

Copyright	xiii
1. Creative Commons BY-ND-NC	xiii
Avant propos: 0.9-SNAPSHOT	xv
1. Historique de Modifications	xv
1.1. Changes in Edition 0.7.2	xv
1.2. Changes in Edition 0.7.1	xix
1.3. Modifications de l'Edition 0.7	xx
1.4. Modifications de l'Edition 0.6	xxi
1.5. Modifications de l'Edition 0.5	xxi
1.6. Modifications de l'Edition 0.4	xxii
Préface	xxv
1. Comment utiliser ce livre	xxv
2. Vos Retours	xxvi
3. Conventions de Police	xxvi
4. Conventions d'écriture Maven	xxvii
5. Remerciements	xxvii
6. Remerciements des traducteurs	xxviii
1. Introduction à Apache Maven	1
1.1. Maven... De quoi s'agit-il ?	1
1.2. Convention plutôt que configuration	1
1.3. Une Interface Commune	2
1.4. Réutilisation universelle grâce aux plugins Maven	3
1.5. Le Modèle conceptuel d'un "Projet"	4
1.6. Maven est-il une alternative à XYZ ?	5
1.7. Comparaison de Maven et de Ant	6
2. Installation de Maven	11
2.1. Vérifier votre installation de Java	11
2.2. Téléchargement de Maven	11
2.3. Installer Maven	12
2.3.1. Installer Maven sur Mac OSX	12
2.3.2. Installer Maven sur Microsoft Windows	13
2.3.3. Installer Maven sur GNU/Linux	14
2.3.4. Installer Maven sur FreeBSD ou OpenBSD	14
2.4. Tester une installation Maven	14
2.5. Détails de l'installation de Maven	14
2.5.1. Configuration et dépôt spécifiques à l'utilisateur	15
2.5.2. Mettre à jour une installation de Maven	16
2.5.3. Migrer de Maven 1.x à Maven 2.x	16
2.6. Désinstaller Maven	17
2.7. Obtenir de l'aide avec Maven	17
2.8. À propos de l'Apache Software License	18
I. Maven par l'exemple	21
3. Mon premier projet avec Maven	23

3.1. Introduction	23
3.1.1. Télécharger l'exemple de ce chapitre	23
3.2. Création du projet Simple	23
3.3. Construire le projet Simple	26
3.4. Modèle Objet du projet Simple	27
3.5. Les concepts principaux	28
3.5.1. Plugins Maven et Goals	28
3.5.2. Cycle de vie de Maven	30
3.5.3. Les coordonnées Maven	33
3.5.4. Les dépôts Maven	36
3.5.5. La gestion des dépendances de Maven	37
3.5.6. Rapports et production du site	39
3.6. En résumé	39
4. Personnalisation d'un projet Maven	41
4.1. Introduction	41
4.1.1. Télécharger l'exemple de ce chapitre	41
4.2. Présentation du projet Simple Weather	41
4.2.1. Yahoo! Météo RSS	42
4.3. Créer le Projet Simple Weather	42
4.4. Personnaliser les informations du projet	44
4.5. Ajout de nouvelles dépendances	45
4.6. Code source de Simple Weather	46
4.7. Ajouter des Ressources	52
4.8. Exécuter le programme Simple Weather	53
4.8.1. Le plugin Maven Exec	54
4.8.2. Explorer les dépendances de votre projet	55
4.9. Ecrire des tests unitaires	57
4.10. Ajouter des dépendances dans le scope <code>test</code>	59
4.11. Ajouter des ressources pour les tests unitaires	60
4.12. Exécuter les test unitaires	62
4.12.1. Ignorer les tests en échec	62
4.12.2. Court-circuiter les tests unitaires	63
4.13. Construire une application packagée et exécutable en ligne de commande	64
4.13.1. Rattacher le goal <code>Assembly</code> à la phase <code>Package</code>	66
5. Une simple application web	69
5.1. Introduction	69
5.1.1. Télécharger l'exemple de ce chapitre	69
5.2. Définition de l'application web simple-webapp	69
5.3. Création du projet web simple-web	69
5.4. Configurer le plugin Jetty	72
5.5. Ajouter une simple servlet	73
5.6. Ajouter les dépendances J2EE	75
5.7. Conclusion	77

6. Un projet multimodule	79
6.1. Introduction	79
6.1.1. Télécharger l'exemple de ce chapitre	79
6.2. Le projet parent	79
6.3. Le module simple-weather	81
6.4. Le module simple-web	83
6.5. Construction du projet multimodule	85
6.6. Exécution de l'application web	87
7. Un projet multimodule d'entreprise	89
7.1. Introduction	89
7.1.1. Télécharger les sources de ce chapitre	89
7.1.2. Projet multimodule d'entreprise	89
7.1.3. Technologies utilisées dans cet exemple	92
7.2. Le projet simple-parent	92
7.3. Le module simple-model	94
7.4. Le module simple-weather	98
7.5. Le module simple-persist	101
7.6. Le module simple-webapp	108
7.7. Exécution de l'application web	118
7.8. Le module de ligne de commande	119
7.9. Exécution de l'application en ligne de commande	125
7.10. Conclusion	127
7.10.1. Programmation avec des projets API	128
8. Optimiser et remanier les POMs	131
8.1. Introduction	131
8.2. Nettoyer le POM	132
8.3. Optimiser les dépendances	132
8.4. Optimiser les plugins	136
8.5. Optimisation avec le plugin Maven Dependency	138
8.6. Les POMs finaux	141
8.7. Conclusion	149
II. Maven - La Reference	151
9. Le Modèle Objet de Projet	153
9.1. Introduction	153
9.2. Le POM	153
9.2.1. Le Super POM	155
9.2.2. Le POM le plus simple possible	158
9.2.3. Le POM effectif	159
9.2.4. Véritables POMs	159
9.3. Syntaxe de POM	160
9.3.1. Les versions d'un projet	160
9.3.2. Référence à une propriété	162
9.4. Dépendances d'un projet	163

9.4.1. Scope de dépendance	164
9.4.2. Dépendances optionnelles	165
9.4.3. Intervalle de versions pour une dépendance	167
9.4.4. Dépendances transitives	168
9.4.5. Résolution des conflits	169
9.4.6. Gestion des dépendances	171
9.5. Relations entre projets	172
9.5.1. Au sujet des coordonnées	173
9.5.2. Projets multimodules	174
9.5.3. Héritage de projet	175
9.6. Les bonnes pratiques du POM	178
9.6.1. Regrouper les dépendances	178
9.6.2. Multimodule ou héritage	180
10. Cycle de vie du build	185
10.1. Introduction	185
10.1.1. Cycle de vie Clean (clean)	185
10.1.2. Cycle de vie par défaut (default)	188
10.1.3. Cycle de vie Site (site)	190
10.2. Cycles de vie spécifiques par type de package	190
10.2.1. JAR	191
10.2.2. POM	191
10.2.3. Plugin Maven	191
10.2.4. EJB	192
10.2.5. WAR	192
10.2.6. EAR	193
10.2.7. Autres types de packaging	193
10.3. Goals communs aux cycles de vie	194
10.3.1. Traiter les ressources	195
10.3.2. Compilation	198
10.3.3. Traiter les ressources des tests	199
10.3.4. Compilation des tests	199
10.3.5. Tester	200
10.3.6. Installer l'artefact	201
10.3.7. Déploiement	201
11. Profils de Build	203
11.1. À quoi servent-ils ?	203
11.1.1. Qu'est ce que la Portabilité du Build ?	203
11.1.2. Choisir le bon niveau de portabilité	204
11.2. Portabilité grâce aux profils Maven	205
11.2.1. Surcharger un POM	207
11.3. Activation de profil	208
11.3.1. Configuration de l'activation	210
11.3.2. Activation par l'absence d'une propriété	211

11.4. Lister les profils actifs	211
11.5. Trucs et Astuces	212
11.5.1. Environnements communs	212
11.5.2. Protéger les mots de passe	214
11.5.3. Classifieurs de plateforme	215
11.6. En résumé	217
12. Exécuter Maven	219
12.1. Options de ligne de commande Maven	219
12.1.1. Définition de propriété	219
12.1.2. Obtenir de l'aide	219
12.1.3. Utilisation de profils de build	221
12.1.4. Afficher les informations relatives à la version	221
12.1.5. Travailler en mode déconnecté	221
12.1.6. Utiliser le POM et le fichier settings de votre choix	222
12.1.7. Chiffrer les mots de passe	222
12.1.8. Gestion des erreurs	222
12.1.9. Contrôle de la verbosité de Maven	223
12.1.10. Exécution de Maven en mode batch	223
12.1.11. Téléchargement et vérification des dépendances	223
12.1.12. Contrôle de la mise à jour des plugins	224
12.1.13. Builds non-récurifs	224
12.2. Utilisation des options avancées du Reactor	225
12.2.1. Reprise de build	225
12.2.2. Spécifier un sous ensemble de projets	225
12.2.3. Construire des sous-ensembles	226
12.2.4. Modifier simple-weather et vérifier que nous n'avons rien cassé grâce à --also-make-dependents	226
12.2.5. Reprise d'un build "make"	226
12.3. Usage du plugin Maven Help	227
12.3.1. Décrire un plugin Maven	227
13. Configuration Maven	231
13.1. Configuration des plugins Maven	231
13.1.1. Paramètres du plugin Configuration	231
13.1.2. Ajouter des dépendances à un plugin	234
13.1.3. Configurer les paramètres globaux d'un plugin	235
13.1.4. Modifier les paramètres spécifiques à une exécution	235
13.1.5. Configuration des paramètres par défaut pour une exécution en ligne de commande	236
13.1.6. Configuration des paramètres pour les goals rattachés au cycle de vie par défaut	236
14. Maven Assemblies	239
14.1. Introduction	239
14.2. Les bases du plugin Assembly	239

14.2.1. Les descripteurs Assembly prédéfinis	240
14.2.2. Construire un Assembly	241
14.2.3. Utilisation des assemblies comme dépendances	244
14.2.4. Construction d'assemblies à partir d'assemblies dépendances	244
14.3. Vue d'ensemble du descripteur d'assembly	248
14.4. Le descripteur d'assembly	249
14.4.1. Référence de propriété dans un descripteur d'assembly	249
14.4.2. Informations obligatoires pour un assembly	249
14.5. Choisir les contenus d'un assembly	251
14.5.1. Section <code>files</code>	251
14.5.2. Section <code>fileSets</code>	252
14.5.3. Patterns d'exclusion par défaut pour la balise <code>fileSets</code>	254
14.5.4. Section <code>dependencySets</code>	255
14.5.5. La balise <code>moduleSets</code>	265
14.5.6. Balise <code>repositories</code>	271
14.5.7. Gestion du répertoire racine de l'assembly	272
14.5.8. <code>componentDescriptors</code> et <code>containerDescriptorHandlers</code>	273
14.6. Best Practices	273
14.6.1. Descripteurs d'assembly standards et réutilisables	274
14.6.2. Assembly de distribution (agrégation)	277
14.7. En résumé	281
15. Propriétés et filtrage des ressources	283
15.1. Introduction	283
15.2. Propriétés Maven	283
15.2.1. Propriétés d'un projet Maven	284
15.2.2. Propriétés des Settings Maven	286
15.2.3. Propriétés des variables d'environnement	286
15.2.4. Propriétés système Java	286
15.2.5. Propriétés définies par l'utilisateur	287
15.3. Filtrage des ressources	289
16. Génération du Site	293
16.1. Introduction	293
16.2. Contruire le site d'un projet avec Maven	293
16.3. Personnaliser le descripteur de site	295
16.3.1. Personnaliser les images des en-têtes du site	296
16.3.2. Personnaliser le menu navigation	297
16.4. Structure de répertoire d'un site	298
16.5. Écrire la documentation d'un projet	299
16.5.1. Exemple de fichier APT	299
16.5.2. Exemple de fichier FML	300
16.6. Déployez le site de votre projet	301
16.6.1. Configurer l'authentification de votre serveur	302
16.6.2. Configurer les permissions des fichiers et dossiers	302

16.7. Personnaliser l'apparence de votre site	303
16.7.1. Personnaliser la CSS du site	303
16.7.2. Créer un modèle de site personnalisé	304
16.7.3. Réutilisation des skins	308
16.7.4. Création d'un thème CSS personnalisé	309
16.8. Trucs et Astuces	311
16.8.1. Intecter du XHTML dans le HEAD	311
16.8.2. Ajouter des liens sous le logo de votre site	311
16.8.3. Ajouter un chemin de navigation à votre site	312
16.8.4. Ajouter la version de votre projet	312
16.8.5. Modifier le format et l'emplacement de la date de publication	313
16.8.6. Utiliser des macros Doxia	314
17. Création de Plugins	317
17.1. Introduction	317
17.2. Programmation Maven	317
17.2.1. Qu'est ce que l'inversion de contrôle ?	317
17.2.2. Introduction à Plexus	318
17.2.3. Pourquoi Plexus ?	319
17.2.4. Qu'est ce qu'un Plugin ?	320
17.3. Descripteur de Plugin	320
17.3.1. Éléments haut-niveau du descripteur de plugin	322
17.3.2. Configuration du Mojo	323
17.3.3. Dépendances d'un Plugin	325
17.4. Écrire un plugin personnalisé	326
17.4.1. Création d'un projet Plugin	326
17.4.2. Un simple Mojo Java	327
17.4.3. Configuration d'un préfixe de Plugin	328
17.4.4. Les traces d'un plugin	331
17.4.5. Annotations de Mojo	332
17.4.6. Lorsque un Mojo échoue	333
17.5. Paramètres d'un Mojo	334
17.5.1. Affecter des valeurs aux paramètres de Mojo	334
17.5.2. Paramètres de Mojo multi-valeurs	336
17.5.3. Dépendre de composants Plexus	338
17.5.4. Paramètres des annotations d'un Mojo	339
17.6. Plugins et le cycle de vie Maven	340
17.6.1. Exécution dans un cycle de vie parallèle	340
17.6.2. Création d'un cycle de vie personnalisé	341
17.6.3. Surcharge du cycle de vie par défaut	342
18. Utilisation des archetypes Maven	345
18.1. Introduction aux archetypes Maven	345
18.2. Utilisation des archétypes	345
18.2.1. Utilisation d'un archétype à partir de la ligne de commande	345

18.2.2. Utilisation du goal Generate en mode interactif	346
18.2.3. Utilisation d'un archétype à partir du plugin Eclipse m2eclipse	348
18.3. Archétypes disponibles	349
18.3.1. Archétypes Maven communs	349
18.3.2. Archétypes tiers notables	350
18.4. Publication d'archétypes	353
19. Développement avec Flexmojos	355
19.1. Introduction	355
19.2. Configuration de l'environnement de build pour Flexmojos	355
19.2.1. Faire référence à un dépôt contenant le Framework Flex	355
19.2.2. Configuration de l'environnement pour les tests Flex Unit	360
19.2.3. Ajouter FlexMojos aux groupes de plugins de votre configuration Maven	362
19.3. Création d'un projet FlexMojos à partir d'un archétype	362
19.3.1. Création d'une bibliothèque Flex	363
19.3.2. Création d'une application Flex	367
19.3.3. Creation d'un projet multimodule : Une application web avec une dépendance Flex	370
19.4. Le cycle de vie de FlexMojos	375
19.4.1. Le cycle de vie SWC	376
19.4.2. Le cycle de vie SWF	377
19.5. Les goals du plugin FlexMojos	378
19.5.1. Génération de la documentation ActionScript	379
19.5.2. Compilation des sources Flex	379
19.5.3. Génération des fichiers de projet Flex Builder	381
19.6. Rapports du plugin FlexMojos	381
19.6.1. Produire le rapport de documentation ActionScript	381
19.7. Développement et personnalisation de Flexmojos	382
19.7.1. Obtenir le code source Flexmojos	383
A. Annexe : détails des settings	385
A.1. Aperçu rapide	385
A.2. Détails des settings	385
A.2.1. Valeurs simples	385
A.2.2. Balise servers	386
A.2.3. Balise mirrors	387
A.2.4. Balise proxies	388
A.2.5. Balise profiles	389
A.2.6. Balise activation	389
A.2.7. Balise properties	390
A.2.8. Balise repositories	391
A.2.9. Balise pluginRepositories	393
A.2.10. Balise activeProfiles	394
A.2.11. Chiffrement des mots de passe dans les Settings Maven	394

B. Annexe : alternatives aux spécifications Sun 399

Copyright

Copyright 2009 Sonatype, Inc.

Online version published by Sonatype, Inc., 654 High Street, Suite 220, Palo Alto, CA, 94301.

Print version published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Nutshell Handbook, the Nutshell Handbook logo, and the O'Reilly logo are registered trademarks of O'Reilly Media, Inc. The Developer's Notebook series designations, the look of a laboratory notebook, and related trade dress are trademarks of O'Reilly Media, Inc.

Java(TM) and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc., in the United States and other countries. Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and Sonatype, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

1. Creative Commons BY-ND-NC

This work is licensed under a Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States license. For more information about this license, see <http://creativecommons.org/licenses/by-nc-nd/3.0/us/>. You are free to share, copy, distribute, display, and perform the work under the following conditions:

- You must attribute the work to Sonatype, Inc. with a link to <http://www.sonatype.com>.
- You may not use this work for commercial purposes.
- You may not alter, transform, or build upon this work.

If you redistribute this work on a web page, you must include the following link with the URL in the about attribute listed on a single line (remove the backslashes and join all URL parameters):

```
<div xmlns:cc="http://creativecommons.org/ns#"
  about="http://creativecommons.org/licenses/results-one?q_1=2&q_1=1\
    &field_commercial=n&field_derivatives=n&field_jurisdiction=us\
    &field_format=StillImage&field_worktitle=Maven%3A+\Guide\
    &field_attribute_to_name=Sonatype%2C+Inc.\
    &field_attribute_to_url=http%3A%2F%2Fwww.sonatype.com\
    &field_sourceurl=http%3A%2F%2Fwww.sonatype.com%2Fbook\
    &lang=en_US&language=en_US&n_questions=3">
```

```
<a rel="cc:attributionURL" property="cc:attributionName"
  href="http://www.sonatype.com">Sonatype, Inc.</a> /
<a rel="license"
  href="http://creativecommons.org/licenses/by-nc-nd/3.0/us/">
  CC BY-NC-ND 3.0</a>
</div>
```

When downloaded or distributed in a jurisdiction other than the United States of America, this work shall be covered by the appropriate ported version of Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 license for the specific jurisdiction. If the Creative Commons Attribution-Noncommercial-No Derivative Works version 3.0 license is not available for a specific jurisdiction, this work shall be covered under the Creative Commons Attribution-Noncommercial-No Derivate Works version 2.5 license for the jurisdiction in which the work was downloaded or distributed. A comprehensive list of jurisdictions for which a Creative Commons license is available can be found on the Creative Commons International web site at <http://creativecommons.org/international>.

If no ported version of the Creative Commons license exists for a particular jurisdiction, this work shall be covered by the generic, unported Creative Commons Attribution-Noncommercial-No Derivative Works version 3.0 license available from <http://creativecommons.org/licenses/by-nc-nd/3.0/>.

Avant propos: 0.9-SNAPSHOT

Jusqu'à présent nous avons eu beaucoup de retours, ne vous arrêtez pas. Ceux-ci sont très appréciés, aussi envoyez les à book@sonatype.com¹. Pour être averti des mises à jour, lisez le blog du livre: <http://blogs.sonatype.com/book>². Tout le monde à Sonatype a mis la main à la pâte pour cette version du livre, c'est pourquoi l'auteur est officiellement "Sonatype".

Tim O'Brien (tobrien@sonatype.com)

Evanston, IL

August 7, 2009

1. Historique de Modifications

De nombreux lecteurs nous ont demandé de tracer les modifications que nous apportons au contenu du livre, la section suivante les liste par ordre retro-chronologique depuis la version 0.9-SNAPSHOT.

1.1. Changes in Edition 0.7.2

The following changes were made:

- Added documentation for Default Mojo Execution settings in a new chapter focused on Maven Configuration Chapitre 13, Configuration Maven. (MVNDEF-140³)
- Added some clarification to Section 5.4, « Configurer le plugin Jetty » instructing users to run `jetty:run` from the `simple-webapp/` directory. (MVNDEF-115⁴)
- Added a warning note to Section 5.4, « Configurer le plugin Jetty » warning Windows users of potential problems starting Jetty with a local Maven repository stored under "C:\Documents and Settings". (MVNDEF-114⁵)
- Update Section 9.2.1, « Le Super POM » to include the Super POM from Maven 2.2.1. (MVNDEF-176⁶)
- Removed summary section from the Introduction, it was useless and served no purpose.
- Addressing feedback from a proofread of the 0.7.1 PDF (MVNDEF-271⁷)
 - Simplified sentence structure on page 88 in Section 7.1.2, « Projet multimodule d'entreprise ». (MVNDEF-278⁸)

¹ <mailto:book@sonatype.com>

² <http://blogs.sonatype.com/book>

- Fixed a spelling related typo on Page 7 in Section 1.7, « Comparaison de Maven et de Ant ». (MVNDEF-289⁹)
- Split a sentence on Page 5 in Section 1.6, « Maven est-il une alternative à XYZ ? ». (MVNDEF-302¹⁰)
- Fixed a sentence structure issues on Page 48 in Section 4.6, « Code source de Simple Weather ». (MVNDEF-304¹¹)
- Fixed a spelling typo on Page 39 in Section 3.5.5, « La gestion des dépendances de Maven ». (MVNDEF-310¹²)
- Added missing punctuation on Page 360 in Section A.2.1, « Valeurs simples ». (MVNDEF-313¹³)
- Fixed a grammar error on Page 356 in Section 19.7.1, « Obtenir le code source Flexmojos ». (MVNDEF-314¹⁴)
- Fixed a wording issues on Page 353 in Section 19.5.2, « Compilation des sources Flex ». (MVNDEF-315¹⁵)
- Fixed a spelling problem on Page 324 in Section 18.3.2.2, « Plugins Confluence et JIRA ». (MVNDEF-317¹⁶)
- Fixed a spelling problem on Page 320 in Section 18.2.2, « Utilisation du goal Generate en mode interactif ». (MVNDEF-318¹⁷)
- Fixed two sentence structure issue on Page 312 in Section 17.6.1, « Exécution dans un cycle de vie parallèle ». (MVNDEF-320¹⁸)
- Fixed a spelling issue on Page 311 in Section 17.5.4, « Paramètres des annotations d'un Mojo ». (MVNDEF-321¹⁹)
- Fixed several sentence structure issues on Page 30 in Section 3.5.1, « Plugins Maven et Goals ». (MVNDEF-323²⁰)
- Fixed a tense issue on Page 298 in Section 17.4.1, « Création d'un projet Plugin ». (MVNDEF-324²¹)
- Capitalized "Java" on Page 296 in Section 17.3.2, « Configuration du Mojo ». Changed the note for `executionStrategy`. Added a missing question mark. (MVNDEF-325²²)
- Fixed some sentence structure issues on Page 294 in Section 17.3.1, « Éléments haut-niveau du descripteur de plugin ». (MVNDEF-326²³)

- Fixed some sentence structure issues on Page 290 in Section 17.2.1, « Qu'est ce que l'inversion de contrôle ? ». (MVNDEF-327²⁴)
- Fixed some wording issues on Page 282 in ???. (MVNDEF-328²⁵)
- Added a missing word on Page 281 in Section 16.7.4, « Création d'un thème CSS personnalisé ». (MVNDEF-329²⁶)
- Fixed a wording issue on Page 273 in Section 16.6, « Déployez le site de votre projet ». (MVNDEF-330²⁷)
- Fixed a spelling issue on Page 244 in Section 14.5.6, « Balise repositories ». (MVNDEF-331²⁸)
- Fixed wording issues on Page 229 in Section 14.5.4, « Section dependencySets ». (MVNDEF-332²⁹)
- Modified sentence structure in Section 11.2, « Portabilité grâce aux profils Maven ». (MVNDEF-337³⁰)
- Removed unnecessary comma from Page 168 in Section 9.5.2, « Projets multimodules ». (MVNDEF-339³¹)
- Fixed sentence structure on Page 166 in Section 9.5.1, « Au sujet des coordonnées ». (MVNDEF-340³²)
- Fixed various spelling and grammar typos on Page 155 in Section 9.3, « Syntaxe de POM ». (MVNDEF-341³³)
- Fixed punctuation issues on Page 149 in Section 9.2.1, « Le Super POM ». (MVNDEF-342³⁴)
- Fixed some spelling and wording issues on Page 119 in Section 7.8, « Le module de ligne de commande ». (MVNDEF-344³⁵)
- Fixed some punctuation issues on Page 110 in Section 7.6, « Le module simple-webapp ». (MVNDEF-345³⁶)
- Responding to Grant Birchmeier's proofread of the Preface material from GetSatisfaction. (MVNDEF-346³⁷, MVNDEF-347³⁸)
 - Split sentence in second paragraph of Section 1, « Comment utiliser ce livre ». (MVNDEF-349³⁹)
 - Fixed mismatch between contact address and footnote in first paragraph of Section 2, « Vos Retours ». (MVNDEF-350⁴⁰)
 - Making sure that the Italic is really italic in the font conventions section. (MVNDEF-351⁴¹)

- Adopted the proposed language for the "plugin" bullet point. (MVNDEF-352⁴²)
- Added a missing article to the first sentence of the first paragraph of Section 1.3, « Une Interface Commune ». (MVNDEF-353⁴³)
- Rearranged a long, ungrammatical sentence at the start of the last paragraph in Section 1.4, « Réutilisation universelle grâce aux plugins Maven ». (MVNDEF-354⁴⁴)
- Added a missing preposition to first bullet in Section 1.5, « Le Modèle conceptuel d'un "Projet" ». (MVNDEF-355⁴⁵)
- Reworded the universal reuse bullet item in Section 1.5, « Le Modèle conceptuel d'un "Projet" ». (MVNDEF-356⁴⁶)
- Removed final sentence in Section 1.5, « Le Modèle conceptuel d'un "Projet" ». (MVNDEF-357⁴⁷)
- Removed the last sentence of Section 1.6, « Maven est-il une alternative à XYZ ? ». (MVNDEF-358⁴⁸)
- Removed a superfluous "the" from the first paragraph of Section 1.7, « Comparaison de Maven et de Ant ». (MVNDEF-359⁴⁹)
- Rewrote the first paragraph of Section 1.7, « Comparaison de Maven et de Ant ». (MVNDEF-360⁵⁰)
- Rewrote second and third sentence fragments in the sixth paragraph of Section 1.7, « Comparaison de Maven et de Ant ». (MVNDEF-362⁵¹)
- Made sure that the comparison bullet points used consistent tense. Fixed a number of sentence fragment issues in the comparison bullet points. (MVNDEF-363⁵², MVNEF-364⁵³, MVNDEF-365⁵⁴, and MVNDEF-366⁵⁵)
- Addressed a few grammar errors in the third to last paragraph of Section 1.7, « Comparaison de Maven et de Ant ». (MVNDEF-367⁵⁶)
- Combined the first two sentences of Section 2.5.2, « Mettre à jour une installation de Maven ». (MVNDEF-369⁵⁷)
- Italicized a book title in Section 2.5.3, « Migrer de Maven 1.x à Maven 2.x ». (MVNDEF-371⁵⁸)
- Separating URLs with a colon twice in Section 2.8, « À propos de l'Apache Software License ». (MVNDEF-372⁵⁹)

- Fixed an incorrect reference to Part II in the third paragraph of Partie I, « Maven par l'exemple ». (MVNDEF-373⁶⁰)

1.2. Changes in Edition 0.7.1

The following changes were made:

- Various changes in Chapitre 19, Développement avec Flexmojos to support the FlexMojos 3.3.0 release:
 - Modified Section 19.2, « Configuration de l'environnement de build pour Flexmojos » to include instructions for referencing Sonatype's Flexmojos repository in a project's POM. (MVNDEF-260⁶¹ and MVNDEF-263⁶²)
 - Update Figure 19.2, « Configuration du dépôt Sonatype Flexmojos Proxy » to reflect the switch to the Sonatype Flexmojos Repository. (MVNDEF-264⁶³)
 - Update Figure 19.3, « Ajout du proxy de Sonatype Flexmojos au groupe `Public Repositories` » to reflect the switch to the Sonatype Flexmojos Repository. (MVNDEF-265⁶⁴)
- Updated Maven Version to 2.2.1. (MVNDEF-268⁶⁵)
- Moving most introduction examples to the `archetype:generate` goal (instead of `archetype:create`) (MVNDEF-41⁶⁶)
 - Chapitre 3, Mon premier projet avec Maven now introduces the `archetype:generate` goal first. `archetype:create` is still useful, but `archetype:generate` is much friendlier and a more appropriate way to introduce new Maven users to the power of Maven Archetypes.
 - Chapitre 4, Personnalisation d'un projet Maven now uses the `archetype:generate` goal instead of `archetype:create` goal.
 - Chapitre 5, Une simple application web new uses the `archetype:generate` goal instead of the `archetype:create` goal.
- Added `${project.baseUri}` to Section 15.2.1, « Propriétés d'un projet Maven ». (MVNDEF-141⁶⁷)
- Fixed XML element ordering error in Exemple 4.21, « Configurer l'exécution du goal `attached` durant la phase `Package` du cycle de vie ». (MVNDEF-32⁶⁸)
- A few readers were confused about an example in Chapter 7. A clarification was added to Section 7.7, « Exécution de l'application web » to instruct the reader to run `mvn clean install` from the top-level directory before attempting to build the database with `hbm2ddl`. (MVNDEF-43⁶⁹)

- Verified that examples can be compiled without the need for Sonatype repositories in response to a reader question. (MVNDEF-72⁷⁰)
- Minor formatting typo fixed in Section 14.2.2, « Construire un Assembly ». (MVNDEF-42⁷¹)
- Resized all Vector Images to fit within Print Margins.
- Resized PDF to Royal Quatro sizing for print-on-demand.
- Automated generation of print figures.
- PDF now bundles fonts to satisfy pre-print requirements.

1.3. Modifications de l'Édition 0.7

Les modifications suivantes ont été apportées:

- Le build Maven de Maven: Nous avons configuré le Definitive Guide pour qu'il utilise le pugin Maven Scribd⁷². Pour voir la configuration du build de ce livre, allez voir le projet maven-guide-en⁷³ sur GitHub. (MVNDEF-128⁷⁴ and MVNDEF-127⁷⁵)
- Correction de références incorrectes vers le livre Eclipsedans les sections Section 18.3.2, « Archétypes tiers notables » et Section 18.2.3, « Utilisation d'un archétype à partir du plugin Eclipse m2eclipse ». Cette section référençait une liste d'archétypes disponible lorsque l'on utilise le wizard de création de projet de m2eclipse. (MVNDEF-79⁷⁶ and MVNDEF-78⁷⁷)
- Correction de références incorrectes dans le chapitre traitant de Spring Web Section 7.6, « Le module simple-webapp » et. (MVNDEF-77⁷⁸, MVNDEF-76⁷⁹, et MVNDEF-75⁸⁰)
- Correction d'une faute de frappe dans Section 8.3, « Optimiser les dépendances ». (MVNDEF-25⁸¹)
- Correction d'un problème de dépassement de ligne dans Section 7.9, « Exécution de l'application en ligne de commande », Exemple 10.1, « Exécuter un goal lors du pre-clean », Section 18.2.2, « Utilisation du goal Generate en mode interactif », Section 18.3.2.1, « AppFuse », Section 18.4, « Publication d'archétypes », Section 19.3.1, « Création d'une bibliothèque Flex », Section 19.3.2, « Création d'une application Flex » et Section 19.3.3, « Creation d'un projet multimodule : Une application web avec une dépendance Flex »
- Correction de deux références non échappées à `${basedir}` dans ???, Section 19.5.2, « Compilation des sources Flex », et ???. (MVNDEF-191⁸², MVNDEF-192⁸³, et MVNDEF-193⁸⁴)
- Suppression de quotes en trop avec les références croisées. (MVNDEF-196⁸⁵, cette modification correspond à la première requête fork + pull sur GitHub par Larry Shatzer)
- Correction de problèmes d'espacement dans l'Appendix B, et vérification que les coordonnées GA de l'artefact Geronimo ne dépassent pas sur la colonne version. (MVNDEF-2⁸⁶)

- Correction de fautes de frappe mineures dans Section 11.5.3, « Classifieurs de plateforme ». (MVNDEF-124 ⁸⁷)
- Correction de la faute de frappe repository.sonatype.com instead au lieu de repository.sonatype.org. Erreurs corrigées dans Chapitre 19, Développement avec Flexmojos. (MVNDEF-129 ⁸⁸)

1.4. Modifications de l'Édition 0.6

Les modifications suivantes ont été apportées:

- MVNDEF-23 ⁸⁹ - Correction d'une faute de frappe dans Section 17.6.3, « Surcharge du cycle de vie par défaut » - "Maven won't know anything it" -> "Maven won't know anything about it"

1.5. Modifications de l'Édition 0.5

Les modifications suivantes impactent le livre dans son ensemble:

- MVNDEF-101 ⁹⁰ - Réduction de la Largeur des Marges PDF
- MVNDEF-100 ⁹¹ - Création d'une de Colonnes de Tble de Largeurs Spécifiques pour le chapitre Flex
- MVNDEF-99 ⁹² - Réduction de la Taille de la Police dans le Livre PDF
- MVNDEF-98 ⁹³ - Created a Table with Custom Column Widths and Multiple Named Spans

Le gros des changements dans la version 0.5 concerne le chapitre FlexMojos, un chapitre qui est toujours à l'état de brouillon puisque le projet FlexMojos est toujours en cours de développement.

- MVNDEF-85 ⁹⁴ - Ajout de documentation pour les cycles de vie spécifiques SWC et SWF dans le chapitre FlexMojos - Section 19.4, « Le cycle de vie de FlexMojos »
- MVNDEF-83 ⁹⁵ - Ajout des instructions pour configurer le Flash Player afin qu'il supporte les tests unitaires de FlexMojos - Section 19.2.2, « Configuration de l'environnement pour les tests Flex Unit »
- MVNDEF-82 ⁹⁶ - Mise à jour des Archétypes FlexMojos Archetypes selon la version 3.1.0. Les archétypes FlexMojos 3.1.0 ne dépendent plus du POM parent pour personnaliser le build pour Flex - Section 19.3, « Création d'un projet FlexMojos à partir d'un archétype »
- MVNDEF-84 ⁹⁷ - Ajout d'une section qui documente tous les goals du plugin FlexMojos - Section 19.5, « Les goals du plugin FlexMojos »
- MVNDEF-103 ⁹⁸ - Ajout d'une section au Chapitre Flexsur l'ajout du Sonatype Plugin Group à votre configuration Maven - Section 19.2.3, « Ajouter FlexMojos aux groupes de plugins de votre configuration Maven »

- MVNDEF-102⁹⁹ - Mise à jour du livre pour référencer la version 3.2.0 de FlexMojos
- MVNDEF-94¹⁰⁰ - Documentation des goals test-compile et test-run. - ???
- MVNDEF-89¹⁰¹ - Documentation du goal flexbuilder - Section 19.5.3, « Génération des fichiers de projet Flex Builder »
- MVNDEF-87¹⁰² - Documentation des goals compile-swc et compile-swf - Section 19.5.2, « Compilation des sources Flex »
- MVNDEF-86¹⁰³ - Documentation du goal et du rapport Actionscript Documentation - Section 19.5.1, « Génération de la documentation ActionScript » et Section 19.6, « Rapports du plugin FlexMojos »

1.6. Modifications de l'Édition 0.4

Les modifications suivantes ont été apportées pour l'Édition 0.4:

- MVNDEF-51¹⁰⁴ - Ajout d'une section dans le Settings Appendix sur le cryptage des mots de passe du fichier Maven Settings - Section A.2.11, « Chiffrement des mots de passe dans les Settings Maven »
- MVNDEF-54¹⁰⁵ - Ajout des informations sur le timestamp de compilation aux données sur la notion de version dans Maven de la section Section 2.4, « Tester une installation Maven »
- MVNDEF-52¹⁰⁶ - Ajout des informations sur Java Home dans les traces de la commande version de Maven de la section Section 2.4, « Tester une installation Maven »

Correction de fautes de frappe:

- MVNDEF-59¹⁰⁷ - Correction d'une faute de frappe dans le chapitre sur les relations entre POM. Section 9.6.1, « Regrouper les dépendances » qui contenait une note avec un type "dependenctManagement" corrigé en "dependencyManagement"
- MVNDEF-46¹⁰⁸ - Correction d'une faute de frappe dans Section 1.7, « Comparaison de Maven et de Ant », "execute the a" est corrigé en "execute a"
- MVNDEF-45¹⁰⁹ - Correction d'une faute de frappe dans Section 16.7, « Personnaliser l'apparence de votre site », "is created many" est corrigé en "is creating many"
- MVNDEF-44¹¹⁰ - Correction d'une faute de frappe dans Section 3.5.2, « Cycle de vie de Maven », "execute all proceeding phases" est corrigé en "execute all preceding phases"
- MVNDEF-31¹¹¹ - Vérification que la correction de "weather-servley" en "weather-servlet" est bien faite dans Figure 7.3, « Contrôleurs Spring MVC référençant les modules simple-weather et simple-persist. »

- MVNDEF-39¹¹² - Le prefixe du goal du plugin Compiler est "compiler" et non "compile" comme il était précédemment écrit dans Section 17.3.1, « Éléments haut-niveau du descripteur de plugin »

Préface

Maven est un outil de "build", de gestion de projet, un conteneur abstrait où s'exécutent les différentes étapes de construction du projet. C'est un outil qui s'est révélé indispensable pour les projets qui deviennent complexes et qui ont besoin de construire et de gérer de manière cohérente de nombreux modules et bibliothèques interdépendants, eux-même utilisant des dizaines voir des centaines de composants tiers. C'est un outil qui a fortement allégé le fardeau quotidien de la gestion des dépendances vers les bibliothèques tierces pour des millions d'ingénieurs, et a permis à de nombreuses organisations de se sortir de l'ornière de la gestion du build de projet pour atteindre un monde où l'effort requis pour construire et maintenir un logiciel n'est plus le facteur limitant dans sa conception.

Ce travail est la première tentative d'un livre complet sur Maven. Il se base sur les expériences et le travail combinés des auteurs des livres précédents sur Maven, aussi vous ne devez pas le voir comme une étape finale mais comme la première édition d'une longue liste de mises à jour. Alors que Maven n'a que quelques années d'existence, les auteurs de ce livre pensent qu'il a juste commencé à remplir les audacieuses promesses faites. Les auteurs, et l'entreprise derrière ce livre, Sonatype¹, pensent que la publication de ce livre marque le début d'une nouvelle phase d'innovation et de développement de Maven et de son écosystème environnant.

1. Comment utiliser ce livre

Prenez le, lisez le contenu de ses pages. Une fois arrivé à la fin d'une page, vous voudrez soit cliquer sur le lien, si vous regardez la version HTML, ou alors, si vous avez la version imprimée, vous soulevez un coin de la page et vous la tournerez. Si vous êtes assis à côté d'un ordinateur, vous pouvez taper certains des exemples et suivre au fur et à mesure. Par pitié, ne lancez pas ce gros volume à la tête de quelqu'un sous le coup de la colère.

Ce livre se compose de trois parties: une Introduction, une Partie I, « Maven par l'exemple », et une Partie II, « Maven - La Reference ». L'introduction se compose de deux chapitres: Chapitre 1, Introduction à Apache Maven et Chapitre 2, Installation de Maven. La Partie I, « Maven par l'exemple » introduit Maven par sa mise en oeuvre sur des exemples concrets tout en expliquant le comment et le pourquoi de leur structure. Si vous êtes novice en ce qui concerne Maven, commencez par la Partie I, « Maven par l'exemple ». La Partie II, « Maven - La Reference » est moins une introduction qu'une référence, chaque chapitre de la Partie II, « Maven - La Reference » traite d'un sujet en particulier et en donne le maximum de détails possible. Par exemple, le Chapitre 17, Création de Plugins dans la Partie II, « Maven - La Reference » traite de la manière d'écrire des plugins au travers des quelques exemples associés à un ensemble de tableaux et de listes.

Même si les deux parties la Partie I, « Maven par l'exemple » et la Partie II, « Maven - La Reference » fournissent des explications, chacune a sa propre stratégie. Là où la Partie I, « Maven par l'exemple » se concentre sur le contexte d'un projet Maven, la Partie II, « Maven - La Reference » se concentre sur un

¹ <http://www.sonatype.com>

sujet particulier. Vous pouvez sauter certaines parties du livre, la Partie I, « Maven par l'exemple » n'est en aucune sorte un pré-requis pour la Partie II, « Maven - La Reference », mais vous pourrez mieux apprécier la Partie II, « Maven - La Reference » si vous avez lu la Partie I, « Maven par l'exemple ». Maven s'apprend mieux par l'exemple, mais une fois ceux-ci faits, vous aurez besoin d'éléments de référence pour commencer à adapter Maven à votre environnement.

2. Vos Retours

Nous n'avons pas écrit ce livre afin de produire un document Word que nous enverrions à notre maison d'édition avant d'aller en fête le lancement en nous autocongratulant pour un travail terminé. Ce livre n'est pas "terminé" ; en fait, ce livre ne le sera jamais complètement. Le sujet qu'il couvre est en perpétuelle évolution et expansion, aussi nous considérons ce travail comme une discussion vivante avec la communauté. Publier ce livre signifie juste que le véritable travail vient de commencer, et vous, notre lecteur, vous avez un rôle essentiel pour nous aider à maintenir et améliorer ce livre. Si vous voyez une erreur quelconque dans ce livre, une faute d'orthographe, du code de mauvaise qualité, un mensonge éhonté, envoyez-nous un e-mail à: book@sonatype.com².

C'est grâce à vous et à vos retours que ce livre restera pertinent. Nous voulons savoir ce qui marche et ce qui ne marche pas. Nous voulons savoir s'il existe des points qui restent obscurs. Notamment, nous voulons savoir si vous trouvez ce livre affreux. Les commentaires positifs ou négatifs sont les bienvenus. Bien sûr nous nous réservons le droit de ne pas être d'accord avec vous, mais toute remarque sera récompensée par une jolie réponse.

3. Conventions de Police

Ce livre respecte certaines conventions quant à l'utilisation des polices de caractère. Comprendre ces conventions dès le début facilite l'utilisation de ce livre.

Italic

Utilisée pour les fichiers, les extensions, les URLs, les noms des applications, la mise en valeur, et les termes nouveaux lors de leur première utilisation.

Largeur Fixe

Utilisée pour les classes, les méthodes, les variables Java, les propriétés, les éléments en relation avec les bases de données, et les extraits de code qui apparaissent dans le texte.

Largeur Fixe Gras

Utilisée pour les commandes que vous devez taper sur une ligne de commande et pour mettre en valeur un nouvel élément de code introduit dans un exemple qui fonctionne.

Largeur fixe italique

Utilisée pour annoter les affichages.

² <mailto:tobrien@sonatype.com>

4. Conventions d'écriture Maven

Le livre respecte certaines conventions de nommage et d'utilisation des polices de caractère en accord avec Maven. Comprendre ces conventions facilite la lecture de ce livre.

`plugin Compiler`

Les plugins Maven commencent par des majuscules.

`create goal`

Les noms de goal sont affichés avec une police à largeur fixe.

`plugin`

Alors que la réelle orthographe "plug-in" (avec un tiret) est probablement plus répandue, ce livre utilise le terme "plugin" pour deux raisons : il est plus facile à lire et écrire et c'est devenu le standard pour la communauté Maven.

Cycle de vie Maven, Structure Standard Maven des Répertoires, Plugin Maven, Modèle Objet de Projet (Project Object Model)

Les concepts fondamentaux de Maven commencent par des majuscules lorsqu'il y est fait référence dans le texte.

`goalParameter`

Le paramètre d'un goal Maven est affiché avec une police à largeur fixe.

`compile phase`

Les phases du cycle de vie de Maven sont affichées avec une police à largeur fixe.

5. Remerciements

Sonatype souhaite remercier les contributeurs suivants. Les personnes citées ci-dessous ont fourni des retours qui ont permis l'amélioration de la qualité de cet ouvrage. Merci donc à Raymond Toal, Steve Daly, Paul Strack, Paul Reinerfelt, Chad Gorshing, Marcus Biel, Brian Dols, Mangalaganesh Balasubramanian, Marius Kruger et Mark Stewart. Et plus spécifiquement, merci à Joel Costigliola pour son aide à débogger et corriger le chapitre sur Spring web. Stan Guillory était pratiquement un contributeur au vu du nombre de corrections qu'il a posté sur le site Get Satisfaction pour ce livre. Merci Stan. Un grand merci à Richard Coasby de Bamboo pour son rôle de consultant en grammaire.

Merci à tous nos auteurs contributeurs, y compris Eric Redmond.

Merci aux contributeurs suivants qui nous ont signalé des erreurs soit par courriel soit par le site Get Satisfaction: Paco Soberón, Ray Krueger, Steinar Cook, Henning Saul, Anders Hammar, "george_007", "ksangani", Niko Mahle, Arun Kumar, Harold Shinsato, "mimil", "-thrown-", Matt Gumbley. Si vous voyez votre pseudo Get Satisfaction dans cette liste, et que vous souhaitez le voir remplacé par votre véritable nom, envoyez nous un courriel à book@sonatype.com³.

³ <mailto:book@sonatype.com>

6. Remerciements des traducteurs

Les traducteurs de la version française souhaitent remercier les contributeurs suivants. Tout d'abord Jérôme Van der Linden et Stéphane Coutant qui nous ont accompagnés sur certaines étapes de cette aventure. Un grand merci à tous nos relecteurs et à leurs très nombreux retours sans qui cette traduction ne serait pas de cette qualité : Bakann DY, Damien Bertizzolo, Cedrick Luven, Alexis Hassler, Thierry Laude et Raphaël Flores. Nous tenons tout particulièrement à remercier Laurent Huet pour sa relecture approfondie et ses remarques argumentées sur certains points de traduction ou de formulation.

Chapitre 1. Introduction à Apache Maven

Bien qu'il existe de nombreuses références à Maven sur internet, on ne trouve pas un seul document correctement écrit sur Maven et qui puisse servir à la fois de véritable référence et d'introduction. Ce que nous avons essayé de faire ici est d'écrire un tel document avec son matériel de référence.

1.1. Maven... De quoi s'agit-il ?

La réponse à cette question dépend de votre point de vue. La plus grande partie des utilisateurs de Maven vont l'appeler un "outil de build" : c'est-à-dire un outil qui permet de produire des artefacts déployables à partir du code source. Pour les gestionnaires de projet et les ingénieurs en charge du build, Maven ressemble plus à un outil de gestion de projet. Quelle est la différence ? Un outil de build comme Ant se concentre essentiellement sur les tâches de prétraitement, de compilation, de packaging, de test et de distribution. Un outil de gestion de projet comme Maven fournit un ensemble de fonctionnalités qui englobe celles d'un outil de build. Maven apporte, en plus de ses fonctionnalités de build, sa capacité à produire des rapports, générer un site web et ainsi facilite la communication entre les différents membres de l'équipe.

Voici une définition plus formelle d'Apache Maven ¹ : Maven est un outil de gestion de projet qui comprend un modèle objet pour définir un projet, un ensemble de standards, un cycle de vie, et un système de gestion des dépendances. Il embarque aussi la logique nécessaire à l'exécution d'actions pour des phases bien définies de ce cycle de vie, par le biais de plugins. Lorsque vous utilisez Maven, vous décrivez votre projet selon un modèle objet de projet clair, Maven peut alors lui appliquer la logique transverse d'un ensemble de plugins (partagés ou spécifiques).

Ne vous laissez pas impressionner par le fait que Maven est un "outil de gestion de projet". Si vous cherchiez juste un outil de build alors Maven fera l'affaire. D'ailleurs, les premiers chapitres de ce livre ne traiteront que du cas d'utilisation le plus courant : comment utiliser Maven pour construire et distribuer votre projet.

1.2. Convention plutôt que configuration

Le paradigme "Convention over Configuration" (en français convention plutôt que configuration) repose sur une idée simple. Par défaut, les systèmes informatiques, les bibliothèques et les frameworks devraient avoir un comportement raisonnable. Un système devrait être "prêt à l'emploi" sans demander de configuration superflue. De célèbres frameworks comme Ruby on Rails² et EJB3 ont commencé à appliquer ces principes en réaction à la complexité du paramétrage de frameworks tels que les

¹ <http://maven.apache.org>

² <http://www.rubyonrails.org/>

spécifications initiales EJB 2.1. On retrouve une illustration de ce principe au travers de la persistance EJB3 : pour rendre une classe persistante tout ce que vous avez à faire est de l'annoter avec `@Entity`. Le framework va considérer que les noms de la table et des colonnes seront ceux de la classe et de ses attributs. Si le besoin s'en ressent, vous pouvez surcharger ces noms prédéfinis, mais la plupart du temps, l'usage de ces conventions implicites du framework procurera un gain de temps appréciable au projet.

Maven intègre ce concept en ayant un comportement logique par défaut. Sans configuration spécifique, le code source est supposé se trouver dans `/${basedir}/src/main/java` et les différentes ressources dans `/${basedir}/src/main/resources`. Les tests, eux, sont supposés être dans `/${basedir}/src/test`, et un projet est supposé produire un fichier JAR. Maven suppose que vous voulez compiler en bytecode dans `/${basedir}/target/classes` et ensuite créer votre fichier JAR distribuable dans `/${basedir}/target`. Même si tout cela peut sembler trivial, n'oubliez pas que pour la plupart des scripts Ant vous devez définir les emplacements de ces différents répertoires. Ant n'a pas la moindre idée d'où se trouve le code source et les différentes ressources, vous devez le lui indiquer. L'adoption par Maven de ce principe de "convention plutôt que configuration" va plus loin que les répertoires, les plugins au cœur de Maven appliquent un ensemble de conventions pour compiler le code source, packager les éléments à distribuer, produire des sites web, et bien d'autres traitements. La force de Maven vient de ses "convictions", il a un cycle de vie bien défini et un ensemble de plugins de base pour construire et assembler un logiciel. Si vous suivez les conventions, Maven ne va vous demander quasiment aucun effort - vous n'avez qu'à mettre votre code source dans le bon répertoire et Maven s'occupe du reste.

Une des conséquences des systèmes respectant le principe de "convention plutôt que configuration" est que leurs utilisateurs peuvent se sentir contraints de suivre une certaine méthodologie. S'il est vrai que Maven a fait certains choix qui ne doivent pas être remis en cause, la plupart des valeurs par défaut peuvent être adaptées. Par exemple, il est tout à fait possible de modifier l'emplacement du code source et des ressources pour un projet, de redéfinir les noms des fichiers JAR, et il est possible d'adapter presque tous les comportements aux spécificités de votre projet par le développement de plugins spécifiques. Si vous ne souhaitez pas suivre les conventions, Maven vous permettra de changer les valeurs par défaut selon vos propres besoins.

1.3. Une Interface Commune

Avant que Maven ne fournisse une interface commune pour construire un logiciel, chaque projet avait une personne dédiée pour gérer son système de build complètement personnalisé. Les développeurs devaient prendre du temps sur leurs développements pour apprendre les arcanes de chaque nouveau projet auquel ils voulaient contribuer. En 2001, vous aviez une approche très différente pour construire un projet comme Turbine³ par rapport à un projet comme Tomcat⁴. Si un nouvel outil d'analyse statique du code source sortait, ou si un nouveau framework de tests unitaires était développé, tout le monde devrait s'arrêter de développer et voir comment l'intégrer dans l'environnement de build spécifique à chaque projet. Comment exécuter les tests unitaires ? Il existait des milliers de réponses à cette question.

³ <http://turbine.apache.org/>

⁴ <http://tomcat.apache.org>

Cette époque se caractérisait par des discussions sans fin sur les outils et les procédures pour construire un logiciel. Le monde d'avant Maven était un monde inefficace, l'âge de "l'Ingénieur du Build".

Aujourd'hui, la plupart des développeurs du libre ont utilisé ou utilisent Maven pour gérer leurs nouveaux projets logiciels. Cette transition n'est pas le simple passage d'un outil de build à un autre, mais l'adoption d'une interface commune de construction de projet. Pendant que les logiciels devenaient modulaires, les systèmes de build devenaient de plus en plus complexes et le nombre de projets a crevé le plafond. Avant Maven, lorsque vous vouliez récupérer le code source de projets comme Apache ActiveMQ⁵ ou Apache ServiceMix⁶ depuis Subversion et le construire à partir de ses sources, vous deviez passer plus d'une heure à essayer de comprendre comment fonctionnait le système de build de chacun de ces projets. De quoi a-t-on besoin pour construire ce projet ? Quelles bibliothèques dois-je télécharger ? Ensuite, où dois-je les mettre ? Quelles tâches dois-je exécuter dans le build ? Dans le meilleur des cas, il fallait quelques minutes pour comprendre comment construire un logiciel, dans le pire (par exemple l'antique implémentation de l'API Servlet du projet Jakarta), construire le logiciel était si complexe qu'il fallait plusieurs heures à un nouveau contributeur pour pouvoir modifier le code source et compiler le projet. De nos jours, il suffit de récupérer le source et d'exécuter la commande **mvn install**.

Même si Maven fournit tout un ensemble d'avantages, dont la gestion des dépendances et la réutilisation de comportements communs de build par ses plugins, la raison principale de son succès vient de la création d'une interface unifiée pour construire un logiciel. Si vous voyez qu'un projet comme Apache Wicket⁷ utilise Maven, vous pouvez supposer qu'après avoir récupéré son code source, la commande **mvn install** vous permettra de le construire sans trop de problèmes. Vous savez où insérer la clef de contact, que la pédale d'accélérateur se trouve à droite et le frein à gauche.

1.4. Réutilisation universelle grâce aux plugins Maven

Le cœur de Maven est assez stupide, il ne sait pas faire grand-chose à part parser quelques documents XML et garder les traces d'un cycle de vie et de l'exécution de quelques plugins. Maven a été conçu pour déléguer la responsabilité du build à un ensemble de plugins Maven qui vont affecter le cycle de vie de Maven et fournir différentes actions : les goals. Avec Maven, tout se passe dans les goals des plugins, c'est là que le code source est compilé, que le bytecode est packagé, que les sites sont publiés et que toute autre tâche nécessaire à votre build se produit. Le Maven que vous téléchargez chez Apache n'y connaît rien en packaging de fichier WAR ou en exécution de tests JUnit ; la plus grande partie de l'intelligence de Maven se trouve dans les plugins, plugins qui sont récupérés du dépôt Maven. En effet, la première fois que vous exécutez une commande comme **mvn install** avec une installation de Maven vierge, elle télécharge les plugins Maven de base du dépôt Maven Central. C'est plus qu'une astuce pour réduire la taille de la distribution Maven à télécharger, c'est par ce moyen que vous pouvez mettre à jour un plugin pour apporter de nouvelles possibilités au build de votre projet. C'est parce que

⁵ <http://activemq.apache.org>

⁶ <http://servicemix.apache.org>

⁷ <http://wicket.apache.org>

Maven récupère les dépendances et les plugins depuis des dépôts distants que vous pouvez réutiliser une logique de build universelle.

Le plugin Maven Surefire est le plugin qui a en charge l'exécution des tests unitaires. À un moment donné, entre la version 1.0 et la version utilisée actuellement quelqu'un a décidé d'apporter le support du framework de tests unitaires TestNG en plus de celui de JUnit. Cette mise à jour s'est faite sans casser la compatibilité ascendante. Si vous utilisez le plugin Surefire pour compiler et exécuter vos tests unitaires JUnit 3, et que vous le mettez à jour, vos tests continueront de s'exécuter sans erreur. Mais vous avez obtenu une nouvelle fonctionnalité, vous pouvez exécuter des tests avec TestNG. Et en plus, vous pouvez exécuter des tests unitaires JUnit 4 annotés. Tout cela sans avoir à mettre à jour votre installation de Maven ou à installer quoi que ce soit. Et plus important encore, vous n'avez rien changé à votre projet si ce n'est le numéro de version d'un plugin dans un unique fichier de configuration de Maven, le Project Object Model (POM).

C'est ce même mécanisme que l'on retrouve dans tout Maven. Maven dispose de plugins pour tout faire, de la compilation du code Java à la génération de rapports et au déploiement sur un serveur d'applications. Maven a extrait les tâches de la construction d'un projet dans des plugins qui sont centralisés pour leur maintenance et partagés universellement. Si l'état de l'art change pour une étape quelconque du build, si un nouveau framework de tests unitaires sort, si de nouveaux outils deviennent disponibles, vous n'avez plus à ajouter une nouvelle verrue à votre système personnalisé de build pour en profiter. Vous allez bénéficier du fait que les plugins sont téléchargés depuis un dépôt distant et maintenus centralement. C'est tout cela qu'implique la notion de réutilisation universelle par les plugins Maven.

1.5. Le Modèle conceptuel d'un "Projet"

Avec Maven vous modélisez un projet. Vous ne faites plus simplement de la compilation de code en bytecode, vous décrivez un projet logiciel et vous lui assignez un ensemble unique de coordonnées. Vous définissez les attributs qui lui sont propres. Quelle est sa licence ? Quels sont ses développeurs et ses contributeurs ? De quels autres projets dépend-il ? Maven est plus qu'un simple "outil de build", c'est plus qu'une amélioration des outils tels que Ant et make, c'est une plateforme qui s'appuie sur de nouvelles sémantiques pour les projets logiciels et le développement. La définition d'un modèle pour tous les projets fait émerger de nouvelles caractéristiques telles que :

La gestion des dépendances

Puisque chaque projet est identifié de manière unique par un triplet composé d'un identifiant de groupe, un identifiant d'artefact et un numéro de version, les projets peuvent utiliser ces coordonnées pour déclarer leurs dépendances.

Des dépôts distants

En liaison avec la gestion de dépendance, nous pouvons utiliser les coordonnées définies dans le Project Object Model (POM) de Maven pour construire des dépôts d'artefacts Maven.

Réutilisation universelle de la logique de build

Les plugins contiennent toute la logique de traitement. Ils s'appuient sur les données et paramètres de configuration définis dans le Project Object Model (POM). Ils ne sont pas conçus pour fonctionner avec des fichiers spécifiques à des endroits connus.

Portabilité / Intégration dans des outils

Les outils tels qu'Eclipse, NetBeans, et IntelliJ ont maintenant un endroit unique pour aller récupérer les informations sur un projet. Avant Maven, chaque EDI conservait à sa manière ce qui était, plus ou moins, son propre Project Object Model (POM). Maven a standardisé cette description, et alors que chaque EDI continue à maintenir ses propres fichiers décrivant le projet, ils peuvent être facilement générés à partir du modèle.

Facilités pour la recherche et le filtrage des artefacts d'un projet

Des outils tels que Nexus vous permettent d'indexer et de rechercher les contenus d'un dépôt à partir des informations contenues dans le POM.

1.6. Maven est-il une alternative à XYZ ?

Bien sûr, Maven est une alternative à Ant, mais Apache Ant⁸ continue à être un outil excellent, et largement utilisé. Il fut le champion des builds Java pendant des années, et vous pouvez intégrer vos scripts Ant au build Maven de votre projet très facilement. C'est une utilisation très commune dans un projet Maven. D'un autre côté, tandis que de plus en plus de projets open source migrent vers la plateforme de gestion de projet Maven, les développeurs se rendent compte que Maven ne fait pas que simplifier la gestion du build, il permet l'émergence d'une interface commune entre les développeurs et les projets logiciels. Maven est plus une plateforme qu'un outil, lorsque vous considérez Maven comme une alternative à Ant, vous comparez des pommes à des oranges. "Maven" est plus qu'un simple outil de build.

C'est cela qui rend les débats sur Maven ou Ant, Maven ou Buildr, Maven ou Gradle inintéressants. Maven ne se réduit pas aux seuls mécanismes de votre système de build, il ne se contente pas de scripter les différentes tâches de votre build, mais il encourage l'émergence d'un ensemble de standards, d'une interface commune, d'un cycle de vie, d'un format de dépôt, d'un standard d'organisation des répertoires, etc. Peu importe le format du POM (XML ou YAML ou Ruby). Maven est bien plus que tout cela, et Maven fait référence à bien plus que l'outil en lui-même. Quand ce livre parle de Maven, il fait référence à la constellation de logiciels, de systèmes, et de standards qui le supportent. Buildr, Ivy, Gradle, tous ces outils interagissent avec le format de dépôt que Maven a permis de créer, et vous pourriez tout aussi facilement utiliser un gestionnaire de dépôt comme Nexus pour gérer des builds écrits entièrement avec Ant.

Bien que Maven soit une alternative à ces outils, la communauté doit abandonner cet esprit de lutte sanglante pour s'approprier utilisateurs et développeurs. C'est peut-être ainsi que ça se passe entre grosses entreprises, mais cela ne correspond pas à la manière de faire des communautés Open Source.

⁸ <http://ant.apache.org>

Les gros titres tels que "Qui sera le vainqueur ? Ant ou Maven ?" ne sont pas très constructifs. Si vous exigez de nous une réponse, bien sûr que nous dirons que Maven est une meilleure alternative que Ant comme technologie fondamentale pour un build ; en même temps, les frontières de Maven changent tout le temps, et la communauté Maven cherche en permanence de nouvelles voies pour le rendre plus œcuménique, plus interopérable, plus coopératif. Les principes au cœur de Maven sont le build déclaratif, la gestion des dépendances, les dépôts, la réutilisation universelle grâce aux plugins, cependant la concrétisation de ces idées n'a que peu d'importance par rapport au fait que la communauté Open Source collabore pour améliorer l'efficacité des builds à "l'échelle de l'entreprise".

1.7. Comparaison de Maven et de Ant

Les auteurs de ce livre n'ont pas pour objectif de créer de l'animosité entre Apache Ant et Apache Maven. Cependant nous sommes conscients que, pour la plupart des organisations, il faut faire un choix entre Apache Ant et Apache Maven. Dans cette section, nous allons comparer ces deux outils.

Ant excelle dans le processus de build, c'est un système de build héritant de make avec des cibles et des dépendances. Chaque cible se compose d'un ensemble d'instructions codées en XML. Il existe une tâche `copy`, une tâche `javac`, tout comme une tâche `jar`. Quand vous utilisez Ant, vous lui fournissez un ensemble d'instructions spécifiques pour compiler et packager le résultat. Prenons par exemple ce simple fichier `build.xml` :

Exemple 1.1. Simple fichier `build.xml` pour Ant

```
<project name="my-project" default="dist" basedir=". ">
  <description>
    simple example build file
  </description>
  <!-- set global properties for this build -->
  <property name="src" location="src/main/java"/>
  <property name="build" location="target/classes"/>
  <property name="dist" location="target"/>

  <target name="init">
    <!-- Create the time stamp -->
    <tstamp/>
    <!-- Create the build directory structure used by compile -->
    <mkdir dir="${build}"/>
  </target>

  <target name="compile" depends="init"
    description="compile the source " >
    <!-- Compile the java code from ${src} into ${build} -->
    <javac srcdir="${src}" destdir="${build}"/>
  </target>

  <target name="dist" depends="compile"
    description="generate the distribution" >
    <!-- Create the distribution directory -->
    <mkdir dir="${dist}/lib"/>
```

```

    <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
    <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
</target>

<target name="clean"
        description="clean up" >
    <!-- Delete the ${build} and ${dist} directory trees -->
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
</target>
</project>

```

Dans cet exemple, on peut voir qu'il faut dire à Ant exactement ce qu'il doit faire. On a une cible compile qui inclut la tâche `javac` pour compiler le code source du répertoire `src/main/java` dans le répertoire `target/classes`. Vous devez indiquer à Ant exactement où se trouve votre code source, où le bytecode produit devra être sauvé, et comment packager tout cela dans un fichier JAR. Même si des développements récents tendent à rendre Ant moins procédural, pour un développeur, Ant reste un langage de code procédural en XML.

Comparons l'exemple Ant précédent avec un exemple Maven. Avec Maven, pour créer un fichier JAR à partir de code source Java, tout ce que vous avez à faire est de créer un simple fichier `pom.xml`, mettre votre code source dans `${basedir}/src/main/java` et exécuter la commande `mvn install` depuis la ligne de commande. Le contenu du fichier `pom.xml` Maven qui permet d'obtenir le même résultat que le simple fichier Ant décrit dans l'Exemple 1.1, « Simple fichier build.xml pour Ant » est le suivant :

Exemple 1.2. Simple fichier pom.xml pour Maven

```

<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
</project>

```

C'est tout ce dont vous avez besoin dans votre `pom.xml`. Exécuter `mvn install` depuis la ligne de commande va traiter les ressources, compiler le source, exécuter les tests unitaires, créer un JAR, et installer ce JAR dans le dépôt local pour qu'il soit réutilisable par d'autres projets. Toujours sans rien modifier, vous pouvez exécuter la commande `mvn site` et vous trouverez un fichier `index.html` dans `target/site` contenant des liens vers la JavaDoc et quelques rapports sur votre code source.

Certes, c'est l'exemple de projet le plus simple possible. Il ne contient rien d'autre que du code source et produit un JAR. C'est un projet qui suit les conventions de Maven et qui ne demande aucune dépendance ou personnalisation. Si nous voulons personnaliser le comportement, notre fichier `pom.xml` va grossir, et dans les plus gros projets, vous pouvez trouver une collection de POMs Maven très complexes qui contiennent beaucoup de configuration de plugins et de déclarations de dépendances. Mais, même lorsque les fichiers POM de votre projet grossissent, ils contiennent des informations d'une nature différente de celles du fichier de build Ant d'un projet de taille similaire. Les POMs Maven contiennent

des déclarations : "C'est un projet JAR" ou "Le code source se trouve dans `src/main/java`". Les scripts Ant contiennent des instructions explicites : "Ceci est le projet", "Le code source se trouve dans `src/main/java`", "Exécuter javac sur ce répertoire", "Mettre les résultats dans `target/classes`", "Créer un JAR à partir de ...", etc. Là où Ant se doit d'être explicite avec le traitement, il y a quelque chose "d'inné" chez Maven qui sait où se trouve le code source et ce qu'il doit en faire.

Les différences entre Ant et Maven dans cet exemple sont :

Apache Ant

- Ant ne dispose d'aucune convention formelle ni comportement par défaut. Il vous faut indiquer à Ant exactement où trouver le source et où mettre les résultats. Des conventions informelles ont émergé au cours du temps, mais elles n'ont pas été intégrées au produit.
- Ant est procédural. Il faut lui indiquer exactement ce qu'il doit faire et quand le faire. Il faut lui dire de compiler, puis de copier, puis de compresser.
- Ant n'a pas de cycle de vie. Vous devez définir des fonctions et les dépendances de ces fonctions. Vous devez ensuite rattacher une série de tâches à chaque fonction manuellement.

Apache Maven

- Maven possède ses conventions. Il sait où se trouve le code source car vous suivez une convention. Le plugin Maven Compiler place le bytecode dans `target/classes` et produit un fichier JAR dans le répertoire `target`.
- Maven est déclaratif. Tout ce que vous avez à faire est de créer un fichier `pom.xml` et mettre votre code source dans le répertoire par défaut. Maven s'occupe du reste.
- Maven possède son propre cycle de vie que vous invoquez lorsque vous exécutez la commande **`mvn install`**. Cette commande demande à Maven d'exécuter une série d'étapes jusqu'à la fin du cycle de vie. Une des conséquences de ce parcours du cycle de vie, est que Maven exécute un certain nombre de goals de plugins par défaut qui vont réaliser certaines tâches comme de compiler et de créer un JAR.

Maven intègre une certaine intelligence sur les tâches communes sous la forme de plugins Maven. Si vous voulez écrire et exécuter des tests unitaires, tout ce que vous avez à faire est d'écrire ces tests et de les mettre dans `${basedir}/src/test/java`, ajouter une dépendance de type test sur TestNG ou JUnit, et exécuter la commande **`mvn test`**. Si vous voulez déployer une application web et non plus un JAR, tout ce que vous avez à faire est de changer le type de votre projet en `war` et de positionner le répertoire racine de l'application web sur `${basedir}/src/main/webapp`. Bien sûr, vous pourriez faire tout cela avec Ant mais vous devriez tout écrire depuis le début. Dans Ant, vous devriez commencer par définir où devrait se trouver le fichier JAR de JUnit, ensuite vous devriez créer un classpath qui contienne ce fichier JAR de JUnit, puis vous devriez indiquer à Ant où devrait se trouver le code source des tests, écrire une fonction pour compiler le code source des tests en bytecode, et pour exécuter ces tests unitaires avec JUnit.

Sans l'apport des technologies comme antlibs et Ivy (et même avec ces technologies complémentaires), Ant laisse l'impression d'un build procédural personnalisé. Un ensemble efficace de POMs Maven dans un projet qui respecte les conventions de Maven a peu de XML, aussi surprenant que cela puisse paraître, en comparaison de Ant. Un autre avantage de Maven est qu'il s'appuie sur des plugins Maven très largement diffusés et partagés. Tout le monde utilise le plugin Maven Surefire pour les tests unitaires, et si quelqu'un ajoute le support d'un nouveau framework de tests unitaires, vous obtenez cette nouvelle fonctionnalité dans votre build juste en incrémentant le numéro de version de ce plugin particulier dans le POM de votre projet.

La décision d'utiliser Maven ou Ant n'est pas une décision binaire, et Ant a toujours sa place pour les builds complexes. Si votre build contient un traitement très spécifique, ou si vous avez écrit des scripts Ant pour remplir une tâche particulière de manière spécifique qui ne peut être adaptée aux standards Maven, vous pouvez toujours utiliser ces scripts dans Maven. Ant est disponible sous la forme d'un des principaux plugins de Maven. Des plugins personnalisés de Maven peuvent être écrits en utilisant Ant, et les projets Maven peuvent être configurés pour exécuter des scripts Ant durant le cycle de vie du projet Maven.

Chapitre 2. Installation de Maven

Ce chapitre contient les instructions détaillées pour installer Maven sur de nombreuses plateformes. Plutôt que de supposer que vous sachiez comment installer un logiciel et définir des variables d'environnement, nous avons fait le choix d'être aussi détaillés que possible afin de réduire le nombre de problèmes que vous pourriez rencontrer suite à une installation partielle. La seule chose que nous allons supposer dans ce chapitre est que vous avez déjà installé un JDK (Java Development Kit) approprié. Si vous n'êtes intéressé que par l'utilisation elle-même, vous pouvez passer à la suite du livre après avoir lu les paragraphes "Téléchargement de Maven" et "Installer Maven". Si vous vous intéressez aux détails de votre installation de Maven, ce chapitre y est entièrement consacré et vous donne une vision d'ensemble de ce que vous avez installé et de la licence Apache Software License, Version 2.0.

2.1. Vérifier votre installation de Java

Même si Maven (pour les versions inférieures à la 2.1) peut s'exécuter avec Java 1.4, dans ce livre nous supposons que vous avez installé au minimum Java 5. Utilisez la version stable la plus récente du JDK (Java Development Kit) disponible pour votre système d'exploitation. Java 5 comme Java 6 permettront d'exécuter les exemples de ce livre.

```
% java -version
java version "1.5.0_16"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.5.0_16-b06-284)
Java HotSpot(TM) Client VM (build 1.5.0_16-133, mixed mode, sharing)
```

Maven fonctionne avec tous les kits de développement Java TM certifiés compatibles et également avec certaines implémentations de Java non certifiées. Les exemples fournis avec ce livre ont été écrits et testés avec les versions officielles du Java Development Kit téléchargées depuis le site internet de Sun Microsystems. Si vous êtes sous GNU/Linux, vous pouvez avoir besoin de télécharger le JDK de Sun vous-même et vérifier qu'il s'agit de la version que vous utilisez (en exécutant la commande `java -version`). Maintenant que Sun a mis Java en Open Source, cela devrait s'améliorer et nous devrions rapidement trouver la JRE et le JDK de Sun par défaut même dans les distributions puristes de GNU/Linux. En attendant ce jour, vous pouvez avoir à le télécharger.

2.2. Téléchargement de Maven

Vous pouvez télécharger Maven depuis le site internet du projet Apache Maven avec l'url suivante <http://maven.apache.org/download.html> ¹.

Lorsque vous téléchargez Maven, faites attention à choisir la dernière version de Apache Maven disponible sur le site de Maven. La dernière version de Maven lorsque ce livre a été écrit était Maven 2.2.1. Si l'Apache Software License ne vous dit rien, nous vous suggérons de vous familiariser avec

¹ <http://maven.apache.org/download.html>

les termes de cette licence avant de commencer à utiliser le produit. Plus d'informations sur l'Apache Software License se trouvent dans la Section 2.8, « À propos de l'Apache Software License ».

2.3. Installer Maven

Il existe de grandes différences entre un système d'exploitation comme Mac OS X et Microsoft Windows, il en est de même entre les différentes versions de Windows. Heureusement, le processus d'installation de Maven est relativement simple et sans douleur sur ces systèmes d'exploitation. Les sections qui suivent mettent en relief les bonnes pratiques pour installer Maven sur une palette de systèmes d'exploitation.

2.3.1. Installer Maven sur Mac OSX

Vous pouvez télécharger une version binaire de Maven depuis <http://maven.apache.org/download.html>. Téléchargez la version courante de Maven dans le format qui vous convient le mieux. Choisissez l'emplacement où vous voulez l'installer et décompressez l'archive à cet endroit. Si vous avez décompressé l'archive dans le répertoire `/usr/local/apache-maven-2.2.1`, vous pouvez vouloir créer un lien symbolique afin de vous faciliter la vie et éviter d'avoir à modifier l'environnement.

```
/usr/local % cd /usr/local
/usr/local % ln -s apache-maven-2.2.1 maven
/usr/local % export M2_HOME=/usr/local/maven
/usr/local % export PATH=${M2_HOME}/bin:${PATH}
```

Une fois Maven installé, il vous reste quelques petites choses à faire afin de pouvoir l'utiliser correctement. Vous devez ajouter le répertoire `bin` de la distribution (dans notre exemple `/usr/local/maven/bin`) au `PATH` de votre système. Vous devez aussi positionner la variable d'environnement `M2_HOME` sur le répertoire racine de votre installation (pour notre exemple, `/usr/local/maven`).



Note

Les instructions d'installation sont les mêmes pour OSX Tiger et OSX Leopard. On nous a rapporté que Maven 2.0.6 est livré avec la préversion de XCode. Si vous avez installé XCode, exécutez la commande `mvn` en ligne de commande pour vérifier sa disponibilité. XCode installe Maven dans le répertoire `/usr/share/maven`. Il est recommandé d'utiliser la dernière version de Maven 2.2.1, de nombreux bugs critiques ont été corrigés depuis la version 2.0.6 de Maven.

Vous allez devoir ajouter les variables `M2_HOME` et `PATH` à un script qui sera exécuté à chaque fois que vous vous connecterez à votre système. Pour cela, ajouter les lignes suivantes au `.bash_login`.

```
export M2_HOME=/usr/local/maven
export PATH=${M2_HOME}/bin:${PATH}
```

Maintenant que vous avez mis à jour votre environnement avec ces quelques lignes, vous pouvez exécuter Maven en ligne de commande.



Note

Pour ces instructions d'installation nous avons supposé que vous utilisiez le shell bash.

2.3.1.1. Installer Maven sur OSX avec MacPorts

Si vous utilisez MacPorts, vous pouvez installer le port maven2 en exécutant les instructions suivantes en ligne de commande.

```
$sudo port install maven2
Password: *****
---> Fetching maven2
---> Attempting to fetch apache-maven-2.2.1-bin.tar.bz2
from http://www.apache.org/dist/maven/binaries
---> Verifying checksum(s) for maven2
---> Extracting maven2
---> Configuring maven2
---> Building maven2 with target all
---> Staging maven2 into destroot
---> Installing maven2 2.2.1_0
---> Activating maven2 2.2.1_0
---> Cleaning maven2
```

Pour plus d'informations sur le port maven2, allez voir [maven2 Portfile](#)². Pour plus d'informations sur le projet MacPorts et son installation, allez voir la [page du projet MacPorts](#)³.

2.3.2. Installer Maven sur Microsoft Windows

L'installation de Maven sur Windows est très proche de celle sur Mac OSX. Les principales différences sont le répertoire d'installation et la définition des variables d'environnement. Dans ce livre, nous allons supposer que le répertoire d'installation de Maven est `c:\Program Files\apache-maven-2.2.1`. Cela ne change pas grand-chose si vous installez Maven dans un autre répertoire, tant que vous configurez correctement les variables d'environnement. Une fois que vous avez décompressé Maven dans le répertoire d'installation, il vous faut définir deux variables d'environnement : `PATH` et `M2_HOME`. En ligne de commande, vous pouvez définir ces variables d'environnement en tapant les instructions suivantes :

```
C:\Users\tobrien >set M2_HOME=c:\Program Files\apache-maven-2.2.1
C:\Users\tobrien >set PATH=%PATH%;%M2_HOME%\bin
```

Définir ces variables d'environnement en ligne de commande va vous permettre d'exécuter Maven dans la même console. Mais à moins de les définir comme des variables du Système avec le Panneau de Configuration, vous devrez exécuter ces deux lignes à chaque fois que vous vous connecterez à votre système. Vous devriez modifier ces deux variables avec le Panneau de Configuration de Microsoft Windows.

² <http://trac.macports.org/browser/trunk/dports/java/maven2/Portfile>

³ <http://www.macports.org/index.php>

2.3.3. Installer Maven sur GNU/Linux

Pour installer Maven sur une machine GNU/Linux, suivez la même procédure que Section 2.3.1, « Installer Maven sur Mac OSX ».

2.3.4. Installer Maven sur FreeBSD ou OpenBSD

Pour installer Maven sur une machine FreeBSD ou OpenBSD, suivez la même procédure que Section 2.3.1, « Installer Maven sur Mac OSX ».

2.4. Tester une installation Maven

Une fois Maven installé, vous pouvez vérifier sa version en exécutant la commande suivante `mvn -v` en ligne de commande. Si Maven a été correctement installé, vous devriez voir en sortie quelque chose ressemblant à cela.

```
$mvn -v
Apache Maven 2.2.1 (r801777; 2009-08-06 12:16:01-0700)
Java version: 1.5.0_16
Java home: /System/Library/Frameworks/JavaVM.framework/Versions/1.5.0/Home
Default locale: en_US, platform encoding: MacRoman
OS name: "mac os x" version: "10.5.6" arch: "i386" Family: "unix"
```

Si vous obtenez cette sortie, vous savez que Maven est disponible et prêt à être utilisé. Si vous n'obtenez pas cette sortie et que votre système d'exploitation ne peut pas trouver la commande `mvn`, vérifiez que vos variables d'environnement `PATH` et `M2_HOME` sont correctement définies.

2.5. Détails de l'installation de Maven

L'archive à télécharger de Maven a une taille d'environ 1.5 MiB⁴, cette taille réduite a été obtenue car le cœur de Maven a été conçu pour récupérer à la demande plugins et dépendances depuis des dépôts distants. Lorsque vous commencez à utiliser Maven, celui-ci va télécharger les plugins depuis un dépôt local décrit dans Section 2.5.1, « Configuration et dépôt spécifiques à l'utilisateur ». Comme vous êtes curieux, jetons un œil au répertoire d'installation de Maven.

```
/usr/local/maven $ls -pl
LICENSE.txt
NOTICE.txt
README.txt
bin/
boot/
```

⁴Avez-vous déjà acheté un disque dur de 200 Go pour vous rendre compte qu'une fois installé il faisait moins de 200 GiB ? Les ordinateurs fonctionnent en Gibibytes, mais les boutiques vendent des produits en Gigaoctets. MiB représente des Mebibyte soit 2^{20} ou 1024^2 . Ces mesures standardisées sont validées et reconnues par l'IEEE, le CIPM, et l'IEC. Pour plus d'informations sur Kibibytes, Mebibytes, Gibibytes, et Tebibytes, lisez <http://en.wikipedia.org/wiki/Mebibyte> [<http://en.wikipedia.org/wiki/Mebibyte>].

```
conf/  
lib/
```

Le fichier `LICENSE.txt` contient la licence logicielle pour Apache Maven. Cette licence sera détaillée dans la Section 2.8, « À propos de l'Apache Software License ». Le fichier `NOTICE.txt` contient des remarques et les attributions exigées par les bibliothèques dont dépend Maven. Le fichier `README.txt` contient lui les instructions d'installation. Le répertoire `bin/` contient le script `mvn` qui permet l'exécution de Maven. Dans `boot/` se trouve le fichier JAR (`classworlds-1.1.jar`) qui a pour fonction de créer le chargeur de classes (Classloader) dans lequel s'exécute Maven. Le répertoire `conf/` contient un fichier `settings.xml` global qui permet de personnaliser le comportement d'une installation de Maven. Si vous devez personnaliser Maven, il est d'usage de surcharger les paramètres dans le fichier `settings.xml` qui se trouve dans `~/.m2`. Le répertoire `lib/` contient un fichier JAR unique (`maven-core-2.2.1-uber.jar`) qui contient le cœur de Maven.



Note

A moins que vous ne travailliez sur un système Unix partagé, vous ne devriez pas avoir à modifier le fichier `settings.xml` du répertoire `M2_HOME/conf`. Modifier le fichier `settings.xml` global dans l'installation de Maven est inutile dans la plupart des cas et risque de compliquer inutilement toute mise à jour de Maven, puisque vous aurez à vous souvenir de copier ce fichier `settings.xml` modifié depuis votre ancienne installation de Maven dans la nouvelle. Si vous devez modifier le fichier `settings.xml`, vous devriez modifier le fichier `settings.xml` qui vous est propre : `~/.m2/settings.xml`.

2.5.1. Configuration et dépôt spécifiques à l'utilisateur

Une fois que vous avez commencé à réellement utiliser Maven, vous vous apercevrez que Maven a créé localement des fichiers de configuration spécifiques à l'utilisateur ainsi qu'un dépôt local dans votre répertoire utilisateur. Dans `~/.m2`, vous trouverez :

`~/.m2/settings.xml`

Un fichier contenant la configuration propre à l'utilisateur, pour l'authentification, les dépôts et les différentes informations nécessaires à la personnalisation du comportement de Maven.

`~/.m2/repository/`

Ce répertoire contient le dépôt local de Maven. Lorsque vous téléchargez une dépendance depuis un dépôt Maven distant, Maven enregistre une copie de cette dépendance dans votre dépôt local.



Note

Sous Unix (et OSX), votre répertoire utilisateur est symbolisé par un tilde (i.e. `~/bin` correspond au répertoire `/home/tobrien/bin`). Sous Windows, nous utiliserons aussi ce caractère `~` pour symboliser votre répertoire utilisateur. Ainsi, sous Windows XP votre répertoire utilisateur est `C:\Documents and Settings\tobrien` et sous Windows

Vista, votre répertoire utilisateur est `C:\Users\tobrien`. À partir de maintenant, vous devrez traduire les chemins du type `~/m2` en fonction de votre système d'exploitation.

2.5.2. Mettre à jour une installation de Maven

Si vous avez installé Maven sur une machine sous Mac OSX ou sous Unix en suivant les instructions de la Section 2.3.1, « Installer Maven sur Mac OSX » et de la Section 2.3.3, « Installer Maven sur GNU/Linux » alors installation de nouvelles versions de Maven devrait être simple. Installez tout simplement la nouvelle version de Maven (`/usr/local/maven-2.future`) à côté de l'installation existante (`/usr/local/maven-2.2.1`). Puis modifiez le lien symbolique `/usr/local/maven` de `/usr/local/maven-2.2.1` en `/usr/local/maven-2.future`. Comme vous avez déjà défini votre variable d'environnement `M2_HOME` pour qu'elle pointe sur `/usr/local/maven`, vous n'aurez pas à modifier la moindre variable d'environnement.

Si vous avez installé Maven sur une machine sous Windows, décompressez Maven dans `c:\Program Files\maven-2.future` et mettez à jour votre variable d'environnement `M2_HOME`.



Note

Si vous avez modifié le fichier global `settings.xml` de `M2_HOME/conf`, vous devrez copier le `settings.xml` dans le répertoire `conf` de la nouvelle installation de Maven.

2.5.3. Migrer de Maven 1.x à Maven 2.x

Si vous migrez de Maven 1 à Maven 2, vous allez utiliser un nouveau POM et une nouvelle structure de dépôt. Si vous avez déjà votre propre dépôt Maven 1 pour gérer vos artefacts, vous pouvez utiliser Nexus Repository Manager pour exposer un dépôt au format Maven 1 de manière à ce qu'il soit compris par des clients Maven 2. Pour plus d'informations sur le Nexus Repository Manager, nous vous encourageons à lire le livre *Repository Management with Nexus*⁵. En plus d'outils comme Nexus, vous avez la possibilité de configurer des références à des dépôts utilisant la structure `legacy`. Pour plus d'informations sur la configuration de dépôt `legacy`, regardez la Section A.2.8, « Balise `repositories` ».

Si vous avez tout un ensemble de projets Maven 1, le plugin Maven One devrait vous intéresser. Le plugin Maven One a été conçu pour faciliter la migration des projets de Maven 1 à Maven 2. Si vous avez un projet Maven 1, vous pouvez convertir son POM en exécutant le goal `one:convert` de la manière suivante:

```
$ cd my-project
$ mvn one:convert
```

`one:convert` va lire le fichier `project.xml` et produire un `pom.xml` compatible avec Maven 2. Si votre build Maven 1 a été personnalisé avec un script Jelly dans le fichier `maven.xml`, il va vous falloir tester d'autres options. Alors que Maven 1 préconisait l'utilisation de scripts Jelly pour personnaliser

⁵ <http://www.sonatype.com/books/nexus-book/reference/>

les builds, Maven 2, lui, préconise la personnalisation au travers de vos propres plugins, de plugins de script ou du plugin Maven Antrun.

Ce qu'il faut retenir lorsque l'on migre de Maven 1 à Maven 2, c'est que Maven 2 est un framework de build complètement différent. Maven 2 introduit le concept du Cycle de Vie Maven, et redéfinit les relations entre plugins. Si vous migrez de Maven 1 à Maven 2, vous devez passer du temps à apprendre les différences qui existent entre ces deux versions. Bien qu'il puisse sembler plus important de commencer par apprendre la nouvelle structure du POM, vous devriez commencer par le cycle de vie. Une fois le cycle de vie de Maven compris, vous pourrez utiliser toute la puissance de Maven.

2.6. Désinstaller Maven

La plupart des instructions d'installation demandent de décompresser l'archive de la distribution de Maven dans un répertoire et de définir quelques variables d'environnement. Si vous devez supprimer Maven de votre ordinateur, tout ce que vous avez à faire est de supprimer le répertoire d'installation de Maven ainsi que les variables d'environnement. Vous voudrez sûrement supprimer le répertoire `~/ .m2` qui contient votre dépôt local.

2.7. Obtenir de l'aide avec Maven

Même si ce livre essaye de se rapprocher d'une référence complète, nous oublierons certains sujets : nous ne couvrirons pas certaines situations particulières ainsi que certaines astuces. Le cœur de Maven est très simple, cependant la véritable valeur ajoutée de Maven s'effectue par l'intermédiaire de ces plugins, et étant donné leur nombre nous n'arriverons pas à tous les couvrir en un seul ouvrage. Vous allez rencontrer des problèmes et des fonctionnalités qui ne sont pas traités dans ce livre ; dans ce cas, nous vous suggérons ces différents points pour y chercher vos réponses :

<http://maven.apache.org>

C'est le premier endroit où chercher, le site web de Maven contient une grande quantité d'information et de documentation. Chaque plugin possède quelques pages de documentation, et il existe une série de documents "quick start" qui pourront vous servir de complément à ce livre. Bien que le site internet de Maven soit une véritable mine d'informations, il peut vite devenir frustrant d'autant qu'on s'y perd facilement. Il existe une boîte de recherche Google sur la page d'accueil du site qui va rechercher dans un ensemble de sites reconnus sur Maven. Cela donne de meilleurs résultats qu'une recherche Google classique.

La Liste de Diffusion des Utilisateurs de Maven

La liste de diffusion Maven Users est l'endroit où les utilisateurs peuvent poser leurs questions. Avant de poser votre question sur cette liste de diffusion, recherchez toute discussion antérieure qui pourrait avoir un rapport avec votre question. En effet, il est mal vu de poser une question qui a déjà été posée sans avoir vérifié auparavant les archives pour voir si quelqu'un y avait déjà répondu. Il y a un certain nombre de sites très utiles pour parcourir les archives des listes de diffusion. Nous avons pour notre part trouvé Nabble très pratique. Vous pouvez parcourir les archives de la liste de diffusion des utilisateurs ici : <http://www.nabble.com/Maven---Users->

f178.html⁶. Vous pouvez vous abonner à cette liste de diffusion en suivant les instructions disponibles ici: <http://maven.apache.org/mail-lists.html>⁷.

<http://www.sonatype.com>

Sonatype maintient une version en ligne de ce livre ainsi que des formations autour d'Apache Maven.

2.8. À propos de l'Apache Software License

Apache Maven est sous Apache Software License, Version 2.0. Si vous voulez lire cette licence, vous la trouverez dans `${M2_HOME}/LICENSE.txt` ou sur le site internet de l'Open Source Initiative ici : <http://www.opensource.org/licenses/apache2.0.php>⁸.

Il y a une forte probabilité, si vous lisez ce livre, que vous ne soyez pas un homme de loi. Si vous vous demandez ce que signifie Apache License, Version 2.0, l'Apache Software Foundation a écrit une Foire Aux Questions (FAQ) très utile sur cette licence ici : <http://www.apache.org/foundation/licence-FAQ.html>⁹. Voici la réponse à la question "Je ne suis pas un homme de loi. Que cela signifie t'il ?"

[Cette licence] vous autorise à:

- télécharger et utiliser librement tout ou partie d'un logiciel Apache, dans un but personnel, interne à une entreprise ou commercial;
- utiliser un logiciel Apache dans les packages ou les distributions que vous créez.

Elle vous interdit de:

- redistribuer n'importe quelle partie d'un logiciel Apache en se l'attribuant;
- utiliser les marques appartenant à l'Apache Software Foundation de manière à laisser entendre que la Foundation approuve votre distribution;
- utiliser les marques de l'Apache Software Foundation de manière à laisser entendre que vous avez créé le logiciel Apache en question.

Elle vous demande:

- d'inclure une copie de la licence avec chaque redistribution qui contient un logiciel Apache;
- d'attribuer clairement à l'Apache Software Foundation toute distribution qui contient un logiciel Apache.

⁶ <http://www.nabble.com/Maven---Users-f178.html>

⁷ <http://maven.apache.org/mail-lists.html>

⁸ <http://www.opensource.org/licenses/apache2.0.php>

⁹ <http://www.apache.org/foundation/licence-FAQ.html>

Elle ne vous demande pas:

- d'inclure le code source du logiciel Apache lui-même, ou toute modification que vous pourriez lui avoir apportées, dans vos distributions qui le contiennent;
- de donner à l'Apache Software Foundation toute modification que vous auriez pu faire du logiciel (même si de tels retours sont appréciés).

Ceci clôt le chapitre sur l'installation. La suite du livre contient des exemples d'utilisation de Maven.

Partie I. Maven par l'exemple

Le premier livre à traiter de Maven fut *Maven: A Developer's Notebook* (O'Reilly). Ce livre introduisait Maven en plusieurs étapes via une conversation entre vous et un collègue qui savait déjà comment utiliser Maven. L'idée derrière la série des *Developer's Notebook* (maintenant arrêtée) était que les développeurs apprennent mieux lorsqu'ils sont assis à côté d'autres développeurs et qu'ils passent par les mêmes schémas de pensée, apprenant à coder en réalisant et en expérimentant. Même si cette série fut un succès, le format Cahier avait des limitations. Les Cahiers du Développeur sont, par essence, des livres "avec un but précis" qui vous font passer par une série d'étapes pour atteindre des buts très spécifiques. Au contraire, les gros livres de référence (les livres avec un animal chez O'Reilly) apportent un matériel de référence qui couvre un sujet dans son entier.

Si vous lisez *Maven: A Developer's Notebook*, vous apprendrez comment créer un projet simple ou un projet qui construit un WAR à partir de fichiers source. Mais si vous voulez comprendre les spécificités d'un point particulier comme le plugin Assembly, vous vous retrouverez dans une impasse. Comme il n'existe pas de matériel de référence correctement écrit sur Maven, vous devrez partir à la chasse dans la documentation des plugins sur le site de Maven ou fouiller les listes de diffusion. Une fois que vous vous êtes réellement plongé dans Maven, vous avez dû lire des milliers de pages HTML sur le site de Maven écrites par des centaines de développeurs différents, chacun avec sa propre idée de ce que doit contenir la documentation d'un plugin. Malgré les efforts de contributeurs de bonne volonté, lire la documentation des plugins sur le site de Maven est au mieux frustrant, et au pire, suffisamment décourageant pour abandonner Maven. Assez souvent, les utilisateurs de Maven restent bloqués car ils ne trouvent pas de réponse.

Cette absence d'un manuel de référence (voire ultime) a bloqué Maven pendant quelques années, et a été un frein à son adoption. Avec *Maven: The Definitive Guide*, nous avons l'intention de changer tout ça en fournissant une référence complète dans la Partie II, « Maven - La Référence ». Dans la Partie I, nous avons conservé la progression narrative d'un Cahier du Développeur ; elle est intéressante car elle permet d'apprendre Maven par l'exemple. Dans cette partie nous "introduisons en faisant", et dans la Partie II, « Maven - La Référence », nous comblons les manques et nous nous plongeons dans les détails. Là où la Partie II, « Maven - La Référence » pourrait utiliser un tableau de référence et un listing de programme déconnecté d'un projet exemple, la Partie II est animée par de véritables exemples.

Après avoir lu cette partie, vous devriez en savoir suffisamment pour commencer à utiliser Maven. Vous auriez besoin de vous référer à la Partie II, « Maven - La Référence » uniquement lorsque vous commencer à personnaliser Maven en écrivant vos propres plugins ou si vous voulez plus de détails sur un plugin particulier.

Chapitre 3. Mon premier projet avec Maven

3.1. Introduction

Dans ce chapitre, nous présentons un projet simple créé à partir de rien en utilisant le plugin Maven Archetype. Cette application basique va nous fournir l'occasion de discuter certains des concepts au cœur de Maven tout en suivant le développement du projet.

Avant de nous lancer dans l'utilisation de Maven sur des builds complexes avec plusieurs modules, nous allons commencer par les bases. Si vous avez déjà utilisé Maven auparavant, vous verrez qu'il fait bien son travail et s'occupe des détails. Vos builds tendent à "fonctionner correctement" et vous n'avez à vous plonger dans les détails de Maven que lorsque vous voulez personnaliser un comportement par défaut ou écrire votre propre plugin. Cependant, quand vous devez vous plonger dans ces détails, une bonne compréhension des principaux concepts est essentielle. Ce chapitre a pour but de vous présenter le projet Maven le plus simple possible puis de vous montrer les principaux concepts de Maven qui en font une solide plateforme de build. Une fois que vous l'aurez lu, vous aurez une compréhension fondamentale du cycle de vie du build, des dépôts Maven et du Project Object Model (POM).

3.1.1. Télécharger l'exemple de ce chapitre

Ce chapitre développe un exemple très simple qui sera utilisé pour explorer les principaux concepts de Maven. Si vous suivez les étapes décrites dans ce chapitre, vous ne devriez pas avoir besoin de télécharger les exemples pour recréer le code produit par Maven. Nous allons utiliser le plugin Maven Archetype pour créer ce projet simple que ce chapitre ne modifie en rien. Si vous préférez lire ce chapitre avec le code source final de l'exemple, le projet qui sert d'exemple dans ce chapitre peut être téléchargé avec le code source des exemples du livre depuis :

```
http://www.sonatype.com/books/maven-book/mavenbook-examples-0.9-SNAPSHOT-project.zip
```

Décompressez cette archive dans le répertoire de votre choix, puis allez dans le répertoire `ch-simple/`. Vous y trouverez un répertoire `simple/` qui contient le code source de ce chapitre.

3.2. Création du projet Simple

Pour commencer un nouveau projet Maven, utilisez le plugin Maven Archetype depuis la ligne de commande. Exécutez le goal `archetype:generate`, sélectionnez l'archetype numéro 15, et tapez "Y" pour confirmer et produire le nouveau projet :

```
$ mvn archetype:generate -DgroupId=org.sonatype.mavenbook.simple \  
                        -DartifactId=simple \  
                        -DpackageName=org.sonatype.mavenbook \  
                        -DprojectName=org.sonatype.mavenbook.simple
```

```

-Dversion=1.0-SNAPSHOT
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] -----
[INFO] Building Maven Default Project
[INFO] task-segment: [archetype:generate] (aggregator-style)
[INFO] -----
[INFO] Preparing archetype:generate
[INFO] No goals needed for project - skipping
[INFO] Setting property: velocimacro.messages.on => 'false'.
[INFO] Setting property: resource.loader => 'classpath'.
[INFO] Setting property: resource.manager.logwhenfound => 'false'.
[INFO] [archetype:generate {execution: default-cli}]
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart \
      (org.apache.maven.archetypes:maven-archetype-quickstart:1.0)
Choose archetype:
...
12: internal -> maven-archetype-mojo (A Maven Java plugin development project)
13: internal -> maven-archetype-portlet (A simple portlet application)
14: internal -> maven-archetype-profiles ()
15: internal -> maven-archetype-quickstart ()
16: internal -> maven-archetype-site-simple (A simple site generation project)
17: internal -> maven-archetype-site (A more complex site project)
18: internal -> maven-archetype-webapp (A simple Java web application)
19: internal -> jini-service-archetype (Archetype for Jini service project creation)
Choose a number: (...) 15: : 15
Confirm properties configuration:
groupId: org.sonatype.mavenbook.simple
artifactId: simple
version: 1.0-SNAPSHOT
package: org.sonatype.mavenbook.simple
Y: : Y
...
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.simple
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook.simple
[INFO] Parameter: package, Value: org.sonatype.mavenbook.simple
[INFO] Parameter: artifactId, Value: simple
[INFO] Parameter: basedir, Value: /private/tmp
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] BUILD SUCCESSFUL

```

La commande `mvn` correspond à l'instruction d'exécution de Maven 2. L'élément `archetype:generate` correspond à un goal Maven. Si vous connaissez Apache Ant, un goal Maven est analogue à une target Ant ; tous les deux décrivent une unité de tâche à accomplir lors d'un build. La paire `-Dname=value` représente un argument passé au goal sous la forme de propriété `-D`, comme pour les propriétés système que vous pourriez passer à la Machine Virtuelle Java via la ligne de commande. Le but du goal `archetype:generate` est de créer rapidement un projet à partir d'un archétype. Dans ce contexte, un archétype se définit comme "un modèle original ou idéal d'après lequel sont bâtis un ouvrage, une œuvre ; un prototype"¹. Il existe un grand nombre d'archétypes disponibles

¹Selon l'American Heritage Dictionary of the English Language.

pour Maven, depuis une simple application Swing jusqu'à une application web complexe. Le goal `archetype:generate` propose de choisir parmi environ 40 archétypes. Dans ce chapitre, nous allons utiliser l'archétype le plus basique pour créer un simple squelette de projet permettant de démarrer. Le préfixe `archetype` correspond au plugin, et `generate` correspond au goal.

Une fois ce projet généré, allons regarder la structure de répertoire Maven qui a été créée sous le répertoire simple.

```
simple/❶
simple/pom.xml❷
  /src/
    /src/main/❸
      /main/java
    /src/test/❹
      /test/java
```

Ce répertoire suit les recommandations de la disposition Maven standard des répertoires. Nous détaillerons cela plus tard dans ce chapitre, pour l'instant essayons de comprendre ces quelques répertoires :

- ❶ Le plugin Maven Archetype crée un répertoire `simple/` dont le nom correspond à l'`artifactId`. C'est ce qu'on appelle le répertoire racine du projet.
- ❷ Chaque projet Maven possède ce qu'on appelle un Project Object Model (POM) dans un fichier `pom.xml`. Ce fichier décrit le projet, configure les plugins, et déclare les dépendances.
- ❸ Les sources et les ressources de notre projet se retrouvent sous le répertoire `src/main`. Dans le cas de notre projet Java simple, cela consistera en quelques classes Java et fichiers de propriétés. Pour un autre projet, on pourrait y trouver le répertoire racine d'une application web ou les fichiers de configuration d'un serveur d'applications. Dans un projet Java, les classes Java sont dans `src/main/java` et les ressources disponibles dans le classpath vont dans `src/main/resources`.
- ❹ Les tests de notre projet vont dans `src/test`. Dans ce répertoire, les classes Java, comme par exemple les tests JUnit ou TestNG, sont dans `src/test/java` et les ressources du classpath pour les tests vont dans `src/test/resources`.

Le plugin Maven Archetype a produit une classe unique `org.sonatype.mavenbook.App`, qui contient 13 lignes de Java avec une fonction statique `main` qui affiche un message :

```
package org.sonatype.mavenbook;

/**
 * Hello world!
 *
 */
public class App
{
    public static void main( String[] args )
    {
        System.out.println( "Hello World!" );
    }
}
```

```
}  
}
```

Le plus simple des archétypes Maven produit le programme le plus simple possible : un programme qui affiche "Hello World!" sur la sortie standard.

3.3. Construire le projet Simple

Une fois le projet créé grâce au plugin Maven Archetype suivant les instructions de la section précédente (Section 3.2, « Création du projet Simple »), il faut construire et packager l'application. Afin d'y parvenir, exécutez `mvn install` depuis le répertoire qui contient le fichier `pom.xml` :

```
$ cd simple  
$ mvn install  
[INFO] Scanning for projects...  
[INFO] -----  
[INFO] Building simple  
[INFO]    task-segment: [install]  
[INFO] -----  
[INFO] [resources:resources]  
[INFO] Using default encoding to copy filtered resources.  
[INFO] [compiler:compile]  
[INFO] Compiling 1 source file to /simple/target/classes  
[INFO] [resources:testResources]  
[INFO] Using default encoding to copy filtered resources.  
[INFO] [compiler:testCompile]  
[INFO] Compiling 1 source file to /simple/target/test-classes  
[INFO] [surefire:test]  
[INFO] Surefire report directory: /simple/target/surefire-reports  
  
-----  
T E S T S  
-----  
Running org.sonatype.mavenbook.AppTest  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.105 sec  
  
Results:  
  
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0  
  
[INFO] [jar:jar]  
[INFO] Building jar: /simple/target/simple-1.0-SNAPSHOT.jar  
[INFO] [install:install]  
[INFO] Installing /simple/target/simple-1.0-SNAPSHOT.jar to \  
  ~/.m2/repository/com/sonatype/maven/simple/simple/1.0-SNAPSHOT/ \  
  simple-1.0-SNAPSHOT.jar
```

Vous venez juste de créer, compiler, tester, packager et installer le projet Maven le plus simple possible. Pour vous prouver que ce programme fonctionne, exécutez-le depuis la ligne de commande.

```
$ java -cp target/simple-1.0-SNAPSHOT.jar org.sonatype.mavenbook.App
```

3.4. Modèle Objet du projet Simple

Lors de son exécution, Maven lit les informations du projet dans le Project Object Model. Le POM répond aux questions telles que : De quel type de projet s'agit-il ? Quel est le nom du projet ? Ce projet a-t-il été personnalisé ? L'Exemple 3.1, « Fichier `pom.xml` du projet Simple » montre le fichier `pom.xml` par défaut créé par le goal `generate` du plugin Maven Archetype.

Exemple 3.1. Fichier `pom.xml` du projet Simple

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.simple</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Ce fichier `pom.xml` est le POM le plus basique que vous rencontrerez dans un projet Maven. La plupart du temps un fichier POM est considérablement plus complexe : il définit de nombreuses dépendances et personnalise le comportement des plugins. Les premiers éléments — `groupId`, `artifactId`, `packaging`, `version` — sont ce que l'on appelle les coordonnées Maven qui identifient de manière unique un projet. Les balises `name` et `url` sont des éléments descriptifs du POM donnant un nom compréhensible par un être humain et associant le projet à un site web. L'élément `dependencies` définit une dépendance unique, dans le scope `test`, sur un framework de test appelé JUnit. Nous reverrons ces sujets par la suite dans la Section 3.5, « Les concepts principaux ». Tout ce que vous devez savoir, pour le moment, c'est que c'est le fichier `pom.xml` qui permet à Maven de s'exécuter.

Maven s'exécute toujours selon un POM effectif, une combinaison de la configuration du fichier `pom.xml` du projet, de l'ensemble des POMs parents, d'un super-POM défini dans Maven, de la configuration de l'utilisateur et des profils actifs. Tous les projets étendent au final le super-POM, qui

définit une configuration par défaut raisonnable et que nous décrivons en détails dans le Chapitre 9, Le Modèle Objet de Projet. Même si votre projet a un fichier `pom.xml` minimal, le contenu du POM de celui-ci est interpolé à partir des contenus des POMs parents, de la configuration de l'utilisateur et de tout profil actif. Pour voir ce POM "effectif", exécutez la commande suivante depuis le répertoire racine de votre simple projet.

```
$mvn help:effective-pom
```

Lorsque vous exécutez cette commande vous pouvez voir un POM nettement plus consistant qui montre la configuration par défaut de Maven. Ce goal devient vite pratique lorsque vous essayez de déboguer un build et que vous voulez voir comment les POMs des parents de ce projet contribuent au POM effectif. Pour plus d'informations sur le plugin Maven Help, lisez la Section 12.3, « Usage du plugin Maven Help ».

3.5. Les concepts principaux

Maintenant que nous avons exécuté Maven pour la première fois, il est temps de s'intéresser aux concepts qui sont au cœur de Maven. Dans l'exemple précédent, vous avez généré un projet qui consistait en un POM et un peu de code respectant la disposition Maven standard des répertoires. Puis vous avez exécuté Maven en lui passant une phase de son cycle de vie, ce qui a demandé à Maven d'exécuter toute une série de goals de plugins Maven. Enfin, vous avez installé l'artefact produit par Maven dans votre dépôt local. Une minute ? Qu'est ce que le "cycle de vie" ? Qu'est donc un "dépôt local" ? La section qui suit définit certains des concepts au cœur de Maven.

3.5.1. Plugins Maven et Goals

Dans la section précédente, nous avons exécuté Maven en ligne de commande avec deux types d'arguments différents. La première commande appelait uniquement un goal d'un plugin, le goal `generate` du plugin `Archetype`. La deuxième exécution de Maven était une phase du cycle de vie, `install`. Pour exécuter un unique goal d'un plugin Maven, nous avons utilisé la syntaxe `mvn archetype:generate`, où `archetype` est l'identifiant du plugin et `generate` l'identifiant du goal. Lorsque Maven exécute le goal d'un plugin, il affiche l'identifiant du plugin et l'identifiant du goal sur la sortie standard :

```
$ mvn archetype:generate -DgroupId=org.sonatype.mavenbook.simple \
                        -DartifactId=simple \
                        -DpackageName=org.sonatype.mavenbook
...
[INFO] [archetype:generate]
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: \
        checking for updates from central
...
```

Un plugin Maven se compose d'un ou plusieurs goals. On peut prendre comme exemples de plugins Maven ceux qui constituent le cœur de Maven comme le plugin `Jar` dont les goals permettent de créer des

fichiers JAR, le plugin Compiler avec ses goals pour compiler le code source et le code des tests unitaires, ou le plugin Surefire dont les goals permettent l'exécution des tests unitaires et la production des rapports. On trouve aussi d'autres plugins, plus spécialisés, comme le plugin Hibernate3 pour l'intégration de la bibliothèque très connue de persistance Hibernate, le plugin JRuby qui permet l'exécution de code Ruby durant un build Maven ou l'écriture de plugins Maven en Ruby. Maven permet aussi de définir ses propres plugins. Vous pouvez écrire votre propre plugin en Java ou dans de nombreux autres langages dont Ant, Groovy, Beanshell et, comme indiqué plus haut, Ruby.

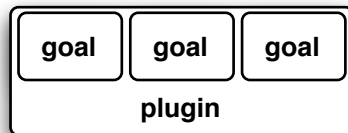


Figure 3.1. Un plugin possède des Goals

Un goal est une tâche spécifique qui peut être exécutée individuellement ou combinée à d'autres goals pour un build plus large. Un goal est une "tâche unitaire" dans Maven. On peut prendre comme exemple le goal `compile` du plugin Compiler, qui, comme son nom l'indique, compile tout le code source du projet, ou le goal `test` du plugin Surefire, qui exécute les tests unitaires. La configuration des goals s'effectue au travers de propriétés qui sont utilisées pour personnaliser le comportement. Par exemple, le goal `compile` du plugin Compiler définit un ensemble de paramètres de configuration qui vous permettent de préciser la version du JDK cible ou les options d'optimisation du compilateur. Dans l'exemple précédent, nous avons passé les paramètres de configuration `groupId` et `artifactId` au goal `generate` du plugin Archetype via les paramètres de la ligne de commande `-DgroupId=org.sonatype.mavenbook.simple` et `-DartifactId=simple`. Nous avons aussi passé le paramètre `packageName` au goal `generate` avec la valeur `org.sonatype.mavenbook`. Si nous n'avions pas précisé le paramètre `packageName`, le nom du package aurait été par défaut `org.sonatype.mavenbook.simple`.



Note

Lorsque l'on fait référence au goal d'un plugin, il est courant d'utiliser le raccourci : `pluginId:goalId`. Par exemple, pour utiliser le goal `generate` du plugin Archetype, nous pouvons écrire `archetype:generate`.

Les goals ont des paramètres qui peuvent avoir des valeurs par défaut raisonnables. Dans l'exemple de `archetype:generate`, nous n'avons pas précisé, via la ligne de commande, quel type d'archétype devait être utilisé pour créer le projet ; nous avons simplement indiqué un `groupId` et un `artifactId`. Comme nous n'avons pas fourni le type d'archétype à créer, le goal `generate` nous a demandé

d'intervenir, le goal `generate` s'est donc interrompu et nous a demandé de choisir dans une liste. Par contre, si nous avons utilisé le goal `archetype:create`, Maven aurait supposé que nous voulions créer un nouveau projet avec l'archétype `maven-archetype-quickstart`. C'est notre premier contact avec le concept de convention plutôt que configuration. La convention, ou le comportement par défaut, pour le goal `create` est de créer un projet basique à partir de l'archétype Quickstart. Le goal `create` définit une propriété de configuration `archetypeArtifactId` dont la valeur par défaut est `maven-archetype-quickstart`. L'archétype Quickstart génère le squelette d'un projet minimal qui contient un POM et une unique classe. Le plugin Archetype est bien plus puissant que cet exemple ne le laisse supposer, mais c'est une façon efficace de commencer rapidement de nouveaux projets. Plus tard dans ce livre, nous vous montrerons comment il est possible d'utiliser le plugin Archetype pour produire des projets plus complexes, comme des applications web, et comment utiliser le plugin Archetype pour définir vos propres structures de projets.

Le cœur de Maven n'intervient que très peu dans les tâches spécifiques qui composent le build de votre projet. Maven, tout seul, ignore comment compiler votre code ou produire un fichier JAR. Il délègue tout cela à des plugins Maven comme le plugin Compiler et le plugin Jar, qui sont téléchargés selon les besoins et mis à jour régulièrement depuis le dépôt central de Maven. Lorsque vous téléchargez Maven, vous n'obtenez que le cœur de la plateforme qui ne sait que parser une ligne de commande, gérer un classpath, parser un fichier POM et télécharger des plugins Maven selon les besoins. Les utilisateurs ont facilement accès aux dernières options du compilateur grâce à Maven qui maintient le plugin Compiler en dehors de son cœur et fournit un mécanisme de mise à jour. Ainsi, les plugins Maven apportent une logique de build réutilisable universellement. Vous ne définissez pas la tâche de compilation dans un fichier de build ; vous utilisez le plugin Compiler plugin qui est partagé par tous les utilisateurs de Maven. Si le plugin Compiler est amélioré, tout projet qui utilise Maven en bénéficie immédiatement. (Et, si vous n'aimez pas le plugin Compiler, vous pouvez le remplacer par votre propre implémentation.)

3.5.2. Cycle de vie de Maven

La deuxième commande que nous avons exécutée dans la section précédente était `mvn install`. Cette commande n'appelle pas le goal d'un plugin, mais une phase du cycle de vie de Maven. Une phase est une des étapes de ce que Maven appelle "le cycle de vie du build". Le cycle de vie du build est une suite ordonnée de phases aboutissant à la construction d'un projet. Maven peut supporter différents cycles de vie, mais celui qui reste le plus utilisé est le cycle de vie par défaut de Maven, qui commence par une phase de validation de l'intégrité du projet et se termine par une phase qui déploie le projet en production. Les phases du cycle de vie sont laissées vagues intentionnellement, définies comme validation, test, ou déploiement elles peuvent avoir un sens différent selon le contexte des projets. Par exemple, pour un projet qui produit une archive Java, la phase `package` va construire un JAR ; pour un projet qui produit une application web, elle produira un fichier WAR.

Les goals des plugins peuvent être rattachés à une phase du cycle de vie. Pendant que Maven parcourt les phases du cycle de vie, il exécute les goals rattachés à chacune d'entre elles. On peut rattacher de zéro à plusieurs goals à chaque phase. Dans la section précédente, quand vous avez exécuté `mvn install`, il se peut que vous ayez remarqué que plusieurs goals s'étaient exécutés. Regardez la sortie après avoir exécuté `mvn install` et relevez les différents goals qui ont été exécutés. Lorsque ce simple exemple

atteint la phase `package`, il exécute le goal `jar` du plugin `Jar`. Puisque notre projet `Quickstart` a (par défaut) un packaging de type `jar`, le goal `jar:jar` est rattaché à la phase `package`.

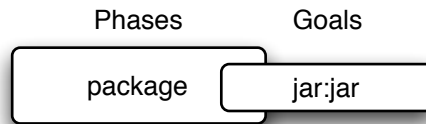


Figure 3.2. Un goal est rattaché à une phase

Nous savons que la phase `package` va créer un fichier JAR pour un projet ayant un packaging de type `jar`. Mais qu'en est-il des goals qui la précèdent, comme `compiler:compile` et `surefire:test` ? Ces goals s'exécutent lors des phases qui précèdent la phase `package` selon le cycle de vie de Maven ; exécuter une phase provoque l'exécution de l'ensemble des phases qui la précèdent, le tout se terminant par la phase spécifiée sur la ligne de commande. Chaque phase se compose de zéro à plusieurs goals, et puisque nous n'avons configuré ou personnalisé aucun plugin, cet exemple rattache un ensemble de goals standards au cycle de vie par défaut. Les goals suivants ont été exécutés dans l'ordre pendant que Maven parcourait le cycle de vie par défaut jusqu'à la phase `package` :

`resources:resources`

Le goal `resources` du plugin `Resources` est rattaché à la phase `process-resources`. Ce goal copie toutes les ressources du répertoire `src/main/resources` et des autres répertoires configurés comme contenant des ressources dans le répertoire cible.

`compiler:compile`

Le goal `compile` du plugin `Compiler` est lié à la phase `compile`. Ce goal compile tout le code source du répertoire `src/main/java` et des autres répertoires configurés comme contenant du code source dans le répertoire cible.

`resources:testResources`

Le goal `testResources` du plugin `Resources` est lié à la phase `process-test-resources`. Ce goal copie toutes les ressources du répertoire `src/test/resources` et des autres répertoires configurés comme contenant des ressources de test dans le répertoire cible pour les tests.

`compiler:testCompile`

Le goal `testCompile` du plugin `Compiler` est rattaché à la phase `test-compile`. Ce goal compile les tests unitaires du répertoire `src/test/java` et des autres répertoires configurés comme contenant du code source de test dans le répertoire cible pour les tests.

`surefire:test`

Le goal `test` du plugin `Surefire` est rattaché à la phase `test`. Ce goal exécute tous les tests et produit des fichiers contenant leurs résultats détaillés. Par défaut, le goal arrêtera le build en cas d'échec de l'un des tests.

jar:jar

Le goal `jar` du plugin `Jar` est lié à la phase `package`. Ce goal `package` le contenu du répertoire cible en un fichier JAR.

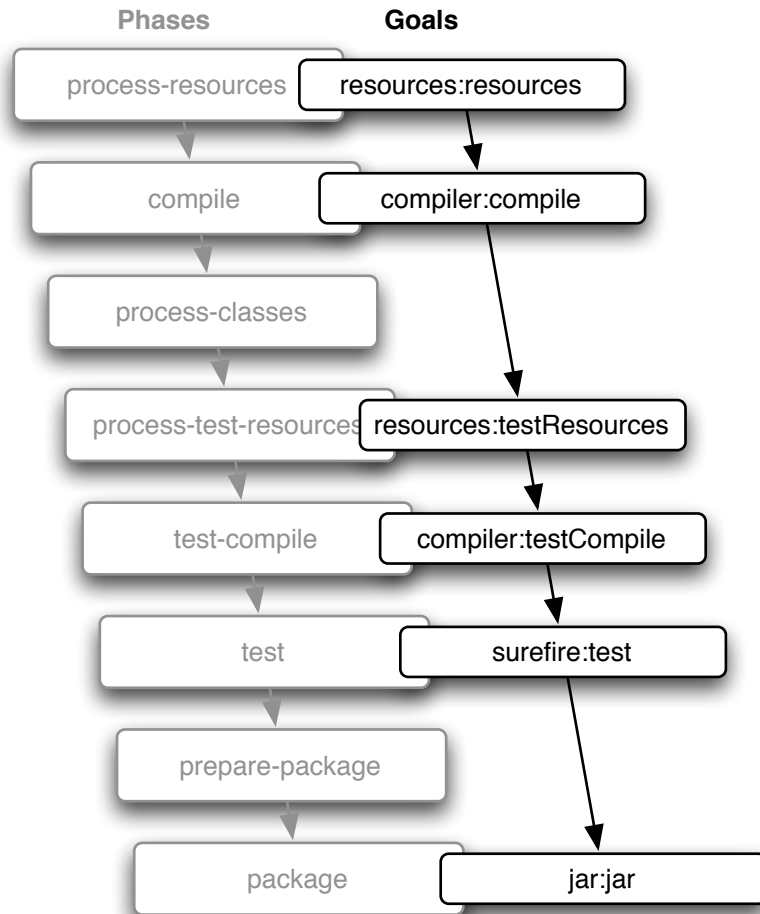


Figure 3.3. Les goals sont lancés à l'exécution de la phase à laquelle ils sont rattachés

Pour résumer, lorsque nous avons exécuté `mvn install`, Maven exécute toutes les phases jusqu'à la phase `install`, et pour cela il parcourt toutes les phases du cycle de vie, exécutant tous les goals liés à chacune de ces phases. Au lieu d'exécuter une des phases du cycle de vie de Maven, vous pourriez obtenir le même résultat en spécifiant une liste de goals de plugin de la manière suivante :

```
mvn resources:resources \  
    compiler:compile \  
    jar:jar
```

```
resources:testResources \  
compiler:testCompile \  
surefire:test \  
jar:jar \  
install:install
```

Il est beaucoup plus simple d'exécuter les phases du cycle de vie plutôt que de lister explicitement les goals à exécuter en ligne de commande, de plus, ce cycle de vie commun permet à chaque projet utilisant Maven d'adhérer à un ensemble de standards bien définis. C'est ce cycle de vie qui permet à un développeur de passer d'un projet à l'autre sans connaître tous les détails du build de chacun d'entre eux. Si vous savez construire un projet Maven, vous savez tous les construire.

3.5.3. Les coordonnées Maven

Le plugin Archetype a créé un projet avec un fichier `pom.xml`. Ceci est le Project Object Model (POM), une description déclarative d'un projet. Quand Maven exécute un goal, celui-ci a accès aux informations définies dans le POM du projet. Lorsque le goal `jar:jar` veut créer un fichier JAR, il va chercher dans le POM le nom du fichier JAR. Quand le goal `compiler:compile` compile le source code Java en bytecode, il va regarder dans le POM pour voir si on a précisé des options de compilation. Les goals s'exécutent dans le contexte d'un POM. Les goals sont les actions que nous voulons appliquer à un projet et un projet est défini par un POM. Le POM donne son nom au projet, il fournit un ensemble d'identifiants uniques (les coordonnées) du projet et précise les relations entre ce projet et d'autres qu'il s'agisse de dépendances, de parents ou de prérequis. Un POM permet aussi de personnaliser le comportement d'un plugin, et de fournir des informations sur la communauté et les développeurs du projet.

Les coordonnées Maven définissent un ensemble d'identifiants qui permet d'identifier de manière unique un projet, une dépendance, ou un plugin dans un POM Maven. Regardez le POM qui va suivre.

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>mavenbook</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>Maven Quick Start Archetype</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>

```

coordinates

Figure 3.4. Coordonnées d'un projet Maven

Nous avons mis en évidence les coordonnées Maven de ce projet : `groupId`, `artifactId`, `version` et `packaging`. Ces identifiants combinés forment ce qu'on appelle les coordonnées d'un projet². Comme dans tout système à coordonnées, les coordonnées Maven permettent de définir un point spécifique dans "l'espace". Lorsqu'un projet est relié à un autre en tant que dépendance, plugin ou comme projet parent, Maven l'identifie via ses coordonnées. Les coordonnées Maven sont souvent écrites en utilisant les deux points comme séparateur selon le format suivant : `groupId:artifactId:packaging:version`. Dans le fichier `pom.xml` ci-dessus, les coordonnées de notre projet sont les suivantes : `mavenbook:my-app:jar:1.0-SNAPSHOT`.

`groupId`

Le groupe, l'entreprise, l'équipe, l'organisation, le projet ou autre groupe. La convention pour les identifiants du groupe est qu'ils commencent par le nom de domaine inversé de l'organisation qui crée le projet. Les projets de Sonatype devraient avoir un `groupId` qui commence par `com.sonatype`, et les projets de l'Apache Software Foundation devraient avoir un `groupId` qui commence par `org.apache`.

²Il existe une cinquième coordonnée, plus rarement utilisée, appelée `classifier` dont nous parlerons plus tard dans le livre. Vous pouvez l'oublier pour l'instant.

`artifactId`

Identifiant unique sous le `groupId` qui représente un projet unique.

`version`

Version spécifique d'un projet. Les projets qui ont été livrés ont un identifiant de version fixe qui fait référence à une version bien spécifique du projet. Les projets en cours de développement peuvent utiliser un identifiant de version qui indique que cette version n'est pas stable : `SNAPSHOT`.

Le format de packaging est aussi un composant important des coordonnées Maven, mais il ne fait pas partie des identifiants uniques d'un projet. Le triplet `groupId:artifactId:version` d'un projet identifie de manière unique un projet ; vous ne pouvez pas avoir un projet avec le même triplet `groupId`, `artifactId`, et `version`.

`packaging`

Le type du projet, `jar` par défaut, décrit le résultat packagé produit par le projet. Un projet avec un packaging `jar` produit une archive JAR ; un projet avec un packaging `war` produit, quant à lui, une application web.

Ces quatre éléments sont la clef pour trouver et utiliser un projet particulier dans le vaste monde des projets "Mavenisés". Les dépôts Maven (publics, privés, et locaux) sont organisés autour de ces identifiants. Lorsque ce projet est installé dans le dépôt Maven local, il devient immédiatement disponible pour tout autre projet qui voudrait l'utiliser. Tout ce que vous avez à faire est de l'ajouter comme dépendance à un autre projet en utilisant les coordonnées Maven uniques propres à l'artefact.

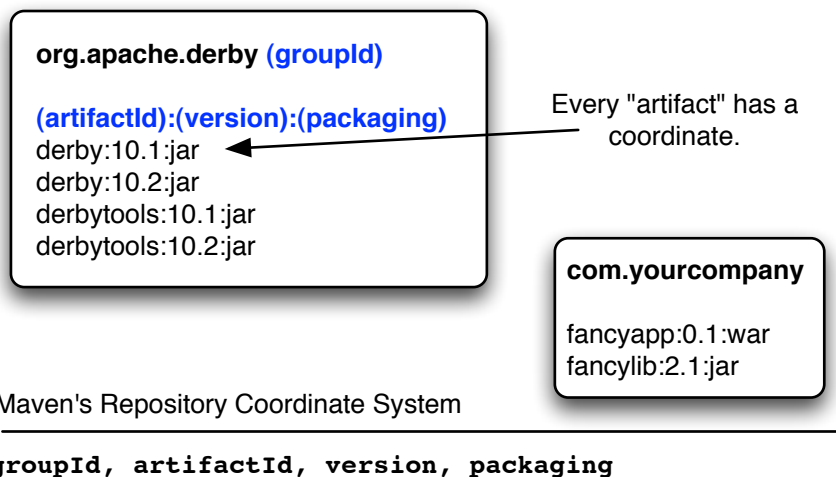


Figure 3.5. L'Espace des projets Maven est un système à coordonnées

3.5.4. Les dépôts Maven

Lors de sa première exécution, vous allez vous apercevoir que Maven télécharge un grand nombre de fichiers depuis un dépôt Maven distant. Si vous exécutez Maven pour la première fois pour le projet simple, la première chose qu'il va faire est de télécharger la dernière version du plugin Ressources lorsqu'il va déclencher le goal `resources:resource`. Avec Maven, artefacts et plugins sont téléchargés depuis des dépôts distants au moment où on en a besoin. Une des raisons pour lesquelles Maven est si léger à télécharger (1.5 MiB) est que cette version initiale ne vient qu'avec très peu de plugins. Maven est livré avec le strict minimum et récupère de dépôts distants ce dont il a besoin quand il en a besoin. Maven est livré avec l'emplacement d'un dépôt distant par défaut (<http://repo1.maven.org/maven2>) qu'il utilise pour télécharger les principaux plugins et les dépendances Maven.

Souvent vous aurez des projets qui dépendent de bibliothèques propriétaires ou qui ne sont pas disponibles publiquement. Dans ce cas vous devrez installer votre propre dépôt au sein du réseau de votre organisation, ou télécharger et installer ces dépendances manuellement. Les dépôts distants par défaut peuvent être remplacés ou complétés par des références à des dépôts Maven personnalisés gérés par votre organisation. Il existe de nombreux produits sur le marché pour permettre à des organisations de gérer et maintenir des miroirs des dépôts Maven publics.

Quelles sont les caractéristiques d'un dépôt qui en font un dépôt Maven ? Un dépôt Maven est un ensemble d'artefacts de projet rangés selon une structure de répertoires correspondant aux coordonnées Maven. Vous pouvez voir cette organisation en ouvrant un navigateur internet et en parcourant le dépôt central de Maven sur <http://repo1.maven.org/maven2/>. Vous verrez qu'un artefact ayant pour coordonnées `org.apache.commons:commons-email:1.1` se retrouve dans le répertoire `/org/apache/commons/commons-email/1.1/` dans le fichier qui s'appelle `commons-email-1.1.jar`. Le comportement standard pour un dépôt Maven est de ranger un artefact sous le répertoire racine du dépôt dans un répertoire respectant le format suivant.

```
/<groupId>/<artifactId>/<version>/<artifactId>-<version>.<packaging>
```

Maven télécharge artefacts et plugins depuis un dépôt distant et les enregistre dans le dépôt Maven local de votre machine. Une fois que Maven a téléchargé un artefact depuis un dépôt distant, il n'a plus besoin de le télécharger à nouveau. En effet, Maven cherchera toujours dans le dépôt local avant d'aller chercher ailleurs. Sur Windows XP, votre dépôt local se trouvera probablement dans `C:\Documents and Settings\USERNAME\.m2\repository` et sur Windows Vista, il se trouve dans `C:\Users\USERNAME\.m2\repository`. Sur les systèmes Unix, votre dépôt local Maven se trouve dans `~/.m2/repository`. Quand vous construisez un projet comme celui de la section précédente, la phase `install` exécute un goal qui installe les artefacts de votre projet dans votre dépôt Maven local.

Dans votre dépôt local, vous devriez voir l'artefact créé par notre projet simple. Si vous exécutez la commande `mvn install`, Maven installera l'artefact de notre projet dans votre dépôt local. Essayez-le.

```
$mvn install
...
[INFO] [install:install]
[INFO] Installing ../simple-1.0-SNAPSHOT.jar to \
```



```
~/m2/repository/com/sonatype/maven/simple/1.0-SNAPSHOT/ \
simple-1.0-SNAPSHOT.jar
```

Comme le montrent les traces de cette commande, Maven a installé le fichier JAR de notre projet dans notre dépôt Maven local. Maven utilise le dépôt local pour partager les dépendances entre projets locaux. Si vous développez deux projets — projet A et projet B — où le projet B dépend de l'artefact produit par le projet A, Maven récupérera l'artefact du projet A depuis votre dépôt local quand il construira le projet B. Les dépôts Maven sont à la fois un cache local des artefacts téléchargés depuis un dépôt distant et un mécanisme qui permet à vos projets de dépendre les uns des autres.

3.5.5. La gestion des dépendances de Maven

Dans l'exemple de ce chapitre, Maven a résolu les coordonnées de la dépendance JUnit — `junit:junit:3.8.1` — sous la forme d'un chemin dans un dépôt Maven `/junit/junit/3.8.1/junit-3.8.1.jar`. C'est cette capacité à trouver un artefact dans un dépôt à partir de ses coordonnées qui nous permet de définir les dépendances dans le POM du projet. Si vous examinez le fichier `pom.xml` du projet simple, vous verrez qu'il comporte une section `dependencies` pour traiter des dépendances et que cette section contient une seule dépendance — JUnit.

Un projet plus complexe contiendrait sûrement plusieurs dépendances, ou pourrait avoir des dépendances qui dépendent elles-mêmes d'autres artefacts. L'une des principales forces de Maven est sa gestion des dépendances transitives. Supposons que votre projet dépende d'une bibliothèque qui à son tour dépend de 5 ou 10 autres bibliothèques (comme Spring ou Hibernate par exemple). Au lieu d'avoir à trouver et lister explicitement toutes ces dépendances dans votre fichier `pom.xml`, vous pouvez ne déclarer que la dépendance à la bibliothèque qui vous intéresse, Maven se chargera d'ajouter ses dépendances à votre projet implicitement. Maven va aussi gérer les conflits de dépendances, vous fournira le moyen de modifier son comportement par défaut et d'exclure certaines dépendances transitives.

Regardons la dépendance téléchargée dans votre dépôt local lors de l'exemple précédent. Ouvrez dans votre dépôt local le répertoire `~/m2/repository/junit/junit/3.8.1/`. Si vous avez suivi les instructions de ce chapitre vous trouverez un fichier `junit-3.8.1.jar` et un fichier `junit-3.8.1.pom` avec quelques fichiers de checksum que Maven utilise pour vérifier l'intégrité des artefacts téléchargés. Remarquez que Maven n'a pas juste téléchargé le JAR de JUnit, mais aussi un fichier POM pour les dépendances de JUnit. C'est le téléchargement des fichiers POM en plus des artefacts qui est au cœur de la gestion des dépendances transitives par Maven.

Quand vous installez l'artefact produit par votre projet dans le dépôt local, vous noterez que Maven publie une version légèrement modifiée du fichier `pom.xml` de votre projet dans le répertoire contenant le fichier JAR. Ce fichier POM enregistré dans le dépôt fournit aux autres projets des informations sur ce projet, dont notamment ses dépendances. Si le Projet B dépend du Projet A, il dépend aussi des dépendances du Projet A. Quand Maven résout une dépendance à partir de ses coordonnées, il récupère, en plus de l'artefact, le POM, puis il analyse les dépendances de ce POM pour trouver les dépendances transitives. Ces dépendances transitives sont ensuite ajoutées à la liste des dépendances du projet.

Dans le monde de Maven, une dépendance n'est plus simplement un fichier JAR ; c'est un fichier POM qui à son tour peut déclarer de nouvelles dépendances. Ce sont ces dépendances de dépendances que l'on appelle dépendances transitives et cela est rendu possible par le fait que les dépôts Maven contiennent plus que du bytecode ; ils contiennent des métadonnées sur les artefacts.

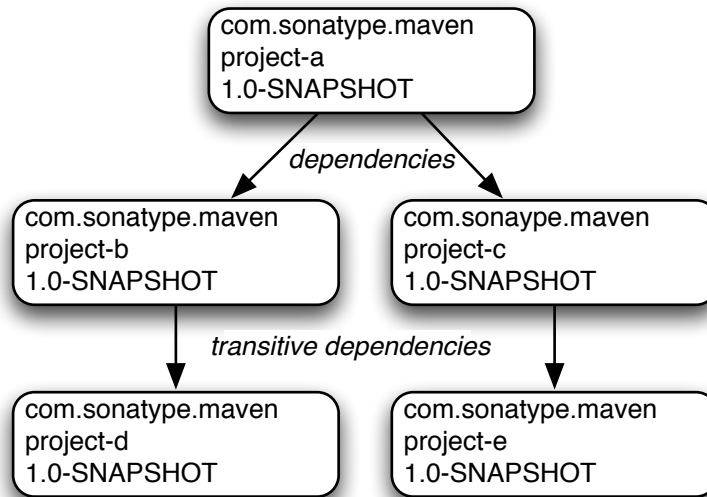


Figure 3.6. Résolution des dépendances transitives par Maven

Dans le schéma précédent, le projet A dépend des projets B et C. Le Projet B dépend du projet D et le projet C dépend du projet E. L'ensemble des dépendances directes et transitives du projet A serait donc les projets B, C, D et E, mais tout ce que le projet A doit faire, c'est de déclarer ses dépendances aux projets B et C. Les dépendances transitives sont pratiques lorsque votre projet dépend d'autres projets qui ont leurs propres dépendances (comme Hibernate, Apache Struts, ou Spring Framework). Maven vous permet d'exclure certaines dépendances transitives du classpath du projet.

Maven fournit enfin différentes portées pour les dépendances. Le fichier `pom.xml` du projet simple contient une unique dépendance — `junit:junit:jar:3.8.1` — ayant pour portée `test` indiquée dans la balise `scope`. Lorsqu'une dépendance Maven a une portée de type `test`, elle n'est pas disponible pour le goal `compile` du plugin `Compiler`. Cette dépendance sera ajoutée au classpath des goals `compiler:testCompile` et `surefire:test`.

Durant la création du JAR d'un projet, les dépendances ne sont pas intégrées à l'artefact produit ; elles ne sont utilisées que lors de la compilation. Par contre lorsque vous utilisez Maven pour produire un WAR ou un EAR, vous pouvez le configurer de manière à packager les dépendances avec l'artefact produit et vous pouvez même configurer Maven pour exclure certaines dépendances du fichier WAR par l'utilisation de la portée `provided`. La portée `provided` indique à Maven que la dépendance est nécessaire à la compilation, mais qu'elle ne doit pas être intégrée à l'artefact produit par le build. Cette

portée est donc très pratique lorsque vous développez une application web. Vous aurez besoin du jar des spécifications Servlet pour compiler, mais vous ne voulez pas inclure le JAR de l'API Servlet dans le répertoire `WEB-INF/lib`.

3.5.6. Rapports et production du site

Une fonction importante de Maven est sa capacité à produire de la documentation et des rapports. Dans le répertoire de votre projet simple, exécutez la commande suivante :

```
$mvn site
```

Cette commande exécute la phase `site` du cycle de vie. Contrairement au cycle de vie du build, qui gère la génération de code, la manipulation des ressources, la compilation, le packaging, etc., ce cycle de vie ne concerne que le traitement du contenu du site qui se trouve dans le répertoire `src/site` et la production de rapports. Une fois la commande exécutée, vous devriez voir un site web du projet dans le répertoire `target/site`. Ouvrez `target/site/index.html` et vous devriez voir le squelette du site du projet Simple. Ce squelette contient certains rapports sous le menu de navigation "Project Reports" à gauche, ainsi que des informations sur le projet, les dépendances et les développeurs dans le menu "Project Information". Le site web du projet simple est quasiment vide puisque le POM contient très peu d'informations, juste des coordonnées Maven, un nom, une URL et une unique dépendance de test.

Sur ce site, vous noterez que des rapports par défaut sont disponibles. Un rapport de test fournit les succès et les échecs de l'ensemble des tests unitaires du projet. Un autre rapport produit la javadoc pour l'API du projet. Maven fournit un ensemble de rapports configurables, comme le rapport Clover qui examine la couverture des tests unitaires, le rapport JXR qui produit des listings de code source en HTML avec des références croisées, utile pour les revues de code, le rapport PMD qui analyse le code source à la recherche de différentes erreurs de codage et le rapport JDepend qui analyse les dépendances entre packages dans un code source. Vous pouvez personnaliser les rapports du site en configurant quels sont ceux qui seront inclus dans le build via le fichier `pom.xml`.

3.6. En résumé

Dans ce chapitre, nous avons créé un projet simple que nous avons packagé sous la forme d'un fichier JAR, puis nous avons installé ce JAR dans le dépôt local de Maven pour qu'il soit utilisable par d'autres projets, enfin nous avons produit un site web contenant de la documentation. Nous avons fait tout cela sans écrire une seule ligne de code ni modifier un seul fichier de configuration. Nous nous sommes arrêtés en chemin pour développer les définitions de certains concepts au cœur de Maven. Dans le chapitre suivant, nous allons commencer à personnaliser et modifier le fichier `pom.xml` de notre projet pour lui ajouter des dépendances et configurer les tests unitaires.

Chapitre 4. Personnalisation d'un projet Maven

4.1. Introduction

Ce chapitre va développer ce que nous avons appris dans le Chapitre 3, Mon premier projet avec Maven. Nous allons créer un projet simple grâce au plugin Maven Archetype, lui ajouter quelques dépendances, du code source et le personnaliser pour l'adapter à notre besoin. À la fin de ce chapitre, vous devriez être capable d'utiliser Maven pour créer de vrais projets.

4.1.1. Télécharger l'exemple de ce chapitre

Nous allons développer un programme utile qui va interagir avec le service en ligne Yahoo! Météo. Bien que vous devriez être capable de suivre ce chapitre sans le code source d'exemple, nous vous recommandons d'en télécharger une copie et de l'utiliser comme référence. Le projet exemple de ce chapitre peut être téléchargé avec le code d'exemple de ce livre à l'adresse :

```
http://www.sonatype.com/books/maven-book/mavenbook-examples-0.9-SNAPSHOT-project.zip
```

Décompressez cette archive dans le répertoire de votre choix et allez dans le répertoire `custom/`. Vous y trouverez un répertoire nommé `simple-weather/` qui contient le projet Maven qui sera développé dans ce chapitre. Si vous souhaitez suivre avec le code d'exemple affiché dans un navigateur web, allez à <http://www.sonatype.com/book/examples-1.0> et cliquez sur le répertoire `custom/`.

4.2. Présentation du projet Simple Weather

Avant de commencer à personnaliser ce projet, prenons un peu de recul et discutons de ce projet Simple Weather. De quoi s'agit-il ? Il s'agit d'un exemple dont le but est de présenter certaines fonctions clés de Maven. C'est une application représentative de celles que vous pourriez avoir à réaliser. L'application Simple Weather est une application basique en ligne de commande qui prend en paramètre un code postal et va chercher des données depuis le flux RSS de Yahoo! Météo. Ensuite, elle parse la réponse et affiche le résultat sur la sortie standard.

Nous avons choisi cet exemple pour un certain nombre de raisons. Premièrement, il est simple à comprendre. Un utilisateur fournit une donnée en entrée via la ligne de commande, l'application prend ce code postal, en fait une requête à Yahoo! Météo, parse la réponse, et formate les données avant de les afficher à l'écran. Cet exemple est une simple fonction `main()` avec quelques classes ; il n'y a pas de framework d'entreprise à introduire ni à expliquer, juste un peu de parsing XML et quelques traces. Deuxièmement, il nous fournit une excellente excuse pour introduire des bibliothèques intéressantes telles que Velocity, Dom4J, et Log4J. Bien qu'il s'agisse d'un livre sur Maven, nous ne résistons pas à

l'idée d'introduire des outils utiles. Enfin, un unique chapitre suffit à introduire, développer, et déployer cet exemple.

4.2.1. Yahoo! Météo RSS

Avant de commencer à réaliser cette application, voici quelques informations concernant le flux RSS Yahoo! Météo. Pour commencer, ce service est accessible selon les termes suivants :

Les flux sont fournis gratuitement pour une utilisation personnelle, non-commerciale par des individus ou des organisations à but non lucratif. Nous vous demandons d'attribuer les résultats à Yahoo! Météo dans votre utilisation de ces flux.

En d'autres termes, si vous pensez intégrer ces flux dans votre site internet commercial, revoyez votre position — ce flux n'est utilisable qu'à titre personnel et non commercial. L'usage que nous allons en faire dans ce chapitre est personnel et à but d'apprentissage. Pour plus d'informations sur les termes de ce service, lisez la documentation de l'API Yahoo Weather! ici : <http://developer.yahoo.com/weather/>.

4.3. Créer le Projet Simple Weather

Tout d'abord, utilisons le plugin Maven Archetype pour créer le squelette basique du projet Simple Weather. Exécutez la commande suivante pour créer un nouveau projet :

```
$ mvn archetype:generate -DgroupId=org.sonatype.mavenbook.custom \
                        -DartifactId=simple-weather \
                        -DpackageName=org.sonatype.mavenbook \
                        -Dversion=1.0

[INFO] Preparing archetype:generate
...
[INFO] [archetype:generate {execution: default-cli}]
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart \
      (org.apache.maven.archetypes:maven-archetype-quickstart:1.0)
Choose archetype:
...
15: internal -> maven-archetype-quickstart ()
...
Choose a number: (...) 15: : 15
Confirm properties configuration:
groupId: org.sonatype.mavenbook.custom
artifactId: simple-weather
version: 1.0
package: org.sonatype.mavenbook.custom
Y: : Y
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.custom
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook.custom
[INFO] Parameter: package, Value: org.sonatype.mavenbook.custom
[INFO] Parameter: artifactId, Value: simple-weather
[INFO] Parameter: basedir, Value: /private/tmp
[INFO] Parameter: version, Value: 1.0
[INFO] BUILD SUCCESSFUL
```

Une fois que le plugin Maven Archetype a créé le projet, allez dans le répertoire `simple-weather` et lisez le fichier `pom.xml`. Vous devriez voir le document XML présenté dans Exemple 4.1, « POM Initial pour le projet `simple-weather` ».

Exemple 4.1. POM Initial pour le projet `simple-weather`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.custom</groupId>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>simple-weather2</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Puis, nous allons configurer le plugin Maven Compiler pour compiler du code Java 5. Pour cela, ajoutez la balise `build` au POM initial comme montré dans l'Exemple 4.2, « POM du projet `simple-weather` avec la configuration du compilateur ».

Exemple 4.2. POM du projet `simple-weather` avec la configuration du compilateur

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.custom</groupId>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>
  <version>1.0</version>
  <name>simple-weather2</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.1</version>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

```

    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>

```

Notez que nous avons passé le paramètre `version` au goal `archetype:generate`. Ce paramètre remplace la valeur par défaut `1.0-SNAPSHOT`. Dans ce projet, nous développons la version `1.0` du projet `simple-weather` comme vous pouvez le voir dans la balise `version` du fichier `pom.xml`.

4.4. Personnaliser les informations du projet

Avant de commencer à écrire du code, personnalisons un peu les informations du projet. Nous voulons ajouter des informations sur la licence du projet, l'organisation, et les quelques développeurs associés à ce projet. Il s'agit d'informations standards que vous attendriez de n'importe quel projet. L'Exemple 4.3, « Ajout des Informations Organisationnelles, Légales et la liste des développeurs au fichier `pom.xml` » montre le XML qui fournit les informations sur l'organisation, la licence et les développeurs.

Exemple 4.3. Ajout des Informations Organisationnelles, Légales et la liste des développeurs au fichier `pom.xml`

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  ...

  <name>simple-weather</name>
  <url>http://www.sonatype.com</url>

  <licenses>
    <license>
      <name>Apache 2</name>
      <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
      <distribution>repo</distribution>
      <comments>A business-friendly OSS license</comments>
    </license>
  </licenses>

  <organization>
    <name>Sonatype</name>

```



```

    <url>http://www.sonatype.com</url>
  </organization>

  <developers>
    <developer>
      <id>jason</id>
      <name>Jason Van Zyl</name>
      <email>jason@maven.org</email>
      <url>http://www.sonatype.com</url>
      <organization>Sonatype</organization>
      <organizationUrl>http://www.sonatype.com</organizationUrl>
      <roles>
        <role>developer</role>
      </roles>
      <timezone>-6</timezone>
    </developer>
  </developers>
  ...
</project>

```

Les ellipses dans l'Exemple 4.3, « Ajout des Informations Organisationnelles, Légales et la liste des développeurs au fichier `pom.xml` » sont un moyen pour que le listing soit plus court et plus lisible. Lorsque vous rencontrez dans un `pom.xml` des "..." juste après la balise ouvrante `project` et juste avant la balise fermante `project`, cela signifie que nous ne vous montrons pas le fichier `pom.xml` dans son intégralité. Dans notre cas, les balises `licenses`, `organization` et `developers` ont été ajoutées avant la balise `dependencies`.

4.5. Ajout de nouvelles dépendances

L'application Simple Weather va devoir accomplir les trois tâches suivantes : récupérer les données sous forme XML depuis Yahoo! Météo, parser le XML de Yahoo et enfin afficher proprement le résultat sur la sortie standard. Pour réaliser toutes ces tâches, nous devons introduire de nouvelles dépendances dans le `pom.xml` du projet. Pour parser la réponse XML de Yahoo!, nous allons utiliser Dom4J et Jaxen, pour formater le résultat de ce programme en ligne de commande nous utiliserons Velocity et nous allons aussi devoir ajouter une dépendance vers Log4J qui sera utilisé pour les logs. Après avoir ajouté ces dépendances, notre balise `dependencies` va ressembler à l'exemple suivant.

Exemple 4.4. Ajout de Dom4J, Jaxen, Velocity, et Log4J comme dépendances

```

<project>
  [...]
  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
    </dependency>
    <dependency>
      <groupId>dom4j</groupId>

```

```

    <artifactId>dom4j</artifactId>
    <version>1.6.1</version>
  </dependency>
  <dependency>
    <groupId>jaxen</groupId>
    <artifactId>jaxen</artifactId>
    <version>1.1.1</version>
  </dependency>
  <dependency>
    <groupId>velocity</groupId>
    <artifactId>velocity</artifactId>
    <version>1.5</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
[... ]
</project>

```

Comme vous pouvez le voir nous avons ajouté quatre balises de dépendance à celle existante qui référençait la dépendance de scope `test` sur JUnit. Si vous ajoutez ces dépendances au fichier `pom.xml` du projet et exécutez la commande `mvn install`, vous verrez Maven télécharger toutes ces dépendances et les dépendances transitives dans votre dépôt local Maven.

Comment avons-nous trouvé ces dépendances ? "Savons" nous à l'avance quelles seraient les bonnes valeurs pour le `groupId` et l'`artifactId` ? Certaines dépendances sont tellement utilisées (comme Log4J) que vous vous souviendrez du `groupId` et de l'`artifactId` à chaque fois que vous en aurez besoin. Velocity, Dom4J et Jaxen ont été localisés grâce au site internet très utile <http://repository.sonatype.org>. Il s'agit d'une version publique du Nexus de Sonatype. Elle fournit une interface permettant d'effectuer des recherches sur différents dépôts Maven, vous pouvez l'utiliser pour trouver vos dépendances. Vous pouvez tester ceci par vous même, ouvrez <http://repository.sonatype.org> et cherchez des bibliothèques très utilisées comme Hibernate ou le framework Spring. Quand vous recherchez un artefact sur ce site, il vous montrera un `artifactId` et toutes ses versions disponibles sur le dépôt central de Maven. Si vous cliquez sur une version spécifique, vous obtiendrez une page qui contient les balises de dépendance que vous aurez à copier/coller dans le `pom.xml` de votre projet. Si vous avez besoin de trouver une dépendance, vous devrez consulter repository.sonatype.org¹, car vous allez rapidement vous rendre compte que certaines bibliothèques ont plus d'un `groupId`. Avec cet outil, vous pourrez utiliser le dépôt Maven.

4.6. Code source de Simple Weather

L'application en ligne de commande Simple Weather se compose de cinq classes Java.

¹ <http://repository.sonatype.org>

```
org.sonatype.mavenbook.weather.Main
```

La classe `Main` contient une méthode `static main()`. Il s'agit du point d'entrée de votre système.

```
org.sonatype.mavenbook.weather.Weather
```

La classe `Weather` est un simple Java bean qui contient la position du rapport météo et quelques éléments clés, comme la température et l'humidité.

```
org.sonatype.mavenbook.weather.YahooRetriever
```

La classe `YahooRetriever` se connecte à Yahoo! Météo et renvoie un `InputStream` des données du flux RSS.

```
org.sonatype.mavenbook.weather.YahooParser
```

La classe `YahooParser` parse le XML de Yahoo! Météo, et renvoie un objet de type `Weather`.

```
org.sonatype.mavenbook.weather.WeatherFormatter
```

La classe `WeatherFormatter` prend un objet de type `Weather` en paramètre, construit un `VelocityContext` et applique le template `Velocity`.

Même si nous ne détaillerons pas le code ici, nous vous fournirons tout le code nécessaire pour vous permettre d'exécuter cet exemple. Nous supposons que la plupart des lecteurs ont téléchargé les exemples qui accompagnent ce livre, mais nous prenons en compte ceux qui veulent suivre l'exemple de ce chapitre étape par étape. Les sections qui suivent listent les classes du projet `simple-weather`. Chacune de ces classes devrait être mise dans le même package : `org.sonatype.mavenbook.weather`.

Supprimons les classes `App` et `AppTest` générées par `archetype:generate` et ajoutons notre nouveau package. Dans un projet Maven, tout le code source doit se trouver dans `src/main/java`. Depuis le répertoire racine de ce nouveau projet, exécutez les commandes suivantes :

```
$ cd src/test/java/org/sonatype/mavenbook
$ rm AppTest.java
$ cd ../../../../..
$ cd src/main/java/org/sonatype/mavenbook
$ rm App.java
$ mkdir weather
$ cd weather
```

Ainsi, nous avons créé un nouveau package appelé `org.sonatype.mavenbook.weather`. Maintenant, il nous faut mettre nos classes dans ce répertoire. A l'aide de votre éditeur de texte favori, créez un nouveau fichier appelé `Weather.java` avec le contenu de l'Exemple 4.5, « Modèle Objet de la classe `Weather` du projet Simple Weather ».

Exemple 4.5. Modèle Objet de la classe `Weather` du projet Simple Weather

```
package org.sonatype.mavenbook.weather;

public class Weather {
```

```

private String city;
private String region;
private String country;
private String condition;
private String temp;
private String chill;
private String humidity;

public Weather() {}

public String getCity() { return city; }
public void setCity(String city) { this.city = city; }

public String getRegion() { return region; }
public void setRegion(String region) { this.region = region; }

public String getCountry() { return country; }
public void setCountry(String country) { this.country = country; }

public String getCondition() { return condition; }
public void setCondition(String condition) { this.condition = condition; }

public String getTemp() { return temp; }
public void setTemp(String temp) { this.temp = temp; }

public String getChill() { return chill; }
public void setChill(String chill) { this.chill = chill; }

public String getHumidity() { return humidity; }
public void setHumidity(String humidity) { this.humidity = humidity; }
}

```

La classe `Weather` définit un simple bean utilisé pour stocker les données météo extraites du flux RSS de Yahoo! Météo. Ce flux fournit un grand nombre d'informations, depuis les heures de lever et de coucher du soleil, à la vitesse et la direction des vents. Pour garder cet exemple relativement simple, le modèle objet de `Weather` ne garde que la température, le facteur vent, l'humidité et une description textuelle des conditions actuelles.

Maintenant, dans le même répertoire, créez un fichier `Main.java`. Cette classe `Main` contiendra la méthode statique `main()` — point d'entrée de cet exemple.

Exemple 4.6. Classe `Main` du projet Simple Weather

```

package org.sonatype.mavenbook.weather;

import java.io.InputStream;
import org.apache.log4j.PropertyConfigurator;

public class Main {

    public static void main(String[] args) throws Exception {

```

```

// Configure Log4J
PropertyConfigurator.configure(Main.class.getClassLoader()
                               .getResource("log4j.properties"));
// Read the Zip Code from the Command-line (if none supplied, use 60202)
String zipcode = "60202";
try {
    zipcode = args[0];
} catch( Exception e ) {}
// Start the program
new Main(zipcode).start();
}

private String zip;

public Main(String zip) {
    this.zip = zip;
}

public void start() throws Exception {
    // Retrieve Data
    InputStream dataIn = new YahooRetriever().retrieve( zip );
    // Parse Data
    Weather weather = new YahooParser().parse( dataIn );
    // Format (Print) Data
    System.out.print( new WeatherFormatter().format( weather ) );
}
}

```

La méthode `main()` ci-dessus configure Log4J en récupérant une ressource depuis le classpath, puis essaie de lire un code postal depuis la ligne de commande. Si une exception survient lors de la lecture du code postal, le programme utilisera par défaut la valeur 60202 comme code postal. Une fois qu'il a un code postal, il instancie la classe `Main` et appelle la méthode `start()` de cette instance. La méthode `start()` appelle le `YahooRetriever` pour qu'il récupère le XML représentant les prévisions météo. Le `YahooRetriever` renvoie un `InputStream` qui est ensuite passé en paramètre du `YahooParser`. Le `YahooParser` parse le XML de Yahoo! Météo et renvoie un objet de type `Weather`. Pour terminer, le `WeatherFormatter` prend en paramètre un objet de type `Weather` et retourne une `String` formatée qui est affichée sur la sortie standard.

Créez un fichier `YahooRetriever.java` dans le même répertoire avec le contenu présenté dans l'Exemple 4.7, « La classe `YahooRetriever` du projet Simple Weather ».

Exemple 4.7. La classe `YahooRetriever` du projet Simple Weather

```

package org.sonatype.mavenbook.weather;

import java.io.InputStream;
import java.net.URL;
import java.net.URLConnection;

import org.apache.log4j.Logger;

```

```

public class YahooRetriever {

    private static Logger log = Logger.getLogger(YahooRetriever.class);

    public InputStream retrieve(int zipcode) throws Exception {
        log.info( "Retrieving Weather Data" );
        String url = "http://weather.yahooapis.com/forecastrss?p=" + zipcode;
        URLConnection conn = new URL(url).openConnection();
        return conn.getInputStream();
    }
}

```

Cette classe très simple ouvre une `URLConnection` sur l'API de Yahoo! Météo et renvoie un `InputStream`. Pour créer quelque chose capable de parser ce flux, nous allons devoir créer le fichier `YahooParser.java` dans le même répertoire.

Exemple 4.8. Classe `YahooParser` du projet Simple Weather

```

package org.sonatype.mavenbook.weather;

import java.io.InputStream;
import java.util.HashMap;
import java.util.Map;

import org.apache.log4j.Logger;
import org.dom4j.Document;
import org.dom4j.DocumentFactory;
import org.dom4j.io.SAXReader;

public class YahooParser {

    private static Logger log = Logger.getLogger(YahooParser.class);

    public Weather parse(InputStream inputStream) throws Exception {
        Weather weather = new Weather();

        log.info( "Creating XML Reader" );
        SAXReader xmlReader = createXmlReader();
        Document doc = xmlReader.read( inputStream );

        log.info( "Parsing XML Response" );
        weather.setCity( doc.valueOf("/rss/channel/y:location/@city" ) );
        weather.setRegion( doc.valueOf("/rss/channel/y:location/@region" ) );
        weather.setCountry( doc.valueOf("/rss/channel/y:location/@country" ) );
        weather.setCondition( doc.valueOf("/rss/channel/item/y:condition/@text" ) );
        weather.setTemp( doc.valueOf("/rss/channel/item/y:condition/@temp" ) );
        weather.setChill( doc.valueOf("/rss/channel/y:wind/@chill" ) );
        weather.setHumidity( doc.valueOf("/rss/channel/y:atmosphere/@humidity" ) );

        return weather;
    }
}

```

```

private SAXReader createXmlReader() {
    Map<String,String> uris = new HashMap<String,String>();
        uris.put( "y", "http://xml.weather.yahoo.com/ns/rss/1.0" );

    DocumentFactory factory = new DocumentFactory();
    factory.setXPathNamespaceURIs( uris );

    SAXReader xmlReader = new SAXReader();
    xmlReader.setDocumentFactory( factory );
    return xmlReader;
}
}

```

La classe `YahooParser` est la classe la plus complexe de cet exemple. Nous n'allons pas détailler l'utilisation de `Dom4J` ou de `Jaxen` ici, cependant cette classe mérite quelques explications. Dans la classe `YahooParser`, la méthode `parse()` prend un `InputStream` en paramètre et renvoie un objet de type `Weather`. Pour cela, elle doit parser un document XML avec `Dom4J`. Puisque nous nous intéressons aux éléments dans l'espace de nommage XML de Yahoo! Météo, nous avons besoin d'un `SAXReader` sensible aux espaces de nommage dans la méthode `createXmlReader()`. Une fois que nous avons créé ce parseur et que nous avons parsé le document, nous obtenons en retour un objet de type `org.dom4j.Document`. Au lieu de parcourir tous ses éléments fils, nous récupérons uniquement les informations qui nous intéressent en les sélectionnant avec des expressions `XPath`. `Dom4J` permet de parser le XML dans cet exemple et `Jaxen` apporte ses capacités `XPath`.

Une fois que nous avons créé un objet de type `Weather`, nous avons besoin de formater ce résultat pour qu'il soit lisible. Créez un fichier `WeatherFormatter.java` dans le même répertoire que les autres classes.

Exemple 4.9. Classe `WeatherFormatter` du projet `Simple Weather`

```

package org.sonatype.mavenbook.weather;

import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StringWriter;

import org.apache.log4j.Logger;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

public class WeatherFormatter {

    private static Logger log = Logger.getLogger(WeatherFormatter.class);

    public String format( Weather weather ) throws Exception {
        log.info( "Formatting Weather Data" );
        Reader reader =
            new InputStreamReader( getClass().getClassLoader()
                .getResourceAsStream( "output.vm" ) );
        VelocityContext context = new VelocityContext();
    }
}

```

```

    context.put("weather", weather );
    StringWriter writer = new StringWriter();
    Velocity.evaluate(context, writer, "", reader);
    return writer.toString();
}
}

```

La classe `WeatherFormatter` utilise Velocity pour appliquer un masque. La méthode `format()` prend un bean de type `Weather` en paramètre et renvoie une `String` formatée. La première chose que fait la méthode `format()` est de charger depuis le classpath un template Velocity appelé `output.vm`. Puis nous créons un `VelocityContext` qui contient un unique objet de type `Weather` appelé `weather`. Un objet de type `StringWriter` est créé pour contenir le résultat de l'application du masque. Le masque est évalué par un appel à `Velocity.evaluate()` et le résultat est renvoyé sous la forme d'une `String`.

Avant de pouvoir exécuter cet exemple, nous allons devoir ajouter des ressources à notre classpath.

4.7. Ajouter des Ressources

Ce projet nécessite deux ressources dans son classpath : la classe `Main` nécessite une ressource appelée `log4j.properties` pour configurer Log4J, et la classe `WeatherFormatter` utilise le template Velocity `output.vm`. Ces deux ressources doivent être dans le package par défaut (ou à la racine du classpath).

Pour ajouter ces ressources, nous aurons besoin d'un nouveau répertoire dans le répertoire racine du projet : `src/main/resources`. Puisque ce répertoire n'a pas été créé par la tâche `archetype:generate`, nous allons devoir le créer en exécutant les commandes suivantes à partir du répertoire racine du projet :

```

$ cd src/main
$ mkdir resources
$ cd resources

```

Une fois le répertoire `resources` créé, nous pouvons y ajouter ces deux ressources. Tout d'abord, ajoutez le fichier `log4j.properties` dans le répertoire `resources`, avec le contenu affiché dans l'Exemple 4.10, « Fichier de configuration Log4J du projet Simple Weather ».

Exemple 4.10. Fichier de configuration Log4J du projet Simple Weather

```

# Set root category priority to INFO and its only appender to CONSOLE.
log4j.rootCategory=INFO, CONSOLE

# CONSOLE is set to be a ConsoleAppender using a PatternLayout.
log4j.appender.CONSOLE=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE.Threshold=INFO
log4j.appender.CONSOLE.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE.layout.ConversionPattern=%-4r %-5p %c{1} %x - %m%n

```


Ce fichier `log4j.properties` configure Log4J pour qu'il affiche tous les messages de log sur la sortie standard en utilisant un `PatternLayout`. Enfin, nous avons besoin de créer `output.vm`, le template Velocity utilisé pour formater le résultat de notre programme en ligne de commande. Créez le fichier `output.vm` dans le répertoire `resources/`.

Exemple 4.11. Template Velocity pour le projet Simple Weather

```
*****
Current Weather Conditions for:
  ${weather.city}, ${weather.region}, ${weather.country}

Temperature: ${weather.temp}
  Condition: ${weather.condition}
  Humidity: ${weather.humidity}
  Wind Chill: ${weather.chill}
*****
```

Ce template contient un certain nombre de références à une variable appelée `weather`, il s'agit du bean `Weather` qui est passé à la classe `WeatherFormatter`. La syntaxe `${weather.temp}` est un raccourci pour récupérer et afficher la valeur de la propriété `temp` du bean. Maintenant que nous avons tout le code de projet au bon endroit, nous pouvons utiliser Maven pour exécuter cet exemple.

4.8. Exécuter le programme Simple Weather

Avec le plugin `Exec` du projet Codehaus Mojo², nous allons pouvoir exécuter ce programme. Pour lancer la classe `Main`, exécutez les commandes suivantes en ligne de commande depuis le répertoire racine du projet :

```
$ mvn install
$ mvn exec:java -Dexec.mainClass=org.sonatype.mavenbook.weather.Main
...
[INFO] [exec:java]
0   INFO  YahooRetriever - Retrieving Weather Data
134 INFO  YahooParser - Creating XML Reader
333 INFO  YahooParser - Parsing XML Response
420 INFO  WeatherFormatter - Formatting Weather Data
*****
Current Weather Conditions for:
  Evanston, IL, US

Temperature: 45
  Condition: Cloudy
  Humidity: 76
  Wind Chill: 38
*****
...
```

² <http://mojo.codehaus.org>

Nous n'avons pas passé de paramètre en ligne de commande à la classe `Main`, aussi c'est le code postal par défaut, 60202, qui a été utilisé. Pour passer un code postal en paramètre, nous devrions utiliser l'argument `-Dexec.args` et lui passer un code postal :

```
$ mvn exec:java -Dexec.mainClass=org.sonatype.mavenbook.weather.Main \
-Dexec.args="70112"
...
[INFO] [exec:java]
0   INFO  YahooRetriever - Retrieving Weather Data
134 INFO  YahooParser  - Creating XML Reader
333 INFO  YahooParser  - Parsing XML Response
420 INFO  WeatherFormatter - Formatting Weather Data
*****
Current Weather Conditions for:
  New Orleans, LA, US

Temperature: 82
  Condition: Fair
  Humidity: 71
  Wind Chill: 82
*****
[INFO] Finished at: Sun Aug 31 09:33:34 CDT 2008
...
```

Comme vous pouvez le voir, nous avons exécuté avec succès notre programme Simple Weather. Nous avons récupéré des données depuis Yahoo! Météo, données que nous avons parsées et formatées avec Velocity. Nous avons réalisé tout cela sans faire grand chose d'autre que d'écrire le code source du projet et un peu de configuration dans le `pom.xml`. Remarquez l'absence du "processus de build". Nous n'avons pas eu à définir comment le compilateur Java devait compiler notre code source en bytecode et nous n'avons pas eu besoin d'indiquer au système de build où se trouvait le bytecode pour qu'il puisse exécuter l'exemple. Tout ce que nous avons eu à faire pour ajouter quelques dépendances a été de trouver leurs coordonnées Maven.

4.8.1. Le plugin Maven Exec

Le plugin Exec vous permet d'exécuter des classes Java ou des scripts. Ce n'est pas un plugin du cœur de Maven, mais il est disponible depuis le projet Mojo³ hébergé chez Codehaus⁴. Pour une description complète du plugin Exec, exécutez la commande suivante :

```
$ mvn help:describe -Dplugin=exec -Dfull
```

Vous listerez ainsi l'ensemble des goals disponibles pour le plugin Maven Exec. Le plugin Help vous fournira aussi les paramètres valides pour le plugin Exec. Si vous désirez personnaliser le comportement du plugin Exec, vous devriez utiliser la documentation fournie par `help:describe` comme guide. Aussi utile que soit le plugin Exec, vous ne devriez pas l'utiliser pour exécuter votre application

³ <http://mojo.codehaus.org>

⁴ <http://www.codehaus.org>

autrement que pour la tester dans un environnement de développement. Pour une solution plus robuste, utilisez le plugin Maven Assembly qui est présenté dans la Section 4.13, « Construire une application packagée et exécutable en ligne de commande », plus loin dans ce chapitre.

4.8.2. Explorer les dépendances de votre projet

Le plugin Exec nous permet d'exécuter le programme Simple Weather sans avoir à charger les dépendances requises dans le classpath. Avec un autre système de build, nous aurions dû copier toutes les dépendances du programme dans une espèce de répertoire fourre-tout `lib/` qui aurait contenu les fichiers JAR. Ensuite, nous aurions dû écrire un script basique qui aurait ajouté notre bytecode et nos dépendances dans un classpath. C'est seulement après tout cela que nous aurions pu exécuter `java org.sonatype.mavenbook.weather.Main`. Le plugin Exec profite du fait que Maven sait déjà comment créer et gérer votre classpath et vos dépendances.

C'est très pratique, mais il est agréable de savoir de quoi se compose exactement le classpath de votre projet. Même si le projet ne dépend que de quelques bibliothèques dont Dom4J, Log4J, Jaxen et Velocity, il dépend aussi de dépendances transitives. Si vous avez besoin de connaître la composition de votre classpath, vous pouvez utiliser le plugin Maven Dependency pour afficher une liste des dépendances résolues. Afin d'afficher cette liste pour le projet Simple Weather, exécutez le goal `dependency:resolve` :

```
$ mvn dependency:resolve
...
[INFO] [dependency:resolve]
[INFO]
[INFO] The following files have been resolved:
[INFO]   com.ibm.icu:icu4j:jar:2.6.1 (scope = compile)
[INFO]   commons-collections:commons-collections:jar:3.1 (scope = compile)
[INFO]   commons-lang:commons-lang:jar:2.1 (scope = compile)
[INFO]   dom4j:dom4j:jar:1.6.1 (scope = compile)
[INFO]   jaxen:jaxen:jar:1.1.1 (scope = compile)
[INFO]   jdom:jdom:jar:1.0 (scope = compile)
[INFO]   junit:junit:jar:3.8.1 (scope = test)
[INFO]   log4j:log4j:jar:1.2.14 (scope = compile)
[INFO]   oro:oro:jar:2.0.8 (scope = compile)
[INFO]   velocity:velocity:jar:1.5 (scope = compile)
[INFO]   xalan:xalan:jar:2.6.0 (scope = compile)
[INFO]   xerces:xercesImpl:jar:2.6.2 (scope = compile)
[INFO]   xerces:xmlParserAPIs:jar:2.6.2 (scope = compile)
[INFO]   xml-apis:xml-apis:jar:1.0.b2 (scope = compile)
[INFO]   xom:xom:jar:1.0 (scope = compile)
```

Comme vous pouvez vous en rendre compte, notre projet possède un grand nombre de dépendances. Alors que nous n'avons inclus que quatre bibliothèques comme dépendances directes, nous avons en fait 15 dépendances. Dom4J dépend de Xerces et des XML Parser APIs, Jaxen dépend de la présence de Xalan dans le classpath. Le plugin Dependency va afficher la combinaison finale des dépendances utilisées pour compiler votre projet. Si vous voulez connaître l'arbre complet des dépendances de votre projet, vous pouvez exécuter le goal `dependency:tree`.

```

$ mvn dependency:tree
...
[INFO] [dependency:tree]
[INFO] org.sonatype.mavenbook.custom:simple-weather:jar:1.0
[INFO] +- log4j:log4j:jar:1.2.14:compile
[INFO] +- dom4j:dom4j:jar:1.6.1:compile
[INFO] | \- xml-apis:xml-apis:jar:1.0.b2:compile
[INFO] +- jaxen:jaxen:jar:1.1.1:compile
[INFO] | +- jdom:jdom:jar:1.0:compile
[INFO] | +- xerces:xercesImpl:jar:2.6.2:compile
[INFO] | \- xom:xom:jar:1.0:compile
[INFO] |     +- xerces:xmlParserAPIs:jar:2.6.2:compile
[INFO] |     +- xalan:xalan:jar:2.6.0:compile
[INFO] |     \- com.ibm.icu:icu4j:jar:2.6.1:compile
[INFO] +- velocity:velocity:jar:1.5:compile
[INFO] | +- commons-collections:commons-collections:jar:3.1:compile
[INFO] | +- commons-lang:commons-lang:jar:2.1:compile
[INFO] | \- oro:oro:jar:2.0.8:compile
[INFO] +- org.apache.commons:commons-io:jar:1.3.2:test
[INFO] \- junit:junit:jar:3.8.1:test
...

```

Si vous êtes aventurier dans l'âme ou si vous voulez voir le chemin complet des dépendances, avec les artefacts qui ont été rejetés à cause d'un conflit ou pour toute autre raison, exécutez Maven avec le flag debug.

```

$ mvn install -X
...
[DEBUG] org.sonatype.mavenbook.custom:simple-weather:jar:1.0 (selected for null)
[DEBUG] log4j:log4j:jar:1.2.14:compile (selected for compile)
[DEBUG] dom4j:dom4j:jar:1.6.1:compile (selected for compile)
[DEBUG] xml-apis:xml-apis:jar:1.0.b2:compile (selected for compile)
[DEBUG] jaxen:jaxen:jar:1.1.1:compile (selected for compile)
[DEBUG] jaxen:jaxen:jar:1.1-beta-6:compile (removed - )
[DEBUG] jaxen:jaxen:jar:1.0-FCS:compile (removed - )
[DEBUG] jdom:jdom:jar:1.0:compile (selected for compile)
[DEBUG] xml-apis:xml-apis:jar:1.3.02:compile (removed - nearer: 1.0.b2)
[DEBUG] xerces:xercesImpl:jar:2.6.2:compile (selected for compile)
[DEBUG] xom:xom:jar:1.0:compile (selected for compile)
[DEBUG] xerces:xmlParserAPIs:jar:2.6.2:compile (selected for compile)
[DEBUG] xalan:xalan:jar:2.6.0:compile (selected for compile)
[DEBUG] xml-apis:xml-apis:1.0.b2.
[DEBUG] com.ibm.icu:icu4j:jar:2.6.1:compile (selected for compile)
[DEBUG] velocity:velocity:jar:1.5:compile (selected for compile)
[DEBUG] commons-collections:commons-collections:jar:3.1:compile (selected for compile)
[DEBUG] commons-lang:commons-lang:jar:2.1:compile (selected for compile)
[DEBUG] oro:oro:jar:2.0.8:compile (selected for compile)
[DEBUG] junit:junit:jar:3.8.1:test (selected for test)

```

Dans la sortie de debug, vous voyez les rouages du mécanisme de gestion de dépendances. Ce que vous voyez ici c'est l'arbre des dépendances de ce projet. Maven affiche les coordonnées Maven complètes de toutes les dépendances de votre projet ainsi que les dépendances de vos dépendances (et les dépendances

des dépendances de vos dépendances). Vous pouvez voir que le projet `simple-weather` dépend de `jaxen`, qui dépend de `xom`, qui à son tour dépend de `icu4j`. Vous pouvez voir aussi que Maven construit ce graphe de dépendances en supprimant les doublons et en résolvant les conflits entre différentes versions. Si vous avez un problème avec certaines dépendances, il est souvent utile d'aller plus loin que la simple liste produite par `dependency:resolve`. Activer les traces en mode debug, vous permet de voir comment fonctionne le mécanisme de résolution des dépendances de Maven.

4.9. Ecrire des tests unitaires

Maven intègre le support des tests unitaires et l'exécution de ces tests fait partie intégrante de son cycle de vie par défaut. Ajoutons quelques tests unitaires à notre projet Simple Weather. Tout d'abord, créons le package `org.sonatype.mavenbook.weather` dans le répertoire `src/test/java` :

```
$ cd src/test/java
$ cd org/sonatype/mavenbook
$ mkdir -p weather/yahoo
$ cd weather/yahoo
```

Maintenant, nous allons créer deux tests unitaires. Le premier testera la classe `YahooParser`, et le second la classe `WeatherFormatter`. Dans le package `weather`, créez un fichier `YahooParserTest.java` avec le contenu présenté dans l'exemple qui suit.

Exemple 4.12. Test unitaire `YahooParserTest` du projet Simple Weather

```
package org.sonatype.mavenbook.weather.yahoo;

import java.io.InputStream;

import junit.framework.TestCase;

import org.sonatype.mavenbook.weather.Weather;
import org.sonatype.mavenbook.weather.YahooParser;

public class YahooParserTest extends TestCase {

    public YahooParserTest(String name) {
        super(name);
    }

    public void testParser() throws Exception {
        InputStream nyData =
            getClass().getClassLoader().getResourceAsStream("ny-weather.xml");
        Weather weather = new YahooParser().parse( nyData );
        assertEquals( "New York", weather.getCity() );
        assertEquals( "NY", weather.getRegion() );
        assertEquals( "US", weather.getCountry() );
        assertEquals( "39", weather.getTemp() );
        assertEquals( "Fair", weather.getCondition() );
        assertEquals( "39", weather.getChill() );
    }
}
```

```
    assertEquals( "67", weather.getHumidity() );
}
}
```

Cette classe `YahooParserTest` étend la classe `TestCase` de JUnit. Elle respecte le modèle habituel d'un test JUnit : un constructeur qui prend un unique paramètre de type `String` et qui appelle le constructeur de la classe mère et un ensemble de méthodes publiques dont les noms commencent par "test", et qui sont invoquées en tant que tests unitaires. Nous avons une seule méthode de test, `testParser`, qui teste le `YahooParser` en parsant un document XML connu. Le document XML de test s'appelle `ny-weather.xml` et est chargé depuis le classpath. Nous ajouterons les ressources pour les tests dans la Section 4.11, « Ajouter des ressources pour les tests unitaires ». Dans la structure de répertoires Maven, le fichier `ny-weather.xml` se trouve dans le répertoire qui contient les ressources de test — `${basedir}/src/test/resources` sous `org/sonatype/mavenbook/weather/yahoo/ny-weather.xml`. Ce fichier est lu sous la forme d'un `InputStream` et passé à la méthode `parse()` de `YahooParser`. La méthode `parse()` renvoie un objet de type `Weather`, qui est testé par une série d'appels à la méthode `assertEquals()`, une méthode définie par `TestCase`.

Dans le même répertoire, créez un fichier `WeatherFormatterTest.java`.

Exemple 4.13. Test unitaire `WeatherFormatterTest` du projet Simple Weather

```
package org.sonatype.mavenbook.weather.yahoo;

import java.io.InputStream;

import org.apache.commons.io.IOUtils;

import org.sonatype.mavenbook.weather.Weather;
import org.sonatype.mavenbook.weather.WeatherFormatter;
import org.sonatype.mavenbook.weather.YahooParser;

import junit.framework.TestCase;

public class WeatherFormatterTest extends TestCase {

    public WeatherFormatterTest(String name) {
        super(name);
    }

    public void testFormat() throws Exception {
        InputStream nyData =
            getClass().getClassLoader().getResourceAsStream("ny-weather.xml");
        Weather weather = new YahooParser().parse( nyData );
        String formattedResult = new WeatherFormatter().format( weather );
        InputStream expected =
            getClass().getClassLoader().getResourceAsStream("format-expected.dat");
        assertEquals( IOUtils.toString( expected ).trim(),
            formattedResult.trim() );
    }
}
```

```
}
```

Ce second test unitaire du projet teste la classe `WeatherFormatter`. Comme pour le test `YahooParserTest`, la classe `WeatherFormatterTest` étend elle aussi la classe `TestCase` de JUnit. L'unique méthode de test lit la même ressource depuis `${basedir}/src/test/resources` dans le répertoire `org/sonatype/mavenbook/weather/yahoo` via le classpath des tests unitaires. Nous ajouterons les ressources de test dans la Section 4.11, « Ajouter des ressources pour les tests unitaires ». `WeatherFormatterTest` passe ce fichier d'exemple au `YahooParser` qui renvoie un objet de type `Weather`. Celui-ci est ensuite formaté par `WeatherFormatter`. Etant donné que `WeatherFormatter` affiche une `String`, nous allons devoir la tester par rapport à une valeur attendue. La valeur attendue a été mise dans un fichier texte `format-expected.dat` qui se trouve dans le même répertoire que `ny-weather.xml`. Pour comparer le résultat du test à la valeur attendue, nous lisons cette valeur attendue dans un `InputStream` et nous utilisons la classe `IOUtils` de Commons IO pour convertir ce fichier en `String`. Cette `String` est ensuite comparée au résultat obtenu grâce à la méthode `assertEquals()`.

4.10. Ajouter des dépendances dans le scope `test`

Le test `WeatherFormatterTest` utilise une classe utilitaire fournie par Apache Commons IO — la classe `IOUtils`. `IOUtils` apporte un certain nombre de fonctions static pratiques qui éliminent la plus grande partie du travail de la manipulation des entrées/sorties. Dans le cas particulier de ce test unitaire, nous avons utilisé `IOUtils.toString()` pour copier la ressource `format-expected.dat` du classpath dans une `String`. Nous aurions pu faire cela sans Commons IO, mais nous aurions dû écrire six ou sept lignes de code supplémentaires avec différents objets de type `InputStreamReader` et `StringWriter`. La véritable raison pour laquelle nous avons utilisé Commons IO, c'est qu'elle nous fournit une excuse pour ajouter la dépendance Commons IO au scope `test`.

Une dépendance dans le scope `test` est une dépendance qui est disponible dans le classpath uniquement durant la compilation et l'exécution des tests. Si votre projet était un `war` ou un `ear`, une dépendance dans le scope `test` ne serait pas incluse dans l'archive résultant du projet. Pour ajouter une dépendance dans le scope `test`, ajoutez la balise `dependency` à la section `dependencies` de votre projet, comme cela est fait dans l'exemple qui suit :

Exemple 4.14. Ajout d'une dépendance dans le scope `test`

```
<project>
...
<dependencies>
...
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-io</artifactId>
  <version>1.3.2</version>
  <scope>test</scope>
</dependency>
...
</dependencies>
```

```
</project>
```

Après avoir ajouté cette dépendance au `pom.xml`, exécutez `mvn dependency:resolve` et vous devriez voir que `commons-io` fait maintenant partie des dépendances avec le scope `test`. Nous devons encore faire une chose avant de pouvoir exécuter les tests unitaires de ce projet. Nous devons créer dans le classpath les ressources dont ces tests unitaires ont besoin. Les différents scopes pour les dépendances sont expliqués en détail dans la Section 9.4.1, « Scope de dépendance ».

4.11. Ajouter des ressources pour les tests unitaires

Les tests unitaires ont accès à des ressources qui leurs sont spécifiques. Souvent vous aurez des fichiers contenant les résultats attendus ou des données de test dans le classpath des tests. Dans ce projet, le fichier XML `ny-weather.xml` est utilisé dans le test `YahooParserTest`. Le résultat attendu de la classe `WeatherFormatter` se trouve dans le fichier `format-expected.dat`.

Pour ajouter ces ressources de test, vous allez devoir créer le répertoire `src/test/resources`. C'est le répertoire par défaut dans lequel Maven recherche les ressources de test. Pour créer ce répertoire, exécutez les commandes suivantes depuis le répertoire racine de votre projet.

```
$ cd src/test
$ mkdir resources
$ cd resources
```

Une fois que vous avez créé le répertoire de ressources, créez un fichier `format-expected.dat` dans le répertoire `resources`.

Exemple 4.15. Résultat attendu du `WeatherFormatterTest` du projet Simple Weather

```
*****
Current Weather Conditions for:
  New York, NY, US

Temperature: 39
  Condition: Fair
  Humidity: 67
  Wind Chill: 39
*****
```

Ce fichier devrait vous rappeler quelque chose. C'est le même résultat que celui que nous avons obtenu lorsque nous avons exécuté le projet Simple Weather project avec le plugin Maven Exec. Le deuxième fichier que vous devrez ajouter au répertoire de ressources est `ny-weather.xml`.

Exemple 4.16. Données XML en entrée de `YahooParserTest` du projet Simple Weather

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<rss version="2.0" xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0"
```



```

xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
<channel>
<title>Yahoo! Weather - New York, NY</title>
<link>http://us.rd.yahoo.com/dailynews/rss/weather/New_York__NY/</link>
<description>Yahoo! Weather for New York, NY</description>
<language>en-us</language>
<lastBuildDate>Sat, 10 Nov 2007 8:51 pm EDT</lastBuildDate>

<ttl>60</ttl>
<yweather:location city="New York" region="NY" country="US" />
<yweather:units temperature="F" distance="mi" pressure="in" speed="mph" />
<yweather:wind chill="39" direction="0" speed="0" />
<yweather:atmosphere humidity="67" visibility="1609" pressure="30.18"
    rising="1" />
<yweather:astronomy sunrise="6:36 am" sunset="4:43 pm" />
<image>
<title>Yahoo! Weather</title>

<width>142</width>
<height>18</height>
<link>http://weather.yahoo.com/</link>
<url>http://l.yimg.com/us.yimg.com/i/us/nws/th/main_142b.gif</url>
</image>
<item>
<title>Conditions for New York, NY at 8:51 pm EDT</title>

<geo:lat>40.67</geo:lat>
<geo:long>-73.94</geo:long>
<link>http://us.rd.yahoo.com/dailynews/rss/weather/New_York__NY/</link>
<pubDate>Sat, 10 Nov 2007 8:51 pm EDT</pubDate>
<yweather:condition text="Fair" code="33" temp="39"
    date="Sat, 10 Nov 2007 8:51 pm EDT" />
<description><![CDATA[
<br />
<b>Current Conditions:</b><br />
Fair, 39 F<br /><br />
<b>Forecast:</b><br />
Sat - Partly Cloudy. High: 45 Low: 32<br />
Sun - Sunny. High: 50 Low: 38<br />
<br />
]]></description>
<yweather:forecast day="Sat" date="10 Nov 2007" low="32" high="45"
    text="Partly Cloudy" code="29" />

<yweather:forecast day="Sun" date="11 Nov 2007" low="38" high="50"
    text="Sunny" code="32" />
<guid isPermaLink="false">10002_2007_11_10_20_51_EDT</guid>
</item>
</channel>
</rss>

```

Ce fichier contient un document XML de test pour YahooParserTest. Nous utilisons ce fichier afin de tester le parseur YahooParser sans avoir à récupérer la réponse XML de Yahoo! Météo.

4.12. Exécuter les test unitaires

Maintenant que votre projet possède ses tests unitaires, exécutons-les. Vous n'avez rien à faire de spécial pour exécuter un test unitaire ; la phase `test` est une étape standard du cycle de vie de Maven. Vous lancez les tests avec Maven à chaque fois que vous exécutez `mvn package` ou `mvn install`. Si vous voulez lancer toutes les phases du cycle de vie jusqu'à la phase `test` incluse, exécutez `mvn test` :

```
$ mvn test
...
[INFO] [surefire:test]
[INFO] Surefire report directory: ~/examples/ch-custom/simple-weather/target/\
surefire-reports

-----
T E S T S
-----

Running org.sonatype.mavenbook.weather.yahoo.WeatherFormatterTest
0   INFO  YahooParser - Creating XML Reader
177 INFO  YahooParser - Parsing XML Response
239 INFO  WeatherFormatter - Formatting Weather Data
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.547 sec
Running org.sonatype.mavenbook.weather.yahoo.YahooParserTest
475 INFO  YahooParser - Creating XML Reader
483 INFO  YahooParser - Parsing XML Response
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.018 sec

Results :

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Exécuter `mvn test` depuis la ligne de commande, a demandé à Maven d'exécuter toutes les phases du cycle de vie jusqu'à la phase `test`. Le plugin Maven Surefire possède un goal `test` rattaché à la phase `test`. Ce goal `test` exécute tous les tests unitaires du projet qu'il peut trouver dans `src/test/java` avec un nom de fichier de la forme `**/Test*.java`, `**/*Test.java` ou `**/*TestCase.java`. Pour ce projet, vous pouvez vous rendre compte que le goal `test` du plugin Surefire a exécuté `WeatherFormatterTest` et `YahooParserTest`. Quand le plugin Maven Surefire lance les tests JUnit, il produit des rapports XML et texte dans le répertoire `${basedir}/target/surefire-reports`. Si vos tests sont en échec, il vous faudra regarder dans ce répertoire pour trouver les messages d'erreurs et les traces de vos tests unitaires.

4.12.1. Ignorer les tests en échec

Souvent vous vous retrouverez à développer sur un système dont certains tests unitaires sont en échec. Si vous pratiquez le développement piloté par les tests (Test Driven Development ou TDD), vous pourriez utiliser ces échecs comme indicateur du reste à faire sur votre projet. Si vous avez des tests en échec et que vous souhaitez construire votre projet tout de même, il va vous falloir indiquer à Maven de ne pas tenir compte des tests en échec. Quand Maven rencontre un échec lors du build, son comportement par défaut est d'arrêter le build. Pour continuer à construire un projet même lorsque le

plugin Surefire rencontre des tests en échec, vous allez devoir mettre la propriété de configuration de Surefire `testFailureIgnore` à `true`.

Exemple 4.17. Ignorer les tests unitaires en échec

```
<project>
[... ]
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <testFailureIgnore>true</testFailureIgnore>
      </configuration>
    </plugin>
  </plugins>
</build>
[... ]
</project>
```

La documentation du plugin (<http://maven.apache.org/plugins/maven-surefire-plugin/test-mojo.html>) indique que ce paramètre correspond à une expression :

Exemple 4.18. Expressions pour un paramètre de plugin

```
testFailureIgnore Set this to true to ignore a failure during \
testing. Its use is NOT RECOMMENDED, but quite \
convenient on occasion.

* Type: boolean
* Required: No
* Expression: ${maven.test.failure.ignore}
```

Cette expression peut être configurée depuis la ligne de commande en utilisant le paramètre `-D` :

```
$ mvn test -Dmaven.test.failure.ignore=true
```

4.12.2. Court-circuiter les tests unitaires

Il se peut que vous vouliez configurer Maven pour qu'il ne lance pas les tests unitaires. Peut-être que vous avez un énorme système pour lequel les tests unitaires prennent plusieurs minutes à s'exécuter et vous ne voulez pas attendre tout ce temps pour avoir le résultat de votre build. Vous pouvez aussi travailler sur un système avec un historique dont toute une série de tests unitaires sont en échec, et au lieu de corriger ces tests, vous voulez produire un JAR. Maven permet de court-circuiter les tests unitaires grâce au paramètre `skip` du plugin Surefire. Pour ne pas lancer les tests depuis la ligne de commande, ajoutez simplement la propriété `maven.test.skip` à n'importe quel goal :

```
$ mvn install -Dmaven.test.skip=true
...
[INFO] [compiler:testCompile]
[INFO] Not compiling test sources
[INFO] [surefire:test]
[INFO] Tests are skipped.
...
```

Quand le plugin Surefire atteint le goal `test`, il ne va pas lancer les tests unitaires si la propriété `maven.test.skip` est à `true`. Une autre façon de configurer Maven pour qu'il ne lance pas les tests unitaires est d'ajouter cette configuration au `pom.xml` de votre projet. Pour cela, vous devez ajouter une balise `plugin` à la configuration de votre `build`.

Exemple 4.19. Court-circuiter les tests unitaires

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-surefire-plugin</artifactId>
        <configuration>
          <skip>true</skip>
        </configuration>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

4.13. Construire une application packagée et exécutable en ligne de commande

Dans la section Section 4.8, « Exécuter le programme Simple Weather » un peu plus tôt dans ce chapitre, nous avons exécuté notre application avec le plugin Maven Exec. Même si ce plugin a permis d'exécuter le programme et d'obtenir des résultats, vous ne devriez pas voir Maven comme un moyen d'exécuter vos applications. Si vous distribuez cette application en ligne de commande, vous voudrez probablement distribuer un JAR ou une archive de type ZIP ou TAR GZIP. Cette section décrit un processus d'utilisation d'un descripteur d'assemblage prédéfini avec le plugin Maven Assembly pour produire un fichier JAR distribuable, qui contienne le bytecode du projet ainsi que toutes ses dépendances.

Le plugin Maven Assembly est un plugin qui vous permet de créer toutes les distributions que vous voulez pour vos applications. Vous pouvez utiliser le plugin Maven Assembly en définissant un descripteur d'assemblage personnalisé pour produire n'importe quel format distribuable de votre projet comme vous le voulez. Dans l'un des chapitres suivants, nous vous montrerons comment créer un

descripteur d'assemblage personnalisé pour produire une archive plus complexe pour l'application Simple Weather. Dans ce chapitre, nous nous contenterons d'utiliser le format prédéfini `jar-with-dependencies`. Pour configurer le plugin Maven Assembly, nous devons ajouter la configuration de plugin suivante à notre configuration de build existante dans le `pom.xml`.

Exemple 4.20. Configurer le descripteur Maven Assembly

```
<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <configuration>
          <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
          </descriptorRefs>
        </configuration>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Une fois que vous avez ajouté cette configuration, vous pouvez construire l'assemblage en exécutant le goal `assembly:assembly`. Dans les traces affichées à l'écran, le goal `assembly:assembly` est exécuté une fois que le build Maven atteint la phase `install` du cycle de vie :

```
$ mvn install assembly:assembly
...
[INFO] [jar:jar]
[INFO] Building jar:
~/examples/ch-custom/simple-weather/target/simple-weather-1.0.jar
[INFO] [assembly:assembly]
[INFO] Processing DependencySet (output=)
[INFO] Expanding: \
  .m2/repository/dom4j/dom4j/1.6.1/dom4j-1.6.1.jar into \
  /tmp/archived-file-set.1437961776.tmp
[INFO] Expanding: .m2/repository/commons-lang/commons-lang/2.1/\
  commons-lang-2.1.jar
  into /tmp/archived-file-set.305257225.tmp
... (Maven Expands all dependencies into a temporary directory) ...
[INFO] Building jar: \
  ~/examples/ch-custom/simple-weather/target/\
  simple-weather-1.0-jar-with-dependencies.jar
```

Une fois que notre application est assemblée dans `target/simple-weather-1.0-jar-with-dependencies.jar`, nous pouvons de nouveau lancer la classe `Main` depuis la ligne de commande. Pour lancer la classe `Main` de l'application Simple Weather, exécutez les commandes suivantes depuis le répertoire racine de votre projet.

```

$ cd target
$ java -cp simple-weather-1.0-jar-with-dependencies.jar \
      org.sonatype.mavenbook.weather.Main 10002
0   INFO  YahooRetriever - Retrieving Weather Data
221 INFO  YahooParser - Creating XML Reader
399 INFO  YahooParser - Parsing XML Response
474 INFO  WeatherFormatter - Formatting Weather Data
*****
Current Weather Conditions for:
  New York, NY, US

Temperature: 44
  Condition: Fair
  Humidity: 40
  Wind Chill: 40
*****

```

Le format `jar-with-dependencies` crée un unique fichier JAR qui contient tout le bytecode du projet `simple-weather` ainsi que celui des dépendances désassemblées. Ce format assez original produit un fichier JAR de 9 MiB qui contient environ 5.290 classes, mais il facilite la distribution des applications développées avec Maven. Plus tard dans ce livre, nous vous montrerons comment créer votre propre descripteur d'assemblage pour produire une distribution plus standard.

4.13.1. Rattacher le goal Assembly à la phase Package

Dans Maven 1, un build était personnalisé en chaînant à la suite les goals de plugin. Chaque goal de plugin avait des prérequis et définissait sa relation aux autres goals de plugin. Maven 2 a introduit le concept de cycle de vie. Maintenant, les goals de plugin sont liés à une série de phases d'un cycle de vie par défaut de Maven. Le cycle de vie pose les bases solides qui facilitent la gestion et la prédiction des goals de plugin qui seront exécutés lors d'un build. Dans Maven 1, les goals de plugin étaient reliés entre eux directement, avec Maven 2 les goals sont liés à des étapes communes du cycle de vie. Même s'il est certainement valide d'exécuter un goal depuis la ligne de commande comme nous l'avons déjà montré, configurer le plugin Assembly pour qu'il exécute le goal `assembly:assembly` durant une phase du cycle de vie de Maven est plus cohérent.

La configuration suivante configure le plugin Maven Assembly pour qu'il exécute le goal `attached` durant la phase `package` du cycle de vie par défaut de Maven. Le goal `attached` fait la même chose que le goal `assembly`. Pour rattacher le goal `assembly:attached` à la phase `package` nous utilisons la balise `executions` sous la balise `plugin` dans la section `build` du POM du projet.

Exemple 4.21. Configurer l'exécution du goal `attached` durant la phase Package du cycle de vie

```

<project>
  [...]
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>

```

```
<configuration>
  <descriptorRefs>
    <descriptorRef>jar-with-dependencies</descriptorRef>
  </descriptorRefs>
</configuration>
<executions>
  <execution>
    <id>simple-command</id>
    <phase>package</phase>
    <goals>
      <goal>attached</goal>
    </goals>
  </execution>
</executions>
</plugin>
</plugins>
</build>
[...]
```

Une fois votre POM configuré, tout ce qu'il vous reste à faire pour produire l'assemblage est d'exécuter la commande `mvn package`. La configuration de l'exécution va s'assurer que le goal `assembly:attached` est exécuté quand Maven atteint la phase `package` durant son parcours du cycle de vie.

Chapitre 5. Une simple application web

5.1. Introduction

Dans ce chapitre, nous allons créer une simple application web avec le plugin Maven Archetype. Nous allons exécuter cette application web dans un conteneur de Servlet nommé Jetty, ajouter quelques dépendances, écrire une simple Servlet, et générer une archive WAR. À la fin du chapitre, vous serez capable de démarrer rapidement le développement d'applications web en utilisant Maven.

5.1.1. Télécharger l'exemple de ce chapitre

L'exemple de ce chapitre est généré avec le plugin Maven Archetype. Même si vous devriez pouvoir suivre le développement de ce chapitre sans le code source de l'exemple, nous vous recommandons d'en télécharger une copie pour vous y référer. Le projet d'exemple de ce chapitre peut être téléchargé avec le code des autres exemples du livre sur :

```
http://www.sonatype.com/books/maven-book/mavenbook-examples-0.9-SNAPSHOT-project.zip
```

Décompressez cette archive dans un répertoire, puis ouvrez le répertoire `ch-simple-web/`. Vous y trouverez un répertoire nommé `simple-webapp/` qui contient le projet Maven développé dans ce chapitre.

5.2. Définition de l'application web simple-webapp

Nous avons volontairement gardé ce chapitre axé sur une bonne vieille application web (Plain-Old Web Applications, POWA) —une Servlet et une page JavaServer Pages (JSP). Nous n'allons pas vous expliquer comment développer une application Struts 2, Tapestry, Wicket, Java Server Faces (JSF), ou Waffle, et nous n'allons pas non plus intégrer un conteneur IoC (Inversion of Control) comme Plexus, Guice, ou Spring. Le but de ce chapitre est de vous montrer les possibilités fournies par Maven pour développer des applications web—ni plus, ni moins. Plus tard dans ce livre nous développerons deux applications web : l'une basée sur Hibernate, Velocity et Spring ; l'autre utilisant Plexus.

5.3. Création du projet web simple-web

Pour créer votre projet d'application web, exécutez la commande `mvn archetype:generate` en précisant un `artifactId` et un `groupId`. Paramétrez l'`archetypeArtifactId` à `maven-archetype-webapp`. L'exécution de cette commande va créer la structure des répertoires appropriée et le POM Maven :

```
$ mvn archetype:generate -DgroupId=org.sonatype.mavenbook.simpleweb \
```

```

-DartifactId=simple-webapp \
-DpackageName=org.sonatype.mavenbook \
-Dversion=1.0-SNAPSHOT
...
[INFO] [archetype:generate {execution: default-cli}]
Choose archetype:
...
15: internal -> maven-archetype-quickstart ()
16: internal -> maven-archetype-site-simple (A simple site generation project)
17: internal -> maven-archetype-site (A more complex site project)
18: internal -> maven-archetype-webapp (A simple Java web application)
...
Choose a number: (...) 15: : 18
Confirm properties configuration:
groupId: org.sonatype.mavenbook.simpleweb
artifactId: simple-webapp
version: 1.0-SNAPSHOT
package: org.sonatype.mavenbook.simpleweb
Y: : Y
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook.simpleweb
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook.simpleweb
[INFO] Parameter: package, Value: org.sonatype.mavenbook.simpleweb
[INFO] Parameter: artifactId, Value: simple-webapp
[INFO] Parameter: basedir, Value: /private/tmp
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
...
[INFO] BUILD SUCCESSFUL

```

Une fois que le plugin Maven Archetype a généré le projet, ouvrez le répertoire `simple-web` et regardez le fichier `pom.xml`. Vous devriez voir le document XML présenté dans l'exemple suivant :

Exemple 5.1. POM initial du projet simple-web

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.simpleweb</groupId>
  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple-webapp Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>

```

```
    <finalName>simple-webapp</finalName>
  </build>
</project>
```

Ensuite, vous allez devoir configurer le plugin Maven Compiler pour utiliser Java 5. Pour cela, ajoutez l'élément `plugins` au POM initial comme décrit dans l'Exemple 5.2, « POM du projet simple-web avec la configuration du compilateur ».

Exemple 5.2. POM du projet simple-web avec la configuration du compilateur

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.simpleweb</groupId>
  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple-webapp Maven Webapp</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
  </build>
</project>
```

Notez que l'élément `packaging` contient la valeur `war`. Ce type de `packaging` permet de configurer Maven pour produire une archive WAR. Un projet avec le `packaging war` va produire un fichier WAR dans le répertoire `target/`. Le nom par défaut de ce fichier est `${artifactId}-${version}.war`. Dans ce projet, le WAR par défaut serait donc `target/simple-webapp-1.0-SNAPSHOT.war`. Dans le projet `simple-webapp`, nous avons personnalisé le nom du fichier WAR généré en ajoutant l'élément `finalName` dans la configuration du `build`. En précisant `simple-webapp` comme valeur de `finalName`, la phase `package` produit le fichier WAR `target/simple-webapp.war`.

5.4. Configurer le plugin Jetty

Une fois que vous avez compilé, testé et packagé votre application, vous allez vouloir la déployer dans un conteneur de Servlet et tester la page `index.jsp` qui a été créée par le plugin Maven Archetype. Normalement, cela implique de télécharger un serveur comme Jetty ou Apache Tomcat, décompresser la distribution, copier votre fichier WAR dans un répertoire `webapps/` et ensuite démarrer le conteneur. Bien que vous puissiez procéder ainsi, ce n'est pas nécessaire. À la place, vous pouvez utiliser le plugin Maven Jetty pour exécuter votre application web avec Maven. Pour ce faire, vous allez devoir configurer le plugin Maven Jetty dans le fichier `pom.xml` de votre projet. Ajoutez l'élément `plugin` à votre configuration de build comme le montre l'exemple ci-dessous :

Exemple 5.3. Configurer le plugin Jetty

```
<project>
  [...]
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
      </plugin>
    </plugins>
  </build>
  [...]
</project>
```

Une fois que vous avez configuré le plugin Maven Jetty dans le `pom.xml` de votre projet, vous pouvez invoquer son goal `run` pour démarrer votre application web dans le conteneur de Servlet Jetty. Exécutez la commande `mvn jetty:run` à partir du répertoire du projet `simple-webapp/` de la manière suivante :

```
~/examples/ch-simple-web/simple-webapp $ mvn jetty:run
...
[INFO] [jetty:run]
[INFO] Configuring Jetty for project: simple-webapp Maven Webapp
[INFO] Webapp source directory = \
      ~/svnw/sonatype/examples/ch-simple-web/simple-webapp/src/main/webapp
[INFO] web.xml file = \
      ~/svnw/sonatype/examples/ch-simple-web/
simple-webapp/src/main/webapp/WEB-INF/web.xml
[INFO] Classes = ~/svnw/sonatype/examples/ch-simple-web/
simple-webapp/target/classes
2007-11-17 22:11:50.532::INFO: Logging to STDERR via org.mortbay.log.StdErrLog
[INFO] Context path = /simple-webapp
[INFO] Tmp directory = determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides = none
[INFO] Webapp directory = \
      ~/svnw/sonatype/examples/ch-simple-web/simple-webapp/src/main/webapp
```

```
[INFO] Starting jetty 6.1.6rc1 ...
2007-11-17 22:11:50.673::INFO: jetty-6.1.6rc1
2007-11-17 22:11:50.846::INFO: No Transaction manager found
2007-11-17 22:11:51.057::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```



Avertissement

Si vous utilisez le plugin Maven Jetty à partir d'un environnement Windows, vous devrez probablement déplacer votre dépôt local dans un répertoire qui ne contient pas d'espaces. Des lecteurs nous ont remonté des problèmes lors du démarrage de Jetty car leur dépôt local était stocké dans le répertoire "C:\Documents and Settings\". La solution pour régler ce problème est de déplacer votre dépôt local Maven dans un répertoire qui ne contient pas d'espaces et de redéfinir son emplacement dans le fichier `~/.m2/settings.xml` comme décrit dans la Section A.2.1, « Valeurs simples ».

Une fois que Maven a démarré le conteneur de servlet Jetty, ouvrez l'URL `http://localhost:8080/simple-webapp/` dans un navigateur web. La page `index.jsp` générée par l'Archetype est triviale ; elle contient un titre de second niveau avec le texte "Hello World!". Maven s'attend à ce que la racine du site web soit dans `src/main/webapp`. C'est dans ce répertoire que vous trouverez le fichier `index.jsp` utilisé dans l'Exemple 5.4, « Contenu de `src/main/webapp/index.jsp` ».

Exemple 5.4. Contenu de `src/main/webapp/index.jsp`

```
<html>
  <body>
    <h2>Hello World!</h2>
  </body>
</html>
```

Dans `src/main/webapp/WEB-INF`, nous allons trouver le plus petit descripteur d'application `web.xml` imaginable, présenté dans l'exemple suivant :

Exemple 5.5. Contenu de `src/main/webapp/WEB-INF/web.xml`

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
</web-app>
```

5.5. Ajouter une simple servlet

Une application web avec une seule page JSP et sans servlet configurée est très peu utile. Ajoutons une simple servlet à cette application et modifions les fichiers `pom.xml` et `web.xml` en conséquence.

Pour commencer, nous allons devoir créer un nouveau package dans `src/main/java` nommé `org.sonatype.mavenbook.web` :

```
$ mkdir -p src/main/java/org/sonatype/mavenbook/web
$ cd src/main/java/org/sonatype/mavenbook/web
```

Une fois que vous avez créé ce package, allez dans le répertoire `src/main/java/org/sonatype/mavenbook/web` et créez une classe `SimpleServlet` dans le fichier `SimpleServlet.java`, qui contient le code suivant :

Exemple 5.6. Classe `SimpleServlet`

```
package org.sonatype.mavenbook.web;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {

        PrintWriter out = response.getWriter();
        out.println( "SimpleServlet Executed" );
        out.flush();
        out.close();
    }
}
```

Notre classe `SimpleServlet` est très simple : une servlet qui écrit un message simple sur le `Writer` de la réponse. Pour ajouter cette servlet à votre application web et l'associer à l'URL d'une requête, ajoutez les balises `servlet` et `servlet-mapping` de l'Exemple 5.7, « Configurer la Servlet » dans le fichier `web.xml` de votre projet. Le fichier `web.xml` se trouve dans `src/main/webapp/WEB-INF`.

Exemple 5.7. Configurer la Servlet

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>simple</servlet-name>
    <servlet-class>org.sonatype.mavenbook.web.SimpleServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>simple</servlet-name>
    <url-pattern>/simple</url-pattern>
```

```
</servlet-mapping>
</web-app>
```

Tout est en place pour tester la servlet ; la classe est bien dans `src/main/java` et le fichier `web.xml` a été mis à jour. Avant que vous ne lanciez le plugin Jetty, compilez votre projet en exécutant **mvn compile** :

```
~/examples/ch-simple-web/simple-webapp $ mvn compile
...
[INFO] [compiler:compile]
[INFO] Compiling 1 source file to \
~/examples/ch-simple-web/simple-webapp/target/classes
[INFO] -----
[ERROR] BUILD FAILURE
[INFO] -----
[INFO] Compilation failure

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[4,0] \
package javax.servlet does not exist

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[5,0] \
package javax.servlet.http does not exist

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[7,35] \
cannot find symbol
symbol: class HttpServlet
public class SimpleServlet extends HttpServlet {

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[8,22] \
cannot find symbol
symbol : class HttpServletRequest
location: class org.sonatype.mavenbook.web.SimpleServlet

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[9,22] \
cannot find symbol
symbol : class HttpServletResponse
location: class org.sonatype.mavenbook.web.SimpleServlet

/src/main/java/org/sonatype/mavenbook/web/SimpleServlet.java:[10,15] \
cannot find symbol
symbol : class ServletException
location: class org.sonatype.mavenbook.web.SimpleServlet
```

La compilation échoue car votre projet Maven n'a pas la dépendance vers l'API Servlet. Dans la section suivante, nous allons ajouter l'API Servlet au POM de notre projet.

5.6. Ajouter les dépendances J2EE

Pour écrire une servlet, nous avons besoin d'ajouter la dépendance à l'API Servlet dans notre projet. Pour ajouter l'API Servlet en tant que dépendance dans le POM du projet, ajoutez l'élément `dependency` comme présenté dans l'exemple :

Exemple 5.8. Ajouter la dépendance vers la spécification Servlet 2.4

```
<project>
  [...]
  <dependencies>
    [...]
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  [...]
</project>
```

Il est important de noter que nous utilisons le scope `provided` pour cette dépendance. Cela permet de préciser à Maven que le jar est "provided" (fourni) par le conteneur et n'a ainsi pas besoin d'être inclus dans le war. Si vous souhaitez écrire votre propre tag JSP pour cette simple application web, vous devrez ajouter une dépendance sur la spécification JSP 2.0. Utilisez la configuration présentée dans l'exemple :

Exemple 5.9. Ajouter la dépendance aux spécifications JSP 2.0

```
<project>
  [...]
  <dependencies>
    [...]
    <dependency>
      <groupId>javax.servlet.jsp</groupId>
      <artifactId>jsp-api</artifactId>
      <version>2.0</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
  [...]
</project>
```

Une fois que vous avez ajouté la dépendance sur l'API Servlet, exécutez la commande `mvn clean install` suivie de la commande `mvn jetty:run`.

```
[tobrien@t1 simple-webapp]$ mvn clean install
...
[tobrien@t1 simple-webapp]$ mvn jetty:run
[INFO] [jetty:run]
...
2007-12-14 16:18:31.305::INFO: jetty-6.1.6rc1
2007-12-14 16:18:31.453::INFO: No Transaction manager found
2007-12-14 16:18:32.745::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

À ce stade, vous devriez pouvoir récupérer le résultat de la `SimpleServlet`. Depuis la ligne de commande, vous pouvez utiliser `curl` pour afficher la réponse de cette servlet sur la sortie standard :


```
~/examples/ch-simple-web $ curl http://localhost:8080/simple-webapp/simple
SimpleServlet Executed
```

5.7. Conclusion

Après avoir lu ce chapitre, vous devriez être en mesure de démarrer rapidement une application web. Ce chapitre ne s'est pas étendu sur les millions de façons différentes de créer une application web complète, d'autres chapitres fourniront plus de détails sur des projets qui mettent en œuvre quelques-uns des frameworks web et autres technologies parmi les plus populaires.

Chapitre 6. Un projet multimodule

6.1. Introduction

Dans ce chapitre, nous allons créer un projet multimodule qui combine les exemples des deux chapitres précédents. Le code du projet `simple-weather` développé dans le Chapitre 4, Personnalisation d'un projet Maven sera combiné avec le projet `simple-webapp` défini dans le Chapitre 5, Une simple application web pour créer une application web qui récupère et affiche des données météorologiques sur une page web. À la fin de ce chapitre, vous serez capable d'utiliser Maven pour développer des projets multimodules complexes.

6.1.1. Télécharger l'exemple de ce chapitre

Le projet multimodule de cet exemple consiste à développer une version modifiée des projets des chapitres 4 et 5 sans utiliser le plugin Maven Archetype. Nous vous recommandons fortement de télécharger une copie du code exemple pour l'utiliser comme référence supplémentaire lors de la lecture de ce chapitre. Le projet exemple de ce chapitre est téléchargeable avec le reste des exemples du livre à l'adresse suivante :

```
http://www.sonatype.com/books/maven-book/mavenbook-examples-0.9-SNAPSHOT-project.zip
```

Décompressez cette archive dans n'importe quel répertoire et ouvrez le répertoire `ch-multi/`. Celui-ci inclut un répertoire `simple-parent/` qui contient le projet Maven multimodule développé dans ce chapitre. Dans ce dernier, vous trouverez un fichier `pom.xml` et deux sous-modules, `simple-weather/` et `simple-webapp/`.

6.2. Le projet parent

Un projet multimodule est défini par un POM parent référençant un ou plusieurs sous-modules. Dans le répertoire `simple-parent/`, vous trouverez le POM parent (également appelé POM de plus haut niveau) dans le fichier `simple-parent/pom.xml`. Consultez l'Exemple 6.1, « POM du projet simple-parent ».

Exemple 6.1. POM du projet simple-parent

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.sonatype.mavenbook.multi</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
```

```

<version>1.0</version>
<name>Multi Chapter Simple Parent Project</name>

<modules>
  <module>simple-weather</module>
  <module>simple-webapp</module>
</modules>

<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

Remarquez que le parent définit un ensemble de coordonnées Maven : le `groupId` avec pour valeur `org.sonatype.mavenbook.multi`, l'`artifactId` ayant pour valeur `simple-parent` et la version `1.0`. Le projet parent ne crée pas de JAR ni de WAR comme c'était le cas de nos projets précédents. À la place, il s'agit simplement d'un POM qui fait référence à d'autres projets Maven. Le packaging approprié pour un projet du type `simple-parent` qui fournit un simple Modèle Objet de Projet (Project Object Model) est `pom`. La section suivante du fichier `pom.xml` liste les sous-modules du projet. Ces modules sont déclarés via une balise XML `modules` et chacune des balises `module` qu'elle contient correspond à un des sous-répertoires du dossier `simple-parent/`. Maven sait qu'il faut chercher pour chacun de ces répertoires un fichier `pom.xml`. Ainsi, il ajoutera ces sous-modules à la liste des projets Maven inclus dans le build.

Enfin, nous définissons quelques préférences qui seront héritées dans tous les sous-modules. La configuration du build du projet `simple-parent` définit la JVM Java 5 comme cible de compilation. Comme le plugin Maven Compiler est attaché au cycle de vie par défaut, il est possible d'utiliser la section `pluginManagement` du fichier `pom.xml` pour effectuer cela. Nous discuterons de la balise `pluginManagement` plus en détail dans les chapitres suivants, mais la configuration des plugins par

défaut et l'ajout de nouveaux plugins sont ainsi beaucoup plus faciles à voir lorsqu'ils sont dissociés de cette manière. La balise `dependencies` ajoute JUnit 3.8.1 comme une dépendance globale. La configuration du build comme les dépendances sont héritées dans tous les sous-modules. L'utilisation de l'héritage de POM vous permet d'ajouter des dépendances communes pour les dépendances universelles comme JUnit ou Log4J.

6.3. Le module simple-weather

Commençons par regarder le sous-module `simple-weather`. Ce module contient toutes les classes qui s'occupent des interactions et du parsing du flux RSS de Yahoo! Météo.

Exemple 6.2. POM du module simple-weather

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.multi</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-weather</artifactId>
  <packaging>jar</packaging>

  <name>Multi Chapter Simple Weather API</name>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-surefire-plugin</artifactId>
          <configuration>
            <testFailureIgnore>>true</testFailureIgnore>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>

  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.14</version>
    </dependency>
    <dependency>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
```

```

    <version>1.6.1</version>
  </dependency>
  <dependency>
    <groupId>jaxen</groupId>
    <artifactId>jaxen</artifactId>
    <version>1.1.1</version>
  </dependency>
  <dependency>
    <groupId>velocity</groupId>
    <artifactId>velocity</artifactId>
    <version>1.5</version>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.3.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

Le fichier pom.xml du module `simple-weather` fait référence à un POM parent via un triplet de coordonnées Maven. Le POM parent du module `simple-weather` est identifié par son `groupId` avec pour valeur `org.sonatype.mavenbook.multi`, par l'artifactId `simple-parent` et par la version `1.0`. Consultez l'Exemple 6.3, « La classe `WeatherService` ».

Exemple 6.3. La classe `WeatherService`

```

package org.sonatype.mavenbook.weather;

import java.io.InputStream;

public class WeatherService {

    public WeatherService() {}

    public String retrieveForecast( String zip ) throws Exception {
        // Retrieve Data
        InputStream dataIn = new YahooRetriever().retrieve( zip );

        // Parse Data
        Weather weather = new YahooParser().parse( dataIn );

        // Format (Print) Data
        return new WeatherFormatter().format( weather );
    }
}

```

Dans le dossier `src/main/java/org/sonatype/mavenbook/weather` se trouve la classe `WeatherService`. Celle-ci fait référence à trois objets définis dans le Chapitre 4, Personnalisation d'un

projet Maven. Dans l'exemple de ce chapitre, nous créons un projet séparé qui contient les services utilisés par le projet de l'application web. C'est un modèle commun dans le développement d'entreprise Java, une application complexe se compose souvent de plusieurs applications web. Vous pourriez donc avoir une application d'entreprise composée de plusieurs applications web et de plusieurs applications en ligne de commande. Souvent, vous voudrez extraire certains éléments communs dans un service externalisé pour pouvoir les réutiliser sur plusieurs projets. C'est donc dans cette optique que nous créons la classe `WeatherService`. Ainsi, nous verrons comment le projet `simple-webapp` fait référence à un service défini dans module séparé : `simple-weather`.

La méthode `retrieveForecast()` prend en paramètre une chaîne de caractères contenant un code postal. Celle-ci est passé à la méthode `retrieve()` de la classe `YahooRetriever` qui retournera un flux XML provenant de Yahoo! Météo. Ce même XML est ensuite passé à la méthode `parse()` de la classe `YahooParser` qui analyse ce flux et le retourne sous la forme d'un objet `Weather`. Ce dernier est ensuite formaté par la classe `WeatherFormatter` en une chaîne de caractères présentable.

6.4. Le module simple-web

Le module `simple-webapp` est le second sous-module référençant le projet `simple-parent`. Cette application web dépend du sous-module `simple-weather` car elle contient des servlets mettant en forme les résultats de la recherche sur Yahoo! Météo.

Exemple 6.4. POM du module `simple-webapp`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.multi</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <name>simple-webapp Maven Webapp</name>
  <dependencies>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.multi</groupId>
      <artifactId>simple-weather</artifactId>
      <version>1.0</version>
```

```

    </dependency>
</dependencies>
<build>
  <finalName>simple-webapp</finalName>
  <plugins>
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty-plugin</artifactId>
    </plugin>
  </plugins>
</build>
</project>

```

Ce module `simple-webapp` contient une servlet très simple qui se contente de récupérer un code postal à partir d'une requête HTTP, d'appeler la classe `WeatherService` que nous venons de voir précédemment dans l'Exemple 6.3, « La classe `WeatherService` » et d'écrire le résultat dans le `Writer` de la réponse.

Exemple 6.5. Servlet `WeatherServlet` du projet `simple-webapp`

```

package org.sonatype.mavenbook.web;

import org.sonatype.mavenbook.weather.WeatherService;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class WeatherServlet extends HttpServlet {
    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws ServletException, IOException {
        String zip = request.getParameter("zip");
        WeatherService weatherService = new WeatherService();
        PrintWriter out = response.getWriter();
        try {
            out.println( weatherService.retrieveForecast( zip ) );
        } catch( Exception e ) {
            out.println( "Error Retrieving Forecast: " + e.getMessage() );
        }
        out.flush();
        out.close();
    }
}

```

La `WeatherServlet` instancie la classe `WeatherService` définie dans le projet `simple-weather`. Le code postal fourni en paramètre de la requête est passé à la méthode `retrieveForecast()` et le résultat est écrit dans le `Writer` de la réponse.

Enfin, pour que tout cela fonctionne, il faut déclarer la servlet dans le fichier `web.xml` du module `simple-webapp`. Ce fichier se trouve dans le répertoire `src/main/webapp/WEB-INF`. Les

balises `servlet` et `servlet-mapping` du fichier `web.xml` associent l'URI `/weather` à la servlet `WeatherServlet`, comme le montre l'Exemple 6.6, « Fichier `web.xml` du module `simple-webapp` ».

Exemple 6.6. Fichier `web.xml` du module `simple-webapp`

```
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd" >

<web-app>
  <display-name>Archetype Created Web Application</display-name>
  <servlet>
    <servlet-name>simple</servlet-name>
    <servlet-class>org.sonatype.mavenbook.web.SimpleServlet</servlet-class>
  </servlet>
  <servlet>
    <servlet-name>weather</servlet-name>
    <servlet-class>org.sonatype.mavenbook.web.WeatherServlet</servlet-class>
  </servlet>
  <servlet-mapping>
    <servlet-name>simple</servlet-name>
    <url-pattern>/simple</url-pattern>
  </servlet-mapping>
  <servlet-mapping>
    <servlet-name>weather</servlet-name>
    <url-pattern>/weather</url-pattern>
  </servlet-mapping>
</web-app>
```

6.5. Construction du projet multimodule

Nous venons de créer le projet `simple-weather` qui contient le code interagissant avec le service Yahoo! Météo et le projet `simple-webapp` qui contient une simple servlet de présentation. C'est le moment de compiler et de packager notre application sous la forme d'un fichier WAR. Pour cela, vous devez compiler et installer les deux projets précédemment créés dans un ordre précis. Comme le projet `simple-webapp` dépend du projet `simple-weather`, nous devons transformer ce dernier (`simple-weather`) en JAR avant de pouvoir compiler la `webapp`. Pour ce faire, utilisons la commande `mvn clean install` à partir du projet de plus haut niveau `simple-parent` :

```
~/examples/ch-multi/simple-parent$ mvn clean install
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Simple Parent Project
[INFO]   simple-weather
[INFO]   simple-webapp Maven Webapp
[INFO] -----
[INFO] Building simple-weather
[INFO]   task-segment: [clean, install]
[INFO] -----
[...]
```

```

[INFO] [install:install]
[INFO] Installing simple-weather-1.0.jar to simple-weather-1.0.jar
[INFO] -----
[INFO] Building simple-webapp Maven Webapp
[INFO]   task-segment: [clean, install]
[INFO] -----
[...]
[INFO] [install:install]
[INFO] Installing simple-webapp.war to simple-webapp-1.0.war
[INFO] -----
[INFO] Reactor Summary:
[INFO] -----
[INFO] Simple Parent Project ..... SUCCESS [3.041s]
[INFO] simple-weather ..... SUCCESS [4.802s]
[INFO] simple-webapp Maven Webapp ..... SUCCESS [3.065s]
[INFO] -----

```

Lorsque Maven est exécuté dans un projet avec des sous-modules, il commence par charger le POM parent et localiser tous les POMs des sous-modules. Maven regroupe alors tous ces POMs dans le plugin Maven Reactor. Il s'agit d'un plugin chargé d'analyser les dépendances entre les modules. Le plugin Reactor s'occupe de l'ordonnement des composants pour garantir que les modules interdépendants soient compilés et installés dans le bon ordre.



Note

Le plugin Reactor préserve l'ordre des modules défini dans le POM à moins que des changements ne soient nécessaires. Une bonne technique pour comprendre ce mécanisme consiste à imaginer que les modules avec les dépendances inter projets soient descendus dans une liste jusqu'à ce que l'ordre des dépendances soit satisfaisant. Dans quelques rares occasions, il peut être bon de réarranger manuellement l'ordre de vos modules — par exemple si vous voulez qu'un module fréquemment instable soit remonté en début de build.

Dès que le plugin Reactor a trouvé l'ordre dans lequel les projets doivent être construits, Maven exécute les goals indiqués pour chaque module afin d'effectuer le build du projet multimodule. Dans cet exemple, vous pouvez effectivement remarquer que Maven commence par construire le module `simple-weather` avant l'application web `simple-webapp` en exécutant la commande `mvn clean install`.



Note

Quand vous exécutez Maven à partir de la ligne de commande vous serez amenés à utiliser fréquemment la phase `clean` avant l'exécution d'autres phases du cycle de vie. Lorsque vous précisez `clean` sur la ligne de commande, vous vous assurez que Maven enlève les fichiers générés lors des builds précédents avant qu'il ne compile et package votre application. Exécuter la commande `clean` n'est pas nécessaire, mais c'est une précaution utile pour s'assurer que vous effectuez un "build propre".

6.6. Exécution de l'application web

Une fois que le projet multimodule a été installé via la commande `mvn clean install` depuis le projet parent, `simple-project`, vous pouvez vous rendre dans le répertoire de la `simple-webapp` et lancer le goal `run` du plugin Jetty :

```
~/examples/ch06/simple-parent/simple-webapp $ mvn jetty:run
[INFO] -----
[INFO] Building simple-webapp Maven Webapp
[INFO]    task-segment: [jetty:run]
[INFO] -----
[...]
[INFO] [jetty:run]
[INFO] Configuring Jetty for project: simple-webapp Maven Webapp
[...]
[INFO] Webapp directory = ~/examples/ch06/simple-parent/\
                       simple-webapp/src/main/webapp
[INFO] Starting jetty 6.1.6rc1 ...
2007-11-18 1:58:26.980::INFO: jetty-6.1.6rc1
2007-11-18 1:58:26.125::INFO: No Transaction manager found
2007-11-18 1:58:27.633::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Une fois que Jetty est démarré, chargez la page <http://localhost:8080/simple-webapp/weather?zip=01201> dans votre navigateur pour consulter le service météo que nous venons de développer.

Chapitre 7. Un projet multimodule d'entreprise

7.1. Introduction

Dans ce chapitre, nous allons créer un projet multimodule à partir des exemples du Chapitre 6, Un projet multimodule et du Chapitre 5, Une simple application web. Nous ferons évoluer ces exemples en un projet utilisant les frameworks Spring et Hibernate pour créer deux applications : une application web et une application en ligne de commande. Ces deux applications permettront de lire le flux provenant du service Yahoo! Météo. Le code du projet `simple-weather` développé dans le Chapitre 4, Personnalisation d'un projet Maven sera donc combiné au code du projet `simple-webapp` du Chapitre 5, Une simple application web. Pour créer ce projet multimodule, nous allons explorer Maven et discuter des différentes manières de l'utiliser pour créer des projets modulaires encourageant la réutilisation.

7.1.1. Télécharger les sources de ce chapitre

Le projet multimodule développé dans cet exemple se compose des versions modifiées des projets développés dans le Chapitre 4, Personnalisation d'un projet Maven et le Chapitre 5, Une simple application web, sans utiliser le plugin Maven Archetype pour le générer. Nous recommandons de télécharger le code des exemples pour utiliser celui-ci comme code de référence lors de la lecture de ce chapitre. Sans ces exemples, vous ne serez pas capable de recréer la totalité du projet produit dans ce chapitre. Le projet contenant les exemples de ce chapitre peut être téléchargé à l'adresse suivante :

```
http://www.sonatype.com/books/maven-book/mavenbook-examples-0.9-SNAPSHOT-project.zip
```

Une fois téléchargée, décompressez l'archive dans n'importe quel répertoire et entrez dans le répertoire `ch-multi-spring/`. Vous trouverez alors dans celui-ci un sous-répertoire `simple-parent/` qui contient le projet Maven multimodule que nous allons développer dans ce chapitre. Dans le répertoire de ce projet, vous trouverez un fichier `pom.xml` et cinq sous-modules : `simple-model/`, `simple-persist/`, `simple-command/`, `simple-weather/` et `simple-webapp/`.

7.1.2. Projet multimodule d'entreprise

Présenter la complexité d'un projet d'entreprise d'envergure dépasse de loin la portée de ce livre. De tels projets sont souvent caractérisés par des problématiques bien spécifiques : bases de données multiples, intégration avec des systèmes externes, sous-projets divisés par départements... Ce genre de projets comporte en général plusieurs milliers de lignes de codes et peut impliquer des dizaines, voire des centaines, de développeurs. S'il est évident que nous ne traiterons pas d'un projet de ce type dans son intégralité dans ce livre, nous allons tout de même vous fournir un exemple assez complet pour

appréhender la complexité d'une application d'entreprise. Enfin, en conclusion, au-delà de ce qui est présenté dans ce chapitre, nous explorerons quelques pistes pour rendre vos applications modulaires.

Dans ce chapitre, nous allons étudier un projet Maven multimodule qui produira deux applications : un outil en ligne de commande et une application web ; chacune de ces applications permettra d'interroger le flux Yahoo! Météo. Ces deux applications conserveront les résultats des différentes requêtes dans une base de données embarquée et permettront de récupérer l'historique des prévisions météorologiques stockées dans celle-ci. Les deux applications réutiliseront le même code métier et partageront une bibliothèque de persistance. L'exemple de ce chapitre est construit sur la base du code du parseur développé dans le Chapitre 4, Personnalisation d'un projet Maven. Ce projet est divisé en cinq sous-modules, dont voici la présentation sur la Figure 7.1, « Relations entre les modules de l'application d'entreprise ».

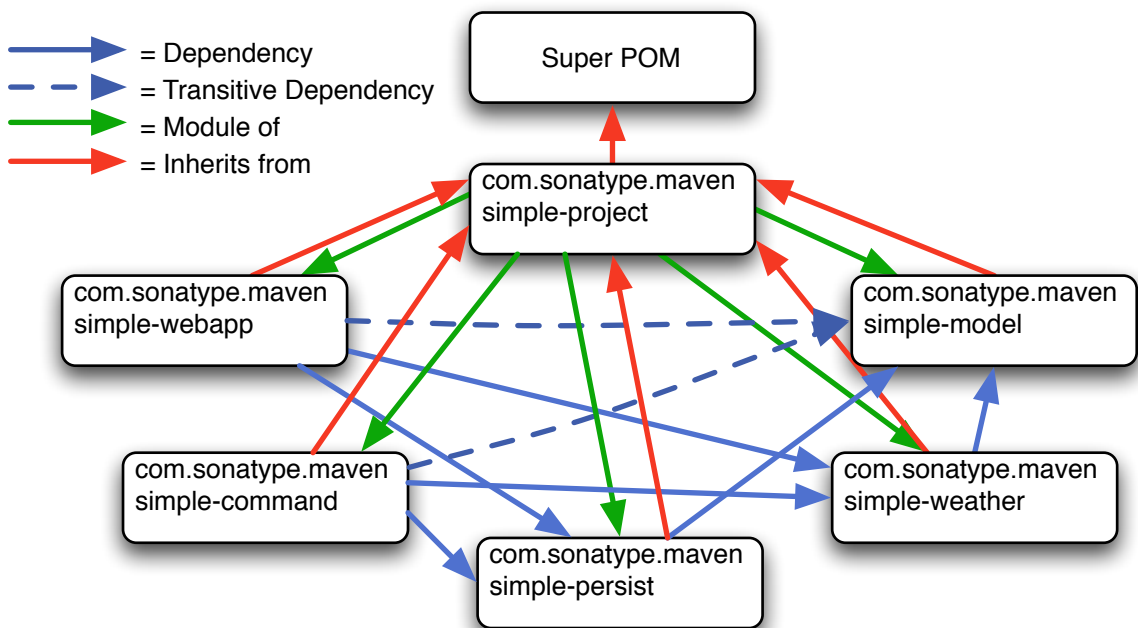


Figure 7.1. Relations entre les modules de l'application d'entreprise

Sur la Figure 7.1, « Relations entre les modules de l'application d'entreprise », vous pouvez voir que projet `simple-parent` se compose de cinq sous-modules :

simple-model

Ce module définit un simple modèle d'objets correspondant aux données retournées par le flux Yahoo! Météo. Ce modèle contient les objets suivants : `Weather`, `Condition`, `Atmosphere`, `Location`, et `Wind`. Lorsque notre application parse le flux Yahoo! Météo, les parseurs définis

dans le module `simple-weather` transforment le XML en une liste d'objets `Weather` qui sera ensuite utilisée dans les applications. Ce projet contient des objets du modèle annotés avec les annotations Hibernate3. Le module `simple-persist` utilise ces objets pour associer chacun d'entre eux à une table de la base de données.

`simple-weather`

Ce module contient toute la logique nécessaire pour récupérer des données du flux Yahoo! Météo et parser le XML obtenu. Ce XML est converti sous la forme d'objets du module `simple-model`. C'est pour cela que le module `simple-weather` doit dépendre du module `simple-model`. Enfin, `simple-weather` définit un service `WeatherService` qui est référencé par deux autres projets : `simple-command` et `simple-webapp`.

`simple-persist`

Ce module contient les DAOs (Data Access Objects) qui permettent d'enregistrer les objets `Weather` dans la base de données. Ces DAOs seront utilisés par les deux applications de ce projet multimodule. Ils permettront également de faire le lien avec les objets du modèle du module `simple-model`. Le module `simple-persist` a donc une dépendance directe au module `simple-model` et une dépendance vers le JAR contenant les Annotations Hibernate.

`simple-webapp`

L'application web contient deux contrôleurs Spring MVC qui utilisent le service `WeatherService` défini dans le module `simple-weather` et les DAOs provenant du module `simple-persist`. De plus, `simple-webapp` a une dépendance directe sur les modules `simple-weather` et `simple-persist`. Ce module dépend donc transitivement du module `simple-model`.

`simple-command`

Ce module contient une application en ligne de commande qui permet d'interroger le flux Yahoo! Météo. Ce projet contient une classe avec une méthode statique `main()` qui interagit avec le service `WeatherService` défini dans le module `simple-weather` et les DAOs du projet `simple-persist`. Le module `simple-command` a une dépendance directe sur les modules `simple-weather` et `simple-persist` et une dépendance transitive au module `simple-model`.

Ce chapitre contient un exemple assez simple pour être décrit dans un livre et assez complexe pour justifier la création de cinq sous-modules. Notre exemple est organisé autour d'un modèle contenant cinq classes, une bibliothèque de persistance avec deux classes de service et une bibliothèque d'analyse grammaticale du flux météo contenant cinq ou six classes. Dans un cas réel, un projet peut avoir un modèle de plusieurs centaines de classes, plusieurs bibliothèques de persistance et des bibliothèques de services partagées. Bien que nous ayons essayé de garder un exemple suffisamment simple pour être compris rapidement, nous vous proposons un projet bien plus complexe que le projet multimodule du chapitre précédent. Vous pourriez être tentés de regarder rapidement les exemples dans ce chapitre et penser que Maven génère trop de complexité comparée à la taille du modèle d'objet. Bien que l'utilisation de Maven suggère un certain niveau de modularité, comprenez bien que nous avons volontairement

compliqué ici nos exemples dans le but de montrer les fonctionnalités Maven dans le cas de projets multimodule.

7.1.3. Technologies utilisées dans cet exemple

L'exemple de ce chapitre implique un peu de technologies qui, même si elles sont populaires, ne sont pas directement liées à Maven. Ces technologies sont Spring et Hibernate. Spring est un conteneur IoC (Inversion of Control) accompagné par un ensemble de frameworks dont le but est de simplifier les interactions entre diverses bibliothèques J2EE. Utiliser Spring comme fondation pour le développement d'applications vous donne accès à un certain nombre d'abstractions très pratiques qui permettent, entre autres, de faciliter l'intégration des framework de persistance comme Hibernate, iBatis ou d'API d'entreprise comme JDBC, JNDI et JMS. La popularité de Spring n'a cessé de s'accroître ces dernières années, permettant d'offrir une solution alternative aux lourds standards proposés par Sun Microsystems. Quant à Hibernate, il s'agit d'un framework d'ORM (Object-Relational Mapping) très largement utilisé qui permet d'interagir avec une base de données comme s'il s'agissait d'une liste d'objets Java. Cet exemple se concentre sur la construction d'une application web simple et d'une application en ligne de commande qui utilisent Spring pour exposer un ensemble réutilisable de composants et Hibernate pour sauvegarder les données météo dans une base embarquée.

Nous avons décidé d'inclure des références à ces frameworks pour montrer comment on construirait des projets en utilisant ces mêmes technologies conjointement à Maven. Bien que nous introduisions ces technologies, ne s'agissant pas des priorités de ce chapitre, elles ne seront pas présentées dans leur intégralité. Pour plus d'informations sur Spring, référez-vous au site officiel dont l'adresse est <http://www.springframework.org/>. De même, pour plus d'informations sur Hibernate, en particulier sur ses Annotations, référez-vous au site du projet <http://www.hibernate.org>. Ce chapitre utilise HSQLDB (Hyper-threaded Structured Query Language Database) comme base de données embarquée. Pour plus d'informations sur celle-ci consultez le site du projet <http://hsqldb.org/>.

7.2. Le projet simple-parent

Ce projet `simple-parent` contient un fichier `pom.xml` qui référence cinq sous-modules : `simple-command`, `simple-model`, `simple-weather`, `simple-persist` et `simple-webapp`. Le fichier `pom.xml` de plus haut niveau est affiché dans l'Exemple 7.1, « POM du projet `simple-parent` ».

Exemple 7.1. POM du projet `simple-parent`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.sonatype.mavenbook.multispring</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
```



```

<version>1.0</version>
<name>Multi-Spring Chapter Simple Parent Project</name>

<modules>
  <module>simple-command</module>
  <module>simple-model</module>
  <module>simple-weather</module>
  <module>simple-persist</module>
  <module>simple-webapp</module>
</modules>

<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```



Note

Si les POMs Maven vous sont déjà familiers, vous pouvez remarquer que ce POM de plus haut niveau ne définit pas de balise `dependencyManagement`. La balise `dependencyManagement` permet de définir la version des dépendances à un endroit centralisé dans un POM de haut-niveau. Ce mécanisme sera abordée dans le Chapitre 8, Optimiser et remanier les POMs.

Notez les similarités de ce POM parent et celui de l'Exemple 6.1, « POM du projet simple-parent ». La seule réelle différence entre ces deux POMs est la liste de leurs sous-modules. Là où l'exemple précédent comptait deux sous-modules, ce POM en dénombre cinq. Les sections suivantes explorent en détail chacun de ces cinq sous-modules. Notre exemple utilisant des annotations, nous avons configuré le compilateur pour cibler les JVM Java 5.

7.3. Le module simple-model

La première chose dont ont besoin la plupart des projets d'entreprise est un modèle objet. Un modèle objet rassemble la liste des principaux objets du domaine d'un système. Par exemple, un système bancaire pourrait avoir un modèle objet qui se compose des objets suivants : `Compte`, `Client` et `Transaction`. De la même manière, un système publiant des résultats sportifs pourrait contenir des objets `Equipe` et `Match`. Quoique vous fassiez, il est probable que vous ayez intégré les concepts de votre système dans un modèle objet. C'est pratique courante dans les projets Maven de séparer ce type de projet et de le référencer dans le reste du projet. Dans notre système, nous transformons chaque requête du flux Yahoo! Météo dans un objet `Weather` qui référence quatre autres objets. La direction, l'effet de froid relatif et la vitesse du vent sont stockées dans un objet `Wind`. Les données de localisation telles que le code postal, la ville, la région et le pays sont stockées dans la classe `Location`. Les conditions atmosphériques telles que l'humidité, la visibilité, la pression barométrique et la tendance sont conservées dans une classe `Atmosphere`. Enfin, la description textuelle des conditions, la température et la date de l'observation sont conservées dans une classe `Condition`.

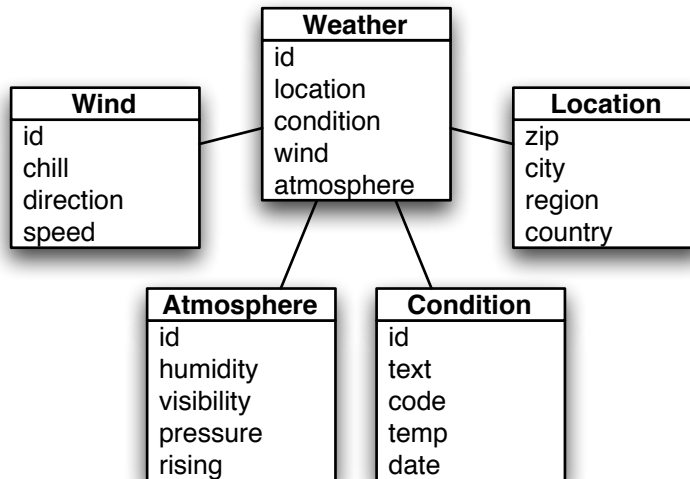


Figure 7.2. Modèle objet pour les données météo

Le fichier `pom.xml` de ce modèle objet contient une dépendance qui nécessite une explication. Notre modèle objet est annoté avec Hibernate Annotations. Celles-ci sont utilisées pour associer ce modèle aux tables de la base de données relationnelle. Cette dépendance est `org.hibernate:hibernate-annotations:3.3.0.ga`. Regardons le `pom.xml` affiché dans l'Exemple 7.2, « POM du module simple-model », ainsi que les quelques exemples d'utilisation de ces annotations.

Exemple 7.2. POM du module simple-model

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<parent>
  <groupId>org.sonatype.mavenbook.multispring</groupId>
  <artifactId>simple-parent</artifactId>
  <version>1.0</version>
</parent>
<artifactId>simple-model</artifactId>
<packaging>jar</packaging>

<name>Simple Object Model</name>

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-annotations</artifactId>
    <version>3.3.0.ga</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-commons-annotations</artifactId>
    <version>3.3.0.ga</version>
  </dependency>
</dependencies>
</project>

```

Dans le dossier `src/main/java/org/sonatype/mavenbook/weather/model` se trouve le fichier `Weather.java`. Celui-ci contient l'objet `Weather` annoté. Il s'agit d'un simple Java bean. Cela veut dire qu'il contient des variables d'instance privées comme `id`, `location`, `condition`, `wind`, `atmosphere` et `date`. Celles-ci sont exposées par l'intermédiaire d'accesseurs publics en suivant ce pattern : une propriété nommée `name` disposera d'un getter public sans argument nommé `getName()`, et d'un setter prenant en paramètre un argument nommé `setName(String name)`. Si nous montrons le getter et le setter de la propriété `id`, nous avons omis volontairement la plupart des getters et des setters des autres propriétés afin de sauver quelques arbres. Consultez l'Exemple 7.3, « `Objet Weather annoté` ».

Exemple 7.3. Objet Weather annoté

```

package org.sonatype.mavenbook.weather.model;

import javax.persistence.*;

import java.util.Date;

@Entity
@NamedQueries({
  @NamedQuery(name="Weather.byLocation",
              query="from Weather w where w.location = :location")
})
public class Weather {

```

```

@Id
@GeneratedValue(strategy=GenerationType.IDENTITY)
private Integer id;

@ManyToOne(cascade=CascadeType.ALL)
private Location location;

@OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
private Condition condition;

@OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
private Wind wind;

@OneToOne(mappedBy="weather",cascade=CascadeType.ALL)
private Atmosphere atmosphere;

private Date date;

public Weather() {}

public Integer getId() { return id; }
public void setId(Integer id) { this.id = id; }

// Nous avons omis les autres getter et setter...
}

```

La classe `Weather` utilise des annotations qui permettent de guider Hibernate pour associer cet objet à une table de la base de données relationnelle. Bien qu'une explication détaillée des annotations Hibernate dépasse les limites de ce chapitre, en voici quelques grandes lignes. L'annotation `@Entity` marque la classe comme entité persistante. Nous avons omis l'annotation `@Table` sur cette classe, ainsi, Hibernate va utiliser le nom de la classe comme nom de table. L'annotation `@NamedQueries` définit une requête qui sera utilisée par le `WeatherDAO` dans le module `simple-persist`. Le langage utilisé dans la requête de l'annotation `@NamedQuery` est écrit en HQL (Hibernate Query Language). Chaque champ de la classe est annoté par des annotations définissant le type des différentes colonnes à associer ainsi que les relations impliquées sur celles-ci :

Id

La propriété `id` est annotée avec `@Id`. Cette annotation marque ce champ comme propriété contenant la clé primaire de la table de base de données. L'annotation `@GeneratedValue` permet de contrôler comment les nouvelles valeurs de la clé primaire sont générées. Dans le cas de notre propriété `id`, nous utilisons la valeur `IDENTITY` de l'énumération `GenerationType` qui permet d'utiliser la génération d'identité fournie par la base de données sous-jacente.

Location

Chaque instance de la classe `Weather` contient un objet `Location`. Ce dernier représente un code postal. Son annotation `@ManyToOne` vous assure que tous les objets `Weather` qui possèdent la même `Location` pointent effectivement vers les mêmes instances. L'attribut `cascade` de

l'annotation `@ManyToOne` permet d'assurer qu'un objet `Location` soit enregistré à chaque sauvegarde d'un objet `Weather`.

`Condition`, `Wind`, `Atmosphere`

Chacun de ces objets est mappé avec une annotation `@OneToOne` dont la propriété `cascade` est à `CascadeType.ALL`. Cette propriété signifie qu'à chaque sauvegarde d'un objet `Weather` une ligne sera créée dans chacune des tables suivantes : `Condition`, `Wind` et `Atmosphere`.

`Date`

Le champ `Date` n'est pas annoté. Cela signifie qu'Hibernate va utiliser la configuration par défaut pour son mapping. Le nom de la colonne sera `date` et possédera le type `timestamp` approprié pour un objet `Date`.



Note

Si vous désirez omettre une propriété du mapping, vous pouvez annoter celle-ci avec `@Transient`.

Ensuite, jetons un coup d'oeil à un second objet du modèle, `Condition`, affiché dans l'Exemple 7.4, « Classe `Condition` du module `simple-model` ». Cette classe se trouve également dans le dossier `src/main/java/org/sonatype/mavenbook/weather/model`.

Exemple 7.4. Classe `Condition` du module `simple-model`

```
package org.sonatype.mavenbook.weather.model;

import javax.persistence.*;

@Entity
public class Condition {

    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    private Integer id;

    private String text;
    private String code;
    private String temp;
    private String date;

    @OneToOne(cascade=CascadeType.ALL)
    @JoinColumn(name="weather_id", nullable=false)
    private Weather weather;

    public Condition() {}

    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }
```

```
// Nous avons omis les autres getter et setter...  
}
```

La classe `Condition` ressemble à la classe `Weather`. Elle est annotée avec `@Entity` et possède les mêmes annotations sur la propriété `id`. Les champs `text`, `code`, `temp` et `date` ne possèdent pas d'annotation et restent donc liés à la base de données selon la configuration par défaut. La propriété `weather` est annotée avec `@OneToOne` et `@JoinColumn` qui permettent de référencer l'objet `Weather` associé via la clé étrangère `weather_id`.

7.4. Le module simple-weather

Le prochain module que nous allons examiner pourrait être considéré comme un “service”. Ce module contient toute la logique nécessaire pour récupérer et parser les données provenant du flux RSS Yahoo! Météo. Bien qu'il ne contienne que trois classes Java et un test JUnit, il permet d'exposer un composant simple, `WeatherService`, qui sera utilisé par l'application web et par l'application en ligne de commande. Un projet d'entreprise contient très souvent plusieurs modules API permettant de centraliser la logique métier et les interactions avec des systèmes externes. Un système bancaire pourrait avoir un module qui récupère et analyse des données provenant d'un fournisseur tiers, et un système affichant des résultats sportifs pourrait interagir avec un flux XML qui fournit les scores du basket ou du football en temps réel. Dans l'Exemple 7.5, « POM du module simple-weather », ce module renferme toute l'activité réseau et le parsing du flux XML provenant du service Yahoo! Météo. Les autres modules peuvent dépendre de celui-ci et simplement appeler la méthode `retrieveForecast()` du service `WeatherService` qui prend en paramètre un code postal et retourne un objet `Weather`.

Exemple 7.5. POM du module simple-weather

```
<project xmlns="http://maven.apache.org/POM/4.0.0"  
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0  
                             http://maven.apache.org/maven-v4_0_0.xsd">  
  <modelVersion>4.0.0</modelVersion>  
  <parent>  
    <groupId>org.sonatype.mavenbook.multispring</groupId>  
    <artifactId>simple-parent</artifactId>  
    <version>1.0</version>  
  </parent>  
  <artifactId>simple-weather</artifactId>  
  <packaging>jar</packaging>  
  
  <name>Simple Weather API</name>  
  
  <dependencies>  
    <dependency>  
      <groupId>org.sonatype.mavenbook.multispring</groupId>  
      <artifactId>simple-model</artifactId>
```

```

    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>
  <dependency>
    <groupId>dom4j</groupId>
    <artifactId>dom4j</artifactId>
    <version>1.6.1</version>
  </dependency>
  <dependency>
    <groupId>jaxen</groupId>
    <artifactId>jaxen</artifactId>
    <version>1.1.1</version>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.3.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

Le POM du module `simple-weather` étend celui du `simple-parent`, configure le type de packaging à `jar` et ajoute les dépendances suivantes :

```
org.sonatype.mavenbook.multispring:simple-model:1.0
```

Le module `simple-weather` parse et le transforme le flux RSS Yahoo! Météo en objet `Weather`. Il a donc une dépendance directe sur le module `simple-model`.

```
log4j:log4j:1.2.14
```

Le module `simple-weather` utilise `Log4J` pour afficher ses messages de log.

```
dom4j:dom4j:1.6.1 et jaxen:jaxen:1.1.1
```

Ces deux dépendances sont utilisées pour parser le XML provenant de Yahoo! Météo.

```
org.apache.commons:commons-io:1.3.2 (scope=test)
```

Cette dépendance dont le scope est `test` est utilisée par `YahooParserTest`.

Analysons maintenant le service `WeatherService` de l'Exemple 7.6, « La classe `WeatherService` ». Cette classe ressemble beaucoup au service `WeatherService` de l'Exemple 6.3, « La classe `WeatherService` ». Bien qu'ils aient le même nom, le service de cet exemple comporte quelques légères différences. Dans cette version, la méthode `retrieveForecast()` retourne un objet `Weather` et le formatage est délégué à l'application appelant le service `WeatherService`. L'autre modification

majeure concerne les classes `YahooRetriever` et `YahooParser`, qui sont maintenant des propriétés du service `WeatherService`.

Exemple 7.6. La classe `WeatherService`

```
package org.sonatype.mavenbook.weather;

import java.io.InputStream;

import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherService {

    private YahooRetriever yahooRetriever;
    private YahooParser yahooParser;

    public WeatherService() {}

    public Weather retrieveForecast(String zip) throws Exception {
        // Récupération des données
        InputStream dataIn = yahooRetriever.retrieve(zip);

        // Parsing des données
        Weather weather = yahooParser.parse(zip, dataIn);

        return weather;
    }

    public YahooRetriever getYahooRetriever() {
        return yahooRetriever;
    }

    public void setYahooRetriever(YahooRetriever yahooRetriever) {
        this.yahooRetriever = yahooRetriever;
    }

    public YahooParser getYahooParser() {
        return yahooParser;
    }

    public void setYahooParser(YahooParser yahooParser) {
        this.yahooParser = yahooParser;
    }
}
```

Pour finir, ce projet contient un fichier XML utilisé par Spring pour créer ce qu'on appelle l'`ApplicationContext` dont voici le fonctionnement : nos applications web et en ligne de commande ont besoin d'interagir avec le service `WeatherService`, pour cela elles récupèrent une instance de cette classe par l'intermédiaire de l'`ApplicationContext` Spring en utilisant le nom de bean `weatherService`. Notre application web utilise un contrôleur Spring MVC, celui-ci est associé à une instance du service `WeatherService`. L'application en ligne de commande charge ce même

service à partir de la méthode statique `main()` grâce à `ApplicationContext`. Afin d'encourager la réutilisation du code, nous avons inclus le fichier `applicationContext-weather.xml` dans le répertoire `src/main/resources`, le rendant ainsi accessible dans le classpath. Les projets qui dépendent du module `simple-weather` peuvent donc charger ce fichier en utilisant la classe `ClasspathXmlApplicationContext` fournie par Spring et ainsi récupérer une instance du service `WeatherService` par l'intermédiaire de son nom : `weatherService`.

Exemple 7.7. ApplicationContext Spring du module simple-weather

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

<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-2.0.xsd
       default-lazy-init="true">

    <bean id="weatherService"
          class="org.sonatype.mavenbook.weather.WeatherService">
        <property name="yahooRetriever" ref="yahooRetriever"/>
        <property name="yahooParser" ref="yahooParser"/>
    </bean>

    <bean id="yahooRetriever"
          class="org.sonatype.mavenbook.weather.YahooRetriever"/>

    <bean id="yahooParser"
          class="org.sonatype.mavenbook.weather.YahooParser"/>
</beans>
```

Ce fichier déclare trois beans : `yahooParser`, `yahooRetriever` et `weatherService`. Le bean `weatherService` est une instance du service `WeatherService`. Le fichier XML configure également les propriétés de ce bean `yahooParser` et `yahooRetriever` qui référencent les noms des deux instances de classes en question. Vous pouvez comparer ce fichier `applicationContext-weather.xml` à la définition de l'architecture d'un sous-système de ce projet multimodule. Certains projets comme `simple-webapp` et `simple-command` peuvent référencer ce contexte et récupérer une instance du service `WeatherService` configurée avec des instances des classes `YahooRetriever` et `YahooParser` en propriété.

7.5. Le module simple-persist

Ce module définit deux DAOs (Data Access Objects). Un DAO est un objet qui fournit une interface aux opérations de persistance. Dans une application qui utilise un framework de mapping ORM (Object-Relational Mapping) comme Hibernate, un DAO est défini autour d'un objet. Dans ce projet, nous définissons deux objets DAOs : `WeatherDAO` et `LocationDAO`. La classe `WeatherDAO` nous permet de sauvegarder un objet `Weather` dans la base de données, de le récupérer à partir de son `id` ou d'une `Location` spécifique. Le DAO `LocationDAO` contient une méthode qui permet la récupération d'un

objet `Location` à partir d'un code postal. Commençons par regarder le POM du module `simple-persist` dans l'Exemple 7.8, « POM du module `simple-persist` ».

Exemple 7.8. POM du module `simple-persist`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.multispring</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-persist</artifactId>
  <packaging>jar</packaging>

  <name>Simple Persistence API</name>

  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
      <artifactId>simple-model</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>3.2.5.ga</version>
      <exclusions>
        <exclusion>
          <groupId>javax.transaction</groupId>
          <artifactId>jta</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</project>
```

```

<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring</artifactId>
  <version>2.0.7</version>
</dependency>
</dependencies>
</project>

```

Ce POM référence celui du module `simple-parent` en POM parent et définit quelques dépendances. Les dépendances listées dans le POM du module `simple-persist` sont :

```
org.sonatype.mavenbook.multispring:simple-model:1.0
```

Tout comme le module `simple-weather`, ce module de persistance référence les objets du modèle définis dans le projet `simple-model`.

```
org.hibernate:hibernate:3.2.5.ga
```

Nous définissons une dépendance sur la version 3.2.5.ga d'Hibernate. Notez l'exclusion d'une dépendance transitive d'Hibernate, nous effectuons cela à cause de la dépendance `javax.transaction:jta` qui n'est pas récupérable à partir du repository Maven public. Cette dépendance est l'une de ces dépendances Sun qui ne sont pas encore publiées dans le dépôt Maven libre 'central'. Afin d'éviter le message nous demandant d'aller télécharger cette dépendance, nous excluons simplement celle-ci d'Hibernate.

```
javax.servlet:servlet-api:2.4
```

Comme le projet contient une Servlet, nous devons inclure une version de Servlet API, ici la 2.4.

```
org.springframework:spring:2.0.7
```

Cette dépendance inclut l'intégration de Spring Framework.



Note

C'est généralement une bonne pratique de dépendre uniquement des composants Spring que vous utilisez. Spring dispose d'ailleurs d'artefacts spécifiques tels que `spring-hibernate3`.

Pourquoi dépendre de Spring ? Lorsqu'il est utilisé avec son intégration Hibernate, Spring permet l'utilisation des classes Helpers comme `HibernateDaoSupport` facilitant l'utilisation d'Hibernate. Pour savoir ce qu'il est possible de faire avec la classe `HibernateDaoSupport`, regardez le code de la classe `WeatherDAO` de l'Exemple 7.9, « Classe `WeatherDAO` du module `simple-persist` ».

Exemple 7.9. Classe `WeatherDAO` du module `simple-persist`

```

package org.sonatype.mavenbook.weather.persist;

import java.util.ArrayList;
import java.util.List;

import org.hibernate.Query;
import org.hibernate.Session;

```

```

import org.springframework.orm.hibernate3.HibernateCallback;
import org.springframework.orm.hibernate3.support.HibernateDaoSupport;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherDAO extends HibernateDaoSupport ❶ {

    public WeatherDAO() {}

    public void save(Weather weather) {❷
        getHibernateTemplate().save( weather );
    }

    public Weather load(Integer id) {❸
        return (Weather) getHibernateTemplate().load( Weather.class, id);
    }

    @SuppressWarnings("unchecked")
    public List<Weather> recentForLocation( final Location location ) {
        return (List<Weather>) getHibernateTemplate().execute(
            new HibernateCallback() {❹
                public Object doInHibernate(Session session) {
                    Query query = getSession().getNamedQuery("Weather.byLocation");
                    query.setParameter("location", location);
                    return new ArrayList<Weather>( query.list() );
                }
            });
    }
}

```

C'est tout ? Pas vraiment, mais cela suffit pour obtenir une classe qui permet d'insérer de nouvelles lignes, en récupérer par leur clé primaire et récupérer toutes les lignes `Weather` qui réfèrent à un `id` de la table `Location`. Nous ne pouvons clairement pas arrêter ce livre et insérer les cinq cents pages qu'il faudrait pour comprendre les subtilités d'Hibernate, contentons-nous de quelques rapides explications :

- ❶ Cette classe étend `HibernateDaoSupport`. Ce qui signifie que cette classe est associée à une `SessionFactory` Hibernate qui permet la création d'objets `Session` Hibernate. Avec Hibernate, chaque opération s'effectue par l'intermédiaire d'un objet `Session`, celle-ci s'occupe des accès à la base de données et de la connexion via la `DataSource` JDBC. Étendre `HibernateDaoSupport` vous permet également d'accéder au template `HibernateTemplate` en utilisant la méthode `getHibernateTemplate()`.
- ❷ La méthode `save()` prend une instance de la classe `Weather` et appelle la méthode `save()` de la classe `HibernateTemplate`. L'`HibernateTemplate` simplifie les appels aux opérations standards d'Hibernate et convertit les exceptions typées des différentes bases de données en `RuntimeException`. Appelons la méthode `save()` qui insère une nouvelle ligne dans la table `Weather`. La mise à jour d'une entité déjà en base passe par la méthode `update()`. La méthode `saveOrUpdate()` crée une nouvelle ligne ou la modifie en fonction de la présence d'une valeur non-nulle dans la propriété `id` de la classe.

- ③ De la même manière, la méthode `load()` se contente d'appeler la méthode du même nom de l'instance `HibernateTemplate`. Cette méthode prend un objet `Class` et un `Serializable` en paramètres. Dans notre exemple, l'objet `Serializable` correspond à la valeur de `id` de l'objet `Weather` à charger.
- ④ Cette dernière méthode `recentForLocation()` appelle une `NamedQuery` définie dans l'objet modèle `Weather`. Il s'agit de la requête nommée "Weather.byLocation" dont le code est "from Weather w where w.location = :location". Nous chargeons cette `NamedQuery` en utilisant une référence de la `Session` `Hibernate` dans un `HibernateCallback` qui est lancé via un appel à la méthode `execute()` de l'`HibernateTemplate`. Dans cette méthode, nous remplissons le paramètre `location` avec le paramètre passé dans la méthode `recentForLocation()`.

C'est maintenant le bon moment de clarifier certains points. Les classes `HibernateDaoSupport` et `HibernateTemplate` proviennent du framework Spring. Elles ont été créées par Spring pour faciliter l'écriture de DAO. Pour cela, nous avons besoin de modifier la configuration de l'`ApplicationContext` Spring du module `simple-persist`. Le fichier XML de l'Exemple 7.10, « `ApplicationContext` Spring du module `simple-persist` » se trouve dans le dossier `src/main/resources` dans un fichier nommé `applicationContext-persist.xml`.

Exemple 7.10. `ApplicationContext` Spring du module `simple-persist`

```
<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-2.0.xsd"
  default-lazy-init="true">

  <bean id="sessionFactory"
class="org.springframework.orm.hibernate3.annotation.AnnotationSessionFactoryBean">
    <property name="annotatedClasses">
      <list>
        <value>org.sonatype.mavenbook.weather.model.Atmosphere</value>
        <value>org.sonatype.mavenbook.weather.model.Condition</value>
        <value>org.sonatype.mavenbook.weather.model.Location</value>
        <value>org.sonatype.mavenbook.weather.model.Weather</value>
        <value>org.sonatype.mavenbook.weather.model.Wind</value>
      </list>
    </property>
    <property name="hibernateProperties">
      <props>
        <prop key="hibernate.show_sql">false</prop>
        <prop key="hibernate.format_sql">>true</prop>
        <prop key="hibernate.transaction.factory_class">
          org.hibernate.transaction.JDBCTransactionFactory
        </prop>
        <prop key="hibernate.dialect">
          org.hibernate.dialect.HSQLDialect
        </prop>
        <prop key="hibernate.connection.pool_size">0</prop>
        <prop key="hibernate.connection.driver_class">

```

```

        org.hsqldb.jdbcDriver
    </prop>
    <prop key="hibernate.connection.url">
        jdbc:hsqldb:data/weather;shutdown=true
    </prop>
    <prop key="hibernate.connection.username">sa</prop>
    <prop key="hibernate.connection.password"></prop>
    <prop key="hibernate.connection.autocommit">true</prop>
    <prop key="hibernate.jdbc.batch_size">0</prop>
</props>
</property>
</bean>

<bean id="locationDAO"
    class="org.sonatype.mavenbook.weather.persist.LocationDAO">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>

<bean id="weatherDAO"
    class="org.sonatype.mavenbook.weather.persist.WeatherDAO">
    <property name="sessionFactory" ref="sessionFactory"/>
</bean>
</beans>

```

Cet `ApplicationContext` définit plusieurs choses. Le bean `sessionFactory` est le bean par lequel les DAOs pourront récupérer les `Session` Hibernate. Ce bean est une instance de la classe `AnnotationSessionFactoryBean` et fournit une liste d'`annotatedClasses`. Notez que cette liste de classes annotées est la liste des classes définies dans le module `simple-model`. Ensuite, la `sessionFactory` est configurée par l'intermédiaire d'une liste de propriétés de configuration Hibernate (`hibernateProperties`). Dans cet exemple, nous utilisons les propriétés Hibernate suivantes :

`hibernate.dialect`

Cette propriété permet de contrôler le SQL qui sera généré pour notre base de données. Comme nous utilisons HSQLDB, nous choisissons le dialecte `org.hibernate.dialect.HSQLDialect`. Hibernate possède nativement des dialectes pour la majorité des bases de données comme Oracle, MySQL, Postgres et SQL Server.

`hibernate.connection.*`

Dans cet exemple, nous configurons une connexion JDBC à partir de la configuration Spring. Nos applications sont configurées pour se lancer avec la base de données HSQLDB du répertoire `./data/weather`. Dans une vraie application d'entreprise, vous utiliseriez plutôt quelque chose comme JNDI pour externaliser la configuration de la base de données de votre code applicatif.

Dans la suite du fichier XML, sont définis les deux DAOs du module `simple-persist`. Chacun d'entre eux contient une référence vers la `sessionFactory` Hibernate que nous venons de définir. Comme l'`ApplicationContext` du module `simple-weather`, ce fichier `applicationContext-persist.xml` définit l'architecture du sous-module. Si vous travailliez avec un plus grand nombre de classes persistantes, vous pourriez trouver utile de les externaliser dans un `ApplicationContext` séparé.

Il reste encore une dernière pièce du puzzle. Plus tard dans ce chapitre, nous verrons comment nous pouvons utiliser le plugin Maven Hibernate3 pour générer notre schéma de base de données à partir des objets annotés du modèle. Pour cela, le plugin Maven Hibernate3 a besoin de récupérer les paramètres de connexion JDBC, la liste des classes annotées et un autre fichier de configuration nommé `hibernate.cfg.xml` présent dans le répertoire `src/main/resources`. Le contenu de ce fichier (qui duplique une partie de la configuration du fichier `applicationContext-persist.xml`) nous permet l'utilisation du plugin Maven Hibernate3 pour générer le DDL (Data Definition Language) du schéma de la base de données. Consultez l'Exemple 7.11, « Fichier `hibernate.cfg.xml` du module `simple-persist` ».

Exemple 7.11. Fichier `hibernate.cfg.xml` du module `simple-persist`

```
<!DOCTYPE hibernate-configuration PUBLIC
    "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
    "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>
  <session-factory>

    <!-- dialecte SQL -->
    <property name="dialect">org.hibernate.dialect.HSQLDialect</property>

    <!-- Configuration des connexions -->
    <property name="connection.driver_class">org.hsqldb.jdbcDriver</property>
    <property name="connection.url">jdbc:hsqldb:data/weather</property>
    <property name="connection.username">sa</property>
    <property name="connection.password"></property>
    <property name="connection.shutdown">>true</property>

    <!-- Pool de connexion JDBC (utilisation du built-in) -->
    <property name="connection.pool_size">1</property>

    <!-- Définit le scope des context Hibernate -->
    <property name="current_session_context_class">thread</property>

    <!-- Désactivation du cache de second niveau -->
    <property name="cache.provider_class">
      org.hibernate.cache.NoCacheProvider
    </property>

    <!-- Affiche les requêtes sur stdout -->
    <property name="show_sql">>true</property>

    <!-- Désactivation du batching, permet à HSQLDB de propager ses erreurs correctement. -->
    <property name="jdbc.batch_size">0</property>

    <!-- Liste de toutes les classes annotées -->
    <mapping class="org.sonatype.mavenbook.weather.model.Atmosphere"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Condition"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Location"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Weather"/>
    <mapping class="org.sonatype.mavenbook.weather.model.Wind"/>
```

```
</session-factory>  
</hibernate-configuration>
```

Les contenus de l'Exemple 7.10, « ApplicationContext Spring du module simple-persist » et de l'Exemple 7.1, « POM du projet simple-parent » sont redondants. Le XML de l'ApplicationContext Spring sera utilisé par l'application web et l'application en ligne de commande, le fichier `hibernate.cfg.xml` n'est présent que pour faire fonctionner le plugin Maven Hibernate3. Plus tard dans ce chapitre, nous verrons comment utiliser ce fichier `hibernate.cfg.xml` et le plugin Maven Hibernate3 pour générer le schéma de la base de données basé sur les objets annotés du projet `simple-model`. Ce fichier `hibernate.cfg.xml` configure les propriétés des connexions JDBC et énumère la liste des objets annotés.

7.6. Le module simple-webapp

L'application Web est définie dans le module `simple-webapp`. Ce projet va définir deux contrôleurs Spring MVC : `WeatherController` et `HistoryController`. Ces deux contrôleurs vont référencer les composants des modules `simple-weather` et `simple-persist`. Le conteneur Spring est configuré dans le fichier `web.xml` de l'application web, celui-ci réfère le fichier `applicationContext-weather.xml` du module `simple-weather` et le fichier `applicationContext-persist.xml` du module `simple-persist`. L'architecture des composants de cette application web simple est montrée sur la Figure 7.3, « Contrôleurs Spring MVC référençant les modules `simple-weather` et `simple-persist`. ».

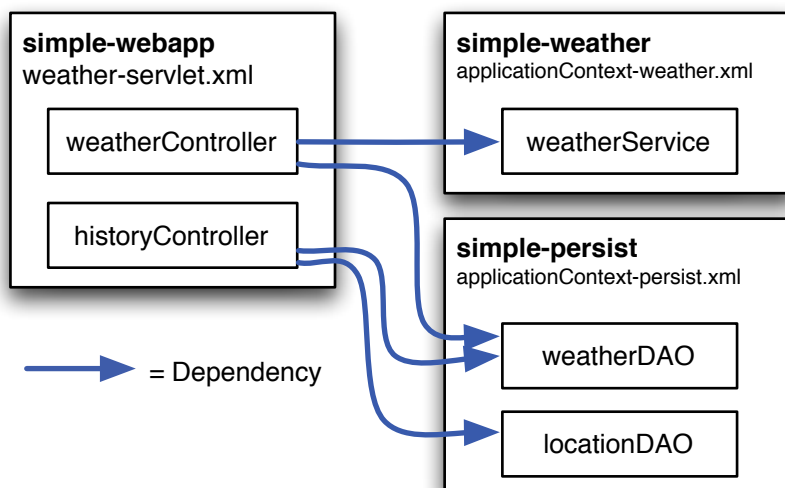


Figure 7.3. Contrôleurs Spring MVC référençant les modules `simple-weather` et `simple-persist`.

Le POM du module `simple-webapp` est affiché dans l'Exemple 7.12, « POM du module `simple-webapp` ».

Exemple 7.12. POM du module `simple-webapp`

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.multispring</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-webapp</artifactId>
  <packaging>war</packaging>
  <name>Simple Web Application</name>
  <dependencies>
    <dependency> ❶
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
      <artifactId>simple-weather</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.multispring</groupId>
      <artifactId>simple-persist</artifactId>
      <version>1.0</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
      <version>1.5</version>
    </dependency>
  </dependencies>
  <build>
    <finalName>simple-webapp</finalName>
    <plugins>
      <plugin> ❷
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>maven-jetty-plugin</artifactId>
        <dependencies> ❸
```

```

    <dependency>
      <groupId>hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>1.8.0.7</version>
    </dependency>
  </dependencies>
</plugin>
<plugin>
  <groupId>org.codehaus.mojo</groupId> ❹
  <artifactId>hibernate3-maven-plugin</artifactId>
  <version>2.0</version>
  <configuration>
    <components>
      <component>
        <name>hbm2ddl</name>
        <implementation>annotationconfiguration</implementation> ❺
      </component>
    </components>
  </configuration>
  <dependencies>
    <dependency>
      <groupId>hsqldb</groupId>
      <artifactId>hsqldb</artifactId>
      <version>1.8.0.7</version>
    </dependency>
  </dependencies>
</plugin>
</plugins>
</build>
</project>

```

Au fur et à mesure que nous avançons dans ce livre, les exemples deviennent de plus en plus substantiels. Vous avez d'ailleurs probablement noté que le fichier `pom.xml` commence à devenir volumineux. Dans ce POM, nous configurons quatre dépendances et deux plugins. Regardons ce POM en détail et étendons-nous sur certains des points de configuration importants :

- ❶ Ce projet `simple-webapp` définit quatre dépendances : les spécifications Servlet 2.4, la bibliothèque de services `simple-weather`, la bibliothèque de persistance `simple-persist` et l'intégralité du Spring Framework 2.0.7.
- ❷ Le plugin Maven Jetty est utilisé de la manière la plus simple qui soit dans ce projet. Nous ajoutons simplement la balise `plugin` qui fait référence aux bons `groupId` et `artifactId`. Le fait que ce plugin soit facile à configurer signifie que les développeurs du plugin ont fait du bon travail en fournissant les valeurs par défaut adéquates pour la plupart des cas. Si vous avez besoin de surcharger cette configuration, vous pouvez le faire en ajoutant une balise `configuration`.
- ❸ Dans notre configuration de build, nous allons configurer le plugin Maven Hibernate3 pour qu'il utilise une instance de la base de données embarquée HSQLDB. Pour que le plugin Maven Hibernate3 arrive à se connecter à la base, il doit référencer le driver JDBC d'HSQLDB dans son classpath. Pour rendre cette dépendance disponible au plugin, nous ajoutons sa déclaration directement dans la déclaration du `plugin`. Ici, nous référençons `hsqldb:hsqldb:1.8.0.7`. Le plugin

Hibernate a également besoin des drivers JDBC pour créer cette base de données, nous avons donc ajouté cette dépendance dans sa configuration.

- ④ C'est à partir de l'utilisation du plugin Maven Hibernate3 que ce POM devient le plus intéressant. Dans la section suivante, nous allons exécuter le goal `hbm2ddl` pour générer une base de données HSQLDB. Nous avons inclus dans ce `pom.xml` une référence vers la version 2.0 du `hibernate3-maven-plugin` récupérable à partir du dépôt Codehaus Mojo.
- ⑤ Le plugin Maven Hibernate3 dispose de plusieurs moyens pour récupérer le mapping Hibernate en fonction du cas d'utilisation. Si vous utilisez des fichiers XML pour le Mapping Hibernate (`.hbm.xml`), et que vous voulez générer les classes de votre modèle par l'intermédiaire du goal `hbm2java`, vous devez indiquer votre implémentation à la configuration. Si vous utilisez le plugin Hibernate3 en reverse engineering pour générer les fichiers `.hbm.xml` et les objets du modèle à partir d'une base de données existante, vous voudrez utiliser l'implémentation `jdbccconfiguration`. Dans notre exemple, nous utilisons simplement les objets annotés du modèle pour générer la base de données. En d'autres termes, nous avons notre mapping Hibernate, mais nous n'avons pas encore de base de données. Pour ce scénario, `annotationconfiguration` est l'implémentation la plus appropriée. Le plugin Maven Hibernate3 sera plus largement détaillé dans la Section 7.7, « Exécution de l'application web ».



Note

Une erreur classique consiste à utiliser la balise `extensions` de la configuration pour ajouter des dépendances nécessaires à un plugin. Il est fortement découragé de procéder de la sorte car les extensions peuvent polluer le classpath de votre projet, et provoquer des effets de bord désagréables. De plus, le comportement des extensions a été modifié depuis la version 2.1 de Maven. Le seul usage correct de la balise `extensions` est l'ajout d'implémentations complémentaires.

Regardons maintenant les deux contrôleurs Spring MVC, chacun d'entre eux référence des beans déclarés dans les `simple-weather` et `simple-persist`.

Exemple 7.13. WeatherController du module `simple-webapp`

```
package org.sonatype.mavenbook.web;

import org.sonatype.mavenbook.weather.model.Weather;
import org.sonatype.mavenbook.weather.persist.WeatherDAO;
import org.sonatype.mavenbook.weather.WeatherService;
import javax.servlet.http.*;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;

public class WeatherController implements Controller {

    private WeatherService weatherService;
    private WeatherDAO weatherDAO;

    public ModelAndView handleRequest(HttpServletRequest request,
```

```

        HttpServletResponse response) throws Exception {

    String zip = request.getParameter("zip");
    Weather weather = weatherService.retrieveForecast(zip);
    weatherDAO.save(weather);
    return new ModelAndView("weather", "weather", weather);
}

public WeatherService getWeatherService() {
    return weatherService;
}

public void setWeatherService(WeatherService weatherService) {
    this.weatherService = weatherService;
}

public WeatherDAO getWeatherDAO() {
    return weatherDAO;
}

public void setWeatherDAO(WeatherDAO weatherDAO) {
    this.weatherDAO = weatherDAO;
}
}

```

WeatherController implémente l'interface Controller qui assure la présence d'une méthode `handleRequest()` dont la signature est montrée dans cet exemple. Si vous regardez le corps de cette méthode, vous constaterez que celle-ci invoque la méthode `retrieveForecast()` du service `weatherService`. Contrairement au chapitre précédent, dans lequel une servlet instancie la classe `WeatherService`, le `WeatherController` est un bean contenant une propriété `weatherService`. Le conteneur Spring IoC a la responsabilité de charger le service `weatherService` dans le contrôleur. Notez également que nous n'utilisons pas le `WeatherFormatter` dans ce contrôleur. Au lieu de cela, nous passons l'objet `Weather` retourné par la méthode `retrieveForecast()` au constructeur de la classe `ModelAndView`. Cette classe `ModelAndView` est utilisée pour effectuer le rendu du template Velocity. Ce template disposera d'une référence sur la variable `#{weather}` contenant ce même objet. Le template `weather.vm` se trouve dans le répertoire `src/main/webapp/WEB-INF/vm`, celui-ci est affiché dans l'Exemple 7.14, « Modèle `weather.vm` interprété par le `WeatherController` ».

Dans ce `WeatherController`, avant d'effectuer le rendu des prévisions météo, nous passons l'objet `Weather` retourné par le service `WeatherService` à la méthode `save()` de la classe `WeatherDAO`. Cet objet est sauvegardé par Hibernate en base de données. Plus tard, dans le contrôleur `HistoryController`, nous verrons comment récupérer l'historique de ces prévisions météorologiques.

Exemple 7.14. Modèle `weather.vm` interprété par le `WeatherController`

```

<b>Current Weather Conditions for:
  #{weather.location.city}, #{weather.location.region},
  #{weather.location.country}</b><br/>

```

```

<ul>
  <li>Temperature: ${weather.condition.temp}</li>
  <li>Condition: ${weather.condition.text}</li>
  <li>Humidity: ${weather.atmosphere.humidity}</li>
  <li>Wind Chill: ${weather.wind.chill}</li>
  <li>Date: ${weather.date}</li>
</ul>

```

La syntaxe de ce modèle Velocity est rapide à expliquer, les variables sont référencées par l'intermédiaire de la notation `{}`. L'expression entre les accolades fait référence à une propriété, ou à une propriété d'une propriété de la variable `weather` passée à ce modèle par le `WeatherController`.

Le contrôleur `HistoryController` est utilisé pour récupérer la liste des prévisions météorologiques les plus récentes parmi celles demandées par le `WeatherController`. Chaque fois que nous récupérons une prévision dans le `WeatherController`, celui-ci enregistre l'objet `Weather` récupéré dans la base de données via le `WeatherDAO`. Ce DAO utilise ensuite Hibernate pour transformer l'objet `Weather` en une série de lignes dans un ensemble de tables de la base de données. Le contrôleur `HistoryController` est affiché dans l'Exemple 7.15, « `HistoryController` du module `simple-web` ».

Exemple 7.15. `HistoryController` du module `simple-web`

```

package org.sonatype.mavenbook.web;

import java.util.*;
import javax.servlet.http.*;
import org.springframework.web.servlet.ModelAndView;
import org.springframework.web.servlet.mvc.Controller;
import org.sonatype.mavenbook.weather.model.*;
import org.sonatype.mavenbook.weather.persist.*;

public class HistoryController implements Controller {

    private LocationDAO locationDAO;
    private WeatherDAO weatherDAO;

    public ModelAndView handleRequest(HttpServletRequest request,
        HttpServletResponse response) throws Exception {
        String zip = request.getParameter("zip");
        Location location = locationDAO.findByZip(zip);
        List<Weather> weathers = weatherDAO.recentForLocation( location );

        Map<String, Object> model = new HashMap<String, Object>();
        model.put( "location", location );
        model.put( "weathers", weathers );

        return new ModelAndView("history", model);
    }

    public WeatherDAO getWeatherDAO() {
        return weatherDAO;
    }
}

```

```

}

public void setWeatherDAO(WeatherDAO weatherDAO) {
    this.weatherDAO = weatherDAO;
}

public LocationDAO getLocationDAO() {
    return locationDAO;
}

public void setLocationDAO(LocationDAO locationDAO) {
    this.locationDAO = locationDAO;
}
}

```

Le contrôleur `HistoryController` est chargé avec les deux DAOs du module `simple-persist`. Les instances de ces deux DAOs sont portées par les propriétés `WeatherDAO` et `LocationDAO`. Le but de l'`HistoryController` est de récupérer une `List` d'objets `Weather` en fonction du paramètre `zip`. Quand le `WeatherDAO` enregistre un objet `Weather` dans la base de données, il ne se contente pas de stocker le code postal, il enregistre un objet `Location` qui est lié à l'objet `Weather` du module `simple-model`. Pour récupérer la `List` des objets `Weather`, l'`HistoryController` commence par récupérer l'objet `Location` qui correspond au paramètre `zip` représentant le code postal. Pour cela, on invoque la méthode `findByZip()` du DAO `LocationDAO`.

Une fois cet objet `Location` récupéré, l'`HistoryController` essaiera de récupérer les objets `Weather` les plus récents associés à cette `Location`. Une fois la `List<Weather>` récupérée, une `HashMap` est créée contenant les deux variables du modèle Velocity `history.vm` de l'Exemple 7.16, « Modèle `history.vm` utilisé par l'`HistoryController` ».

Exemple 7.16. Modèle `history.vm` utilisé par l'`HistoryController`

```

<b>
Weather History for: ${location.city}, ${location.region}, ${location.country}
</b>
<br/>

#foreach( $weather in $weathers )
  <ul>
    <li>Temperature: $weather.condition.temp</li>
    <li>Condition: $weather.condition.text</li>
    <li>Humidity: $weather.atmosphere.humidity</li>
    <li>Wind Chill: $weather.wind.chill</li>
    <li>Date: $weather.date</li>
  </ul>
#end

```

Le modèle `history.vm` du dossier `src/main/webapp/WEB-INF/vm` référence la variable `location` pour afficher les prévisions météo provenant du `WeatherDAO`. Ce template utilise une structure de

contrôle Velocity, `#foreach`, pour itérer sur chaque élément de la variable `weathers`. Chaque élément de la liste `weathers` est assigné à une variable nommée `weather`. Le cœur de la boucle, entre `#foreach` et `#end`, permet d'afficher les informations de chaque prévision.

Nous venons de voir les implémentations de nos deux Controller. Nous venons également de voir comment ils référencent les beans des modules `simple-weather` et `simple-persist`. Ces Controller répondent aux requêtes HTTP par l'intermédiaire de mystérieux systèmes qui savent comment effectuer le rendu de templates Velocity. Toute la magie est configurée dans l'ApplicationContext Spring du fichier `src/main/webapp/WEB-INF/weather-servlet.xml`. Ce XML configure les contrôleurs et référence d'autres beans managés par Spring. Il est chargé par un `ServletContextListener` qui est configuré pour charger également les fichiers `applicationContext-weather.xml` et `applicationContext-persist.xml` à partir du classpath. Regardons de plus près le fichier `weather-servlet.xml` de l'Exemple 7.17, « Configuration des contrôleurs Spring du fichier `weather-servlet.xml` ».

Exemple 7.17. Configuration des contrôleurs Spring du fichier `weather-servlet.xml`

```
<beans>
  <bean id="weatherController" ❶
    class="org.sonatype.mavenbook.web.WeatherController">
    <property name="weatherService" ref="weatherService"/>
    <property name="weatherDAO" ref="weatherDAO"/>
  </bean>

  <bean id="historyController"
    class="org.sonatype.mavenbook.web.HistoryController">
    <property name="weatherDAO" ref="weatherDAO"/>
    <property name="locationDAO" ref="locationDAO"/>
  </bean>

  <!-- you can have more than one handler defined -->
  <bean id="urlMapping"
    class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="urlMap">
      <map>
        <entry key="/weather.x"> ❷
          <ref bean="weatherController" />
        </entry>
        <entry key="/history.x">
          <ref bean="historyController" />
        </entry>
      </map>
    </property>
  </bean>

  <bean id="velocityConfig" ❸
    class="org.springframework.web.servlet.view.velocity.VelocityConfigurer">
    <property name="resourceLoaderPath" value="/WEB-INF/vm"/>
  </bean>
```

```

    <bean id="viewResolver" ❷
class="org.springframework.web.servlet.view.velocity.VelocityViewResolver">
    <property name="cache" value="true"/>
    <property name="prefix" value=""/>
    <property name="suffix" value=".vm"/>
    <property name="exposeSpringMacroHelpers" value="true"/>
    </bean>
</beans>

```

- ❶ Le fichier `weather-servlet.xml` définit deux contrôleurs Spring, `weatherController` a deux propriétés qui référencent `weatherService` et `weatherDAO`. `historyController` référence les beans `weatherDAO` et `locationDAO`. Quand l'`ApplicationContext` est créé, il est créé dans un environnement qui donne accès aux `ApplicationContexts` définis dans les fichiers `simple-persist` et `simple-weather`. Dans l'Exemple 7.18, « `web.xml` du module `simple-webapp` » vous pouvez voir comment Spring est configuré pour fusionner plusieurs composants contenant des fichiers de configuration Spring différents.
- ❷ Le bean `urlMapping` définit les formats des URL pouvant invoquer les contrôleurs `WeatherController` et `HistoryController`. Dans cet exemple, nous utilisons le `SimpleUrlHandlerMapping` et associons `/weather.x` au `WeatherController` et `/history.x` à l'`HistoryController`.
- ❸ Comme nous utilisons le moteur de template Velocity, nous avons besoin de fournir quelques options de configuration spécifiques. Dans le bean `velocityConfig`, nous demandons à Velocity de rechercher toutes les modèles présents dans le répertoire `/WEB-INF/vm`.
- ❹ Enfin, le `viewResolver` est configuré avec la classe `VelocityViewResolver`. Il existe un bon nombre d'implémentations du `ViewResolver` dans Spring, du standard `ViewResolver` pour afficher une JSP ou JSTL au `ViewResolver` capable d'effectuer le rendu de modèles Freemarker. Dans cet exemple, nous configurons le moteur de template Velocity et modifions les préfixes et suffixes par défaut pour modifier automatiquement les noms des modèles passés aux objets `ModelAndView`.

Pour finir, le projet `simple-webapp` possède un fichier `web.xml` qui fournit la configuration de base à l'application web. Le fichier `web.xml` est affiché dans l'Exemple 7.18, « `web.xml` du module `simple-webapp` ».

Exemple 7.18. `web.xml` du module `simple-webapp`

```

<web-app id="simple-webapp" 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>Simple Web Application</display-name>

  <context-param> ❶
    <param-name>contextConfigLocation</param-name>
    <param-value>

```



```

    classpath:applicationContext-weather.xml
    classpath:applicationContext-persist.xml
  </param-value>
</context-param>

<context-param> ❷
  <param-name>log4jConfigLocation</param-name>
  <param-value>/WEB-INF/log4j.properties</param-value>
</context-param>

<listener> ❸
  <listener-class>
    org.springframework.web.util.Log4jConfigListener
  </listener-class>
</listener>

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

<servlet> ❺
  <servlet-name>weather</servlet-name>
  <servlet-class>
    org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping> ❻
  <servlet-name>weather</servlet-name>
  <url-pattern>*.x</url-pattern>
</servlet-mapping>
</web-app>

```

- ❶ Voici un peu de magie qui nous permet la réutilisation des fichiers `applicationContext-weather.xml` et `applicationContext-persist.xml` dans ce projet. Le `contextConfigLocation` est utilisé par `ContextLoaderListener` pour créer un `ApplicationContext`. Quand la servlet `weather` est créée, le fichier `weather-servlet.xml` de l'Exemple 7.17, « Configuration des contrôleurs Spring du fichier `weather-servlet.xml` » évalue l'`ApplicationContext` créé à partir du `contextConfigLocation`. Par ce moyen, vous pouvez définir un ensemble de beans dans un autre projet et les référencer par l'intermédiaire du `classpath`. Comme les JARs des modules `simple-persist` et `simple-weather` seront placés dans `WEB-INF/lib`, tout ce que nous avons à faire est d'utiliser le préfixe `classpath:` pour référencer ces fichiers. (Une autre option consisterait à copier ces fichiers dans `/WEB-INF` et de les référencer avec quelque chose ressemblant à `/WEB-INF/applicationContext-persist.xml`).
- ❷ Le paramètre `log4jConfigLocation` est utilisé par le `Log4jConfigListener` pour savoir où chercher le fichier de configuration de logging Log4J. Dans cet exemple, nous demandons à Log4J de regarder dans le fichier `/WEB-INF/log4j.properties`.

- ③ Cela assure que le système Log4J est configuré lors du démarrage de l'application. Il est important de mettre ce `Log4JConfigListener` avant le `ContextLoaderListener`. Dans le cas contraire, vous pourriez manquer d'importants messages indiquant un éventuel problème survenu lors du démarrage d'application. Si vous avez un ensemble particulièrement grand de beans gérés par Spring et que l'un d'entre eux n'arrive pas à s'initialiser au démarrage de l'application, votre application ne se lancera pas. Si vous avez initialisé votre système de log avant le démarrage de Spring, vous aurez la chance de récupérer une alerte ou un message d'erreur. Au contraire, si vous n'avez pas initialisé le système de log avant le démarrage de Spring, préparez vous à naviguer dans le noir pour comprendre pourquoi votre application refuse de démarrer.
- ④ Le `ContextLoaderListener` est essentiel pour le conteneur Spring. Quand l'application démarre, ce listener construit un `ApplicationContext` grâce au paramètre `contextConfigLocation`.
- ⑤ Nous définissons un `DispatcherServlet` de Spring MVC nommé `weather`. Cela forcera Spring à regarder dans le fichier `/WEB-INF/weather-servlet.xml`. Vous pouvez avoir autant de `DispatcherServlet` que vous le désirez, une `DispatcherServlet` peut contenir un ou plusieurs `Controller` Spring MVC.
- ⑥ Toutes les requêtes terminant par `.x` seront routées vers la servlet `weather`. Notez que l'extension `.x` n'a pas de signification particulière, c'est un choix arbitraire, il vous est possible de choisir n'importe quel format pour vos URLs.

7.7. Exécution de l'application web

Pour exécuter l'application web, vous devez tout d'abord construire votre projet multimodule dans son intégralité et ensuite construire la base de données en utilisant le plugin `Hibernate3`. D'abord, exécutez la commande `mvn clean install` à partir du répertoire du projet de plus haut niveau `simple-parent` :

```
$ mvn clean install
```

L'exécution de la commande `mvn clean install` dans le répertoire de plus haut niveau de votre projet installera tous ces modules dans votre dépôt local. Vous devez faire ceci avant de construire la base de données du projet `simple-webapp`. Pour construire la base de données à partir du projet `simple-webapp`, exécutez la commande suivante à partir du répertoire du projet `simple-webapp` :

```
$ mvn hibernate3:hbm2ddl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hibernate3'.
[INFO] org.codehaus.mojo: checking for updates from central
[INFO] -----
[INFO] Building Multi-Spring Chapter Simple Web Application
[INFO]   task-segment: [hibernate3:hbm2ddl]
[INFO] -----
[INFO] Preparing hibernate3:hbm2ddl
...
10:24:56,151 INFO org.hibernate.tool.hbm2ddl.SchemaExport - export complete
[INFO] -----
[INFO] BUILD SUCCESSFUL
```

```
[INFO] -----
```

Une fois cela fait, vous devriez avoir un répertoire `${basedir}/data` qui contient la base de données HSQLDB. Vous pouvez démarrer l'application web avec la commande suivante :

```
$ mvn jetty:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'jetty'.
[INFO] -----
[INFO] Building Multi-Spring Chapter Simple Web Application
[INFO]    task-segment: [jetty:run]
[INFO] -----
[INFO] Preparing jetty:run
...
[INFO] [jetty:run]
[INFO] Configuring Jetty for project:
Multi-Spring Chapter Simple Web Application
...
[INFO] Context path = /simple-webapp
[INFO] Tmp directory = determined at runtime
[INFO] Web defaults = org/mortbay/jetty/webapp/webdefault.xml
[INFO] Web overrides = none
[INFO] Starting jetty 6.1.7 ...
2008-03-25 10:28:03.639::INFO: jetty-6.1.7
...
2147 INFO  DispatcherServlet - FrameworkServlet 'weather': \
        initialization completed in 1654 ms
2008-03-25 10:28:06.341::INFO: Started SelectChannelConnector@0.0.0.0:8080
[INFO] Started Jetty Server
```

Une fois que Jetty est démarré, vous pouvez lancer votre navigateur et ouvrir la page <http://localhost:8080/simple-webapp/weather.x?zip=60202>. Vous devriez y voir les prévisions météorologiques d'Evanston (Illinois). Modifiez le code postal pour obtenir votre propre rapport de prévisions.

```
Current Weather Conditions for: Evanston, IL, US

* Temperature: 42
* Condition: Partly Cloudy
* Humidity: 55
* Wind Chill: 34
* Date: Tue Mar 25 10:29:45 CDT 2008
```

7.8. Le module de ligne de commande

Le projet `simple-command` est la version ligne de commande du projet `simple-webapp`. Cet utilitaire possède donc les mêmes dépendances : `simple-persist` et `simple-weather`. Au lieu d'interagir avec l'application par l'intermédiaire d'un navigateur web, vous pourrez exécuter cet utilitaire à partir de la ligne de commande.

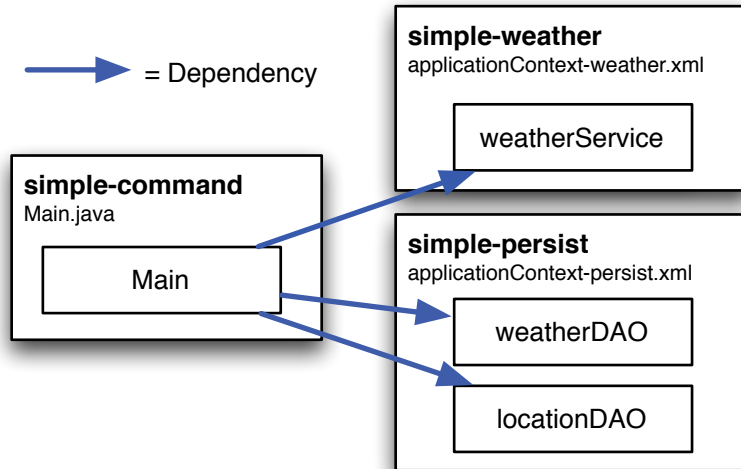


Figure 7.4. L'application en ligne de commande référence simple-weather et simple-persist

Exemple 7.19. POM du module simple-command

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.multispring</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-command</artifactId>
  <packaging>jar</packaging>
  <name>Simple Command Line Tool</name>

  <build>
    <finalName>${project.artifactId}</finalName>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
```

```

    <artifactId>maven-surefire-plugin</artifactId>
    <configuration>
      <testFailureIgnore>>true</testFailureIgnore>
    </configuration>
  </plugin>
  <plugin>
    <artifactId>maven-assembly-plugin</artifactId>
    <configuration>
      <descriptorRefs>
        <descriptorRef>jar-with-dependencies</descriptorRef>
      </descriptorRefs>
    </configuration>
  </plugin>
  <plugin>
    <groupId>org.codehaus.mojo</groupId>
    <artifactId>hibernate3-maven-plugin</artifactId>
    <version>2.1</version>
    <configuration>
      <components>
        <component>
          <name>hbm2ddl</name>
          <implementation>annotationconfiguration</implementation>
        </component>
      </components>
    </configuration>
    <dependencies>
      <dependency>
        <groupId>hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>1.8.0.7</version>
      </dependency>
    </dependencies>
  </plugin>
</plugins>
</build>

<dependencies>
  <dependency>
    <groupId>org.sonatype.mavenbook.multispring</groupId>
    <artifactId>simple-weather</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>org.sonatype.mavenbook.multispring</groupId>
    <artifactId>simple-persist</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
    <version>2.0.7</version>
  </dependency>
</dependencies>

```

```

    <groupId>hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>1.8.0.7</version>
  </dependency>
</dependencies>
</project>

```

Ce POM crée un fichier JAR qui contient la classe `org.sonatype.mavenbook.weather.Main` présentée dans l'Exemple 7.20, « La classe Main du module simple-command ». Dans ce POM nous configurons le plugin Maven Assembly pour utiliser le descripteur d'assemblage intégré appelé `jar-with-dependencies` qui crée un JAR contenant tout le bytecode nécessaire au projet pour s'exécuter : le bytecode du projet et celui de ses dépendances.

Exemple 7.20. La classe Main du module simple-command

```

package org.sonatype.mavenbook.weather;

import java.util.List;

import org.apache.log4j.PropertyConfigurator;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;
import org.sonatype.mavenbook.weather.persist.LocationDAO;
import org.sonatype.mavenbook.weather.persist.WeatherDAO;

public class Main {

    private WeatherService weatherService;
    private WeatherDAO weatherDAO;
    private LocationDAO locationDAO;

    public static void main(String[] args) throws Exception {
        // Configuration de Log4J
        PropertyConfigurator.configure(Main.class.getClassLoader().getResource(
            "log4j.properties"));

        // Lecture du code postal à partir de la ligne de commande
        String zipcode = "60202";
        try {
            zipcode = args[0];
        } catch (Exception e) {
        }

        // Lecture de l'opération à partir de la ligne de commande
        String operation = "weather";
        try {
            operation = args[1];
        } catch (Exception e) {
        }
    }
}

```

```

// Démarrage du programme
Main main = new Main(zipcode);

ApplicationContext context =
    new ClassPathXmlApplicationContext(
        new String[] { "classpath:applicationContext-weather.xml",
                       "classpath:applicationContext-persist.xml" });
main.weatherService = (WeatherService) context.getBean("weatherService");
main.locationDAO = (LocationDAO) context.getBean("locationDAO");
main.weatherDAO = (WeatherDAO) context.getBean("weatherDAO");
if( operation.equals("weather") ) {
    main.getWeather();
} else {
    main.getHistory();
}

private String zip;

public Main(String zip) {
    this.zip = zip;
}

public void getWeather() throws Exception {
    Weather weather = weatherService.retrieveForecast(zip);
    weatherDAO.save( weather );
    System.out.print(new WeatherFormatter().formatWeather(weather));
}

public void getHistory() throws Exception {
    Location location = locationDAO.findByZip(zip);
    List<Weather> weathers = weatherDAO.recentForLocation(location);
    System.out.print(new WeatherFormatter().formatHistory(location, weathers));
}
}

```

La classe Main possède des références vers WeatherDAO, LocationDAO et WeatherService. La méthode statique main() de cette classe :

- Lit le code postal passé en premier argument de la ligne de commande
- Lit l'opération à effectuer. Il s'agit du second argument à passer à la ligne de commande. Si l'opération est "weather", la dernière prévision sera récupérée à partir du service web. Si l'opération est "history", le programme récupérera l'historique des prévisions à partir de la base de données.
- Charge l'ApplicationContext Spring en utilisant deux fichiers XML provenant des modules simple-persist et simple-weather
- Crée une instance de la classe Main
- Récupère les propriétés weatherService, weatherDAO, et locationDAO à partir des beans Spring de l'ApplicationContext

- Appelle la méthode appropriée `getWeather()` ou `getHistory()` en fonction de l'opération demandée.

Dans l'application web, nous utilisons le `VelocityViewResolver` proposé par Spring pour effectuer le rendu d'un template Velocity. Dans l'implémentation en ligne de commande, nous avons besoin d'écrire manuellement une classe qui permet d'afficher les données météorologiques à partir d'un template Velocity. L'Exemple 7.21, « `WeatherFormatter` affiche les prévisions météo en utilisant un template Velocity » affiche la classe `WeatherFormatter`. Cette classe possède donc deux méthodes qui permettent d'afficher respectivement les prévisions météorologiques et leur historique.

Exemple 7.21. `WeatherFormatter` affiche les prévisions météo en utilisant un template Velocity

```
package org.sonatype.mavenbook.weather;

import java.io.InputStreamReader;
import java.io.Reader;
import java.io.StringWriter;
import java.util.List;

import org.apache.log4j.Logger;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

import org.sonatype.mavenbook.weather.model.Location;
import org.sonatype.mavenbook.weather.model.Weather;

public class WeatherFormatter {

    private static Logger log = Logger.getLogger(WeatherFormatter.class);

    public String formatWeather( Weather weather ) throws Exception {
        log.info( "Formatting Weather Data" );
        Reader reader =
            new InputStreamReader( getClass().getClassLoader().
                getResourceAsStream("weather.vm"));
        VelocityContext context = new VelocityContext();
        context.put("weather", weather );
        StringWriter writer = new StringWriter();
        Velocity.evaluate(context, writer, "", reader);
        return writer.toString();
    }

    public String formatHistory( Location location, List<Weather> weathers )
        throws Exception {
        log.info( "Formatting History Data" );
        Reader reader =
            new InputStreamReader( getClass().getClassLoader().
                getResourceAsStream("history.vm"));
        VelocityContext context = new VelocityContext();
        context.put("location", location );
        context.put("weathers", weathers );
        StringWriter writer = new StringWriter();
```



```

Velocity.evaluate(context, writer, "", reader);
return writer.toString();
}
}

```

Le template `weather.vm` affiche le code postal, la ville, le pays et les prévisions de température. Le template `history.vm` affiche le lieu et itère sur la liste des prévisions stockées dans la base de données. Ces deux templates se trouvent dans le dossier `${basedir}/src/main/resources`.

Exemple 7.22. Le template Velocity `weather.vm`

```

*****
Current Weather Conditions for:
  ${weather.location.city},
  ${weather.location.region},
  ${weather.location.country}
*****

* Temperature: ${weather.condition.temp}
* Condition: ${weather.condition.text}
* Humidity: ${weather.atmosphere.humidity}
* Wind Chill: ${weather.wind.chill}
* Date: ${weather.date}

```

Exemple 7.23. Le template Velocity `history.vm`

```

Weather History for:
${location.city},
${location.region},
${location.country}

#foreach( $weather in $weathers )
*****
* Temperature: $weather.condition.temp
* Condition: $weather.condition.text
* Humidity: $weather.atmosphere.humidity
* Wind Chill: $weather.wind.chill
* Date: $weather.date
#end

```

7.9. Exécution de l'application en ligne de commande

Le projet `simple-command` est configuré pour créer un unique JAR contenant le bytecode du projet et celui de toutes ses dépendances. Pour créer cet assemblage, il nous faut exécuter le goal `assembly` du plugin Maven Assembly à partir du répertoire du projet `simple-command` :

```

$ mvn assembly:assembly
[INFO] -----
[INFO] Building Multi-spring Chapter Simple Command Line Tool
[INFO]    task-segment: [assembly:assembly] (aggregator-style)
[INFO] -----

```

```

[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [surefire:test]
...
[INFO] [jar:jar]
[INFO] Building jar: ../simple-parent/simple-command/target/simple-command.jar
[INFO] [assembly:assembly]
[INFO] Processing DependencySet (output=)
[INFO] Building jar: ../simple-parent/simple-command/target
                /simple-command-jar-with-dependencies.jar

```

Le build suit le cycle de vie suivant : compilation, exécution des tests unitaires et construction du JAR. Le goal `assembly:assembly` crée un JAR contenant les dépendances. Pour cela, toutes les dépendances sont dézippées dans un répertoire temporaire, ensuite tout le bytecode est regroupé dans un unique JAR créé dans le répertoire `target/` et nommé `simple-command-jar-with-dependencies.jar`. Ce "super" JAR pèse 15 MO.

Avant d'exécuter notre utilitaire à partir de la ligne de commande, nous devons appeler le goal `hbm2ddl` du plugin `Hibernate3` pour créer la base de données `HSQLDB`. Pour cela, exécutez la commande suivante à partir du répertoire du projet `simple-command` :

```

$ mvn hibernate3:hbm2ddl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'hibernate3'.
[INFO] org.codehaus.mojo: checking for updates from central
[INFO] -----
[INFO] Building Multi-spring Chapter Simple Command Line Tool
[INFO]    task-segment: [hibernate3:hbm2ddl]
[INFO] -----
[INFO] Preparing hibernate3:hbm2ddl
...
10:24:56,151 INFO org.hibernate.tool.hbm2ddl.SchemaExport - export complete
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----

```

Une fois cette commande exécutée, vous devriez voir apparaître un répertoire `data/` dans le dossier `simple-command`. Ce répertoire `data/` contient la base de données `HSQLDB`. Pour exécuter l'outil de prévisions en ligne de commande, lancez la commande suivante à partir du répertoire du projet `simple-command` :

```

$ java -cp target/simple-command-jar-with-dependencies.jar \
    org.sonatype.mavenbook.weather.Main 60202
2321 INFO  YahooRetriever - Retrieving Weather Data
2489 INFO  YahooParser   - Creating XML Reader

```

```

2581 INFO YahooParser - Parsing XML Response
2875 INFO WeatherFormatter - Formatting Weather Data
*****
Current Weather Conditions for:
  Evanston,
  IL,
  US
*****

* Temperature: 75
* Condition: Partly Cloudy
* Humidity: 64
* Wind Chill: 75
* Date: Wed Aug 06 09:35:30 CDT 2008

```

Pour afficher l'historique des prévisions, exécutez la commande suivante :

```

$ java -cp target/simple-command-jar-with-dependencies.jar \
      org.sonatype.mavenbook.weather.Main 60202 history
2470 INFO WeatherFormatter - Formatting History Data
Weather History for:
Evanston, IL, US

*****
* Temperature: 39
* Condition: Heavy Rain
* Humidity: 93
* Wind Chill: 36
* Date: 2007-12-02 13:45:27.187
*****
* Temperature: 75
* Condition: Partly Cloudy
* Humidity: 64
* Wind Chill: 75
* Date: 2008-08-06 09:24:11.725
*****
* Temperature: 75
* Condition: Partly Cloudy
* Humidity: 64
* Wind Chill: 75
* Date: 2008-08-06 09:27:28.475

```

7.10. Conclusion

Nous avons passé beaucoup de temps sur des sujets ne se rapportant pas directement à Maven pour parcourir tout ce chemin. Nous l'avons fait pour présenter un projet exemple aussi complet et significatif que possible pour que vous puissiez l'utiliser pour réaliser de véritables systèmes. Pour arriver à ce résultat, nous n'avons pas eu besoin de prendre de raccourcis, ni de vous émerveiller avec des tours de magie à la Ruby on Rails, ni de vous laisser croire qu'on peut créer une application Java d'entreprise en "10 minutes !". Trop de monde essayent de vous vendre le framework révolutionnaire qui vous demanderait zéro temps d'apprentissage. Ce que nous avons essayé de faire dans ce chapitre est de

présenter le tableau dans son intégralité, l'écosystème complet d'un build multimodule. Nous vous avons présenté Maven dans le contexte d'une application qui ressemble à ce que vous pourriez voir dans la vraie vie, et non 10 minutes de vidéo roulant Apache Ant dans la boue et essayant de vous donner envie d'adopter Apache Maven.

Si vous terminez ce chapitre en vous demandant en quoi il traite de Maven, notre mission est réussie. Nous avons présenté un ensemble complexe de projets en utilisant des frameworks populaires que nous avons assemblés par des build déclaratifs. Le fait que plus de 60% de ce chapitre ait été consacré au fonctionnement de Spring et Hibernate doit montrer que Maven, pour sa plus grosse partie, n'est pas l'étape la plus compliquée d'un projet. Il nous a permis de nous concentrer sur l'application elle-même, et non sur le processus de build. Au lieu de passer du temps à discuter du fonctionnement de Maven et le travail que vous devriez faire pour construire un projet qui intègre Spring et Hibernate, nous avons parlé presque exclusivement des technologies utilisées. Si vous commencez à utiliser Maven et que vous prenez du temps pour l'apprendre, vous commencez vraiment à profiter du fait que vous ne devez pas passer votre temps à écrire vos scripts de build. Vous ne devez pas perdre votre temps à vous inquiéter des aspects classiques de votre build.

Vous pouvez utiliser le squelette du projet de ce chapitre comme fondation pour votre propre projet. Fondation à laquelle vous ajouterez des modules au fur et à mesure de vos besoins. Par exemple, le véritable projet dont ce chapitre est tiré comporte deux projets modèles, deux modules de persistance qui permettent de stocker des objets dans différents type de base de données, plusieurs applications web et une application Java mobile. Au total, le système du monde réel duquel j'ai tiré cet exemple contient au moins 15 modules interdépendants. Vous avez vu l'exemple de projet multimodule le plus complet que nous allons présenter dans ce livre, notez cependant que cet exemple se contente d'aborder qu'une infime partie de ce qu'il est possible de faire avec Maven.

7.10.1. Programmation avec des projets API

Ce chapitre a exploré un projet multimodule qui était plus complexe que l'exemple présenté dans le Chapitre 6, Un projet multimodule, pourtant il s'agit encore d'une simplification par rapport à un vrai projet. Dans un plus grand projet, vous pourriez construire un système ressemblant à la Figure 7.5, « Programmation avec des projets API ».

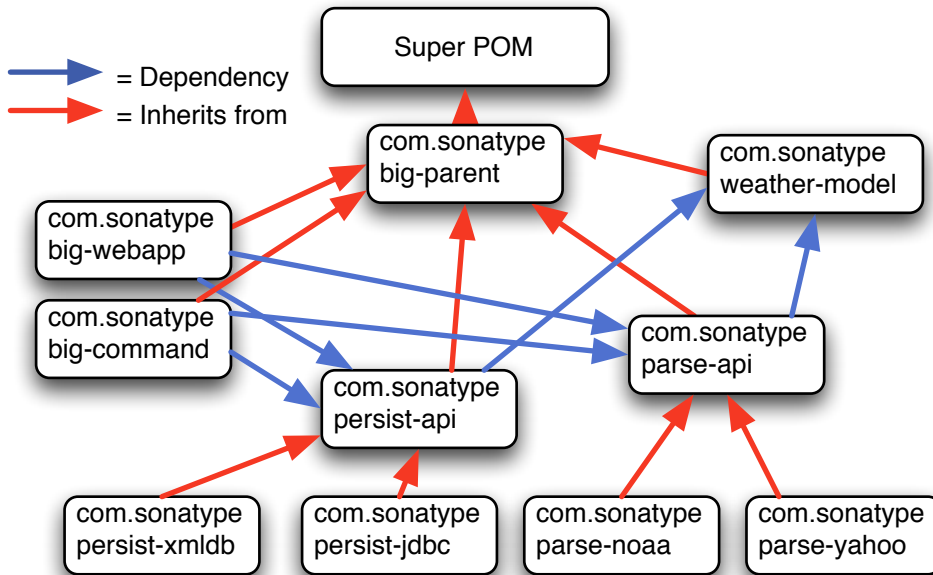


Figure 7.5. Programmation avec des projets API

Quand nous utilisons le terme projet API, nous faisons allusion à un projet Maven qui contient uniquement des interfaces et des constantes. Sur la Figure 7.5, « Programmation avec des projets API », deux des projets sont des projets API : `persist-api` et `parse-api`. Si les projets `big-command` et `big-webapp` utilisent les interfaces définies dans `persist-api`, il sera facile de changer d'implémentations et de framework de persistance. Ce diagramme en particulier montre deux implémentations du projet `persist-api` : une qui stocke ses données dans une base de données XML et l'autre qui stocke ses données dans une base de données relationnelle. Si vous décidez de mettre en place certains des concepts de ce chapitre, vous pourriez réfléchir à comment passer un signal au programme pour permettre de changer d'`ApplicationContext` Spring et changer ainsi de base de données à la volée. Comme pour la conception OO de l'application elle-même, il est souvent prudent de séparer les interfaces d'une API de leurs implémentations en utilisant des projets Maven séparés.

Chapitre 8. Optimiser et remanier les POMs

8.1. Introduction

Dans le Chapitre 7, Un projet multimodule d'entreprise, nous avons vu combien d'éléments de Maven doivent intervenir de concert pour produire un build multimodule complètement fonctionnel. Bien que l'exemple du chapitre précédent représente une véritable application — une application qui interagit avec une base de données, un web service, et qui elle-même présente deux interfaces : une au travers d'une application web, et l'autre via la ligne de commande — cet exemple de projet reste convenu. Présenter la complexité d'un véritable projet demanderait un livre bien plus épais que celui que vous avez entre les mains. Dans la réalité les applications évoluent année après année et sont souvent maintenues par des équipes de développeurs nombreuses et variées, chacune se concentrant sur sa propre partie. Dans un véritable projet, vous devez souvent évaluer les décisions et la conception faites par d'autres. Dans ce chapitre, nous allons prendre du recul par rapport aux exemples que vous avez vus dans la Partie I, « Maven par l'exemple », et nous allons nous demander quelles sont les optimisations à réaliser avec ce que nous avons appris sur Maven. Maven est un outil très adaptable qui peut devenir simple ou complexe selon votre besoin. C'est pour cette raison qu'il existe des centaines de manières différentes de configurer votre projet Maven pour réaliser une même tâche mais aucune qui ne saurait prétendre être la “bonne”.

N'allez pas voir dans cette dernière phrase une autorisation à détourner à Maven pour lui faire réaliser des choses pour lesquelles il n'a pas été conçu. Même si Maven permet des approches diverses et variées, il existe sûrement une approche "à la Maven" qui utilise Maven comme il a été conçu pour être utilisé et vous rend ainsi plus efficace. Dans ce chapitre, nous allons vous montrer certaines optimisations que vous pouvez appliquer à un projet Maven existant. Pourquoi n'avons nous pas commencé avec un POM optimisé en premier lieu ? Concevoir des POMs avec une valeur pédagogique est bien différent de la conception de POMs efficaces. S'il est sûrement plus facile de définir certaines valeurs dans votre `~/m2/settings.xml` que de déclarer un profil dans un `pom.xml`, l'écriture d'un livre, comme sa lecture, dépend de la fréquence à laquelle on va introduire de nouveaux concepts ainsi que du moment où ils seront introduits, ni trop tôt ni trop tard. Dans la Partie I, « Maven par l'exemple », nous avons fait attention à ne pas vous noyer sous trop d'informations. Et ce faisant, nous avons dû éviter certains concepts de base comme la balise `dependencyManagement` dont nous parlerons dans ce chapitre.

À plusieurs moments dans la Partie I, « Maven par l'exemple » les auteurs de ce livre ont choisi des raccourcis ou ont évité un point de détail important pour rester concentrés sur l'essentiel du chapitre. Vous avez appris à créer un projet Maven, vous l'avez compilé et installé sans avoir à parcourir des centaines de pages qui vous décrivaient toutes les options et possibilités. Nous avons procédé ainsi car nous pensons qu'il est important pour un nouvel utilisateur de Maven d'obtenir rapidement des résultats plutôt que de suivre une très longue, voire interminable histoire. Une fois que vous avez commencé à utiliser Maven, vous devriez savoir comment analyser vos propres projets et POMs. Dans

ce chapitre, nous prenons du recul pour étudier ce qu'il nous reste après l'exemple du Chapitre 7, Un projet multimodule d'entreprise.

8.2. Nettoyer le POM

L'optimisation du POM d'un projet multimodules est plus facile à faire en plusieurs passes, car il y a plusieurs points à traiter. En général, nous recherchons des répétitions dans un POM et ses POMs frères. Quand vous commencez un projet ou que celui-ci évolue fréquemment, il est tout à fait acceptable de dupliquer des dépendances et des configurations de plugins. Mais au fur et à mesure que votre projet se stabilise et que le nombre de sous-modules augmente, vous devrez prendre le temps de remanier ces éléments communs de configuration et de dépendances. Rendre vos POMs plus efficaces vous aidera énormément à gérer la complexité de votre projet durant sa croissance. À chaque fois qu'il y a duplication d'information, il existe une meilleure manière de faire.

8.3. Optimiser les dépendances

Si vous jetez un oeil aux différents POMs créés dans le Chapitre 7, Un projet multimodule d'entreprise, vous remarquerez plusieurs types de répétition. Le premier type que vous pouvez voir est la duplication de dépendances comme `spring` et `hibernate-annotations` dans plusieurs modules. La dépendance `hibernate` a en outre l'exclusion de `javax.transaction` à chacune de ses définitions. Le second type de répétition rencontré vient du fait que parfois plusieurs dépendances sont liées entre elles et partagent la même version. C'est souvent le cas lorsqu'un projet livre plusieurs composants couplés entre eux. Par exemple, regardez les dépendances `hibernate-annotations` et `hibernate-commons-annotations`. Toutes les deux ont la même version `3.3.0.ga`, et nous pouvons nous attendre à ce que les versions de ces deux dépendances évoluent de concert. Les deux artefacts `hibernate-annotations` et `hibernate-commons-annotations` sont des composants du même projet mis à disposition par JBoss, et donc quand une nouvelle version de ce projet sort, ces deux dépendances changent. Le troisième et dernier type de répétition est la duplication de dépendances et de version de modules frères. Maven fournit un mécanisme simple pour vous permettre de factoriser ces duplications dans un POM parent.

Comme pour le code source de votre projet, chaque duplication dans vos POMs, est une porte ouverte pour de futurs problèmes. La cohérence des versions sur un gros projet sera difficile à assurer si des déclarations de dépendance sont dupliquées. Tant que vous avez deux ou trois modules, cela reste gérable, mais lorsque votre organisation utilise un énorme build multimodule pour gérer des centaines de composants produits par plusieurs services, une seule erreur de dépendance peut entraîner chaos et confusion. Une simple erreur de version sur une dépendance permettant la manipulation de bytecode comme ASM dans les profondeurs des dépendances du projet peut devenir le grains de sable qui gripperait une application web, maintenue par une autre équipe de développeurs, et qui dépendrait de ce module. Les tests unitaires pourraient passer car ils sont exécutés avec une certaine version de la dépendance, mais ils échoueraient lamentablement en production là où le package (un WAR, dans ce cas) serait réalisé avec une version différente. Si vous avez des dizaines de projets qui utilisent une même dépendance comme Hibernate Annotations, chaque recopie et duplication des dépendances et des

exclusions vous rapproche du moment où un build échouera. Au fur et à mesure que la complexité de vos projets Maven augmente, la liste de vos dépendances s'allonge et vous allez donc devoir stabiliser les déclarations de dépendance et de version dans des POMs parent.

La duplication des versions des modules frères peut produire un problème assez redoutable qui ne résulte pas directement de Maven, et dont on ne se souvient qu'après l'avoir rencontré plusieurs fois. Si vous utilisez le plugin Maven Release pour effectuer vos livraisons, toutes les versions de dépendance soeurs seront automatiquement mises à jour pour vous, aussi ce n'est pas là que réside le problème. Si `simple-web` version `1.3-SNAPSHOT` dépend de `simple-persist` version `1.3-SNAPSHOT`, et si vous produisez la version 1.3 de ces deux projets, le plugin Maven Release est suffisamment intelligent pour changer les versions dans les POMs de votre projet multimodule automatiquement. Produire la livraison avec le plugin Release va automatiquement incrémenter les versions de votre build à `1.4-SNAPSHOT`, et le plugin Release va commiter ces modifications sur le dépôt de source. Livrer un énorme projet multimodule ne pourrait être plus facile, à moins que...

Les problèmes arrivent lorsque les développeurs fusionnent les modifications du POM et perturbent une livraison en cours. Un développeur fusionne souvent des modifications et parfois il se trompe lors de la gestion du conflit sur la dépendance sur un module frère, revenant par inadvertance à la version de la livraison précédente. Comme les versions consécutives d'une dépendance sont souvent compatibles, cela n'apparaît pas lorsque le développeur lance le build, ni avec un système d'intégration continue. Imaginez un build très complexe où le tronc est rempli de composants à la version `1.4-SNAPSHOT`, et maintenant imaginez que le Développeur A a mis à jour le Composant A tout au fond de la hiérarchie du projet pour qu'il dépende de la version `1.3-SNAPSHOT` du Composant B. Même si la plupart des développeurs utilisent la version `1.4-SNAPSHOT`, le build fonctionne correctement si les versions `1.3-SNAPSHOT` et `1.4-SNAPSHOT` du Composant B sont compatibles. Maven continuera à construire le projet en utilisant la version `1.3-SNAPSHOT` du Composant B depuis le dépôt local des développeurs. Tout semble bien se passer — le projet est construit, l'intégration continue est au vert, etc. Quelqu'un peut alors rencontrer un bug étrange en rapport avec le Composant B, mais il va se dire que c'est la faute à "pas de chance" et va poursuivre son développement. Pendant ce temps, dans la salle des machines la pression monte, jusqu'à ce qu'une des pièces explose ...

Quelqu'un, appelons le Mr. Distract, a rencontré un conflit lors de la fusion du Composant A, et a malencontreusement indiqué que le Composant A dépend de la version `1.3-SNAPSHOT` du Composant B alors que le projet continuait sa marche en avant. Une équipe de développeurs essaye de corriger un bug dans le Composant B depuis tout ce temps et ils ne comprennent pas pourquoi ils n'arrivent pas à le corriger en production. Finalement, quelqu'un regarde le Composant A et s'aperçoit de cette dépendance sur une mauvaise version. Heureusement, ce bug n'était pas suffisamment important pour coûter de l'argent ou des vies, mais Monsieur Distract se sent un peu bête et les autres ont perdu un peu de leur confiance en lui pour ce problème de dépendances entre composants. (Heureusement, Monsieur Distract se rend compte que ce problème est une erreur d'utilisateur et ne vient pas de Maven, mais il est plus que probable qu'il commence un vilain blog dans lequel il se plaint de Maven sans arrêt pour se sentir mieux.)

Heureusement, la duplication de dépendance et les erreurs de dépendance entre projets frères peuvent être facilement évitées avec quelques modifications. La première chose à faire est de trouver toutes les dépendances utilisées sur plus d'un projet et de les déplacer dans la section `dependencyManagement` du POM parent. Nous allons laisser de côté les dépendances entre modules frères pour l'instant. Le pom `simple-parent` contient maintenant :

```
<project>
...
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
      <version>2.0.7</version>
    </dependency>
    <dependency>
      <groupId>org.apache.velocity</groupId>
      <artifactId>velocity</artifactId>
      <version>1.5</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
      <version>3.3.0.ga</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>3.2.5.ga</version>
      <exclusions>
        <exclusion>
          <groupId>javax.transaction</groupId>
          <artifactId>jta</artifactId>
        </exclusion>
      </exclusions>
    </dependency>
  </dependencies>
</dependencyManagement>
...
</project>
```

Une fois ces dépendances remontées, nous allons devoir supprimer les versions de ces dépendances de tous les POMs ; sinon, elles vont surcharger ce qui est défini dans la balise `dependencyManagement` du projet parent. Regardons juste le `simple-model` pour plus de clarté :

```
<project>
...
```

```

<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-annotations</artifactId>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
  </dependency>
</dependencies>
...
</project>

```

La seconde chose à faire est de corriger la duplication des versions de `hibernate-annotations` et `hibernate-commons-annotations` puisqu'elles devraient rester identiques. Nous allons faire cela en créant une propriété appelée `hibernate.annotations.version`. La section `simple-parent` résultante ressemble à ceci :

```

<project>
  ...
  <properties>
    <hibernate.annotations.version>3.3.0.ga</hibernate.annotations.version>
  </properties>

  <dependencyManagement>
    ...
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>${hibernate.annotations.version}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
      <version>${hibernate.annotations.version}</version>
    </dependency>
    ...
  </dependencyManagement>
  ...
</project>

```

Notre dernier problème à corriger est celui des dépendances entre modules frères. Une technique que nous pourrions utiliser est de les déplacer dans la section `dependencyManagement`, comme toutes les autres, et de définir les versions des projets frères dans le projet parent qui les contient. Cette approche est certainement valide, mais nous pouvons aussi résoudre ce problème de version en utilisant deux propriétés prédéfinies — `${project.groupId}` et `${project.version}`. Puisqu'il s'agit de dépendances entre frères, il n'y a pas grand-chose à gagner à les lister dans le parent, aussi nous allons faire confiance à la propriété prédéfinie `${project.version}`. Comme ces projets font tous partie du même groupe, nous pouvons sécuriser d'avantage ces déclarations en faisant référence au groupe du

POM courant via la propriété prédéfinie `${project.groupId}`. La section `dependency` de `simple-command` ressemble maintenant à cela :

```
<project>
...
<dependencies>
...
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>simple-weather</artifactId>
  <version>${project.version}</version>
</dependency>
<dependency>
  <groupId>${project.groupId}</groupId>
  <artifactId>simple-persist</artifactId>
  <version>${project.version}</version>
</dependency>
...
</dependencies>
...
</project>
```

Voici un résumé des deux optimisations que nous avons réalisées pour réduire ce problème de duplication des dépendances :

Remonter les dépendances communes dans `dependencyManagement`

Si plus d'un projet dépend d'une dépendance particulière, vous pouvez ajouter celle-ci à `dependencyManagement`. Le POM parent peut contenir une version et un ensemble d'exclusions ; tout ce que les POMs fils ont besoin de faire pour référencer cette dépendance est d'utiliser le `groupId` et l'`artifactId`. Les projets fils peuvent ne pas déclarer la version et les exclusions si la dépendance fait partie des dépendances de la balise `dependencyManagement`.

Utiliser la `version` et le `groupId` pré-définis pour les projets frères

Utiliser `${project.version}` et `${project.groupId}` pour faire référence à un projet frère. Les projets frères ont la plupart du temps le même `groupId`, et presque toujours la même version. Utiliser `${project.version}` vous aidera à éviter les incohérences de versions entre projets frères comme nous l'avons vu plus tôt.

8.4. Optimiser les plugins

Si nous regardons les configurations des différents plugins, nous pouvons remarquer la réplication en plusieurs endroits des dépendances HSQLDB. Malheureusement, `dependencyManagement` ne s'applique pas aux dépendances des plugins, mais nous pouvons tout de même utiliser une propriété pour uniformiser les versions. Les projets Maven multimodule complexes ont tendance à définir toutes les versions dans le POM de plus haut niveau. Ce POM de plus haut niveau devient le point central pour les modifications qui impactent tout le projet. Voyez les numéros de versions comme des chaînes de caractères dans une classe Java. Si vous répétez constamment une phrase, vous allez probablement en faire une variable afin de n'avoir à la modifier qu'à un seul endroit lorsque vous devrez la changer.

Remonter la version de HSQLDB dans une propriété du POM de plus haut niveau se fait au travers de la balise XML `properties` :

```
<project>
...
<properties>
  <hibernate.annotations.version>3.3.0.ga</hibernate.annotations.version>
  <hsqldb.version>1.8.0.7</hsqldb.version>
</properties>
...
</project>
```

Nous pouvons noter que la configuration de `hibernate3-maven-plugin` est dupliquée dans les modules `simple-webapp` et `simple-command`. Il est possible de gérer la configuration des plugins dans le POM de plus haut niveau de la même manière que pour la gestion des dépendances avec la section `dependencyManagement` de ce POM. Pour ce faire, nous allons utiliser la balise XML `pluginManagement` sous la balise XML `build` du POM de plus haut niveau :

```
<project>
...
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.5</source>
          <target>1.5</target>
        </configuration>
      </plugin>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <components>
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation>
            </component>
          </components>
        </configuration>
        <dependencies>
          <dependency>
            <groupId>hsqldb</groupId>
            <artifactId>hsqldb</artifactId>
            <version>${hsqldb.version}</version>
          </dependency>
        </dependencies>
      </plugin>
    </plugins>
  </pluginManagement>
```

```
</build>
...
</project>
```

8.5. Optimisation avec le plugin Maven Dependency

Sur des projets plus importants, des dépendances ont souvent tendance à se glisser dans un POM au fur et à mesure que leur nombre augmente. Lorsque les dépendances changent, vous vous retrouvez avec des dépendances inutilisées ou avec certaines des bibliothèques nécessaires qui n'ont pas été déclarées explicitement. Comme Maven 2.x inclut les dépendances transitives du scope compile, votre projet peut compiler correctement mais ne pas fonctionner en production. Prenons par exemple le cas d'un projet qui utilise les classes d'une librairie très utilisée comme Jakarta Commons BeanUtils. Au lieu de déclarer une dépendance explicite sur BeanUtils, votre projet dépend d'un projet comme Hibernate qui référence BeanUtils comme dépendance transitive. Votre projet peut donc compiler et s'exécuter correctement. Mais si vous mettez à jour votre version d'Hibernate et que cette nouvelle version ne dépend plus de BeanUtils, vous allez commencer à rencontrer des erreurs de compilation et d'exécution, et il ne sera pas évident de trouver pourquoi votre projet a soudainement arrêté de compiler. De plus, comme vous n'avez pas explicitement défini de version pour cette dépendance, Maven ne peut résoudre correctement un éventuel conflit de version.

Une bonne pratique avec Maven est de toujours déclarer explicitement les dépendances pour les classes auxquelles vous faites référence dans votre code. Si vous importez des classes de Commons BeanUtils, vous devriez aussitôt déclarer une dépendance directe sur Commons BeanUtils. Heureusement, grâce à l'analyse du bytecode, le plugin Maven Dependency peut vous aider à découvrir les références directes à des dépendances. Prenons les POMs que nous venons d'optimiser et regardons si on peut y trouver des erreurs :

```
$ mvn dependency:analyze
[INFO] Scanning for projects...
[INFO] Reactor build order:
[INFO]   Chapter 8 Simple Parent Project
[INFO]   Chapter 8 Simple Object Model
[INFO]   Chapter 8 Simple Weather API
[INFO]   Chapter 8 Simple Persistence API
[INFO]   Chapter 8 Simple Command Line Tool
[INFO]   Chapter 8 Simple Web Application
[INFO]   Chapter 8 Parent Project
[INFO] Searching repository for plugin with prefix: 'dependency'.
...
[INFO] -----
[INFO] Building Chapter 8 Simple Object Model
[INFO]   task-segment: [dependency:analyze]
[INFO] -----
[INFO] Preparing dependency:analyze
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
```

```

[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [dependency:analyze]
[WARNING] Used undeclared dependencies found:
[WARNING]   javax.persistence:persistence-api:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]   org.hibernate:hibernate-annotations:jar:3.3.0.ga:compile
[WARNING]   org.hibernate:hibernate:jar:3.2.5.ga:compile
[WARNING]   junit:junit:jar:3.8.1:test

...

[INFO] -----
[INFO] Building Chapter 8 Simple Web Application
[INFO]   task-segment: [dependency:analyze]
[INFO] -----
[INFO] Preparing dependency:analyze
[INFO] [resources:resources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:compile]
[INFO] Nothing to compile - all classes are up to date
[INFO] [resources:testResources]
[INFO] Using default encoding to copy filtered resources.
[INFO] [compiler:testCompile]
[INFO] No sources to compile
[INFO] [dependency:analyze]
[WARNING] Used undeclared dependencies found:
[WARNING]   org.sonatype.mavenbook.optimize:simple-model:jar:1.0:compile
[WARNING] Unused declared dependencies found:
[WARNING]   org.apache.velocity:velocity:jar:1.5:compile
[WARNING]   javax.servlet:jstl:jar:1.1.2:compile
[WARNING]   taglibs:standard:jar:1.1.2:compile
[WARNING]   junit:junit:jar:3.8.1:test

```

Dans ces traces tronquées vous pouvez voir le résultat du goal `dependency:analyze`. Ce goal analyse le projet pour voir s'il existe des dépendances indirectes, ou si on fait référence à des dépendances qui ne sont pas déclarées directement. Dans le projet `simple-model`, le plugin `Dependency` indique une “dépendance utilisée mais pas déclarée” sur `javax.persistence:persistence-api`. Pour obtenir plus d'informations, allez dans le répertoire `simple-model` et exécutez le goal `dependency:tree` qui va lister toutes les dépendances du projet, directes et transitives.

```

$ mvn dependency:tree
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'dependency'.
[INFO] -----
[INFO] Building Chapter 8 Simple Object Model
[INFO]   task-segment: [dependency:tree]
[INFO] -----
[INFO] [dependency:tree]
[INFO] org.sonatype.mavenbook.optimize:simple-model:jar:1.0

```

```

[INFO] +- org.hibernate:hibernate-annotations:jar:3.3.0.ga:compile
[INFO] | \- javax.persistence:persistence-api:jar:1.0:compile
[INFO] +- org.hibernate:hibernate:jar:3.2.5.ga:compile
[INFO] | +- net.sf.ehcache:ehcache:jar:1.2.3:compile
[INFO] | +- commons-logging:commons-logging:jar:1.0.4:compile
[INFO] | +- asm:asm-attrs:jar:1.5.3:compile
[INFO] | +- dom4j:dom4j:jar:1.6.1:compile
[INFO] | +- antlr:antlr:jar:2.7.6:compile
[INFO] | +- cglib:cglib:jar:2.1_3:compile
[INFO] | +- asm:asm:jar:1.5.3:compile
[INFO] | \- commons-collections:commons-collections:jar:2.1.1:compile
[INFO] \- junit:junit:jar:3.8.1:test
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----

```

Dans cette trace, nous pouvons voir que la dépendance `persistence-api` est apportée par `hibernate`. Une analyse rapide du code source de ce module va trouver de nombreux imports de `javax.persistence` ce qui confirme que nous faisons directement référence à cette dépendance. La correction la plus simple est d'ajouter une référence directe à cette dépendance. Dans cet exemple, nous indiquons la version de la dépendance dans la section `dependencyManagement` de `simple-parent` car cette dépendance est liée à Hibernate et c'est là qu'on déclare la version d'Hibernate. Un jour va arriver où vous allez vouloir changer la version d'Hibernate pour votre projet. Mettre la version de `persistence-api` à côté de la version d'Hibernate permettra d'éviter d'oublier de la mettre à jour lorsque vous éditez le POM parent pour modifier la version d'Hibernate.

Si vous regardez le résultat du `dependency:analyze` pour le module `simple-web`, vous verrez qu'il faut aussi ajouter une référence directe à `simple-model`. Le code de `simple-webapp` fait directement référence aux objets métier de `simple-model`, et `simple-model` est visible dans `simple-webapp` en tant que dépendance transitive via `simple-persist`. Puisqu'il s'agit d'une dépendance vers un module frère qui partage la même version et le même `groupId`, la dépendance peut être déclarée dans le `pom.xml` de `simple-webapp` avec `${project.groupId}` et `${project.version}`.

Comment fait le plugin Maven Dependency pour trouver ces problèmes ? Comment `dependency:analyze` sait-il à quelles classes et à quelles dépendances le bytecode de votre projet fait référence ? Le plugin Dependency utilise l'outil ASM d'ObjectWeb (<http://asm.objectweb.org/>) pour analyser le bytecode brut. Le plugin Dependency utilise ASM pour parcourir toutes les classes du projet, et lister toutes les classes référencées qui n'en font pas partie. Ensuite, il parcourt toutes les dépendances directes et transitives, et marque les classes trouvées dans les dépendances directes. Toute classe qui n'a pas été trouvée dans les dépendances directes, est recherchée dans les dépendances transitives et ainsi il génère la liste des “dépendances utilisées, mais non déclarées”.

Par contre, la liste des dépendances déclarées mais non utilisées est un peu plus délicate à valider et bien moins utile que les “dépendances utilisées mais non déclarées”. Premièrement, car certaines dépendances ne sont utilisées qu'à l'exécution ou que pour les tests, et donc ne seront pas référencées dans le bytecode. Celles-ci sont assez faciles à voir dans les traces ; par exemple, JUnit apparaît dans cette liste comme on pouvait s'y attendre car elle n'est utilisée que pour les tests unitaires. Vous remarquerez

aussi les dépendances vers Velocity et l'API Servlet pour le module `simple-web`. Là encore, on pouvait s'y attendre, car le projet n'a aucune référence directe aux classes de ces artefacts, même s'ils restent essentiels durant l'exécution.

Soyez prudents lorsque vous supprimez une dépendance déclarée mais inutilisée, à moins que vous n'ayez une très bonne couverture de tests, vous risquez d'avoir des surprises pendant l'exécution. Un problème beaucoup plus dangereux peut apparaître avec l'optimisation du bytecode. Par exemple, le compilateur a le droit de substituer la valeur d'une constante et d'optimiser en supprimant la référence. Supprimer cette dépendance va empêcher la compilation, alors que l'outil indique qu'elle n'est pas utilisée. Les versions futures du plugin Maven Dependency fourniront de meilleures techniques pour détecter ou ignorer ces problèmes.

Vous devriez utiliser `dependency:analyze` régulièrement pour détecter ces erreurs classiques dans vos projets. On peut configurer le build pour qu'il échoue si ce plugin rencontre certaines conditions, et il peut aussi produire un rapport.

8.6. Les POMs finaux

En résumé, tous les fichiers des POMs finaux sont présentés en tant que référence pour ce chapitre. L'Exemple 8.1, « POM final de simple-parent » montre le POM de plus haut niveau, celui de `simple-parent`.

Exemple 8.1. POM final de simple-parent

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.sonatype.mavenbook.optimize</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 8 Simple Parent Project</name>

  <modules>
    <module>simple-command</module>
    <module>simple-model</module>
    <module>simple-weather</module>
    <module>simple-persist</module>
    <module>simple-webapp</module>
  </modules>

  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
```

```

<artifactId>maven-compiler-plugin</artifactId>
<configuration>
  <source>1.5</source>
  <target>1.5</target>
</configuration>
</plugin>
<plugin>
<groupId>org.codehaus.mojo</groupId>
<artifactId>hibernate3-maven-plugin</artifactId>
<version>2.1</version>
<configuration>
  <components>
    <component>
      <name>hbm2ddl</name>
      <implementation>annotationconfiguration</implementation>
    </component>
  </components>
</configuration>
<dependencies>
  <dependency>
    <groupId>hsqldb</groupId>
    <artifactId>hsqldb</artifactId>
    <version>${hsqldb.version}</version>
  </dependency>
</dependencies>
</plugin>
</plugins>
</pluginManagement>
</build>

<properties>
  <hibernate.annotations.version>3.3.0.ga</hibernate.annotations.version>
  <hsqldb.version>1.8.0.7</hsqldb.version>
</properties>
<dependencyManagement>
<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
    <version>2.0.7</version>
  </dependency>
  <dependency>
    <groupId>org.apache.velocity</groupId>
    <artifactId>velocity</artifactId>
    <version>1.5</version>
  </dependency>
  <dependency>
    <groupId>javax.persistence</groupId>
    <artifactId>persistence-api</artifactId>
    <version>1.0</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>

```

```

    <artifactId>hibernate-annotations</artifactId>
    <version>${hibernate.annotations.version}</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-commons-annotations</artifactId>
    <version>${hibernate.annotations.version}</version>
  </dependency>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.5.ga</version>
    <exclusions>
      <exclusion>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
</dependencyManagement>

<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

Le POM présenté dans l'Exemple 8.2, « POM final de simple-command » correspond au POM pour simple-command, la version utilisable en ligne de commande de l'outil.

Exemple 8.2. POM final de simple-command

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

  <artifactId>simple-command</artifactId>
  <packaging>jar</packaging>
  <name>Chapter 8 Simple Command Line Tool</name>

  <build>

```

```

<pluginManagement>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <mainClass>org.sonatype.mavenbook.weather.Main</mainClass>
            <addClasspath>>true</addClasspath>
          </manifest>
        </archive>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <testFailureIgnore>>true</testFailureIgnore>
      </configuration>
    </plugin>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </plugin>
  </plugins>
</pluginManagement>
</build>

<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>simple-weather</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>simple-persist</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
  </dependency>
  <dependency>
    <groupId>org.apache.velocity</groupId>
    <artifactId>velocity</artifactId>
  </dependency>
</dependencies>

```

```
</project>
```

Le POM présenté dans l'Exemple 8.3, « POM final de simple-model » correspond au POM du projet simple-model. Le projet simple-model contient les objets métiers utilisés tout au long de l'application.

Exemple 8.3. POM final de simple-model

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>
  <artifactId>simple-model</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Object Model</name>

  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
    </dependency>
    <dependency>
      <groupId>javax.persistence</groupId>
      <artifactId>persistence-api</artifactId>
    </dependency>
  </dependencies>
</project>
```

Le POM présenté dans l'Exemple 8.4, « POM final de simple-persist » correspond au POM du projet simple-persist. Le projet simple-persist contient toute la logique de persistance qui est mise en œuvre avec Hibernate.

Exemple 8.4. POM final de simple-persist

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
```

```

    <version>1.0</version>
  </parent>
  <artifactId>simple-persist</artifactId>
  <packaging>jar</packaging>

  <name>Chapter 8 Simple Persistence API</name>

  <dependencies>
    <dependency>
      <groupId>${project.groupId}</groupId>
      <artifactId>simple-model</artifactId>
      <version>${project.version}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-commons-annotations</artifactId>
    </dependency>
    <dependency>
      <groupId>javax.servlet</groupId>
      <artifactId>servlet-api</artifactId>
      <version>2.4</version>
      <scope>provided</scope>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring</artifactId>
    </dependency>
  </dependencies>
</project>

```

Le POM présenté dans l'Exemple 8.5, « POM final de simple-weather » correspond au POM du projet simple-weather. Le projet simple-weather est le projet qui contient toute la logique pour parser le flux RSS de Yahoo! Météo. Ce projet dépend du projet simple-model.

Exemple 8.5. POM final de simple-weather

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>

```

```

</parent>
<artifactId>simple-weather</artifactId>
<packaging>jar</packaging>

<name>Chapter 8 Simple Weather API</name>

<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>simple-model</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>
  <dependency>
    <groupId>dom4j</groupId>
    <artifactId>dom4j</artifactId>
    <version>1.6.1</version>
  </dependency>
  <dependency>
    <groupId>jaxen</groupId>
    <artifactId>jaxen</artifactId>
    <version>1.1.1</version>
  </dependency>
  <dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-io</artifactId>
    <version>1.3.2</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>

```

Enfin, le POM présenté dans l'Exemple 8.6, « POM final de simple-webapp » correspond au POM du projet simple-webapp. Le projet simple-webapp contient une application web qui enregistre les prévisions météo obtenues dans une base de données HSQLDB et interagit avec les bibliothèques produites par le projet simple-weather.

Exemple 8.6. POM final de simple-webapp

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <parent>
    <groupId>org.sonatype.mavenbook.optimize</groupId>
    <artifactId>simple-parent</artifactId>
    <version>1.0</version>
  </parent>

```

```

<artifactId>simple-webapp</artifactId>
<packaging>war</packaging>
<name>Chapter 8 Simple Web Application</name>
<dependencies>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.4</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>simple-model</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>simple-weather</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>simple-persist</artifactId>
    <version>${project.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring</artifactId>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>jstl</artifactId>
    <version>1.1.2</version>
  </dependency>
  <dependency>
    <groupId>>taglibs</groupId>
    <artifactId>standard</artifactId>
    <version>1.1.2</version>
  </dependency>
  <dependency>
    <groupId>org.apache.velocity</groupId>
    <artifactId>velocity</artifactId>
  </dependency>
</dependencies>
<build>
  <finalName>simple-webapp</finalName>
  <plugins>
    <plugin>
      <groupId>org.mortbay.jetty</groupId>
      <artifactId>maven-jetty-plugin</artifactId>
      <version>6.1.9</version>
    </plugin>
  </plugins>
  <dependencies>

```



```
<dependency>
  <groupId>hsqldb</groupId>
  <artifactId>hsqldb</artifactId>
  <version>${hsqldb.version}</version>
</dependency>
</dependencies>
</plugin>
</plugins>
</build>
</project>
```

8.7. Conclusion

Ce chapitre vous a présenté plusieurs techniques pour améliorer votre maîtrise de vos dépendances et de vos plugins afin de faciliter la maintenance de vos builds. Nous vous recommandons de revoir périodiquement vos builds afin de vérifier que les duplications et donc les risques potentiels soient minimisés. Au cours de la vie d'un projet, il est inévitable que de nouvelles dépendances soient ajoutées. Vous pouvez vous retrouver dans une situation où une dépendance qui autrefois n'était utilisée qu'à un seul endroit l'est maintenant en plus de 10, celle-ci devant donc être remontée. Les listes des dépendances utilisées ou inutilisées changent dans le temps et elles peuvent être nettoyées facilement avec le plugin Maven Dependency.

Partie II. Maven - La Reference



Chapitre 9. Le Modèle Objet de Projet

9.1. Introduction

Ce chapitre traite du concept au cœur de Maven — le Project Object Model (Modèle Objet de Projet). C'est dans ce POM que sont déclarées l'identité et la structure d'un projet, que sont configurés les builds et où sont définies les relations entre projets. C'est l'existence d'un fichier `pom.xml` qui définit un projet Maven.

9.2. Le POM

Les projets Maven, les dépendances, les builds, les artefacts : tous sont des objets qu'il va falloir modéliser et décrire. Ces objets sont décrits dans un fichier XML appelé Modèle Objet de Projet. Le POM indique à Maven quel type de projet il va devoir traiter et comment il va devoir s'adapter pour transformer les sources et produire le résultat attendu. Ainsi, comme le fichier `web.xml` décrit, configure et personnalise une application web Java, c'est la présence d'un fichier `pom.xml` qui définit un projet Maven. Il s'agit d'une déclaration décrivant un projet Maven; c'est le “plan” abstrait que Maven doit comprendre et suivre pour construire votre projet.

Vous pouvez aussi voir dans ce fichier `pom.xml` un équivalent à un fichier `Makefile` ou à un fichier `build.xml` pour Ant. Quand vous utilisez GNU make pour construire une application comme par exemple MySQL, vous aurez le plus souvent un fichier `Makefile` contenant toutes les instructions explicites pour produire un binaire à partir des sources. Quand vous utilisez Apache Ant, vous avez très probablement un fichier `build.xml` qui contient les instructions explicites pour nettoyer, compiler, packager et déployer une application. make, Ant, et Maven ont en commun le fait qu'il leur faut un fichier avec un nom pré-déterminé, qu'il s'agisse de `Makefile`, `build.xml`, ou `pom.xml`, mais c'est là que les similarités s'arrêtent. Si vous regardez un `pom.xml` Maven, la plus grande partie du POM se compose de descriptions. Où se trouve le code source ? Où sont les ressources ? Quel est le packaging ? Si vous regardez un fichier `build.xml` pour Ant, vous verrez quelque chose d'entièrement différent. Vous verrez des instructions explicites pour des tâches comme la compilation de classes Java. Le POM Maven est déclaratif, et même si vous pouvez y inclure des personnalisations procédurales via le plugin Maven Ant, en général vous n'aurez pas besoin de vous plonger dans les délicats détails procéduraux de la construction de votre projet.

Le POM n'est pas spécifique à la construction de projets Java. Même si la plupart des exemples de ce livre sont des applications Java, il n'y a rien de spécifique à Java dans la définition d'un Modèle Objet de Projet Maven. Si effectivement les plugins par défaut de Maven permettent de construire des artefacts sous forme de JAR à partir d'un ensemble de sources, de tests et de ressources, rien ne vous empêche de définir un POM pour un projet qui contient des sources C# et produit un binaire Microsoft propriétaire avec des outils Microsoft. De même, rien ne vous empêche d'utiliser un POM pour un livre technique. En effet, le source de ce livre et ses exemples sont répartis dans un projet Maven multimodule qui utilise l'un des nombreux plugins Maven pour Docbook afin d'appliquer une feuille XSL Docbook standard à

un ensemble de chapitres présents sous la forme de fichiers XML. Certains ont créé des plugins Maven pour transformer du code Adobe Flex en SWCs et SWFs ; d'autres utilisent Maven pour construire des projets écrits en C.

Nous avons donc établi que le POM décrit et déclare, et qu'il se différencie de Ant ou de Make en ne fournissant pas d'instructions explicites. Nous avons vu que les concepts du POM ne sont pas propres à Java. Regardons tout cela plus en détail au travers de la Figure 9.1, « Le Modèle Objet de Projet » pour une analyse du contenu d'un POM.

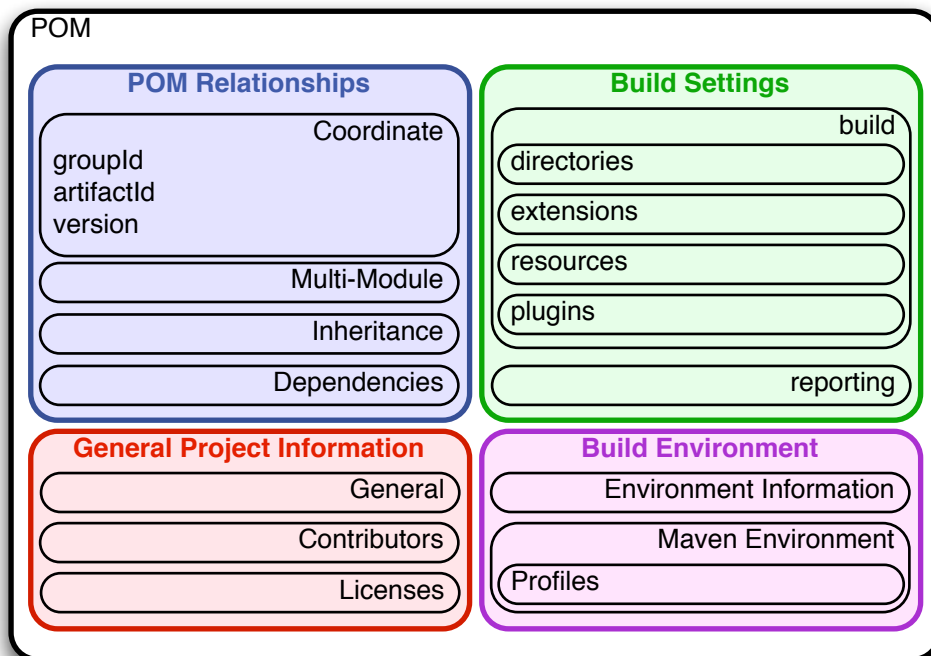


Figure 9.1. Le Modèle Objet de Projet

Le POM se compose de quatre catégories de description et de configuration :

Informations générales sur le projet

Cette catégorie regroupe le nom l'URL et la licence du projet, l'organisation qui produit ce projet, et une liste de développeurs et de contributeurs.

Configuration du build

Dans cette section, nous configurons le build Maven en personnalisant le comportement par défaut. Nous pouvons changer l'endroit où se trouvent les sources et les tests, ajouter de nouveaux plugins, lier des goals de plugins au cycle de vie et personnaliser les paramètres de génération du site web.

Environnement du build

L'environnement du build consiste en un ensemble de profils qui peuvent être activés pour être utilisés dans différents environnements. Par exemple, au cours du développement vous pouvez vouloir déployer sur un serveur qui sera différent de celui sur lequel vous déploierez en production. L'environnement de build adapte la configuration du build pour un environnement spécifique et il s'accompagne souvent d'un fichier `settings.xml` personnalisé dans le répertoire `~/.m2`. Ce fichier `settings.xml` est détaillé dans le Chapitre 11, Profils de Build et dans la Section A.2, « Détails des settings ».

Relations entre POM

Un projet est rarement isolé. Il dépend souvent d'autres projets, hérite d'une configuration de POM de projets parent, définit ses propres coordonnées, et peut comporter des sous-modules.

9.2.1. Le Super POM

Avant de se plonger dans des exemples de POMs, jetons un rapide coup d'œil au Super POM. Tous les POMs de projet Maven étendent le Super POM qui définit un ensemble de valeurs par défaut partagé par tous les projets. Ce Super POM fait partie de l'installation de Maven, et se trouve dans le fichier `maven-2.2.1-uber.jar` du répertoire `${M2_HOME}/lib`. Si vous regardez ce fichier JAR, vous y trouverez un fichier `pom-4.0.0.xml` dans le package `org.apache.maven.project`. Le Super POM de Maven est présenté dans l'Exemple 9.1, « Le Super POM ».

Exemple 9.1. Le Super POM

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <name>Maven Default Project</name>

  <repositories>
    <repository>
      <id>central</id> ❶
      <name>Maven Repository Switchboard</name>
      <layout>default</layout>
      <url>http://repo1.maven.org/maven2</url>
      <snapshots>
        <enabled>>false</enabled>
      </snapshots>
    </repository>
  </repositories>

  <pluginRepositories>
    <pluginRepository>
      <id>central</id> ❷
      <name>Maven Plugin Repository</name>
      <url>http://repo1.maven.org/maven2</url>
      <layout>default</layout>
      <snapshots>
        <enabled>>false</enabled>
      </snapshots>
      <releases>
```

```

    <updatePolicy>never</updatePolicy>
  </releases>
</pluginRepository>
</pluginRepositories>

<build> ❸
  <directory>target</directory>
  <outputDirectory>target/classes</outputDirectory>
  <finalName>${pom.artifactId}-${pom.version}</finalName>
  <testOutputDirectory>target/test-classes</testOutputDirectory>
  <sourceDirectory>src/main/java</sourceDirectory>
  <scriptSourceDirectory>src/main/scripts</scriptSourceDirectory>
  <testSourceDirectory>src/test/java</testSourceDirectory>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
  </resources>
  <testResources>
    <testResource>
      <directory>src/test/resources</directory>
    </testResource>
  </testResources>
</build> ❹
<pluginManagement>
  <plugins>
    <plugin>
      <artifactId>maven-antrun-plugin</artifactId>
      <version>1.1</version>
    </plugin>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.2-beta-1</version>
    </plugin>
    <plugin>
      <artifactId>maven-clean-plugin</artifactId>
      <version>2.2</version>
    </plugin>
    <plugin>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>2.0.2</version>
    </plugin>
    <plugin>
      <artifactId>maven-dependency-plugin</artifactId>
      <version>2.0</version>
    </plugin>
    <plugin>
      <artifactId>maven-deploy-plugin</artifactId>
      <version>2.3</version>
    </plugin>
    <plugin>
      <artifactId>maven-ear-plugin</artifactId>
      <version>2.3.1</version>
    </plugin>
  </plugins>

```



```

<plugin>
  <artifactId>maven-ejb-plugin</artifactId>
  <version>2.1</version>
</plugin>
<plugin>
  <artifactId>maven-install-plugin</artifactId>
  <version>2.2</version>
</plugin>
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <version>2.2</version>
</plugin>
<plugin>
  <artifactId>maven-javadoc-plugin</artifactId>
  <version>2.4</version>
</plugin>
<plugin>
  <artifactId>maven-plugin-plugin</artifactId>
  <version>2.3</version>
</plugin>
<plugin>
  <artifactId>maven-rar-plugin</artifactId>
  <version>2.2</version>
</plugin>
<plugin>
  <artifactId>maven-release-plugin</artifactId>
  <version>2.0-beta-7</version>
</plugin>
<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <version>2.2</version>
</plugin>
<plugin>
  <artifactId>maven-site-plugin</artifactId>
  <version>2.0-beta-6</version>
</plugin>
<plugin>
  <artifactId>maven-source-plugin</artifactId>
  <version>2.0.4</version>
</plugin>
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.4.2</version>
</plugin>
<plugin>
  <artifactId>maven-war-plugin</artifactId>
  <version>2.1-alpha-1</version>
</plugin>
</plugins>
</pluginManagement>

<reporting>
  <outputDirectory>target/site</outputDirectory>
</reporting>

```

```
</project>
```

Le Super POM définit des variables de configuration standards qui seront héritées par tous les projets. Les valeurs de ces variables sont définies dans les sections numérotées :

- ❶ Le Super POM par défaut déclare un unique dépôt Maven distant ayant pour ID `central`. C'est ce dépôt central de Maven qui est configuré par défaut dans toutes les installations de Maven pour être interrogé. Cette configuration peut être surchargée par un fichier `settings.xml` personnalisé. Attention, le Super POM par défaut a désactivé la récupération des artefacts snapshots depuis le dépôt central de Maven. Si vous avez besoin d'un dépôt snapshot, vous devrez configurer vos dépôts dans votre fichier `pom.xml` ou votre fichier `settings.xml`. Le fichier `settings.xml` et les profils sont traités dans le Chapitre 11, Profils de Build ainsi que dans Section A.2, « Détails des settings ».
- ❷ Le dépôt central de Maven contient aussi des plugins Maven. Le dépôt de plugins est par défaut le dépôt central de Maven. La récupération des snapshots est désactivée par défaut et la règle de gestion des mises à jour indique "never", ce qui signifie que Maven ne met jamais automatiquement à jour un plugin si une nouvelle version est publiée.
- ❸ L'élément `build` définit les positions par défaut des répertoires selon la disposition Maven standard des répertoires.
- ❹ Depuis la version 2.0.9 de Maven les versions par défaut des plugins du cœur sont définies dans le Super POM. Cela a été mis en place pour apporter une certaine stabilité aux utilisateurs qui ne précisent pas de version dans leurs POMs.

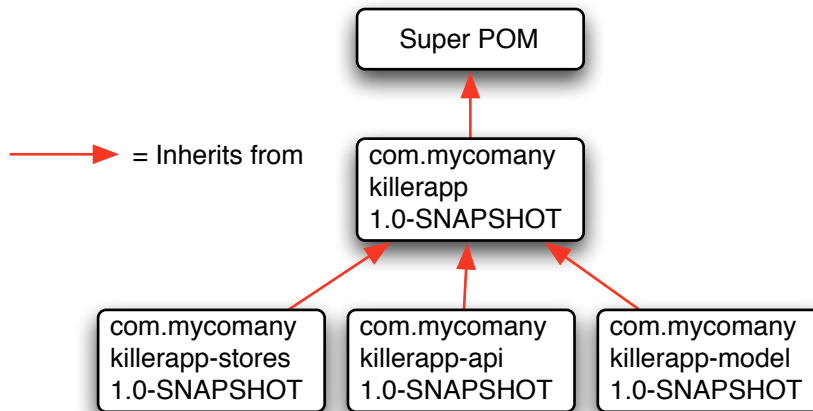


Figure 9.2. Le Super POM est toujours le parent de plus haut niveau

9.2.2. Le POM le plus simple possible

Tous les POMs Maven héritent leurs valeurs par défaut du Super POM (dont nous avons parlé dans la Section 9.2.1, « Le Super POM »). Si vous écrivez un projet tout simple qui produit un JAR à partir de

fichiers source se trouvant dans `src/main/java` dont les tests JUnit à exécuter sont dans `src/test/java` et que vous voulez produire le site web de ce projet avec la commande `mvn site`, vous n'avez rien à configurer. Vous n'avez besoin, dans ce cas, que du POM le plus simple possible, tel celui présenté dans l'Exemple 9.2, « Le POM le plus simple possible ». Ce POM définit un `groupId`, un `artifactId` et une `version` : les trois coordonnées exigées pour tout projet.

Exemple 9.2. Le POM le plus simple possible

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch08</groupId>
  <artifactId>simplest-project</artifactId>
  <version>1</version>
</project>
```

Un POM aussi simple serait plus qu'adéquat pour un projet très simple — par exemple, une bibliothèque Java qui produirait un fichier JAR. Ce POM ne fait aucune référence à un autre projet, ne dépend de rien et il lui manque des informations basiques comme son nom et une URL. Si vous créez un tel fichier et un sous-répertoire `src/main/java` contenant du code source, l'exécution de la commande `mvn package` produirait le JAR `target/simple-project-1.jar`.

9.2.3. Le POM effectif

Ce POM, le plus simple soit-il, introduit le concept de “POM effectif”. Comme les POMs héritent leur configuration d'autres POMs, vous devez toujours voir un POM Maven comme la combinaison du Super POM, plus tous les POMs parents intermédiaires et enfin le POM du projet en cours. Maven commence par le Super POM et surcharge la configuration par défaut avec un ou plusieurs POMs parents. Puis, il surcharge la configuration résultante avec les valeurs du POM du projet. Au final, vous obtenez un POM effectif qui est le résultat de la combinaison de plusieurs POMs. Si vous voulez voir le POM effectif d'un projet, vous allez devoir exécuter le goal `effective-pom` du plugin Maven Help dont nous parlerons dans la Section 12.3, « Usage du plugin Maven Help ». Pour lancer le goal `effective-pom`, exécutez la commande suivante dans un répertoire contenant un fichier `pom.xml` :

```
$ mvn help:effective-pom
```

L'exécution du goal `effective-pom` devrait afficher un document XML résultant de la fusion du Super POM et du POM de l'Exemple 9.2, « Le POM le plus simple possible ».

9.2.4. Véritables POMs

Plutôt que de taper un ensemble de POMs convenu pour vous guider pas-à-pas, vous pouvez regarder les exemples fournis dans Partie I, « Maven par l'exemple ». Maven est un véritable caméléon, vous pouvez choisir et utiliser ce dont vous voulez profiter. Certains projets libres peuvent accorder de

l'importance à la possibilité de lister les développeurs et les contributeurs, de pouvoir produire une documentation claire pour un projet et de pouvoir gérer automatiquement les livraisons grâce au plugin Maven Release. Par contre, pour quelqu'un qui travaille dans le contexte d'une petite équipe au sein d'une entreprise, les capacités de Maven pour gérer les distributions ou lister les développeurs peuvent avoir un moindre intérêt. La suite de ce chapitre va traiter les caractéristiques du POM en lui-même. Au lieu de vous bombarder avec un listing de 10 pages contenant tout un ensemble de POMs, nous allons nous concentrer sur la création d'une bonne référence pour chacune des sections spécifiques du POM. Dans ce chapitre, nous allons parler des relations entre les POMs mais sans ajouter un nouvel exemple pour l'illustrer. Si vous voulez un tel exemple, vous le trouverez dans le Chapitre 7, Un projet multimodule d'entreprise.

9.3. Syntaxe de POM

Le POM se trouve toujours dans un fichier `pom.xml` dans le répertoire racine d'un projet Maven. Ce document XML peut commencer par la déclaration XML, mais elle n'est pas obligatoire. Toutes les valeurs dans un POM se présentent sous la forme d'éléments XML.

9.3.1. Les versions d'un projet

La balise `version` d'un projet Maven contient le numéro de version stable qui est utilisé pour regrouper et ordonner les distributions. Les versions dans Maven se décomposent ainsi : version majeure, version mineure, version incrémentale et qualifieur. Dans un numéro de version, ces différents éléments se présentent selon le format suivant :

```
<version majeure>.<version mineure>.<version incrémentale>-<qualifieur>
```

Par exemple, la version "1.3.5" correspond à la version majeure 1, mineure 3 et incrémentale 5. La version "5" correspond à la version majeure 5 sans version mineure ou incrémentale. Le qualifieur est utilisé pour les builds des étapes intermédiaires : distributions alpha ou beta, et il est séparé des autres éléments de version par un tiret. Par exemple, la version "1.3-beta-01" a une version majeure 1, mineure 3 et un qualifieur "beta-01".

Suivre ces préconisations pour les numéros de version prend tout son sens lorsque vous commencez à utiliser des intervalles pour vos versions dans les POMs. Les intervalles de versions seront abordés dans la Section 9.4.3, « Intervalle de versions pour une dépendance ». Ils permettent de spécifier une dépendance dont la version est comprise dans cet intervalle. Cela n'est possible que parce que Maven est capable de trier les versions en se basant sur le format de numéro de version dont nous venons de parler.

Si votre numéro de version respecte le format `<majeure>.<mineure>.<incrémentale>-<qualifieur>` alors vos versions seront correctement ordonnées, "1.2.3" sera donc bien considérée comme plus récente que "1.0.2". La comparaison se fera en utilisant les valeurs numériques des versions majeure, mineure et incrémentale. Si votre numéro de version ne respecte pas ce standard, alors vos versions seront comparées comme des chaînes de caractères ; la chaîne "1.0.1b" sera comparée à la chaîne "1.2.0b".

9.3.1.1. Numéro de version de build

Un des problèmes avec les numéros de version est l'ordonnancement des qualifieurs. Prenez par exemple, les numéros de version "1.2.3-alpha-2" et "1.2.3-alpha-10" où "alpha-2" correspond au second build alpha et "alpha-10" au dixième build alpha. Alors que le build "alpha-10" devrait être considéré comme plus récent que le build "alpha-2", Maven va mettre "alpha-10" avant "alpha-2". Ceci est dû à un problème connu dans la façon dont Maven traite les numéros de version.

Maven doit, en théorie, considérer le nombre après le qualifieur comme le numéro du build. En d'autres termes, le qualifieur devrait être "alpha", et le numéro du build 2. Même si Maven a été conçu pour séparer le numéro du build du qualifieur, cette fonctionnalité ne fonctionne pas actuellement. En conséquence, "alpha-2" et "alpha-10" sont comparés comme des chaînes de caractères, ce qui positionne "alpha-10" avant "alpha-2" alphabétiquement. Pour contourner cette limitation, vous devez compléter à gauche vos numéros de version de build qualifiés. Si vous utilisez "alpha-02" et "alpha-10" vous n'aurez plus ce problème, et tout continuera à bien fonctionner le jour où Maven traitera correctement les numéros de version de build.

9.3.1.2. Les versions SNAPSHOT

Les numéros de versions dans Maven peuvent contenir une chaîne de caractères pour indiquer que le projet est en cours de développement. Si une version contient la chaîne "SNAPSHOT", alors Maven va prendre en compte cette clef et la convertir en une valeur de type date et heure au format UTC (Coordinated Universal Time) quand vous installerez ou publierez ce composant. Par exemple, si votre projet est en version "1.0-SNAPSHOT" et que vous déployez ses artefacts sur un dépôt Maven, Maven va alors convertir cette version en "1.0-20080207-230803-1" si vous réalisez votre déploiement le 7 Février 2008 à 23:08 UTC. En d'autres termes, quand vous déployez un snapshot, vous ne livrez pas un composant logiciel ; vous livrez un instantané d'un composant.

Pourquoi utiliser cette fonctionnalité ? Les versions SNAPSHOT sont utilisées pour les projets en cours de développement. Si votre projet dépend d'un composant logiciel en cours de développement, vous pouvez dépendre d'une version SNAPSHOT, Maven essaiera alors de télécharger périodiquement la dernière version snapshot du dépôt lorsque vous lancerez votre build. De même, si la prochaine livraison de votre système est la version "1.4", tant que votre projet n'est pas livré définitivement, il devrait être en version "1.4-SNAPSHOT".

Par défaut, Maven ne va pas vérifier la présence de versions SNAPSHOT sur les dépôts distants. Pour pouvoir dépendre de versions SNAPSHOT les utilisateurs doivent donc activer explicitement le téléchargement des snapshots au travers des balises XML `repository` ou `pluginRepository` dans le POM.

Lorsque vous livrez un projet vous devriez remplacer toutes vos dépendances sur des versions SNAPSHOT par des dépendances vers des versions stables. Si un projet dépend d'un SNAPSHOT, il n'est pas stable car ses dépendances peuvent évoluer dans le temps. Les artefacts publiés sur les dépôts Maven stables comme <http://repo1.maven.org/maven2> ne peuvent pas dépendre de versions

SNAPSHOT. Le Super POM de Maven désactive la publication de snapshots sur le dépôt Central. Les versions SNAPSHOT sont utilisées uniquement pour le développement.

9.3.2. Référence à une propriété

Un POM peut contenir des références à des propriétés, elles sont précédées par le signe dollar et entourées par des accolades. Par exemple, regardons le POM suivant :

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <build>
    <finalName>${project.groupId}-${project.artifactId}</finalName>
  </build>
</project>
```

Si vous écrivez ce XML dans un fichier `pom.xml` et que vous exécutez la commande `mvn help:effective-pom`, vous verrez le message suivant s'afficher sur la sortie standard :

```
...
<finalName>org.sonatype.mavenbook-project-a</finalName>
...
```

Quand Maven lit un POM, il remplace les références vers des propriétés lorsqu'il charge le XML du POM. On rencontre fréquemment des propriétés dans un usage avancé de Maven. Ces propriétés sont similaires à celles que l'on trouve dans d'autres systèmes comme Ant ou Velocity. Il s'agit tout simplement de variables délimitées par `${ . . . }`. Maven fournit trois variables implicites qui peuvent être utilisées pour accéder aux variables d'environnement, aux informations du POM et à votre configuration de Maven :

env

La variable `env` permet d'accéder aux variables d'environnement de votre système d'exploitation ou de votre shell. Par exemple, une référence à `${env.PATH}` dans un POM Maven serait remplacée par le contenu de la variable d'environnement `${PATH}` (ou `%PATH%` sous Windows).

project

La variable `project` permet d'accéder au POM. Vous pouvez utiliser un chemin pavé de points ('.') pour référencer la valeur d'un élément du POM. Par exemple, dans cette section nous avons utilisé le `groupId` et l'`artifactId` pour définir la valeur de l'élément `finalName` dans la configuration du `build`. La syntaxe pour cette référence était : `${project.groupId}-${project.artifactId}`.

settings

La variable `settings` permet d'accéder aux informations de votre configuration de Maven. Là encore, vous pouvez utiliser un chemin pavé de points (.) pour référencer la valeur d'un élément

du fichier `settings.xml`. Par exemple, `${settings.offline}` ferait référence à la valeur de l'élément `offline` du fichier `~/ .m2/settings.xml`.



Note

Vous pouvez rencontrer d'anciens builds qui utilisent `${pom.xxx}` ou simplement `${xxx}` pour référencer des propriétés du POM. Ces méthodes ont été marquées comme abandonnées, et vous ne devriez utiliser que `${project.xxx}`.

En plus de ces trois variables implicites, vous pouvez référencer les propriétés système et toute propriété configurée dans un POM Maven ou dans un profil de build :

Propriétés système en Java

Toutes les propriétés accessibles via la méthode `getProperties()` de la classe `java.lang.System` sont visibles comme propriétés du POM. Voici quelques exemples de propriétés système : `${user.name}`, `${user.home}`, `${java.home}`, et `${os.name}`. Une liste complète des propriétés système se trouve dans la Javadoc de la classe `java.lang.System`.

x

Il est possible de définir des propriétés supplémentaires grâce à la balise `properties`, soit dans un fichier `pom.xml`, dans le fichier `settings.xml`, ou enfin en les chargeant depuis des fichiers externes. Si vous définissez une propriété `fooBar` dans votre fichier `pom.xml`, cette propriété est référencée par `${fooBar}`. Ces propriétés de configuration sont très pratiques lorsque pour construire votre système, vous devez filtrer des ressources que vous déployez sur différentes plateformes. Voici la syntaxe pour déclarer `${foo}=bar` dans un POM:

```
<properties>
  <foo>bar</foo>
</properties>
```

Pour une liste plus détaillée des propriétés disponibles, lisez le Chapitre 15, Propriétés et filtrage des ressources.

9.4. Dépendances d'un projet

Maven sait gérer des dépendances internes et externes. Pour un projet Java, une dépendance externe est une bibliothèque comme Plexus, Spring ou Log4J. Un exemple de dépendance interne est un projet d'application web qui dépend d'un autre projet contenant les classes des services, des objets du domaine ou assurant la persistance. L'Exemple 9.3, « Dépendances d'un projet » donne un exemple des dépendances d'un projet.

Exemple 9.3. Dépendances d'un projet

```
<project>
  ...
  <dependencies>
    <dependency>
```

```

    <groupId>org.codehaus.xfire</groupId>
    <artifactId>xfire-java5</artifactId>
    <version>1.2.5</version>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
  <dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>servlet-api</artifactId>
    <version>2.4</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
...
</project>

```

La première dépendance est une dépendance de compilation vers la bibliothèque XFire SOAP de chez Codehaus. Vous pouvez utiliser une dépendance de ce type dans votre projet lorsque celui-ci dépend d'une bibliothèque pour compiler, exécuter les tests et s'exécuter. La deuxième dépendance est une dépendance sur JUnit avec la portée (scope) `test`. Vous pouvez utiliser une dépendance dans le scope `test` lorsque vous n'avez besoin de cette bibliothèque que pour compiler et exécuter les tests. La dernière dépendance de l'Exemple 9.3, « Dépendances d'un projet » est une dépendance sur l'API Servlet 2.4. Cette dernière dépendance se trouve dans le scope `provided`. Vous pouvez utiliser le scope `provided` quand votre application a besoin d'une bibliothèque à la compilation ainsi que pour les tests et que cette bibliothèque est fournie par un conteneur à l'exécution.

9.4.1. Scope de dépendance

L'Exemple 9.3, « Dépendances d'un projet » nous a permis d'introduire brièvement trois des cinq scopes de dépendance : `compile`, `test` et `provided`. Le scope contrôle dans quel classpath vont se retrouver les dépendances et quelles seront celles qui seront intégrées à l'application. Regardons ces scopes plus en détail :

compile

`compile` est le scope par défaut ; toutes les dépendances sont dans ce scope `compile` si aucun scope n'est précisé. Les dépendances du scope `compile` se retrouvent dans tous les classpaths et sont packagées avec l'application.

provided

Les dépendances du scope `provided` sont utilisées lorsqu'elles doivent être fournies par le JDK ou un conteneur. Par exemple, si vous développez une application web, vous aurez besoin de l'API Servlet dans votre classpath pour pouvoir compiler une servlet, mais vous ne voudrez pas inclure l'API Servlet dans votre fichier WAR ; le JAR de l'API Servlet est fourni par votre serveur d'applications ou par votre conteneur de servlet. Les dépendances du scope `provided` font partie

du classpath de compilation (mais pas de celui d'exécution). Elles ne sont pas transitives et ne seront pas packagées avec l'application.

runtime

Les dépendances du scope `runtime` sont des dépendances nécessaires à l'exécution de l'application et des tests, mais qui sont inutiles à la compilation. Par exemple, vous pouvez avoir besoin d'un JAR pour l'API JDBC à la compilation et uniquement de l'implémentation du driver JDBC à l'exécution.

test

Les dépendances du scope `test` sont des dépendances qui ne sont pas nécessaires à l'application durant son fonctionnement normal, elles ne servent que durant les phases de compilation et d'exécution des tests. Nous avons déjà parlé du scope `test` dans la Section 4.10, « Ajouter des dépendances dans le scope `test` ».

system

Le scope `system` est assez proche du scope `provided` sauf que vous devez fournir un chemin explicite vers le JAR sur le système de fichiers local. Il permet la compilation utilisant des objets natifs faisant partie des bibliothèques système. On suppose que cet artefact est toujours présent et donc il ne sera pas cherché dans un dépôt. Si vous utilisez le scope `system`, vous devez automatiquement lui adjoindre une balise `systemPath`. Il est important de noter que l'utilisation de ce scope n'est pas recommandée (vous devriez toujours essayer de référencer des dépendances qui se trouvent dans un dépôt Maven public ou privé).

9.4.2. Dépendances optionnelles

Supposons que vous travaillez sur une bibliothèque qui fournit un service de cache. Au lieu d'écrire votre propre système de cache en partant de zéro, vous voulez utiliser certaines des bibliothèques existantes qui gèrent un cache sur le système de fichiers ou un cache distribué. Supposons encore que vous voulez permettre à l'utilisateur final de choisir entre un cache sur le système de fichiers et un cache distribué en mémoire. Vous voulez utiliser la bibliothèque libre EHCACHE (<http://ehcache.sourceforge.net/>) pour le cache sur le système de fichiers, et pour le cache distribué en mémoire, vous voulez utiliser SwarmCache (<http://swarmcache.sourceforge.net/>), une autre bibliothèque libre. Vous allez écrire une interface et créer une bibliothèque qui pourra être configurée pour utiliser EHCACHE ou SwarmCache, cependant vous ne voulez pas ajouter une dépendance vers ces deux bibliothèques de cache dans chaque projet qui dépendrait de votre bibliothèque.

En d'autres termes, vous avez besoin de ces deux bibliothèques à la compilation de votre projet, mais vous ne voulez pas que toutes les deux apparaissent comme des dépendances transitives nécessaires à l'exécution dans un projet qui utiliserait votre bibliothèque. Vous pouvez réaliser cela en utilisant des dépendances optionnelles comme dans l'Exemple 9.4, « Déclaration de dépendances optionnelles ».

Exemple 9.4. Déclaration de dépendances optionnelles

```
<project>  
<modelVersion>4.0.0</modelVersion>
```

```

<groupId>org.sonatype.mavenbook</groupId>
<artifactId>my-project</artifactId>
<version>1.0.0</version>
<dependencies>
  <dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
    <version>1.4.1</version>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>swarmcache</groupId>
    <artifactId>swarmcache</artifactId>
    <version>1.0RC2</version>
    <optional>true</optional>
  </dependency>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.13</version>
  </dependency>
</dependencies>
</project>

```

Une fois que vous avez déclaré ces dépendances comme optionnelles, vous devrez les ajouter de manière explicite dans les projets qui dépendront de `my-project`. Par exemple, si vous écrivez une application qui dépend de `my-project` et que vous voulez utiliser l'implémentation EHCACHE, vous devrez ajouter la balise `dependency` suivante à votre projet.

```

<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>my-application</artifactId>
  <version>1.0.0</version>
  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>my-project</artifactId>
      <version>1.0.0</version>
    </dependency>
    <dependency>
      <groupId>net.sf.ehcache</groupId>
      <artifactId>swarmcache</artifactId>
      <version>1.4.1</version>
    </dependency>
  </dependencies>
</project>

```

Dans un monde idéal, vous ne devriez pas avoir besoin de dépendances optionnelles. Au lieu d'avoir un gros projet avec un ensemble de dépendances optionnelles, vous devriez isoler le code spécifique à EHCACHE dans un sous-module `my-project-ehcache` et le code spécifique à SwarmCache dans un autre sous-module `my-project-swarmcache`. Ainsi, les projets qui référencent `my-project`

et qui doivent ajouter une dépendance spécifique, pourraient référencer une dépendance vers une implémentation spécifique et bénéficieraient ainsi des dépendances transitives.

9.4.3. Intervalle de versions pour une dépendance

Vous n'avez pas à dépendre d'une version spécifique d'une dépendance, vous pouvez définir un ensemble de versions acceptables pour une dépendance donnée. Par exemple, vous pouvez configurer votre projet pour qu'il dépende de JUnit 3.8 ou supérieure, ou toutes les versions de JUnit comprises entre 1.2.10 et 1.2.14. Pour cela, vous devez entourer les numéros de versions avec les caractères suivants :

(,)
pour définir un intervalle ouvert

[,]
pour définir un intervalle fermé

Par exemple, si vous voulez accéder aux versions de JUnit supérieures ou égales à 3.8 mais inférieures à 4.0, vous déclareriez votre dépendance comme dans l'Exemple 9.5, « Définition d'un intervalle de versions : JUnit 3.8 - JUnit 4.0 ».

Exemple 9.5. Définition d'un intervalle de versions : JUnit 3.8 - JUnit 4.0

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[3.8,4.0)</version>
  <scope>test</scope>
</dependency>
```

Si vous voulez dépendre de JUnit pour toutes les versions inférieures à 3.8.1, vous ne devez spécifier que la borne supérieure comme dans l'Exemple 9.6, « Définition d'un intervalle de versions : JUnit <= 3.8.1 ».

Exemple 9.6. Définition d'un intervalle de versions : JUnit <= 3.8.1

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[,3.8.1]</version>ex-de
  <scope>test</scope>
</dependency>
```

Une version avant ou après la virgule n'est pas obligatoire et signifie +/- l'infini. Par exemple, "[4.0,)" signifie toute version supérieure ou égale à 4.0. "(2.0)" signifie toute version strictement inférieure à 2.0. "[1.2]" signifie uniquement la version 1.2, et rien d'autre.



Note

Quand vous déclarez une version "normale" telle que la version 3.8.2 de JUnit, Maven voit cette déclaration comme "accepter toute version, mais en préférant la version 3.8.2". Ce

qui signifie que lorsqu'un conflit est détecté, Maven s'autorise l'utilisation des algorithmes de résolution de conflit pour choisir la meilleure version. Si vous spécifiez [3.8.2], cela indique que seule la version 3.8.2 sera utilisée et aucune autre. Si ailleurs il se trouve une dépendance qui spécifie [3.8.1], le build va échouer en vous indiquant le conflit en question. Nous vous précisons tout cela pour que vous soyez au courant de cette option, mais n'utilisez là que lorsque c'est véritablement nécessaire. Pour résoudre ces conflits de version, il est préférable de passer par la balise `dependencyManagement`.

9.4.4. Dépendances transitives

Une dépendance transitive est la dépendance d'une dépendance. Si le `projet-a` dépend du `projet-b`, qui lui-même dépend à son tour du `projet-c`, alors le `projet-c` est une dépendance transitive du `projet-a`. Si le `projet-c` dépendait du `projet-d`, alors le `projet-d` serait lui aussi une dépendance transitive du `projet-a`. Un des grands intérêts de Maven et qu'il sait gérer les dépendances transitives. Il évite ainsi au développeur d'avoir à gérer l'ensemble des dépendances nécessaires pour compiler et exécuter une application. Vous pouvez ainsi dépendre uniquement de Spring sans avoir à gérer toutes les dépendances de Spring.

Maven réalise cette opération en construisant un graphe des dépendances et en gérant les conflits et les recouvrements qui pourraient arriver. Par exemple, si Maven s'aperçoit que deux projets dépendent des mêmes `groupId` et `artifactId`, il va automatiquement trouver la dépendance à utiliser, en favorisant toujours la version la plus récente. Même si tout cela semble fort pratique, il existe un certain nombre de cas où les dépendances transitives peuvent entraîner des problèmes de configuration. Dans ce cas, vous pouvez utiliser l'exclusion de dépendance.

9.4.4.1. Dépendances transitives et scope

Chacun des scopes présentés plus tôt dans la Section 9.4.1, « Scope de dépendance » affecte non seulement le scope de la dépendance dans le projet, mais aussi le comportement transitif de cette dépendance. La manière la plus simple de présenter tout cela est sous la forme d'un tableau, comme celui du Tableau 9.1, « Comment le scope affecte les dépendances transitives ». Les scopes dans la première ligne représentent le scope d'une dépendance transitive. Les scopes dans la colonne de gauche représentent le scope de la dépendance directe. L'intersection des lignes et des colonnes donne le scope de la dépendance transitive. Une cellule vide indique que la dépendance transitive ne sera pas prise en compte.

Tableau 9.1. Comment le scope affecte les dépendances transitives

Scope Direct	Scope Transif			
	compile	provided	runtime	test
compile	compile	-	runtime	-
provided	provided	provided	provided	-
runtime	runtime	-	runtime	-
test	test	-	test	-

Pour illustrer la relation entre le scope d'une dépendance transitive et le scope d'une dépendance directe, voici quelques exemples. Si le `projet-a` possède une dépendance dans le scope `test` vers le `projet-b` qui a une dépendance dans le scope `compile` vers le `projet-c`. Le `projet-c` sera donc une dépendance transitive du `projet-a` dans le scope `test`.

Vous pouvez voir cela comme une ligne de transitivité qui agit comme un filtre sur le scope de dépendance. Les dépendances transitives qui sont dans les scopes `test` et `provided` n'ont en général pas d'effet sur un projet. L'exception à cette règle concerne les dépendances transitives `provided` de dépendances elles-aussi `provided` qui seront donc des dépendances dans le scope `provided` du projet. Les dépendances transitives qui sont dans les scopes `compile` et `runtime` affectent le projet quelque soit le scope de la dépendance directe. Les dépendances transitives qui sont dans le scope `compile` resteront dans ce scope quelque soit le scope de la dépendance directe. Les dépendances transitives qui sont dans le scope `runtime` seront en général dans le même scope que la dépendance directe sauf lorsque cette dernière est dans le scope `compile`. Quand une dépendance transitive est dans le scope `runtime` et que la dépendance directe est dans le scope `compile` alors la dépendance transitive sera dans le scope effectif `runtime`.

9.4.5. Résolution des conflits

À certains moments, vous aurez besoin d'exclure une dépendance transitive lorsque votre projet dépend d'un autre projet et que vous voulez retirer complètement la dépendance transitive ou que vous souhaitez la remplacer par une autre qui apporte la même fonctionnalité. L'Exemple 9.7, « Exclusion d'une dépendance transitive » présente une dépendance vers le `projet-a` qui exclut la dépendance transitive vers le `projet-b`.

Exemple 9.7. Exclusion d'une dépendance transitive

```
<dependency>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>project-a</artifactId>
  <version>1.0</version>
  <exclusions>
    <exclusion>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>project-b</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```

Souvent, vous voudrez remplacer une dépendance transitive par une autre implémentation de la fonctionnalité. Par exemple, si vous dépendez d'une bibliothèque qui elle-même dépend de l'API JTA de Sun, vous pouvez vouloir remplacer cette dépendance transitive. Hibernate en est un bon exemple. Hibernate dépend du JAR de l'API JTA de Sun qui n'est pas disponible sur le dépôt central de Maven car il n'est pas librement distribuable. Heureusement, le projet Apache Geronimo a créé une implémentation indépendante de cette bibliothèque qui est librement redistribuable. Pour remplacer une dépendance transitive par une autre, vous devrez exclure la dépendance transitive et ajouter l'autre dépendance à votre

projet. L'Exemple 9.8, « Exclusion et remplacement d'une dépendance transitive » donne un exemple d'un tel remplacement.

Exemple 9.8. Exclusion et remplacement d'une dépendance transitive

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.5.ga</version>
    <exclusions>
      <exclusion>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jta_1.1_spec</artifactId>
    <version>1.1</version>
  </dependency>
</dependencies>
```

Dans l'Exemple 9.8, « Exclusion et remplacement d'une dépendance transitive », rien n'indique que la dépendance `geronimo-jta_1.1_spec` en remplace une autre, il se trouve juste qu'il s'agit d'une bibliothèque qui présente la même API que la dépendance JTA originale. Voici d'autres raisons pour lesquelles vous voudriez exclure ou remplacer des dépendances transitives :

1. Le `groupId` ou l'`artifactId` de l'artefact a changé alors que le projet nécessite une version avec l'autre nom de cet artefact, ce qui fait que vous vous retrouvez avec deux copies du même projet dans votre classpath. Normalement, Maven détecte ce genre de conflit et utilise une unique version du projet, mais quand les `groupId` et `artifactId` sont différents, Maven les considère comme deux bibliothèques distinctes.
2. Un artefact n'est pas utilisé dans votre projet et la dépendance transitive n'a pas été rendue optionnelle. Dans ce cas, vous voulez exclure cette dépendance non requise pour votre projet car vous essayez de réduire le nombre de bibliothèques que vous redistribuez avec votre application.
3. Un artefact est fourni par votre conteneur à l'exécution et donc il ne doit pas être inclus par votre build. Par exemple, si une dépendance dépend de l'API Servlet et que vous voulez être sûr que cette dépendance transitive ne se retrouve pas dans le répertoire `WEB-INF/lib` de l'application web.
4. Pour exclure une dépendance qui peut être une API avec plusieurs implémentations. C'est le cas présenté dans l'Exemple 9.8, « Exclusion et remplacement d'une dépendance transitive » ; où une API de Sun qui demande une acceptation d'une licence et une installation manuelle dans un dépôt personnel (le JAR JTA de Sun) peut être remplacée par une version librement redistribuable de la même API disponible sur le dépôt central de Maven (l'implémentation JTA de Geronimo).

9.4.6. Gestion des dépendances

Une fois que vous avez adopté Maven dans votre multinationale, vous allez vous demander s'il n'existe pas une meilleure manière de gérer les versions des dépendances dans vos deux cent vingt projets Maven interdépendants. Si chaque projet qui utilise une dépendance comme le connecteur Java de MySQL doit gérer dans son coin le numéro de version de cette dépendance, vous allez au-devant de graves problèmes le jour où vous allez faire évoluer cette version. Puisque les numéros de version sont répartis sur toute l'arborescence du projet, vous allez devoir éditer à la main chacun des fichiers `pom.xml` qui fait référence à cette dépendance pour être sûr de mettre à jour ce numéro de version partout. Même avec des commandes comme `find`, `xargs`, et `awk`, vous risquez d'oublier un POM.

Heureusement, Maven fournit un moyen de consolider le numéro de version d'une dépendance grâce à la balise `dependencyManagement`. Vous rencontrerez habituellement la balise `dependencyManagement` dans un POM parent de haut niveau d'une organisation ou d'un projet. L'utilisation de la balise `dependencyManagement` dans un fichier `pom.xml` vous permet de référencer une dépendance dans un projet fils sans avoir à spécifier la version. Maven va parcourir la hiérarchie des POMs jusqu'à ce qu'il trouve un projet avec une balise `dependencyManagement`, il utilisera alors la version déclarée dans cette balise `dependencyManagement`.

Par exemple, si vous avez un grand nombre de projets qui utilisent le connecteur Java MySQL dans sa version 5.1.2, vous pourriez écrire la balise `dependencyManagement` suivante dans le POM de plus haut niveau de votre projet multimodule.

Exemple 9.9. Définition des versions dans un POM de haut niveau

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>mysql</groupId>
        <artifactId>mysql-connector-java</artifactId>
        <version>5.1.2</version>
      </dependency>
      ...
    </dependencies>
  </dependencyManagement>
```

Vous pourrez alors, dans un projet fils, ajouter une dépendance au connecteur Java MySQL avec le XML suivant :

```
<project>
  <modelVersion>4.0.0</modelVersion>
```

```

<parent>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
</parent>
<artifactId>project-a</artifactId>
...
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
</dependencies>
</project>

```

Faites attention au fait que le projet fils n'a pas déclaré explicitement la version de la dépendance à `mysql-connector-java`. Comme cette dépendance a été déclarée dans la balise `dependencyManagement` du POM de plus haut niveau, le numéro de version va se propager jusqu'à la dépendance `mysql-connector-java` du projet fils. Attention, si le projet fils avait défini une version, celle-ci aurait remplacé la version définie dans la section `dependencyManagement` du POM de plus haut niveau. Tout cela signifie que la version déclarée dans `dependencyManagement` n'est utilisée que lorsque le projet fils ne déclare pas de version.

La gestion des dépendances avec un POM de haut niveau est différente de la déclaration d'une dépendance partagée au travers d'un POM parent. Pour commencer, toutes les dépendances sont héritées. Si `mysql-connector-java` était déclarée comme une dépendance du projet de plus haut niveau, chaque projet fils ferait référence à cette dépendance. Au lieu d'ajouter des dépendances inutiles, l'utilisation de la balise `dependencyManagement` vous permet de consolider et de centraliser la gestion des versions des dépendances sans ajouter de dépendances héritées par tous les projets fils. En d'autres termes, la balise `dependencyManagement` est équivalente à une variable d'environnement qui vous permet de déclarer une dépendance dans un sous-projet sans avoir à préciser de numéro de version.

9.5. Relations entre projets

Une des raisons pour lesquelles nous utilisons Maven est qu'il facilite le suivi des dépendances (et des dépendances de dépendances). Quand un projet dépend d'un artefact produit par un autre projet alors cet artefact est une dépendance. Dans le cas d'un projet Java, cela peut être aussi simple que de dépendre d'une dépendance externe comme `Log4J` ou `JUnit`. Les dépendances peuvent donc représenter des dépendances externes, mais elles permettent aussi de gérer les dépendances entre projets. Si le `projet-a` dépend du `projet-b`, Maven est suffisamment intelligent pour savoir que le `projet-b` doit être construit avant le `projet-a`.

Les relations d'un projet ne se limitent pas aux dépendances et à déterminer tout ce qui est lui nécessaire pour construire un artefact. Maven sait modéliser les relations d'un projet vers son parent et entre sous-modules. Cette section présente les différentes relations qui peuvent exister entre projets et comment les configurer.

9.5.1. Au sujet des coordonnées

Les coordonnées permettent de définir de manière unique un projet, nous en avons déjà parlé dans le Chapitre 3, Mon premier projet avec Maven. Les relations entre projets se font au moyen des coordonnées Maven. Il ne suffit pas de dire que le projet-a dépend du projet-b ; en réalité, un projet avec un `groupId`, un `artifactId` et une version dépend d'un autre projet avec son `groupId`, son `artifactId` et sa version. Pour résumer, les coordonnées Maven se composent de trois éléments :

groupId

Un `groupId` regroupe un ensemble d'artefacts. Ces identifiants de groupe ont, en général, la forme d'un package Java. Par exemple, le `groupId` `org.apache.maven` est le `groupId` de base de l'ensemble des artefacts produits par le projet Apache Maven. Les identifiants de groupe sont convertis en chemin dans les dépôts Maven ; par exemple le `groupId` `org.apache.maven` correspondra au répertoire `/maven2/org/apache/maven` sur `repo1.maven.org`¹.

artifactId

L'`artifactId` est l'identifiant principal du projet. Quand vous produisez un artefact, il va prendre le nom de l'`artifactId`. Quand vous faites référence à un projet vous utiliserez l'`artifactId`. La combinaison `artifactId`, `groupId` doit être unique. En d'autres termes, vous ne pouvez pas avoir deux projets distincts avec les mêmes `artifactId` et `groupId` ; les `artifactIds` sont uniques pour un `groupId` donné.



Note

Si l'utilisation des `!` est très courante pour les `groupIds`, vous devriez éviter d'en utiliser pour les `artifactIds`. En effet, cela peut provoquer des problèmes lors de l'interprétation d'un nom complet jusqu'aux sous-composants.

version

Quand un artefact est délivré, on lui affecte un numéro de version. Ce numéro de version peut être un identifiant numérique tel que "1.0", "1.1.1" ou "1.1.2-alpha-01". Vous pouvez aussi utiliser ce que l'on appelle une version snapshot. Une version snapshot est une version d'un composant en cours de développement. Les numéros de version snapshot se terminent toujours par `SNAPSHOT` ; par exemple "1.0-SNAPSHOT", "1.1.1-SNAPSHOT" ou "1-SNAPSHOT". La Section 9.3.1.1, « Numéro de version de build » décrit les numéros de versions et les intervalles de versions.

Il existe un quatrième identifiant moins utilisé :

classifier

On utilise un `classifier` lorsqu'on livre le même code, mais sous la forme de plusieurs artefacts distincts pour des raisons techniques. Par exemple, si vous voulez construire deux artefacts d'un JAR, un compilé avec le compilateur Java 1.4 et un autre avec le compilateur Java 6,

¹ <http://repo1.maven.org/maven2/org/apache/maven>

vous utiliserez le classifieur pour produire deux JARs différents avec la même combinaison `groupId:artifactId:version`. Si votre projet utilise des extensions en code natif, vous utiliserez le classifieur pour distinguer les artefacts selon chaque plate-forme cible. Les classifieurs sont couramment utilisés lors du packaging du code source, de la JavaDoc d'un artefact ou d'assemblage de binaires.

Dans ce livre, lorsque nous évoquons des dépendances, nous utilisons souvent le format simplifié suivant pour décrire une dépendance : `groupId:artifactId:version`. Pour référencer la version 2.5 de Spring, nous utiliserions `org.springframework:spring:2.5`. Quand vous demandez à Maven d'afficher une liste des dépendances grâce au plugin Maven Dependency, vous verrez que Maven a tendance à produire des traces en utilisant cette écriture simplifiée.

9.5.2. Projets multimodules

Les projets multimodules sont des projets qui se composent d'un ensemble de modules à construire. Le packaging d'un projet multimodule a toujours pour valeur `pom`, et ce projet produit rarement un artefact. Un projet multimodule n'existe que pour regrouper un ensemble de projets dans un même build. La Figure 9.3, « Relations dans un projet multimodule » nous montre une hiérarchie de projets qui se compose de deux projets parents avec un packaging de type `pom`, et de trois projets de type `jar`.

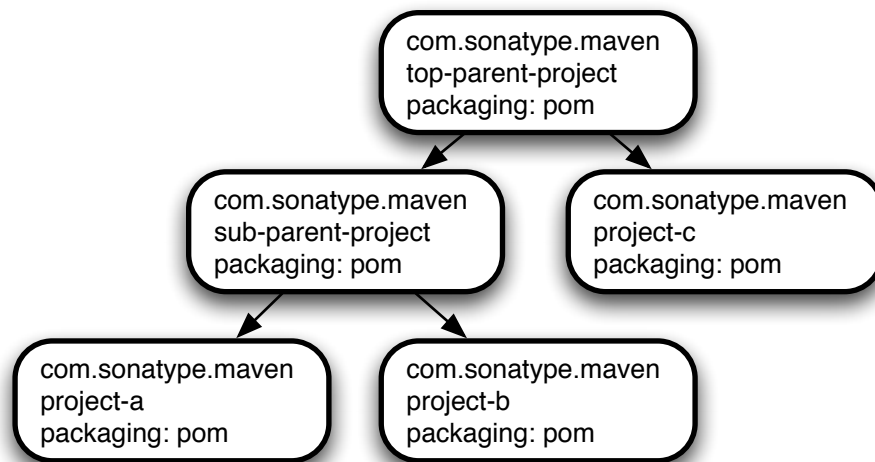


Figure 9.3. Relations dans un projet multimodule

La structure des répertoires sur le système de fichiers suit les relations entre les modules. L'ensemble des projets illustré par la Figure 9.3, « Relations dans un projet multimodule » aurait la structure de répertoires suivante :

```
top-parent-project/pom.xml
top-parent-project/subparent-parent-project/pom.xml
```

```
top-parent-project/subparent-parent-project/project-a/pom.xml
top-parent-project/subparent-parent-project/project-b/pom.xml
top-parent-project/project-c/pom.xml
```

Les projets sont liés entre eux par les projets `top-parent-project` et `subparent-parent-project` comme sous-modules dans un POM. Par exemple, le projet `org.sonatype.mavenbook:top-parent-project` est un projet multimodule avec un packaging de type `pom`. Le fichier `pom.xml` du projet `top-parent-project` doit posséder la balise `modules` suivante :

Exemple 9.10. Balise `modules` du projet `top-parent-project`

```
<project>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>top-parent-project</artifactId>
  ...
  <modules>
    <module>subparent-parent-project</module>
    <module>project-c</module>
  </modules>
  ...
</project>
```

Quand Maven parse le POM du projet `top-parent-project`, il analyse la balise `modules` pour s'apercevoir que le projet `top-parent-project` référence les projets `subparent-parent-project` et `project-c`. Maven va alors traiter les fichiers `pom.xml` dans chacun de ces sous-répertoires. Maven répète cette opération pour chacun des sous-modules : il lira le fichier `subparent-parent-project/pom.xml` pour s'apercevoir que le projet `subparent-parent-project` fait référence à deux projets dans la balise `modules` suivante :

Exemple 9.11. Balise `modules` du projet `subparent-parent-project`

```
<project>
  ...
  <modules>
    <module>project-a</module>
    <module>project-b</module>
  </modules>
  ...
</project>
```

Attention au fait que nous appelons "modules" les projets qui composent un projet multimodule et non projets "enfants" ou "fils". Nous agissons ainsi délibérément pour ne pas confondre les projets qui sont liés dans un projet multimodule de ceux qui héritent d'un même POM.

9.5.3. Héritage de projet

Parfois, nous voulons qu'un projet hérite de certaines valeurs d'un POM parent. Vous pourriez être en train de construire un énorme système, et vous ne voulez pas répéter les mêmes balises de dépendance encore et encore. Vous pouvez éviter de vous répéter dans vos projets par l'héritage grâce à la balise

parent. Quand un projet spécifie un parent, il hérite de l'ensemble des informations du POM du projet parent. Il peut redéfinir les valeurs du POM parent et en ajouter.

Tous les POMs de Maven héritent d'un POM parent. Si un POM ne spécifie pas de parent direct au travers de la balise `parent`, ce POM héritera des valeurs définies dans le Super POM. L'Exemple 9.12, « Héritage entre projets » présente la balise `parent` du `project-a` qui hérite du POM du projet `a-parent`.

Exemple 9.12. Héritage entre projets

```
<project>
  <parent>
    <groupId>com.training.killerapp</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <artifactId>project-a</artifactId>
  ...
</project>
```

L'exécution de la commande `mvn help:effective-pom` dans le `project-a` va afficher un POM qui est le résultat de la fusion du Super POM avec le POM du projet `a-parent` et celui du `project-a`. Les héritages implicites et explicites du `project-a` sont présentés dans Figure 9.4, « Héritage pour les projets `a-parent` et `project-a` ».

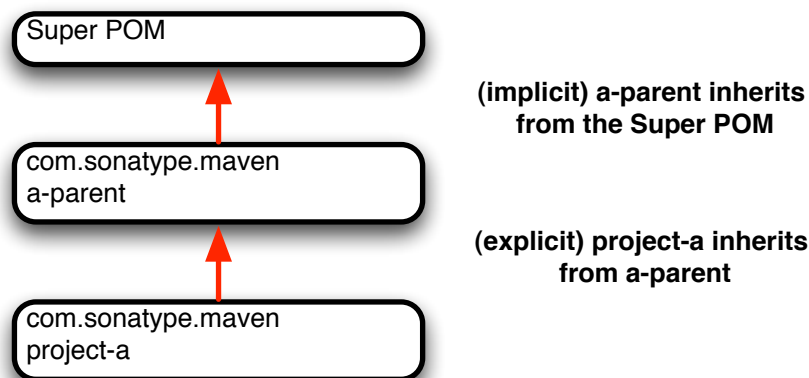


Figure 9.4. Héritage pour les projets `a-parent` et `project-a`

Quand un projet spécifie un projet parent, Maven utilise le POM de ce parent comme point de départ avant de traiter le POM du projet courant. Il hérite de tout, jusqu'au `groupId` et au numéro de `version`. Vous pouvez remarquer que le `project-a` ne spécifie aucun des deux, le `groupId` et la `version` sont tous les deux hérités du projet `a-parent`. Lorsqu'une balise `parent` est présente, tout ce qu'un POM a

besoin de définir est l'`artifactId`. Cela n'est pas obligatoire, le `project-a` peut avoir un `groupId` et une `version` différents de ceux du parent, mais s'ils ne sont pas définis alors Maven utilisera ceux du POM parent. Si vous commencez à utiliser Maven pour gérer et construire de gros projets multimodules, vous aurez souvent à créer des projets qui auront un même `groupId` et une même `version`.

Quand vous héritez d'un POM, vous avez le choix entre utiliser les valeurs héritées de ce POM ou les surcharger sélectivement. Ce qui suit est une liste d'éléments qu'un POM Maven hérite de son POM parent :

- les identifiants (il faut au moins surcharger le `groupId` ou l'`artifactId`).
- les dépendances
- les développeurs et les contributeurs
- les listes de plugins
- les listes de rapports
- les exécutions de plugin (les exécutions qui ont le même id sont fusionnées)
- les configuration de plugin

Quand Maven hérite de dépendances, il va ajouter les dépendances définies dans les projets fils à celles des projets parents. C'est ainsi que vous pouvez spécifier des dépendances utilisées tout au long de projets qui héritent d'un même POM parent. Par exemple, si votre projet fait usage de Log4J pour ses traces, vous pouvez ajouter cette dépendance dans votre POM de plus haut niveau. Tous les projets qui hériteront de ce POM auront automatiquement Log4J comme dépendance. De même, si vous voulez vous assurer que tous vos projets utilisent la même version d'un plugin Maven, vous pouvez définir la version de ce plugin Maven dans la section `pluginManagement` d'un POM parent de haut niveau.

Maven assume que le POM parent est disponible dans le dépôt local ou dans le répertoire parent (`../pom.xml`) du projet courant. Si aucune de ces options n'est valide, ce comportement par défaut peut être redéfini par la balise `relativePath`. Par exemple, certaines organisations préfèrent une structure à plat des projets où le fichier `pom.xml` du projet parent ne se trouve pas dans le répertoire parent d'un projet fils. Il peut se trouver dans un répertoire au même niveau que celui du projet. Si votre projet fils se trouve dans le répertoire `./projet-a` et que le projet parent se trouve dans `./a-parent`, vous préciserez dans le POM le chemin relatif du projet `parent-a` ainsi :

```
<project>
  <parent>
    <groupId>org.sonatype.mavenbook</groupId>
    <artifactId>a-parent</artifactId>
    <version>1.0-SNAPSHOT</version>
    <relativePath>../a-parent/pom.xml</relativePath>
  </parent>
  <artifactId>projet-a</artifactId>
</project>
```

9.6. Les bonnes pratiques du POM

Maven peut être utilisé pour tout gérer, depuis des systèmes composés d'un projet tout simple à des systèmes comprenant des dizaines de sous-modules liés les uns aux autres. Pour utiliser Maven, il ne suffit pas d'apprendre la syntaxe nécessaire à sa configuration, il faut apprendre le "Maven Way" (la "manière Maven"). Il s'agit d'un ensemble de bonnes pratiques pour organiser et construire des projets avec Maven. Cette section essaye d'en donner certains éléments pour vous éviter d'avoir à creuser dans les strates de fils de discussions des listes de diffusion de Maven qui se sont accumulées au cours des années.

9.6.1. Regrouper les dépendances

Si vous avez un ensemble de dépendances qui sont logiquement liées les unes aux autres vous pouvez créer un projet de type `pom` qui va les regrouper. Par exemple, supposons que votre application utilise Hibernate, un célèbre framework de mapping Objet-Relationnel. Tout projet qui utilise Hibernate dépend aussi de Spring et du pilote JDBC pour MySQL. Au lieu d'avoir à déclarer ces dépendances dans chaque projet qui utilise Hibernate, Spring et MySQL vous pourriez créer un POM spécial qui ne fait rien d'autre que de déclarer un ensemble de dépendances communes. Vous pourriez ainsi créer un projet `persistence-deps` (un raccourci pour Dépendances de Persistance) et déclarer dans chaque projet qui nécessite une fonction de persistance, une dépendance à ce projet :

Exemple 9.13. Consolidation des dépendances dans un projet POM

```
<project>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>persistence-deps</artifactId>
  <version>1.0</version>
  <packaging>pom</packaging>
  <dependencies>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate</artifactId>
      <version>${hibernateVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.hibernate</groupId>
      <artifactId>hibernate-annotations</artifactId>
      <version>${hibernateAnnotationsVersion}</version>
    </dependency>
    <dependency>
      <groupId>org.springframework</groupId>
      <artifactId>spring-hibernate3</artifactId>
      <version>${springVersion}</version>
    </dependency>
    <dependency>
      <groupId>mysql</groupId>
      <artifactId>mysql-connector-java</artifactId>
      <version>${mysqlVersion}</version>
    </dependency>
  </dependencies>
</project>
```

```

</dependencies>
<properties>
  <mysqlVersion>(5.1,)</mysqlVersion>
  <springVersion>(2.0.6,)</springVersion>
  <hibernateVersion>3.2.5.ga</hibernateVersion>
  <hibernateAnnotationsVersion>3.3.0.ga</hibernateAnnotationsVersion>
</properties>
</project>

```

Si vous créez ce projet dans un répertoire `persistence-deps`, tout ce que vous avez à faire ensuite est d'écrire ce fichier `pom.xml` et d'exécuter la commande `mvn install`. Comme c'est un projet dont le packaging est de type `pom`, ce POM s'installe dans votre dépôt local. Vous pouvez maintenant ajouter ce projet comme dépendance de votre projet et toutes ses dépendances seront automatiquement ajoutées à votre projet. Quand vous ajoutez une dépendance vers ce projet `persistence-deps`, n'oubliez pas de spécifier qu'il s'agit d'une dépendance dont le packaging est de type `pom`.

Exemple 9.14. Déclaration d'une dépendance vers un POM

```

<project>
  <description>Ce projet nécessite JDBC</description>
  ...
  <dependencies>
    ...
    <dependency>
      <groupId>org.sonatype.mavenbook</groupId>
      <artifactId>persistence-deps</artifactId>
      <version>1.0</version>
      <type>pom</type>
    </dependency>
  </dependencies>
</project>

```

Si plus tard vous décidez de changer de pilote JDBC (en utilisant par exemple JTDS), vous n'avez qu'à remplacer les dépendances dans le projet `persistence-deps` pour utiliser `net.sourceforge.jtds:jtds` au lieu de `mysql:mysql-java-connector` puis mettre à jour le numéro de version. Tous les projets qui dépendent de `persistence-deps` utiliseront JTDS s'ils décident d'utiliser la nouvelle version. La consolidation des dépendances est une bonne pratique qui permet de réduire la taille des fichiers `pom.xml` qui ont un grand nombre de dépendances. Si vous devez partager un grand nombre de dépendances entre plusieurs projets, vous pouvez aussi établir des relations parent-enfant entre ceux-ci et extraire toutes les dépendances communes dans le projet parent. La limite de cette approche parent-enfant est qu'un projet peut n'avoir qu'un seul parent. Parfois, il est donc plus simple de regrouper ces dépendances et de référencer une dépendance de type `pom`. Ainsi, votre projet peut référencer autant de POM de dépendance selon ses besoins.



Note

Maven utilise le niveau d'une dépendance dans l'arbre lorsqu'il résout un conflit, ne retenant que la dépendance la plus proche. Cette technique de regroupement de

dépendances les fait descendre d'un niveau dans l'arbre. Ne l'oubliez pas lorsque vous devez choisir entre regrouper des dépendances dans un fichier `pom.xml` ou utiliser la balise `dependencyManagement` dans un POM parent.

9.6.2. Multimodule ou héritage

Il existe une différence entre hériter d'un projet parent et être géré dans un projet multimodule. Un projet parent transfère sa configuration à ses enfants. Un projet multimodule gère un ensemble de sous-projets ou modules. Les relations dans un projet multimodule vont du haut vers le bas. Quand vous configurez un projet multimodule, vous indiquez tout simplement que votre build doit intégrer les modules spécifiés. Les builds multimodules servent à regrouper un ensemble de modules dans un même build. La relation parent-enfant va du noeud feuille vers le haut. La relation parent-enfant concerne plutôt la définition d'un projet particulier. Quand vous associez un projet fils à son parent, vous indiquez à Maven que le POM d'un projet dérive d'un autre.

Pour illustrer la prise de décision entre une approche par héritage et une approche multimodule regardez les deux exemples qui vont suivre : le projet Maven utilisé pour produire ce livre et un projet hypothétique qui contient un ensemble logique de modules.

9.6.2.1. Projet simple

Tout d'abord, jetons un oeil au projet `maven-book`. L'héritage et les relations multimodules sont présentées dans la Figure 9.5, « Héritage dans le projet multimodule `maven-book` ».

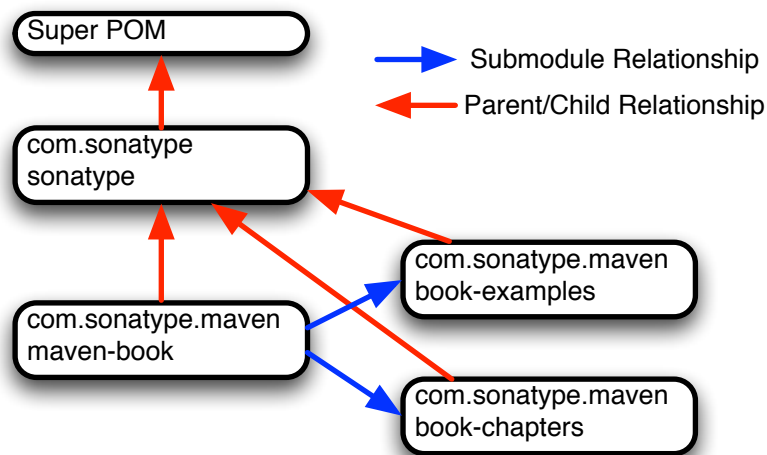


Figure 9.5. Héritage dans le projet multimodule `maven-book`

Quand nous construisons ce livre sur Maven que vous êtes en train de lire, nous exécutons la commande `mvn package` dans un projet multimodule qui s'appelle `maven-book`. Ce projet multimodule se

compose de deux sous-modules : `book-examples` et `book-chapters`. Aucun de ces projets ne partage le même parent, ils ne sont liés que par le fait qu'ils sont tous les deux des modules du projet `maven-book`. Le projet `book-examples` construit les archives ZIP et TGZ que vous avez téléchargées pour obtenir les exemples de ce livre. Quand nous exécutons le build du projet `book-examples` depuis le répertoire `book-examples/` avec la commande `mvn package`, il ne sait pas qu'il n'est qu'une partie du projet `maven-book`. Le projet `book-examples` ne s'intéresse pas au projet `maven-book`, tout ce qu'il sait est que son parent est le POM `sonatype` de plus haut-niveau et qu'il construit une archive d'exemples. Dans ce cas, le projet `maven-book` existe uniquement comme facilitateur en regroupant des modules.

Chacun de ces trois projets : `maven-book`, `book-examples` et `book-chapters` ont le même parent "d'entreprise" — `sonatype`. C'est une pratique courante dans les organisations qui ont adopté Maven. Au lieu d'avoir chaque projet qui étende le Super POM par défaut, certaines organisations définissent un POM d'entreprise de haut niveau qui sert de parent par défaut quand un projet n'a aucune raison d'en dépendre d'un autre. Dans ce livre, par exemple, il n'y a aucune raison pour que les projets `book-examples` et `book-chapters` partagent le même POM parent. Il s'agit de projets complètement différents avec leurs propres dépendances et configurations. Ils utilisent des plugins très différents pour construire ce que vous êtes en train de lire. Le POM `sonatype` permet à une organisation de personnaliser le comportement par défaut de Maven et de fournir des informations propres à l'organisation pour le déploiement et les profils.

9.6.2.2. Projet multimodule d'entreprise

Regardons un exemple qui fournit une image plus précise de ce que pourrait être un véritable projet où l'héritage et les relations multimodules existent côte à côte. La Figure 9.6, « Héritage dans un projet multimodule d'entreprise » montre un ensemble de projets qui ressemblent à ceux que l'on pourrait rencontrer dans une véritable application d'entreprise. Il y a un POM de haut niveau pour l'entreprise avec un `artifactId` ayant pour valeur `sonatype`. Il y a ensuite un projet multimodule `big-system` qui référence les sous-modules `server-side` et `client-side`.

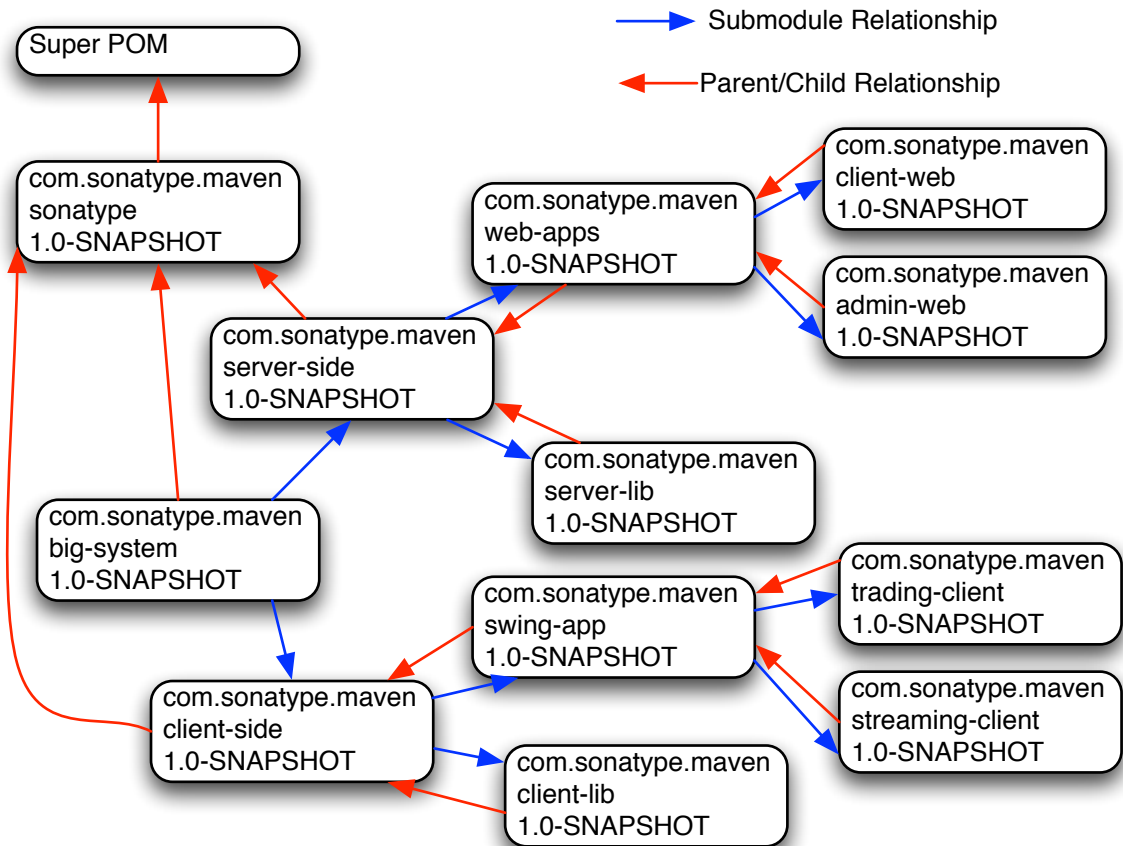


Figure 9.6. Héritage dans un projet multimodule d'entreprise

De quoi s'agit-il ? Essayons de donner un sens à toutes ces flèches. Tout d'abord, regardons `big-system`. Le projet `big-system` pourrait être le projet dans lequel vous exécuteriez la commande `mvn package` pour construire et tester le système dans son entier. Le projet `big-system` référence les sous-modules `client-side` et `server-side`. Chacun de ces projets regroupe le code qui s'exécute côté serveur et côté client. Concentrons-nous sur le projet `server-side`. Sous ce projet `server-side` se trouve un projet appelé `server-lib` et un projet multimodule `web-apps`. Sous `web-apps` nous retrouvons deux applications web Java : `client-web` et `admin-web`.

Commençons par les relations parent/enfant de `client-web` et `admin-web` avec `web-apps`. Puisque ces deux applications web utilisent le même framework (Wicket par exemple), ces deux projets partagent un même ensemble de dépendances. Les dépendances vers l'API Servlet, l'API JSP et Wicket seraient toutes regroupées dans le projet `web-apps`. Ces deux projets `client-web` et `admin-web` doivent aussi dépendre de `server-lib`. Cette dépendance sera définie comme une dépendance de `web-apps`

vers `server-lib`. Puisque `client-web` et `admin-web` partagent tous les deux cette configuration en héritant de `web-apps`, ces deux projets auront des POMs succincts contenant tout au plus les identifiants, le parent et le nom de l'artefact final.

Maintenant, concentrons-nous sur la relation parent/enfant de `web-apps` et `server-lib` vers `server-side`. Dans ce cas-ci, supposons qu'il existe deux groupes de développeurs séparés, l'un qui travaille sur le code côté serveur et l'autre sur le code côté client. La liste des développeurs serait configurée dans le POM du projet `server-side` et héritée par ses projets fils : `web-apps`, `server-lib`, `client-web` et `admin-web`. Nous pourrions aussi imaginer que le projet `server-side` soit configuré différemment pour prendre en compte les spécificités du développement côté serveur. Le projet `server-side` pourrait utiliser un profil qui n'aurait de sens que pour tous les projets `server-side`. Ce profil pourrait contenir les informations de connexion à la base de données, ou le POM de ce projet `server-side` pourrait configurer une version spécifique du plugin Maven Jetty qui serait récupérée par tous les projets qui hériteraient du POM `server-side`.

Dans cet exemple, la principale raison d'utiliser les relations parent/enfant est cet ensemble de dépendances partagées et la configuration commune à un groupe de projets qui sont liés logiquement. Tous les projets sous `big-system` sont liés les uns aux autres en tant que sous-modules, cependant tous les sous-modules ne sont pas obligés de déclarer comme parent le projet qui les a déclarés comme sous-module. Tout est sous-module pour des raisons de simplification, pour construire le système dans son ensemble allez dans le répertoire du projet `big-system` et exécutez la commande `mvn package`. Si vous regardez attentivement la figure vous noterez qu'il n'existe pas de relation parent/enfant entre les projets `server-side` et `big-system`. Pourquoi donc ? L'héritage de POM est très puissant, mais il ne doit pas être utilisé à tort et à travers. Quand le partage de dépendances et de configuration a du sens, il faut utiliser la relation parent/enfant. Mais elle n'a pas de sens lorsque les deux projets sont très différents. Ce qui est le cas, par exemple, des projets `server-side` et `client-side`. Il est tout à fait possible de concevoir un système où `client-side` et `server-side` héritent du POM commun de `big-system`. Cependant à chaque apparition d'une divergence significative entre ces deux projets il va falloir trouver des manières subtiles pour factoriser la configuration commune dans `big-system` sans perturber tous les projets fils. Même si `client-side` et `server-side` partagent la même dépendance à Log4J, ils peuvent avoir des configurations de plugins très différentes.

Il existe un certain point, dont la position dépend de votre style et de votre expérience, au-delà duquel vous estimerez que la duplication de configuration est acceptable pour garder des projets comme `client-side` et `server-side` indépendants . Concevoir un système qui se compose de dizaines de projets sur plus de cinq niveaux d'héritage de POM n'est pas la meilleure idée qu'on puisse avoir. Dans une telle configuration, vous n'avez pas dupliqué votre dépendance à Log4J, mais vous devrez naviguer entre les cinq niveaux de POM pour comprendre comment Maven calcule votre POM effectif. Toute cette complexité pour éviter d'avoir à dupliquer cinq fois les quelques lignes d'une dépendance. Avec Maven, il y a toujours "une manière Maven de faire", mais il en existe bien d'autres. Au final, tout est question de préférences et de style. Dans la plupart des cas vous n'aurez pas de soucis si vos sous-modules déclarent le même projet comme parent, mais il se peut que votre utilisation de Maven évolue dans temps.

Chapitre 10. Cycle de vie du build

10.1. Introduction

Maven modélise les projets en entités qui sont définies par un POM. Le POM capture donc l'identité d'un projet : Quel est son contenu ? De quel type de packaging a-t-il besoin ? Le projet a-t-il un parent ? Quelles sont ses dépendances ? Nous avons exploré les principes de description d'un projet dans les chapitres précédents, mais nous n'avons pas introduit le mécanisme qui permet à Maven d'agir sur ces objets. Dans Maven, les "actions" (ou verbe) sont représentées par des goals. Ceux-ci sont packagés dans des plugins qui sont attachés à des phases du cycle de vie du build. Le cycle de vie Maven se compose d'une séquence de phases nommées comme `prepare-resources`, `compile`, `package` ou `install`. Il y a donc, entre autres, une phase pour la compilation et une autre pour le packaging. Il existe également des phases du type `pre-` et `post-`. Celles-ci peuvent être utilisées, par exemple, pour enregistrer des goals devant être exécutés avant la compilation ou effectuer certaines tâches après l'exécution d'une phase spécifique. Quand vous demandez à Maven de construire un projet, vous lui demandez de parcourir la liste des différentes phases du cycle de vie et d'exécuter chacun des goals associés à celles-ci.

Le cycle de vie d'un build est organisé en une séquence de phases qui permettent d'ordonner des groupes de goals. Ces goals sont choisis et rattachés en fonction du type de packaging de leur projet. Il existe trois types de cycle de vie : `clean`, `default` (quelquefois appelé `build`) et `site`. Dans ce chapitre, vous allez apprendre comment Maven rattache des goals aux phases de ces cycles de vie et comment personnaliser un cycle de vie. Vous apprendrez également les différentes phases du cycle de vie par défaut.

10.1.1. Cycle de vie Clean (`clean`)

Le premier cycle de vie auquel nous allons nous intéresser est le cycle de vie le plus simple de Maven. L'exécution de la commande `mvn clean` invoque le cycle de vie 'clean'. Celui-ci contient trois phases :

- `pre-clean`
- `clean`
- `post-clean`

La phase la plus intéressante de ce cycle de vie est celle nommée `clean`. Le goal 'clean' du plugin Clean (`clean:clean`) est attaché à la phase `clean` du cycle de vie `clean`. Le goal `clean:clean` efface les traces laissées par la dernière construction en supprimant le répertoire de build. Si vous n'avez pas modifié l'emplacement du répertoire de build, celui-ci se trouve par défaut dans le répertoire `${basedir}/target`. Lorsque vous exécutez le goal `clean:clean`, vous ne l'exécutez pas directement avec `mvn clean:clean`, vous l'utilisez par l'intermédiaire de la phase `clean` du cycle de vie du même nom. Exécuter la phase `clean` donne à Maven l'opportunité d'exécuter les autres goals rattachés à la phase `pre-clean`.

Par exemple, supposez que vous vouliez exécuter un goal `antrun:run` pour afficher une notification lors du `pre-clean`, ou archiver le répertoire de build du projet avant qu'il ne soit supprimé. Exécuter directement le goal `clean:clean` ne lancera pas le cycle de vie, vos tâches annexes ne seront donc pas exécutées. Par contre, si vous utilisez la phase `clean`, vous utiliserez le cycle de vie du même nom (`clean`) et traverserez de ce fait les trois phases du cycle de vie de jusqu'à la phase `clean`. L'Exemple 10.1, « Exécuter un goal lors du pre-clean » montre une configuration de build qui permet de rattacher le goal `antrun:run` à la phase `pre-clean` pour afficher une alerte avant la suppression de l'artefact en cours de nettoyage. Dans cet exemple, le goal `antrun:run` est utilisé pour exécuter des commandes Ant qui permettent de tester l'existence d'un artefact de projet. Ainsi, si l'artefact du projet est trouvé et est donc sur le point d'être effacé, une alerte s'affichera à l'écran.

Exemple 10.1. Exécuter un goal lors du pre-clean

```
<project>
...
<build>
  <plugins>... <plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
    <execution>
      <id>file-exists</id>
      <phase>pre-clean</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <!-- adds the ant-contrib tasks (if/then/else used below) -->
          <taskdef resource="net/sf/antcontrib/antcontrib.properties" />
          <available
            file="${project.build.directory}/${project.build.finalName}.${project.packaging}"
            property="file.exists" value="true" />

          <if>
            <not>
              <isset property="file.exists" />
            </not>
            <then>
              <echo>No
                ${project.build.finalName}.${project.packaging} to
                delete</echo>
            </then>
            <else>
              <echo>Deleting
                ${project.build.finalName}.${project.packaging}</echo>
            </else>
          </if>
        </tasks>
      </configuration>
    </execution>
  </executions>
</project>
```

```

<dependencies>
  <dependency>
    <groupId>ant-contrib</groupId>
    <artifactId>ant-contrib</artifactId>
    <version>1.0b2</version>
  </dependency>
</dependencies>
</plugin>
</plugins>
</build>
</project>

```

Exécuter la commande `mvn clean` avec une telle configuration sur un projet produira un résultat ressemblant à celui-ci :

```

[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Your Project
[INFO]   task-segment: [clean]
[INFO] -----
[INFO] [antrun:run {execution: file-exists}]
[INFO] Executing tasks
  [echo] Deleting your-project-1.0-SNAPSHOT.jar
[INFO] Executed tasks
[INFO] [clean:clean]
[INFO] Deleting directory ~/corp/your-project/target
[INFO] Deleting directory ~/corp/your-project/target/classes
[INFO] Deleting directory ~/corp/your-project/target/test-classes
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 1 second
[INFO] Finished at: Wed Nov 08 11:46:26 CST 2006
[INFO] Final Memory: 2M/5M
[INFO] -----

```

En plus de configurer Maven pour exécuter un goal lors de la phase `pre-clean`, vous pouvez personnaliser le plugin Clean pour effacer des dossiers en plus du répertoire contenant le build. Vous pouvez configurer le plugin pour supprimer une liste de fichier en lui spécifiant un `fileSet`. L'exemple ci-dessous configure le plugin Clean pour supprimer tous les fichiers du répertoire `target-other/` en utilisant des wildcards standard Ant : `*` et `**`.

Exemple 10.2. Personnaliser le comportement du plugin Clean

```

<project>
  <modelVersion>4.0.0</modelVersion>
  ...
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-clean-plugin</artifactId>
        <configuration>

```

```

<filesets>
  <fileset>
    <directory>target-other</directory>
    <includes>
      <include>*.class</include>
    </includes>
  </fileset>
</filesets>
</configuration>
</plugin>
</plugins>
</build>
</project>

```

10.1.2. Cycle de vie par défaut (default)

La plupart des utilisateurs Maven sont familiers avec le cycle de vie par défaut. C'est un modèle général de processus de build pour une application. Ce cycle de vie commence par la phase `validate` et se termine avec la phase `deploy`. Les phases du cycle de vie Maven par défaut sont présentées dans le Tableau 10.1, « Phases du cycle de vie Maven ».

Tableau 10.1. Phases du cycle de vie Maven

Phase du cycle de vie	Description
<code>validate</code>	Valide le projet et la disponibilité de toutes les informations requises pour le build
<code>generate-sources</code>	Génère le code source nécessaire pour l'inclure à la compilation
<code>process-sources</code>	Traite le code source, pour filtrer certaines valeurs par exemple
<code>generate-resources</code>	Génère les ressources à inclure dans le package
<code>process-resources</code>	Copie et traite les ressources dans leur répertoire destination, afin qu'elles soient prêtes pour le packaging
<code>compile</code>	Compile le code source du projet
<code>process-classes</code>	Traite à posteriori les fichiers générés par la compilation, pour modifier du bytecode par exemple
<code>generate-test-sources</code>	Génère le code source des tests pour l'inclure à la compilation
<code>process-test-sources</code>	Traite le code source des tests, pour filtrer certaines valeurs par exemple
<code>generate-test-resources</code>	Génère les ressources pour les tests

Phase du cycle de vie	Description
process-test-resources	Copie et traite les ressources de test dans le répertoire destination des tests
test-compile	Compile le code source des tests dans le répertoire destination des tests
test	Exécute les tests en utilisant le framework de test approprié. Le code de ces tests ne doit pas être nécessaire au packaging ni au déploiement.
prepare-package	Effectue les opérations nécessaires pour la préparation du package avant que celui-ci ne soit réellement créé. Il en résulte souvent une version dézippée et prête à être packagée du futur package (Maven 2.1+)
package	Package le code compilé dans un format distribuable tel que JAR, WAR ou EAR
pre-integration-test	Effectue les actions nécessaires avant de lancer les tests d'intégration, comme configurer un environnement par exemple.
integration-test	Traite et déploie si nécessaire le package dans l'environnement où les tests pourront être exécutés
post-integration-test	Effectue les actions nécessaires à la fin de l'exécution des tests d'intégration, comme nettoyer l'environnement par exemple
verify	Lance des points de contrôle pour vérifier que le package est valide et qu'il passe les critères qualité
install	Installe le package dans le dépôt local, celui-ci pourra ainsi être utilisé comme dépendance par d'autres projets locaux
deploy	Copie le package final sur le dépôt distant. Permet de partager le package avec d'autres utilisateurs et projets (souvent pertinent pour une vraie livraison)

10.1.3. Cycle de vie Site (site)

Maven fait plus que construire des artefacts, il peut également produire de la documentation et différents rapports d'un ou plusieurs projets. La documentation du projet et la génération du site web ont un cycle de vie séparé. Celui-ci contient quatre phases :

1. pre-site
2. site
3. post-site
4. site-deploy

Les goals suivants sont rattachés par défaut à ce cycle de vie :

1. site - site:site
2. site-deploy - site:deploy

Le type de packaging n'a aucune influence sur ce cycle de vie. Les types de packaging influent sur la création des artefacts, pas sur le type de site généré. Le plugin Site lance l'exécution de la production d'un document Doxia¹ et de différents plugins permettant de générer des rapports. Vous pouvez générer un site à partir d'un projet Maven en exécutant la commande suivante :

```
$ mvn site
```

Pour plus d'informations à propos de la génération de site Maven, consultez le Chapitre 16, Génération du Site.

10.2. Cycles de vie spécifiques par type de package

Des goals spécifiques sont rattachés à chaque phase en fonction du packaging du projet. Un projet de type `jar` ne dispose pas des mêmes goals par défaut qu'un projet de type `war`. La balise `packaging` influe donc sur les étapes requises pour construire un projet. Pour illustrer cela, prenons un exemple avec deux projets : l'un avec un packaging de type `pom` et l'autre avec un packaging de type `jar`. Le projet avec le packaging de type `pom` lancera le goal `site:attach-descriptor` durant la phase `package`, alors que le projet de type `jar` exécutera le goal `jar:jar`.

Les paragraphes suivants décrivent le cycle de vie pour tous les types de packaging disponibles dans Maven. Ils vous permettront de trouver quels goals sont rattachés au cycle de vie par défaut.

¹ <http://maven.apache.org/doxia/>

10.2.1. JAR

JAR est le type de packaging par défaut, c'est également le plus commun et le plus souvent rencontré. Les goals par défaut rattachés au cycle de vie pour un projet possédant ce type de packaging sont montrés dans le Tableau 10.2, « Goals par défaut pour le packaging de type JAR ».

Tableau 10.2. Goals par défaut pour le packaging de type JAR

Phase du cycle de vie	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

10.2.2. POM

POM est le plus simple des types de packaging. L'artefact généré est le pom lui-même plutôt qu'un fichier JAR, SAR ou EAR. Il ne possède pas de code à tester ni même à compiler, et n'a pas de ressources à traiter. Les goals par défaut d'un projet du type POM sont affichés dans le Tableau 10.3, « Goals par défaut d'un projet du type POM ».

Tableau 10.3. Goals par défaut d'un projet du type POM

Phase du cycle de vie	Goal
package	site:attach-descriptor
install	install:install
deploy	deploy:deploy

10.2.3. Plugin Maven

Ce type de packaging ressemble au packaging JAR avec trois particularités : `plugin:descriptor`, `plugin:addPluginArtifactMetadata` et `plugin:updateRegistry`. Ces goals génèrent un descripteur de fichier et effectuent quelques modifications sur les données. Les goals par défaut des projets possédant un packaging de type Plugin sont présentés sur le Tableau 10.4, « Goals par défaut d'un projet du type Plugin ».

Tableau 10.4. Goals par défaut d'un projet du type Plugin

Phase du cycle de vie	Goal
generate-resources	plugin:descriptor
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar, plugin:addPluginArtifactMetadata
install	install:install, plugin:updateRegistry
deploy	deploy:deploy

10.2.4. EJB

Les EJBs (Enterprise Java Beans) proposent un mécanisme standard d'accès aux données. Ils proposent une approche orientée modèle (model-driven development) pour le développement d'applications Java d'entreprise. Maven fournit un support pour les EJB 2 et 3. Les paramètres par défaut du plugin recherchent la présence des fichiers de configuration EJB 2.1, vous devez configurer le plugin EJB pour packager spécifiquement un EJB3. Les goals par défaut pour des projets du type EJB sont présentés dans le Tableau 10.5, « Goals par défaut d'un projet du type EJB ».

Tableau 10.5. Goals par défaut d'un projet du type EJB

Phase du cycle de vie	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	ejb:ejb
install	install:install
deploy	deploy:deploy

10.2.5. WAR

Le packaging de type WAR ressemble également aux types JAR et EJB, à l'exception près du goal `war:war` de la phase `package`. Notez que le plugin `war:war` nécessite la présence d'un fichier de configuration `web.xml` dans le répertoire `src/main/webapp/WEB-INF`. Les goals par défaut d'un projet de type WAR sont présentés dans le Tableau 10.6, « Goals par défaut d'un projet de type WAR ».

Tableau 10.6. Goals par défaut d'un projet de type WAR

Phase du cycle de vie	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy

10.2.6. EAR

Les EARs sont probablement les artefacts les plus simples d'une application Java EE. Ils sont constitués d'un descripteur de déploiement nommé `application.xml`, de ressources et de modules. Le plugin EAR possède un goal nommé `generate-application-xml`. Celui-ci génère le fichier `application.xml` à partir de la configuration définie dans le POM du projet EAR. Les goals par défaut d'un projet de type EAR sont présentés dans le Tableau 10.7, « Goals par défaut d'un projet de type EAR ».

Tableau 10.7. Goals par défaut d'un projet de type EAR

Phase du cycle de vie	Goal
generate-resources	ear:generate-application-xml
process-resources	resources:resources
package	ear:ear
install	install:install
deploy	deploy:deploy

10.2.7. Autres types de packaging

La liste que nous venons de parcourir n'est pas exhaustive. Nombre d'autres types de packaging sont disponibles par l'intermédiaire de projets externes et de plugins, dont : le type de packaging NAR (native archive), les types SWF et SWC pour les projets qui produisent des contenus Adobe Flash et Flex. Vous pouvez également définir un type de packaging personnalisé et modifier le cycle de vie par défaut pour répondre à vos contraintes.

Pour utiliser l'un de ces types de packaging personnalisé, vous avez besoin de deux choses : un plugin qui définit le cycle de vie de ce type de packaging personnalisé et un dépôt qui contient ce plugin. Plusieurs types de packaging personnalisés sont définis dans des plugins téléchargeables à partir du dépôt Maven

central. Voici l'exemple d'un projet qui référence le plugin Israfil Flex et qui utilise un type de packaging personnalisé SWF comme résultat de la construction du code source Adobe Flex.

Exemple 10.3. Type de packaging personnalisé pour Adobe Flex (SWF)

```
<project>
...
<packaging>swf</packaging>
...
<build>
  <plugins>
    <plugin>
      <groupId>net.israfil.mojo</groupId>
      <artifactId>maven-flex2-plugin</artifactId>
      <version>1.4-SNAPSHOT</version>
      <extensions>true</extensions>
      <configuration>
        <debug>true</debug>
        <flexHome>${flex.home}</flexHome>
        <useNetwork>true</useNetwork>
        <main>org/sonatype/mavenbook/Main.mxml</main>
      </configuration>
    </plugin>
  </plugins>
</build>
...
</project>
```

Dans la Section 17.6, « Plugins et le cycle de vie Maven », nous vous montrons comment créer votre propre type de packaging avec un cycle de vie personnalisé. Cet exemple devrait vous donner une idée de ce que vous avez à faire pour référencer un nouveau type de packaging. Tout ce dont vous avez besoin est de référencer le plugin qui fournit votre nouveau type de packaging. Le plugin Israfil Flex est un plugin Maven non officiel hébergé sur Google Code, pour plus d'informations sur son fonctionnement et son utilisation, rendez-vous à l'adresse <http://code.google.com/p/israfil-mojo>. Ce plugin fournit le cycle de vie suivant pour les projets de type SWF :

Tableau 10.8. Cycle de vie par défaut d'un projet de type SWF

Phase du cycle de vie	Goal
compile	flex2:compile-swc
install	install
deploy	deploy

10.3. Goals communs aux cycles de vie

La plupart des cycles de vie de packaging possèdent des goals en commun. Si vous regardez les goals rattachés aux cycles de vie WAR et JAR, vous remarquerez que seule la phase `package` diffère. La

phase `package` du cycle de vie WAR appelle `war:war` alors que cette même phase appelle `jar:jar` pour le cycle de vie JAR. La plupart des cycles de vie que vous rencontrerez partageront certains goals pour gérer les ressources, exécuter les tests et compiler votre code. Dans cette section, nous explorerons quelques-uns des goals utilisés sur plusieurs cycles de vie.

10.3.1. Traiter les ressources

La plupart des cycles de vie rattachent le goal `resources:resources` à la phase `process-resources`. La phase `process-resources` "traite" les ressources et les copie dans le répertoire destination. Si vous n'avez pas personnalisé les répertoires par défaut définis dans le Super POM, Maven copiera les fichiers du répertoire `${basedir}/src/main/resources` dans le répertoire `${basedir}/target/classes` ou dans le répertoire défini dans `${project.build.outputDirectory}`. En plus de les copier, Maven peut également appliquer des filtres aux ressources pour remplacer certains de leurs éléments. Tout comme dans le POM où les variables étaient définies par la notation `${...}`, vous pouvez référencer des variables dans les ressources de votre projet en utilisant cette même syntaxe. Couplée aux profils, cette fonctionnalité peut être utilisée pour construire des artefacts configurés pour différentes plateformes de déploiement. Il est courant d'avoir besoin de construire des artefacts multi-environnement, en effet, un projet a souvent plusieurs environnements cibles : développement, tests, recette, production. Pour plus d'informations à propos des profils, consultez le Chapitre 11, Profils de Build.

Pour illustrer ce filtrage des ressources, prenons l'exemple d'un projet possédant un fichier XML : `src/main/resources/META-INF/service.xml`. Vous voulez externaliser certaines variables de configuration dans un fichier `properties`. En d'autres termes, vous voulez référencer l'URL JDBC, l'utilisateur et le mot de passe de votre base de données, sans directement mettre ces valeurs en dur dans le fichier `service.xml`. À la place, vous désirez utiliser un fichier `properties` pour centraliser toutes les valeurs de configuration de votre programme. Cela vous permet de consolider toute la configuration dans un seul fichier `properties`, et simplifie la mise à jour des valeurs de configuration lors de la configuration d'un nouvel environnement. Commençons par regarder le contenu du fichier `service.xml` du répertoire `src/main/resources/META-INF`.

Exemple 10.4. Utilisation des propriétés dans les ressources du projet

```
<service>
  <!-- This URL was set by project version ${project.version} -->
  <url>${jdbc.url}</url>
  <user>${jdbc.username}</user>
  <password>${jdbc.password}</password>
</service>
```

Ce fichier XML utilise la même syntaxe pour gérer ses propriétés que celle que du POM. La première variable référencée est la variable `project`, celle-ci est une variable implicite qui est également disponible dans le POM. La variable `project` propose un moyen d'accéder à certaines informations du POM. Les trois variables suivantes sont `jdbc.url`, `jdbc.username` et `jdbc.password`. Ces variables sont définies dans un fichier de `properties` `src/main/filters/default.properties`.

Exemple 10.5. default.properties dans src/main/filters

```
jdbc.url=jdbc:hsqldb:mem:mydb
jdbc.username=sa
jdbc.password=
```

Pour configurer le filtrage des ressources avec le fichier `default.properties`, nous avons besoin de préciser deux choses dans le POM du projet : une liste de fichiers de propriétés dans la balise `filters` de la configuration du build et un booléen qui permet à Maven de savoir si un répertoire doit être filtré. Le comportement par défaut de Maven est de ne pas effectuer de filtre mais de juste copier les ressources dans le répertoire destination : vous devez donc configurer explicitement le filtrage des ressources. Ce comportement par défaut permet d'éviter les mauvaises surprises comme, par exemple, filtrer sans que vous le désiriez des fichiers contenant des éléments `${...}`.

Exemple 10.6. Filtrage des ressources (remplacer les propriétés)

```
<build>
  <filters>
    <filter>src/main/filters/default.properties</filter>
  </filters>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
```

Comme pour tous les répertoires dans Maven, celui contenant les ressources du projet ne se trouve pas forcément dans `src/main/resources`. Il ne s'agit que de la valeur par défaut définie dans le Super POM. Notez également que vous n'avez pas besoin de centraliser toutes vos ressources dans un unique répertoire. Vous pouvez les séparer dans plusieurs sous-répertoires dans `src/main`. Imaginez que votre projet contienne des centaines de fichiers XML et des centaines d'images. Au lieu de centraliser toutes vos ressources dans le répertoire `src/main/resources`, il voudrez probablement créer deux répertoires `src/main/xml` et `src/main/images`. Pour ajouter des répertoires à la liste des répertoires contenant les ressources, ajoutez la balise `resource` suivante dans la configuration de votre build.

Exemple 10.7. Ajouter des répertoire ressources complémentaires

```
<build>
  ...
  <resources>
    <resource>
      <directory>src/main/resources</directory>
    </resource>
    <resource>
      <directory>src/main/xml</directory>
    </resource>
    <resource>
```



```

    <directory>src/main/images</directory>
  </resource>
</resources>
...
</build>

```

Lorsque vous construisez un projet qui produit une application en ligne de commande ou console, vous voudrez souvent écrire des scripts shell et les référencer dans le JAR produit par un build. Quand vous utilisez le plugin Assembly pour distribuer votre application via un ZIP ou un TAR, vous pouvez placer tous vos scripts dans un répertoire du type `src/main/command`. Dans la configuration des ressources du POM suivant, vous verrez comment nous pouvons utiliser le filtrage des ressources et avec une référence vers une variable pour modifier le nom du JAR. Pour plus d'informations sur le plugin Maven Assembly, consultez le Chapitre 14, Maven Assemblies.

Exemple 10.8. Filtrage de ressources Scripts

```

<build>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>simple-cmd</artifactId>
  <version>2.3.1</version>
  ...
  <resources>
    <resource>
      <filtering>>true</filtering>
      <directory>${basedir}/src/main/command</directory>
      <includes>
        <include>run.bat</include>
        <include>run.sh</include>
      </includes>
      <targetPath>${basedir}</targetPath>
    </resource>
    <resource>
      <directory>${basedir}/src/main/resources</directory>
    </resource>
  </resources>
  ...
</build>

```

Si vous exécutez la commande `mvn process-resources` dans ce projet, vous vous retrouverez avec deux fichiers dans `${basedir}` : `run.sh` et `run.bat`. Nous avons inclus ces deux fichiers dans la balise `resource`, configuré le filtrage, et affecté le `targetPath` à la valeur `${basedir}`. Dans une seconde balise `resource`, nous avons configuré le chemin par défaut des ressources pour qu'elles soient copiées sans filtrage dans le répertoire de destination. L'Exemple 10.8, « Filtrage de ressources Scripts » montre comment déclarer deux répertoires ressources et les configurer différemment. Le projet de l'Exemple 10.8, « Filtrage de ressources Scripts » doit disposer un fichier `run.bat` dans le dossier `src/main/command` qui contient le code suivant :

```

@echo off
java -jar ${project.build.finalName}.jar %*

```

Après avoir exécuté la commande `mvn process-resources`, un fichier nommé `run.bat` devrait apparaître dans `${basedir}`, en voici son contenu :

```
@echo off
java -jar simple-cmd-2.3.1.jar %*
```

La possibilité de personnaliser le filtrage pour un sous-ensemble spécifique de ressources est l'une des raisons de séparer vos ressources par type sur un projet. Plus un projet est complexe, plus il sera avantageux de séparer les ressources dans différents répertoires. L'alternative au fait de conserver différentes sortes de ressources nécessitant différentes configurations de filtre dans des répertoires différents est d'utiliser un ensemble complexe de pattern 'include' et 'exclude' pour différencier les types de configuration. Cette seconde solution est beaucoup plus complexe à maintenir et à mettre en place.

10.3.2. Compilation

La plupart des cycles de vie rattachent le goal `compile` du plugin Compiler à la phase `compile`. Cette phase appelle `compile:compile` qui est configuré pour compiler tout le code source et copier le bytecode dans le répertoire destination du build. Si vous n'avez pas personnalisé les valeurs définies dans le Super POM, `compile:compile` compilera toutes les sources du répertoire `src/main/java` dans `target/classes`. Le plugin Compiler appelle `javac` et utilise les paramètres par défaut : 1.3 pour le code source, et 1.1 pour le bytecode produit. Autrement dit, le plugin Compiler présume que le code source de votre projet est conforme Java 1.3 et qu'il doit tourner sur une JVM Java 1.1. Si vous voulez modifier ce paramétrage, vous devez fournir la configuration suivante du plugin Compiler dans le POM de votre projet.

Exemple 10.9. Modifier les versions du code source et du bytecode pour le plugin Compiler

```
<project>
...
<build>
...
<plugins>
  <plugin>
    <artifactId>maven-compiler-plugin</artifactId>
    <configuration>
      <source>1.5</source>
      <target>1.5</target>
    </configuration>
  </plugin>
</plugins>
...
</build>
...
</project>
```

Notez que nous avons configuré le plugin Compiler, pas le goal `compile:compile`. Si nous voulions configurer le source et le bytecode généré juste pour le goal `compile:compile`, nous aurions placé la balise `configuration` sous la balise `execution` du goal `compile:compile`. Nous avons effectué

cette configuration au niveau du plugin, car `compile:compile` n'est pas le seul goal intéressé par cette configuration. Le plugin Compiler est réutilisé pour la compilation des tests via le goal `compile:testCompile`, cette configuration nous permet donc de faire d'une pierre deux coups.

Si vous voulez personnaliser l'emplacement du code source, vous pouvez modifier la configuration du build. Si vous désirez placer le code source de votre projet dans le répertoire `src/java` au lieu du répertoire `src/main/java`, et si vous voulez que votre projet soit généré dans `classes` au lieu `target/classes`, vous pouvez toujours modifier la configuration par défaut du `sourceDirectory` définie dans le Super POM.

Exemple 10.10. Modifier le répertoire du code source par défaut

```
<build>
...
<sourceDirectory>src/java</sourceDirectory>
<outputDirectory>classes</outputDirectory>
...
</build>
```



Avertissement

Alors qu'il pourrait vous sembler nécessaire de configurer Maven pour qu'il utilise votre propre structure de répertoires, nous n'essayerons jamais assez de vous dissuader de modifier la configuration Maven par défaut. Nous n'essayons pas ici de vous laver le cerveau pour vous forcer à utiliser la structure Maven, mais il reste beaucoup simple de suivre quelques conventions pour que tout le monde puisse comprendre rapidement votre projet.

10.3.3. Traiter les ressources des tests

La phase `process-test-resources` est quasi-identique à la phase `process-resources`. Si le POM comporte quelques légères différences, la plus grande partie de la configuration reste identique. Vous pouvez filtrer les ressources de vos tests de la même manière que les ressources du projet. L'emplacement par défaut des ressources pour les tests, défini dans le Super POM, est `src/test/resources`, le répertoire destination est configuré par défaut dans la variable `${project.build.testOutputDirectory}` et correspond à `target/test-classes`.

10.3.4. Compilation des tests

La phase `test-compile` est quasiment identique à la phase `compile`. La seule différence est que `test-compile` appelle `compile:testCompile` pour compiler le code source des tests. Si vous n'avez pas personnalisé les répertoires par défaut du Super POM, `compile:testCompile` compilera le code source contenu du répertoire `src/test/java` dans `target/test-classes`.

Comme pour le répertoire du code source, si vous désirez personnaliser l'emplacement du code source des tests ou du répertoire destination de la compilation, vous pouvez surcharger

`testSourceDirectory` et `testOutputDirectory`. Si vous voulez stocker le code source de vos tests dans `src-test/` au lieu du répertoire `src/test/java` et que vous voulez enregistrer le bytecode dans le répertoire `classes-test/` au lieu `target/test-classes`, utilisez la configuration suivante.

Exemple 10.11. Modifier l'emplacement du code source et du bytecode des tests

```
<build>
...
<testSourceDirectory>src-test</testSourceDirectory>
<testOutputDirectory>classes-test</testOutputDirectory>
...
</build>
```

10.3.5. Tester

La plupart des cycles de vie rattachent le goal `test` du plugin Surefire à leur phase `test`. Le plugin Surefire est un plugin Maven permettant d'exécuter des tests unitaires. Le comportement par défaut du plugin Surefire est de rechercher toutes les class se terminant par `*Test` dans le répertoire source des tests, puis de les exécuter comme des tests JUnit². Le plugin Surefire peut également être configuré pour exécuter des tests unitaires TestNG³.

Après avoir exécuté la commande `mvn test`, vous aurez probablement noté que le plugin Surefire a produit des rapports d'exécution dans le répertoire `target/surefire-reports`. Ce répertoire de rapports contient deux fichiers pour chaque test exécuté par le plugin : un fichier XML qui contient les informations d'exécution du test, un fichier texte qui contient la sortie des tests unitaires. Si un problème est survenu durant la phase de test et qu'un test unitaire a échoué, vous pouvez utiliser la sortie standard de Maven et ce répertoire pour trouver la cause du problème. Ce répertoire `surefire-reports/` est également utilisé durant la génération du site pour créer un résumé 'facile à lire' du résultat des tests unitaires.

Si vous travaillez sur un projet qui contient des tests unitaires qui échouent, mmais que vous voulez tout de même générer votre artefact, vous devez configurer le plugin Surefire pour qu'il poursuive le build même en cas d'échec. Le comportement par défaut est d'arrêter le build lorsqu'un test unitaire échoue. Pour modifier ce comportement, vous devez affecter la propriété `testFailureIgnore` du plugin Surefire à `'true'`.

Exemple 10.12. Configurez le plugin Surefire pour ignorer les tests en échec

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <testFailureIgnore>true</testFailureIgnore>
      </configuration>
    </plugin>
  </plugins>
</build>
```

² <http://www.junit.org>

³ <http://www.testng.org>

```
</configuration>
</plugin>
...
</plugins>
</build>
```

Si vous souhaitez ne pas exécuter l'ensemble des tests, vous pouvez exécuter la commande suivante :

```
$ mvn install -Dmaven.test.skip=true
```

La variable `maven.test.skip` agit sur les plugins Compiler et Surefire. En précisant cette variable, vous demandez à Maven d'ignorer les tests.

10.3.6. Installer l'artefact

Le goal `install` du plugin Install est toujours rattaché à la phase `install` du cycle de vie. Ce goal `install:install` installe simplement les artefacts d'un projet dans le dépôt local. Si vous avez un projet avec un `groupId` à `org.sonatype.mavenbook`, un `artifactId` à `simple-test`, et une `version` à `1.0.2`, le goal `install:install` copiera le fichier JAR de `target/simple-test-1.0.2.jar` vers `~/.m2/repository/org/sonatype/mavenbook/simple-test/1.0.2/simple-test-1.0.2.jar`. Si le projet a un type de packaging POM, ce goal copiera le POM dans le dépôt local.

10.3.7. Déploiement

Le goal `deploy` du plugin Deploy est souvent rattaché à la phase `deploy` du cycle de vie. Cette phase est utilisée pour déployer un artefact sur un dépôt Maven distant. Cela est souvent utilisé pour mettre à jour un dépôt distant quand vous effectuez une release. La procédure de déploiement peut aller d'une simple copie de fichier d'un répertoire dans un autre, à un ensemble complexe de transferts SCP de fichiers en utilisant une clé publique. Les préférences de déploiement nécessitent souvent des accès à un dépôt distant, et donc, elles sont souvent stockées hors du fichier `pom.xml`. Ces préférences se retrouvent donc souvent dans un fichier propre à chaque utilisateur : `~/.m2/settings.xml`. Tout ce que vous avez besoin de savoir pour le moment est que le goal `deploy:deploy` est rattaché à la phase `deploy` et qu'il permet le transfert d'un artefact sur un dépôt de publication et la mise à jour des informations d'un dépôt qui peuvent être affectées par un tel déploiement.

Chapitre 11. Profils de Build

11.1. À quoi servent-ils ?

Les profils permettent d'adapter un build à l'environnement, ils assurent la portabilité entre différents environnements de build.

Qu'entendons-nous par différents environnements de build ? La production et le développement sont deux environnements typiques. Quand vous travaillez dans l'environnement de développement, votre système est certainement configuré pour travailler sur une base de données de développement installée sur le poste local tandis qu'en production, votre système est configuré pour accéder à la base de données de production. Maven permet de définir différents environnements de build (profils de build) qui peuvent remplacer n'importe quel paramètre du fichier `pom.xml`. Vous pourriez configurer votre application pour lire dans votre base de données de développement locale dans un profil "development", et la configurer dans le profil "production" pour lire dans la base de données de production. Les profils peuvent également être activés en fonction de l'environnement et de la plateforme, vous pouvez personnaliser un build pour qu'il s'exécute différemment selon le système d'exploitation ou la version du JDK installé. Avant que nous parlions de l'utilisation et la configuration des profils Maven, nous devons définir le concept de Portabilité du Build.

11.1.1. Qu'est ce que la Portabilité du Build ?

La "portabilité" du build est la mesure de la facilité avec laquelle on peut prendre un projet particulier et le construire sur différents environnements. Un build qui fonctionne sans aucune configuration particulière ni personnalisation des fichiers properties est plus portable qu'un build qui nécessite un certain travail pour construire le projet à partir de rien. Les projets les plus portables sont bien souvent des projets Open Source très utilisés comme Apache Commons ou Apache Velocity qui fonctionnent avec Maven sans aucune personnalisation ou presque. Pour faire simple, les builds les plus portables fonctionnent immédiatement et les moins portables vous demandent de réaliser des acrobaties plus ou moins périlleuses dans les fichiers de configuration pour modifier les chemins vers les outils de build en fonction des plateformes. Avant que nous vous dévoilions comment obtenir un build portable, étudions les différents types de portabilité dont nous allons parler.

11.1.1.1. Builds non portables

L'absence de portabilité est exactement ce que les outils de build essayent d'éviter - cependant, n'importe quel outil peut être configuré pour être non-portable (même Maven). Un projet n'est pas portable lorsqu'il ne peut être construit que dans certaines circonstances spécifiques (ex : votre machine locale). À moins que vous travailliez seul et que vous n'envisagiez pas de déployer votre application sur une autre machine, il vaut mieux éviter complètement la non-portabilité. Un build non-portable fonctionne seulement sur une machine, c'est un build "à usage unique". Maven est conçu pour décourager les builds non-portables en offrant la possibilité de personnaliser les builds grâce aux profils.

Quand un développeur récupère le code source d'un projet non-portable, il n'est pas en mesure de construire le projet sans remanier une grosse partie du script de build.

11.1.1.2. Portabilité sur l'environnement

Un build est qualifié de portable d'un environnement à l'autre s'il possède un mécanisme pour personnaliser son comportement et sa configuration en fonction de l'environnement. Par exemple, un projet qui contient une référence à une base de données de test dans un environnement de test et à une base de données de production dans un environnement de production, est portable sur ces deux environnements. Il est probable que ce build ait différentes propriétés pour chaque environnement. Quand vous changez pour un environnement qui n'est pas défini et qui ne possède pas de profil associé, le projet ne fonctionnera pas. Donc un projet n'est portable que sur des environnements bien définis.

Quand un nouveau développeur récupère le code source d'un projet dépendant de l'environnement, il devra exécuter le build dans cet environnement ou créer l'environnement adéquat pour réussir à construire le projet.

11.1.1.3. Portabilité interne à une organisation

Le point clef de cet environnement est que seuls quelques-uns ont accès à des ressources internes à l'organisation, comme le gestionnaire de configuration ou le dépôt interne Maven. Un projet dans une grande entreprise peut dépendre de la présence d'une base de données accessible uniquement pour les développeurs internes. Un projet Open Source peut exiger un certain niveau de droits pour pouvoir publier le site web et les artefacts produits sur un dépôt public.

Si vous essayez de construire un projet interne hors du réseau interne de l'entreprise (par exemple de l'autre côté du firewall), le build va échouer. Il se peut que cela vienne de plugins propres à l'entreprise qui ne sont pas disponibles ou de dépendances du projet inaccessibles car vous n'avez pas les droits nécessaires pour accéder au dépôt distant d'entreprise. Un tel projet n'est portable que sur les environnements au sein d'une organisation.

11.1.1.4. Véritable Portabilité (Universelle)

Tout le monde peut télécharger le code source d'un projet véritablement portable, le compiler et l'installer sans avoir à configurer le build pour son environnement spécifique. C'est le plus haut niveau de portabilité ; aucun travail supplémentaire n'est nécessaire pour construire ce projet. Ce niveau de portabilité est important surtout pour les projets libres qui dépendent de la facilité avec laquelle les contributeurs potentiels vont pouvoir construire le projet à partir du code source téléchargé.

N'importe quel développeur peut télécharger le code source d'un projet véritablement portable.

11.1.2. Choisir le bon niveau de portabilité

Évidemment, vous voulez éviter le pire cas : le build non-portable. Vous avez peut-être connu le malheur de travailler ou d'étudier pour une organisation dont les builds des applications critiques sont non-portables. Dans une organisation comme celle-là, impossible de déployer une application sans une personne bien spécifique sur une machine bien spécifique elle aussi. Avec une telle organisation, il

est très difficile d'introduire de nouvelles dépendances ou des changements sans se coordonner avec la personne qui gère ce build non-portable. Ces builds non-portables ont tendances à se développer dans des environnements très politiques où une personne ou un groupe veut contrôler quand et comment un projet est construit et déployé. "Comment construit-on le système ? Oh, nous devons appeler Jack et lui demander de nous le construire, personne d'autre ne peut déployer en production". C'est une situation périlleuse qui est beaucoup plus fréquente qu'on ne le pense. Si vous travaillez dans cette organisation, Maven et les profils Maven sont votre porte de sortie.

À l'autre bout du spectre de la portabilité se trouvent les builds très portables. Ces builds sont des builds très difficiles à réaliser. Ils restreignent vos dépendances aux projets et aux outils qui sont librement distribuables et facilement accessibles. De nombreux packages commerciaux doivent être exclus des builds les plus portables car ils ne peuvent être téléchargés sans que vous n'ayez à accepter une licence spécifique. Cette large portabilité va également restreindre les dépendances aux logiciels qui sont distribués sous la forme d'artefacts Maven. Par exemple, si vous dépendez des pilotes JDBC d'Oracle, vos utilisateurs devront les télécharger et les installer manuellement ; ce n'est pas très portable car vous devrez fournir les instructions nécessaires pour configurer l'environnement afin que les personnes intéressées puissent construire votre application. D'un autre côté, vous pourriez utiliser un pilote JDBC disponible sur le dépôt public de Maven comme MySQL ou HSQLDB.

Comme nous l'avons vu précédemment, les projets libres ont intérêt à avoir des builds le plus portable possible. Les builds très portables réduisent le coût de contribution d'un projet. Dans un projet libre (comme Maven), on trouve deux groupes d'utilisateurs très distincts : les développeurs et les utilisateurs finaux. Quand un utilisateur final utilise Maven et qu'il décide de contribuer en proposant un patch à Maven, d'utilisateur du produit final d'un build, il doit devenir capable de construire ce produit par lui-même. Il doit donc tout d'abord se transformer en développeur, mais il risque de perdre de sa motivation à contribuer au projet s'il dépense trop d'énergie à apprendre à le construire. Avec un projet très portable, un utilisateur n'a pas à connaître les arcanes du build d'un projet pour se transformer en développeur, il peut télécharger le code source, le modifier, construire son artefact et proposer sa contribution sans avoir à demander de l'aide pour configurer son environnement. Plus le coût d'entrée pour contribuer à un projet libre est bas et plus le nombre de contributions augmente, surtout ces petites contributions qui font la différence pour la réussite d'un projet. Une des conséquences de l'adoption de Maven par un grand nombre de projets Open Source est que contribuer à ces projets est devenu beaucoup plus facile.

11.2. Portabilité grâce aux profils Maven

Un profil dans Maven est un ensemble alternatif de valeurs qui définissent ou surchargent les valeurs par défaut. En utilisant un profil, vous pouvez personnaliser un build pour différents environnements. Les profils sont configurés dans le fichier `pom.xml` et possèdent un identifiant. Vous pouvez ensuite exécuter Maven en précisant sur la ligne de commande le profil à utiliser. Le `pom.xml` suivant utilise un profil `production` pour écraser les paramètres par défaut du plugin Compiler.

Exemple 11.1. Surcharge des paramètres de compilation en production par un profil Maven

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                    http://maven.apache.org/maven-v4_0_0.xsd">
<modelVersion>4.0.0</modelVersion>
<groupId>org.sonatype.mavenbook</groupId>
<artifactId>simple</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>simple</name>
<url>http://maven.apache.org</url>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>3.8.1</version>
    <scope>test</scope>
  </dependency>
</dependencies>
<profiles>
  <profile>
    <id>production</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <debug>>false</debug>
            <optimize>>true</optimize>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
</project>

```

Dans cet exemple, nous avons ajouté un profil nommé `production` pour surcharger la configuration par défaut du plugin Maven Compiler. Examinons la syntaxe de ce profil en détail.

- ❶ L'élément `profiles` est dans le `pom.xml`, il contient un ou plusieurs éléments `profile`. Puisque les profils écrasent les paramètres par défaut du `pom.xml`, l'élément `profiles` est habituellement positionné en fin de `pom.xml`.
- ❷ Chaque profil doit avoir un élément `id`. Cette balise `id` contient le nom qui est utilisé pour invoquer ce profil en ligne de commande. Un profil est invoqué en passant l'argument `-P<profile_id>` à la ligne de commande Maven.
- ❸ Un élément `profile` peut contenir de nombreux éléments présents dans l'élément `project` d'un POM. Dans cet exemple, nous surchargeons le comportement du plugin Compiler et nous devons écraser la configuration du plugin qui se trouve normalement dans l'élément `build/plugins`.

- 4 Nous surchargeons la configuration du plugin Maven Compiler. Nous nous assurons que le bytecode produit par le profil de production ne contient pas les informations de débogage et qu'il a été optimisé par le compilateur.

Pour exécuter la commande `mvn install` avec le profil de `production`, vous devez passer l'argument `-Pproduction` en ligne de commande. Pour vérifier que le profil de `production` surcharge la configuration par défaut du plugin Compiler, exécutez Maven avec les options de debug (`-X`) activées :

```
~/examples/profile $ mvn clean install -Pproduction -X
... (omitting debugging output) ...
[DEBUG] Configuring mojo 'o.a.m.plugins:maven-compiler-plugin:2.0.2:testCompile'
[DEBUG] (f) basedir = ~\examples\profile
[DEBUG] (f) buildDirectory = ~\examples\profile\target
...
[DEBUG] (f) compilerId = javac
[DEBUG] (f) debug = false
[DEBUG] (f) failOnError = true
[DEBUG] (f) fork = false
[DEBUG] (f) optimize = true
[DEBUG] (f) outputDirectory = \
    ~\svn\sonatype\examples\profile\target\test-classes
[DEBUG] (f) outputFileName = simple-1.0-SNAPSHOT
[DEBUG] (f) showDeprecation = false
[DEBUG] (f) showWarnings = false
[DEBUG] (f) staleMillis = 0
[DEBUG] (f) verbose = false
[DEBUG] -- end configuration --
... (omitting debugging output) ...
```

Cet extrait de la sortie en mode debug de Maven nous montre la configuration du plugin Compiler avec le profil de production. Comme nous pouvons le voir, la variable `debug` est à `false` et la variable `optimize` est à `true`.

11.2.1. Surcharger un POM

L'exemple précédent vous a montré comment surcharger la configuration par défaut d'un plugin Maven seul, cependant vous ignorez toujours ce qu'il est possible de faire avec un profil Maven. Pour faire court, un profil Maven peut surcharger presque tout ce que vous pouvez trouver dans un fichier `pom.xml`. Le POM Maven possède une balise appelée `profiles` qui contient les différentes configurations d'un projet, et dans cette balise se trouve une balise `profile` qui définit chacun d'entre eux. Chaque profil doit avoir une balise `id`, cela mit à part, il peut contenir presque tous les éléments que l'on peut trouver sous la balise `project`. Le document XML qui suit nous montre toutes les balises qu'un profil peut surcharger.

Exemple 11.2. Balises autorisées dans un profil

```
<project>
  <profiles>
    <profile>
```

```

<build>
  <defaultGoal>...</defaultGoal>
  <finalName>...</finalName>
  <resources>...</resources>
  <testResources>...</testResources>
  <plugins>...</plugins>
</build>
<reporting>...</reporting>
<modules>...</modules>
<dependencies>...</dependencies>
<dependencyManagement>...</dependencyManagement>
<distributionManagement>...</distributionManagement>
<repositories>...</repositories>
<pluginRepositories>...</pluginRepositories>
<properties>...</properties>
</profile>
</profiles>
</project>

```

Un profil peut donc redéfinir tous ces éléments. Il peut redéfinir le nom de l'artefact final, les dépendances et le build d'un projet en surchargeant la configuration des plugins. Un profil peut aussi redéfinir les paramètres de distribution. Par exemple, si pour une étape de validation vous devez publier vos artefacts sur un serveur de pré-production, vous allez créer un profil de validation qui va surcharger la balise `distributionManagement`.

11.3. Activation de profil

Dans la section précédente, nous vous avons présenté comment créer un profil qui redéfinit le comportement par défaut selon un environnement cible spécifique. Le build précédent était configuré par défaut pour le développement, et le profil `production` existe pour apporter les éléments de configuration propres à l'environnement de production. Que se passe-t'il si vous devez adapter votre build selon le système d'exploitation ou la version du JDK ? Maven fournit un mécanisme pour "activer" un profil selon différents paramètres liés à l'environnement, c'est ce qu'on appelle l'activation de profil.

Prenons l'exemple suivant, supposons que nous avons une bibliothèque Java dont certaines fonctions ne sont disponibles que pour Java 6 et son moteur de scripts défini dans la JSR-223¹. Une fois que vous avez séparé le bloc de code qui utilise le moteur de scripts dans un projet Maven à part, vous voulez que les personnes qui utilisent Java 5 puissent construire votre projet sans essayer de construire l'extension spécifique Java 6. Vous pouvez faire cela au moyen d'un profil Maven qui ajoute le module de gestion des scripts uniquement lorsque le build se fait sur un JDK Java 6. Tout d'abord, jetons un oeil à la structure des répertoires de notre projet et comment nous souhaitons que les développeurs construisent notre système.

Lorsque l'on exécute la commande `mvn install` avec un JDK Java 6, nous voulons que le build construise le projet `simple-script`. Si par contre cette commande est exécutée sur Java 5 il faut

¹ <http://jcp.org/en/jsr/detail?id=223>

exclure le projet `simple-script` du build. Si vous n'excluez pas le projet `simple-script` de votre build en Java 5, celui-ci va échouer car Java 5 ne fournit pas la classe `ScriptEngine`. Étudions plus en détail le fichier `pom.xml` du projet de bibliothèque :

Exemple 11.3. Inclusion dynamique de sous-modules par activation de profil

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>simple</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>simple</name>
  <url>http://maven.apache.org</url>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <profiles>
    <profile>
      <id>jdk16</id>
      <activation>
        <jdk>1.6</jdk>
      </activation>
      <modules>
        <module>simple-script</module>
      </modules>
    </profile>
  </profiles>
</project>
```

Si vous exécutez la commande `mvn install` en Java 6, vous vous apercevrez que Maven descend dans le sous-répertoire `simple-script` pour y construire le projet `simple-script`. Si vous exécutez la même commande `mvn install` en Java 5, le build n'essayera pas de construire le sous-module `simple-script`. Analysons cette configuration d'activation en détail :

- 1 La balise `activation` indique les conditions d'activation du profil. Dans cet exemple, nous avons indiqué que ce profil sera activé pour les versions de Java à partir de Java "1.6". Ce qui inclut donc "1.6.0_03", "1.6.0_02" ou tout autre version commençant par "1.6". Les conditions d'activation ne sont pas limitées à la version de Java, pour une liste complète des paramètres d'activation rendez-vous à la section Configuration de l'activation.
- 2 Nous ajoutons le module `simple-script` dans ce profil. L'ajout de ce module fait que Maven va aller regarder dans le sous-répertoire `simple-script/` à la recherche d'un fichier `pom.xml`.

11.3.1. Configuration de l'activation

L'activation contient une ou plusieurs conditions sur les versions du JDK, les systèmes d'exploitation ou des propriétés. Un profil est activé lorsque toutes les conditions d'activation sont satisfaites. Par exemple, un profil peut indiquer comme conditions d'être sur un système d'exploitation Windows avec un JDK version 1.4. Ce profil ne sera donc activé que si le build est exécuté sur une machine Windows avec Java 1.4. Si le profil est actif, alors tous ses éléments surchargent les éléments correspondants du projet comme si ce profil était inclus via l'argument `-P` en ligne de commande. L'exemple suivant présente un profil qui n'est activé que par une combinaison très spécifique de propriétés, de version du JDK et de système d'exploitation.

Exemple 11.4. Paramètres d'activation du profil : version du JDK, système d'exploitation et propriétés

```
<project>
...
<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <activeByDefault>>false</activeByDefault>❶
      <jdk>1.5</jdk>❷
      <os>
        <name>Windows XP</name>❸
        <family>Windows</family>
        <arch>x86</arch>
        <version>5.1.2600</version>
      </os>
      <property>
        <name>mavenVersion</name>❹
        <value>2.0.5</value>
      </property>
      <file>
        <exists>file2.properties</exists>❺
        <missing>file1.properties</missing>
      </file>
    </activation>
    ...
  </profile>
</profiles>
</project>
```

L'exemple précédent définit un ensemble très précis de paramètres d'activation. Regardons chacun d'entre eux plus en détail :

- ❶ La balise `activeByDefault` contrôle si ce profil est actif par défaut.
- ❷ Ce profil n'est actif que pour les versions du JDK commençant par "1.5". Ce qui inclut "1.5.0_01" et "1.5.1".

- ③ Ce profil cible aussi une version très spécifique de Windows XP, la version 5.1.2600 sur une plateforme 32-bit. Si votre projet utilise le plugin natif pour compiler du code en C, vous risquez de vous retrouver à écrire plusieurs projets suivant les plateformes.
- ④ La balise `property` indique à Maven qu'il doit activer ce profil si la propriété `mavenVersion` a pour valeur `2.0.5`. La propriété `mavenVersion` est une propriété implicite qui est disponible pour tous les builds Maven.
- ⑤ La balise `file` nous permet d'activer un profil sur la présence (ou l'absence) d'un fichier. Le profil `dev` sera activé si un fichier `file2.properties` existe à la racine du projet. Le profil `dev` ne sera activé que s'il n'existe pas de fichier `file1.properties` à la racine du projet.

11.3.2. Activation par l'absence d'une propriété

Vous pouvez activer un profil sur la valeur d'une propriété comme `environment.type`. Vous pouvez donc activer un profil `development` si la propriété `environment.type` a pour valeur `dev` et un profil `production` si cette propriété vaut maintenant `prod`. Il est aussi possible d'activer un profil en cas d'absence d'une propriété. La configuration suivante active un profil si la propriété `environment.type` n'est pas présente durant l'exécution de Maven.

Exemple 11.5. Activation de profils en cas d'absence d'une propriété

```
<project>
...
<profiles>
  <profile>
    <id>development</id>
    <activation>
      <property>
        <name>!environment.type</name>
      </property>
    </activation>
  </profile>
</profiles>
</project>
```

Attention au point d'exclamation qui préfixe le nom de la propriété. Le point d'exclamation est souvent appelé le caractère "bang" ce qui signifie "non". Ce profil est activé quand aucune propriété `environment.type` n'est définie.

11.4. Lister les profils actifs

Les profils Maven peuvent être définis soit dans un fichier `pom.xml`, dans un fichier `profiles.xml`, dans le fichier `~/.m2/settings.xml` ou dans le fichier `${M2_HOME}/conf/settings.xml`. Avec ces quatre possibilités, il est difficile de savoir quels sont les profils disponibles pour un projet sans avoir à parcourir les quatre fichiers en question. Pour faciliter la gestion des profils, connaître ceux qui sont disponibles et où ils sont définis, il existe le goal du plugin Maven Help `active-profiles` qui liste

l'ensemble des profils actifs et où ils sont définis. Voici les instructions pour exécuter le goal `active-profiles` :

```
$ mvn help:active-profiles
Active Profiles for Project 'My Project':

The following profiles are active:

- my-settings-profile (source: settings.xml)
- my-external-profile (source: profiles.xml)
- my-internal-profile (source: pom.xml)
```

11.5. Trucs et Astuces

Les profils encouragent la portabilité du build. Si vous avez besoin de configurer avec délicatesse votre build pour qu'il puisse s'exécuter sur différentes plateformes ou pour construire différents artefacts selon la plateforme cible, alors les profils peuvent améliorer sa portabilité. Les profils définis dans les fichiers `settings.xml` diminuent la portabilité d'un build puisque les développeurs doivent échanger cette information supplémentaire. Les sections qui vont suivre présentent des guides et des suggestions pour l'utilisation de profils Maven dans votre projet.

11.5.1. Environnements communs

Une des principales raisons à l'utilisation de profils Maven est de fournir la configuration spécifique à un environnement. Dans un environnement de développement, vous voudrez produire du bytecode avec les informations nécessaires pour le débogage et vous voudrez configurer votre système pour qu'il utilise une base de données de développement. Dans un environnement de production, vous souhaitez produire un JAR signé et configurer votre système pour qu'il utilise la base de données de production. Dans ce chapitre, nous avons défini un certain nombre d'environnements avec des identifiants comme `dev` et `prod`. On peut faire plus simple en définissant des profils qui seront activés par des propriétés de l'environnement et en utilisant ces propriétés pour tous vos projets. Par exemple, si chaque projet a un profil `development` activé par une propriété appelée `environment.type` ayant pour valeur `dev`, et si ces mêmes projets avaient un profil `production` activé par la présence d'une propriété `environment.type` ayant pour valeur `prod`, vous pourriez alors créer un profil dans votre fichier `settings.xml` qui définirait une propriété `environment.type` avec pour valeur `dev` sur votre machine de développement. Ainsi, chaque projet doit définir un profil `dev` activé par la même variable d'environnement. Voyons comment appliquer tout cela : le fichier `settings.xml` suivant définit un profil dans `~/ .m2/settings.xml` qui spécifie la valeur `dev` pour la propriété `environment.type`.

Exemple 11.6. Le fichier `~/ .m2/settings.xml` définit un profil par défaut qui spécifie la propriété `environment.type`

```
<settings>
  <profiles>
    <profile>
      <activation>
        <activeByDefault>true</activeByDefault>
      </activation>
    </profile>
  </profiles>
</settings>
```



```

    </activation>
    <properties>
      <environment.type>dev</environment.type>
    </properties>
  </profile>
</profiles>
</settings>

```

Lorsque vous exécutez Maven sur votre poste avec cette configuration, ce profil sera activé et la propriété `environment.type` aura pour valeur `dev`. Vous pouvez utiliser cette propriété pour activer les profils définis dans le fichier `pom.xml` d'un projet comme nous allons le voir. Regardons donc, comment le fichier `pom.xml` définirait un profil qui serait activé par le fait que la propriété `environment.type` ait pour valeur `dev`.

Exemple 11.7. Profil d'un projet activé quand `environment.type` vaut 'dev'

```

<project>
  ...
  <profiles>
    <profile>
      <id>development</id>
      <activation>
        <property>
          <name>environment.type</name>
          <value>dev</value>
        </property>
      </activation>
      <properties>
        <database.driverClassName>com.mysql.jdbc.Driver</database.driverClassName>
        <database.url>
          jdbc:mysql://localhost:3306/app_dev
        </database.url>
        <database.user>development_user</database.user>
        <database.password>development_password</database.password>
      </properties>
    </profile>
    <profile>
      <id>production</id>
      <activation>
        <property>
          <name>environment.type</name>
          <value>prod</value>
        </property>
      </activation>
      <properties>
        <database.driverClassName>com.mysql.jdbc.Driver</database.driverClassName>
        <database.url>jdbc:mysql://master01:3306,slave01:3306/app_prod</database.url>
        <database.user>prod_user</database.user>
      </properties>
    </profile>
  </profiles>
</project>

```

Ce projet définit de nouvelles propriétés comme `database.url` et `database.user` qui pourraient être utilisées pour configurer un plugin Maven ailleurs dans le fichier `pom.xml`. Il existe de nombreux plugins qui peuvent manipuler une base de données, exécuter du SQL, ou comme le plugin Maven Hibernate3, peuvent générer un ensemble d'objets annotés utilisés par les frameworks de persistance. Certains de ces plugins peuvent être configurés dans un fichier `pom.xml` grâce à ces propriétés. Ces propriétés peuvent aussi être utilisées pour filtrer des ressources. Dans cet exemple, comme nous avons défini un profil dans le fichier `~/m2/settings.xml` qui spécifie la valeur `dev` pour `environment.type`, le profil de développement sera toujours actif à chaque exécution de Maven sur cette machine. Par contre, si nous voulions modifier ce comportement par défaut, nous pourrions préciser la valeur de cette propriété en ligne de commande. Si nous avons besoin d'activer le profil de production, nous pourrions exécuter Maven avec la commande :

```
~/examples/profiles $ mvn install -Denvironment.type=prod
```

Spécifier une propriété en ligne de commande surchargera la valeur par défaut définie dans le fichier `~/m2/settings.xml`. Nous aurions pu aussi définir un profil avec un `id` "dev" et l'invoquer directement avec l'argument `-P` en ligne de commande, mais l'utilisation de cette propriété `environment.type` nous permet d'écrire de nouveaux fichiers `pom.xml` en respectant ce standard. Tous vos projets pourraient posséder un profil qui serait activé par la même propriété `environment.type` définie dans les fichiers `~/m2/settings.xml` de chaque développeur. Ainsi, les développeurs peuvent partager une configuration commune pour le développement sans avoir à la définir dans des fichiers `settings.xml` non-portables.

11.5.2. Protéger les mots de passe

Cette bonne pratique est la conséquence de la section précédente. Dans la partie Profil d'un projet activé quand `environment.type` vaut 'dev', le profil de production ne contient pas la propriété `database.password`. J'ai fait cela délibérément pour illustrer le fait de stocker les mots de passe dans votre fichier `settings.xml`. Si vous développez une application dans une grande organisation qui accorde de l'importance à la sécurité, il est probable que la majorité des développeurs ne connaissent pas le mot de passe de production de la base de données. C'est la norme dans une organisation qui trace des lignes infranchissables entre équipe de développement et équipe de production. Les développeurs ont accès aux environnements de développement et de pré-production, mais ils pourraient ne pas avoir (ou ne pas vouloir) accès à la base de données de production. Il y a un certain nombre de raisons pour lesquelles cela aurait un sens, particulièrement, si une organisation gère des données sensibles financières, médicales ou autres. Dans ce scénario, l'environnement de production ne pourrait être utilisé que par le responsable technique ou par un membre de l'équipe de production. Quand ils exécutent le build avec la valeur `prod` pour la propriété `environment.type`, ils vont devoir définir cette variable dans leur fichier `settings.xml` comme suit :

Exemple 11.8. Enregistrement de mots de passe dans un profil du fichier `settings.xml` propre à l'utilisateur

```
<settings>
  <profiles>
```

```

<profile>
  <activeByDefault>true</activeByDefault>
  <properties>
    <environment.type>prod</environment.type>
    <database.password>m1ss10nimp0ss1b13</database.password>
  </properties>
</profile>
</profiles>
</settings>

```

Cet utilisateur a défini un profil par défaut qui donne à `environment.type` la valeur `prod` et qui spécifie le mot de passe de production. Quand le build du projet est exécuté, le profil de production est activé par la propriété `environment.type` et la propriété `database.password` est remplie. Ainsi, vous pouvez mettre toute la configuration spécifique à la production dans le fichier `pom.xml` du projet et retirer seulement le mot de passe nécessaire pour accéder à la base de données de production.



Note

Ces informations privées sont en général contraires à la portabilité, mais ce que nous venons de faire a du sens. Vous ne voudriez pas partager vos secrets avec n'importe qui.

11.5.3. Classifieurs de plateforme

Supposons que vous avez une bibliothèque ou un projet qui produit des artefacts spécifiques selon la plateforme. Même si Java est indépendant de la plateforme, parfois vous pouvez avoir besoin d'écrire du code qui invoque du code natif, spécifique à une plateforme. Vous pouvez aussi avoir écrit du code C compilé avec le plugin Maven Native et que vous voulez produire un artefact qualifié selon la plateforme sur laquelle il a été construit. Vous pouvez définir un classificateur avec le plugin Maven Assembly ou le plugin Maven Jar. Le fichier `pom.xml` suivant produit un artefact qualifié grâce à des profils activés en fonction du système d'exploitation. Pour plus d'informations sur le plugin Maven Assembly vous pouvez consulter le Chapitre 14, Maven Assemblies.

Exemple 11.9. Qualification d'artefacts avec des profils activés selon la plateforme

```

<project>
  ...
  <profiles>
    <profile>
      <id>windows</id>
      <activation>
        <os>
          <family>windows</family>
        </os>
      </activation>
      <build>
        <plugins>
          <plugin>
            <artifactId>maven-jar-plugin</artifactId>
            <configuration>

```

```

        <classifier>win</classifier>
    </configuration>
</plugin>
</plugins>
</build>
</profile>
<profile>
    <id>linux</id>
    <activation>
        <os>
            <family>unix</family>
        </os>
    </activation>
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <classifier>linux</classifier>
                </configuration>
            </plugin>
        </plugins>
    </build>
</profile>
</profiles>
</project>

```

Si le système d'exploitation fait partie de la famille Windows, ce fichier `pom.xml` qualifie l'artefact Jar avec "-win". Si le système d'exploitation est un membre de la famille Unix, l'artefact est qualifié avec "-linux". Ce fichier `pom.xml` permet donc d'ajouter des classificateurs aux artefacts, mais il est plus verbeux que nécessaire, car il recopie la configuration du plugin Maven Jar dans les deux profils. Cet exemple pourrait être réécrit en utilisant des variables pour réduire la redondance :

Exemple 11.10. Qualification des artefacts avec des profils activés selon la plateforme d'exécution et en utilisant des variables

```

<project>
    ...
    <build>
        <plugins>
            <plugin>
                <artifactId>maven-jar-plugin</artifactId>
                <configuration>
                    <classifier>${envClassifier}</classifier>
                </configuration>
            </plugin>
        </plugins>
    </build>
    ...
    <profiles>
        <profile>

```

```

<id>windows</id>
<activation>
  <os>
    <family>windows</family>
  </os>
</activation>
<properties>
  <envClassifier>win</envClassifier>
</properties>
</profile>
<profile>
  <id>linux</id>
  <activation>
    <os>
      <family>unix</family>
    </os>
  </activation>
  <properties>
    <envClassifier>linux</envClassifier>
  </properties>
</profile>
</profiles>
</project>

```

Dans ce fichier `pom.xml`, chaque profil n'a pas besoin d'inclure une balise `build` pour configurer le plugin Jar. Au lieu de cela, le profil est activé par la famille à laquelle appartient le système d'exploitation, et valorise la propriété `envClassifier` soit à `win` soit à `linux`. Cette propriété `envClassifier` est référencée dans la balise `build` du fichier `pom.xml` pour ajouter un classificateur à l'artefact JAR produit par le projet. Ainsi l'artefact JAR produit s'appellera `${finalName}-${envClassifier}.jar` et sera utilisé comme dépendance grâce à la syntaxe suivante :

Exemple 11.11. Dépendance vers un artefact qualifié

```

<dependency>
  <groupId>com.mycompany</groupId>
  <artifactId>my-project</artifactId>
  <version>1.0</version>
  <classifier>linux</classifier>
</dependency>

```

11.6. En résumé

Lorsqu'ils sont utilisés judicieusement, les profils facilitent grandement la configuration du build pour différentes plateformes. Si quelque part dans votre build vous avez besoin de définir un chemin spécifique à une plateforme pour un serveur d'applications, vous pouvez mettre ces éléments de configuration dans un profil qui sera activé selon la nature de votre système d'exploitation. Si vous avez un projet qui doit produire différents artefacts pour différents environnements, vous pouvez personnaliser le comportement du build selon les différents environnements et plateformes par des

profils spécifiques à ceux-ci. L'utilisation de profils permet de rendre les builds portables, il n'est plus nécessaire de réécrire votre logique de construction pour l'adapter à un nouvel environnement. Surchargez la configuration qui doit être modifiée et partagez celle qui peut l'être.

-amd,--also-make-dependents	If project list is specified, also build projects that depend on projects on the list
-B,--batch-mode	Run in non-interactive (batch) mode
-C,--strict-checksums	Fail the build if checksums don't match
-c,--lax-checksums	Warn if checksums don't match
-cpu,--check-plugin-updates	Force upToDate check for any relevant registered plugins
-D,--define <arg>	Define a system property
-e,--errors	Produce execution error messages
-emp,--encrypt-master-password <arg>	Encrypt master security password
-ep,--encrypt-password <arg>	Encrypt server password
-f,--file	Force the use of an alternate POM file.
-fae,--fail-at-end	Only fail the build afterwards; allow all non-impacted builds to continue
-ff,--fail-fast	Stop at first failure in reactorized builds
-fn,--fail-never	NEVER fail the build, regardless of project result
-gs,--global-settings <arg>	Alternate path for the global settings file
-h,--help	Display help information
-N,--non-recursive	Do not recurse into sub-projects
-npr,--no-plugin-registry	Don't use ~/m2/plugin-registry.xml for plugin versions
-npu,--no-plugin-updates	Suppress upToDate check for any relevant registered plugins
-o,--offline	Work offline
-P,--activate-profiles <arg>	Comma-delimited list of profiles to activate
-pl,--projects <arg>	Build specified reactor projects instead of all projects
-q,--quiet	Quiet output - only show errors
-r,--reactor	Dynamically build reactor from subdirectories
-rf,--resume-from <arg>	Resume reactor from specified project
-s,--settings <arg>	Alternate path for the user settings file
-U,--update-snapshots	Forces a check for updated releases and snapshots on remote repositories
-up,--update-plugins	Synonym for cpu
-V,--show-version	Display version information WITHOUT stopping build
-v,--version	Display version information
-X,--debug	Produce execution debug output

Si vous voulez de l'aide sur les goals et les paramètres disponibles pour un plugin Maven spécifique, allez à la Section 12.3, « Usage du plugin Maven Help ».

12.1.3. Utilisation de profils de build

Pour activer un ou plusieurs profils depuis la ligne de commande, il vous faut utiliser l'option suivante :

```
-P, --activate-profiles <arg>  
    Liste de profils à activer séparés par des virgules
```

Pour plus d'informations sur les profils de build, rendez-vous au Chapitre 11, Profils de Build.

12.1.4. Afficher les informations relatives à la version

Pour afficher les informations sur la version de Maven, voici l'option à utiliser :

```
-V, --show-version  
    Affiche la version SANS arrêter le build en cours  
  
-v, --version  
    Affiche la version
```

Ces deux options affichent le même résultat, cependant l'option `-v` va arrêter Maven après avoir affiché la version. Utilisez l'option `-V` si vous voulez afficher les informations sur la version de Maven au début des traces de votre build. Cela peut-être pratique si vous exécutez Maven dans un environnement d'intégration continue et que vous voulez connaître quelle version de Maven a été utilisée pour un build particulier.

Exemple 12.1. Informations relatives à la version de Maven

```
$ mvn -v  
Apache Maven 2.2.1 (r801777; 2009-08-06 14:16:01-0500)  
Java version: 1.6.0_15  
Java home: /System/Library/Frameworks/JavaVM.framework/Versions/1.6.0/Home  
Default locale: en_US, platform encoding: MacRoman  
OS name: "mac os x" version: "10.6.1" arch: "x86_64" Family: "mac"
```

12.1.5. Travailler en mode déconnecté

Si vous devez utiliser Maven sans connexion réseau, vous devez utiliser l'option suivante afin d'éviter que Maven vérifie les éventuelles mises à jour des plugins ou des dépendances via le réseau :

```
-o, --offline  
    Travailler en mode déconnecté
```

Quand Maven s'exécute avec cette option, il n'essaye pas de se connecter à un dépôt distant pour récupérer des artefacts.

12.1.6. Utiliser le POM et le fichier settings de votre choix

Si vous n'aimez pas le nom de votre fichier `pom.xml`, l'endroit où sont enregistrés vos préférences Maven ou le répertoire par défaut où se trouve le fichier de configuration globale de Maven, vous pouvez personnaliser tout cela avec les options suivantes :

- f, --file <file>
Force l'utilisation d'un fichier POM alternatif
- s, --settings <arg>
Chemin alternatif vers votre fichier de configuration personnel
- gs, --global-settings <file>
Chemin alternatif vers le fichier de configuration global

12.1.7. Chiffrer les mots de passe

Les commandes Maven suivantes vous permettent de chiffrer les mots de passe enregistrés dans vos fichiers de préférences Maven :

- emp, --encrypt-master-password <password>
Chiffre le mot de passe principal
- ep, --encrypt-password <password>
Chiffre le mot de passe du serveur

Le chiffrement des mots de passe est décrit plus en détails dans la Section A.2.11, « Chiffrement des mots de passe dans les Settings Maven ».

12.1.8. Gestion des erreurs

Les options suivantes permettent de contrôler le comportement de Maven lors de l'échec d'un build en plein milieu d'un build multimodule :

- fae, --fail-at-end
Le build n'échoue qu'à la fin ; tous les modules non-impactés sont construits
- ff, --fail-fast
Arrêt au premier échec dans les builds
- fn, --fail-never
AUCUN échec, quelque soit le résultat de la construction du projet

Les options `-fn` et `-fae` sont utiles pour les builds multimodules qui sont exécutés par un outil d'intégration continue comme Hudson. L'option `-ff` est très utile pour les développeurs qui exécutent des builds interactifs et qui veulent rapidement des retours durant le développement.

12.1.9. Contrôle de la verbosité de Maven

Si vous voulez contrôler le niveau de trace de Maven, vous pouvez utiliser une de ces trois options de ligne de commande :

- e, --errors
Affiche les messages d'erreur lors de l'exécution
- X, --debug
Affiche les traces en mode debug
- q, --quiet
Mode silencieux - n'affiche que les erreurs

Avec l'option -q un message n'est affiché que lorsqu'il y a une erreur ou un problème.

Avec l'option -x vous serez submergé de messages de debug. Cette option est surtout utilisée par ceux qui développent Maven et ses plugins pour diagnostiquer un problème avec du code Maven lors du développement. Cette option -x est très utile lorsque vous essayez de trouver la cause d'un problème très délicat avec une dépendance ou un classpath.

L'option -e est pratique si vous êtes un développeur Maven ou si vous avez besoin de diagnostiquer une erreur dans un plugin Maven. Si vous remontez une anomalie sur Maven ou l'un de ses plugins, n'oubliez pas de fournir les traces obtenues avec les deux options -x et -e.

12.1.10. Exécution de Maven en mode batch

Pour exécuter Maven en mode batch, utilisez l'option suivante :

- B, --batch-mode
Exécution en mode non-interactif (batch)

Le mode batch est nécessaire lorsque vous devez exécuter Maven dans un environnement d'intégration continue sans interactions possibles. Quand vous exécutez Maven en mode non-interactif, celui-ci n'accepte aucune entrée utilisateur. Au lieu de cela, il utilise les valeurs par défaut 'intelligentes' lorsqu'il a besoin d'entrées.

12.1.11. Téléchargement et vérification des dépendances

Les options suivantes de la ligne de commande modifient le comportement de Maven vis-à-vis des dépôts distants ainsi que la vérification des artefacts téléchargés :

- C, --strict-checksums
Fait échouer le build si les checksums ne correspondent pas
- c, --lax-checksums
Affiche une alerte si les checksums ne correspondent pas

`-U, --update-snapshots`

Force la recherche de mises à jour de versions stables et snapshots sur les dépôts distants

Si pour vous la sécurité est importante, utilisez l'option `-C` à l'exécution de Maven. Les dépôts Maven possèdent un checksum MD5 et SHA1 pour chaque artefact stocké. Maven est configuré pour avertir l'utilisateur final lorsque le checksum ne correspond pas à l'artefact téléchargé. L'utilisation de l'option `-c` fera échouer le build si Maven rencontre un artefact avec un mauvais checksum.

L'option `-σ` est très utile lorsque vous voulez vous assurer que Maven vérifie qu'il utilise bien les dernières versions stables et SNAPSHOT des dépendances.

12.1.12. Contrôle de la mise à jour des plugins

Les options de ligne de commande suivantes indiquent à Maven s'il doit (ou pas) mettre à jour les plugins à partir des dépôts distants :

`-npu, --no-plugin-updates`

Élimine la recherche de mises à jour pour tous les plugins enregistrés. L'utilisation de cette option aura pour effet de fixer Maven sur les versions des plugins disponibles dans le dépôt local. Avec l'option `-npu`, Maven n'ira pas consulter les dépôts distants à la recherche de mises à jour.

`-cpu, --check-plugin-updates`

Force la recherche de mises à jour pour les plugins enregistrés. Force Maven à vérifier l'existence de nouvelle version stable d'un plugin Maven. Faites attention au fait que cela n'affectera pas vos builds si vous spécifiez explicitement les versions des plugins Maven dans le POM de votre projet.

`-up, --update-plugins`

Synonyme de `cpu`.

Les options suivantes modifient la manière dont Maven télécharge les plugins depuis un dépôt distant :

`-npr, --no-plugin-registry`

Ne pas utiliser le fichier `~/ .m2/plugin-registry.xml` pour les versions de plugin.

Lorsqu'elle est utilisée, l'option `-npr` demande à Maven de ne pas consulter le Registre de plugins. Pour plus de détails sur le Registre de plugins, consultez la page : <http://maven.apache.org/guides/introduction/introduction-to-plugin-registry.html>.

12.1.13. Builds non-récurifs

Parfois vous voudrez exécuter Maven sans qu'il descende dans tous les sous-modules d'un projet. Vous pouvez obtenir cela grâce à l'option suivante :

`-N, --non-recursive`

Empêche Maven de construire les sous-modules. Ne construit que le projet contenu dans le répertoire courant.

Avec cette option Maven exécutera un goal ou une étape du cycle de vie du projet dans le répertoire courant. Maven n'essayera pas de construire tous les projets d'un build multimodule quand vous utilisez l'option `-n`.

12.2. Utilisation des options avancées du Reactor

Depuis la version 2.1 de Maven, il existe de nouvelles options qui vous permettent de manipuler la façon dont Maven va construire des projets multimodule. Si vous utilisez Maven 2.0, ces nouveaux arguments ne sont pas disponibles. Ces nouvelles options sont :

- `-r, --reactor`
Construit le reactor dynamiquement depuis les sous-répertoires
- `-rf, --resume-from`
Reprend le reactor depuis le projet spécifié
- `-pl, --projects`
Construit le reactor spécifié plutôt que tous les projets
- `-am, --also-make`
Si une liste de projets est spécifiée, construit aussi tous les projets demandés par cette liste
- `-amd, --also-make-dependents`
Si une liste de projets est spécifiée, construit aussi tous les projets dont dépendent les projets de cette liste

12.2.1. Reprise de build

Supposons que nous soyons en train de travailler sur du code et que nous essayions d'exécuter `mvn install` depuis `simple-parent`, et que nous ayons un test en échec dans `simple-weather`. Nous corrigeons donc `simple-weather` sans changer `simple-model` ; nous savons que `simple-model` est bon et que donc il n'est pas nécessaire de le reconstruire ou de le tester. Nous pouvons alors utiliser l'argument `--resume-from` ainsi :

```
$ mvn --resume-from simple-weather install
```

Ainsi, `simple-model` ne sera pas reconstruit et le build reprendra là où nous l'avions laissé dans `simple-weather`. Si `simple-weather` est construit avec succès, Maven poursuivra et construira les autres projets.

12.2.2. Spécifier un sous ensemble de projets

Supposons que nous ayons modifié `simple-command` et `simple-webapp` et que nous voulions reconstruire ces deux projets. Nous pouvons utiliser l'argument `--projects` ainsi :

```
$ mvn --projects simple-command,simple-webapp install
```

Seuls ces deux projets seront construits, ce qui nous évitera d'avoir à exécuter Maven séparément dans chaque répertoire.

12.2.3. Construire des sous-ensembles

Supposons que nous soyons des développeurs travaillant sur `simple-command`. Nous ne désirons pas travailler sur `simple-webapp` pour l'instant mais juste avoir une version fonctionnelle de `simple-command`. Nous pouvons utiliser `--also-make` ainsi :

```
$ mvn --projects simple-command --also-make install
```

Lorsque nous utilisons `--also-make`, Maven va examiner la liste de projets (ici, uniquement `simple-command`) et va descendre dans l'arbre des dépendances à la recherche des projets qu'il va devoir construire. Dans ce cas, il va automatiquement construire `simple-model`, `simple-weather` et `simple-persist` mais sans construire `simple-webapp`.

12.2.4. Modifier `simple-weather` et vérifier que nous n'avons rien cassé grâce à `--also-make-dependents`

Supposons que nous ayons modifié `simple-weather`. Nous voulons nous assurer que nous n'avons cassé aucun des projets qui en dépendent. (Dans ce cas-ci, nous voulons nous assurer que nous n'avons pas cassé `simple-command` et `simple-webapp`, mais dans un Reactor plus complexe cela ne serait pas si évident.) Nous voulons aussi éviter d'avoir à reconstruire et tester les projets dont nous savons qu'ils n'ont pas changé. Dans ce cas-ci, nous voulons éviter de construire `simple-persist`. Nous pouvons utiliser `--also-make-dependents` ainsi :

```
$ mvn --projects simple-weather --also-make-dependents install
```

Quand nous utilisons `--also-make-dependents`, Maven va examiner tous les projets de notre reactor pour trouver ceux qui dépendent de `simple-weather` et va automatiquement construire ceux-ci et aucun autre. Dans notre cas, il va automatiquement construire `simple-weather` puis `simple-command` et `simple-webapp`.

12.2.5. Reprise d'un build "make"

Quand nous utilisons `--also-make`, nous exécutons un sous-ensemble de projets, mais cela ne signifie pas que le build ne va pas échouer en plein milieu. Nous pouvons reprendre notre build `--also-make` depuis le projet qui l'a arrêté en utilisant les options `--resume-from` et `--also-make` ainsi :

```
$ mvn --projects simple-command --also-make \  
  --resume-from simple-weather install
```

L'argument `--resume-from` fonctionne aussi avec `--also-make-dependents`.

12.3. Usage du plugin Maven Help

Tout au long de ce livre, nous allons introduire des plugins Maven, parler de fichiers Maven Project Object Model (POM), de fichiers de settings et de profils. Parfois vous aurez besoin d'un outil vous permettant de connaître les modèles utilisés par Maven et les goals disponibles pour un plugin donné. Le plugin Maven Help vous permet de lister tous les profils Maven actifs, d'afficher un POM effectif, les settings effectifs ou les attributs d'un plugin Maven.



Note

Pour une vision plus conceptuelle du POM et des plugins, lisez le Chapitre 3, Mon premier projet avec Maven.

Le plugin Maven Help a quatre goals. Les trois premiers goals — `active-profiles`, `effective-pom` et `effective-settings` — décrivent un projet particulier et doivent être exécutés depuis le répertoire racine d'un projet. Le dernier goal — `describe` — est un peu plus complexe, il affiche des informations sur un plugin ou le goal d'un plugin. Voici des informations générales sur ces quatre goals :

`help:active-profiles`

Liste les profils (du projet, de l'utilisateur, globaux) qui sont actifs pour le build courant.

`help:effective-pom`

Affiche le POM effectif pour le build en cours, en prenant en compte les profils activés.

`help:effective-settings`

Affiche la configuration calculée des settings pour le projet, en prenant en compte la configuration de l'utilisateur et les modifications apportées par les profils actifs.

`help:describe`

Décrit les attributs d'un plugin. Il n'est pas nécessaire de l'exécuter dans le répertoire d'un projet. Vous devez fournir au moins le `groupId` et le `artifactId` du plugin dont vous voulez la description.

12.3.1. Décrire un plugin Maven

Une fois que vous commencez à utiliser Maven, vous passerez le plus clair de votre temps à rechercher des informations sur les plugins Maven. Comment fonctionnent les plugins ? Quels sont les paramètres de configuration ? Quels sont les goals ? Vous utiliserez fréquemment le goal `help:describe` pour obtenir ce type d'information. Le paramètre `plugin` vous permet de spécifier le plugin sur lequel vous voulez des informations, en passant son préfixe (par exemple `maven-help-plugin` ou `help`) ou son nom complet `groupId:artifact[:version]`, la version étant optionnelle. Par exemple, la commande suivante utilise le goal `describe` du plugin Maven Help pour afficher des informations sur le plugin Maven Help.

```
$ mvn help:describe -Dplugin=help
```

```

...
Group Id:  org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version:    2.0.1
Goal Prefix: help
Description:

The Maven Help plugin provides goals aimed at helping to make sense out of
the build environment. It includes the ability to view the effective
POM and settings files, after inheritance and active profiles
have been applied, as well as a describe a particular plugin goal to give
usage information.
...

```

L'exécution du goal `describe` en passant le paramètre `plugin` a affiché les coordonnées Maven du plugin, le préfixe du goal et une description succincte du plugin. Même si ces informations sont utiles, la plupart du temps on voudra en savoir un peu plus. Pour afficher la liste des goals avec leurs paramètres grâce au plugin Help, il faut exécuter le goal `help:describe` avec le paramètre `full` comme ci-dessous :

```

$ mvn help:describe -Dplugin=help -Dfull
...
Group Id:  org.apache.maven.plugins
Artifact Id: maven-help-plugin
Version:    2.0.1
Goal Prefix: help
Description:

The Maven Help plugin provides goals aimed at helping to make sense out of
the build environment. It includes the ability to view the effective
POM and settings files, after inheritance and active profiles
have been applied, as well as a describe a particular plugin goal to
give usage information.

Mojos:
=====
Goal: 'active-profiles'
=====
Description:

Lists the profiles which are currently active for this build.

Implementation: org.apache.maven.plugins.help.ActiveProfilesMojo
Language: java

Parameters:
-----

[0] Name: output
Type: java.io.File
Required: false
Directly editable: true
Description:

```



```
This is an optional parameter for a file destination for the output of
this mojo...the listing of active profiles per project.
```

```
-----
```

```
[1] Name: projects
Type: java.util.List
Required: true
Directly editable: false
Description:
```

```
This is the list of projects currently slated to be built by Maven.
```

```
-----
```

```
This mojo doesn't have any component requirements.
```

```
=====
```

```
... removed the other goals ...
```

Cette option est fort utile pour découvrir un plugin, tous ses goals et leurs paramètres. Mais parfois on veut encore plus d'informations. Pour obtenir les informations sur un unique goal, renseignez le paramètre `mojo` en plus du paramètre `plugin`. La commande suivante liste l'ensemble des informations sur le goal `compile` du plugin `Compiler`.

```
$ mvn help:describe -Dplugin=compiler -Dmojo=compile -Dfull
```


Chapitre 13. Configuration Maven

13.1. Configuration des plugins Maven

Pour personnaliser le comportement d'un plugin Maven, vous devez configurer celui-ci dans le POM du projet. Les sections suivantes présentent les différents moyens à votre disposition pour personnaliser la configuration d'un plugin Maven.

13.1.1. Paramètres du plugin Configuration

Les plugins Maven sont configurés par l'intermédiaire de propriétés définies par goals. Par exemple, si vous jetez un coup d'oeil au goal `compile` du plugin Maven Compiler, vous verrez une liste de paramètres de configuration, comme `source`, `target`, `compilerArgument`, `fork`, `optimize`, ... Maintenant, si vous regardez le goal `testCompile`, vous trouverez une liste différente de paramètres. Pour trouver la liste des paramètres de configuration d'un goal d'un plugin, vous pouvez utiliser le plugin Maven Help. Celui-ci permet d'afficher la description d'un plugin ou d'un goal en particulier.

Pour afficher la description d'un plugin, utilisez le goal `help:describe` de la manière suivante à partir de la ligne de commande :

```
$ mvn help:describe -Dcmd=compiler:compile
[INFO] [help:describe {execution: default-cli}]
[INFO] 'compiler:compile' is a plugin goal (aka mojo).
Mojo: 'compiler:compile'
compiler:compile
  Description: Compiles application sources
  Deprecated. No reason given
```

Pour plus d'informations sur les paramètres de configuration disponibles, utilisez cette même commande en y ajoutant l'argument `-Ddetail` :

```
$ mvn help:describe -Dcmd=compiler:compile -Ddetail
[INFO] [help:describe {execution: default-cli}]
[INFO] 'compiler:compile' is a plugin goal (aka mojo).
Mojo: 'compiler:compile'
compiler:compile
  Description: Compiles application sources
  Deprecated. No reason given
  Implementation: org.apache.maven.plugin.CompilerMojo
  Language: java
  Bound to phase: compile

  Available parameters:

  compilerArgument
    Sets the unformatted argument string to be passed to the compiler if fork
    is set to true.

  This is because the list of valid arguments passed to a Java compiler
```

varies based on the compiler version.
Deprecated. No reason given

compilerArguments

Sets the arguments to be passed to the compiler (prepending a dash) if fork is set to true.

This is because the list of valid arguments passed to a Java compiler varies based on the compiler version.
Deprecated. No reason given

compilerId (Default: javac)

The compiler id of the compiler to use. See this guide for more information.
Deprecated. No reason given

compilerVersion

Version of the compiler to use, ex. '1.3', '1.5', if fork is set to true.
Deprecated. No reason given

debug (Default: true)

Set to true to include debugging information in the compiled class files.
Deprecated. No reason given

encoding

The -encoding argument for the Java compiler.
Deprecated. No reason given

excludes

A list of exclusion filters for the compiler.
Deprecated. No reason given

executable

Sets the executable of the compiler to use when fork is true.
Deprecated. No reason given

failOnError (Default: true)

Indicates whether the build will continue even if there are compilation errors; defaults to true.
Deprecated. No reason given

fork (Default: false)

Allows running the compiler in a separate process. If 'false' it uses the built in compiler, while if 'true' it will use an executable.
Deprecated. No reason given

includes

A list of inclusion filters for the compiler.
Deprecated. No reason given

maxmem

Sets the maximum size, in megabytes, of the memory allocation pool, ex. '128', '128m' if fork is set to true.
Deprecated. No reason given

```

meminitial
  Initial size, in megabytes, of the memory allocation pool, ex. '64',
  '64m' if fork is set to true.
  Deprecated. No reason given

optimize (Default: false)
  Set to true to optimize the compiled code using the compiler's
  optimization methods.
  Deprecated. No reason given

outputFileName
  Sets the name of the output file when compiling a set of sources to a
  single file.
  Deprecated. No reason given

showDeprecation (Default: false)
  Sets whether to show source locations where deprecated APIs are used.
  Deprecated. No reason given

showWarnings (Default: false)
  Set to true to show compilation warnings.
  Deprecated. No reason given

source
  The -source argument for the Java compiler.
  Deprecated. No reason given

staleMillis (Default: 0)
  Sets the granularity in milliseconds of the last modification date for
  testing whether a source needs recompilation.
  Deprecated. No reason given

target
  The -target argument for the Java compiler.
  Deprecated. No reason given

verbose (Default: false)
  Set to true to show messages about what the compiler is doing.
  Deprecated. No reason given

```

Si vous désirez récupérer la liste des goals d'un plugin, vous pouvez exécuter le goal `help:describe` et lui passer un paramètre. Ce paramètre accepte soit un préfixe de plugin soit un `groupId` et un `artifactId` comme le montrent les exemples suivants :

```

$ mvn help:describe -Dplugin=compiler
[INFO] [help:describe {execution: default-cli}]
[INFO] org.apache.maven.plugins:maven-compiler-plugin:2.0.2

Name: Maven Compiler Plugin
Description: Maven Plugins
Group Id: org.apache.maven.plugins
Artifact Id: maven-compiler-plugin
Version: 2.0.2
Goal Prefix: compiler

```

This plugin has 2 goals:

```
compiler:compile
  Description: Compiles application sources
  Deprecated. No reason given

compiler:testCompile
  Description: Compiles application test sources
  Deprecated. No reason given
```

Vous pouvez utiliser le `groupId` et l'`artifactId` du plugin pour obtenir la même liste de goals.

```
$ mvn help:describe -Dplugin=org.apache.maven.plugins:maven-compiler-plugin
```

En passant l'argument `-Ddetail` au goal `help:describe`, vous demandez à Maven d'afficher tous les goals d'un plugin avec tous leurs paramètres.

13.1.2. Ajouter des dépendances à un plugin

Si vous désirez configurer un plugin pour qu'il utilise des versions spécifiques de ses dépendances, vous pouvez utiliser la balise `dependencies`. Quand le plugin s'exécute, il se lance avec un classpath contenant ses dépendances. L'Exemple 13.1, « Ajout d'une dépendance à un plugin » est un exemple de configuration de plugin qui surcharge les versions de ses dépendances et en ajoute une nouvelle pour faciliter l'exécution du goal.

Exemple 13.1. Ajout d'une dépendance à un plugin

```
<plugin>
  <groupId>com.agilejava.docbkx</groupId>
  <artifactId>docbkx-maven-plugin</artifactId>
  <version>2.0.9</version>
  <dependencies>
    <dependency>
      <groupId>docbook</groupId>
      <artifactId>docbook-xml</artifactId>
      <version>4.5</version>
    </dependency>
    <dependency>
      <groupId>org.apache.fop</groupId>
      <artifactId>fop-pdf-images</artifactId>
      <version>1.3</version>
    </dependency>
    <dependency>
      <groupId>org.apache.fop</groupId>
      <artifactId>fop-pdf-images-res</artifactId>
      <version>1.3</version>
      <classifier>res</classifier>
    </dependency>
    <dependency>
      <groupId>pdfbox</groupId>
      <artifactId>pdfbox</artifactId>
```

```
<version>0.7.4-dev</version>
  <classifier>dev</classifier>
</dependency>
</dependencies>
</plugin>
```

13.1.3. Configurer les paramètres globaux d'un plugin

Pour affecter une valeur à un paramètre de configuration sur un projet, utilisez le fichier XML de l'Exemple 13.2, « Configurer un plugin Maven ». À moins que cette configuration ne soit surchargée par une configuration de paramètre de plugin plus spécifique, Maven utilisera les valeurs définies dans la balise `plugin` pour tous les goals exécutés par ce plugin.

Exemple 13.2. Configurer un plugin Maven

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
</plugin>
```

13.1.4. Modifier les paramètres spécifiques à une exécution

Vous pouvez configurer les paramètres d'un plugin spécifiquement pour l'exécution d'un goal Maven. L'Exemple 13.3, « Surcharge des paramètres de configuration d'une exécution » montre comment passer un paramètre de configuration au goal exécuté par le plugin `AntRun` durant la phase de validation. Cette exécution héritera ainsi des paramètres de la balise `configuration` du plugin dont les valeurs seront fusionnées avec celles définies dans l'exécution personnalisée.

Exemple 13.3. Surcharge des paramètres de configuration d'une exécution

```
<plugin>
  <artifactId>maven-antrun-plugin</artifactId>
  <executions>
    <execution>
      <phase>validate</phase>
      <goals>
        <goal>run</goal>
      </goals>
      <configuration>
        <tasks>
          <echo>${PATH}=${env.PATH}</echo>
          <echo>User's Home Directory: ${user.home}</echo>
          <echo>Project's Base Director: ${basedir}</echo>
        </tasks>
      </configuration>
    </execution>
  </executions>
```

```
</plugin>
```

13.1.5. Configuration des paramètres par défaut pour une exécution en ligne de commande

Depuis Maven 2.2.0, vous pouvez fournir des paramètres de configuration pour les goals qui sont exécutés à partir de la ligne de commande. Pour cela, utilisez un id spécial pour l'exécution, `default-cli`. L'Exemple 13.4, « Configuration d'un plugin pour une exécution en ligne de commande » montre comment rattacher un goal à la phase `package` du cycle de vie qui produit le binaire. Cet exemple configure également l'exécution `default-cli` du plugin Assembly pour qu'il utilise le descripteur `jar-with-dependencies`. Le descripteur `bin.xml` sera utilisé durant le cycle de vie, et `jar-with-dependencies` sera utilisé lors de l'exécution de la commande `mvn assembly:assembly` à partir de la console.

Exemple 13.4. Configuration d'un plugin pour une exécution en ligne de commande

```
<plugin>
  <artifactId>maven-assembly-plugin</artifactId>
  <configuration>
    <appendAssemblyId>>false</appendAssemblyId>
  </configuration>
  <executions>
    <execution>
      <id>assemble-binary</id>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <descriptors>
          <descriptor>src/main/assembly/bin.xml</descriptor>
        </descriptors>
      </configuration>
    </execution>
    <execution>
      <id>default-cli</id>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </execution>
  </executions>
</plugin>
```

13.1.6. Configuration des paramètres pour les goals rattachés au cycle de vie par défaut

Depuis Maven 2.2.0, si vous désirez personnaliser le comportement d'un goal qui est déjà rattaché au cycle de vie par défaut, vous pouvez utiliser un id spécial pour l'exécution, "default-<goal>". Ainsi,

vous pouvez par exemple personnaliser le comportement du goal `jar` du plugin `Jar` qui est rattaché à la phase `package` du cycle de vie par défaut. Vous pouvez personnaliser les paramètres de configuration d'une exécution comme le présente l'Exemple 13.5, « Configuration d'un paramètre pour l'exécution d'un goal par défaut ».

Exemple 13.5. Configuration d'un paramètre pour l'exécution d'un goal par défaut

```
<plugin>
  <artifactId>maven-jar-plugin</artifactId>
  <executions>
    <execution>
      <id>default-jar</id>
      <configuration>
        <excludes>
          <exclude>**/somepackage/*</exclude>
        </excludes>
      </configuration>
    </execution>
    <execution>
      <id>special-jar</id>
      <phase>package</phase>
      <goals>
        <goal>jar</goal>
      </goals>
      <configuration>
        <includes>
          <include>**/somepackage/*</include>
        </includes>
        <classifier>somepackage</classifier>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Dans cet exemple, le goal par défaut `jar` est configuré pour exclure le contenu d'un certain package. Un autre goal, `jar`, est rattaché à la phase `package` pour créer un fichier JAR qui contient seulement le contenu d'un package spécifique.

Configurer les paramètres des goals par défaut peut aussi être intéressant si vous avez besoin de configurer deux goals avec des valeurs de configuration différentes pour certains paramètres. L'Exemple 13.6, « Configurer deux paramètres d'un goal d'un plugin » montre comment configurer le goal `resources:resources` pour qu'il ne tienne pas compte des répertoires vides tout en configurant le goal `resources:testResources` pour qu'il les prenne en compte.

Exemple 13.6. Configurer deux paramètres d'un goal d'un plugin

```
<plugin>
  <artifactId>maven-resources-plugin</artifactId>
  <executions>
    <execution>
      <id>default-resources</id>
```

```
<configuration>
  <includeEmptyDirs>false</includeEmptyDirs>
</configuration>
</execution>
<execution>
  <id>default-testResources</id>
  <configuration>
    <includeEmptyDirs>true</includeEmptyDirs>
  </configuration>
</execution>
</executions>
</plugin>
```

Chapitre 14. Maven Assemblies

14.1. Introduction

Maven fournit des plugins qui sont utilisés pour créer des archives dans les formats les plus courants, et qui seront, pour la plupart, utilisées comme dépendances d'autres projets. Par exemple, nous avons les plugins JAR, WAR, EJB et EAR. Comme nous l'avons déjà vu dans le Chapitre 10, Cycle de vie du build ces plugins correspondent à différents formats de packaging de projet, chacun avec son processus de build légèrement différent. Même si Maven supporte les différents formats standards de packaging grâce à ses plugins et des cycles de vie personnalisés, il va arriver un moment où vous aurez besoin de créer une archive ou un répertoire avec une structure qui lui est propre. Ces archives personnalisées sont appelées Assemblies Maven.

Il existe de nombreuses raisons pour lesquelles vous voudrez construire ces archives personnalisées pour votre projet. La plus courante est peut-être la distribution du projet. Ce mot 'distribution' peut signifier plusieurs choses selon la personne qui l'emploie (ou selon le projet), tout cela dépendant de la manière dont le projet doit être utilisé. Essentiellement, il s'agit d'archives qui fournissent un moyen simple pour les utilisateurs d'installer ou d'utiliser le produit du projet. Dans certains cas, cela peut signifier fournir un serveur d'applications comme Jetty avec l'application Web. Dans d'autres, il s'agit de fournir un paquet contenant le code source et la documentation de l'API avec le binaire compilé comme, par exemple, un fichier jar. C'est souvent lorsque vous êtes en train de construire la version distribuable d'un produit que les assemblies vont pouvoir vous aider. Par exemple, les produits comme Nexus, dont on parle plus en détail dans *Repository Management with Nexus*¹, sont le résultat de plusieurs énormes projets Maven multimodules, et c'est un assembly Maven qui a construit l'archive finale que vous téléchargez depuis Sonatype.

Dans la plupart des cas, le plugin Assembly est idéalement taillé pour construire des packages distribuables de projets. Cependant, les assemblies ne sont pas forcément des archives distribuables ; les assemblies doivent apporter aux utilisateurs de Maven la flexibilité dont ils ont besoin pour produire des archives personnalisées de tout type. En bref, les assemblies doivent combler les trous laissés par les formats standards d'archives fournis par les types de packaging des projets. Bien sûr, vous pourriez écrire un plugin Maven complet pour produire votre propre format d'archive, avec une nouvelle association au cycle de vie et la configuration de gestion d'artefact pour indiquer à Maven comment le déployer. Mais le plugin Assembly rend tout cela superflu dans la plupart des cas en vous donnant les moyens de construire votre archive selon votre propre recette sans avoir à écrire une ligne de code Maven.

14.2. Les bases du plugin Assembly

Avant d'aller plus loin, prenons une minute pour parler des deux principaux goals du plugin Assembly : `assembly:assembly` et le `mojo single`. J'ai cité ces deux goals de manière différente pour indiquer

¹ <http://www.sonatype.com/books/nexus-book/reference/>

qu'on ne les utilise pas de la même manière. Le goal `assembly:assembly` est conçu pour être invoqué directement depuis la ligne de commande et il ne doit jamais être lié à une phase du cycle de vie. Au contraire, le mojo `single` est lui conçu pour faire partie de votre build de tous les jours et doit être rattaché à une phase du cycle de vie du build de votre projet.

La raison de cette différence est que le goal `assembly:assembly` est ce que Maven appelle un mojo agrégateur ; c'est à dire un mojo qui a été conçu pour être exécuté au plus une fois dans un build, quelque soit le nombre de projets qui sont construits. Il tire sa configuration du projet racine - habituellement le POM de plus haut niveau ou la ligne de commande. Quand il est rattaché à un cycle de vie, un mojo agrégateur peut provoquer de désagréables effets secondaires. Il peut forcer l'exécution de la phase `package` du cycle de vie en avance de phase, ce qui fait que le build exécute cette phase `package` deux fois.

Comme le goal `assembly:assembly` est un mojo agrégateur, cela peut poser des problèmes avec les builds Maven multimodules et il doit donc être appelé seul en ligne de commande. Ne rattachez jamais l'exécution de `assembly:assembly` à une phase du cycle de vie. `assembly:assembly` était le goal originel du plugin Assembly et il n'a jamais été conçu pour faire partie du processus de build standard d'un projet. Quand il est devenu évident que les archives produites par `assembly` étaient une exigence légitime des projets, le mojo `single` a été développé. Ce mojo suppose qu'il a été rattaché à la bonne partie du processus de build et que, donc, il aura accès aux fichiers et artefacts du projet dont il a besoin pour s'exécuter au sein d'un grand projet Maven multimodule. Dans un environnement multimodule, il s'exécutera autant de fois qu'il est lié aux POMs des différents modules. Contrairement à `assembly:assembly`, `single` ne forcera jamais l'exécution d'une étape du cycle de vie en avance de phase.

Le plugin Assembly propose plusieurs autres goals en plus de ces deux là. Cependant, le détail de ces autres mojos dépasse le cadre de ce chapitre, car ils servent pour des cas d'utilisation obsolètes ou exotiques : on a très rarement besoin d'eux. Autant que possible, pour produire vos packages utilisez `assembly:assembly` depuis la ligne de commande et `single` pour rattacher cette opération aux phases du cycle de vie.

14.2.1. Les descripteurs Assembly prédéfinis

Nombreux sont ceux qui choisissent de créer leurs propres recettes - appelées descripteurs d'assembly - cependant cela n'est pas forcément nécessaire. Le plugin Assembly fournit des descripteurs prêts à l'emploi pour plusieurs types d'archives communs. Vous pouvez donc les utiliser immédiatement sans écrire une ligne de configuration. Voici la liste des descripteurs d'assembly prédéfinis dans le plugin Maven Assembly :

`bin`

Le descripteur `bin` est utilisé pour packager les fichiers `LICENSE`, `README` et `NOTICE` du projet avec son artefact principal, pour peu que ce dernier soit un jar. Vous pouvez voir cela comme la plus petite distribution binaire pour un projet autosuffisant.

`jar-with-dependencies`

Le descripteur `jar-with-dependencies` construit une archive JAR avec le contenu du jar du projet principal et les contenus des dépendances d'exécution de ce projet. Associé avec la bonne définition de la `Main-Class` dans le fichier Manifest (dont on parle dans “Configuration du Plugin” ci-dessous), ce descripteur permet de produire un jar exécutable autosuffisant votre projet, même si ce dernier possède des dépendances.

`project`

Le descripteur `project` construit une archive à partir de la structure de répertoire du projet telle qu'elle existe sur votre système de fichier, et probablement dans votre outil de gestion de configuration. Bien sûr, le répertoire `target` n'est pas pris en compte ainsi que les fichiers de métadonnées comme les répertoires `CVS` et `.svn` que nous avons l'habitude de voir. En bref, le but de ce descripteur est de créer une archive du projet qui, une fois décompressée, permet de construire le projet avec Maven.

`src`

Le descripteur `src` produit une archive du code source de votre projet avec les fichiers `pom.xml`, ainsi que les éventuels fichiers `LICENSE`, `README` et `NOTICE` qui se trouvent dans le répertoire racine du projet. Ce descripteur produit une archive qui peut être construite par Maven dans la plupart des cas. Cependant, il suppose que tout le code source et les ressources soient dans le répertoire standard `src` et donc, qu'il peut oublier les fichiers et les répertoires non-standards même s'ils sont critiques pour le build.

14.2.2. Construire un Assembly

Le plugin Assembly peut être exécuté de deux manières : vous pouvez l'invoquer directement depuis la ligne de commande ou le configurer comme un élément standard de votre processus de build en le rattachant à une phase du cycle de vie du build de votre projet. L'invocation directe a son utilité, particulièrement pour les assemblies qui ne font pas partie des principaux livrables de votre projet. Dans la plupart des cas, vous voudrez produire les assemblies de votre projet au cours de son processus de build standard. Ainsi, vos packagings personnalisés sont inclus lorsque votre projet est installé ou déployé dans les dépôts Maven et ils sont donc toujours disponibles pour vos utilisateurs.

Comme exemple d'invocation directe du plugin Assembly, imaginez que vous voulez livrer une copie de votre projet que l'on puisse construire à partir des sources. Au lieu de déployer uniquement le produit final du build, vous voulez aussi inclure le code source. Ce n'est pas une opération que vous avez besoin de répéter souvent, donc ça n'a pas de sens que d'ajouter cette configuration à votre POM. Au lieu de cela, vous pouvez utiliser la commande suivante :

```
$ mvn -DdescriptorId=project assembly:single
...
[INFO] [assembly:single]
[INFO] Building tar : /Users/~/.mvn-examples-1.0/assemblies/direct-invocation/\
target/direct-invocation-1.0-SNAPSHOT-project.tar.gz
[INFO] Building tar : /Users/~/.mvn-examples-1.0/assemblies/direct-invocation/\
target/direct-invocation-1.0-SNAPSHOT-project.tar.bz2
```

```
[INFO] Building zip: /Users/~/.mvn-examples-1.0/assemblies/direct-invocation/\
target/direct-invocation-1.0-SNAPSHOT-project.zip
...
```

Imaginez que vous voulez produire un JAR exécutable à partir de votre projet. Si votre projet est autosuffisant et sans dépendance, vous pouvez obtenir ce résultat à partir de votre artefact avec un peu de configuration du plugin JAR. Cependant, la plupart des projets ont des dépendances et celles-ci doivent être incorporées pour obtenir un JAR exécutable. Dans ce cas, vous voulez vous assurer qu'à chaque fois que vous installez ou déployez l'artefact JAR de votre projet, le JAR exécutable le soit aussi.

Supposons que la classe principale de votre projet est `org.sonatype.mavenbook.App`, la configuration de POM suivante permet de créer un JAR exécutable :

Exemple 14.1. Descripteur assembly pour un JAR exécutable

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.assemblies</groupId>
  <artifactId>executable-jar</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
  <name>Assemblies Executable Jar Example</name>
  <url>http://sonatype.com/book</url>
  <dependencies>
    <dependency>
      <groupId>commons-lang</groupId>
      <artifactId>commons-lang</artifactId>
      <version>2.4</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.2-beta-2</version>
        <executions>
          <execution>
            <id>create-executable-jar</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
            <configuration>
              <descriptorRefs>
                <descriptorRef>
                  jar-with-dependencies
                </descriptorRef>
              </descriptorRefs>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </build>
</project>
```

```

        </descriptorRefs>
        <archive>
            <manifest>
                <mainClass>org.sonatype.mavenbook.App</mainClass>
            </manifest>
        </archive>
    </configuration>
</execution>
</executions>
</plugin>
</plugins>
</build>
</project>

```

Il y a deux points auxquels il faut prêter attention dans la configuration ci-dessus. Premièrement, nous utilisons l'élément `descriptorRefs` dans configuration plutôt que le paramètre `descriptorId` que nous avons utilisé précédemment. Cela nous permet de construire différents packages durant la même exécution du plugin Assembly tout en supportant notre cas d'utilisation avec très peu de configuration supplémentaire. Deuxièmement, la balise `archive` sous `configuration` spécifie l'attribut `Main-Class` du fichier manifest dans le JAR produit. Cette section est généralement disponible dans les plugins qui créent des fichiers JAR comme le plugin JAR utilisé pour le packaging par défaut des projets.

Maintenant, vous pouvez produire un JAR exécutable simplement en exécutant la commande `mvn package`. Après, nous listerons le contenu du répertoire `target` pour vérifier que le JAR exécutable a bien été généré. Enfin, pour prouver qu'il s'agit bien d'un JAR exécutable, nous essayerons de l'exécuter :

```

$ mvn package
... (output omitted) ...
[INFO] [jar:jar]
[INFO] Building jar: ~/mvn-examples-1.0/assemblies/executable-jar/target/\
    executable-jar-1.0-SNAPSHOT.jar
[INFO] [assembly:single {execution: create-executable-jar}]
[INFO] Processing DependencySet (output=)
[INFO] Building jar: ~/mvn-examples-1.0/assemblies/executable-jar/target/\
    executable-jar-1.0-SNAPSHOT-jar-with-dependencies.jar
... (output omitted) ...
$ ls -l target
... (output omitted) ...
executable-jar-1.0-SNAPSHOT-jar-with-dependencies.jar
executable-jar-1.0-SNAPSHOT.jar
... (output omitted) ...
$ java -jar \
    target/executable-jar-1.0-SNAPSHOT-jar-with-dependencies.jar
Hello, World!

```

Des traces présentées ci-dessus vous pouvez voir que maintenant le build normal du projet produit un nouvel artefact en plus du principal fichier JAR. Ce nouvel artefact a le classifieur `jar-with-dependencies`. Enfin, nous avons vérifié que ce nouveau JAR est réellement exécutable et que son exécution produit le résultat attendu : l'affichage de "Hello, World!".

14.2.3. Utilisation des assemblies comme dépendances

Lorsque des assemblies sont produits durant le processus de build normal du projet, les archives résultantes sont jointes à l'artefact principal du projet. Cela signifie qu'elles seront installées et déployées au côté de l'artefact et qu'elles seront donc accessibles comme ce dernier. Chaque artefact assembly aura les mêmes coordonnées de base que le projet principal (à savoir le `groupId`, l'`artifactId` et la `version`). Cependant ces artefacts sont des pièces rapportées ce qui signifie pour Maven qu'il s'agit de produits secondaires au build du projet principal. Par exemple, les assemblies `source` contiennent les données d'entrée brutes du build du projet et les assemblies `jar-with-dependencies` contiennent l'ensemble des classes du projet et de ses dépendances. Les artefacts ainsi rattachés peuvent ne pas respecter la règle Maven un projet un artefact de par leur nature de produits secondaires.

Comme les assemblies sont (normalement) des artefacts rattachés, chacun doit avoir son classifieur en plus des coordonnées de l'artefact principal pour le distinguer de ce dernier. Par défaut ce classifieur est l'identifiant du descripteur de l'assembly. Quand on utilise les descripteurs pré-définis l'identifiant du descripteur d'assembly est le même que l'identifiant utilisé dans la balise `descriptorRef` pour ce type d'assembly.

Maintenant que vous avez déployé l'assembly au côté de votre artefact principal, comment pouvez-vous l'utiliser comme dépendance dans un autre projet ? La réponse est simple. Rappelez-vous la discussion à propos des dépendances entre projets avec Maven dans la Section 3.5.3, « Les coordonnées Maven » et dans la Section 9.5.1, « Au sujet des coordonnées », les projets dépendent les uns des autres grâce à quatre éléments de base que l'on appelle coordonnées d'un projet : `groupId`, `artifactId`, `version` et `packaging`. Dans la Section 11.5.3, « Classifieurs de plateforme », il existe de nombreuses variantes de l'artefact du projet selon la plate-forme cible et le projet spécifie un élément `classifieur` qui prend la valeur `win` ou `linux` de manière à pouvoir choisir le bon artefact selon la plate-forme cible. Les artefacts assembly peuvent être utilisés comme dépendance grâce aux coordonnées de base du projet combinées au classifieur avec lequel l'assembly a été installé ou déployé. Si l'assembly n'est pas une archive JAR il faudra aussi déclarer son type.

14.2.4. Construction d'assemblies à partir d'assemblies dépendances

Assez perturbant ce titre n'est ce pas ? Essayons de mettre en place un scénario pour expliquer cette notion d'assemblage d'assembly. Imaginez que vous voulez construire une archive qui elle-même contient des assemblies d'un projet. Supposons que vous ayez un build multimodule et que vous vouliez déployer un assembly qui contient un ensemble d'assemblies de ce projet liés entre eux. Dans l'exemple de cette section, nous créons un package qui rassemble des répertoires "constructibles" des sous-modules d'un projet qui sont utilisés ensemble. Pour essayer de rester simple nous utiliserons les descripteurs d'assembly dont nous venons de parler - `project` et `jar-with-dependencies`. Dans cet exemple particulier, nous supposerons que chaque projet construit son assembly en plus de son artefact JAR principal. Nous supposerons aussi que chaque projet de ce build multimodule rattache le `goal single` à la phase `package` et qu'il utilise la balise `descriptorRef`. Tous les projets de ce build multimodule héritent de la configuration du fichier `pom.xml` de plus haut niveau et notamment de sa

balise `pluginManagement` qui est décrite dans l'Exemple 14.2, « Configuration de l'assembly du projet dans le POM de plus haut niveau ».

Exemple 14.2. Configuration de l'assembly du projet dans le POM de plus haut niveau

```
<project>
...
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.2-beta-2</version>
        <executions>
          <execution>
            <id>create-project-bundle</id>
            <phase>package</phase>
            <goals>
              <goal>single</goal>
            </goals>
            <configuration>
              <descriptorRefs>
                <descriptorRef>project</descriptorRef>
              </descriptorRefs>
            </configuration>
          </execution>
        </executions>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
...
</project>
```

Le POM de chaque projet référence le plugin dont la configuration est décrite dans l'Exemple 14.2, « Configuration de l'assembly du projet dans le POM de plus haut niveau » par une déclaration minimale dans son build comme le montre l'Exemple 14.3, « Activation de la configuration du plugin Assembly dans les projets fils ».

Exemple 14.3. Activation de la configuration du plugin Assembly dans les projets fils

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
    </plugin>
  </plugins>
</build>
```

Pour produire l'ensemble des assemblies du projet, exécutez la commande `mvn install` depuis le répertoire racine. Vous devriez voir Maven installer les artefacts avec des classifieurs dans votre dépôt.

```

$ mvn install
...
Installing ~/mvn-examples-1.0/assemblies/as-dependencies/project-parent/\
second-project/target/second-project-1.0-SNAPSHOT-project.tar.gz to
~/ .m2/repository/org/sonatype/mavenbook/assemblies/second-project/1.0-SNAPSHOT/\
second-project-1.0-SNAPSHOT-project.tar.gz
...
Installing ~/mvn-examples-1.0/assemblies/as-dependencies/project-parent/\
second-project/target/second-project-1.0-SNAPSHOT-project.tar.bz2 to
~/ .m2/repository/org/sonatype/mavenbook/assemblies/second-project/1.0-SNAPSHOT/\
second-project-1.0-SNAPSHOT-project.tar.bz2
...
Installing ~/mvn-examples-1.0/assemblies/as-dependencies/project-parent/\
second-project/target/second-project-1.0-SNAPSHOT-project.zip to
~/ .m2/repository/org/sonatype/mavenbook/assemblies/second-project/1.0-SNAPSHOT/\
second-project-1.0-SNAPSHOT-project.zip
...

```

Lorsque vous exécutez `install`, Maven va copier l'artefact principal de chaque projet ainsi que chaque assembly dans votre dépôt Maven local. Tous ces artefacts sont maintenant disponibles comme dépendance pour d'autres projets localement. Si votre but final est de créer un paquet qui inclut les assemblies de différents projets vous pouvez créer un autre projet qui référence comme dépendances les assemblies de ces projets. Ce projet chapeau prend la responsabilité de construire l'assembly englobant tous les autres. Le POM de ce projet assembly chapeau ressemblerait au document XML décrit dans l'Exemple 14.4, « POM du projet assembly chapeau ».

Exemple 14.4. POM du projet assembly chapeau

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.assemblies</groupId>
  <artifactId>project-bundle</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>Assemblies-as-Dependencies Example Project Bundle</name>
  <url>http://sonatype.com/book</url>
  <dependencies>
    <dependency>
      <groupId>org.sonatype.mavenbook.assemblies</groupId>
      <artifactId>first-project</artifactId>
      <version>1.0-SNAPSHOT</version>
      <classifier>project</classifier>
      <type>zip</type>
    </dependency>
    <dependency>
      <groupId>org.sonatype.mavenbook.assemblies</groupId>
      <artifactId>second-project</artifactId>
      <version>1.0-SNAPSHOT</version>
      <classifier>project</classifier>
    </dependency>
  </dependencies>
</project>

```

```

    <type>zip</type>
  </dependency>
</dependencies>
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.2-beta-2</version>
      <executions>
        <execution>
          <id>bundle-project-sources</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
          <configuration>
            <descriptorRefs>
              <descriptorRef>
                jar-with-dependencies
              </descriptorRef>
            </descriptorRefs>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
</project>

```

Le POM de ce projet assembly chapeau fait référence aux assemblées des projets `first-project` et `second-project`. Au lieu de référencer l'artefact principal de chaque projet, le POM de ce projet assembly spécifie un classifieur `project` et un type `zip`. Cela indique à Maven qu'il va devoir résoudre une archive de type ZIP qui a été créée par l'assembly `project`. Remarquez que le projet assembly produit lui-même un assembly `jar-with-dependencies`. L'assembly produit par `jar-with-dependencies` n'est pas une archive particulièrement élégante puisqu'il construit un fichier JAR contenant toutes les dépendances décompressées. `jar-with-dependencies` demande à Maven de récupérer toutes les dépendances, de les décompresser et ensuite de créer une archive unique en y incluant le résultat du projet courant. Dans ce projet, cela produit un unique fichier JAR qui agrège les contenus des assemblées de `first-project` et de `second-project`.

Cet exemple illustre comment les capacités de base du plugin Maven Assembly peuvent être combinées sans nécessiter un descripteur d'assembly. Il crée une unique archive qui contient les répertoires de plusieurs projets les uns à côté des autres. Cette fois, le `jar-with-dependencies` est juste un format de stockage et nous n'avons pas besoin de spécifier l'attribut de manifest `Main-Class`. Pour construire ce paquet, nous exécutons le projet `project-bundle` comme d'habitude :

```

$ mvn package
...
[INFO] [assembly:single {execution: bundle-project-sources}]
[INFO] Processing DependencySet (output=)

```

```
[INFO] Building jar: ~/downloads/mvn-examples-1.0/assemblies/as-dependencies/\
project-bundle/target/project-bundle-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Pour vérifier que l'assembly `project-bundle` contient bien les éléments décompressés et combinés des assemblées dépendances, exécutez la commande `jar tf` :

```
$ jar tf \
  target/project-bundle-1.0-SNAPSHOT-jar-with-dependencies.jar
...
first-project-1.0-SNAPSHOT/pom.xml
first-project-1.0-SNAPSHOT/src/main/java/org/sonatype/mavenbook/App.java
first-project-1.0-SNAPSHOT/src/test/java/org/sonatype/mavenbook/AppTest.java
...
second-project-1.0-SNAPSHOT/pom.xml
second-project-1.0-SNAPSHOT/src/main/java/org/sonatype/mavenbook/App.java
second-project-1.0-SNAPSHOT/src/test/java/org/sonatype/mavenbook/AppTest.java
```

Maintenant que vous avez lu cette section, son titre devrait avoir un peu plus de sens. Vous avez combiné les assemblées de deux projets dans un assembly grâce à un projet assembly dépendant de chacun de ces assemblées.

14.3. Vue d'ensemble du descripteur d'assembly

Quand les descripteurs d'assembly standards dont nous avons parlé dans la Section 14.2, « Les bases du plugin Assembly » ne sont pas pertinents, il va vous falloir définir votre propre descripteur. Un descripteur d'assembly est un document XML qui définit la structure et les contenus d'un assembly. Ce descripteur se compose de cinq sections principales de configuration et de deux sections additionnelles : une pour spécifier des fragments de descripteur d'assembly standard, que l'on appelle descripteurs de composant, et une autre pour définir des classes de traitement de fichiers personnalisés pour aider à gérer la production de l'assembly.

Configuration de base

Cette section contient les informations nécessaires à tous les assemblées ainsi que quelques options de configuration pour l'archive finale, comme le chemin de base à utiliser pour toutes les entrées d'archive. Pour que le descripteur soit valide vous devez spécifier l'identifiant de l'assembly, au moins un format et au moins une des sections présentées ci-dessus.

Informations concernant les fichiers

La configuration de ce segment du descripteur d'assembly concerne les fichiers du système de fichier contenus dans les répertoires du projet. Ce segment se compose de deux sections principales : `files` et `fileSets`. Vous pouvez utiliser les balises `files` et `fileSets` pour contrôler quels fichiers seront inclus ou exclus de l'assembly et avec quelles permissions.

Informations concernant les dépendances

Presque tous les projets, quelque soit leur taille, dépendent d'autres projets. Durant la création d'archives de distribution, les dépendances d'un projet sont souvent ajoutées dans l'assembly du produit final. Cette section gère la manière dont les dépendances sont ajoutées dans

l'archive finale. C'est dans cette section que vous configurez si les dépendances doivent être décompressées, ajoutées directement au répertoire `lib/` ou renommées. Cette section vous permet aussi de contrôler quelles dépendances doivent être ajoutées à l'assembly et avec quelles permissions.

Informations concernant les dépôts

Parfois il est utile d'isoler l'ensemble des artefacts nécessaires à la construction d'un projet, qu'il s'agisse d'artefacts de dépendance, des POMs des artefacts de dépendance ou même d'un POM ancêtre du projet (le POM parent, son parent, etc.). Cette section vous permet d'inclure une ou plusieurs structures de répertoires de dépôt d'artefact dans votre assembly avec différentes options de configuration. Le plugin Assembly n'est pas capable d'inclure les artefacts de plugin dans ces dépôts pour l'instant.

Informations concernant les modules

Cette section du descripteur d'assembly vous permet de profiter des relations parent-enfant lors de la construction de votre archive personnalisée pour inclure des fichiers de code source, des artefacts et des dépendances des sous-modules de votre projet. C'est la section la plus complexe du descripteur d'assembly car elle vous permet de travailler avec des modules et des sous-modules de deux façons : comme un ensemble de balises `fileSets` (via la section `sources`) ou comme un ensemble de `dependencySets` (via la section `binaries`).

14.4. Le descripteur d'assembly

Cette section présente une vision générale de ce qu'est un descripteur d'assembly et donne les grandes lignes à suivre lorsque l'on écrit son propre descripteur. Le plugin Assembly est l'un des plus gros plugins de l'écosystème Maven, mais aussi l'un des plus souples.

14.4.1. Référence de propriété dans un descripteur d'assembly

Toutes les propriétés dont nous avons parlé dans la Section 15.2, « Propriétés Maven » peuvent être référencées dans un descripteur d'assembly. Avant qu'un descripteur d'assembly soit utilisé par Maven, il doit être interpolé en utilisant les données du POM et de l'environnement de build. Toutes les propriétés du POM supportées pour l'interpolation sont utilisables dans les descripteurs d'assembly, qu'il s'agisse de propriétés du POM, de valeurs d'éléments du POM, de propriétés système, de propriétés définies par l'utilisateur ou de variables d'environnement du système d'exploitation.

Les seules exceptions à cette étape d'interpolation sont les balises `outputDirectory`, `outputDirectoryMapping` ou `outputFileNameMapping` des différentes sections du descripteur. Ces éléments sont conservés sous leur forme primitive pour permettre d'appliquer les données spécifiques à chaque artefact ou module pour la résolution des expressions.

14.4.2. Informations obligatoires pour un assembly

Tout assembly a besoin obligatoirement de deux données : la balise `id` et la liste des formats d'archive à produire. En pratique il faut y ajouter au moins une autre section du descripteur — car la plupart des

archiveurs vont se bloquer s'il y a aucun fichier à incorporer — mais sans une balise `format` et une balise `id` il n'y a aucune archive à réaliser. La balise `id` est utilisée à la fois pour le nom et le classifieur dans le dépôt Maven de l'artefact de l'archive produite. Le format détermine quel composant archiveur sera utilisé pour produire l'archive assembly finale. Tous les descripteurs d'assembly doivent contenir une balise `id` et au moins une balise `format` :

Exemple 14.5. Balises obligatoires d'un descripteur d'assembly

```
<assembly>
  <id>bundle</id>
  <formats>
    <format>zip</format>
  </formats>
  ...
</assembly>
```

L'`id` de l'assembly est une chaîne de caractères sans espace. La pratique standard est d'utiliser des tirets entre les mots de l'`id` de l'assembly. Si vous produisez un assembly qui construit une structure unique d'un package intéressant, vous lui donnerez une `id` comme `unique-package-intéressant`. Il est possible de déclarer plusieurs formats différents dans un même descripteur d'assembly, ce qui vous permet de créer les archives de distribution habituelles `.zip`, `.tar.gz` et `.tar.bz2` facilement. Si vous ne trouvez pas le format d'archive qu'il vous faut, vous pouvez créer un format personnel. Les formats personnels sont décrits dans la Section 14.5.8, « `componentDescriptors` et `containerDescriptorHandlers` ». Le plugin Assembly supporte nativement plusieurs formats d'archive, ce qui inclut :

- `jar`
- `zip`
- `tar`
- `bzip2`
- `gzip`
- `tar.gz`
- `tar.bz2`
- `rar`
- `war`
- `ear`

- `src`
- `dir`

L'`id` et le `format` sont indispensables car ils vont faire partie des coordonnées de l'archive assemblée. L'exemple de l'Exemple 14.5, « Balises obligatoires d'un descripteur d'assembly » va créer un artefact d'assembly de type `zip` avec comme classifieur `bundle`.

14.5. Choisir les contenus d'un assembly

En théorie, l'`id` et le `format` sont les seuls éléments obligatoires pour un descripteur d'assembly valide ; cependant la plupart des composants archiveurs d'assembly échoueront s'ils n'ont pas au moins un fichier à inclure dans l'archive finale. La définition des fichiers à inclure dans l'assembly se fait dans les cinq sections principales du descripteur d'assembly : `files`, `fileSets`, `dependencySets`, `repositories` et `moduleSets`. Pour explorer ces sections le plus efficacement possible, nous allons commencer par étudier la plus simple : `files`. Puis nous traiterons les deux sections les plus utilisées `fileSets` et `dependencySets`. Une fois que vous avez compris le fonctionnement de `fileSets` et de `dependencySets`, il est beaucoup plus facile de comprendre comment fonctionnent `repositories` et `moduleSets`.

14.5.1. Section `files`

La section `files` est la partie la plus simple du descripteur d'assembly, elle a été conçue pour les fichiers qui ont une position relative au répertoire du projet. Avec cette section vous avez un total contrôle sur les fichiers qui seront inclus dans votre assembly, quel seront leurs noms et où ils se positionneront dans l'archive.

Exemple 14.6. Ajout d'un fichier JAR dans un assembly avec la balise `files`

```
<assembly>
...
<files>
  <file>
    <source>target/my-app-1.0.jar</source>
    <outputDirectory>lib</outputDirectory>
    <destName>my-app.jar</destName>
    <fileMode>0644</fileMode>
  </file>
</files>
...
</assembly>
```

Supposons que vous construisiez le projet `my-app` dans sa version `1.0`, la configuration de l'Exemple 14.6, « Ajout d'un fichier JAR dans un assembly avec la balise `files` », permet d'inclure le JAR de votre projet dans le répertoire `lib/` de l'assembly, en supprimant le numéro de version pour

obtenir au final un fichier renommé en `my-app.jar`. Ensuite, il faut rendre ce JAR lisible par tout le monde et éditable son propriétaire (c'est ce que que représente le mode 0644 pour les fichiers avec le format sur quatre chiffres des permissions Unix). Pour plus d'informations sur le format de la valeur de la balise `fileMode`, consultez l'explication de Wikipédia sur le système octal de représentation des droits².

Vous pouvez construire un assembly très complexe avec ces balises `file`, si vous connaissez la liste exhaustive des fichiers à inclure. Même si vous ne connaissez pas cette liste au lancement du build, vous pouvez très facilement utiliser un plugin Maven personnalisé pour produire ce descripteur d'assembly avec des références comme celui ci-dessus. Si la section `files` vous permet un contrôle très fin des permissions, de la position et du nom de chaque fichier dans l'archive assembly, écrire une balise `file` pour chaque fichier est une tâche pénible. La plupart du temps vous allez travailler sur des groupes de fichiers et de dépendances grâce à la balise `fileSets`. Les quatre sections suivantes sont conçues pour vous permettre de sélectionner des listes de fichiers correspondant à des critères particuliers.

14.5.2. Section `fileSets`

Comme pour la section `files`, les `fileSets` correspondent à des fichiers qui ont une position relative par rapport aux répertoires du projet. Cependant, au contraire de la section `files`, `fileSets` décrit des ensembles de fichiers, dont le nom et le chemin respectent (ou ne respectent pas) un certain format, et les répertoires dans lesquels on peut les trouver. La forme la plus simple de `fileSet` spécifie uniquement un répertoire où se trouvent les fichiers :

```
<assembly>
...
<fileSets>
  <fileSet>
    <directory>src/main/java</directory>
  </fileSet>
</fileSets>
...
</assembly>
```

Cet ensemble de fichiers se compose de tout le contenu du répertoire `src/main/java` de notre projet. Il profite des valeurs par défaut des nombreux paramètres de cette section, aussi regardons les brièvement.

Tout d'abord, vous avez remarqué que nous n'avons pas indiqué où nous souhaitons mettre les fichiers à inclure dans l'assembly. Par défaut, le répertoire de destination (que l'on peut préciser au moyen de la balise `outputDirectory`) est le même que le répertoire source (dans notre cas, `src/main/java`). De plus, nous n'avons pas spécifié de format à respecter pour l'inclusion ou l'exclusion de fichiers. Dans ce cas, le comportement par défaut est de sélectionner tous les fichiers qui se trouvent dans le répertoire source avec quelques exceptions importantes. Les exceptions à cette règle sont surtout les fichiers et les répertoires de métadonnées des outils de gestion de configuration. Ces exceptions par défaut sont utilisées lorsque la balise `useDefaultExcludes` est à `true`, ce qui est le cas par défaut. Lorsque l'option `useDefaultExcludes` est activée, les répertoires tels que `.svn/` et `CVS/` sont exclus d'office

² http://en.wikipedia.org/wiki/File_system_permissions#Octal_notation_and_additional_permissions

de la liste des fichiers à inclure dans l'archive assembly. La Section 14.5.3, « Patterns d'exclusion par défaut pour la balise `fileSets` » présente une liste détaillée des formats d'exclusion par défaut.

Si vous voulez contrôler plus finement cet ensemble de fichiers, vous pouvez le spécifier explicitement. L'Exemple 14.7, « Inclusion de fichiers avec la balise `fileSet` » montre une balise `fileSet` qui spécifie toutes les valeurs par défaut.

Exemple 14.7. Inclusion de fichiers avec la balise `fileSet`

```
<assembly>
...
<fileSets>
  <fileSet>
    <directory>src/main/java</directory>
    <outputDirectory>src/main/java</outputDirectory>
    <includes>
      <include>**</include>
    </includes>
    <useDefaultExcludes>true</useDefaultExcludes>
    <fileMode>0644</fileMode>
    <directoryMode>0755</directoryMode>
  </fileSet>
</fileSets>
...
</assembly>
```

Les sections `includes` se composent d'une liste de balises `include` qui contiennent des patterns de chemin et de nom de fichier. Ces formats peuvent avoir un ou plusieurs jokers comme `***` qui correspond à un ou plusieurs répertoires, `*` qui correspond à une partie d'un nom de fichier ou `?` qui remplace un caractère quelconque dans un nom de fichier. L'Exemple 14.7, « Inclusion de fichiers avec la balise `fileSet` » utilise une balise `fileMode` pour spécifier que les fichiers de ce groupe devront être lisibles par tous mais éditables uniquement par le propriétaire. Comme la balise `fileSet` peut inclure des répertoires, il existe aussi une balise `directoryMode` pour spécifier le mode du répertoire. Elle fonctionne comme `fileMode`. Comme il faut avoir les droits d'exécution sur un répertoire pour pouvoir en lister le contenu, il faut s'assurer que les répertoires sont bien exécutable en plus d'être lisibles. Comme pour les fichiers, seul le propriétaire peut modifier les répertoires et leurs contenus dans cet exemple.

La balise `fileSet` possède des options supplémentaires. Premièrement, elle peut contenir une balise `excludes` de la même forme que la balise `includes`. Les patterns d'exclusion vous permettent d'exclure des fichiers spécifiques d'un `fileSet` s'ils correspondent au pattern spécifié. Les patterns d'inclusion prennent le pas sur ceux d'exclusion. Vous pouvez aussi mettre à `true` l'élément `filtering` si vous voulez remplacer les expressions dans les fichiers ainsi sélectionnés par les valeurs des propriétés. Les expressions sont délimitées soit par `${` et `}` (expression standard Maven, par exemple `${project.groupId}`) ou par `@` suivi de `@` (expression standard Ant, par exemple `@project.groupId@`). Vous pouvez choisir la fin de ligne dans vos fichiers grâce à la balise `lineEnding`. Les valeurs autorisées pour la balise `lineEnding` sont :

keep

Conserve la fin de ligne des fichiers originaux. (C'est la valeur par défaut).

unix

Fin de ligne de type Unix

lf

Juste le caractère Nouvelle Ligne

dos

Fin de ligne de type MS-DOS

crlf

Caractère Retour à la Ligne suivi du caractère Nouvelle Ligne

Enfin, si vous voulez vous assurer que tous les éléments de sélection de fichiers sont utilisés, vous pouvez mettre la balise `useStrictFiltering` à `true` (la valeur par défaut est `false`). Cela peut-être très utile si des patterns inutilisés indiquent des fichiers manquants dans un répertoire intermédiaire. Lorsque la balise `useStrictFiltering` est à `true`, le plugin Assembly échouera si un pattern d'inclusion n'est pas satisfait. En d'autres termes, si vous avez un pattern d'inclusion qui inclut un fichier d'un build et que ce fichier est absent alors, lorsque la balise `useStrictFiltering` est à `true` le build Maven échouera lorsqu'il ne trouvera pas ce fichier.

14.5.3. Patterns d'exclusion par défaut pour la balise `fileSets`

Quand vous utilisez les patterns d'exclusion par défaut, le plugin Maven Assembly va ignorer d'autres fichiers en plus des répertoires SVN et CVS. Par défaut, les patterns d'exclusion sont la classe `DirectoryScanner`³ du projet `plexus-utils`⁴ hébergé chez Codehaus. La liste des patterns d'exclusion est définie sous la forme d'un tableau statique de `Strings` appelé `DEFAULTEXCLUDES` dans `DirectoryScanner`. Le contenu de cette variable est présenté dans l'Exemple 14.8, « Définitions des patterns d'exclusion de Plexus Utils ».

Exemple 14.8. Définitions des patterns d'exclusion de Plexus Utils

```
public static final String[] DEFAULTEXCLUDES = {
    // Miscellaneous typical temporary files
    "**/*~",
    "**/#**#",
    "**/.#**",
    "**/.*",
    "**/.*",

    // CVS
    "**/CVS",
```

³ <http://svn.codehaus.org/plexus/plexus-utils/trunk/src/main/java/org/codehaus/plexus/util/DirectoryScanner.java>

⁴ <http://plexus.codehaus.org/plexus-utils/>

```

    "**/CVS/**",
    "**/.cvsignore",

    // SCCS
    "**/SCCS",
    "**/SCCS/**",

    // Visual SourceSafe
    "**/vssver.scc",

    // Subversion
    "**/.svn",
    "**/.svn/**",

    // Arch
    "**/.arch-ids",
    "**/.arch-ids/**",

    //Bazaar
    "**/.bZR",
    "**/.bZR/**",

    //SurroundSCM
    "**/.MySCMServerInfo",

    // Mac
    "**/.DS_Store"
};

```

Ce tableau de patterns par défaut exclut les fichiers temporaire d'éditeurs tels que GNU Emacs⁵, les fichiers temporaires les plus classiques sous Mac et les fichiers des outils de gestion de configuration (même si Visual SourceSafe est plus une malédiction qu'un outil de gestion de configuration). Si vous avez besoin de redéfinir ces patterns d'exclusion par défaut, mettez `useDefaultExcludes` à `false` et définissez votre propre liste de patterns d'exclusion dans votre descripteur d'assembly.

14.5.4. Section `dependencySets`

Une des fonctionnalités les plus demandées pour les assemblées est l'ajout des dépendances d'un projet dans l'archive assembly. Les balises `files` et `fileSets` traitent les fichiers de votre projet, cependant une dépendance n'est pas un fichier qui se trouve dans les répertoires de votre projet. Les artefacts dont dépend votre projet sont résolus par Maven lors du build. Les artefacts de dépendance sont une notion abstraite, il n'ont pas un chemin bien défini et sont résolus en utilisant un ensemble symbolique de coordonnées Maven. Puisque `file` et `fileSet` exigent un chemin réel, les dépendances seront incluses ou exclues d'un assembly par une combinaison de coordonnées Maven et de scope.

La forme la plus simple de la balise `dependencySet` est une balise vide :

```
<assembly>
```

⁵ <http://www.gnu.org/software/emacs/>

```
...
<dependencySets>
  <dependencySet/>
</dependencySets>
...
</assembly>
```

La balise `dependencySet` ci-dessus va rechercher toutes les dépendances d'exécution de votre projet (le scope runtime scope inclut le contenu du scope compile implicitement), et va ensuite les ajouter au répertoire racine de l'archive `assembly`. Elle va aussi copier l'artefact principal de votre projet, s'il existe, dans ce répertoire racine.



Note

Une minute ? Je pensais que la balise `dependencySet` gérait l'inclusion des dépendances de mon projet, pas son archive principale ? Cet effet secondaire non-intuitif était un bug très utilisé de la version 2.1 du plugin `Assembly`, et, comme Maven accorde une très grande importance à la compatibilité ascendante, ce comportement non-intuitif et incorrect a dû être préservé pour la version 2.2. Vous pouvez interdire ce comportement en mettant à `false` la balise `useProjectArtifact`.

Même si l'ensemble des dépendances par défaut peut être très utile sans configuration supplémentaire, cette section du descripteur d'`assembly` supporte de nombreuses options de configuration pour vous permettre d'adapter son comportement à vos besoins spécifiques. Par exemple, la première chose que vous pourriez vouloir faire par rapport à cette liste de dépendances est d'exclure l'artefact du projet en mettant `useProjectArtifact` à `false` (sa valeur par défaut est à `true` pour des raisons historiques). Cela vous permettra de gérer le résultat du build de votre projet indépendamment des fichiers des dépendances. De même, vous pouvez choisir de décompresser les dépendances en mettant la balise `unpack` à `true` (elle est à `false` par défaut). Lorsque l'option `unpack` est activée, le plugin `Assembly` va décompresser les contenus des dépendances dans le répertoire racine de l'archive.

À partir maintenant, vous pouvez faire plusieurs choses avec cet ensemble de dépendances. Les sections qui vont suivre présentent comment définir l'endroit où seront enregistrées ces dépendances et comment inclure et exclure des dépendances selon leur scope. Enfin, nous nous étendrons sur cette fonctionnalité de décompression des dépendances en explorant certaines de ses options avancées.

14.5.4.1. Configurer l'emplacement des dépendances

C'est la combinaison de deux options de configuration qui détermine où va se retrouver un fichier de dépendance dans l'archive `assembly` : `outputDirectory` et `outputFileNameMapping`. Vous pouvez configurer l'emplacement des dépendances dans votre `assembly` selon les propriétés des artefacts des dépendances. Disons que vous voulez mettre toutes vos dépendances dans des répertoires qui correspondent au `groupId` de l'artefact. Dans ce cas, vous pouvez utiliser la balise `outputDirectory` sous `dependencySet` et lui donner pour valeur :

```
<assembly>
```

```

...
<dependencySets>
  <dependencySet>
    <outputDirectory>${artifact.groupId}</outputDirectory>
  </dependencySet>
</dependencySets>
...
</assembly>

```

Cela aura pour effet de mettre chaque dépendance dans un répertoire dont le nom correspond au `groupId` de son artefact.

Si vous voulez aller plus loin dans la personnalisation et que vous souhaitez retirer le numéro de version du nom de fichier des dépendances. Il suffit de configurer le renommage des dépendances grâce à la balise `outputFileNameMapping` comme suit :

```

<assembly>
...
<dependencySets>
  <dependencySet>
    <outputDirectory>${artifact.groupId}</outputDirectory>
    <outputFileNameMapping>
      ${artifact.artifactId}.${artifact.extension}
    </outputFileNameMapping>
  </dependencySet>
</dependencySets>
...
</assembly>

```

Dans l'exemple précédent, une dépendance sur `commons:commons-codec` version 1.3, se retrouverait dans le fichier `commons/commons-codec.jar`.

14.5.4.2. Interpolation de propriétés pour l'emplacement des dépendances

Comme nous l'avons déjà mentionné dans la Section 14.4.1, « Référence de propriété dans un descripteur d'assembly », aucun de ces éléments n'est interpolé avec le reste du descripteur d'assembly. En effet, leurs valeurs brutes doivent être traitées par des interpréteurs d'expression d'artefact spécifiques.

Les expressions utilisables pour ces deux éléments sont légèrement différentes. Les expressions `${project.*}`, `${pom.*}` et `${*}` qui sont disponibles dans le POM et dans le reste du descripteur le sont aussi dans ces deux cas. Pour la balise `outputFileNameMapping` le traitement suivant est appliqué pour résoudre les expressions :

1. Si l'expression correspond au pattern `${artifact.*}` :
 - a. Recherche dans l'instance `Artifact` de la dépendance (résolution des attributs : `groupId`, `artifactId`, `version`, `baseVersion`, `scope`, `classifier` et `file.*`)
 - b. Recherche dans l'instance `ArtifactHandler` de la dépendance (résolution de : `expression`)

- c. Recherche dans l'instance du projet associé à l'artefact de la dépendance (résolution : principalement les propriétés du POM)
2. Si l'expression correspond au pattern `${pom.*}` ou `${project.*}`:
 - a. Recherche dans l'instance du projet (`MavenProject`) du build en cours.
3. Si l'expression correspond au pattern `${dashClassifier?}` et que l'instance `Artifact` de la dépendance a un classifieur, résoud jusqu'au classifieur précédé d'un tiret (`-classifier`). Sinon résoud comme étant une chaîne vide.
4. Essaye de résoudre l'expression avec l'instance du projet du build en cours.
5. Essaye de résoudre l'expression avec les propriétés du POM du build en cours.
6. Essaye de résoudre l'expression avec les propriétés système disponibles.
7. Essaye de résoudre l'expression avec les variables d'environnement du système d'exploitation.

La valeur de la balise `outputDirectory` est interpolée de manière assez semblable, à la différence près qu'il n'y a pas d'information du type `${artifact.*}` disponible. Seules les instances `${project.*}` des artefacts permettent de résoudre les expressions. Donc les expressions associées à ces patterns ne sont pas disponibles (il s'agit des éléments 1a, 1b, et 3 de la liste ci-dessus).

Comment sait-on quand utiliser `outputDirectory` et `outputFileNameMapping` ? Quand les dépendances sont décompressées, seul `outputDirectory` est utilisé pour calculer l'emplacement final. Quand les dépendances sont gérées comme un tout (c'est à dire non décompressées), les deux éléments `outputDirectory` et `outputFileNameMapping` peuvent être utilisés ensemble. Quand ils le sont, le résultat est équivalent à :

```
<archive-root-dir>/<outputDirectory>/<outputFileNameMapping>
```

Quand `outputDirectory` est absent, il n'est pas utilisé. Quand c'est `outputFileNameMapping` qui est absent, on prend sa valeur par défaut qui est : `${artifact.artifactId}-${artifact.version}${dashClassifier?}.${artifact.extension}`

14.5.4.3. Inclusion et exclusion de dépendance par scope

Dans la Section 9.4, « Dépendances d'un projet », nous avons noté que toutes les dépendances d'un projet ont un scope. Le scope détermine le moment où une dépendance sera utilisée au cours du processus de build. Par exemple, les dépendances dans le scope `test` ne font pas partie du classpath au moment de la compilation du code source du projet ; par contre, elles font partie du classpath de compilation des tests unitaires. C'est fait ainsi car le code source de votre projet ne devrait pas contenir du code spécifique pour les tests. En effet, tester n'est pas une des fonctions de votre projet (c'est une fonction du processus de build du projet). De même, les dépendances dans le scope `provided` doivent être présentes dans l'environnement où sera déployé le projet. Cependant, si un projet dépend d'une dépendance `provided`

particulière, il est possible qu'il ait besoin de celle-ci pour compiler. C'est pour cette raison que les dépendances du scope `provided` sont présentes dans le classpath de compilation mais qu'elles ne sont pas empaquetées avec l'artefact ou un assembly du projet.

Dans cette même Section 9.4, « Dépendances d'un projet », nous avons vu que les scopes de dépendances en induisent d'autres. Par exemple, le scope de dépendance `runtime` inclut les dépendances du scope `compile`, car toutes les dépendances utilisées au moment de la compilation (sauf celles du scope `provided`) seront nécessaires pour l'exécution du code. Les manières dont un scope d'une dépendance directe affecte le scope des dépendances transitives sont très complexes. Dans un descripteur d'assembly Maven, nous pouvons utiliser les scopes pour filtrer des ensembles de dépendances et leur appliquer des traitements différents .

Par exemple, si nous projetons de packager une application web avec Jetty⁶ et créer ainsi une application autonome, nous allons devoir inclure toutes les dépendances du scope `provided` dans les répertoires du Jetty que nous intégrons. Cela nous assurera que ces dépendances `provided` sont effectivement présentes dans l'environnement d'exécution. Les dépendances d'exécution qui ne font pas partie du scope `provided` continuent à aller dans le répertoire `WEB-INF/lib`, donc ces deux ensembles de dépendances doivent être traités distinctement. Ces blocs de dépendances ressemblent au document XML qui suit.

Exemple 14.9. Définition de blocs de dépendances par l'utilisation des scopes

```
<assembly>
...
<dependencySets>
  <dependencySet>
    <scope>provided</scope>
    <outputDirectory>lib/${project.artifactId}</outputDirectory>
  </dependencySet>
  <dependencySet>
    <scope>runtime</scope>
    <outputDirectory>
      webapps/${webContextName}/WEB-INF/lib
    </outputDirectory>
  </dependencySet>
</dependencySets>
...
</assembly>
```

Les dépendances du scope `provided` sont ajoutées au répertoire `lib/` situé à la racine de l'assembly. Ce répertoire va contenir l'ensemble des bibliothèques qui seront incluses dans le classpath global d'exécution de Jetty. Nous utilisons un sous-répertoire dont le nom est l'`artifactId` du projet pour retrouver plus facilement d'où vient une bibliothèque particulière. Les dépendances d'exécution sont incluses dans le répertoire `WEB-INF/lib` de l'application web, qui se trouve dans le sous-répertoire standard de Jetty : `webapps/`, et dont le nom est donné par la configuration de la propriété `webContextName` du POM. Ce que nous venons de faire dans cet exemple est de séparer les

⁶ <http://www.mortbay.org/jetty-6/>

dépendances spécifiques à l'application de celles qui sont présentes dans le classpath global d'un conteneur de servlets.

Cependant, séparer en ne tenant compte que du scope peut s'avérer insuffisant, notamment dans le cas d'applications web. Il est tout à fait concevable qu'une ou plusieurs des dépendances pour l'exécution soient un assemblage de ressources standardisées et non-compilées qui seront utilisées par l'application web. Par exemple, nous pourrions avoir un ensemble d'applications qui réutiliseraient le même ensemble de fichiers de Javascript, CSS, SWF et d'images. Pour faciliter la standardisation de ces ressources, une bonne pratique consiste à les mettre dans une archive et de les déployer sur un dépôt Maven. À partir de ce moment là, elles peuvent être référencées comme une dépendance Maven standard - éventuellement comme une dépendance de type `zip` - normalement dans le scope `runtime`. Souvenez-vous qu'il s'agit de ressources et non d'une dépendance du code de l'application vers un binaire ; et donc il ne faut pas les inclure dans le répertoire `WEB-INF/lib`. Ces archives de ressources doivent être séparées des dépendances binaires d'exécution et décompressées à la racine de l'application web. Pour pouvoir effectuer ce type de séparation, nous allons utiliser les patterns d'inclusion et d'exclusion qui se basent sur les coordonnées d'une dépendance.

En d'autres termes, nous allons supposer que vous avez trois ou quatre applications web qui réutilisent les mêmes ressources et que vous voulez créer un assembly qui met les dépendances du scope `provided` dans le répertoire `lib/`, les dépendances du scope `runtime` dans `webapps/<contextName>/WEB-INF/lib` et enfin décompresse une dépendance particulière du scope `runtime` à la racine de votre application web. Nous pouvons réaliser tout ça car le plugin Assembly permet de définir plusieurs patterns d'inclusion et d'exclusion pour un même élément `dependencySet`. Dans la prochaine section, nous allons essayer de développer cette idée.

14.5.4.4. Configuration fine : inclusion et exclusion de dépendances

Une dépendance de ressources peut être une simple liste de ressources (CSS, Javascript, images ...) dans un projet dont un assembly crée une archive de type ZIP. En fonction des particularités de votre application web, vous voudrez distinguer les dépendances de ressources des dépendances de binaires grâce à leur type. La plupart des applications web vont dépendre d'autres dépendances de type `jar`. Du coup, il est possible de dire avec certitude que toutes les dépendances de type `zip` sont des dépendances de ressources. Or, nous pourrions avoir une situation où les ressources sont conservées sous un format `jar` mais que nous pouvons distinguer les ressources par un classifieur `resources`. Dans tous les cas, nous pouvons spécifier un pattern d'inclusion pour cibler ces dépendances de ressources et appliquer une logique différente que celle utilisée pour les binaires. Nous pouvons effectuer cette distinction par l'intermédiaire de balises `includes` et `excludes` dans le `dependencySet`.

Les balises `includes` et `excludes` sont des listes. Elles acceptent des sous-éléments du type `include` et `exclude`. Chaque balise `include` et `exclude` contient une valeur sous forme de chaîne de caractère et peuvent contenir des caractères jokers. Les dépendances peuvent respecter ces patterns de plusieurs façons. Généralement, trois formats de pattern sont supportés :

```
groupId:artifactId - sans la version
```

Utilisez ce pattern pour trouver des dépendances par `groupId` et `artifactId`

groupId:artifactId:type[:classifieur] - id de conflit

Ce pattern vous permet de fournir un ensemble plus large de coordonnées pour créer des patterns include/exclude plus spécifiques.

groupId:artifactId:type[:classifieur]:version - identité complète de l'artefact

Si vous avez à récupérer un artefact bien spécifique, vous pouvez renseigner l'intégralité des coordonnées.

Tous ces formats de patterns supportent l'utilisation du joker '*', il permet de remplacer n'importe quelle sous-section de l'identité et peut être employé dans plusieurs sous-sections (bloc entre ':'). En outre, notez que la section classifieur du pattern ci-dessus est facultative, les dépendances qui n'ont pas de classifieur ne prennent pas en compte la section 'classificateur' de ce pattern.

Dans l'exemple donné ci-dessus, la distinction essentielle est le type zip de l'artefact et il faut noter qu'aucune des dépendances n'a de classifieur, le pattern suivant serait respecté par toutes les ressources de type :

```
*:zip
```

Le pattern ci-dessus utilise la seconde identité de la dépendance : l'id de conflit. Maintenant que nous avons un pattern qui distingue les dépendances de ressources des dépendances de binaires, nous pouvons modifier notre liste de dépendances pour gérer ces archives différemment :

Exemple 14.10. Utilisation des l'inclusion et d'exclusion de dépendances dans le **dependencySets**

```
<assembly>
...
<dependencySets>
  <dependencySet>
    <scope>provided</scope>
    <outputDirectory>lib/${project.artifactId}</outputDirectory>
  </dependencySet>
  <dependencySet>
    <scope>runtime</scope>
    <outputDirectory>
      webapps/${webContextName}/WEB-INF/lib
    </outputDirectory>
    <excludes>
      <exclude>*:zip</exclude>
    </excludes>
  </dependencySet>
  <dependencySet>
    <scope>runtime</scope>
    <outputDirectory>
      webapps/${webContextName}/resources
    </outputDirectory>
    <includes>
      <include>*:zip</include>
    </includes>
    <unpack>true</unpack>
  </dependencySet>
</dependencySets>
</assembly>
```

```
</dependencySet>
</dependencySets>
...
</assembly>
```

Dans l'Exemple 14.10, « Utilisation des l'inclusion et d'exclusion de dépendances dans le `dependencySets` », la liste de dépendances du scope `runtime` de notre dernier exemple a été mise à jour pour exclure les dépendances de type ressource. Seules les dépendances binaires (c'est à dire celles qui ne sont pas de type `zip`) sont ajoutées dans le répertoire `WEB-INF/lib` de l'application. Les dépendances de ressources sont regroupées dans un seul ensemble de dépendances. Cet ensemble est configuré pour copier ses dépendances dans le répertoire ressource de l'application. La balise `includes` du dernier `dependencySet` annule les exclusions du `dependencySet` précédent. Ainsi, les dépendances ressources sont incluses en utilisant un seul pattern d'identité : `*:zip`. La dernière balise `dependencySet` fait référence à des ressources partagées, elle est configurée pour décompresser celles-ci à la racine de l'application web.

L'Exemple 14.10, « Utilisation des l'inclusion et d'exclusion de dépendances dans le `dependencySets` » présume que le projet contenant les ressources partagées dispose d'un type différent des autres dépendances. Que se passerait-il si celles-ci avaient le même type que celui des autres dépendances ? Comment différencieriez-vous cette dépendance ? Dans ce cas, si une dépendance de ressources partagées a été packagée comme un JAR avec le classificateur de type `resources`, vous pouvez changer le pattern d'identité pour qu'il filtre cette dépendance :

```
*:jar:resources
```

Au lieu d'accepter les artefacts de type `zip` sans classifieur, nous retenons ici les artefacts de type `jar` avec un classifieur `resources`.

Comme pour la section `fileSets`, `dependencySets` supporte l'utilisation du flag `useStrictFiltering`. Lorsque celui-ci est activé, n'importe quel pattern qui n'accepte aucune dépendance fera échouer l'assembly, et donc par conséquent, le build. Cela est très pratique comme soupape de sécurité, pour vous assurer que les dépendances de votre projet et votre descripteur d'assembly sont bien synchronisés. Par défaut, ce flag est désactivé pour des raisons de rétrocompatibilité.

14.5.4.5. Dépendances transitives, pièces jointes et artefacts de projet

La section `dependencySet` vous propose deux autres mécanismes pour vous aider dans le choix de vos artefacts : la sélection de dépendances transitives et la possibilité de travailler avec des artefacts du projet. Ces deux fonctionnalités proviennent de la nécessité de supporter des configurations existantes qui utilisent une définition un peu plus libérale du mot "dépendance". Comme premier exemple, examinons l'artefact principal du projet. Dans la majorité des cas, il ne doit pas être considéré comme une dépendance. Pourtant, les plus anciennes versions du plugin Assembly l'utilisaient dans le calcul des dépendances. Pour fournir une rétrocompatibilité avec cette "fonctionnalité", la version 2.2 du plugin Assembly dispose d'un flag à mettre dans le `dependencySet`, celui-ci est appelé

`useProjectArtifact`. La valeur par défaut de ce flag est `true`. Par défaut, un ensemble de dépendances prendra en compte l'artefact du projet dans la sélection de ses dépendances. Si vous préférez gérer l'artefact du projet séparément, affectez ce flag à `false`.



Astuce

Les auteurs de ce livre vous recommandent de toujours laisser le flag `useProjectArtifact` à `false`.

Comme extension naturelle à l'inclusion d'un artefact projet, les artefacts rattachés à un projet peuvent également être gérés par un `dependencySet` en utilisant le flag `useProjectAttachments` (celui-ci est désactivé par défaut). Activer ce flag permet aux patterns qui précisent des classifieurs et des types d'artefacts de prendre en compte les artefacts rattachés. Ils partagent la même identité `groupId/artifactId/version`, mais différents `type` et `classifieur`. Cette fonctionnalité peut s'avérer utile pour inclure les JARs de Javadoc ou de source dans un `assembly`.

En plus de traiter avec des artefacts du projet, il est également possible de prendre en compte les dépendances transitives pour calculer un ensemble de dépendances grâce à deux paramètres. Le premier, appelé `useTransitiveDependencies` (et activé par défaut), permet d'activer l'inclusion des dépendances transitives lors de la sélection par pattern. Comme exemple pour savoir comment il peut être utilisé, considérez ce qui arrive lorsque votre POM a une dépendance sur un autre `assembly`. Cet `assembly` aura (probablement) un classifieur qui permet de le distinguer de l'artefact principal du projet, ce qui en fait une pièce jointe. Cependant, une particularité du processus de résolution des dépendances Maven est que les informations des dépendances transitives pour l'artefact principal sont toujours utilisées pour résoudre l'artefact de l'`assembly`. Si l'`assembly` package les dépendances en lui, dans ce cas, utiliser la résolution des dépendances transitives dupliquerait ces dépendances. Pour éviter cela, nous pouvons simplement utiliser le flag `useTransitiveDependencies` en le positionnant à `false`.

L'autre flag permettant de résoudre les dépendances est plus subtil. Il est appelé `useTransitiveFiltering` et sa valeur par défaut est `false`. Pour comprendre ce que fait ce flag, nous devons d'abord comprendre quelles informations sont disponibles pour un artefact donné lors du processus de résolution des dépendances. Quand un artefact est une dépendance d'un autre (c'est à dire avec au moins un niveau intermédiaire entre lui et votre POM), il possède ce que Maven appelle un "chemin de dépendances". Il s'agit d'une liste de chaînes de caractères qui correspondent à l'identité complète de chaque artefact (`groupId:artifactId:type:[classifieur:]version`) de dépendances entre cet artefact et votre POM. Si vous vous rappelez des trois types d'identités des artefacts disponibles pour écrire un pattern, vous remarquerez que les entrées dans ce chemin de dépendances - l'identité complète de l'artefact - correspondent au troisième type. Lorsque le flag `useTransitiveFiltering` est positionné à `true`, toutes les entrées du chemin de dépendances d'un artefact peuvent agir sur l'inclusion ou l'exclusion de cet artefact.

Si vous envisagez l'utilisation de ce filtrage transitif, prenez garde ! Un artefact peut être inclus à partir de nombreux emplacement dans un graphe de dépendance, mais dans Maven 2.0.9, seul le premier chemin

de dépendances est utilisé pour ce type de sélection. Cela peut conduire à des problèmes difficiles à résoudre dans la collecte des dépendances de votre projet.



Avertissement

La plupart des assemblies ne nécessitent pas ce niveau de contrôle sur les listes de dépendances. Réfléchissez attentivement pour savoir si vous en avez vraiment besoin. Astuce : ce n'est probablement pas le cas.

14.5.4.6. Options avancées de dépaquetage

Comme nous l'avons vu précédemment, certaines dépendances de projet ont besoin d'être décompressées durant la création d'un assembly. Dans les exemples ci-dessus, la décision de savoir s'il fallait décompresser ou pas était simple. Elle ne tient pas compte de ce qui doit être dépaqueté, ou plus important, de ce qui ne doit pas l'être. Pour obtenir plus de contrôle sur le processus de dépaquetage, vous pouvez configurer la balise `unpackOptions` dans votre `dependencySet`. Ainsi, vous avez la possibilité de choisir quels fichiers vous voulez inclure dans votre assembly, et quels fichiers doivent être filtrés pour résoudre des expressions à partir des informations du POM. En fait, les options disponibles pour dépaqueter vos ensembles de dépendances sont similaires à celles disponibles pour l'inclusion ou l'exclusion de fichier.

Pour continuer notre exemple d'application web, supposons que certaines dépendances de ressources aient été zippées avec un fichier qui décrit les détails de leur licence de distribution. Dans le cas de notre application web, ces licences se présenteront sous la forme d'un fichier `NOTICES` inclu dans notre paquetage. Pour exclure ce fichier, ajoutons-le simplement aux options de dépaquetage dans la balise `dependencySet` qui gère les artefacts de ressources :

Exemple 14.11. Exclusion de fichiers dans le dépaquetage d'une dépendance

```
<assembly>
...
<dependencySets>
  <dependencySet>
    <scope>runtime</scope>
    <outputDirectory>
      webapps/${webContextName}/resources
    </outputDirectory>
    <includes>
      <include>*:zip</include>
    </includes>
    <unpack>true</unpack>
    <unpackOptions>
      <excludes>
        <exclude>**/LICENSE*</exclude>
      </excludes>
    </unpackOptions>
  </dependencySet>
</dependencySets>
...
```

```
</assembly>
```

Notez que la balise `exclude` que nous utilisons ressemble beaucoup à celle de la déclaration du `fileSet`. Ici, nous excluons les fichiers qui commencent par le mot `LICENSE` des répertoires de nos artefacts de ressources. Vous pouvez considérer la section des options de dépaquetage comme une seconde balise `fileSet` appliquée à chaque dépendance faisant partie de cet ensemble. En d'autres mots, il s'agit d'un `fileSet` utilisé pour le dépaquetage des dépendances. Tout comme nous avons spécifié un modèle d'exclusion pour les dossiers dans les dépendances de ressources afin de bloquer certains fichiers, vous pouvez également choisir un ensemble restreint de fichiers à inclure en utilisant la section `includes`. La même configuration que celle définissant inclusions et exclusions dans les `fileSets` peut-être utilisée dans le cas de balises `unpackOptions`.

En plus de mécanisme d'inclusion et d'exclusion, les options de dépaquetage sur une liste de dépendances peuvent également utiliser un flag `filtering`, dont la valeur par défaut est `false`. Encore une fois, cela ressemble beaucoup au mécanisme proposé par les ensembles de fichiers discuté ci-dessus. Les expressions peuvent utiliser, soit la syntaxe Maven : `${property}`, soit la syntaxe Ant : `@property@`. Le filtrage des ressources est particulièrement intéressant pour les ensembles de dépendances. Elle permet de créer des modèles de ressources normalisés et versionnés qui peuvent être personnalisés à chaque intégration à un `assembly`. Une fois que vous maîtrisez cette fonctionnalité de filtrage, que vous avez dépaqueté les dépendances de ressources partagées, vous serez en mesure de commencer à éliminer et centraliser certaines ressources partagées.

14.5.4.7. Résumé des ensembles de dépendances

Au final, il est important de mentionner que les ensembles de dépendances supportent les mêmes options de configuration `fileMode` et `directoryMode` que les ensembles de fichiers. Cependant, vous devrez vous rappeler que l'option `directoryMode` ne sera utilisée que si les dépendances sont dépaquetées.

14.5.5. La balise `moduleSets`

Les builds multimodules sont généralement liés par les balises `parent` et `modules` dans les POMs. Typiquement, les POMs parents spécifient leurs fils dans une section `modules`, qui, en temps normal, aura pour effet de les inclure dans la procédure d'exécution du build du projet parent. La relation entre ces deux projets, et comment elle a été contruite, peut avoir des implications importantes sur la manière dont le plugin `Assembly` participe à ce processus. Nous discuterons de cela un peu plus tard. Pour l'instant, contentons-nous de garder à l'esprit la relation parent-enfant pendant que nous disctons de la section `moduleSets`.

Les projets sont construits sous la forme d'un projet multimodule parce qu'ils font partie d'un système plus vaste. Ces projets sont conçus pour être utilisés ensemble, un module unique dans un build important n'a que peu de valeur en lui-même. De cette façon, la structure du build du projet est liée à la façon dont nous espérons que ce projet (et ses modules) soit utilisé. Si vous considérez le projet du point de vue de l'utilisateur, il semble logique que l'objectif final de ce build soit de construire et distribuer un seul fichier qu'il pourra directement déployer sans trop de soucis. Comme les builds multimodules Maven s'appuient habituellement sur une structure top-down, où les informations de dépendances, les configurations de

plugin et bien d'autres informations sont héritées du projet parent par l'enfant, il semble naturel que la tâche de transformation de ces modules en un fichier unique et distribuable doive incomber au projet de plus haut niveau dans la hiérarchie. C'est là qu'intervient la balise `moduleSet`.

Les ensembles de modules permettent l'inclusion de ressources appartenant à chacun des modules du projet dans l'assembly final. Tout comme vous pouvez choisir un groupe de fichiers à inclure dans un assembly en utilisant les balises `fileSet` et `dependencySet`, vous pouvez inclure un ensemble de fichiers et de ressources en utilisant la balise `moduleSet` pour se référer aux modules d'un build multimodule. Cela est rendu possible grâce aux deux types d'inclusion spécifiques aux modules : l'un basé sur les fichiers, l'autre sur les artefacts. Avant d'entrer dans les particularités ces deux types d'inclusion de modules ressources dans un assembly, regardons un peu de comment sélectionner les modules à traiter.

14.5.5.1. Sélection des modules

Maintenant, vous devriez commencer à maîtriser les patterns `includes/excludes`. Ils sont également utilisés dans les descripteurs d'assembly pour sélectionner les fichiers et les dépendances. Dans un descripteur d'assembly, lorsque vous faites référence à des modules, vous pouvez également utiliser ces patterns `includes/excludes` pour définir les règles qui s'appliquent sur différents ensembles de modules. La particularité des patterns `includes` et `excludes` d'un `moduleSet` est qu'ils ne permettent pas d'utiliser de jokers. (Du moins dans Maven 2.2-beta-2, cette fonctionnalité n'ayant pas été très demandée, elle n'a donc pas été implémentée.) À la place de cela, chaque balise `include` et `exclude` correspond simplement au `groupId` et à l'`artifactId` du module, séparés par un caractère ':' ainsi :

```
groupId:artifactId
```

En plus des balises `includes` et `excludes`, le `moduleSet` propose d'autres outils de sélection : le flag `includeSubModules` (dont la valeur par défaut est `true`). La relation parent-enfant d'une structure multimodule n'est pas limitée à deux niveaux d'un projet. En fait, vous pouvez inclure un module, peu importe son niveau de profondeur dans votre build. Chaque projet qui est un module d'un module du projet courant est considéré comme un sous-module. Dans certains cas, vous souhaitez construire chaque module séparément (y compris les sous-modules). Pour cela, affectez simplement la valeur du flag `useSubModules` à `true`.

Lorsque vous essayez d'inclure des fichiers de la structure de répertoires de chaque module, vous souhaitez déclarer cette structure de répertoires une seule fois. Si votre structure de répertoires du projet correspond aux déclarations des relations parent-enfant définies dans les POMs, alors les patterns de fichiers comme `**/src/main/java` peuvent s'appliquer non seulement aux répertoires directs de ce module, mais également pour les répertoires de ses propres modules. Dans ce cas, il n'est pas nécessaire de traiter les sous-modules directement (ils seront traités comme des sous-répertoires des modules de votre projet), et donc il faut affecter la valeur du flag `useSubModules` à `false`.

Une fois que nous avons déterminé comment la sélection module doit s'exécuter sur l'ensemble des modules, nous sommes prêts à choisir ce qu'ils doivent contenir. Comme mentionné ci-dessus, il est possible d'inclure des fichiers ou des artefacts provenant du module du projet.

14.5.5.2. Balise `sources`

Supposez que vous désirez inclure le code sources de tous les modules dans votre assembly, mais que vous voulez exclure un module en particulier. Peut-être avez-vous un projet appelé `secret-sauce` qui contient du code secret et sensible que vous ne voulez pas distribuer dans votre projet. Le moyen le plus simple d'effectuer cela est d'utiliser la balise `moduleSet` qui inclus chaque répertoire d'un projet dans `${module.basedir.name}` et qui exclu le module `secret-sauce` de l'assembly.

Exemple 14.12. Inclusion et exclusion de modules dans un `moduleSet`

```
<assembly>
...
<moduleSets>
  <moduleSet>
    <includeSubModules>false</includeSubModules>
    <excludes>
      <exclude>
        com.mycompany.application:secret-sauce
      </exclude>
    </excludes>
    <sources>
      <outputDirectoryMapping>
        ${module.basedir.name}
      </outputDirectoryMapping>
      <excludeSubModuleDirectories>
        false
      </excludeSubModuleDirectories>
      <fileSets>
        <fileSet>
          <directory></directory>
          <excludes>
            <exclude>**/target</exclude>
          </excludes>
        </fileSet>
      </fileSets>
    </sources>
  </moduleSet>
</moduleSets>
...
</assembly>
```

Dans l'Exemple 14.12, « Inclusion et exclusion de modules dans un `moduleSet` », puisque nous devons gérer les sources de chaque module, il est plus simple de traiter seulement les modules directs du projet en cours, en manipulant les sous-modules avec le joker sur le chemin/fichier. Renseignez l'élément `includeSubModules` à `false`. Ainsi, nous n'avons donc pas à nous soucier de l'apparition de sous-modules dans le répertoire racine de l'archive assebly. La balise `exclude` s'occupera d'exclure votre module secret `secret-sauce`.

Normalement, les sources d'un module sont incluses dans l'assembly dans un sous-répertoire qui porte le nom de l'`artifactId` du module. Toutefois, comme Maven permet d'avoir des

modules dans des répertoires dont le nom ne correspond pas à leur `artifactId`, il est souvent préférable d'utiliser une expression `${module.basedir.name}` pour préserver le nom du répertoire du module courant. (`${module.basedir.name}` revient au même que d'appeler la méthode `MavenProject.getBasedir().getName()`). Il est important de se rappeler que les modules ne sont pas nécessairement des sous-répertoires du projet qui les déclare. Si votre projet possède une structure un peu particulière, vous pouvez avoir besoin de recourir à la déclaration de balises `moduleSet` spéciales qui sauront comprendre et tenir compte des particularités de votre projet.



Avertissement

Pour essayer de minimiser les particularités votre projet, comme Maven est flexible, si vous vous surprenez à écrire trop de configuration, c'est qu'il y a probablement un moyen plus facile d'y arriver.

Continuons à parcourir l'Exemple 14.12, « Inclusion et exclusion de modules dans un `moduleSet` ». Comme nous ne traitons pas les sous-modules de manière explicite, nous devons faire en sorte que les répertoires des sous-modules ne soient pas exclus des répertoires sources que nous avons pour chaque module direct. Affecter le flag `excludeSubModuleDirectories` à `false` permet d'appliquer les mêmes patterns de fichiers sur les structures de répertoires du module en cours de traitement et sur ses sous-modules. Enfin, dans l'Exemple 14.12, « Inclusion et exclusion de modules dans un `moduleSet` », le résultat produit par chacun des modules ne nous intéresse pas. Nous avons donc exclu le répertoire `target/` de tous les modules.

Il est intéressant de mentionner que la balise `sources` peut inclure des éléments de type `fileSet` directement ou dans ses sous-balises imbriquées. Ces balises de configuration sont utilisées pour fournir une rétrocompatibilité avec les anciennes versions du plugin Assembly (versions 2.1 et précédentes) qui ne prenaient pas en charge plusieurs ensembles de fichiers pour un même module sans créer de `modulesSet` séparés. Elles sont dépréciées, vous ne devez pas les utiliser.

14.5.5.3. Interpolation de l'`outputDirectoryMapping` dans les `moduleSets`

Dans la Section 14.5.4.1, « Configurer l'emplacement des dépendances », nous avons utilisé la balise `outputDirectoryMapping` pour changer le nom du répertoire sous lequel est incluse chaque source des modules. Les expressions contenues dans cette balise sont résolues exactement de la même manière que celles de la balise `outputFileNameMapping`, qui utilisait des sets de dépendances (référez-vous à l'explication de cet algorithme dans la section Section 14.5.4, « Section `dependencySets` »).

Dans l'Exemple 14.12, « Inclusion et exclusion de modules dans un `moduleSet` », vous avons utilisé l'expression `${module.basedir.name}`. Vous avez peut-être remarqué que le début de cette expression, `module`, n'est pas listé dans l'algorithme de résolution des expressions de la section Section 14.5.4, « Section `dependencySets` ». Cet objet à la base de cette expression est spécifique à la configuration des `moduleSets`. Il fonctionne de la même manière que les références à `${artifact.*}` disponibles pour la balise `outputFileNameMapping`, à la différence qu'il s'applique aux instances `MavenProject`, `Artifact` et `ArtifactHandler` du module au lieu de celles d'un artefact de dépendance.

14.5.5.4. Balise `binaries`

Tout comme la balise `sources` se charge de l'inclusion des sources d'un module, la balise `binaries` se charge d'inclure le résultat du build d'un module, ou ses artefacts. Bien que cette section fonctionne essentiellement comme une façon de spécifier un `dependencySets` à appliquer à chaque module de la série, quelques fonctionnalités propres aux artefacts des modules méritent d'être explorées : `attachmentClassifier` et `includeDependencies`. En plus de cela, la balise `binaries` contient des options de configuration similaires à la balise `dependencySet`, en rapport avec la manipulation de l'artefact du module lui-même. Il s'agit des balises : `unpack`, `outputFileNameMapping`, `outputDirectory`, `directoryMode` et `fileMode`. Enfin, ces balises `binaries` d'un module peuvent contenir une balise `dependencySets` pour spécifier comment les dépendances de chaque module doivent être incluses dans l'assembly. D'abord, jetons un coup d'oeil à la façon dont ces options peuvent être utilisées pour gérer les artefacts propres au module.

Supposons que nous voulons inclure les `jars` de Javadoc de nos modules dans notre assembly. Dans ce cas, nous ne nous soucions pas de l'inclusion des dépendances, nous voulons simplement ajouter le `jar` de la Javadoc. Toutefois, comme ce `jar` un peu particulier est toujours présent en tant que pièce jointe de l'artefact principal, nous devons spécifier le classifieur à utiliser pour le récupérer. Pour simplifier, nous ne couvrirons pas dépaquetage des `jars` de Javadoc des modules, puisque la configuration est exactement la même que celle utilisée pour les dépendances que nous avons déjà traitée dans ce chapitre. Le `moduleSet` devrait ressembler à l'Exemple 14.13, « Inclure la Javadoc des modules dans un assembly ».

Exemple 14.13. Inclure la Javadoc des modules dans un assembly

```
<assembly>
...
<moduleSets>
  <moduleSet>
    <binaries>
      <attachmentClassifier>javadoc</attachmentClassifier>
      <includeDependencies>>false</includeDependencies>
      <outputDirectory>apidoc-jars</outputDirectory>
    </binaries>
  </moduleSet>
</moduleSets>
...
</assembly>
```

Dans l'Exemple 14.13, « Inclure la Javadoc des modules dans un assembly », vous ne spécifiez pas directement de valeur au flag `includeSubModules` flag, celui-ci est activé par défaut. Cependant, nous tenons absolument à traiter tous les modules - même les sous-modules - en utilisant ce `moduleSet` : nous n'utilisons aucune sorte de pattern de fichier qui pourrait correspondre à des structures de sous-répertoire à l'intérieur du module. Pour chaque module, la balise `attachmentClassifier` récupère l'artefact qui lui est attaché avec le classifieur Javadoc. La balise `includeDependencies` signale au plugin Assembly que les dépendances des modules ne nous intéressent pas, nous récupérerons juste les

pièces jointes. Enfin, la balise `outputDirectory` demande au plugin Assembly de mettre tous les jars de Javadoc dans un répertoire nommé `apidoc-jars/` en dehors du répertoire de l'assembly.

Nous ne faisons rien de très compliqué dans cet exemple. Cependant, il est important de noter que les mêmes changements pour la résolution des expressions, dont nous avons parlé à propos de la balise `outputDirectoryMapping` de la section `sources`, s'appliquent également ici. Tout ce qui est accessible par `${artifact.*}` dans la configuration de la balise `outputFileNameMapping` du `dependencySet` est également disponible dans `${module.*}`. La même chose s'applique pour toute balise `outputFileNameMapping` lorsqu'elle est utilisée directement dans une balise `binaries`.

Enfin, examinons un exemple dans lequel nous voulons simplement traiter l'artefact du module et ses dépendances dans le scope `runtime`. Dans ce cas, nous voulons mettre les artefacts de chaque module une structure de répertoires séparée, en fonction de l'`artifactId` et de la `version` des modules. Cette configuration du `moduleSet` reste simple et ressemble au code de l'Exemple 14.14, « Inclusion des artefacts d'un module et de ses dépendances dans un assembly » :

Exemple 14.14. Inclusion des artefacts d'un module et de ses dépendances dans un assembly

```
<assembly>
...
<moduleSets>
  <moduleSet>
    <binaries>
      <outputDirectory>
        ${module.artifactId}-${module.version}
      </outputDirectory>
      <dependencySets>
        <dependencySet/>
      </dependencySets>
    </binaries>
  </moduleSet>
</moduleSets>
...
</assembly>
```

Dans l'Exemple 14.14, « Inclusion des artefacts d'un module et de ses dépendances dans un assembly », nous utilisons la balise `dependencySet` en la laissant vide. Comme vous devez inclure toutes les dépendances, par défaut, vous n'avez pas besoin d'effectuer de configuration. Lorsque la balise `outputDirectory` est spécifiée dans la balise `binaires`, toutes les dépendances vont être incluses dans le même répertoire, aux côtés de l'artefact du module. Ainsi, nous n'avons pas besoin de configurer tout cela dans le `dependencySet`.

La plupart du temps, les binaires d'un module restent assez simples. Dans les deux parties - la partie principale, chargée de la manipulation de l'artefact du module lui-même, et la partie chargée des ensembles de dépendances, et qui traite les dépendances du module - les options de configuration restent très similaires à celles des ensembles de dépendances. Bien entendu, la balise `binaires` fournit

également des options pour contrôler quelles dépendances sont incluses et quel artefact principal de projet vous souhaitez utiliser.

Comme pour la balise `source`, la balise `binaries` dispose d'options de configuration qui sont fournies uniquement pour des causes de rétrocompatibilité. Celles-ci devraient être dépréciées, comment l'utilisation des sous-sections `includes` et `excludes`.

14.5.5.5. `moduleSets`, POMs parents et balise `binaries`

Enfin, clôturons cette discussion avec un avertissement. En ce qui concerne les relations parents-module, il existe des interactions subtiles entre le fonctionnement interne de Maven et l'exécution d'un `moduleSet` dans une balise `binaires`. Lorsqu'un POM déclare un parent, ce parent doit être résolu d'une façon ou d'une autre avant que le POM en question puisse être construit. Si un parent est dans un dépôt Maven, pas de problème. Cependant, vous vous exposez à de gros problèmes si le parent dispose d'un POM de plus haut niveau dans le même build, en particulier si le POM parent utilise les binaires de ces modules.

Maven 2.0.9 trie les projets d'un build multimodule en fonction de leurs dépendances, de manière à construire les dépendances d'un projet avant celui-ci. Le problème est que l'élément parent est considéré comme une dépendance, ce qui signifie que le build du projet parent doit être effectué avant que le projet enfant soit construit. Si une partie du build de ce parent inclut la création d'un assembly qui utilise les binaires des modules, ces binaires ne seront pas encore créés et ne pourront donc pas être inclus. Cela provoquera ainsi l'échec de la construction de l'assembly. Il s'agit d'une question complexe et subtile. Elle limite sérieusement l'utilité de la section `binaires` de la partie module du descripteur d'assembly. En fait, à ce sujet, un bug a été créé sur le gestionnaire d'anomalie du plugin Assembly : <http://jira.codehaus.org/browse/MASSEMBLY-97>. Il faut espérer que les futures versions de Maven vont trouver un moyen de réparer cette fonctionnalité, puisque l'obligation de construire le parent en premier n'est pas forcément nécessaire.

14.5.6. Balise `repositories`

Dans le descripteur d'assembly, la balise `repositories` est un élément un peu plus exotique. Peu d'applications (autre que Maven) peuvent profiter pleinement de la structure de répertoires d'un dépôt Maven. Pour cette raison, et parce que nombre de ces fonctionnalités ressemblent étroitement à la balise `dependencySets`, nous passerons très rapidement sur la présentation de cette balise. Dans la plupart des cas, les utilisateurs qui ont compris comment fonctionnent les `dependencySets` n'auront aucun souci à utiliser la balise `repositories` du plugin Assembly. Nous n'allons donc pas illustrer cette balise par un cas d'utilisation. Nous allons simplement nous contenter de vous donner quelques mises en garde pour ceux d'entre vous qui ressentent le besoin d'utiliser la balise `repositories`.

Cela dit, nous avons deux fonctionnalités à mentionner en particulier à propos des balises `repositories`. La première est le flag `includeMetadata`. Lorsque ce flag est activé les métadonnées, comme la liste des versions réelles qui correspondent aux versions virtuelles `-SNAPSHOT`, sont incluses. Par défaut ce flag est désactivé. À l'heure actuelle, les seules métadonnées incluses lorsque ce flag est à `true` sont celles téléchargées comme informations à partir du dépôt central de Maven.

La seconde fonctionnalité est appelée `groupVersionAlignments`. Ici encore, cette balise représente une liste de configurations individuelles `groupVersionAlignment`, dont le but est de normaliser tous les artefacts inclus pour un `groupId` particulier de manière à n'utiliser qu'une seule `version`. Chaque entrée se compose de deux éléments obligatoires : un `id` et une `version`, ainsi qu'une section optionnelle appelée `excludes` qui fournit une liste d'`artifactId` qui doivent être exclus de ce réaligement. Malheureusement, ce remaniement ne semble pas modifier les POMs impliqués dans le dépôt, ni ceux liés à des artefacts réalignés, ni ceux qui dépendent des artefacts réalignés. De ce fait, il est difficile d'imaginer un réel cas d'utilisation pour ce genre de réaligement.

En général, le plus simple est d'utiliser les mêmes principes que ceux des `dependencySets` de votre descripteur d'assembly. Même si la balise `repositories` supporte d'autres options, elles sont principalement fournies pour des raisons de rétrocompatibilité, et seront probablement dépréciées dans les prochaines releases.

14.5.7. Gestion du répertoire racine de l'assembly

Maintenant que nous avons parcouru les principales fonctionnalités du descripteur d'assembly, nous pouvons clôturer les discussions sur le contenu de ce descripteur et terminer avec quelque chose de plus léger : le nommage du répertoire racine et la manipulation des répertoires de site.

Il est souvent important de pouvoir définir le nom du répertoire racine de votre assembly, ou, a minima, de savoir où il se trouve. Deux options de configurations sont disponibles dans le descripteur d'assembly pour vous permettre de gérer le répertoire racine de vos archives : `includeBaseDirectory` et `baseDirectory`. Dans le cas de fichier jar exécutable, vous n'avez pas besoin de répertoire racine. Pour cela, il vous suffit d'ajouter la balise `includeBaseDirectory` et de la mettre à `false` (sa valeur par défaut est `true`). Ainsi, vous obtiendrez une archive, qui une fois dépaquetée, peut contenir plusieurs répertoires. Ce type d'archive, avec plusieurs répertoires à la racine, est considéré comme une mauvaise pratique pour les archives que vous devez dépaqueter avant de pouvoir les utiliser. Cependant, dans le cas d'une archive à utiliser telle quelle, comme un jar exécutable, cela est acceptable.

Dans les autres cas, vous voudrez garantir le nom du répertoire racine de votre archive quelque soient les informations contenues dans votre POM (comme la `version` par exemple). Par défaut, la balise `baseDirectory` a pour valeur `${project.artifactId}-${project.version}`. Il est aisé de changer cette valeur en combinant chaînes de caractères et expressions interpolées à partir du POM, comme par exemple : `${project.groupId}-${project.artifactId}`. Cette fonctionnalité peut s'avérer très pratique pour les équipes de documentation (tout le monde en a, n'est-ce pas ?).

Continuons avec la présentation d'un autre flag de configuration : `includeSiteDirectory`. Par défaut, celui-ci a pour valeur `false`. Si le build de votre projet construit le répertoire racine pour un site en utilisant le cycle de vie ou les goals du plugin Maven Site, le site ainsi produit peut être inclus à l'assembly en positionnant ce flag à `true`. Cependant, cette fonctionnalité est un peu limitée car on se contente d'inclure le répertoire `outputDirectory` de la section `reporting` du POM courant (par défaut, `target/site`) sans prendre en considération les répertoires des éventuels autres modules. Utilisez cette option si vous le désirez, mais vous pouvez obtenir le même résultat en utilisant un `fileSet` ou un `moduleSet`. Il s'agit ici encore d'un autre exemple de configuration legacy supportée par le plugin

Assembly pour assurer la rétrocompatibilité. Vos besoins peuvent évoluer, si vous désirez inclure un site provenant de plusieurs modules, privilégiez l'utilisation des `fileSet` ou des `moduleSet` plutôt que d'activer le flag `includeSiteDirectory`.

14.5.8. `componentDescriptors` et `containerDescriptorHandlers`

Terminons l'exploration du descripteur d'assembly avec la présentation de deux dernières balises : `containerDescriptorHandlers` et `componentDescriptors`. La balise `containerDescriptorHandlers` prend en compte des composants vous permettant d'étendre les fonctionnalités du plugin Assembly. Précisément, ces composants personnalisés vous permettent de définir et de gérer certains types de fichiers qui peuvent être le produit de la fusion de différents éléments utilisés lors de la création de votre assembly. Par exemple, nous pouvons utiliser ce mécanisme pour construire un fichier `web.xml` unique à partir de plusieurs fragments pour l'intégrer à l'assembly.

La balise `componentDescriptors` permet de référencer des descripteurs d'assembly externes et de les inclure dans le descripteur courant. Les références vers ses composants peuvent être sous l'une des formes suivantes :

1. Chemins relatifs : `src/main/assembly/component.xml`
2. Références d'artefact : `groupId:artifactId:version[:type[:classifier]]`
3. Ressources du classpath : `/assemblies/component.xml`
4. URLs : `http://www.sonatype.com/component.xml`

Lors de la résolution d'un descripteur de composant, le plugin Assembly essaye ces différentes stratégies dans cet ordre précis. La première stratégie qui fonctionne est alors utilisée.

Les descripteurs de composant peuvent contenir les mêmes sections qu'un descripteur d'Assembly, à l'exception de la balise `moduleSets`. Celle-ci est considérée comme spécifique à chaque projet. La balise `containerDescriptorHandlers`, dont nous avons parlé brièvement, est également incluse dans le descripteur de composant. Les descripteurs de composants ne peuvent pas contenir de format, d'id d'assembly ou toute autre configuration en rapport avec le répertoire racine de l'archive. En effet, ces éléments sont tous considérés comme propres à un descripteur d'assembly. Bien qu'il semble intéressant de permettre le partage de la balise `formats`, cela n'a pas été fait jusqu'à ce jour (version 2.2-beta-2-release du plugin Assembly).

14.6. Best Practices

Le plugin Assembly est assez flexible pour permettre de résoudre la plupart des problèmes de différentes manières. Si vous avez un besoin unique pour votre projet, il ya de bonnes chances que vous puissiez utiliser directement l'une des méthodes documentées dans ce chapitre pour obtenir votre structure d'assembly désirée. Cette section décrit quelques-unes des bonnes pratiques qui, si elles sont respectées, rendront l'utilisation du plugin Assembly plus productive et moins pénible.

14.6.1. Descripteurs d'assembly standards et réutilisables

Jusqu'à présent, nous avons principalement parlé des différentes solutions pour construire certains types d'assembly. Mais que ferez-vous si vous avez des dizaines de projets qui ont tous besoin d'un type particulier d'assembly ? En bref, comment pouvons-nous réutiliser les efforts investis sur plusieurs projets sans avoir à copier-coller notre descripteur d'assembly ?

Le moyen le plus simple est de créer un artefact standardisé et versionné pour externaliser le descripteur d'assembly, et de le déployer. Une fois cela fait, vous pouvez configurer le plugin Assembly dans votre POM pour inclure ce descripteur d'assembly comme une dépendance de plugin. À ce stade, vous pouvez utiliser ce descripteur de l'Assembly par l'intermédiaire de la balise de configuration `descriptorRefs` dans la déclaration du plugin Assembly. Pour illustrer ceci, voici, en exemple, ce descripteur d'assembly :

```
<assembly>
  <id>war-fragment</id>
  <formats>
    <format>zip</format>
  </formats>
  <includeBaseDirectory>>false</includeBaseDirectory>
  <dependencySets>
    <dependencySet>
      <outputDirectory>WEB-INF/lib</outputDirectory>
    </dependencySet>
  </dependencySets>
  <fileSets>
    <fileSet>
      <directory>src/main/webapp</directory>
      <outputDirectory></outputDirectory>
      <excludes>
        <exclude>**/web.xml</exclude>
      </excludes>
    </fileSet>
  </fileSets>
</assembly>
```

Inclus dans votre projet, ce descripteur serait un bon moyen de packager le contenu afin qu'il puisse être décompressé directement dans une application web existante pour fusionner avec celle-ci (pour ajouter une fonctionnalité, par exemple). Toutefois, si votre équipe construit plusieurs de ces projets "web-fragment", vous aurez probablement envie de réutiliser ce descripteur plutôt que de le dupliquer. Pour déployer ce descripteur dans son propre artefact, nous allons le mettre dans le répertoire `src/main/resources/assemblies` d'un projet qui lui sera dédié.

La structure de projet de cet artefact `assembly-descriptor` devrait ressembler à cela :

```
|-- pom.xml
|-- src
    |-- main
        |-- resources
            |-- assemblies
                |-- web-fragment.xml
```

Notez le chemin vers le fichier descripteur `web-fragment`. Par défaut, Maven inclus les fichiers du répertoire `src/main/resources` dans le JAR final. Ainsi, notre descripteur d'assembly sera inclus sans aucune autre configuration de notre part. Notez également le préfixe de chemin `assemblies/`, le plugin Assembly s'attend à ce préfixe de chemin pour tous les descripteurs fournis dans le classpath des plugins. Il est important de mettre notre descripteur à l'emplacement relatif approprié. Ainsi, il sera récupéré par le plugin Assembly lors de son exécution.

Souvenez-vous, ce projet est maintenant en dehors de votre projet `web-fragment`. Le descripteur d'assembly possède son propre artefact avec sa propre version et, peut-être, son propre cycle de release. Une fois que vous avez installé ce nouveau projet via Maven, vous pourrez le référencer dans vos projets `web-fragment`. Pour plus de clarté, le processus devrait ressembler à ceci :

```
$ mvn install
(...)
[INFO] [install:install]
[INFO] Installing (...) /web-fragment-descriptor/target/\
        web-fragment-descriptor-1.0-SNAPSHOT.jar
        to /Users/~/.m2/repository/org/sonatype/mavenbook/assemblies/\
        web-fragment-descriptor/1.0-SNAPSHOT/\
        web-fragment-descriptor-1.0-SNAPSHOT.jar
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 5 seconds
(...)
```

Comme notre projet `web-fragment-descriptor` ne contient pas de code source, le JAR résultant ne contiendra rien d'autre que notre descripteur d'assembly `web-fragment`. Maintenant, utilisons ce nouvel artefact :

```
<project>
  (...)
  <artifactId>my-web-fragment</artifactId>
  (...)
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-assembly-plugin</artifactId>
        <version>2.2-beta-2</version>
        <dependencies>
          <dependency>
            <groupId>org.sonatype.mavenbook.assemblies</groupId>
            <artifactId>web-fragment-descriptor</artifactId>
            <version>1.0-SNAPSHOT</version>
          </dependency>
        </dependencies>
        <executions>
          <execution>
            <id>assemble</id>
            <phase>package</phase>
            <goals>
```

```

        <goal>single</goal>
    </goals>
    <configuration>
        <descriptorRefs>
            <descriptorRef>web-fragment</descriptorRef>
        </descriptorRefs>
    </configuration>
</execution>
</executions>
</plugin>
(...)
</plugins>
</build>
(...)
</project>

```

Deux choses sont particulières sur cette configuration du plugin Assembly :

- Nous devons inclure la déclaration de la dépendance vers notre `web-fragment-descriptor` au niveau de la déclaration des plugins afin d'avoir accès au descripteur d'Assembly par l'intermédiaire du classpath du plugin.
- Comme nous utilisons des références du classpath plutôt qu'un fichier local, nous devons utiliser la balise `descriptorRefs` à la place de la balise `descriptor`. Notez également, que même si le descripteur d'assembly se trouve en fait dans `assemblies/web-fragment.xml` dans le classpath du plugin, nous pouvons le référencer sans utiliser le préfixe `assemblies/`. Ceci est rendu possible car le plugin Assembly suppose que les descripteurs d'assembly se trouvent effectivement à cet emplacement.

Maintenant, vous êtes libre de réutiliser la configuration du POM ci-dessus dans autant de projets que vous voulez, avec l'assurance que la totalité des fragments web de l'assembly soient pris en compte. Si vous avez besoin de faire des ajustements sur le format de votre assembly - peut-être pour d'inclure d'autres ressources, ou pour affiner les dépendances ou le `fileSet` - vous pouvez tout simplement incrémenter la version du projet de votre descripteur d'assembly, et le déployer à nouveau. Les POMs référençant l'artefact `assembly-descriptor` peuvent ainsi adopter cette nouvelle version du descripteur s'ils le souhaitent.

Un dernier point concernant la réutilisation de l'`assembly-descriptor` : vous souhaitez peut-être partager la configuration du plugin. Pour cela, ajoutez la configuration ci-dessus à la section `pluginManagement` de votre POM parent, puis référez la configuration de votre plugin dans votre POM comme ceci :

```

(...)
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
    </plugin>
  </plugins>
</build>
(...)

```


Si vous avez ajouté le reste de la configuration du plugin - comme décrit dans l'exemple précédent - dans la section `pluginManagement` du POM parent de votre projet, tous les projets qui en héritent profiteront du format d'assemblage avancé dans leurs propres builds en ajoutant ces quelques lignes.

14.6.2. Assembly de distribution (agrégation)

Comme mentionné ci-dessus, le plugin Assembly fournit plusieurs façons de créer de nombreux formats d'archives. Les assemblages de distribution sont généralement de très bons exemples, car ils combinent souvent des modules à partir d'un build multimodule, avec leurs dépendances et, éventuellement, d'autres fichiers en plus de ces artefacts. La distribution vise à inclure ces différentes sources en une seule archive que l'utilisateur pourra télécharger, décompresser et exécuter. Toutefois, nous avons également vu un certain nombre d'inconvénients potentiels pouvant être provoqués par l'utilisation de la balise `moduleSets` du descripteur d'assembly - les relations parent-enfant entre les POMs d'un build peuvent, dans certains cas, rendre indisponible les artefacts des modules.

Plus précisément, si les POMs des modules référencent comme parent le POM qui contient la configuration du plugin-assembly, le projet parent devra être construit avant les projets modules lors de l'exécution du build. Or, l'assembly du parent s'attend à trouver les artefacts de ses modules, mais ces projets attendent également la fin de la construction de leur parent. On arrive donc à une situation de blocage empêchant la construction du parent. En d'autres termes, le projet enfant dépend du projet parent qui dépend à son tour du projet enfant.

À titre d'exemple, considérons le descripteur d'assembly ci-dessous. Il est conçu pour être utilisé à partir du projet de plus haut niveau de la hiérarchie multimodule :

```
<assembly>
  <id>distribution</id>
  <formats>
    <format>zip</format>
    <format>tar.gz</format>
    <format>tar.bz2</format>
  </formats>

  <moduleSets>
    <moduleSet>
      <includes>
        <include>*-web</include>
      </includes>
      <binaries>
        <outputDirectory></outputDirectory>
        <unpack>true</unpack>
        <includeDependencies>true</includeDependencies>
        <dependencySets>
          <dependencySet>
            <outputDirectory>/WEB-INF/lib</outputDirectory>
          </dependencySet>
        </dependencySets>
      </binaries>
    </moduleSet>
  </moduleSets>
</assembly>
```

```

<moduleSet>
  <includes>
    <include>*-addons</include>
  </includes>
  <binaries>
    <outputDirectory>/WEB-INF/lib</outputDirectory>
    <includeDependencies>>true</includeDependencies>
    <dependencySets>
      <dependencySet/>
    </dependencySets>
  </binaries>
</moduleSet>
</moduleSets>
</assembly>

```

Pour un projet parent donné - appelé `app-parent` - contenant trois modules : `app-core`, `app-web` et `app-addons`, notez ce qu'il se passe lorsque nous essayons d'exécuter ce build multimodule :

```

$ mvn package
[INFO] Reactor build order:
[INFO]   app-parent <----- PARENT BUILDS FIRST
[INFO]   app-core
[INFO]   app-web
[INFO]   app-addons
[INFO] -----
[INFO] Building app-parent
[INFO]   task-segment: [package]
[INFO] -----
[INFO] [site:attach-descriptor]
[INFO] [assembly:single {execution: distro}]
[INFO] Reading assembly descriptor: src/main/assembly/distro.xml
[INFO] -----
[ERROR] BUILD ERROR
[INFO] -----
[INFO] Failed to create assembly: Artifact:
org.sonatype.mavenbook.assemblies:app-web:jar:1.0-SNAPSHOT (included by module)
does not have an artifact with a file. Please ensure the package phase is
run before the assembly is generated.
...

```

Le projet parent - `app-parent` - est le premier à être construit. C'est parce que chacun des autres projets désigne ce POM comme son parent, forçant ainsi l'ordre de construction. Le module `app-web`, qui est le premier module désigné dans le descripteur d'assembly, n'a pas encore été construit. Par conséquent, il ne dispose d'aucun des artefacts qui lui sont associés, et donc la construction de l'assembly est impossible.

Une solution de contournement consiste à supprimer la balise `executions` de la déclaration du plugin `Assembly`. Celle-ci lie le plugin à la phase `package` du cycle de vie dans le fichier POM parent. Une fois cette suppression effectuée, exécutez ces deux tâches Maven : la première, `package`, permet de construire le projet multimodule ; et la seconde, `assembly:assembly`, pour invoquer directement le plugin `assembly` qui va utiliser les artefacts construits lors de l'exécution précédente pour construire le paquet de distribution. Pour effectuer cela, utilisez la ligne de commande suivante :

```
$ mvn package assembly:assembly
```

Pendant, cette approche présente plusieurs inconvénients. Premièrement, elle rend le processus d'assemblage plus complexe en lui rajoutant une tâche manuelle qui peut accroître les risques d'erreur. En outre, cela pourrait signifier que les artefacts joints - qui sont associés en mémoire lors de l'exécution de la construction du projet - ne seront pas accessibles au second passage sans recourir à des références au système de fichiers.

Au lieu d'utiliser la balise `moduleSet` pour lister les artefacts du projet multimodule, il est souvent préférable d'utiliser une approche moins technique : à l'aide d'un module dédié à la distribution et aux dépendances inter-projet. Avec cette approche, vous créez un nouveau module dans votre build dont le seul but est d'effectuer l'assemblage. Le POM de ce module contient des références aux dépendances des autres modules, et il configure le plugin Assembly pour qu'il soit rattaché à la phase `package` de son cycle de vie. Le descripteur d'assembly lui-même utilise une section `dependencySets` à la place d'un `moduleSets` pour lister les artefacts et déterminer où les inclure dans l'archive résultante. Cette approche évite les inconvénients liés à la relation parent-enfant mentionnés plus haut, et offre l'avantage d'utiliser une section de configuration plus simple dans le descripteur d'assembly.

Pour cela, nous pouvons créer une nouvelle structure de projet qui ressemble fortement à celle utilisée par l'approche module-set présentée précédemment. Avec l'ajout de ce nouveau projet de distribution, vous devriez avoir cinq POMs au total : `app-parent`, `app-core`, `app-web`, `app-addons` et `app-distribution`. Le nouveau POM `app-distribution` devrait ressembler à cela :

```
<project>
  <parent>
    <artifactId>app-parent</artifactId>
    <groupId>org.sonatype.mavenbook.assemblies</groupId>
    <version>1.0-SNAPSHOT</version>
  </parent>
  <modelVersion>4.0.0</modelVersion>
  <artifactId>app-distribution</artifactId>
  <name>app-distribution</name>

  <dependencies>
    <dependency>
      <artifactId>app-web</artifactId>
      <groupId>org.sonatype.mavenbook.assemblies</groupId>
      <version>1.0-SNAPSHOT</version>
      <type>war</type>
    </dependency>
    <dependency>
      <artifactId>app-addons</artifactId>
      <groupId>org.sonatype.mavenbook.assemblies</groupId>
      <version>1.0-SNAPSHOT</version>
    </dependency>
    <!-- Not necessary since it's brought in via app-web.
  <dependency> [2]
    <artifactId>app-core</artifactId>
    <groupId>org.sonatype.mavenbook.assemblies</groupId>
```

```

    <version>1.0-SNAPSHOT</version>
  </dependency>
  -->
</dependencies>
</project>

```

Notez que nous devons inclure les dépendances vers les autres modules dans la structure du projet puisque nous n'avons pas de section `modules` dans ce POM. Notez aussi que nous n'utilisons pas de dépendance explicite vers `app-core`. Puisqu'il s'agit également d'une dépendance du projet `app-web`, nous n'avons pas besoin de la traiter (ou, d'éviter de la traiter) à deux reprises.

Ensuite, lorsque nous déplaçons le descripteur d'assembly `distro.xml` dans le projet `app-distribution`, nous devons également modifier la configuration de la section `dependencySets` :

```

<assembly>
  ...
  <dependencySets>
    <dependencySet>
      <includes>
        <include>*-web</include>
      </includes>
      <useTransitiveDependencies>false</useTransitiveDependencies>
      <outputDirectory></outputDirectory>
      <unpack>true</unpack>
    </dependencySet>
    <dependencySet>
      <excludes>
        <exclude>*-web</exclude>
      </excludes>
      <useProjectArtifact>false</useProjectArtifact>
      <outputDirectory>/WEB-INF/lib</outputDirectory>
    </dependencySet>
  </dependencySets>
  ...
</assembly>

```

Cette fois, si nous lançons la construction à partir du répertoire de plus haut niveau du projet, nous obtiendrons de meilleurs résultats :

```

$ mvn package
(...)
[INFO] -----
[INFO] Reactor Summary:
[INFO] -----
[INFO] module-set-distro-parent .....SUCCESS [3.070s]
[INFO] app-core ..... SUCCESS [2.970s]
[INFO] app-web ..... SUCCESS [1.424s]
[INFO] app-addons ..... SUCCESS [0.543s]
[INFO] app-distribution ..... SUCCESS [2.603s]
[INFO] -----
[INFO] -----
[INFO] BUILD SUCCESSFUL

```

```
[INFO] -----  
[INFO] Total time: 10 seconds  
[INFO] Finished at: Thu May 01 18:00:09 EDT 2008  
[INFO] Final Memory: 16M/29M  
[INFO] -----
```

Comme vous pouvez le voir, l'approche par dependency-set est beaucoup plus stable et - aussi longtemps que la logique interne de tri de Maven ne sera pas aussi perfectionnée que celle du plugin Assembly - offre moins de possibilités de se tromper lors de l'exécution d'un build.

14.7. En résumé

Comme nous l'avons vu dans ce chapitre, le plugin Maven Assembly permet de la création de formats d'archives personnalisées. Bien que dans le détail ces archives peuvent être complexes, ce n'est pas nécessairement toujours le cas - comme nous l'avons vu avec les descripteurs d'assembly prédéfinis. Même si votre but est d'inclure les dépendances et certains fichiers de votre projet dans une structure de répertoires unique et archivée, l'écriture d'un descripteur d'assembly ne doit pas être trop complexe.

Les assemblies sont utiles pour un large éventail de cas d'utilisation. Ils sont le plus couramment utilisées pour distribuer des applications. Même s'il existe de nombreuses manières différentes d'utiliser le plugin Assembly, les deux façons les plus conseillées pour vous éviter des problèmes est d'utiliser des descripteurs d'assembly standardisés et d'éviter l'utilisation des `moduleSets` lors de la création des archives de distribution.

Chapitre 15. Propriétés et filtrage des ressources

15.1. Introduction

En lisant ce ce livre, vous avez été confronté à l'utilisation de propriétés dans des fichiers POM. Ainsi, les dépendances d'un build multimodule peuvent être référencées par les propriétés comme `${project.groupId}` et `${project.version}`, et n'importe quelle partie du POM peut être référencée en préfixant la variable par "project.". Les variables d'environnement et propriétés Système Java peuvent également être référencées, tout comme les valeurs définies dans le fichier `~/.m2/settings.xml`. Nous allons parcourir dans ce chapitre l'ensemble des valeurs possibles et nous verrons comment ces propriétés peuvent être utilisées pour créer des build portables.

Si vous avez déjà utilisé des références de propriétés dans votre POM, vous devriez savoir que cette fonctionnalité est appelée 'filtrage des ressources'. Celle-ci permet de remplacer les références des propriétés dans n'importe quel fichier stocké dans `src/main/resources`. Par défaut, cette fonctionnalité est désactivée pour éviter des filtrages accidentels. Cette fonctionnalité peut être utilisée pour effectuer des builds spécifiques à une certaine plateforme ou pour externaliser les variables de build dans des fichiers de propriétés, des POMs ou des profils. Ce chapitre présente donc cette fonctionnalité de filtrage des ressources.

15.2. Propriétés Maven

Vous pouvez utiliser les propriétés Maven dans des fichiers `pom.xml` ou dans n'importe quel fichier ressource contrôlé par la fonctionnalité de filtrage du plugin Maven Resource. Une propriété est toujours préfixée par `${` et suffixée par `}`. Par exemple, pour référencer la propriété `project.version`, vous devez utiliser la notation suivante :

```
${project.version}
```

Des propriétés implicites sont directement disponibles à partir de n'importe quel projet Maven, en voici la liste :

`project.*`

POM (Project Object Model) Maven. Vous pouvez utiliser le préfixe `project.*` pour référencer des valeurs du POM.

`settings.*`

Configuration Maven. Vous pouvez utiliser le préfixe `settings.*` pour référencer des valeurs du fichier `~/.m2/settings.xml`.

env.*

Les variables d'environnement, comme `PATH` et `M2_HOME`, peuvent être référencées en utilisant le préfixe `env.*`.

Propriétés Système

Les propriétés pouvant être récupérées par la méthode `System.getProperty()` peuvent être utilisées directement comme propriété Maven.

En plus de ces propriétés implicites, l'utilisateur peut définir ses propres propriétés à partir du POM, du fichier de paramétrage `~/ .m2/settings.xml` ou dans un profil Maven. Les paragraphes suivants fournissent ainsi le détail des différentes propriétés disponibles dans un projet Maven.

15.2.1. Propriétés d'un projet Maven

Quand une propriété Maven est utilisée, son nom référence une propriété du POM. Plus précisément, vous référencez une propriété de la classe `org.apache.maven.model.Model` qui est exposée implicitement comme variable du projet. Quand vous référencez une propriété en utilisant l'une de ces variables implicites, vous utilisez la notation avec des 'points' pour référencer la propriété du bean `Model` correspondante. Par exemple, lorsque nous utilisons `${project.version}`, en fait, vous invoquez la méthode `getVersion()` de l'instance du `Model` qui est exposée via la variable `project`.

Le POM est également représenté dans le document `pom.xml` disponible dans tous les projets Maven. Tout ce qui se trouve dans le POM peut donc être utilisé en propriété. Pour une présentation complète de la structure du POM, référez-vous à l'adresse <http://maven.apache.org/ref/2.2.1/maven-model/maven.html>. La liste ci-dessous récapitule quelques-unes des propriétés habituellement utilisées dans un projet Maven.

`project.groupId` et `project.version`

La majorité des projets multimodule partagent les mêmes `groupId` et `version`. Lorsque vous déclarez une interdépendance entre deux modules possédant les mêmes `groupId` et `version`, c'est souvent une bonne idée que d'utiliser des propriétés pour référencer ces valeurs :

```
<dependencies>
  <dependency>
    <groupId>${project.groupId}</groupId>
    <artifactId>sibling-project</artifactId>
    <version>${project.version}</version>
  </dependency>
</dependencies>
```

`project.artifactId`

L'`artifactId` d'un projet est souvent utilisé pour le nom de l'archive résultant du projet. Par exemple, dans un projet avec un packaging du type `WAR`, vous voudrez peut-être produire un fichier `WAR` sans que sa version apparaisse dans son nom. Pour cela, utilisez la propriété `project.artifactId` dans votre POM comme ceci :

```
<build>
```



```
<finalName>${project.artifactId}</finalName>
</build>
```

`project.name` et `project.description`

Les nom et description du projet sont souvent utilisés dans la documentation. Au lieu d'avoir à maintenir ce contenu à plusieurs endroits, vous pouvez utiliser ces propriétés.

`project.build.*`

Vous ne devez jamais utiliser la valeur en dur `target/classes` pour référencer le répertoire destination de Maven. Une bonne pratique consiste à utiliser des propriétés pour référencer ce genre de répertoire.

- `project.build.sourceDirectory`
- `project.build.scriptSourceDirectory`
- `project.build.testSourceDirectory`
- `project.build.outputDirectory`
- `project.build.testOutputDirectory`
- `project.build.directory`

Les propriétés `sourceDirectory`, `scriptSourceDirectory` et `testSourceDirectory` fournissent un accès aux répertoires source du projet. Les propriétés `outputDirectory` et `testOutputDirectory` fournissent un accès aux répertoires utilisés par Maven pour produire le bytecode et le résultat de son build. La propriété `directory` désigne le répertoire qui contient tous ces répertoires destination.

Autres types de propriété

Des centaines de propriétés sont utilisables à partir POM. La présentation complète de la structure du POM est disponible à l'adresse <http://maven.apache.org/ref/2.2.1/maven-model/maven.html>.

Pour consulter la liste complète des propriétés disponibles dans le `Model` Maven, tournez-vous vers la Javadoc du projet `maven-model` à cette adresse <http://maven.apache.org/ref/2.2.1/maven-model/apidocs/index.html>. Une fois la Javadoc chargée, ouvrez la classe `Model`. À partir de la Javadoc de cette classe, vous devriez pouvoir naviguer sur les différentes propriétés du POM que vous pouvez utiliser. Par exemple, si vous désirez référencer le répertoire destination du build, vous pouvez donc utiliser cette Javadoc pour trouver quelle propriété référencer : `model.getBuild().getOutputDirectory()`. Cette méthode est accessible sous la forme d'une propriété par l'intermédiaire de la notation suivante : `${project.build.outputDirectory}`.

Pour plus d'informations à propos du module Maven Model, le module qui définit la structure du POM, consultez la page du projet Maven Model à l'adresse <http://maven.apache.org/ref/2.2.1/maven-model>.

15.2.2. Propriétés des Settings Maven

Vous pouvez également référencer n'importe quelle propriété définie dans le fichier des préférences Maven locales. Ces préférences locales se trouvent d'ordinaire dans le fichier `~/.m2/settings.xml`. En général, ce fichier contient des configuration spécifiques aux utilisateurs comme par exemple : l'emplacement du dépôt local, des différents serveurs, et des différents miroirs, ainsi que les différents profils.

La liste complète des propriétés des paramètres utilisateurs est disponible à cette adresse <http://maven.apache.org/ref/2.2.1/maven-settings/settings.html>.

15.2.3. Propriétés des variables d'environnement

Les variables d'environnement peuvent être référencées par le préfixe `env.*`. Voici quelques-unes des variables d'environnement les plus utilisées :

`env.PATH`

Contient la valeur courante du `PATH` dans l'environnement dans lequel Maven est lancé. Ce `PATH` contient la liste des répertoires utilisés pour rechercher les scripts et exécutables.

`env.HOME`

(Sur les systèmes `*nix`) cette variable définit le répertoire `home` de l'utilisateur. Nous vous conseillons cependant d'utiliser la propriété `${user.home}` qui donne le même résultat.

`env.JAVA_HOME`

Cette propriété contient le chemin vers le répertoire d'installation Java. Elle peut pointer soit vers un JDK (Java Development Kit), soit vers une JRE (Java Runtime Environment). Nous vous conseillons également d'utiliser la propriété `${java.home}` qui donne, là encore, le même résultat.

`env.M2_HOME`

Contient le chemin vers le répertoire d'installation de Maven.

Si vous avez le choix, privilégiez l'utilisation des propriétés système Java. Préférez donc l'utilisation de la propriété `${user.home}` à celle de `${env.HOME}`. Cela vous permet d'augmenter la portabilité de votre build en le faisant adhérer davantage au paradigme WORA (Write-One-Run-Anywhere) mis en avant par Java.

15.2.4. Propriétés système Java

Maven expose toutes les propriétés `java.lang.System`. Tout ce que vous pouvez récupérer à partir de la méthode `System.getProperty()` peut être référencé par une propriété Maven. Le tableau suivant liste ces propriétés :

Tableau 15.1. Java System Properties

Propriété Système	Description
<code>java.version</code>	Version de la JRE
<code>java.vendor</code>	Fabriquant de la JRE
<code>java.vendor.url</code>	URL du fabriquant de la JRE
<code>java.home</code>	Répertoire d'installation de Java
<code>java.vm.specification.version</code>	Version des spécifications de la JVM
<code>java.vm.specification.vendor</code>	Fabriquant des spécifications de la JVM
<code>java.vm.specification.name</code>	Nom des spécifications de la JVM
<code>java.vm.version</code>	Version de l'implémentation de la JVM
<code>java.vm.vendor</code>	Fabriquant de l'implémentation de la JVM
<code>java.vm.name</code>	Nom de l'implémentation de la JVM
<code>java.specification.version</code>	Version des spécifications de la JRE
<code>java.specification.vendor</code>	Fabriquant des spécifications de la JRE
<code>java.specification.name</code>	Nom des spécifications de la JRE
<code>java.class.version</code>	Version du format des classes
<code>java.class.path</code>	Classpath Java
<code>java.ext.dirs</code>	Répertoires des extensions
<code>os.name</code>	Nom du système d'exploitation
<code>os.arch</code>	Architecture du système d'exploitation
<code>os.version</code>	Version du système d'exploitation
<code>file.separator</code>	Caractère de séparation des fichiers et répertoires ("/" sous UNIX, "\" sous Windows)
<code>path.separator</code>	Caractère de séparation des chemins (":" sous UNIX, ";" sous Windows)
<code>line.separator</code>	Séparateur de lignes ("\n" sous UNIX et Windows)
<code>user.name</code>	Nom de l'utilisateur
<code>user.home</code>	Répertoire personnel de l'utilisateur
<code>user.dir</code>	Répertoire de travail de l'utilisateur

15.2.5. Propriétés définies par l'utilisateur

En plus des propriétés implicites fournies par le POM, les paramètres, les variables d'environnement et les propriétés système, vous avez la possibilité de définir vos propres propriétés. Les propriétés

peuvent être définies dans un POM ou dans un profil. Les propriétés définies de la sorte peuvent être référencées de la même manière que toute autre propriété disponible dans Maven. Les propriétés définies par l'utilisateur peuvent être utilisées directement dans un POM ou dans le filtrage des ressources par l'intermédiaire du plugin Maven Resource. Voici un exemple de définition d'une propriété dans un POM Maven.

Exemple 15.1. Définition d'une propriété dans un POM

```
<project>
...
<properties>
  <arbitrary.property.a>This is some text</arbitrary.property.a>
  <hibernate.version>3.3.0.ga</hibernate.version>
</properties>
...
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>${hibernate.version}</version>
  </dependency>
</dependencies>
...
</project>
```

L'exemple précédent définit deux propriétés : `arbitrary.property.a` et `hibernate.version`. La propriété `hibernate.version` est utilisée dans la déclaration d'une dépendance. L'utilisation du caractère `'` en séparateur dans les noms de propriété est pratique courante dans les POMs et profils. Cette notation n'offre rien de spécial, la clé `hibernate.version` est simplement utilisée pour référencer la valeur `"3.3.0.ga"`. L'exemple ci-dessous montre comment définir une propriété à partir d'un profil dans un POM Maven.

Exemple 15.2. Propriété utilisateur définie dans un profil d'un POM

```
<project>
...
<profiles>
  <profile>
    <id>some-profile</id>
    <properties>
      <arbitrary.property>This is some text</arbitrary.property>
    </properties>
  </profile>
</profiles>
...
</project>
```

L'exemple précédent présente comment définir une propriété utilisateur dans un profil. Pour plus d'information à propos des propriétés utilisateurs et des profils, consultez le Chapitre 11, Profils de Build.

15.3. Filtrage des ressources

Vous pouvez utiliser Maven pour remplacer des variables dans les ressources d'un projet par leur véritable valeur. Lorsque cette fonctionnalité de filtrage des ressources est activée, Maven parcourt les ressources à la recherche de références à des propriétés. Lorsque Maven trouve l'une de ces références, il remplace celle-ci par la valeur appropriée selon un mécanisme similaire aux propriétés définies dans un POM. Cette fonctionnalité est très pratique pour paramétrer un build avec différentes valeurs de configuration selon la plateforme cible.

Il est fréquent qu'un fichier `.properties` ou qu'un document XML présent dans le répertoire `src/main/resources` contienne des références vers des ressources externes du type base de données ou emplacement réseau qui doivent être configurées différemment en fonction de l'environnement cible. Par exemple, un système qui fait appel à une base de données aura un fichier de configuration qui contiendra l'URL JDBC, le nom d'utilisateur et le mot de passe permettant de s'y connecter. Si vous avez besoin d'utiliser une base de données différente en développement, vous pouvez soit utiliser une technologie du type JNDI pour externaliser votre configuration, soit utiliser un mécanisme de build qui saura remplacer les différentes variables par les valeurs correspondant à la plateforme cible.

En utilisant le filtrage des ressources, vous pouvez référencer des propriétés Maven et les spécifier dans les profils pour définir différentes valeurs de configuration selon les environnements cibles. Pour illustrer cela, prenons l'exemple d'un projet qui utilise Spring pour configurer une `BasicDataSource` à partir du projet Commons DBCP¹. Ce projet contient un fichier nommé `applicationContext.xml` dans le répertoire `src/main/resources` qui possède le contenu XML de l'Exemple 15.3, « Référencer des propriétés Maven à partir d'une ressource ».

Exemple 15.3. Référencer des propriétés Maven à partir d'une ressource

```
<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="someDao" class="com.example.SomeDao">
    <property name="dataSource" ref="dataSource"/>
  </bean>

  <bean id="dataSource" destroy-method="close"
        class="org.apache.commons.dbcp.BasicDataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}"/>
    <property name="url" value="${jdbc.url}"/>
    <property name="username" value="${jdbc.username}"/>
    <property name="password" value="${jdbc.password}"/>
  </bean>
</beans>
```

¹ <http://commons.apache.org/dbcp>

Votre programme utilisera ce fichier lors de son exécution. Avant cela, votre build remplacera les références des différentes propriétés (`jdbc.url`, `jdbc.username`, ...) par les valeurs définies dans votre fichier `pom.xml`. Le filtrage des ressources est désactivé par défaut pour vous éviter de mauvaises surprises. Aussi, pour activer cette fonctionnalité, vous devez modifier le contenu de la balise `resources` dans votre POM. Dans l'Exemple 15.4, « Définition de variables et activation du filtrage des ressources », le POM définit les variables utilisées dans l'Exemple 15.3, « Référencer des propriétés Maven à partir d'une ressource » et active le filtrage des ressources pour tout le répertoire `src/main/resources`.

Exemple 15.4. Définition de variables et activation du filtrage des ressources

```
<project>
...
<properties>
  <jdbc.driverClassName>
    com.mysql.jdbc.Driver</jdbc.driverClassName>
  <jdbc.url>jdbc:mysql://localhost:3306/development_db</jdbc.url>
  <jdbc.username>dev_user</jdbc.username>
  <jdbc.password>s3cr3tw0rd</jdbc.password>
</properties>
...
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
</build>
...
<profiles>
  <profile>
    <id>production</id>
    <properties>
      <jdbc.driverClassName>oracle.jdbc.driver.OracleDriver</jdbc.driverClassName>
      <jdbc.url>jdbc:oracle:thin:@proddb01:1521:PROD</jdbc.url>
      <jdbc.username>prod_user</jdbc.username>
      <jdbc.password>s00p3rs3cr3t</jdbc.password>
    </properties>
  </profile>
</profiles>
</project>
```

Les quatre variables sont définies dans la balise `properties` et le filtrage des ressources est activé pour les ressources du répertoire `src/main/resources`. Pour activer la fonctionnalité de filtrage, vous devez affecter la valeur `true` à la balise `filtering` pour chaque répertoire que vous désirez filtrer. La construction d'un projet contenant les ressources de l'Exemple 15.3, « Référencer des propriétés Maven à partir d'une ressource » et le POM de l'Exemple 15.4, « Définition de variables et activation du filtrage des ressources » va créer dans le répertoire `target/classes` le fichier suivant, vous remarquerez d'ailleurs que toutes les propriétés ont bien été remplacées :

```

$ mvn install
...
$ cat target/classes/applicationContext.xml
...
<bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName" value="com.mysql.jdbc.Driver"/>
  <property name="url" value="jdbc:mysql://localhost:3306/development_db"/>
  <property name="username" value="dev_user"/>
  <property name="password" value="s3cr3tw0rd"/>
</bean>
...

```

Le POM de l'Exemple 15.4, « Définition de variables et activation du filtrage des ressources » définit également le profil `production`. Celui-ci surcharge les propriétés par défaut avec des valeurs appropriées à l'environnement de production. Dans ce POM, la configuration par défaut des connexions à la base de données cible une base de données MySQL locale installée sur la machine du développeur. Lorsque ce projet est construit avec le profil 'production' activé, Maven mettra à jour cette configuration pour se connecter à une base de données Oracle en utilisant un autre driver, une autre URL, et un autre couple utilisateur / mot de passe. Si vous construisez un projet avec la ressource de l'Exemple 15.3, « Référencer des propriétés Maven à partir d'une ressource », le POM de l'Exemple 15.4, « Définition de variables et activation du filtrage des ressources » et le profil 'production' activé lorsque vous listerez le contenu du répertoire `target/classes` vous constaterez qu'il contient un fichier correctement filtré avec les valeurs de production :

```

$ mvn -Pproduction install
...
$ cat target/classes/applicationContext.xml
...
<bean id="dataSource" destroy-method="close"
      class="org.apache.commons.dbcp.BasicDataSource">
  <property name="driverClassName"
            value="oracle.jdbc.driver.OracleDriver"/>
  <property name="url" value="jdbc:oracle:thin:@proddb01:1521:PROD"/>
  <property name="username" value="prod_user"/>
  <property name="password" value="s00p3rs3cr3t"/>
</bean>
...

```


Chapitre 16. Génération du Site

16.1. Introduction

Les applications les mieux réussies sont rarement l'oeuvre d'un seul homme. Les équipes sont souvent découpées en cellules pouvant aller d'une poignée de personnes dans un bureau à des armées de plusieurs centaines de personnes réparties sur un environnement distribué. La réussite ou l'échec de la plupart des projets Open Source (comme Maven) dépend l'existence ou non d'une documentation bien écrite pour un public très distribué de développeurs et d'utilisateurs. Dans chaque environnement, il est important que les projets disposent d'un moyen simple pour publier et maintenir leur documentation en ligne. Le développement d'applications est avant tout un exercice de collaboration et de communication, la publication d'un site Maven est l'une des manières de s'assurer d'une bonne communication entre votre projet et ses utilisateurs.

Pour un projet Open Source, un site Internet est souvent le point de départ tant pour les communautés de développeurs que pour les utilisateurs. Là où les utilisateurs finaux recherchent sur le site du projet des tutoriaux, des guides utilisateur, la documentation de API et les archives de la mailing-list, les développeurs y recherchent des documents de conception, des rapports de code, un gestionnaire d'anomalies ou les release notes. Les plus grands des projets Open Source peuvent intégrer des wikis, des gestionnaires d'anomalies et des systèmes d'intégration continue qui permettent d'augmenter la couverture de la documentation en ligne avec des éléments qui reflètent l'état présent du développement. Si le site d'un nouveau projet Open Source se révèle incapable de fournir les renseignements de base à ses utilisateurs potentiels, c'est souvent le signe avant-coureur d'un futur échec, ce projet aura probablement du mal à être adopté. En d'autres termes, pour un projet Open Source, le site et la documentation sont aussi essentiels pour la formation de la communauté que le code lui-même.

Maven peut être utilisé pour créer des sites web de projet qui contiennent des informations aussi bien pour les utilisateurs finaux que les développeurs. Maven peut générer une multitude de rapports, des résultats des tests unitaires à un rapport sur la qualité du code en passant par des rapports sur les dépendances inter-package. Maven vous donne la possibilité d'écrire de simples pages web et d'afficher ces pages au travers d'un modèle élaboré pour le projet. Maven peut publier le contenu d'un site sous plusieurs formats : XHTML et PDF. Maven peut être utilisé pour générer la documentation de l'API de votre projet mais aussi pour embarquer la Javadoc et le code source dans l'archive binaire de votre projet. Une fois que vous avez produit toute la documentation de votre projet, Maven vous permet également de publier votre site sur un serveur distant.

16.2. Construire le site d'un projet avec Maven

Pour illustrer le processus de construction du site web d'un projet, créons un projet en utilisant le plugin Archetype :

```
$ mvn archetype:create -DgroupId=org.sonatype.mavenbook -DartifactId=sample-project
```

Cette commande crée un projet Maven minimaliste qui contient une classe Java dans le répertoire `src/main/java` et un simple POM. Ensuite, vous pouvez construire le site en exécutant la commande `mvn site`. Pour construire le site et afficher le résultat dans un navigateur web, vous pouvez utiliser la commande `mvn site:run`. Celle-ci construira le site et démarrera une instance de Jetty.

```
$ cd sample-project
$ mvn site:run
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'site'.
[INFO] -----
[INFO] Building sample-project
[INFO]    task-segment: [site:run] (aggregator-style)
[INFO] -----
[INFO] Setting property: classpath.resource.loader.class =>
    'org.codehaus.plexus.velocity.ContextClassLoaderResourceLoader'.
[INFO] Setting property: velocimacro.messages.on => 'false'.
[INFO] Setting property: resource.loader => 'classpath'.
[INFO] Setting property: resource.manager.logwhenfound => 'false'.
[INFO] [site:run]
2008-04-26 11:52:26.981::INFO: Logging to STDERR via org.mortbay.log.StdErrLog
[INFO] Starting Jetty on http://localhost:8080/
2008-04-26 11:52:26.046::INFO: jetty-6.1.5
2008-04-26 11:52:26.156::INFO: NO JSP Support for /, did not find
    org.apache.jasper.servlet.JspServlet
2008-04-26 11:52:26.244::INFO: Started SelectChannelConnector@0.0.0.0:8080
```

Une fois Jetty démarré sur le port 8080, le site du projet est disponible à l'adresse `http://localhost:8080/` à partir de votre navigateur web. Vous devriez y découvrir un résultat similaire à la Figure 16.1, « Site généré par Maven ».



Figure 16.1. Site généré par Maven

Si vous cliquez sur ce site web, vous constaterez qu'il n'est pas très pratique comme véritable site de projet. Il ne contient aucune information (et n'est pas très joli). Tant que le projet `sample-project` n'est

pas configuré pour afficher la liste des développeurs, le gestionnaire d'anomalies ou le code source, toutes les pages du projet ne contiendront aucune information intéressante. Même la page d'index du site est vide de contenu, elle affiche un message indiquant qu'aucune description n'a été associée à ce projet. Pour personnaliser ce site, vous devez commencer par ajouter du contenu à votre projet et dans votre POM.

Si vous décidez d'utiliser le plugin Maven Site pour construire le site de votre projet, vous voudrez donc le personnaliser. Vous voudrez renseigner les champs du POM qui permettent de lister les personnes participant au projet. Vous voudrez personnaliser également le menu gauche du site et les liens affichés en haut de la page. Pour personnaliser le contenu de ce site, dont le menu de navigation, vous devez éditer le descripteur de site.

16.3. Personnaliser le descripteur de site

Quand vous ajoutez du contenu à votre site, vous allez vouloir modifier le menu gauche de navigation qui est généré avec votre site. Le descripteur suivant personnalise le logo affiché dans le coin en haut à gauche de votre site. En plus de personnaliser l'entête de votre site, ce même descripteur ajoute une entrée dans le menu de navigation : un simple lien vers une page de présentation générale.

Exemple 16.1. Descripteur de site initial

```
<project name="Sample Project">
  <bannerLeft>
    <name>Sonatype</name>
    <src>images/logo.png</src>
    <href>http://www.sonatype.com</href>
  </bannerLeft>
  <body>
    <menu name="Sample Project">
      <item name="Overview" href="index.html"/>
    </menu>
    <menu ref="reports"/>
  </body>
</project>
```

Ce descripteur référence une image, `logo.png`. Celle-ci doit être placée dans le répertoire `${basedir}/src/site/resources/images`. En plus des changements du descripteur de site, vous voudrez créer une page `index.apx` à placer dans le répertoire `${basedir}/src/site/apx`. Éditez ce fichier pour lui donner le contenu suivant. Ce fichier sera transformé en `index.html` et servira de page d'accueil à votre site.

```
Welcome to the Sample Project, we hope you enjoy your time
on this project site. We've tried to assemble some
great user documentation and developer information, and
we're really excited that you've taken the time to visit
this site.
```

```
What is Sample Project
```

```
Well, it's easy enough to explain. This sample project is
a sample of a project with a Maven-generated site from
Maven: The Definitive Guide. A dedicated team of volunteers
help maintain this sample site, and so on and so forth.
```

Pour visualiser votre site, exécutez les commandes `mvn clean site` et `mvn site:run` :

```
$ mvn clean site
$ mvn site:run
```

Une fois cela fait, ouvrez un navigateur et rendez-vous à l'adresse `http://localhost:8080`. La page affichée devrait ressembler à la capture d'écran de la Figure 16.2, « Site web personnalisé du projet ».

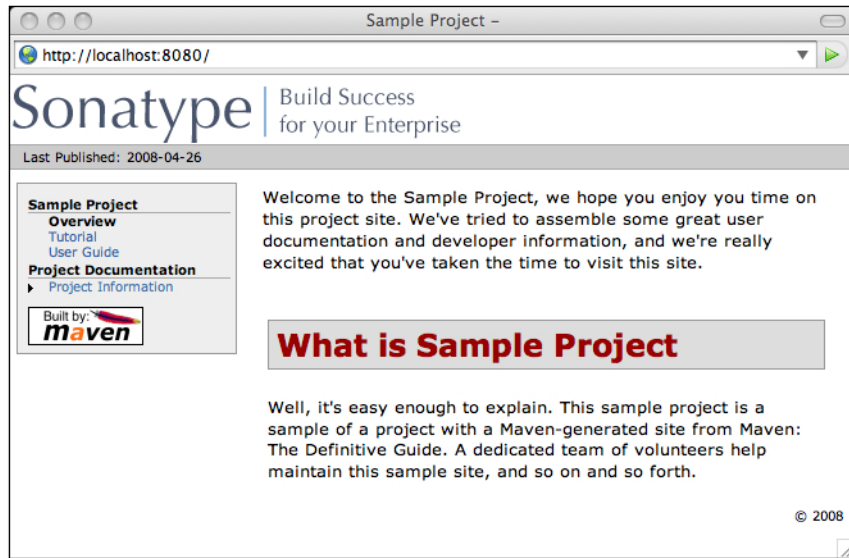


Figure 16.2. Site web personnalisé du projet

16.3.1. Personnaliser les images des en-têtes du site

Pour personnaliser les éléments graphiques qui apparaissent dans les coins en haut à gauche et en haut à droite de la page, vous pouvez utiliser les balises `bannerLeft` et `bannerRight` du descripteur de site.

Exemple 16.2. Descripteur avec ajout d'images en haut à gauche et à droite du site

```
<project name="Sample Project">
  <bannerLeft>
    <name>Left Banner</name>
    <src>images/banner-left.png</src>
    <href>http://www.google.com</href>
  </bannerLeft>
```

```

<bannerRight>
  <name>Right Banner</name>
  <src>images/banner-right.png</src>
  <href>http://www.yahoo.com</href>
</bannerRight>
...
</project>

```

Les deux balises `bannerLeft` et `bannerRight` sont configurées avec : un nom (`name`), un emplacement d'image (`src`) et un lien (`href`). Dans le descripteur de site ci-dessus, le plugin Maven Site utilisera les images `banner-left.png` et `banner-right.png` pour le coins en haut à gauche et en haut à droite du site. Maven recherchera ces images dans le répertoire `${basedir}/src/site/resources/images`.

16.3.2. Personnaliser le menu navigation

Pour personnaliser le contenu du menu navigation, utilisez la balise `menu` avec une sous-balise `item`. La balise `menu` ajoute une section au menu gauche de navigation. Chaque `item` représente un lien dans ce menu.

Exemple 16.3. Descripteur de site avec ajout d'entrées dans le menu

```

<project name="Sample Project">
  ...
  <body>

    <menu name="Sample Project">
      <item name="Introduction" href="index.html"/>
      <item name="News" href="news.html"/>
      <item name="Features" href="features.html"/>
      <item name="Installation" href="installation.html"/>
      <item name="Configuration" href="configuration.html"/>
      <item name="FAQ" href="faq.html"/>
    </menu>
    ...
  </body>
</project>

```

Les balises `item` peuvent également être imbriquées. En imbriquant vos items, vous créez un sous-menu dans le menu navigation sous la forme d'un arbre extensible ou rétractable. L'exemple suivant ajoute un lien "Developer Resources" qui pointe vers la page `/developer/index.html`. Quand un utilisateur regardera cette page, les menus au dessus de l'élément sélectionné seront développés.

Exemple 16.4. Ajout d'un lien au menu du site

```

<project name="Sample Project">
  ...
  <body>
    ...

```

```

<menu name="Sample Project">
  ...
  <item name="Developer Resources" href="/developer/index.html"
        collapse="true">
    <item name="System Architecture" href="/developer/architecture.html"/>
    <item name="Embedder's Guide" href="/developer/embedding.html"/>
  </item>
</menu>
...
</body>
</project>

```

Lorsque l'attribut `collapse` d'un `item` a pour valeur `true`, Maven pliera celui-ci jusqu'à ce qu'un utilisateur consulte cette page particulière. Dans l'exemple précédent, lorsque l'utilisateur ne regarde pas la page "Developer Resources", Maven n'affiche pas les liens "System Architecture" et "Embedder's Guide". À la place, il affichera une flèche pointant sur le lien "Developer Resources". Quand l'utilisateur regardera cette page, ces liens s'afficheront avec une flèche pointée vers le bas.

16.4. Structure de répertoire d'un site

Maven place tous les documents du site dans le répertoire `src/site`. Les documents avec des formats similaires sont regroupés dans des sous-répertoires de celui-ci. Ainsi, les fichiers APT se trouvent dans le répertoire `src/site/apt`, les fichiers FML dans `src/site/fml` et les documents XDoc dans `src/site/xdoc`. Le descripteur de site se trouve à l'emplacement `src/site/site.xml` et toutes les références sont stockées dans le répertoire `src/site/resources`. Lorsque le plugin Maven Site construit un site web, il copie tout le contenu de ce répertoire à la racine du site. Si vous stockez une image dans `src/site/resources/images/test.png`, vous pouvez l'utiliser dans votre site en utilisant le chemin relatif `images/test.png`.

L'exemple suivant récapitule l'emplacement dans un projet de tous les fichiers APT, FML, HTML, XHTML, et XDoc. Notez que le contenu XHTML est simplement stocké dans le répertoire dédié aux ressources. Le fichier `architecture.html` ne sera donc pas traité par Doxia, mais simplement copié dans le répertoire destination. Vous pouvez utiliser ce principe pour inclure du contenu HTML brut sur lequel vous ne voulez pas appliquer de modèle ou que vous ne voulez pas formater avec Doxia ou le plugin Maven Site.

```

sample-project
+- src/
  +- site/
    +- apt/
      | +- index.appt
      | +- about.appt
      | |
      | +- developer/
      |   +- embedding.appt
      |
    +- fml/
      | +- faq.fml

```

```
|
+- resources/
| +- images/
| | +- banner-left.png
| | +- banner-right.png
| |
| +- architecture.html
| +- jira-roadmap-export-2007-03-26.html
|
+- xdoc/
| +- xml-example.xml
|
+- site.xml
```

Notez que la documentation pour les développeurs se trouve dans le fichier `src/site/apt/developer/embedding.apt`. Celui-ci se trouve dans un sous-répertoire d'`apt`, il en sera de même pour sa page HTML correspondante. Quand le plugin Site effectue le rendu du répertoire `src/site/apt`, il génère les fichiers HTML dans des répertoires relatifs à la racine du site. Si un fichier se trouve dans le répertoire `apt`, son fichier généré se trouvera à la racine du site web. Si un fichier se trouve dans le répertoire `apt/developer`, son fichier généré se trouvera dans le répertoire `developer/` du site web.

16.5. Écrire la documentation d'un projet

Maven utilise en moteur de traitement de documentation appelé Doxia. Doxia peut transformer différents formats de fichiers avec un même modèle de document, manipuler ces modèles et effectuer le rendu dans différents formats de sorties dont PDF ou XHTML. Pour écrire un document pour votre projet, vous devez écrire votre contenu dans un format reconnu par Doxia. À ce jour, Doxia supporte la plupart des formats "plein texte" (autrement dit "Almost Plain Text", ou APT), XDoc (format de la documentation de Maven 1.x), XHTML et FML (utilisé pour écrire des FAQ).

Ce chapitre présente succinctement le format APT. Pour une plus de précisions sur ce format, ou pour une introduction à XDoc ou à FML, consultez les ressources suivantes :

- Référence APT : <http://maven.apache.org/doxia/format.html>
- Référence XDoc : <http://jakarta.apache.org/site/jakarta-site2.html>
- Référence FML : <http://maven.apache.org/doxia/references/fml-format.html>

16.5.1. Exemple de fichier APT

L'Exemple 16.5, « Document APT » affiche un document APT simple dont le contenu est composé d'un paragraphe d'introduction et d'une liste. Notez que la liste se termine par le pseudo élément "[]".

Exemple 16.5. Document APT

```
---
```

```
Introduction to Sample Project
---
Brian Fox
---
26-Mar-2008
---
```

```
Welcome to Sample Project
```

```
This is a sample project, welcome! We're excited that you've decided to
read the index page of this Sample Project. We hope you enjoy the simple
sample project we've assembled for you.
```

```
Here are some useful links to get you started:
```

- * {{{news.html}News}}
 - * {{{features.html}Features}}
 - * {{{faq.html}FAQ}}
- ```
[]
```

Si le document APT de l'Exemple 16.5, « Document APT » est placé dans `src/site/apt/index.apt`, le plugin Maven Site en analysera le contenu APT avec Doxia et produira un fichier `index.html` contenant le XHTML généré.

## 16.5.2. Exemple de fichier FML

Beaucoup de projets contiennent une FAQ (Foire Aux Questions). L'Exemple 16.6, « Document FML (FAQ Markup Language) » présente un exemple de document FML.

Exemple 16.6. Document FML (FAQ Markup Language)

```
<?xml version="1.0" encoding="UTF-8"?>
<faqs title="Frequently Asked Questions">
 <part id="General">
 <faq id="sample-project-sucks">
 <question>Sample project doesn't work. Why does sample
project suck?</question>
 <answer>
 <p>
 We resent that question. Sample wasn't designed to work, it was
 designed to show you how to use Maven. If you really think
 this project sucks, then keep it to yourself. We're not
 interested in your pestering questions.
 </p>
 </answer>
 </faq>
 <faq id="sample-project-source">
 <question>I want to put some code in Sample Project,
how do I do this?</question>
```



```

<answer>
 <p>
 If you want to add code to this project, just start putting
 Java source in src/main/java. If you want to put some source
 code in this FAQ, use the source element:
 </p>
 <source>
 for(int i = 0; i < 1234; i++) {
 // do something brilliant
 }
 </source>
</answer>
</faq>
</part>
</faqs>

```

## 16.6. Déployez le site de votre projet

Une fois que la documentation de votre projet est écrite et que vous êtes fier du site que vous avez créé, il vous faut le déployer sur un serveur. Pour déployer un site, vous pouvez utiliser le plugin Maven Site. Celui-ci propose différentes méthodes pour déployer votre site sur un serveur distant, dont FTP, SCP et DAV. Par exemple, pour déployer votre site via DAV, configurez la balise `distributionManagement` dans votre POM comme ceci :

Exemple 16.7. Configurer le déploiement d'un site

```

<project>
 ...
 <distributionManagement>
 <site>
 <id>sample-project.website</id>
 <url>dav:https://dav.sample.com/sites/sample-project</url>
 </site>
 </distributionManagement>
 ...
</project>

```

L'url contenue dans une balise incluse de la section `distributionManagement` est préfixée par un indicateur `dav`. Celui-ci indique au plugin Maven Site qu'il va lui falloir déployer le site vers une URL avec le protocole WebDAV. Une fois la configuration effectuée dans le POM de votre projet `sample-project`, vous pouvez lancer le déploiement via la commande suivante :

```
$ mvn clean site-deploy
```

Si vous disposez d'un serveur configuré pour comprendre le protocole WebDAV, Maven déploiera le site de votre projet sur ce serveur distant. Si vous déployez ce site sur un serveur accessible au public, vous devrez probablement ajouter de la configuration d'authentification pour l'accès sécurisé. Si votre serveur serveur vous demande un identifiant et un mot de passe (ou tout autre moyen d'authentification), vous pouvez configurer ces valeurs dans votre fichier `~/.m2/settings.xml`.

## 16.6.1. Configurer l'authentification de votre serveur

Pour configurer votre identifiant et votre mot de passe pour le déploiement du site dans le fichier `$HOME/.m2/settings.xml`, inspirez vous de l'Exemple 16.8, « Authentification serveur dans les préférences utilisateur » :

Exemple 16.8. Authentification serveur dans les préférences utilisateur

```
<settings>
...
<servers>
 <server>
 <id>sample-project.website</id>
 <username>jdcasey</username>
 <password>b@dp@ssw0rd</password>
 </server>
 ...
</servers>
...
</settings>
```

La section authentification peut contenir différents types d'éléments. Par exemple, si vous utilisez un déploiement par SCP, vous voudrez peut-être utiliser une authentification par clé publique. Pour cela, utilisez les balises spécifiques `publicKey` et `passphrase` plutôt que `password`. Il se peut que vous deviez configurer l'élément `username` selon la configuration de votre serveur.

## 16.6.2. Configurer les permissions des fichiers et dossiers

Si vous travaillez dans un groupe de développeurs conséquent, vous voudrez vous assurer que les dossiers de votre site Internet ont les bonnes permissions utilisateur et groupe après leur déploiement. Pour configurer ces permissions lors du déploiement, inspirez-vous du fichier `$HOME/.m2/settings.xml` suivant :

Exemple 16.9. Configurer les permissions des fichiers et répertoires sur un serveur distant

```
<settings>
...
<servers>
 ...
 <server>
 <id>hello-world.website</id>
 ...
 <directoryPermissions>0775</directoryPermissions>
 <filePermissions>0664</filePermissions>
 </server>
</servers>
...
</settings>
```

Les préférences ci-dessus rendent les répertoires lisibles et modifiables par leur propriétaire et les membres de son groupe. Les utilisateurs anonymes, quant à eux, pourront uniquement lire et lister le

contenu du répertoire. De la même manière, le propriétaire et les membres de son groupe disposeront d'un accès en lecture et écriture sur n'importe quel fichier, alors que les utilisateurs anonymes disposeront d'un accès en lecture seule.

## 16.7. Personnaliser l'apparence de votre site

Le modèle Maven par défaut laisse beaucoup à désirer. Si vous voulez personnaliser le site de votre projet au delà de l'ajout de contenu, d'options dans le menu navigation et de modification des logos, Maven vous offre plusieurs mécanismes qui le permettent. Il est ainsi possible de modifier plus en profondeur la structure et l'aspect du site web. Pour effectuer de petites modifications, l'ajout d'un fichier `site.css` est souvent suffisant. Cependant, si vous voulez que ces modifications soient réutilisables, ou si vous désirez modifier le XHTML généré, la création d'une 'skin' Maven est souvent nécessaire.

### 16.7.1. Personnaliser la CSS du site

Le moyen le plus simple pour personnaliser l'apparence de votre site est de fournir à votre projet un fichier `site.css`. Tout comme pour les images ou pour le contenu XHTML, cette feuille de style doit se trouver dans le répertoire `src/site/resources/css`. Avec ce fichier CSS, il vous est possible de modifier la disposition et le style des textes de votre site. Il vous est également possible d'ajouter une image de fond ou de personnaliser les images de vos listes. Par exemple, si vous décidez de faire ressortir davantage le titre du menu, vous pourriez utiliser le style suivant dans le fichier `src/site/resources/css/site.css` :

```
#navcolumn h5 {
 font-size: smaller;
 border: 1px solid #aaaaaa;
 background-color: #bbb;
 margin-top: 7px;
 margin-bottom: 2px;
 padding-top: 2px;
 padding-left: 2px;
 color: #000;
}
```

Après une nouvelle génération de votre site, le titre de votre menu sera ainsi tramé par un fond gris et séparé du reste du menu par un peu plus d'espace. En utilisant ce fichier, n'importe quel élément de la structure d'un site Maven peut être décoré par l'intermédiaire de styles personnalisés. Lorsque vous modifiez le fichier `site.css` d'un projet Maven, les modifications impactent uniquement son site. Si vous désirez effectuer des modifications qui s'appliquent sur plusieurs projets, vous pouvez créer une skin personnalisée.



#### Astuce

Il n'existe pas de bonne documentation sur la structure du modèle par défaut d'un site Maven. Si vous essayez de personnaliser le style de votre projet Maven, le mieux est

d'utiliser une extension Firefox comme Firebug pour vous aider à naviguer dans le DOM de vos pages web.

## 16.7.2. Créer un modèle de site personnalisé

Si la structure du site Maven par défaut ne vous convient pas, vous pouvez toujours personnaliser son modèle. Grâce à cela, il est possible de contrôler complètement le résultat du plugin Maven Site. Il est également possible de personnaliser le modèle du site de telle sorte que vous ne reconnaissiez pas la structure du modèle par défaut.

Le plugin Maven Site utilise le moteur de rendu Doxia, qui utilise à son tour en interne un modèle Velocity pour effectuer le rendu de chaque page. Pour changer la structure d'une page, il est possible de configurer le plugin Site à partir du POM pour utiliser un modèle personnalisé. Comme le modèle du site est relativement complexe, il est préférable de disposer d'un bon point de départ pour effectuer votre personnalisation. Commencez par copier le modèle Velocity par défaut de Doxia, qui est disponible sur le dépôt Subversion de Doxia, `default-site.vm`<sup>1</sup> dans le fichier `src/site/site.vm`. Ce fichier s'appuie donc une syntaxe Velocity. Velocity est un langage simple pour écrire et appliquer des modèles. Il supporte la définition de macros et permet l'accès aux méthodes et propriétés des objets en utilisant un formalisme simple. Une présentation plus détaillée de la syntaxe de ce framework n'est pas l'objet de ce livre, aussi, pour plus d'informations sur Velocity, référez-vous au site officiel du projet à l'adresse suivante <http://velocity.apache.org>.

Si le modèle `default-site.xml` est assez complexe, les modifications nécessaires à la personnalisation du menu gauche sont relativement simples. Si vous voulez modifier l'apparence d'un `menuItem`, rechercher la macro Velocity du même nom. Elle se trouve dans une section qui ressemble à cela :

```
#macro (menuItem $item)
 ...
#end
```

Si vous remplacez la définition de la macro par celle présentée ci-dessous, vous injecterez du javascript dans chaque élément du menu pour fermer et ouvrir l'arbre sans avoir à recharger entièrement la page :

```
#macro (menuItem $item $listCount)
 #set ($collapse = "none")
 #set ($currentItemHref = $PathTool.calculateLink($item.href,
 $relativePath))
 #set ($currentItemHref = $currentItemHref.replaceAll("\\\"", "/"))

 #if ($item && $item.items && $item.items.size() > 0)
 #if ($item.collapse == false)
 #set ($collapse = "collapsed")
 #endif
 #endif
#end
```

<sup>1</sup> <http://svn.apache.org/viewvc/maven/doxia/doxia-sitetools/trunk/doxia-site-renderer/src/main/resources/org/apache/maven/doxia/siterenderer/resources/default-site.vm?revision=595592>

```

#else
 ## By default collapsed
 #set ($collapse = "collapsed")
#end

#set ($display = false)
#displayTree($display $item)

#if ($alignedFileName == $currentItemHref || $display)
 #set ($collapse = "expanded")
#end
#end
<li class="$collapse">
#if ($item.img)
 #if (! ($item.img.toLowerCase().startsWith("http") ||
 $item.img.toLowerCase().startsWith("https")))
 #set ($src = $PathTool.calculateLink($item.img, $relativePath))
 #set ($src = $item.img.replaceAll("\\\"", "/"))

 #else

 #end
#end
#if ($alignedFileName == $currentItemHref)
 $item.name
#else
 #if ($item && $item.items && $item.items.size() > 0)
 <a onclick="expand('list$listCount')"
 style="cursor:pointer">$item.name
 #else
 $item.name
 #end
#end
#if ($item && $item.items && $item.items.size() > 0)
#if ($collapse == "expanded")
<ul id="list$listCount" style="display:block">
#else
<ul id="list$listCount" style="display:none">
#end
 #foreach($subitem in $item.items)
 #set ($listCounter = $listCounter + 1)
 #menuItem($subitem $listCounter)
 #end

#end

#end

```

Cette modification rajoute un nouveau paramètre à la macro `menuItem`. Pour que cette modification fonctionne, vous devez donc mettre à jour toutes les références à cette macro; dans le cas contraire, le XHTML résultant sera incorrect ou incomplet. Pour effectuer ces mises à jour, apportez la même modification à la macro `mainMenu`. Trouvez cette macro en recherchant un bloc semblable à celui-là :

```
#macro (mainMenu $menus)
```

```
...
#end
```

Remplacez maintenant la macro `mainMenu` avec cette implémentation :

```
#macro (mainMenu $menus)
 #set ($counter = 0)
 #set ($listCounter = 0)
 #foreach($menu in $menus)
 #if ($menu.name)
 <h5 onclick="expand('menu$counter') ">$menu.name</h5>
 #end
 <ul id="menu$counter" style="display:block">
 #foreach($item in $menu.items)
 #menuItem($item $listCounter)
 #set ($listCounter = $listCounter + 1)
 #end

 #set ($counter = $counter + 1)
 #end
#end
```

Cette nouvelle macro `mainMenu` est entièrement compatible avec la version précédente, elle fournit un support javascript aux éléments racines du menu. Un clic sur un de ces éléments racines permet aux utilisateurs d'afficher l'arbre complet dans devoir attendre le chargement de la page.

La mise à jour de la macro `menuItem` rajoute un appel à une fonction javascript `expand()`. Cette fonction doit donc être ajoutée au template XHTML principal, par exemple en bas du modèle. Recherchez la section avec un contenu ressemblant à ceci :

```
<head>
...
<meta http-equiv="Content-Type"
 content="text/html; charset=${outputEncoding}" />
...
</head>
```

et remplacez-la avec ce contenu :

```
<head>
...
<meta http-equiv="Content-Type"
 content="text/html; charset=${outputEncoding}" />
<script type="text/javascript">
 function expand(item) {
 var expandIt = document.getElementById(item);
 if(expandIt.style.display == "block") {
 expandIt.style.display = "none";
 expandIt.parentNode.className = "collapsed";
 } else {
 expandIt.style.display = "block";
 expandIt.parentNode.className = "expanded";
 }
 }
</script>
```

```

 }
 }
</script>
#if ($decoration.body.head)
 #foreach($item in $decoration.body.head.getChildren())
 #if ($item.name == "script")
 $item.toUnescapedString()
 #else
 $item.toString()
 #end
 #end
#end
#end
</head>

```

Après avoir modifié le modèle du site, vous devez mettre à jour le POM de votre projet pour qu'il utilise votre nouveau modèle. Pour personnaliser le modèle du site, vous devez utiliser la balise `templateDirectory` dans les propriétés de configuration du plugin Maven Site.

Exemple 16.10. Personnaliser le modèle de page dans le POM du projet

```

<project>
...
<build>
 <plugins>
 <plugin>
 <artifactId>maven-site-plugin</artifactId>
 <configuration>
 <templateDirectory>src/site</templateDirectory>
 <template>site.vm</template>
 </configuration>
 </plugin>
 </plugins>
</build>
...
</project>

```

Vous devriez pouvoir maintenant régénérer le site de votre projet. En effectuant cela, vous pourrez remarquer que les ressources et CSS du site Maven manquent. Lorsqu'un projet Maven personnalise le template du site, le plugin Maven Site s'attend que le projet fournisse toutes les images et CSS. Pour obtenir les ressources initiales pour le site de votre projet, le plus simple reste de les copier à partir du dépôt Subversion du projet de modèle par défaut de Doxia. Cela peut se faire en exécutant les commandes suivantes :

```

$ svn co \
 http://svn.apache.org/repos/asf/maven/doxia/doxia-sitetools/\
 trunk/doxia-site-renderer
$ rm \
 doxia-site-renderer/src/main/resources/org/apache/maven/\
 doxia/siterenderer/resources/css/maven-theme.css

```

```
$ cp -rf \
 doxia-site-renderer/src/main/resources/org/apache/maven/\
 doxia/siterenderer/resources/* \
 sample-project/src/site/resources
```

Récupérez le projet `doxia-site-renderer` et copiez toutes ses ressources dans le répertoire `src/site/resources` de votre projet.

Lorsque vous régénérez un site, vous pouvez remarquer également que certains éléments du menu ne possèdent pas de style CSS. Cela est dû à une mauvaise intégration entre la CSS du site et votre modèle personnalisé. Pour corriger cela, modifiez votre fichier `site.css` pour changer la couleur de vos liens dans le menu en ajoutant les lignes suivantes :

```
li.collapsed, li.expanded, a:link {
 color:#36a;
}
```

Après avoir régénéré le site, la couleur du lien du menu devrait être réparée. Si vous appliquez le nouveau modèle au projet `sample-project` de ce chapitre, vous noterez que le menu s'affiche maintenant sous la forme d'un arbre. Un clic sur "Developer Resources" ne vous emmène plus sur la page du même nom ; à la place, son sous-menu est affiché. Du coup, suite à la modification de l'élément "Developer Resources" en un sous-menu dynamique, vous ne pouvez plus accéder à la page `developer/index.ap`. Le plus simple reste donc d'ajouter au sous-menu un élément "Overview" qui pointe sur cette même page :

Exemple 16.11. Ajouter un élément du menu dans le descripteur de site

```
<project name="Hello World">
 ...
 <menu name="Main Menu">
 ...
 <item name="Developer Resources" collapse="true">
 <item name="Overview" href="/developer/index.html"/>
 <item name="System Architecture" href="/developer/architecture.html"/>
 <item name="Embedder's Guide" href="/developer/embedding.html"/>
 </item>
 </menu>
 ...
</project>
```

### 16.7.3. Réutilisation des skins

Si votre entreprise crée de nombreux sites Maven, vous voudrez probablement réutiliser vos modèles et vos feuilles CSS personnalisés sur d'autres projets. Pour cela, Maven vous propose un mécanisme vous permettant de créer des skins. Les skins du plugin Maven Site permettent de packager les ressources et les templates pour les réunir sur différents projets. Elles vous permettent ainsi d'éviter de dupliquer ces ressources à chaque utilisation.



Plus rapide que de définir votre propre skin, vous pouvez également utiliser l'une des skins alternatives fournies par Maven. Chaque skin propose son propre layout pour la navigation, le contenu, les logos et les templates :

- Maven Classic Skin - `org.apache.maven.skins:maven-classic-skin:1.0`
- Maven Default Skin - `org.apache.maven.skins:maven-default-skin:1.0`
- Maven Stylus Skin - `org.apache.maven.skins:maven-stylus-skin:1.0.1`

Pour obtenir la liste complète des skins disponibles, rendez-vous à l'adresse suivante : <http://repo1.maven.org/maven2/org/apache/maven/skins/>.

La création d'une skin consiste simplement à construire un projet Maven qui contienne votre personnalisation de la feuille de style `maven-theme.css`. Ainsi, elle peut être identifiée par le triplet `groupId`, `artifactId` et `version`. Elle peut également contenir des ressources (comme des images) et remplacer le modèle du site par défaut (modèle décrit en utilisant la syntaxe Velocity). Cela permet de générer des structures de page XHTML complètement différentes de celle proposée par défaut. Dans la plupart des cas, la personnalisation de la CSS peut s'avérer suffisante. Pour illustrer cela, créons une skin pour le projet `sample-project`. La première étape consiste en la personnalisation de la feuille de style `maven-theme.css`.

Avant de commencer à écrire cette CSS, nous devons créer un projet Maven séparé. Celui-ci sera référencé par le descripteur de site du projet `sample-project`. Utilisons donc le plugin Maven Archetype pour créer un projet vide. Pour cela, exécutez la commande suivante à partir du répertoire parent du projet `sample-project` :

```
$ mvn archetype:create -DartifactId=sample-site-skin
-DgroupId=org.sonatype.mavenbook
```

Cette commande crée un projet (et un répertoire) appelé `sample-site-skin`. Rendez vous dans celui-ci et supprimez tout le code source et les tests qui y ont été générés. Créez-y ensuite un répertoire pour y mettre les ressources (de la skin) :

```
$ cd sample-site-skin
$ rm -rf src/main/java src/test
$ mkdir src/main/resources
```

#### 16.7.4. Création d'un thème CSS personnalisé

Ensuite, écrivons une CSS personnalisée pour notre skin. Celle-ci doit se trouver à l'emplacement suivant : `src/main/resources/css/maven-theme.css`. Contrairement au fichier `site.css` qui se retrouvera au cœur des sources du projet, le fichier `maven-theme.css` sera empaqueté dans un JAR installé dans votre dépôt local. Pour que ce fichier soit inclus dans le JAR de la skin, il doit se trouver dans le répertoire principal des ressources du projet : `src/main/resources`.

Comme pour la personnalisation du modèle, commencer en personnalisant la CSS existante est une bonne idée. Copiez donc cette CSS de la skin par défaut dans le fichier `maven-theme.css` de votre projet. Pour récupérer une copie de ce fichier, enregistrez le contenu du fichier `maven-theme.css`<sup>2</sup> dans le répertoire `src/main/resources/css/` de votre projet skin.

Une fois ce fichier récupéré, personnalisez-le en utilisant la CSS de votre ancien fichier `site.css`. Remplacez le bloc `#navcolumn h5` par le code ci-dessous :

```
#navcolumn h5 {
 font-size: smaller;
 border: 1px solid #aaaaaa;
 background-color: #bbb;
 margin-top: 7px;
 margin-bottom: 2px;
 padding-top: 2px;
 padding-left: 2px;
 color: #000;
}
```

Ceci fait, construisez l'artefact de votre projet `sample-site-skin` et installez le JAR produit dans votre dépôt local. Pour cela, exécutez la commande suivante :

```
$ mvn clean install
```

Revenez ensuite dans le répertoire du projet `sample-project`. Si vous avez déjà personnalisé le fichier `site.css` en début de ce chapitre, renommez-le en `site.css.bak` pour qu'il ne soit pas utilisé par le plugin Maven Site :

```
$ mv src/site/resources/css/site.css src/site/resources/css/site.css.bak
```

Pour utiliser votre skin `sample-site-skin` dans le site du projet `sample-project`, ajoutez la référence à cet artefact (`sample-site-skin`) dans le descripteur de site de votre projet (`sample-project`). Pour référencer une skin dans votre site, utilisez la balise du même nom :

Exemple 16.12. Configurer une skin personnalisée dans le descripteur de site

```
<project name="Sample Project">
 ...
 <skin>
 <groupId>org.sonatype.mavenbook</groupId>
 <artifactId>sample-site-skin</artifactId>
 </skin>
 ...
</project>
```

Vous pouvez considérer la skin de votre site comme une dépendance. Les skins ont un `groupId` et un `artifactId` comme n'importe quel artefact. Utiliser une skin pour votre site vous permet de consolider

<sup>2</sup> <http://svn.apache.org/viewvc/maven/skins/trunk/maven-default-skin/src/main/resources/css/maven-theme.css?view=co>

les personnalisations d'un projet pour réutiliser les CSS et modèles aussi facilement que pour n'importe quel autre artefact.

## 16.8. Trucs et Astuces

Dans cette section, nous allons passer en revue quelques trucs et astuces à utiliser pour la création de votre site Maven.

### 16.8.1. Injecter du XHTML dans le HEAD

Pour injecter du XHTML dans la balise `HEAD`, ajoutez une balise `head` à la balise `body` dans votre descripteur de site. L'exemple suivant ajoute un lien vers un flux RSS sur toutes les pages du site du projet `sample-project`.

Exemple 16.13. Injecter du XHTML dans la balise HEAD

```
<project name="Hello World">
 ...
 <body>
 <head>
 <link href="http://sample.com/sites/sample-project/feeds/blog"
 type="application/atom+xml"
 id="auto-discovery"
 rel="alternate"
 title="Sample Project Blog" />
 </head>
 ...
 </body>
</project>
```

### 16.8.2. Ajouter des liens sous le logo de votre site

Si vous travaillez sur un projet développé par une entreprise, vous voudrez probablement ajouter des liens en dessous du logo de votre projet. Supposons que votre projet appartienne à l'Apache Software Foundation. Par exemple, vous voudrez ajouter un lien vers le site web officiel juste en dessous de votre logo, mais également un lien vers le site de projet parent du votre. Pour ajouter des liens en dessous du logo d'un site, ajoutez une balise `links` à la balise `body` du descripteur de site. Ensuite, à chaque balise `item` correspond un lien dans la barre juste en dessous du logo du projet. Dans l'exemple suivant, nous désirons créer un lien vers l'Apache Software Foundation et un lien vers le projet Apache Maven.

Exemple 16.14. Ajouter des liens sous le logo de votre site

```
<project name="Hello World">
 ...
 <body>
 ...
 <links>
 <item name="Apache" href="http://www.apache.org"/>
 </links>
 </body>
</project>
```

```
<item name="Maven" href="http://maven.apache.org"/>
</links>
...
</body>
</project>
```

### 16.8.3. Ajouter un chemin de navigation à votre site

Si vos pages sont organisées hiérarchiquement, vous voudrez probablement mettre en place un chemin de navigation que vos utilisateurs sachent dans quel contexte ils se trouvent et qu'ils puissent naviguer facilement dans les pages et les catégories. Pour configurer un chemin de navigation, ajoutez la balise `breadcrumbs` à la balise `body` dans votre descripteur de site. Ainsi, chaque balise `item` de ce chemin de fer affichera un lien en respectant l'ordre dans lequel ces balises ont été mises. Les éléments du chemin de navigation doivent donc être ordonnés en fonction du niveau dans la hiérarchie de pages, du niveau le plus haut à celui le plus bas. Dans l'exemple suivant, dans le descripteur de site, l'élément Codehaus contiendra un élément Mojo.

Exemple 16.15. Configurer le chemin de navigation de votre site

```
<project name="Sample Project">
...
<body>
...
<breadcrumbs>
 <item name="Codehaus" href="http://www.codehaus.org"/>
 <item name="Mojo" href="http://mojo.codehaus.org"/>
</breadcrumbs>
...
</body>
</project>
```

### 16.8.4. Ajouter la version de votre projet

Lorsque vous documentez un projet qui propose plusieurs versions, c'est souvent important de mettre la version du projet sur chacune des pages du site. Pour afficher la version de votre projet sur le site web, ajoutez simplement la balise `version` à votre descripteur de site :

Exemple 16.16. Afficher la version de votre projet

```
<project name="Sample Project">
...
<version position="left"/>
...
</project>
```

Ce morceau de code XML permet d'afficher la version (dans le cas du projet `sample-project`, on affichera "Version: 1.0-SNAPSHOT") dans le coin en haut à gauche de votre site, juste à côté de la dernière date de publication. Il est possible d'afficher la version de votre projet aux positions suivantes :

- left  
Dans la barre en haut à gauche, juste en dessous du logo
- right  
Dans la barre en haut à droite, juste en dessous du logo
- navigation-top  
En haut du menu
- navigation-bottom  
En bas du menu
- none  
Masque complètement la version

### 16.8.5. Modifier le format et l'emplacement de la date de publication

Vous voudrez peut-être modifier le format ou repositionner l'affichage de la date de publication "Last Published" sur le site de votre projet. Comme nous venons de le faire pour la version dans l'astuce précédente, vous pouvez choisir l'emplacement de la date de publication en utilisant l'une de ces positions :

- left  
Dans la barre en haut à gauche, juste en dessous du logo
- right  
Dans la barre en haut à droite, juste en dessous du logo
- navigation-top  
En haut du menu
- navigation-bottom  
En bas du menu
- none  
Masque complètement la date de publication

Exemple 16.17. Positionner la date de publication

```
<project name="Sample Project">
 ...
 <publishDate position="navigation-bottom"/>
 ...
</project>
```

Par défaut, la date de publication sera affichée selon le format `MM/dd/yyyy`. Il est possible de modifier celui-ci en utilisant la notation standard présentée dans la Javadoc de la classe

`java.text.SimpleDateFormat` (pour plus d'informations, consultez la Javadoc de la classe `SimpleDateFormat`<sup>3</sup>) Par exemple, pour modifier le format de la date pour qu'il utilise le masque `yyyy-MM-dd`, utilisez la balise `publishDate` suivante.

Exemple 16.18. Configurer le format de la date de publication

```
<project name="Sample Project">
 ...
 <publishDate position="navigation-bottom" format="yyyy-MM-dd"/>
 ...
</project>
```

## 16.8.6. Utiliser des macros Doxia

En plus de ces fonctionnalités avancées de rendu, Doxia fournit également un moteur de macros. Celui-ci permet de déclencher l'injection de contenu dynamique pour chaque entrée. Prenons un exemple pour illustrer l'utilisation de ce moteur, la macro `snippet` qui permet à un document de récupérer un extrait de code depuis un fichier disponible via HTTP. En utilisant cette macro, un petit morceau d'APT peut être transformé en XHTML. Le morceau de code APT suivant appelle la macro `snippet`. Notez que ce code doit être sur une seule ligne, un antislash peut tout de même être inséré pour faire des retours à la ligne lors de son affichage.

```
%{snippet|id=modello-model|url=http://svn.apache.org/repos/asf/maven/\
archetype/trunk/maven-archetype/maven-archetype-model/src/main/\
mdo/archetype.mdo}
```

Exemple 16.19. Résultat de la macro `snippet` en XHTML

```
<div class="source"><pre>

<model>
 <id>archetype</id>
 <name>Archetype</name>
 <description><![CDATA[Maven's model for the archetype descriptor.
]]></description>
 <defaults>
 <default>
 <key>package</key>
 <value>org.apache.maven.archetype.model</value>
 </default>
 </defaults>
 <classes>
 <class rootElement="true" xml.tagName="archetype">
 <name>ArchetypeModel</name>
 <description>Describes the assembly layout and packaging.</description>
 <version>1.0.0</version>
 <fields>
 <field>
```

<sup>3</sup> <http://java.sun.com/j2se/1.5.0/docs/api/java/text/SimpleDateFormat.html>

```
<name>id</name>
<version>1.0.0</version>
<required>>true</required>
<type>String</type>
</field>
...
</fields>
</class>
</classes>
</model>

</pre></div>
```



## Avertissement

Les macros Doxia NE DOIVENT PAS être indentées dans les documents APT. Si vous le faites, le parseur APT ignorera ces macros complètement.

Pour plus d'information sur l'utilisation de la macro `snippet`, référez-vous au guide de cette macro sur le Site Maven à l'adresse suivante <http://maven.apache.org/guides/mini/guide-snippet-macro.html>.





# Chapitre 17. Création de Plugins

## 17.1. Introduction

Ce chapitre aborde des fonctionnalités avancées de Maven, ce n'est pas pour autant que l'écriture d'un plugin Maven doit vous intimider. Malgré la complexité de cet outil, les concepts fondamentaux sont faciles à appréhender, l'écriture de plugins reste relativement simple. Après la lecture de ce chapitre, vous aurez une bonne idée de ce qu'implique la création d'un plugin Maven.

## 17.2. Programmation Maven

La plus grande partie de cet ouvrage a été dédié à l'utilisation de Maven, de ce fait, vous n'avez pas vu beaucoup d'exemples de code portant sur la personnalisation de Maven. En fait, vous n'en avez pas vu du tout. Maven a été pensé pour que 99% de ses utilisateurs n'aient jamais besoin d'écrire le moindre plugin pour personnaliser Maven. En effet, il existe de nombre de plugins et ceux-ci peuvent être configurés pour répondre à la grande majorité des cas, vous n'aurez donc probablement pas besoin d'écrire de plugins. Cela dit, si votre projet comporte des demandes très spécifiques qui vous obligent à personnaliser le comportement de Maven, vous pouvez alors vous lancer dans la création d'un plugin. Modifier le code du cœur de Maven est considéré comme hors de leur portée par la plupart des développeurs, comme s'il s'agissait de modifier la pile TCP/IP d'un système d'exploitation.

Avant de commencer à écrire votre plugin, vous devez en apprendre un peu plus sur les entrailles de Maven : Comment gère t-il les composants logiciels ? Qu'est-ce qu'un plugin ? Comment puis-je personnaliser un cycle de vie ? Cette section répond à certaines de ces questions et introduit les concepts clés du cœur de Maven. Apprendre à écrire un plugin Maven est le moyen le plus simple de personnaliser Maven. Si vous vous demandiez par où démarrer pour comprendre le code de Maven, vous avez trouvé le bon point de départ.

### 17.2.1. Qu'est ce que l'inversion de contrôle ?

Le cœur de Maven est basé sur un conteneur d'inversion de contrôle (IoC) nommé Plexus. Que fait-il ? Il s'agit d'un système de gestion des relations entre composants. Bien qu'il existe un livre dédié sur l'IoC écrit par Martin Fowler, le concept et le terme ont été si souvent utilisés à bon et à mauvais escient ces dernières années qu'il devient difficile de trouver une bonne définition de cette notion sans que s'agisse d'une auto-référence. Au lieu de recourir à une citation Wikipédia, nous allons résumer l'inversion de contrôle et l'injection de dépendances par l'intermédiaire d'une analogie.

Supposons que vous devez connecter une série de composants hi-fi ensemble. Imaginez donc plusieurs composants stéréo branchés à une Playstation 3 et un TiVo qui doivent s'interfacer à la fois avec un boîtier Apple TV et à une télévision LCD 50 pouces. Vous venez de ramener tout ce matériel d'un magasin d'électronique à la maison et vous avez acheté les câbles pour connecter tout cela. Vous déballez donc l'ensemble de ces éléments, vous les installez chacun à leur place, puis vous commencez

---

à brancher les milliers de câbles coaxiaux et de prises stéréo aux milliers d'entrées numériques et analogiques. Éloignez-vous de votre Home cinéma et allumez la télé, vous venez de réaliser de l'injection de dépendances, et vous étiez vous-même un conteneur d'inversion de contrôle.

Quel est le rapport avec tout ça ? Transposons cette analogie en Java. Votre Playstation 3 comme votre Java Bean fournissent tous les deux une interface. La Playstation 3 à deux entrées : l'alimentation et la prise réseau, et une sortie vers la TV. Votre Java Bean possède trois propriétés : `power`, `network` et `tvOutput`. Quand vous ouvrez la boîte de votre Playstation 3, le manuel ne vous décrit pas en détail comment la connecter à tous les types de téléviseurs. De même, votre bean Java fournit une liste de propriétés sans pour autant vous donner une recette explicite pour créer et gérer un système complet de composants. Dans un conteneur d'IoC tel que Plexus, il vous incombe de déclarer les relations dans un ensemble de composants qui fournit une interface d'entrée et de sortie. Vous n'instanciez pas d'objets, Plexus s'en charge. Le code de votre application n'est pas responsable de la gestion des états des composants, c'est le rôle de Plexus. Lorsque vous démarrez Maven, Plexus est démarré pour gérer un système d'éléments connexes, comme votre système stéréo.

Quels sont les avantages apportés par l'utilisation d'un conteneur IoC ? Quel est l'avantage d'acheter des composants stéréo distincts ? Si l'un des composants se casse, vous pouvez le remplacer sans avoir à dépenser les 15 000 € nécessaires pour remplacer l'ensemble de votre système. Si votre téléviseur vous lâche, vous pouvez le remplacer sans que cela n'affecte le lecteur CD. Et le plus important pour vous, les composants de votre chaîne hi-fi coûtent moins chers et sont plus fiables car les fabricants peuvent se contenter de construire des composants en s'appuyant un ensemble d'entrées et de sorties connues. Les conteneurs d'inversion de contrôle et d'injection de dépendances encouragent donc la catégorisation et l'émergence de standards. L'industrie du logiciel aime s'imaginer comme source de toutes les nouvelles idées, mais l'injection de dépendances et l'inversion de contrôle ne sont réellement que de nouveaux mots pour définir des concepts bien connus : la standardisation et l'interchangabilité des appareils. Si vous voulez en savoir plus sur l'IoC et l'injection de dépendances, vous pouvez vous renseigner sur la Ford T, le Cotton Gin ou l'émergence d'un standard pour le chemin de fer au 19ème siècle.

## 17.2.2. Introduction à Plexus

La fonctionnalité la plus importante d'un conteneur d'IoC Java est son mécanisme d'injection de dépendances. Le principe de base d'un conteneur IoC est de contrôler la création et la gestion des objets. Ce ne sont plus les objets qui instancient leurs dépendances mais une tierce personne : le conteneur IoC. En utilisant l'injection de dépendances dans une application utilisant la programmation par interface, vous pouvez créer des composants qui n'ont aucune liaison particulière avec les différentes implémentations de vos services. Bien que votre code utilise des interfaces, vous pouvez décrire les dépendances entre les classes et les composants dans un fichier XML qui définit les composants, les classes d'implémentation et les relations entre vos composants. En d'autres termes, vous pouvez écrire des composants isolés et les lier entre eux en utilisant un fichier XML. Dans Plexus, les différents composants d'un système sont définis par un document XML qui se trouve dans `META-INF/plexus/components.xml`.

Dans un conteneur Java IoC, il existe plusieurs moyens d'injecter des dépendances dans un objet : injection par constructeur, accesseur ou propriété. Bien que Plexus soit capable d'utiliser ces trois techniques d'injection de dépendances, Maven en utilise seulement deux : l'injection par propriété et par accesseur.

#### Injection par constructeur

L'injection par constructeur permet de fournir des objets nécessaires lors de la création de l'objet. Par exemple, si vous avez un objet du type `Person` qui possède un constructeur `Person (String name, Job job)`, vous pouvez lui passer ainsi deux valeurs, donc deux dépendances.

#### Injection par accesseur

Ce type d'injection utilise les accesseurs des propriétés des Java Beans pour remplir ses dépendances. Par exemple, si vous avez un objet du type `Person` qui possède deux propriétés `name` et `job`, un conteneur IoC utilisant l'injection par accesseur créera une instance de la classe `Person` en utilisant le constructeur par défaut et appellera ensuite les méthodes `setName()` et `setJob()`.

#### Injection par propriété

Les deux types d'injection précédents se basent sur l'appel de méthodes et constructeurs publics. En utilisant l'injection par propriété, un conteneur IoC peut remplir les dépendances d'un composant en utilisant directement ses propriétés. Par exemple, si vous avez un objet du type `Person` qui contient deux propriétés `name` et `job`, votre conteneur IoC utilisera ces champs directement pour remplir les dépendances (i.e. `person.name = "Thomas"; person.job = job;`)

### 17.2.3. Pourquoi Plexus ?

Spring est le conteneur IoC le plus populaire du moment. Il a de bons arguments à faire valoir : il a affecté "l'écosystème Java" en forçant les entreprises comme Sun Microsystems à donner plus contrôle à la communauté open source et en permettant d'ouvrir les standards par son "bus" orienté composant sur lequel on vient se brancher. Mais Spring n'est pas le seul conteneur IoC open source, il en existe d'autres (comme `PicoContainer`<sup>1</sup>).

Il y a des années, quand Maven a été créé, Spring n'était pas si mature. L'équipe initiale des développeurs Maven connaissait bien le conteneur Plexus, qu'elle avait d'ailleurs inventé, et donc c'est ce dernier qu'elle choisit. S'il n'est pas aussi populaire que Spring, il en est pas pour autant moins efficace. Comme Plexus a été créé par la même personne que celle qui est à l'origine de Maven, il répond parfaitement à ses besoins. Après la lecture de ce chapitre, vous en saurez plus sur le fonctionnement de Plexus. Si vous avez déjà utilisé un conteneur IoC, vous pourrez noter les similarités et les différences de ce conteneur.

---

<sup>1</sup> <http://www.picocontainer.org/>



## Note

Si Maven est basé sur Plexus, cela ne veut pas dire que la communauté Maven est "anti-Spring" (nous avons d'ailleurs consacré un chapitre complet à Spring dans cet ouvrage). La question "Pourquoi vous n'avez pas utilisé Spring ?" revient assez souvent pour que nous effectuons cet aparté. Nous le savons, Spring est la star, et c'est mérité. Cependant, nous avons une tâche dans notre liste qui consiste à introduire (et documenter) Plexus auprès des développeurs : dans l'industrie logicielle c'est toujours bien d'avoir le choix.

### 17.2.4. Qu'est ce qu'un Plugin ?

Un plugin Maven est un artefact Maven qui contient un descripteur de plugin et un ou plusieurs Mojos. Un Mojo peut se comparer à un goal Maven, d'ailleurs derrière chaque goal se cache un Mojo. Au goal `compiler:compile` correspond la classe `CompilerMojo` dans le plugin Maven Compiler, le goal `jar:jar` correspond à la classe `JarMojo` du plugin Maven Jar... Lorsque vous écrivez votre propre plugin, vous regroupez un ensemble de Mojos (ou goals) dans un seul artefact.<sup>2</sup>



## Note

Mojo ? Qu'est-ce qu'un Mojo ? Le mot *mojo* définit dans le dictionnaire par plusieurs définitions comme un "charme magique ou un sort", une "amulette, souvent dans un petit sac de flanelle contenant un ou plusieurs objets magiques" et du "magnétisme personnel, charme". Maven utilise le terme Mojo comme jeu de mots autour du terme Pojo (Plain-old Java Object), un Pojo Maven est donc appelé Mojo.

Un Mojo est bien plus qu'un goal Maven, c'est un composant géré par Plexus qui peut inclure des références à d'autres composants Plexus.

## 17.3. Descripteur de Plugin

Un plugin Maven contient une 'carte' qui donne à Maven le chemin des différents Mojos et leur configuration. Ce descripteur de plugin se trouve dans le fichier JAR du plugin, à l'emplacement `META-INF/maven/plugin.xml`. Quand Maven charge un plugin, il commence par lire ce fichier XML puis instancie et configure les objets du plugin pour en rendre les Mojos disponibles.

Lorsque vous écrivez des plugins Maven, nous n'avez quasiment jamais besoin de vous soucier du descripteur. Dans le Chapitre 10, Cycle de vie du build, les goals du cycle de vie rattachés au type de packaging `maven-plugin` montrent que le goal `plugin:descriptor` est rattaché à la phase `generate-resources`. Ce goal génère un descripteur de plugin à partir de annotations présentes dans le code source du plugin. Dans la suite du chapitre, nous verrons comment les Mojos sont annotés et comment le contenu de ces annotations se retrouve dans le fichier `META-INF/maven/plugin.xml`.

---

<sup>2</sup>"mojo." The American Heritage® Dictionary of the English Language, Fourth Edition. Houghton Mifflin Company, 2004. Answers.com 02 Mar. 2008. <http://www.answers.com/topic/mojo-1>

L'Exemple 17.1, « Plugin Descriptor » montre le descripteur du plugin Maven Zip. Comme son nom l'indique, ce plugin permet de construire des archives Zips à partir des résultats du build. Normalement, vous ne devriez jamais à avoir écrire votre propre plugin pour construire des archives avec Maven. Utilisez simplement le plugin Maven Assembly, il est capable de produire des archives dans de multiples formats. Regardez le descripteur de site suivant, cela vous donnera une idée de ce genre de fichier.

### Exemple 17.1. Plugin Descriptor

```
<plugin>
 <description></description>
 <groupId>com.training.plugins</groupId>
 <artifactId>maven-zip-plugin</artifactId>
 <version>1-SNAPSHOT</version>
 <goalPrefix>zip</goalPrefix>
 <isolatedRealm>>false</isolatedRealm>
 <inheritedByDefault>>true</inheritedByDefault>
 <mojos>
 <mojo>
 <goal>zip</goal>
 <description>Zips up the output directory.</description>
 <requiresDirectInvocation>>false</requiresDirectInvocation>
 <requiresProject>>true</requiresProject>
 <requiresReports>>false</requiresReports>
 <aggregator>>false</aggregator>
 <requiresOnline>>false</requiresOnline>
 <inheritedByDefault>>true</inheritedByDefault>
 <phase>package</phase>
 <implementation>com.training.plugins.ZipMojo</implementation>
 <language>java</language>
 <instantiationStrategy>per-lookup</instantiationStrategy>
 <executionStrategy>once-per-session</executionStrategy>
 <parameters>
 <parameter>
 <name>baseDirectory</name>
 <type>java.io.File</type>
 <required>>false</required>
 <editable>>true</editable>
 <description>Base directory of the project.</description>
 </parameter>
 <parameter>
 <name>buildDirectory</name>
 <type>java.io.File</type>
 <required>>false</required>
 <editable>>true</editable>
 <description>Directory containing the build files.</description>
 </parameter>
 </parameters>
 <configuration>
 <buildDirectory implementation="java.io.File">
 ${project.build.directory}</buildDirectory>
 <baseDirectory implementation="java.io.File">
 ${basedir}</baseDirectory>
 </configuration>
 </mojo>
 </mojos>
</plugin>
```

```

</configuration>
<requirements>
 <requirement>
 <role>org.codehaus.plexus.archiver.Archiver</role>
 <role-hint>zip</role-hint>
 <field-name>zipArchiver</field-name>
 </requirement>
</requirements>
</mojo>
</mojos>
<dependencies>
 <groupId>org.apache.commons</groupId>
 <artifactId>commons-io</artifactId>
 <version>1.3.2</version>
</dependencies>
</plugin>

```

Un descripteur de site se compose de trois parties : la configuration haut-niveau qui contient les éléments du type `groupId` et `artifactId`, la déclaration des mojos et la déclaration des dépendances. Regardons chacune de ces sections dans le détail.

### 17.3.1. Éléments haut-niveau du descripteur de plugin

Les valeurs de configuration haut-niveau du plugin sont :

#### description

Cet élément contient une courte description du plugin. Dans le plugin Zip, cette description est laissée vide.

#### groupId, artifactId, version

Comme tout artefact Maven, un plugin doit posséder son triplet de coordonnées unique. Le `groupId`, l'`artifactId` et la `version` sont utilisés pour identifier le plugin dans le dépôt Maven.

#### goalPrefix

Cet élément contrôle le préfixe utilisé pour référencer les différents goals d'un plugin. Si vous regardez le descripteur du plugin Compiler, vous verrez que la valeur du `goalPrefix` est `compiler`. De la même manière, si vous regardez le descripteur du plugin Jar, son `goalPrefix` aura pour valeur `jar`. Pour votre nouveau plugin il est important de choisir un préfixe qui n'existe pas.

#### isolatedRealm (deprecated)

C'est une propriété legacy qui n'est plus utilisée par Maven. Elle reste présente dans le système pour assurer la compatibilité avec les plugins les plus anciens. Les anciennes versions du Maven utilisaient un mécanisme de chargement des dépendances des plugins dans un `ClassLoader` isolé. Maven utilise de manière intensive le projet ClassWorlds<sup>3</sup> provenant de la

<sup>3</sup> <http://classworlds.codehaus.org/>

communauté Codehaus<sup>4</sup> pour créer une hiérarchie de `ClassLoader` qui est modélisée par un objet `ClassRealm`. N'hésitez pas à ignorer cette propriété et à laisser sa valeur à `false`.

#### `inheritedByDefault`

Si cette propriété est à `true`, il vous est possible de configurer vos mojos dans un projet enfant en reprenant la configuration définie dans son projet parent. Si vous configurez un mojo pour être exécuté durant une phase spécifique dans un projet parent et que cette propriété est à `true`, l'exécution de celui-ci aura lieu dans le projet fils. À l'inverse, si cette propriété est à `false`, l'exécution ne tiendra pas compte du projet fils.

## 17.3.2. Configuration du Mojo

Vient ensuite la déclaration de chaque Mojo. La balise `plugin` contient une balise `mojos` possédant un élément par mojo présent dans le plugin. Chaque balise `mojo` contient les éléments suivants :

#### `goal`

Il s'agit du nom du goal. Si nous prenons l'exemple du goal `compiler:compile`, `compiler` est le `goalPrefix` et `compile` désigne le nom du goal.

#### `description`

Cette balise contient la description du goal. Celle-ci est utilisée par le plugin Help pour produire la documentation du plugin.

#### `requiresDirectInvocation`

Si vous affectez cette propriété à `true`, le goal pourra seulement s'exécuter s'il est appelé directement en la ligne de commande par un utilisateur. Si quelqu'un essaye de rattacher ce goal à une phase du cycle de vie dans un POM, Maven affichera un message d'erreur. La valeur par défaut pour cette propriété est `false`.

#### `requiresProject`

Spécifie si le goal peut être exécuté en dehors d'un projet. Si cette propriété est affectée à `true` (la valeur par défaut), le goal nécessitera un POM.

#### `requiresReports`

Si vous créez un plugin qui nécessite la présence de rapports, par exemple, si vous écrivez un plugin qui agrège les informations à partir de plusieurs rapports, vous pouvez setter la propriété `requiresReports` à `true`. La valeur par défaut de cette propriété est `false`.

#### `aggregator`

Un descripteur de Mojo dont la valeur de la balise `aggregator` est affectée à `true` est censé n'être lancé qu'une seule fois durant une exécution Maven. Ce flag a été créé pour donner la possibilité aux développeurs de plugins d'agréger la sortie d'une série de builds. Par exemple, on peut utiliser

---

<sup>4</sup> <http://www.codehaus.org>

celui-ci pour créer un plugin qui agrège un rapport sur tous les projets d'un build. Un goal qui possède ce flag `aggregator` positionné à `true` doit être lancé uniquement à partir d'un projet haut-niveau d'un build Maven. La valeur par défaut de cette propriété est `false`. Elle est marquée comme `deprecated` pour les prochaines releases.

#### `requiresOnline`

Spécifie si un goal donné peut s'exécuter en mode hors connexion (option `-o` de la ligne de commande). Si un goal nécessite l'utilisation de ressources réseau et que ce flag est activé, Maven affichera une erreur si le goal est exécuté en mode hors connexion. La valeur par défaut de cette propriété est `false`.

#### `inheritedByDefault`

Si cette propriété est positionnée à `true`, un mojo configuré dans un projet parent reprendra cette configuration pour le projet enfant. Si vous configurez un mojo pour s'exécuter dans une phase spécifique d'un projet parent et que ce flag est activé, cette exécution sera héritée pour le projet enfant. Cette propriété est positionnée par défaut à `false`.

#### `phase`

Si vous ne rattachez pas ce goal à une phase spécifique, cet élément définit la phase par défaut du mojo. Si vous ne précisez pas cet élément, Maven demandera à l'utilisateur de spécifier explicitement la phase dans le POM.

#### `implementation`

Il s'agit de la classe du Mojo à instancier par Maven. Cette propriété est une propriété de composant Plexus (définie dans le `ComponentDescriptor` de Plexus).

#### `language`

Java est le langage par défaut d'un Mojo Maven. Cette propriété contrôle le `ComponentFactory` utilisé par Plexus pour créer et instancier le Mojo. Ce chapitre se concentre sur l'écriture de plugin Java, cependant notez qu'il est possible d'utiliser d'autres langages : Groovy, Beanshell, Ruby... Dans le cas où vous utiliseriez un de ces langages alternatifs, vous auriez à configurer cet élément.

#### `instantiationStrategy`

Cette propriété est une propriété de configuration de composants Plexus, elle permet de contrôler la stratégie utilisée par Plexus pour gérer ses instances de composants. Dans Maven, tous les mojos sont configurés avec une `instantiationStrategy` ayant pour valeur `per-lookup`. Une nouvelle instance du composant (mojo) est créée à chaque fois qu'on demande à Plexus de nous le fournir.

#### `executionStrategy`

Il s'agit du choix de la stratégie utilisée par Maven pour exécuter un Mojo. Les différentes valeurs utilisables dans cet élément sont `once-per-session` et `always`. Note : cette propriété est dorénavant `deprecated` et n'est plus utilisée par Maven, elle sera supprimée dans les prochaines releases.



## parameters

Cette balise décrit chacun des paramètres du Mojo. Quel est son nom ? Quel est son type ? Est-il obligatoire ? Chaque paramètre possède les éléments suivants :

### name

Il s'agit du nom du paramètre (exemple `baseDirectory`)

### type

Il s'agit du type Java du paramètre (exemple `java.io.File`)

### required

Indique si le paramètre est obligatoire. Affecté à `true`, le paramètre doit obligatoirement être passé au goal exécuté avec une valeur non nulle.

### editable

Si un paramètre n'est pas éditable (si cette propriété est positionnée à `false`), alors la valeur de ce paramètre ne pourra pas être modifiée dans un POM. Par exemple, si le descripteur de plugin définit la valeur de la propriété `buildDirectory` à `${basedir}`, aucun POM ne pourra pas surcharger cette valeur.

### description

Courte description utilisée lors de la génération de la documentation du plugin (utilisée par le plugin Help)

## configuration

Cet élément fournit les valeurs par défaut de tous les paramètres du Mojo utilisant la notation pour les expressions Maven. Cet exemple fournit la valeur par défaut des paramètres Mojo `baseDir` et `buildDirectory`. Dans cet élément, l'implémentation spécifie le type (`java.io.File`), la valeur du paramètre peut contenir soit une valeur par défaut en dur soit une référence à une propriété Maven.

## requirements

C'est ici que le descripteur de plugin devient intéressant. Un Mojo est un composant géré par Plexus. Grâce à cela, il est possible de référencer n'importe quel autre composant géré par Plexus. Cette propriété permet de définir des dépendances vers d'autres composants Plexus.

Même s'il est intéressant de savoir comment lire un descripteur de plugin, vous ne devriez pratiquement jamais avoir à en écrire un. Les descripteurs de plugin sont générés automatiquement à partir d'annotations présentes dans le code source du Mojo.

### 17.3.3. Dépendances d'un Plugin

Pour finir, le descripteur de plugin déclare une liste de dépendances comme le ferait un projet Maven. Quand Maven utilise un plugin, toutes ses dépendances sont téléchargées avant d'exécuter un de ses goals. Dans cet exemple, le plugin dépend de Jakarta Commons IO version 1.3.2.

## 17.4. Écrire un plugin personnalisé

Pour écrire un plugin, vous devez créer un ou plusieurs Mojo (goal). Chaque Mojo est représenté par une classe Java qui contient des annotations pour indiquer à Maven comment générer le descripteur de plugin. Avant de commencer à écrire vos classes Mojo, vous devez commencer par créer un projet Maven avec le type de packaging approprié.

### 17.4.1. Création d'un projet Plugin

Pour créer un projet Plugin, vous pouvez utiliser le plugin Maven Archetype. La ligne de commande suivante se chargera de générer un projet yant pour `groupId org.sonatype.mavenbook.plugins`, et `first-maven-plugin` comme `artifactId` :

```
$ mvn archetype:create \
-DgroupId=org.sonatype.mavenbook.plugins \
-DartifactId=first-maven-plugin \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-mojo
```

Le plugin Archetype va créer ainsi le répertoire nommé `my-first-plugin` qui contiendra le POM suivant.

Exemple 17.2. Le POM d'un projet de plugin

```
<?xml version="1.0" encoding="UTF-8"?><project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>org.sonatype.mavenbook.plugins</groupId>
 <artifactId>first-maven-plugin</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>maven-plugin</packaging>
 <name>first-maven-plugin Maven Mojo</name>
 <url>http://maven.apache.org</url>
 <dependencies>
 <dependency>
 <groupId>org.apache.maven</groupId>
 <artifactId>maven-plugin-api</artifactId>
 <version>2.0</version>
 </dependency>
 <dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>3.8.1</version>
 <scope>test</scope>
 </dependency>
 </dependencies>
</project>
```

L'élément le plus important d'un POM de projet de plugin est la balise `packaging`, celle-ci doit contenir la valeur `maven-plugin`. Ce type de packaging personnalise le cycle de vie Maven pour

inclure les goals nécessaires à la création du descripteur de plugin. Le cycle de vie Plugin a été présenté dans la Section 10.2.3, « Plugin Maven », il ressemble fortement au cycle de vie Jar à ces trois exceptions près : le goal `plugin:descriptor` est rattaché à la phase `generate-resources`, le goal `plugin:addPluginArtifactMetadata` est rattaché à la phase `package`, et le goal `plugin:updateRegistry` est rattaché à la phase `install`.

L'autre partie importante d'un POM de projet de plugin est la dépendance sur Maven Plugin API. Ici, le projet dépend de la version 2.0 de `maven-plugin-api` et ainsi que de JUnit pour les tests unitaires.

## 17.4.2. Un simple Mojo Java

Dans ce chapitre, nous allons vous présenter comment écrire des Mojos en Java. Chaque Mojo de votre projet doit implémenter l'interface `org.apache.maven.plugin.Mojo`, l'exemple suivant présente une classe `Mojo` qui implémente cette interface `Mojo` par l'intermédiaire de la classe `org.apache.maven.plugin.AbstractMojo`. Avant de rentrer dans le code de ce Mojo, commençons par regarder les méthodes de l'interface `Mojo` :

```
void setLog(org.apache.maven.monitor.logging.Log log)
```

Chaque implémentation de `Mojo` doit fournir un moyen de communiquer son avancement. L'exécution du goal a-t-elle réussi ? Ou, au contraire, une erreur est-elle survenue durant l'exécution du goal ? Lorsque Maven charge et exécute un Mojo, il appelle la méthode `setLog()` et passe ainsi au plugin ainsi un journal pour y écrire ses traces.

```
protected Log getLog()
```

Comme nous venons de voir, Maven appelle la méthode `setLog()` avant d'exécuter votre `Mojo`, celui-ci peut donc récupérer le journal injecté par l'intermédiaire de la méthode `getLog()`. Au lieu d'afficher l'avancement sur la sortie standard ou dans la console, il est préférable d'utiliser les méthodes de l'objet `Log`.

```
void execute() throws org.apache.maven.plugin.MojoExecutionException
```

Cette méthode est appelée par Maven au moment de lancer l'exécution du goal.

L'interface `Mojo` est responsable de deux choses : exécuter un goal et en tracer les résultats. Lorsque vous écrivez un plugin, vous pouvez faire hériter vos Mojos de la classe `AbstractMojo`. `AbstractMojo` implémente les méthodes `setLog()` et `getLog()`. Il vous reste donc à implémenter la méthode `execute()` qui est déclarée comme abstraite dans cette classe mère. L'Exemple 17.3, « Un simple EchoMojo » présente une implémentation simple d'un `Mojo` qui se contente d'afficher un message dans la console.

### Exemple 17.3. Un simple EchoMojo

```
package org.sonatype.mavenbook.plugins;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugin.MojoFailureException;
```

```

/**
 * Echos an object string to the output screen.
 * @goal echo
 * @requiresProject false
 */
public class EchoMojo extends AbstractMojo
{
 /**
 * Any Object to print out.
 * @parameter expression="${echo.message}" default-value="Hello World..."
 */
 private Object message;

 public void execute()
 throws MojoExecutionException, MojoFailureException
 {
 getLog().info(message.toString());
 }
}

```

Si vous créez ce Mojo dans le répertoire `${basedir}` sous `src/main/java` à l'emplacement `org/sonatype/mavenbook/mojo/EchoMojo.java` dans le projet créé précédemment et que vous exécutez la commande `mvn install`, vous devriez être capable d'appeler votre directement goal à partir de la ligne de commande :

```
$ mvn org.sonatype.mavenbook.plugins:first-maven-plugin:1.0-SNAPSHOT:echo
```

Cette (longue) ligne de commande est un simple appel à `mvn` suivi d'un argument : `groupId:artifactId:version:goal`. Lorsque vous exécutez cette ligne de commande, vous devriez voir apparaître le message "Hello Maven World..." dans votre console. Pour personnaliser ce message, vous pouvez passer le paramètre suivant à cette même ligne de commande :

```
$ mvn org.sonatype.mavenbook.plugins:first-maven-plugin:1.0-SNAPSHOT:echo \
 -Decho.message="The Eagle has Landed"
```

Cette ligne de commande exécute le goal `EchoMojo` et affiche le message "The Eagle has Landed".

### 17.4.3. Configuration d'un préfixe de Plugin

Spécifier à chaque fois le `groupId`, l'`artifactId`, la `version` et `goal` est un peu lourd (c'est le moindre que l'on puisse dire). Pour éviter cela, vous avez la possibilité d'utiliser un préfixe de plugin. Par exemple, au lieu de taper :

```
$ mvn org.apache.maven.plugins:maven-jar-plugin:2.3:jar
```

Vous pouvez utiliser le préfixe `jar`. Ainsi, la ligne de commande devient beaucoup plus digeste : `mvn jar:jar`. Comment Maven sait-il transformer `jar:jar` en `org.apache.maven.plugins:maven-jar:2.3` ? Maven regarde dans un fichier du dépôt Maven pour obtenir la liste des plugins pour un `groupId` spécifique. Par défaut, Maven est configuré pour rechercher les plugins dans deux groupes :

`org.apache.maven.plugins` et `org.codehaus.mojo`. Lorsque vous spécifiez un nouveau préfixe comme `mvn hibernate3:hbm2ddl`, Maven scanne les métadonnées du dépôt Maven à la recherche du plugin approprié. Maven commence par parcourir les groupes `org.apache.maven.plugins` à la recherche du préfixe `hibernate3`. S'il n'y trouve pas de préfixe `hibernate3`, il continuera en parcourant les métadonnées du groupe `org.codehaus.mojo`.

Lorsque Maven parcourt les métadonnées d'un `groupId`, il récupère deus le dépôt Maven le fichier XML qui contient les métadonnées des artefacts de ce groupe. Ce fichier XML est spécifique à chaque dépôt. Si vous n'avez pas configuré de dépôt, Maven se contentera de chercher dans les métadonnées du groupe `org.apache.maven.plugins` dans votre dépôt Maven local (`~/m2/repository`) dans le fichier `org/apache/maven/plugins/maven-metadata-central.xml`. L'Exemple 17.4, « Métadonnées Maven du groupe Maven Plugin » présente une partie du fichier XML `maven-metadata-central.xml` du groupe `org.apache.maven.plugin`.

#### Exemple 17.4. Métadonnées Maven du groupe Maven Plugin

```
<?xml version="1.0" encoding="UTF-8"?>
<metadata>
 <plugins>
 <plugin>
 <name>Maven Clean Plugin</name>
 <prefix>clean</prefix>
 <artifactId>maven-clean-plugin</artifactId>
 </plugin>
 <plugin>
 <name>Maven Compiler Plugin</name>
 <prefix>compiler</prefix>
 <artifactId>maven-compiler-plugin</artifactId>
 </plugin>
 <plugin>
 <name>Maven Surefire Plugin</name>
 <prefix>surefire</prefix>
 <artifactId>maven-surefire-plugin</artifactId>
 </plugin>
 ...
 </plugins>
</metadata>
```

Comme vous pouvez le voir dans l'Exemple 17.4, « Métadonnées Maven du groupe Maven Plugin », c'est ce fichier `maven-metadata-central.xml` qui rend possible l'exécution de la commande `mvn surefire:test`. Maven parcourt `org.apache.maven.plugins` et `org.codehaus.mojo` : les plugins du groupe `org.apache.maven.plugins` sont considérés comme des plugins du cœur Maven, alors que les plugins du groupe `org.codehaus.mojo` sont considérés comme des plugins moins importants. Le projet Apache Maven gère le groupe `org.apache.maven.plugins`, et c'est une communauté open source indépendante qui a la charge du projet Codehaus Mojo. Si vous désirez publier vos plugins avec votre `groupId`, et que Maven parcourt automatiquement le préfixe de plugin de votre nouveau `groupId`, vous pouvez personnaliser les groupes traités par Maven dans vos fichiers Maven de configuration personnelle.

Pour exécuter votre goal `echo first-maven-plugin` en utilisant la commande `first:echo`, ajoutez le `groupId org.sonatype.mavenbook.plugins` dans votre fichier `~/ .m2/settings.xml` comme le montre l'Exemple 17.5, « Personnaliser les groupes de plugins dans les Settings Maven ». Ainsi, un nouveau groupe sera ajouté en début de liste des groupes analysés par Maven.

#### Exemple 17.5. Personnaliser les groupes de plugins dans les Settings Maven

```
<settings>
...
<pluginGroups>
 <pluginGroup>org.sonatype.mavenbook.plugins</pluginGroup>
</pluginGroups>
</settings>
```

Ceci fait, vous pouvez maintenant exécuter votre goal par l'intermédiaire de la commande `mvn first:echo` à partir de n'importe quel répertoire. Vous constaterez que Maven saura interpréter correctement votre préfixe. Cela fonctionne car le projet respecte certaines conventions de nommage. Si votre projet plugin a un `artifactId` qui respecte le format `maven-first-plugin` ou `first-maven-plugin`, Maven affectera automatiquement le préfixe `first` à votre plugin. En d'autres termes, lorsque le plugin Maven Plugin a généré le descripteur de votre plugin et que vous n'avez pas spécifié de `goalPrefix` dans votre projet, le goal `plugin:descriptor` récupérera le préfixe de votre plugin à partir de son `artifactId` lorsqu'il suit l'un des formats suivants :

- `${prefix}-maven-plugin`, OU
- `maven-${prefix}-plugin`

Si vous désirez définir explicitement un préfixe de plugin, you avez besoin de configurer le plugin Maven Plugin. Le plugin Maven Plugin est responsable de la construction du descripteur de plugin et doit également effectuer certaines tâches durant les phases de packaging et de chargement. Le plugin Maven Plugin est configurable comme n'importe quel autre plugin via la balise `build`. Pour configurer le préfixe de votre plugin, ajoutez le code XML dans la balise `build` du projet `first-maven-plugin`.

#### Exemple 17.6. Configuration d'un préfixe de plugin

```
<?xml version="1.0" encoding="UTF-8"?><project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>org.sonatype.mavenbook.plugins</groupId>
 <artifactId>first-maven-plugin</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>maven-plugin</packaging>
 <name>first-maven-plugin Maven Mojo</name>
 <url>http://maven.apache.org</url>
 <build>
 <plugins>
 <plugin>
 <artifactId>maven-plugin-plugin</artifactId>
 <version>2.3</version>
 <configuration>
 <goalPrefix>blah</goalPrefix>
 </configuration>
 </plugin>
 </plugins>
 </build>
</project>
```

```

 </configuration>
 </plugin>
</plugins>
</build>
<dependencies>
 <dependency>
 <groupId>org.apache.maven</groupId>
 <artifactId>maven-plugin-api</artifactId>
 <version>2.0</version>
 </dependency>
 <dependency>
 <groupId>junit</groupId>
 <artifactId>junit</artifactId>
 <version>3.8.1</version>
 <scope>test</scope>
 </dependency>
</dependencies>
</project>

```

L'Exemple 17.6, « Configuration d'un préfixe de plugin » configure le préfixe du plugin pour qu'il prenne la valeur `blah`. Si vous avez ajouté le groupe `org.sonatype.mavenbook.plugins` à la liste des groupes `pluginGroups` de votre fichier `~/.m2/settings.xml`, vous devriez pouvoir exécuter `EchoMojo` en lançant la commande suivante `mvn echo:blah`.

#### 17.4.4. Les traces d'un plugin

Maven s'occupe de connecter votre `mojo` à un logger en appelant la méthode `setLog()` avant l'exécution de votre `Mojo`. Il fournit une implémentation de la classe `org.apache.maven.monitor.logging.Log`. Cette classe expose des méthodes qui vous pouvez utiliser pour afficher des informations à vos utilisateurs. La classe `Log` fournit plusieurs niveaux de trace, elle propose donc un mécanisme similaire à l'API fournie par `Log4J`<sup>5</sup>. Ces différents niveaux sont utilisables par l'intermédiaire de plus méthodes dédiées : `debug`, `info`, `error` et `warn`. Afin de sauver les arbres, nous ne listerons ici que les méthodes du niveau `debug`.

```
void debug(CharSequence message)
```

Affiche un message en utilisant le niveau `debug`.

```
void debug(CharSequence message, Throwable t)
```

Affiche un message en utilisant le niveau `debug` et ajoute une trace de l'état de la pile à partir d'une `Throwable Exception` ou `Error`)

```
void debug(Throwable t)
```

Affiche la trace de l'état de la pile de la `Throwable (Exception` ou `Error)`

Chacun de ces quatre niveaux expose ces trois méthodes. Ces quatre niveaux répondent à quatre buts différents. Le niveau `debug` est dédié aux personnes qui désirent une vue très détaillée de ce qui se

<sup>5</sup> <http://logging.apache.org/>

passer lors de l'exécution d'un Mojo. Vous ne devez donc jamais présumer que vos utilisateurs utilisent ce niveau. Pour cela, il est préférable d'utiliser le niveau `info` qui est destiné à afficher des messages d'informations généraux lors d'une utilisation normale du mojo. Par exemple, si vous construisez un plugin qui compile du code, utilisez ce niveau `info` pour afficher la sortie du compilateur à l'écran.

Le niveau de logging `warn` est utilisé pour afficher des événements et messages d'erreurs que votre Mojo pourrait rencontrer. Par exemple, si vous essayez d'exécuter un plugin qui compile du code source Ruby mais que ce code n'est pas disponible, affichez un `warning`. Contrairement aux erreurs (qui disposent d'un niveau dédié : `error`), les `warnings` ne sont pas destinés à afficher des erreurs bloquantes. Vous utiliseriez plutôt le niveau `error` lorsque votre Mojo qui doit compiler du code, ne trouve pas de compilateur. Dans ce cas, vous voudrez probablement afficher un message d'erreur et lever une exception. Vous pouvez considérer que vos utilisateurs verront les messages en `info` et tous les messages d'erreurs.

### 17.4.5. Annotations de Mojo

Dans le plugin `first-maven-plugin`, vous n'avez pas écrit de descripteur de plugin, Maven s'est chargé d'en générer un pour vous. Pour cela, Maven utilise les informations du POM du projet et une liste d'annotations présentes dans votre classe `EchoMojo`. La classe `EchoMojo` contient une seule annotation : `@goal`. Voici la liste complète des annotations utilisables dans un `Mojo`.

`@goal <goalName>`

C'est la seule annotation obligatoire, elle donne un nom unique au goal que vous êtes en train d'écrire.

`@requiresDependencyResolution <requireScope>`

Marque le mojo comme un mojo qui nécessite la résolution des dépendances d'un certain scope (explicite ou implicite). Les valeurs autorisées sont : `compile`, `runtime` ou `test`. Par exemple, si la valeur de cette annotation est passée à `test`, Maven ne peut exécuter ce Mojo qu'une fois que les dépendances du scope `test` sont résolues.

`@requiresProject (true|false)`

Marque ce goal comme goal devant être exécuté à l'intérieur d'un projet. La valeur par défaut est `true`. À l'opposé, certains plugins comme le plugin `Archetype` ne le sont pas : ils n'ont pas besoin d'un POM pour s'exécuter.

`@requiresReports (true|false)`

Si vous voulez créer un plugin qui repose sur la présence de rapports, affectez cette annotation à `true`. La valeur par défaut de ce flag est `false`.

`@aggregator (true|false)`

Un goal possédant cette propriété affectée à `true` est supposé ne s'exécuter qu'une seule fois lors d'une exécution de Maven. Cette propriété a été créée pour donner aux développeurs la possibilité d'agréger la sortie d'une série de builds. Par exemple, si vous voulez créer un plugin qui effectue un rapport consolidant la sortie des projets d'un build. Un goal avec `aggregator` affecté à `true`



ne devrait s'exécuter qu'à partir du projet de plus haut-niveau d'un build Maven. Sa valeur par défaut est `false`.

#### `@requiresOnline (true|false)`

Spécifie si un goal donné peut s'exécuter en mode hors connexion. Si un goal nécessite l'utilisation de ressources réseau et que ce flag est activé, Maven affichera une erreur si le goal est exécuté en mode hors connexion. La valeur par défaut pour cette propriété est `false`.

#### `@requiresDirectInvocation`

Si vous affectez cette propriété à `true`, le goal pourra seulement s'exécuter via un appel direct en ligne de commande par un utilisateur. Si quelqu'un essaye de rattacher ce goal à une phase du cycle de vie dans un POM, Maven affichera un message d'erreur. La valeur par défaut pour est propriété est `false`.

#### `@phase <phaseName>`

Cette annotation spécifie la phase par défaut d'un goal. Si vous n'avez pas spécifié de phase pour exécuter ce goal, Maven rattachera celui-ci à la phase définie par cette annotation.

#### `@execute [goal=goalName|phase=phaseName [lifecycle=lifecycleId]]`

Cette annotation peut être utilisée de différentes manières. Si une phase est fournie, Maven exécutera un cycle de vie parallèle en fin d'une phase spécifique. Le résultat de cette exécution parallèle est rendu disponible dans la propriété Maven `${executedProperty}`.

La seconde manière d'utiliser cette annotation est de spécifier explicitement un goal en utilisant la notation `prefix:goal`. Lorsque vous spécifiez un goal, Maven exécute celui-ci dans un environnement parallèle, qui n'affecte pas le build Maven courant.

La troisième manière d'utiliser cette annotation est de spécifier une phase dans un cycle de vie alternatif en utilisant un identifiant de cycle de vie.

```
@execute phase="package" lifecycle="zip"
@execute phase="compile"
@execute goal="zip:zip"
```

Si vous regardez le code source du `EchoMojo`, vous noterez que Maven n'utilise pas des annotations standards au format Java 5. À la place, il utilise les Commons Attributes<sup>6</sup>. Commons Attributes fournissait un moyen d'utiliser des annotations avant qu'elles ne fassent partie intégrante des spécifications Java. Pourquoi ne pas utiliser les annotations Java 5 ? Maven ne les utilise pas car il a été conçu pour des JVMs pré-Java 5. Comme Maven désire rester compatible avec les versions antérieures de Java, il ne peut pas utiliser les fonctionnalités disponibles dans Java 5.

## 17.4.6. Lorsque un Mojo échoue

La méthode `execute()` d'un Mojo lance deux types d'exceptions : `MojoExecutionException` et `MojoFailureException`. La différence entre ces deux exceptions est aussi subtile

<sup>6</sup> <http://commons.apache.org/attributes/>

qu'importante, ces exception ont un impact sur ce qui arrive lorsqu'une exécution "échoue". Une `MojoExecutionException` se doit d'être fatale, quelque chose d'irréparable est arrivé. Lancez cette exception lorsque quelque chose arrive et que vous souhaitez arrêter le build. Par exemple, vous essayez d'écrire sur le disque qui n'a plus d'espace libre ou que vous essayez de vous connecter à un dépôt distant qui ne répond pas. Lancez donc `MojoExecutionException` si vous n'avez aucune chance que votre build puisse continuer correctement, lorsque quelque chose d'horrible est arrivé et que vous voulez arrêter le build et afficher à l'utilisateur un message "BUILD ERROR".

L'exception `MojoFailureException` correspond à un événement moins catastrophique, quelque chose qui ne doit pas déclencher la fin du build. Un test unitaire peut échouer, un checksum MD5 peut échouer, ces deux exemples sont des problèmes, mais pas assez important pour arrêter votre build. C'est dans ce type situation que vous devez utiliser `MojoFailureException`. Maven prévoit plusieurs comportements différents lorsqu'un projet échoue. En voici les descriptions.

Lorsque vous lancez un build Maven, vous pouvez invoquer une série de projets qui peuvent chacun soit réussir soit échouer. Vous pouvez démarrer Maven sous trois modes différents :

`mvn -ff`

Mode fail-fast : Maven échoue (s'arrête) au premier échec.

`mvn -fae`

Mode Fail-at-end : dans ce mode, Maven échouera à la fin du build. Si un projet du reactor de Maven échoue, Maven continuera l'exécution de son build et n'affichera l'échec qu'en fin de build.

`mvn -fn`

Mode Fail never : Maven ne s'arrête pas en cas d'échec et ne reportera pas d'erreur.

Vous pourrez vouloir ignorer les erreurs si vous exécutez un build d'intégration continue et que vous voulez le poursuivre même lorsque le build de l'un des projets échoue. En tant que développeur de plugin, vous devez prendre soin de lancer la bonne exception `MojoExecutionException` ou `MojoFailureException` en fonction de votre type d'erreur.

## 17.5. Paramètres d'un Mojo

Cette notion est aussi importante que celles de la méthode `execute()` et des annotations de Mojo. Un Mojo est configuré par l'intermédiaire de paramètres. Cette section se concentre sur les sujets et la configuration de ces paramètres de Mojo.

### 17.5.1. Affecter des valeurs aux paramètres de Mojo

Dans notre `EchoMojo`, nous avons déclaré un paramètre 'message' en utilisant l'annotation suivante :

```
/**
 * Any Object to print out.
```

```

* @parameter
* expression="${echo.message}"
* default-value="Hello Maven World"
*/
private Object message;

```

L'expression par défaut pour ce paramètre est `${echo.message}`. Cela veut dire que Maven essaiera d'utiliser la valeur de la propriété `echo.message` pour affecter la valeur du message. Si cette propriété est nulle, le paramètre prendra la valeur définie grâce à l'attribut `default-value` de l'annotation `@parameter`. Au lieu d'utiliser la propriété `echo.message`, vous pouvez configurer une valeur pour ce message directement à partir du POM de votre projet.

Plusieurs moyens existent pour renseigner la valeur du paramètre 'message' de notre `EchoMojo`. Premièrement, vous pouvez passer une valeur à partir de la ligne de commande en utilisant la syntaxe suivante (à supposer que vous ayez ajouté `org.sonatype.mavenbook.plugins` à votre `pluginGroups`) :

```
$ mvn first:echo -Decho.message="Hello Everybody"
```

Vous pouvez également spécifier la valeur de ce message en définissant une propriété dans votre POM ou dans votre fichier `settings.xml`.

```

<project>
 ...
 <properties>
 <echo.message>Hello Everybody</echo.message>
 </properties>
</project>

```

Ce paramètre peut également être configuré directement par l'intermédiaire d'une valeur de configuration de votre plugin. Si vous voulez personnaliser directement le paramètre 'message', vous pouvez utiliser la configuration suivante pour votre build. Celle-ci court-circuite la propriété `echo.message` et renseigne le paramètre du Mojo à partir de la configuration du plugin.

```

<project>
 ...
 <build>
 <plugins>
 <plugin>
 <groupId>org.sonatype.mavenbook.plugins</groupId>
 <artifactId>first-maven-plugin</artifactId>
 <version>1.0-SNAPSHOT</version>
 <configuration>
 <message>Hello Everybody!</message>
 </configuration>
 </plugin>
 </plugins>
 </build>
</project>

```

Si vous désirez exécuter votre `EchoMojo` deux fois dans différentes phases du cycle de vie, et si vous voulez configurer le paramètre 'message' avec deux valeurs différentes, vous pouvez configurer la valeur de ce paramètre à partir de la balise `execution` dans votre POM de la manière suivante :

```
<build>
 <build>
 <plugins>
 <plugin>
 <groupId>org.sonatype.mavenbook.plugins</groupId>
 <artifactId>first-maven-plugin</artifactId>
 <version>1.0-SNAPSHOT</version>
 <executions>
 <execution>
 <id>first-execution</id>
 <phase>generate-resources</phase>
 <goals>
 <goal>echo</goal>
 </goals>
 <configuration>
 <message>The Eagle has Landed!</message>
 </configuration>
 </execution>
 <execution>
 <id>second-execution</id>
 <phase>validate</phase>
 <goals>
 <goal>echo</goal>
 </goals>
 <configuration>
 <message>${project.version}</message>
 </configuration>
 </execution>
 </executions>
 </plugin>
 </plugins>
 </build>
</build>
```

Même si ce dernier exemple est assez verbeux, il illustre la flexibilité de Maven. Dans l'exemple précédent, vous avez rattaché l'`EchoMojo` aux phases `validate` et `generate-resources` du cycle de vie par défaut. La première balise `execution` est rattachée à la phase `generate-resources`, elle fournit la valeur suivante à la propriété 'message' : "The Eagle has Landed!". La seconde balise `execution` est rattachée à la phase `validate`, elle fournit une référence à la propriété `${project.version}`. Lorsque vous exécutez la commande `mvn install` sur ce projet, vous verrez que le goal `first:echo` s'exécute deux fois et affiche deux messages différents.

## 17.5.2. Paramètres de Mojo multi-valeurs

Les plugins peuvent avoir des paramètres qui acceptent plusieurs valeurs. Jetez un coup d'oeil au `zipMojo` affiché dans l'Exemple 17.7, « Un plugin avec des paramètres multi-valeurs ». Les paramètres

`includes` et `excludes` acceptent un tableau de `String` qui spécifient des paramètres d'inclusion et l'exclusion pour créer le fichier ZIP.

### Exemple 17.7. Un plugin avec des paramètres multi-valeurs

```
package org.sonatype.mavenbook.plugins

/**
 * Zips up the output directory.
 * @goal zip
 * @phase package
 */
public class ZipMojo extends AbstractMojo
{
 /**
 * The Zip archiver.
 * @parameter \
 * expression="${component.org.codehaus.plexus.archiver.Archiver#zip}"
 */
 private ZipArchiver zipArchiver;

 /**
 * Directory containing the build files.
 * @parameter expression="${project.build.directory}"
 */
 private File buildDirectory;

 /**
 * Base directory of the project.
 * @parameter expression="${basedir}"
 */
 private File baseDirectory;

 /**
 * A set of file patterns to include in the zip.
 * @parameter alias="includes"
 */
 private String[] mIncludes;

 /**
 * A set of file patterns to exclude from the zip.
 * @parameter alias="excludes"
 */
 private String[] mExcludes;

 public void setExcludes(String[] excludes) { mExcludes = excludes; }

 public void setIncludes(String[] includes) { mIncludes = includes; }

 public void execute()
 throws MojoExecutionException
 {
 try {
 zipArchiver.addDirectory(buildDirectory, includes, excludes);
 }
 }
}
```

```

 zipArchiver.setDestFile(new File(baseDirectory, "output.zip"));
 zipArchiver.createArchive();
 } catch(Exception e) {
 throw new MojoExecutionException("Could not zip", e);
 }
}
}
}

```

Pour configurer un paramètre de Mojo multi-valeurs, vous pouvez utiliser une liste d'éléments. Si le nom d'un paramètre multi-valeurs est `includes`, vous utiliserez une balise `includes` contenant des éléments `include`. Si le paramètre multi-valeurs est `excludes`, vous utiliserez une balise `excludes` contenant des éléments `exclude`. Pour configurer le `ZipMojo` pour qu'il ignore tous les fichiers `.txt` et tous les fichiers qui terminent par un tilde, vous pouvez utiliser la configuration de plugin suivante.

```

<project>
...
 <build>
 <plugins>
 <plugin>
 <groupId>org.sonatype.mavenbook.plugins</groupId>
 <artifactId>zip-maven-plugin</artifactId>
 <configuration>
 <excludes>
 <exclude>**/*.txt</exclude>
 <exclude>**/*~</exclude>
 </excludes>
 </configuration>
 </plugin>
 </plugins>
 </build>
</project>

```

### 17.5.3. Dépendre de composants Plexus

Un Mojo est un composant géré par un conteneur IoC appelé Plexus. Un Mojo peut dépendre d'un autre composant géré par Plexus en déclarant un paramètre Mojo et en utilisant les annotations `@parameter` ou `@component`. L'Exemple 17.7, « Un plugin avec des paramètres multi-valeurs » présente le Mojo `ZipMojo` qui dépend d'un composant Plexus en utilisant l'annotation `@parameter`. Nous aurions pu déclarer cette dépendance en utilisant l'annotation `@component`.

Exemple 17.8. Dépendre de composants Plexus

```

/**
 * The Zip archiver.
 * @component role="org.codehaus.plexus.archiver.Archiver" roleHint="zip"
 */
private ZipArchiver zipArchiver;

```

Lorsque Maven instancie ce Mojo, il essaye de récupérer le composant Plexus ayant le rôle et le raccourci de rôle spécifiés. Dans cet exemple, le Mojo sera rattaché au composant ZipArchiver. ce qui permettra à notre `ZipMojo` de créer des fichiers ZIP.

## 17.5.4. Paramètres des annotations d'un Mojo

À moins d'insister pour écrire vos descripteurs de plugin vous-même, vous ne devriez pas à avoir à écrire la moindre ligne de XML. À la place de cela, le plugin Maven Plugin dispose d'un goal `plugin:descriptor` qui est rattaché à la phase `generate-resources`. Ce goal génère le descripteur de plugin à partir d'annotations sur votre Mojo. Pour configurer un paramètre de Mojo, vous pouvez utiliser les annotations présentées ci-dessous sur chacun des des champs privés du Mojo. Il est aussi possible de mettre ces annotations sur les accesseurs publics du Mojo. Cependant, la convention veut que l'on configure les plugins Maven en utilisant les annotations directement sur les champs.

`@parameter [alias="someAlias"] [expression="{someExpression}"] [default-value="value"]`

Marque comme paramètre un champ privé (ou un accesseur). L'`alias` fournit le nom du paramètre. Si l'`alias` n'est pas renseigné, Maven utilise le nom de la variable comme nom du paramètre. L'`expression` est l'expression utilisée par Maven pour obtenir une valeur du paramètre. L'expression fait souvent référence à une propriété : `{echo.message}`. La valeur par défaut est renseignée par `default-value`, celle-ci est utilisée si aucune valeur ne peut être obtenue à partir de l'expression ou si aucune valeur n'a été fournie par la configuration du plugin dans le POM.

`@required`

Si cette annotation est présente, ce paramètre doit obligatoirement être renseigné avant l'exécution du Mojo. Si Maven essaye d'exécuter le Mojo alors que ce paramètre est nul alors Maven va lancer une erreur.

`@readonly`

Si cette annotation est présente, alors l'utilisateur ne peut pas configurer ce paramètre via le fichier POM. Cette annotation est utilisée en combinaison avec l'attribut `expression` de l'annotation de ce paramètre. Par exemple, si vous voulez être sûr que cette propriété possède la valeur que la propriété `finalName` du POM, vous utiliserez l'expression `{build.finalName}` et ajouterez l'annotation `@readonly`. Ainsi, l'utilisateur ne pourra changer la valeur de ce paramètre qu'en changeant la valeur du `finalName` dans le POM.

`@component`

Demande à Maven d'injecter un composant Plexus dans le champ. Voici un exemple d'utilisation de cette annotation `@component` :

```
@component role="org.codehaus.plexus.archiver.Archiver" roleHint="zip"
```

Cette ligne a pour effet de récupérer le composant `ZipArchiver` à partir de Plexus. Il s'agit de l'Archiver qui correspond au `role hint zip`. Au lieu d'injecter des composants, vous pouvez également utiliser l'annotation `@parameter` en utilisant une expression :

```
@parameter expression="\${component.org.codehaus.plexus.archiver.Archiver#zip}"
```

Si ces deux annotations ont le même comportement, l'annotation `@component` est à privilégier pour configurer des dépendances sur des composants Plexus.

### `@deprecated`

Indique que ce paramètre est déprécié. Les utilisateurs peuvent continuer de configurer ce paramètre, mais un warning sera affiché.

## 17.6. Plugins et le cycle de vie Maven

Dans le chapitre Chapitre 10, Cycle de vie du build, nous avons vu que les cycles de vie peuvent être personnalisés en fonction du type de packaging. Un plugin peut soit introduire un nouveau type de packaging, soit personnaliser le cycle de vie. Dans cette section, nous allons voir comment personnaliser le cycle de vie à partir d'un plugin Maven. Nous verrons également comment dire à un Mojo de d'exécuter un cycle de vie parallèle.

### 17.6.1. Exécution dans un cycle de vie parallèle

Imaginons que vous devez écrire un plugin qui dépend du résultat du build précédent. Par exemple, peut-être que le goal du `zipMojo` ne peut être lancé que si un élément du résultat existe pour être inclus dans l'archive. Il est possible de préciser des goal prérequis en utilisant l'annotation `@execute` sur un Mojo. Cette annotation forcera Maven à lancer un build parallèle et exécuter un goal ou un cycle de vie dans cette seconde instance de Maven sans que tout cela affecte le build courant. Si vous avez écrit un Mojo que vous exécutez une fois par jour pour lancer la commande `mvn install` et packager le résultat sous un format de distribution personnalisé. Votre descripteur de Mojo peut demander à Maven d'exécuter le cycle de vie par défaut jusqu'à la phase `install` et d'exposer le résultat de ce projet dans votre mojo en utilisant la propriété `${executedProject}`. Vous pouvez ensuite référencer cette propriété dans un projet et effectuer ainsi un traitement à posteriori.

Autre possibilité, si vous avez un goal qui effectue quelque chose de complètement indépendant du cycle de vie par défaut. Imaginons quelque chose de complètement inattendu, vous disposez peut-être d'un goal qui transforme un fichier WAV en MP3 en utilisant quelque chose comme LAME. Avant de faire cela, vous voulez parcourir un cycle de vie qui transforme un fichier MIDI en un fichier WAV (vous pouvez faire faire tout ce que vous voulez à Maven, et cet exemple n'est pas aussi "tiré par les cheveux" qu'on pourrait le croire). Vous avez donc créé un cycle de vie `midi-sound` et vous voulez utiliser le résultat de la phase `install` de ce cycle de vie dans une application web qui possède un type de packaging `war`. Comme votre projet est exécuté dans le cycle de vie `war`, vous devez avoir un goal qui va lancer votre cycle de vie `midi-source` en parallèle de ce build. Pour cela, vous devez annoter votre mojo avec `@execute lifecycle="midi-source" phase="install"`.

### `@execute goal="<goal>"`

Cette annotation provoque l'exécution d'un goal donné avant celui annoté. Le nom du goal doit être donné en utilisant la notation `prefix:goal`.



`@execute phase="<phase>"`

Un cycle de vie alternatif, spécifié par la phase donnée, sera exécuté en parallèle avant de reprendre l'exécution courante.

`@execute lifecycle="<lifecycle>" phase="<phase>"`

Cette annotation provoque l'exécution du cycle de vie alternatif. Un cycle de vie personnalisé peut être défini dans le fichier `META-INF/maven/lifecycle.xml`.

## 17.6.2. Création d'un cycle de vie personnalisé

Un cycle de vie personnalisé doit être packagé dans un plugin dans un fichier `META-INF/maven/lifecycle.xml` dans l'un des dossiers de ressources : par exemple `src/main/resources`. Le fichier `lifecycle.xml` suivant déclare un cycle de vie nommé `zipcycle`. Celui-ci contient un unique goal, `zip`, dans une seule phase, `package`.

Exemple 17.9. Définition d'un cycle de vie par défaut dans le fichier `lifecycle.xml`

```
<lifecycles>
 <lifecycle>
 <id>zipcycle</id>
 <phases>
 <phase>
 <id>package</id>
 <executions>
 <execution>
 <goals>
 <goal>zip</goal>
 </goals>
 </execution>
 </executions>
 </phase>
 </phases>
 </lifecycle>
</lifecycles>
```

Si vous voulez exécuter la phase `zipcycle` au sein d'un autre build, vous pouvez créer un `ZipForkMojo` qui utilise l'annotation `@execute` pour demander à Maven de traverser la phase `zipcycle` avant l'exécution du build courant.

Exemple 17.10. Fork d'un cycle de vie à partir d'un Mojo

```
/**
 * Forks a zip lifecycle.
 * @goal zip-fork
 * @execute lifecycle="zipcycle" phase="package"
 */
public class ZipForkMojo extends AbstractMojo
{
```

```

public void execute()
 throws MojoExecutionException
{
 getLog().info("doing nothing here");
}
}

```

Exécuter le `ZipForkMojo` lancera un fork du cycle de vie. Si vous avez configuré votre plugin pour qu'il s'exécute avec le préfixe `zip`, l'exécution de `zip-fork` devrait produire une sortie ressemblant à cela.

```

$ mvn zip:zip-fork
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'zip'.
[INFO] -----
[INFO] Building Maven Zip Forked Lifecycle Test
[INFO] task-segment: [zip:zip-fork]
[INFO] -----
[INFO] Preparing zip:zip-fork
[INFO] [site:attach-descriptor]
[INFO] [zip:zip]
[INFO] Building zip: \
~/maven-zip-plugin/src/projects/zip-lifecycle-test/target/output.zip
[INFO] [zip:zip-fork]
[INFO] doing nothing here
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 1 second
[INFO] Finished at: Sun Apr 29 16:10:06 CDT 2007
[INFO] Final Memory: 3M/7M
[INFO] -----

```

L'exécution de `zip-fork` a créé un nouveau cycle de vie, Maven a ensuite exécuté le cycle de vie `zipcycle` et a affiché le message contenu dans notre `ZipFormMojo`.

### 17.6.3. Surcharge du cycle de vie par défaut

Une fois que vous avez créé votre propre cycle de vie et que vous l'avez lancé depuis un Mojo, la question suivante que vous pouvez vous poser est comment surcharger le cycle de vie par défaut. Comment créer un cycle de vie personnalisé et comment le rattacher à vos projets ? Dans le Chapitre 10, Cycle de vie du build, nous avons vu que le type de packaging d'un projet définit son cycle de vie. Le comportement diffère selon les types de packaging, ainsi `war` a rattaché différents goals au cycle de vie pour construire son package. Les cycles de vie personnalisés, comme `swf`, provenant du plugin Israfil Flex 3, rattachent différents goals à la phase compile. Lorsque vous créez un cycle de vie personnalisé, vous pouvez rattacher celui-ci à un type de packaging en fournissant une configuration Plexus dans l'archive de votre plugin.

Pour définir un nouveau cycle de vie pour un nouveau type de packaging, vous devez configurer le composant `LifecycleMapping` dans Plexus. Dans votre projet plugin, créez un fichier `META-INF/`

plexus/components.xml sous src/main/resources. Éditez ce fichier et ajoutez lui le contenu de l'Exemple 17.11, « Surcharge du cycle de vie par défaut ». Mettez le nom du type de packaging dans role-hint, et rattachez les goals aux phases en utilisant leurs coordonnées (sans la version). Vous pouvez associer plusieurs goals à une phase en utilisant une liste séparée par des virgules.

#### Exemple 17.11. Surcharge du cycle de vie par défaut

```
<component-set>
 <components>
 <component>
 <role>org.apache.maven.lifecycle.mapping.LifecycleMapping</role>
 <role-hint>zip</role-hint>
 <implementation>
 org.apache.maven.lifecycle.mapping.DefaultLifecycleMapping
 </implementation>
 <configuration>
 <phases>
 <process-resources>
 org.apache.maven.plugins:maven-resources-plugin:resources
 </process-resources>
 <compile>
 org.apache.maven.plugins:maven-compiler-plugin:compile
 </compile>
 <package>org.sonatype.mavenbook.plugins:maven-zip-plugin:zip</package>
 </phases>
 </configuration>
 </component>
 </components>
</component-set>
```

Si vous créez un plugin qui définit un nouveau type de packaging et un nouveau cycle de vie, vous devez affecter la balise extensions à true dans le POM de votre projet. Cela a pour effet de demander à Maven de scanner votre plugin à la recherche du fichier components.xml dans le répertoire META-INF/plexus. Ainsi, votre nouveau type de packaging sera rendu disponible dans votre projet.

#### Exemple 17.12. Configuration d'un plugin en Extension

```
<project>
 ...
 <build>
 ...
 <plugins>
 <plugin>
 <groupId>com.training.plugins</groupId>
 <artifactId>maven-zip-plugin</artifactId>
 <extensions>true</extensions>
 </plugin>
 </plugins>
 </build>
</project>
```

Une fois que vous avez ajouté le plugin avec sa balise `extensions` activée, vous pourrez utiliser votre nouveau type de packaging. Votre projet pourra ainsi exécuter le cycle de vie personnalité associé à celui-ci.

# Chapitre 18. Utilisation des archetypes Maven

## 18.1. Introduction aux archetypes Maven

Un archetype est un modèle de projet Maven. Le plugin Maven Archetype les utilise pour créer de nouveaux projets. Les archétypes permettent à des projets Open Source comme Apache Wicket ou Apache Cocoon de présenter à leurs utilisateurs les fondations sur lesquelles construire une nouvelle application. Les archétypes peuvent également être utilisés dans une société pour encourager la standardisation d'un ensemble de projets. Si vous travaillez dans une société avec de grandes équipes de développement et que tous vos projets doivent respecter une même structure, vous pouvez publier un archetype. Ainsi, n'importe quel membre de l'équipe de développement sera capable de démarrer rapidement un nouveau projet en respectant la structure préconisée. Vous pouvez créer un nouveau projet avec un archetype en utilisant le plugin Maven Archetype à partir de la ligne de commande ou en utilisant le wizard de création de nouveau projet proposé par le plugin `m2eclipse` comme le présente le livre "Developing with Eclipse and Maven"<sup>1</sup>.

## 18.2. Utilisation des archétypes

Vous pouvez utiliser un archetype en utilisant le goal `generate` du plugin Maven Archetype à partir de la ligne de commande ou du plugin Eclipse `m2eclipse`.

### 18.2.1. Utilisation d'un archetype à partir de la ligne de commande

La ligne de commande suivante peut être utilisée pour générer un projet à partir de l'archetype `quickstart`.

```
mvn archetype:generate \
-DgroupId=org.sonatype.mavenbook \
-DartifactId=quickstart \
-Dversion=1.0-SNAPSHOT \
-DpackageName=org.sonatype.mavenbook \
-DarchetypeGroupId=org.apache.maven.archetypes \
-DarchetypeArtifactId=maven-archetype-quickstart \
-DarchetypeVersion=1.0 \
-DinteractiveMode=false
```

Le goal `generate` accepte les paramètres suivants :

`groupId`

Il s'agit du `groupId` du projet que vous voulez créer.

---

<sup>1</sup> <http://www.sonatype.com/books/m2eclipse-book/reference/>

`artifactId`

Il s'agit de l'`artifactId` du projet que vous voulez créer.

`version`

Il d'agit de la `version` du projet que vous voulez créer (valeur par défaut : 1.0-SNAPSHOT).

`packageName`

Le package par défaut de votre projet (par défaut, il s'agit du `groupId`).

`archetypeGroupId`

Le `groupId` de l'archétype que vous désirez utiliser.

`archetypeArtifactId`

L'`artifactId` de l'archétype que vous désirez utiliser.

`archetypeVersion`

La `version` de l'archétype que vous désirez utiliser.

`interactiveMode`

Lorsque le goal `generate` est exécuté dans ce mode, les paramètres listés précédemment seront demandés à l'utilisateur les uns après les autres durant l'exécution. Dans le cas contraire, le goal `generate` utilise les valeurs passées en la ligne de commande.

Une fois que vous avez exécuté le goal `generate` avec la ligne de commande précédente, un répertoire `quickstart` contenant un nouveau projet Maven est créé. Cette ligne de commande est relativement difficile à retenir. Dans la section suivante, nous générerons ce même projet en utilisant le mode interactif.

## 18.2.2. Utilisation du goal Generate en mode interactif

Le moyen le plus simple d'utiliser le plugin Maven Archetype pour générer un nouveau projet Maven à partir d'un archétype est d'utiliser le goal `archetype:generate` en mode interactif. Ainsi, le goal `generate` vous demandera de choisir un archétype parmi une liste d'archétypes disponibles. Par défaut, le mode interactif est activé, la seule chose que vous avez à faire pour générer un nouveau projet est d'exécuter cette simple commande : `mvn archetype:generate`.

```
$ mvn archetype:generate
[INFO] -----
[INFO] Building Maven Default Project
[INFO] task-segment: [archetype:generate] (aggregator-style)
[INFO] [archetype:generate]
[INFO] Generating project in Interactive mode
[INFO] No archetype defined. Using maven-archetype-quickstart
Choose archetype:
1: internal -> appfuse-basic-jsf
2: internal -> appfuse-basic-spring
3: internal -> appfuse-basic-struts
4: internal -> appfuse-basic-tapestry
5: internal -> appfuse-core
```

```

6: internal -> appfuse-modular-jsf
7: internal -> appfuse-modular-spring
8: internal -> appfuse-modular-struts
9: internal -> appfuse-modular-tapestry
10: internal -> maven-archetype-j2ee-simple
11: internal -> maven-archetype-marmalade-mojo
12: internal -> maven-archetype-mojo
13: internal -> maven-archetype-portlet
14: internal -> maven-archetype-profiles
15: internal -> maven-archetype-quickstart
16: internal -> maven-archetype-site-simple
17: internal -> maven-archetype-site
18: internal -> maven-archetype-webapp
19: internal -> jini-service-archetype
20: internal -> softeu-archetype-seam
21: internal -> softeu-archetype-seam-simple
22: internal -> softeu-archetype-jsf
23: internal -> jpa-maven-archetype
24: internal -> spring-osgi-bundle-archetype
25: internal -> confluence-plugin-archetype
26: internal -> jira-plugin-archetype
27: internal -> maven-archetype-har
28: internal -> maven-archetype-sar
29: internal -> wicket-archetype-quickstart
30: internal -> scala-archetype-simple
31: internal -> lift-archetype-blank
32: internal -> lift-archetype-basic
33: internal -> cocoon-22-archetype-block-plain
34: internal -> cocoon-22-archetype-block
35: internal -> cocoon-22-archetype-webapp
36: internal -> myfaces-archetype-helloworld
37: internal -> myfaces-archetype-helloworld-facelets
38: internal -> myfaces-archetype-trinidad
39: internal -> myfaces-archetype-jsfcomponents
40: internal -> gmaven-archetype-basic
41: internal -> gmaven-archetype-mojo
Choose a number: 15

```

Pour commencer, le goal `archetype:generate` en mode interactif affiche la liste des archétypes disponibles pour l'utilisateur. Le plugin Maven Archetype est fourni avec un catalogue d'archétypes correspondants à la plupart des types de projets standards (archétypes 10 à 18). Le catalogue d'archétypes du plugin peut également être complété par des archétypes tiers, comme des archétypes permettant de générer des projets AppFuse, des plugins Confluence ou Jira, des applications Wicket, des applications Scala, des projets Groovy, ... Vous pouvez consulter une liste non exhaustive des archétypes proposés par des tiers dans la partie Section 18.3.2, « Archétypes tiers notables ».

Une fois que vous avez choisi un archétype, le plugin Maven Archetype télécharge celui-ci et vous demande de fournir les valeurs suivantes pour générer votre nouveau projet :

- `groupId`
- `artifactId`

- version
- package

```
[INFO] artifact org.apache.maven.archetypes:maven-archetype-quickstart: checking for updates from central
Downloading: http://repo1.maven.org/maven2/org/apache/maven/archetypes/maven-archetype-quickstart/1.0-SNAPSHOT
4K downloaded
Define value for groupId: : org.sonatype.mavenbook
Define value for artifactId: : quickstart
Define value for version: 1.0-SNAPSHOT: : 1.0-SNAPSHOT
Define value for package: org.sonatype.mavenbook: : org.sonatype.mavenbook
Confirm properties configuration:
groupId: org.sonatype.mavenbook
artifactId: quickstart
version: 1.0-SNAPSHOT
package: org.sonatype.mavenbook
Y: : Y
```

Une fois ceci fait, le plugin Maven Archetype génère le projet dans le répertoire que vous avez indiqué.

```
[INFO] -----
[INFO] Using following parameters for creating OldArchetype: maven-archetype-quickstart:RELEASE
[INFO] -----
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: basedir, Value: /Users/tobrien/tmp
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: quickstart
[INFO] ***** End of debug info from resources from generated POM *****
[INFO] OldArchetype created in dir: /Users/tobrien/tmp/quickstart
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 1 minute 57 seconds
[INFO] Finished at: Sun Oct 12 15:39:14 CDT 2008
[INFO] Final Memory: 8M/15M
[INFO] -----
```

### 18.2.3. Utilisation d'un archétype à partir du plugin Eclipse m2eclipse

Le plugin `m2eclipse` vous permet de créer des projets Maven à partir d'archétypes en proposant un wizard intuitif pour rechercher, sélectionner et configurer un archétype Maven. Pour plus d'informations sur la génération de projet Maven à partir d'un archétype et du plugin Eclipse `m2eclipse`, référez-vous au chapitre "Creating a Maven Project from a Maven Archetype"<sup>2</sup> du livre "Developing with Eclipse and Maven"<sup>3</sup>.

<sup>2</sup> <http://www.sonatype.com/books/m2eclipse-book/reference/eclipse-sect-creating-project.html#eclipse-sect-m2e-create-archetype>

<sup>3</sup> <http://www.sonatype.com/books/m2eclipse-book/>



## 18.3. Archétypes disponibles

Comme de plus en plus de projets adoptent Maven, de plus en plus d'artefacts sont publiés et fournissent aux utilisateurs un moyen simple de créer des projets à partir de modèles existants. Cette section présente quelques-uns de ces archétypes.

### 18.3.1. Archétypes Maven communs

La plupart des archétypes les plus simples sont regroupés sous le `groupId` : `org.apache.maven.archetypes`. Ces archétypes ne possèdent que quelques options et proposent la création de projets relativement basiques. Vous pouvez les utiliser pour générer les quelques éléments de structure qui distinguent un projet Maven d'un autre. Par exemple, l'archétype `webapp` présenté ici se contente de générer un fichier `web.xml` dans le répertoire `${basedir}/src/main/webapp/WEB-INF` sans même proposer le squelette d'une Servlet. Dans le paragraphe Section 18.3.2, « Archétypes tiers notables » vous pouvez effectuer un survol rapide de quelques-uns des archétypes tiers les plus remarquables comme ceux d'AppFuse ou de Cocoon.

Les archétypes suivants se trouvent dans le `groupId` `org.apache.maven.archetypes` :

#### 18.3.1.1. maven-archetype-quickstart

L'archétype `quickstart` est un simple projet de type JAR avec une simple dépendance sur JUnit. Après avoir généré un projet avec cet archétype, une seule classe sera produite : `App`. Celle-ci sera générée dans le package par défaut et contiendra une méthode `main()` qui affiche "Hello World!" sur la sortie standard. Vous aurez également un test unitaire JUnit dans une classe nommée `AppTest` avec une méthode `testApp()` dont le code est trivial.

#### 18.3.1.2. maven-archetype-webapp

Cet archétype crée un simple projet de type WAR avec une dépendance sur JUnit. Le dossier `${basedir}/src/main/webapp` contient le squelette minimale d'une application web : une page `index.jsp` et un fichier `web.xml` le plus simple possible. Même si cet archétype inclut une dépendance sur JUnit, aucun test n'est créé. Si vous cherchez à produire une application web fonctionnelle, cet archétype vous décevra probablement. Consulter la partie Section 18.3.2, « Archétypes tiers notables » pour trouver des exemples d'archétypes plus évolués.

#### 18.3.1.3. maven-archetype-mojo

Cet archétype crée un simple projet de type `maven-plugin` contenant une seule classe nommée `MyMojo` dans le package par défaut du projet. La classe `MyMojo` contient un goal `touch` rattaché à la phase `process-resources` et qui crée un fichier `touch.txt` dans le répertoire `target/` lorsqu'on l'exécute. Ce nouveau projet a une dépendance sur `maven-plugin-api` et JUnit.

## 18.3.2. Archétypes tiers notables

Cette section vous propose un bref aperçu de quelques-uns des archétypes fournis par des tiers non associés au projet Maven. Si vous recherchez une liste plus complète des archétypes disponibles, jetez un oeil à la liste des archétypes disponibles dans `m2eclipse`. `m2eclipse` vous permet de créer un nouveau projet Maven à partir d'une liste sans cesse plus nombreuse de plus de 80 archétypes couvrant la plupart des technologies. Le chapitre "Creating a Maven Project from a Maven Archetype"<sup>4</sup> du livre "Developing with Eclipse and Maven"<sup>5</sup> contient la liste des archétypes directement disponibles à partir du plugin `m2eclipse`. Les archétypes listés dans cette section sont disponibles dans la liste des archétypes par défaut générée par le mode interactif du goal `generate`.

### 18.3.2.1. AppFuse

AppFuse est un framework développé par Matt Raible. Vous pouvez le voir comme la Pierre de Rosette des technologies Java les plus connues comme Spring Framework, Hibernate ou iBatis. En utilisant AppFuse, vous pouvez rapidement créer une application multi-tier complète qui peut s'interfacer à plusieurs frameworks de présentation web, comme Java Server Faces, Struts ou Tapestry. En créant AppFuse 2.0, Matt Raible a mis à jour son framework en utilisant Maven 2 pour profiter de ses fonctionnalités de gestion de dépendances et de création d'archétypes. AppFuse 2 fournit les archétypes suivants, tous sont regroupés dans le `groupId org.appfuse.archetypes` :

```
appfuse-basic-jsf et appfuse-modular-jsf
```

Application complète utilisant JSF (Java Server Faces) comme couche de présentation.

```
appfuse-basic-spring et appfuse-modular-spring
```

Application complète utilisant Spring MVC comme couche de présentation.

```
appfuse-basic-struts et appfuse-modular-struts
```

Application complète utilisant Struts 2 comme couche de présentation.

```
appfuse-basic-tapestry et appfuse-modular-tapestry
```

Application complète utilisant Tapestry comme couche de présentation.

```
appfuse-core
```

Modèle objet et sa couche de persistance sans couche présentation.

Les archétypes dont l'artifactId respecte le format `appfuse-basic-*` produisent une applications complète correspondant à un unique projet Maven. Les archétypes dont l'artifactId respecte le format `appfuse-modular-*` produisent une applications complète sous la forme d'un projet Maven multimodule qui sépare les objets du modèle métier, la couche de persistance et la couche présentation. Voici un exemple permettant de générer une application web modulaire utilisant Spring MVC :

```
$ mvn archetype:generate \
-DarchetypeArtifactId=appfuse-modular-spring \
-
```

<sup>4</sup> <http://www.sonatype.com/books/m2eclipse-book/reference/eclipse-sect-creating-project.html#eclipse-sect-m2e-create-archetype>

<sup>5</sup> <http://www.sonatype.com/books/m2eclipse-book/>

```
-DarchetypeGroupId=org.appfuse.archetypes \
-DgroupId=org.sonatype.mavenbook \
-DartifactId=mod-spring \
-Dversion=1.0-SNAPSHOT \
-DinteractiveMode=false[INFO] Scanning for projects...
...
[INFO] [archetype:generate]
[INFO] Generating project in Batch mode
[INFO] Archetype [org.appfuse.archetypes:appfuse-modular-spring:RELEASE] found in catalog internal
[INFO] -----
[INFO] Using following parameters for creating OldArchetype: appfuse-modular-spring:RELEASE
[INFO] -----
[INFO] Parameter: groupId, Value: org.sonatype.mavenbook
[INFO] Parameter: packageName, Value: org.sonatype.mavenbook
[INFO] Parameter: basedir, Value: /Users/tobrien/tmp
[INFO] Parameter: package, Value: org.sonatype.mavenbook
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: mod-spring
...
[INFO] OldArchetype created in dir: /Users/tobrien/tmp/mod-spring
[INFO] -----
[INFO] BUILD SUCCESSFUL
$ cd mod-spring
$ mvn
... (an overwhelming amount of activity ~5 minutes)
$ cd web
$ mvn jetty:run-war
... (Maven Jetty plugin starts a Servlet Container on port 8080)
```

Il faut moins de cinq minutes entre la génération du projet à partir d'un archétype AppFuse et l'exécution de cette application web avec son authentification et son système de gestion des utilisateurs. Voici d'un bon exemple de la véritable force qu'apporte l'utilisation des archétypes Maven pour générer de nouvelles applications. Nous avons un peu simplifié le processus d'installation d'AppFuse, car nous avons oublié tout ce qui concerne le téléchargement et l'installation de la base de données MySQL. Mais cela n'a rien de très compliqué au final, surtout si vous lisez "AppFuse Quickstart Documentation"<sup>6</sup>.

### 18.3.2.2. Plugins Confluence et JIRA

Atlassian a créé quelques archétypes pour aider les personnes intéressées dans le développement de plugins Confluence ou JIRA. Confluence et JIRA sont respectivement un wiki et un gestionnaire d'anomalies. Ces deux produits ont acquis un grand nombre d'utilisateurs en distribuant des licences gratuites pour les projets open source. Les deux artefacts `jira-plugin-archetype` et `confluence-maven-archetype` ont le même `groupId` `com.atlassian.maven.archetypes`. Lorsque vous générez un plugin Confluence, l'archétype va créer un fichier `pom.xml` qui contient les références au dépôt Atlassian et des dépendances sur l'artefact Confluence. Au final, le projet plugin Confluence disposera d'un exemple de macro et d'un descripteur `atlassian-plugin.xml`. En utilisant l'archétype JIRA, vous créez un projet avec une classe `MyPlugin` vierge et un descripteur `atlassian-plugin.xml` dans le répertoire `${basedir}/src/main/resources`.

<sup>6</sup> <http://appfuse.org/display/APF/AppFuse+QuickStart>

Pour plus d'information sur le développement de plugin Confluence avec Maven 2, référez-vous à la page "Developing Confluence Plugins with Maven 2"<sup>7</sup> disponibles sur le wiki du projet. De même, pour plus d'informations sur le développement de plugins JIRA en utilisant Maven 2, référez-vous à la page "How to Build and Atlassian Plugin"<sup>8</sup> disponible sur l'Atlassian Developer Network.

### 18.3.2.3. Wicket

Apache Wicket est un framework web orienté composant. Il se focalise sur la gestion de l'état de composants serveur écrits en Java et en simple HTML. Là où un framework comme Spring MVC ou Ruby on Rails se focalise sur la fusion d'objets d'une requête avec une série de modèles de page, Wicket se concentre plutôt sur la capture des interactions et sur la structure de la page à partir d'une série de classes POJO Java. Vous pouvez générer un projet Wicket en utilisant le plugin Maven Archetype :

```
$ mvn archetype:generate
... (select the "wicket-archetype-quickstart" artifact from the interactive menu) ...
... (supply a groupId, artifactId, version, package) ...
... (assuming the artifactId is "ex-wicket") ...
$ cd ex-wicket
$ mvn install
... (a lot of Maven activity) ...
$ mvn jetty:run
... (Jetty will start listening on port 8080) ...
```

Tout comme l'archétype AppFuse, cet archétype crée une application web directement exécutable par le plugin Maven Jetty. Une fois lancée, rendez-vous à l'adresse <http://localhost:8080/ex-wicket>, pour consulter votre application web nouvellement produite.



## Note

Pensez à la puissance qu'apportent les archétypes Maven face à une simple approche 'copier-coller' qui a caractérisé les dernières années de développement web. Il y a six ans, sans le plugin Maven Archetype, vous auriez dû parcourir un livre sur AppFuse ou sur Wicket et apprendre comment bien utiliser ces frameworks avant de pouvoir en exécuter le moindre bout de code dans un conteneur de servlets. Ou alors, vous auriez juste copié collé un projet existant et commencé à le personnaliser selon vos besoins. Avec le plugin Maven Archetype, les développeurs de framework peuvent vous fournir une application personnalisée à vos besoins en quelques minutes. Il s'agit d'un changement profond qui n'a pas encore atteint toutes les entreprises. On peut donc s'attendre à voir se multiplier le nombre d'artefacts dans les prochaines années.

<sup>7</sup> <http://confluence.atlassian.com/display/DISC/Developing+Confluence+Plugins+with+Maven+2>

<sup>8</sup> <http://confluence.atlassian.com/display/DEVNET/How+to+Build+an+Atlassian+Plugin>

## 18.4. Publication d'archétypes

Une fois que vous avez produit quelques artefacts, vous voudrez probablement les partager avec le reste du monde. Pour cela, vous devez créer un catalogue d'archétypes. Un catalogue d'archétypes est un fichier XML que le plugin Maven Archetype peut consulter pour rechercher des archétypes dans un repository. L'Exemple 18.1, « Catalogue d'archétypes du projet Apache Cocoon » affiche le contenu du catalogue d'archétypes du projet Apache Cocoon qui peut être trouvé à l'adresse <http://cocoon.apache.org/archetype-catalog.xml>.

Exemple 18.1. Catalogue d'archétypes du projet Apache Cocoon

```
<archetype-catalog>
 <archetypes>
 <archetype>
 <groupId>org.apache.cocoon</groupId>
 <artifactId>cocoon-22-archetype-block-plain</artifactId>
 <version>1.0.0</version>
 <description>Creates an empty Cocoon block; useful if you want to add
 another block to a Cocoon application</description>
 </archetype>
 <archetype>
 <groupId>org.apache.cocoon</groupId>
 <artifactId>cocoon-22-archetype-block</artifactId>
 <version>1.0.0</version>
 <description>Creates a Cocoon block containing some small
 samples</description>
 </archetype>
 <archetype>
 <groupId>org.apache.cocoon</groupId>
 <artifactId>cocoon-22-archetype-webapp</artifactId>
 <version>1.0.0</version>
 <description>Creates a web application configured to host Cocoon blocks.
 Just add the block dependencies</description>
 </archetype>
 </archetypes>
</archetype-catalog>
```

Pour générer un tel catalogue, vous devez parcourir un dépôt Maven puis produire ce fichier XML catalogue. Le plugin Maven Archetype possède un goal nommé `crawl` qui effectue cela. En exécutant le goal `archetype:crawl` à partir de la ligne de commande sans argument, le plugin Maven Archetype va parcourir votre dépôt local à la recherche d'archétypes et créer un fichier `archetype-catalog.xml` dans le répertoire `~/.m2/repository`.

```
[tobrien@MACBOOK repository]$ mvn archetype:crawl
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] -----
[INFO] Building Maven Default Project
```

```

[INFO] task-segment: [archetype:crawl] (aggregator-style)
[INFO] -----
[INFO] [archetype:crawl]
repository /Users/tobrien/.m2/repository
catalogFile null
[INFO] Scanning /Users/tobrien/.m2/repository/ant/ant/1.5/ant-1.5.jar
[INFO] Scanning /Users/tobrien/.m2/repository/ant/ant/1.5.1/ant-1.5.1.jar
[INFO] Scanning /Users/tobrien/.m2/repository/ant/ant/1.6/ant-1.6.jar
[INFO] Scanning /Users/tobrien/.m2/repository/ant/ant/1.6.5/ant-1.6.5.jar
...
[INFO] Scanning /Users/tobrien/.m2/repository/xmlrpc/xmlrpc/1.2-b1/xmlrpc-1.2-b1.jar
[INFO] Scanning /Users/tobrien/.m2/repository/xom/xom/1.0/xom-1.0.jar
[INFO] Scanning /Users/tobrien/.m2/repository/xom/xom/1.0b3/xom-1.0b3.jar
[INFO] Scanning /Users/tobrien/.m2/repository/xpp3/xpp3_min/1.1.3.4.0/xpp3_min-1.1.3.4.0.jar
[INFO] -----
[INFO] BUILD SUCCESSFUL
[INFO] -----
[INFO] Total time: 31 seconds
[INFO] Finished at: Sun Oct 12 16:06:07 CDT 2008
[INFO] Final Memory: 6M/12M
[INFO] -----

```

Si vous êtes intéressé par la création d'un catalogue d'archétypes, c'est généralement parce que vous êtes un projet Open Source ou une société qui dispose d'un ensemble d'archétypes à partager. Ces archétypes sont probablement déjà disponibles dans un dépôt que vous avez besoin d'explorer pour générer un catalogue. En d'autres termes, vous aurez probablement envie de parcourir un répertoire sur un dépôt Maven existant et générer un fichier `archetype-catalog.xml` à la racine du dépôt. Pour cela, il vous faut passer le catalogue et le dépôt en paramètre du goal `archetype:crawl`.

La ligne de commande suivante permet de créer un fichier catalogue à l'emplacement `/var/www/html/archetype-catalog.xml`, le dépôt se trouve dans le répertoire `/var/www/html/maven2`.

```

$ mvn archetype:crawl -Dcatalog=/var/www/html/archetype-catalog.xml \
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] -----
[INFO] Building Maven Default Project
[INFO] task-segment: [archetype:crawl] (aggregator-style)
[INFO] -----
[INFO] [archetype:crawl]
repository /Users/tobrien/tmp/maven2
catalogFile /Users/tobrien/tmp/blah.xml
[INFO] Scanning /Users/tobrien/tmp/maven2/com/discursive/cas/extend/cas-extend-client-java/2.1.1/cas-
[INFO] Scanning /Users/tobrien/tmp/maven2/com/discursive/cas/extend/cas-extend-client-java/2.2/cas-ex
-Drepository=/var/www/html/maven2
...

```

# Chapitre 19. Développement avec Flexmojos

## 19.1. Introduction

Ce chapitre propose un aperçu du projet Flexmojos pour les personnes intéressées par l'utilisation de Maven pour développer des applications et des bibliothèques Flex.

## 19.2. Configuration de l'environnement de build pour Flexmojos

Avant que vous ne tentiez de compiler des bibliothèques et des applications Flex avec Maven, vous devrez réaliser deux tâches de configuration :

- Configurez vos paramètres Maven afin de référencer un dépôt qui contienne le framework Flex
- Ajoutez Flash Player à votre PATH pour pouvoir exécuter les tests unitaires Flex
- (Optionnel) Configurez vos paramètres Maven pour inclure le groupe de plugin de Sonatype

### 19.2.1. Faire référence à un dépôt contenant le Framework Flex

Pour configurer votre environnement Maven pour utiliser Flexmojos, vous avez deux options : vous pouvez faire référence au dépôt Flexmojos de Sonatype directement dans le fichier `pom.xml`, ou vous pouvez installer Nexus et ajouter le dépôt Sonatype pour Flexmojos en tant que dépôt mandataire (proxy) dans votre propre gestionnaire de dépôts. Si l'option la plus simple consiste à faire référence directement au dépôt, télécharger et installer Nexus vous procurera le contrôle et la flexibilité dont vous avez besoin pour cacher et gérer les artefacts générés par vos propres builds. Si vous souhaitez juste commencer rapidement avec Flexmojos, lisez ci-dessous la Section 19.2.1.1, « Faire référence depuis le POM au dépôt Flexmojos de Sonatype ». Si ce qui vous intéresse est une solution à long terme qui peut être déployée pour servir de support à une équipe de développement, allez à la Section 19.2.1.2, « Utiliser Nexus comme dépôt mandataire de Flexmojos de Sonatype ».

Si votre organisation utilise déjà la solution Nexus de Sonatype en tant que mandataire pour les dépôts distants, vous avez probablement un fichier `~/.m2/settings.xml` personnalisé qui pointe vers un seul groupe Nexus. Si telle est votre situation, vous devrez ajouter au groupe référencé par votre équipe de développement un dépôt mandataire pour le groupe du dépôt de Flexmojos avec l'adresse `http://repository.sonatype.org/content/groups/flexgroup/`. Ajouter un dépôt mandataire pour ce groupe distant et puis ajouter ce groupe au groupe public de votre installation Nexus, donnera aux clients de votre instance Nexus un accès aux artefacts Sonatype `repository.sonatype.org`.

---

### 19.2.1.1. Faire référence depuis le POM au dépôt Flexmojos de Sonatype

Flexmojos dépend de certains artefacts qui ne sont actuellement pas disponibles depuis le dépôt central Maven. Ces artefacts sont disponibles depuis un dépôt hébergé par Sonatype. Pour utiliser Flexmojos, vous devrez faire référence à ce dépôt depuis le `pom.xml` de votre projet. Dans ce but, ajoutez l'élément `repositories` indiqué dans Exemple 19.1, « Ajouter une référence au dépôt Flexmojos de Sonatype au sein du POM » dans le fichier `pom.xml` de votre projet.

Exemple 19.1. Ajouter une référence au dépôt Flexmojos de Sonatype au sein du POM

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <groupId>test</groupId>
 <artifactId>test</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>pom</packaging>

 <modules>
 <module>swc</module>
 <module>swf</module>
 <module>war</module>
 </modules>

 <repositories>
 <repository>
 <id>flexmojos</id>
 <url>http://repository.sonatype.org/content/groups/flexgroup/</url>
 </repository>
 </repositories>
</project>
```

Le XML illustré dans Exemple 19.1, « Ajouter une référence au dépôt Flexmojos de Sonatype au sein du POM », ajoute ce dépôt à la liste des dépôts consultés par Maven quand il tente de télécharger les artefacts et les plugins.

### 19.2.1.2. Utiliser Nexus comme dépôt mandataire de Flexmojos de Sonatype

Au lieu de pointer directement vers le dépôt Flexmojos de Sonatype, Sonatype recommande d'installer un gestionnaire de dépôt et de le mettre en proxy du dépôt public de Sonatype. Quand vous mettez en place un proxy d'un dépôt distant avec un gestionnaire de dépôt comme Nexus, vous gagnez un niveau supplémentaire de contrôle et de stabilité qu'il n'est pas possible d'atteindre lorsque votre build dépend directement de ressources externes. En plus de ce contrôle et de cette stabilité, un gestionnaire de dépôt fournit aussi une cible pour le déploiement des artefacts binaires produits par vos builds. Les instructions pour télécharger, installer et configurer Nexus sont disponibles dans le *Installation chapter in Repository Management with Nexus*<sup>1</sup>. Une fois Nexus installé et démarré, effectuez les opérations suivantes pour lui ajouter une fonction de mandataire du dépôt public de Sonatype.

<sup>1</sup> <http://www.sonatype.com/books/nexus-book/reference/install.html>



Pour ajouter un nouveau proxy d'un dépôt, cliquez sur le lien Repositories sous Views/Repositories dans le menu Nexus de la partie gauche de l'interface utilisateur de Nexus. Ce click sur Repositories va charger le panneau Repositories. Dans ce panneau Repositories, cliquez sur le bouton Add.. et sélectionnez Proxy Repository comme le montre la Figure 19.1, « Ajout du proxy d'un dépôt sur Sonatype Nexus ».

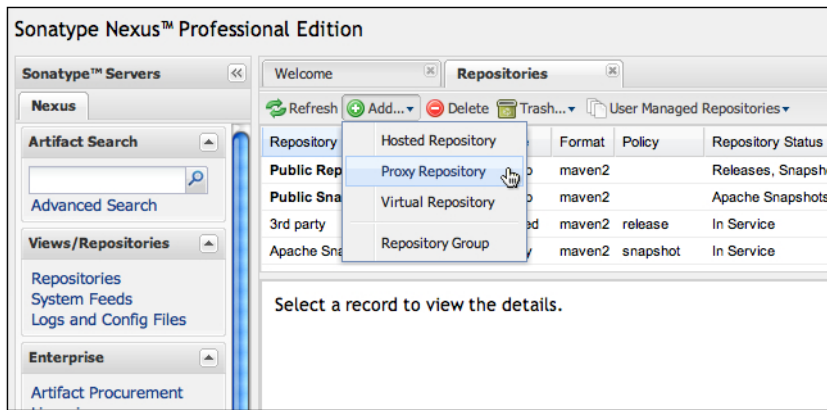


Figure 19.1. Ajout du proxy d'un dépôt sur Sonatype Nexus

Une fois que vous avez créé ce nouveau dépôt proxy, il vous faut le configurer pour qu'il pointe vers le dépôt public Flexmojos de Sonatype. Sélectionnez ce nouveau dépôt puis l'onglet de Configuration en bas de la fenêtre. Remplissez les champs suivants avec les valeurs indiquées comme le montre la Figure 19.2, « Configuration du dépôt Sonatype Flexmojos Proxy ».

- Repository ID prend pour valeur "sonatype-flexmojos"
- Repository Name prend pour valeur "Sonatype Flexmojos Proxy"
- La "Remote Storage Location" prend pour valeur <http://repository.sonatype.org/content/groups/flexgroup/>

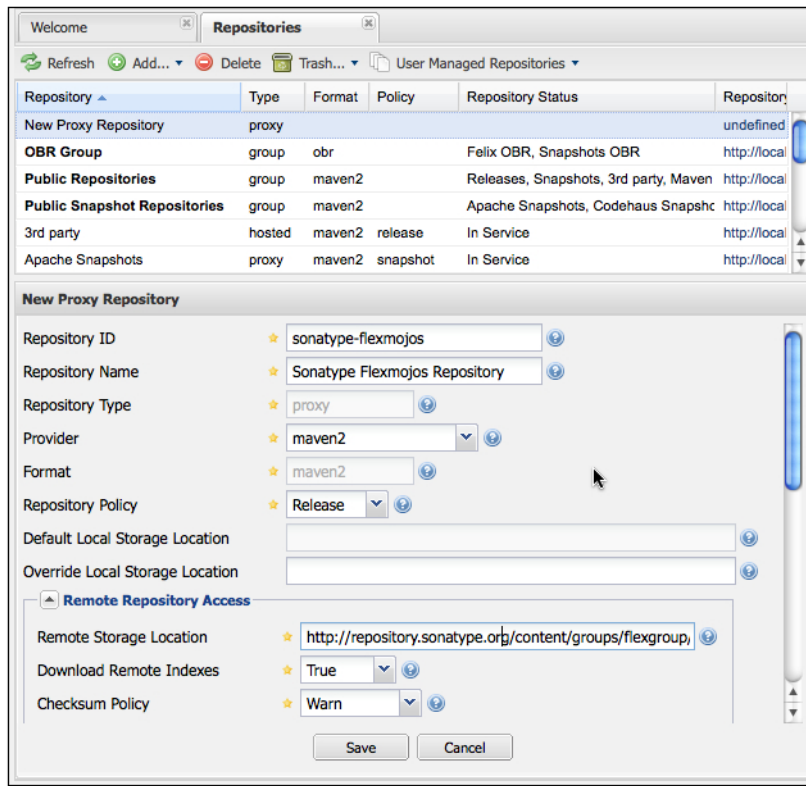


Figure 19.2. Configuration du dépôt Sonatype Flexmojos Proxy

Une fois que vous avez rempli les champs présentés dans Figure 19.2, « Configuration du dépôt Sonatype Flexmojos Proxy » cliquez sur le bouton Save pour enregistrer ce dépôt et commencer à l'utiliser comme proxy du dépôt Sonatype Flexmojos. Nexus est fourni avec un groupe public de dépôts, qui regroupe plusieurs dépôts en un seul point d'entrée pour les clients Maven. Pour compléter la configuration de ce nouveau dépôt mandataire, vous devez ajouter celui-ci au groupe Nexus `Public Repositories`. Pour ce faire, allez à la liste des dépôts qui devrait être visible dans le haut du panneau `Repositories` comme le montre Figure 19.2, « Configuration du dépôt Sonatype Flexmojos Proxy ». Cliquez sur le groupe `Public Repositories` puis sur l'onglet `Configuration` en bas du panneau `Repository`. Cliquer sur l'onglet `Configuration` fait apparaître le formulaire `Group configuration` comme le montre Figure 19.3, « Ajout du proxy de Sonatype Flexmojos au groupe `Public Repositories` ».

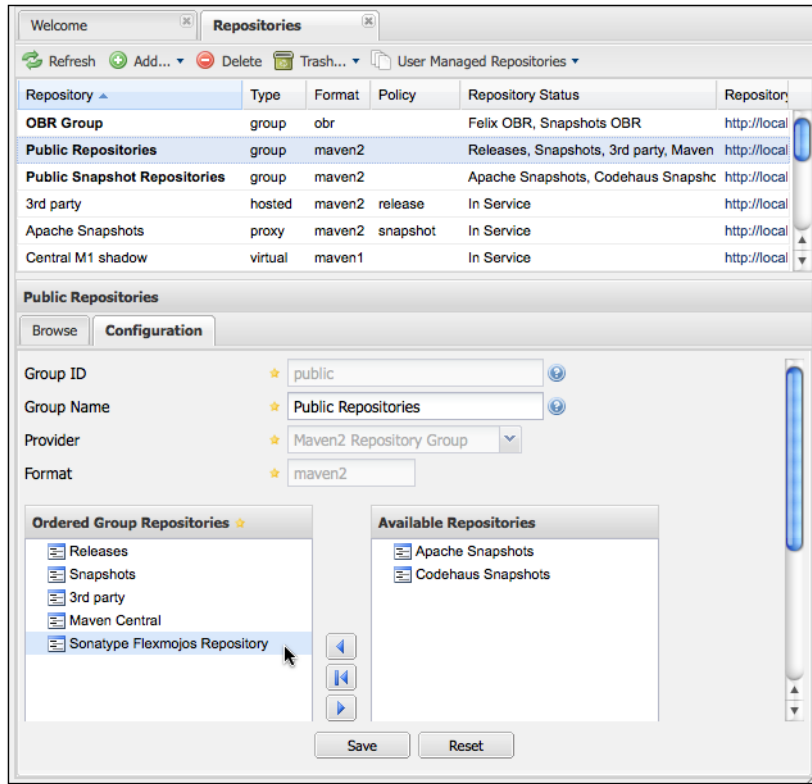


Figure 19.3. Ajout du proxy de Sonatype Flexmojos au groupe **Public Repositories**

Pour ajouter le Sonatype Public Proxy au groupe Public Repositories faites un glisser/déposer du dépôt Sonatype Public Proxy de la liste Available Repositories dans la liste Ordered Group Repositories. Cliquez sur le bouton Save et vous aurez ajouté avec succès un proxy du dépôt Sonatype Flexmojos à votre installation de Nexus. À chaque fois qu'un client va demander un artefact à ce groupe de dépôt, si Nexus ne l'a pas déjà mis en cache il va le demander au dépôt Sonatype Flexmojos situé à <http://repository.sonatype.org/content/groups/flexgroup/>. Votre installation de Nexus maintient un cache local de tous les artefacts récupérés depuis le dépôt Sonatype Flexmojos. Ce cache local vous donne plus de contrôle et rend l'environnement de build plus stable. Si vous montez une équipe de développeurs qui dépend des artefacts du dépôt public de Sonatype, vous aurez ainsi un environnement de build autonome qui ne dépendra pas de la disponibilité du dépôt de Sonatype une fois que les artefacts nécessaires auront été mis en cache par votre instance de Nexus.

La dernière étape consiste à faire pointer votre installation de Maven vers l'instance de Nexus que vous venez de configurer. Vous devez donc modifier votre configuration Maven pour qu'il utilise le groupe du dépôt Nexus comme miroir pour tous les dépôts. Pour cela, vous devez éditer votre fichier `~/ .m2/ settings.xml` et lui ajouter le XML suivant.

## Exemple 19.2. Configuration de l'instance Nexus dans le fichier settings.xml

```
<settings>
 <mirrors>
 <mirror>
 <!--This sends everything else to /public -->
 <id>nexus</id>
 <mirrorOf>*</mirrorOf>
 <url>http://localhost:8081/nexus/content/groups/public</url>
 </mirror>
 </mirrors>
 <profiles>
 <profile>
 <id>nexus</id>
 <!--Enable snapshots for the built in central repo to direct -->
 <!--all requests to nexus via the mirror -->
 <repositories>
 <repository>
 <id>central</id>
 <url>http://central</url>
 <releases><enabled>true</enabled></releases>
 <snapshots><enabled>true</enabled></snapshots>
 </repository>
 </repositories>
 <pluginRepositories>
 <pluginRepository>
 <id>central</id>
 <url>http://central</url>
 <releases><enabled>true</enabled></releases>
 <snapshots><enabled>true</enabled></snapshots>
 </pluginRepository>
 </pluginRepositories>
 </profile>
 </profiles>
 <activeProfiles>
 <!--make the profile active all the time -->
 <activeProfile>nexus</activeProfile>
 </activeProfiles>
</settings>
```

Ce fichier XML configure Maven pour qu'il aille consulter le groupe de dépôts publics plutôt que les dépôts d'artefacts et de plugins configurés par ailleurs. C'est une façon simple de garantir que toute demande d'artefact passe par le Nexus installé.

### 19.2.2. Configuration de l'environnement pour les tests Flex Unit

Flexmojos s'attend à pouvoir lancer le lecteur autonome Flash Play pour l'exécution des tests unitaires. Pour que cela soit opérationnel, il vous faudra ou bien ajouter le lecteur Flash Player à votre PATH, or alors indiquer la localisation de l'exécutable Flash Player lors du build en utilisant l'option `-DflashPlayer.command`. Au moment de l'exécution d'un test unitaire, Flex Mojos s'attend à pouvoir lancer l'exécutable Flash Player en fonction de la plateforme :

## Microsoft Windows

FlexMojos tentera de lancer le binaire `FlashPlayer.exe`. Pour permettre l'exécution des tests unitaires, ajoutez à votre PATH le répertoire contenant `FlashPlayer.exe`, ou alors précisez la localisation du binaire `FlashPlayer.exe` via l'option de ligne de commande Maven `-DflashPlayer.command=${filepath}`.

## Macintosh OSX

FlexMojos tentera de lancer l'application "Flash Player". Pour permettre l'exécution des tests unitaires, ajoutez le répertoire contenant "Flash Player" à votre PATH, ou alors précisez le chemin de l'exécutable via l'option de ligne de commande Maven `-DflashPlayer.command=${filepath}`.

## Unix (Linux, Solaris, etc.)

FlexMojos tentera de lancer l'exécutable `flashplayer`. Pour permettre l'exécution des tests unitaires, ajoutez le répertoire contenant `flashplayer` à votre PATH, ou alors précisez le chemin de l'exécutable via l'option de ligne de commande Maven `-DflashPlayer.command=${filepath}`.



## Note

Sur une machine Linux, il vous faudra installer un serveur X Virtual Frame Buffer (Xvfb) pour pouvoir lancer les tests unitaires sans interface graphique. Pour plus d'information, suivez le lien [Xvfb<sup>2</sup>](#).

Si vous avez déjà développé des applications Flash avec Adobe Flash CS4, Adobe Flex Builder ou si vous visionnez du contenu flash dans un navigateur, alors il est probable que Flash Player soit installé sur votre station de travail. Bien qu'il soit possible de configurer Maven pour qu'il utilise l'un de ces players durant la campagne de tests unitaires Flex, vous préférerez vous assurer que vous lancez bien la version debug de Flash Player. Pour minimiser les risques d'incompatibilité, vous devriez télécharger l'un des Flash Players listés ci-dessous et l'installer sur votre station de travail. Pour télécharger Flash Player selon l'environnement :

- Windows : [http://download.macromedia.com/pub/flashplayer/updaters/10/flashplayer\\_10\\_sa\\_debug.exe](http://download.macromedia.com/pub/flashplayer/updaters/10/flashplayer_10_sa_debug.exe)<sup>3</sup>
- Mac OSX : [http://download.macromedia.com/pub/flashplayer/updaters/10/flashplayer\\_10\\_sa\\_debug.app.zip](http://download.macromedia.com/pub/flashplayer/updaters/10/flashplayer_10_sa_debug.app.zip)<sup>4</sup>
- Linux : [http://download.macromedia.com/pub/flashplayer/updaters/10/flash\\_player\\_10\\_linux\\_dev.tar.gz](http://download.macromedia.com/pub/flashplayer/updaters/10/flash_player_10_linux_dev.tar.gz)<sup>5</sup>

Lancez la commande suivante pour installer ce player et l'ajouter à votre PATH sur une machine OSX :

```
$ wget http://download.macromedia.com/pub/flashplayer/updaters/10/\
flashplayer_10_sa_debug.app.zip
```

```
$ unzip flashplayer_10_sa_debug.app.zip
$ sudo cp -r Flash\ Player.app /Applications/
$ export PATH=/Applications/Flash\ Player.app/Contents/MacOS:${PATH}
```

Plutôt que d'ajouter en ligne de commande le chemin de Flash Player à votre PATH, vous devriez configurer votre environnement. Sur OSX, il vous suffit d'ajouter cette dernière commande d'export à votre fichier `~/.bash_profile`.

### 19.2.3. Ajouter FlexMojos aux groupes de plugins de votre configuration Maven

Si vous exécutez des goals FlexMojos en ligne de commande, vous trouverez qu'il est plus pratique d'ajouter les groupes de plugins Sonatype à votre configuration Maven. Pour cela, éditez le fichier `~/.m2/settings.xml` et ajoutez les groupes de plugins suivants :

Exemple 19.3. Ajouter les plugins Sonatype à votre configuration Maven

```
<pluginGroups>
 <pluginGroup>com.sonatype.maven.plugins</pluginGroup>
 <pluginGroup>org.sonatype.plugins</pluginGroup>
</pluginGroups>
```

Une fois que vous avez ajouté ces groupes de plugins à votre configuration Maven, vous pouvez invoquer les goals FlexMojos en utilisant le préfixe de plugin `flexmojos`. Sans cette configuration, invocation du goal `flexbuilder` nécessite la ligne de commande suivante :

```
$ mvn org.sonatype.flexmojos:flexmojos-maven-plugin:3.2.0:flexbuilder
```

Avec le groupe `org.sonatype.plugins` dans votre configuration Maven, le même goal peut être invoqué comme suit :

```
$ mvn flexmojos:flexbuilder
```

## 19.3. Création d'un projet FlexMojos à partir d'un archétype

Flexmojos a un ensemble d'archétypes qui peuvent être utilisés rapidement pour créer un nouveau projet Flex. Les archétypes ci-dessous sont tous dans le groupe `org.sonatype.flexmojos` avec version `3.3.0` :

`flexmojos-archetypes-library`

Crée un simple projet de bibliothèque Flex qui produit un artefact SWC

`flexmojos-archetypes-application`

Crée un simple projet Flex qui produit un artefact SWF

flexmojos-archetypes-modular-webapp

Crée un projet multimodule qui présente un projet qui produit un SWC qui est consommé par un projet pour produire un SWF lequel est affiché dans uen application web produite par un dernier projet.

### 19.3.1. Création d'une bibliothèque Flex

Pour créer un projet de bibliothèque Flex, il suffit d'exécuter la ligne de commande suivante :

```
$ mvn archetype:generate \
-DarchetypeRepository=http://repository.sonatype.org/content/groups/public \
-DarchetypeGroupId=org.sonatype.flexmojos \
-DarchetypeArtifactId=flexmojos-archetypes-library \
-DarchetypeVersion=3.3.0
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] com.sonatype.maven.plugins: checking for updates from central
...
[INFO] [archetype:generate]
[INFO] Generating project in Interactive mode
[INFO] Archetype defined by properties
...
Define value for groupId: : org.sonatype.test
Define value for artifactId: : sample-library
Define value for version: 1.0-SNAPSHOT: : 1.0-SNAPSHOT
Define value for package: org.sonatype.test: : org.sonatype.test
Confirm properties configuration:
groupId: org.sonatype.test
artifactId: sample-library
version: 1.0-SNAPSHOT
package: org.sonatype.test
Y: : Y[INFO] Parameter: groupId, Value: org.sonatype.test
[INFO] Parameter: packageName, Value: org.sonatype.test
[INFO] Parameter: basedir, Value: /Users/Tim
[INFO] Parameter: package, Value: org.sonatype.test
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: sample-library
[INFO] -----
[INFO] BUILD SUCCESSFUL
```

En regardant dans le répertoire `sample-library/`, vous verrez que le projet présente une arborescence comme dans la Figure 19.4, « Arborescence de l'archétype de bibliothèque Flexmojo ».

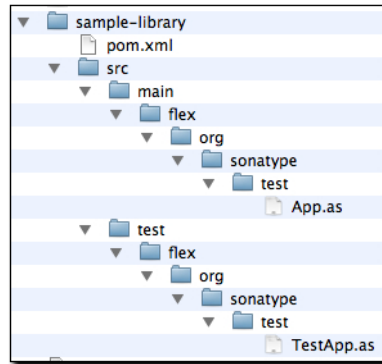


Figure 19.4. Arborescence de l'archétype de bibliothèque Flexmojo

Le produit du seul archétype de bibliothèque Flex contient trois fichiers : un POM, un fichier source et un fichier de test unitaire. Regardons chacun de ces trois fichiers. Premièrement, le Project Object Model (POM).

#### Exemple 19.4. POM d'un archétype pour projet de bibliothèque Flex

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/maven-v4_0_0.xsd">

 <modelVersion>4.0.0</modelVersion>

 <groupId>org.sonatype.test</groupId>
 <artifactId>sample-library</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>swc</packaging>

 <name>test Flex</name>

 <build>
 <sourceDirectory>src/main/flex</sourceDirectory>
 <testSourceDirectory>src/test/flex</testSourceDirectory>
 <plugins>
 <plugin>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-maven-plugin</artifactId>
 <version>3.3.0</version>
 <extensions>true</extensions>
 </plugin>
 </plugins>
 </build>

 <dependencies>
 <dependency>
 <groupId>com.adobe.flex.framework</groupId>
 <artifactId>flex-framework</artifactId>
```



```

 <version>3.2.0.3958</version>
 <type>pom</type>
 </dependency>

 <!-- flexmojos Unit testing support -->
 <dependency>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-unittest-support</artifactId>
 <version>3.3.0</version>
 <type>swc</type>
 <scope>test</scope>
 </dependency>
</dependencies>

</project>

```

L'Exemple 19.4, « POM d'un archétype pour projet de bibliothèque Flex » est très simple, le point clé de ce POM c'est la configuration de `flexmojos-maven-plugin` qui positionne `extensions` à `true`. Cette configuration personnalise le cycle de vie pour le packaging `swc` qui est défini dans le plugin `flexmojos-maven-plugin`. L'archétype inclut alors la dépendance `flex-framework` ainsi que la dépendance de type `test` `flexmojos-unittest-support`. La dépendance `flex-framework` est un POM qui contient les références vers les bibliothèques et les ressources nécessaires à la compilation une application Flex.

Dans l'Exemple 19.4, « POM d'un archétype pour projet de bibliothèque Flex », le packaging est primordial. Le type de packaging d'un POM contrôle le cycle de vie qui est utilisé pour produire le résultat de build. La valeur `swc` dans l'élément `packaging` est l'indice qui indique le cycle de vie spécifique Flex dont les spécificités sont fournies par le plugin `flexmojos-maven-plugin`. L'autre partie importante de ce POM est l'élément `build` qui précise l'emplacement du code source Flex ainsi que celui des tests unitaires. Maintenant, jetons un coup d'oeil à l'Exemple 19.5, « App, l'application exemple de l'archétype de bibliothèque Flex » qui contient l'exemple de code d'ActionScript créé par l'archétype.

Exemple 19.5. App, l'application exemple de l'archétype de bibliothèque Flex

```

package org.sonatype.test {
 public class App {
 public static function greeting(name:String):String {
 return "Hello, " + name;
 }
 }
}

```

Bien que ce code soit très simple, il apporte un exemple et un repère immédiat : "Ecrire le reste du code ici". Bien qu'il puisse sembler idiot de tester un code aussi simple, un exemple de test nommé `TestApp.as` est proposé dans le répertoire `src/test/flex`. Ce test est présenté dans l'Exemple 19.6, « Test unitaire de la classe App pour l'archétype de bibliothèque Flex ».

Exemple 19.6. Test unitaire de la classe App pour l'archétype de bibliothèque Flex

```

package org.sonatype.test {

```

```

import flexunit.framework.TestCase;

public class TestApp extends TestCase {

 /**
 * Tests our greeting() method
 */
 public function testGreeting():void {
 var name:String = "Buck Rogers";
 var expectedGreeting:String = "Hello, Buck Rogers";

 var result:String = App.greeting(name);
 assertEquals("Greeting is incorrect", expectedGreeting, result);
 }
}

```

Pour exécuter ce build, allez au répertoire du projet `sample-library` et lancez la commande : `run mvn install`.

```

$ mvn install
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building sample-library Flex
[INFO] task-segment: [install]
[INFO] -----
[INFO] [resources:resources]
[INFO] [flexmojos:compile-swc]
[INFO] flexmojos 3.3.0 - GNU GPL License (NO WARRANTY) - \
See COPYRIGHT file
[WARNING] Nothing expecified to include. Assuming source and resources folders.
[INFO] Flex compiler configurations:
-compiler.headless-server=false
-compiler.keep-all-type-selectors=false
-compiler.keep-generated-actionscript=false
-compiler.library-path ~/.m2/repository/com/adobe/flex/framework/flex/\
3.2.0.3958...
-compiler.namespaces.namespace http://www.adobe.com/2006/mxml
 target/classes/configs/mxml-manifest.xml
-compiler.optimize=true
-compiler.source-path src/main/flex
...
[INFO] [resources:testResources]
[WARNING] Using platform encoding (MacRoman actually) to copy filtered \
resources, i.e. build is platform dependent!
[INFO] skip non existing resourceDirectory src/test/resources
[INFO] [flexmojos:test-compile]
[INFO] flexmojos 3.3.0 - GNU GPL License (NO WARRANTY) - \
See COPYRIGHT file
[INFO] Flex compiler configurations:
-compiler.include-libraries ~/.m2/repository/org/sonatype/flexmojos/\
flexmojos-unittest-support...
-compiler.keep-generated-actionscript=false

```

```

-compiler.library-path ~/.m2/repository/com/adobe/flex/framework/flex
 3.2.0.3958/flex-3.2.0....
-compiler.optimize=true
-compiler.source-path src/main/flex target/test-classes src/test/flex
-compiler.strict=true
-target-player 9.0.0
-use-network=true
-verify-digests=true -load-config=
[INFO] Already trust on target/test-classes/TestRunner.swf
[INFO] [flexmojos:test-run]
[INFO] flexmojos 3.3.0 - GNU GPL License (NO WARRANTY) - \
See COPYRIGHT file
[INFO] flexunit setup args: null
[INFO] -----
[INFO] Tests run: 1, Failures: 0, Errors: 0, Time Elapsed: 0 sec
[INFO] [install:install]

```



## Note

Pour pouvoir exécuter les tests unitaires Flex il vous faudra configurer votre variable d'environnement PATH afin d'inclure le lecteur Flash Player. Pour plus d'information concernant la configuration de FlexMojos pour les tests unitaires, se référer à la Section 19.2.2, « Configuration de l'environnement pour les tests Flex Unit ».

Quand vous exécutez la commande `mvn install` pour ce projet, vous pouvez noter dans l'output que Maven et le plugin Flexmojos prennent en charge la gestion de toutes les bibliothèques et dépendances pour le compilateur Flex. De la même façon que Maven est excellent pour aider les développeurs Java à gérer le contenu d'un classpath Java, Maven peut aider les développeurs Flex à gérer la complexité de génération des paths. Vous avez peut-être été surpris quand votre projet Flexmojos a démarré un navigateur web ou un lecteur Flash Player et l'a utilisé pour exécuter l'application TestApp construite à partir de votre code source.

## 19.3.2. Création d'une application Flex

Pour créer une application Flex à partir d'un archétype Maven, lancer la commande suivante :

```

$ mvn archetype:generate \
 -DarchetypeRepository=http://repository.sonatype.org/content/groups/public \
 -DarchetypeGroupId=org.sonatype.flexmojos \
 -DarchetypeArtifactId=flexmojos-archetypes-application \
 -DarchetypeVersion=3.3.0
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] com.sonatype.maven.plugins: checking for updates from central
...
[INFO] [archetype:generate]
[INFO] Generating project in Interactive mode
[INFO] Archetype defined by properties
...
Define value for groupId: : org.sonatype.test

```

```

Define value for artifactId: : sample-application
Define value for version: 1.0-SNAPSHOT: : 1.0-SNAPSHOT
Define value for package: org.sonatype.test: : org.sonatype.test
Confirm properties configuration:
groupId: org.sonatype.test
artifactId: sample-library
version: 1.0-SNAPSHOT
package: org.sonatype.test
Y: : Y
[INFO] Parameter: groupId, Value: org.sonatype.test
[INFO] Parameter: packageName, Value: org.sonatype.test
[INFO] Parameter: basedir, Value: /Users/Tim/flex-sample
[INFO] Parameter: package, Value: org.sonatype.test
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: sample-application
[INFO] BUILD SUCCESSFUL

```

Si vous regardez dans le répertoire `sample-application/` vous verrez l'arborescence de fichiers illustrée dans la Figure 19.5, « Structure de fichiers issue de l'archétype Application Flex ».

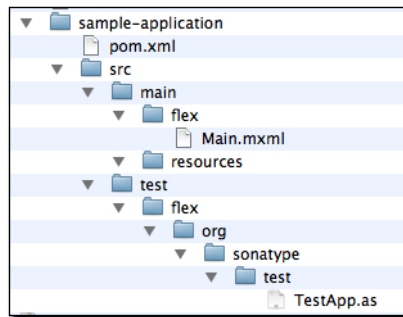


Figure 19.5. Structure de fichiers issue de l'archétype Application Flex

La génération d'une application Flex via l'archétype produit le POM suivant.

#### Exemple 19.7. POM généré par l'archétype Application Flex

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/maven-v4_0_0.xsd">

 <modelVersion>4.0.0</modelVersion>

 <groupId>org.sonatype.test</groupId>
 <artifactId>sample-application</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>swf</packaging>

 <name>sample-application Flex</name>

```

```

<build>
 <sourceDirectory>src/main/flex</sourceDirectory>
 <testSourceDirectory>src/test/flex</testSourceDirectory>
 <plugins>
 <plugin>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-maven-plugin</artifactId>
 <version>3.3.0</version>
 <extensions>>true</extensions>
 </plugin>
 </plugins>
</build>

<dependencies>
 <dependency>
 <groupId>com.adobe.flex.framework</groupId>
 <artifactId>flex-framework</artifactId>
 <version>3.2.0.3958</version>
 <type>pom</type>
 </dependency>

 <!-- flexmojos Unit testing support -->
 <dependency>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-unittest-support</artifactId>
 <version>3.3.0</version>
 <type>swc</type>
 <scope>test</scope>
 </dependency>
</dependencies>

</project>

```

La différence entre l'Exemple 19.7, « POM généré par l'archétype Application Flex » et l'Exemple 19.4, « POM d'un archétype pour projet de bibliothèque Flex » se trouve dans le type de packaging : `swf` plutôt que `swc`. En positionnant le packaging à `swf`, le projet produira une application Flex, à savoir l'application `target/sample-application-1.0-SNAPSHOT.swf`. L'application exemple produite par cet archétype affichera le texte "Hello World". Le fichier source `Main.mxml` se trouve dans `src/main/flex`.

#### Exemple 19.8. Application exemple `Main.mxml`

```

<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute">
 <mx:Text text="Hello World!"/>
</mx:Application>

```

L'archétype produit également un test unitaire `FlexUnit` simple qui ne fait rien d'autre que d'afficher un message de trace. L'exemple de test unitaire se trouve dans `src/test/flex/org/sonatype/test`.

#### Exemple 19.9. Test unitaire de `Main.mxml`

```

package org.sonatype.test

```

```

{

 import flexunit.framework.TestCase;
 import Main;

 public class TestApp extends TestCase
 {

 public function testNothing():void
 {

 //TODO un implemented
 trace("Hello test");

 }

 }

}

```

### 19.3.3. Creation d'un projet multimodule : Une application web avec une dépendance Flex

La ligne de commande suivante crée un projet multimodule contenant un projet de bibliothèque Flex référencée par une application Flex qui elle-même est référencée par une application web :

```

$ mvn archetype:generate \
 -DarchetypeRepository=http://repository.sonatype.org/content/groups/public \
 -DarchetypeGroupId=org.sonatype.flexmojos \
 -DarchetypeArtifactId=flexmojos-archetypes-modular-webapp \
 -DarchetypeVersion=3.3.0
[INFO] Scanning for projects...
[INFO] Searching repository for plugin with prefix: 'archetype'.
[INFO] com.sonatype.maven.plugins: checking for updates from central
...
[INFO] [archetype:generate]
[INFO] Generating project in Interactive mode
[INFO] Archetype defined by properties
...
Define value for groupId: : org.sonatype.test
Define value for artifactId: : sample-multimodule
Define value for version: 1.0-SNAPSHOT: : 1.0-SNAPSHOT
Define value for package: org.sonatype.test: : org.sonatype.test
Confirm properties configuration:
groupId: org.sonatype.test
artifactId: sample-library
version: 1.0-SNAPSHOT
package: org.sonatype.test
Y: : Y
[INFO] Parameter: groupId, Value: org.sonatype.test
[INFO] Parameter: packageName, Value: org.sonatype.test
[INFO] Parameter: basedir, Value: /Users/Tim
[INFO] Parameter: package, Value: org.sonatype.test
[INFO] Parameter: version, Value: 1.0-SNAPSHOT
[INFO] Parameter: artifactId, Value: sample-multimodule
[INFO] -----

```

[INFO] BUILD SUCCESSFUL

Si vous regardez dans le répertoire `sample-multimodule/`, vous verrez une arborescence qui contient trois projets `swc`, `swf` et `war`.

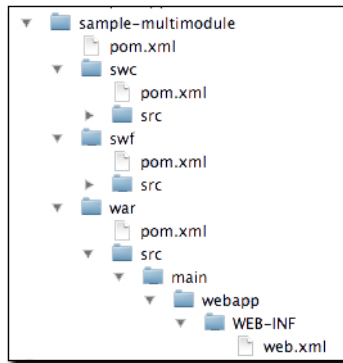


Figure 19.6. Arborescence de fichiers issue de l'archétype multimodule Flex

Le POM parent du projet multimodule est simple comme vous pouvez le voir ci-dessous. Il est constitué de références aux modules `swc`, `swf` et `war`.

#### Exemple 19.10. POM parent produit par l'archétype multimodule Flex

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/maven-v4_0_0.xsd">
 <modelVersion>4.0.0</modelVersion>
 <groupId>org.sonatype.test</groupId>
 <artifactId>sample-multimodule</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>pom</packaging>

 <modules>
 <module>swc</module>
 <module>swf</module>
 <module>war</module>
 </modules>
</project>
```

Le projet `swc` a un POM simple qui ressemble au POM illustré dans l'Exemple 19.4, « POM d'un archétype pour projet de bibliothèque Flex ». Notez que l'`artifactId` de ce POM ne reflète pas le nom du module mais est `swc-sw-c`.

#### Exemple 19.11. POM du module `swc`

```
<project>

 <modelVersion>4.0.0</modelVersion>
```

```

<parent>
 <groupId>org.sonatype.test</groupId>
 <artifactId>sample-multimodule</artifactId>
 <version>1.0-SNAPSHOT</version>
</parent>

<groupId>org.sonatype.test</groupId>
<artifactId>swc-swc</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>swc</packaging>
<name>swc Library</name>
<build>
 <sourceDirectory>src/main/flex</sourceDirectory>
 <testSourceDirectory>src/test/flex</testSourceDirectory>
 <plugins>
 <plugin>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-maven-plugin</artifactId>
 <version>3.3.0</version>
 <extensions>>true</extensions>
 <configuration>
 <locales>
 <locale>en_US</locale>
 </locales>
 </configuration>
 </plugin>
 </plugins>
</build>

<dependencies>
 <dependency>
 <groupId>com.adobe.flex.framework</groupId>
 <artifactId>flex-framework</artifactId>
 <version>3.2.0.3958</version>
 <type>pom</type>
 </dependency>

 <!-- flexmojos Unit testing support -->
 <dependency>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-unittest-support</artifactId>
 <version>3.3.0</version>
 <type>swc</type>
 <scope>test</scope>
 </dependency>
</dependencies>

</project>

```

Le POM du module swf est semblable au POM de l'Exemple 19.7, « POM généré par l'archétype Application Flex » avec une dépendance vers l'artefact `swc-swc` en plus. Notez que l'`artifactId` de ce POM ne reflète pas le nom du répertoire qui contient le module ; l'`artifactId` dans le POM ci-dessous est `swf-swf`.



## Exemple 19.12. POM du module swf

```
<project>

 <modelVersion>4.0.0</modelVersion>

 <parent>
 <groupId>org.sonatype.test</groupId>
 <artifactId>sample-multimodule</artifactId>
 <version>1.0-SNAPSHOT</version>
 </parent>

 <groupId>org.sonatype.test</groupId>

 <artifactId>swf-swf</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>swf</packaging>
 <name>swf Application</name>

 <build>
 <sourceDirectory>src/main/flex</sourceDirectory>
 <testSourceDirectory>src/test/flex</testSourceDirectory>
 <plugins>
 <plugin>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-maven-plugin</artifactId>
 <version>3.3.0</version>
 <extensions>true</extensions>
 <configuration>
 <locales>
 <locale>en_US</locale>
 </locales>
 </configuration>
 </plugin>
 </plugins>
 </build>

 <dependencies>
 <dependency>
 <groupId>com.adobe.flex.framework</groupId>
 <artifactId>flex-framework</artifactId>
 <version>3.2.0.3958</version>
 <type>pom</type>
 </dependency>

 <!-- flexmojos Unit testing support -->
 <dependency>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-unittest-support</artifactId>
 <version>3.3.0</version>
 <type>swc</type>
 <scope>test</scope>
 </dependency>

 <dependency>
```

```
<groupId>org.sonatype.test</groupId>
<artifactId>swf-swc</artifactId>
<version>1.0-SNAPSHOT</version>
<type>swc</type>
</dependency>
</dependencies>
</project>
```



## Avertissement

Dans l'Exemple 19.12, « POM du module swf », la dépendance vers "swf-swc" doit être changée en "swc-swc". C'est un bug de l'archétype multimodule Flex qui est présent dans la version 3.3.0 de FlexMojos. Il sera corrigé dans la version FlexMojos 3.2.0.

Quand vous déclarez une dépendance vers un SWC, vous devez préciser le type de dépendance afin que Maven puisse localiser les artefacts adéquats dans le dépôt distant ou local. Dans ce cas, le projet `swf-swf` dépend du SWC produit par le projet `swc-swc`. Quand vous ajoutez la dépendance au projet `swf-swf`, le plugin FlexMojos ajoutera le fichier SWC approprié dans le chemin des bibliothèques de compilation Flex.

Maintenant, jetez un coup d'oeil dans le POM simple du module war.

### Exemple 19.13. POM du module war

```
<project>
 <modelVersion>4.0.0</modelVersion>
 <parent>
 <artifactId>sample-multimodule</artifactId>
 <groupId>org.sonatype.test</groupId>
 <version>1.0-SNAPSHOT</version>
 </parent>
 <groupId>org.sonatype.test</groupId>
 <artifactId>war-war</artifactId>
 <version>1.0-SNAPSHOT</version>
 <packaging>war</packaging>
 <build>
 <plugins>
 <plugin>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-maven-plugin</artifactId>
 <version>3.3.0</version>
 <executions>
 <execution>
 <goals>
 <goal>copy-flex-resources</goal>
 </goals>
 </execution>
 </executions>
 </plugin>
 <plugin>
 <groupId>org.mortbay.jetty</groupId>
```

```
<artifactId>maven-jetty-plugin</artifactId>
</plugin>
</plugins>
</build>
<dependencies>
<dependency>
<groupId>org.sonatype.test</groupId>
<artifactId>war-swf</artifactId>
<version>1.0-SNAPSHOT</version>
<type>swf</type>
</dependency>
</dependencies>
</project>
```



## Avertissement

Dans l'Exemple 19.13, « POM du module war », la dépendance vers "war-swf" doit être remplacée par "swf-swf". C'est un bug dans l'archétype multimodule Flex qui est présent dans la version 3.3.0 de FlexMojos. Il sera corrigé dans la version FlexMojos 3.2.0.

Le POM illustré dans l'Exemple 19.13, « POM du module war » configure le plugin FlexMojos pour exécuter le goal `copy-flex-resources` pour ce projet. Le goal `copy-flex-resources` copiera l'application SWF à la racine de l'application web. Dans ce projet, l'exécution du build et la production du WAR copieront le fichier `swf-swf-1.0-SNAPSHOT.swf` dans le répertoire racine de l'application web `target/war-war-1.0-SNAPSHOT`.

Pour construire l'application web multimodule, exécutez `mvn install` depuis le répertoire racine. Ceci doit générer les artefacts `swc-swc`, `swf-swf` et `war-war` et produire le fichier WAR `/target/war-war-1.0-SNAPSHOT.war` qui contient le `swf-swf-1.0-SNAPSHOT.swf` à la racine de l'application web.



## Note

Pour pouvoir exécuter les tests unitaires Flex, il vous faudra configurer votre variable d'environnement `PATH` afin d'inclure le lecteur Flash Player. Pour plus d'information concernant la configuration de FlexMojos pour les tests unitaires, se référer à la Section 19.2.2, « Configuration de l'environnement pour les tests Flex Unit ».

## 19.4. Le cycle de vie de FlexMojos

Le plugin Maven FlexMojos personnalise le cycle de vie en se basant sur le type de packaging. Si votre projet a un `packaging` de type `swc` ou `swf`, alors le plugin FlexMojos exécute un cycle de vie personnalisé, si toutefois votre configuration de plugin positionne les extensions à `true`. L'Exemple 19.14, « Configuration de l'élément extensions à true pour un cycle de vie personnalisé Flex » illustre la configuration de plugin pour le `flexmojos-maven-plugin` avec la balise `extensions` positionnée à `true`.

## Exemple 19.14. Configuration de l'élément extensions à true pour un cycle de vie personnalisé Flex

```
<build>
 <sourceDirectory>src/main/flex</sourceDirectory>
 <testSourceDirectory>src/test/flex</testSourceDirectory>
 <plugins>
 <plugin>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-maven-plugin</artifactId>
 <version>3.3.0</version>
 <extensions>true</extensions>
 <configuration>
 <locales>
 <locale>en_US</locale>
 </locales>
 </configuration>
 </plugin>
 </plugins>
</build>
```

### 19.4.1. Le cycle de vie SWC

Quand le packaging est du type `swc`, FlexMojos exécute le cycle de vie illustré dans la Figure 19.7, « Le cycle de vie SWC de FlexMojos ». Les goals surlignés sont des goals spécifiques au plugin FlexMojos, ceux qui ne le sont pas sont des goals standards du plugin Core Maven.

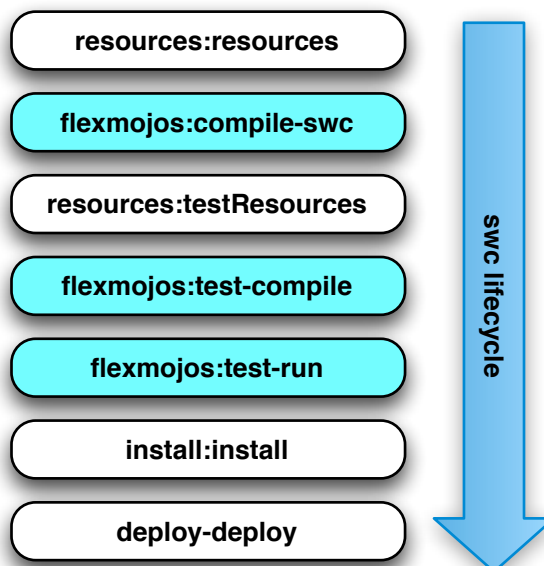


Figure 19.7. Le cycle de vie SWC de FlexMojos

Les goals FlexMojos participants sont les suivants :

`flexmojos:compile-swc`

Ce goal compile en une bibliothèque SWC tous les fichiers ActionScript et MXML présents dans le répertoire `sourceDirectory`. Un fichier SWC est une bibliothèque Adobe qui contient des composants et des ressources utilisées dans des applications Flex.

`flexmojos:test-compile`

Ce goal compile tous les fichiers ActionScript et MXML qui se trouvent dans le répertoire `testSourceDirectory`.

`flexmojos:test-run`

Ce goal exécute les tests unitaires en utilisant le lecteur Flash Player. Ce goal ne peut s'exécuter que si le Flash Player a été correctement configuré comme décrit dans la Section 19.2.2, « Configuration de l'environnement pour les tests Flex Unit ».

## 19.4.2. Le cycle de vie SWF

Quand le type de packaging est `swf`, FlexMojos exécute le cycle de vie illustré dans la Figure 19.8, « Cycle de vie SWF de FlexMojos ». Les goals surlignés sont spécifiques au plugin FlexMojos, ceux qui ne le sont pas sont des goals standards du plugin Core Maven.

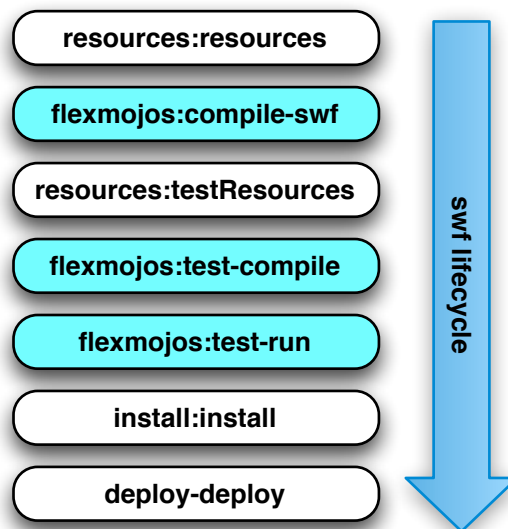


Figure 19.8. Cycle de vie SWF de FlexMojos

Les goals FlexMojos participants sont les suivants :

`flexmojos:compile-swf`

Ce goal compile en un fichier SWF tous les fichiers ActionScript et MXML du répertoire `sourceDirectory`. Un fichier SWF est un fichier qui contient une application qui peut être exécutée dans le lecteur Flash Player ou le l'environnement d'exécution Adobe AIR.

`flexmojos:test-compile`

Ce goal compile tous les fichiers ActionScript et MXML du répertoire `testSourceDirectory`.

`flexmojos:test-run`

Ce goal exécute les tests unitaires en utilisant le lecteur Flash Player. Ce goal ne peut être exécuté que si le lecteur Flash Player été à correctement configuré comme décrit dans la Section 19.2.2, « Configuration de l'environnement pour les tests Flex Unit ».

## 19.5. Les goals du plugin FlexMojos

Le plugin Maven FlexMojos contient les goals suivants :

`flexmojos:asdoc`

Génère la documentation pour les fichiers source ActionScript

`flexmojos:asdoc-report`

Génère la documentation pour les fichiers source ActionScript sous la forme d'un rapport qui peut être inclus dans un site Maven

`flexmojos:compile-swc`

Compile les sources Flex (ActionScript et MXML) en une bibliothèque SWC pour une intégration dans une application Flex ou AIR

`flexmojos:compile-swf`

Compile les sources Flex (ActionScript et MXML) en une application SWF pour lecteur Adobe Flash Player ou l'environnement d'exécution Adobe AIR

`flexmojos:copy-flex-resources`

Copie les ressources Flex dans le projet de l'application web

`flexmojos:flexbuilder`

Génère les fichiers de configuration du projet pour l'environnement de développement Adobe Flex Builder

`flexmojos:generate`

Génère, via Granite GAS3, des fichiers sources ActionScript 3 sur la base de classes Java

`flexmojos:optimize`

Goal qui exécute une optimisation des fichiers swc, postérieurement à leur inclusion dans une application SWF. Ce goal est utilisé pour produire des fichiers RSL.

flexmojos:sources

Génère un JAR contenant toutes les sources du projet Flex

flexmojos:test-compile

Compile toutes les classes de test du projet Flex

flexmojos:test-run

Exécute les tests unitaires en utilisant le lecteur Adobe Flash Player

flexmojos:test-swc

produit un fichier SWC contenant les classes de test du projet

flexmojos:wrapper

Génère une page HTML qui encapsule l'application SWF

### 19.5.1. Génération de la documentation ActionScript

Vous pouvez exécuter les goals `asdoc` ou `asdoc-report` pour générer la documentation des sources ActionScript. Une fois générée, la documentation est sauvegardée en HTML dans le répertoire `$(basedir)/target/asdoc`. La Figure 19.9, « Documentation ActionScript produite par le plugin FlexMojos » illustre le résultat issu de l'exécution du goal `asdoc` pour le projet obtenu via l'archétype Flexmojos.

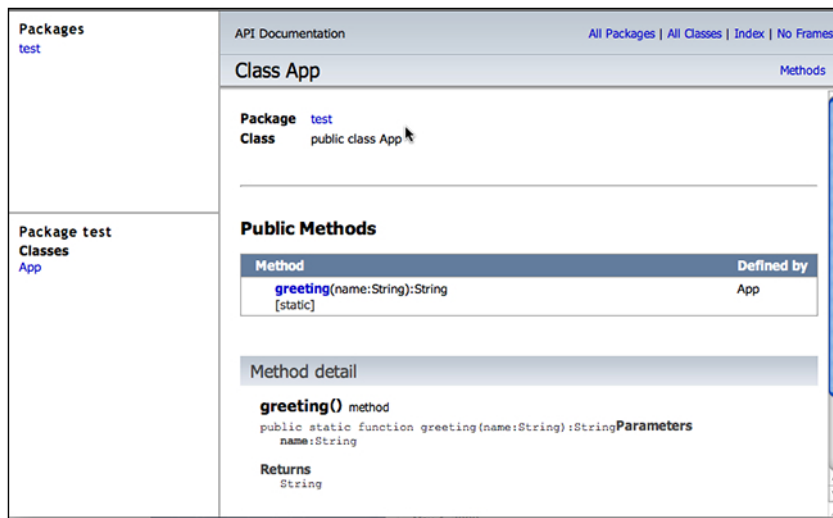


Figure 19.9. Documentation ActionScript produite par le plugin FlexMojos

### 19.5.2. Compilation des sources Flex

FlexMojos présente nombre de goals compilant ActionScript et MXML en fichiers SWC et SWF. Les goals `compile-swc` et `compile-swf` sont utilisés pour produire les artefacts à partir des sources du

projet. Le goal `test-compile` est utilisé pour compiler les tests unitaires. Dans les projets simples créés par les archétypes FlexMojos, les goals `compile-swc` et `compile-swf` sont invoqués parce que le projet personnalise le cycle de vie et rattache `compile-swc` ou `compile-swf` à la phase de compilation et `test-compile` à la phase `test-compile`. Si vous avez besoin de configurer les options du compilateur FlexMojos, vous le ferez à travers les options de configuration du plugin FlexMojos. Par exemple, si vous voulez que l'application avec le POM illustré dans l'Exemple 19.7, « POM généré par l'archétype Application Flex » ignore les alertes de compilation au niveau du code et utilise des polices personnalisées, vous pouvez utiliser une configuration de plugin comme celle illustrée dans l'Exemple 19.15, « Configuration du plugin pour une compilation personnalisée ».

#### Exemple 19.15. Configuration du plugin pour une compilation personnalisée

```
<build>
 <sourceDirectory>src/main/flex</sourceDirectory>
 <testSourceDirectory>src/test/flex</testSourceDirectory>
 <plugins>
 <plugin>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-maven-plugin</artifactId>
 <version>3.3.0</version>
 <extensions>true</extensions>
 <configuration>
 <configurationReport>true</configurationReport>
 <warnings>
 <arrayToStringChanges>true</arrayToStringChanges>
 <duplicateArgumentNames>>false</duplicateArgumentNames>
 </warnings>

 <advancedAntiAliasing>true</advancedAntiAliasing>
 <flashType>true</flashType>
 <languages>
 <englishRange>U+0020-U+007E</englishRange>
 </languages>
 <localFontsSnapshot>
 ${basedir}/src/main/resources/fonts.ser
 </localFontsSnapshot>
 <managers>
 <manager>flash.fonts.BatikFontManager</manager>
 </managers>
 <maxCachedFonts>20</maxCachedFonts>
 <maxGlyphsPerFace>1000</maxGlyphsPerFace>

 </configuration>
 </plugin>
 </plugins>
</build>
```



### 19.5.3. Génération des fichiers de projet Flex Builder

Pour générer les fichiers de projet Flex Builder pour un projet FlexMojos, configurez les groupes de plugin comme décrit dans la Section 19.2.3, « Ajouter FlexMojos aux groupes de plugins de votre configuration Maven » et lancez le goal `flexbuilder` :

```
$ mvn flexmojos:flexbuilder
```

L'exécution de ce goal produira les fichiers : `.project`, `.settings/org.eclipse.core.resources.prefs`, `.actionScriptProperties` et `.flexLibProperties`.

## 19.6. Rapports du plugin FlexMojos

Le plugin Maven FlexMojos propose le rapport suivant :

`flexmojos:asdoc-report`

Produit la documentation du code source ActionScript sous la forme d'un rapport pouvant être intégré dans un site Maven

### 19.6.1. Produire le rapport de documentation ActionScript

Pour que `asdoc-report` soit intégré au build du site Maven, il suffit d'ajouter le code XML suivant à votre POM :

Exemple 19.16. Configuration pour une génération de documentation ActionScript

```
<reporting>
 <plugins>
 <plugin>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-maven-plugin</artifactId>
 <version>3.3.0</version>
 <reportSets>
 <reportSet>
 <id>flex-reports</id>
 <reports>
 <report>asdoc-report</report>
 </reports>
 </reportSet>
 </reportSets>
 </plugin>
 </plugins>
</reporting>
```

Quand vous exécutez la commande `mvn site`, Maven générera cette documentation et la placera dans le menu "Project Reports" comme vous pouvez le voir dans la Figure 19.10, « Documentation ActionScript incluse dans le site Maven ».



Figure 19.10. Documentation ActionScript incluse dans le site Maven

Si vous avez besoin de fournir des options de configuration au goal `asdoc-report`, il vous faudra ajouter un élément de configuration `reportSets` comme le montre l'Exemple 19.17, « Configuration de `asdoc-report` ».

Exemple 19.17. Configuration de `asdoc-report`

```
<reporting>
 <plugins>
 <plugin>
 <groupId>org.sonatype.flexmojos</groupId>
 <artifactId>flexmojos-maven-plugin</artifactId>
 <version>3.3.0</version>
 <reportSets>
 <reportSet>
 <id>flex-reports</id>
 <reports>
 <report>asdoc-report</report>
 </reports>
 <configuration>
 <windowTitle>My TEST API Doc</windowTitle>
 <footer>Copyright 2009 Sonatype</footer>
 </configuration>
 </reportSet>
 </reportSets>
 </plugin>
 </plugins>
</reporting>
```

## 19.7. Développement et personnalisation de Flexmojos

Les sections suivantes vous guideront à travers les premières étapes de personnalisation de Flexmojos, voire de contribution au projet lui-même. Flexmojos est plus qu'un simple outil de compilation de l'ActionScript en artefacts SWF et SWC ; c'est une communauté de développeurs. Cette section ne concerne pas tout le monde, mais si quelque chose vous dérange et que vous voulez participer, vous êtes les bienvenus.

## 19.7.1. Obtenir le code source Flexmojos

Flexmojos est un projet Open Source hébergé par Sonatype dont le code source est enregistré dans le dépôt Subversion de la forge Sonatype. Vous pouvez visualiser le contenu du dépôt Subversion de Flexmojos en ouvrant, dans un navigateur web, le lien <http://svn.sonatype.org/flexmojos/trunk>.

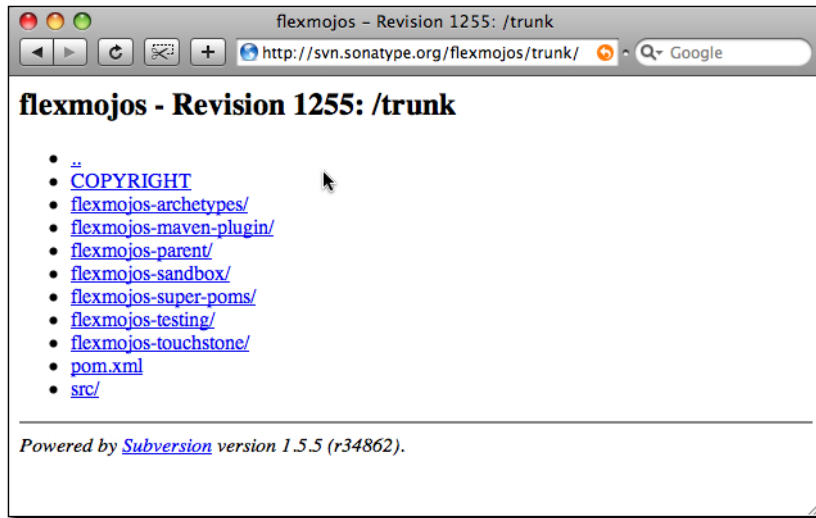


Figure 19.11. Le dépôt Subversion de Flexmojos

Si participer au projet Flexmojos vous intéresse, vous récupérerez probablement la version courante du projet sur votre machine. Pour récupérer le code source Flexmojos en utilisant Subversion, saisissez la ligne de commande suivante :

```
$ svn co http://svn.sonatype.org/flexmojos/trunk flexmojos
A flexmojos
...
$ ls
COPYRIGHT flexmojos-sandbox pom.xml
flexmojos-archetypes flexmojos-super-poms src
flexmojos-maven-plugin flexmojos-testing
flexmojos-parent flexmojos-touchstone
```



# Annexe A. Annexe : détails des settings

## A.1. Aperçu rapide

La balise `settings` du fichier `settings.xml` contient les balises utilisées pour définir les valeurs qui permettent de configurer l'exécution de Maven. Les paramètres de ce fichier doivent s'appliquer à plusieurs projets, ils ne doivent donc pas s'appliquer à une configuration spécifique. Ils permettent par exemple de configurer les dépôts locaux, les serveurs de repository alternatifs, les informations d'authentification. Ce fichier `settings.xml` peut se trouver à deux emplacements :

- Répertoire d'installation de Maven : `$M2_HOME/conf/settings.xml`
- Fichier spécifique à un utilisateur : `~/.m2/settings.xml`

Voici un aperçu des balises de haut niveau à insérer dans la balise `settings` :

Exemple A.1. Aperçu des balises haut niveau du `settings.xml`

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/settings-1.0.0.xsd">
 <localRepository/>
 <interactiveMode/>
 <usePluginRegistry/>
 <offline/>
 <pluginGroups/>
 <servers/>
 <mirrors/>
 <proxies/>
 <profiles/>
 <activeProfiles/>
</settings>
```

## A.2. Détails des settings

### A.2.1. Valeurs simples

La moitié des éléments de haut niveau des paramètres sont des valeurs simples qui permettent de configurer le comportement de base de Maven :

Exemple A.2. Balises simples de haut niveau du `settings.xml`

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/settings-1.0.0.xsd">
<localRepository>${user.dir}/.m2/repository</localRepository>
<interactiveMode>true</interactiveMode>
<usePluginRegistry>false</usePluginRegistry>
<offline>false</offline>
<pluginGroups>
 <pluginGroup>org.codehaus.mojo</pluginGroup>
</pluginGroups>
...
</settings>

```

Les balises simples de haut niveau sont :

#### localRepository

Cette valeur correspond au chemin du dépôt local. Sa valeur par défaut est `${user.dir}/.m2/repository`.

#### interactiveMode

`true` si Maven doit essayer d'interagir avec les entrées de l'utilisateur, `false` sinon. Sa valeur par défaut est `true`.

#### usePluginRegistry

`true` si Maven doit utiliser le fichier `${user.dir}/.m2/plugin-registry.xml` pour gérer les versions des plugins, sa valeur par défaut est `false`.

#### offline

`true` si le système de build doit fonctionner en mode hors connexion, sa valeur par défaut est `false`. Cette balise est très utile pour les serveurs de build qui ne peuvent pas se connecter à des dépôts distants, soit par ce qu'il ne dispose pas de réseau, soit pour des raisons de sécurité.

#### pluginGroups

Cette balise contient une liste de balises `pluginGroup`, chaque balise contenant un `groupId`. La liste est utilisée lorsqu'un plugin est lancé et que le son `groupId` n'est pas fourni par la ligne de commande. Cette liste contient `org.apache.maven.plugins` par défaut.

## A.2.2. Balise `servers`

La balise `distributionManagement` du POM définit les `repository` à utiliser pour le déploiement. Cependant, certains paramètres tels que les certificats de sécurité ne doivent pas être distribués avec le `pom.xml`. Ce type d'information doit se trouver dans le fichier `settings.xml` du serveur de construction.

Exemple A.3. Configuration serveur du `settings.xml`

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
```

```

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/settings-1.0.0.xsd">
...
<servers>
 <server>
 <id>server001</id>
 <username>my_login</username>
 <password>my_password</password>
 <privateKey>${user.home}/.ssh/id_dsa</privateKey>
 <passphrase>some_passphrase</passphrase>
 <filePermissions>664</filePermissions>
 <directoryPermissions>775</directoryPermissions>
 <configuration></configuration>
 </server>
</servers>
...
</settings>

```

Les balises utilisables dans `servers`:

`id`

Il s'agit de l'id du serveur (ce n'est pas le login utilisateur) il correspond à l'id de la balise `distributionManagement` du référentiel.

`username`, `password`

Ces éléments apparaissent sous la forme d'une paire indiquant l'identifiant et le mot de passe requis pour vous authentifier sur ce serveur.

`privateKey`, `passphrase`

Comme pour les deux éléments précédents, cette paire indique le chemin vers une clé privée (qui se trouve par défaut dans `${user.home}/.ssh/id_dsa`) et une passphrase, si nécessaire. Les balises `passphrase` et `password` pourront être externalisées dans le futur, pour le moment leur valeur doit être rentré en plain-text dans le fichier `settings.xml`.

`filePermissions`, `directoryPermissions`

Lorsqu'un fichier ou un répertoire est créé lors du déploiement dans un dépôt, ces permissions sont utilisées. Les valeurs autorisées des nombres à trois chiffres correspondant aux permissions fichiers \*nix, comme 664 ou 775.

### A.2.3. Balise `mirrors`

Exemple A.4. Configuration des miroirs dans le fichier `settings.xml`

```

<settings xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/settings-1.0.0.xsd">

```

```

...
<mirrors>
 <mirror>
 <id>planetmirror.com</id>
 <name>PlanetMirror Australia</name>
 <url>http://downloads.planetmirror.com/pub/maven2</url>
 <mirrorOf>central</mirrorOf>
 </mirror>
</mirrors>
...
</settings>

```

id, name

Identifiant unique d'un miroir. L'id est utilisé pour différencier plusieurs miroirs.

url

L'URL de base de ce miroir. Le système de build utilisera cette URL pour se connecter à un dépôt plutôt que d'utiliser l'URL du serveur par défaut.

mirrorOf

L'id du serveur dont ce miroir fait référence. Par exemple, utilisez cette balise pour pointer vers un miroir du serveur central Maven (<http://repo1.maven.org/maven2>). Il ne doit pas s'agir de 'id' du miroir.

## A.2.4. Balise `proxies`

Exemple A.5. Configuration d'un proxy à partir du `settings.xml`

```

<settings xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/settings-1.0.0.xsd">
...
 <proxies>
 <proxy>
 <id>myproxy</id>
 <active>true</active>
 <protocol>http</protocol>
 <host>proxy.somewhere.com</host>
 <port>8080</port>
 <username>proxyuser</username>
 <password>somepassword</password>
 <nonProxyHosts>*.google.com|ibiblio.org</nonProxyHosts>
 </proxy>
 </proxies>
...
</settings>

```

id

Identifiant unique du proxy. Celui-ci est utilisé pour différencier les éléments `proxy`.



active

true si ce proxy est activé. Vous pouvez ainsi configurer plusieurs proxy, mais d'en activer un seul à la fois à la demande.

protocol, host, port

Les `protocol://host:port` d'un proxy.

username, password

Ces éléments apparaissent sous la forme d'une paire indiquant l'identifiant et le mot de passe requis pour vous authentifier sur ce serveur proxy.

nonProxyHosts

Liste des Hosts pour lesquels il ne faut pas utiliser de proxy. L'exemple ci-dessus est délimité par des pipes (!), il est également possible des les délimiter par des virgules.

### A.2.5. Balise `profiles`

La balise `profile` du fichier `settings.xml` est comparable à une version tronquée de la balise du même nom disponible dans un `pom.xml`. Elle contient des balises `activation`, `repositories`, `pluginRepositories` et `properties`. Cette balise `profile` ne peut inclure que ces quatre éléments car ils se préoccupent du système de construction dans son intégralité.

Si un profil est activé à partir des `settings`, ces valeurs surchargeront n'importe quels autres profils provenant qui matchent les identifiants des fichiers POM ou `profiles.xml`.

### A.2.6. Balise `activation`

Une `activation` est la clé d'un profil. Comme les profils du POM, la puissance d'un profil vient de sa capacité à modifier certaines des valeurs sous certaines conditions. Ces conditions sont indiquées dans la balise `activation`.

Exemple A.6. Balise `activation` du fichier `settings.xml`

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/settings-1.0.0.xsd">
 ...
 <profiles>
 <profile>
 <id>test</id>
 <activation>
 <activeByDefault>>false</activeByDefault>
 <jdk>1.5</jdk>
 <os>
 <name>Windows XP</name>
 <family>Windows</family>
 </os>
 </activation>
 </profile>
 </profiles>
</settings>
```

```

 <arch>x86</arch>
 <version>5.1.2600</version>
 </os>
 <property>
 <name>mavenVersion</name>
 <value>2.0.3</value>
 </property>
 <file>
 <exists>${basedir}/file2.properties</exists>
 <missing>${basedir}/file1.properties</missing>
 </file>
</activation>
...
</profile>
</profiles>
...
</settings>

```

L'activation d'un profil se produit lorsque les critères définis ont été atteints. Tous ne sont pas forcément nécessaires à en même temps.

#### jdk

L'activation a une vérification centrée Java intégrée dans la balise `jdk`. Celui se déclenchera si le test est exécuté avec un numéro de version de JDK qui correspond au préfixe donné. Dans l'exemple ci-dessus, `1.5.0_06` correspondra.

#### os

La balise `os` peut définir des propriétés spécifiques à un système d'exploitation.

#### property

Le profil s'activera si Maven détecte la présence d'une certaine propriété (une valeur qui peut être référencée dans un POM par `${name}`) qui correspond à une paire nom=valeur.

#### file

Enfin, un nom de fichier donné peut activer le profil en fonction de l'existence d'un fichier.

La balise `activation` n'est pas le seul moyen d'activer un profil. La balise `activeProfile` du fichier `settings.xml` peut contenir un id de profil. Ils peuvent également être activés à partir de la ligne de commande en utilisant une liste séparée par virgules après un flag `P` (Exemple : `-P test`).

Pour voir quels profils sont activés sur un build, utiliser le `maven-help-plugin` :

```
mvn help:active-profiles
```

## A.2.7. Balise `properties`

Les propriétés Maven permettent de définir des valeurs, comme celles des propriétés Ant. Leurs valeurs sont accessibles à partir de `n'importe où` dans un POM en utilisant la notation `${X}`, où `X` est la propriété.

Ils existe cinq styles différents de propriétés, qui sont tous disponibles à partir du fichier `settings.xml` :

`env.X`

Préfixer une variable avec `env.` retournera une variable d'environnement. Par exemple, `${env.PATH}` contient la valeur de la variable d'environnement `$path`. (%PATH% sous Windows.)

`project.x`

Ce genre de propriété contiendra la valeur de l'élément correspondant du POM.

`settings.x`

Permet de référencer un élément du fichier `settings.xml`.

Propriétés systèmes Java

Toutes les propriétés disponibles via la méthode `java.lang.System.getProperties()` sont accessibles comme propriété de POM (Exemple : `${java.home}`).

`x`

Affecté avec la balise `properties` ou un par l'intermédiaire d'un fichier externe, cet valeur peut être utilisée comme `${someVar}`.

Exemple A.7. Affecter la propriété `${user.install}` à partir du fichier `settings.xml`

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/settings-1.0.0.xsd">
 ...
 <profiles>
 <profile>
 ...
 <properties>
 <user.install>${user.dir}/our-project</user.install>
 </properties>
 ...
 </profile>
 </profiles>
 ...
</settings>
```

La propriété `${user.install}` est accessible dans un POM si ce profil est activé.

## A.2.8. Balise `repositories`

Les dépôts sont des collections distantes de projets utilisables par Maven pour remplir le dépôt local du système de build. C'est à partir de ce dépôt local que Maven appelle ses plugins et ses dépendances. Différents dépôts distants peuvent contenir des projets, et en fonction du profil actif, on peut les utiliser pour rechercher une certaine release ou un snapshot d'un artefact.

## Exemple A.8. Configuration des dépôts à partir du fichier settings.xml

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/settings-1.0.0.xsd">
 ...
 <profiles>
 <profile>
 ...
 <repositories>
 <repository>
 <id>codehausSnapshots</id>
 <name>Codehaus Snapshots</name>
 <releases>
 <enabled>>false</enabled>
 <updatePolicy>always</updatePolicy>
 <checksumPolicy>warn</checksumPolicy>
 </releases>
 <snapshots>
 <enabled>>true</enabled>
 <updatePolicy>never</updatePolicy>
 <checksumPolicy>fail</checksumPolicy>
 </snapshots>
 <url>http://snapshots.maven.codehaus.org/maven2</url>
 <layout>default</layout>
 </repository>
 </repositories>
 <pluginRepositories>
 ...
 </pluginRepositories>
 </profile>
 </profiles>
 ...
</settings>
```

### releases, snapshots

Ce sont les policy pour chaque type d'artefact, release ou snapshot. Avec ces deux ensembles, un pom a le pouvoir de modifier les policy de chaque type au sein d'un référentiel unique. Un peut décider d'activer seulement le téléchargement de snapshots, par exemple à des fins de développement.

### enabled

true ou false, permet de savoir si un dépôt est activé pour un certain type (releases ou snapshots).

### updatePolicy

Cet élément spécifie la fréquence des mises à jour. Maven va comparer les timestamp des POMs locaux avec les POMs distants. Les choix possibles sont : `always`, `daily` (par défaut), `interval:X` (où X représente un nombre de minute) ou `never`.

## checksumPolicy

Lorsque Maven déploie des fichiers dans le dépôt, il déploie également les fichiers checksum correspondants. Plusieurs options sont disponibles lorsqu'un checksum est manquant ou invalide : `ignore`, `fail` ou `warn`.

## layout

Dans la description des dépôts ci-dessus, il est mentionné que tous les dépôts suivent un layout commun. Maven 2 a un layout par défaut pour ses dépôts, Maven 1.x en avait un différent. Utilisez cet élément pour préciser lesquels sont par défaut ou `legacy`. Si vous êtes passé de Maven 1 à Maven 2 et que vous souhaitez utiliser le même dépôt, renseignez celui-ci comme `legacy`.

## A.2.9. Balise `pluginRepositories`

La structure de la balise `pluginRepositories` est similaire à celle de la balise `repositories`. Les balises `pluginRepositories` spécifient un emplacement distant où l'on peut trouver des artefacts de plugins Maven.

### Exemple A.9. `pluginRepositories` dans le fichier `settings.xml`

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/settings-1.0.0.xsd">
 ...
 <profiles>
 <profile>
 ...
 <repositories>
 ...
 </repositories>
 <pluginRepositories>
 <pluginRepository>
 <id>codehausSnapshots</id>
 <name>Codehaus Snapshots</name>
 <releases>
 <enabled>>false</enabled>
 <updatePolicy>always</updatePolicy>
 <checksumPolicy>warn</checksumPolicy>
 </releases>
 <snapshots>
 <enabled>>true</enabled>
 <updatePolicy>never</updatePolicy>
 <checksumPolicy>fail</checksumPolicy>
 </snapshots>
 <url>http://snapshots.maven.codehaus.org/maven2</url>
 <layout>default</layout>
 </pluginRepository>
 </pluginRepositories>
 ...
 </profile>
```

```
</profiles>
...
</settings>
```

## A.2.10. Balise `activeProfiles`

Exemple A.10. Activer des profils à partir du fichier `settings.xml`

```
<settings xmlns="http://maven.apache.org/POM/4.0.0"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
 http://maven.apache.org/xsd/settings-1.0.0.xsd">
 ...
 <activeProfiles>
 <activeProfile>env-test</activeProfile>
 </activeProfiles>
</settings>
```

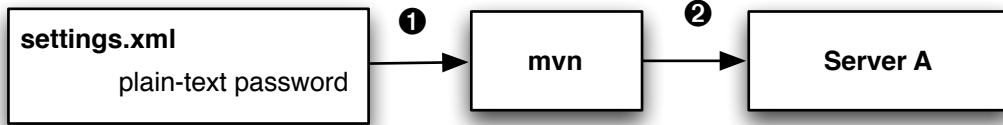
La dernière pièce du puzzle du `settings.xml` est la balise `activeProfiles`. Elle contient un ensemble de balises `activeProfile`, qui ont chacun pour valeur un id de profil. Tout profil dont l'id est présente dans une balise `activeProfile` sera activé, quels que soient les paramètres d'environnement. Si aucun profil n'est trouvé, rien ne se passera. Par exemple, si `env-test` est un `activeProfile`, un `profile` dans un `pom.xml` (ou `profile.xml`) sera activé. Si aucun profil n'est trouvé, l'exécution se poursuivra tout de même normalement.

## A.2.11. Chiffrement des mots de passe dans les Settings Maven

Si vous utilisez Maven pour déployer vos artefacts sur des dépôts distants ou tout autre service distant nécessitant l'utilisation de mots de passe, la meilleure solution est de stocker ces mots de passe dans vos Settings Maven. Sans un mécanisme pour chiffrer ces mots de passe, le fichier `~/.m2/settings.xml` devient rapidement une faille de sécurité, car il contient les mots de passe en clair d'accès à vos différents serveurs. Pour éviter ce problème, Maven 2.1 a introduit une fonctionnalité qui permet de chiffrer vos mots de passe. Pour ce faire, vous devez commencer par créer un mot de passe maître et stocker celui-ci dans un fichier de `security-settings.xml` à l'emplacement `~/.m2/settings-security.xml`. Vous pouvez ensuite utiliser ce dernier pour chiffrer vos autres mots de passe, ceux que vous deviez stocker en clair dans vos Settings Maven (`~/.m2/settings.xml`).

Pour illustrer cette fonctionnalité, regardons le processus utilisé par Maven pour récupérer le mot de passe non chiffré d'un serveur à partir des paramètres Maven d'un utilisateur. La Figure A.1, « Stockage de mot de passe non crypté dans les Settings Maven » présente ce processus. Un utilisateur peut faire référence à un serveur en utilisant un identifiant dans le `POM` d'un projet, Maven recherche alors le serveur correspondant dans ses settings. Une fois trouvé, Maven utilisera le mot de passe associé à celui-ci. Le mot de passe est stocké en clair dans `~/.m2/settings.xml`, il est donc facilement accessible à toute personne qui dispose des droits de lecture sur ce fichier.

~/m2

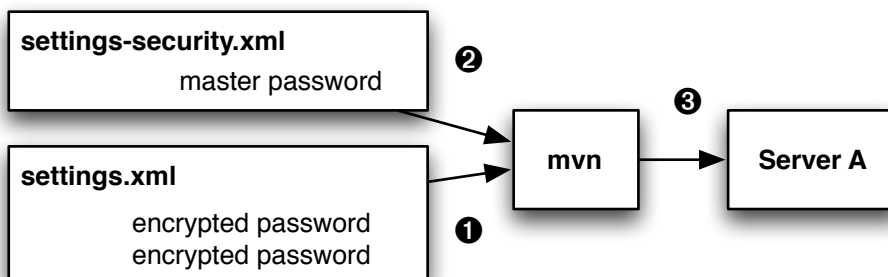


- ❶ Maven Retrieves password for Server A from ~/m2/settings.
- ❷ Maven sends the password to the remote server.

Figure A.1. Stockage de mot de passe non crypté dans les Settings Maven

Maintenant, regardons comment Maven utilise des mots de passe chiffrés. La Figure A.2, « Stockage d'un mot de passe crypté dans les Settings Maven » présente ce processus.

~/m2



- ❶ Maven Retrieves the Encrypted Password for Server A from ~/m2/settings.
- ❷ Maven retrieves the master password from ~/m2/security-settings.xml
- ❸ Maven decrypts the password and sends the decrypted password to the remote server.

Figure A.2. Stockage d'un mot de passe crypté dans les Settings Maven

Pour configurer la fonctionnalité de chiffrement des mots de passe, créez le mot de passe maître en exécutant l'une des commandes `mvn -emp` ou `mvn --encrypt-master-password` suivi par votre mot de passe maître.

```
$ mvn -emp mypassword
{rsB56BJcqoEHZqEZ0R1VR4TIspmODx1Ln8/PVvsGgAw=}
```

Maven affiche alors une copie de la version chiffrée du mot de passe sur la sortie standard. Copiez celui-ci et collez-le dans le fichier `~/m2/settings-security.xml` comme le montre l'exemple suivant.

### Exemple A.11. settings-security.xml avec un mot de passe maître

```
<settingsSecurity>
 <master>{rsB56BJcqoEHZqEZ0R1VR4TIspmODx1Ln8/PVvsgaGw=}</master>
</settingsSecurity>
```

Après avoir créé le mot de passe maître, vous pouvez commencer à chiffrer vos mots de passe pour les utiliser dans vos settings. Pour chiffrer un mot de passe à l'aide du mot de passe maître, utilisez l'une des commande `mvn -ep` ou `mvn --encrypt-password`. Imaginons que vous disposez d'un gestionnaire de dépôt et que vous avez besoin d'un utilisateur "deployment" et d'un mot de passe "qualityFIRST" pour vous y connecter. Pour chiffrer ce mot de passe, utilisez la ligne de commande suivante :

```
$ mvn -ep qualityFIRST
{uMrbEOEf/VQHnc0W2X49Qab75j9LSTwiM3mg2LCrOzI=}
```

Copiez ensuite le mot de passe chiffré affiché sur la sortie standard et collez-le dans vos settings Maven.

### Exemple A.12. Stocker un mot de passe encrypté dans les Settings Maven (~/.m2/settings.xml)

```
<settings>
 <servers>
 <server>
 <id>nexus</id>
 <username>deployment</username>
 <password>{uMrbEOEf/VQHnc0W2X49Qab75j9LSTwiM3mg2LCrOzI=}</password>
 </server>
 </servers>
 ...
</settings>
```

Lorsque vous exécutez un build Maven qui a besoin d'interagir avec le gestionnaire de repository, Maven récupérera le mot de passe principal à partir du fichier `~/.m2/settings-security.xml` et utilisera celui-ci pour déchiffrer le mot de passe stocké dans votre fichier `~/.m2/settings.xml`. Maven utilisera ce mot de passe déchiffré pour se connecter au serveur.

Quels en sont les bénéfices d'un tel mécanisme ? Il vous permet d'éviter de stocker en clair vos mots de passe dans le fichier `~/.m2/settings.xml`. Notez que cette fonctionnalité ne prévoit pas le chiffage du mot de passe lors de l'envoi de celui-ci sur le serveur distant. Il est toujours possible, pour une personne mal intentionnée, de récupérer vos mots de passe en analysant les flux réseau.

Pour encore plus de sécurité, vous pouvez demander à vos développeurs de stocker le mot de passe maître chiffré sur un périphérique de stockage amovible comme un disque dur USB. En utilisant cette méthode, un développeur doit brancher son disque amovible sur sa station de travail lorsqu'il veut effectuer un déploiement ou une quelconque interaction avec un serveur distant. Pour cela, votre fichier `~/.m2/settings-security.xml` doit contenir une référence vers l'emplacement réel de votre fichier `settings-security.xml`. Cette configuration passe par l'utilisation de la balise `relocation`.



Exemple A.13. Configuration de la balise **relocation** du mot de passe maître

```
<settingsSecurity>
 <relocation>/Volumes/usb-key/settings-security.xml</relocation>
</settingsSecurity>
```

Ainsi, le développeur peut stocker son fichier `settings-security.xml` sur son emplacement personnalisé `/Volumes/usb-key/settings-security.xml` et s'arranger pour que ce fichier ne soit disponible que s'il est assis devant sa station de travail.



# Annexe B. Annexe : alternatives aux spécifications Sun

Le projet Apache Geronimo maintient les implémentations de plusieurs spécifications de Java EE. Le tableau Tableau B.1, « Autres implémentations des artefacts de Specs » liste les `artifactId` et `version` de toutes les spécifications implémentées par le projet Geronimo. Pour utiliser l'une de ces dépendances, utilisez le `groupId` `org.apache.geronimo.specs` et recherchez la version des spécifications que vous désirez utiliser en dépendance.



## Note

Tous les artefacts du tableau Tableau B.1, « Autres implémentations des artefacts de Specs » possèdent le même `groupId` : `org.apache.geronimo.specs`.

Tableau B.1. Autres implémentations des artefacts de Specs

Spécification	Version de la Spec	Id de l'artefact	Version de l'artefact
Activation	1.0.2	geronimo-activation_1.0.2_spec	1.2
Activation	1.1	geronimo-activation_1.1_spec	1.0.1
Activation	1.0	geronimo-activation_1.0_spec	1.1
CommonJ	1.1	geronimo-commonj_1.1_spec	1.0
Corba	2.3	geronimo-corba_2.3_spec	1.1
Corba	3.0	geronimo-corba_3.0_spec	1.2
EJB	2.1	geronimo-ejb_2.1_spec	1.1
EJB	3.0	geronimo-ejb_3.0_spec	1.0
EL	1.0	geronimo-el_1.0_spec	1.0
Interceptor	3.0	geronimo-interceptor_3.0_spec	1.0
J2EE Connector	1.5	geronimo-j2ee-connector_1.5_spec	1.1.1
J2EE Deployment	1.1	geronimo-j2ee-deployment_1.1_spec	1.1
J2EE JACC	1.0	geronimo-j2ee-jacc_1.0_spec	1.1.1
J2EE Management	1.0	geronimo-j2ee-management_1.0_spec	1.1
J2EE Management	1.1	geronimo-j2ee-management_1.1_spec	1.0
J2EE	1.4	geronimo-j2ee_1.4_spec	1.1
JACC	1.1	geronimo-jacc_1.1_spec	1.0

Spécification	Version de la Spec	Id de l'artefact	Version de l'artefact
JEE Deployment	1.1MR3	geronimo-javaee-deployment_1.1MR3_spec	1.0
JavaMail	1.3.1	geronimo-javamail_1.3.1_spec	1.3
JavaMail	1.4	geronimo-javamail_1.4_spec	1.2
JAXR	1.0	geronimo-jaxr_1.0_spec	1.1
JAXRPC	1.1	geronimo-jaxrpc_1.1_spec	1.1
JMS	1.1	geronimo-jms_1.1_spec	1.1
JPA	3.0	geronimo-jpa_3.0_spec	1.1
JSP	2.0	geronimo-jsp_2.0_spec	1.1
JSP	2.1	geronimo-jsp_2.1_spec	1.0
JTA	1.0.1B	geronimo-jta_1.0.1B_spec	1.1.1
JTA	1.1	geronimo-jta_1.1_spec	1.1
QName	1.1	geronimo-qname_1.1_spec	1.1
SAAJ	1.1	geronimo-saaaj_1.1_spec	1.1
Servlet	2.4	geronimo-servlet_2.4_spec	1.1.1
Servlet	2.5	geronimo-servlet_2.5_spec	1.1.1
STaX API	1.0	geronimo-stax-api_1.0_spec	1.0.1
WS Metadata	2.0	geronimo-ws-metadata_2.0_spec	1.1.1



## Note

Ces versions d'artefacts seront probablement dépassées lorsque vous lirez ce livre. Pour vérifier les dernières versions disponibles, rendez visite à l'adresse <http://repo1.maven.org/maven2/org/apache/geronimo/specs/> et cliquez sur l'artefactId que vous désirez ajouter.

Pour illustrer l'utilisation du tableau Tableau B.1, « Autres implémentations des artefacts de Specs », si vous voulez écrire du code dans votre projet qui nécessite les spécifications JTA 1.0.1B, ajoutez la dépendance suivante à votre projet :

Exemple B.1. Ajout de JTA 1.0.1B à un projet MavenProject

```
<dependency>
 <groupId>org.apache.geronimo.specs</groupId>
 <artifactId>geronimo-jta_1.0.1B_spec</artifactId>
 <version>1.1.1</version>
</dependency>
```

Notez que la version de cet artefact n'est pas alignée avec la version des spécifications. La dépendance précédente ajoute la version 1.0.1B des spécifications JTA en utilisant la version 1.1.1. de l'artefact. Soyez conscients de cela lorsque vous récupérez les dernières versions des artefacts, vérifiez qu'elle est compatible avec vos spécifications.

