [1](#)

ACID, propriétés des transactions [253](#)

Actions

multiples, regrouper dans des contrôleur
[383-388](#)

Struts

accéder au contexte d'application [411-413](#)

déclarer dans le fichier de configuration des beans [414-415](#)

ActionServlet (Struts) [410](#)

ActionSupport, classe (Spring) [409, 412-413](#)

Affichage des beans

sous forme de graphe [38](#)
Spring Explorer [37-38](#)

After returning, greffons [160, 177](#)

After throwing, greffons [161-162, 178-179](#)

After, greffons [177](#)

afterCompletion(), méthode de rappel [344](#)

@AfterReturning, annotation [178](#)

afterThrowing(), méthode de rappel [162](#)

@AfterThrowing, annotation [179](#)

Ajax (Asynchronous JavaScript and XML), DWR et [404](#)

AJDT (AspectJ Development Tools) [33](#)

Annotations

AnnotationBeanConfigurerAspect [208, 289](#)

AnnotationMethodHandlerAdapter, classe [395, 400](#)

AnnotationSessionFactoryBean, classe [309](#)

Annotations (suite)

- AspectJ
 - AnnotationTransactionAspect 288, 291
 - déclarer des aspects avec 175-180
 - prise en charge dans Spring, activer 172
- communes pour les tests dans Spring 469-471

configuration basée sur les annotations (Spring 2.0) 24

développer des contrôleurs avec 393-400

JPA, utiliser dans Hibernate 300-302

Annulation, attribut transactionnel 285-286**AOP (Aspect-Oriented Programming),
Voir POA (Programmation orientée aspect)****Apache**

bibliothèques

Commons 142-144

POI 389

Derby 214

projet Commons 17

Struts

1.x, intégrer à Spring 409-415

définition 403

Tomcat 6.0 340

API de transaction

attribut de niveau d'isolation, fixer dans 284-285

gestionnaire de transactions 260-262

ApplicationEvent, classe 127**ApplicationListener, interface 127, 129****Applications web**

accéder à Spring dans 404-408

développer avec MVC 328-331

génériques, accéder à Spring dans 404-408

nouvelles

configurer (MVC) 332

contrôleurs pour (MVC) 335-338

déployer 340

fichiers de configuration pour (MVC) 333-335

Arguments, valider (application de calculatrice) 145**ArithmeticCalculator, interface 176, 186-189****Around, greffons 162-164, 180****AspectJ**

annotations

déclarer des aspects avec 175-180

prise en charge dans Spring 172

vue d'ensemble 171

AspectJExpressionPointcutAdvisor 166

aspects

AnnotationBeanConfigurerAspect 289

AnnotationTransactionAspect 288

configurer dans Spring 205-206

bibliothèques 174

configurer dans Spring 205-206

expressions de points d'action 166

écrire 186-191

guide de programmation 186

télécharger 140

tisseur 204

à la compilation 200

au chargement 201

vue d'ensemble 171

aspectOf(), méthode statique de fabrique 205-206**Aspects**

déclarer avec des configurations XML 197-200

définition 140

préférence 182-184

Assistance au contenu dans les fichiers de configuration 39**Atomicité, propriété des transactions 253****Attributs**

de session, déterminer la localisation avec 348

transactionnels, Voir Gestion des transactions

autodetect, mode 65**autowire, attribut 61-62, 66**

- @Autowired**
 annotation 210, 289
 liaison automatique des beans 66-72
- AutowiredAnnotationBeanPostProcessor, classe 66**
- Aware, interfaces 116**
- B**
- Bases de données**
 accéder dans des tests d'intégration 465-469
 configurer une application (JDBC) 214-215
 interroger avec SimpleJdbcTemplate 235-237
 mettre à jour
 avec SimpleJdbcTemplate 234-235
 avec un configurateur de requête 224
 avec un créateur de requête 222-223
 en série 225-226
 templates JDBC
 pour l'interrogation 226-231
 pour la mise à jour 221
- Batch, projet (Spring Portfolio) 27**
- BeanCreationException, classe 100, 102**
- BeanDoc, projet (Spring Portfolio) 27**
- BeanNameAutoProxyCreator, classe 168**
- BeanNameGenerator, interface 92**
- BeanNameUrlHandlerMapping, classe 340**
- BeanPostProcessor, interface 118**
- BeanPropertyRowMapper, classe 228**
- BeanPropertySqlParameterSource, classe 239**
- Beans**
 afficher
 dans Spring Explorer 37-38
 dans un graphe 38
 anonymes 45
 chercher 40
 configuration, externaliser 122-124
 conscients du conteneur 116-117
- correspondance de requêtes web par les noms de beans 341
 création automatique de proxies 167-169 créer
 avec des méthodes d'instance de fabrique 100-102
 avec des méthodes statiques de fabrique 99-100
 avec un bean de fabrique Spring 102-104
 par invocation de constructeurs 96-99
 dans le conteneur Spring IoC 32
 de fabrique
 définir des collections avec 84-86
 définition 84
 FactoryBean, interface 102
 FieldRetrievingFactoryBean, classe 104-105
 déclarer
 à partir de champs statiques 104-105
 à partir de propriétés d'objets 106-107
 dans un fichier de configuration 44-45
 des beans JSF gérés dans un fichier de configuration 420
 enfant, définition 72
 exposer pour une invocation à distance 422-423
 initialisation et destruction, personnaliser 110-115
 injecter dans des objets de domaine 207-210
 internes, déclarer 56
 introduire
 des comportements 191-194
 des états 194-196
 liaison automatique, *Voir* Liaison automatique des beans par configuration XML
 motifs de noms 189
 obtenir depuis des conteneurs IoC 48-49
 parent 72, 81-82
 portée, fixer 108-110
 postprocesseur de beans, créer 118-122
 résoudre en JSF 419
- before(), méthode 157**
- @Before, annotation 175**

Before, greffons 157-159, 175-177
bindOnNewForm, propriété 369
Bouchons
 définition 435
 tests unitaires, créer avec 438-442
buildPdfDocument(), méthode 392

C

Champs statiques, déclarer des beans à partir de 104-105
Chemin d'accès aux classes
 ClassPathXmlApplicationContext, classe 48
 configurer 30
Classes
 de beans, créer 43
 de support pour l'accès aux données 313
 dépendantes, créer des tests unitaires pour 438-443
 isolées, créer des tests unitaires pour 435-438
 persistantes, définition 294
Code
 dispersion 147
 mélange 146
Codes d'erreur, base de données 246-249
Cohérence, propriété des transactions 253
Collections
 autonomes, définition 86
 classe concrète pour, spécifier 85
 dans un bean parent, fusionner 81-82
 définir
 avec des beans de fabrique et le schéma util 84-86
 pour des propriétés de beans 75-82
 types de données pour les éléments de 82-84
@Column, annotation 301
commit(), méthode (JDBC) 257-258
CommonAnnotationBeanPostProcessor, classe 115
common-annotations.jar 72

Commons BeanUtils, bibliothèque 17
Commons Logging, bibliothèque 17
Complex(int, int), constructeur 204
ComplexFormatter, classe 207-208
@Component, annotation 89
Comportements
 introduire dans des beans 191-194
 vérifier 435
Composants
 définition 2
 gérer avec des conteneurs 2-7
 scan, Voir Scan de composants
@Configurable, annotation 208, 289
Configurateur de requête, mise à jour de la base de données avec 224
Configuration des beans
 dans le conteneur Spring IoC
 ambiguïté sur le constructeur, lever 49-52
 configuration de bean, hériter 72-75
 configuration XML, lier
 automatiquement des beans avec 61-65
 définir des collections avec des beans de fabrique et le schéma util 84-86
 définir des collections pour les propriétés de beans 75-82
 définir des propriétés de beans par raccourci 46
 dépendances, vérifier 56-58
 fichier de configuration, déclarer des beans dans 44-45
 instancier un conteneur Spring IoC 46-49
 liaison automatique des beans avec @Autowired et @Resource 66-72
 propriétés, contrôler avec @Required 59-61
 références de beans, préciser 52-56
 scan de composants dans le chemin d'accès aux classes 87
 types de données pour les éléments de collection 82-84
 vue d'ensemble 42
 externaliser 122-124

- Conseiller, définition** 165
- Constructeurs**
- ambiguïté, résoudre 49-52
 - arguments, préciser des références de beans pour 55
 - éléments <constructor-arg> 45, 50, 55, 100, 101
 - invoyer pour créer des beans 96-99
 - liaison automatique des beans 64
- Conteneurs**
- beans conscients de l'existence des 116-117
 - composants
 - définition 2
 - gérer 2-7
 - configurer avec des fichiers de configuration 16-19
 - définition 2
 - employer 5-7
 - Localisateur de service, design pattern 8-9
 - séparer l'interface de l'implémentation 2-4
 - Voir aussi* Spring IoC, conteneurs
- Context (module), définition** 22
- <**context:annotation-config**>, élément 66, 115, 122, 210
- <**context:component-scan**>, élément 325, 394
- <**context:load-time-weaver**>, élément 204
- <**context:spring-configured**>, élément 208
- contextConfigLocation**, paramètre de servlet 333
- Contexte**
- d'application
 - accéder à, avec la prise en charge ancienne de JUnit 3.8 450-451
 - accéder à, avec un framework TestContext 453
 - accéder à, dans des actions Struts 411-413
 - ApplicationContextAware, interface 135, 448, 451, 454
 - charger dans une application Struts 410
 - définition 22, 46
 - gérer dans des tests d'intégration 448-454
 - interface 48, 124, 126
 - de test, définition 428
- ContextLoaderListener, classe** 334, 404-406, 409, 410
- ContextLoaderPlugin (Struts)** 411, 412
- Contrôleurs**
- Controller, interface 335
 - créer avec des vues paramétrées 359-361
 - de formulaire
 - créer 362-366
 - design pattern Post-Redirect-Get, appliquer 366
 - développer 397-400
 - données de formulaire, valider 372-374
 - données de référence d'un formulaire 369-371
 - objets de commande, initialiser 367-369
 - simples, classe SimpleFormController (MVC) 361-362, 367, 371-372
 - types personnalisés, lier des propriétés 371-372
 - Voir aussi* Formulaires assistants, contrôleurs
 - vue d'ensemble 361
 - définition 327-328
 - développer avec des annotations 393-400
 - frontaux 328
 - monoaction, développer 395-397
 - multiactions
 - créer 385-387
 - développer 395-397
 - MultiActionController, classe 383
 - MVC
 - à base d'annotations, tests unitaires 446
 - tests unitaires 443-446
 - pour les nouvelles applications web 335-338
 - regrouper plusieurs actions dans 383-388
- @Controller, annotation** 91, 393

- ControllerClassNameHandlerMapping**, classe [341-342, 361](#)
- Cookies**
CookieLocaleResolver, classe [348](#)
paramètres régionaux, déterminer avec [348](#)
- Core (module), définition** [22](#)
- Correspondance objet-relationnel (ORM), Voir ORM (Object/Relational Mapping), frameworks**
- Correspondances, Hibernate XML** [296-300](#)
- Counter, interface** [195-196](#)
- court-views.xml, fichier de configuration** [353](#)
- createProduct(), méthode statique de fabrique** [99](#)
- Créateurs**
automatiques de proxies [167](#)
de requêtes, mettre à jour la base de données avec [222-223](#)
- CustomDateEditor, classe** [131-132](#)
- CustomEditorConfigurer, classe** [130, 131](#)
- CustomSQLErrorCodesTranslation, bean** [249](#)
- D**
- DAO (Data Access Object)**
classes de support [317-318](#)
design pattern [216](#)
exécuter [220](#)
implémenter avec JDBC [216-218](#)
interface [87](#)
- DateFormat.parse(), méthode** [130](#)
- @DeclareParents, annotation** [193](#)
- DefaultAdvisorAutoProxyCreator, classe** [169](#)
- DefaultAnnotationHandlerMapping, classe** [395](#)
- DefaultTransactionDefinition**
classe [261](#)
objet [285](#)
- DelegatingActionProxy (Struts)** [415](#)
- DelegatingRequestProcessor (Struts)** [415](#)
- DelegatingVariableResolver (JSF)** [416, 420](#)
- Déploiement de nouvelles applications web** [340](#)
- Derby, Apache** [214](#)
- Design patterns**
DAO [216](#)
GoF (Gang of Four) [149](#)
Localisateur de service [8](#)
Post-Redirect-Get [366](#)
- destroy-method, attribut** [114](#)
- Destruction et initialisation des beans** [110-115](#)
- Détection automatique, liaison automatique des beans par** [65](#)
- DI (Dependency Injection)**
appliquer [10-12](#)
définition [10](#)
DependencyInjectionTestExecutionListener, classe [448](#)
types de [12-16](#)
- DirtiesContextTestExecutionListener, classe** [448](#)
- DispatcherServlet, classe** [328-330](#)
- Dispersion de code** [147](#)
- DisposableBean, interface** [113-114](#)
cycle de vie [111](#)
- Données**
accès aux
classes de support [313](#)
DataAccessException, classe [261](#)
DataAccessException, hiérarchie [244, 316, 321](#)
éléments de programmation [296](#)
gestion des exceptions, personnaliser [248-249](#)
- DataIntegrityViolationException, classe [246](#)
- DataSourceTransactionManager, classe [259, 262](#)
- de formulaire, valider [372-374](#)

- extraire
 - avec des gestionnaires de rappel pour les lignes 227
 - par correspondance de ligne 228-229
- sources de, configurer dans Spring 218-220
- types de données, convertir 132
- DriverManagerDataSource, classe 218-220, 255**
- Durabilité, propriété des transactions 253**
- DWR (Direct Web Remoting)**
 - définition 404
 - DwrServlet 421
 - DwrSpringServlet 423
 - éléments
 - <dwr:configuration> 424
 - <dwr:remote> 420, 424
 - intégrer Spring à 420-424
- Dynamic Modules, projet (Spring Portfolio) 27**

- E**
- EasyMock, bibliothèque 441-442**
- Eclipse**
 - créer un projet 34
 - Data Tools Platform (DTP) 215
 - Mylyn 33
 - plug-in 26, 33
 - Web Tools Platform (WTP) 28
- Écouteur d'exécution du test, définition 428**
- Éditeurs de propriétés**
 - convertir des types date 132
 - enregistrer 129-132
 - personnalisés, créer 133-134
- Ensembles (java.util.Set), définition 78**
- Enterprise Integration (module), définition 24**
- Entité (classe d'), définition 294**
- @Entity, annotation 301**
- equals(), méthode 78**

- Erreurs d'objets 373**
- États**
 - introduire dans des beans 194-196
 - vérification 435
- Événements**
 - d'application, communiquer par 129
 - écouter 129
 - publier 128
- Excel, créer des vues 391-392**
- Exceptions**
 - d'accès aux données, personnaliser la gestion 248-249
 - FileNotFoundException, classe 119
 - gérer dans le framework Spring JDBC 244-248
 - IllegalArgumentException, classe 100
 - lancer 178-179, 296, 325
- Exemples d'applications**
 - calcul des intérêts 429
 - calculatrice (préoccupations transversales) 141
 - chariot d'achat 108-110
 - classement des produits par date 130
 - distance entre deux villes 409
 - enregistrement de véhicules 214
 - génération de rapports 2-4
 - librairie en ligne 253-257
 - magasin en ligne 96-99
 - système de gestion des cours 294
- Expression régulières, points d'action 166**

- F**
- Fabriques**
 - de beans
 - définition 46
 - instancier 47
 - de gestionnaires d'entités (JPA) 305, 310-312
 - de sessions, configurer Hibernate 306-310
- FacesServlet (JSF) 417**

Fichiers de configuration

- configurer des conteneurs 16-19
- des beans
 - actions Struts, déclarer dans 414-415
 - assistance de contenu, utiliser 39
 - beans JSF gérés, déclarer dans 420
 - configurer DWR dans 423-424
 - créer dans Spring IDE 31-32
 - valider 39
- pour les nouvelles applications web 333-335
- XML, déterminer des vues à partir de 352

FieldRetrievingFactoryBean, classe 104-105**File.mkdirs(), méthode** 120**FileNotFoundException, classe** 119**FileSystemXmlApplicationContext, classe** 48**Fixtures de test**

- initialiser 430
- injecter
 - avec le framework TestContext 456-458
 - dans des tests d'intégration 455-459

fmt, bibliothèque de balises 339**Formulaires**

- assistants, contrôleurs
 - créer 379-381
 - formulaires multipages, gérer avec 374-375
 - valider les données 381-383
- contrôleur, *Voir* Contrôleurs de formulaire
 - <form>, balise 365, 377
 - formBackingObject(), méthode 379
 - multipages 374-375

G**Gestion des transactions**

- attributs
 - d'annulation 285-286
 - d'isolation 277-285

de lecture seule 286-288

de propagation 271-277

de temporisation 286-288

commit() et rollback(), méthodes JDBC 257-258

implémentations de gestionnaires, choisir 258-259

par déclaration

avec les greffons transactionnels 268-270

fondamentaux 252

Spring AOP classique 265-268

@Transactional, annotation 270-271

par programmation

API de gestionnaire de transactions 260-262

fondamentaux 252

templates de transaction 262-265

problèmes 252-258

transactions concurrentes 277-280

tissage au chargement pour 288-291

vue d'ensemble 251-252

Gestionnaires

associer

des requêtes web à 340

des URL à des méthodes de 387-388

d'entités, fabrique de (JPA) 310-312

HandlerExceptionResolver, interface 355

HandlerMapping, interface 329, 340

getApplicationContext(), méthode 448**getAsString(), méthode** 133**getBean(), méthode** 108-110**getConfigLocations(), méthode** 450**getConfigPath(), méthode** 450**getCurrentSession(), méthode** 319**getInputStream(), méthode** 136**getJdbcTemplate(), méthode** 466**getResource(), méthode** 135-136**getTransaction(), méthode** 260-262**getWebApplicationContext(), méthode** 412**GoF (Gang of Four), design patterns** 149

Greffons

- After 177
- After returning 177
- After throwing 178-179
- Around 180
- Before 175-177
- déclarer 199
- introduction 191
- Spring AOP classique
 - After returning 160
 - After throwing 161-162
 - Around 162-164
 - Before 157-159
 - définition 155
 - types de 156
- transactionnels
 - attribut de niveau d'isolation, fixer dans 284-285
 - gérer par déclaration 268-270

H

- <h:form>, élément (JSP) 419
- <h:inputText>, élément (JSP) 419
- hbm2ddl.auto, propriété 297
- Héritage de la configuration de bean 72-75
- Hibernate, framework ORM**
 - étendre une classe Hibernate de support de DAO 317-318
 - fabrique de sessions, configurer 306-310
 - fondamentaux 293
 - Hibernate Entity-Manager 294
 - HibernateCallback, interface 316
 - HibernateCourseDao, classe 313
 - HibernateTemplate, classe 313, 316
 - HibernateTransactionManager, classe 259, 315
 - JPA avec Hibernate comme moteur 302-306
 - objets persistants avec l'API d'Hibernate 296, 300, 302
 - templates, utiliser 312-315
- HQL (Hibernate Query Language)** 294

HTTP (HyperText Transfer Protocol)

- en-tête de requête pour déterminer les paramètres régionaux 347
- méthode HTTP GET 398
- méthode POST 399

I

@Id, annotation

IDE

- Java, installer 28
- projet (Spring Portfolio) 26
- Spring, Voir Spring IDE

IllegalArgumentException, classe

@InitBinder, annotation

Initialisation et destruction des beans

InitializingBean, interface

init-method, attribut

Injection

- beans Spring dans des objets de domaine 207
- de contexte JPA 322-325
- de dépendance (DI), Voir DI (Dependency Injection)
- de ressources 137
- par constructeur
 - fondamentaux 13-14
 - vérifier les dépendances 58
- par mutateur
 - exemple 10-11
 - fondamentaux 12-13
 - templates JDBC 232

Installation

- framework Spring 28-29
- Java IDE 28
- Spring IDE 32-34

Integration, projet (Spring Portfolio)

Intercepteurs de gestionnaires

- définition 344
- requêtes web, intercepter avec 344-347

Interfaces

- injection par 14-16
- séparer de l'implémentation 2-4

- InternalPathMethodNameResolver, classe** 387
- InternalResourceViewResolver, classe** 352, 354
- Interrogation**
des bases de données
 plusieurs lignes 229-230
 une valeur 230-231
query(), méthode template (JdbcTemplate) 227
queryForList(), méthode 229
queryForObject(), méthode (JdbcTemplate) 228, 231
- Introduction**
déclarer 199
greffon 191
- Inversion de contrôle (IoC), Voir IOC (Inversion of Control) pour l'obtention de ressource**
- Invocation à distance, exposer des beans pour** 422-423
- InvocationHandler, interface** 150
- invoke(), méthode** 150-151
- IOC (Inversion of Control) pour l'obtention de ressources** 10-12
- IoC, conteneurs**
beans du, obtenir 48-49
Spring, *Voir* Spring IoC, conteneurs
- Isolation**
de transaction, attribut
 définition 277
 fixer 277-285
propriété des transactions 253
- iText, bibliothèque** 389
- J**
- Java**
API
de persistance (JPA), *Voir* JPA (Java Persistence API)
de réflexion 52, 150
bibliothèque de balises standard (JSTL) 328, 339
- Collections, framework 75
- IDE, installer 28
- Java 1.5, utiliser un template simple avec 234-237
- java.beans.PropertyEditor, interface 133
- java.sql.SQLException 244
- java.text.DecimalFormat 82
- java.util.Collection, interface 78
- java.util.Date, propriétés 131
- java.util.List, type 76-77
- java.util.Map 358
- java.util.Properties
 collection 80
 objet 17
- JavaConfig, projet (Spring Portfolio) 27
- javax.sql.DataSource, interface 218
- JDBC, Voir JDBC (Java Database Connectivity)**
plate-forme JEE 1
- JavaScript, Spring, Voir Spring, framework**
- JBoss AOP** 140
- JDBC (Java Database Connectivity)**
commit(), méthode 257-258
configurer
 une base de données d'application 214-215
 une source de données dans Spring 218-220
- DAO**
design pattern 216
exécuter 220
implémenter avec 216-218
exceptions, gérer dans le framework JDBC 244-249
- JdbcTemplate, classe** 221, 232, 234
- mettre à jour des bases de données
 avec un configurateur de requête 224
 avec un créateur de requête 222-223
 en série 225-226
- module, définition 23
- opérations, modéliser avec des objets élémentaires 240-244
- paramètres nommés dans les templates, utiliser 238-240

- propriétés pour la connexion à une base de données d'application 215, 253
 rollback(), méthode 257-258
 templates
 pour interroger des bases de données 226-231
 pour mettre à jour des bases de données 221
 simples, utiliser avec Java 1.5 234-237
 simplifier la création 232-234
 vue d'ensemble 214
- JdbcDaoSupport, classe**
 étendre 233-234
 jdbcTemplate, propriété 232
 méthodes 233
- JDK**
 API de journalisation 144
 installer 28
 version 1.5 28
- JExcelAPI, bibliothèque** 389
- JNDI (Java Naming and Directory Interface)** 1
- jndi-lookup, élément** 220
- JndiObjectFactoryBean, classe** 102, 220
- JPA (Java Persistence API)**
 classe JPA de support de DAO, étendre 318
 fabrique de gestionnaires d'entités, configurer 310-312
 fondamentaux 294
 JpaCallback, interface 316
 JpaTemplate, classe 313, 316, 322, 325
 JpaTransactionManager, classe 258
 objets persistants
 avec des annotations JPA 300-302
 Hibernate comme moteur 302-306
 templates, utiliser 315-317
- JSF (Java Server Faces)**
 définition 404
 intégrer Spring à 416-420
 JSF Reference Implementation 416
- JSP (JavaServer Page)**
 templates 338
 vues, créer 338-340
- JSR-250** 111, 114
- JSTL (Java Standard Tag Library)** 328, 339
- JUnit**
 tester des applications avec 429-432
 version 4 429
 vue d'ensemble 428
- JUnit 3.8, prise en charge ancienne**
 accéder
 à des bases de données 466-467
 à un contexte d'application 450-451, 453
 annotations communes pour les tests, utiliser avec 470
 fixtures de test, injecter 455-456, 458
 transactions, gérer 462, 464
 vue d'ensemble 428
- JUnit 4.4**
 contexte d'application, accéder à 451-453
 fixtures de test, injecter 456-457
 transactions, gérer 462-464
- L**
- Lancer des exceptions** 178-179
- LDAP (Lightweight Directory Access Protocol)**
 projet Spring 27
- Lecture seule, attribut transactionnel** 286-288
- Lectures**
 fantômes 277, 284
 non reproductibles 277, 283
 sales 277
- Liaison automatique des beans**
 avec @Autowired et @Resource
 beans uniques de type compatible 67-68
 par nom 71
 par type avec qualificateur 70-71
 tous les beans de type compatible 69-70
 modes reconnus par Spring 61

- Liaison automatique des beans (suite)**
par configuration XML
par constructeur 64
par détection automatique 65
par nom 63
par type 62-63
vérifier les dépendances 65
vue d'ensemble 61-62
- Lignes (bases de données)**
extraire des données
 avec des gestionnaires de rappel pour les lignes 227
 par correspondance de ligne 228-229 multiples, obtenir 229-230
- Listes (java.util.List), définition** 76-77
- LocalContainerEntityManagerFactoryBean, classe** 311
- LocaleChangeInterceptor, classe** 349
- LocaleResolver, interface** 347
- Localisateur de service, design pattern** 8-9
- LocalSessionFactoryBean, classe** 102
- Log4J, bibliothèque (Apache)** 144-145
- @LoggingRequired, annotation** 188-189
- M**
- <map>, élément** 405
- MappingSqlQuery, classe** 242
- mapRow(), méthode** 228
- MapSqlParameterSource, classe** 239
- MaxCalculator, interface** 192
- Mélange de code** 146
- Messages**
 MessageSource, interface 124-125, 350
 multilingues
 obtenir 124-126
 externaliser 350-351
- MethodBeforeAdvice, interface** 157
- Méthodes**
 correspondance avec des points d'action Spring classique 164-167
- d'instance de fabrique, créer des beans 100-102
de fabrique
 attribut 205
 créer des beans 99-100
 suivi 142
 tracer 142
- MethodInterceptor, interface** 162
- MinCalculator, interface** 192
- Mise à jour**
base de données
 avec des configurateurs de requêtes 224
 avec des créateurs de requêtes 222-223
 avec des requêtes SQL et des valeurs de paramètres 224-225
 avec des templates JDBC 221
 avec SimpleJdbcTemplate 234-235
 en série 225-226
 verrous de mise à jour 282
 objets d'opération (JDBC) 241-242
 update(), méthode (JdbcTemplate) 221, 224-225
- ModelAndView**
 classe 330, 357
 objets 357-359
- @ModelAttribute, annotation** 399
- Modèles, définition** 327
- Modèle-Vue-Contrôleur (MVC), Voir**
 MVC (Modèle-Vue-Contrôleur), framework
- ModelMap, classe** 358
- Modularisation des préoccupations transversales**
 greffons Spring classiques 155-164
 proxies dynamiques 148-154
- Modules**
 framework Spring 22-24
 Modules, projet (Spring Portfolio) 27
- Monoaction, développer des contrôleurs** 395-397
- Motifs de signatures**
 méthode 186
 type 188-189

Mutateurs, références aux beans pour 53-
55

MVC (Modèle-Vue-Contrôleur), framework

- associer
 - des exceptions à des vues 355-356
 - des requêtes web à des gestionnaires 340
 - des requêtes web par des définitions de correspondances personnalisées 342
- contrôleurs
 - créer avec des vues paramétrées 359-361
 - de formulaire, *Voir* Contrôleurs de formulaire
 - de formulaires multipages, *Voir* Formulaires assistants, contrôleurs développer avec des annotations 393-400
 - MVC, tests unitaires 443-446
 - regrouper plusieurs actions dans 383-388
 - correspondance de requêtes web
 - par les noms de beans 341
 - par les noms de classes de contrôleurs 341-342
 - par stratégies multiples 343-344
 - définition 327
 - développer des applications web avec 328-331
 - intercepter des requêtes web avec des intercepteurs de gestionnaire 344-347
 - messages textuels localisés, externaliser 350-351
 - ModelAndView objects, construire 357-359
 - nouvelles applications web
 - configurer 332
 - contrôleurs 335-338
 - déployer 340
 - fichiers de configuration 333-335
 - paramètres régionaux de l'utilisateur, déterminer 348-350
 - vue d'ensemble 327

vues

- déterminer par les noms 351-354
- Excel et PDF, créer 389-393
- JSP, créer 338-340

Mylyn, Eclipse 33

N

NestedRuntimeException, classe 246

new

- créateur (DWR) 422
- opérateur 207

Niveaux d'isolation (transactions)

- READ_UNCOMMITTED / READ_COMMITTED 280-282
- reconnus par Spring 278
- REPEATABLE_READ 282-284
- SERIALIZABLE 284

Nommage des composants scannés 92

Noms de méthodes, points d'action 165-166

noRollbackFor, attribut 285

NullPointerException, classe 113

O

Objets

- cibles, définition 181
- d'opération
 - de type fonction 243-244
 - fonction 243-244
 - interroger 242-243
 - mettre à jour 241-242
- de commande, initialisation 367-369
- de domaine, injecter des beans dans 207-210
- élémentaires, modéliser des opérations JDBC sous forme de 240-244
- persistants (ORM)
 - avec des sessions contextuelles Hibernate 319-322
 - avec des templates Spring ORM 312-318

Objets (suite)

- persistants (ORM)
 - avec injection de contexte JPA [322-325](#)
 - utiliser JPA avec Hibernate comme moteur [302-306](#)
 - utiliser l'API d'Hibernate avec des annotations JPA [300-302](#)
 - utiliser l'API d'Hibernate avec des correspondances XML Hibernate [296-300](#)
- simulacres
 - créer des tests unitaires avec [438-442](#)
 - définition [435](#)
 - pour l'API des servlets [443](#)
- @Order, annotation** [182-184](#)
- Ordered, interface** [182-184](#)
- org.springframework.orm.hibernate, paquetage** [294](#)
- org.springframework.transaction.TransactionDefinition, interface** [272, 277](#)
- ORM (Object/Relational Mapping), framework**
 - éléments de programmation pour l'accès aux données [296](#)
 - fabriques de ressources ORM, configurer [306-312](#)
 - module, définition [24](#)
 - objets persistants, *Voir* Objets persistants (ORM)
 - problèmes avec l'utilisation directe de [294-296](#)
 - vue d'ensemble [293](#)

P**Paiement, exemple de fonction (chariot d'achat)** [111-113](#)**Paramètres**

- de vues, créer des contrôleurs avec [359-361](#)
- nommés, dans des templates JDBC [238-240](#)
- ParameterizableViewController, classe [359-361, 366](#)

ParameterizedRowMapper, interface[235-237](#)**ParameterMethodNameResolver, classe**[388](#)**parse(), méthode** [130](#)**PDF, créer des vues** [392-393](#)**persistence.xml, fichier** [302](#)**@PersistenceContext, annotation** [322](#)**@PersistenceUnit, annotation** [324](#)**Personnalisation**éditeurs de propriétés [133-134](#)gérer les exceptions d'accès aux données [248-249](#)initialisation et destruction de beans [110-115](#)**PlatformTransactionManager (Spring)**[259, 462](#)**Plug-in**Eclipse [26, 33](#)Struts [411, 412](#)**POA (Programmation orientée aspect)**API dans Spring 2.0 [25](#)

éléments

<aop:aspect> [198](#)<aop:config> [198, 268](#)<aop:declare-parents> [199](#)module, définition [23](#)Spring 2.x [171](#)transactions, gérer avec Spring classique [265-268](#)*Voir aussi* Spring AOP classiquevue d'ensemble [139-140](#)**POI (Apache), bibliothèque** [389](#)**Points d'action**attributs [199](#)déclarer [198](#)définitions, réutiliser [184-186](#)

expressions

AspectJ, écrire [186-191](#)combiner [189](#)@Pointcut, annotation [184](#)publics [185](#)

- Spring classique
 - correspondance de méthodes avec 165-167
 - définition 164
 - expression AspectJ de points d'action 166
 - expression régulière de points d'action 166
 - points d'action de nom de méthode 165-166
- Points de jonction**
 - définition 176
 - informations sur, accéder à 181-182
- POJO (Plain Old Java Object)**
 - aspects avec des annotations AspectJ 171
 - définition 41
 - utiliser la POA 140
- populateProtectedVariables, propriété 456**
- Portées**
 - attribut (élément <bean>) 108
 - de bean, fixer 108-110
- Portfolio, projets Spring 26-27**
- Portlet MVC (module), définition 24**
- @PostConstruct, annotation 114-115**
- postHandle(), méthode de rappel 344**
- Postprocesseurs de beans 118-122**
- Post-Redirect-Get, design pattern 366**
- Précédence des aspects 182-184**
- @PreDestroy, annotation 114-115**
- Préfixes**
 - de ressource 136
 - PrefixGenerator, interface 52
 - redirect 354
- preHandle(), méthode de rappel 344**
- Préoccupations transversales**
 - contre préoccupations centrales 146
 - définition 141
 - modulariser
 - avec des greffons Spring classique 155-164
 - avec un proxy dynamique 148-154
 - non modularisées 141-148
- processEntry(), méthode 19**
- processSubmit(), méthode 399**
- Programmation orientée objet (POO) 139**
- Propagation des transactions**
 - attribut 271-277
 - comportements
 - reconnus par Spring 272
 - REQUIRED 274-275
 - REQUIRES_NEW 275-277
- Propriétés**
 - d'objets, déclarer des beans à partir de 106-107
 - de beans
 - définir des collections pour 75-82
 - tables d'association et 78-80
 - de type
 - objet, contrôler 58
 - quelconque, contrôler 58
 - simple, contrôler 57
 - des transactions
 - ACID 253
 - atomicité 253
 - cohérence 253
 - durabilité 253
 - isolation 253
 - fichier de, définition 16
 - java.util.Properties, collection 80
 - PropertiesMethodNameResolver, classe 388
 - PropertyPathFactoryBean, classe 106-107
 - PropertyPlaceholderConfigurer, classe 123-124
 - PropertyUtils, classe 18
 - prototype, portée de bean 108, 110**
 - Proxies**
 - attribut de niveau d'isolation, fixer dans 284-285
 - CGLIB 150
 - créer automatiquement pour des beans 167-169
 - de journalisation, créer 150-153
 - dynamiques 149
 - greffons introduction et 193

Proxies (suite)

- dynamiques
- modulariser des préoccupations transversales 148-154
- prise en charge par le JDK 149
- fondamentaux 148-150
- pour la validation 153-154
- ProxyFactoryBean, classe 102, 157-159, 265-268
- publishEvent()**, méthode 127

Q**@Qualifier, annotation** 70**query(), méthode** 227**R****Raccourcis, définir des propriétés de beans avec** 46**READ_UNCOMMITTED / READ_COMMITTED, niveaux d'isolation** 280-282**Recherches**

- actives, définition 10
- de beans 40
- localisateur de service et 8-9

redirect, préfixe 354**<ref>, élément** 52**Références à des beans, préciser**

- pour des arguments de constructeur 55
- pour des mutateurs 53-55

RegexpMethodPointcutAdvisor, classe 166**REPEATABLE_READ, niveau d'isolation** 282-284**Réertoires de l'installation de Spring** 29**@Repository**

- annotation 321, 325
- stéréotype 90

RequestContextListener, classe 420**@RequestMapping, annotation** 393**@RequestParam, annotation** 398**Requêtes web**

- associer
 - à des gestionnaires 340
 - avec des définitions de correspondances personnalisées 342
- correspondances
 - avec les noms de beans 341
 - avec les noms de classes des contrôleurs 341-342
 - par stratégies multiples 343-344
- intercepter avec des intercepteurs de gestionnaire 344-347

@Required, annotation, vérifier des propriétés avec 59**REQUIRED, comportement de la propagation** 274-275**RequiredAnnotationBeanPostProcessor, classe** 59-61**REQUIRES_NEW, comportement de la propagation** 275-277**Résolveurs multiples, pour les vues** 354**@Resource, annotation** 66, 71**Ressources**

- bundles 124, 353
- externes, charger 134-137
- fabriques de, configurer un ORM 306-312
- injecter 137
- préfixes 136
- Resource, interface 47, 135
- ResourceBundleMessageSource, classe 124-125, 350
- ResourceBundleViewResolver, classe 353
- ResourceEditor, classe 307
- ResourceLoaderAware, interface 135

Rich Client, projet (Spring Portfolio) 27**rollback(), méthode (JDBC)** 257-258**S****Scan de composants**

- automatique 89-91
- composants détectés, nommer 92

- dans le chemin d'accès aux classes 87-89
- définition 87
- filtres 91
- Security, projet (Spring Portfolio)** 27
- Séparer l'interface de l'implémentation, principe** 2
- SequenceGenerator, classe** 43, 53-54
- SERIALIZABLE, niveau d'isolation** 284
- Service Locator, design pattern** 8
- @Service, stéréotype 91
- Servlet, API** 407
- ServletRequestUtils, classe** 337
- Session, interface (Hibernate)** 296
- @SessionAttributes, annotation 398
- sessionForm, propriété 369
- SessionLocaleResolver, classe** 348
- Sessions contextuelles d'Hibernate** 319-322
- setAsText(), méthode 133
- setAutoCommit(), méthode 258
- setDataSource(), méthode (classe JdbcDaoSupport) 233
- setDirty(), méthode 451
- setExceptionTranslator(), méthode 249
- setJdbcTemplate(), méthode (classe JdbcDaoSupport) 233
- setTransactionIsolation(), méthode 277
- setupForm(), méthode 398
- Signatures**
 - de méthodes
 - afterThrowing() 162
 - motifs 186
 - de type, motifs 188-189
- SimpleJdbcTemplate, classe** 234-235
 - bases de données
 - interroger 235-237
 - mettre à jour 234-235
 - utiliser avec Java 1.5 234-237
- SimpleUrlHandlerMapping, classe** 342
- Simplification de la création d'un template (JDBC)** 232-234
- SingleConnectionDataSource, classe** 219
- singleton, portée de bean 108, 110
- Sites web**
 - pour des informations
 - Apache Struts 403
 - DWR 404
 - framework Spring 22
 - guide de programmation AspectJ 186
 - JSF 404
 - SpringSource 22
 - pour le téléchargement
 - Apache Commons, projet 17
 - Apache Derby 214
 - Apache POI, bibliothèque 389
 - AspectJ 140
 - Eclipse Web Tools Platform (WTP) 28
 - framework Spring 28
 - iText, bibliothèque 389
 - JBoss AOP 140
 - JExcelAPI, bibliothèque 389
 - JSF Reference Implementation (JSF-R) 1.2 416
- Soumission redondante d'un formulaire** 366
- Spring 2.x**
 - AOP, vue d'ensemble 171
 - greffon transactionnel dans 268
- Spring AOP classique**
 - greffons, *Voir* Greffons
 - points d'action, *Voir* Points d'action
 - Spring classique
 - préoccupations transversales, *Voir* Préoccupations transversales
 - transactions, gérer avec 265-268
 - vue d'ensemble 139-140
- Spring IDE**
 - fichiers de configuration des beans, créer dans 31-32
 - fonctions de prise en charge des beans 34-40
 - installer 32-34

Spring IoC, conteneurs

- beans
 - configuration externalisée 122-124
 - configuration, *Voir Configuration des beans dans le conteneur Spring IoC*
 - conscients de l'existence des 116-117
 - créer avec des constructeurs 96-99
 - créer avec des méthodes d'instance de fabrique 100-102
 - créer avec des méthodes statiques de fabrique 99-100
 - créer avec un bean de fabrique Spring 102-104
 - déclarer à partir de champs statiques 104-105
 - déclarer à partir de propriétés d'objets 106-107
 - déclarés dans 32
 - initialisation et destruction, personnaliser 110-115
 - éditeurs de propriétés
 - enregister 129-132
 - personnalisés, créer 133-134
 - instancier 46-49
 - messages multilingues, obtenir 124-126
 - portée de bean, fixer 108-110
 - postprocesseur de beans, créer 118-122
 - ressources externes, charger 134-137

Spring, framework

- accéder dans des applications web génériques 404-408
- beans, créer avec un bean de fabrique Spring 102-104
- fondamentaux 22
- installer 28-29
- intégrer
 - à DWR 420-424
 - à JSF 416-420
 - à Struts 1.x 409-415
- modules 22-24
- nouvelles fonctionnalités de Spring 2.0 24-26
- projets
 - configurer 30-32
 - créer dans Eclipse 34
 - Spring Portfolio 26-27

Spring AOP classique, *Voir Spring AOP classique*

- Spring XSD 44
- spring, créateur 422
- <spring:message>, balise 350
- SpringJUnit4ClassRunner 460
- SpringSource 22
- tisseur au chargement de Spring 2.5 204

SQL (Structured Query Language)

- mettre à jour une base de données avec des requêtes 224-225
- objet d'opération SqlFunction 243
- paramètres nommés dans des templates 238-240
- SQLExceptionCodes, classe 248-249
- SQLException, classe 216, 218
- SQLExceptionTranslator, interface 249
- SqlParameterSource, interface 239

Struts 1.x (Apache), intégrer à Spring
409-415**T****Tableaux, définition** 77**Tables d'association** 78-80**Templates**

- Hibernate, utiliser 312-315
- JDBC
 - bases de données, interroger 226-231
 - bases de données, interroger avec SimpleJDBC 235-237
 - bases de données, mettre à jour 221
 - bases de données, mettre à jour avec SimpleJDBC 234-235
 - injecter 232
 - paramètres nommés, utiliser 238-240
 - template simple, utiliser avec Java 1.5 234-237
- JPA, utiliser 315-317
- ORM, objets persistants avec 312-318
- transaction 262-265

Temporisation, attribut transactionnel
286-288**@Test, type d'annotation** 429

-
- TestContext, framework**
- accéder
 - à des bases de données avec 467-469
 - à un contexte d'application avec 451, 453-454
 - annotations communes pour les tests, utiliser avec 471
 - classes de support pour la gestion des transactions 460
 - du contexte 449
 - gérer les transactions avec 462-465
 - injecter des fixtures de test 456-459
- Testing (module), définition** 24
- TestNG**
- applications, tester avec 432-434
 - contexte d'application, accéder à 454
 - fixtures de test, injecter 458
 - prise en charge des tests orientés données 433-434
 - transactions, gérer dans 465
 - vue d'ensemble 428
- Tests**
- d'intégration
 - accéder à des bases de données dans 465-469
 - contexte d'application, gérer dans 448-454
 - créer 442
 - définition 435
 - fixtures de test, injecter 455-459
 - transactions, gérer dans 459-465
 - des applications
 - annotations communes pour les tests dans Spring 469-471
 - avec JUnit 429-432
 - avec TestNG 432-434
 - bases de données, accéder à dans des tests d'intégration 465-469
 - contexte d'application, gérer dans des tests d'intégration 448-454
 - fixtures de test, injecter dans des tests d'intégration 455-459
 - tests d'intégration, créer 442
 - tests unitaires des contrôleurs MVC 443-446
- tests unitaires, créer 434-443
- transactions, gérer dans des tests d'intégration 459-465
- vue d'ensemble 427-428
- unitaires
- créer pour des classes dépendantes 438-442
 - créer pour des classes isolées 435-438
 - définition 434
 - des contrôleurs MVC 443-446
- this, objet, définition** 181
- ThrowsAdvice, interface** 161
- Tissage**
- à la compilation, AspectJ 200
 - AspectJ
 - à la compilation 200
 - au chargement 201
 - au chargement
 - gestion de transaction avec 288-291
 - par un tisseur AspectJ 201, 204
 - par un tisseur Spring 2.5 204
 - vue d'ensemble 201
 - définition 200
- Transactions**
- concurrentes, problèmes 277-280
 - définition 252
 - gérer dans les tests d'intégration 459-465
 - @Transactional, annotation 270-271, 288
 - TransactionalTestExecutionListener, classe 459
 - TransactionCallback, interface 262-264
 - TransactionInterceptor
 - classe 284
 - greffon 265
 - TransactionProxyFactoryBean, classe 266, 267, 284
 - TransactionStatus, objet 261
- TX (module), définition** 23
- <tx:advice>, élément 268
- <tx:annotation-driven>, élément 270, 291, 315
- <tx:attributes>, élément 269
- <tx:method>, élément 269, 286-287

Types

d'objets, vérifier les propriétés de 58
personnalisés, liaison des propriétés de 371-372
simples, vérifier les propriétés de 57

U**Unités de persistance, définition** 302**URL (Uniform Resource Locator)**

associer à des méthodes de gestionnaire 387-388
déterminer des vues en fonction de 352

util, schéma

balises
de collection dans 85
<util:constant> 104-105
<util:property-path> 106-107
<util:set> 86
collections, définir avec 84-86

Utilisateurs, déterminer les paramètres régionaux 348-350**V****Valeurs uniques, interroger** 230-231**Validation**

arguments (application de la calculatrice) 145
automatique, comportement 257-258
données de formulaires 372-374
assistants 381-383
fichiers de configuration des beans 39
proxies de validation, créer 153-154
validate(), méthode 381
ValidationUtils, classe 373

Vérification des dépendances

contrôler les propriétés par 56-58
injection par constructeur et 58
liaison automatique des beans 65
modes reconnus par Spring 57

Verrous de lecture, base de données 284**Vues**

associer des exceptions à 355-356
définition 327

Excel, créer 391-392

JSP, créer 338-340
paramétrées, créer des contrôleurs avec 359-361

PDF, créer 392-393
résolveurs multiples 354

résoudre
à partir de bundles de ressources 353
à partir des fichiers de configuration XML 352
en fonction des URL 352
par les noms 351-354
ViewResolver, interface 351

W**Web Flow, projet (Spring Portfolio)** 27**Web Framework Integration (module), définition** 24**Web MVC (module), définition** 24**Web Services, projet (Spring Portfolio)** 27**Web Tools Platform, Eclipse (WTP)** 28**web.xml, descripteur de déploiement** 333**WEB-INF, répertoire** 332**X****XML (Extensible Markup Language)**

configurations
à base de schéma dans Spring 2.0 24
XML, déclarer des aspects 197-200
XML, lier automatiquement des beans 61-65
correspondances, objets persistants avec Hibernate 296-300
fichiers de configuration, résoudre des vues 352
XmlBeanFactory, classe 48
XmlPortletApplicationContext, classe 48
XmlViewResolver, classe 352
XmlWebApplicationContext, classe 48

XSD (XML Schema Definition), définir des fichiers XML de configuration des beans 44

Spring par l'exemple

Développez facilement des applications Java avec Spring

« Spring représente le framework applicatif Java EE le plus simple et le plus puissant que j'aie jamais utilisé. Depuis l'inversion de contrôle et la programmation orientée aspect à la persistance, les transactions, le design pattern MVC, les tests et autres, ce livre explique comment tirer profit de Spring au travers d'exemples de code détaillés. Ces exemples vont du scan de composants dans le chemin d'accès aux classes au test unitaire des contrôleurs Spring MVC.

Spring propose des solutions simples à des problèmes difficiles. Cette simplicité a été également mon premier objectif dans la conception de cet ouvrage. J'ai souhaité trouver un équilibre entre l'étendue des sujets traités, le niveau de détail, la rapidité d'apprentissage et les connaissances requises.

Chaque chapitre explore un sujet important au travers d'un exemple réel complet. Lors de la première lecture d'un chapitre, je vous recommande de suivre l'exemple de manière séquentielle. Ensuite, vous pouvez vous en servir comme référence en examinant la liste des problèmes-solutions. Si vous cherchez à résoudre un problème précis, vous pouvez aller directement aux solutions dont vous avez besoin en consultant la table des matières ou l'index.

Bon apprentissage et utilisation de Spring ! »

Gary Mak

Programmation
Java JEE

Niveau : Tous niveaux

Configuration : Multiplate-forme

PEARSON

Pearson Education France
47 bis, rue des Vinaigriers
75010 Paris
Tél. : 01 72 74 90 00
Fax : 01 42 05 22 17
www.pearson.fr



ISBN : 978-2-7440-4107-5



9 782744 041075

TABLE DES MATIÈRES

- Inversion de contrôle et conteneurs
- Introduction au framework Spring
- Configuration des beans
- Fonctions élaborées du conteneur Spring IoC
- Proxy dynamique et Spring AOP classique
- Spring 2.x AOP et prise en charge d'AspectJ
- Prise en charge de JDBC
- Gestion des transactions
- Prise en charge de l'ORM
- Framework Spring MVC
- Intégration avec d'autres frameworks web
- Prise en charge des tests

À propos de l'auteur :

Gary Mak est architecte technique et développeur d'applications sur la plate-forme Java Entreprise depuis six ans. Il utilise le framework Spring dans ses projets depuis sa version 1.0 et intervient en tant que formateur sur Java Enterprise, Spring, Hibernate, les services web et le développement agile. Auteur de plusieurs didacticiels sur Spring et Hibernate, sa popularité ne cesse de grandir au sein de la communauté Java.