

Copyright	xxi
Avant-propos	xxiii
Préface	xxv
1. Audience	xxv
2. Organisation du livre	xxv
3. Jenkins ou Hudson?	xxv
4. Conventions sur les polices	xxvi
5. Conventions de ligne de commande	xxvi
6. Contributeurs	xxvii
6.1. Les traducteurs du présent livre en français	xxviii
7. L'équipe de revue	xxix
8. Sponsors du livre	xxix
8.1. Wakaleo Consulting	xxix
8.2. CloudBees	xxix
8.3. Odd-e	xxx
9. Utilisation des exemples de code	xxx
10. Safari® Books Online	xxxi
11. Comment nous contacter	xxxi
12. Remerciements	xxxii
1. Introduction à Jenkins	1
1.1. Introduction	1
1.2. Les fondamentaux de l'Intégration Continue	1
1.3. Introduction à Jenkins (né Hudson)	3
1.4. De Hudson à Jenkins — Un rapide historique	3
1.5. Dois-je utiliser Jenkins ou Hudson?	4
1.6. Mettre en place l'Intégration Continue au sein de votre organisation	5
1.6.1. Phase 1 — Pas de serveur de build	5
1.6.2. Phase 2 — Builds quotidiens	6
1.6.3. Phase 3 — Builds quotidiens et tests automatisés basiques	6
1.6.4. Phase 4 — Arrivée des métriques	6
1.6.5. Phase 5 — Prendre les tests au sérieux	6
1.6.6. Phase 6 — Tests d'acceptance automatisés et un déploiement plus automatisé	6
1.6.7. Phase 7 — Déploiement Continu	7
1.7. Et maintenant ?	7
2. Vos premiers pas avec Jenkins	9
2.1. Introduction	9
2.2. Préparation de votre environnement	9
2.2.1. Installation de Java	10
2.2.2. Installation de Git	11
2.2.3. Configurer un compte GitHub	11
2.2.4. Configurer les clefs SSH	12
2.2.5. Forker le dépôt des exemples	12

2.3. Démarrer Jenkins	14
2.4. Configurer les Outils	18
2.4.1. Configurer votre configuration Maven	19
2.4.2. Configurer le JDK	20
2.4.3. Notification	21
2.4.4. Configurer Git	21
2.5. Votre premier travail Jenkins	22
2.6. Votre premier travail Jenkins en action	27
2.7. Plus de rapports—Afficher les Javadocs	34
2.8. Ajouter la couverture de code et d'autres métriques	36
2.9. Conclusion	42
3. Installer Jenkins	43
3.1. Introduction	43
3.2. Télécharger et installer Jenkins	43
3.3. Préparation d'un serveur de build pour Jenkins	46
3.4. Le répertoire de travail de Jenkins	48
3.5. Installer Jenkins sur Debian ou Ubuntu	49
3.6. Installer Jenkins sur Redhat, Fedora ou CentOS	50
3.7. Installer Jenkins sur SUSE ou OpenSUSE	51
3.8. Exécuter Jenkins comme une application autonome	52
3.9. Running Jenkins Behind an Apache Server	56
3.10. Running Jenkins on an Application Server	57
3.11. Memory Considerations	58
3.12. Installing Jenkins as a Windows Service	59
3.13. What's in the Jenkins Home Directory	63
3.14. Backing Up Your Jenkins Data	67
3.15. Upgrading Your Jenkins Installation	67
3.16. Conclusion	68
4. Configurer votre serveur Jenkins	69
4.1. Introduction	69
4.2. Le tableau de bord de configuration — L'écran Administrer Jenkins	69
4.3. Configurer l'environnement système	72
4.4. Configurer les propriétés globales	73
4.5. Configurer vos JDKs	74
4.6. Configurer vos outils de build	77
4.6.1. Maven	77
4.6.2. Ant	78
4.6.3. Langage de scripts Shell	79
4.7. Configurer vos outils de gestion de version	79
4.7.1. Configurer Subversion	80
4.7.2. Configurer CVS	80
4.8. Configurer le serveur de messagerie électronique	80
4.9. Configurer un Proxy	81

4.10. Conclusion	82
5. Configurer vos tâches de Build	83
5.1. Introduction	83
5.2. Tâches de Build Jenkins	83
5.3. Créer une tâche de build free-style	84
5.3.1. Options Générales	84
5.3.2. Options avancées du projet	86
5.4. Configurer la Gestion du Code Source	88
5.4.1. Travailler avec Subversion	88
5.4.2. Travailler avec Git	91
5.5. Déclencheurs de build	103
5.5.1. Déclencher une tâche de build lorsqu'une autre tâche de build se termine	104
5.5.2. Tâches de build périodiques	104
5.5.3. Scruter le SCM	105
5.5.4. Déclencher des builds à distance	106
5.5.5. Construction manuelle de tâches	108
5.6. Les étapes de builds	108
5.6.1. Les étapes de build Maven	109
5.6.2. Les étapes de build Ant	111
5.6.3. Exécuter une commande Batch Shell ou Windows	111
5.6.4. Utiliser les variables d'environnement Jenkins dans vos builds	113
5.6.5. Exécuter des scripts Groovy	115
5.6.6. Construire des projets dans d'autres langages	117
5.7. Les actions à la suite du build	117
5.7.1. Rapport sur les résultats de tests	117
5.7.2. Archiver les résultats de build	118
5.7.3. Notifications	122
5.7.4. Construire d'autres projets	122
5.8. Démarrer votre nouvelle tâche de build	123
5.9. Travailler avec des tâches de build Maven	123
5.9.1. Construire dès lors qu'une dépendance SNAPSHOT est construite	124
5.9.2. Configurer un build Maven	124
5.9.3. Les actions à la suite du build	126
5.9.4. Déployer vers un gestionnaire de dépôt d'entreprise	127
5.9.5. Déployer vers des gestionnaires de dépôt d'entreprise commerciales	130
5.9.6. Gérer les modules	131
5.9.7. Les étapes de build supplémentaires dans votre tâche de build Maven	132
5.10. Utiliser Jenkins avec d'autres langages	132
5.10.1. Construire des projets avec Grails	133
5.10.2. Construire des projets avec Gradle	134
5.10.3. Construire des projets avec Visual Studio MSBuild	137
5.10.4. Construire des projets avec NAnt	138
5.10.5. Construire des projets avec Ruby et Ruby on Rails	139

5.11. Conclusion	141
6. Tests automatisés	143
6.1. Introduction	143
6.2. Automatisez vos tests unitaires et d'intégration	144
6.3. Configuration des rapports de test dans Jenkins	145
6.4. Afficher les résultats de test	147
6.5. Ignorer des tests	150
6.6. Couverture de code	152
6.6.1. Mesurer la couverture de code avec Cobertura	153
6.6.2. Mesurer la couverture de code avec Clover	162
6.7. Tests d'acceptation automatisés	164
6.8. Tests de performance automatisés avec JMeter	167
6.9. A l'aide ! Mes tests sont trop lents !	175
6.9.1. Ajouter plus de matériel	175
6.9.2. Lancer moins de tests d'intégration/fonctionnels	176
6.9.3. Exécutez vos tests en parallèle	176
6.10. Conclusion	177
7. Sécuriser Jenkins	179
7.1. Introduction	179
7.2. Activer la sécurité dans Jenkins	179
7.3. Sécurité simple dans Jenkins	180
7.4. Domaines de sécurité — Identifier les utilisateurs Jenkins	181
7.4.1. Utiliser la base de données intégrée à Jenkins	181
7.4.2. Utiliser un annuaire LDAP	185
7.4.3. Utiliser Microsoft Active Directory	186
7.4.4. Utiliser les utilisateurs et les groupes Unix	187
7.4.5. Déléguer au conteneur de Servlet	187
7.4.6. Utiliser Atlassian Crowd	188
7.4.7. S'intégrer avec d'autres systèmes	190
7.5. Autorisation — Qui peut faire quoi	191
7.5.1. Sécurité basée sur une matrice	192
7.5.2. Sécurité basée sur le projet	196
7.5.3. Sécurité basée sur les rôles	198
7.6. Audit — Garder la trace des actions utilisateurs	201
7.7. Conclusion	204
8. Notification	205
8.1. Introduction	205
8.2. Notification par email	205
8.3. Notification par email avancée	207
8.4. Revendiquer des builds	210
8.5. Flux RSS	211
8.6. Radars de build	212
8.7. Messagerie instantanée	214

8.7.1. Notification par IM avec Jabber	214
8.7.2. Notification avec IRC	219
8.8. Notification par IRC	219
8.9. Notificateurs de bureau	223
8.10. Notifications via Notifo	227
8.11. Notifications vers mobiles	229
8.12. Notifications via SMS	230
8.13. Faire du bruit	232
8.14. Appareils de retour extrêmes	234
8.15. Conclusion	236
9. Qualité du Code	237
9.1. Introduction	237
9.2. La qualité du code dans votre processus de build	238
9.3. Les outils d'analyse de qualité du code populaires pour Java et Groovy	239
9.3.1. Checkstyle	239
9.3.2. PMD/CPD	242
9.3.3. FindBugs	246
9.3.4. CodeNarc	248
9.4. Rapports de problèmes de qualité de code avec le plugin Violations	249
9.4.1. Travailler avec des tâches de build free-style	250
9.4.2. Travailler avec des tâches de build Maven	253
9.5. Utiliser les rapports Checkstyle, PMD, et FindBugs	255
9.6. Les rapports sur la complexité du code	258
9.7. Les rapports sur les tâches ouvertes	259
9.8. Intégration avec Sonar	261
9.9. Conclusion	265
10. Builds avancés	267
10.1. Introduction	267
10.2. Tâches de build paramétrées	267
10.2.1. Créer des tâches de build paramétrées	267
10.2.2. Adapter vos build pour travailler avec des scripts de builds paramétrés	269
10.2.3. Types de paramètres plus avancés	271
10.2.4. Construire à partir d'un tag Subversion	273
10.2.5. Réaliser un build à partir d'un tag Git	274
10.2.6. Démarrer une tâche de build paramétrée à distance	275
10.2.7. Historique des tâches de build paramétrées	275
10.3. Déclencheurs paramétrés	276
10.4. Tâches de build multiconfiguration	279
10.4.1. Configurer un build multiconfiguration	279
10.4.2. Configurer un axe Esclave	280
10.4.3. Configurer un axe JDK	281
10.4.4. Axe personnalisé	282
10.4.5. Exécuter un Build Multiconfiguration	282

10.5. Générer vos tâches de build Maven automatiquement	285
10.5.1. Configurer une tâche	286
10.5.2. Réutiliser une configuration de tâche par héritage	288
10.5.3. Le support des plugins	290
10.5.4. Les tâches Freestyle	293
10.6. Coordonner vos builds	293
10.6.1. Les builds parallèles dans Jenkins	293
10.6.2. Graphes de dépendance	294
10.6.3. Jonctions	295
10.6.4. Plugin Locks and Latches	297
10.7. Pipelines de build et promotions	298
10.7.1. Gestion des releases Maven avec le plugin M2Release	298
10.7.2. Copier des artefacts	301
10.7.3. Promotions de build	305
10.7.4. Agréger des résultats de tests	313
10.7.5. Pipelines de Build	314
10.8. Conclusion	317
11. Builds distribués	319
11.1. Introduction	319
11.2. L'Architecture de build distribuée de Jenkins	319
11.3. Stratégies Maître/Esclave dans Jenkins	320
11.3.1. Le maître démarre l'agent esclave en utilisant SSH	321
11.3.2. Démarrer l'agent esclave manuellement via Java Web Start	325
11.3.3. Installer un esclave Jenkins en tant que service Windows	328
11.3.4. Démarrer le noeud esclave en mode Headless	329
11.3.5. Démarrer un esclave Windows en tant que service distant	329
11.4. Associer une tâche de build avec un esclave ou un groupe d'esclaves	330
11.5. Surveillance des noeuds	332
11.6. Cloud computing	333
11.6.1. Utiliser Amazon EC2	333
11.7. Utiliser le service CloudBees DEV@cloud	338
11.8. Conclusion	339
12. Déploiement automatisé et livraison continue	341
12.1. Introduction	341
12.2. Mise en oeuvre du déploiement automatisé et continu	342
12.2.1. Le script de déploiement	342
12.2.2. Mises à jour de base de données	342
12.2.3. Tests fumigatoires	345
12.2.4. Revenir sur des changements	345
12.3. Déployer vers un serveur d'application	346
12.3.1. Déployer une application Java	346
12.3.2. Déployer des applications à base de scripts telles Ruby et PHP	356
12.4. Conclusion	359

13. Maintenir Jenkins	361
13.1. Introduction	361
13.2. Surveillance de l'espace disque	361
13.2.1. Utiliser le plugin "Disk Usage"	362
13.2.2. Disk Usage et les projets Jenkins de type Apache Maven	364
13.3. Surveiller la charge serveur	365
13.4. Sauvegarde de votre configuration	367
13.4.1. Fondamentaux de la sauvegarde Jenkins	367
13.4.2. Utilisation du Backup Plugin	369
13.4.3. Des sauvegardes automatisées plus légères	371
13.5. Archiver les tâches de build	372
13.6. Migrer les tâches de build	373
13.7. Conclusion	377
A. Automatiser vos tests unitaires et d'intégration	379
A.1. Automatiser vos tests avec Maven	379
A.2. Automatiser vos tests avec Ant	384
Index	389

List of Figures

2.1. Installer Java	10
2.2. Créer un compte GitHub	12
2.3. Forker le dépôt des exemples de code	13
2.4. Exécuter Jenkins en utilisant Java Web Start à partir du site web du livre	15
2.5. Java Web Start téléchargera et exécutera la dernière version de Jenkins	16
2.6. Java Web Start exécutant Jenkins	16
2.7. La page de démarrage Jenkins	17
2.8. The Manage Jenkins screen	18
2.9. The Configure Jenkins screen	19
2.10. Configuring a Maven installation	20
2.11. Configuring a JDK installation	21
2.12. Managing plugins in Jenkins	22
2.13. Installing the Git plugin	22
2.14. Setting up your first build job in Jenkins	24
2.15. Telling Jenkins where to find the source code	25
2.16. Scheduling the build jobs	26
2.17. Adding a build step	26
2.18. Configuring JUnit test reports and artifact archiving	27
2.19. Your first build job running	28
2.20. The Jenkins dashboard	28
2.21. A failed build	31
2.22. The list of all the broken tests	32
2.23. Details about a failed test	32
2.24. Now the build is back to normal	34
2.25. Adding a new build step and report to generate Javadoc	35
2.26. Jenkins will add a Javadoc link to your build results	36
2.27. Jenkins has a large range of plugins available	37
2.28. Adding another Maven goal to generating test coverage metrics	38
2.29. Configuring the test coverage metrics in Jenkins	39
2.30. Jenkins displays code coverage metrics on the build home page	40
2.31. Jenkins lets you display code coverage metrics for packages and classes	41
2.32. Jenkins also displays a graph of code coverage over time	42
3.1. Vous pouvez télécharger les binaires de Jenkins sur le site web de Jenkins	44
3.2. L'assistant de configuration sous Windows	45
3.3. La page d'accueil de Jenkins	46
3.4. Starting Jenkins using Java Web Start	60
3.5. Installing Jenkins as a Windows service	61
3.6. Configuring the Jenkins Windows Service	62
3.7. The Jenkins home directory	64
3.8. The Jenkins jobs directory	65

3.9. The builds directory	66
3.10. Upgrading Jenkins from the web interface	68
4.1. Configurer son installation Jenkins dans l'écran Administrer Jenkins	70
4.2. Configuration du système dans Jenkins	72
4.3. Configurer les variables d'environnement dans Jenkins	74
4.4. Utiliser une variable d'environnement configurée	74
4.5. Configuration des JDKs dans Jenkins	75
4.6. Installer un JDK automatiquement	76
4.7. Configurer Maven dans Jenkins	77
4.8. Configurer la variable système MVN_OPTS	78
4.9. Configurer Ant dans Jenkins	79
4.10. Configurer un serveur d'email dans Jenkins	80
4.11. Configurer un serveur d'email pour utiliser un domaine Google Apps	81
4.12. Configurer Jenkins pour utiliser un proxy	82
5.1. Jenkins supporte quatre principaux types de tâches de build	84
5.2. Créer une nouvelle tâche de build	85
5.3. Conserver un build sans limite de temps	86
5.4. Pour afficher les options avancées, vous devez cliquer sur le bouton Avancé...	86
5.5. L'option "Empêcher le build quand un projet en aval est en cours de build" est utile quand un simple commit affecte plusieurs projets dépendants les uns des autres.	87
5.6. Jenkins embarque par défaut le support pour Subversion	88
5.7. Navigateur de code source montrant les changements dans le code qui ont causé le build.....	90
5.8. Configuration système du plugin Git	92
5.9. Remplir une URL de dépôt Git	93
5.10. Configuration avancée d'une URL de dépôt Git	94
5.11. Configuration avancée des branches Git à construire	94
5.12. Branches et régions	95
5.13. Choix de la stratégie	97
5.14. Configuration globale de l'exécutable de Git	98
5.15. Navigateur de dépôt	98
5.16. Journal de scrutation	99
5.17. Résultats de la scrutation de Git	99
5.18. Déclenchement par Gerrit	100
5.19. Git Publisher	101
5.20. Fusionner les résultats	102
5.21. Navigateur de dépôt GitHub	103
5.22. Navigateur de dépôt GitHub	103
5.23. Il y a de multiples manières de configurer Jenkins pour le démarrage d'une tâche de build	103
5.24. Déclencher une autre tâche de build même si celle-ci est instable.	104
5.25. Déclencher un build via une URL en utilisant un jeton	108
5.26. Ajouter une étape de build à une tâche de build Freestyle	110
5.27. Configurer une étape de build Ant	111

5.28. Configurer une étape Exécuter un script Shell	112
5.29. Ajouter une installation Groovy à Jenkins	116
5.30. Lancer des commandes Groovy dans le cadre d'une tâche de build	116
5.31. Lancer des scripts Groovy dans le cadre d'une tâche de build	117
5.32. Rapport sur les résultats de tests	118
5.33. Configurer les artefacts de builds	118
5.34. Les artefacts de build sont affichés sur la page de résultat d'un build et la page d'accueil d'un job	119
5.35. Archiver le code source et un paquet binaire	121
5.36. Notification par email	122
5.37. Créer une nouvelle tâche de build Maven	123
5.38. Spécifier les goals Maven	125
5.39. Les tâches de build Maven — les options avancées	125
5.40. Déployer des artefacts vers un dépôt Maven	127
5.41. Après déploiement, l'artefact devrait être disponible sur votre gestionnaire de dépôt d'entreprise	128
5.42. Redéployer un artefact	129
5.43. Déployer vers Artifactory depuis Jenkins	129
5.44. Jenkins affiche un lien vers le dépôt Artifactory correspondant	130
5.45. Voir l'artefact déployé sur Artifactory	130
5.46. Voir les artefacts déployés et le build Jenkins correspondant dans Artifactory	131
5.47. Gérer les modules dans une tâche de build Maven	131
5.48. Configurer des étapes de build Maven supplémentaires	132
5.49. Ajouter une installation Grails à Jenkins	133
5.50. Configurer une étape de build Grails	134
5.51. Configurer le plugin Gradle	135
5.52. Configurer une tâche de build Gradle	137
5.53. Tâche incrémentale de Gradle	137
5.54. Configurer les outils de build .NET avec Jenkins	138
5.55. Une étape de build utilisant MSBuild	138
5.56. Une étape de build utilisant NAnt	139
5.57. Une étape de build utilisant Rake	140
5.58. Publier des métriques de qualité de code pour Ruby et Rails	141
6.1. Vous configurez votre installation Jenkins dans l'écran Administrer Jenkins	145
6.2. Configurer les rapports de test Maven dans un projet free-style	146
6.3. Installer le plugin xUnit	146
6.4. Publier les résultat de test xUnit	147
6.5. Jenkins affiche la tendance des résultats de test sur la page d'accueil du projet	148
6.6. Jenkins affiche une vue synthétique des résultats de test	148
6.7. Les détails d'un échec de test	149
6.8. Les tendances de temps de build peuvent vous donner un bon indicateur de la rapidité de vos tests	150
6.9. Jenkins vous permet de voir combien de temps les tests ont mis pour s'exécuter	151

6.10. Jenkins affiche les tests ignorés en jaune	152
6.11. Installer le plugin Cobertura	158
6.12. Votre build de couverture de code doit produire les données de couverture de code	159
6.13. Configurer les métriques de couverture de code dans Jenkins	159
6.14. Les résultats des tests de couverture de code contribuent à l'état du projet sur le tableau de bord	160
6.15. Configurer les métriques de couverture de code dans Jenkins	161
6.16. Afficher les métriques de couverture de code	162
6.17. Configurer les rapports Clover dans Jenkins	163
6.18. Tendence de couverture de code Clover	164
6.19. Utilisation de conventions de nommage orientées métier pour des tests JUnit	165
6.20. Installer le plugin HTML Publisher	165
6.21. Publier les rapports HTML	166
6.22. Jenkins affiche un lien spécial sur la page d'accueil de la tâche de build pour votre rapport	166
6.23. Le plugin DocLinks vous permet d'archiver des documents HTML et non-HTML	167
6.24. Préparer un script de test de performance dans JMeter	169
6.25. Préparer un script de tests de performance dans JMeter	171
6.26. Mise en place du build de performance pour s'exécuter chaque nuit à minuit	172
6.27. Les tests de performance peuvent demander de grandes quantités de mémoire	172
6.28. Configurer le plugin Performance dans votre tâche de build	173
6.29. Le plugin Jenkins Performance garde une trace des temps de réponse et des erreurs	173
6.30. Vous pouvez aussi visualiser les résultats de performance par requête	175
7.1. Activer la sécurité dans Jenkins	180
7.2. La page de connexion Jenkins	181
7.3. La liste des utilisateurs connus de Jenkins	182
7.4. Afficher les builds auxquels un utilisateur participe	182
7.5. Créer un nouveau compte utilisateur en s'enregistrant	183
7.6. Synchroniser les adresses email	183
7.7. Vous pouvez aussi gérer les utilisateurs Jenkins depuis la page de configuration Jenkins.....	184
7.8. La base de données des utilisateurs de Jenkins	184
7.9. Configurer LDAP dans Jenkins	185
7.10. Utiliser des groupes LDAP dans Jenkins	186
7.11. Sélectionner le domaine de sécurité	188
7.12. Utiliser Atlassian Crowd comme domaine de sécurité Jenkins	189
7.13. Utiliser Atlassian Crowd comme domaine de sécurité Jenkins	189
7.14. Utiliser les groupes Atlassian Crowd dans Jenkins	190
7.15. Utiliser des scripts personnalisés pour gérer l'authentification	190
7.16. Sécurité basée sur une matrice	192
7.17. Configurer un administrateur	193
7.18. Configurer les autres utilisateurs	193
7.19. Sécurité basée sur le projet	196
7.20. Configurer la sécurité basée sur le projet	197

7.21. Voir un projet	197
7.22. Configurer les permissions de droit de lecture étendus	198
7.23. Configurer la sécurité basée sur les rôles	198
7.24. Le menu de configuration Gérer les rôles	199
7.25. Gérer les rôles globaux	199
7.26. Gérer les rôles de projets	200
7.27. Assigner des rôles à des utilisateurs	200
7.28. Configurer le plugin Audit Trail	201
7.29. Mettre en place l'historique des configurations de tâches	202
7.30. Présentation de l'historique de configuration des tâches	203
7.31. Voir les différences dans l'historique de configuration des tâches	203
8.1. Configurer les notifications par email	205
8.2. Configurer les notifications par email avancées	207
8.3. Configurer les déclencheurs de notification par email	209
8.4. Message de notification personnalisé	210
8.5. Revendiquer un build échoué	211
8.6. Flux RSS dans Jenkins	212
8.7. Créer une vue radar	213
8.8. Afficher une vue radar	214
8.9. Installation des plugins Jenkins de messagerie instantanée	215
8.10. Jenkins nécessite son propre compte de messagerie instantanée	215
8.11. Mise en place de notifications de base Jabber dans Jenkins	216
8.12. Configuration avancée Jabber	217
8.13. Messages Jenkins Jabber en action	219
8.14. Installation des plugins Jenkins IRC	220
8.15. Configuration avancée des notifications par IRC	221
8.16. Configuration avancée de notifications par IRC pour une tâche de build	222
8.17. Messages de notification par IRC en action	223
8.18. Notifications Jenkins dans Eclipse	224
8.19. Connexion de Jenkins dans NetBeans	225
8.20. Lancement de Jenkins Tray Application	226
8.21. Exécution de Jenkins Tray Application	227
8.22. Créer un service Notifo pour votre instance Jenkins	228
8.23. Configurer les notifications via Notifo dans votre tâche de build Jenkins	229
8.24. Recevoir une notification via Notifo sur un iPhone	229
8.25. Utiliser l'application iPhone Hudson Helper	230
8.26. Envoyer des notifications SMS via une passerelle SMS	231
8.27. Recevoir des notifications via SMS	232
8.28. Configurer les règles de Jenkins Sounds dans une tâche de build	233
8.29. Configurer Jenkins Sounds	233
8.30. Configurer Jenkins Speaks	234
8.31. Un Nabaztag	235
8.32. Configurer votre Nabaztag	236

9.1. C'est facile de configurer les règles Checkstyle avec Eclipse	240
9.2. Configurer les règles PMD dans Eclipse	243
9.3. Générer les rapports de qualité de code dans un build Maven	250
9.4. Configurer le plugin violation pour un projet free-style	251
9.5. Les violations au cours du temps	251
9.6. Les violations pour un build particulier	252
9.7. Configurer le plugin de violations pour un projet free-style	253
9.8. Configurer le plugin Violations pour un projet Maven.	254
9.9. Les tâches de build Maven de Jenkins comprennent les structures multi-modules de Maven	254
9.10. Activer le plugin Violations pour un module individuel	255
9.11. Installer les plugins Checkstyle et Static Analysis Utilities.	256
9.12. Configurer le plugin Checkstyle	257
9.13. Afficher les tendances Checkstyle	257
9.14. Un nuage de point couverture/complexité.	258
9.15. Vous pouvez cliquer sur n'importe quel point du graphique pour poursuivre l'enquête	259
9.16. Configurer le plugin Task Scanner est simple	260
9.17. Le graphique de tendances des tâches ouvertes	260
9.18. Rapport de qualité de code par Sonar.	261
9.19. Jenkins et Sonar	262
9.20. Configurer Sonar dans Jenkins	263
9.21. Configurer Sonar dans une tâche de build	264
9.22. Planifier les builds Sonar	264
10.1. Créer une tâche de build paramétrée	268
10.2. Ajouter un paramètre à la tâche de build	268
10.3. Ajouter un paramètre à la tâche de build	269
10.4. Démonstration d'un paramètre de build	269
10.5. Ajouter un paramètre à la tâche de build Maven	270
10.6. Différents types de paramètres sont disponibles	271
10.7. Configurer un paramètre Choix	271
10.8. Configurer un paramètre Run	272
10.9. Configurer un paramètre Fichier	272
10.10. Ajouter des paramètres pour réaliser un build à partir d'un tag Subversion	273
10.11. Réaliser un build à partir d'un tag Subversion	273
10.12. Configurer un paramètre pour un tag Git	274
10.13. Réaliser un build à partir d'un tag Git	275
10.14. Jenkins stocke les valeurs des paramètres utilisées pour chaque build	276
10.15. Tâche de build paramétré pour des tests unitaires	277
10.16. Ajouter un déclencheur paramétré à une tâche de build	277
10.17. La tâche de build que vous déclenchez doit aussi être une tâche paramétrée	278
10.18. Passer un paramètre prédéfini à une tâche de build paramétré	279
10.19. Créer une tâche de build multiconfiguration	280
10.20. Ajouter un axe à un build multiconfiguration	280

10.21. Définir un axe de noeuds esclave	281
10.22. Définir un axe de versions de JDK	282
10.23. Définir un axe spécifique à l'utilisateur	282
10.24. Résultats de build multiconfiguration	283
10.25. Mettre en place un filtre de combinaison	284
10.26. Résultats de build utilisant un filtre de combinaison	285
10.27. Une tâche générée avec le Maven Jenkins plugin	287
10.28. Tâche générée jenkins-master	288
10.29. Configuration du plugin Jenkins pour Artifactory	292
10.30. Déclencher plusieurs autres builds après une tâche de build	294
10.31. Un graphe de dépendance de tâche de build	295
10.32. Configurer une jonction dans la tâche de build phoenix-web-tests	296
10.33. Un graphe de dépendance de tâche de build plus compliqué	296
10.34. Ajouter un nouveau verrou	297
10.35. Configurer une tâche de build pour utiliser un verrou	298
10.36. Configurer une release Maven en utilisant le plugin M2Release	299
10.37. L'option de menu Perform Maven Release	300
10.38. Effectuer une release Maven dans Jenkins	301
10.39. Ajouter une étape de build "Copier des artefacts d'un autre projet"	302
10.40. Exécuter des tests web sur un fichier WAR copié	304
10.41. Copier à partir d'un build multiconfiguration	305
10.42. Tâches de build dans le processus de promotion	306
10.43. Configurer un processus de promotion de build	307
10.44. Configurer un processus manuel de promotion de build	308
10.45. Voir les détails d'une promotion de build	309
10.46. Utiliser fingerprints dans le processus de promotion de build	310
10.47. Récupérer le fichier WAR depuis la tâche de build amont	311
10.48. Archiver le fichier WAR dans la tâche aval	311
10.49. Récupérer le fichier WAR depuis la tâche d'intégration	311
10.50. Nous avons besoin de déterminer le fingerprint du fichier WAR que nous utilisons	312
10.51. Récupérer le dernier fichier WAR promu	312
10.52. Les builds promus sont indiqués par une étoile dans l'historique de build	313
10.53. Rapport sur l'agrégation des résultats de test	313
10.54. Visualisation des résultats de tests agrégés	314
10.55. Configurer une étape manuelle dans le pipeline de build	315
10.56. Créer une vue Build Pipeline	315
10.57. Configurer une vue Build Pipeline	316
10.58. Un Pipeline de Build en action	317
11.1. Gérer les noeuds esclaves	320
11.2. Créer un nouveau noeud esclave	321
11.3. Créer un noeud esclave Unix	322
11.4. Mettre un esclave hors-ligne lorsqu'il est inactif	323
11.5. Configurer l'emplacement des outils	324

11.6. Votre nouveau noeud esclave en action	325
11.7. Créer un noeud esclave pour JNLP	326
11.8. Lancer un esclave via Java Web Start	326
11.9. L'agent esclave Jenkins en action	327
11.10. L'esclave Jenkins échouant à la connexion au maître	327
11.11. Configurer le port de l'esclave Jenkins	328
11.12. Installer l'esclave Jenkins en tant que service Windows	328
11.13. Gérer le service Windows Jenkins	328
11.14. Permettre à Jenkins de contrôler un esclave Windows comme un service Windows	330
11.15. Exécuter une tâche de build sur un noeud esclave particulier	331
11.16. Jenkins surveille proactivement vos agents de build	333
11.17. Vous gérez vos instances EC2 en utilisant la console de gestion Amazon AWS	334
11.18. Configurer un esclave Amazon EC2	335
11.19. Configurer un esclave Amazon EC2	336
11.20. Créer une nouvelle image Amazon EC2	337
11.21. Ajouter un nouvel esclave Amazon EC2 manuellement	338
12.1. Une chaîne simple de déploiement automatisé	347
12.2. Copier l'artefact binaire à déployer	348
12.3. Déployer sur Tomcat avec le plugin Deploy	348
12.4. Ajouter un paramètre “Build selector for Copy Artifact”	350
12.5. Configurer le sélecteur de paramétrage de build	350
12.6. Spécifier où trouver les artefacts à déployer	351
12.7. Choix du build à redéployer	351
12.8. Utiliser l'option “Specified by permalink”	352
12.9. Utiliser un build spécifique	352
12.10. Utiliser un dépôt d'entreprise Maven	353
12.11. Déployer un artefact depuis un dépôt Maven	356
12.12. Préparer le WAR à déployer	356
12.13. Configurer un hôte distant	357
12.14. Déployer des fichiers vers un hôte distant dans la section build	358
12.15. Déployer des fichiers vers un hôte distant depuis les actions réalisées après le build	359
13.1. Suppression des anciens builds	361
13.2. Supprimer les anciens builds — options avancées	362
13.3. Voir l'utilisation d'espace disque	363
13.4. Affichage de l'utilisation disque d'un projet	363
13.5. Affichage de l'espace disque d'un projet au cours du temps	364
13.6. Tâches de build Maven—options avancées	364
13.7. Statistiques de charge Jenkins	366
13.8. Le plugin Jenkins Monitoring	367
13.9. Le dossier des builds	368
13.10. Le plugin Jenkins Backup Manager	370
13.11. Configurer Jenkins Backup Manager	370
13.12. Configurer le plugin Thin Backup	371

13.13. Restaurer une configuration précédente	372
13.14. Recharger la configuration à partir du disque	373
13.15. Jenkins vous informe si vos données ne sont pas compatibles avec la version actuelle.....	374
13.16. Gestion de configuration périmée	375
A.1. Un projet contenant des classes de tests nommées librement	382

Copyright

Copyright © 2010 John Ferguson Smart

Version imprimée publiée par O'Reilly Media, 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Version en ligne publiée par Wakaleo Consulting, 111 Donald Street, Karori, Wellington 6012, New Zealand.

Ce travail est diffusé sous licence Creative Commons Attribution-Noncommercial-No Derivative Works 3.0 United States. Pour plus d'informations au sujet de cette licence, voir <http://creativecommons.org/licenses/by-nc-nd/3.0/us/>. Vous êtes libre de partager, copier, distribuer, afficher et exécuter les travaux sous les conditions suivantes :

- Vous devez attribuer le travail à John Ferguson Smart
- Vous ne devez pas utiliser ce travail à des fins commerciales.
- Vous ne devez pas modifier, transformer, ou vous baser sur ce travail.

Java™ et tous les logos et marques basés sur Java sont des marques commerciales ou des marques déposées de Sun Microsystems, Inc., aux États-Unis et dans d'autres pays.

Eclipse™ est une marque de l'Eclipse Foundation, Inc., aux États-Unis et dans d'autres pays.

Apache et le logo plume d'Apache sont des marques de l'Apache Software Foundation.

Linux® est une marque déposée de Linus Torvalds aux États-Unis et dans d'autres pays.

Beaucoup d'appellations utilisées par les fabricants et les vendeurs pour distinguer leurs produits sont des marques déposées. Lorsque ces appellations apparaissent dans ce livre, et que Wakaleo Consulting était au courant d'une marque déposée, les désignations ont été inscrites en majuscules ou avec des initiales en majuscule.

Bien que toutes les précautions aient été prises lors de la préparation de ce livre, l'éditeur et l'auteur n'assument aucune responsabilité pour les erreurs ou omissions, ou pour les dommages résultant de l'utilisation de l'information contenue dans ce document.

Avant-propos

Kohsuke Kawaguchi

Il y a 7 ans, j'écrivais la première ligne de code qui a démarré le projet aujourd'hui connu sous le nom de Jenkins, et qui s'appelait à l'origine Hudson. J'avais l'habitude d'être celui qui cassait le build¹, j'avais donc besoin d'un programme qui détecte mes erreurs avant que mes collègues ne le fassent. C'était alors un outil simple qui faisait une chose simple. Cependant, cet outil a rapidement évolué et j'aime à penser aujourd'hui qu'il est devenu le serveur d'intégration continue le plus répandu sur le marché, englobant un large écosystème de plugins, des distributions commerciales, du Jenkins-as-a-Service hébergé, des groupes utilisateurs, des rencontres utilisateurs, des formations, etc.

Comme la plupart de mes autres projets, celui-ci a été rendu open source dès sa création. Au cours de sa vie, il a reposé essentiellement sur l'aide et l'amour d'autres personnes, sans lesquelles ce projet ne serait pas dans son état actuel. Durant cette période, j'ai aussi appris une chose ou deux sur le fonctionnement des projets open source. De par cette expérience, j'ai constaté que les gens ignorent souvent qu'il y a plusieurs façons de contribuer à un projet open source, et qu'écrire du code n'en est qu'une parmi d'autres. On peut en parler autour de soi, aider les autres utilisateurs, organiser des conférences, et bien sûr, on peut rédiger de la documentation.

En cela, John est une personne importante au sein de la communauté Jenkins, même s'il n'a pas fourni de code, il rend Jenkins plus accessible aux nouveaux utilisateurs. Par exemple, il a un blog populaire qui est suivi par de nombreuses personnes, sur lequel il parle régulièrement des pratiques liées à l'intégration continue, ainsi que d'autres sujets liés au développement logiciel. Il a vraiment un talent pour expliquer les choses afin que même des utilisateurs néophytes puissent les comprendre, ce qui est souvent difficile pour des gens comme moi qui développent Jenkins jour après jour. Il est aussi connu pour ses formations, dont Jenkins fait partie. C'est encore une autre façon par laquelle il rend Jenkins accessible à davantage de personnes. Il a clairement une passion pour évangéliser de nouvelles idées et enseigner à ses pairs développeurs comment être plus productifs.

Ces temps-ci je consacre mon temps chez CloudBees à la version open source de Jenkins ainsi qu'à la version pro sur laquelle nous construisons des plugins, et à faire fonctionner Jenkins dans les clouds privés et publics via le service DEV@cloud de CloudBees. Avec ce rôle, j'ai maintenant une plus grande interaction avec John qu'auparavant, et mon respect pour sa passion n'a fait qu'augmenter.

Je fus donc véritablement ravi qu'il prenne en charge l'immense tâche que représente l'écriture d'un livre sur Jenkins. Cela donne une formidable vue d'ensemble sur les principaux ingrédients de l'intégration continue. En plus, à titre personnel, vu que l'on me demande souvent s'il y a un livre sur Jenkins, je peux enfin répondre à cette question par l'affirmative ! Mais plus important encore, ce livre reflète, entre autre, sa passion et sa longue expérience dans l'enseignement de Jenkins. Mais ne me croyez pas sur parole, vous devrez le lire pour le voir par vous-même.

¹Note des traducteurs : nous n'avons pas traduit le terme build car nous n'avons pas trouvé de traduction pertinente et parce que ce terme reste majoritairement utilisé dans les métiers des TI.

Préface

1. Audience

Ce livre est destiné à des lecteurs relativement techniques, bien qu'aucune expérience préalable de l'Intégration Continue ne soit requise. Vous découvrez peut-être l'Intégration Continue, et désirez connaître les bénéfices que cela pourrait apporter à votre équipe de développement. Ou, vous pourriez déjà utiliser Jenkins ou Hudson, et vous voudriez découvrir comment emmener plus loin votre infrastructure d'Intégration Continue.

Une partie importante de ce livre traite de Jenkins dans le contexte de projets Java ou liés à une JVM. Cependant, si vous utilisez une autre pile technologique, ce livre devrait vous donner de bonnes bases en ce qui concerne l'Intégration Continue avec Jenkins. La construction de projets utilisant des technologies non-Java est abordée, comme Grails, Ruby on Rails et .Net. De plus, plusieurs sujets, comme la configuration générale, la notification, les builds distribués et la sécurité sont applicables quel que soit le langage que vous utilisez.

2. Organisation du livre

L'Intégration Continue ressemble à beaucoup d'autres choses : plus vous investissez dessus, plus vous en tirez de bénéfices. Bien qu'une configuration d'Intégration Continue basique produira tout de même des améliorations dans le fonctionnement de votre équipe, il est aussi très avantageux d'assimiler et d'implémenter graduellement quelques-unes des techniques avancées. Dans ce but, ce livre est organisé comme un trek progressif dans le monde de l'Intégration Continue avec Jenkins, allant du plus simple au plus avancé. Dans le premier chapitre, nous effectuerons un balayage de tout ce qui fait Jenkins, sous la forme d'une promenade à haut-niveau. Ensuite, nous progresserons dans l'installation et la configuration de votre serveur Jenkins et expliquerons comment configurer des tâches de build basiques. Une fois que nous aurons maîtrisé les bases, nous explorerons des sujets plus avancés, en passant par les tests automatisés, la sécurité, des techniques avancées de notification et la mesure et les rapports sur les métriques de qualité de code. Ensuite, nous passerons à des techniques de construction plus poussées comme les matrix builds, les builds distribués et l'IC basée sur le Cloud, avant de parler de l'implémentation du déploiement continu avec Jenkins. Et enfin, nous traiterons des astuces pour la maintenance de votre serveur Jenkins.

3. Jenkins ou Hudson?

Comme nous l'évoquons dans l'introduction, Jenkins était originellement, et jusqu'à récemment, connu sous le nom de Hudson. En 2009, Oracle racheta Sun et hérita de la base de code de Hudson. Début 2011, des tensions entre Oracle et la communauté open source atteignirent un point de rupture et le projet se scinda en deux entités distinctes : Jenkins, conduit par la plupart des développeurs originels de Hudson, et Hudson, qui resta sous le contrôle d'Oracle.

Comme le titre le suggère, ce livre se concentre principalement sur Jenkins. Toutefois, une bonne partie de ce livre fut initialement écrit avant la scission, et les produits restent très similaires. Ainsi, bien que les exemples et les illustrations se réfèrent habituellement à Jenkins, presque tout ce qui est discuté s'appliquera aussi à Hudson.

4. Conventions sur les polices

Ce livre suit certaines conventions pour l'utilisation des polices. Comprendre ces conventions d'emblée facilitera l'utilisation de ce livre.

Italique

Utilisée pour les noms de fichiers, extensions de fichiers, URLs, noms d'applications, emphases et les nouveaux termes lorsqu'ils sont introduits pour la première fois.

Chasse fixe

Utilisée pour les noms de classes Java, les méthodes, variables, propriétés, types de données, éléments de base de données et extraits de code apparaissant dans le texte.

Chasse fixe en gras

Utilisée pour les commandes à entrer dans la ligne de commande et pour mettre en valeur du code nouveau introduit dans un exemple fonctionnel.

Chasse fixe en italique

Utilisée pour annoter la sortie.

5. Conventions de ligne de commande

De temps en temps, ce livre traite des instructions en ligne de commande. Quand c'est le cas, la sortie produite par la console (e.g. les invites de commande ou la sortie écran) est affichée en caractères normaux, et les commandes (ce que vous tapez) sont écrites en **gras**. Par exemple :

```
$ ls -al
total 168
drwxr-xr-x  16 johnsmart  staff    544 21 Jan 07:20 .
drwxr-xr-x+ 85 johnsmart  staff   2890 21 Jan 07:10 ..
-rw-r--r--   1 johnsmart  staff    30 26 May 2009 .owner
-rw-r--r--@  1 johnsmart  staff   1813 16 Apr 2009 config.xml
drwxr-xr-x 181 johnsmart  staff   6154 26 May 2009 fingerprints
drwxr-xr-x  17 johnsmart  staff    578 16 Apr 2009 jobs
drwxr-xr-x   3 johnsmart  staff    102 15 Apr 2009 log
drwxr-xr-x  63 johnsmart  staff   2142 26 May 2009 plugins
-rw-r--r--   1 johnsmart  staff    46 26 May 2009 queue.xml
-rw-r--r--@  1 johnsmart  staff    64 13 Nov 2008 secret.key
-rw-r--r--   1 johnsmart  staff  51568 26 May 2009 update-center.json
drwxr-xr-x   3 johnsmart  staff    102 26 May 2009 updates
drwxr-xr-x   3 johnsmart  staff    102 15 Apr 2009 userContent
drwxr-xr-x  12 johnsmart  staff    408 17 Feb 2009 users
drwxr-xr-x  28 johnsmart  staff    952 26 May 2009 war
```

Lorsque nécessaire, l'antislash à la fin de la ligne est utilisé pour indiquer un saut de ligne : vous pouvez taper l'ensemble sur une seule ligne (sans l'antislash) si vous préférez. N'oubliez pas d'enlever le caractère “>” au début des lignes concernées — c'est un caractère de l'invite Unix :

```
$ wget -O - http://jenkins-ci.org/debian/jenkins-ci.org.key \  
> | sudo apt-key add -
```

Pour des raisons de cohérence, à moins que nous ne traitions une question spécifique à Windows, nous utiliserons des invites de commande de style Unix (le signe dollar, “\$”), comme montré ici :

```
$ java -jar jenkins.war
```

ou :

```
$ svn list svn://localhost
```

Cependant, à moins qu'on indique le contraire, les utilisateurs Windows peuvent utiliser ces commandes depuis la console de commande Windows :

```
C:\Documents and Settings\Owner> java -jar jenkins.war
```

ou :

```
C:\Documents and Settings\Owner> svn list svn://localhost
```

6. Contributeurs

Ce livre n'a pas été écrit par une seule personne. Cela a plutôt été un effort collaboratif impliquant plusieurs personnes jouant différents rôles. En particulier, les personnes suivantes ont généreusement donné de leur temps, de leur savoir et de leur talent d'écriture pour rendre ce livre meilleur :

- Evgeny Goldin est un ingénieur logiciel né en Russie vivant en Israël. Il est lead developer chez Thomson Reuters où il est responsable d'un certain nombre d'activités, dont certaines sont directement liées à Maven, Groovy, et des outils de build comme Artifactory et Jenkins. Il possède une vaste expérience dans un panel assez large de technologies, dont Perl, Java, JavaScript et Groovy. Les outils de build et les langages dynamiques sont les sujets favoris de Evgeny, à propos desquels il écrit, présente ou blogue souvent. Il écrit actuellement pour GroovyMag, Methods & Tools et développe sur deux projets open source qu'il a créés : Maven-plugins¹ et GCommons². Il blogue sur <http://evgeny-goldin.com/blog> et peut être trouvé sur Twitter comme @evgeny_goldin.

Evgeny a réalisé une section sur la génération automatique de tâche de build Maven dans Chapter 10, Builds avancés.

- Matthew McCullough est un vétéran énergique de 15 ans d'expérience dans le développement logiciel d'entreprise, l'éducation open source, et co-fondateur de LLC, une entreprise de consulting de Denver. Matthew est actuellement formateur pour GitHub.com, auteur de la série Git Master Class pour O'Reilly, conférencier à plus de 30 conférences nationales et internationales, auteur de 3 des 10 plus importantes RefCards DZone, et président du Groupe Utilisateur d'Open Source

de Denver. Ses sujets de recherche sont aujourd'hui concentrés autour de l'automatisation de projet : outils de build (Maven, Leiningen, Gradle), contrôle de version distribué (Git), Intégration Continue (Jenkins) et les métriques de qualité (Sonar). Matthew réside à Denver, dans le Colorado, avec sa magnifique femme et ses deux jeunes filles, qui sont actives dans pratiquement toutes les activités d'extérieur que le Colorado puisse offrir.

Matthew a écrit la section sur l'intégration de Git à Jenkins dans Chapter 5, Configurer vos tâches de Build.

- Juven Xu est un ingénieur logiciel venant de Chine qui travaille pour Sonatype. Membre actif de la communauté open source et expert Maven reconnu, Juven était responsable de la traduction chinoise de Maven: The Definitive Guide et aussi d'un livre original de référence chinois sur Maven. Il travaille aussi actuellement à la traduction chinoise du présent livre.

Juven a écrit la section sur les notifications IRC dans Chapter 8, Notification.

- Rene Groeschke est ingénieur logiciel chez Cassidian Systems, connu précédemment comme EADS Deutschland GmbH, et aussi un enthousiaste de l'open source. ScrumMaster certifié avec 7 ans d'expérience comme programmeur dans plusieurs projets Java d'entreprise, il se concentre plus particulièrement sur les méthodes Agiles comme l'Intégration Continue et le développement guidé par les tests. En dehors de son travail quotidien, l'Université de l'Education d'Entreprise de Friedrichshafen lui permet de faire passer le mot à propos de scrum et de sujets connexes en donnant des cours aux étudiants en informatique.

Rene a réalisé la section sur la construction de projets avec Gradle dans Chapter 5, Configurer vos tâches de Build.

6.1. Les traducteurs du présent livre en français

- Alexis Morelle³
- Alexis Thomas⁴
- Antoine Meausoone⁵
- Baptiste Mathus⁶
- Joseph Pachod⁷
- Emmanuel Hugonnet⁸
- Frederic Bouquet⁹
- Jeff Maury¹⁰
- Kevin Lecouvey¹¹
- Michael Pailloncy¹²

- Nacef Labidi¹³
- Thibault Richard¹⁴

7. L'équipe de revue

Le processus de revue technique de ce livre a été un peu différent de l'approche utilisée pour la plupart des livres. Plutôt que d'avoir un ou deux relecteurs techniques pour le livre entier une fois que le livre aurait été en passe d'être terminé, une équipe de volontaires de la communauté Jenkins, incluant plusieurs développeurs clé de Jenkins, a relu les chapitres au fil de leur écriture. Cette équipe de relecture était composée des personnes suivantes : Alan Harder, Andrew Bayer, Carlo Bonamico, Chris Graham, Eric Smalling, Gregory Boissinot, Harald Soevik, Julien Simpson, Juven Xu, Kohsuke Kawaguchi, Martijn Verberg, Ross Rowe et Tyler Ballance.

8. Sponsors du livre

Ce livre n'aurait pas été possible sans l'aide de plusieurs organisations qui étaient désireuses d'assister et de subventionner le processus d'écriture.

8.1. Wakaleo Consulting

Wakaleo Consulting¹⁵ est une entreprise de consulting qui aide les organisations à optimiser leur processus de développement logiciel. Dirigée par John Ferguson Smart, auteur de ce livre et de Java Power Tools¹⁶, Wakaleo Consulting fournit des services de consulting, de formation et de mentoring dans le développement Agile Java et dans les pratiques de test, l'optimisation de cycle vie de développement logiciel et les méthodes Agiles.

Wakaleo aide les entreprises avec des formations et de l'assistance dans des domaines comme l'Intégration Continue, l'automatisation de build, le développement guidé par les tests, les tests Web automatisés et le code propre, en utilisant des outils open source tels que Maven, Jenkins, Selenium 2 et Nexus. Wakaleo Consulting donne aussi des formations publiques et sur site autour de l'Intégration Continue et le Déploiement Continu, l'automatisation de build, les pratiques de codage propre, le développement guidé par les tests (TDD) et le Behavior-Driven Development, incluant des cours Certified Scrum Developer (CSD).

8.2. CloudBees

CloudBees¹⁷ est la seule entreprise spécialisée pour offrir un service de gestion du cycle de vie, depuis le développement jusqu'au déploiement, d'applications Java web dans le cloud. L'entreprise est aussi l'experte mondiale sur l'outil d'intégration continue Jenkins/Hudson.

¹⁵ <http://www.wakaleo.com>

¹⁶ <http://oreilly.com/catalog/9780596527938>

¹⁷ <http://www.cloudbees.com>

Le créateur de Jenkins/Hudson Kohsuke Kawaguchi dirige une équipe d'experts CloudBees à travers le monde. Ils ont créé Nectar, une version supportée et améliorée de Jenkins disponible par abonnement. Si vous dépendez de Jenkins pour des processus logiciels critiques, Nectar fournit une version hautement testée, stable et totalement supportée de Jenkins. Cela inclut aussi des fonctionnalités disponibles seulement dans Nectar comme le scaling automatique de machines virtuelles VMWare.

Si vous êtes prêts à explorer la puissance de l'Intégration Continue dans le cloud, CloudBees propose Jenkins/Hudson comme brique de sa plateforme de build DEV@cloud. Vous pouvez démarrer avec Jenkins instantanément et vous pouvez monter en puissance comme vous le souhaitez — pas de gros investissements dès le début dans vos serveurs de build, plus de capacités limitées pour vos builds et plus de soucis de maintenance. Une fois qu'une application est prête à partir, vous pouvez la déployer sur l'offre Platform as a Service CloudBees RUN@cloud en seulement quelques clics.

Avec les services CloudBees DEV@cloud et RUN@cloud, vous n'avez pas à vous soucier des serveurs, machines virtuelles ou des équipes informatique. Et avec Nectar, vous profitez du Jenkins le plus puissant, stable et supporté existant.

8.3. Odd-e

Odd-e¹⁸ est une entreprise basée en Asie qui construit des produits de façon innovante et aide les autres à faire de même. L'équipe est composée de coaches expérimentés et de développeurs produit travaillant dans des valeurs scrum, agile, lean, et artisan du logiciel, et l'entreprise est structurée de la même façon. Par exemple, Odd-e n'a pas d'organisation hiérarchique de responsables prenant les décisions pour les autres. Au lieu de cela, les individus s'organisent eux-mêmes et utilisent leurs talents pour constamment améliorer leurs compétences. L'entreprise propose des formations et du coaching suivi pour aider les autres à chercher et développer une meilleure façon de travailler.

Ce n'est pas le travail mais les valeurs qui lient Odd-e. Ses membres adorent construire du logiciel, accordent davantage d'importance à l'apprentissage et la contribution qu'à la maximisation du profit, et ont décidé de supporter le développement open source en Asie.

9. Utilisation des exemples de code

Ce livre est un livre open source, publié sous licence Creative Commons. Ce livre a été écrit en DocBook, en utilisant XmlMind. Le code source du livre est disponible à <http://www.github.org/wakaleo/jenkins-the-definitive-guide>.

Les exemples de projets Jenkins de ce livre sont open source et librement disponibles en ligne — référez-vous à la page web du livre à <http://www.wakaleo.com/books/jenkins-the-definitive-guide> pour plus de détails.

Ce livre existe pour vous aider à faire votre travail. En général, vous pouvez utiliser le code de ce livre dans vos programmes et documentation. Vous n'avez pas besoin de nous contacter pour obtenir

¹⁸ <http://www.odd-e.com>

une permission à moins que vous ne reproduisiez une part importante du code. Par exemple, écrire un programme qui utilise plusieurs extraits de code de ce livre ne nécessite pas de permission. Vendre ou distribuer un CD-ROM d'exemples des livres O'Reilly requiert une permission. Répondre à une question en citant ce livre et utiliser des exemples de code du livre ne nécessite pas de permission. Incorporer une part importante d'exemples de code de ce livre dans votre documentation produit requiert une permission.

Nous apprécions, mais ne requérons pas, une référence. Une référence inclut habituellement le titre, l'auteur, l'éditeur et l'ISBN. Par exemple : "Jenkins: The Definitive Guide par John Ferguson Smart (O'Reilly). Copyright 2011 John Ferguson Smart, 978-1-449-30535-2."

Si vous pensez que votre utilisation des exemples de code tombe en dehors d'un usage raisonnable (NdT : fair-use) des permissions données ci-dessus, n'hésitez pas à nous contacter à <permissions@oreilly.com>.

10. Safari® Books Online

Note

Safari Books Online est une bibliothèque numérique à la demande qui vous permet de chercher facilement parmi 7500 livres et vidéos de référence de technologie ou de création pour trouver les réponses dont vous avez besoin rapidement.

Avec un abonnement, vous pouvez lire n'importe quelle page ou regarder n'importe quelle vidéo de notre bibliothèque en ligne. Lisez les livres depuis votre téléphone portable ou vos périphériques mobiles. Accédez aux nouveaux titres avant qu'ils soient disponibles en impression papier, et obtenez des accès exclusifs à des manuscrits en développements et postez des retours aux auteurs. Copiez et collez des exemples de code, organisez vos favoris, téléchargez des chapitres, mettez des sections clés en favoris, créez des notes, imprimez des pages et bénéficiez d'une tonne d'autres fonctionnalités vous faisant gagner du temps.

O'Reilly Media a déposé ce livre sur le service Safari Books Online. Pour obtenir l'accès numérique complet à ce livre et à d'autres dans des sujets similaires de chez O'Reilly et d'autres éditeurs, enregistrez-vous gratuitement chez <http://my.safaribooksonline.com>¹⁹.

11. Comment nous contacter

Merci d'adresser vos commentaires et vos questions concernant ce livre à l'éditeur :

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

¹⁹ <http://my.safaribooksonline.com/?portal=oreilly>

800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

Nous avons une page web pour ce livre, sur laquelle nous listons les erreurs, les exemples et toute autre information additionnelle. Vous pouvez accéder à cette page à :

<http://www.oreilly.com/catalog/9781449305352>

Pour commenter ou poser des questions techniques à propos de ce livre, envoyez un email à :

[<bookquestions@oreilly.com>](mailto:bookquestions@oreilly.com)

Pour plus d'information à propos de nos livres, cours, conférences et nouvelles, visitez notre site web à <http://www.oreilly.com>.

Trouvez-nous sur Facebook : <http://facebook.com/oreilly>

Suivez-nous sur Twitter : <http://twitter.com/oreillymedia>

Regardez-nous sur YouTube : <http://www.youtube.com/oreillymedia>

12. Remerciements

Pour commencer et avant tout, à ma merveilleuse femme, et aux garçons, James et William, sans qui, grâce à leur amour, leur soutien et leur tolérance, ce livre n'aurait pas été possible.

J'aimerais remercier Mike Loukides pour avoir travaillé avec moi une fois de plus sur ce projet de livre, et toute l'équipe O'Reilly pour leur haut niveau de travail.

Merci à Kohsuke Kawaguchi d'avoir créé Jenkins, et d'être encore la force motrice derrière ce brillant produit. Merci aussi à François Dechery, Sacha Labourey, Harpreet Singh et le reste de l'équipe CloudBees pour leur aide et leur soutien.

Je suis aussi très reconnaissant envers ceux qui ont consacré du temps et de l'énergie pour contribuer à ce livre : Evgeny Goldin, Matthew McCullough, Juven Xu, et Rene Groeschke.

Un grand merci revient aux relecteurs suivants, qui ont fourni des retours de grande valeur tout le long du processus d'écriture : Alan Harder, Andrew Bayer, Carlo Bonamico, Chris Graham, Eric Smalling, Gregory Boissinot, Harald Soevik, Julien Simpson, Juven Xu, Kohsuke Kawaguchi, Martijn Verberg, Ross Rowe, et Tyler Ballance.

Merci à Andrew Bayer, Martijn Verburg, Matthew McCullough, Rob Purcell, Ray King, Andrew Walker, et beaucoup d'autres, dont les discussions et les retours m'ont fourni l'inspiration et les idées qui ont fait de ce livre ce qu'il est.

Et beaucoup d'autres personnes ont aidé de différentes façons pour enrichir et compléter ce livre tel qu'il n'aurait pu l'être autrement : Geoff et Alex Bullen, Pete Thomas, Gordon Weir, Jay Zimmerman, Tim

O'Brien, Russ Miles, Richard Paul, Julien Simpson, John Stevenson, Michael Neale, Arnaud H ritier, et Manfred Moser.

Et enfin un grand merci aux d veloppeurs et   la communaut  utilisateur de Hudson/Jenkins pour les encouragements et le soutien constants.

Chapter 1. Introduction à Jenkins

1.1. Introduction

L'Intégration Continue, aussi connue sous le terme IC, est l'un des piliers du développement logiciel moderne. En fait, elle est un véritable tournant — quand l'Intégration Continue est mise en place dans une organisation, elle change radicalement la manière dont les équipes pensent le processus de développement. Elle est capable de permettre et d'induire toute une série d'améliorations et de transformations, depuis le build régulier automatisé jusqu'à la livraison continue en production. Une bonne infrastructure d'IC peut fluidifier le processus de développement jusqu'au déploiement, aide à détecter et corriger les bogues plus rapidement, fournit un écran de contrôle très utile aux développeurs mais aussi aux non-développeurs, et poussée à l'extrême, elle permet aux équipes de fournir plus de valeur métier aux utilisateurs finaux. Toute équipe de développement professionnelle, quelque soit sa taille, devrait mettre en œuvre l'IC.

1.2. Les fondamentaux de l'Intégration Continue

A l'époque des projets en V et des diagrammes de Gantt, avant l'introduction des pratiques de l'IC, une équipe de développement dépensait son temps et son énergie sans compter durant la phase amenant à la livraison, appelée Phase d'Intégration. Durant cette phase, toutes les modifications apportées au code par les développeurs ou de petites équipes étaient rassemblées puis agrégées et fusionnées en un produit fonctionnel. Ce travail était dur, intégrant ainsi des mois de modifications qui entraient en conflit. Il était très difficile d'anticiper les problèmes qui allaient surgir, et encore plus de les corriger car cela impliquait de reprendre du code qui avait été écrit des semaines, voire des mois auparavant. Ce douloureux processus, plein de risques et de dangers, amenait souvent des retards significatifs de livraison, des coûts supplémentaires et, au final, des clients mécontents. L'Intégration Continue est née comme une réponse à ces problèmes.

L'Intégration Continue, dans sa forme la plus simple, se compose d'un outil qui surveille les modifications de code dans votre gestionnaire de configuration. Dès qu'un changement est détecté, cet outil va automatiquement compiler et tester votre application. Si la moindre erreur arrive alors l'outil va immédiatement avertir les développeurs afin qu'ils puissent tout de suite corriger le problème.

Mais l'Intégration Continue est bien plus que cela. L'Intégration Continue peut aussi suivre la santé de votre projet en surveillant la qualité du code et les métriques de couverture et ainsi aider à maintenir la dette technique à un niveau bas et à abaisser les coûts de maintenance. Rendre visible publiquement les métriques de qualité de code encourage les développeurs à être fiers de la qualité de leur code et à essayer de toujours l'améliorer. Combinée à une suite de tests d'acceptance automatisés, l'IC peut aussi se transformer en outil de communication en affichant une image claire de l'état des développements en cours. Et elle peut simplifier et accélérer la livraison en vous aidant à automatiser le processus de déploiement, vous permettant de déployer la dernière version de votre application soit automatiquement soit d'un simple clic.

Dans le fond, l'Intégration Continue c'est réduire les risques en fournissant des retours rapides. En premier lieu, de par sa conception, elle permet d'identifier et de corriger les problèmes d'intégration et les regressions plus rapidement ce qui permet de livrer plus facilement et avec moins de bogues. En donnant une meilleure visibilité sur l'état du projet à tous les membres de l'équipe, techniques comme non-techniques, l'Intégration Continue facilite la communication au sein de l'équipe et encourage la collaboration pour résoudre les problèmes ainsi que l'amélioration du processus. Et, en automatisant le processus de déploiement, l'Intégration Continue vous permet de mettre votre logiciel dans les mains des testeurs et des utilisateurs finaux plus rapidement, avec plus de fiabilité et avec moins d'efforts.

Ce concept de déploiement automatisé est important. En effet, si vous poussez ce concept de déploiement automatisé à sa conclusion logique, vous pourriez mettre en production tout build qui passerait sans encombre les tests automatisés nécessaires. Cette pratique de déployer en production tous les builds ayant réussi est appelée communément Déploiement Continu.

Cependant, une approche pure du Déploiement Continu n'est pas toujours souhaitable. Par exemple, de nombreux utilisateurs n'apprécieraient pas d'avoir une nouvelle version disponible plusieurs fois par semaine et préféreraient un cycle de livraison plus prévisible (et transparent). Des considérations commerciales et marketing peuvent aussi entrer en compte pour déterminer quand une nouvelle version doit être déployée.

La notion de Livraison Continue est très proche de cette idée de Déploiement Continu en prenant en compte ces considérations. Lors d'une Livraison Continue tout build qui a réussi à passer les tests automatisés pertinents et les contrôles qualité peut virtuellement être déployé en production au moyen d'un processus lancé par un simple clic, et ainsi se retrouver dans les mains de l'utilisateur final en quelques minutes. Cependant, ce processus n'est pas automatique : c'est au métier, plutôt qu'à l'Informatique de décider quel est le moment opportun pour livrer les dernières modifications.

Ainsi les techniques d'Intégration Continue, et plus particulièrement le Déploiement Continu et la Livraison Continue, permettent d'apporter la valeur à l'utilisateur final plus rapidement. Combien de temps faut-il à votre équipe pour mettre une petite modification du code en production ? Dans ce temps quelle est la part des problèmes qui auraient pu être corrigés plus tôt si vous aviez su quelles modifications faisait Joe au bout du couloir ? Quelle part est prise par le pénible travail des équipes de la qualité pour tester manuellement ? Combien d'étapes manuelles, dont les secrets ne sont connus que de quelques initiés seulement, sont nécessaires à un déploiement ? L'Intégration Continue n'est pas la solution miracle, elle permet de rationaliser certains de ces problèmes.

Mais l'Intégration Continue est tout autant une mentalité que des outils. Pour tirer un maximum de profit de l'IC, une équipe doit adopter un comportement IC. Par exemple, vos projets doivent avoir un build fiable, reproductible et automatisé, sans intervention humaine. La correction des builds en erreur devrait être une priorité absolue, et ne devrait pas être oubliée dans un coin. Le processus de déploiement devrait être automatisé, sans étape manuelle. Et puisque la confiance que vous mettez dans votre serveur d'intégration dépend en grande partie de la qualité de vos tests, l'équipe doit mettre l'accent sur la qualité des tests et des pratiques associées.

Dans ce livre nous verrons comment mettre en œuvre une solution d'Intégration Continue robuste et complète avec Jenkins ou Hudson.

1.3. Introduction à Jenkins (né Hudson)

Jenkins, qui s'appelait à l'origine Hudson, est un outil d'Intégration Continue open source écrit en Java. Bénéficiant d'une part de marché dominante, Jenkins est utilisé par des équipes de toutes tailles, pour des projets dans des langages et des technologies variés, incluant .NET, Ruby, Groovy, Grails, PHP et d'autres, ainsi que Java bien sûr. Qu'est ce qui est à l'origine du succès de Jenkins ? Pourquoi utiliser Jenkins pour votre infrastructure d'IC ?

Tout d'abord, Jenkins est facile à utiliser. L'interface utilisateur est simple, intuitive et visuellement agréable, et Jenkins dans son ensemble a une très petite courbe d'apprentissage. Comme nous le verrons dans le chapitre suivant, vous pouvez démarrer avec Jenkins en quelques minutes.

Cependant Jenkins n'a pas sacrifié puissance ou extensibilité : il est aussi extrêmement flexible et s'adapte facilement à vos moindres désirs. Des centaines d'extensions open source sont disponibles, et de nouvelles apparaissent toutes les semaines. Ces extensions couvrent tout : les outils de gestion de configuration, les outils de build, les métriques de qualité de code, les annonceurs de build, l'intégration avec des systèmes externes, la personnalisation de l'interface utilisateur, des jeux et bien d'autres fonctionnalités encore. Les installer est simple et rapide.

Enfin, mais ce n'est pas négligeable, une bonne part de la popularité de Jenkins vient de la taille et du dynamisme de sa communauté. La communauté Jenkins est immense, dynamique, réactive et c'est un groupe accueillant, avec ses champions passionnés, ses listes de diffusion actives, ses canaux IRC et son blog et son compte twitter très bruyants. Le rythme de développement est très intense, avec des livraisons hebdomadaires comportant les dernières évolutions, corrections et mises à jour des extensions.

Cependant, Jenkins répond tout aussi bien aux attentes des utilisateurs qui ne souhaitent pas mettre à jour toutes les semaines. Pour ceux qui veulent un rythme moins trépidant, il existe une version Long-term Support, ou LTS. Il s'agit d'un ensemble de versions qui sont à la traîne par rapport à la toute dernière en termes de nouveautés mais qui apportent plus de stabilité et moins de changements. Les versions LTS sortent environ tous les trois mois, les correctifs importants y sont intégrés. Ce concept est identique aux versions Ubuntu LTS .

1.4. De Hudson à Jenkins — Un rapide historique

Jenkins est le produit d'un développeur visionnaire, Kohsuke Kawaguchi, qui a commencé ce projet comme un loisir sous le nom d'Hudson à la fin de l'année 2004 alors qu'il travaillait chez Sun. Comme Hudson évoluait au cours des années, il était de plus en plus utilisé par les équipes de Sun pour leurs propres projets. Début 2008, Sun reconnaissait la qualité et la valeur de cet outil et demandait à Kohsuke de travailler à plein temps sur Hudson, commençant ainsi à fournir des services professionnels et du support sur Hudson. En 2010, Hudson était devenu la principale solution d'Intégration Continue avec une part de marché d'environ 70%.

En 2009, Oracle racheta Sun. Vers la fin 2010, des tensions apparurent entre la communauté de développeurs d'Hudson et d'Oracle, dont la source était les problèmes de l'infrastructure Java.net, aggravées par la politique d'Oracle au sujet de la marque Hudson. Ces tensions révélaient de profonds désaccords sur la gestion du projet par Oracle. En effet, Oracle voulait orienter le projet vers un processus de développement plus contrôlé, avec des livraisons moins fréquentes, alors que le cœur des développeurs d'Hudson, mené par Kohsuke, préfère continuer à fonctionner selon le modèle communautaire ouvert, flexible et rapide qui avait si bien fonctionné pour Hudson par le passé.

En Janvier 2011, la communauté des développeurs Hudson vota pour renommer le projet en Jenkins. Par la suite ils migrèrent le code originel d'Hudson vers un nouveau projet Github¹ et y poursuivirent leur travail. La grande majorité des développeurs du cœur et des plugins leva le camp et suivit Kohsuke Kawaguchi et les autres principaux contributeurs dans le camp de Jenkins, où le gros de l'activité de développement peut être observée aujourd'hui.

Après ce fork, la majorité des utilisateurs suivit la communauté des développeurs Jenkins et passa à Jenkins. Au moment où ces lignes sont écrites, les sondages montrent qu'environ 75% des utilisateurs d'Hudson sont passés à Jenkins, 13 % utilisent encore Hudson et 12% utilisent Jenkins et Hudson ou sont en cours de migration vers Jenkins.

Cependant, Oracle et Sonatype (la compagnie derrière Maven et Nexus) ont poursuivi leur travail à partir du code d'Hudson (qui est maintenant lui aussi hébergé chez GitHub à <https://github.com/hudson>), mais selon un axe différent. En effet, les développeurs de Sonatype se sont concentrés sur des modifications dans l'architecture, notamment sur l'intégration avec Maven, sur le framework d'injection de dépendances, et sur l'architecture des plugins.

1.5. Dois-je utiliser Jenkins ou Hudson?

Donc faut-il utiliser Jenkins ou Hudson ? Puisqu'il s'agit ici d'un livre traitant de Jenkins, voici plusieurs raisons pour choisir Jenkins :

- Jenkins est le nouvel Hudson. En fait, Jenkins est simplement le bon vieil Hudson renommé, donc si vous avez apprécié Hudson, vous aimerez Jenkins ! Jenkins utilise le code d'Hudson et l'équipe de développement ainsi que la philosophie du projet sont restées identiques. En résumé, les développeurs initiaux qui ont écrit la plus grande partie du cœur d'Hudson, ont simplement continué en travaillant sur le projet Jenkins après la séparation.
- La communauté Jenkins. Comme de nombreux projets Open Source ayant du succès, la force d'Hudson venait de sa grande et dynamique communauté et de son adoption massive. Les bogues sont détectés (et généralement corrigés) beaucoup plus rapidement, et si par malheur vous rencontrez un souci il y a de fortes chances que quelqu'un d'autre l'ait déjà rencontré ! Si vous avez un problème, écrivez une question sur la liste de diffusion ou sur le canal IRC — vous trouverez sûrement quelqu'un pour vous aider.

¹ <https://github.com/jenkinsci>

- Le rythme de développement intense. Jenkins conserve la fréquence élevée des sorties, typique d'Hudson, et que de nombreux développeurs apprécient. Les nouvelles fonctionnalités, les nouveaux plugins et les corrections de bogues apparaissent hebdomadairement et le temps de correction des bogues est vraiment très court. Si, par contre, vous préférez plus de stabilité, il y a toujours les versions LTS.

Et, pour équilibrer les choses, voici quelques raisons qui peuvent vous faire préférer Hudson :

- Si ça fonctionne, ne le touchez pas. Vous avez déjà un Hudson installé dont vous êtes très satisfait et vous ne ressentez pas le besoin d'installer la dernière version.
- L'intégration professionnelle et les outils Sonatype. Hudson va probablement mettre l'accent sur son intégration avec des outils professionnels comme un annuaire LDAP/Active Directory et les produits de Sonatype tels que Maven 3, Nexus et M2Eclipse, alors que Jenkins sera plus ouvert à des outils concurrents comme Artifactory et Gradle.
- L'architecture des plugins. Si vous avez l'intention d'écrire vos propres plugins Jenkins/Hudson plugins, il vous faut savoir que Sonatype travaille pour proposer une injection de dépendance JSR-330 pour les plugins d'Hudson. Les nouveaux développeurs peuvent trouver cette approche plus facile à utiliser même si cela soulève des questions quant à la compatibilité entre Jenkins et Hudson.

La bonne nouvelle est que quelque soit l'outil que vous utilisiez entre Hudson et Jenkins, ils restent globalement très proches et la plupart des techniques et des astuces présentées dans ce livre seront valables sur les deux. En effet, pour illustrer ce point, de nombreuses captures d'écran dans ce livre font référence à Hudson plutôt qu'à Jenkins.

1.6. Mettre en place l'Intégration Continue au sein de votre organisation

L'Intégration Continue n'est pas une histoire de tout ou rien. En fait, mettre en place l'IC au sein d'une organisation suit un chemin qui vous fera passer par différentes phases. A chacune de ces phases se rattachent des améliorations de l'infrastructure technique, mais aussi, et c'est peut-être le plus important, des améliorations des pratiques et de la culture de l'équipe de développement elle-même. Dans les paragraphes qui vont suivre j'ai essayé d'esquisser chacune de ces phases.

1.6.1. Phase 1 — Pas de serveur de build

Au tout début, l'équipe n'a pas de serveur central de build quelqu'il soit. Le logiciel est construit manuellement sur la machine du développeur, même si cela peut être réalisé par un script Ant ou équivalent. Le code source peut être enregistré dans un gestionnaire de configuration centralisé, mais les développeurs n'ont pas l'habitude de commiter leurs modifications régulièrement. Peu de temps avant la prochaine livraison prévue, un développeur intègre manuellement les modifications, une opération qui produit beaucoup de peine et de souffrance.

1.6.2. Phase 2 — Builds quotidiens

Dans cette phase, l'équipe possède un serveur de build et des builds automatisés sont exécutés régulièrement (généralement la nuit). Ce build compile tout simplement le code car il n'existe pas de tests unitaires pertinents et répétables. En effet, les tests automatisés, même s'ils sont écrits, ne font pas partie du processus de build, et peuvent ne pas s'exécuter correctement du tout. Cependant, maintenant les développeurs commitent leurs modifications régulièrement au moins à la fin de la journée. Si un développeur commite des modifications de code qui entrent en conflit avec le travail d'un autre développeur, le serveur de build avertit l'équipe par mail le lendemain matin. Toutefois, l'équipe ne se sert du serveur de build qu'à titre informatif — ils ne se sentent pas obligés de réparer un build en échec immédiatement, et le build peut rester en échec pendant plusieurs jours sur le serveur de build.

1.6.3. Phase 3 — Builds quotidiens et tests automatisés basiques

L'équipe commence à prendre l'Intégration Continue et les tests au sérieux. Le serveur de build est configuré pour lancer un build dès qu'un nouveau code est commité dans l'outil de gestion de configuration et les membres de l'équipe peuvent facilement voir quelles sont les modifications du code qui correspondent à chaque build et de quel problème elles traitent. A cela s'ajoute le fait que le script de build compile l'application et lance une série de tests unitaires ou d'intégration automatisés. En plus des emails, le serveur de build avertit aussi les membres de l'équipe des problèmes d'intégration en utilisant des canaux plus proactifs tel que la messagerie instantanée. Les builds en échec sont maintenant réparés rapidement.

1.6.4. Phase 4 — Arrivée des métriques

Des métriques de qualité et de couverture de code automatisées sont maintenant mesurées pour aider à évaluer la qualité du code et (jusqu'à un certain point) la pertinence et l'efficacité des tests. Le build mesurant la qualité de code produit aussi automatiquement la documentation de l'API de l'application. Tout ceci permet à l'équipe de maintenir un code de très bonne qualité, alertant les membres de l'équipe en cas de dérive des bonnes pratiques de tests. L'équipe a aussi monté un “build radiator”, un tableau de bord qui présente l'état du projet sur un écran visible de l'ensemble des membres de l'équipe.

1.6.5. Phase 5 — Prendre les tests au sérieux

Les bénéfices de l'Intégration Continue sont étroitement liés à de solides pratiques de test. De nos jours, des techniques comme le développement piloté par les tests sont très utilisées ce qui améliore la confiance que l'on peut avoir dans le résultat d'un build automatisé. L'application n'est plus simplement compilée et testée, mais si les tests passent, elle est automatiquement déployée sur un serveur d'application pour des tests plus complets de bout en bout et des tests de performance.

1.6.6. Phase 6 — Tests d'acceptance automatisés et un déploiement plus automatisé

Le Développement piloté par les tests d'acceptance est utilisé, guidant les efforts de développement et fournissant des rapports de haut niveau sur l'état du projet. Ces tests automatisés profitent des outils issus

du monde du Développement piloté par le comportement (BDD) et du Développement piloté par les tests d'acceptance non seulement comme outils de tests mais aussi comme outils de communication et de documentation, en produisant des rapports de tests utilisant les termes métier qu'un non-développeur peut comprendre. Puisque ces tests de haut-niveau sont automatisés dès le début du processus de développement, ils permettent de savoir très simplement quelles sont les fonctionnalités qui sont réalisées et celles qu'il reste à faire. L'application est automatiquement déployée sur un environnement de test pour l'équipe qualité (QA) soit à chaque modification soit tous les soirs ; une version peut être déployée (ou “promue”) pour des tests de conformité utilisateurs et il est possible de la déployer dans des environnements de production au moyen d'un build lancé manuellement quand les testeurs la considère prête. L'équipe peut aussi utiliser le serveur de build pour revenir en arrière d'une version en cas de catastrophe.

1.6.7. Phase 7— Déploiement Continu

La confiance dans les tests unitaires, d'intégration et d'acceptance automatisés est telle que les équipes peuvent utiliser les techniques de déploiement continu développées dans les phases précédentes pour pousser les modifications directement en production.

La progression entre les niveaux décrite ici est approximative et peut ne pas correspondre exactement à des situations dans le monde réel. Par exemple, vous pouvez tout à fait introduire des tests de l'interface web avant de mettre en places les métriques de qualité de code et les rapports sur la couverture de code. Cependant, cela devrait vous donner une idée globale de la stratégie d'Intégration Continue à mettre en oeuvre dans une organisation du monde réel.

1.7. Et maintenant ?

Tout au long du reste du livre, pendant que nous étudierons les différentes fonctionnalités de Jenkins ainsi que les pratiques associées pour en profiter au maximum, nous verrons comment progresser suivant ces différents niveaux avec Jenkins. Et souvenez-vous, la plupart des exemples de ce livre sont disponibles en ligne (cf. <http://www.wakaleo.com/books/jenkins-the-definitive-guide> pour plus d'informations), vous pouvez donc mettre les mains dans le cambouis vous aussi !

Chapter 2. Vos premiers pas avec Jenkins

2.1. Introduction

Dans ce chapitre, nous allons faire un tour rapide des fonctionnalités clé de Jenkins. Tout d'abord, vous verrez combien il est facile d'installer Jenkins et d'y configurer votre première de build automatique. Nous n'entrerons pas dans les détails - ceux-ci vous seront donnés dans les chapitres suivants, ainsi que dans un chapitre détaillé sur l'administration de Jenkins à la fin de ce livre (Chapter 13, Maintenir Jenkins). Ce chapitre n'est qu'une introduction. Cependant, quand vous terminerez la lecture de ce chapitre vous aurez aussi eu un aperçu des rapports de sur les résultats des tests, la production de javadoc et la publication de la couverture de test! Nous avons un long chemin à parcourir, alors allons-y!

2.2. Préparation de votre environnement

Vous pouvez aborder ce chapitre de deux manières différentes. Vous le lisez entièrement, sans toucher un clavier, pour vous faire une idée sur ce qu'est Jenkins. Ou, vous pouvez mettre les mains dans le cambouis, et suivre les étapes sur votre machine.

Si vous voulez suivre tranquillement, il se peut que cela nécessite d'installer certains logiciels sur votre machine. Souvenez-vous que la fonction principale de tout outil d'Intégration Continue est de surveiller du code source dans un outil de gestion de configuration, d'en extraire la dernière version et de la construire dès que des modifications sont apportées. Donc, il vous faudra un outil de gestion de configuration. Dans le cas qui nous occupe, nous utiliserons Git¹. Le code source de notre simple projet se trouve hébergé sur GitHub². Ne vous inquiétez pas, vous ne risquerez pas de perturber ce référentiel avec vos modifications : vous travaillerez sur votre propre réplique et vous pourrez y faire ce que vous voulez. Si vous n'avez jamais utilisé Git et/ou si vous n'avez pas un compte sur GitHub, ne vous inquiétez pas nous vous guiderons pour vos premiers pas et l'installation est très bien documentée sur le site web de GitHub. Nous vous expliquerons comment mettre tout cela en place par la suite.

Dans ce chapitre, nous utiliserons Jenkins pour construire une application Java avec Apache Maven. Apache Maven est un outil de build très largement utilisé dans le monde Java, avec de nombreuses et puissantes fonctionnalités telles que la gestion déclarative des dépendances, le principe de convention plutôt que configuration et un très grand nombre de plugins. Pour construire tout cela, nous utiliserons les dernières versions du Java Development Kit (JDK) et d'Apache Maven, mais si vous ne les avez pas déjà installées sur votre machine ne vous inquiétez pas! Comme nous le verrons, Jenkins saura les installer pour vous.

¹ <http://git-scm.com>

² <https://github.com>

2.2.1. Installation de Java

La première chose que vous allez devoir installer sur votre machine est Java. Hudson est une application web Java web application, aussi vous aurez besoin au minimum du Java Runtime Environment, ou JRE pour l'exécuter. Pour les exemples de ce chapitre, vous devrez avoir une version récente de Java 6 (ces exemples ont été écrits avec Java 6 update 17 et la dernière version disponible au moment où j'écris ces mots est Java 6 update 19). Si vous n'êtes pas sûr, vous pouvez vérifier votre version depuis la ligne de commande (en ouvrant une console DOS sous Windows) et en exécutant `java -version`. Si Java est installé sur votre machine vous devriez obtenir un résultat ressemblant à ceci :

```
$ java -version
java version "1.6.0_17"
Java(TM) SE Runtime Environment (build 1.6.0_17-b04-248-10M3025)
Java HotSpot(TM) 64-Bit Server VM (build 14.3-b01-101, mixed mode)
```

Si vous n'avez une version déjà installée ou si votre version est plus ancienne, téléchargez et installez la dernière version de la JRE depuis le site web de Java³, comme le montre Figure 2.1, "Installer Java".

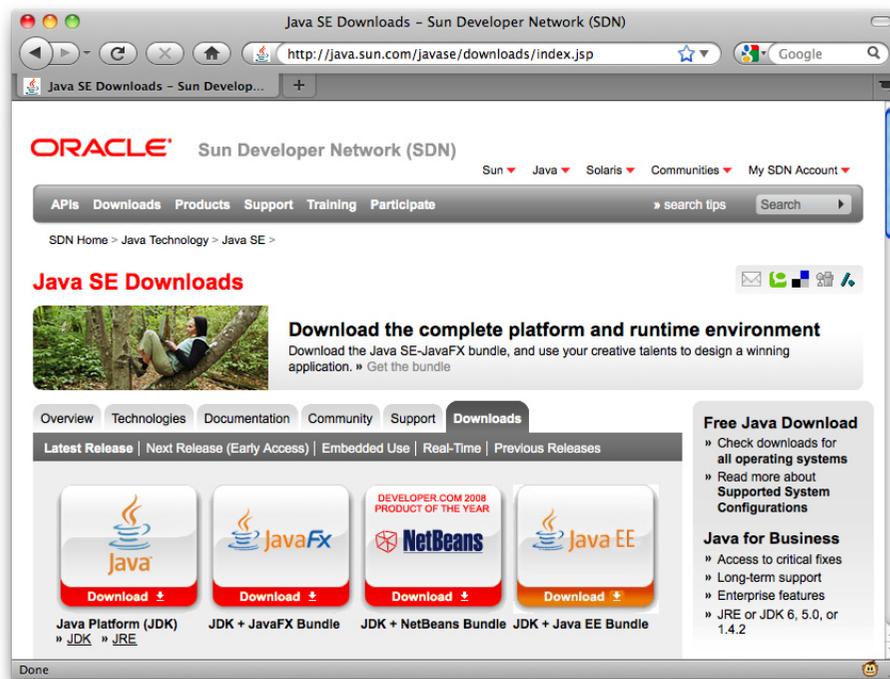


Figure 2.1. Installer Java

³ <http://java.sun.com/javase/downloads/index.jsp>

2.2.2. Installation de Git

Etant donné que nous utiliserons Git, il va vous falloir installer et configurer Git sur votre machine. Si vous êtes novice en ce qui concerne Git, un petit tour sur le site de référence de Git⁴ peut vous être utile. Et si vous vous perdez en cours de route, la manière de procéder est entièrement décrite dans les pages d'aide de GitHub⁵.

Avant toute chose, il vous faut installer Git sur votre machine. Cela exige de télécharger le programme d'installation approprié pour votre système d'exploitation depuis le site web de Git⁶. Il existe des programmes d'installation packagés pour Windows et Mac OS X. Si vous utilisez GNU/Linux vous êtes sur le terrain de Git : les distributions GNU/Linux. Sur Ubuntu ou toute autre distribution basées sur Debian, vous pouvez exécuter une commande comme celle-ci :

```
$ sudo apt-get install git-core
```

Sur Fedora ou toute distribution basée sur RPM, il pouvez utiliser `yum` à la place :

```
$ sudo yum install git-core
```

Et, puisqu'il s'agit de GNU/Linux, vous avez aussi la possibilité d'installer l'application à partir des sources. Les instructions sont disponibles sur le site web de Git.

Une fois cette opération réalisée, vérifiez que Git est correctement installé en exécutant la commande suivante :

```
$ git --version  
git version 1.7.1
```

2.2.3. Configurer un compte GitHub

Ensuite, si vous n'en possédez pas déjà un, vous devrez vous créer un compte Github. Cela est simple et (pour notre usage du moins) gratuit, sans compter que tous les gars cools en ont un. Allez sur la page d'enregistrement de GitHub⁷ et choisissez l'option "Create a free account". Vous n'aurez qu'à fournir un nom d'utilisateur, un mot de passe et votre adresse e-mail (cf. Figure 2.2, "Créer un compte GitHub").

⁴ <http://gitref.org>

⁵ <http://help.github.com>

⁶ <http://git-scm.com>

⁷ <https://github.com/plans>

Figure 2.2. Créer un compte GitHub

2.2.4. Configurer les clefs SSH

GitHub utilise les clefs SSH pour établir une connexion sécurisée entre votre ordinateur et les serveurs Github. Les configurer n'est pas compliqué mais demande un peu d'effort : heureusement il existe des instructions claires et détaillées pour chaque système d'exploitation sur le site de GitHub⁸.

2.2.5. Forker le dépôt des exemples

Comme nous l'avons mentionné plus tôt, tout les exemples de code de ce livre sont stockés sur GitHub, à l'URL suivante : <https://github.com/wakaleo/game-of-life>. C'est un dépôt public, vous pouvez donc librement voir les sources en ligne et récupérer votre propre copie de travail. Toutefois, si vous voulez effectuer des changements, vous devrez créer votre propre fork. Un fork est une copie personnelle d'un dépôt que vous pouvez utiliser à votre guise. Pour créer un fork, connectez à votre compte GitHub et cliquez sur l'URL du dépôt⁹. Cliquez ensuite sur le bouton Fork (see Figure 2.3, “Forker le dépôt des exemples de code”). Cela créera votre propre copie du dépôt.

Une fois que vous avez forké le dépôt, vous devez cloner une copie locale pour vérifier que tout est correctement configuré. Ouvrez la ligne de commande et exécutez la commande suivante (en remplaçant `<username>` avec votre propre nom d'utilisateur GitHub):

⁸ <http://help.github.com/set-up-git-redirect>

⁹ <https://github.com/wakaleo/game-of-life>

```
$ git clone git@github.com:<username>/game-of-life.git
```

Ceci va “cloner” (ou faire un check out, en termes Subversion) une copie du projet sur votre disque local :

```
git clone git@github.com:john-smart/game-of-life.git
Initialized empty Git repository in /Users/johnsmart/.../game-of-life/.git/
remote: Counting objects: 1783, done.
remote: Compressing objects: 100% (589/589), done.
remote: Total 1783 (delta 1116), reused 1783 (delta 1116)
Receiving objects: 100% (1783/1783), 14.83 MiB | 119 KiB/s, done.
Resolving deltas: 100% (1116/1116), done.
```

Vous devriez à présent disposer d'une copie locale du projet que vous pouvez construire et exécuter. Nous utiliserons ce projet plus tard pour déclencher des changements dans le dépôt.

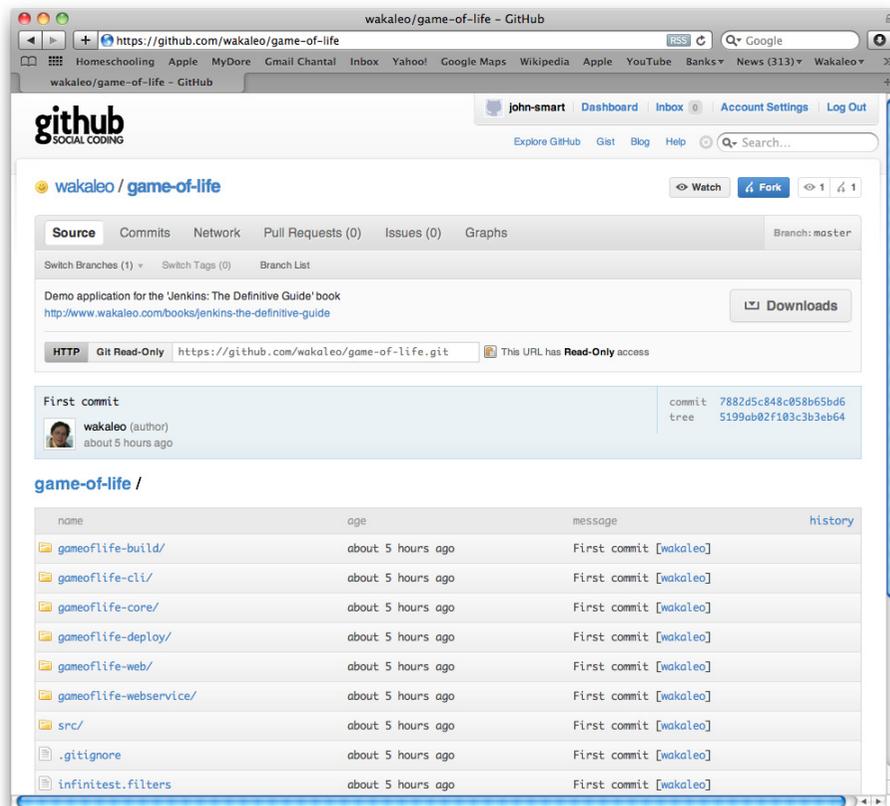


Figure 2.3. Forker le dépôt des exemples de code

2.3. Démarrer Jenkins

Il y a plusieurs façons d'exécuter Jenkins sur votre machine. Une des plus faciles pour exécuter Jenkins pour la première fois est d'utiliser Java Web Start. Java Web Start est une technologie qui vous permet de démarrer une application Java sur votre machine locale via une URL sur une page web — cela vient préinstallé avec le JRE Java. Dans notre cas, cela démarrera un serveur Jenkins s'exécutant sur votre machine, et vous permettra de l'expérimenter comme si vous l'aviez installé localement. Tout ce dont vous avez besoin pour cela est de travailler avec une version récente (Java 6 ou plus) du Java Runtime Environment (JRE), que nous avons installé dans la section précédente.

Par commodité, il y a un lien vers l'instance Java Web Start de Jenkins sur la page de ressources du livre¹⁰. Vous y trouverez un large bouton orange dans la section Ressources du Livre (voir Figure 2.4, "Exécuter Jenkins en utilisant Java Web Start à partir du site web du livre"). Vous pouvez aussi trouver ce lien sur la page Meet Jenkins sur le site Web de Jenkins¹¹, où, si vous scrollez assez loin, vous devriez trouver une section "Test Drive" avec un bouton "Launch" identique.

¹⁰ <http://www.wakaleo.com/books/jenkins-the-definitive-guide>

¹¹ <http://wiki.jenkins-ci.org/display/JENKINS/Meet+Jenkins>

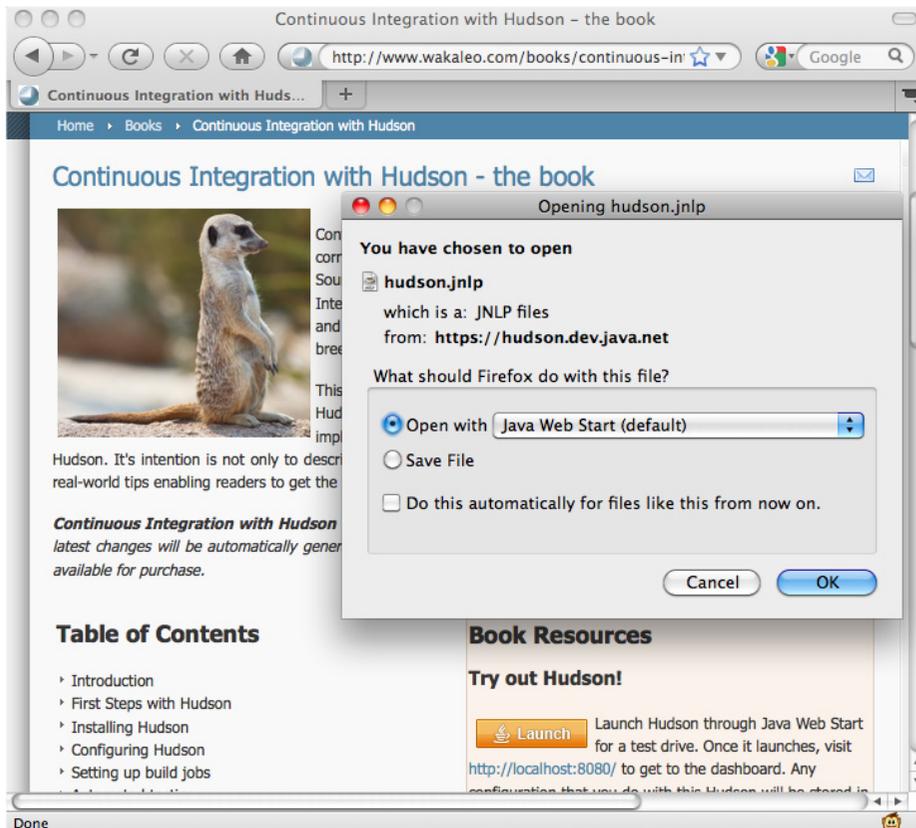


Figure 2.4. Exécuter Jenkins en utilisant Java Web Start à partir du site web du livre

Java Web Start semble fonctionner le mieux sur Firefox. Quand vous cliquez sur le bouton Launch sur l'un de ces sites dans Firefox, le navigateur vous demandera si vous voulez ouvrir un fichier appelé `jenkins.jnlp` en utilisant Java Web Start. Cliquez sur OK — cela téléchargera Jenkins et le démarrera sur votre machine (voir Figure 2.5, “Java Web Start téléchargera et exécutera la dernière version de Jenkins”).

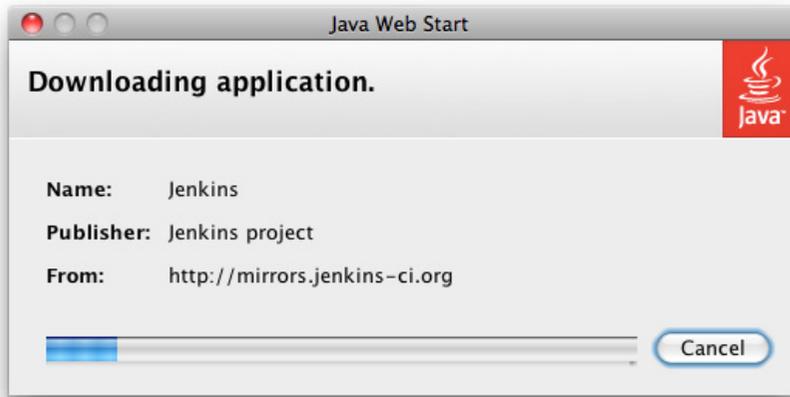


Figure 2.5. Java Web Start téléchargera et exécutera la dernière version de Jenkins

Dans d'autres navigateurs, cliquer sur le bouton pourrait simplement télécharger le fichier JNLP. Dans Internet Explorer, vous pourriez même avoir à cliquer avec le bouton droit sur le lien et sélectionner "Enregistrer sous" pour enregistrer le fichier JNLP, et l'exécuter depuis Windows Explorer. Toutefois, dans chacun de ces cas, quand vous ouvrez le fichier JNLP, Java Web Start téléchargera et démarrera Jenkins.

Java Web Start n'aura besoin de télécharger une version particulière de Jenkins qu'une seule fois. A partir de ce moment, quand vous cliquez sur le bouton "Launch" à nouveau, Java Web Start utilisera une copie de Jenkins qu'il a déjà téléchargée (jusqu'à ce qu'une nouvelle version sorte). Ignorez tout message que votre système d'exploitation ou votre anti-virus pourrait afficher — il est parfaitement sûr d'exécuter Jenkins sur votre machine locale.

Une fois qu'il aura fini de télécharger, il démarrera Jenkins sur votre machine. Vous pourrez le voir s'exécuter dans une petite fenêtre appelée "Console Jenkins" (voir Figure 2.6, "Java Web Start exécutant Jenkins"). Pour arrêter Jenkins à tout moment, fermez simplement cette fenêtre.

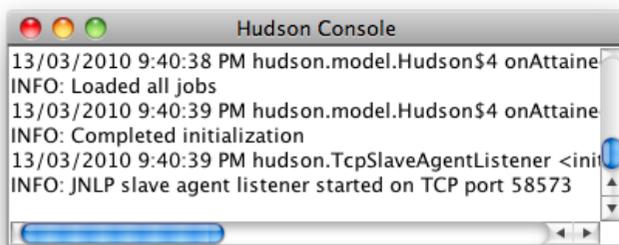


Figure 2.6. Java Web Start exécutant Jenkins

Il y a aussi des installeurs disponibles pour les principaux systèmes d'exploitations sur le site Web de Jenkins¹². Ou, si vous êtes un utilisateur Java expérimenté versé dans les manipulations de fichiers WAR, vous pourriez préférer simplement télécharger la dernière version de Jenkins et l'exécuter depuis la ligne de commande. Jenkins est fourni sous la forme d'un WAR exécutable — vous pouvez télécharger la version la plus récente à partir de la page d'accueil¹³ du site de Jenkins. Par facilité, il y a un lien vers la dernière version de Jenkins dans la section Ressources du site web¹⁴ de ce livre.

Une fois téléchargé, vous pouvez démarrer Jenkins depuis la ligne de commande comme montré ici :

```
$ java -jar jenkins.war
```

Si vous avez démarré Jenkins en utilisant Java Web Start ou via la ligne de commande, Jenkins devrait à présent tourner sur votre machine locale. Par défaut, Jenkins s'exécutera sur le port 8080, vous pouvez donc accéder à Jenkins dans votre navigateur web sur <http://localhost:8080>.

Sinon, si vous êtes familier avec les serveurs d'application comme Tomcat, vous pouvez simplement déployer le WAR de Jenkins dans votre serveur d'application — avec Tomcat, par exemple, vous pouvez simplement placer le fichier `jenkins.war` dans le répertoire `webapps` de Tomcat. Si vous exécutez Jenkins sur un serveur d'application, l'URL à utiliser pour accéder à Jenkins sera légèrement différente. Sur une installation Tomcat par défaut, par exemple, vous pouvez accéder à Jenkins dans votre navigateur web à l'adresse <http://localhost:8080/jenkins>.

Quand vous ouvrez Jenkins dans votre navigateur, vous devez voir un écran comme celui montré dans Figure 2.7, “La page de démarrage Jenkins”. Vous êtes maintenant prêt à faire vos premiers pas avec Jenkins !

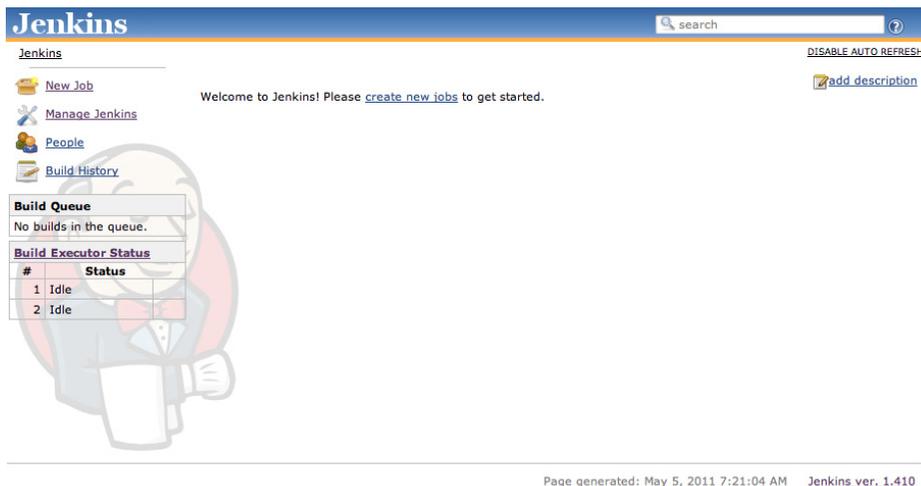


Figure 2.7. La page de démarrage Jenkins

¹² <http://jenkins-ci.org>

¹³ <http://http://jenkins-ci.org>

¹⁴ <http://www.wakaleo.com/books/jenkins-the-definitive-guide>

2.4. Configuring the Tools

Before we get started, we do need to do a little configuration. More precisely, we need to tell Jenkins about the build tools and JDK versions we will be using for our builds.

Click on the Manage Jenkins link on the home page (see Figure 2.7, “La page de démarrage Jenkins”). This will take you to the Manage Jenkins page, the central one-stop-shop for all your Jenkins configuration. From this screen, you can configure your Jenkins server, install and upgrade plugins, keep track of system load, manage distributed build servers, and more! For now, however, we’ll keep it simple. Just click on the Configuring System link at the top of the list (see Figure 2.8, “The Manage Jenkins screen”).

Jenkins [DISABLE AUTO REFRESH](#)

[New Job](#)
[Manage Jenkins](#)
[People](#)
[Build History](#)

Build Queue
No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle

Manage Jenkins

- [Configure System](#)
Configure global settings and paths.
- [Reload Configuration from Disk](#)
Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files directly on disk.
- [Manage Plugins](#)
Add, remove, disable or enable plugins that can extend the functionality of Jenkins.
- [System Information](#)
Displays various environmental information to assist trouble-shooting.
- [System Log](#)
System log captures output from java.util.logging output related to Jenkins.
- [Load Statistics](#)
Check your resource utilization and see if you need more computers for your builds.
- [Jenkins CLI](#)
Access/manage Jenkins from your shell, or from your script.
- [Script Console](#)
Executes arbitrary script for administration/trouble-shooting/diagnostics.
- [Manage Nodes](#)
Add, remove, control and monitor the various nodes that Jenkins runs jobs on.
- [About Jenkins](#)
See the version and license information
- [Prepare for Shutdown](#)
Stops executing new builds, so that the system can be eventually shut down safely.

Page generated: May 5, 2011 7:24:07 AM [Jenkins ver. 1.410](#)

Figure 2.8. The Manage Jenkins screen

This will take you to Jenkins’s main configuration screen (see Figure 2.9, “The Configure Jenkins screen”). From here you can configure everything from security configuration and build tools to email servers, version control systems and integration with third-party software. The screen contains a lot of information, but most of the fields contain sensible default values, so you can safely ignore them for now.

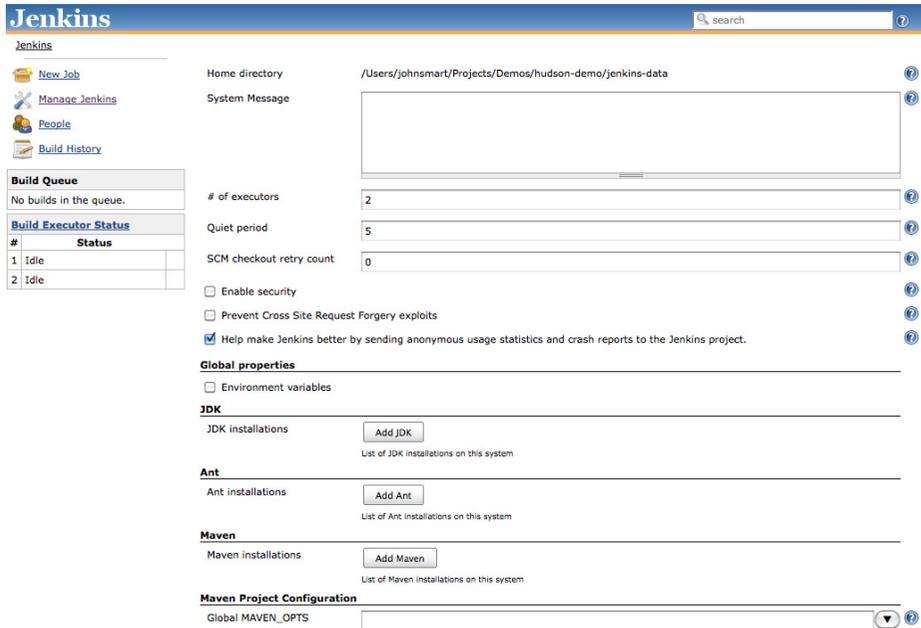


Figure 2.9. The Configure Jenkins screen

For now, you will just need to configure the tools required to build our sample project. The application we will be building is a Java application, built using Maven. So in this case, all we need to do is to set up a recent JDK and Maven installation.

However before we start, take a look at the little blue question mark icons lined to the right of the screen. These are Jenkins’s contextual help buttons. If you are curious about a particular field, click on the help icon next to it and Jenkins will display a very detailed description about what it is and how it works.

2.4.1. Configuring Your Maven Setup

Our sample project uses Maven, so we will need to install and configure Maven first. Jenkins provides great out-of-the-box support for Maven. Scroll down until you reach the Maven section in the Configure System screen (see Figure 2.10, “Configuring a Maven installation”).

Jenkins provides several options when it comes to configuring Maven. If you already have Maven installed on your machine, you can simply provide the path in the MAVEN_HOME field. Alternatively, you can install a Maven distribution by extracting a zip file located in a shared directory, or execute a home-rolled installation script. Or you can let Jenkins do all the hard work and download Maven for you. To choose this option, just tick the Install automatically checkbox. Jenkins will download and install Maven from the Apache website the first time a build job needs it. Just choose the Maven version you want to install and Jenkins will do the rest. You will also need to give a name for your Maven version (imaginatively called “Maven 2.2.1” in the example), so that you can refer to it in your build jobs.

For this to work, you need to have an Internet connection. If you are behind a proxy, you'll need to provide your proxy information—we discuss how to set this up in Section 4.9, “Configurer un Proxy”.

The screenshot shows the 'Maven' configuration page in Jenkins. At the top, there is a section titled 'Maven installations'. Below this, there is a form for configuring a new installation. The 'name' field contains 'Maven 2.2.1'. There is a checked checkbox for 'Install automatically' with a help icon to its right. Below that, there is a section for 'Install from Apache' with a version dropdown menu set to '2.2.1'. At the bottom of the form, there are four buttons: 'Add Installer' (with a dropdown arrow), 'Delete Installer', 'Add Maven', and 'Delete Maven'. Below the buttons, there is a small text label: 'List of Maven installations on this system'.

Figure 2.10. Configuring a Maven installation

One of the nice things about the Jenkins Maven installation process is how well it works with remote build agents. Later on in the book, we'll see how Jenkins can also run builds on remote build servers. You can define a standard way of installing Maven for all of your build servers (downloading from the Internet, unzipping a distribution bundle on a shared server, etc.)—all of these options will work when you add a new remote build agent or set up a new build server using this Jenkins configuration.

2.4.2. Configuring the JDK

Once you have configured your Maven installation, you will also need to configure a JDK installation (see Figure 2.11, “Configuring a JDK installation”). Again, if you have a Java JDK (as opposed to a Java Runtime Environment—the JDK contains extra development tools such as the Java compiler) already installed on your workstation, you can simply provide the path to your JDK in the `JAVA_HOME` field. Otherwise, you can ask Jenkins to download the JDK from the Oracle website¹⁵ the first time a build job requires it. This is similar to the automatic Maven installation feature—just pick the JDK version you need and Jenkins will take care of all the logistics. However, for licensing reasons, you will also need to tick a checkbox to indicate that you agree with the Java SDK License Agreement.

¹⁵ <http://www.oracle.com/technetwork/java/index.html>

JDK

JDK installations

name

Install automatically ?

 **Install from java.sun.com**

Version

I agree to the Java SE Development Kit License Agreement

Figure 2.11. Configuring a JDK installation

Now go to the bottom of the screen and click on the Save button.

2.4.3. Notification

Another important aspect you would typically set up is notification. When a Jenkins build breaks, and when it works again, it can send out email messages to the team to spread the word. Using plugins, you can also get it to send instant messages or SMS messages, post entries on Twitter, or get people notified in a few other ways. It all depends on what works best for your organizational culture. Email notification is easy enough to set up if you know your local SMTP server address—just provide this value in the Email Notification section towards the bottom of the main configuration page. However, to keep things simple, we’re not going to worry about notifications just yet.

2.4.4. Setting Up Git

The last thing we need to configure for this demo is to get Jenkins working with Git. Jenkins comes with support for Subversion and CVS out of the box, but you will need to install the Jenkins Git plugin to be able to complete the rest of this tutorial. Don’t worry, the process is pretty simple. First of all, click on the Manage Jenkins link to the left of the screen to go back to the main configuration screen (see Figure 2.8, “The Manage Jenkins screen”). Then click on Manage Plugins. This will open the plugin configuration screen, which is where you manage the extra features you want to install on your Jenkins server. You should see four tabs: Updates, Available, Installed, and Advanced (see Figure 2.12, “Managing plugins in Jenkins”).

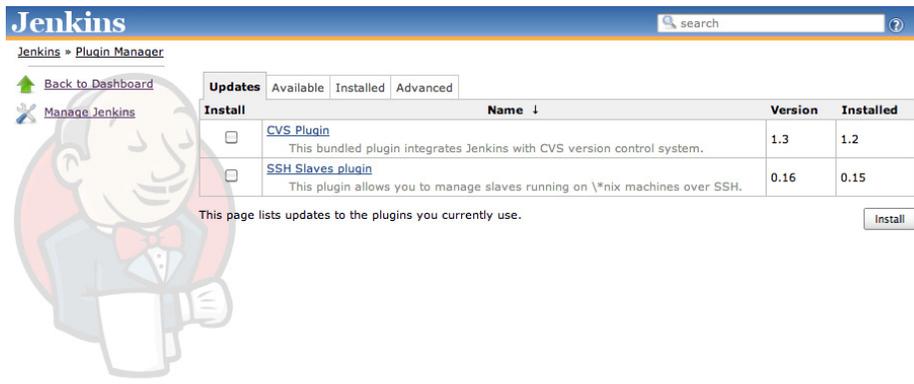


Figure 2.12. Managing plugins in Jenkins

For now, just click on the Available tab. Here you will see a very long list of available plugins. Find the Git Plugin entry in this list and tick the corresponding checkbox (see Figure 2.13, “Installing the Git plugin”), and then scroll down to the bottom of the screen and click on Install. This will download and install the Jenkins Git plugin into your local Jenkins instance.

<input type="checkbox"/>	CMVC Plugin This plugin integrates CMVC to Hudson.	0.3
<input type="checkbox"/>	Darcs Plugin This plugin integrates Darcs version control system to Jenkins. The plugin requires the Darcs binary (darcs) to be installed on the target machine.	0.3.5
<input type="checkbox"/>	Dimensions Plugin This plugin integrates Hudson with Dimensions , the Serena SCM solution.	0.8.1
<input type="checkbox"/>	File System SCM Use File System as SCM.	1.10
<input checked="" type="checkbox"/>	Git Plugin This plugin allows use of GIT as a build SCM. Git 1.3.3 or newer is required.	1.1.6
<input type="checkbox"/>	Harvest Plugin This plugin allows you to use CA Harvest as a SCM.	0.4

Figure 2.13. Installing the Git plugin

Once it is done, you will need to restart Jenkins for the changes to take effect. To do this, you can simply click on the “Restart Jenkins when no jobs are running” button displayed on the installation screen, or alternatively shut down and restart Jenkins by hand.

That is all we need to configure at this stage. You are now ready to set up your first Jenkins build job!

2.5. Your First Jenkins Build Job

Build jobs are at the heart of the Jenkins build process. Simply put, you can think of a Jenkins build job as a particular task or step in your build process. This may involve simply compiling your source code

and running your unit tests. Or you might want a build job to do other related tasks, such as running your integration tests, measuring code coverage or code quality metrics, generating technical documentation, or even deploying your application to a web server. A real project usually requires many separate but related build jobs.

Our sample application is a simple Java implementation of John Conway’s “Game of Life.”¹⁶ The Game of Life is a mathematical game which takes place on a two dimensional grid of cells, which we will refer to as the Universe. Each cell can be either alive or dead. Cells interact with their direct neighbors to determine whether they will live or die in the next generation of cells. For each new generation of cells, the following rules are applied:

- Any live cell with fewer than two live neighbors dies of underpopulation.
- Any live cell with more than three live neighbors dies of overcrowding.
- Any live cell with two or three live neighbors lives on to the next generation.
- Any dead cell with exactly three live neighbors becomes a live cell.

Our application is a Java module, built using Maven, that implements the core business logic of the Game of Life. We’ll worry about the user interfaces later on. For now, let’s see how we can automate this build in Jenkins. If you are not familiar with Maven, or prefer Ant or another build framework—don’t worry! The examples don’t require much knowledge of Maven, and we’ll be looking at plenty of examples of using other build tools later on in the book.

For our first build job, we will keep it simple: we are just going to compile and test our sample application. Click on the New Job link. You should get to a screen similar to Figure 2.14, “Setting up your first build job in Jenkins”. Jenkins supports several different types of build jobs. The two most commonly-used are the freestyle builds and the Maven 2/3 builds. The freestyle projects allow you to configure just about any sort of build job: they are highly flexible and very configurable. The Maven 2/3 builds understand the Maven project structure, and can use this to let you set up Maven build jobs with less effort and a few extra features. There are also plugins that provide support for other types of build jobs. Nevertheless, although our project does use Maven, we are going to use a freestyle build job, just to keep things simple and general to start with. So choose “Build a freestyle software project”, as shown in Figure 2.14, “Setting up your first build job in Jenkins”.

You’ll also need to give your build job a sensible name. In this case, call it `gameoflife-default`, as it will be the default CI build for our Game of Life project.

¹⁶See http://en.wikipedia.org/wiki/Conway%27s_Game_of_Life.

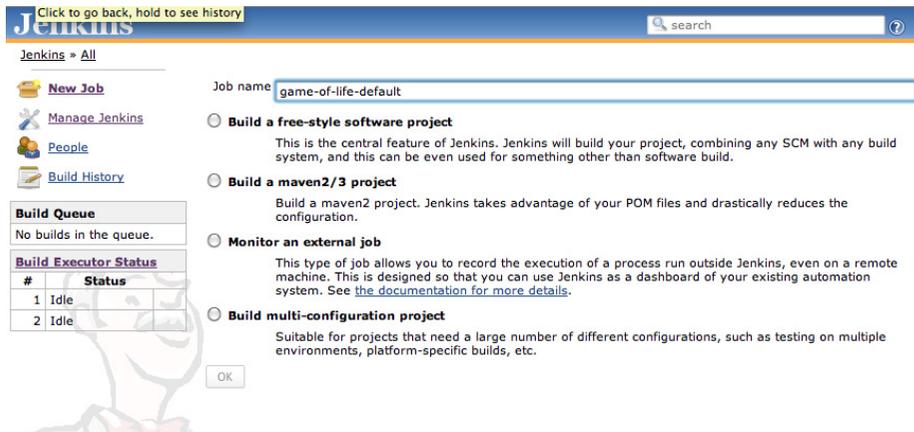


Figure 2.14. Setting up your first build job in Jenkins

Once you click on OK, Jenkins will display the project configuration screen (see Figure 2.15, “Telling Jenkins where to find the source code”).

In a nutshell, Jenkins works by checking out the source code of your project and building it in its own workspace. So the next thing you need to do is to tell Jenkins where it can find the source code for your project. You do this in the Source Code Management section (see Figure 2.15, “Telling Jenkins where to find the source code”). Jenkins provides support for CVS and Subversion out of the box, and many others such as Git, Mercurial, ClearCase, Perforce and many more via plugins.

For this project, we will be getting the source code from the GitHub repository we set up earlier. On the Jenkins screen, choose “Git” and enter the Repository URL we defined in Section 2.2.5, “Forker le dépôt des exemples” (see Figure 2.15, “Telling Jenkins where to find the source code”). Make sure this is the URL of your fork, and not of the original repository: it should have the form `git@github.com:<username>/game-of-life.git`, where `<username>` is the username for your own GitHub account. You can leave all of the other options up until here with their default values.

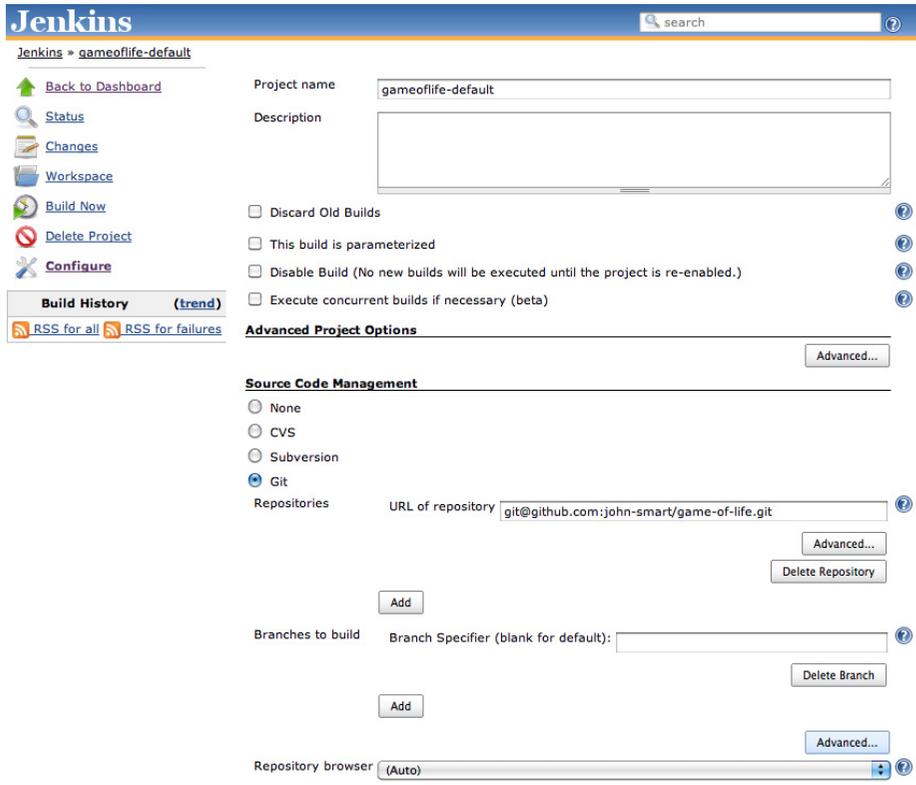


Figure 2.15. Telling Jenkins where to find the source code

Once we have told Jenkins where to find the source code for our application, we need to tell it how often it should check for updates. We want Jenkins to monitor the repository and start a build whenever any changes have been committed. This is a common way to set up a build job in a Continuous Integration context, as it provides fast feedback if the build fails. Other approaches include building on regular intervals (for example, once a day), requiring a user to kick of the build manually, or even triggering a build remotely using a “post-commit” hook in your SCM.

We configure all of this in the Build Triggers section (see Figure 2.16, “Scheduling the build jobs”). Pick the Poll SCM option and enter “* * * * *” (that’s five asterisks separated by spaces) in the Schedule box. Jenkins schedules are configured using the `CRON` syntax, well-known in the Unix world. The `CRON` syntax consists of five fields separated by white space, indicating respectively the minute (0–59), hour (0–23), day of the month (1–31), month (1–12) and the day of the week (0–7, with 0 and 7 being Sunday). The star is a wildcard character which accepts any valid value for that field. So five stars basically means “every minute of every hour of every day.” You can also provide ranges of values: “* 9-17 * * *” would mean “every minute of every day, between 9am and 5pm.” You can also space out the schedule using intervals: “*/5 * * * *” means “every 5 minutes,” for example. Finally, there are some other convenient short-hands, such as “@daily” and “@hourly”.

Don't worry if your Unix skills are a little rusty—if you click on the blue question mark icon on the side of the schedule box, Jenkins will bring up a very complete refresher.

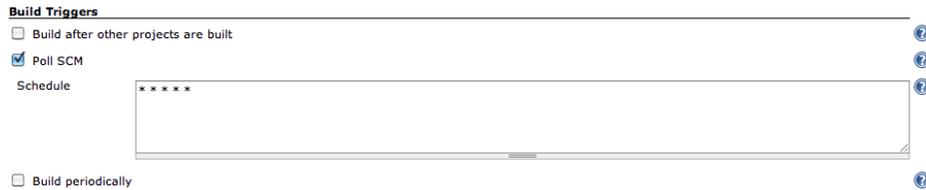


Figure 2.16. Scheduling the build jobs

The next step is to configure the actual build itself. In a freestyle build job, you can break down your build job into a number of build steps. This makes it easier to organize builds in clean, separate stages. For example, a build might run a suite of functional tests in one step, and then tag the build in a second step if all of the functional tests succeed. In technical terms, a build step might involve invoking an Ant task or a Maven target, or running a shell script. There are also Jenkins plugins that let you use additional types of build steps: Gant, Grails, Gradle, Rake, Ruby, MSBuild and many other build tools are all supported.

For now, we just want to run a simple Maven build. Scroll down to the Build section and click on the “Add build step” and choose “Invoke top-level Maven targets” (see Figure 2.17, “Adding a build step”). Then enter “clean package” in the Goals field. If you are not familiar with Maven, this will delete any previous build artifacts, compile our code, run our unit tests, and generate a JAR file.



Figure 2.17. Adding a build step

By default, this build job will fail if the code does not compile or if any of the unit tests fail. That's the most fundamental thing that you'd expect of any build server. But Jenkins also does a great job of helping you display your test results and test result trends.

The de facto standard for test reporting in the Java world is an XML format used by JUnit. This format is also used by many other Java testing tools, such as TestNG, Spock and Easyb. Jenkins understands this format, so if your build produces JUnit XML test results, Jenkins can generate nice graphical test reports and statistics on test results over time, and also let you view the details of any test failures. Jenkins also

keeps track of how long your tests take to run, both globally, and per test—this can come in handy if you need to track down performance issues.

So the next thing we need to do is to get Jenkins to keep tabs on our unit tests.

Go to the Post-build Actions section (see Figure 2.18, “Configuring JUnit test reports and artifact archiving”) and tick “Publish JUnit test result report” checkbox. When Maven runs unit tests in a project, it automatically generates the XML test reports in a directory called `surefire-reports` in the `target` directory. So enter “`**/target/surefire-reports/*.xml`” in the “Test report XMLs” field. The two asterisks at the start of the path (“`**`”) are a best practice to make the configuration a bit more robust: they allow Jenkins to find the target directory no matter how we have configured Jenkins to check out the source code.

Another thing you often want to do is to archive your build results. Jenkins can store a copy of the binary artifacts generated by your build, allowing you to download the binaries produced by a build directly from the build results page. It will also post the latest binary artifacts on the project home page, which is a convenient way to distribute the latest and greatest version of your application. You can activate this option by ticking the “Archive the artifacts” checkbox and indicating which binary artifacts you want Jenkins to archive. In Figure 2.18, “Configuring JUnit test reports and artifact archiving”, for example, we have configured Jenkins to store all of the JAR files generated by this build job.

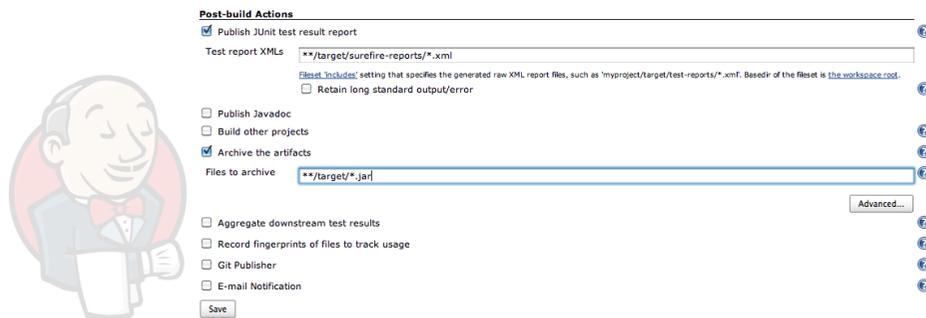


Figure 2.18. Configuring JUnit test reports and artifact archiving

Now we’re done—just click on the Save button at the bottom of the screen. Our build job should now be ready to run. So let’s see it in action!

2.6. Your First Build Job in Action

Once you save your new build job, Jenkins will display the home page for this job (see Figure 2.19, “Your first build job running”). This is where Jenkins displays details about the latest build results and the build history.

If you wait a minute or so, the build should kick off automatically—you can see the stripy progress bar in the Build History section in the bottom left hand corner of Figure 2.19, “Your first build job running”. Or, if you are impatient, you can also trigger the build manually using the Build Now button.

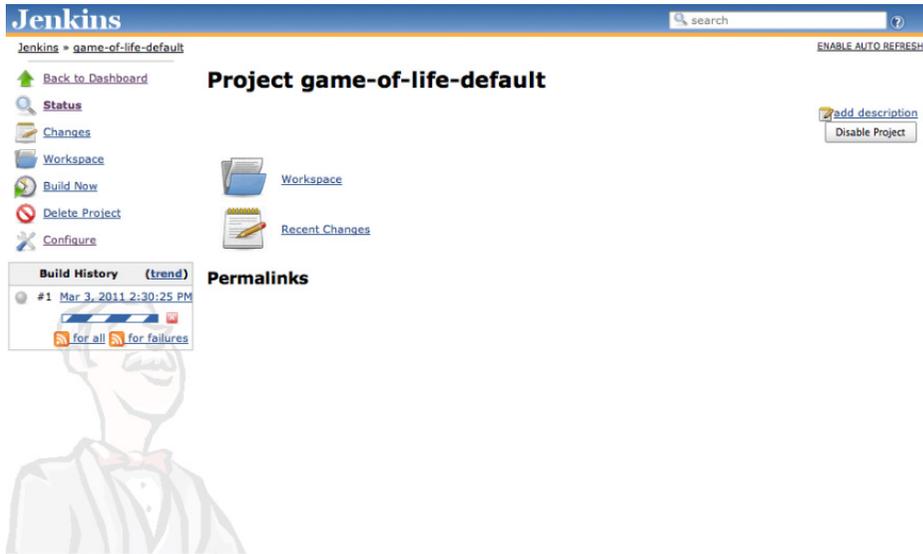


Figure 2.19. Your first build job running

The build will also now figure proudly on your Jenkins server’s home page (see Figure 2.20, “The Jenkins dashboard”). This page shows a summary of all of your build jobs, including the current build status and general state of health of each of your builds. It tells you when each build ran successfully for the last time, and when it last failed, and also the result of the last build.

One of Jenkins’s specialities is the way it lets you get an idea of build behavior over time. For example, Jenkins uses a weather metaphor to help give you an idea of the stability of your builds. Essentially, the more your builds fail, the worse the weather gets. This helps you get an idea of whether a particular broken build is an isolated event, or if the build is breaking on a regular basis, in which case it might need some special attention.

You can also manually trigger a build job here, using the build schedule button (that’s the one that looks a bit like a green play button on top of a clock).



Figure 2.20. The Jenkins dashboard

When the build finishes, the ball in the Build History box becomes solid blue. This means the build was a success. Build failures are generally indicated by a red ball. For some types of project, you can also distinguish between a build error (such as a compiler error), indicated by a red ball, and other sorts of build failures, such as unit test failures or insufficient code coverage, which are indicated by a yellow ball. There are also some other details about the latest test results, when the last build was run, and so on. But before we look at the details, let's get back to the core business model of a Continuous Integration server—kicking off builds when someone changes the code!

We are going to commit a code change to GitHub and see what happens, using the source code we checked out in Section 2.2.5, “Forker le dépôt des exemples”. We now have Jenkins configured to monitor our GitHub fork, so if we make any changes, Jenkins should be able to pick them up.

So let's make a change. The idea is to introduce a code change that will cause the unit tests to fail. If your Java is a bit rusty, don't worry, you won't need to know any Java to be able to break the build—just follow the instructions!

Now in normal development, you would first modify the unit test that describes this behaviour. Then you would verify that the test fails with the existing code, and implement the code to ensure that the test passes. Then you would commit your changes to your version control system, allowing Jenkins to build them. However this would be a poor demonstration of how Jenkins handles unit test failures. So in this example, we will, against all best practices, simply modify the application code directly.

First of all, open the `Cell.java` file, which you will find in the `gameoflife-core/src/main/java/com/wakaleo/gameoflife/domain` directory. Open this file in your favorite text editor. You should see something like this:

```
package com.wakaleo.gameoflife.domain;

public enum Cell {
    LIVE_CELL("*"), DEAD_CELL(".");

    private String symbol;

    private Cell(String symbol) {
        this.symbol = symbol;
    }

    @Override
    public String toString() {
        return symbol;
    }

    static Cell fromSymbol(String symbol) {
        Cell cellRepresentedBySymbol = null;
        for (Cell cell : Cell.values()) {
            if (cell.symbol.equals(symbol)) {
                cellRepresentedBySymbol = cell;
                break;
            }
        }
    }
}
```

```

    }
    return cellRepresentedBySymbol;
}

public String getSymbol() {
    return symbol;
}
}

```

The application can print the state of the grid as a text array. Currently, the application prints our live cells as an asterisk (*), and dead cells appear as a minus character (-). So a five-by-five grid containing a single living cell in the center would look like this:

```

-----
--*--
-----

```

Now users have asked for a change to the application—they want pluses (+) instead of stars! So we are going to make a slight change to the `Cell` class method, and rewrite it as follows (the modifications are in **bold**):

```

package com.wakaleo.gameoflife.domain;

public enum Cell {
    LIVE_CELL("+"), DEAD_CELL(".");

    private String symbol;

    private Cell(String symbol) {
        this.symbol = symbol;
    }

    @Override
    public String toString() {
        return symbol;
    }

    static Cell fromSymbol(String symbol) {
        Cell cellRepresentedBySymbol = null;
        for (Cell cell : Cell.values()) {
            if (cell.symbol.equals(symbol)) {
                cellRepresentedBySymbol = cell;
                break;
            }
        }
        return cellRepresentedBySymbol;
    }

    public String getSymbol() {
        return symbol;
    }
}

```

Save this change, and then commit them to the local Git repository by running `git commit`:

```
$ git commit -a -m "Changes stars to pluses"
[master 61ce946] Changes stars to pluses
 1 files changed, 1 insertions(+), 1 deletions(-)
```

This will commit the changes locally, but since Git is a distributed repository, you now have to push these changes through to your fork on GitHub. You do this by running `git push`:

```
$ git push
Counting objects: 21, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (11/11), 754 bytes, done.
Total 11 (delta 4), reused 0 (delta 0)
To git@github.com:john-smart/game-of-life.git
 7882d5c..61ce946  master -> master
```

Now go back to the Jenkins web page. After a minute or so, a new build should kick off, and fail. In fact, there are several other places which are affected by this change, and the regression tests related to these features are now failing. On the build job home page, you will see a second build in the build history with an ominous red ball (see Figure 2.21, “A failed build”)—this tells you that the latest build has failed.

You might also notice some clouds next to the Build History title—this is the same “weather” icon that we saw on the home page, and serves the same purpose—to give you a general idea of how stable your build is over time.

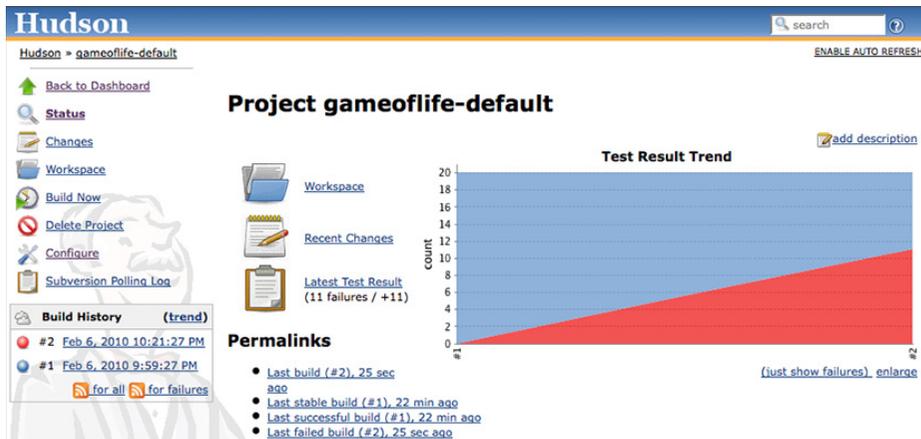


Figure 2.21. A failed build

If you click on the new build history entry, Jenkins will give you some more details about what went wrong (see Figure 2.22, “The list of all the broken tests”). Jenkins tells us that there were 11 new test

failures in this build, something which can be seen at a glance in the Test Result Trend graph—red indicates test failures. You can even see which tests are failing, and how long they have been broken.

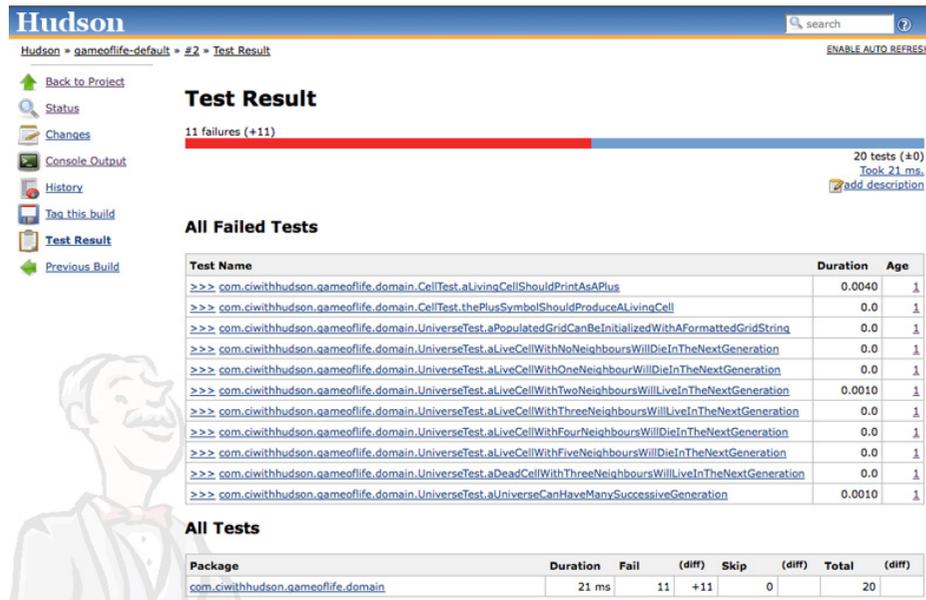


Figure 2.22. The list of all the broken tests

If you want to know exactly what went wrong, that's easy enough to figure out as well. If you click on the failed test classes, Jenkins brings up the actual details of the test failures (see Figure 2.23, “Details about a failed test”), which is a great help when it comes to reproducing and fixing the issue.



Figure 2.23. Details about a failed test

Jenkins displays a host of information about the failed test in a very readable form, including the error message the test produced, the stack trace, how long the test has been broken, and how long it took to run. Often, this in itself is enough to put a developer on the right track towards fixing the issue.

Now let's fix the build. To make things simple, we'll just back out our changes and recommit the code in its original state (the end users just changed their mind about the asterisks, anyway). So just undo the changes you made to the `Cell` class (again, the changes are highlighted in **bold**):

```
package com.wakaleo.gameoflife.domain;

public enum Cell {
    LIVE_CELL("*"), DEAD_CELL(".");

    private String symbol;

    private Cell(String symbol) {
        this.symbol = symbol;
    }

    @Override
    public String toString() {
        return symbol;
    }

    static Cell fromSymbol(String symbol) {
        Cell cellRepresentedBySymbol = null;
        for (Cell cell : Cell.values()) {
            if (cell.symbol.equals(symbol)) {
                cellRepresentedBySymbol = cell;
                break;
            }
        }
        return cellRepresentedBySymbol;
    }

    public String getSymbol() {
        return symbol;
    }
}
```

When you've done this, commit your changes again:

```
$ git commit -a -m "Restored the star"
[master bc924be] Restored the star
 1 files changed, 1 insertions(+), 1 deletions(-)
$ git push
Counting objects: 21, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (7/7), done.
Writing objects: 100% (11/11), 752 bytes, done.
Total 11 (delta 4), reused 6 (delta 0)
To git@github.com:john-smart/game-of-life.git
 61ce946..bc924be master -> master
```

Once you've committed these changes, Jenkins should pick them up and kick off a build. Once this is done, you will be able to see the fruit of your work on the build job home page (see Figure 2.24, “Now the build is back to normal”)—the build status is blue again and all is well. Also notice the way we are building up a trend graph showing the number of succeeding unit tests over time—this sort of report really is one of Jenkins's strong points.

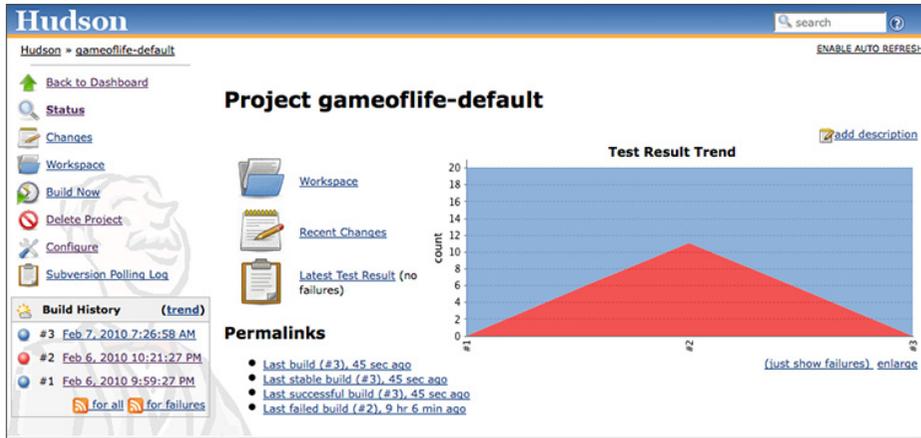


Figure 2.24. Now the build is back to normal

2.7. More Reporting—Displaying Javadocs

For many Java projects, Javadoc comments are an important source of low-level technical documentation. There are even tools, such as UmlGraph, that let you produce Javadoc with embedded UML diagrams to give you a better picture of how the classes fit together in the application. This sort of technical documentation has the advantage of being cheap to produce, accurate and always up-to-date.

Jenkins can integrate Javadoc API documentation directly into the Jenkins website. This way, everyone can find the latest Javadoc easily, in a well known place. Often, this sort of task is performed in a separate build job, but for simplicity we are going to add another build step to the gameoflife-default build job to generate and display Javadoc documentation for the Game of Life API.

Start off by going into the “gameoflife-default” configuration screen again. Click on “Add build step”, and add a new build step to “Invoke top level Maven targets” (see Figure 2.25, “Adding a new build step and report to generate Javadoc”). In the Goals field, place `javadoc:javadoc`—this will tell Maven to generate the Javadoc documentation.



Figure 2.25. Adding a new build step and report to generate Javadoc

Now go to the “Post-build Action” and tick the “Publish Javadoc” checkbox. This project is a multimodule project, so a separate subdirectory is generated for each module (core, services, web and so forth). For this example, we are interested in displaying the documentation for the core module. In the Javadoc directory field, enter `gameoflife-core/target/site/apidocs`—this is where Maven will place the Javadocs it generates for the core module. Jenkins may display an error message saying that this directory doesn’t exist at first. Jenkins is correct—this directory won’t exist until we run the `javadoc:javadoc` goal, but since we haven’t run this command yet we can safely ignore the message at this stage.

If you tick “Retain Javadoc for each successful build”, Jenkins will also keep track of the Javadocs for previous builds—not always useful, but it can come in handy at times.

Now trigger a build manually. You can do this either from the build job’s home page (using the Build Now link), or directly from the server home page. Once the build is finished, open the build job summary page. You should now see a Javadoc link featuring prominently on the screen—this link will open the latest version of the Javadoc documentation (see Figure 2.26, “Jenkins will add a Javadoc link to your build results”). You will also see this link on the build details page, where it will point to the Javadoc for that particular build, if you have asked Jenkins to store Javadoc for each build.

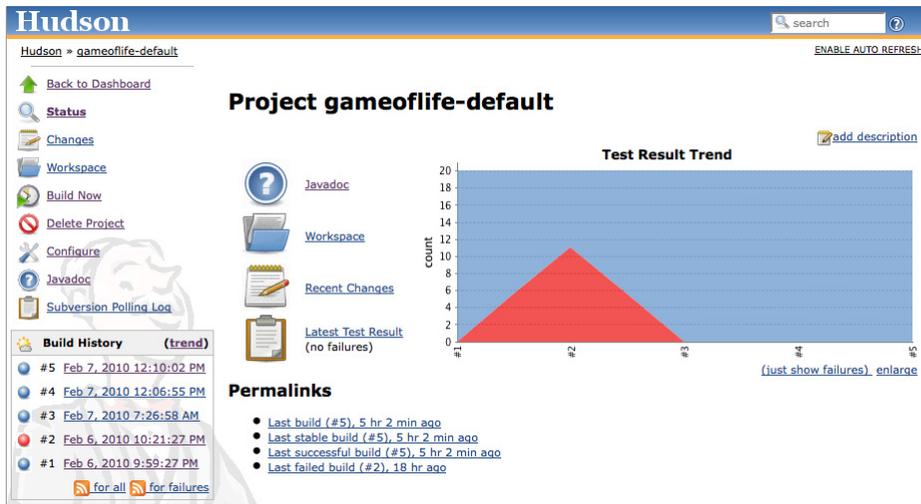


Figure 2.26. Jenkins will add a Javadoc link to your build results

2.8. Adding Code Coverage and Other Metrics

As we mentioned earlier, reporting is one of Jenkins's strong points. We have seen how easy it is to display test results and to publish Javadocs, but you can also publish a large number of other very useful reports using Jenkins's plugins.

Plugins are another one of Jenkins's selling points—there are plugins for doing just about anything, from integrating new build tools or version control systems to notification mechanisms and reporting. In addition, Jenkins plugins are very easy to install and integrate smoothly into the existing Jenkins architecture.

To see how the plugins work, we are going to integrate code coverage metrics using the Cobertura plugin. Code coverage is an indication of how much of your application code is actually executed during your tests—it can be a useful tool in particular for finding areas of code that have not been tested by your test suites. It can also give some indication as to how well a team is applying good testing practices such as Test-Driven Development or Behavior-Driven Development.

Cobertura¹⁷ is an open source code coverage tool that works well with both Maven and Jenkins. Our Maven demonstration project is already configured to record code coverage metrics, so all we need to do is to install the Jenkins Cobertura plugin and generate the code coverage metrics for Jenkins to record and display.

¹⁷ <http://cobertura.sourceforge.net>

Updates		
Available	Installed	Advanced
Install	Name	Version
Artifact Uploaders		
<input type="checkbox"/>	ArtifactDeployer Plugin This plugin makes it possible to copy artifacts to remote locations.	0.1
<input type="checkbox"/>	Artifact Plugin This plugin allows deploying Maven 2, Maven 3, Ivy and Gradle artifacts and build info to the Artifactory artifacts manager.	2.0.0
<input type="checkbox"/>	Build Publisher Plugin This plugin allows records from one Hudson to be published on another Hudson.	1.10
<input type="checkbox"/>	CIFS-Publisher Plugin This plugin uploads build artifacts to repository sites using CIFS (SMB) protocol.	1.1.14
<input type="checkbox"/>	Confluence Publisher Plugin This plugin allows you to publish build artifacts as attachments to an Atlassian Confluence wiki page.	1.0.1
<input type="checkbox"/>	CopyArchiver Plugin The objective is to aggregate archived artifacts from several jobs into a shared directory. For each job, only archived artifacts of the last success build will be copied.	0.5.1
<input type="checkbox"/>	Deploy Plugin This plugin takes a war/ear file and deploys that to a running remote application server at the end of a build	1.7
<input type="checkbox"/>	Deploy WebSphere Plugin This plugin is an extension of the [Deploy Plugin]. It takes a war/ear file and deploys that to a running remote WebSphere Application Server at the end of a build.	1.0
<input type="checkbox"/>	Dimensions Plugin This plugin integrates Hudson with Dimensions , the Serena SCM solution.	0.8.1
<input type="checkbox"/>	FTP-Publisher Plugin This plugin can be used to upload project artifacts and whole directories to an ftp server.	1.0

Figure 2.27. Jenkins has a large range of plugins available

To install a new plugin, go to the Manage Jenkins page and click on the Manage Plugins entry. This will display a list of the available plugins as well as the plugins already installed on your server (see Figure 2.27, “Jenkins has a large range of plugins available”). If your build server doesn’t have an Internet connection, you can also manually install a plugin by downloading the plugin file elsewhere and uploading it to your Jenkins installation (just open the Advanced tab in Figure 2.27, “Jenkins has a large range of plugins available”), or by copying the plugin to the `$JENKINS_HOME/plugins` directory.

In our case, we are interested in the Cobertura plugin, so go to the Available tab and scroll down until you find the Cobertura Plugin entry in the Build Reports section. Click on the checkbox and then click on the Install button at the bottom of the screen.

This will download and install the plugin for you. Once it is done, you will need to restart your Jenkins instance to see the fruits of your labor. When you have restarted Jenkins, go back to the Manage Plugins screen and click on the Installed tab—there should now be a Cobertura Plugin entry in the list of installed plugins on this page.

Once you have made sure the plugin was successfully installed, go to the configuration page for the `gameoflife-default` build job.

To set up code coverage metrics in our project, we need to do two things. First we need to generate the Cobertura coverage data in an XML form that Jenkins can use; then we need to configure Jenkins to display the coverage reports.

Our Game of Life project already has been configured to generate XML code coverage reports if we ask it. All you need to do is to run `mvn cobertura:cobertura` to generate the reports in XML form. Cobertura can also generate HTML reports, but in our case we will be letting Jenkins take care of the reporting, so we can save on build time by not generating the For this example, for simplicity, we will just add the `cobertura:cobertura` goal to the second build step (see Figure 2.28, “Adding another Maven goal to generating test coverage metrics”). You could also add a new build step just for the code

coverage metrics. In a real-world project, code quality metrics like this are typically placed in a distinct build job, which is run less frequently than the default build.

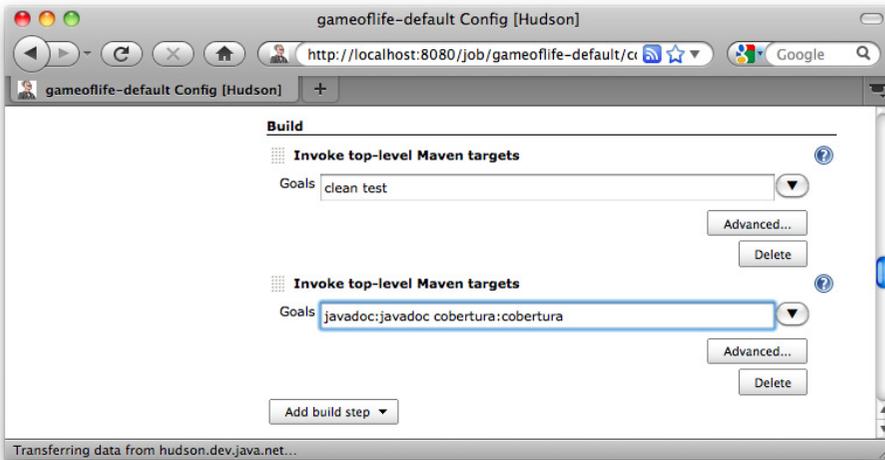


Figure 2.28. Adding another Maven goal to generating test coverage metrics

Next, we need to tell Jenkins to keep track of our code coverage metrics. Scroll down to the “Post-build Actions” section. You should see a new checkbox labeled Publish Cobertura Reports. Jenkins will often add UI elements like this when you install a new plugin. When you tick this box, Jenkins will display the configuration options for the Cobertura plugin that we installed earlier (see Figure 2.29, “Configuring the test coverage metrics in Jenkins”).

Like most of the code-quality related plugins in Jenkins, the Cobertura plugin lets you fine-tune not only the way Jenkins displays the report data, but also how it interprets the data. In the Coverage Metrics Targets section, you can define what you consider to be the minimum acceptable levels of code coverage. In Figure 2.29, “Configuring the test coverage metrics in Jenkins”, we have configured Jenkins to list any builds with less than 50% test coverage as “unstable” (indicated by a yellow ball), and notify the team accordingly.

Publish Cobertura Coverage Report

Cobertura xml report pattern

This is a file name pattern that can be used to locate the cobertura xml report files (for example with Maven2 use ****/target/site/cobertura/coverage.xml**). The path is relative to the module root unless you have configured your SCM with multiple modules, in which case it is relative to the workspace root. Note that the module root is SCM-specific, and may not be the same as the workspace root. Cobertura must be configured to generate XML reports for this plugin to function.

Consider only stable builds

Include only stable builds, i.e. exclude unstable and failed ones.

Coverage Metric Targets

Conditionals	<input type="button" value="Delete"/>	<input type="text" value="98"/>	<input type="text" value="75"/>	<input type="text" value="75"/>
Lines	<input type="button" value="Delete"/>	<input type="text" value="98"/>	<input type="text" value="75"/>	<input type="text" value="75"/>
Methods	<input type="button" value="Delete"/>	<input type="text" value="100"/>	<input type="text" value="80"/>	<input type="text" value="80"/>
Packages	<input type="button" value="Delete"/>	<input type="text" value="100"/>	<input type="text" value="95"/>	<input type="text" value="95"/>
<input type="button" value="Add"/>				

Configure health reporting thresholds.
For the  row, leave blank to use the default value (i.e. 80).
For the  and  rows, leave blank to use the default values (i.e. 0).

Figure 2.29. Configuring the test coverage metrics in Jenkins

This fine-tuning often comes in handy in real-world builds. For example, you may want to impose a special code coverage constraint in release builds, to ensure high code coverage in release versions. Another strategy that can be useful for legacy projects is to gradually increase the minimum tolerated code coverage level over time. This way you can avoid having to retro-fit unit tests on legacy code just to raise the code coverage, but you do encourage all new code and bug fixes to be well tested.

Now trigger a build manually. The first time you run the build job with Cobertura reporting activated, you will see coverage statistics for your build displayed on the build home page, along with a Coverage Report link when you can go for more details (see Figure 2.30, “Jenkins displays code coverage metrics on the build home page”). The Cobertura report shows different types of code coverage for the build we just ran. Since we have only run the test coverage metrics once, the coverage will be displayed as red and green bars.

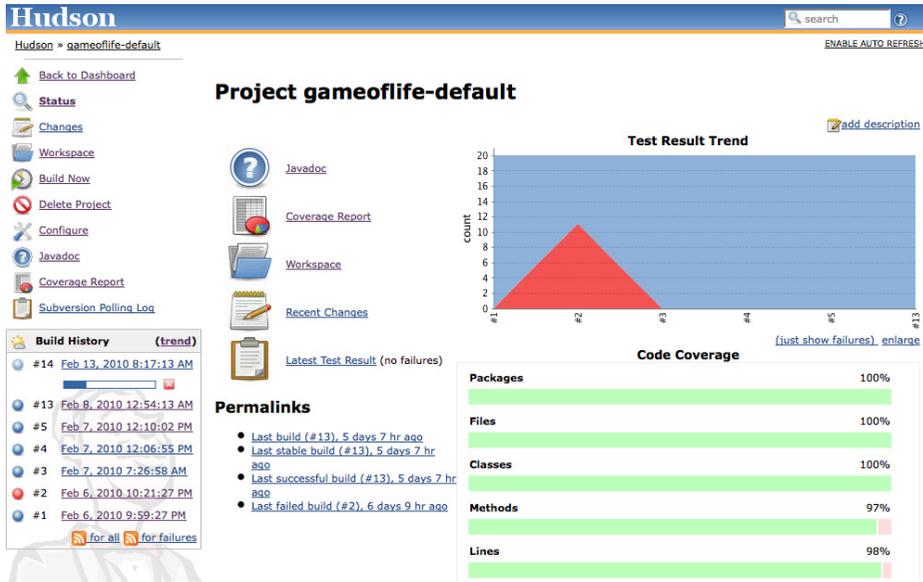


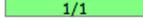
Figure 2.30. Jenkins displays code coverage metrics on the build home page

If you click on the Coverage Report icon, you will see code coverage for each package in your application, and even drill down to see the code coverage (or lack thereof) for an individual class (see Figure 2.31, “Jenkins lets you display code coverage metrics for packages and classes”). When you get to this level, Jenkins displays both the overall coverage statistics for the class, and also highlights the lines that were executed in green, and those that weren’t in red.

This reporting gets better with time. Jenkins not only reports metrics data for the latest build, but also keeps track of metrics over time, so that you can see how they evolve throughout the life of the project.

For example, if you drill down into the coverage reports, you will notice that certain parts of this code are not tested (for example the `Cell.java` class in Figure 2.31, “Jenkins lets you display code coverage metrics for packages and classes”).

File Coverage Summary

Name	Classes	Methods	Lines	Conditionals
Cell.java	100%  1/1	67%  2/3	71%  5/7	50%  3/6

Source

```
com/ciwithudson/gameoflife/domain/Cell.java
1 package com.ciwithudson.gameoflife.domain;
2
3 /**
4  * A single cell, which can be alive or dead.
5  */
6 819 abstract public class Cell {
7
8     public abstract Boolean isAlive();
9
10    public Boolean isDead() {
11 0      return !isAlive();
12    }
13
14    public abstract Cell nextGeneration(int neighbourCount);
15
16    public static Cell fromChar(char cellValue) {
17 92      if (cellValue == LivingCell.SYMBOL) {
18 35          return new LivingCell();
19 57      } else if (cellValue == DeadCell.SYMBOL) {
20 57          return new DeadCell();
21      }
22 0      throw new IllegalArgumentException("Illegal cell value character: " + cellValue);
23    }
24
25 }
```

Figure 2.31. Jenkins lets you display code coverage metrics for packages and classes

Code coverage metrics are a great way to isolate code that has not been tested, in order to add extra tests for corner cases that were not properly tested during the initial development, for example. The Jenkins code coverage graphs are also a great way of keeping track of your code coverage metrics as the project grows. Indeed, as you add new tests, you will notice that Jenkins will display a graph of code coverage over time, not just the latest results (see Figure 2.32, “Jenkins also displays a graph of code coverage over time”).

Code Coverage

Cobertura Coverage Report

Trend

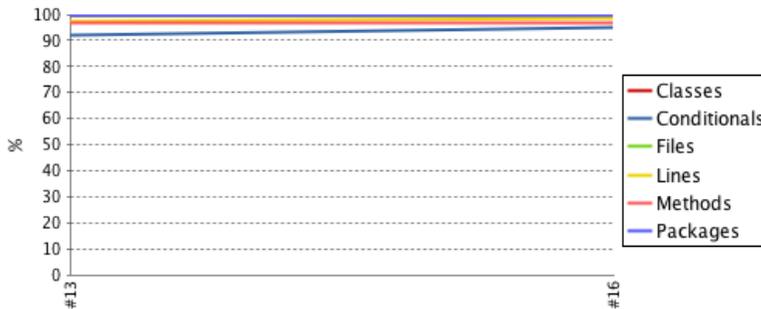


Figure 2.32. Jenkins also displays a graph of code coverage over time

Note that our objective here is not to improve the code coverage just for the sake of improving code coverage—we are adding an extra test to verify some code that was not previously tested, and as a result the code coverage goes up. There is a subtle but important difference here—code coverage, as with any other metric, is very much a means to an end (high code quality and low maintenance costs), and not an end in itself.

Nevertheless, metrics like this can give you a great insight into the health of your project, and Jenkins presents them in a particularly accessible way.

This is just one of the code quality metrics plugins that have been written for Jenkins. There are many more (over fifty reporting plugins alone at the time of writing). We'll look at some more of them in Chapter 9, *Qualité du Code*.

2.9. Conclusion

In this chapter, we have gone through what you need to know to get started with Jenkins. You should be able to set up a new build job, and setting up reporting on JUnit test results and javadocs. And you have seen how to add a reporting plugin and keep tabs on code coverage. Well done! But there's still a lot more to learn about Jenkins—in the following chapters, we will be looking at how Jenkins can help you improve your build automation process in many other areas as well.

Chapter 3. Installer Jenkins

3.1. Introduction

L'une des premières choses que vous avez dû remarquer à propos de Jenkins, c'est qu'il est très simple à installer. En effet, en moins de cinq minutes, vous pouvez avoir un serveur Jenkins installé et disponible. Néanmoins, comme toujours, dans le monde réel, les choses ne sont pas aussi simples, et il y a quelques détails auxquelles vous devez penser lorsque vous installez un serveur Jenkins en production. Dans ce chapitre, nous allons voir comment installer Jenkins sur votre poste de travail ou sur un véritable serveur. Nous allons également voir comment entretenir votre installation de Jenkins une fois en production, et comment réaliser les actions de maintenance simples comme les sauvegardes et les mises à jour.

3.2. Télécharger et installer Jenkins

Jenkins est simple à installer, et vous pouvez le faire tourner n'importe où. Vous pouvez le lancer soit comme une application, ou le déployer dans un conteneur d'application Java classique comme Tomcat ou JBoss. La première option est simple à mettre à oeuvre, et permet de tester Jenkins sur votre poste de travail, en quelques minutes, vous pouvez installer et lancer une version minimaliste de Jenkins.

Jenkins étant une application Java, vous aurez besoin d'une version récente de Java sur votre machine. Plus précisément, vous aurez besoin de Java 5. En fait, sur votre serveur, vous aurez même certainement besoin du Java Development Kit (JDK) 5.0 ou d'une version supérieure pour exécuter vos builds. Si vous n'en êtes pas sûr, vous pouvez vérifier en exécutant la commande suivante sur votre machine :

```
java -version
```

```
$
java -version
java version "1.6.0_17"
Java(TM) SE Runtime Environment (build 1.6.0_17-b04-248-10M3025)
Java HotSpot(TM) 64-Bit Server VM (build 14.3-b01-101, mixed mode)
```

Jenkins est disponible sous la forme d'une application Java packagée (un fichier WAR). Vous pouvez télécharger la dernière version sur le site web de Jenkins (<http://jenkins-ci.org> —voir Figure 3.1, “Vous pouvez télécharger les binaires de Jenkins sur le site web de Jenkins ”) ou depuis le site web du livre. Jenkins est un projet actif, et de nouvelles versions sont disponibles régulièrement.

Pour les utilisateurs de Windows, il existe un installateur Windows pour Jenkins. L'installateur se présente sous la forme d'un fichier zip contenant un package MSI pour Jenkins, ainsi qu'un fichier `setup.exe` pouvant être utilisé pour installer les bibliothèques .NET si elles n'ont pas déjà été installées sur votre poste. Dans la plupart des cas, tout ce que vous aurez à faire sera de décompresser le fichier zip et de lancer le fichier `jenkins-x.x.msi` (voir Figure 3.2, “L'assistant de configuration sous Windows”

). L'installateur MSI est livré avec une version du JRE intégré, il n'est donc pas nécessaire d'avoir une version de Java installée.



Figure 3.1. Vous pouvez télécharger les binaires de Jenkins sur le site web de Jenkins

Une fois que vous avez lancé l'installateur, Jenkins démarrera automatiquement sur le port 8080 (voir Figure 3.3, “La page d'accueil de Jenkins”). L'installateur créera un service Jenkins pour vous, que vous pourrez démarrer et arrêter comme n'importe quel autre service Windows.

Il existe également de très bon installateurs pour Mac OS X et également pour la plupart des distributions Linux, comme Ubuntu, RedHat (incluant CentOS et Fedora) et OpenSolaris. Nous expliquerons comment installer Jenkins sur Ubuntu et RedHat plus loin.

Si vous n'installez pas Jenkins via l'une des installations natives, vous pouvez simplement télécharger la dernière installation depuis le site web de Jenkins. Une fois que vous aurez téléchargé la dernière installation de Jenkins, copiez le dans un répertoire approprié sur votre serveur de build. Dans un environnement Windows, vous devriez installer Jenkins dans un répertoire comme `C:\Tools\Jenkins` (il est préférable de ne pas installer Jenkins dans un répertoire contenant des espaces, comme par exemple `C:\Program Files`, cela peut causer des problèmes dans certains cas). Sur un serveur Linux ou Unix, vous pouvez l'installer dans `/usr/local/jenkins`, `/opt/jenkins`, ou dans un autre répertoire, selon vos conventions et les préférences de votre administrateur système.

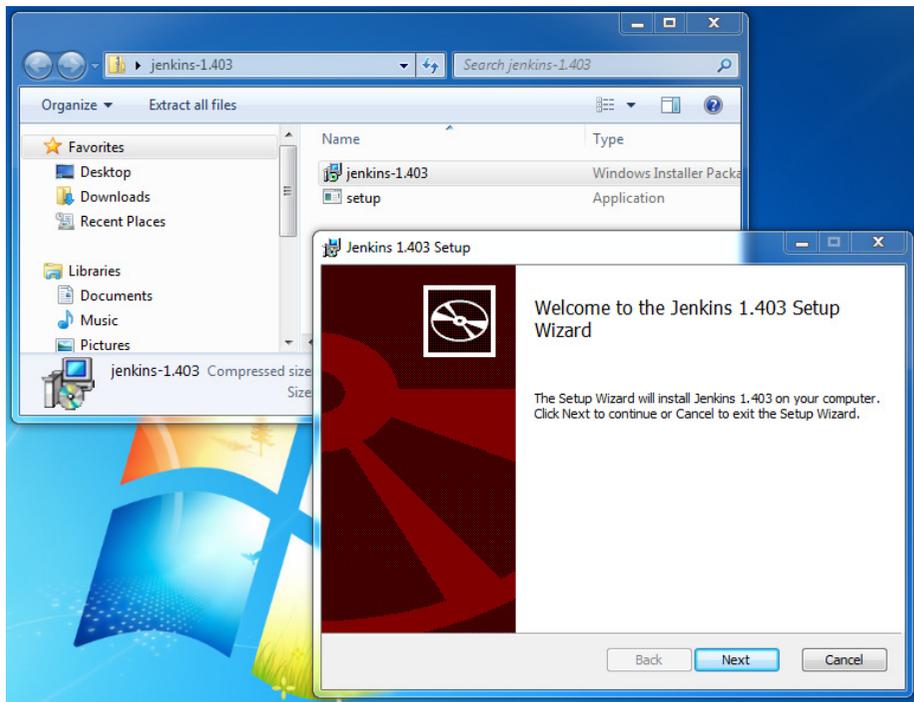


Figure 3.2. L'assistant de configuration sous Windows

Avant d'aller plus loin, démarrons simplement Jenkins et jetons-y un coup d'oeil. Si vous ne l'avez pas encore testé dans les chapitres précédents, il est temps de vous salir les mains. Ouvrez une invite de commande dans le répertoire contenant le fichier `jenkins.war` et lancez la commande suivante:

```
$  
java -jar jenkins.war  
[Winstone 2008/07/01 20:54:53] - Beginning extraction from war file  
...  
INFO: Took 35 ms to load  
...  
[Winstone 2008/07/01 20:55:08] - HTTP Listener started: port=8080  
[Winstone 2008/07/01 20:55:08] - Winstone Servlet Engine v0.9.10 running:  
controlPort=disabled  
[Winstone 2008/07/01 20:55:08] - AJPL3 Listener started: port=8009
```

Jenkins devrait être disponible sur le port 8080. Ouvrez votre navigateur et allez à l'adresse `http://localhost:8080` et jetez-y un oeil. (voir Figure 3.3, “La page d'accueil de Jenkins”).

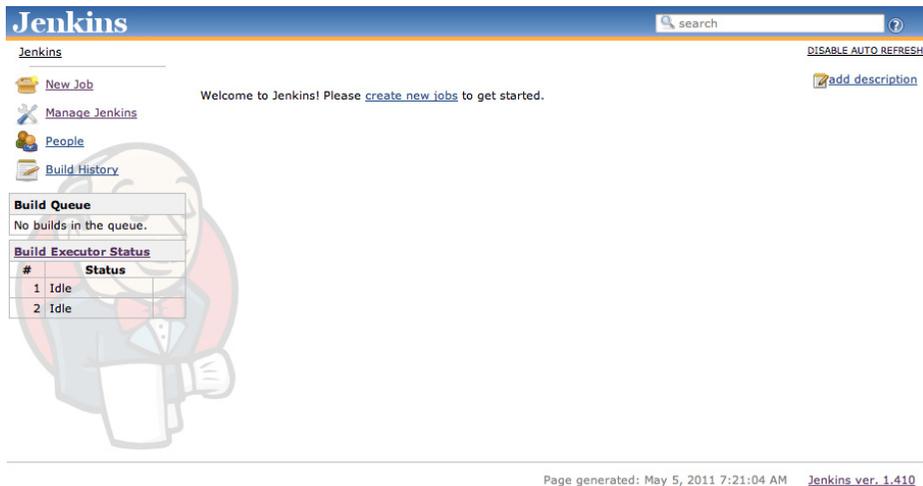


Figure 3.3. La page d'accueil de Jenkins

3.3. Préparation d'un serveur de build pour Jenkins

L'installation de Jenkins sur votre machine locale de développement est une chose, mais l'installation de Jenkins sur un bon serveur de build mérite un peu plus de prévoyance et de planification.

Avant de commencer votre installation, la première chose dont vous aurez besoin est un serveur de build. Pour fonctionner correctement, Jenkins a besoin à la fois d'un processeur puissant et de mémoire. Jenkins, pour sa part, est une application web Java relativement modeste. Cependant, pour la majorité des configurations, au moins une partie des builds sera exécutée sur le serveur de build principal. Les builds ont tendance à être à la fois gourmand en mémoire et en temps processeur, et Jenkins peut être configuré pour exécuter plusieurs builds en parallèle. Selon le nombre de tâches de build que vous gérez, Jenkins aura aussi besoin de mémoire dédiée pour son propre usage interne. La quantité de mémoire nécessaire dépendra en grande partie de la nature de vos builds, mais la mémoire n'est pas cher ces temps-ci (du moins pour des environnements non hébergés), et il vaut mieux ne pas être avare.

Un serveur de build a aussi besoin d'un CPU puissant. En règle générale, vous aurez besoin d'un processeur par build parallèle, même si, dans la pratique, vous pouvez capitaliser sur les délais d'E/S pour faire un peu mieux que cela. Il est également dans votre intérêt de dédier votre serveur de build, autant que possible, à la tâche de gestion des builds continus. En particulier, vous devriez éviter les applications gourmandes en mémoire ou en CPU tels que les serveurs de test, les applications d'entreprise fortement utilisées, les bases de données d'entreprise tel que Oracle, les serveurs de messagerie d'entreprise, et ainsi de suite.

Une option vraiment pratique, disponible dans de nombreuses organisations aujourd'hui, est d'utiliser une machine virtuelle. De cette façon, vous pouvez choisir la quantité de mémoire et le nombre de processeurs que vous jugez appropriés pour votre installation initiale, et d'ajouter facilement de la

mémoire et des processeurs plus tard si nécessaire. Cependant, si vous utilisez une machine virtuelle, assurez-vous qu'elle dispose de suffisamment de mémoire pour supporter le nombre maximal de builds en parallèle auquel vous vous attendez à être exécuté. L'utilisation de la mémoire d'un serveur d'intégration continue peut-être vue comme des dents de scie — Jenkins créera, au besoin, des JVMs supplémentaires pour ses tâches de build, et celles-ci ont besoin de mémoire.

Une autre approche utile est de configurer plusieurs machines de build. Jenkins rend facile la mise en place d'“esclaves” sur d'autres machines qui peuvent être utilisées pour exécuter des tâches de build additionnelles. Les esclaves restent inactifs jusqu'à ce qu'une nouvelle tâche de build soit demandée — puis l'installation principale de Jenkins envoie la tâche de build à un esclave et rend compte des résultats. C'est une excellente façon d'absorber les pointes soudaines de l'activité de build, par exemple juste avant une livraison majeure de votre principal produit. C'est aussi une stratégie utile si certains gros builds ont tendance à “accaparer” le serveur de build principal — il suffit de les mettre sur leur propre agent de build ! Nous verrons comment faire cela en détail plus loin dans ce livre.

Si vous installez Jenkins sur un serveur de build Linux ou Unix, cela peut-être une bonne idée de créer un utilisateur spécial (et un groupe utilisateur) pour Jenkins. Cela rend plus facile de surveiller d'un coup d'oeil les ressources système utilisées par les builds de Jenkins, et ainsi résoudre les builds qui posent problème en conditions réelles. Les paquets d'installation des binaires natifs décrits ci-dessous font cela pour vous. Si vous n'avez pas utilisé l'un d'entre eux, vous pouvez créer un utilisateur Jenkins dédié depuis la ligne de commande comme montré ici :

```
$  
sudo groupadd build  
$  
sudo useradd --create-home --shell  
  /bin/bash --groups build jenkins
```

Les détails exacts peuvent varier en fonction de votre environnement. Par exemple, vous préférerez peut-être utiliser une console d'administration graphique au lieu de la ligne de commande, ou, sur un serveur Linux basé sur une Debian (comme Ubuntu), vous pouvez utiliser des commandes plus conviviales comme `:adduser` et `addgroup` .

Dans la majorité des environnements, vous devrez configurer correctement Java pour cet utilisateur. Par exemple, vous pouvez le faire en définissant les variables `JAVA_HOME` et `PATH` dans le fichier `.bashrc` , comme montré ici :

```
export  
  JAVA_HOME=/usr/local/java/jdk1.6.0  
export PATH=$JAVA_HOME/bin:$PATH
```

Vous devriez maintenant être en mesure d'utiliser cet utilisateur pour exécuter Jenkins dans un environnement isolé.

3.4. Le répertoire de travail de Jenkins

Avant que nous installions Jenkins, toutefois, il y a certaines choses que vous devez savoir sur la façon dont Jenkins stocke ses données. En effet, peu importe où vous stockez le fichier WAR de Jenkins, Jenkins conserve toutes ses données importantes dans un répertoire spécial séparé appelé le répertoire de travail Jenkins. Ici, Jenkins stocke les informations sur la configuration de votre serveur de build, vos tâches de build, les artefacts de build, les comptes utilisateur, et d'autres informations utiles, ainsi que tous les plugins que vous avez installé. Le format du répertoire de travail de Jenkins est rétro compatible entre les versions, donc vous pouvez librement mettre à jour ou ré-installer votre exécutable Jenkins sans affecter votre répertoire de travail Jenkins.

Il va sans dire que ce répertoire aura besoin de beaucoup d'espace disque.

Par défaut, le répertoire de travail Jenkins sera appelé `.jenkins`, et sera placé dans votre répertoire de travail. Par exemple, si vous utilisez une machine sous Windows 7 et si votre nom d'utilisateur est "john", vous trouverez le répertoire de travail Jenkins sous `C:\Users\john\.jenkins`. Sous Windows XP, ce serait `C:\Documents and Settings\John\.jenkins`. Sur une machine Linux, ce serait probablement sous `/home/john/.jenkins`. Et ainsi de suite.

Vous pouvez forcer Jenkins à utiliser un répertoire différent pour son répertoire de travail en définissant la variable d'environnement `JENKINS_HOME`. Vous pourriez avoir besoin de le faire sur un serveur de build pour vous conformer aux conventions du répertoire local ou pour rendre votre administrateur système heureux. Par exemple, si votre fichier WAR Jenkins est installé dans `/usr/local/jenkins`, et que le répertoire de travail Jenkins a besoin d'être dans le répertoire `/data/jenkins`, vous pourriez écrire un script de démarrage comme suit :

```
export
  JENKINS_BASE=/usr/local/jenkins
export JENKINS_HOME=/var/jenkins-data
java -jar ${JENKINS_BASE}/jenkins.war
```

Si vous utilisez Jenkins dans un conteneur Java EE comme Tomcat ou JBoss, vous pouvez configurer la webapp pour exposer ses propres variables d'environnement. Par exemple, si vous utilisez Tomcat, vous pouvez créer un fichier appelé `jenkins.xml` dans le répertoire `$CATALINA_BASE/conf/localhost` :

```
<Context
  docBase="../jenkins.war">
  <Environment name="JENKINS_HOME" type="java.lang.String"
  value="/data/jenkins" override="true"/>
</Context>
```

Dans une vie antérieure, Jenkins était connu sous le nom de Hudson. Jenkins reste compatible avec les installations précédentes de Hudson, et le passage de Hudson à Jenkins peut-être aussi simple que de remplacer l'ancien fichier `hudson.war` par `jenkins.war`. Jenkins cherchera son répertoire de travail dans les places suivantes (par ordre de préséance):

1. Une entrée d'environnement JNDI appelée JENKINS_HOME
2. Une entrée d'environnement JNDI appelée HUDSON_HOME
3. Une propriété système nommée JENKINS_HOME
4. Une propriété système nommée HUDSON_HOME
5. Une variable d'environnement nommée JENKINS_HOME
6. Une variable d'environnement nommée HUDSON_HOME
7. Le répertoire `.hudson` dans le répertoire de travail utilisateur, s'il existe déjà
8. Le répertoire `.jenkins` dans le répertoire de travail utilisateur

3.5. Installer Jenkins sur Debian ou Ubuntu

Si vous installez Jenkins sur Debian et Ubuntu, il est commode d'installer le paquet binaire natif pour ces plates-formes. Cela est assez simple à faire, même si ces binaires ne sont pas fournis dans les dépôts standard en raison de la fréquence élevée des mises à jour. Premièrement, vous devez ajouter la clé à votre système comme indiqué ici:

```
$  
wget -q -O -  
  http://pkg.jenkins-ci.org/debian/jenkins-ci.org.key \  
| sudo apt-key add -  
$  
sudo echo "deb  
  http://pkg.jenkins-ci.org/debian binary/" > \  
/etc/apt/sources.list.d/jenkins.list
```

Maintenant, mettez à jour le référentiel des paquets Debian :

```
$  
sudo aptitude update
```

Une fois que cela est fait, vous pouvez installer Jenkins en utilisant l'outil `aptitude` :

```
$  
sudo aptitude install -y jenkins
```

Ceci installera Jenkins comme un service, avec un script de démarrage correctement configuré dans `/etc/init.d/jenkins` et un utilisateur système correspondant nommé "jenkins". Si vous n'avez pas

déjà Java d'installé sur votre serveur, ceci installera aussi la verion OpenJDK de Java. Par défaut, vous trouverez le fichier WAR de Jenkins dans le répertoire `/usr/share/jenkins`, et le répertoire de travail de Jenkins dans `/var/lib/jenkins`.

Le processus d'installation devrait avoir démarré Jenkins. En général, pour démarrer Jenkins, exécutez simplement ce script:

```
$ sudo /etc/init.d/jenkins start
```

Jenkins va maintenant être exécuté sur le port par défaut : 8080 (`http://localhost:8080/`).

Vous pouvez arrêter Jenkins comme il suit :

```
$ sudo /etc/inid.d/jenkins stop
```

Jenkins écrira les fichiers de journalisation dans `/var/log/jenkins/jenkins.log`. Vous pouvez aussi affiner les paramètres de configuration dans le fichier `/etc/default/jenkins`. Ceci peut-être utile si vous avez besoin de modifier les arguments de de démarrage de Java (`JAVA_ARGS`). Vous pouvez aussi utiliser ce fichier pour configurer les arguments qui seront passés à Jenkins, comme le port HTTP ou le contexte d'application `web` (voir Section 3.8, “Exécuter Jenkins comme une application autonome”).

3.6. Installer Jenkins sur Redhat, Fedora ou CentOS

Il existe également des paquets binaires natifs pour Redhat, Fedora et CentOS. Il vous faut d'abord configurer le référentiel comme il suit :

```
$ sudo wget -O
/etc/yum.repos.d/jenkins.repo \
http://jenkins-ci.org/redhat/jenkins.repo
$ sudo rpm --import
http://pkg.jenkins-ci.org/redhat/jenkins-ci.org.key
```

Sur une nouvelle installation, vous devrez peut-être installer le JDK:

```
$ sudo yum install java-1.6.0-openjdk
```

Ensuite, vous pouvez installer le paquet comme montré ici :

```
$  
sudo yum install jenkins
```

Ceci installera la dernière version de Jenkins dans le répertoire `/usr/lib/jenkins`. Le répertoire de travail par défaut de Jenkins sera `/var/lib/jenkins`.

Vous pouvez maintenant démarrer Jenkins en utilisant la commande de `service` :

```
$  
sudo service jenkins start
```

Jenkins va maintenant être exécuté sur le port par défaut : 8080 (`http://localhost:8080/`).

Les paramètres de configuration de Jenkins sont placés dans le fichier `/etc/sysconfig/jenkins`. Cependant, au moment de l'écriture de ce livre, les options de configuration sont plus limitées que celles fournies par le paquet Ubuntu : vous pouvez définir le port HTTP en utilisant le paramètre `JENKINS_PORT`, par exemple, mais pour spécifier un contexte d'application vous devez modifier le script de démarrage à la main. Les principales options de configuration sont listées ici :

`JENKINS_JAVA_CMD`

La version de Java que vous voulez utiliser pour exécuter Jenkins

`JENKINS_JAVA_OPTIONS`

Les options de ligne de commande à passer à Java, tels que les options de mémoire

`JENKINS_PORT`

Le port sur lequel Jenkins sera exécuté

3.7. Installer Jenkins sur SUSE ou OpenSUSE

Les paquets binaires sont aussi disponibles pour SUSE et OpenSUSE, donc le processus d'installation process sur ces plates-formes est simple. Premièrement, vous devez ajouter le dépôt Jenkins à la liste des dépôts SUSE :

```
$  
sudo zypper addrepo  
http://pkg.jenkins-ci.org/opensuse/ jenkins
```

Enfin, il vous suffit d'installer Jenkins en utilisant la commande `zypper` :

```
$  
sudo zypper install jenkins
```

Comme vous pouvez le voir sur la sortie de la console, ceci installera à la fois Jenkins et le dernier JDK de Sun, si ce dernier n'est pas déjà installé. Les installations OpenSuse ont généralement la version OpenJDK de Java, mais Jenkins préfère celle de Sun. Lors du téléchargement JDK de Sun, il vous sera demandé de valider la licence de Java Sun avant de continuer l'installation.

Ce processus d'installation créera aussi un utilisateur `jenkins` et installera Jenkins en tant que service, de sorte qu'il se lancera automatiquement à chaque démarrage de la machine. Pour démarrer manuellement Jenkins, vous pouvez appeler le script de démarrage `jenkins` depuis le répertoire `/etc/init.d`:

```
$  
sudo /etc/init.d/jenkins jenkins start
```

Jenkins sera maintenant exécuté par défaut sur le port 8080 (`http://localhost:8080/`).

Les options de configuration sont similaires à celles de l'installation pour Redhat (voir Section 3.6, "Installer Jenkins sur Redhat, Fedora ou CentOS"). Vous pouvez définir un nombre limité de variables de configuration dans le fichier `/etc/sysconfig/jenkins` , mais pour toutes les options de configuration avancées, vous devrez modifier le script de démarrage dans `/etc/init.d/jenkins` .

L'outil `zypper` facilite aussi la mise à jour de votre instance Jenkins:

```
$  
sudo zypper update jenkins
```

Ceci téléchargera et installera la dernière version de Jenkins depuis le site de Jenkins .

3.8. Exécuter Jenkins comme une application autonome

Vous pouvez exécuter le serveur Jenkins d'une des deux manières suivantes : soit comme une application autonome, soit déployé comme une application web standard sur un conteneur de servlets Java ou un serveur d'application comme Tomcat, JBoss, ou GlassFish. Les deux approches ont leurs avantages et leurs inconvénients, nous allons donc examiner les deux ici.

Jenkins est fourni sous la forme d'un fichier WAR que vous pouvez exécuter directement en utilisant un conteneur de servlet intégré. Jenkins utilise le moteur de servlet léger Winstone pour vous permettre d'exécuter le serveur directement, sans avoir à configurer un serveur web par vous-même. C'est probablement la meilleure façon de commencer, vous permettant d'être opérationnel avec Jenkins en quelques minutes. C'est aussi une option très flexible, offrant quelques fonctionnalités supplémentaires

non-accessibles si vous déployez Jenkins sur un serveur d'application classique. En particulier, si vous exécutez Jenkins en tant que serveur autonome, vous serez en mesure d'installer et mettre à jour les plugins à la volée, et de redémarrer Jenkins directement depuis les écrans d'administration.

Pour exécuter Jenkins en utilisant le conteneur de servlet intégré, allez à la ligne de commande et entrez la commande suivante :

```
C:\Program
Files\Jenkins>
java -jar jenkins.war
[Winstone 2011/07/01 20:54:53] - Beginning extraction from war file
[Winstone 2011/07/01 20:55:07] - No webapp classes folder found -
C:\Users\john\
.jenkins\war\WEB-INF\classes
jenkins home directory: C:\Users\john\.jenkins
...
INFO: Took 35 ms to load
...
[Winstone 2011/07/01 20:55:08] - HTTP Listener started: port=8080
[Winstone 2011/07/01 20:55:08] - Winstone Servlet Engine v0.9.10 running:
controlPort=disabled
[Winstone 2011/07/01 20:55:08] - AJPl3 Listener started: port=8009
```

Dans un environnement Linux, la procédure est similaire. Notez comment nous démarrons le serveur Jenkins à partir du compte utilisateur “jenkins” que nous avons créé plus tôt :

```
john@lambton:~$ sudo su - jenkins
jenkins@lambton:~$ java -jar /usr/local/jenkins/jenkins.war
[Winstone 2011/07/16 02:11:24] - Beginning extraction from war file
[Winstone 2011/07/16 02:11:27] - No webapp classes folder found -
/home/jenkins/
.jenkins/war/WEB-INF/classes
jenkins home directory: /home/jenkins/.jenkins
...
[Winstone 2011/07/16 02:11:31] - HTTP Listener started: port=8080
[Winstone 2011/07/16 02:11:31] - AJPl3 Listener started: port=8009
[Winstone 2011/07/16 02:11:31] - Winstone Servlet Engine v0.9.10 running:
controlPort=disabled
```

Cela démarrera le moteur de servlet intégré dans la fenêtre de la console. L'application Web Jenkins sera maintenant disponible sur le port 8080. Lorsque vous exécutez Jenkins en utilisant le serveur intégré, il n'y a pas de contexte d'application Web, donc vous accédez à Jenkins directement en utilisant l'URL du serveur (e.g., <http://localhost:8080>).

Pour arrêter Jenkins, pressez simplement Ctrl-C.

Par défaut, Jenkins s'exécutera sur le port 8080. Si cela ne convient pas à votre environnement, vous pouvez spécifier le port manuellement, en utilisant l'option `--httpPort` :

```
$  
java -jar jenkins.war --httpPort=8081
```

Dans une architecture réelle, Jenkins peut ne pas être la seule application Web à s'exécuter sur votre serveur de build. Selon la capacité de votre serveur, Jenkins peut avoir à cohabiter avec d'autres applications Web ou des gestionnaires de dépôts Maven, par exemple. Si vous exécutez Jenkins aux côtés d'un autre serveur d'application, tels que Tomcat, Jetty ou GlassFish, vous aurez aussi besoin de remplacer le port `ajp13`, en utilisant l'option `--ajp13Port` :

```
$  
java -jar jenkins.war --httpPort=8081  
--ajp13Port=8010
```

Quelques autres options utiles sont :

`--prefix`

Cet option vous permet de définir un chemin de contexte pour votre serveur Jenkins. Par défaut Jenkins s'exécutera sur le port 8080 sans chemin de contexte (`http://localhost:8080`). Toutefois, si vous utilisez cette option, vous pouvez forcer Jenkins à utiliser n'importe quel chemin de contexte qui vous plaît, par exemple:

```
$  
java -jar jenkins.war  
--prefix=jenkins
```

Dans ce cas, Jenkins sera accessible depuis `http://localhost:8080/jenkins` .

Cette option est souvent utilisée lors de l'intégration d'une instance autonome de Jenkins avec Apache.

`--daemon`

Si vous exécutez Jenkins sur une machine Unix, vous pouvez utiliser cette option pour démarrer Jenkins comme une tâche de fond, s'exécutant comme un démon unix.

`--logfile`

Par défaut, Jenkins écrit son fichier de journalisation dans le répertoire courant. Cependant, sur un serveur, vous avez souvent besoin d'écrire vos fichiers de journalisation dans un répertoire prédéterminé. Vous pouvez utiliser cette option pour rediriger vos messages vers un autre fichier :

```
$  
java -jar jenkins.war  
--logfile=/var/log/jenkins.log
```

Stopping Jenkins using Ctrl-C is a little brutal, of course—in practice, you would set up a script to start and stop your server automatically.

If you are running Jenkins using the embedded Winstone application server, you can also restart and shutdown Jenkins elegantly by calling the Winstone server directly. To do this, you need to specify the `controlPort` option when you start Jenkins, as shown here:

```
$
java -jar jenkins.war
  --controlPort=8001
```

A slightly more complete example in a Unix environment might look like this:

```
$
nohup java -jar jenkins.war
  --controlPort=8001 > /var/log/jenkins.log 2>&1 &
```

The key here is the `controlPort` option. This option gives you the means of stopping or restarting Jenkins directly via the Winstone tools. The only problem is that you need a matching version of the Winstone JAR file. Fortunately, one comes bundled with your Jenkins installation, so you don't have to look far.

To restart the server, you can run the following command:

```
$
java -cp $JENKINS_HOME/war/winstone.jar
  winstone.tools.WinstoneControl reload: \
  --host=localhost --port=8001
```

And to shut it down completely, you can use the following:

```
$
java -cp $JENKINS_HOME/war/winstone.jar
  winstone.tools.WinstoneControl shutdown \
  --host=localhost --port=8001
```

Another way to shut down Jenkins cleanly is to invoke the special “/exit” URL, as shown here:

```
$
wget http://localhost:8080/exit
```

On a real server, you would typically have set up security, so that only a system administrator could access this URL. In this case, you will need to provide a username and a password:

```
$
wget --user=admin --password=secret
http://localhost:8080/exit
```

Note that you can actually do this from a different server, not just the local machine:

```
$
wget --user=admin --password=secret
http://buildserver.acme.com:8080/exit
```

Note that while both these methods will shut down Jenkins relatively cleanly (more so than killing the process directly, for example), they will interrupt any builds in progress. So it is recommended practice to prepare the shutdown cleanly by using the Prepare for Shutdown button on the Manage Jenkins screen (see Section 4.2, “Le tableau de bord de configuration — L’écran Administrer Jenkins”).

Running Jenkins as a stand-alone application may not be to everyone’s taste. For a production server, you might want to take advantage of the more sophisticated monitoring and administration features of a full blown Java application server such as JBoss, GlassFish, or WebSphere Application Server. And system administrators may be wary of the relatively little-known Winstone server, or may simply prefer Jenkins to fit into a known pattern of Java web application development. If this is the case, you may prefer to, or be obliged to, deploy Jenkins as a standard Java web application. We look at this option in the following section.

3.9. Running Jenkins Behind an Apache Server

If you are running Jenkins in a Unix environment, you may want to hide it behind an Apache HTTP server in order to harmonize the server URLs and simplify maintenance and access. This way, users can access the Jenkins server using a URL like `http://myserver.myorg.com/jenkins` rather than `http://myserver.myorg.com:8081` .

One way to do this is to use the Apache `mod_proxy` and `mod_proxy_ajp` modules. These modules let you use implement proxying on your Apache server using the AJP13 (Apache JServer Protocol version 1.3). Using this module, Apache will transfer requests to particular URL patterns on your Apache server (running on port 80) directly to the Jenkins server running on a different port. So when a user opens a URL like `http://www.myorg.com/jenkins` , Apache will transparently forward traffic to your Jenkins server running on `http://buildserver.myorg.com:8081/jenkins` .Technically, this is known as “Reverse Proxying,” as the client has no knowledge that the server is doing any proxying, or where the proxied server is located. So you can safely tuck your Jenkins server away behind a firewall, while still providing broader access to your Jenkins instance via the public-facing URL.

The exact configuration of this module will vary depending on the details of your Apache version and installation details, but one possible approach is shown here.

First of all, if you are running Jenkins as a stand-alone application, make sure you start up Jenkins using the `--prefix` option. The prefix you choose must match the suffix in the public-facing URL you want to use. So if you want to access Jenkins via the URL `http://myserver.myorg.com/jenkins`, you will need to provide `jenkins` as a prefix:

```
$
java -jar jenkins.war --httpPort=8081
  --ajp13Port=8010 --prefix=jenkins
```

If you are running Jenkins on an application server such as Tomcat, it will already be running under a particular web context (`/jenkins` by default).

Next, make sure the `mod_proxy` and `mod_proxy_ajp` modules are activated. In your `httpd.conf` file (often in the `/etc/httpd/conf` directory), you should have the following line:

```
LoadModule proxy_module modules/mod_proxy.so
```

The proxy is actually configured in the `proxy_ajp.conf` file (often in the `/etc/httpd/conf.d` directory). Note that the name of the proxy path (`/jenkins` in this example) must match the prefix or web context that Jenkins is using. An example of such a configuration file is given here:

```
LoadModule proxy_ajp_module
modules/mod_proxy_ajp.so

ProxyPass /jenkins http://localhost:8081/jenkins
ProxyPassReverse /jenkins http://localhost:8081/jenkins
ProxyRequests Off
```

Once this is done, you just need to restart your Apache server:

```
$
sudo /etc/init.d/httpd restart
Stopping httpd: [ OK ]
Starting httpd: [ OK ]
```

Now you should be able to access your Jenkins server using a URL like `http://myserver.myorg.com/jenkins`.

3.10. Running Jenkins on an Application Server

Since Jenkins is distributed as an ordinary WAR file, it is easy to deploy it on any standard Java application server such as Tomcat, Jetty, or GlassFish. Running Jenkins on an application server is

arguably more complicated to setup and to maintain. You also lose certain nice administration features such as the ability to upgrade Jenkins or restart the server directly from within Jenkins. On the other hand, your system administrators might be more familiar with maintaining an application running on Tomcat or GlassFish than on the more obscure Winstone server.

Let's look at how you would typically deploy Jenkins onto a Tomcat server. The easiest approach is undoubtedly to simply unzip the Tomcat binary distribution onto your disk (if it is not already installed) and copy the `jenkins.war` file into the Tomcat `webapps` directory. You can download the Tomcat binaries from the Tomcat website¹.

You start Tomcat by running the `startup.bat` or `startup.sh` script in the Tomcat bin directory. Jenkins will be available when you start Tomcat. You should note that, in this case, Jenkins will be executed in its own web application context (typically “jenkins”), so you will need to include this in the URL you use to access your Jenkins server (e.g., `http://localhost:8080/jenkins`).

However, this approach is not necessarily the most flexible or robust option. If your build server is a Windows box, for example, you probably should install Tomcat as a Windows service, so that you can ensure that it starts automatically whenever the server reboots. Similarly, if you are installing Tomcat in a Unix environment, it should be set up as a service.

3.11. Memory Considerations

Continuous Integration servers use a lot of memory. This is the nature of the beast—builds will consume memory, and multiple builds being run in parallel will consume still more memory. So you should ensure that your build server has enough RAM to cope with however many builds you intend to run simultaneously.

Jenkins naturally needs RAM to run, but if you need to support a large number of build processes, it is not enough just to give Jenkins a lot of memory. In fact Jenkins spans a new Java process each time it kicks off a build, so during a large build, the build process needs the memory, not Jenkins.

You can define build-specific memory options for your Jenkins build jobs—we will see how to do this later on in the book. However if you have a lot of builds to maintain, you might want to define the `JAVA_OPTS`, `MAVEN_OPTS` and `ANT_OPTS` environment variables to be used as default values for your builds. The `JAVA_OPTS` options will apply for the main Jenkins process, whereas the other two options will be used when Jenkins kicks off new JVM processes for Maven and Ant build jobs respectively.

Here is an example of how these variables might be configured on a Unix machine in the `.profile` file:

```
export
  JAVA_OPTS=-Djava.awt.headless=true -Xmx512m
  -DJENKINS_HOME=/data/jenkins
export MAVEN_OPTS="-Xmx512m -XX:MaxPermSize=256m"
export ANT_OPTS="-Xmx512m -XX:MaxPermSize=256m"
```

¹ <http://tomcat.apache.org>

3.12. Installing Jenkins as a Windows Service

If you are running a production installation of Jenkins on a Windows box, it is essential to have it running as a Windows service. This way, Jenkins will automatically start whenever the server reboots, and can be managed using the standard Windows administration tools.

One of the advantages of running Jenkins on an application server such as Tomcat is that it is generally fairly easy to configure these servers to run as a Windows service. However, it is also fairly easy to install Jenkins as a service, without having to install Tomcat.

Jenkins has a very convenient feature designed to make it easy to install Jenkins as a Windows servers. There is currently no graphical installer that does this for you, but you get the next best thing—a web-based graphical installer.

First, you need to start the Jenkins server on your target machine. The simplest approach is to run Jenkins using Java Web Start (see Figure 3.4, “Starting Jenkins using Java Web Start”). Alternatively, you can do this by downloading Jenkins and running it from the command line, as we discussed earlier:

```
C:\jenkins>  
java -jar jenkins.war
```

This second option is useful if the default Jenkins port (8080) is already being used by another application. It doesn't actually matter which port you use—you can change this later.

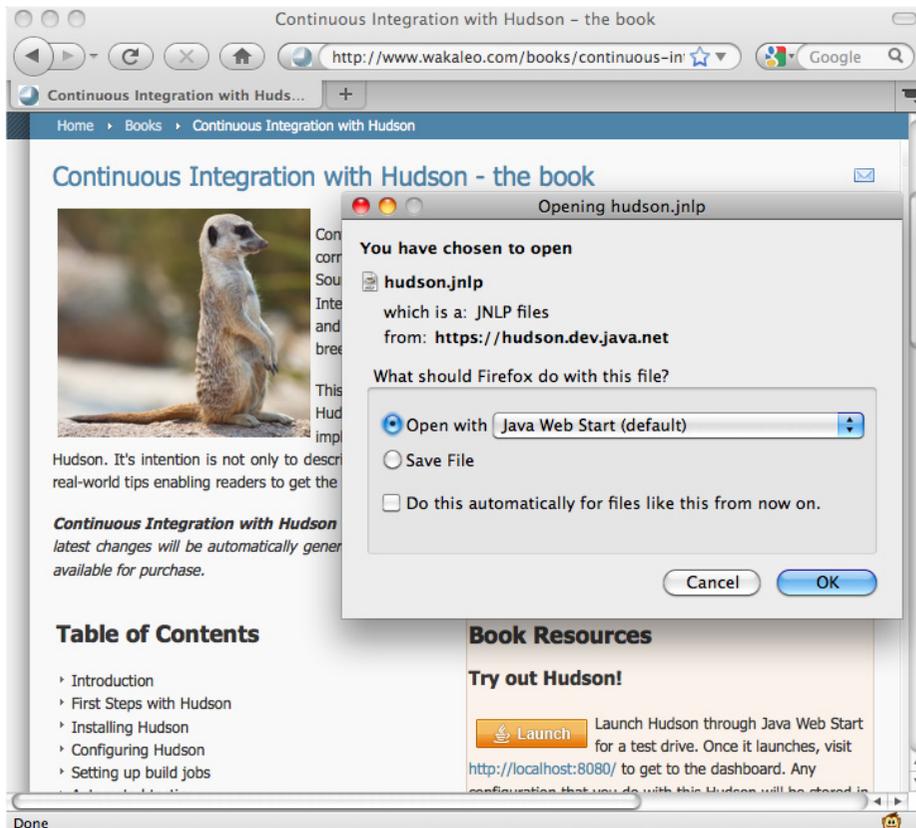


Figure 3.4. Starting Jenkins using Java Web Start

Once you have Jenkins running, connect to this server and go to the Manage Jenkins screen. Here you will find an Install as Windows Service button. This will create a Jenkins service on the server that will automatically start and stop Jenkins in an orderly manner (see Figure 3.5, “Installing Jenkins as a Windows service”).

Jenkins will prompt you for an installation directory. This will be the Jenkins home directory (`JENKINS_HOME`). The default value is the default `JENKINS_HOME` value: a directory called `.jenkins` in the current user’s home directory. This is often not a good choice for a Windows installation. When running Jenkins on Windows XP, you should avoid installing your Jenkins home directory anywhere near your `C:\Documents And Settings` directory—not only is it a ridiculously long name, the spaces can wreak havoc with your Ant and Maven builds and any tests using classpath-based resources. It is much better to use a short and sensible name such as `C:\Jenkins`. The Vista and Windows 7 home directory paths like `C:\Users\john` will also work fine.

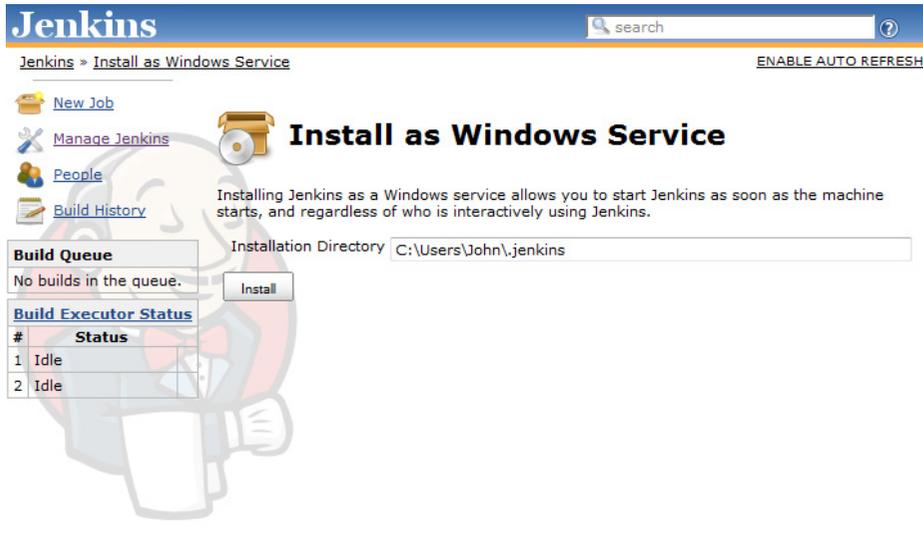


Figure 3.5. Installing Jenkins as a Windows service

A short home directory path is sometimes required for other reasons, too. On many versions of Windows (Windows XP, Windows Server 2003, etc.), file path lengths are limited to around 260 characters. If you combine a nested Jenkins work directory and a deep class path, you can often overrun this, which will result in very obscure build errors. To minimize the risks of over-running the Windows file path limits, you need to redefine the `JENKINS_HOME` environment variable to point to a shorter path, as we discussed above.

This approach won't always work with Windows Vista or Windows 7. An alternative strategy is to use the `jenkins.exe` program that the Web Start installation process will have installed in the directory you specified above. Open the command line prompt as an administrator (right-click, "Run as administrator") and run the `jenkins.exe` executable with the `install` option:

```
C:\Jenkins>
jenkins.exe install
```

This basic installation will work fine in a simple context, but you will often need to fine-tune your service. For example, by default, the Jenkins service will be running under the local System account. However, if you are using Maven, Jenkins will need an `.m2` directory and a `settings.xml` file in the home directory. Similarly, if you are using Groovy, you might need a `.groovy/lib` directory. And so on. To allow this, and to make testing your Jenkins install easier, make sure you run this service under a real user account with the correct development environment set up (see Figure 3.6, "Configuring the Jenkins Windows Service"). Alternatively, run the application as the system user, but use the System Information page in Jenkins to check the `/jenkins-guide-complet` directory, and place any files that must be placed in the user home directory here.

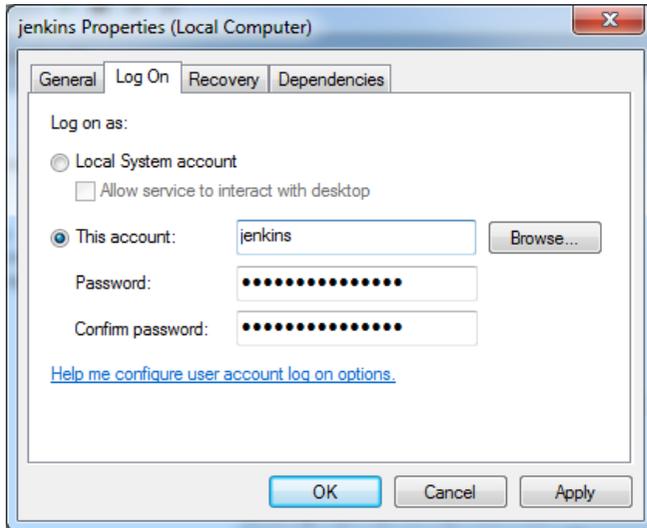


Figure 3.6. Configuring the Jenkins Windows Service

You configure the finer details of the Jenkins service in a file called `jenkins.xml`, in the same directory as your `jenkins.war` file. Here you can configure (or reconfigure) ports, JVM options, and the Jenkins work directory. In the following example, we give Jenkins a bit more memory and get it to run on port 8081:

```
<service>
  <id>jenkins</id>
  <name>Jenkins</name>
  <description>This service runs the Jenkins continuous integration system
</description>
  <env name="JENKINS_HOME" value="D:\jenkins" />
  <executable>java</executable>
  <arguments>-Xrs -Xmx512m
-Dhudson.lifecycle=hudson.lifecycle.WindowsServiceLifecycle
-jar "%BASE%\jenkins.war" --httpPort=8081
--ajp13Port=8010</arguments>
</service>
```

Finally, if you need to uninstall the Jenkins service, you can do one of two things. The simplest is to run the Jenkins executable with the `uninstall` option:

```
C:\jenkins>
jenkins.exe uninstall
```

The other option is to use the Windows `service` tool `sc`:

```
C:>
```

3.13. What's in the Jenkins Home Directory

The Jenkins home directory contains all the details of your Jenkins server configuration, details that you configure in the Manage Jenkins screen. These configuration details are stored in the form of a set of XML files. Much of the core configuration, for example, is stored in the `config.xml` file. Other tools-specific configuration is stored in other appropriately-named XML files: the details of your Maven installations, for example, are stored in a file called `hudson.tasks.Maven.xml`. You rarely need to modify these files by hand, though occasionally it can come in handy.

The Jenkins home directory also contains a number of subdirectories (see Figure 3.7, “The Jenkins home directory”). Not all of the files and directories will be present after a fresh installation, as some are created when required by Jenkins. And if you look at an existing Jenkins installation, you will see additional XML files relating to Jenkins configuration and plugins.

The main directories are described in more detail in Table 3.1, “The Jenkins home directory structure”.

Table 3.1. The Jenkins home directory structure

Directory	Description
<code>.jenkins</code>	The default Jenkins home directory (may be <code>.hudson</code> in older installations).
<code>fingerprints</code>	This directory is used by Jenkins to keep track of artifact fingerprints. We look at how to track artifacts later on in the book.
<code>jobs</code>	This directory contains configuration details about the build jobs that Jenkins manages, as well as the artifacts and data resulting from these builds. We look at this directory in detail below.
<code>plugins</code>	This directory contains any plugins that you have installed. Plugins allow you to extend Jenkins by adding extra feature. Note that, with the exception of the Jenkins core plugins (subversion, cvs, ssh-slaves, maven, and scid-ad), plugins are not stored with the <code>jenkins</code> executable, or in the expanded web application directory. This means that you can update your Jenkins executable and not have to reinstall all your plugins.
<code>updates</code>	This is an internal directory used by Jenkins to store information about available plugin updates.
<code>userContent</code>	You can use this directory to place your own custom content onto your Jenkins server. You can access files in this directory at http://myserver/hudson/userContent (if you are running Jenkins on

Directory	Description
	an application server) or <code>http://myserver/userContent</code> (if you are running in stand-alone mode).
<code>users</code>	If you are using the native Jenkins user database, user accounts will be stored in this directory.
<code>war</code>	This directory contains the expanded web application. When you start Jenkins as a stand-alone application, it will extract the web application into this directory.

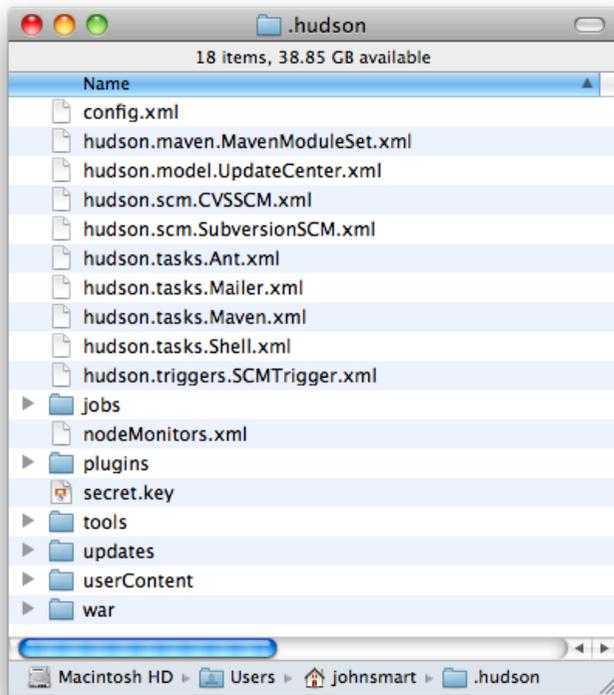


Figure 3.7. The Jenkins home directory

The `jobs` directory is a crucial part of the Jenkins directory structure, and deserves a bit more attention. You can see an example of a real Jenkins jobs directory in Figure 3.8, “The Jenkins jobs directory” .

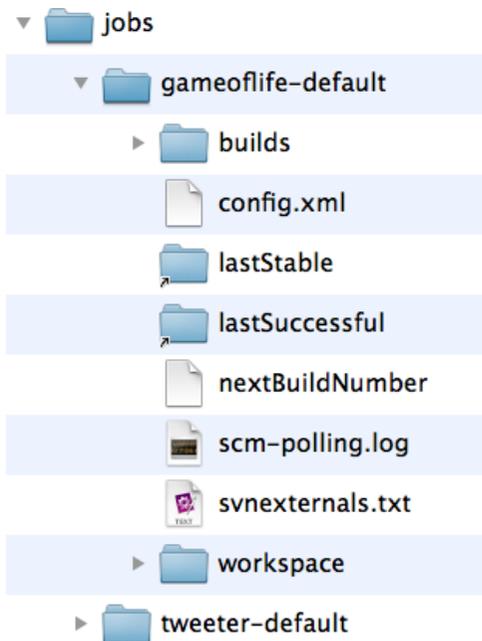


Figure 3.8. The Jenkins jobs directory

This directory contains a subdirectory for each Jenkins build job being managed by this instance of Jenkins. Each job directory in turn contains two subdirectories: `builds` and `workspace`, along with some other files. In particular, it contains the build job `config.xml` file, which contains, as you might expect, the configuration details for this build job. There are also some other files used internally by Jenkins, that you usually wouldn't touch, such as the `nextBuildNumber` file (which contains the number that will be assigned to the next build in this build job), as well as symbolic links to the most recent successful build and the last stable one. A successful build is one that does not have any compilation errors. A stable build is a successful build that has passed whatever quality criteria you may have configured, such as unit tests, code coverage and so forth.

Both the `build` and the `workspace` directories are important. The `workspace` directory is where Jenkins builds your project: it contains the source code Jenkins checks out, plus any files generated by the build itself. This workspace is reused for each successive build—there is only ever one `workspace` directory per project, and the disk space it requires tends to be relatively stable.

The `builds` directory contains a history of the builds executed for this job. You rarely need to intervene directly in these directories, but it can be useful to know what they contain. You can see a real example of the `builds` directory in Figure 3.9, “The builds directory”, where three builds have been performed. Jenkins stores build history and artifacts for each build it performs in a directory labeled with a timestamp (“2010-03-12_20-42-05” and so forth in Figure 3.9, “The builds directory”). It also contains symbolic links with the actual build numbers that point to the build history directories.

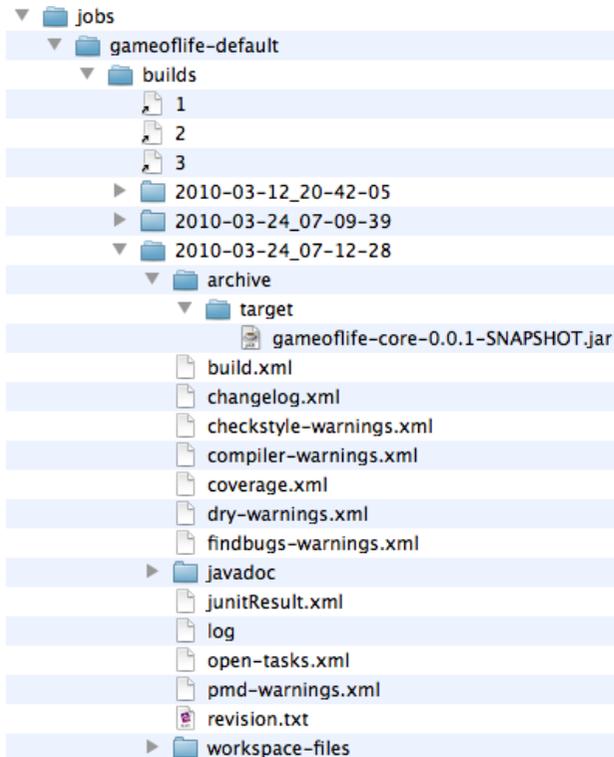


Figure 3.9. The builds directory

Each build directory contains information such as the build result log file, the Subversion revision number used for this build (if you are using Subversion), the changes that triggered this build, and any other data or metrics that you have asked Jenkins to keep track of. For example, if your build job keeps track of unit test results or test coverage metrics, this data will be stored here for each build. The build directory also contains any artifacts you are storing—binary artifacts, but also other generated files such as javadoc or code coverage metrics. Some types of build jobs, such as the Jenkins Maven build jobs, will also archive binary artifacts by default.

The size of the `build` directory will naturally grow over time, as the build history cumulates. You will probably want to take this into account when designing your build server directory structure, especially if your build server is running in a Unix-style environment with multiple disk partitions. A lot of this data takes the form of text or XML files, which does not consume a large amount of extra space for each build. However, if your build archives some of your build artifacts, such as JAR or WAR files, they too will be stored here. The size of these artifacts should be factored into your disk space requirements. We will see later on how to limit the number of builds stored for a particular build job if space is an issue. Limiting the number of build jobs that Jenkins stores is always a trade-off between disk space and keeping useful build statistics, as Jenkins does rely on this build history for its powerful reporting features.

Jenkins uses the files in this directory extensively to display build history and metrics data, so you should be particularly careful not to delete any of the build history directories without knowing exactly what you are doing.

3.14. Backing Up Your Jenkins Data

It is important to ensure that your Jenkins data is regularly backed up. This applies in particular to the Jenkins home directory, which contains your server configuration details as well as your build artifacts and build histories. This directory should be backed up frequently and automatically. The Jenkins executable itself is less critical, as it can easily be reinstalled without affecting your build environment.

3.15. Upgrading Your Jenkins Installation

Upgrading Jenkins is easy—you simply replace your local copy of the `jenkins.war` file and restart Jenkins. However you should make sure there are no builds running when you restart your server. Since your build environment configuration details, plugins, and build history are stored in the Jenkins home directory, upgrading your Jenkins executable will have no impact on your installation. You can always check what version of Jenkins you are currently running by referring to the version number in the bottom right corner of every screen.

If you have installed Jenkins using one of the Linux packages, Jenkins can be upgraded using the same process as the other system packages on the server.

If you are running Jenkins as a stand-alone instance, you can also upgrade your Jenkins installation directly from the web interface, in the Manage Jenkins section. Jenkins will indicate if a more recent version is available, and give you the option to either download it manually or upgrade automatically (see Figure 3.10, “Upgrading Jenkins from the web interface”).



Figure 3.10. Upgrading Jenkins from the web interface

Once Jenkins has downloaded the upgrade, you can also tell it to restart when no jobs are running. This is probably the most convenient way to upgrade Jenkins, although it will not work in all environments. In particular, you need to be running Jenkins as a stand-alone application, and the user running Jenkins needs to have read-write access to the `jenkins.war` file.

If you are running Jenkins on an application server such as Tomcat or JBoss, you might need to do a bit more tidying up when you upgrade your Jenkins instance. Tomcat, for example, places compiled JSP pages in the `CATALINA_BASE/work` directory. When you upgrade your Jenkins version, these files need to be removed to prevent the possibility of any stale pages being served.

Any plugins you have installed will be unaffected by your Jenkins upgrades. However, plugins can also be upgraded, independently of the main Jenkins executable. You upgrade your plugins directly in the Jenkins web application, using the Jenkins Plugin Manager. We discuss plugins in more detail further on in this book.

3.16. Conclusion

In this chapter, we have seen how to install and run Jenkins in different environments, and learned a few basic tips on how to maintain your Jenkins installation once running. Jenkins is easy to install, both as a stand-alone application and as a WAR file deployed to an existing application server. The main things you need to consider when choosing a build server to host Jenkins are CPU, memory, and disk space.

Chapter 4. Configurer votre serveur Jenkins

4.1. Introduction

Avant de commencer à créer vos tâches de build dans Jenkins, vous devez faire un peu de configuration pour vous assurer que votre serveur Jenkins fonctionnera sans problème dans votre environnement spécifique. Jenkins est hautement configurable, et bien que la plupart des options soient fournies avec des valeurs raisonnables par défaut, ou que l'outil soit capable de trouver les bons outils de build dans le PATH ou dans les variables d'environnement, c'est toujours une bonne idée de savoir exactement ce que votre serveur de build fait.

Jenkins est globalement très simple à configurer. Les écrans d'administration sont intuitifs, et l'aide contextuelle (les icônes en forme de point d'interrogation bleu à côté de chaque champ) est détaillée et précise. Dans ce chapitre, nous allons voir comment configurer votre serveur basique en détail. Nous verrons notamment comment configurer Jenkins pour qu'il utilise différentes versions de Java, d'outils de build comme Ant ou Maven, et d'outils de gestion de version comme CVS et Subversion. Plus loin dans le livre, nous regarderons aussi des configurations de serveur plus avancées, comme l'utilisation d'autres systèmes de gestion de version ou d'outils de notifications.

4.2. Le tableau de bord de configuration — L'écran Administrer Jenkins

Dans Jenkins, vous gérez pratiquement tous les aspects de la configuration du système dans l'écran Administrer Jenkins (voir Figure 4.1, “Configurer son installation Jenkins dans l'écran Administrer Jenkins”). Vous pouvez aussi atteindre cet écran directement depuis n'importe où dans l'application en tapant “manage” dans la boîte de recherche Jenkins. Cet écran change en fonction des plugins que vous installez, ne soyez donc pas surpris si vous voyez plus de choses que ce que nous montrons ici.

Manage Jenkins

[Configure System](#)
Configure global settings and paths.

[Reload Configuration from Disk](#)
Discard all the loaded data in memory and reload everything from file system. Useful when you modified config files directly on disk.

[Manage Plugins](#)
Add, remove, disable or enable plugins that can extend the functionality of Jenkins.

[System Information](#)
Displays various environmental information to assist trouble-shooting.

[System Log](#)
System log captures output from `java.util.logging` output related to Jenkins.

[Load Statistics](#)
Check your resource utilization and see if you need more computers for your builds.

[Jenkins CLI](#)
Access/manage Jenkins from your shell, or from your script.

[Script Console](#)
Executes arbitrary script for administration/trouble-shooting/diagnostics.

[Manage Nodes](#)
Add, remove, control and monitor the various nodes that Jenkins runs jobs on.

[About Jenkins](#)
See the version and license information

[Prepare for Shutdown](#)
Stops executing new builds, so that the system can be eventually shut down safely.

Figure 4.1. Configurer son installation Jenkins dans l'écran Administrer Jenkins

Cet écran vous permet de configurer différents aspects de votre serveur Jenkins. Chaque lien sur cette page vous amène à un écran de configuration dédié, où vous pouvez gérer différentes parties du serveur Jenkins. Quelques-unes des options les plus intéressantes sont discutées ici :

Configurer le système

C'est là que vous gérez les chemins vers les différents outils que vous utilisez dans vos builds, comme les JDKs, les versions de Ant et Maven, les options de sécurité, les serveurs d'email, et autres détails de configuration de niveau système. Plusieurs des plugins que vous installerez nécessiteront aussi d'être configurés ici — Jenkins ajoutera les champs dynamiquement à l'installation des plugins.

Recharger la configuration à partir du disque

Comme nous l'avons vu dans le précédent chapitre, Jenkins stocke tous les détails de configuration du système et des tâches de build dans des fichiers XML localisés dans le répertoire de travail de Jenkins (voir Section 3.4, "Le répertoire de travail de Jenkins"). Il stocke aussi tout l'historique des builds dans le même répertoire. Si vous migrez des tâches de build d'une instance Jenkins à une autre, ou archivez de vieilles tâches de build, vous aurez besoin d'ajouter ou d'enlever les différents répertoires de tâches de build du répertoire `builds` de Jenkins. Vous n'avez pas besoin de désactiver Jenkins pour faire cela — vous pouvez simplement utiliser l'option "Recharger la configuration à partir du disque" pour recharger directement la

configuration système de Jenkins et des tâches de build. Ce processus peut être un peu lent s'il y a beaucoup d'historique de build, pendant que Jenkins charge non seulement la configuration des builds mais aussi les données de l'historique.

Gestion des plugins

L'une des meilleures fonctionnalités de Jenkins est son architecture extensible. Il y a un large écosystème de plugins open source tierces disponibles, vous permettant d'ajouter des fonctionnalités à votre serveur de build, du support des différents outils de gestion de sources comme Git, Mercurial ou ClearCase, aux métriques de qualité du code et de couverture de code. Nous regarderons plusieurs des plugins les plus populaires et utiles à travers ce livre. Les plugins peuvent être installés, mis à jour et enlevés via l'écran Gérer les Plugins. Notez qu'enlever des plugins nécessite un certain soin, parce que cela peut parfois affecter la stabilité de votre instance Jenkins — nous verrons cela plus en détails dans Section 13.6, “Migrer les tâches de build”.

Information Système

Cet écran affiche une liste de toutes les propriétés système Java et les variables d'environnement système. Ici, vous pouvez vérifier dans quelle version exacte de Java Jenkins est en train de fonctionner, avec quel utilisateur, etc. Vous pouvez aussi vérifier que Jenkins utilise le bon paramétrage des variables d'environnement. Cet écran sert principalement pour le dépannage. Il vous permet de vous assurer que votre serveur fonctionne avec les propriétés système et les variables d'environnement que vous pensez.

Log système

L'écran Log système est un moyen pratique de voir les fichiers de log Jenkins en temps réel. Encore une fois, ceci sert principalement au dépannage.

Vous pouvez aussi souscrire aux flux RSS pour différents niveaux de messages de logs. Par exemple, en tant qu'administrateur Jenkins, il peut être une bonne idée de souscrire à tous les messages de log de niveau ERROR et WARNING.

Statistiques d'utilisation

Jenkins garde la trace du niveau d'activité de votre serveur en fonction du nombre de builds concurrents et de la longueur de la file d'attente de build (ce qui vous donne une idée de la durée pendant laquelle vos builds doivent attendre avant d'être exécutés). Ces statistiques peuvent vous aider à savoir si vous avez besoin d'ajouter de la capacité additionnelle ou des nœuds supplémentaires à votre infrastructure.

Console de script

Cet écran vous permet d'exécuter des scripts Groovy sur le serveur. C'est utile pour le dépannage avancé : cela requiert en effet une connaissance profonde de l'architecture interne de Jenkins. Cet écran est donc principalement utile pour les développeurs de plugins et consorts.

Gérer les nœuds

Jenkins gère aussi bien les builds parallèles que distribués. Dans cet écran, vous pouvez configurer le nombre de builds que vous voulez. Jenkins les exécute simultanément, et, si vous

utilisez des builds distribués, configure les nœuds de build. Un nœud de build est une autre machine que Jenkins peut utiliser pour exécuter ses builds. Nous regarderons comment configurer les builds distribués en détail dans Chapter 11, Builds distribués.

Préparer à la fermeture

Si vous avez besoin d'éteindre Jenkins, ou le serveur sur lequel il fonctionne, c'est mieux de ne pas le faire lorsqu'un build est en cours. Pour fermer Jenkins proprement, vous pouvez utiliser le lien Préparer à la fermeture, qui empêche le démarrage de tout nouveau build. Finalement, lorsque tous les builds en cours seront terminés, vous pourrez éteindre Jenkins proprement.

Nous reviendrons sur certaines de ces fonctionnalités en détails plus loin dans le livre. Dans les sections suivantes, nous nous concentrerons sur comment configurer les paramètres systèmes les plus importants de Jenkins .

4.3. Configurer l'environnement système

La page d'administration la plus importante de Jenkins est l'écran Configurer le système (Figure 4.2, "Configuration du système dans Jenkins"). Ici, vous paramétrez la plupart des outils fondamentaux dont Jenkins a besoin pour son travail quotidien. L'écran par défaut contient un certain nombre de sections, chacune concernant un domaine différent ou un outil externe. De plus, quand vous installez des plugins, leur configuration système globale est aussi souvent effectuée dans cet écran.

The screenshot shows the Jenkins 'Configure System' page. The top navigation bar includes 'Jenkins', a search box, and a help icon. The left sidebar contains links for 'New Job', 'Manage Jenkins', 'People', and 'Build History'. Below these are sections for 'Build Queue' (showing 'No builds in the queue.') and 'Build Executor Status' (a table with 2 executors in 'Idle' status).

The main configuration area includes:

- Home directory: `/Users/johnsmart/Projects/Demos/hudson-demo/jenkins-data`
- System Message: A large empty text area.
- # of executors:
- Quiet period:
- SCM checkout retry count:
- Global properties: Enable security, Prevent Cross Site Request Forgery exploits, Help make Jenkins better by sending anonymous usage statistics and crash reports to the Jenkins project.
- JDK: Environment variables, (List of JDK installations on this system)
- Ant: (List of Ant installations on this system)
- Maven: (List of Maven installations on this system)
- Maven Project Configuration: Global MAVEN_OPTS

Figure 4.2. Configuration du système dans Jenkins

L'écran Configurer le système vous permet de définir les paramètres globaux pour votre installation Jenkins, et aussi pour vos outils externes nécessaires au processus de build. La première partie de cet écran permet de définir certains paramètres de niveau système.

Le répertoire de travail de Jenkins est indiqué, pour référence. De cette façon, vous pouvez vérifier d'un coup d'œil que vous travaillez avec le répertoire de travail auquel vous vous attendez. Rappelez-vous, vous pouvez changer ce répertoire en positionnant la variable d'environnement `JENKINS_HOME` dans votre environnement (voir Section 3.4, “Le répertoire de travail de Jenkins”).

Le champ Message du système sert à plusieurs choses. Ce texte est affiché en haut de votre page d'accueil Jenkins. Vous pouvez utiliser des balises HTML, c'est donc un moyen simple de personnaliser votre serveur de build en incluant le nom de votre serveur et un petit laïus sur son rôle. Vous pouvez aussi l'utiliser pour afficher des messages pour tous les utilisateurs, pour annoncer par exemple des indisponibilités du système, etc.

La Période d'attente est utile pour les outils de gestion de sources comme CVS qui committent les fichiers un par un, au lieu de les grouper ensemble en une seule transaction atomique. Normalement, Jenkins déclenchera un build dès qu'il détectera un changement dans le dépôt de sources. Toutefois, cela ne convient pas à tous les environnements. Si vous utilisez un outil comme CVS, vous ne devriez pas lancer un build dès que le premier changement arrive, parce que le dépôt sera dans un état inconsistant tant que tous les changements n'auront pas été committés. Vous pouvez utiliser le champ Période d'attente pour éviter des problèmes de ce genre. Si vous mettez une valeur à cet endroit, Jenkins attendra qu'aucun changement n'ait été détecté pendant le nombre spécifié de secondes avant de déclencher le build. Ceci permet de s'assurer que tous les changements ont été committés et que le dépôt est dans un état stable avant de démarrer le build.

Pour les systèmes de gestion de version modernes, comme Subversion, Git ou Mercurial, les commits sont atomiques. Cela signifie que des changements dans plusieurs fichiers sont soumis au dépôt comme unité simple, et le code source sur le dépôt est garanti d'être à tout moment dans un état stable. Toutefois, certaines équipes utilisent encore une approche où un changement logique est livré en plusieurs commits. Dans ce cas, vous pouvez utiliser la Période d'attente pour vous assurer que le build utilise toujours une version stable de code source.

La valeur de Période d'attente spécifiée est la valeur par défaut au niveau système — si nécessaire, vous pouvez redéfinir cette valeur individuellement pour chaque projet.

Vous pouvez aussi gérer les comptes utilisateurs et les droits ici. Par défaut, Jenkins laisse n'importe quel utilisateur faire ce qu'il souhaite. Si vous souhaitez une approche plus restrictive, vous devrez activer la sécurité de Jenkins en sélectionnant le champ Activer la sécurité. Il y a plusieurs façons de gérer cela, nous regarderons cet aspect de Jenkins plus tard (voir Chapter 7, Sécuriser Jenkins).

4.4. Configurer les propriétés globales

La section Propriétés globales (voir Figure 4.3, “Configurer les variables d'environnement dans Jenkins”) vous permet de définir des variables de façon centralisée et de les utiliser dans toutes vos

tâches de build. Vous pouvez ajouter autant de propriétés que vous voulez, et les utiliser dans vos tâches. Jenkins les rendra disponible à l'intérieur de l'environnement de vos tâches de build, vous permettant de les utiliser facilement dans vos scripts Ant ou Maven. A noter que vous ne devez pas mettre de points (“.”) dans les noms de propriétés, parce qu'ils ne seront pas traités correctement. Utilisez donc `ldapserver` ou `ldap_server`, mais pas `ldap.server`.

Global properties

Environment variables

List of key-value pairs

name	<input type="text" value="ldapserver"/>	
value	<input type="text" value="wanaka"/>	

Figure 4.3. Configurer les variables d'environnement dans Jenkins

Il y a deux façons principales d'utiliser ces variables. Premièrement, vous pouvez les utiliser directement dans votre script de build, en utilisant la notation `${key}` ou `$key` (donc `${ldapserver}` ou `$ldapserver` dans l'exemple donné ci-dessus). C'est l'approche la plus simple, mais cela signifie qu'il y a un couplage fort entre la configuration de votre tâche et vos scripts de build.

Si votre script utilise un nom de propriété différent (un contenant des points, par exemple), vous pouvez aussi passer la valeur à votre script de build dans la configuration de votre tâche de build. Dans Figure 4.4, “Utiliser une variable d'environnement configurée” nous passons la valeur de la propriété `ldapserver` définie dans Figure 4.3, “Configurer les variables d'environnement dans Jenkins” à une tâche de build Maven. Utiliser l'option `-D` signifie que cette valeur sera accessible à l'intérieur du script. C'est une approche flexible, parce qu'on peut assigner les propriétés globales définies dans Jenkins à des variables spécifiques à nos scripts de build. Dans Figure 4.4, “Utiliser une variable d'environnement configurée”, par exemple, la propriété `ldapserver` sera disponible à l'intérieur du build Maven via la propriété `${ldap.server}`.

Build

Invoke top-level Maven targets

Goals

Figure 4.4. Utiliser une variable d'environnement configurée

4.5. Configurer vos JDKs

Historiquement, l'une des utilisations les plus communes de Jenkins était de construire des applications Java. Jenkins fournit donc naturellement un support intégré pour Java.

Par défaut, Jenkins construira les applications Java en utilisant la version de Java qu'il trouve dans le PATH, qui est habituellement celle avec laquelle Jenkins fonctionne. Toutefois, pour un serveur de

build de production, vous voudrez probablement plus de contrôle que cela. Par exemple, vous pourriez exécuter votre serveur Jenkins avec Java 6, pour des raisons de performance. Par contre, votre serveur de production pourrait tourner avec Java 5 ou même Java 1.4. Les grosses organisations sont souvent très précautionneuses lorsqu'il s'agit de mettre à jour les versions de Java dans leurs environnements de production, et certains des serveurs d'application poids-lourds du marché sont de notoriété publique lents à être certifiés avec les derniers JDKs.

Dans tous les cas, c'est toujours une sage pratique que de construire votre application en utilisant une version de Java proche de celle utilisée sur votre serveur de production. Bien qu'une application compilée avec Java 1.4 tournera généralement avec Java 6, l'inverse n'est pas toujours vrai. Vous pourriez aussi avoir plusieurs applications qui nécessitent d'être construites avec différentes versions de Java.

Jenkins fournit un bon support pour travailler avec de multiples JVMs. En effet, Jenkins rend très facile la configuration et l'utilisation de plusieurs versions de Java. Comme la plupart des configurations de niveau système, cela se paramètre dans l'écran Configurer le système (voir Figure 4.2, "Configuration du système dans Jenkins"). A cet endroit, vous trouverez une section appelée JDK qui vous permettra de gérer les installations de JDK avec lesquelles vous voulez que Jenkins travaille.

Le moyen le plus simple de déclarer une installation de JDK est de fournir un nom approprié (qui sera ensuite utilisé pour identifier cette installation de Java lors de la configuration de vos builds), et un chemin vers le répertoire de l'installation Java (le même chemin que vous utiliseriez pour la variable `JAVA_HOME`), comme montré sur Figure 4.5, "Configuration des JDKs dans Jenkins". Bien que vous deviez entrer le chemin manuellement, Jenkins vérifiera en temps réel à la fois que le répertoire existe et que ça ressemble à un répertoire JDK valide.

The screenshot shows the 'JDK' configuration page in Jenkins. At the top, there is a section titled 'JDK installations'. Below this, there are two entries for installed JDKs. Each entry consists of a 'name' field, a 'JAVA_HOME' field, an 'Install automatically' checkbox, and a 'Delete JDK' button. The first entry has 'name' set to 'Java 1.6.0' and 'JAVA_HOME' set to '/usr/java/jdk1.6.0_17'. The second entry has 'name' set to 'Java 1.5.0' and 'JAVA_HOME' set to '/usr/java/jdk1.5.0'. Below these entries is an 'Add JDK' button and a link that says 'List of JDK installations on this system'.

Figure 4.5. Configuration des JDKs dans Jenkins

Vous pouvez aussi demander à Jenkins d'installer Java pour vous. Dans ce cas, Jenkins téléchargera l'installateur du JDK et installera une copie sur votre machine (voir Figure 4.6, "Installer un JDK

automatiquement”). La première fois qu'une construction a besoin d'utiliser ce JDK, Jenkins téléchargera et installera la version spécifiée de Java dans le répertoire `tools` du répertoire racine de Jenkins. Si le build est exécuté sur un nouvel agent de build qui n'a pas ce JDK installé, il le téléchargera et l'installera sur celui-ci de la même façon.

C'est aussi un formidable moyen de configurer les agents de construction. Comme nous le verrons plus loin dans le livre, Jenkins peut déléguer des tâches de build à d'autres machines, ou d'autres agents de construction. Un agent de construction (ou “esclave”) est simplement un autre ordinateur que Jenkins peut utiliser pour exécuter certains de ses builds. Si vous utilisez l'option d'installation automatique de Jenkins, vous n'avez pas à installer toutes les versions du JDK dont vous avez besoin sur les machines agent de construction — Jenkins le fera pour vous la première fois que cela lui sera nécessaire.

Par défaut, Jenkins propose de télécharger le JDK à partir du site web d'Oracle. Si votre installation Jenkins est derrière un serveur proxy, vous pourrez avoir besoin de modifier votre configuration de proxy pour permettre à Jenkins d'accéder aux sites externes de téléchargement (voir Section 4.9, “Configurer un Proxy”). Une autre option est de fournir une URL pointant sur votre propre copie interne des binaires du JDK (soit sous la forme d'un ZIP soit sous la forme d'un fichier TAR compressé avec GZip), stocké sur un serveur local à l'intérieur de votre organisation. Cela vous permet de fournir des installations normalisées sur un serveur local et d'accélérer les installations automatiques. Quand vous utilisez cette option, Jenkins vous permet aussi de spécifier un label, ce qui restreindra l'utilisation de cette installation aux noeuds ayant ce label. C'est une technique utile si vous avez besoin d'installer une version spécifique d'un outil sur certaines machines de build. La même approche peut aussi être utilisée pour les autres outils de build (comme Maven et Ant).

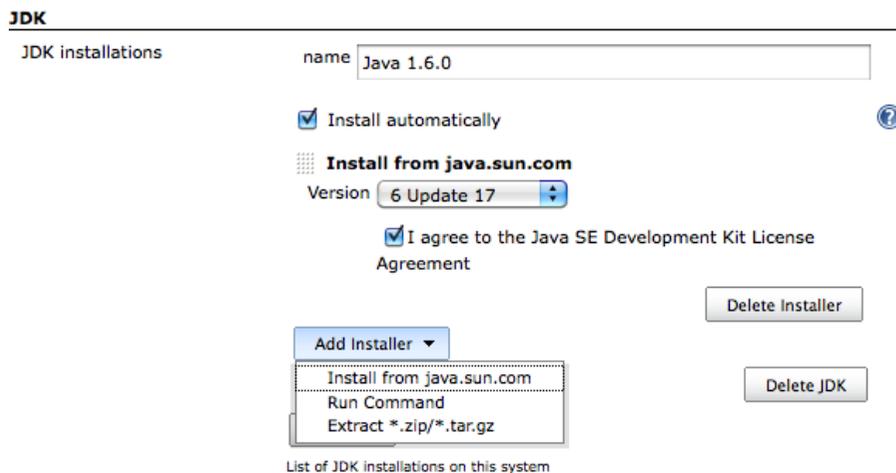


Figure 4.6. Installer un JDK automatiquement

L'installateur automatique ne fonctionnera pas dans tous les environnements (s'il ne parvient pas à trouver ou identifier un système d'exploitation satisfaisant, par exemple, l'installation échouera), mais c'est néanmoins un moyen utile et agréable de configurer de nouveaux serveurs de build ou des agents de construction distribués de manière consistante.

4.6. Configurer vos outils de build

Les outils de build sont l'essence même de tout serveur de build, et Jenkins n'est pas une exception. En standard, Jenkins supporte trois outils de build principaux : Ant, Maven, et le basique shell-script (ou script Batch sous Windows). En utilisant les plugins Jenkins, vous pouvez aussi ajouter le support d'autres outils de build et d'autres langages, comme Gant, Grails, MSBuild, et beaucoup d'autres.

4.6.1. Maven

Maven est un framework de scripting de haut-niveau pour Java qui utilise des notions telles qu'une structure normalisée d'arborescence et des cycles de vie normalisés, "Convention over Configuration", et une gestion déclarative des dépendances pour simplifier le scripting de bas-niveau que l'on trouve souvent dans un script de build Ant typique. Avec Maven, votre projet utilise un cycle de vie normalisé et bien défini — compile, test, package, deploy, etc. Chaque phase de cycle est associée avec un plugin Maven. Les différents plugins Maven utilisent la structure d'arborescence normalisée pour traiter ces tâches avec un minimum d'intervention de votre part. Vous pouvez aussi étendre Maven en redéfinissant les configurations de plugin par défaut ou en invoquant des plugins supplémentaires.

Jenkins fournit un excellent support pour Maven, et comprend parfaitement les structures de projet et les dépendances Maven. Vous pouvez soit demander à Jenkins d'installer automatiquement une version de Maven (comme nous le faisons avec Maven 3 dans l'exemple), ou fournir un chemin vers une installation locale de Maven (voir Figure 4.7, "Configurer Maven dans Jenkins"). Vous pouvez configurer autant de versions de Maven pour vos projets de build que vous le voulez, et utiliser différentes versions de Maven pour différents projets.

The screenshot shows the 'Maven' configuration page in Jenkins. It features a section titled 'Maven installations' with a table of existing installations and an 'Install from Apache' section. The table lists two installations: 'Maven 2.2.1' with MAVEN_HOME set to '/usr/local/maven' and 'Maven 3.0'. The 'Maven 3.0' installation has 'Install automatically' checked. Below the table is an 'Add Maven' button. The 'Install from Apache' section shows a version dropdown set to '3.0-alpha-6' and a 'Delete Installer' button. At the bottom, there is an 'Add Installer' button and another 'Delete Maven' button. A footer note reads 'List of Maven installations on this system'.

name	MAVEN_HOME	Install automatically	Actions
Maven 2.2.1	/usr/local/maven	<input type="checkbox"/>	Delete Maven
Maven 3.0		<input checked="" type="checkbox"/>	Delete Maven

Install from Apache
Version: 3.0-alpha-6
Delete Installer

Add Installer

Add Maven

List of Maven installations on this system

Figure 4.7. Configurer Maven dans Jenkins

Si vous cochez la case à cocher Installer automatiquement, Jenkins téléchargera et installera la versions demandée de Maven pour vous. Vous pouvez soit demander à Jenkins de télécharger Maven directement depuis le site Apache, soit depuis une URL (a priori locale) de votre choix. C'est un excellent choix si vous utilisez des builds distribués, et puisque Maven est multi-plateformes, cela fonctionnera sur n'importe quelle machine. Vous n'avez pas besoin d'installer explicitement Maven sur chaque machine de build — la première fois qu'une machine de build aura besoin d'utiliser Maven, elle en téléchargera une copie et l'installera dans le répertoire `tools` du répertoire racine de Jenkins.

Il est parfois nécessaire de passer des options système à votre processus de construction Maven. Par exemple, il est souvent utile de donner à Maven un peu plus de mémoire pour des tâches lourdes comme la couverture de code ou la génération de site. Maven vous permet de faire cela en positionnant la variable `MAVEN_OPTS`. Dans Jenkins, vous pouvez définir une valeur par défaut pour tout le système, afin de l'utiliser dans tous les projets (voir Figure 4.8, “Configurer la variable système `MVN_OPTS`”). C'est pratique si vous voulez utiliser des options standards de mémoire (par exemple) pour tous vos projets, sans avoir à le configurer dans chaque projet à la main.



Figure 4.8. Configurer la variable système `MVN_OPTS`

4.6.2. Ant

Ant est un langage de script de build pour Java largement utilisé et très connu. C'est un langage de scripting flexible, extensible, relativement bas-niveau utilisé dans un grand nombre de projets opensource. Un script de build Ant (typiquement nommé `build.xml`) est constitué d'un certain nombre de targets. Chaque target effectue une tâche particulière dans le processus de build, comme compiler votre code ou exécuter vos tests unitaires. Il fonctionne en exécutant des tasks, qui portent une partie spécifique de la tâche de build, comme invoquer `javac` pour compiler votre code, ou créer un nouveau répertoire. Les targets ont aussi des dependencies, indiquant l'ordre dans lequel vos tâches de build doivent être exécutées. Par exemple, vous devez compiler votre code avant de lancer vos tests unitaires.

Jenkins fournit en standard un support excellent pour Ant — vous pouvez invoquer les tâches Ant depuis votre tâche de build, en fournissant les propriétés permettant de personnaliser le processus comme cela est nécessaire. Nous regardons comment faire cela en détail plus loin dans le livre.

Si Ant est disponible dans le path système, Jenkins le trouvera. Toutefois, si vous voulez savoir précisément quelle version de Ant vous êtes en train d'utiliser, ou si vous avez besoin de pouvoir utiliser différentes versions de Ant sur différentes tâches de build, vous pouvez configurer autant d'installations de Ant que vous le souhaitez (voir Figure 4.9, “Configurer Ant dans Jenkins”). Fournissez simplement un

nom et un répertoire d'installation pour chaque version de Ant dans la section Ant de l'écran Configurer le système. Vous pourrez ensuite choisir quelle version de Ant vous voulez utiliser pour chaque projet.

Si vous cochez la case à cocher Installer automatiquement, Jenkins téléchargera et installera Ant dans le répertoire `tools` du répertoire racine de Jenkins, exactement comme il le fait pour Maven. Il téléchargera une installation de Ant la première fois qu'une tâche de build aura besoin d'utiliser Ant, soit depuis le site web Apache, soit depuis une URL locale. Encore une fois, ceci est un moyen formidable de normaliser vos serveurs de build et de faciliter l'ajout de nouveaux serveurs de builds distribués à une infrastructure existante.

Ant

Ant installations

name

Install automatically ?

Install from Apache

Version

[List of Ant installations on this system](#)

Figure 4.9. Configurer Ant dans Jenkins

4.6.3. Langage de scripts Shell

Si vous exécutez votre serveur de build sous Unix ou Linux, Jenkins vous permettra d'insérer des scripts shells dans vos tâches de build. C'est pratique pour effectuer des tâches bas-niveau, liées à l'OS que vous ne voulez pas faire avec Ant ou Maven. Dans la section Shell, vous définissez le Shell par défaut qui sera utilisé pour exécuter ces scripts Shell. Par défaut, c'est `/bin/sh`, mais parfois vous pouvez vouloir modifier cela pour utiliser un autre interpréteur de commande comme `bash` ou `Perl`.

Sous Windows, la section Shell ne s'applique pas — vous utilisez le scripting batch Windows à la place. Donc, sur un serveur de build Windows, vous devriez laisser ce champ vierge.

4.7. Configurer vos outils de gestion de version

Jenkins arrive de base pré-installé avec des plugins pour CVS et Subversion. Les autres systèmes de gestion de version sont supportés par des plugins que vous pouvez télécharger depuis l'écran Gérer les plugins.

4.7.1. Configurer Subversion

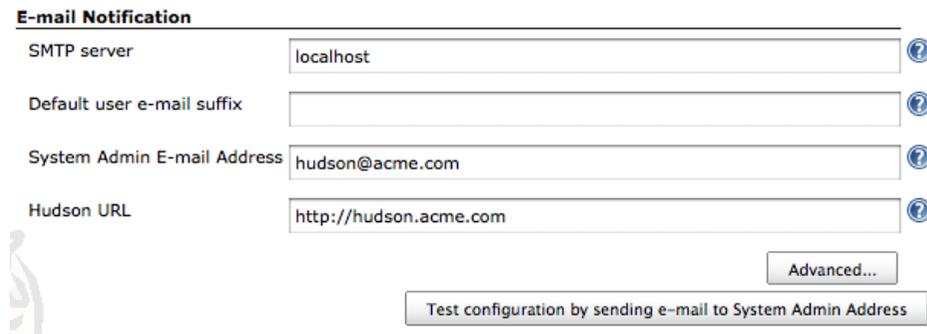
Subversion ne nécessite pas de configuration spéciale, puisque Jenkins utilise des bibliothèques Java natives pour interagir avec des dépôts Subversion. Si vous avez besoin de vous authentifier pour vous connecter à un dépôt, Jenkins vous le demandera quand vous entrerez l'URL Subversion dans la configuration de la tâche de build.

4.7.2. Configurer CVS

CVS nécessite peu voire aucune configuration. Par défaut, Jenkins cherchera des outils comme CVS dans le chemin système, bien que vous puissiez fournir le chemin explicitement s'il ne s'y trouve pas. CVS garde le login et le mot de passe dans un fichier appelé `.cvspass`, qui se trouve habituellement dans votre répertoire utilisateur. Si ce n'est pas le cas, vous pouvez fournir un chemin où Jenkins pourra trouver ce fichier.

4.8. Configurer le serveur de messagerie électronique

La dernière des options de configuration basique que vous devez mettre en place est la configuration du serveur de messagerie électronique. L'email est la technique de notification la plus fondamentale de Jenkins — quand un build échoue, il envoie un email au développeur qui a committé les changements, et optionnellement aux autres membres de l'équipe. Jenkins a donc besoin de connaître votre serveur de messagerie électronique (voir Figure 4.10, “Configurer un serveur d'email dans Jenkins”).



E-mail Notification

SMTP server ?

Default user e-mail suffix ?

System Admin E-mail Address ?

Hudson URL ?

Advanced...

Test configuration by sending e-mail to System Admin Address

Figure 4.10. Configurer un serveur d'email dans Jenkins

L'email de l'administrateur système est l'adresse depuis laquelle les messages de notification sont envoyés. Vous pouvez aussi utiliser ce champ pour tester la configuration email — si vous cliquez sur le bouton Tester la configuration, Jenkins enverra un email de test à cette adresse.

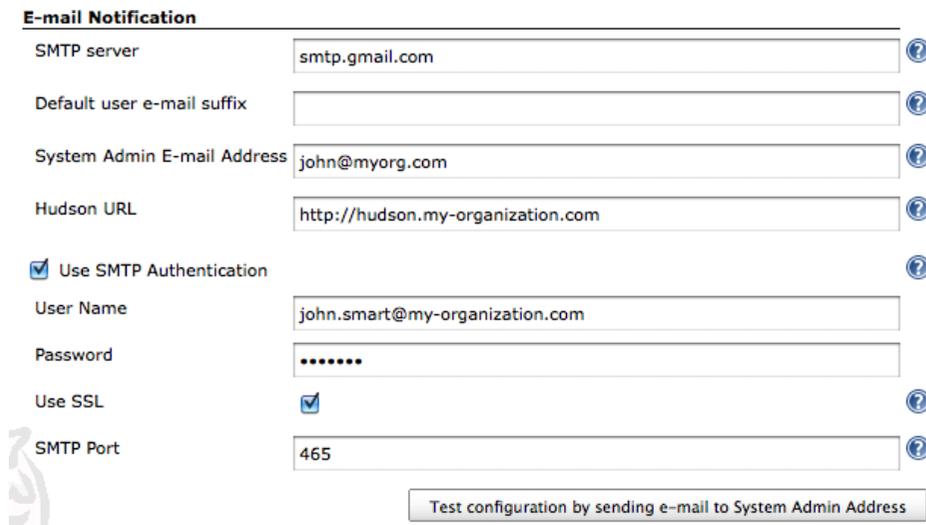
Dans de nombreuses organisations, vous pouvez dériver l'adresse email de l'utilisateur à partir de son login en ajoutant le nom de domaine de l'organisation. Par exemple, chez ACME, l'utilisateur Marcel Tartampion aura un login "mtartampion" et une adresse email "mtartampion@acme.com". Si cela s'étend à votre système de gestion de version, Jenkins peut vous économiser un bon nombre d'efforts de

configuration dans ce domaine. Dans l'exemple précédent, vous pourriez simplement spécifier le suffixe email utilisateur par défaut et Jenkins devinera le reste.

Vous devrez aussi fournir une URL de base correcte pour votre serveur Jenkins (une qui n'utilise pas localhost). Jenkins utilise cette URL dans les notifications email pour que les utilisateurs puissent aller directement de l'email à l'écran d'échec du build sur Jenkins.

Jenkins fournit aussi une configuration email plus sophistiquée, en utilisant des fonctionnalités plus avancées comme l'authentification SMTP et SSL. Si vous êtes dans ce cas, cliquez sur le bouton Avancé pour configurer ces options.

Par exemple, plusieurs organisations utilisent Google Apps pour leurs services de messagerie. Vous pouvez configurer Jenkins pour travailler avec le service Gmail comme montré dans Figure 4.11, “Configurer un serveur d'email pour utiliser un domaine Google Apps”. Tout ce que vous avez besoin de faire dans ce cas est d'utiliser le serveur SMTP Gmail, et de fournir votre nom d'utilisateur Gmail et votre mot de passe dans Authentication SMTP (vous devez aussi utiliser SSL et le port non-standard 465).



E-mail Notification

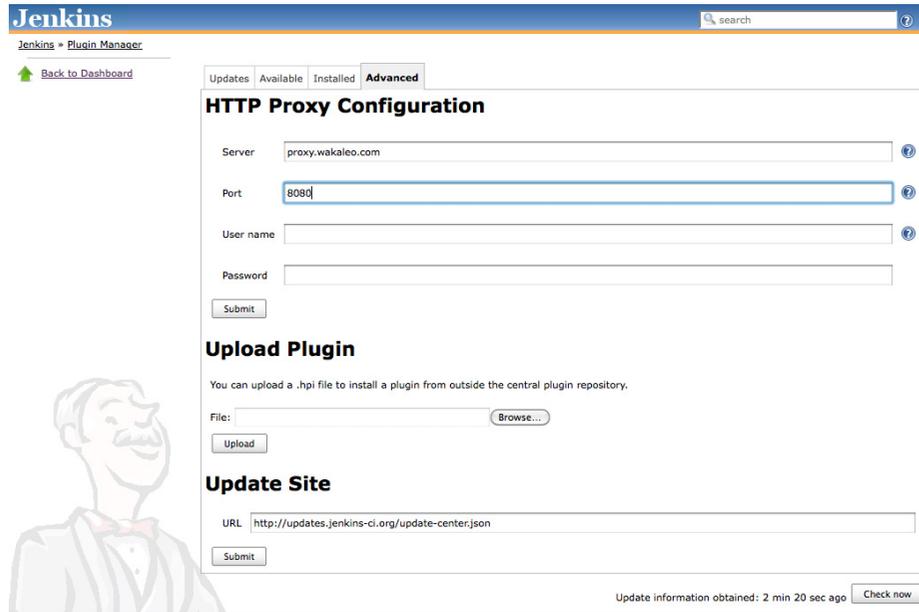
SMTP server	<input type="text" value="smtp.gmail.com"/>	?
Default user e-mail suffix	<input type="text"/>	?
System Admin E-mail Address	<input type="text" value="john@myorg.com"/>	?
Hudson URL	<input type="text" value="http://hudson.my-organization.com"/>	?
<input checked="" type="checkbox"/> Use SMTP Authentication		?
User Name	<input type="text" value="john.smart@my-organization.com"/>	
Password	<input type="password" value="*****"/>	
<input checked="" type="checkbox"/> Use SSL		?
SMTP Port	<input type="text" value="465"/>	?

Figure 4.11. Configurer un serveur d'email pour utiliser un domaine Google Apps

4.9. Configurer un Proxy

Dans la plupart des environnements d'entreprise, votre serveur Jenkins sera situé derrière un pare-feu, et n'aura pas un accès direct à Internet. Jenkins a besoin d'un accès Internet pour télécharger les plugins et les mises à jour, et aussi pour installer les outils tels que le JDK, Ant et Maven depuis des sites distants. Si vous avez besoin de passer par un serveur proxy HTTP pour accéder à Internet, vous pouvez configurer les détails de connexion (le serveur et le port, et si nécessaire le nom utilisateur et le mot de passe) dans l'onglet Avancé de l'écran Gestionnaire de plugins (voir Figure 4.12, “Configurer Jenkins pour utiliser un proxy”).

Si votre proxy utilise le système d'authentification Microsoft NTLM, vous devrez alors fournir un nom de domaine en plus d'un nom d'utilisateur. Vous pouvez placer les deux dans le champ Nom d'utilisateur : entrez simplement le nom de domaine, suivi par anti-slash (\), puis par le nom utilisateur, comme par exemple “MonDomain\Joe Bloggs”.



The screenshot shows the Jenkins Plugin Manager interface. At the top, there's a blue header with the Jenkins logo and a search bar. Below the header, there's a navigation bar with tabs for 'Updates', 'Available', 'Installed', and 'Advanced'. The 'Advanced' tab is selected. The main content area is titled 'HTTP Proxy Configuration' and contains several input fields: 'Server' (with the value 'proxy.wakaleo.com'), 'Port' (with the value '8080'), 'User name', and 'Password'. There are help icons (question marks) next to each field. Below these fields is a 'Submit' button. Underneath the proxy configuration is the 'Upload Plugin' section, which includes a text input for a file name and a 'Browse...' button, followed by an 'Upload' button. Below that is the 'Update Site' section, which has a text input for a URL (with the value 'http://updates.jenkins-ci.org/update-center.json') and a 'Submit' button. At the bottom right of the interface, there's a status bar that says 'Update information obtained: 2 min 20 sec ago' and a 'Check now' button. On the left side of the interface, there is a faint illustration of a man with a mustache, wearing a suit and a bow tie.

Figure 4.12. Configurer Jenkins pour utiliser un proxy

Enfin, si vous mettez en place un accès Proxy sur votre serveur Jenkins, rappelez-vous que tous les autres outils de ce serveur auront besoin de connaître aussi l'existence de ce proxy. En particulier, cela inclut des outils comme Subversion (si vous accédez à un dépôt externe) et Maven (si vous n'utilisez pas un Enterprise Repository Manager).

4.10. Conclusion

Vous n'avez pas besoin d'énormément de configuration pour démarrer avec Jenkins. La configuration requise est plutôt évidente, et est centralisée dans l'écran Configurer le système. Une fois que c'est fait, vous êtes prêt à créer votre première tâche de build Jenkins !

Chapter 5. Configurer vos tâches de Build

5.1. Introduction

Les tâches de build sont les éléments de base d'un serveur d'Intégration Continue.

Une tâche de build est une manière de compiler, tester, emballer, déployer ou d'effectuer des actions sur votre projet. Les tâches de build apparaissent sous plusieurs formes ; vous pouvez compiler et tester unitairement votre application, créer des rapports qualitatifs pour votre code source, générer de la documentation, emballer une application pour une livraison, la déployer en environnement de production, exécuter un test de fumée automatisé ou n'importe quelles autres tâches similaires.

Un projet logiciel aura généralement plusieurs tâches de build attachées. Vous pourriez commencer avec une tâche de build dédiée qui exécute tous les tests unitaires par exemple. Si ceux-ci se terminent avec succès, vous pourriez poursuivre avec une tâche de build exécutant des tests d'intégration plus longs, faire tourner la qualification sur le code ou générer la documentation technique avant d'emballer votre application web pour la déployer sur un serveur de test.

Dans Jenkins, les tâches de build sont simples à configurer. Dans ce chapitre, nous verrons les différents types de tâches de build et la manière de les configurer. Dans les chapitres suivants, nous irons plus loin en regardant comment organiser plusieurs tâches de build, comment configurer un séquençage pour la promotion de builds et comment automatiser la procédure de déploiement. Démarrons pour l'instant avec la manière de configurer une tâche basique de build dans Jenkins.

5.2. Tâches de Build Jenkins

Créer une nouvelle tâche de build dans Jenkins est simple : cliquez simplement sur le lien “Nouveau Job” du menu dans le tableau de bord de Jenkins. Jenkins supporte différents types de tâches de build qui vous sont présentés lorsque vous choisissez de créer un nouveau job (voir Figure 5.1, “Jenkins supporte quatre principaux types de tâches de build”).

Projet free-style

Les tâches de build free-style sont des tâches de build générales apportant une grande flexibilité.

Projet Apache Maven

Le “projet maven2/3” est une tâche de build spécialement adaptée aux projets Apache Maven. Jenkins comprend les fichiers `pom` et la structure des projets Apache Maven et peut utiliser les informations glanées dans le fichier `pom` pour réduire les efforts de configuration nécessaires à la configuration de votre projet.

Contrôler un job externe

La tâche de build “Contrôler un job externe” vous permet de garder un oeil sur des processus non-interactifs externes comme des tâches cron.

Projet multi-configuration

Le “projet multi-configuration” (également référencé comme “projet matrix”) vous permet de faire tourner la même tâche de build avec différentes configurations. Cette puissante fonctionnalité peut être utile pour tester une application dans des environnements différents, avec différentes bases de données ou même sur différentes machines de build. Nous regarderons plus en détails la manière de configurer ces tâches de build multi-configuration plus loin dans ce livre.

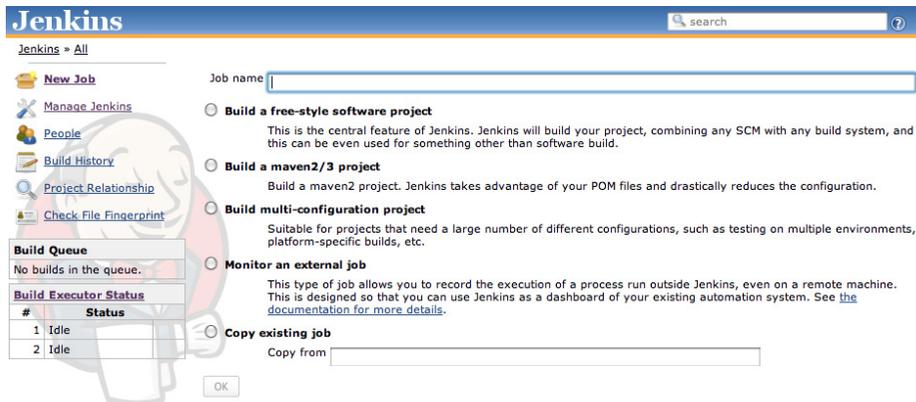


Figure 5.1. Jenkins supporte quatre principaux types de tâches de build

Vous pouvez également copier un job existant ce qui est une très bonne façon de créer un nouveau job avec une configuration très similaire à une tâche de build existante, à l'exception de quelques détails de configuration.

Dans ce chapitre, nous nous concentrerons sur les deux premiers types de tâches de build qui sont les plus couramment utilisées. Les autres seront discutés plus loin. Démarrons avec l'option la plus flexible : la tâche de build free-style.

5.3. Créer une tâche de build free-style

La tâche de build free-style est l'option la plus flexible et la plus configurable et peut être utilisée pour n'importe quel type de projet. Elle est assez rapide à mettre en place et la plupart des options de configuration vues ici sont également disponibles dans les autres types de tâches de build.

5.3.1. Options Générales

La première section que vous voyez lorsque vous créez un job de type free-style contient les informations générales du projet comme son nom unique ou sa description ainsi que d'autres indiquant comment et où la tâche de build doit être exécutée (voir Figure 5.2, “Créer une nouvelle tâche de build”).

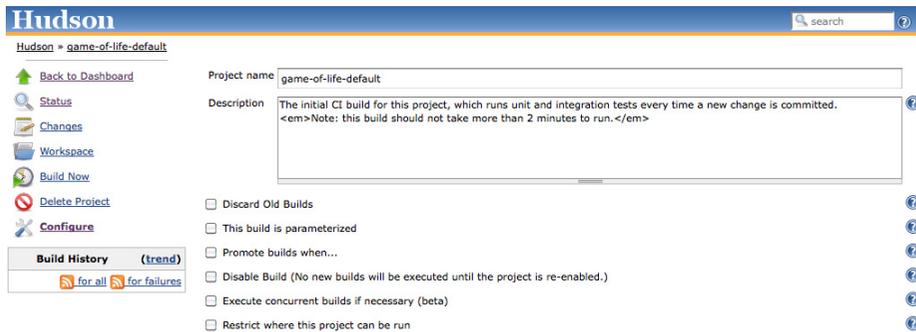


Figure 5.2. Créer une nouvelle tâche de build

Le nom du projet peut être n'importe lequel mais il est bon de noter qu'il sera utilisé comme repertoire du projet et dans les URLs du job. J'évite donc généralement d'utiliser des noms avec des espaces. La description du projet apparaîtra sur la page de démarrage du projet — utilisez la pour donner une idée générale sur le but de la tâche de build et son contexte. Les tags HTML y sont acceptés.

Les autres options sont plus techniques et nous reviendrons sur quelques unes plus en détails dans la suite de ce guide.

Un des aspects importants est la manière dont vous allez gérer l'historique de vos builds. Les tâches de build peuvent consommer beaucoup d'espace disque particulièrement si vous archivez les artefacts de vos builds (les fichiers binaires tels que JARs, WARs, TARs, etc. générés par votre tâche de build). Même sans artefacts, garder un enregistrement pour chaque tâche de build consomme de la mémoire et de l'espace disque supplémentaire et cela n'est peut être pas justifié selon la nature de votre tâche de build. Par exemple, pour un job de qualimétrie générant des rapports sur l'analyse statique et la couverture de votre code dans le temps, vous pourriez vouloir garder une trace de vos builds pendant toute la durée du projet. Cependant, pour une tâche de build qui déploie automatiquement une application sur un serveur de test, garder un historique et les artefacts pour la postérité est peut être beaucoup moins important.

L'option de suppression des anciens builds vous permet de limiter le nombre de builds conservés dans l'historique. Vous pouvez aussi bien demander à Jenkins de ne garder que les builds récents (Jenkins supprimera les builds après un certain nombre de jours) ou ne garder pas plus qu'un nombre déterminé de builds. Si un build en particulier a une valeur sentimentale pour vous, vous pouvez toujours demander à Jenkins de le conserver à tout jamais en utilisant le bouton de conservation sans limite de temps sur la page de détails du build (voir Figure 5.3, “Conserver un build sans limite de temps”). Notez que ce bouton n'apparaîtra que si vous avez demandé à Jenkins de supprimer les anciens builds.



Figure 5.3. Conserver un build sans limite de temps

En plus de cela, Jenkins ne supprimera jamais le dernier build stable qui s'est terminé avec succès, peu importe son ancienneté. Par exemple, si vous limitez Jenkins pour ne conserver que les vingt derniers builds et que votre dernier build qui s'est terminé avec succès s'est exécuté trente builds plus tôt, Jenkins le conservera en plus des vingt derniers builds échoués.

Vous avez aussi la possibilité de désactiver une tâche. Une tâche désactivée ne sera pas exécutée tant que vous ne l'aurez pas réactivée. Utiliser cette option lorsque vous venez de créer votre job est cependant assez rare. D'un autre côté, cette option est souvent utile lorsque vous devez suspendre temporairement une tâche pendant une maintenance ou une grande refactorisation du projet et plus généralement lorsqu'une notification d'échec de la tâche de build ne sera pas utile à l'équipe.

5.3.2. Options avancées du projet

Les options avancées du projet contiennent, comme l'indique le nom de cette section, des options de configuration moins courantes. Vous devrez cliquer sur le bouton Avancé pour les faire apparaître. (voir Figure 5.4, “Pour afficher les options avancées, vous devez cliquer sur le bouton Avancé...”).



Figure 5.4. Pour afficher les options avancées, vous devez cliquer sur le bouton Avancé...

L'option période d'attente dans la configuration du job vous permet simplement d'outrepasser la période d'attente définie globalement dans la configuration du système de Jenkins (voir Section 4.3, “Configurer l'environnement système”). Cette option est principalement utilisée pour les systèmes de gestion de version qui ne supportent pas les commits atomiques, comme par exemple CVS, mais également dans les équipes où les développeurs ont pour habitude de diviser le commit de leur travail en plusieurs petites contributions.

L'option “Empêcher le build quand un projet en amont est en cours de build” est utile lorsque plusieurs projets liés sont affectés par un seul commit mais qu'ils doivent être construits dans un ordre spécifique. Si vous activez cette option, Jenkins attendra que toutes les tâches de build en amont (voir Section 5.5, “Déclencheurs de build”) soient terminées avant de démarrer le build.

Par exemple, lorsque vous livrez une nouvelle version d'un projet Apache Maven multimodule, la mise à jour du numéro de version se fera dans plusieurs, voir l'ensemble, des modules du projet. Supposons que nous ayons ajouté une application web au projet Game of Life que nous avons utilisé dans le Chapter 2, Vos premiers pas avec Jenkins, et que nous l'ayons ajouté sous la forme d'un projet Apache Maven séparé. Lorsque nous livrons une nouvelle version de ce projet, aussi bien le numéro de version du core que celui de l'application web seront mis à jour (voir Figure 5.5, “L'option “Empêcher le build quand un projet en aval est en cours de build” est utile quand un simple commit affecte plusieurs projets dépendants les uns des autres.”). Avant de pouvoir construire l'application web, nous devons construire une nouvelle version du module core de Game of Life. Cependant, si vous avez des tâches de build free-style séparées pour chaque module, les tâches de build de l'application web et du core devraient démarrer simultanément. Le build de l'application web échouera si le build du core n'a pas produit de nouvelle version du module du core, même s'il n'y a pas de test échoué.

Pour éviter ce problème, vous pourriez configurer la tâche de build de l'application web pour démarrer seulement si le build du core s'est terminé avec succès. Cependant, cela signifie que l'application web ne serait jamais construite si des changements étaient effectués uniquement pour celle-ci et non pour le module core. Une meilleure approche est alors d'utiliser l'option “Construire à la suite d'autres projets (projets en amont)”. Dans ce cas, lorsqu'un numéro de version a été mis à jour dans le contrôle de version, Jenkins programmera les deux builds pour leur exécution. Il attendra cependant que le build du module core se soit terminé pour démarrer le build de l'application web.

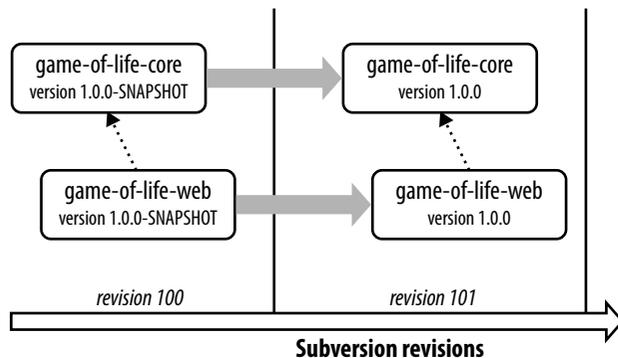


Figure 5.5. L'option “Empêcher le build quand un projet en aval est en cours de build” est utile quand un simple commit affecte plusieurs projets dépendants les uns des autres.

Vous pouvez également surcharger l'espace de travail par défaut utilisé par Jenkins pour y tirer le code source et construire votre projet. De manière générale, Jenkins créera un espace de travail spécifique pour votre projet accessible dans le répertoire de la tâche de build de votre projet (voir Section 3.13, “What’s in the Jenkins Home Directory”). Cela fonctionne dans presque tous les cas. Il y a cependant des cas dans lesquels vous pourriez avoir besoin de forcer Jenkins à utiliser un espace de travail différent grâce à cette option. Un exemple serait le cas où vous auriez besoin de construire plusieurs tâches de build dans un même espace de travail. Vous pouvez définir un répertoire de travail différent en cochant l'option “Utiliser un répertoire de travail spécifique” et en spécifiant le chemin vous même. Ce chemin peut aussi bien être absolu ou relatif au répertoire de base de Jenkins.

Nous verrons d'autres options avancées qui apparaissent dans cette section plus loin dans ce livre.

5.4. Configurer la Gestion du Code Source

Dans son rôle le plus basique, un serveur d'Intégration Continue surveille votre système de gestion de version et vérifie les derniers changements au fur et à mesure. Le serveur compile et test alors la version la plus récente du code. Il peut alternativement récupérer et construire simplement la dernière version de votre code source de manière régulière. Dans tous les cas, une forte intégration avec votre système de gestion de version est essentielle.

De par leur rôle fondamental, les options de configuration du SCM sont identiques au travers de tous les types de tâches de build dans Jenkins. Jenkins supporte CVS et Subversion nativement, embarque un support pour Git et s'intègre avec un grand nombre d'autres systèmes de gestion de version via des plugins. A l'écriture de ce livre, le support par plugin inclus Accurev, Bazaar, BitKeeper, ClearCase, CMVC, Darcs, Dimensions, Git, CA Harvest, Mercurial, Perforce, PVCS, Rational Team Concert, StarTeam, Surround SCM, CM/Synergy, Microsoft Team Foundation Server et même Visual SourceSafe. Dans le reste de cette section, nous verrons comment configurer quelques uns des SCM les plus courants.

5.4.1. Travailler avec Subversion

Subversion est l'un des systèmes de gestion de version parmi les plus utilisés, et Jenkins embarque un support complet de Subversion (voir Figure 5.6, "Jenkins embarque par défaut le support pour Subversion"). Pour utiliser du code source provenant d'un dépôt Subversion, vous devez simplement fournir l'URL correspondante - cela fonctionnera parfaitement avec n'importe lequel des trois protocoles utilisés par Subversion (http, svn ou file). Jenkins vérifiera que l'URL est valide aussitôt que vous l'aurez remplie. Si le dépôt nécessite une authentification, Jenkins vous questionnera alors sur vos identifiants automatiquement et les enregistrera pour tout autre tâche de build qui aura besoin d'accéder à ce même dépôt.

The screenshot shows the 'Source Code Management' configuration page in Jenkins. At the top, there are three radio buttons: 'None', 'CVS', and 'Subversion', with 'Subversion' selected. Below this, there are several input fields and a dropdown menu. The 'Repository URL' field contains 'svn://localhost/gameoflife/trunk'. The 'Local module directory (optional)' field contains a period '.'. A button labeled 'Add more locations...' is positioned below these fields. The 'Check-out Strategy' dropdown menu is open, showing four options: 'Use 'svn update' as much as possible' (selected), 'Use 'svn update' as much as possible, with 'svn revert' before update', 'Emulate clean checkout by first deleting unversioned/ignored files, then 'svn update'', and 'Always check out a fresh copy'. Below the dropdown, the 'Repository browser' section has a 'URL' field containing 'http://localhost:8888/svn/' and a 'Repository Instance' field containing 'gameoflife'. An 'Advanced...' button is located at the bottom right of the configuration area.

Figure 5.6. Jenkins embarque par défaut le support pour Subversion

Par défaut, Jenkins récupérera le contenu du dépôt dans un sous-répertoire de votre espace de travail, dont le nom sera celui du dernier élément de l'URL Subversion. Donc si votre URL Subversion est `svn://localhost/gameoflife/trunk`, Jenkins récupérera le contenu du dépôt dans un répertoire nommé `trunk` dans l'espace de travail de votre tâche de build. Si vous préférez nommer votre répertoire différemment, remplissez le nom que vous souhaitez dans le champ **Local module directory**. Mettez un point (“.”) dans le champ si vous souhaitez mettre le code source directement à la racine de l'espace de travail.

Occasionnellement, vous aurez besoin de récupérer du code source de plusieurs URLs Subversion. Dans ce cas, utilisez le bouton “Add more locations...” pour ajouter autant de dépôts sources que vous en avez besoin.

Une bonne procédure de build ne devrait pas modifier le code source ou laisser de fichiers supplémentaires qui pourrait porter à confusion votre système de gestion de version ou la procédure de build. Les artefacts générés et fichiers temporaires (comme les fichiers de journaux, rapports, données de test ou bases de données fichier) devraient être créés dans un répertoire séparé et créé spécifiquement pour ce besoin (tout comme le répertoire `target` dans les builds Apache Maven), et/ou être configurés pour être ignorés par le dépôt de votre système de gestion de version. Ces fichiers devraient également être supprimés par la procédure de build, une fois que le build n'en a plus besoin. Cela prend également une grande importance dans l'assurance de construire une procédure de build propre et reproductible—pour une version donnée de votre code source, le build devrait se comporter exactement de la même manière, peu importe où et quand il est exécuté. Les changements locaux, et la présence de fichiers temporaires, peuvent potentiellement compromettre cela.

Vous pouvez finement configurer la manière dont Jenkins récupère la dernière version de votre code source en sélectionnant la valeur attendue dans la liste déroulante Check-out Strategy. Si votre projet est correctement configuré, vous pouvez cependant accélérer grandement les choses en choisissant “Use ‘svn update’ as much as possible”. Il s'agit de l'option la plus rapide mais elle laisse les artefacts et fichiers des builds précédents dans votre espace de travail. Pour vous positionner du côté de la sécurité, vous pouvez choisir la seconde option (“Use ‘svn update’ as much as possible, with ‘svn revert’ before update”), qui exécutera systématiquement `svn revert` avant de lancer `svn update`. Cela vous assurera qu'aucun fichier n'aura été modifié localement. Cependant, cela ne supprimera pas les fichiers nouvellement créés pendant la procédure de build. Sinon, vous pouvez demander à Jenkins de supprimer tous les fichiers à ignorer ou non versionnés avant de faire un `svn update` ou encore jouer la carte de la sécurité et récupérer une copie propre et complète à chaque build.

Une autre fonctionnalité de Jenkins particulièrement utile est son intégration avec les navigateurs de code source. Un bon navigateur de code source est un élément important de votre configuration d'Intégration Continue. Cela vous permet de voir d'un coup d'œil les changements qui sont à l'origine du lancement de votre build, ce qui est très utile quand vous avez besoin de localiser un problème lors d'un build échoué (voir Figure 5.7, “Navigateur de code source montrant les changements dans le code qui ont causé le build.”). Jenkins intègre la plupart des principaux navigateurs de source code, y compris des outils open source comme WebSVN ou Sventon, ainsi que ceux commerciaux tels qu'Atlassian FishEye.

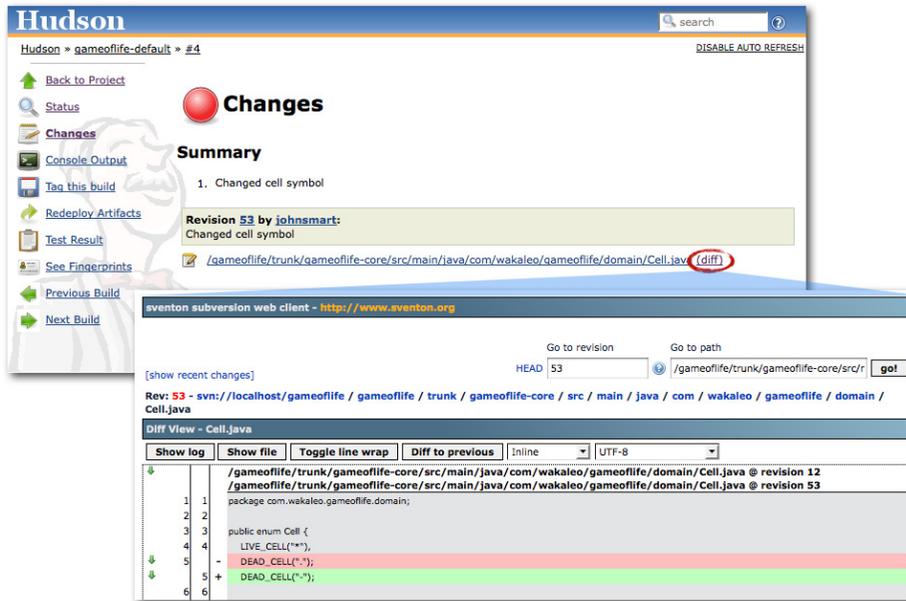


Figure 5.7. Navigateur de code source montrant les changements dans le code qui ont causé le build.

Jenkins vous permet également d'affiner la sélection des changements qui déclencheront un build. Dans la section avancée, vous pouvez utiliser le champ Régions exclues pour dire à Jenkins de ne pas déclencher de build si certains fichiers sont modifiés. Ce champ prend en compte une liste d'expressions régulières qui identifient les fichiers qui ne doivent pas déclencher de build. Par exemple, supposez que vous ne voulez pas que Jenkins démarre un build si seulement des images ont été modifiées. Pour ce faire, vous pouvez utiliser un groupe d'expressions régulières comme celles qui suivent :

```
/trunk/gameoflife/gameoflife-web/src/main/webapp/*.jpg
/trunk/gameoflife/gameoflife-web/src/main/webapp/*.gif
/trunk/gameoflife/gameoflife-web/src/main/webapp/*.png
```

De manière alternative, vous pouvez spécifier uniquement les Régions Inclues, si vous n'êtes intéressé que par les changements d'une partie de votre arbre de code source. Vous pouvez même combiner les champs Régions exclues et Régions Inclues — dans ce cas, un fichier modifié ne déclenchera un build que s'il est inclus dans les Régions Inclues mais pas dans les Régions Exclues.

Vous pouvez également ignorer les changements provenant de certains utilisateurs (Utilisateurs Exclus), ou pour certains messages de commit en particulier (Excluded Commit Messages). Par exemple, si votre projet utilise Maven, vous pourrez être amené à utiliser le plugin Maven Release Plugin pour promouvoir votre application d'une version snapshot vers une version release officielle. Ce plugin poussera automatiquement le numéro de version de votre application depuis sa version snapshot utilisée pendant le développement (comme par exemple 1.0.1-SNAPSHOT) vers une version release (1.0.1), empaquetera, déploiera votre application avec ce numéro de version et le mettra à jour avec le prochain

numéro de version snapshot (par exemple 1.0.2-SNAPSHOT) pour les développements futurs. Pendant cette procédure, Apache Maven s'occupe de multiples étapes comptables comme d'effectuer le commit du nouveau numéro de version, de créer la nouvelle étiquette pour la livraison de votre application et enfin faire l'opération de commit pour le nouveau numéro de version snapshot.

Supposez maintenant que vous avez une tâche de build spécifique pour effectuer une nouvelle livraison utilisant cette procédure. Les différentes opérations de commits générées par le plugin Maven Release Plugin devraient normalement déclencher des tâches de build dans Jenkins. Cependant, comme votre tâche de build de livraison est déjà en train de compiler et tester cette version de votre application, vous n'avez pas besoin que Jenkins le fasse à nouveau dans une autre tâche de build. Pour s'assurer que Jenkins ne déclenche pas un autre build dans ce cas, vous pouvez utiliser le champ Excluded Commit Messages avec la valeur suivante :

```
[maven-release-plugin] prepare release.*
```

Cela vous assurera que Jenkins sautera les changements correspondant à des nouvelles versions de release mais pas ceux correspondant à la prochaine version snapshot.

5.4.2. Travailler avec Git

Contribué par Matthew McCullough

Git¹ est un système de gestion de version distribué qui est le successeur logique de Subversion² et un concurrent à Mercurial³ partageant le même esprit. Le support de Git dans Jenkins est mature et complet. Il y a également plusieurs plugins qui peuvent contribuer au workflow général de Git dans Jenkins. Nous commencerons par nous intéresser au plugin Git qui apporte le support des fonctions principales de Git. Nous aborderons le sujet des plugins supplémentaires brièvement.

5.4.2.1. Installation du plugin

Le plugin Git est disponible dans le Gestionnaire de Plugins Jenkins et est documenté sur sa propre page de wiki⁴. Le plugin suppose que Git (version 1.3.3 ou ultérieure) a été installé sur le serveur de build. Vous devrez donc vous en assurer préalablement. Vous pouvez vous en assurer en exécutant la commande suivante sur votre serveur de build :

```
$ git --version
git version 1.7.1
```

Revenez ensuite à Jenkins et cochez la case correspondante dans le gestionnaire de plugins Jenkins et cliquez sur le bouton d'installation.

¹ <http://git-scm.com/>

² <http://subversion.tigris.org/>

³ <http://mercurial.selenic.com/>

⁴ <https://wiki.jenkins-ci.org/display/JENKINS/Git+Plugin>

5.4.2.1.1. Configuration système du plugin

Après l'installation du plugin Git, une nouvelle section de configuration est disponible dans la page Administrer Jenkins#Configurer le système (voir Figure 5.8, “Configuration système du plugin Git”). Vous devez en particulier fournir le chemin vers l'exécutable de Git. Si Git est déjà installé sur votre système, entrez simplement “git” dans le champ.

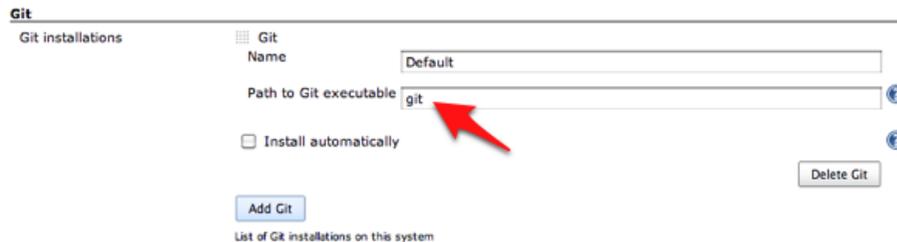


Figure 5.8. Configuration système du plugin Git

5.4.2.1.2. Configuration de la clé SSH

Si le dépôt Git auquel vous accédez utilise SSH sans mot de passe comme moyen d'authentification —par exemple si l'adresse d'accès ressemble à `git@github.com:matthewmccullough/some-repo.git`—vous devrez fournir la partie privée de la clé sous forme de fichier `~/.ssh/id_rsa` où `~` est le répertoire racine du compte utilisateur exécutant Jenkins.

L'empreinte du serveur distant devra additionnellement être ajoutée dans `~/.ssh/known_hosts` pour éviter que Jenkins ne vous invite à accepter l'autorisation vers le serveur Git lors du premier accès alors que la console sera non interactive.

Alternativement, si vous avez la possibilité de vous connecter avec l'utilisateur `jenkins`, accédez par SSH à la machine Jenkins en tant que `jenkins` et faites une tentative manuelle de cloner un dépôt Git distant. Cela testera la configuration de votre clé privée et remplira le fichier `known_hosts` dans le répertoire `~/.ssh`. C'est probablement la solution la plus simple de vous familiariser avec les subtilités de la configuration SSH.

5.4.2.2. Utilisation du plugin

Que cela soit dans un nouveau projet Jenkins ou dans un projet existant, une nouvelle option de Gestion du Code Source pour Git sera affichée. Dès lors, vous pouvez configurer une ou plusieurs adresses de dépôts (voir Figure 5.9, “Remplir une URL de dépôt Git”). Un seul dépôt est utilisé dans la majorité des projets. Ajouter un second dépôt peut être utile dans des cas plus compliqués et vous permet de spécifier des cibles distinctes pour les opérations de `pull` et de `push`.

5.4.2.2.1. Configuration avancée par projet de la gestion du code source

Dans la plupart des cas, l'URL du dépôt Git que vous utilisez devrait être suffisante. Cependant, si vous avez besoin de plus d'options, cliquez sur le bouton Avancé (voir Figure 5.10, "Configuration avancée d'une URL de dépôt Git"). Cela apporte un contrôle plus précis sur le comportement du `pull`.

Le Nom du dépôt est un titre raccourci (ou `remote` en langage Git) pour un dépôt donné auquel vous pouvez vous reporter plus tard dans la configuration de l'action de merge.

La Refspec est un terme⁵ spécifique du langage Git pour contrôler précisément ce qui est récupéré depuis les serveurs distants et sous quel espace de nom c'est stocké localement.

5.4.2.2.2. Branches à construire

Le champ Branch Specifier (Figure 5.11, "Configuration avancée des branches Git à construire") peut être utilisé à l'aide d'un caractère générique ou en spécifiant le nom d'une branche à construire par Jenkins. Si le champ est laissé vide, toutes les branches seront construites. Pour le moment, lorsque vous sauvegardez votre tâche pour la première fois et qu'elle est configurée avec ce champ vide, il est alors rempli avec `**`, ce qui signifie "construire toutes les branches".

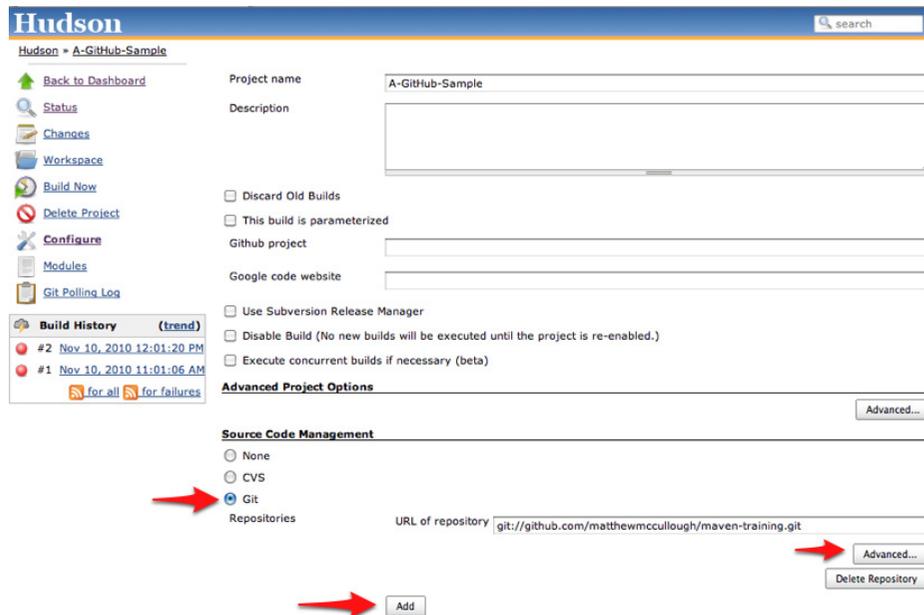


Figure 5.9. Remplir une URL de dépôt Git

⁵ <http://progit.org/book/ch9-5.html>

Source Code Management

None
 CVS
 Git

Repositories

URL of repository

Name of repository (blank to create default)

Refspec (blank to create default)

Branches to build

Branch Specifier (blank for default):

Repository browser

Figure 5.10. Configuration avancée d'une URL de dépôt Git

Source Code Management

None
 CVS
 Git

Repositories

URL of repository

Name of repository (blank to create default)

Refspec (blank to create default)

Branches to build

Branch Specifier (blank for default):

Repository browser

Figure 5.11. Configuration avancée des branches Git à construire

5.4.2.2.3. Régions exclues

Les régions (vues dans Figure 5.12, “Branches et régions”) sont des chemins nommés spécifiquement ou génériques de votre code source qui, même une fois modifiés, ne doivent pas déclencher de build. Il s'agit généralement de fichiers non compilés tels que les archives de locales ou d'images qui n'ont à priori pas d'effet sur les tests unitaires ou d'intégration.

Git

Repositories

URL of repository

Name of repository (blank to create default)

Refspec (blank to create default)

Branches to build

Branch Specifier (blank for default):

Excluded Regions

Excluded Users

Checkout/merge to local branch (optional)

Local subdirectory for repo (optional)

Merge options Merge before build

Prune remote branches before build

Clean after checkout

Figure 5.12. Branches et régions

5.4.2.2.4. Utilisateurs Exclus

Le plugin Git vous permet également d'ignorer certains utilisateurs même s'ils effectuent des changements du code source qui auraient dû normalement déclencher un build.

Cette action n'est pas aussi méchante qu'elle peut paraître : les utilisateurs exclus sont généralement des utilisateurs automatisés ou des développeurs non humains, qui ont des comptes distincts avec des droits de commit sur le gestionnaire de code source. Ces utilisateurs automatisés font généralement de petites opérations comme incrémenter la version d'un fichier `pom.xml` plutôt que de véritables changements dans la logique de votre application. Si vous souhaitez exclure plusieurs utilisateurs, ajoutez les simplement sur des lignes séparées.

5.4.2.2.5. Récupérer/fusionner sur une branche locale

Parfois vous aurez le besoin de récupérer directement le HEAD de manière séparée dans une branche au travers du hash d'un commit. Dans ce cas, spécifiez votre branche locale dans le champ "Checkout/merge to a local branch".

Il est plus simple de l'illustrer par un exemple. Sans spécifier de branche locale, le plugin ferait quelque chose comme cela :

```
git checkout 73434e4a0af0f51c242f5ae8efc51a88383afc8a
```

Autrement, si vous utilisiez une branche nommée `maBranche`, Jenkins ferait la chose suivante :

```
git branch -D maBranche
git checkout -b maBranche 73434e4a0af0f51c242f5ae8efc51a88383afc8a
```

5.4.2.2.6. Dépôt dans un sous-répertoire local

Par défaut, Jenkins clonera le dépôt Git directement dans l'espace de travail de votre tâche de build. Si vous préférez utiliser un répertoire différent, vous pouvez le spécifier ici. Notez que ce répertoire est relatif à l'espace de travail de votre tâche de build.

5.4.2.2.7. Fusionner avant le build

Le cas typique d'utilisation de cette option est le remplissage d'une branche d'intégration proche de la branche `master`. N'oubliez pas que seulement les fusions ne présentant pas de conflit seront effectuées automatiquement. Les fusions plus complexes qui requièrent une intervention manuelle feront échouer le build.

La branche fusionnée qui en résulte ne sera pas poussée automatiquement à moins que l'action de `push` ne soit activée dans les actions post-build.

5.4.2.2.8. Tailler les branches distantes avant le build

Le taillage supprime les copies locales de branches distantes qui proviennent d'un clone précédent mais qui ne sont plus présentes sur le dépôt distant. En résumé, il s'agit du nettoyage du clone local pour qu'il soit parfaitement synchronisé avec son jumeau distant.

5.4.2.2.9. Nettoyer après récupération

Active la purge de tout fichier ou répertoire non versionné, ramenant le votre copie de travail à son état vierge.

5.4.2.2.10. Mise à jour récursive des sous-modules

Si vous utilisez les fonctionnalités de sous-modules de Git dans votre projet, cette option vous assure que chaque sous-module est à jour grâce à un appel explicite à la commande `update`, même si les sous-modules sont imbriqués dans d'autres sous-modules.

5.4.2.2.11. Utiliser l'auteur du commit dans changelog

Jenkins note et affiche l'auteur du changement de code dans une vue synthétique. Git note l'auteur et le commiter du code distinctement, et cette option vous permet de choisir lequel apparaîtra dans le changelog.

5.4.2.2.12. Effacer l'espace de travail

Typiquement Jenkins réutilisera l'espace de travail, le rafraîchissant simplement si nécessaire et si vous avez activé l'option "Clean after checkout", nettoiera les fichiers non versionnés. Cependant, si vous préférez avoir un espace de travail complètement propre, vous pouvez utiliser l'option "Wipe out workspace" pour supprimer et reconstruire l'espace de travail de zéro. Gardez à l'esprit que cela allongera significativement le temps d'initialisation et de construction du projet.

5.4.2.2.13. Choix de la stratégie

Jenkins décide quelle branches il doit construire en se basant sur une certaine stratégie (voir Figure 5.13, "Choix de la stratégie"). Les utilisateurs peuvent influencer cette procédure de recherche de branche. Le choix par défaut est de rechercher tous les HEADs de branche. Si le plugin Gerrit est installé, d'autres options pour construire tous les commits notifiés par Gerrit seront affichées.



Figure 5.13. Choix de la stratégie

5.4.2.2.14. Exécutable Git

Dans les options globales de Jenkins (voir Figure 5.14, "Configuration globale de l'exécutable de Git"), plusieurs exécutables Git peuvent être configurés et utilisés build par build. Cela est rarement utilisé et l'est seulement lorsqu'un clone ou d'autres opérations Git sont sensibles à une version particulière de Git. Git tend à être très flexible quant à ses numéro de version ; les dépôts légèrement anciens peuvent être clonés très facilement avec une nouvelle version de Git et inversement.

Git

Git installations

Git
 Name
 Path to Git executable
 Install automatically

Git
 Name
 Path to Git executable
 Install automatically

List of Git installations on this system

Figure 5.14. Configuration globale de l'exécutable de Git

5.4.2.2.15. Navigateur de dépôt

Tout comme Subversion, Git a plusieurs navigateurs de source qu'il peut utiliser. Les plus couramment utilisés sont Gitorious, Git Web, ou GitHub. Si vous fournissez l'URL correspondante à votre navigateur de dépôt, Jenkins pourra alors afficher un lien direct vers les changements de votre code source qui ont déclenché le build (voir Figure 5.15, "Navigateur de dépôt").

Checkout/merge to local branch (optional)

Local subdirectory for repo (optional)

Merge options Merge before build

Prune remote branches before build

Clean after checkout

Recursively update submodules

Use commit author in changelog

Wipe out workspace

Choosing strategy

Git executable

Repository browser

Figure 5.15. Navigateur de dépôt

5.4.2.3. Déclencheurs de build

Le plugin Git de base offre la possibilité de scruter l'outil de gestion de version régulièrement et de vérifier si de nouveaux changements ont eu lieu depuis la dernière requête. Si des changements sont

présents, un build est alors lancé. Le journal de scrutation (montré dans Figure 5.16, “Journal de scrutation”) est accessible par un lien dans la partie gauche de la page dans la barre de navigation lorsque vous visitez une tâche spécifique. Vous y trouverez les informations sur la dernière fois que le dépôt a été scruté et s’il a renvoyé une liste de changements (voir Figure 5.17, “Résultats de la scrutation de Git”).

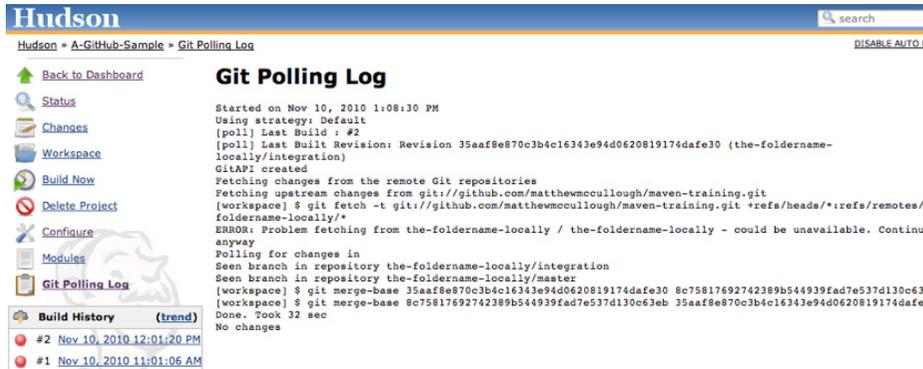


Figure 5.16. Journal de scrutation

La scrutation de Git est disponible dans un format plus orienté développeur qui montre les commentaires de commit ainsi que des hyperliens pointant vers une vue plus détaillée des utilisateurs et des fichiers modifiés.



Figure 5.17. Résultats de la scrutation de Git

Installer le Gerrit Build Trigger ajoute une option Gerrit event qui peut être plus efficace et précise que de simplement scruter le dépôt.

5.4.2.3.1. Déclenchement par Gerrit

Gerrit⁶ est une application web open source qui facilite les revues de code⁷ pour les projets dont les sources sont gérées par Git. Il lit le dépôt Git traditionnel et apporte une comparaison côte-à-côte des changements. Lorsque le code est revu, Gerrit apporte alors un lieu pour commenter et déplacer le patch dans un état ouvert, fusionné ou abandonné.

⁶ <http://code.google.com/p/gerrit/>

⁷ <https://review.source.android.com/#q,status:open,n,z>

Le Gerrit Trigger⁸ est un plugin Jenkins qui peut déclencher un build Jenkins sur du code quand n'importe quelle activité liée à un utilisateur spécifique survient dans le projet d'un utilisateur défini du dépôt Git (voir Figure 5.18, “Déclenchement par Gerrit”). Il s'agit d'une alternative aux options plus régulièrement utilisées comme la construction périodique ou la scrutation de l'outil de gestion de version.

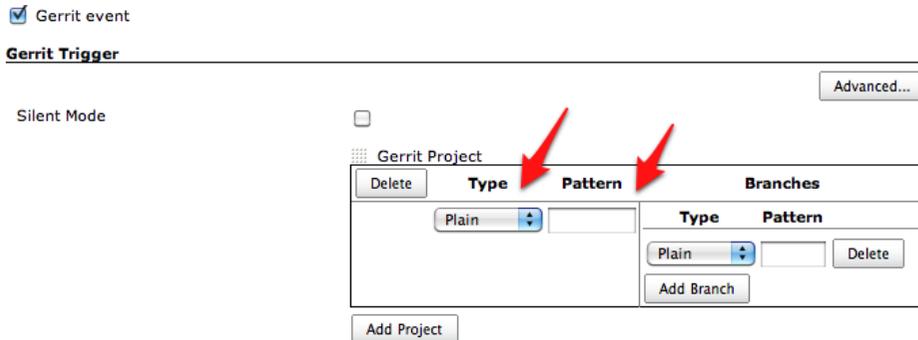


Figure 5.18. Déclenchement par Gerrit

La configuration de ce plugin est minimale et focalisée sur le Type du Projet et son Pattern ainsi que le Type de Branches et leur Pattern. Dans chaque paire, le type peut être Plain, Path ou bien RegExp — descriptif de ce qu'il faut observer — et la valeur (pattern) à évaluer en utilisant le type type comme guide.

5.4.2.4. Actions post-build

Le plugin Git pour Jenkins ajoute des capacités spécifiques à Git au post-traitement des artefacts du build. Plus spécifiquement, le Git Publisher (montré dans Figure 5.19, “Git Publisher”) permet les actions de merge et de push. Cochez la case du Git Publisher pour afficher ses quatres options.

⁸ <http://wiki.hudson-ci.org/display/HUDSON/Gerrit+Trigger>

Git Publisher

Push Only If Build Succeeds

Merge Results

If pre-build merging is configured, push the result back to the origin

Tags

Tag to push

Create new tag

Target remote name

Tags to push to remote repositories

Branches

Branch to push

Target remote name

Branches to push to remote repositories

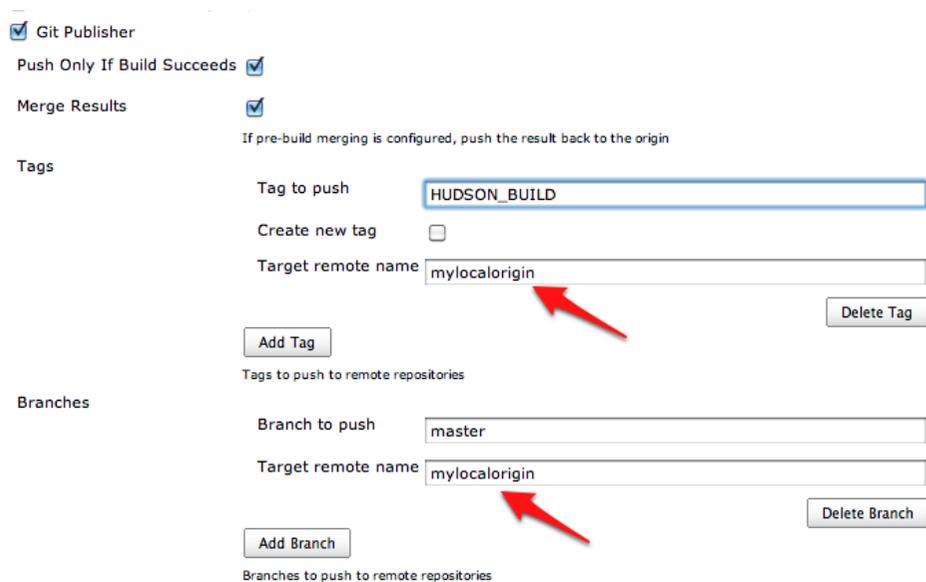


Figure 5.19. Git Publisher

5.4.2.4.1. Pousser seulement si le build est réussi

Si une fusion ou tout autre action entraînant la création d'un commit a été faite pendant le build Jenkins, cette option peut alors être activée pour pousser les changements dans le dépôt distant.

5.4.2.4.2. Fusionner les résultats

Si une fusion au début du build a été configurée, la branche résultante est alors poussée vers son origine (voir Figure 5.20, “Fusionner les résultats”).

Git Publisher

Push Only If Build Succeeds

Merge Results

If pre-build merging is configured, push the result back to the origin

Tags

Tag to push

Create new tag

Target remote name

Tags to push to remote repositories

Branches

Branch to push

Target remote name

Branches to push to remote repositories

Figure 5.20. Fusionner les résultats

5.4.2.4.3. Étiquettes

Lorsque vous poussez des étiquettes, chacune d'elle peut être nommée et choisie d'être créée si elle n'existe pas (ce qui échoue si elle existe déjà). Les variables d'environnement peuvent être utilisées dans le nom des étiquettes. Par exemple vous pouvez utiliser l'ID du process avec `HUDSON_BUILD_$PPID` ou même le numéro de build s'il est fourni par un plugin Jenkins avec `$HUDSON_AUTOTAG_$BUILDNUM`. Les étiquettes peuvent être ciblées sur un dépôt distant spécifique comme `origin` ou `integrationrepo`.

5.4.2.4.4. Branches

Le HEAD courant utilisé dans le build Jenkins d'une application peut être poussé dans d'autres dépôts distants lors d'une étape suivant le build. Vous n'avez à spécifier que la branche de destination et le nom du dépôt distant.

Les noms des dépôts distants sont validés en comparaison à la configuration précédemment faite du plugin. Si le dépôt n'existe pas, un avertissement est affiché.

5.4.2.5. Plugin GitHub

Le plugin GitHub offre deux points d'intégration. Premièrement, il apporte un lien optionnel vers la page de démarrage du projet sur GitHub. Entrez simplement l'URL du projet (sans la partie `tree/master` ou `tree/branch`). Par exemple, `http://github.com/matthewmccullough/git-workshop`.

Ensuite, le plugin GitHub plugin permet d'obtenir un lien par fichier modifié qui sera relié directement au navigateur de dépôt via la section de gestion du code source (voir Figure 5.21, "Navigateur de dépôt GitHub").



Figure 5.21. Navigateur de dépôt GitHub

Avec le choix de `githubweb` comme navigateur de dépôt, tout fichier sur lequel des changements auront été détectés sera lié à la page web de visualisation du source appropriée sur GitHub (Figure 5.22, “Navigateur de dépôt GitHub”).

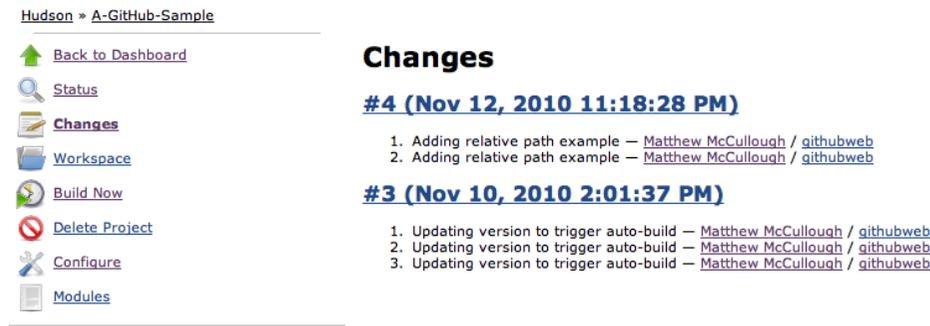


Figure 5.22. Navigateur de dépôt GitHub

5.5. Déclencheurs de build

Une fois que vous avez configuré votre système de gestion de version, vous devez dire à Jenkins quand démarrer un build. Vous pouvez configurer cela dans la section Ce qui déclenche le build.

Dans un build free-style, il y a trois manières basiques de déclencher une tâche de build (voir Figure 5.23, “Il y a de multiples manières de configurer Jenkins pour le démarrage d'une tâche de build”):

- Démarrer une tâche de build une fois qu'une autre s'est terminée
- Lancer des builds à des intervalles périodiques
- Scruter les changements sur le SCM

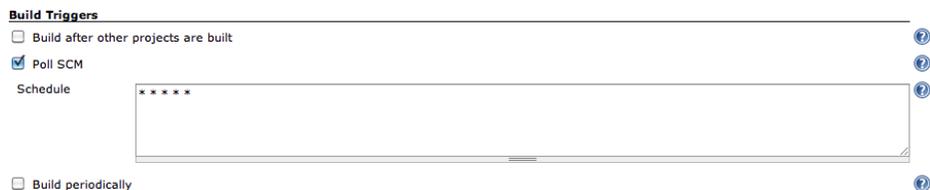


Figure 5.23. Il y a de multiples manières de configurer Jenkins pour le démarrage d'une tâche de build

5.5.1. Déclencher une tâche de build lorsqu'une autre tâche de build se termine

La première option vous permet de configurer un build qui se lancera à chaque fois qu'un autre build se terminera. C'est une manière facile de construire une séquence de build. Par exemple, vous pourriez configurer une tâche de build initiale qui lancera des tests unitaires et des tests d'intégration, suivie d'une autre tâche de build séparée qui lancera de l'analyse statique de code, plus gourmande en CPU. Vous remplissez simplement le nom du job précédent dans ce champ. Si la tâche de build peut être lancée par plusieurs autres tâches de build, listez simplement leurs noms ici en les séparant par des virgules. Dans ce cas là, la tâche de build sera déclenchée à chaque fois que des tâches de build présentes dans la liste se termineront.

Il y a un champ symétrique dans la section des Actions à la suite du build appelée “Construire d'autres projets”. Ce champ sera mis à jour automatiquement dans les tâches de build correspondantes en cohérence avec ce que vous aurez rempli ici. Cependant, au contraire de “Construire à la suite d'autres projets”, vous avez la possibilité de déclencher une autre tâche de build même si le build est instable (voir Figure 5.24, “Déclencher une autre tâche de build même si celle-ci est instable.”). Cela est utile par exemple si vous voulez exécuter une tâche de build sur des indicateurs de qualité du code même s'il y a des tests unitaires qui ont échoué dans la première tâche de build.



Figure 5.24. Déclencher une autre tâche de build même si celle-ci est instable.

5.5.2. Tâches de build périodiques

Une autre stratégie est simplement de déclencher une tâche de build à intervalle régulier. Il est important de faire remarquer qu'il ne s'agit plus d'Intégration Continue ; il s'agit simplement de builds périodiques, chose que vous pourriez tout aussi bien faire, par exemple, avec une tâche cron sous Unix. Aux débuts des builds automatisés, et toujours aujourd'hui dans beaucoup d'entreprises, les builds ne sont pas exécutés en réponse aux changements commités dans le système de contrôle de version mais seulement la nuit (build nocturne journalier). Cependant, pour être efficace, le serveur d'Intégration Continue doit apporter un retour d'information beaucoup plus rapidement qu'une seule fois par jour.

Il y a toutefois quelques cas où les tâches périodiques auront une utilité. Cela inclut les très longues tâches de build, lorsqu'un retour rapide est moins critique. Par exemple, des tests de charge et de performance intensifs qui prennent plusieurs heures à s'exécuter ou des tâches de build Sonar. Sonar est une excellente manière de conserver une vue sur vos indicateurs de qualité du code de vos projets et au fur et à mesure que le temps passe mais il ne permet de conserver qu'un seul lot de données par jour. Il n'est donc pas nécessaire d'exécuter des builds Sonar plus fréquemment que cela.

Pour toutes les tâches périodiques, Jenkins utilise une syntaxe ressemblant à celle de cron. Cette syntaxe est constituée de cinq champs séparés par des espaces blancs au format suivant :

MINUTES HEURES JOURMOIS MOIS JOURSEMAINE

Les valeurs suivantes sont possibles pour chaque champ :

MINUTES

Les minutes dans une heure (0-59)

HEURES

Les heures dans une journée (0-23)

JOURMOIS

Le jour dans un mois (1-31)

MOIS

Le mois (1-12)

JOURSEMAINE

Le jour de la semaine (0-7) où 0 et 7 représentent le dimanche

Il existe également quelques raccourcis :

- “*” représente l'ensemble des valeurs possibles. Par exemple, “* * * * *” signifie “toutes les minutes.”
- Vous pouvez définir des espaces en utilisant la notation “M–N”. Par exemple “1-5” dans le champ JOURSEMAINE signifie “lundi à vendredi.”
- Vous pouvez utiliser la notation slash pour définir des sauts dans un espace. Par exemple, “*/5” dans le champ MINUTES signifie “toutes les cinq minutes.”
- Une liste dont les éléments sont séparés par des virgules indique une liste de valeur prises en compte. Par exemple, “15,45” dans le champ MINUTES signifie “aux minutes 15 et 45 de chaque heure.”
- Vous pouvez aussi utiliser les raccourcis suivants : “@yearly”, “@annually”, “@monthly”, “@weekly”, “@daily”, “@midnight”, et “@hourly”.

Généralement vous n'aurez besoin que d'une seule ligne dans ce champ mais pour des configurations de périodicité plus compliquées, vous aurez peut-être besoin de plusieurs lignes.

5.5.3. Scruter le SCM

Comme nous avons pu le voir, les tâches de build périodiques ne sont généralement pas la meilleure stratégie pour la plupart des tâches de build d'Intégration Continue. La valeur du retour d'information est proportionnelle à la vitesse à laquelle vous recevez ce retour et il n'y a pas d'exception en ce qui concerne l'Intégration Continue. C'est pour cette raison que scruter le SCM est généralement une bien meilleure option.

La scrutation implique l'interrogation à intervalles réguliers du serveur de contrôle de version pour savoir si des changements ont été ajoutés. Si des changements ont été faits dans le code source du projet, Jenkins lance alors un build. Scruter est habituellement une opération peu coûteuse, vous pouvez donc le faire fréquemment pour vous assurer qu'un build sera lancé rapidement après tout commit de code source. Plus vous scruterez fréquemment, plus votre tâche démarrera rapidement et plus précis sera le retour d'information lié aux changements effectués dans le cas où le build échoue.

Dans Jenkins, la scrutation du SCM est très facile à configurer et utilise la même syntaxe cron précédemment présentée.

Naturellement vous aurez l'envie de scruter le SCM le plus souvent possible (par exemple en utilisant “* * * * *” pour chaque minute). Comme Jenkins n'utilise que des requêtes simples et ne lance de build que lorsque le code source a été modifié, cette approche est souvent raisonnable pour de petits projets. Cela montre cependant des limites quand il y a un grand nombre de tâches de build car cela pourrait saturer le serveur SCM et le réseau avec les requêtes dont beaucoup sont inutiles. Dans ce cas, une approche plus précise sera plus appropriée avec un déclenchement de la tâche de build directement par le SCM lorsqu'il reçoit un changement. Cette option est discutée dans Section 5.5.4, “Déclencher des builds à distance”.

Si des changements sont commités très fréquemment et dans un grand nombre de projets, cela peut causer la création d'une longue liste d'attente de tâches de build et ainsi retarder le retour d'information par la suite. Vous pouvez donc partiellement réduire la file d'attente de builds en scrutant moins régulièrement le SCM mais au prix d'un retour d'information moins précis.

Si vous utilisez CVS, scruter n'est peut être pas une bonne option. Lorsque CVS vérifie les nouveaux changements d'un projet, il vérifie chaque fichier un par un ce qui est une procédure lente et fastidieuse. La meilleure solution est de migrer vers un système de contrôle de version plus moderne tel que Git ou Subversion. La deuxième meilleure solution sera de scruter à des intervalles beaucoup plus espacés (toutes les 30 minutes par exemple).

5.5.4. Déclencher des builds à distance

Scruter peut être une stratégie efficace pour des petits projets mais cela ne s'adapte pas très bien à un grand nombre de jobs. Cela utilise inutilement beaucoup de ressources réseau et il y a toujours un court délai entre le commit de nouveau code et le démarrage de la tâche de build. La stratégie sera donc de déléguer à votre système de gestion de version le déclenchement du build dans Jenkins dès qu'un changement est commité.

Il est facile de démarrer une tâche de build Jenkins à distance. Vous devez simplement invoquer une URL de la forme suivante :

```
http://SERVER/jenkins/job/PROJECTNAME/build
```

Par exemple, si mon serveur Jenkins est accessible à `http://myserver:8080/jenkins`, je pourrais démarrer la tâche de build `gameoflife` en invoquant l'URL suivante en utilisant un outil tel que `wget` ou `curl` :

```
§ wget http://myserver:8080/jenkins/job/gameoflife/build
```

L'astuce alors est de faire faire cette invocation directement par votre serveur de contrôle de version dès qu'un changement est commité. Les détails dans la manière de faire sont différents pour chaque système de contrôle de version. Dans Subversion, par exemple, vous devrez écrire un script exécuté automatiquement après la soumission de code pour faire déclencher le build. Votre script pourrait par exemple parcourir l'URL de votre dépôt, en extraire le nom du projet et ensuite effectuer une opération de `wget` sur l'URL correspondante à la tâche de build :

```
JENKINS_SERVER=http://myserver:8080/jenkins
REPOS="$1"
PROJECT=<Expression Régulière de Traitement ici>❶
/usr/bin/wget $JENKINS_SERVER/job/${PROJECT}/build
```

- ❶ Utilisez une expression régulière pour extraire le nom de votre projet de l'URL de votre dépôt Subversion.

Cette approche ne déclenchera cependant qu'un build en particulier et se repose sur une convention de nommage qui veut que la tâche de build par défaut se base sur le nom de votre dépôt Subversion. Vous pouvez opter pour une approche plus flexible avec Subversion en utilisant directement l'API Subversion de Jenkins comme cela est démontré ici :

```
JENKINS_SERVER=http://myserver:8080/jenkins
REPOS="$1"
REV="$2"
UUID=`svnlook uuid $REPOS`
/usr/bin/wget \
  --header "Content-Type:text/plain;charset=UTF-8" \
  --post-data "`svnlook changed --revision $REV $REPOS`" \
  --output-document "-" \
  --timeout=2 \
  $JENKINS_SERVER/subversion/${UUID}/notifyCommit?rev=$REV
```

Cela déclenchera automatiquement n'importe quelle tâche de build Jenkins qui surveille ce dépôt Subversion.

Si vous avez activé la sécurité dans Jenkins, les choses peuvent devenir un peu plus compliquées. Dans le plus simple des cas (où n'importe quel utilisateur peut faire ce qu'il veut), vous n'avez qu'à activer l'option "Déclencher les builds à distance" (voir Figure 5.25, "Déclencher un build via une URL en utilisant un jeton"), et fournir une chaîne de caractères spécifique qui pourra être utilisée dans l'URL :

`http://SERVER/jenkins/job/PROJECTNAME/build?token=DOIT`

Build Triggers

Build after other projects are built

Trigger builds remotely (e.g., from scripts)

Authentication Token

Use the following URL to trigger build remotely: HUDSON_URL/job/game-of-life-default/build?token=TOKEN_NAME or /buildWithParameters?token=TOKEN_NAME
 Optionally append &cause=Cause+Text to provide text that will be included in the recorded build cause.

Build periodically

Poll SCM

Schedule

Figure 5.25. Déclencher un build via une URL en utilisant un jeton

Cela ne fonctionnera pas si les utilisateurs ont besoin d'être authentifiés pour déclencher un build (par exemple si vous utilisez une sécurité par projet ou basée sur une matrice). Dans ce cas, vous aurez besoin de fournir un nom d'utilisateur et un mot de passe, comme montré dans l'exemple suivant :

```
$ wget http://scott:tiger@myserver:8080/jenkins/job/gameoflife/build
```

ou :

```
$ curl -u scott:tiger http://scott:tiger@myserver:8080/jenkins/job/gameoflife/build
```

5.5.5. Construction manuelle de tâches

Une construction ne doit pas forcément être déclenchée automatiquement. Certaines tâches de build doivent seulement être démarrées manuellement, par une intervention humaine. Par exemple, vous pourriez avoir besoin de configurer un déploiement automatique dans un environnement de test de validation (UAT), qui ne devra être démarré qu'à la demande de vos collègues de la validation (QA). Dans ce cas, laissez simplement la section Ce qui déclenche le build vide.

5.6. Les étapes de builds

Maintenant, Jenkins sait où et à quelle fréquence obtenir le code source du projet. La prochaine chose que vous devez expliquer à Jenkins est qu'est ce qu'il doit faire avec le code source. Dans un build Freestyle, vous pouvez faire ceci en définissant des étapes de build. Les étapes de build sont des blocs basiques de construction pour le processus de build Freestyle de Jenkins. C'est ce qui permet de dire à Jenkins exactement comment vous voulez que votre projet soit construit.

Une tâche de build peut avoir une étape, ou plusieurs. Il peut éventuellement n'en avoir aucune. Dans un build Freestyle, vous pouvez ajouter autant d'étapes de build que vous le souhaitez dans la section Build de la configuration de votre projet (voir la figure Figure 5.26, "Ajouter une étape de build à une tâche de build Freestyle"). Dans une installation Jenkins basique, vous serez capable d'ajouter des étapes pour

invoquer Maven et Ant, aussi bien que lancer des commandes shell spécifique à l'OS ou des batchs Windows. Et en installant des plugins additionnels, vous pouvez aussi intégrer d'autres outils, comme Groovy, Gradle, Grailes, Jython, MSBuild, Phing, Python, Rake, et Ruby, juste pour nommer certains des outils les plus connus.

Dans le reste de cette section, nous allons plonger dans quelques-uns des types d'étapes de build les plus communs.

5.6.1. Les étapes de build Maven

Jenkins a un excellent support de Maven, et les étapes de build Maven sont faciles à configurer et très flexibles. Il suffit de choisir « Invoquer les cibles Maven de haut niveau » depuis la liste des étapes de build, choisir une version de Maven à lancer (si vous avez plusieurs versions installées), et entrer les goals Maven que vous souhaitez lancer. Les tâches de build Freestyle de Jenkins fonctionnent biens avec Maven 2 et Maven 3.

Tout comme en ligne de commande, vous pouvez spécifier autant de goals individuels que vous le souhaitez. Vous pouvez aussi fournir des options en ligne de commande. Quelques options utiles de Maven dans un contexte IC sont :

`-B, --batch-mode`

Cette option indique à Maven de ne pas demander d'entrée à l'utilisateur, en utilisant les valeurs par défaut si nécessaire. Si Maven demande n'importe quelle entrée durant un build Jenkins, le build sera bloqué indéfiniment.

`-U, --update-snapshots`

Force Maven à vérifier les mises à jour des dépendances de type release ou snapshot sur le dépôt distant. Cela vous permet d'être sûr que vous êtes en train de construire avec les dernières et les plus grandes dépendances snapshot, et pas uniquement les vieilles copies locales qui ne sont pas forcément synchronisées avec le code source.

`-Dsurefire.useFile=false`

Cette option force Maven à écrire la sortie JUnit dans la console, au lieu de le faire dans des fichiers textes dans le répertoire target comme c'est fait d'habitude. Avec ceci, n'importe quels détails de test en échec seront visibles directement dans la sortie console de la tâche de build. Les fichiers XML dont Jenkins a besoin pour ses rapports de test seront toujours générés.

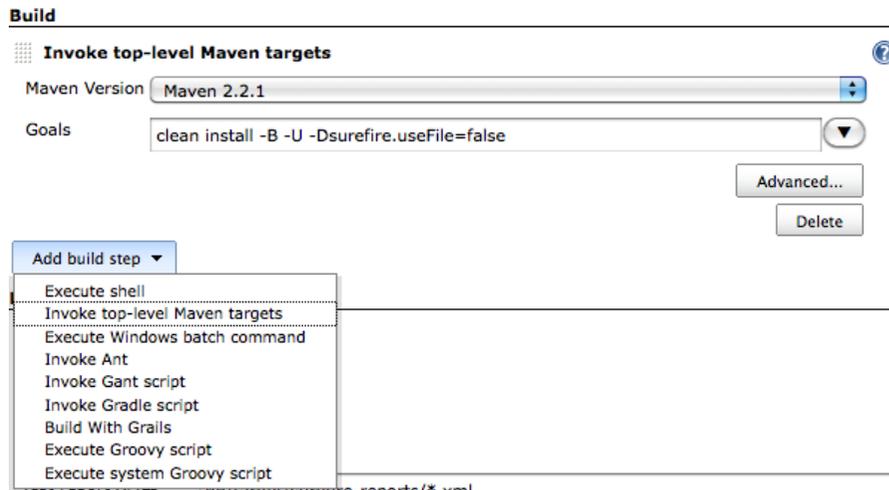


Figure 5.26. Ajouter une étape de build à une tâche de build Freestyle

Les options avancées sont également utiles (cliquez sur le bouton Avancé).

Le champ optionnel **POM** permet de surcharger l'emplacement par défaut du fichier `pom.xml`. C'est l'équivalent de lancer Maven en ligne de commande avec l'option `-f` ou `--file`. C'est utile pour certains projets multi modules où le fichier agrégé `pom.xml` (celui contenant les sections `<modules>`) est situé dans un sous répertoire et non au niveau supérieur.

Le champ Properties vous permet de spécifier des valeurs de propriété qui seront passées au processus de build Maven, en utilisant le format standard de fichier illustré ici :

```
# Selenium test configuration
selenium.host=testserver.acme.com
selenium.port=8080
selenium.browser=firefox
```

Ces propriétés sont passées à Maven en tant qu'options de ligne de commande, comme montré ici :

```
$ mvn verify -Dselenium.host=testserver.acme.com ...
```

Le champ JVM Options vous permet de spécifier des options standards de la machine virtuelle Java pour votre tâche de build. Donc, si votre processus de build est particulièrement consommateur de mémoire, vous pourriez ajouter plus d'espace pour la heap avec l'option `-Xmx` (par exemple, `-Xmx512m` peut spécifier la taille maximum de la heap à 512 Mo).

La dernière option que vous pouvez configurer est un dépôt privé Maven pour cette tâche de build. Normalement, Maven utilisera le dépôt Maven par défaut (usuellement le dossier `.m2/repository` dans le répertoire personnel de l'utilisateur). Parfois, cela peut mener à des interférences entre tâches de build, ou utiliser des versions snapshot inconsistantes d'un build à un autre. Pour être sûr que votre

build est lancé dans des conditions de laboratoire, vous pouvez activer cette option. Votre tâche de build aura son propre dépôt privé, réservé pour son utilisation exclusive. Sur le plan négatif, la première fois que la tâche de build lancera un build, cela prendra du temps pour télécharger tous les artefacts Maven, et les dépôts privés peuvent prendre beaucoup de place. Cependant, c'est la meilleure façon de garantir que votre build est lancé dans un environnement vraiment isolé.

5.6.2. Les étapes de build Ant

Les tâches de build Freestyle fonctionnent également bien avec Ant. Apache Ant⁹ est un outil de scripting de build Java largement utilisé et bien connu. En effet, un nombre important de projets Java sont liés à des scripts de build Ant.

Ant n'est pas seulement utilisé comme un outil de build principal — même si votre projet utilise Maven, vous pouvez recourir à l'appel de scripts Ant pour faire des tâches spécifiques. Il y a des bibliothèques Ant disponibles pour beaucoup d'outils de développement et des tâches bas niveau, comme utiliser SSH, ou travailler avec des serveurs d'application propriétaires.

Dans sa forme la plus basique, configurer une étape de build Ant est très simple, en effet, il vous suffit de fournir la version de Ant que vous souhaitez utiliser et le nom de la target que vous voulez invoquer. Dans la Figure 5.27, “Configurer une étape de build Ant”, par exemple, nous invoquons un script Ant pour démarrer un script de test JMeter.

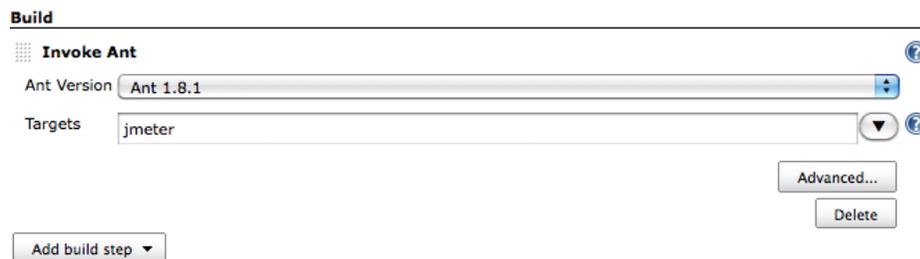


Figure 5.27. Configurer une étape de build Ant

Comme dans une étape de build Maven, le bouton « Avancé... » vous fournit plus d'options détaillées, comme spécifier un script de build différent, ou un script de build dans un répertoire différent (celui par défaut sera `build.xml` dans le répertoire racine). Vous pouvez aussi spécifier des propriétés et des options de la JVM, comme vous pouvez le faire pour Maven.

5.6.3. Exécuter une commande Batch Shell ou Windows

Occasionnellement, vous pouvez avoir besoin d'exécuter une commande directement au niveau du système d'exploitation. Certains processus de build anciens sont liés à des scripts spécifiques à l'OS, par

⁹ <http://ant.apache.org/>

exemple. Dans d'autres cas, vous pourriez avoir besoin d'effectuer un opérateur bas niveau qui serait plus facilement faite avec une commande au niveau OS.

Vous pouvez faire ceci avec Jenkins avec une commande Exécuter un script shell (pour Unix) ou Exécuter une ligne de commande batch Windows (pour Windows). Par exemple, dans la Figure 5.28, "Configurer une étape Exécuter un script Shell", nous avons ajouté une étape pour exécuter la commande Unix `ls`.



Figure 5.28. Configurer une étape Exécuter un script Shell

La sortie de l'étape de build est montrée ici :

```
[workspace] $ /bin/sh -xe /var/folders/.../jenkins2542160238803334344.s
+ ls -al
total 64
drwxr-xr-x  14 johnsmart  staff   476 30 Oct 15:21 .
drwxr-xr-x   9 johnsmart  staff   306 30 Oct 15:21 ..
-rw-r--r--@  1 johnsmart  staff   294 22 Sep 01:40 .checkstyle
-rw-r--r--@  1 johnsmart  staff   651 22 Sep 01:40 .classpath
-rw-r--r--@  1 johnsmart  staff   947 22 Sep 01:40 .project
drwxr-xr-x   5 johnsmart  staff   170 22 Sep 01:40 .settings
-rw-r--r--@  1 johnsmart  staff   437 22 Sep 01:40 .springBeans
drwxr-xr-x   9 johnsmart  staff   306 30 Oct 15:21 .svn
-rw-r--r--@  1 johnsmart  staff  1228 22 Sep 01:40 build.xml
-rw-r--r--@  1 johnsmart  staff    50 22 Sep 01:40 infinitest.filters
-rw-r--r--  1 johnsmart  staff  6112 30 Oct 15:21 pom.xml
drwxr-xr-x   5 johnsmart  staff   170 22 Sep 01:40 src
drwxr-xr-x   3 johnsmart  staff   102 22 Sep 01:40 target
drwxr-xr-x   5 johnsmart  staff   170 22 Sep 01:40 tools
```

Vous pouvez soit exécuter une commande spécifique à l'OS (ex : `ls`), soit stocker un script plus compliqué comme un fichier dans votre gestionnaire de contrôle de version, et exécuter ce script. Si vous exécutez un script, vous n'avez juste qu'à faire référence au nom de votre script relativement par rapport au répertoire de travail.

Les scripts Shell sont exécutés en utilisant l'option `-ex` — les commandes sont affichées dans la console, comme si c'était la sortie. Si n'importe laquelle des commandes exécutées retourne une valeur différente de zéro, le build échouera.

Lorsque Jenkins exécute un script, il spécifie un nombre en tant que variable d'environnement que vous pouvez utiliser à l'intérieur de votre script. Nous discutons de ces variables plus en détail dans la prochaine section.

En réalité, il y a beaucoup de bonnes raisons pour lesquelles vous devriez éviter d'utiliser des scripts de niveau OS dans vos tâches de build si vous pouvez les éviter. En particulier, il rend votre tâche de build au meilleur des cas, spécifique à l'OS, et dans le pire dépendant de la configuration précise de la machine. Une alternative plus portable pour exécuter des scripts spécifiques à l'OS est d'écrire un script équivalent dans un langage de script plus portable, comme Groovy ou Gant.

5.6.4. Utiliser les variables d'environnement Jenkins dans vos builds

Une astuce utile qui peut être utilisée dans pratiquement n'importe quelle étape de build est d'obtenir des informations de la part de Jenkins sur la tâche de build courante. En réalité, lorsque Jenkins démarre une étape de build, il met à disposition les variables d'environnement suivantes dans le script de build :

`BUILD_NUMBER`

Le numéro du build courant, comme "153".

`BUILD_ID`

Un horodatage pour identifier le build courant, sous le format YYYY-MM-DD_hh-mm-ss.

`JOB_NAME`

Le nom du job, comme game-of-life.

`BUILD_TAG`

Un moyen commode d'identifier la tâche de build courante, sous la forme `jenkins-${JOB_NAME}-${BUILD_NUMBER}` (ex : `jenkins-game-of-life-2010-10-30_23-59-59`).

`EXECUTOR_NUMBER`

Un nombre identifiant l'exécuteur ayant démarré cette construction parmi les exécuteurs sur la même machine. C'est le nombre que vous voyez dans « Etat du lanceur de construction », à l'exception que ce nombre commence de 0, pas 1.

`NODE_NAME`

Le nom de l'esclave si ce build est en train d'être lancé sur un esclave, ou "" si le build est en train d'être lancé sur le maître.

`NODE_LABELS`

La liste des libellés associés au nœud sur lequel le build est démarré.

`JAVA_HOME`

Si votre tâche est configurée pour utiliser une version spécifique de JDK, cette variable contient la valeur du `JAVA_HOME` correspondant au JDK spécifié. Lorsque cette variable est fixée, la variable `PATH` est aussi mise à jour pour avoir `$JAVA_HOME/bin`.

`WORKSPACE`

Le chemin absolu du répertoire de travail.

HUDSON_URL

L'URL complète du serveur Jenkins, par exemple `http://ci.acme.com:8080/jenkins/`.

JOB_URL

L'URL complète pour cette tâche de build, par exemple `http://ci.acme.com:8080/jenkins/game-of-life`.

BUILD_URL

L'URL complète de ce build, par exemple `http://ci.acme.com:8080/jenkins/game-of-life/20`.

SVN_REVISION

Pour les projets basés sur Subversion, cette variable contient le numéro de la révision courante.

CVS_BRANCH

Pour les projets basés sur CVS, cette variable contient la branche du module. Si CVS est configuré pour consulter le trunk, cette variable d'environnement ne sera pas spécifiée.

Ces variables sont faciles à utiliser. Dans un script Ant, vous pouvez y accéder avec le tag `<property>` comme montré ici :

```
<target name="printinfo">
  <property environment="env" />
  <echo message="${env.BUILD_TAG}"/>
</target>
```

Dans Maven, vous pouvez accéder aux variables soit de la même manière (en utilisant le prefix « env. »), soit directement en utilisant la variable d'environnement Jenkins. Par exemple, dans le fichier `pom.xml`, l'URL du projet pointera sur la tâche de build Jenkins qui a lancé le build `mvn site` :

```
<project...>
  ...
  <groupId>com.wakaleo.gameoflife</groupId>
  <artifactId>gameoflife-core</artifactId>
  <version>0.0.55-SNAPSHOT</version>
  <name>gameoflife-core</name>
  <url>${JOB_URL}</url>
```

Alternativement, si vous construisez une application web, vous pouvez aussi utiliser `maven-war-plugin` pour insérer le numéro de la tâche de build dans le manifest de l'application web, ex :

```
<project>
  ...
  <build>
    ...
    <plugins>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
```

```

    <configuration>
      <manifest>
        <addDefaultImplementationEntries>>true</addDefaultImplementationEntries>
      </manifest>
      <archive>
        <manifestEntries>
          <Specification-Title>Continuous Integration with Hudson (French Content)</Specification-Title>
          <Specification-Version>0.0.4-SNAPSHOT</Specification-Version>
          <Implementation-Version>${BUILD_TAG}</Implementation-Version>
        </manifestEntries>
      </archive>
    </configuration>
  </plugin>
  ...
</plugins>
</build>
...
</project>

```

Cela produit un fichier `MANIFEST.MF` avec les lignes suivantes :

```

Manifest-Version: 1.0
Archiver-Version: Plexus Archiver
Created-By: Apache Maven
Built-By: johnsmart
Build-Jdk: 1.6.0_22
Jenkins-Build-Number: 63
Jenkins-Project: game-of-life
Jenkins-Version: 1.382
Implementation-Version: jenkins-game-of-life-63
Specification-Title: gameoflife-web
Specification-Version: 0.0.55-SNAPSHOT

```

Dans un script Groovy, elles peuvent être accédées via la méthode `System.getenv()` :

```

def env = System.getenv()
env.each {
    println it
}

```

ou :

```

def env = System.getenv()
println env['BUILD_NUMBER']

```

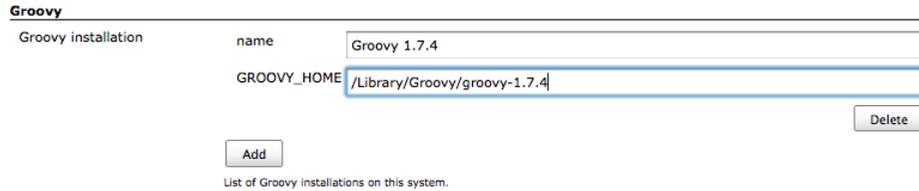
5.6.5. Exécuter des scripts Groovy

Groovy n'est pas seulement un langage dynamique populaire de la JVM, c'est aussi un langage qui convient pour le scripting de bas niveau. Le plugin Groovy¹⁰ de Jenkins vous permet d'exécuter des

¹⁰ <http://wiki.jenkins-ci.org//display/HUDSON/Groovy+Plugin>

commandes Groovy arbitraires, ou invoquer des scripts Groovy, dans le cadre de votre processus de build.

Une fois que vous avez installé le plugin Groovy avec la manière habituelle, vous aurez besoin d’ajouter une référence de votre installation Groovy dans la page de configuration du système (voir la figure Figure 5.29, “Ajouter une installation Groovy à Jenkins”).



Groovy

Groovy installation

name

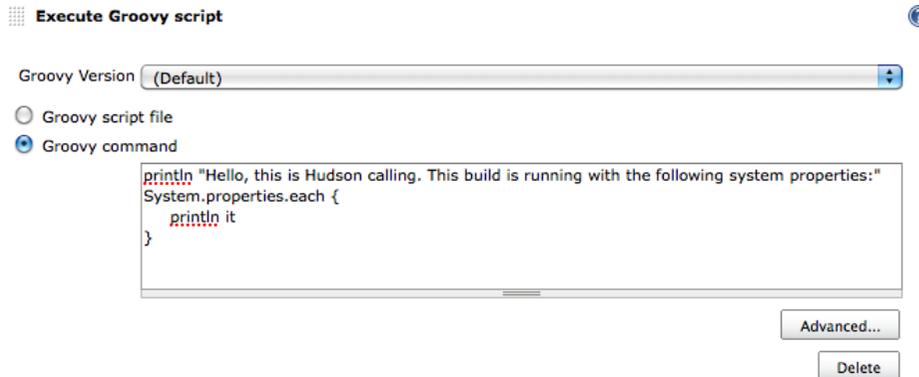
GROOVY_HOME

[List of Groovy installations on this system.](#)

Figure 5.29. Ajouter une installation Groovy à Jenkins

Maintenant, vous pouvez ajouter du script Groovy dans votre tâche de build. Lorsque vous cliquez sur ‘Ajouter une étape de build’, vous verrez deux nouvelles entrées dans le menu déroulant : « Exécuter un script Groovy » et « Exécuter un script Groovy Système ». La première option est généralement celle que vous souhaitez — cela exécutera simplement un script Groovy dans une JVM séparée, comme si vous l’invoquiez depuis la ligne de commande. La deuxième option lance des commandes Groovy depuis la JVM de Jenkins, avec un accès interne complet à Jenkins, et est principalement utilisé pour manipuler les tâches de build de Jenkins ou le processus de build lui-même. C’est un sujet plus avancé dont nous discuterons plus loin dans ce livre.

Une étape de build Groovy peut prendre une forme sur deux. Pour les cas simples, vous pouvez juste ajouter un petit bout de Groovy, comme montré dans la Figure 5.30, “Lancer des commandes Groovy dans le cadre d’une tâche de build”. Pour les cas plus complexes ou compliqués, vous pouvez probablement écrire un script Groovy et le placer sous un système de contrôle de version. Une fois que votre script est en sûreté dans votre SCM, vous pouvez le démarrer en sélectionnant l’option « Fichier de script Groovy » et fournir le chemin de votre script (relatif au répertoire de travail de la tâche de build).



Execute Groovy script

Groovy Version

Groovy script file

Groovy command

```
println "Hello, this is Hudson calling. This build is running with the following system properties:"
System.properties.each {
    println it
}
```

Figure 5.30. Lancer des commandes Groovy dans le cadre d’une tâche de build

Dans la Figure 5.31, “Lancer des scripts Groovy dans le cadre d’une tâche de build”, vous pouvez voir un exemple légèrement plus compliqué. Ici nous lançons un script Groovy appelé `run-fitness-tests.groovy`, qui peut être trouvé dans le répertoire `scripts`. Ce script prend des suites de test pour être exécutés comme ses paramètres — nous pouvons les mettre dans le champ Paramètres Groovy. Sinon vous pouvez aussi fournir des propriétés en ligne de commande dans le champ Propriétés — c’est simplement un moyen plus pratique d’utiliser l’option `-D` en ligne de commande pour passer des valeurs de propriétés au script Groovy.

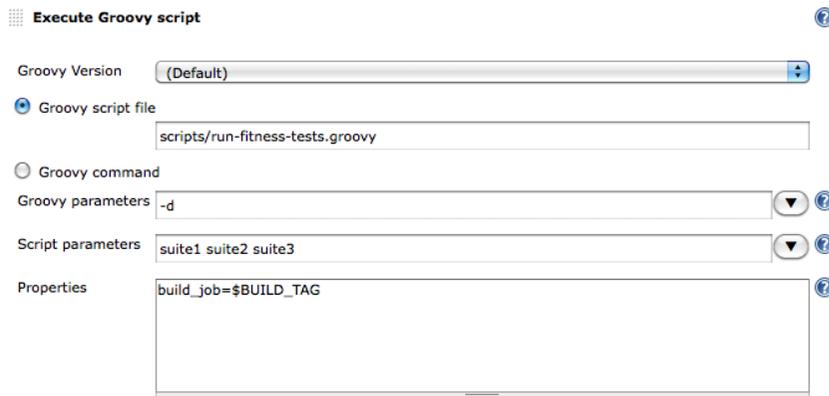


Figure 5.31. Lancer des scripts Groovy dans le cadre d’une tâche de build

5.6.6. Construire des projets dans d’autres langages

Jenkins est un outil flexible, il peut être utilisé avec beaucoup plus de langage que Java et Groovy. Par exemple, Jenkins fonctionne aussi très bien avec Grails, .Net, Ruby, Python et PHP, juste pour en nommer quelques uns. En utilisant d’autres langages, vous aurez généralement besoin d’installer un plugin supportant votre langage favori, qui ajoutera un nouveau type d’étape de build pour ce langage. Nous regarderons d’autres exemples dans la section Section 5.10, “Utiliser Jenkins avec d’autres langages”.

5.7. Les actions à la suite du build

Une fois qu’un build est terminé, il y a toujours des petites choses à voir après. Vous pourriez avoir besoin d’archiver certains artefacts générés, faire des rapports sur les résultats des tests, et notifier des personnes sur les résultats. Dans cette section, nous allons regarder certaines des tâches les plus courantes que vous aurez besoin de configurer après que le build est effectué.

5.7.1. Rapport sur les résultats de tests

L’une des exigences les plus évidentes sur une tâche de build est de faire des rapports sur les résultats des tests. Non seulement s’il y a des échecs aux tests, mais aussi combien de tests sont exécutés, en combien

de temps, et ainsi de suite. Dans le monde Java, JUnit est la librairie de test la plus couramment utilisée, et le format XML JUnit pour les résultats de test est très utilisé et aussi bien compris par les autres outils.

Jenkins fournit un grand support pour les rapports de test. Dans une tâche de build freestyle, vous devez cocher l'option « Publier le rapport des résultats de tests JUnit », et fournir un chemin vers vos fichiers de rapport JUnit (voir Figure 5.32, “Rapport sur les résultats de tests”). Vous pouvez utiliser une expression générique (comme `**/target/surefire-reports/*.xml` dans un projet Maven) pour inclure les rapports JUnit depuis un grand nombre de répertoires différents — Jenkins agrégera les résultats dans un seul rapport.

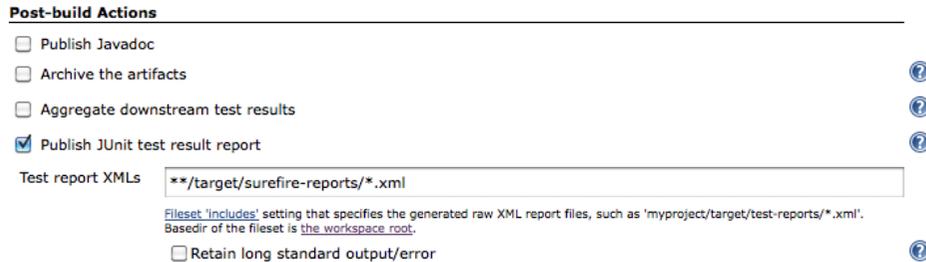


Figure 5.32. Rapport sur les résultats de tests

Nous regarderons les tests automatisés plus en détail dans le chapitre Chapter 6, Tests automatisés.

5.7.2. Archiver les résultats de build

A quelques exceptions près, le but principal d'une tâche de build est généralement de construire quelque chose. Dans Jenkins, nous appelons cette chose un artefact. Un artefact pourrait être un exécutable binaire (un fichier JAR ou WAR pour un projet Java, par exemple), ou certains autres livrables liés, comme de la documentation ou du code source. Une tâche de build peut stocker un ou plusieurs différents artefacts, gardant uniquement la dernière copie ou chaque artefact tous les builds.

Configurer Jenkins pour stocker vos artefacts est simple — cochez la case à cocher « Archiver les artefacts » dans les Actions à la suite de build, et spécifier quels artefacts vous voulez stocker (voir la figure Figure 5.33, “Configurer les artefacts de builds”).

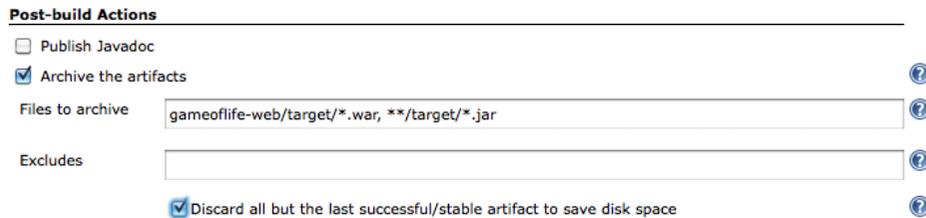


Figure 5.33. Configurer les artefacts de builds

Dans le champ « Fichiers à archiver », vous pouvez spécifier les chemins complets des fichiers que vous souhaitez archiver (relatif au répertoire de travail de la tâche de build), ou, utiliser un caractère générique (ex. : `**/*.jar`, pour tous les fichiers JAR, n'importe où dans le répertoire de travail). L'un des avantages d'utiliser des caractères génériques est qu'il permet de rendre votre build moins dépendant de votre configuration de gestion de version. Par exemple, si vous utilisez Subversion (voir la section Section 5.4, « Configurer la Gestion du Code Source »), Jenkins va parcourir votre projet directement depuis le répertoire de travail, ou depuis un sous-répertoire, en fonction de votre configuration. Si vous utilisez un caractère générique comme `**/target/*.war`, Jenkins trouvera le fichier indépendamment du répertoire où est positionné le projet.

Comme d'habitude, le bouton Avancé donne accès à quelques options supplémentaires. Si vous utilisez un caractère générique pour trouver vos artefacts, vous pourriez avoir besoin d'exclure certains répertoires dans la recherche. Vous pouvez faire ceci en remplissant le champ Exclusions. Vous entrez un modèle de nom de fichier que vous ne souhaitez pas archiver, même s'ils seraient normalement inclus par le champ "Fichiers à archiver".

Les artefacts archivés peuvent prendre beaucoup de place sur le disque, en particulier si les builds sont fréquents. Pour cette raison, vous pouvez vouloir garder uniquement le dernier en succès. Pour faire ceci, il suffit de cocher l'option « Supprime tous les artefacts, à l'exception du dernier artefact stable ou construit avec succès, afin de gagner de l'espace disque ». Jenkins gardera les artefacts des derniers builds stables (s'il y en a). Il gardera aussi les artefacts du dernier build instable construit juste après un build stable (s'il y a), et également du dernier build en échec qui est arrivé.

Les artefacts de build archivés apparaissent sur la page des résultats de build (voir la figure Figure 5.34, « Les artefacts de build sont affichés sur la page de résultat d'un build et la page d'accueil d'un job »). Les artefacts de build les plus récents sont aussi affichés dans la page d'accueil d'un job.



Figure 5.34. Les artefacts de build sont affichés sur la page de résultat d'un build et la page d'accueil d'un job

Vous pouvez aussi utiliser les URLs permanentes pour accéder aux artefacts de build les plus récents. Il s'agit d'une excellente façon de réutiliser les derniers artefacts de vos builds, que ce soit depuis des tâches

de build Jenkins ou depuis des scripts externes, par exemple. Trois URLs sont disponibles : dernier build stable, dernier build en succès et dernier build construit.

Avant que nous regardions les URLs, nous devrions discuter du concept de builds stables et en succès.

Un build est en succès lorsque la compilation n'est pas signalée en erreur.

Un build est considéré stable s'il a été construit avec succès, et qu'aucun éditeur ne l'a signalé comme instable. Par exemple, dépendamment de votre configuration de projet, des échecs de tests unitaires, une couverture de code insuffisante, ou d'autres problèmes de qualité de code, peuvent provoquer la mise à l'état instable d'un build. Donc un build stable est toujours en succès, mais l'inverse n'est pas nécessairement vrai — un build peut être en succès sans être nécessairement stable.

Un build complet est simplement un build qui a fini, peu importe son résultat. A noter que l'étape d'archivage aura lieu quel que soit le résultat de la construction.

Le format des URLs d'artefact est intuitif, et prend la forme suivante :

Dernier build stable

```
<server-url>/job/<build-job>/lastStableBuild/artifact/<path-to-artifact>
```

Dernier build en succès

```
<server-url>/job/<build-job>/lastSuccessfulBuild/artifact/<path-to-artifact>
```

Dernier build complet

```
<server-url>/job/<build-job>/lastCompletedBuild/artifact/<path-to-artifact>
```

C'est mieux illustré par quelques exemples. Supposez que votre serveur Jenkins est démarré sur `http://myserver:8080`, votre job de construction se nomme `game-of-life`, et vous stockez un fichier appelé `gameoflife.war`, qui est dans le répertoire `target` de votre répertoire de travail. Les URLs pour cet artefact seraient les suivantes :

Dernier build stable

```
http://myserver:8080/job/gameoflife/lastStableBuild/artifact/target/gameoflife.war
```

Dernier build en succès

```
http://myserver:8080/job/gameoflife/lastSuccessfulBuild/artifact/target/gameoflife.war
```

Dernier build complet

```
http://myserver:8080/job/gameoflife/lastCompletedBuild/artifact/target/gameoflife.war
```

Les artefacts peuvent ne pas juste être des exécutables binaires. Imaginez, par exemple, que votre processus de build implique de déployer automatiquement tous les builds sur un serveur de test. Pour plus de commodités, vous voulez garder une copie du code source exact associé à chaque fichier WAR déployé. Une manière de faire cela serait de générer le code source associé à un build, et d’archiver à la fois ce fichier le fichier WAR. Nous pouvons faire ceci en générant le fichier JAR contenant le code source de l’application (par exemple, en utilisant le Maven Source Plugin pour un projet Maven), et ensuite inclure celui-ci dans la liste des artefacts à stocker (voir la figure Figure 5.35, “Archiver le code source et un paquet binaire”).

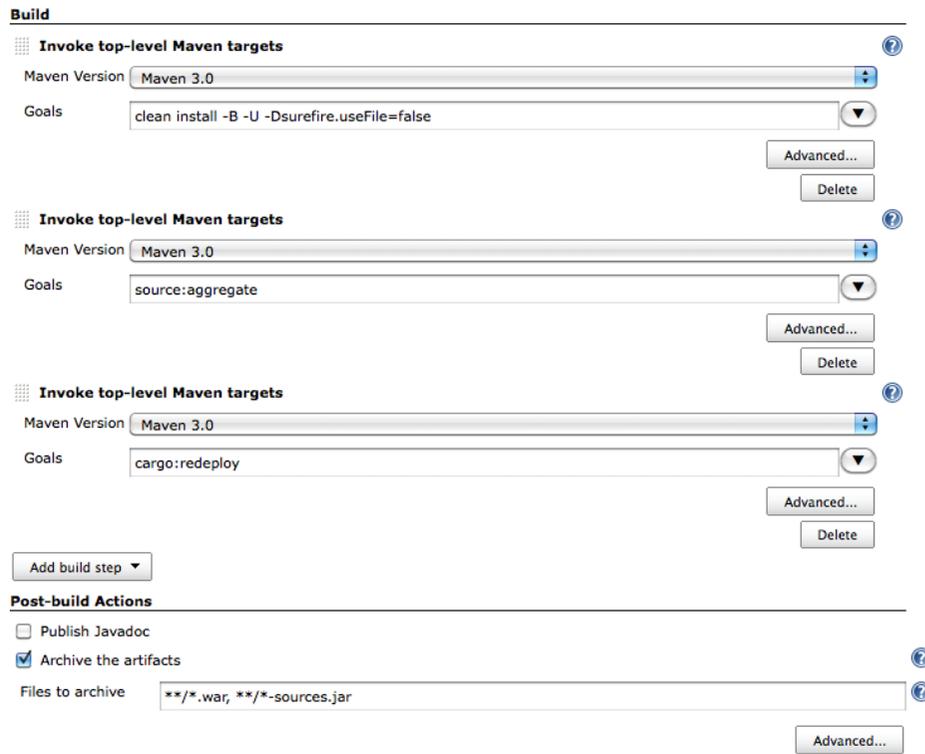


Figure 5.35. Archiver le code source et un paquet binaire

Bien sûr, cet exemple est un peu académique : il serait probablement plus simple de juste utiliser le numéro de révision pour ce build (qui est affiché sur la page de résultat du build) pour retrouver le code source depuis votre système de gestion de version. Mais vous voyez l’idée.

Notez que si vous utiliser un gestionnaire de dépôt d’entreprise comme Nexus ou Artifactory pour stocker vos artefacts binaires, vous n’auriez pas besoin de les garder sur le serveur Jenkins. Vous pourriez simplement préférer déployer automatiquement vos artefacts dans votre gestionnaire de dépôt d’entreprise dans le cadre de la construction de votre job, et de les y retrouver lorsque c’est nécessaire.

5.7.3. Notifications

Le but d'un serveur IC est de permettre d'informer les gens lorsqu'un build est rompu. Dans Jenkins, cela se passe dans la rubrique Notification.

Jenkins fournit le support des notifications par email. Vous pouvez l'activer en cochant la case à cocher « Notifier par email » dans les actions à la suite du build (voir Figure 5.36, "Notification par email"). Ensuite, entrez les adresses emails des membres de l'équipe qui doivent savoir lorsqu'un build est rompu. Lorsqu'un build est rompu, Jenkins enverra un message amical aux utilisateurs dans la liste contenant un lien vers les builds rompus.

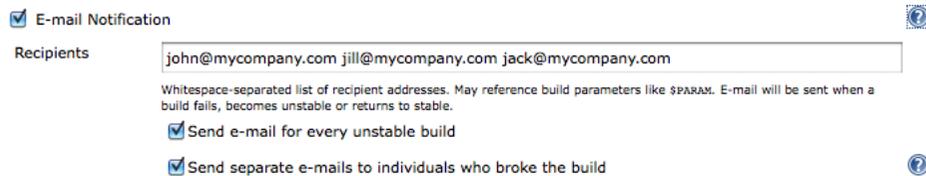


Figure 5.36. Notification par email

Vous pouvez aussi opter pour envoyer un email séparé à l'utilisateur dont le commit a (probablement) rompu le build. Pour que cela fonctionne, vous devez avoir activé la sécurité dans votre serveur Jenkins (voir la figure Chapter 7, Sécuriser Jenkins).

Normalement, Jenkins enverra une notification par email à chaque fois qu'un build échouera (par exemple, à cause d'une erreur de compilation). Il enverra aussi une notification lorsque le build devient instable pour la première fois (par exemple, s'il y a des tests en échecs). A moins que vous le configurer pour faire ça, Jenkins n'enverra pas d'emails pour chaque build instable, mais uniquement pour le premier.

Finalement, Jenkins enverra un message lorsque un build précédemment en échec ou instable réussit, pour permettre d'informer tout le monde que le problème a été résolu.

5.7.4. Construire d'autres projets

Vous pouvez aussi démarrer d'autres constructions de job dans les actions à la suite du build, en utilisant l'option « Construire d'autres projets (projets en aval) ». Ceci est utile si vous souhaitez organiser votre processus de build en plusieurs, plus petites étapes, à la place d'une seule longue construction de job. Il suffit de lister les projets que vous souhaitez démarrer après celui-ci. Normalement, ces projets seront déclenchés uniquement si le build est stable, mais vous pouvez optionnellement déclencher d'autre construction de job même si le build courant est instable. Cela peut être utile, par exemple, si vous voulez démarrer une construction de job effectuant des rapports sur des mesures de qualité de code après une construction de job principal du projet, même s'il y a des tests en échecs dans le build principal.

5.8. Démarrer votre nouvelle tâche de build

Maintenant, tout ce que vous devez faire est sauver votre nouveau job de construction. Vous pouvez alors déclencher le premier build manuellement, ou juste en attendant que celui-ci se déclenche de lui-même. Une fois que le build est terminé, vous pouvez cliquer sur le numéro du build pour voir les résultats de votre travail.

5.9. Travailler avec des tâches de build Maven

Dans cette section, nous allons avoir un aperçu de l'autre type de tâche de build communément utilisé : les tâches de build Maven 2/3.

Les tâches de build Maven sont spécifiquement adaptées pour les builds Maven 2 et Maven 3. Créer une tâche de build Maven nécessite considérablement moins de travail que son équivalent en tâche de build Freestyle. Les tâches de build Maven supportent les fonctionnalités avancées liées à Maven comme les builds incrémentaux sur les projets multi-modules et les builds déclenchés par des changements sur des dépendances en snapshot, et permettent une configuration et des rapports plus simples.

Pendant, il y a un hic : les tâches de build Maven 2/3 sont moins flexibles que les tâches de build Freestyle, et ne supportent pas des étapes de build multiples dans la même tâche de build. Certains utilisateurs ont aussi signalés que les gros projets Maven tendent à être plus lents et utilisent plus de mémoire lorsqu'ils sont configurés avec des tâches de build Maven au lieu de leur équivalent en Freestyle.

Dans cette section, nous allons étudier pour savoir comment configurer des builds Maven 2/3, quand les utiliser, ainsi que leurs avantages et inconvénients.

Pour créer une nouvelle tâche de build, il suffit de choisir l'option « Construire un projet Maven 2/3 » dans la page Nouveau Job (voir Figure 5.37, “Créer une nouvelle tâche de build Maven”).

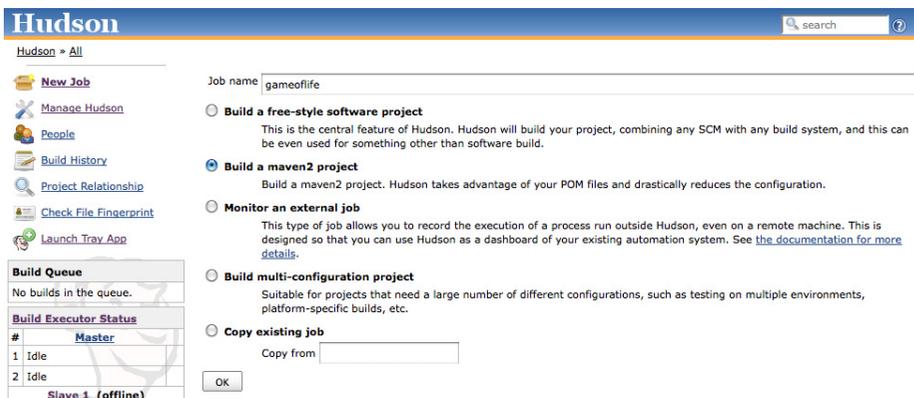


Figure 5.37. Créer une nouvelle tâche de build Maven

5.9.1. Construire dès lors qu'une dépendance SNAPSHOT est construite

A première vue, l'écran de configuration d'une tâche de build Maven 2/3 est très similaire à celle que nous avons vu pour les tâches de build dans la section précédente. La première différence que vous pouvez noter est dans la section Déclencheurs de build. Dans cette section, une option supplémentaire est disponible « Lance un build à chaque fois qu'une dépendance SNAPSHOT est construite ». Si vous sélectionnez cette option, Jenkins examinera votre fichier `pom.xml` (ou les fichiers) pour regarder si aucunes dépendances SNAPSHOT sont en train d'être construites par d'autres tâches de build. Si n'importe laquelle des tâches de build met à jour une dépendance SNAPSHOT que votre projet utilise, Jenkins construira aussi votre projet.

Typiquement dans Maven, les dépendances SNAPSHOT sont utilisées pour partager la toute dernière version d'une librairie avec d'autres projets de la même équipe. Comme elles sont par définition instables, ce n'est pas une pratique recommandée de lier des dépendances SNAPSHOT avec d'autres équipes ou depuis des sources externes.

Par exemple, imaginiez que vous êtes en train de travailler sur une nouvelle application web `game-of-life`. Vous utilisez Maven pour votre projet, donc vous pouvez utiliser une tâche de build Maven dans Jenkins. Votre équipe travaille aussi sur une librairie réutilisable appelée `cooltools`. Comme ces deux projets sont développés par la même équipe, vous utilisez certaines des dernières fonctionnalités de `cooltools` dans l'application `game-of-life`. vous avez une dépendance SNAPSHOT dans la section `<dependencies>` du fichier `pom.xml` de `game-of-life` :

```
<dependencies>
  <dependency>
    <groupId>com.acme.common</groupId>
    <artifactId>cooltools</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <scope>test</scope>
  </dependency>
  ...
</dependencies>
```

Dans votre serveur Jenkins, vous avez configuré des tâches de build Maven à la fois pour l'application `cooltools` et `game-of-life`. Comme votre projet `game-of-life` a besoin de la dernière version SNAPSHOT de `cooltools`, vous cochez l'option « Lance un build à chaque fois qu'une dépendance SNAPSHOT est construite ». Comme cela, dès lors que le projet `cooltools` est reconstruit, le projet `game-of-life` sera aussi automatiquement reconstruit.

5.9.2. Configurer un build Maven

La prochaine section où vous noterez un changement est la section Build. Dans une tâche de build Maven, la section Build est entièrement dévoué au lancement d'un seul goal Maven (voir la figure Figure 5.38, "Spécifier les goals Maven"). Dans cette section, vous spécifiez la version de Maven que vous voulez exécuter (rappelez-vous, en tant que job Maven, cela ne fonctionnera qu'avec Maven), la

position du fichier `pom.xml`, et le goal Maven (ou les goals) à invoquer. Vous pouvez aussi ajouter n'importe quelles options de ligne de commande que vous voulez ici.

The screenshot shows the 'Build' section of a Jenkins configuration page. It contains three input fields: 'Maven Version' with a dropdown menu set to 'Maven 2.2.1', 'Root POM' with a text box containing 'pom.xml', and 'Goals and options' with a text box containing 'clean install -B -U -Dsurefire.useFile=false'. Each field has a help icon to its right. Below these fields is a button labeled 'Advanced...'.

Figure 5.38. Spécifier les goals Maven

Dans beaucoup de cas, c'est tout ce que vous aurez besoin de faire pour configurer une tâche de build Maven. Cependant, si vous cliquez sur le bouton « Avancé... », vous pouvez cocher certaines fonctionnalités avancées (Figure 5.39, “Les tâches de build Maven — les options avancées”).

The screenshot shows the 'Build' section with advanced options. It includes fields for 'Root POM' (pom.xml), 'Goals and options' (clean deploy -B -U -Dsurefire.useFile=false), 'MAVEN_OPTS' (empty), and 'Alternate settings file' (empty). Below these are several checkboxes: 'Incremental build - only build changed modules' (unchecked), 'Disable automatic artifact archiving' (checked), 'Build modules in parallel' (unchecked), 'Use private Maven repository' (unchecked), 'Resolve Dependencies during Pom parsing' (unchecked), and 'Process Plugins during Pom parsing' (unchecked). At the bottom is a dropdown for 'Maven Validation Level' set to 'DEFAULT'. Each field and checkbox has a help icon to its right.

Figure 5.39. Les tâches de build Maven — les options avancées

L'option de build incrémental est très pratique pour les grands builds multi modules Maven. Si vous cochez cette option, lorsqu'un changement est effectué sur l'un des modules du projet, Jenkins reconstruira uniquement ce module et les modules qui utilisent le module modifié. Il fait cette magie en utilisant certaines nouvelles fonctionnalités introduites par Maven 2.1 (donc cela ne fonctionnera pas si vous utilisez Maven 2.0.x). Jenkins détecte quels modules a été modifiés, et utilise l'option `-pl` (`--project-list`) pour construire uniquement les modules mis à jour, et l'option `-amd` (`--also-make-dependents`) pour construire aussi les modules qui utilisent les modules mis à jour. Si rien n'a été modifié dans le code source, tous les modules sont construits.

Par défaut, Jenkins archivera tous les artefacts générés par une tâche de build Maven. Cela peut être utile à certains moments, mais cela peut aussi être très coûteux en place sur le disque. Si vous souhaitez désactiver cette option, il suffit de cocher l'option « Désactive l'archivage automatique des artefacts ».

Alternativement, vous pouvez toujours limiter les artefacts stockés en utilisant l'option « Supprimer les anciens builds » tout en haut de la page de configuration.

L'option « Construire les modules en parallèle » informe Jenkins qu'il peut démarrer chaque module individuellement en parallèle comme un build séparé. En théorie, cela pourrait rendre un peu plus rapide les constructions. En pratique, cela fonctionnera réellement si vos modules sont totalement indépendants (si vous n'utilisez pas d'agrégation), ce qui est rarement le cas. Si vous pensez que construire vos modules en parallèle peut réellement accélérer votre projet multi module, vous pouvez avoir envie d'utiliser un build freestyle avec Maven 3 et sa nouvelle fonctionnalité de construction parallèle.

Une autre option utile est « Utiliser une repository Maven privé ». Normalement, lorsque Jenkins démarre Maven, il se comportera exactement de la même manière que Maven en ligne de commande : il stockera les artefacts et retrouvera les artefacts depuis le dépôt Maven local (dans `~/.m2/repository` si vous ne l'avez pas configuré dans le fichier `settings.xml`). Ceci est efficace en terme d'espace disque, mais pas souvent idéal pour des constructions en IC. En effet, si plusieurs tâches de build sont en construction avec les mêmes artefacts en SNAPSHOT, les constructions peuvent finir par se gêner les unes avec les autres.

Lorsque cette option est sélectionnée, Jenkins demande à Maven d'utiliser `$(WORKSPACE)/.repository` comme dépôt local Maven. Cela signifie que chaque job aura son propre dépôt Maven isolé pour lui tout seul. Cela règle les problèmes ci-dessus, au détriment de la consommation d'espace disque supplémentaire.

Avec cette option, Maven utilisera un dépôt Maven dédié pour cette tâche de build, situé dans le répertoire `$(WORKSPACE)/.repository`. Ceci prend plus d'espace disque, mais garantit une meilleure isolation pour nos tâches de build.

Une autre approche pour ce problème est de redéfinir l'emplacement par défaut du dépôt en utilisant la propriété `maven.repo.local`, comme montré ici :

```
$ mvn install -Dmaven.repo.local=~/.m2/staging-repository
```

Cette approche a l'avantage d'être capable de partager un dépôt entre certaines tâches de builds, ce qui est utile si vous souhaitez faire une série de builds liés. Cela fonctionnera aussi avec des tâches Freestyle.

5.9.3. Les actions à la suite du build

Les actions à la suite du build dans une tâche de build Maven est considérablement simple à configurer comparé à une tâche Freestyle. C'est simplement parce que, puisque c'est un build Maven, Jenkins sait où chercher un certain nombre de sortie du build. Les artefacts, rapports de test, Javadoc, en ainsi de suite, sont tous générés dans les répertoires standards, ce qui signifie que vous n'avez pas besoin de préciser à Jenkins où trouver ces éléments. Donc Jenkins trouvera, et effectuera automatiquement des rapports sur des résultats de test JUnit, par exemple. Plus loin dans ce livre, nous verrons comment les projets Maven simplifient aussi la configuration de beaucoup d'outils de mesure de qualité de code et de rapports.

Beaucoup des autres actions à la suite du build sont similaires à ceux que nous avons vus dans une tâche de build Freestyle.

5.9.4. Déployer vers un gestionnaire de dépôt d'entreprise

Une option supplémentaire qui apparaît dans les tâches de build Maven est la capacité de déployer vos artefacts vers un dépôt Maven (voir la figure Figure 5.40, “Déployer des artefacts vers un dépôt Maven”). Un gestionnaire de dépôt d'entreprise est un serveur qui agit à la fois comme un proxy/cache pour des artefacts publics de Maven, et comme un serveur de stockage centralisé pour vos propres artefacts internes. Des gestionnaires de dépôt d'entreprise open source comme Nexus (de Sonatype) et Artifactory (de JFrog) fournissent des fonctionnalités de maintenance et d'administration puissantes qui permettent de configurer et maintenir vos dépôts Maven très simplement. Ces deux produits ont des versions commerciales, avec des fonctionnalités additionnelles visant à construire des infrastructures plus sophistiquées ou haut de gamme.

L'avantage d'avoir Jenkins qui déploie vos artefacts (à l'opposé de simplement faire `mvn deploy`) est que, si vous avez un build Maven multi module, les artefacts seront déployés uniquement lorsque le build entier a fini avec succès. Par exemple, supposons que vous ayez un projet Maven multi module avec cinq modules. Si vous lancez `mvn deploy`, et que le build échoue après trois modules, les deux premiers modules vont avoir été déployés vers votre dépôt, mais pas les trois premiers, qui laissent votre dépôt dans un état instable. Faire en sorte que Jenkins faire le déploiement assure que les artefacts sont déployés comme un groupe une fois que le build a terminé avec succès.



The screenshot shows the configuration for 'Deploy artifacts to Maven repository'. It includes a checked checkbox for the option, a text field for 'Repository URL' containing 'http://www.com/artifactory/libs-snapshots-local', a text field for 'Repository ID' containing 'manuka', and two unchecked checkboxes: 'Assign unique versions to snapshots' and 'Deploy even if the build is unstable'. Each field has a help icon to its right.

Figure 5.40. Déployer des artefacts vers un dépôt Maven

Pour faire cela, il suffit de cocher l'option « Déployer les artefacts dans le repository Maven » dans les actions à la suite du build. Vous aurez besoin de spécifier l'URL du dépôt sur lequel vous voulez déployer. Il faut que cela soit l'URL complète vers le dépôt (ex : `http://nexus.acme.com/nexus/content/repositories/snapshots`, et pas juste `http://nexus.acme.com/nexus`)

La plupart des dépôts ont besoin de vous authentifier avant de vous laisser déployer des artefacts. La manière standard de Maven pour faire cela est de placer `<server>` dans votre fichier `settings.xml` local, comme montré ici :

```
<settings...>
  <servers>
    <server>
```

```

<id>nexus-snapshots</id>
<username>scott</username>
<password>tiger</password>
</server>
<server>
  <id>nexus-releases</id>
  <username>scott</username>
  <password>tiger</password>
</server>
</servers>
</settings>

```

Pour les personnes qui souhaitent plus de sécurité, vous pouvez aussi encrypter ces mots de passe si besoin est.

Ensuite, entrez l'identifiant correspondant dans le champ Identifiant du repository de Jenkins. Jenkins sera alors capable de retrouver le bon nom d'utilisateur et mot de passe, et de déployer vos artefacts. Une fois que le build sera terminé, vos artefacts devraient être disponibles sur votre dépôt d'entreprise Maven (voir la figure Figure 5.41, "Après déploiement, l'artefact devrait être disponible sur votre gestionnaire de dépôt d'entreprise").

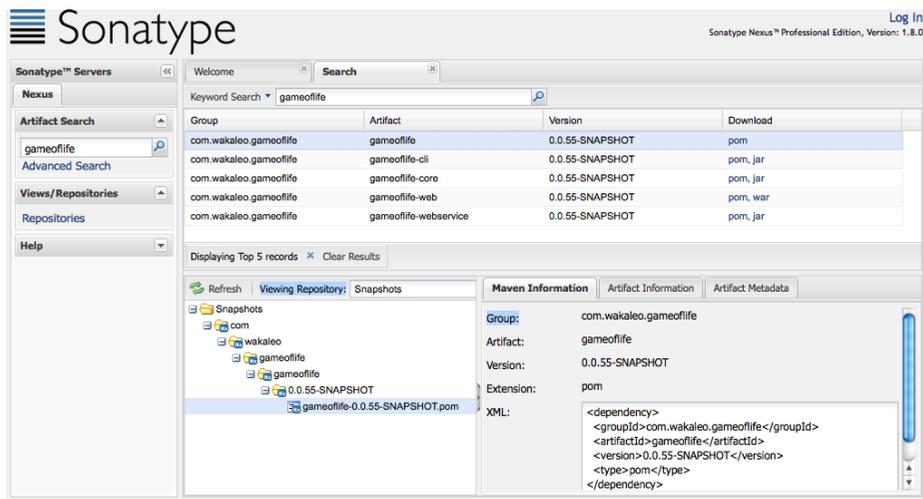


Figure 5.41. Après déploiement, l'artefact devrait être disponible sur votre gestionnaire de dépôt d'entreprise

En utilisant cette option, vous n'avez pas toujours besoin de déployer tout de suite — vous pouvez toujours revenir en arrière et redéployer des artefacts d'une version précédente. Il suffit de cliquer sur le menu « Redéployer les artefacts » à gauche et spécifier l'URL du dépôt sur lequel vous voulez redéployer votre artefact (voir la figure Figure 5.42, "Redéployer un artefact"). Comme dans l'exemple précédent, le bouton Avancé vous permet de spécifier l'identifiant pour le tag `<server>` dans votre fichier `settings.xml` local. Comme vous pourrez voir plus loin dans ce livre, vous pouvez aussi utiliser

ce déploiement dans le cadre d'un processus de promotion de build en configurant un déploiement automatique vers un dépôt différent lorsque certaines métriques de qualité sont satisfaites, par exemple.



Figure 5.42. Redéployer un artefact

Cette approche fonctionnera bien pour n'importe quel gestionnaire de dépôt d'entreprise. Cependant, si vous utilisez Artifactory, vous pourriez préférer installer le plugin Artifactory¹¹ de Jenkins, qui fournit une intégration bidirectionnelle vers le gestionnaire de dépôt d'entreprise Artifactory. Il fonctionne en envoyant des informations supplémentaires au serveur Artifactory durant le déploiement, permettant au serveur d'avoir un lien vers le build qui a généré un artefact donné. Une fois que vous avez installé le plugin, vous pouvez l'activer dans votre tâche de build Maven en cochant l'option « Deploy artifacts to Artifactory » dans les actions à la suite du build. Ensuite vous choisissez sur quel dépôt votre projet doit déployer dans une liste de dépôt sur le serveur, avec le nom d'utilisateur et le mot de passe requis pour faire le déploiement (voir la figure Figure 5.43, «Déployer vers Artifactory depuis Jenkins»).



Figure 5.43. Déployer vers Artifactory depuis Jenkins

Votre tâche de build déploiera automatiquement vers Artifactory. En supplément, un lien vers l'artefact sur le serveur sera maintenant affiché sur la page d'accueil de la tâche de build et la page de résultat des builds (voir la figure Figure 5.44, «Jenkins affiche un lien vers le dépôt Artifactory correspondant»).

¹¹ <http://wiki.jenkins-ci.org/display/JENKINS/Artifactory+Plugin>



Figure 5.44. Jenkins affiche un lien vers le dépôt Artifactory correspondant

Ce lien vous emmène sur une page sur le serveur Artifactory contenant les artefacts déployés (voir la figure Figure 5.45, “Voir l’artefact déployé sur Artifactory”). Depuis cette page, il y a aussi un lien pour vous ramener vers la page du build qui a construit cet artefact.



Figure 5.45. Voir l’artefact déployé sur Artifactory

5.9.5. Déployer vers des gestionnaires de dépôt d’entreprise commerciales

Un gestionnaire de dépôt d’entreprise est une partie essentielle de n’importe quelle infrastructure de développement de logiciel basé sur Maven. Ils jouent aussi un rôle clé pour les projets non basés sur Maven utilisant des outils tels que Ivy ou Gradle, chacun d’eux étant liés aux dépôts standards de Maven.

Chacun des principaux gestionnaires de dépôt d’entreprise, Nexus and Artifactory, propose des versions professionnelles qui fournissent des fonctionnalités supplémentaires avec Jenkins. Plus loin dans ce livre, nous discuterons comment vous pouvez utiliser des fonctionnalités avancées comme la gestion de release et de staging de Nexus Pro pour implémenter des stratégies sophistiquées de promotion de build. Sur l’aspect déploiement, l’édition commerciale d’Artifactory (Artifactory Pro Power Pack) étend l’intégration bidirectionnelle que nous avons vu plus tôt. Lorsque vous voyez un artefact sur le navigateur du dépôt, un onglet ‘Builds’ affiche les détails sur le build Jenkins Jenkins qui a créé

l'artefact, et un lien vers la page du build sur Jenkins (voir la figure Figure 5.46, “Voir les artefacts déployés et le build Jenkins correspondant dans Artifactory”). Artifactory assure également le suivi des dépendances qui ont été utilisées dans le build Jenkins, et vous alertera si vous essayez de les supprimer depuis le dépôt.



Figure 5.46. Voir les artefacts déployés et le build Jenkins correspondant dans Artifactory

5.9.6. Gérer les modules

Lorsque vous utilisez Maven, il est habituel de découper le projet en plusieurs modules. Les tâches de build Maven ont une connaissance intrinsèque des projets multi modules, et ajoute un élément Modules au menu qui vous permet d’afficher la structure du projet d’un coup d’œil (voir la figure Figure 5.47, “Gérer les modules dans une tâche de build Maven”).



Figure 5.47. Gérer les modules dans une tâche de build Maven

Cliquer sur l'un des modules vous permettra d'afficher la page de build pour ce module. A partir de la, vous pouvez voir les résultats détaillés des builds pour chaque module, déclencher un build pour un module isolé, et si nécessaire, affiner la configuration d'un module, surchargeant ainsi la configuration global du projet.

5.9.7. Les étapes de build supplémentaires dans votre tâche de build Maven

Par défaut, une tâche de build Maven ne permet qu'un seul goal Maven. Il y a des fois où c'est assez limitatif, et vous voudriez ajouter des étapes supplémentaires avant ou après la build principal. Vous pouvez faire ceci avec le plugin M2 Extra Steps de Jenkins. Ce plugin vous permet d'ajouter des étapes de builds normales avant et après le goal Maven principal, vous donnant la flexibilité d'une tâche de build Freestyle tout en gardant l'avantage de la configuration d'une tâche de build Maven.

Installez le plugin et allez dans la section Environnements de Build de votre tâche de build. Cochez l'option « Configure M2 Extra Build Steps ». Vous devriez pouvoir maintenant ajouter des étapes de build qui seront ajoutées avant et/ou après que votre goal Maven principal soit exécuté (voir la figure Figure 5.48, « Configurer des étapes de build Maven supplémentaires »).

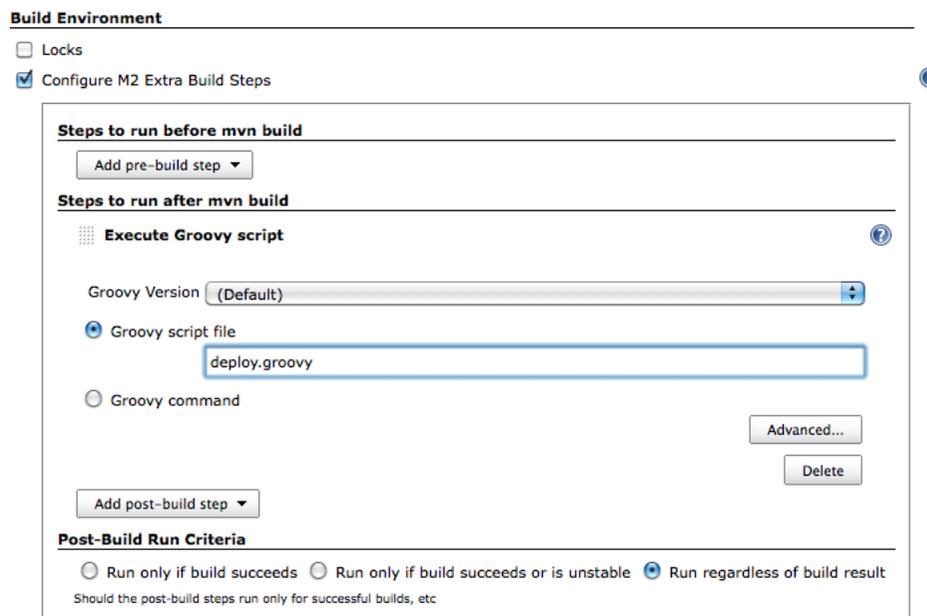


Figure 5.48. Configurer des étapes de build Maven supplémentaires

5.10. Utiliser Jenkins avec d'autres langages

Comme mentionné plus tôt, Jenkins fournit un excellent support pour d'autres langages. Dans cette section, nous allons voir comment utiliser Jenkins avec certains des plus communément utilisés.

5.10.1. Construire des projets avec Grails

Grails est un framework d'application web dynamique open-source construit avec Groovy et beaucoup d'autres frameworks Java open-source bien établis tel que Spring ou Hibernate.

Jenkins fournit un excellent support pour les builds Grails. Premièrement, vous devez installer le plugin Grails¹². Une fois que vous l'avez installé et redémarré Jenkins, vous devrez fournir au moins une version de Grails pour Jenkins dans la section Grails de l'écran de configuration du système (voir la figure Figure 5.49, "Ajouter une installation Grails à Jenkins").

Grails Builder

Grails installation

name	<input type="text" value="Grails 1.3.4"/>
GRAILS_HOME	<input type="text" value="/Library/Grails/grails-1.3.4"/>

[List of Grails installations on this system](#)

Figure 5.49. Ajouter une installation Grails à Jenkins

Maintenant vous pouvez configurer une tâche de build Freestyle pour construire un projet Grails. Le plugin Grails ajoute l'étape de build « Build with Grails », que vous pouvez utiliser pour construire un application Grails (voir la figure Figure 5.50, "Configurer une étape de build Grails"). Ici, vous fournissez la target Grails, ou les targets, que vous voulez exécuter. A la différence de la ligne de commande, vous pouvez exécuter plusieurs targets dans la même ligne de commande. Cependant, si vous souhaitez passer des arguments sur une target particulier, vous devez placer la target et ses arguments entre guillemet. Dans la Figure 5.50, "Configurer une étape de build Grails", par exemple, nous lançons `grails clean`, suivi de `grails test-app -unit -non-interactive`. ur que cela fonctionne correctement, il faut mettre les options de la seconde commande entre guillemet, ce qui nous donne `grails clean "test-app -unit -non-interactive"`.

¹² <http://wiki.jenkins-ci.org/display/HUDSON/Grails+Plugin>

Build

Build With Grails

Grails Installation

Select a Grails installation to use

Force Upgrade

Run 'grails upgrade --non-interactive' first

Targets

Specify target(s) to run separated by spaces (optional).

server.port

Specify a value for the server.port system property (optional)

grails.work.dir

Specify a value for the grails.work.dir system property (optional)

grails.project.work.dir

Specify a value for the grails.project.work.dir system property (optional)

Project Base Directory

Specify a path to the root of the Grails project (optional)

Properties

Additional system properties to set (optional)

Figure 5.50. Configurer une étape de build Grails

L'étape de build Grails peut prendre beaucoup de paramètres optionnels. Par exemple, Grails est minutieux avec les versions — si votre projet a été créé avec une version plus ancienne, Grails vous demandera de la mettre à jour. Pour jouer la sécurité, par exemple, vous pourriez cocher la case à cocher Force Upgrade, qui assure de lancer un `grails upgrade --non-interactive` avant de démarrer les targets principales.

Vous pouvez aussi configurer le port du serveur (utile si vous exécutez des tests web), et n'importe quel autre propriété que vous voulez passer au build.

5.10.2. Construire des projets avec Gradle

Rédigé par René Groeschke

En comparaison des outils de build anciens Ant et Maven, Gradle¹³ est un nouvel outil de build open source pour la machine virtuelle Java. Les scripts de build pour Gradle sont construits dans un Domain Specific Language (DSL) basé sur Groovy. Gradle implémente la convention avant la configuration, permet un accès direct aux tasks Ant, et utilise une gestion des dépendances déclarative comme Maven. La nature concise du scripting Groovy vous permet d'écrire des scripts de build plus expressif avec très peu de code, au prix de la perte du support d'un IDE qui existe pour les outils établis tel que Ant et Maven.

¹³ <http://gradle.org>

Il y a deux manières différentes de lancer des builds Gradle avec Jenkins. Vous pouvez utiliser soit le plugin Gradle pour Jenkins soit la fonctionnalité d’enveloppement (wrapper) de Gradle.

5.10.2.1. Le plugin Gradle de Jenkins

Vous pouvez installer le plugin Gradle avec la façon habituel — il suffit d’aller sur l’écran du gestionnaire de plugin et de sélectionner le plugin Gradle. Cliquez sur installer et redémarrez l’instance de Jenkins.

Une fois que Jenkins a redémarré, il vous faut configurer vous nouveau plugin Gradle. Vous trouverez maintenant une nouvelle section Gradle dans la page de configuration du système. Ici, il vous faudra ajouter une installation de Gradle que vous souhaitez utiliser. Le processus est similaire que pour les autres installations d’outil utilisé. Premièrement, cliquez sur le bouton Ajouter Gradle pour ajouter une nouvelle installation Gradle, et entrez un nom approprié (voir la figure Figure 5.51, “Configurer le plugin Gradle”). Si Gradle a déjà été installé sur votre serveur de build, vous pouvez pointer vers le répertoire local de Gradle. Sinon, vous pouvez utiliser la fonctionnalité « Installer automatiquement » pour télécharger une installation de Gradle, sous la forme d’un fichier ZIP ou GZipped TAR, directement à partir d’une URL. Vous pouvez utiliser une URL publique (voir <http://gradle.org/downloads.html>), ou vous pourriez préférer de mettre ces installations disponibles sur le serveur local uniquement.

The screenshot shows the Jenkins configuration page for Gradle. It includes a form with the following fields and controls:

- name:** Gradle-0.9RC2
- Install automatically:**
- Extract *.zip/*.tar.gz:**
- Label:** (empty text field)
- Download URL for binary archive:** <http://dist.codehaus.org/gradle/gradle-0.9-rc-2-bin.zip>
- Subdirectory of extracted archive:** gradle-0.9-rc-2
- Buttons:** Add Installer, Delete Installer, Add Gradle, Delete Gradle
- Footer:** List of Gradle installations on this system

Figure 5.51. Configurer le plugin Gradle

Vous utilisez typiquement des tâches de build Freestyle pour configurer vos builds Gradle. Lorsque vous ajouter une étape de build a une tâche de build Freestyle, vous aurez maintenant une nouvelle option appelée « Invoque Gradle script », qui vous permet d’ajouter des options spécifiques de Gradle à votre tâche de build.

Par exemple, voici un script Gradle très simple. C’est un projet simple Java qui utilise une structure de répertoire Maven et un gestionnaire de dépôt Maven. Il y a une tâche paramétrable, appelée uploadArchives, pour déployer l’archive générée vers un gestionnaire local de dépôt d’entreprise :

```
apply plugin: 'java'
apply plugin: 'maven'
```

```

version='1.0-SNAPSHOT'
group = "org.acme"

repositories{
    mavenCentral()
    mavenRepo urls: 'http://build.server/nexus/content/repositories/public'
}

dependencies{
    testCompile "junit:junit:4.8.2"
}

uploadArchives {
    repositories.mavenDeployer {
        configuration = configurations.archives
        repository(url: "http://build.server/nexus/content/repositories/snapshots") {
            authentication(userName: "admin", password: "password")
        }
    }
}

```

Dans la Figure 5.52, “Configurer une tâche de build Gradle”, nous utilisons l’instance « Gradle-0.9RC2 » qui vient d’être configurée pour démarrer un build Gradle. Dans ce cas, nous souhaitons démarrer des tests JUnit et télécharger les artefacts de build vers notre dépôt local Maven. De plus, nous configurons notre tâche pour collecter les résultats de test depuis `**/build/test-results`, le répertoire par défaut stockant les résultats de test avec Gradle.

5.10.2.2. Les builds incrémentaux

Lorsqu’on lance une tâche de build Gradle avec des sources non modifiées, Gradle lance ses builds incrémentaux. Si la sortie d’une tâche Gradle est toujours disponible et que les sources n’ont pas changé depuis le dernier build, Gradle est capable de sauter l’exécution de la tâche et de la marquer comme à jour. Cette fonctionnalité de build incrémental peut considérablement diminuer le temps d’une tâche de build.

Si Gradle évalue qu’une tâche de test est à jour, même l’exécution de vos tests unitaires est sautée. Cela peut poser des problèmes lorsque vous lancez votre build Gradle avec Jenkins. Dans notre tâche de build simple ci-dessus, nous avons configuré une action à la suite du build pour publier les rapports JUnit de votre build. Si la tâche de test est sautée par Gradle, Jenkins marquera la tâche en erreur avec le message suivant :

Test reports were found but none of them are new. Did tests run?

Vous pouvez simplement résoudre ceci en invalidant la sortie et forcer une réexécution de vos tests en ajoutant le bout de code suivant dans votre fichier Gradle :

```

test {
    outputs.upToDateWhen { false }
}

```

Build

Invoke Gradle script

Gradle Version:

Build step description:

Switches:

Tasks:

Root Build script:

Build File:

Specify Gradle build file to run. Also, [some environment variables are available to the build script](#)

Post-build Actions

Publish Javadoc

Archive the artifacts

Aggregate downstream test results

Publish JUnit test result report

Test report XMLs:

Best Practices: ***/build/test-results/*.xml' doesn't match anything: '**' exists but not '**/build/test-results/*.xml'**

[Best Practices](#): setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Base dir of the fileset is the [workspace root](#).

Retain long standard output/error

Figure 5.52. Configurer une tâche de build Gradle

Après avoir ajouté le bout de code ci-dessus dans votre fichier de build, la sortie console de la tâche devrait ressembler à celle de la figure Figure 5.53, “Tâche incrémentale de Gradle”.

Console Output

```

Started by user anonymous
[SampleApp1] $ /Users/Rene/.hudson/tools/Gradle-0.9RC2/gradle-0.9-rc-2/bin/gradle test uploadArchives
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test
:jar UP-TO-DATE
:uploadArchivesUploading: org/acme/SampleApp1/1.0-SNAPSHOT/SampleApp1-1.0-20101123.220032-2.jar to repository remote at
http://localhost:8081/artifactory/gradlerepo
Transferring 1K from remote
Uploaded 1K

BUILD SUCCESSFUL

Total time: 24.325 secs
Recording test results
Finished: SUCCESS

```

Figure 5.53. Tâche incrémentale de Gradle

Comme vous pouvez le voir, toutes les tâches exceptées test et uploadArchives ont été marquées comme à jour et n’ont pas été exécutées.

5.10.3. Construire des projets avec Visual Studio MSBuild

Jenkins est une application Java, mais il fournit aussi un excellent support pour les applications .NET.

Pour construire des projets .NET dans Jenkins, vous devez installer le MSBuild plugin¹⁴.

¹⁴ <http://wiki.jenkins-ci.org/display/HUDSON/MSBuild+Plugin>

Vous pourriez aussi vouloir installer le plugin MSTest¹⁵ et le plugin NUnit¹⁶, pour afficher vos résultats de test.

Une fois que vous avez installé les plugins .NET et redémarré Jenkins, vous devez configurer vos outils de build .NET. Allez dans la page de configuration du système et spécifiez le chemin vers l'exécutable MSBuild (voir la figure Figure 5.54, "Configurer les outils de build .NET avec Jenkins").

The screenshot shows two sections of the Jenkins configuration page. The first section is titled "MSBuild Builder" and contains a form for adding a new MSBuild installation. It has two input fields: "name" with the value "MSBuild 3.5" and "Path To msbuild.exe" with the value "C:\Windows\Microsoft.NET\Framework\v3.5\msbuild.exe". There are "Add" and "Delete" buttons. Below the form is a link that says "List of MSBuild installations on this system". The second section is titled "Nant Builder" and contains a form for adding a new NAnt installation. It has two input fields: "name" with the value "NAnt 0.90" and "NANT_HOME" with the value "C:\Program Files\NAnt". There are "Add" and "Delete" buttons.

Figure 5.54. Configurer les outils de build .NET avec Jenkins

Une fois que vous l'avez configuré, vous pouvez retourner dans votre projet Freestyle et ajouter une étape de build .NET à la configuration.

Allez dans la section Build et choisissez l'option « Build a Visual project or solution using MSBuild » dans le menu Ajouter une étape de build. Ensuite, entrez le chemin vers votre script de build MSBuild (un fichier .proj ou .sln), avec n'importe quelle option de ligne de commande que votre script de build a besoin (voir la figure Figure 5.55, "Une étape de build utilisant MSBuild").

The screenshot shows a configuration page for a build step. It has a title "Build a Visual Studio project or solution using MSBuild." and three input fields: "MSBuild Version" with a dropdown menu showing "MSBuild 3.5", "MSBuild Build File" with the value "gameoflife.sln", and "Command Line Arguments" with the value "/p:Configuration=Release". There is a "Delete" button at the bottom right.

Figure 5.55. Une étape de build utilisant MSBuild

5.10.4. Construire des projets avec NAnt

Un autre moyen de construire vos applications .NET est d'utiliser NAnt. NAnt est la version .NET de l'outil de scripting de build largement utilisé dans le monde Java. Les scripts de build NAnt sont des

¹⁵ <http://wiki.jenkins-ci.org/display/HUDSON/MSTest+Plugin>

¹⁶ <http://wiki.jenkins-ci.org/display/HUDSON/NUnit+Plugin>

fichiers XML (typiquement avec une extension `.build`), avec un format très similaire au script de build Ant.

Pour construire avec NAnt dans Jenkins, il vous faut installer le plugin NAnt¹⁷ de Jenkins. Une fois que vous l’avez installé et redémarré Jenkins, allez dans la page de configuration du système et spécifiez un répertoire d’installation NAnt dans la section NAnt Builder (voir la figure Figure 5.54, “Configurer les outils de build .NET avec Jenkins”).

Maintenant, allez dans la section Build de votre projet Freestyle et choisissez « Execute NAnt build » (voir la figure Figure 5.56, “Une étape de build utilisant NAnt”). Ici, vous spécifiez le script de build et la target que vous voulez invoquer. Si vous cliquez sur l’option « Avancée... », vous pouvez aussi spécifier des valeurs de propriété pour les passer dans le script NAnt.



Figure 5.56. Une étape de build utilisant NAnt

5.10.5. Construire des projets avec Ruby et Ruby on Rails

Jenkins est un excellent choix lorsqu’il s’agit de faire de l’IC sur vos projets Ruby et Ruby on Rails. Le plugin Rake vous permet d’ajouter des étapes de build Rake dans vos tâches de build. Vous pouvez aussi utiliser le plugin Ruby qui vous permet de lancer des scripts Ruby directement dans votre tâche de build. Finalement, le plugin Ruby Metrics fournit un support pour les outils de métriques de qualité de code comme RCov, Rails stats, et Flog.

Un autre outil précieux dans ce domaine est `CI:Reporter`. Cette bibliothèque est un add-on de `Test::Unit`, `RSpec`, et `Cucumber` qui génère des rapports compatible avec JUnit de vos tests. Comme vous allez le voir, les résultats de test compatibles avec JUnit peuvent être utilisés directement par Jenkins pour faire des rapports de vos résultats de test. Vous devez installer `CI:Reporter` en utilisant Gem comme illustré ici :

```
$ sudo gem install ci_reporter
Successfully installed ci_reporter-1.6.4
1 gem installed
```

Ensuite, il vous faudra le configurer dans votre Rakefile, en ajoutant ce qui suit :

¹⁷ <http://wiki.jenkins-ci.org/display/HUDSON/NAnt+Plugin>

```
require 'rubygems'
gem 'ci_reporter'
require 'ci/reporter/rake/test_unit' # use this if you're using Test::Unit
```

Dans le chapitre Chapter 9, Qualité du Code, nous discutons de l'intégration des métriques de qualité de code dans vos builds Jenkins. Jenkins fournit aussi un support sur les métriques de couverture de code en Ruby. Le plugin Ruby Metrics supporte les métriques de qualité de code en utilisant `rcov` aussi bien que des statistiques de code générales avec `Rails stats`. Pour installer le plugin `rcov`, vous devez tout d'abord exécuter quelque chose comme les lignes suivantes :

```
$ ./script/plugin install http://svn.codahale.com/rails_rcov
```

Une fois que c'est configuré, vous serez en mesure d'afficher vos résultats de test et la tendance de résultat de test dans Jenkins.

Finalement, vous pouvez configurer un build Rake simplement en utilisant une étape de build Rake, comme illustré dans la Figure 5.57, "Une étape de build utilisant Rake".

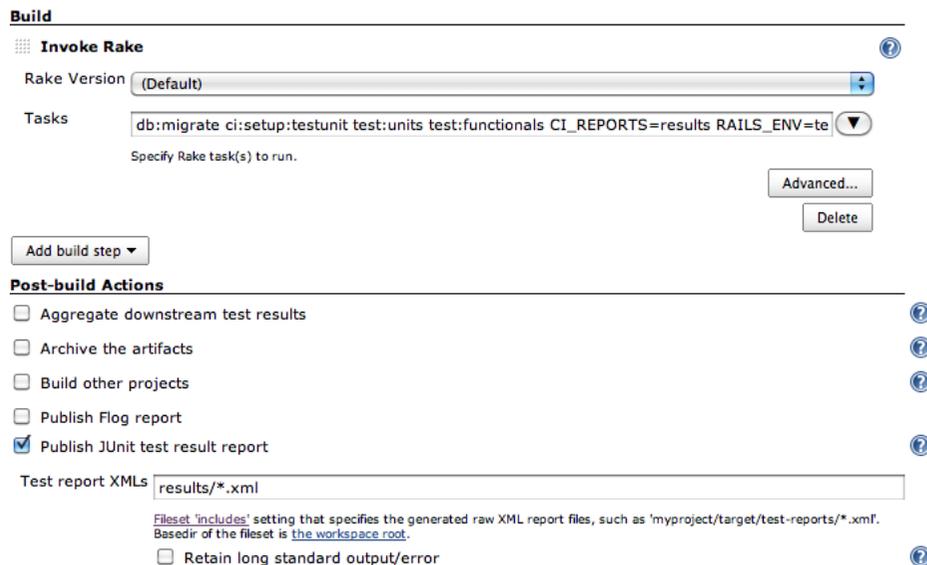


Figure 5.57. Une étape de build utilisant Rake

Vous devez aussi configurer Jenkins pour faire des rapports sur les résultats des tests et les métriques de qualité. Vous pouvez faire ceci en activant les options « Publish JUnit test result report », « Publish Rails stats report », et « Public Rcov report » (voir la figure Figure 5.58, "Publier des métriques de qualité de code pour Ruby et Rails"). Les rapports XML JUnit seront trouvés dans le répertoire `results` (entrez `results/*.xml` dans le champ "Test report XMLs"), et le rapport dans le répertoire `coverage/units`.

Post-build Actions

- Aggregate downstream test results
- Archive the artifacts
- Build other projects
- Publish Flog report
- Publish JUnit test result report

Test report XMLs:

Fileset 'includes' setting that specifies the generated raw XML report files, such as 'myproject/target/test-reports/*.xml'. Basedir of the fileset is [the workspace root](#).

- Retain long standard output/error
- Publish Javadoc
- Publish Rails Notes report
- Publish Rails stats report

Rake Version:

- Publish Rcov report

Rcov report directory:

relative path to the coverage report directory

Coverage metric targets

Total coverage	<input type="text" value="80"/>	<input type="text" value="0"/>	<input type="text" value="0"/>
Code coverage	<input type="text" value="80"/>	<input type="text" value="0"/>	<input type="text" value="0"/>

Configure health reporting thresholds.
For the row, leave blank to use the default value (i.e. 80).
For the and rows, leave blank to use the default values (i.e. 0).

Figure 5.58. Publier des métriques de qualité de code pour Ruby et Rails

5.11. Conclusion

Dans ce chapitre, nous avons couvert les bases sur la création de tâches de build pour les cas les plus communs que vous pouvez rencontrer. Plus tard dans ce livre, nous allons utiliser ces fondamentaux pour discuter sur des options plus avancées comme les builds paramétrés, les builds matriciels, et les stratégies de promotion de build.

Chapter 6. Tests automatisés

6.1. Introduction

Si vous n'utilisez pas les tests automatisés avec votre environnement d'intégration continue, vous passez à côté d'un aspect important. Croyez-moi — l'IC sans les tests automatisés représente juste une petite amélioration pour les tâches de build lancées automatiquement. Maintenant, ne vous méprenez pas, si vous partez de zéro, c'est quand même un grand pas en avant, mais vous pouvez faire mieux. En résumé, si vous utilisez Jenkins sans tests automatisés, vous n'obtenez pas autant de valeur de votre infrastructure d'intégration continue que vous le devriez.

Un des principes de base de l'intégration continue est qu'un build doit être vérifiable. Vous devez être capable de déterminer objectivement si un build donné est prêt à passer à la prochaine étape du processus de construction, et le meilleur moyen de le faire est d'utiliser les tests automatisés. Sans une bonne automatisation des tests, vous allez devoir conserver de nombreux artefacts construits et les tester manuellement, ce qui est contraire à l'esprit de l'intégration continue.

Il y a plusieurs façons d'intégrer les tests automatisés dans votre application. Une des façons les plus efficaces pour écrire des tests de haute qualité est de les écrire en premier, en utilisant des techniques comme Test-Driven Development (TDD) ou Behavior-Driven Development (BDD). Dans cette approche, utilisée couramment dans de nombreux projets agiles, le but de vos tests unitaires est à la fois de clarifier votre compréhension du comportement du code et d'écrire un test automatisé prouvant que le code implémente le comportement. En se concentrant sur le test du comportement attendu de votre code, plutôt que sur son implémentation, cela rend les tests plus compréhensibles et plus pertinents, et par conséquent aide Jenkins à fournir une information plus pertinente.

Bien entendu, l'approche plus classique mettant en oeuvre des tests unitaires, quand le code a été implémenté, est aussi couramment utilisée, et est certainement mieux que pas de tests du tout.

Jenkins n'est pas limité aux tests unitaires, cependant. Il y a plusieurs autres types de tests automatisés que vous devriez envisager, selon la nature de votre application, parmi les tests d'intégration, les tests web, les tests fonctionnels, les tests de performance, les tests de charge et autres. Tous ceux-ci ont leur place dans un environnement de build automatisé.

Jenkins peut aussi être utilisé, en conjonction avec des techniques telles que Behavior-Driven Development et Acceptance Test Driven Development, comme un outil de communication destiné à la fois aux développeurs et aux autres intervenants d'un projet. Des frameworks BDD tels que easyb, fitness, jbehave, rspec, Cucumber, et beaucoup d'autres, essaient de présenter les tests d'acceptation de sorte que les testeurs, les Product Owners, et les utilisateurs puissent les comprendre. Avec de tels outils, Jenkins peut fournir des informations sur l'avancement d'un projet en termes métiers, et ainsi faciliter la communication entre développeurs et les non-développeurs à l'intérieur d'une équipe.

Pour des applications existantes avec peu ou pas de tests automatisés existants, cela peut être difficile et consommateur en temps de mettre en place des tests unitaires complets dans le code. De plus, les tests peuvent ne pas être très efficaces, parce qu'ils auront tendance à valider l'implémentation existante plutôt que vérifier le fonctionnel attendu. Une approche intéressante dans ces situations est d'écrire des tests fonctionnels automatisés (“régression”) qui simulent les manières les plus courantes d'utilisation de l'application. Par exemple, les outils de tests web automatisés tels que Selenium et WebDriver peuvent être utilisés efficacement pour tester les applications web à un haut niveau. Alors que cette approche n'est pas aussi complète que la combinaison de tests unitaires, d'intégration et d'acceptation de bonne qualité, c'est quand même une façon efficace et économique d'intégrer des tests de régression automatisés dans une application existante.

Dans ce chapitre, nous allons voir comment Jenkins vous aide à conserver les traces des résultats des tests automatisés, et comment vous pouvez utiliser cette information pour surveiller et disséquer votre processus de build.

6.2. Automatisez vos tests unitaires et d'intégration

Le premier sujet que nous allons regarder est comment intégrer vos tests unitaires dans Jenkins. Que vous maîtrisiez l'approche Test-Driven Development, ou que vous écriviez des tests unitaires avec une approche plus conventionnelle, ce seront probablement les premiers tests que vous voudrez automatiser avec Jenkins.

Jenkins est excellent dans l'analyse des résultats de vos tests. Cependant, c'est à vous d'écrire les tests appropriés et de configurer votre script de build pour qu'il les exécute automatiquement. Heureusement, intégrer des tests unitaires dans vos builds automatisés est généralement aisé.

Il y a de nombreux outils de tests unitaires, avec la famille xUnit occupant une place prépondérante. Dans le monde Java, JUnit est le standard de facto, bien que TestNG soit aussi un framework de test unitaire populaire avec un certain nombre de fonctionnalités innovantes. Pour les applications C#, le framework NUnit propose des fonctionnalités similaires à celles fournies par JUnit, comme le fait `Test::Unit` pour Ruby. Pour C/C++, il y a CppUnit, et les développeurs PHP peuvent utiliser PHPUnit. Et cette liste n'est pas exhaustive !

Ces outils peuvent aussi être utilisés pour les tests d'intégration, les tests fonctionnels, les tests web et ainsi de suite. De nombreux outils de tests web, comme Selenium, WebDriver, et Watir, génèrent des rapports compatibles xUnit. Les outils Behaviour-Driven Development et de tests d'acceptation automatisés comme easyb, Fitnesse, Concordion sont aussi orientés xUnit. Dans les sections suivantes, nous ne faisons pas de distinction entre ces différents types de test, parce que, d'un point de vue configuration, ils sont traités de la même façon par Jenkins. Cependant, vous aurez certainement à faire la distinction dans vos tâches de build. Afin d'obtenir un compte-rendu plus rapide, vos tests devront être groupés dans des catégories bien définies, en commençant par les rapides tests unitaires, ensuite viendront les tests d'intégration, pour finir par exécuter les tests fonctionnels et web, plus longs.

Une discussion détaillée sur la façon d'automatiser vos tests est en dehors du sujet de ce livre, mais nous couvrons quelques techniques utiles pour Apache Maven et Ant dans Appendix A, Automatiser vos tests unitaires et d'intégration.

6.3. Configuration des rapports de test dans Jenkins

Une fois que votre build génère des résultats de test, vous devez configurer votre tâche de build Jenkins afin de les afficher. Comme mentionné précédemment, Jenkins sait traiter n'importe quel rapport de tests compatible xUnit, quel que soit le langage avec lequel ils ont été écrits.

Pour les tâches de build Maven, aucune configuration spécifique n'est requise — assurez-vous seulement que vous lancez bien un goal qui va exécuter vos tests, tel que `mvn test` (pour vos tests unitaires) ou `mvn verify` (pour les test unitaires et d'intégration). Un exemple de configuration de tâche de build Maven est donné dans Figure 6.1, “Vous configurez votre installation Jenkins dans l'écran Administrer Jenkins”.

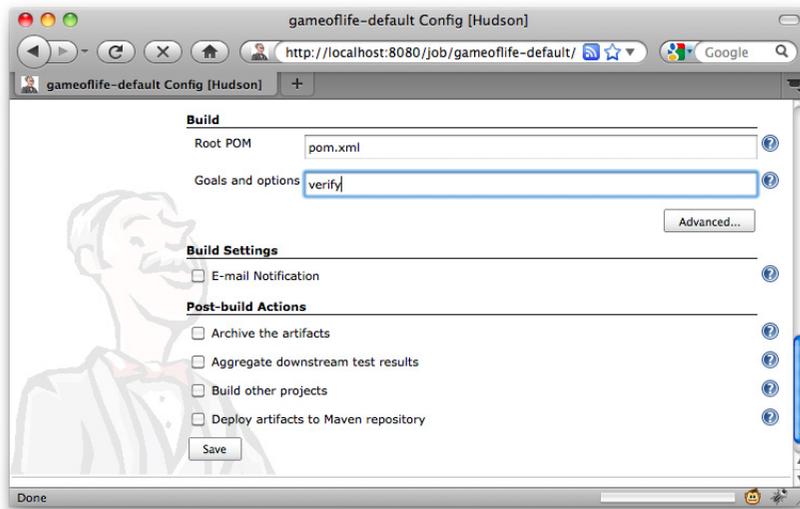


Figure 6.1. Vous configurez votre installation Jenkins dans l'écran Administrer Jenkins

Pour les tâches de build free-style, vous devez effectuer un petit travail de configuration. En plus de vous assurer que votre build exécute les tests, vous devez indiquer à Jenkins de publier les rapports de test JUnit. Vous le configurez dans la section “Actions à la suite du build” (voir Figure 6.2, “Configurer les rapports de test Maven dans un projet free-style”). Ici, vous fournissez un chemin vers les rapports XML JUnit ou TestNG. Leur chemin exact dépend du projet — pour un projet Maven, un chemin tel que `**/target/surefire-reports/*.xml` les trouvera pour la plupart des projets. Pour un projet Ant, cela dépendra de la façon dont vous avez configuré la tâche Ant JUnit, comme discuté précédemment.

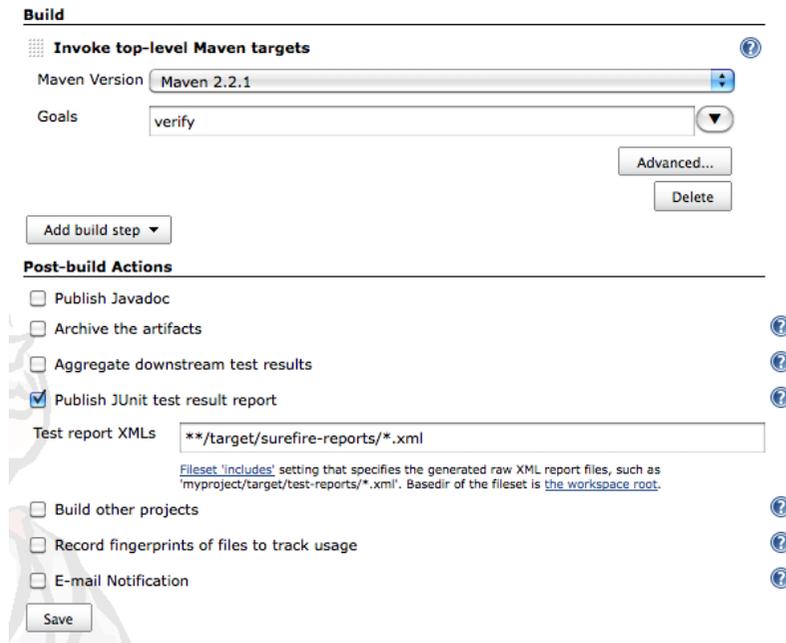


Figure 6.2. Configurer les rapports de test Maven dans un projet free-style

Pour les projets Java, qu'ils utilisent JUnit ou TestNG, Jenkins fournit une excellente intégration de base. Si vous utilisez Jenkins pour des projets non Java, vous aurez besoin du plugin xUnit. Ce plugin permet à Jenkins de traiter les rapports de test de projets non Java d'une façon uniforme. Il fournit un support de MSUnit et NUnit (pour C# et d'autres langages .NET), UnitTest++ et Boost Test (pour C++), PHPUnit (pour PHP), ainsi que quelques autres bibliothèques xUnit via d'autres plugins (voir Figure 6.3, "Installer le plugin xUnit").

<input type="checkbox"/>	eXtreme Feedback Panel This plugin provides an eXtreme Feedback Panel that can be used to expose the status of a selected number of Jobs.	1.0.8
<input checked="" type="checkbox"/>	xUnit Plugin This plugin allows you to publish testing tools test result report.	0.6.1

Build Tools

Figure 6.3. Installer le plugin xUnit

Une fois que vous avez installé le plugin xUnit, vous devez configurer le traitement pour vos rapports xUnit dans la section "Actions à la suite du build". Sélectionnez la case "Publish testing tools result report", et saisissez le chemin vers les rapports XML générés par votre bibliothèque de test (voir Figure 6.4, "Publier les résultat de test xUnit"). Quand la tâche de build s'exécute, Jenkins convertira ces rapports en rapports JUnit de telle manière à ce qu'ils soient affichés dans Jenkins.



Figure 6.4. Publier les résultats de test xUnit

6.4. Afficher les résultats de test

Une fois que Jenkins sait où se trouvent les rapports de test, il fournit un excellent travail de rapport sur ces derniers. En effet, une des tâches principales de Jenkins est de détecter et de fournir des rapports sur des échecs de build. Et un test unitaire échoué est un des symptômes les plus évidents.

Comme nous le mentionnions plus tôt, Jenkins fait la distinction entre les builds échoués et les builds instables. Un build échoué (indiqué par une balle rouge) signale des tests échoués, ou une tâche de build qui est cassée d'une façon brutale, telle qu'une erreur de compilation. Un build instable, en revanche, est un build qui n'est pas considéré de qualité suffisante. C'est intentionnellement vague : ce qui définit la “qualité” est généralement donné par vous, mais c'est typiquement lié aux métriques de code comme la couverture de code ou les standards de codage, ce que nous discuterons plus tard dans le livre. Pour l'instant, concentrons-nous sur les builds échoués.

Dans Figure 6.5, “Jenkins affiche la tendance des résultats de test sur la page d'accueil du projet” nous pouvons voir comment Jenkins affiche une tâche de build Maven contenant des échecs de tests. Ceci est la page d'accueil de la tâche de build, qui devrait être votre premier point d'entrée quand un build échoue. Quand un build contient des tests en échec, le lien Dernier résultats des tests indique le nombre courant d'échecs de test dans cette tâche de build (“5 échecs” dans l'illustration), et aussi la différence dans le nombre d'échecs de test depuis le dernier build (“+5” dans l'illustration — cinq nouveaux échecs de test). Vous pouvez aussi voir comment les tests ont réussi dans le temps — les échecs de test des builds précédents apparaîtront en rouge dans le graphique Tendance des résultats des tests.

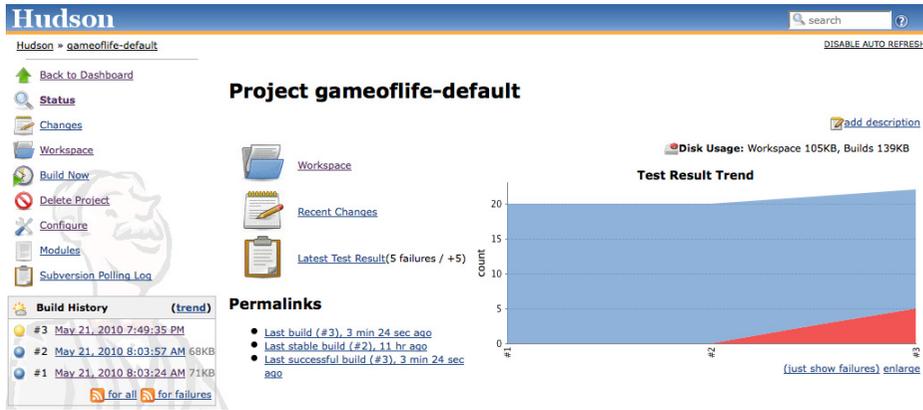


Figure 6.5. Jenkins affiche la tendance des résultats de test sur la page d'accueil du projet

Si vous cliquez sur le lien Derniers résultats des tests, Jenkins vous donnera un aperçu des résultats des derniers tests (voir Figure 6.6, “Jenkins affiche une vue synthétique des résultats de test”). Jenkins supporte les structures de projets multi-modules Maven, et pour une tâche de build Maven, Jenkins va afficher une vue synthétique des résultats de test par module. Pour voir plus de détails sur les tests en échec dans un module particulier, cliquez simplement sur le module qui vous intéresse.

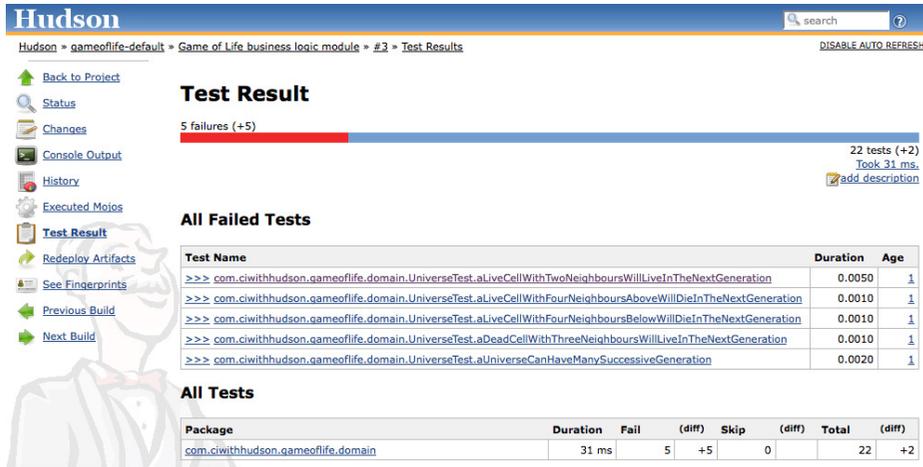


Figure 6.6. Jenkins affiche une vue synthétique des résultats de test

Pour les tâches de build free-style, Jenkins vous donnera directement un aperçu de vos résultats de test, mais organisé par packages de haut niveau plutôt que par modules.

Dans les deux cas, Jenkins commence par présenter un aperçu des résultats de test pour chaque package. A partir d'ici, vous pouvez affiner, voir les résultats de test pour chaque classe de test pour finir par les tests dans les classes de test. Et s'il y a des tests en échec, ils seront surlignés en haut de la page.

Cette vue complète vous donne à la fois un bon aperçu de l'état courant de vos tests, et une indication sur leur historique. La colonne Age indique depuis combien de temps un test a été cassé, avec un hyperlien qui vous ramène vers le premier build dans lequel le test a échoué.

Vous pouvez aussi rajouter une description aux résultats de test, en utilisant le lien ajouter une description dans le coin en haut à droite de l'écran. C'est une bonne manière d'annoter un échec de build avec des détails additionnels, afin de rajouter une information supplémentaire sur l'origine du problème des échecs de test ou des notes sur la façon de les corriger.

Lorsqu'un test échoue, vous voulez généralement savoir pourquoi. Pour voir les détails d'un échec d'un test donné, cliquez sur le lien correspondant sur cet écran. Cela va afficher l'ensemble des détails, y compris le message d'erreur et la pile, ainsi qu'un rappel du temps depuis lequel le test est en échec (voir Figure 6.7, “Les détails d'un échec de test”). Vous devriez vous méfier des tests qui sont en échec depuis plus de deux builds — cela signale soit un problème technique pointu qui doit être investigué, soit une attitude complaisante envers les tests en échec (les développeurs ignorent peut-être les échecs de build), ce qui est plus sérieux et doit sûrement être étudié.

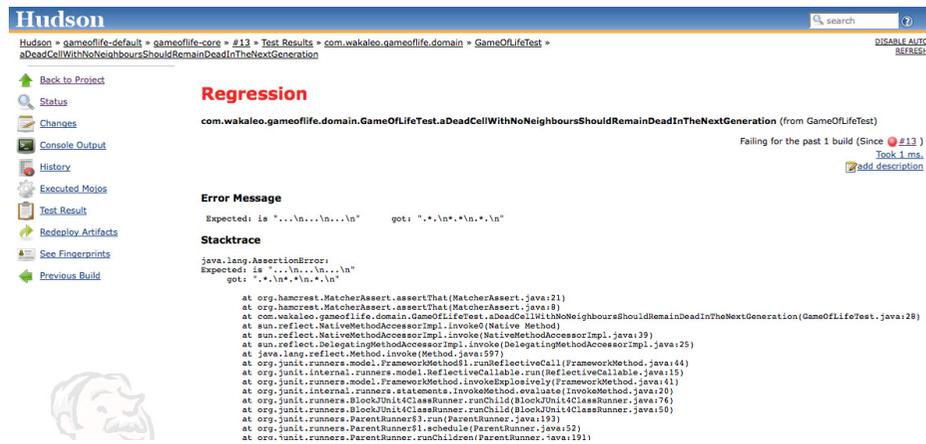


Figure 6.7. Les détails d'un échec de test

Assurez-vous que vous gardez un œil sur le temps que prennent vos tests pour s'exécuter, et pas uniquement s'ils passent ou échouent. Les tests unitaires doivent être conçus pour s'exécuter rapidement, et des tests trop longs peuvent être le signe d'un problème de performance. Des tests unitaires lents retardent aussi le retour, et en IC, un retour rapide est la clé. Par exemple, exécuter un millier de tests unitaires en cinq minutes est bon — prendre une heure ne l'est pas. Donc c'est une bonne idée de vérifier régulièrement combien de temps vos tests unitaires prennent pour s'exécuter, et si nécessaire investiguer pourquoi ils prennent autant de temps.

Heureusement, Jenkins peut facilement vous dire la durée que prennent vos tests pour s'exécuter dans le temps. Sur la page d'accueil de la tâche de build, cliquez sur le lien “tendance” dans la boîte Historique des builds sur la gauche de l'écran. Cela vous donnera un graphique à l'instar de celui dans Figure 6.8,

“Les tendances de temps de build peuvent vous donner un bon indicateur de la rapidité de vos tests”, vous montrant combien de temps chacun de vos builds a pris pour s'exécuter. Cependant, les tests ne sont pas la seule chose qui apparaît dans une tâche de build, mais si vous avez suffisamment de tests à regarder de près, ils vont probablement prendre une grande partie du temps. Ainsi, ce graphique est aussi un excellent moyen de voir comment vos tests se comportent.

Build Time Trend

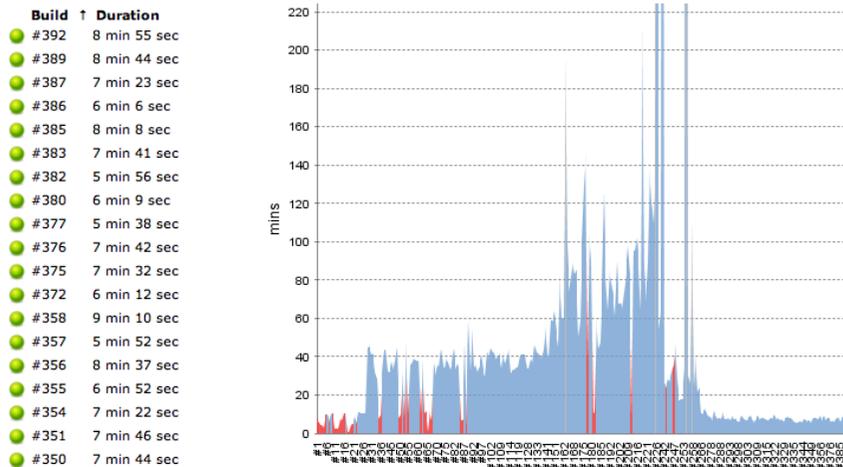


Figure 6.8. Les tendances de temps de build peuvent vous donner un bon indicateur de la rapidité de vos tests

Quand vous êtes sur la page Résultats des tests (voir Figure 6.6, “Jenkins affiche une vue synthétique des résultats de test”), vous pouvez aussi affiner et voir le temps que prennent les tests dans un module, package ou classe donnée. Cliquez sur la durée du test dans la page Résultats des tests (“A duré 31 ms” dans Figure 6.6, “Jenkins affiche une vue synthétique des résultats de test”) pour voir l'historique du test pour un package, une classe, ou un test individuel (voir Figure 6.9, “Jenkins vous permet de voir combien de temps les tests ont mis pour s'exécuter”). Cela rend facile l'isolation d'un test qui prend plus de temps qu'il ne le devrait, ou même décider quand une optimisation générale de vos tests unitaires est requise.

6.5. Ignorer des tests

Jenkins fait la distinction entre les tests échoués et les tests ignorés. Les tests ignorés sont ceux qui ont été désactivés, par exemple en utilisant l'annotation `@Ignore` dans JUnit 4:

```
@Ignore("Pending more details from the BA")
@Test
public void cashWithdrawalShouldDeductSumFromBalance() throws Exception {
    Account account = new Account();
    account.makeDeposit(100);
}
```

```
account.makeCashWithdraw(60);
assertThat(account.getBalance(), is(40));
}
```

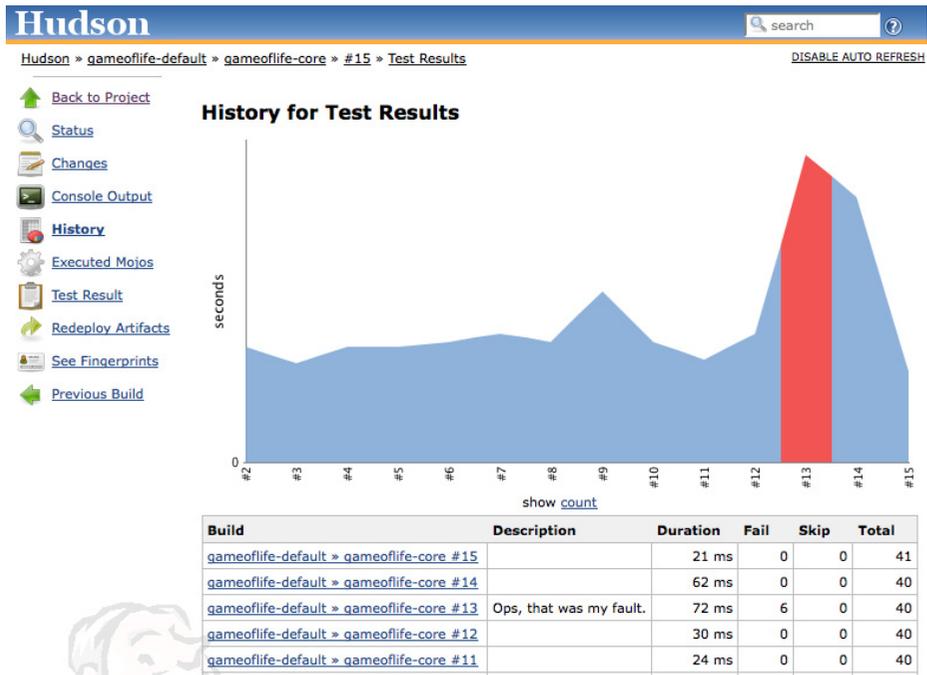


Figure 6.9. Jenkins vous permet de voir combien de temps les tests ont mis pour s'exécuter

Ignorer des tests est parfaitement légitime dans certaines circonstances, comme lors de la mise en place d'un test d'acceptation automatisé, ou un test technique de plus haut niveau, en attente pendant que vous implémentez les couches inférieures. Dans de tels cas, vous ne voulez pas être distrait par le test d'acceptation échoué, mais vous ne voulez pas non plus oublier que le test existe. Utiliser des techniques telles que l'annotation `@Ignore` est certainement meilleur que de commenter simplement le test ou de le renommer (dans JUnit 3), car cela permet à Jenkins de garder un oeil sur les tests ignorés pour vous.

Avec TestNG, vous pouvez aussi ignorer des tests, en utilisant la propriété `enabled`:

```
@Test(enabled=false)
public void cashWithdrawalShouldDeductSumFromBalance() throws Exception {
    Account account = new Account();
    account.makeDeposit(100);
    account.makeCashWithdraw(60);
    assertThat(account.getBalance(), is(40));
}
```

Avec TestNG, vous pouvez aussi définir des dépendances entre les tests, de façon à ce que certains tests s'exécuteront après qu'un autre test ou un groupe de tests se soit exécuté, comme illustré ici:

```

@Test
public void serverStartedOk() {...}

@Test(dependsOnMethods = { "serverStartedOk" })
public void whenAUserLogsOnWithACorrectUsernameAndPasswordTheHomePageIsDisplayed() {...}

```

Ici, si le premier test (`serverStartedOk()`) échoue, le test suivant sera ignoré.

Dans tous ces cas, Jenkins marquera les tests qui n'ont pas été exécutés en jaune, à la fois dans la tendance de résultats de test globale, et dans les détails du test (voir Figure 6.10, “Jenkins affiche les tests ignorés en jaune”). Les tests ignorés ne sont pas aussi mauvais que des tests échoués, mais il est important de ne pas avoir l'habitude de les négliger. Les tests ignorés sont comme des branches dans un système de gestion de version: un test doit être ignoré pour une raison particulière, avec une idée claire de la date à laquelle il sera réactivé. Un test ignoré qui reste ignoré pendant une période trop longue ne sent pas bon.

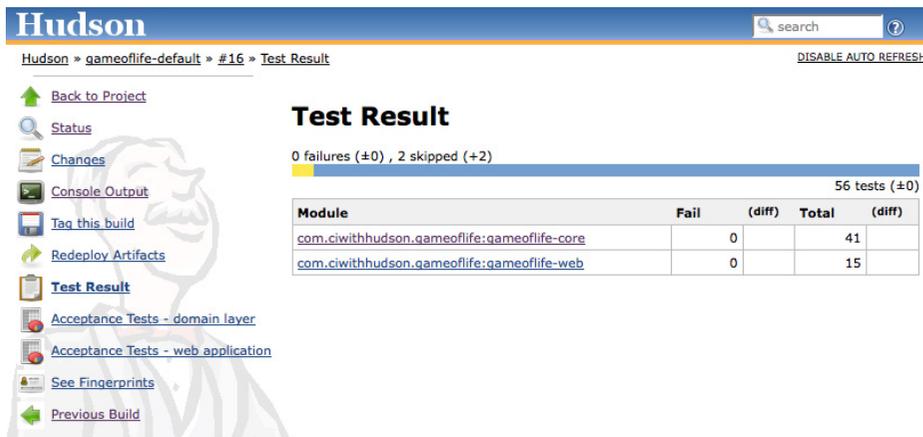


Figure 6.10. Jenkins affiche les tests ignorés en jaune

6.6. Couverture de code

Une autre métrique très utile liée aux tests est la couverture de code. La couverture de code donne une indication sur les parties de votre application qui ont été exécutées pendant les tests. Alors que ce n'est pas en soit une indication suffisante sur la qualité du test (il est facile d'exécuter une application entière sans réellement tester quoique ce soit, et les métriques de couverture de code ne fournissent pas une indication de la qualité ou de l'exactitude de vos tests), c'est une bonne indication du code qui n'a pas été testé. Et, si votre équipe est en train d'adopter des pratiques de test rigoureuses telles que Test-Driven-Development, la couverture de code peut être un bon indicateur sur la façon dont ces pratiques ont été mises en place.

L'analyse de la couverture de code est un traitement consommateur en CPU et mémoire, et va ralentir votre tâche de build de façon significative. Pour cette raison, vous allez généralement exécuter les

métriques de couverture de code dans une tâche de build Jenkins séparée, exécutée après que vos tests unitaires et d'intégration aient réussi.

Il y a de nombreux outils de couverture de code disponibles, et plusieurs sont supportés dans Jenkins, tous via des plugins dédiés. Les développeurs Java peuvent choisir entre Cobertura et Emma, deux outils populaires de couverture de code open source, ou Clover, un puissant outil commercial de couverture de code d'Atlassian. Pour les projets .NET, vous pouvez utiliser NCover.

Le fonctionnement et la configuration de tous ces outils sont semblables. Dans cette section, nous allons examiner Cobertura.

6.6.1. Mesurer la couverture de code avec Cobertura

Cobertura¹ est un outil de couverture de code open source pour Java et Groovy qui est simple d'utilisation et s'intègre parfaitement à la fois dans Maven et Jenkins.

Comme tous les plugins Jenkins de métriques de qualité de code,² le plugin Cobertura pour Jenkins ne va pas lancer les tests de couverture de code pour vous. C'est à vous de générer les informations de couverture de code dans le cadre de votre processus de build automatisé. Jenkins, d'autre part, fournit un excellent travail de rapport sur les métriques de couverture de code, notamment le suivi de la couverture de code dans le temps, et fournissant une couverture de code agrégée sur plusieurs modules applicatifs.

La couverture de code peut être une affaire complexe, et il est utile de comprendre le traitement que Cobertura effectue, surtout quand vous devez le mettre en place dans des outils de scripting de plus bas niveau comme Ant. L'analyse de la couverture de code fonctionne en trois étapes. D'abord, il modifie (ou "instrumente") les classes de votre application, afin qu'elles conservent le nombre de fois que chaque ligne de code a été exécutée.³ Il stocke ces informations dans un fichier spécial (Cobertura utilise un fichier nommé `cobertura.ser`).

Quand le code de l'application a été instrumenté, vous exécutez vos tests avec le code instrumenté. A la fin des tests, Cobertura aura généré un fichier de données indiquant pour chaque ligne le nombre de fois qu'elle a été exécutée au cours des tests.

Une fois que ce fichier a été généré, Cobertura peut utiliser cette donnée pour générer un rapport dans un format plus lisible, comme XML ou HTML.

6.6.1.1. Intégrer Cobertura avec Maven

Produire des métriques de couverture de code avec Cobertura dans Maven est relativement simple. Si vous êtes intéressés par la génération des données de couverture de code, il vous suffit de rajouter le **`cobertura-maven-plugin`** à la section build de votre fichier `pom.xml`:

¹ <http://cobertura.sourceforge.net>

² À l'exception notable de Sonar, que nous allons examiner plus tard dans le livre.

³ C'est en fait une légère simplification ; en fait, Cobertura stocke aussi d'autres informations, telles que le nombre de fois que chaque résultat possible d'un test booléen a été exécuté. Cependant, cela ne modifie pas l'approche générale.

```

<project>
  ...
  <build>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>cobertura-maven-plugin</artifactId>
        <version>2.5.1</version>
        <configuration>
          <formats>
            <format>html</format>
            <format>xml</format>
          </formats>
        </configuration>
      </plugin>
      ...
    </plugins>
  </build>
  ...
</project>

```

Cela va générer les métriques de couverture de code quand vous lancerez le plugin Cobertura directement :

```
$ mvn cobertura:cobertura
```

Les données de couverture de code seront générées dans le répertoire `target/site/cobertura`, dans un fichier nommé `coverage.xml`.

Cependant, cette approche va instrumenter vos classes et produire les données de couverture de code pour chaque build, ce qui est inefficace. Une meilleure approche est de placer cette configuration dans un profil spécifique, comme montré ici :

```

<project>
  ...
  <profiles>
    <profile>
      <id>metrics</id>
      <build>
        <plugins>
          <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>cobertura-maven-plugin</artifactId>
            <version>2.5.1</version>
            <configuration>
              <formats>
                <format>html</format>
                <format>xml</format>
              </formats>
            </configuration>
          </plugin>
        </plugins>
      </build>
    </profile>
  </profiles>

```

```
</profile>
...
</profiles>
</project>
```

Dans ce cas, vous lancerez le plugin Cobertura en utilisant le profil `metrics` pour générer les données de couverture de code :

```
$ mvn cobertura:cobertura -Pmetrics
```

Une autre approche consiste à inclure les rapports de couverture de code dans vos rapports Maven. Cette approche est beaucoup plus lente et plus consommatrice en mémoire que de générer simplement les données de couverture de code, mais cela peut avoir du sens si vous générez aussi d'autres métriques de qualité de code et leurs rapports en même temps. Si vous voulez le faire avec Maven 2, vous devez aussi inclure le plugin Maven Cobertura plugin dans la section `reporting`, comme montré ici :

```
<project>
...
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>cobertura-maven-plugin</artifactId>
      <version>2.5.1</version>
      <configuration>
        <formats>
          <format>html</format>
          <format>xml</format>
        </formats>
      </configuration>
    </plugin>
  </plugins>
</reporting>
</project>
```

A présent, les données de couverture de code seront générées quand vous générerez le site Maven pour ce projet :

```
$ mvn site
```

Si votre projet Maven contient des modules (comme il est de pratique courante pour des gros projets Maven), vous devez mettre en place la configuration Cobertura dans le fichier `pom.xml` parent. Les métriques de couverture de code et le rapport seront générés séparément pour chaque module. Si vous utilisez l'option de configuration `aggregate`, le plugin Maven Cobertura générera aussi un rapport de plus haut niveau combinant les données de couverture de code de tous les modules. Cependant, que vous utilisiez cette option ou non, le plugin Jenkins Cobertura va lire les données de couverture de code de plusieurs fichiers et les combiner dans un seul rapport agrégé.

Au moment de la rédaction, il y a une limitation avec le plugin Maven Cobertura — la couverture de code sera uniquement enregistrée pour les tests exécutés pendant la phase `test`, et pas les tests exécutés

pendant la phase `integration-test`. Cela peut être un problème si vous utilisez cette phase pour lancer des tests d'intégration ou des tests web qui nécessitent une application complètement packagée et déployée — dans ce cas, la couverture de code des tests qui sont uniquement exécutés pendant la phase `integration-test` ne sera pas comptée dans les métriques de couverture de code Cobertura.

6.6.1.2. Intégrer Cobertura avec Ant

Intégrer Cobertura dans votre build Ant est un peu plus compliqué que de le faire avec Maven. Cependant, cela vous permet d'avoir un contrôle plus fin sur les classes qui sont instrumentées, et quand la couverture de code est mesurée.

Cobertura est livré avec une tâche Ant que vous pouvez utiliser pour intégrer Cobertura dans vos builds Ant. Vous devez télécharger la dernière distribution Cobertura, et la décompresser quelque part sur votre disque dur. Afin que votre build soit plus portable, et donc plus facile à déployer dans Jenkins, c'est une bonne idée de placer la distribution Cobertura que vous utilisez dans votre répertoire projet, et de la sauvegarder dans votre système de gestion de version. Ainsi, c'est le moyen le plus facile pour garantir que le build utilisera la même version de Cobertura quelle que soit la façon dont il est exécuté.

En supposant que vous avez téléchargé la dernière installation de Cobertura et que vous l'avez placée à l'intérieur de votre projet dans un répertoire nommé `tools`, vous pourriez faire quelque chose comme ceci :

```
<property name="cobertura.dir" value="/jenkins-guide-complet/hudsonbook-content-fr/tools/cobertura" />
<path id="cobertura.classpath">②
  <fileset dir="${cobertura.dir}">
    <include name="cobertura.jar" />③
    <include name="lib/**/*.jar" />④
  </fileset>
</path>
<taskdef classpathref="cobertura.classpath" resource="tasks.properties" />
```

- ❶ Indique à Ant où se trouve l'installation de Cobertura.
- ❷ Nous devons mettre en place un classpath que Cobertura utilisera pour s'exécuter.
- ❸ Le chemin contient l'application Cobertura elle-même.
- ❹ Et toutes ses dépendances.

Ensuite, vous devez instrumenter les classes de l'application. Vous devez faire attention de placer ces classes instrumentées dans un répertoire différent, de telle manière qu'elles ne seront pas déployées en production par accident :

```
<target name="instrument" depends="init,compile">❶
  <delete file="cobertura.ser"/>②
  <delete dir="${instrumented.dir}" />③
  <cobertura-instrument todir="${instrumented.dir}">④
```

```

<fileset dir="${classes.dir}">
  <include name="**/*.class" />
  <exclude name="**/*Test.class" />
</fileset>
</cobertura-instrument>
</target>

```

- ❶ Nous ne pouvons instrumenter les classes de l'application qu'une fois qu'elles ont été compilées.
- ❷ Détruit toutes les données de couverture de code générées par les builds précédents.
- ❸ Détruit toutes les classes précédemment instrumentées.
- ❹ Instrumente les classes de l'application (mais pas les classes des tests) et les place dans le répertoire `${instrumented.dir}`.

À ce stade, le répertoire `${instrumented.dir}` contient une version instrumentée de nos classes applicatives. Maintenant, tout ce que nous devons faire pour générer des données utiles de couverture de code est d'exécuter nos tests unitaires avec les classes de ce répertoire :

```

<target name="test-coverage" depends="instrument">
  <junit fork="yes" dir="/jenkins-guide-complet/hudsonbook-content-fr">❶
    <classpath location="${instrumented.dir}" />
    <classpath location="${classes.dir}" />
    <classpath refid="cobertura.classpath" />❷

    <formatter type="xml" />
    <test name="${testcase}" todir="${reports.xml.dir}" if="testcase" />
    <batchtest todir="${reports.xml.dir}" unless="testcase">
      <fileset dir="${src.dir}">
        <include name="**/*Test.java" />
      </fileset>
    </batchtest>
  </junit>
</target>

```

- ❶ Exécute les tests JUnit avec les classes instrumentées.
- ❷ Les classes instrumentées utilisent les classes de Cobertura, donc les bibliothèques Cobertura doivent aussi se trouver dans le classpath.

Cela va produire les données brutes de couverture de code dont nous avons besoin pour produire les rapports XML de couverture que Jenkins peut utiliser. Pour produire ces rapports, nous devons lancer une autre tâche, comme montré ici :

```

<target name="coverage-report" depends="test-coverage">
  <cobertura-report srcdir="${src.dir}" destdir="${coverage.xml.dir}"
    format="xml" />
</target>

```

Enfin, n'oubliez pas de faire le ménage une fois que tout est fini : la cible **clean** ne doit pas détruire uniquement les classes générées, mais aussi les classes générées instrumentées, les données de couverture de code Cobertura, et les rapports Cobertura :

```

<target name="clean"
  description="Remove all files created by the build/test process.">
  <delete dir="${classes.dir}" />
  <delete dir="${instrumented.dir}" />
  <delete dir="${reports.dir}" />
  <delete file="cobertura.log" />
  <delete file="cobertura.ser" />
</target>

```

Une fois cela fait, vous êtes prêt à intégrer vos rapports de couverture de code dans Jenkins.

6.6.1.3. Installer le plugin de couverture de code Cobertura

Une fois que les données de couverture de code sont générées dans le cadre de votre processus de build, vous pouvez configurer Jenkins pour les afficher. Ceci implique l'installation du plugin Jenkins Cobertura. Nous avons décrit ce processus dans Section 2.8, “Adding Code Coverage and Other Metrics”, mais nous allons le décrire à nouveau pour rafraîchir votre mémoire. Allez sur l'écran Administrer Jenkins, et cliquez sur Gestion des plugins. Cela va vous amener à l'écran du gestionnaire des plugins. Si Cobertura n'a pas été installé, vous trouverez le plugin Cobertura dans l'onglet Disponibles, dans la section Build Reports (voir Figure 6.11, “Installer le plugin Cobertura”). Pour l'installer, cochez simplement la case et appuyez sur Entrée (ou faites défiler jusqu'au bas de l'écran et cliquez sur le bouton “Installer”). Jenkins va télécharger et installer le plugin pour vous. Une fois que le téléchargement est fini, vous devez redémarrer votre serveur Jenkins.

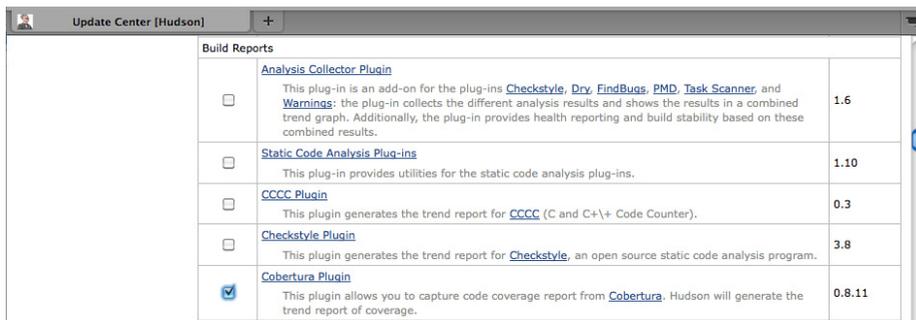


Figure 6.11. Installer le plugin Cobertura

6.6.1.4. Les rapports de couverture de code dans votre build

Une fois que vous avez installé le plugin, vous pouvez mettre en place les rapports de couverture de code dans vos tâches de build. Puisque la couverture de code peut être lente et consommatrice en mémoire, vous devrez généralement créer une tâche de build séparée pour cela et les autres métriques de qualité de code, qui sera exécutée après les tests unitaires et d'intégration. Pour de gros projets, vous pouvez même le mettre en place via un build qui s'exécute sur une base quotidienne. En effet, le retour sur les métriques de couverture de code et autres n'est généralement pas aussi critique que le retour sur les résultats de test, et cela va libérer des exécuteurs de build pour des tâches de build qui peuvent bénéficier de retour rapide.

Comme nous le mentionnions précédemment, Jenkins ne fait pas d'analyse de couverture par lui-même — vous devez configurer votre build pour produire le fichier Cobertura `coverage.xml` (ou fichiers) avant que vous puissiez générer de jolis graphes ou rapports, généralement en utilisant une des techniques dont nous avons discuté précédemment (voir Figure 6.12, “Votre build de couverture de code doit produire les données de couverture de code”).

Build

Root POM

Goals and options

Figure 6.12. Votre build de couverture de code doit produire les données de couverture de code

Une fois que vous avez configuré votre build pour produire des données de couverture de code, vous pouvez configurer Cobertura dans la section “Actions à la suite du build” de votre tâche de build. Si vous cochez la case “Publish Cobertura Coverage Report”, vous devriez voir quelque chose comme Figure 6.13, “Configurer les métriques de couverture de code dans Jenkins”.

Publish Cobertura Coverage Report

Cobertura xml report pattern

This is a file name pattern that can be used to locate the cobertura xml report files (for example with Maven2 use `**/target/site/cobertura/coverage.xml`). The path is relative to the module root unless you have configured your SCM with multiple modules, in which case it is relative to the workspace root. Note that the module root is SCM-specific, and may not be the same as the workspace root. Cobertura must be configured to generate XML reports for this plugin to function.

Consider only stable builds

Include only stable builds, i.e. exclude unstable and failed ones.

Coverage Metric Targets

Conditionals	Delete	98	75	75
Lines	Delete	98	75	75
Methods	Delete	100	80	80
Packages	Delete	100	95	95

Configure health reporting thresholds.
For the row, leave blank to use the default value (i.e. 80).
For the and rows, leave blank to use the default values (i.e. 0).

Figure 6.13. Configurer les métriques de couverture de code dans Jenkins

Le premier et plus important champ ici est le chemin des données XML Cobertura que nous avons générées. Votre projet peut contenir un seul fichier `coverage.xml`, ou plusieurs. Si vous avez un projet Maven multi-modules, par exemple, le plugin Maven Cobertura va générer un fichier `coverage.xml` distinct pour chaque module.

Le chemin accepte des caractères génériques du style Ant, et il est donc facile d'inclure les données de couverture de code à partir de plusieurs fichiers. Pour tout projet Maven, un chemin tel que `**/`

target/site/cobertura/coverage.xml inclura toutes les métriques de couverture de code de tous les modules dans le projet.

Il y a en fait plusieurs types de couverture de code, et il est parfois utile de les distinguer. La plus intuitive est la couverture de lignes, qui compte le nombre de fois qu'une ligne donnée est exécutée pendant les tests automatisés. "Conditional Coverage" (aussi appelé "Branch Coverage") prend en compte si les expressions booléennes dans les instructions `if` et semblables sont testées d'une manière qui vérifie tous les résultats possibles d'une expression conditionnelle. Par exemple, étant donné le fragment de code suivant:

```
if (price > 10000) {
    managerApprovalRequired = true;
}
```

Pour obtenir une couverture de code complète pour ce code, vous devez l'exécuter deux fois : une fois avec une valeur supérieure à 10,000, une autre avec une valeur inférieure ou égale à 10,000.

D'autres métriques de couverture de code plus basiques incluent les méthodes (combien de méthodes dans l'application ont été exécutées par les tests), classes et packages.

Jenkins vous permet de définir lesquelles de ces métriques vous voulez suivre. Par défaut, le plugin Cobertura enregistre les couvertures conditionnelles, lignes, et méthodes, ce qui est généralement suffisant. Cependant, il est facile de rajouter d'autres métriques de couverture de code si vous pensez que ce peut être utile pour votre équipe.

Le traitement par Jenkins des métriques de couverture de code n'est pas uniquement un affichage passif — Jenkins vous permet de définir comment ces métriques affectent le résultat du build. Vous pouvez définir des valeurs de palier pour les métriques de couverture de code qui affectent à la fois le résultat du build et le rapport météo sur le tableau de bord Jenkins (voir Figure 6.14, "Les résultats des tests de couverture de code contribuent à l'état du projet sur le tableau de bord"). Chaque métrique de couverture de code que vous suivez possède trois valeurs de palier.

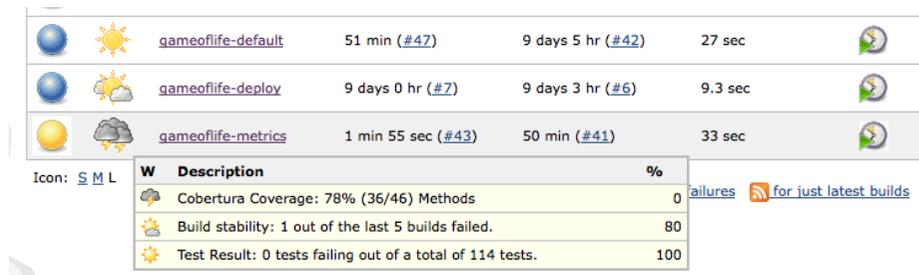


Figure 6.14. Les résultats des tests de couverture de code contribuent à l'état du projet sur le tableau de bord

Le premier (celui avec une icône ensoleillée) est la valeur minimale que le build doit atteindre pour avoir une icône ensoleillée. La seconde indique la valeur en dessous de laquelle le build aura une icône

orageuse. Jenkins va extrapoler les valeurs intermédiaires entre ces deux-ci pour d'autres icônes météo plus nuancées.

Le dernier palier est simplement la valeur en dessous de laquelle un build sera marqué comme “instable” — avec une balle jaune. Bien que n'étant pas aussi mauvais qu'une balle rouge (pour un build cassé), une balle jaune entrainera un message de notification et apparaîtra comme mauvais sur le tableau de bord.

Cette fonctionnalité n'est pas un simple détail esthétique — elle fournit un moyen précieux de fixer des objectifs de qualité de code pour vos projets. Bien qu'elle ne puisse pas être interprétée seule, une mauvaise couverture de code n'est généralement pas un bon signe pour un projet. Donc si vous prenez la couverture de code au sérieux, utilisez ces valeurs de palier pour fournir un retour direct quand les choses ne sont pas à la hauteur.

6.6.1.5. Interpréter les métriques de couverture de code

Jenkins affiche vos rapports de couverture de code sur la page d'accueil de la tâche de build. La première fois qu'il est exécuté, il produit un simple graphique à barres (voir Figure 2.30, “Jenkins displays code coverage metrics on the build home page”). A partir du second build, un graphique est affiché, indiquant les différents types de couverture que vous suivez dans le temps (voir Figure 6.15, “Configurer les métriques de couverture de code dans Jenkins”). Dans les deux cas, le graphique affichera aussi les métriques de couverture de code pour le dernier build.

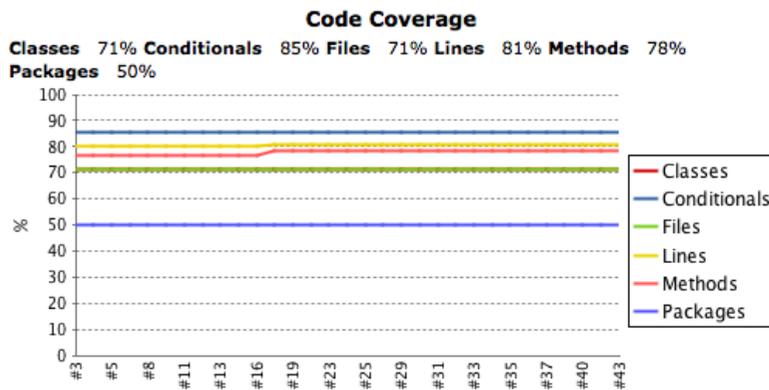


Figure 6.15. Configurer les métriques de couverture de code dans Jenkins

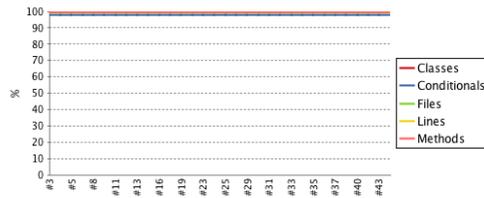
Jenkins offre une grande valeur ajoutée en vous permettant de descendre dans les métriques de couverture de code, affichant les erreurs de couverture de code pour les packages, classes à l'intérieur d'un package, et les lignes de code à l'intérieur d'une classe (voir Figure 6.16, “Afficher les métriques de couverture de code”). Quel que soit le niveau de détail que vous regardiez, Jenkins affichera un graphe en haut de la page montrant la tendance de couverture dans le temps. Plus bas, vous trouverez la répartition par package ou classe.

Code Coverage

[Cobertura Coverage Report](#) >

com.wakaleo.gameoflife.domain

Trend



Package Coverage summary

Name	Classes	Conditionals	Files	Lines	Methods
com.wakaleo.gameoflife.domain	100% 5/5	98% 53/54	100% 5/5	100% 108/108	100% 36/36

Coverage Breakdown by File

Name	Classes	Conditionals	Lines	Methods
Grid.java	100% 1/1	100% 30/30	100% 40/40	100% 16/16
Cell.java	100% 1/1	75% 3/4	100% 14/14	100% 5/5
GridWriter.java	100% 1/1	100% 4/4	100% 7/7	100% 2/2
Universe.java	100% 1/1	100% 12/12	100% 34/34	100% 9/9
GridReader.java	100% 1/1	100% 4/4	100% 13/13	100% 4/4

Figure 6.16. Afficher les métriques de couverture de code

Lorsque vous arrivez au niveau de détail de la classe, Jenkins affiche aussi le code source de la classe, avec les lignes colorées en fonction de leur niveau de couverture. Les lignes qui ont été complètement exécutées pendant les tests sont en vert, et les lignes qui n'ont jamais été exécutées sont marquées en rouge. Un nombre dans la marge indique le nombre de fois que la ligne en question a été exécutée. Enfin, un ombrage jaune dans la marge est utilisée pour indiquée une couverture conditionnelle insuffisante (par exemple, une instruction `if` qui a seulement été testé avec un résultat).

6.6.2. Mesurer la couverture de code avec Clover

Clover est un excellent outil de couverture de code commercial d'Atlassian⁴. Clover fonctionne parfaitement pour des projets Ant, Maven, ou même Grails. La configuration et l'utilisation de Clover est bien documentée sur le site web d'Atlassian, donc nous ne regarderons pas ces aspects en détail. Cependant, en guise d'exemple, voici une configuration Maven typique de Clover pour une utilisation avec Jenkins :

```
<build>
...
<plugins>
...
<plugin>
<groupId>com.atlassian.maven.plugins</groupId>
<artifactId>maven-clover2-plugin</artifactId>
<version>3.0.4</version>
```

⁴ <http://www.atlassian.com/software/clover>

```

<configuration>
  <includesTestSourceRoots>false</includesTestSourceRoots>
  <generateXml>true</generateXml>
</configuration>
</plugin>
</plugins>
</build>
...

```

Cela va générer à la fois des rapports de couverture de code HTML et XML, y compris les données agrégées si le projet Maven contient plusieurs modules.

Pour intégrer Clover dans Jenkins, vous devez installer le plugin Jenkins Clover selon la manière habituelle en utilisant l'écran du gestionnaire de plugins. Une fois que vous avez redémarré Jenkins, vous pourrez intégrer les données de couverture de code Clover dans vos builds.

Exécuter Clover sur votre projet est un projet en plusieurs étapes : vous instrumentez votre code applicatif, exécutez les tests, agrégez les données de test (pour les projets Maven à plusieurs modules) et générez les rapports HTML et XML. Puisque cela peut être une opération assez lente, vous allez généralement l'exécuter dans une tâche de build séparée, et pas avec vos test classiques. Vous pouvez le faire comme suit :

```
$ clover2:setup test clover2:aggregate clover2:clover
```

Ensuite, vous devez mettre en place les rapports Clover dans Jenkins. Cochez la case Publish Clover Coverage Report pour l'activer. La configuration est similaire à celle de Cobertura — vous devez fournir le chemin du répertoire du rapport HTML Clover, et du fichier rapport XML, et vous pouvez aussi définir des valeurs de palier pour les icônes météo ensoleillées et orageuses, et pour les builds instables (voir Figure 6.17, “Configurer les rapports Clover dans Jenkins”).

Publish Clover Coverage Report

Clover report directory

Specify the path to the directory that contains the clover.xml report file, relative to the [workspace root](#). Clover must be configured to generate XML reports for this plugin to function fully.

Clover report file name

Specify the name of the Clover xml file generated relative to the Clover report directory specified above. If not specified - 'clover.xml' is assumed.

Coverage Metric Targets	% Methods	% Conditionals	% Statements
	<input type="text" value="100"/>	<input type="text" value="100"/>	<input type="text" value="100"/>
	<input type="text" value="80"/>	<input type="text" value="80"/>	<input type="text" value="80"/>
	<input type="text" value="75"/>	<input type="text" value="75"/>	<input type="text" value="75"/>

Configure health reporting thresholds.
For the  row, leave blank to use the default values (i.e. 70, 80, and 80 for methods, conditionals and statements respectively).
For the  and  rows, leave blank to use the default values (i.e. 0).

Figure 6.17. Configurer les rapports Clover dans Jenkins

Une fois que vous l'aurez fait, Jenkins affichera le niveau actuel de couverture de code, ainsi qu'un graphe de la couverture de code dans le temps, sur la page d'accueil de votre tâche de build de votre projet (voir Figure 6.18, “Tendance de couverture de code Clover”).

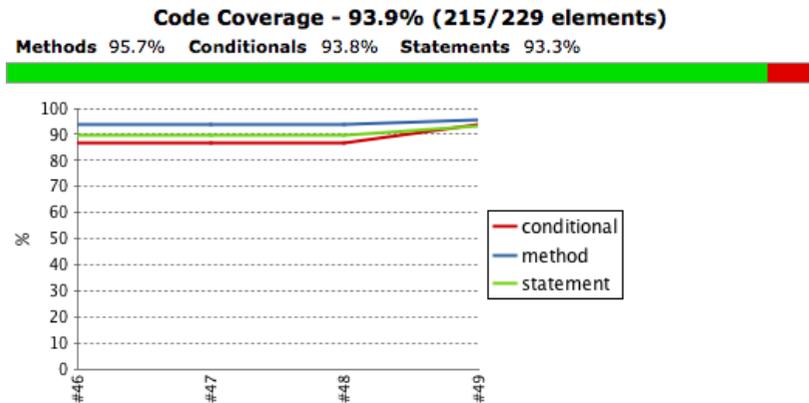


Figure 6.18. Tendence de couverture de code Clover

6.7. Tests d'acceptation automatisés

Les tests d'acceptation automatisés jouent un rôle important dans de nombreux projets agiles, à la fois pour la vérification et pour la communication. Comme outil de vérification, les tests d'acceptation jouent un rôle similaire aux tests d'intégration, et visent à démontrer que l'application fait effectivement ce qui est attendu d'elle. Mais ce n'est qu'un aspect secondaire des tests d'acceptation automatisés. L'objectif principal est en fait la communication — montrer aux non développeurs (experts métier, analystes métiers, testeurs, et ainsi de suite) précisément où en est le projet.

Les tests d'acceptation ne doivent pas être mélangés avec des tests orientés développeurs, parce qu'à la fois leur finalité et leur audience sont très différentes. Les tests d'acceptation doivent être des exemples montrant comment le système fonctionne, avec un accent sur la démonstration plutôt que sur une preuve exhaustive. Les tests exhaustifs doivent être faits au niveau des tests unitaires.

Les tests d'acceptation peuvent être automatisés en utilisant des outils conventionnels tels que JUnit, mais il y a une tendance croissante à utiliser des frameworks Behavior-Driven Development (BDD) à cet effet, parce qu'ils correspondent mieux à l'audience non technique des tests d'acceptation. Les outils Behavior-driven development utilisés pour les tests d'acceptation automatisés génèrent généralement des rapports HTML avec une mise en page bien adaptée aux non développeurs. Ils produisent aussi souvent des rapports compatibles JUnit qui peuvent être traités directement par Jenkins.

Les frameworks Behavior-Driven Development ont aussi la notion de “tests en attente”, tests qui sont automatisés, mais qui n'ont pas encore été implémentés par l'équipe de développement. Cette distinction joue un rôle important dans la communication avec les autres acteurs non développeurs : si vous pouvez automatiser ces tests tôt dans le processus, ils vous donneront un excellent indicateur des fonctionnalités qui ont été implémentées, qui fonctionnent, ou qui n'ont pas encore été démarrées.

En règle générale, vos tests d'acceptation doivent être affichés séparément des autres tests automatisés plus conventionnels. S'ils utilisent le même framework de test que vos tests classiques (e.g., JUnit),

assurez-vous qu'ils sont exécutés dans une tâche de build séparée, de telle façon que les non-développeurs peuvent les visualiser et se concentrer sur les tests orientés métier sans être distraits par ceux de plus bas niveau ou techniques. Il peut être aussi utile d'adopter des conventions de nommage orientées métier et fonctionnelles pour vos tests et classes de test, afin de les rendre plus accessible aux non-développeurs (voir Figure 6.19, "Utilisation de conventions de nommage orientées métier pour des tests JUnit"). La façon dont vous nommez vos tests et classes de test peut faire une réelle différence quant à la lecture des rapports de test et la compréhension des fonctionnalités et comportements métier qui sont testés.

Test Result : WhenTheUserEntersAnInitialGrid

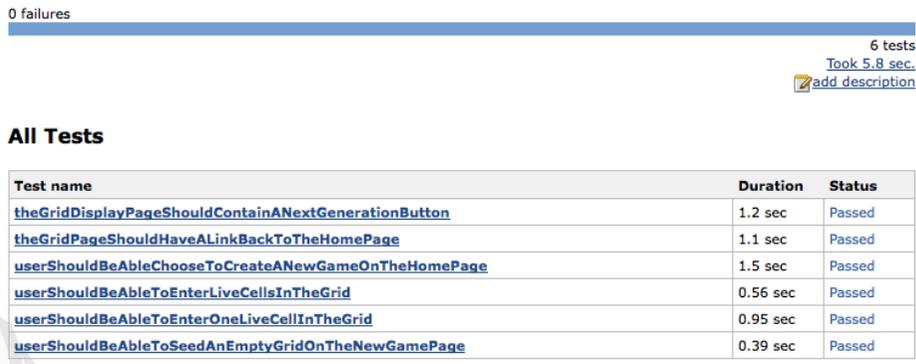


Figure 6.19. Utilisation de conventions de nommage orientées métier pour des tests JUnit

Si vous utilisez un outil qui génère des rapports HTML, vous pouvez les afficher dans le même build que vos tests classiques, tant qu'ils apparaissent dans un rapport séparé. Jenkins fournit un plugin très pratique pour ce genre de rapport HTML, appelé le plugin HTML Publisher (voir Figure 6.20, "Installer le plugin HTML Publisher"). Bien que ce soit à vous de vous assurer que votre build produise les bons rapports, Jenkins peut les afficher sur la page de votre tâche de build, les rendant facilement accessible à tous les membres de l'équipe.

<input checked="" type="checkbox"/>	Deploy to container plugin This plugin allows you to deploy a war to a container after a successful build.	1.5
<input checked="" type="checkbox"/>	Hudson description setter plugin	1.6
<input checked="" type="checkbox"/>	HTML Publisher plugin This plugin publishes HTML reports.	0.4
<input checked="" type="checkbox"/>	M2 Release Plugin A plug-in that enables you to perform releases using the maven-release-plugin from Hudson.	0.4.0

Figure 6.20. Installer le plugin HTML Publisher

Ce plugin est facile à configurer. Allez jusqu'à la section "Actions à la suite du build" et cochez la case "Publish HTML reports" (voir Figure 6.21, "Publier les rapports HTML"). Ensuite, indiquez à Jenkins le répertoire où ont été générés vos rapports HTML, une page d'index, et un titre pour votre rapport.

Vous pouvez aussi demander à Jenkins de stocker les rapports générés pour chaque build, ou de ne conserver que le dernier.



Figure 6.21. Publier les rapports HTML

Une fois que cela est fait, Jenkins affichera une icône spéciale sur votre page d'accueil de votre tâche de build, avec un lien vers votre rapport HTML. Sur Figure 6.22, “Jenkins affiche un lien spécial sur la page d'accueil de la tâche de build pour votre rapport”, vous pouvez voir les rapports easyb que nous avons configuré précédemment en action.

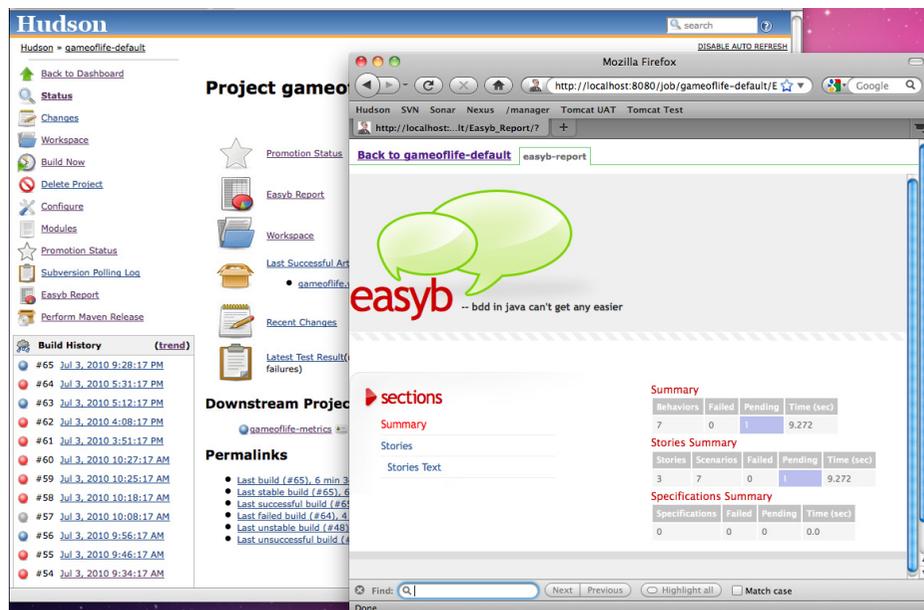


Figure 6.22. Jenkins affiche un lien spécial sur la page d'accueil de la tâche de build pour votre rapport

Le plugin HTML Publisher fonctionne parfaitement pour les rapports HTML. Si, d'autre part, vous voulez (aussi) publier des documents non-HTML, tels que des fichiers texte, PDFs, et ainsi de suite, alors le plugin DocLinks est fait pour vous. Ce plugin est semblable au plugin HTML Publisher, mais il vous permet d'archiver à la fois des rapports HTML aussi bien que des documents dans d'autres formats. Par exemple, dans Figure 6.23, “Le plugin DocLinks vous permet d'archiver des documents HTML et non-HTML”, nous avons configuré une tâche de build qui archive à la fois un document PDF et un rapport HTML. Ces deux documents seront maintenant listés sur la page d'accueil de la tâche de build.

Publish documents

Documents

Title

Description

Directory to archive
 archive recursively
Directory relative to the root of the workspace, such as 'myproject/build/javadoc'. If "archive recursively" checked, the entire directory structure is archived.

Index file
Specify the file to display. If no value is set, then 'index.html' is used.

Title

Description

Directory to archive
 archive recursively
Directory relative to the root of the workspace, such as 'myproject/build/javadoc'. If "archive recursively" checked, the entire directory structure is archived.

Index file
Specify the file to display. If no value is set, then 'index.html' is used.

Figure 6.23. Le plugin DocLinks vous permet d'archiver des documents HTML et non-HTML

6.8. Tests de performance automatisés avec JMeter

La performance applicative est un autre domaine de test important. Les tests de performance peuvent être utilisés pour vérifier beaucoup de choses, telles que la rapidité avec laquelle une application répond aux requêtes d'un nombre donné d'utilisateurs simultanés, ou comment une application réagit à un nombre croissant d'utilisateurs. De nombreuses applications ont des contrats de niveau de service (SLAs), qui définissent contractuellement comment elles doivent réagir.

Les tests de performance sont souvent une activité unique, prise en compte uniquement juste à la fin du projet ou quand les choses commencent à aller mal. Néanmoins, les problèmes de performance sont comme n'importe quelle autre sorte de bug : plus tard ils sont détectés dans le processus, plus coûteux ils sont à corriger. Il est donc logique d'automatiser ces tests de performance et de charge, de façon à pouvoir repérer les zones de dégradation des performances avant qu'elles ne sortent dans la nature.

JMeter⁵ est un outil open source de test de performance et de charge. Il fonctionne en simulant la charge sur votre application, et en mesurant le temps de réponse alors que le nombre d'utilisateurs simulés et les requêtes augmentent. Il simule les actions d'un navigateur ou une application cliente, envoyant des requêtes de toutes sortes (HTTP, SOAP, JDBC, JMS, etc) vers votre serveur. Vous configurez un ensemble de requêtes à envoyer à votre application, ainsi que des pauses aléatoires, conditions et boucles, et d'autres variantes destinées à mieux imiter les actions utilisateurs réelles.

JMeter s'exécute comme une application Swing, dans laquelle vous pouvez configurer vos scripts de test (voir Figure 6.24, "Préparer un script de test de performance dans JMeter"). Vous pouvez même exécuter JMeter comme proxy, et utiliser votre application dans un navigateur traditionnel pour préparer une version initiale de votre script de test.

⁵ <http://jakarta.apache.org/jmeter/>

Un tutoriel complet sur l'utilisation de JMeter sort du cadre de ce livre. Cependant, il est assez facile à apprendre, et vous pouvez trouver de nombreux détails sur son utilisation sur le site de JMeter. Avec un peu de travail, vous pouvez avoir un script de test respectable et l'exécuter en quelques heures.

Ce qui nous intéresse ici est le processus d'automatisation de ces tests de performance. Il y a plusieurs façons pour intégrer des tests JMeter dans votre processus de build Jenkins. Bien qu'à l'écriture de ces lignes, il n'y ait pas de plugin officiel JMeter pour Maven disponible dans les dépôts Maven, il y a un plugin Ant. Donc, la méthode la plus simple est d'écrire un script Ant pour exécuter vos tests de performance, et ensuite soit d'appeler ce script Ant directement, soit (si vous utilisez un projet Maven, et voulez exécuter JMeter via Maven) d'utiliser l'intégration Ant Maven pour lancer le script Ant à partir de Maven. Un simple script Ant exécutant quelques tests JMeter est illustré ici :

```
<project default="jmeter">
  <path id="jmeter.lib.path">
    <pathelement location="/jenkins-guide-complet/hudsonbook-content-fr/tools/jmeter/extras/ant-jmeter" />
  </path>

  <taskdef name="jmeter"
    classname="org.programmerplanet.ant.taskdefs.jmeter.JMeterTask"
    classpathref="jmeter.lib.path" />

  <target name="jmeter">
    <jmeter jmeterhome="/jenkins-guide-complet/hudsonbook-content-fr/tools/jmeter"
      testplan="/jenkins-guide-complet/hudsonbook-content-fr/src/test/jmeter/gameoflife.jmx"
      resultlog="/jenkins-guide-complet/hudsonbook-content-fr/target/jmeter-results.jtl">
      <jvmarg value="-Xmx512m" />
    </jmeter>
  </target>
</project>
```

Cela suppose que l'installation de JMeter est disponible dans le répertoire `tools` de votre projet. Placer des outils tels que JMeter au sein de votre structure de projet est une bonne habitude, car il rend vos scripts de build plus portables et plus faciles à exécuter sur n'importe quelle machine, ce qui est précisément ce dont nous avons besoin pour les exécuter sur Jenkins.

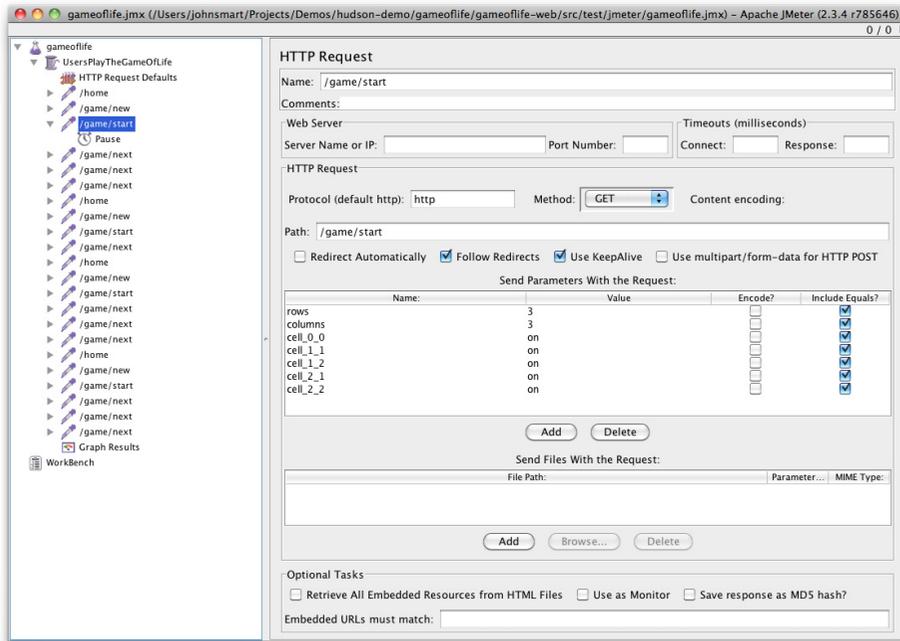


Figure 6.24. Préparer un script de test de performance dans JMeter

Notons que nous utilisons aussi le tag `<jvmarg>` optionnel pour fournir à JMeter une quantité de mémoire suffisante — les tests de performance sont une activité consommatrice de mémoire.

Le script affiché ici exécutera les tests de performance JMeter sur une application lancée. Donc vous devez vous assurer que l'application que vous voulez tester est active et lancée avant que vous lanciez vos tests. Il y a plusieurs façons de le faire. Pour des tests de performance plus lourds, vous voudrez généralement déployer votre application sur un serveur de test avant de lancer les tests. Pour la plupart des applications ce n'est généralement pas trop difficile — le plugin Maven Cargo, par exemple, vous permet d'automatiser le processus de déploiement sur une variété de serveurs locaux et distants. Nous verrons aussi comment le faire dans Jenkins plus loin dans le livre.

Sinon, si vous utilisez Maven pour une application Web, vous pouvez utiliser le plugin Jetty ou Cargo pour vous assurer que l'application est déployée avant le démarrage des tests d'intégration, et ensuite appeler le script Ant JMeter à partir de Maven pendant la phase de test d'intégration. Avec Jetty, par exemple, vous pouvez faire quelque chose comme cela :

```
<project...>
  <build>
    <plugins>
      <plugin>
        <groupId>org.mortbay.jetty</groupId>
        <artifactId>jetty-maven-plugin</artifactId>
        <version>7.1.0.v20100505</version>
        <configuration>
```

```

<scanIntervalSeconds>10</scanIntervalSeconds>
<connectors>
  <connector
    implementation="org.eclipse.jetty.server.nio.SelectChannelConnector">
      <port>${jetty.port}</port>
      <maxIdleTime>60000</maxIdleTime>
    </connector>
  </connectors>
<stopKey>foo</stopKey>
<stopPort>9999</stopPort>
</configuration>
<executions>
  <execution>
    <id>start-jetty</id>
    <phase>pre-integration-test</phase>
    <goals>
      <goal>run</goal>
    </goals>
    <configuration>
      <scanIntervalSeconds>0</scanIntervalSeconds>
      <daemon>true</daemon>
    </configuration>
  </execution>
  <execution>
    <id>stop-jetty</id>
    <phase>post-integration-test</phase>
    <goals>
      <goal>stop</goal>
    </goals>
  </execution>
</executions>
</plugin>
...
</plugins>
</build>
</project>

```

Cela va démarrer une instance de Jetty et y déployer votre application web avant les tests d'intégration, et l'arrêter après.

Enfin, vous devez exécuter vos tests de performance JMeter pendant cette phase. Vous pouvez le faire en utilisant le maven-antrun-plugin pour lancer le script Ant que nous avons écrit précédemment pendant la phase integration-test :

```

<project...>
  ...
  <profiles>
    <profile>
      <id>performance</id>
      <build>
        <plugins>
          <plugin>
            <artifactId>maven-antrun-plugin</artifactId>
            <version>1.4</version>

```

```

    <executions>
      <execution>
        <id>run-jmeter</id>
        <phase>integration-test</phase>
        <goals>
          <goal>run</goal>
        </goals>
        <configuration>
          <tasks>
            <ant antfile="build.xml" target="jmeter" >
          </tasks>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>
</build>
</profile>
</profiles>
...
</project>

```

Maintenant, tout ce que vous devez faire est de lancer les tests d'intégration avec le profil performance pour que Maven exécute la suite de test JMeter. Vous pouvez le faire en invoquant les phases de cycle de vie Maven **integration-test** ou **verify**:

```
$ mvn verify -Pperformance
```

Une fois que vous avez configuré votre script de build pour gérer JMeter, vous pouvez mettre en place un build de tests de performance dans Jenkins. Pour cela, nous allons utiliser le plugin Jenkins Performance Test, qui interprète les logs JMeter et peut générer des statistiques et des jolis graphes en utilisant ces données. Donc, allez sur l'écran du gestionnaire des plugins sur votre serveur Jenkins et installez ce plugin (voir Figure 6.25, "Préparer un script de tests de performance dans JMeter"). Quand vous aurez installé ce plugin, vous devrez redémarrer Jenkins.

<input type="checkbox"/>	NCover Plugin Archive and publish .NET code coverage HTML reports from NCover .	0.3
<input type="checkbox"/>	JUnit Plugin This plugin allows you to publish JUnit test results.	0.10
<input checked="" type="checkbox"/>	Performance Plugin This plugin allows you to capture reports from JMeter and JUnit . Hudson will generate graphic charts with the trend report of performance and robustness. It includes the feature of setting the final build status as good, unstable or failed, based on the reported error percentage.	1.2
<input type="checkbox"/>	PerfPublisher Plugin This plugin generates global and trend reports for tests results analysis. Based on an open XML tests results format, the plugin parses the generated files and publish statistics, reports and analysis on the current health of the project.	7.97

Figure 6.25. Préparer un script de tests de performance dans JMeter

Une fois que le plugin est installé, vous pouvez mettre en place une tâche de build de performance dans Jenkins. Cette tâche de build sera généralement séparée des autres builds. Dans Figure 6.26, "Mise en

place du build de performance pour s'exécuter chaque nuit à minuit”, nous avons mis en place un build de performance fonctionnant sur une base quotidienne, ce qui est probablement assez pour des tests de robustesse ou de performance.

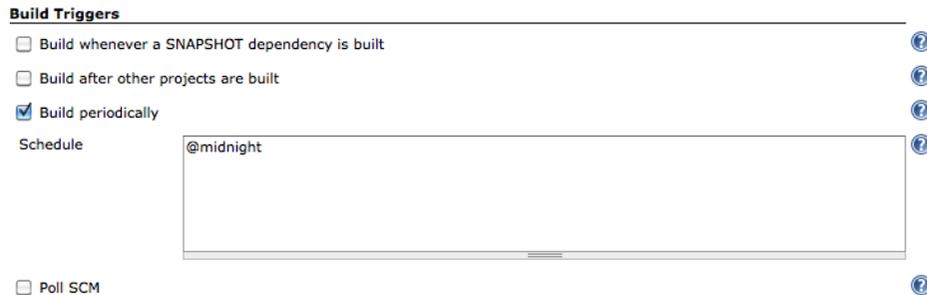


Figure 6.26. Mise en place du build de performance pour s'exécuter chaque nuit à minuit

Il ne reste alors plus qu'à configurer votre tâche de build pour qu'elle exécute vos tests de performance. Dans Figure 6.27, “Les tests de performance peuvent demander de grandes quantités de mémoire”, nous exécutons le build Maven que nous avons configuré précédemment. Notez que nous utilisons le champ MAVEN_OPTS (accessible en cliquant sur le bouton Avancé) pour fournir suffisamment de mémoire à la tâche de build.



Figure 6.27. Les tests de performance peuvent demander de grandes quantités de mémoire

Pour mettre en place des rapports de performance, sélectionnez simplement l'option “Publish Performance test result report” dans la section Actions à la suite du build (voir Figure 6.28, “Configurer le plugin Performance dans votre tâche de build”). Vous devez indiquer à Jenkins où se trouvent les résultats de test JMeter (les fichiers de sortie, pas les scripts de test). Le plugin Performance peut traiter plusieurs résultats JMeter, donc vous pouvez mettre des caractères génériques dans le chemin pour vous assurer que tous vos rapports JMeter seront affichés.

Si vous prenez vos mesures de performance au sérieux, alors le build doit échouer si les niveaux de service attendus ne sont pas atteints. Dans un environnement d'intégration continue, toute mesure qui n'échoue pas si un critère de qualité minimum n'est pas atteint a tendance à être ignorée.

Vous pouvez configurer le plugin Performance afin de marquer un build instable ou échoué si un certain pourcentage des requêtes finissent en erreur. Par défaut, ces valeurs ne seront atteintes que dans les cas d'erreurs applicatives (i.e., bugs) ou plantages de serveur. Toutefois, vous devriez vraiment configurer vos scripts de test JMeter pour placer une limite sur le maximum acceptable pour le temps

de réponse pour vos requêtes. Ceci est particulièrement important si votre application a des obligations contractuelles à cet égard. Une façon de le faire avec JMeter est d'ajouter un élément Duration Assertion à votre script. Cela va provoquer une erreur si une requête prend plus d'un certain temps pour s'exécuter.

Publish Performance test result report ?

Performance report

Specify the path to the Performance report files, relative to the [workspace root](#).
 - If you left this field blank the plugin will look for files matching the pattern: `**/*.jtl` in the workspace.
 - Or you can enclose the search specifying a list of files and folders separated by semicolon.
 - Or use an Ant Fileset pattern.

Performance threshold ● Unstable ● Failed

Thresholds: % %

Specify the error percentage threshold that set the build unstable or failed (a value of 0 means: dont use this threshold).

Figure 6.28. Configurer le plugin Performance dans votre tâche de build

À présent, lorsque la tâche de build s'exécute, le plugin Performance va produire des graphes permettant de suivre les temps de réponse globaux et le nombre d'erreurs (voir Figure 6.29, "Le plugin Jenkins Performance garde une trace des temps de réponse et des erreurs"). Il y aura un graphique séparé pour chaque rapport JMeter que vous avez généré. S'il n'y a qu'un seul graphique, il sera aussi affiché sur la page d'accueil du build ; sinon vous pouvez les visualiser sur une page dédiée à laquelle vous pouvez accéder via le menu Performance Trend.

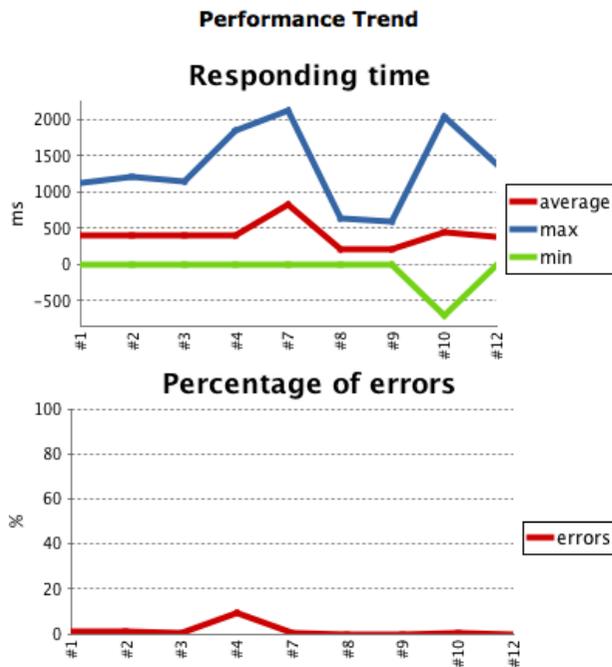


Figure 6.29. Le plugin Jenkins Performance garde une trace des temps de réponse et des erreurs

Ce graphe vous donne un aperçu de la performance dans le temps. Vous utilisez généralement ce graphe pour vous assurer que les temps de réponse moyens sont dans la limite prévue, et aussi repérer des variations anormales des temps de réponse moyens ou maximums. Cependant, si vous avez besoin de suivre et d'isoler des problèmes de performance, l'écran Performance Breakdown peut être plus utile. A partir du rapport Performance Trend, cliquez sur le lien Last Report en haut de l'écran. Cela fera apparaître une répartition des temps de réponse et des erreurs par demande (voir Figure 6.30, "Vous pouvez aussi visualiser les résultats de performance par requête"). Vous pouvez faire la même chose pour les builds précédents, en cliquant sur le lien Performance Report sur la page de détails du build.

Avec quelques variations mineures, un script de test JMeter travaille essentiellement en simulant un nombre donné d'utilisateurs simultanés. Généralement, cependant, vous voudrez voir comment votre application gère des nombres différents d'utilisateurs. Le plugin Jenkins Performance le prend en compte assez bien, et peut traiter des graphes de plusieurs rapports JMeter. Assurez-vous juste que vous utilisez une expression générique quand vous dites à Jenkins où se trouvent les rapports.

Bien sûr, il serait agréable de pouvoir réutiliser le même script de test JMeter pour chaque test exécuté. JMeter supporte les paramètres, donc vous pouvez facilement réutiliser le même script JMeter avec un nombre différent d'utilisateurs simulés. Utilisez simplement une expression de propriété dans votre script JMeter, et passez cette propriété à JMeter quand vous lancez le script. Si votre propriété est nommée `request.threads`, alors l'expression de propriété dans votre script JMeter sera `${__property(request.threads)}`. Ensuite, vous pouvez utiliser l'élément `<property>` dans la tâche Ant `<jmeter>` pour passer la propriété quand vous exécutez le script. La cible Ant suivante, par exemple, exécute JMeter trois fois, pour 200, 500 et 1000 utilisateurs simultanés:

```
<target name="jmeter">
  <jmeter jmeterhome="/jenkins-guide-complet/hudsonbook-content-fr/tools/jmeter"
    testplan="/jenkins-guide-complet/hudsonbook-content-fr/src/test/jmeter/gameoflife.jmx"
    resultlog="/jenkins-guide-complet/hudsonbook-content-fr/target/jmeter-results-200-users" />
  <jvmarg value="-Xmx512m" />
  <property name="request.threads" value="200"/>
  <property name="request.loop" value="20"/>
</jmeter>
<jmeter jmeterhome="/jenkins-guide-complet/hudsonbook-content-fr/tools/jmeter"
  testplan="/jenkins-guide-complet/hudsonbook-content-fr/src/test/jmeter/gameoflife.jmx"
  resultlog="/jenkins-guide-complet/hudsonbook-content-fr/target/jmeter-results-500-users" />
<jvmarg value="-Xmx512m" />
<property name="request.threads" value="500"/>
<property name="request.loop" value="20"/>
</jmeter>
<jmeter jmeterhome="/jenkins-guide-complet/hudsonbook-content-fr/tools/jmeter"
  testplan="/jenkins-guide-complet/hudsonbook-content-fr/src/test/jmeter/gameoflife.jmx"
  resultlog="/jenkins-guide-complet/hudsonbook-content-fr/target/jmeter-results-1000-users" />
<jvmarg value="-Xmx512m" />
<property name="request.threads" value="1000"/>
<property name="request.loop" value="20"/>
</jmeter>
</target>
```

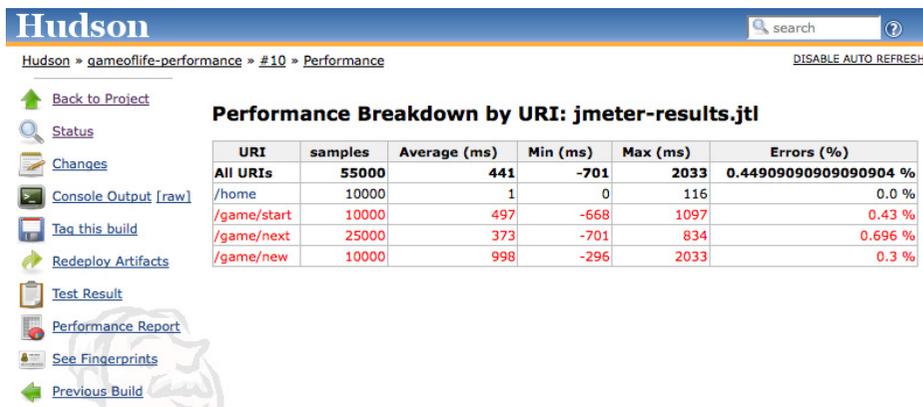


Figure 6.30. Vous pouvez aussi visualiser les résultats de performance par requête

6.9. A l'aide ! Mes tests sont trop lents !

Un des principes fondamentaux de la conception de vos builds d'intégration continue est que la valeur de l'information d'un échec de build diminue rapidement avec le temps. En d'autres termes, plus le retour d'un échec de build met de temps à vous atteindre, moins il vaut la peine, et plus dur il est à corriger.

En effet, si vos tests fonctionnels ou d'intégration prennent plusieurs heures pour s'exécuter, il y a des chances qu'ils ne seront pas exécutés à chaque changement. Ils sont plus susceptibles d'être programmés comme un build quotidien. Le problème est alors que beaucoup de choses peuvent intervenir en vingt-quatre heures, et, si le build quotidien échoue, il sera difficile de déterminer lequel des nombreux changements committés sur le contrôle de version pendant la journée était responsable. C'est un problème sérieux, et pénalise la capacité de votre serveur d'intégration continue de fournir un retour rapide qui le rend utile.

Bien sûr, certains builds sont lents, par nature. Les tests de performance ou de charge rentrent dans cette catégorie, comme d'autres builds lourds de métriques de qualité de code pour de gros projets. Cependant, les tests d'intégration et fonctionnels ne rentrent pas dans cette catégorie. Vous devez faire tout ce que vous pouvez pour rendre ces tests aussi rapides que possible. Moins de dix minutes est probablement acceptable pour une suite complète d'intégration/fonctionnel. Deux heures ne l'est pas.

Donc, si vous vous trouvez vous-même à devoir accélérer vos tests, voici quelques stratégies qui seront utiles, par ordre approximatif de difficulté.

6.9.1. Ajouter plus de matériel

Quelquefois la façon la plus facile pour accélérer vos builds est de rajouter du matériel. Cela pourrait être aussi simple que de mettre à jour votre serveur de build. Comparé au temps et effort passés à identifier et corriger les bugs liés à l'intégration, le coût d'achat d'un nouveau serveur flambant neuf est relativement faible.

Une autre option est d'envisager l'utilisation d'approches virtuelles ou basées sur le cloud. Plus tard dans le livre, nous verrons comment vous pouvez utiliser des machines virtuelles VMWare ou une infrastructure cloud telles que Amazon Web Services (EC2) ou CloudBees pour augmenter votre capacité de build sur une base “à la demande”, sans avoir à investir dans de nouvelles machines permanentes.

Cette approche peut également impliquer la distribution de vos builds sur plusieurs serveurs. Bien que cela ne va pas en tant que tel accélérer vos tests, cela peut mener à un retour plus rapide si votre serveur de build est sous forte demande, et si vos tâches de build sont constamment en attente.

6.9.2. Lancer moins de tests d'intégration/fonctionnels

Dans de nombreuses applications, les tests d'intégration ou fonctionnels sont utilisés par défaut comme moyen standard pour tester presque tous les aspects du système. Cependant, les tests d'intégration et fonctionnels ne sont pas le meilleur moyen de détecter et d'identifier les bugs. En raison du grand nombre de composants impliqués dans un test typique de bout en bout, il peut être très difficile de savoir où s'est produit l'erreur. En outre, avec autant de pièces changeant, il est extrêmement difficile, si ce n'est totalement irréalisable, de couvrir l'ensemble des chemins possibles à travers l'application.

Pour cette raison, dans la mesure du possible, vous devriez préférer les tests unitaires rapides aux tests d'intégration et fonctionnels beaucoup plus lents. Quand vous êtes confiants dans le fait que les composants individuels fonctionnent correctement, vous pouvez compléter le tableau par quelques tests de bout en bout qui passent par des cas d'utilisation communs du système, ou des cas d'utilisation qui ont causé des problèmes par le passé. Cela va garantir que les composants s'emboîtent correctement, ce qui est, après tout, ce que les tests d'intégration sont supposés faire. Mais préférez, dans la mesure du possible, les tests unitaires aux tests plus complets. Cette stratégie est probablement l'approche la plus viable pour maintenir votre retour rapide, mais elle nécessite une certaine discipline et des efforts.

6.9.3. Exécutez vos tests en parallèle

Si vos tests fonctionnels prennent deux heures pour s'exécuter, il est peu probable qu'ils aient tous besoin de s'exécuter à la suite. Il est aussi peu probable qu'ils consomment toute la CPU disponible sur votre machine de build. Alors découper vos tests d'intégration en petits lots et les exécuter en parallèle a beaucoup de sens.

Il y a plusieurs stratégies que vous pouvez essayer, et votre choix va probablement dépendre de la nature de votre application. Une approche, par exemple, est de mettre en place plusieurs tâches de build pour exécuter des sous ensembles différents de vos tests fonctionnels, et de lancer ces tâches en parallèle. Jenkins vous permet d'agrèger les résultats de test. C'est une bonne façon de tirer avantage de l'architecture de build distribuée pour accélérer vos builds encore plus. Le point essentiel de cette stratégie est la capacité d'exécuter des sous-ensembles de vos tests en isolation, ce qui peut demander une certaine restructuration.

A un plus bas niveau, vous pouvez aussi exécuter vos tests en parallèle au niveau des scripts de build. Comme nous avons vu précédemment, TestNG et les versions les plus récentes de JUnit supportent tous

les deux l'exécution de tests en parallèle. Néanmoins, vous devrez vous assurer que vos tests peuvent être exécutés simultanément, ce qui peut nécessiter une restructuration. Par exemple, les fichiers communs ou des variables d'instance partagées vont dans ce cas causer des problèmes.

En général, vous devez être prudent sur les interactions entre vos tests. Si vos tests web démarrent un serveur Web intégré, comme Jetty, par exemple, vous devez vous assurer que le port utilisé est différent pour chaque ensemble de tests simultanés.

Néanmoins, si vous pouvez le faire fonctionner pour votre application, exécuter vos tests en parallèle est un des moyens les plus efficaces pour accélérer vos tests.

6.10. Conclusion

L'automatisation des tests est un élément essentiel de tout environnement d'intégration continue, et doit être pris très au sérieux. Comme dans d'autres domaines dans l'intégration continue, et peut-être plus encore ici, le retour d'information est roi, il est donc important de s'assurer que vos tests s'exécutent rapidement, y compris les tests d'intégration et fonctionnels.

Chapter 7. Sécuriser Jenkins

7.1. Introduction

Jenkins supporte plusieurs modèles de sécurité, et peut s'intégrer avec différents gestionnaires d'utilisateurs. Dans les petites organisations, où les développeurs travaillent proches les uns des autres, la sécurité de votre machine Jenkins n'est peut-être pas un gros problème — vous pourriez juste vouloir éviter que des utilisateurs non identifiés n'altèrent vos configurations de tâches de build. Pour de plus importantes organisations, avec de multiples équipes, une approche plus stricte pourrait être nécessaire, dans laquelle seuls les membres de l'équipe et les administrateurs systèmes ont les droits pour modifier la configuration des tâches de build. Et dans des situations où Jenkins serait exposé à une audience plus large, comme un site web interne d'une entreprise, ou même sur Internet, certaines tâches de build pourraient être visibles à tous les utilisateurs alors que d'autres nécessiteraient d'être cachées aux utilisateurs non autorisés.

Dans ce chapitre, nous regarderons comment configurer différentes configurations de sécurité dans Jenkins, pour différents environnements et circonstances.

7.2. Activer la sécurité dans Jenkins

Configurer une sécurité basique dans Jenkins est assez simple. Allez sur la page de configuration principale et sélectionnez la case à cocher Activer la sécurité (voir Figure 7.1, “Activer la sécurité dans Jenkins”). Ceci affiche plusieurs options que nous allons expliquer en détails dans ce chapitre. La première section, Domaine de sécurité, définit où Jenkins regardera pendant l'authentification, et inclut des options telles que l'utilisation d'utilisateurs stockés dans un serveur LDAP, en utilisant le compte utilisateur Unix sous-jacent (en supposant, bien sûr, que Jenkins est exécuté sur une machine Unix), ou utilisant une simple base de données embarquée gérée par Jenkins.

La seconde section, Autorisations, détermine ce que les utilisateurs peuvent faire une fois qu'ils sont connectés. Cela va de simples options comme "Tout le monde a accès à toutes les fonctionnalités" ou "Les utilisateurs connectés peuvent tout faire" à des rôles plus sophistiqués ou des politiques d'autorisations par projet.

Enable security ?

TCP port for JNLP slave agents Fixed : Random Disable ?

Markup Formatter ?

Treat the text as HTML and use it as is without any translation

Access Control

Security Realm

Delegate to servlet container ?

Jenkins's own user database ?

Allow users to sign up ?

LDAP ?

Unix user/group database ?

Authorization

Anyone can do anything ?

Legacy mode ?

Logged-in users can do anything ?

Matrix-based security ?

Project-based Matrix Authorization Strategy ?

Figure 7.1. Activer la sécurité dans Jenkins

À la fin de ce chapitre, nous regarderons comment configurer la sécurité Jenkins pour un certain nombre de scénarii courants.

7.3. Sécurité simple dans Jenkins

Le modèle de sécurité le plus simple disponible dans Jenkins permet aux utilisateurs authentifiés de faire tout ce qu'ils veulent, alors que les utilisateurs non authentifiés auront seulement une vue en lecture seule de leurs tâches de build. C'est super pour les petites équipes - les développeurs peuvent gérer les tâches de build, alors que les autres utilisateurs (testeur, analyste métier, responsable de projet, etc.) peuvent accéder aux tâches de build pour voir l'état du projet. En effet, certaines tâches pourraient même être configurées uniquement dans ce but, affichant les résultats de tests d'acceptation automatisés ou des métriques de qualité de code, par exemple.

Vous pouvez mettre en place ce type de configuration en choisissant “Les utilisateurs connectés peuvent tout faire” dans la section Autorisations. Il y a plusieurs façons dans Jenkins pour authentifier les utilisateurs (voir Section 7.4, “Domaines de sécurité — Identifier les utilisateurs Jenkins”), mais pour cet exemple, nous allons utiliser l'option la plus simple, qui est d'utiliser la base de données intégrée à Jenkins (voir Section 7.4.1, “Utiliser la base de données intégrée à Jenkins”). C'est la configuration illustrée dans Figure 7.1, “Activer la sécurité dans Jenkins”.

Assurez-vous de cocher l'option "Autoriser les utilisateurs à se connecter". Cette option affichera un lien Se connecter en haut de l'écran permettant aux utilisateurs de créer leurs propres comptes (voir Figure 7.2, “La page de connexion Jenkins”). C'est une bonne idée pour les développeurs d'utiliser ici leur nom d'utilisateur de gestionnaire de sources : dans ce cas, Jenkins sera capable de retrouver quels utilisateurs ont contribué aux changements qui ont déclenché un build particulier.



Figure 7.2. La page de connexion Jenkins

Cette approche est évidemment un peu trop simple pour beaucoup de situations — il est utile pour les petites équipes de travailler en forte proximité, où le but est de connaître le changement de qui a déclenché (ou cassé) un build particulier, plutôt que de gérer l'accès de façon plus restrictive. Dans les sections suivantes, nous discuterons de deux aspects orthogonaux de la sécurité Jenkins : identifier vos utilisateurs (domaines de sécurité) et déterminer ce qu'ils sont autorisés à faire (Autorisation).

7.4. Domaines de sécurité — Identifier les utilisateurs Jenkins

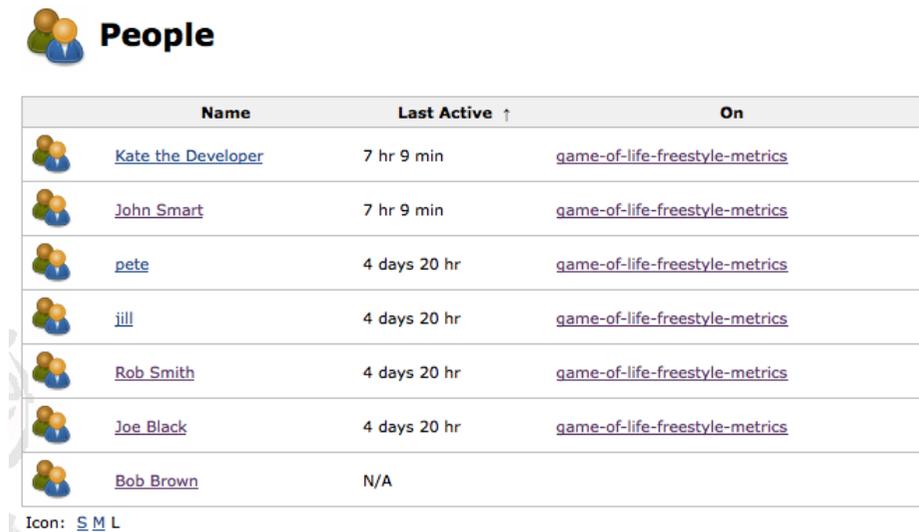
Jenkins vous permet d'identifier et de gérer les utilisateurs de plusieurs façons, depuis une simple base de données intégrée pour les petites équipes jusqu'à l'intégration avec des annuaires d'entreprise, avec de nombreuses autres options entre les deux.

7.4.1. Utiliser la base de données intégrée à Jenkins

Le moyen le plus simple pour gérer des comptes utilisateurs dans Jenkins est d'utiliser la base de données interne de Jenkins. C'est une bonne option si vous voulez garder les choses simples, car peu de configuration est nécessaire. Les utilisateurs qui ont besoin de se connecter au serveur Jenkins peuvent s'enregistrer et créer un compte par eux-mêmes, et, en fonction du modèle de sécurité choisi, un administrateur peut ensuite décider ce que ces utilisateurs sont autorisés à faire.

Jenkins ajoute automatiquement tout utilisateur de gestionnaire de sources à cette base de données dès qu'un changement est effectué dans le code source surveillé par Jenkins. Ces noms d'utilisateurs sont utilisés principalement pour enregistrer le responsable de chaque tâche de build. Vous pouvez voir la liste des utilisateurs actuellement connus en cliquant sur l'entrée de menu Personnes (voir Figure 7.3, “La liste des utilisateurs connus de Jenkins”). Ici, vous pouvez visualiser les utilisateurs que Jenkins connaît actuellement, et aussi voir le dernier projet dans lequel ils ont committé. Notez que cette liste

contient la liste de tous les utilisateurs à avoir jamais committé dans les projets que Jenkins surveille — ils pourraient ne pas être (et en général ne sont pas) tous des utilisateurs actifs de Jenkins capables de se connecter sur le serveur Jenkins.



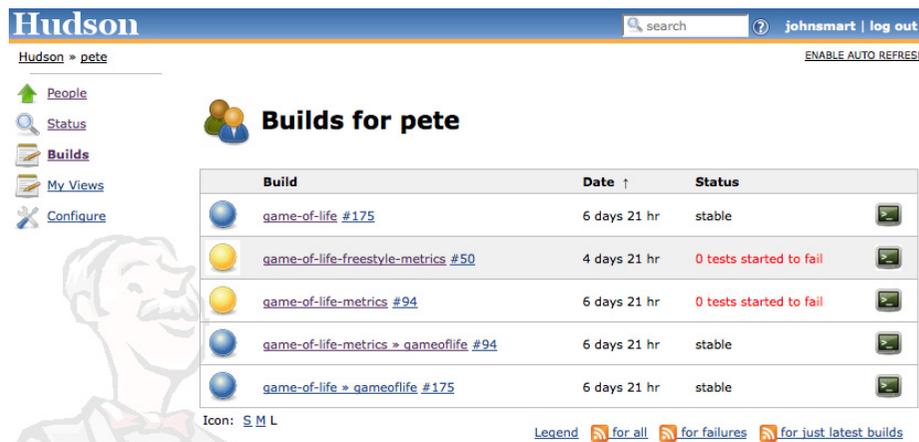
People

Name	Last Active ↑	On
 Kate the Developer	7 hr 9 min	game-of-life-freestyle-metrics
 John Smart	7 hr 9 min	game-of-life-freestyle-metrics
 pete	4 days 20 hr	game-of-life-freestyle-metrics
 jill	4 days 20 hr	game-of-life-freestyle-metrics
 Rob Smith	4 days 20 hr	game-of-life-freestyle-metrics
 Joe Black	4 days 20 hr	game-of-life-freestyle-metrics
 Bob Brown	N/A	

Icon: [S](#) [M](#) [L](#)

Figure 7.3. La liste des utilisateurs connus de Jenkins

Si vous cliquez sur un utilisateur de cette liste, Jenkins vous emmène sur une page affichant différentes informations à propos de cet utilisateur, incluant son nom complet et les tâches de build auxquelles il a contribué (voir Figure 7.4, “Afficher les builds auxquels un utilisateur participe”). De là, vous pouvez aussi modifier ou compléter les détails à propos de cet utilisateur, comme son mot de passe ou son adresse email.



Hudson johnsmart | log out

Hudson » [pete](#) ENABLE AUTO REFRESH

[People](#) [Status](#) [Builds](#) [My Views](#) [Configure](#)

Builds for pete

Build	Date ↑	Status
 game-of-life #175	6 days 21 hr	stable
 game-of-life-freestyle-metrics #50	4 days 21 hr	0 tests started to fail
 game-of-life-metrics #94	6 days 21 hr	0 tests started to fail
 game-of-life-metrics » gameoflife #94	6 days 21 hr	stable
 game-of-life » gameoflife #175	6 days 21 hr	stable

Icon: [S](#) [M](#) [L](#) [Legend](#) [for all](#) [for failures](#) [for just latest builds](#)

Figure 7.4. Afficher les builds auxquels un utilisateur participe

Un utilisateur apparaissant sur cette liste ne peut pas nécessairement se connecter à Jenkins. Pour pouvoir se connecter, l'utilisateur doit avoir un mot de passe configuré. Il y a essentiellement deux façons de faire cela. Si vous avez configuré l'option "Autoriser les utilisateurs à s'enregistrer", les utilisateurs peuvent simplement se connecter avec leur nom d'utilisateur SCM et fournir leur adresse email et leur mot de passe (voir Section 7.3, "Sécurité simple dans Jenkins"). Autrement, vous pouvez activer un utilisateur en cliquant sur l'option de menu Configurer dans l'écran de détails utilisateur, et fournir une adresse email et un mot de passe vous-même (voir Figure 7.5, "Créer un nouveau compte utilisateur en s'enregistrant").

The screenshot shows the Jenkins user configuration interface for a user named 'John Smart'. The page has a blue header with the Jenkins logo and a search bar. On the left, there is a navigation menu with options: People, Status, Builds, My Views, and Configure. The main content area contains several sections:

- Your name:** A text input field containing 'John Smart'.
- Description:** A large text area for providing a description.
- Password:** Two fields for 'Password' and 'Confirm Password', both masked with dots.
- E-mail:** An 'E-mail address' field containing 'john.smart@wakaleo.com'. Below it is a note: 'Your e-mail address, like joe.chin@sun.com'.
- My Views:** A 'Default View' dropdown menu set to 'All'. Below it is a note: 'The view selected by default when navigating to the users private views'.

 A 'Save' button is located at the bottom left of the configuration area. A cartoon character of a man with a bow tie is partially visible on the left side of the page.

Figure 7.5. Créer un nouveau compte utilisateur en s'enregistrant

Il est utile de noter que, si vos adresses email sont synchronisées avec vos noms d'utilisateurs de contrôle de version (par exemple, si vous travaillez chez acme.com, et que l'utilisateur "joe" dans votre système de contrôle de version a une adresse email joe@acme.com), vous pouvez faire que Jenkins dérive l'adresse email de l'utilisateur en ajoutant un suffixe que vous configurez dans la section Notification Email (voir Figure 7.6, "Synchroniser les adresses email"). Si vous avez effectué ce type de configuration, vous n'avez pas besoin de spécifier l'adresse email pour les nouveaux utilisateurs à moins qu'elle ne respecte pas cette convention.

The screenshot shows the 'E-mail Notification' configuration page in Jenkins. It contains the following fields:

- SMTP server:** An empty text input field.
- Default user e-mail suffix:** A text input field containing '@acme.com'.
- System Admin E-mail Address:** A text input field containing 'sysadmin@acme.com'.
- Hudson URL:** A text input field containing 'http://hudson.acme.com'.

 Each field has a help icon (question mark) to its right. At the bottom right, there is an 'Advanced...' button. Below the fields, there is a button that says 'Test configuration by sending e-mail to System Admin Address'.

Figure 7.6. Synchroniser les adresses email

Une autre façon de gérer les utilisateurs courants actifs (ceux qui peuvent vraiment se connecter à Jenkins) s'effectue via le lien Gérer les utilisateurs sur la page de configuration principale de Jenkins (voir Figure 7.7, “Vous pouvez aussi gérer les utilisateurs Jenkins depuis la page de configuration Jenkins”).

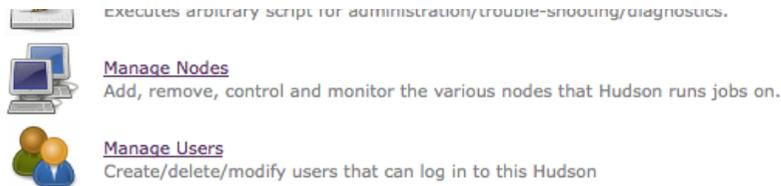


Figure 7.7. Vous pouvez aussi gérer les utilisateurs Jenkins depuis la page de configuration Jenkins

D'ici, vous pouvez voir et éditer les utilisateurs qui peuvent se connecter à Jenkins (voir Figure 7.8, “La base de données des utilisateurs de Jenkins”). Cela inclut à la fois les utilisateurs qui se sont enregistrés manuellement (si cette option a été activée) et les utilisateurs SCM que vous avez activés en leur configurant un mot de passe. Vous pouvez aussi éditer des informations utilisateur (par exemple, modifier leur adresse email ou réinitialiser leur mot de passe), ou même les supprimer de la liste des utilisateurs actifs. Procéder ainsi ne les enlèvera pas de la liste globale des utilisateurs (leurs noms apparaîtront toujours dans l'historique de build, par exemple), mais ils ne seront plus capables de se connecter au serveur Jenkins.

Users

These users can log into Hudson. This is a sub set of [this list](#), which also contains auto-created users who really just made some commits on some projects and have no direct Hudson access.

Name	
 Bob Brown	 
 jill	 
 Joe Black	 
 John Smart	
 Kate the Developer	 
 Rob Smith	 

Figure 7.8. La base de données des utilisateurs de Jenkins

La base de données interne de Jenkins est suffisante pour de nombreuses équipes et organisations. Toutefois, pour des organisations plus importantes, cela peut devenir fastidieux et répétitif de gérer un grand nombre d'utilisateurs à la main. Et plus particulièrement encore si cette information existe déjà quelque part. Dans les sections suivantes, nous regarderons comment brancher Jenkins avec d'autres systèmes de gestion utilisateurs, comme des annuaires LDAP ou des utilisateurs et groupes Unix.

7.4.2. Utiliser un annuaire LDAP

Plusieurs organisations utilisent des annuaires LDAP pour stocker des comptes et mots de passe à travers différentes applications. Jenkins s'intègre bien avec LDAP, sans nécessiter de plugin spécial. Il peut authentifier les utilisateurs en utilisant l'annuaire LDAP, vérifier l'appartenance à un groupe, et récupérer les adresses email des utilisateurs authentifiés.

Pour intégrer Jenkins à votre annuaire LDAP, sélectionnez simplement “LDAP” dans la section Domaine de sécurité, et remplissez les détails concernant votre serveur LDAP (voir Figure 7.9, “Configurer LDAP dans Jenkins”). Le champ le plus important est le serveur de l'annuaire. Si vous utilisez un port non standard, vous devrez aussi l'indiquer (par exemple, `ldap.acme.org:1389`). Si vous utilisez LDAPS, vous devrez aussi le spécifier (par exemple, `ldaps://ldap.acme.org`)

Si votre serveur autorise la connexion anonyme, cela vous suffira probablement pour démarrer. Sinon, vous pouvez utiliser les options avancées pour paramétrer plus finement votre configuration.

La plupart des champs Avancés peuvent sans problème être laissés vides à moins que vous n'ayez une bonne raison de les changer. Si votre annuaire est extrêmement volumineux, vous devriez spécifier une valeur de DN racine (e.g., `dc=acme, dc=com`) et/ou une base de recherche utilisateur et groupe (e.g., `ou=people`) pour réduire la portée des requêtes utilisateur. Ceci n'est habituellement pas nécessaire à moins que vous ne remarquiez des problèmes de performance. Ou, si votre serveur n'autorise pas les connexions anonymes, vous devrez fournir un DN et un mot de passe de gestionnaire, afin que Jenkins puisse se connecter au serveur pour exécuter ses requêtes.

The image shows the 'Security Realm' configuration section in Jenkins. It features a list of radio buttons for selecting a security realm: 'Delegate to servlet container', 'LDAP' (which is selected), 'Unix user/group database', and 'Hudson's own user database'. Below the 'LDAP' option, there are several input fields with labels and help icons: 'Server' (containing 'localhost:1389'), 'root DN', 'User search base', 'User search filter' (containing 'uid={0}'), 'Group search base', 'Manager DN', and 'Manager Password'.

Figure 7.9. Configurer LDAP dans Jenkins

Une fois que vous avez configuré votre serveur LDAP comme domaine de sécurité, vous pouvez configurer votre modèle de sécurité comme décrit précédemment. Quand les utilisateurs se connecteront à Jenkins, ils seront authentifiés sur l'annuaire LDAP.

Vous pouvez aussi utiliser des groupes LDAP, bien que la configuration ne soit pas immédiatement évidente. Supposons que vous ayez défini un group appelé JenkinsAdmin dans votre annuaire LDAP, avec un DN `cn=JenkinsAdmin, ou=Groups, dc=acme, dc=com`. Pour référencer ce groupe dans Jenkins, vous devez prendre le nom commun (cn) en majuscules, et le préfixer avec `ROLE_`. Ainsi `cn=JenkinsAdmin` devient `ROLE_JENKINSADMIN`. Vous pouvez voir un exemple de groupes LDAP utilisés de cette façon dans Figure 7.10, “Utiliser des groupes LDAP dans Jenkins”.

Authorization

Legacy mode
 Project-based Matrix Authorization Strategy
 Logged-in users can do anything
 Anyone can do anything
 Matrix-based security

User/group	Overall		Slave				Job					Run				
	Administer	Read	Configure	Delete	Create	Delete	Configure	Read	Extended	Read	Build	Workspace	Release	Delete	Update	Creat
ROLE_HUDSONADMIN	<input checked="" type="checkbox"/>															
ROLE_HUDSONREADER	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Anonymous	<input type="checkbox"/>															

User/group to add:

Figure 7.10. Utiliser des groupes LDAP dans Jenkins

7.4.3. Utiliser Microsoft Active Directory

Microsoft Active Directory est un logiciel d'annuaire largement utilisé dans les architectures Microsoft. Bien qu'Active Directory fournisse un service LDAP, il peut être compliqué à configurer, et il est plus simple de demander à Jenkins de parler directement au serveur Active Directory. Heureusement, il y a un plugin pour ça.

Le plugin Active Directory de Jenkins vous permet de configurer Jenkins pour authentifier les utilisateurs via un serveur Microsoft Active Directory. Vous pouvez à la fois authentifier les utilisateurs, et récupérer leurs groupes pour la matrice d'autorisations générale ou par projet. Notez que, à l'inverse d'une intégration LDAP conventionnelle (voir Section 7.4.2, “Utiliser un annuaire LDAP”), il n'est pas nécessaire de préfixer les groupes avec `ROLE_` — vous pouvez utiliser l'annuaire des groupes Active Directory (comme “Administrateur de Domaine”).

Pour configurer le plugin, vous devez fournir le nom de domaine complet de votre serveur Active Directory. Si vous avez plus d'un domaine, vous pouvez fournir une liste séparée par des virgules. Si vous fournissez le nom de la forêt (par exemple “acme.com” au lieu de “europe.acme.com”), alors la recherche sera faite à partir du catalogue global. Notez que si vous faites cela sans spécifier le bind DN (voir ci-dessous), l'utilisateur devra se connecter en tant que “europe\joe” ou “joe@europe”.

Les options avancées vous permettent de spécifier un nom de site (pour améliorer les performances en restreignant les contrôleurs de domaine que Jenkins requête), et un DN de liaison et un mot de passe, ce qui peut être pratique si vous vous connectez à une forêt multidomaines. Vous devez fournir des valeurs de DN de liaison et de mot de passe valides, que Jenkins puisse utiliser pour se connecter à votre

serveur afin qu'il établisse l'identité complète de l'utilisateur en cours d'authentification. De cette façon, l'utilisateur peut taper simplement "jack" ou "jill", et faire que le système retrouve automatiquement qu'ils sont jack@europe.acme.com ou jack@asia.acme.com. Vous devez fournir le nom principal complet avec le nom de domaine, tel que admin@europe.acme.com, ou un nom distinctif de style LDAP, tel que CN=Administrator,OU=europe,DC=acme,DC=com.

Une autre bonne chose à propos de ce plugin est qu'il fonctionne à la fois dans un environnement Windows et un environnement Unix. Donc si Jenkins fonctionne sur un serveur Unix, il pourra quand même effectuer les authentifications via un service Microsoft Active Directory d'une autre machine.

Plus précisément, si Jenkins s'exécute sur une machine Windows et que vous ne spécifiez pas de domaine, cette machine doit être un membre du domaine auprès duquel vous souhaitez vous authentifier. Jenkins utilisera ADSI pour retrouver tous les détails, aucune configuration additionnelle n'est donc nécessaire.

Sur une machine non Windows (ou si vous spécifiez un ou plusieurs domaines), vous devez dire à Jenkins le nom du domaine Active Directory auprès duquel s'authentifier. Jenkins utilise alors les enregistrements DNS SRV et le service LDAP d'Active Directory pour authentifier les utilisateurs.

Jenkins peut déterminer les groupes Active Directory auxquels l'utilisateur appartient. Vous pouvez donc les utiliser dans votre stratégie d'autorisations. Par exemple, vous pouvez utiliser ces groupes dans la sécurité basée sur une matrice, ou autoriser les "Administrateurs de domaine" à administrer Jenkins.

7.4.4. Utiliser les utilisateurs et les groupes Unix

Si vous exécutez Jenkins sur une machine Unix, vous pouvez aussi demander à Jenkins d'utiliser les comptes utilisateur et groupes définis sur cette machine. Dans ce cas, les utilisateurs se connecteront à Jenkins en utilisant leurs comptes et mots de passe Unix. On utilise alors le système Pluggable Authentication Modules (PAM), et cela fonctionne aussi bien avec NIS.

Dans sa forme la plus basique, c'est un peu rébarbatif, parce que cela nécessite de créer et de configurer des comptes utilisateurs pour chaque nouvel utilisateur Jenkins. Ce n'est véritablement utile que si ces comptes nécessitent d'être mis en place pour d'autres besoins.

7.4.5. Déléguer au conteneur de Servlet

Une autre façon d'identifier les utilisateurs Jenkins est de laisser le conteneur de Servlet le faire pour vous. Cette approche est utile si vous exécutez Jenkins dans un conteneur de Servlet comme Tomcat ou GlassFish, et que vous avez déjà un moyen établi pour intégrer le conteneur de Servlet avec votre annuaire utilisateur. Tomcat, par exemple, vous permet d'authentifier les utilisateurs par rapport à une base de données relationnelle (en utilisant du JDBC direct ou une DataSource), JNDI, JAAS, ou fichier de configuration XML. Vous pouvez aussi utiliser les rôles définis dans l'annuaire utilisateur du conteneur de Servlet afin de les utiliser pour les stratégies d'autorisation par matrice ou basée sur le projet.

Dans Jenkins, c'est facile à configurer — sélectionnez simplement cette option dans la section Domaine de sécurité (voir Figure 7.11, “Sélectionner le domaine de sécurité”). Une fois que vous avez fait cela, Jenkins laissera le serveur s'occuper de tout.



Figure 7.11. Sélectionner le domaine de sécurité

7.4.6. Utiliser Atlassian Crowd

Si votre organisation utilise les produits Atlassian comme JIRA et Confluence, vous pouvez aussi utiliser Crowd. Crowd est une application commerciale de gestion d'identité et de Single-Sign On (SSO) d'Atlassian qui vous permet de gérer des comptes utilisateur unique à travers différents produits. Vous gérez une base de données interne d'utilisateurs, de groupes et de rôles, et vous vous intégrez avec des annuaires externes comme des annuaires LDAP ou des magasins utilisateur spécifiques.

En utilisant le plugin Jenkins Crowd, vous pouvez utiliser Atlassian Crowd comme source de vos utilisateurs et groupes Jenkins. Avant de commencer, vous devez configurer une nouvelle application dans Crowd (voir Figure 7.12, “Utiliser Atlassian Crowd comme domaine de sécurité Jenkins”). Configurez simplement une nouvelle Application Générique appelée “jenkins” (ou quelque chose du même genre), et avancez d'onglet en onglet. Dans l'onglet Connexions, vous devez fournir l'adresse IP de votre serveur Jenkins. Ensuite, vous devez indiquer l'annuaire Crowd que vous utiliserez pour récupérer les comptes utilisateurs Jenkins et informations de groupes. Enfin, vous devrez dire à Crowd quels utilisateurs de ces annuaires peuvent se connecter à Jenkins. Une option est d'autoriser tous les utilisateurs à s'authentifier, et laisser Jenkins gérer les détails. Sinon, vous pouvez lister les groupes utilisateur Crowd autorisés à se connecter à Jenkins.

Figure 7.12. Utiliser Atlassian Crowd comme domaine de sécurité Jenkins

Une fois que vous avez configuré cela, vous devez installer le plugin Jenkins Crowd, comme vous le faites habituellement via le gestionnaire de plugin de Jenkins. Une fois installé le plugin et Jenkins redémarré, vous pouvez définir Crowd comme domaine de sécurité dans l'écran de configuration principal de Jenkins (voir Figure 7.13, "Utiliser Atlassian Crowd comme domaine de sécurité Jenkins").

Figure 7.13. Utiliser Atlassian Crowd comme domaine de sécurité Jenkins

Avec ce plugin installé et configuré, vous pouvez utiliser les utilisateurs et les groupes de Crowd pour toutes les stratégies d'autorisation que nous avons présentées précédemment dans ce chapitre. Par exemple, dans Figure 7.14, "Utiliser les groupes Atlassian Crowd dans Jenkins", nous utilisons des groupes utilisateurs définis dans Crowd pour configurer une sécurité basée sur la matrice dans l'écran principal de configuration.

User/group	Overall				Job								Run			View			SCM	
	Administer	Read	Configure	Delete	Create	Delete	Configure	Read	Extended	Read	Build	Workspace	Release	Delete	Update	Create	Delete	Configure	Promote	Tag
authenticated	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
hudson-administrators	<input checked="" type="checkbox"/>																			
hudson-read-only	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Anonymous	<input type="checkbox"/>																			

Figure 7.14. Utiliser les groupes Atlassian Crowd dans Jenkins

7.4.7. S'intégrer avec d'autres systèmes

En plus des stratégies d'authentification discutées ici, il y a un certain nombre d'autres plugins permettant à Jenkins d'effectuer l'authentification via d'autres systèmes. Au moment de l'écriture de ces lignes, ceci inclut Central Authentication Service (CAS) — un outil open source de single sign-on — et le serveur Collabnet Source Forge Enterprise Edition (SFEE).

Si aucun plugin n'est disponible, vous pouvez aussi écrire votre propre script d'authentification. Pour faire cela, vous devez installer le plugin Script Security Realm. Une fois que vous avez installé le script et redémarré Jenkins, vous pouvez écrire deux scripts dans votre langage de scripting favori. Un script authentifie l'utilisateur, tandis que l'autre détermine les groupes d'un utilisateur donné (voir Figure 7.15, “Utiliser des scripts personnalisés pour gérer l'authentification”).

Security Realm

Delegate to servlet container
 LDAP
 Unix user/group database
 Hudson's own user database
 Active Directory
 Crowd
 Authenticate via custom script

Login Command:

Groups Command:

Groups Delimiter:

Authorization

Legacy mode
 Project-based Matrix Authorization Strategy
 Logged-in users can do anything
 Anyone can do anything
 Matrix-based security

User/group	Overall				Job								Run			View		
	Administer	Read	Configure	Delete	Create	Delete	Configure	Read	Extended	Read	Build	Workspace	Release	Delete	Update	Create	Delete	Configure
admin	<input checked="" type="checkbox"/>																	
authenticated	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Anonymous	<input type="checkbox"/>																	

Figure 7.15. Utiliser des scripts personnalisés pour gérer l'authentification

Avant d'invoquer le script d'authentification, Jenkins positionne deux variables d'environnement : `U`, contenant le nom d'utilisateur, et `P`, contenant le mot de passe. Ce script utilise ces variables

d'environnement pour authentifier le nom d'utilisateur et le mot de passe spécifié, en retournant 0 en cas de réussite, et une autre valeur sinon. Si l'authentification échoue, la sortie du processus sera renvoyée dans le message d'erreur affiché à l'utilisateur. Voici un simple script script d'authentification Groovy :

```
def env = System.getenv()
def username = env['U']
def password = env['P']

println "Authenticating user $username"

if (authenticate(username, password)) {
    System.exit 0
} else {
    System.exit 1
}

def authenticate(def username, def password) {
    def userIsAuthenticated = true
    // Authentication logic goes here
    return userIsAuthenticated
}
```

Ce script est suffisant si tout ce que vous avez à faire est de gérer une authentification basique sans groupes. Si vous voulez utiliser des groupes de votre source d'authentification personnalisée dans vos autorisations matricielles ou projet (voir Section 7.5, “Autorisation — Qui peut faire quoi”), vous pouvez écrire un second script, qui détermine les groupes pour un utilisateur donné. Ce script utilise la variable d'environnement U pour déterminer quel utilisateur essaie de se connecter, et affiche une liste de groupe séparée par des virgules pour cet utilisateur sur la sortie standard. Si vous n'aimez pas les virgules, vous pouvez redéfinir le caractère de séparation dans la configuration. Voici un simple script Groovy pour faire cela :

```
def env = System.getenv()
def username = env['U']

println findGroupsFor(username)

System.exit 0

def findGroupsFor(def username) {
    return "admin,game-of-life-developer"
}
```

Ces deux scripts doivent renvoyer 0 lorsqu'ils sont appelés pour qu'un utilisateur soit authentifié.

7.5. Autorisation — Qui peut faire quoi

Une fois que vous avez défini comment identifier vos utilisateurs, vous devez décider ce qu'ils sont autorisés à faire. Jenkins supporte différentes stratégies dans ce domaine, allant d'une simple approche où un utilisateur connecté peut faire n'importe quoi à des stratégies impliquant des rôles plus précis et des authentifications par projet.

7.5.1. Sécurité basée sur une matrice

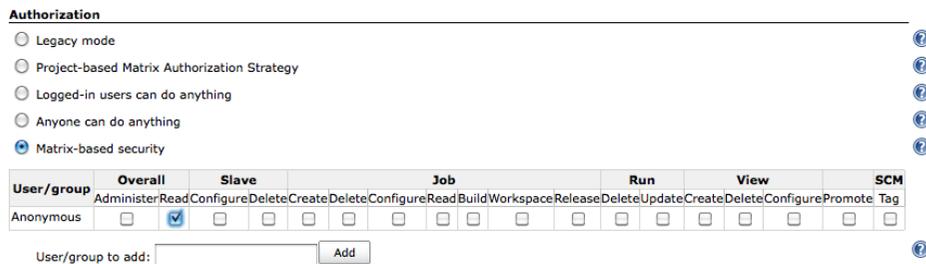
Laisser les utilisateurs connectés faire n'importe quoi est certainement flexible, et pourrait suffire pour une petite équipe. Pour des équipes plus importantes ou multiples, ou des cas où Jenkins est utilisé en dehors d'un environnement de développement, une approche plus sophistiquée est généralement requise.

La sécurité basée sur matrice est une approche plus élaborée, où différents utilisateurs se voient attribuer différents droits, en utilisant une approche basée sur les rôles.

7.5.1.1. Mettre en place la sécurité basée sur une matrice

La première étape dans la configuration de la sécurité basée sur une matrice dans Jenkins est de créer un administrateur. C'est une étape essentielle, et elle doit être effectuée avant toutes les autres. Votre administrateur peut être un utilisateur existant, ou un créé spécialement dans ce but. Si vous voulez créer un utilisateur administrateur dédié, créez le simplement en vous enregistrant de la façon habituelle (voir Figure 7.2, “La page de connexion Jenkins”). Il n'est pas nécessaire qu'il soit associé à un utilisateur SCM.

Une fois que votre utilisateur administrateur est prêt, vous pouvez activer la sécurité basée sur une matrice en sélectionnant “Sécurité basée sur une matrice” dans la section Autorisations de la page de configuration principale. Jenkins affichera un tableau contenant les utilisateurs autorisés, et des cases à cocher correspondant aux différentes permissions que vous pouvez affecter à ces utilisateurs (voir Figure 7.16, “Sécurité basée sur une matrice”).



The screenshot shows the Jenkins 'Authorization' configuration page. Under the 'Authorization' heading, there are five radio button options: 'Legacy mode', 'Project-based Matrix Authorization Strategy', 'Logged-in users can do anything', 'Anyone can do anything', and 'Matrix-based security'. The 'Matrix-based security' option is selected. To the right of these options are five question mark icons. Below the radio buttons is a table with columns for 'User/group', 'Overall', 'Slave', 'Job', 'Run', 'View', and 'SCM'. The 'SCM' column has sub-columns for 'Administer', 'Read', 'Configure', 'Delete', 'Create', 'Delete', 'Configure', 'Read', 'Build', 'Workspace', 'Release', 'Delete', 'Update', 'Create', 'Delete', 'Configure', 'Promote', and 'Tag'. The 'Anonymous' user is listed in the first row, with a checked box under 'Slave' and 'Read' in the 'SCM' column. Below the table is a text input field labeled 'User/group to add:' followed by an 'Add' button and a question mark icon.

Figure 7.16. Sécurité basée sur une matrice

L'utilisateur spécial “anonyme” est toujours présent dans le tableau. Cet utilisateur représente les utilisateurs non authentifiés. Typiquement, vous ne donnez que des droits limités aux utilisateurs non authentifiés, comme des accès en lecture seule, ou pas d'accès du tout (comme montré dans Figure 7.16, “Sécurité basée sur une matrice”).

La première chose que vous devez savoir faire maintenant est de donner les droits d'administration à votre administrateur. Ajoutez cet utilisateur d'administration dans le champ “Utilisateur/groupe à ajouter” et cliquez sur Ajouter. Votre administrateur apparaîtra alors dans la matrice de permissions.

Maintenant assurez-vous d'accorder toutes les permissions à cet utilisateur (voir Figure 7.17, “Configurer un administrateur”), et sauvez votre configuration. Vous devriez à présent être capable de vous connecter avec votre compte utilisateur administrateur (si vous n’êtes pas déjà connecté avec ce compte) et continuer à configurer vos autres utilisateurs.

Matrix-based security

User/group	Overall		Slave		Job				Run		View			SCM				
	Administer	Read	Configure	Delete	Create	Delete	Configure	Read	Build	Workspace	Release	Delete	Update	Create	Delete	Configure	Promote	Tag
Anonymous	<input type="checkbox"/>																	
administrator	<input checked="" type="checkbox"/>																	

User/group to add: Add

Figure 7.17. Configurer un administrateur

7.5.1.2. Configurer plus finement les permissions utilisateurs

Une fois que vous avez configuré votre compte administrateur, vous pouvez ajouter d'autres utilisateurs ayant besoin d'accéder à votre instance Jenkins. Ajoutez simplement les noms d'utilisateur et cochez les permissions que vous voulez leur accorder (voir Figure 7.18, “Configurer les autres utilisateurs”). Si vous utilisez un serveur LDAP ou des utilisateurs ou groupes Unix comme schéma d'authentification sous-jacent (voir Section 7.4.2, “Utiliser un annuaire LDAP”), vous pouvez aussi configurer les permissions pour des groupes d'utilisateurs.

Matrix-based security

User/group	Overall		Slave		Job				Run		View			SCM				
	Administer	Read	Configure	Delete	Create	Delete	Configure	Read	Build	Workspace	Release	Delete	Update	Create	Delete	Configure	Promote	Tag
administrator	<input checked="" type="checkbox"/>																	
bob	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
joe	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
kate	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Anonymous	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

User/group to add: Add

Figure 7.18. Configurer les autres utilisateurs

Vous pouvez accorder différentes permissions, qui sont organisées en plusieurs groupes : Global, Esclave, Job, Lancer, Voir et Gestion de version. La plupart des permissions sont assez évidentes, mais certaines nécessitent un peu plus d'explication. Les permissions individuelles sont comme suit :

Global

Ce groupe couvre des permissions basiques sur l'ensemble du système :

Administrer

Permet à un utilisateur de faire des modifications de configuration à l'ensemble du système ou autres opérations sensibles, par exemple dans les pages de configuration principales de Jenkins. Ceci devrait être réservé à l'administrateur Jenkins.

Lire

Cette permission fournit un accès en lecture seule à pratiquement toutes les pages de Jenkins. Si vous voulez que les utilisateurs anonymes puissent voir les tâches librement, mais qu'ils ne puissent pas les modifier ou les démarrer, donnez le droit Lire à l'utilisateur spécial "anonyme". Sinon, enlevez simplement cette permission à l'utilisateur Anonyme. Et si vous voulez que tous les utilisateurs authentifiés puissent voir les tâches de build, ajoutez alors un utilisateur spécial "authenticated", et donnez lui la permission Global/Lire.

Esclave

Ce groupe couvre des permissions à propos de noeuds de constructions, ou esclaves :

Configurer

Créer ou configurer de nouveaux noeuds de construction.

Supprimer

Supprimer des noeuds de construction.

Job

Ce group couvre des permissions liées aux tâches :

Créer

Créer une nouvelle tâche de build.

Supprimer

Supprimer une tâche de build existante.

Configurer

Mettre à jour la configuration de tâches de build existantes.

Lire

Voir des tâches de build.

Build

Démarrer une tâche de build.

Espace de travail

Voir et télécharger le contenu de l'espace de travail pour une tâche de build. Rappelez-vous, l'espace de travail contient le code source et les artefacts, donc si vous voulez protéger ces derniers d'un accès général, vous devriez enlever cette permission.

Release

Démarrer une release Maven pour un projet configuré avec le plugin M2Release.

Lancer

Ce groupe couvre les droits relatifs à des builds spécifiques de l'historique des builds :

Supprimer

Supprimer un build de l'historique de build.

Mettre à jour

Mettre à jour la description et d'autres propriétés d'un build dans l'historique de build. Ceci peut être utile si un utilisateur veut laisser une note sur la cause des échecs de build, par exemple.

Voir

Ce groupe couvre des vues de gestion :

Créer

Créer une nouvelle vue.

Supprimer

Supprimer une vue existante.

Configurer

Configurer une vue existante.

Gestion de version

Permissions relatives à votre système de contrôle de version :

Tag

Créer un nouveau tag dans le dépôt de code source pour un build donné.

Autres

Il peut y avoir d'autres permissions disponibles, en fonction des plugins installés. Une permission utile est :

Promouvoir

Si le plugin Promoted Builds est installé, cette permission permet aux utilisateurs de promouvoir manuellement un build.

7.5.1.3. A l'aide ! Je me suis verrouillé tout seul !

Il pourrait arriver que, pendant ce processus, vous vous bloquiez tout seul l'accès à Jenkins. Ceci peut arriver si, par exemple, vous sauvez la configuration matricielle sans avoir correctement configuré votre administrateur. Si cela arrive, ne paniquez pas — il y a une correction simple, du moment que vous avez accès au répertoire racine de Jenkins. Ouvrez simplement le fichier `config.xml` à la racine du répertoire de Jenkins. Il contiendra quelque chose de ce genre :

```
<hudson>
  <version>1.391</version>
  <numExecutors>2</numExecutors>
```

```
<mode>NORMAL</mode>
<useSecurity>true</useSecurity>
...
```

La chose à rechercher est l'élément `<useSecurity>`. Pour restaurer votre accès à Jenkins, changez cette valeur à `false`, et redémarrez votre serveur. Vous pourrez alors à nouveau accéder à Jenkins, et mettre en place votre configuration de sécurité correctement.

7.5.2. Sécurité basée sur le projet

La sécurité basée sur le projet vous permet d'utiliser le modèle de sécurité matricielle dont vous venons de discuter, et l'appliquer à des projets individuels. Non seulement vous pouvez assigner des rôles globaux à vos utilisateurs, mais vous pouvez aussi configurer des droits plus spécifiques à certains projets en particulier.

Pour activer la sécurité de niveau projet, sélectionnez “Stratégie d'autorisation matricielle basée sur les projets” dans la section Autorisations de l'écran de configuration principal (voir Figure 7.19, “Sécurité basée sur le projet”). Ici, vous pouvez configurer des droits par défaut pour les utilisateurs et les groupes, comme nous l'avons vu dans la sécurité basée sur une matrice (voir Section 7.5.1, “Sécurité basée sur une matrice”).

Authorization

Legacy mode

Project-based Matrix Authorization Strategy

User/group	Overall		Slave		Job				Run		View			SCM				
	Administer	Read	Configure	Delete	Create	Delete	Configure	Read	Build	Workspace	Release	Delete	Update	Create	Delete	Configure	Promote	Tag
administrator	<input checked="" type="checkbox"/>																	
bob	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
joe	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
johnsmart	<input checked="" type="checkbox"/>																	
kate	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Anonymous	<input type="checkbox"/>																	

User/group to add:

Logged-in users can do anything

Anyone can do anything

Matrix-based security

Figure 7.19. Sécurité basée sur le projet

Ce sont les permissions par défaut à appliquer à tous les projets qui n'ont pas été spécifiquement configurés. Toutefois, quand vous utilisez la sécurité basée sur le projet, vous pouvez aussi positionner des permissions spécifiques au projet. Pour faire cela, sélectionnez “Activer la sécurité basée sur le projet” dans l'écran de configuration du projet (voir Figure 7.20, “Configurer la sécurité basée sur le projet”). Jenkins affichera un tableau de permissions spécifiques au projet. Vous pouvez configurer ces permissions pour différents utilisateurs et groupes comme dans la page de configuration globales. Ces permissions seront ajoutées aux permissions globales pour produire un ensemble de permissions spécifiquement applicables à ce projet.

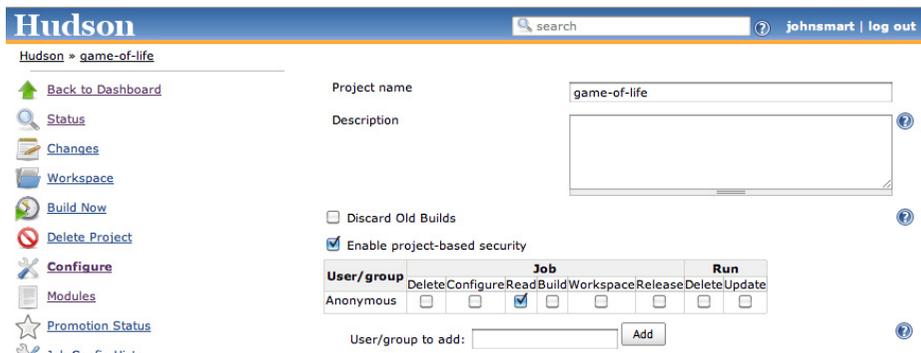


Figure 7.20. Configurer la sécurité basée sur le projet

Comprendre comment cela fonctionne est plus facile avec quelques exemples pratiques. Dans Figure 7.19, “Sécurité basée sur le projet”, par exemple, aucune permission n’a été donnée à l’utilisateur anonyme. Donc, par défaut, toutes les tâches de build resteront invisibles jusqu’à ce que l’utilisateur se connecte. Toutefois, nous utilisons une sécurité basée sur le projet, nous pouvons donc redéfinir cela projet par projet. Dans Figure 7.20, “Configurer la sécurité basée sur le projet”, par exemple, nous avons configuré le projet `game-of-life` pour qu’il offre l’accès en lecture seule à l’utilisateur spécial “anonyme”.

Quand vous aurez sauvé cette configuration, les utilisateurs non authentifiés pourront voir le projet `game-of-life` en mode lecture seule (voir Figure 7.21, “Voir un projet”). Le même principe s’applique avec toutes les permissions spécifiques au projet.

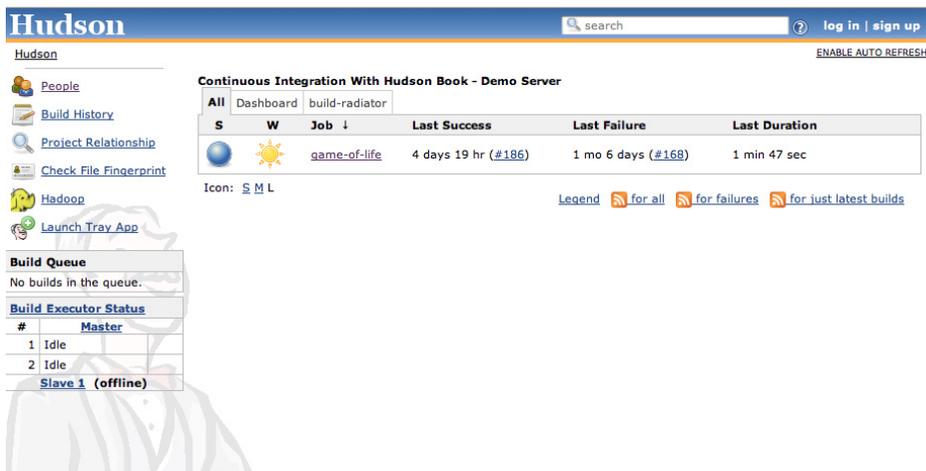


Figure 7.21. Voir un projet

Notez que les permissions sont cumulatives — à l’écriture de ces lignes, il n’y a pas de moyen d’enlever une permission globale pour un projet particulier. Par exemple, si l’utilisateur anonyme a l’accès en lecture seule au niveau global, vous ne pouvez pas lui enlever pour un projet individuel. Donc quand vous utilisez la sécurité basée sur le projet, utilisez la matrice globale au système pour définir

des permissions par défaut minimales à travers tous vos projets, et configurez les projets avec des autorisations additionnelles spécifiques au projet.

Il y a plusieurs approches pour gérer les permissions de projet, et elles dépendent autant sur la culture organisationnelle que sur des considérations techniques. Une stratégie courante est d'autoriser les membres de l'équipe à avoir l'accès complet à leurs propres projets, et l'accès en lecture seule aux autres projets. Le plugin Extended Read Permission est une extension utile à avoir dans ce scénario. Ce plugin vous permet d'offrir aux utilisateurs d'autres équipes une vue en lecture seule de la configuration de votre projet, sans avoir à modifier quoi que ce soit (voir Figure 7.22, “Configurer les permissions de droit de lecture étendus”). C'est un formidable moyen de partager des pratiques de configuration de build et des astuces avec d'autres équipes sans les laisser trafiquer vos builds.

Enable project-based security

User/group	Job							Run	
	Delete	Configure	Read	Extended Read	Build	Workspace	Release	Delete	Update
authenticated	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>				
Anonymous	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 7.22. Configurer les permissions de droit de lecture étendus

Il est intéressant de noter que, aussi importantes ou multiples que soient les équipes impliquées, la base de données interne de Jenkins atteint ses limites assez rapidement, et cela vaut le coup d'envisager l'intégration avec un service d'annuaire plus spécialisé comme un serveur LDAP, Active Directory ou Atlassian Crowd, ou alors un système de permission plus sophistiqué comme une sécurité basée sur les rôles, discutée dans la section suivante.

7.5.3. Sécurité basée sur les rôles

Quelques fois, gérer les permissions utilisateur individuellement peut s'avérer rébarbatif, et vous pourriez ne pas vouloir vous intégrer avec un serveur LDAP pour configurer vos groupes avec. Une alternative plus récente est d'utiliser le plugin Role Strategy, qui permet de définir des rôles globaux ou de niveau projet, et les affecter aux utilisateurs.

Vous installez le plugin de façon habituelle, via le gestionnaire de plugin. Une fois installé, vous pouvez activer cette stratégie d'autorisation sur la page de configuration principale (voir Figure 7.23, “Configurer la sécurité basée sur les rôles”).



Figure 7.23. Configurer la sécurité basée sur les rôles

Quand vous avez configuré cela, vous pouvez définir des rôles regroupant des ensembles de permissions liées les unes aux autres. Vous créez et configurez vos rôles, puis les assignez à vos utilisateurs, dans l'écran de gestion de rôles, auquel vous accédez dans l'écran Administrer Jenkins (voir Figure 7.24, “Le menu de configuration Gérer les rôles”).

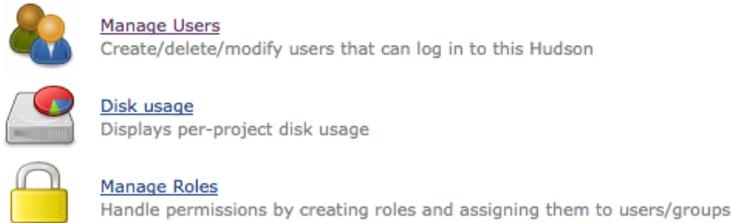


Figure 7.24. Le menu de configuration Gérer les rôles

Dans l'écran Gérer les rôles, vous pouvez mettre en place des permissions globales ou de niveau projet. Les permissions globales s'appliquent à tous les projets, et sont typiquement des permissions d'administration du système ou d'accès général (voir Figure 7.25, “Gérer les rôles globaux”). La mise en oeuvre de ces rôles est intuitive et similaire à la configuration des permissions utilisateur dans les autres modèles de sécurité que nous avons vus.

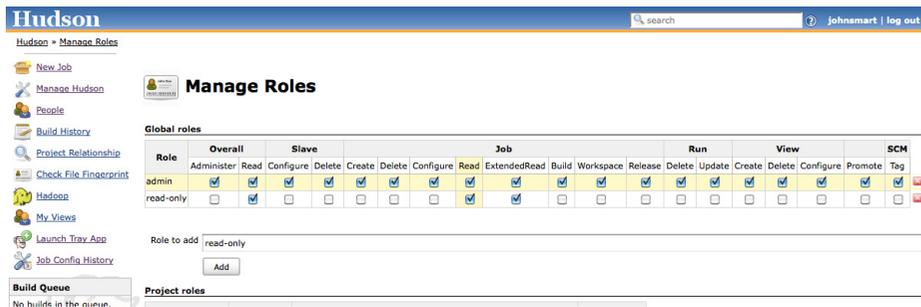


Figure 7.25. Gérer les rôles globaux

Les rôles de projet sont légèrement plus compliqués. Un rôle de projet regroupe un ensemble de permissions applicables à un ou plusieurs projets (a priori reliés). Vous définissez les projets concernés en utilisant une expression régulière, cela aide à avoir un ensemble clair et cohérent de conventions de nommage dans vos noms de projets (voir Figure 7.26, “Gérer les rôles de projets”). Par exemple, vous pouvez vouloir créer des rôles distinguant les développeurs avec des droits complets sur leurs propres projets d'utilisateur qui peuvent simplement déclencher un build et voir le résultat. Ou vous pouvez créer des rôles où les développeurs peuvent configurer certaines tâches de build de déploiement automatisé, mais que seules les équipes de production soient autorisées à exécuter ces tâches.

Project roles

Role	Pattern	Job							Run			
		Delete	Configure	Read	ExtendedRead	Build	Workspace	Release	Delete	Update		
game-of-life-developer	game-of-life.*	<input checked="" type="checkbox"/>										
game-of-life-run-build	game-of-life.*	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
production-deployment	prod-deploy.*	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
uat-deployment	uat-deploy.*	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
deployment-developer	.*-deployment	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	

Role to add:

Pattern:

Figure 7.26. Gérer les rôles de projets

Une fois ces rôles définis, vous pouvez aller dans l'écran Assigner les rôles pour configurer les utilisateurs ou groupes avec ces rôles (voir Figure 7.27, “Assigner des rôles à des utilisateurs”).

Assign Roles

Global roles

User/group	admin	read-only	
administrator	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
authenticated	<input type="checkbox"/>	<input checked="" type="checkbox"/>	
johnsmart	<input checked="" type="checkbox"/>	<input type="checkbox"/>	
Anonymous	<input type="checkbox"/>	<input type="checkbox"/>	

User/group to add:

Project roles

User/group	deployment-developer	game-of-life-developer	game-of-life-run-build	production-deployment	uat-deployment	
bob	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	
joe	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
kate	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
rob	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	
Anonymous	<input type="checkbox"/>					

User/group to add:

Figure 7.27. Assigner des rôles à des utilisateurs

La stratégie basée sur les rôles est relativement nouvelle dans Jenkins, mais c'est une excellente façon de simplifier les tâches de gestion des permissions dans de grosses organisations, multi-équipes et multi-projets.

7.6. Audit — Garder la trace des actions utilisateurs

En plus de configurer les comptes utilisateurs et leurs droits d'accès, il peut être utile de garder des actions de chaque utilisateur : en d'autres termes, qui a fait quoi à votre configuration serveur. Ce type de traçage est même requis dans certaines organisations.

Il y a deux plugins Jenkins qui peuvent vous aider à accomplir cela. Le plugin Audit Trail conserve un enregistrement des changements utilisateur dans un fichier de log spécial. Et le plugin JobConfigHistory vous permet de garder des copies de versions précédentes des diverses configurations de tâches et du système que Jenkins utilise.

Le plugin Audit Trail garde une trace des principales actions utilisateur dans un ensemble de fichiers de logs tournants. Pour mettre cela en place, allez sur la page Gérer les plugins et sélectionnez le plugin Audit Trail dans la liste des plugins disponibles. Ensuite, comme d'habitude, cliquez sur Installer et redémarrez Jenkins une fois que le plugin a été téléchargé.

La configuration de l'audit trail s'effectue dans la section Audit Trail de l'écran de configuration principal de Jenkins (voir Figure 7.28, “Configurer le plugin Audit Trail”). Le champ le plus important est l'emplacement des logs, qui indique où se trouve le répertoire dans lequel les logs doivent être écrits. L'audit trail est conçu pour produire des logs de style système, qui sont souvent placés dans un répertoire système comme `/var/log`. Vous pouvez aussi configurer le nombre de fichiers de logs à maintenir, et la taille maximale (approximative) de chaque fichier. L'option la plus simple est de fournir un chemin absolu (comme `/var/log/hudson.log`), auquel cas Jenkins écrira dans des fichiers de logs avec des noms comme `/var/log/hudson.log.1`, `/var/log/hudson.log.2`, et ainsi de suite. Bien sûr, vous devez vous assurer que l'utilisateur exécutant votre instance Jenkins peut écrire dans ce répertoire.

Audit Trail

Log Location	<input type="text" value="/var/log/hudson-audit-trail.log"/>	?
Log File Size MB	<input type="text" value="1"/>	?
Log File Count	<input type="text" value="10"/>	?
URL Patterns to Log	<input type="text" value=".*(?:configSubmit doDelete postBuildResult cancelQueue stop toggleLogKeep doWipeOutW"/>	?

Log how each build is triggered

Figure 7.28. Configurer le plugin Audit Trail

Vous pouvez aussi utiliser le format défini dans la classe de l'API de logging Java `FileHandler`¹ pour plus de contrôle sur les fichiers de log générés. Dans ce format, vous pouvez insérer des variables telles que `%h`, pour le répertoire racine de l'utilisateur courant, et `%t`, pour le répertoire temporaire du système, afin de construire un chemin de fichier plus dynamique.

Par défaut, les détails enregistrés dans les logs d'audit sont assez légers — ils enregistrent effectivement les actions clés effectuées, comme la création, la modification ou la suppression de

¹ <http://download.oracle.com/javase/1.5.0/docs/api/java/util/logging/FileHandler.html>

configurations de tâches ou de vues, ainsi que l'utilisateur qui a effectué ces actions. Le log montre aussi comment les tâches individuelles ont été démarrées. Voici un extrait du log par défaut :

```
Dec 27, 2010 9:16:08 AM /job/game-of-life/configSubmit by johnsmart
Dec 27, 2010 9:16:42 AM /view/All/createItem by johnsmart
Dec 27, 2010 9:16:57 AM /job/game-of-life-prod-deployment/doDelete by johnsmart
Dec 27, 2010 9:24:38 AM job/game-of-life/ #177 Started by user johnsmart
Dec 27, 2010 9:25:57 AM job/game-of-life-acceptance-tests/ #107 Started by upstream
project "game-of-life" build number 177
Dec 27, 2010 9:25:58 AM job/game-of-life-functional-tests/ #7 Started by upstream
project "game-of-life" build number 177
Dec 27, 2010 9:28:15 AM /configSubmit by johnsmart
```

L'audit trail est certainement utile, particulièrement avec une perspective d'administration système. Toutefois, cela ne produit pas d'information à propos des changements exacts qui ont été faits à la configuration Jenkins. Néanmoins, une des raisons les plus importantes pour garder des traces d'actions utilisateurs dans Jenkins est de savoir quels changements ont été faits aux configurations de tâches de build. Quand quelque chose se passe mal, il peut être utile de savoir quels changements ont été faits et être donc capables de les défaire. Le plugin JobConfigHistory vous permet justement de faire cela.

Le plugin JobConfigHistory est un outil puissant vous permettant de conserver l'historique complet des changements faits à la fois sur les tâches et fichiers de configuration système. Vous l'installez depuis le gestionnaire de plugin de la façon habituelle. Une fois installé, vous pouvez régler finement la configuration de l'historique des tâches dans l'écran Administrer Jenkins (voir Figure 7.29, “Mettre en place l'historique des configurations de tâches”).

Job Config History	
Root history folder	<input type="text" value="config-history"/>
Max number of history entries to keep	<input type="text" value="100"/>
Save system configuration changes	<input checked="" type="checkbox"/>
System configuration exclude file pattern	<input type="text" value="queue nodeMonitors UpdateCenter"/>
Do not save duplicate history	<input checked="" type="checkbox"/>

Figure 7.29. Mettre en place l'historique des configurations de tâches

Ici, vous pouvez configurer un bon nombre d'options utiles non standard. En particulier, vous devriez spécifier un répertoire où Jenkins peut stocker l'historique de configuration, dans le champ "Répertoire racine de l'historique". C'est le répertoire dans lequel Jenkins stockera un enregistrement des changements à la fois liés au système et aux configuration de tâches. Cela peut être à la fois un répertoire absolu (comme `/var/hudson/history`), ou un répertoire relatif, calculé à partir de la racine du répertoire de Jenkins (voir Section 3.4, “Le répertoire de travail de Jenkins”). Si vous ne faites pas cela, l'historique de configuration des tâches sera stocké dans les tâches elles-mêmes, et tout sera perdu si vous supprimez une tâche.

Il y a un certain nombre d'autres options utiles dans la section Avancé. La case à cocher “Sauver les changements de configuration système” vous permet de garder la trace de mise à jour de configuration de niveau système, et non pas seulement de celles spécifiques aux tâches. Et la case à cocher “Ne pas

sauver d'historique dupliqué” vous permet d'éviter d'avoir des enregistrements de configuration si aucun changement réel n'a été effectué. Sinon, une nouvelle version du fichier de configuration sera enregistrée, même si vous avez seulement appuyé sur le bouton Sauver sans faire aucun changement. Jenkins peut aussi provoquer cela en interne - par exemple, le paramétrage de la configuration système est entièrement sauvé à chaque fois que la page principale de configuration est sauvée, même si aucune modification n'a été faite.

Une fois que vous avez mis ce plugin en place, vous pouvez accéder à l'historique pour le serveur complet, incluant les mises à jour de configuration système, aussi bien qu'aux changements effectués à la configuration de chaque projet. Dans les deux cas, vous pouvez voir ces changements en cliquant sur l'icône Job Config History sur la droite de l'écran. Cliquer sur cette icône affichera une vue de tout votre historique de configuration, incluant les changements de tâches et les changements de niveau système (voir Figure 7.30, “Présentation de l'historique de configuration des tâches”).

The screenshot shows the Hudson Job Config History page. At the top, there is a search bar and a user profile for 'johnsmart' with a 'log out' link. Below the search bar, there is a 'Back to Dashboard' link. The main heading is 'All Configuration History'. Below this is a table with the following columns: Date, Job/System configuration, Operation, User, and File(raw). The table contains 10 rows of data, all showing 'Changed' operations by 'johnsmart' on various configuration files. Each row has a 'View as XML (RAW)' link.

Date	Job/System configuration	Operation	User	File(raw)
2010-12-27_09-56-29	game-of-life	Changed	johnsmart	View as XML (RAW)
2010-12-27_09-46-53	config (system)	Changed	johnsmart	View as XML (RAW)
2010-12-27_09-45-03	hudson.plugins.groovy.Groovy (system)	Changed	johnsmart	View as XML (RAW)
2010-12-27_09-45-03	hudson.scm.SubversionSCM (system)	Changed	johnsmart	View as XML (RAW)
2010-12-27_09-42-44	game-of-life	Changed	johnsmart	View as XML (RAW)
2010-12-27_09-35-23	audit-trail (system)	Changed	johnsmart	View as XML (RAW)
2010-12-27_09-35-23	com.g2one.hudson.grails.GrailsBuilder (system)	Changed	johnsmart	View as XML (RAW)
2010-12-27_09-35-23	config (system)	Changed	johnsmart	View as XML (RAW)

Figure 7.30. Présentation de l'historique de configuration des tâches

Si vous cliquez sur un changement de niveau système (indiqué par le suffixe “(system)” dans la liste), Jenkins vous emmène vers un écran listant toutes les versions de ce fichier, et vous permet de voir les différences entre les versions (voir Figure 7.31, “Voir les différences dans l'historique de configuration des tâches”). Les différences sont affichées sous la forme de fichiers diff, ce qui n'est pas particulièrement lisible en soi. Toutefois, pour de petits changements, le format XML lisible de Jenkins rend cela suffisant pour comprendre le changement qui a été effectué.

The screenshot shows the Hudson config Configuration History page. At the top, there is a search bar and a user profile for 'johnsmart' with a 'log out' link. Below the search bar, there is a 'Back to Dashboard' link. The main heading is 'config Configuration History'. Below this is a table with the following columns: Date, Job/System configuration, Operation, User, File(raw), and Diff. The table contains 3 rows of data, all showing 'Changed' operations by 'johnsmart' on the 'config (system)' file. Each row has a 'View as XML (RAW)' link and a 'Diff' column with two radio buttons for 'File A' and 'File B'. The 'Diff' column also has a 'Show Diffs' button at the bottom.

Date	Job/System configuration	Operation	User	File(raw)	Diff
2010-12-27_09-46-53	config (system)	Changed	johnsmart	View as XML (RAW)	File A <input checked="" type="radio"/> File B <input type="radio"/>
2010-12-27_09-35-23	config (system)	Changed	johnsmart	View as XML (RAW)	File A <input type="radio"/> File B <input checked="" type="radio"/>
2010-12-27_09-28-16	config (system)	Changed	johnsmart	View as XML (RAW)	File A <input type="radio"/> File B <input type="radio"/>

Figure 7.31. Voir les différences dans l'historique de configuration des tâches

Le plugin JobConfigHistory est un outil puissant. Toutefois, à l'écriture de ces lignes, il a ses limites. Comme mentionné, le plugin affiche uniquement les différences au format `diff` brut, et vous ne pouvez pas restaurer une version précédente d'un fichier de configuration (faire cela hors contexte pourrait être dangereux dans certaines circonstances, particulièrement pour les fichiers de configuration système). Cependant, il donne un aperçu très clair des changements ayant été effectués, à la fois sur vos tâches de build et sur votre configuration système.

7.7. Conclusion

Dans ce chapitre, nous avons regardé une variété de façons de configurer la sécurité dans Jenkins. Le modèle de sécurité Jenkins, avec les deux concepts orthogonaux d'Authentification et d'Autorisation, est flexible et extensible. Pour une installation Jenkins de n'importe quelle taille, vous devriez essayer d'intégrer votre stratégie de sécurité Jenkins dans l'organisation dans son ensemble. Ceci peut aller d'une simple intégration à votre annuaire local LDAP à la mise en place d'une solution complète de SSO comme Crowd ou CAS. Dans les deux cas, ceci rendra le système considérablement plus facile à administrer à long terme.

Chapter 8. Notification

8.1. Introduction

Bien qu'il soit important que votre serveur de build construise votre logiciel, il est encore plus important qu'il signale lorsqu'il ne peut pas le faire. Une part importante de la proposition de valeur de tout environnement d'intégration continue est d'améliorer le flux d'information sur la santé de votre projet ; que ce soit des tests d'intégration échoués, des régressions dans la suite des tests d'intégration ou d'autres problèmes de qualité comme une baisse dans la couverture de code ou de métriques de qualité de code. Dans tous les cas, un serveur d'intégration continue doit permettre aux bonnes personnes de connaître les nouveaux problèmes, et il doit pouvoir le faire rapidement. C'est ce que nous appelons : Notification.

Il y a deux principales catégories de stratégies de notification, que j'appelle passive et active (ou pull/push). Les notifications passives (pull) demandent aux développeurs de consulter consciemment le statut du dernier build. Ces notifications comprennent les flux RSS, les radars de build, et (dans une certaine mesure) les e-mails. Les notifications actives (push) vont alerter les développeurs de manière pro-active lorsqu'un build échoue. Ces notifications comprennent des méthodes telles que des notifieurs intégrés au bureau, le dialogue en direct et les SMS. Ces deux approches ont leurs points positifs et négatifs. Les stratégies de notifications passives telles que les radars de build peuvent diffuser une connaissance générale sur les builds échoués et aider à installer une culture d'équipe où la correction des builds cassés prend une haute priorité. Des formes plus directes de notification peuvent encourager activement les développeurs à prendre les choses en main et corriger les builds cassés plus rapidement.

8.2. Notification par email

La notification par email est la forme de notification la plus évidente et la plus commune. L'email est bien connu, omniprésent et facile à utiliser et à configurer (voirSection 4.8, “Configurer le serveur de messagerie électronique”). Ainsi, quand les équipes mettent en place leur premier environnement d'intégration continue, c'est généralement la stratégie de notification qu'ils essaient en premier.

Vous activez les notifications email dans Jenkins en cochant la case Notification par email et en fournissant la liste des adresses email des personnes qui doivent être notifiées (voirFigure 8.1, “Configurer les notifications par email”). Par défaut, Jenkins enverra un courrier pour chaque build échoué ou instable. Rappelez-vous, Jenkins enverra aussi un nouvel email dès le premier build réussi après une série de builds échoués ou instables, afin d'indiquer que le problème a été résolu.

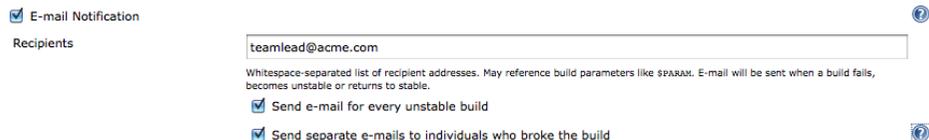


Figure 8.1. Configurer les notifications par email

Normalement, un build ne devrait pas prendre trop d'essais pour fonctionner à nouveau — les développeurs doivent diagnostiquer et reproduire le problème localement, le corriger, et alors seulement committer leur correction dans le gestionnaire de versions. Des erreurs répétées de build indiquent généralement une erreur de configuration chronique ou de mauvaises pratiques de développement (par exemple, des développeurs qui soumettent leurs modifications sans vérifier au préalable leur fonctionnement en local).

Vous pouvez également choisir d'envoyer un email distinct à tous les développeurs qui ont des changements committés dans le build échoué. C'est généralement une bonne idée, car les développeurs qui ont committé des changements depuis le dernier build sont naturellement les personnes qui devraient être les plus intéressées par les résultats du build. Jenkins va récupérer l'adresse email de l'utilisateur à partir du domaine de sécurité couramment configuré (voir Section 7.4, “Domaines de sécurité — Identifier les utilisateurs Jenkins”), ou en dérivant l'adresse email à partir de l'utilisateur de l'outil de gestion de versions s'il a été configuré (voir Section 4.8, “Configurer le serveur de messagerie électronique”).

Si vous utilisez cette option, il peut être moins pertinent d'inclure l'équipe entière dans la liste de diffusion principale. Vous incluez de préférence les personnes qui seront intéressées par le suivi du résultat de chaque build (tels que les responsables techniques), et laisser Jenkins informer les développeurs contributeurs directement.

Cela implique que les changements ont provoqué l'échec du build, ce qui est généralement (mais pas toujours) le cas. Cependant, si les builds sont peu fréquents (par exemple les builds nocturnes, ou si un build est en attente pendant plusieurs heures avant d'être abandonné), de nombreux changements peuvent être committés. Il est alors difficile de savoir quel est le développeur responsable de l'échec du build.

Tous les builds ne sont pas semblables par rapport aux notifications par email. Les développeurs qui ont soumis des changements sont particulièrement intéressés par les résultats des builds des tests unitaires et des tests d'intégration (surtout ceux déclenchés par leurs propres modifications). Les testeurs seront eux plus intéressés par jeter un œil sur le statut des tests d'acceptation automatisés. La configuration des notifications par email sera donc différente pour chaque étape de build. En fait, il est utile de définir des stratégies de notification par email. Par exemple,

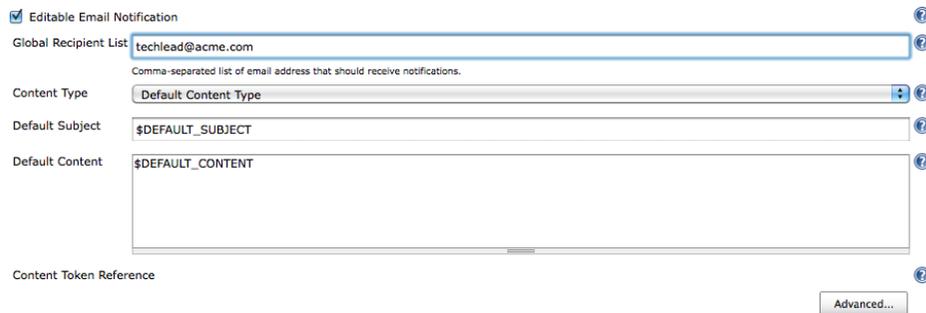
- pour les builds rapides (les tests unitaires/d'intégration s'exécutent en moins de 5 minutes) : les notifications sont envoyées aux chefs d'équipes et aux développeurs qui ont committé des changements,
- pour les builds lents (les builds de tests d'acceptation s'exécutent après les builds rapides) : les notifications sont envoyées aux chefs d'équipes, aux testeurs, ainsi qu'aux développeurs qui ont committé des changements.
- pour les builds nocturnes (les métriques de qualité, tests de performance et autres s'exécutent uniquement si les autres builds fonctionnent): les notifications sont envoyées à tous les membres d'équipe — Ceux-ci fournissent une photo instantanée de la santé du projet avant la réunion quotidienne.

En fait, vous devriez adopter au cas par cas la stratégie de notification appropriée pour chaque tâche de build, plutôt que d'appliquer une politique globale pour toutes les tâches de build.

8.3. Notification par email avancée

Par défaut, la notification par email Jenkins est un outil plutôt brut. Les messages de notification sont toujours envoyés au même groupe de personnes. Vous ne pouvez pas envoyer des messages à différentes personnes en fonction de ce qui a mal tourné. De la même façon, vous ne pouvez pas mettre en œuvre des politiques d'escalade. Par exemple, il pourrait être utile d'être en mesure de notifier les développeurs qui ont committé les modifications la première fois qu'un build échoue, et envoyer un message différent au chef d'équipe ou à l'équipe entière si le build échoue une seconde fois

Le plugin Email-ext vous permet de définir une stratégie de notification par email plus fine. Ce plugin ajoute une case Editable Email Notification (voir Figure 8.2, “Configurer les notifications par email avancées”), qui remplace efficacement la notification par email standard de Jenkins. Ainsi, vous pouvez définir une liste de destinataires par défaut et affiner le contenu du message électronique. Vous pouvez aussi définir une stratégie de notification plus précise avec des messages différents pour des listes de destinataires différents suivant les événements. Notez qu'une fois que vous avez installé et configuré ce plugin pour votre tâche de build, vous pouvez désactiver la configuration normale de notification par email.



The screenshot shows the configuration interface for the 'Editable Email Notification' plugin. It includes a checked checkbox for 'Editable Email Notification', a text input for 'Global Recipient List' containing 'techlead@acme.com', a dropdown for 'Content Type' set to 'Default Content Type', and text inputs for 'Default Subject' and 'Default Content', both containing '\$DEFAULT_SUBJECT' and '\$DEFAULT_CONTENT' respectively. There is also a 'Content Token Reference' section and an 'Advanced...' button.

Figure 8.2. Configurer les notifications par email avancées

Ce plugin a deux fonctionnalités liées mais distinctes. Tout d'abord, il vous permet de personnaliser le message de notification par email. Vous pouvez choisir parmi un grand nombre de mots clés prédéfinis pour créer vos propres titres et corps de messages personnalisés. Vous incluez un mot clé dans votre modèle de message en utilisant la notation habituelle dollar (e.g., `${BUILD_NUMBER}` ou `$BUILD_NUMBER`). Certains mots clés acceptent des paramètres que vous pouvez spécifier en utilisant le format `name=value` (ex : `${BUILD_LOG, maxLines=100}` ou `${ENV, var="PATH"}`). Les mots clés les plus utiles sont:

```
${DEFAULT_SUBJECT}
```

Le sujet par défaut configuré dans la page de configuration Jenkins

`${DEFAULT_CONTENT}`

Le corps du message par défaut configuré dans la page de configuration Jenkins

`${PROJECT_NAME}`

Le nom du projet

`${BUILD_NUMBER}`

Le numéro de build courant

`${BUILD_STATUS}`

Le statut du build courant (échec, succès, etc.)

`${CAUSE}`

La cause du build

`${BUILD_URL}`

Un lien vers la page correspondante du build sur Jenkins

`${FAILED_TESTS}`

Information sur les tests unitaires échoués, si certains ont échoués

`${CHANGES}`

Changements effectués depuis le dernier build

`${CHANGES_SINCE_LAST_SUCCESS}`

Tous les changements effectués depuis le dernier build avec succès

Pour obtenir la liste complète des mots clés disponibles ainsi que leurs options pour ceux qui acceptent des paramètres, cliquez sur l'icône d'aide en face de Contexte Token Reference.

Le bouton Avancé vous permet de définir une stratégie de notification plus sophistiquée basée sur le concept de déclencheurs (voir Figure 8.3, “Configurer les déclencheurs de notification par email”). Les déclencheurs déterminent quand les notifications par email doivent être envoyées. Les déclencheurs supportés sont les suivants :

Failure

Chaque fois qu'un build échoue.

Still Failing

Pour tous les builds qui restent en échec par la suite.

Unstable

Chaque fois qu'un build est instable.

Still Unstable

Pour tous les builds qui restent instables par la suite.

Success

Pour chaque build en succès.

Fixed

Quand un build passe d'échec ou instable à succès.

Before Build

Avant chaque début de build.

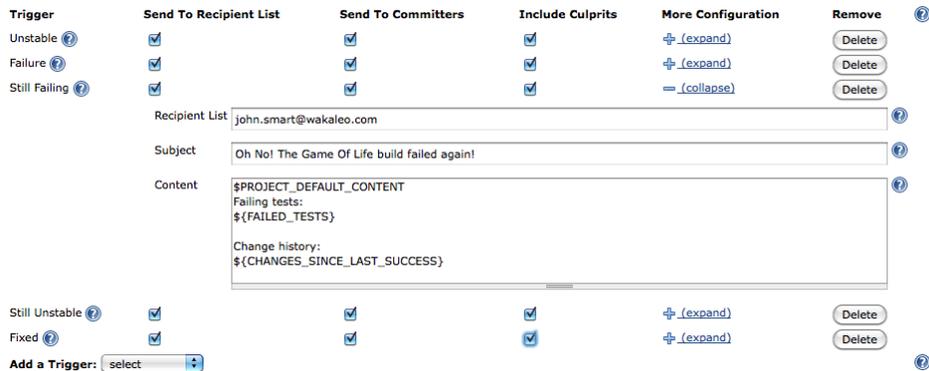


Figure 8.3. Configurer les déclencheurs de notification par email

Vous pouvez configurer autant (ou aussi peu) de déclencheurs que vous le souhaitez. La liste des destinataires et le modèle de message peuvent être personnalisés pour chaque déclencheur, par exemple en utilisant les déclencheurs Still Failing et Still Unstable, vous pouvez mettre en place une stratégie de notification qui n'avertit que le développeur qui a commis des changements la première fois qu'une tâche de build échoue, mais continue par informer le chef d'équipe si elle échoue une seconde fois. Vous pouvez choisir d'envoyer le message uniquement à des développeurs qui ont commis lors du build (« Send to committers »), ou d'inclure également tous ceux qui ont commis depuis le dernier build avec succès. Cela assure que tous ceux qui peuvent être impliqués dans l'échec du build seront notifiés de façon appropriée.

Vous pouvez également personnaliser le contenu du message en cliquant sur l'option More Configuration (comme indiqué pour le déclencheur Still Failing dans Figure 8.3, « Configurer les déclencheurs de notification par email »). De cette façon, vous pouvez personnaliser des messages différents qui seront envoyés dans des cas distincts.

Les déclencheurs interagissent intelligemment entre eux. Donc, si vous configurez à la fois le déclencheur Failing et le déclencheur Still Failing, seul le déclencheur Still Failing sera activé lors du second échec de build.

Un exemple d'un tel message personnalisé est illustré dans Figure 8.4, « Message de notification personnalisé ».

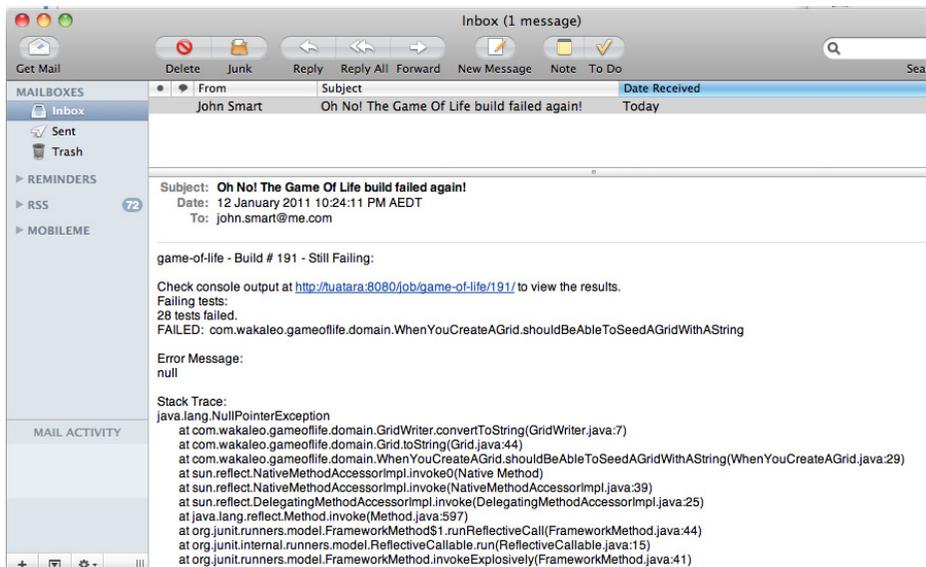


Figure 8.4. Message de notification personnalisé

Toutefois, l'email n'est pas une stratégie de notification sans défaut. Certains développeurs ferment par moment leurs clients de messagerie afin d'éviter d'être interrompu. Dans les grandes organisations, le nombre d'emails qui arrivent chaque jour peut être considérable et les notifications d'échec de build peuvent être cachées parmi une foule d'autres messages moins importants. Les échecs de build peuvent donc ne pas toujours avoir l'attention prioritaire qu'ils nécessitent dans un environnement d'intégration continue finement réglé. Dans les sections suivantes, nous nous pencherons sur certaines stratégies de notification autres qui peuvent être utilisées pour augmenter la sensibilisation des équipes sur les builds échoués et encourager les développeurs à les corriger plus rapidement.

8.4. Revendiquer des builds

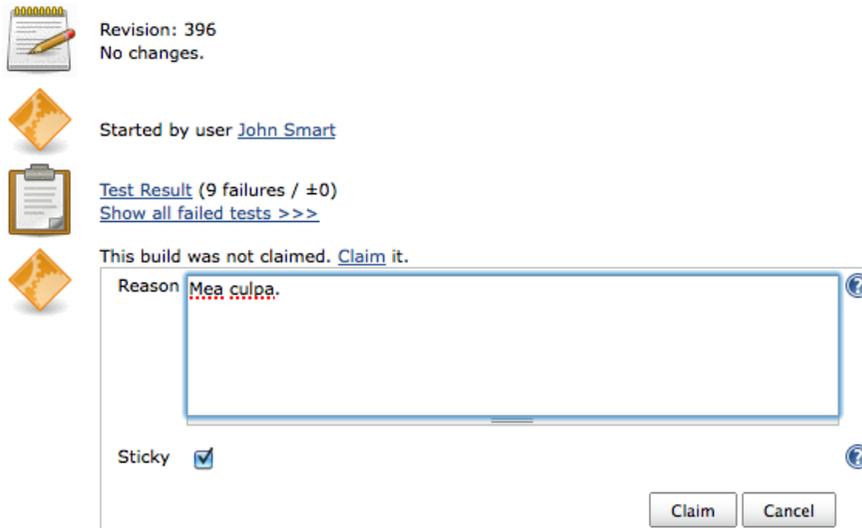
Quand un build échoue, il peut être utile de savoir que quelqu'un a repéré le problème et y travaille dessus. Cela évite d'avoir plusieurs développeurs qui gaspillent leur temps à essayer de corriger le même problème, chacun de leur côté.

Le plugin Claim permet aux développeurs d'indiquer qu'ils se sont appropriés un build cassé et qu'ils essaient de le réparer. Vous pouvez installer ce plugin de la façon habituelle. Une fois installé, les développeurs peuvent revendiquer un build échoué comme le leur. Ils peuvent éventuellement ajouter un commentaire pour expliquer la cause suspectée de l'échec du build et ce qu'ils ont l'intention de faire à ce sujet. Le build revendiqué sera alors marqué comme tel dans l'historique des builds afin d'éviter aux autres développeurs de gaspiller inutilement du temps à investiguer.

Pour activer la revendication pour une tâche de build, vous devez cocher la case "Allow broken build claiming" dans la page de configuration de la tâche de build. Alors, vous pourrez revendiquer un build cassé dans la page de détails du build (voir Figure 8.5, "Revendiquer un build échoué"). Les

builds revendiqués seront affichés avec une icône dans l'historique des builds indiquant qu'ils ont été revendiqués. Vous pouvez aussi effectuer une revendication de build “sticky” afin que tous les échecs ultérieurs de build pour cette tâche soient aussi automatiquement revendiqués par ce développeur, et ce, jusqu'à ce que le problème soit résolu.

Build #194 (Jan 13, 2011 12:11:46 PM)



Revision: 396
No changes.

Started by user [John Smart](#)

Test Result (9 failures / ±0)
[Show all failed tests >>>](#)

This build was not claimed. [Claim it.](#)

Reason Mea culpa.

Sticky

[Claim](#) [Cancel](#)

Module Builds

 gameoflife	0.76 sec
 gameoflife-build	1.5 sec
 gameoflife-cli	1 ms
 gameoflife-core	3 sec
 gameoflife-web	1 ms
 gameoflife-webservice	1 ms

Figure 8.5. Revendiquer un build échoué

8.5. Flux RSS

Jenkins fournit aussi des flux RSS pratiques pour les résultats de build, tant pour les résultats globaux sur l'ensemble de vos builds (ou juste les builds d'une vue particulière), que pour les résultats d'un build spécifique. Les icônes de flux RSS sont disponibles au bas des tableaux de bord des builds (voir Figure 8.6, “Flux RSS dans Jenkins”) et au bas du panneau de l'historique des builds pour les tâches de build individuel. Ils vous donnent soit accès à l'ensemble des résultats des builds, soit accès simplement aux builds échoués.

S	W	Job ↓	Last Success	Last Failure	Last Duration	
		deployment	38 min (#3)	N/A	0.1 sec	
		integration-tests	38 min (#4)	N/A	0.1 sec	
		unit-tests-build	38 min (#7)	N/A	66 ms	

Icon: [S](#) [M](#) [L](#)

Legend  for all  for failures  for just latest builds

Figure 8.6. Flux RSS dans Jenkins

Les URLs des flux RSS sont simples, et fonctionnent pour toute page Jenkins affichant un ensemble de résultats de build. Vous devez juste rajouter `/rssAll` pour obtenir le flux RSS de tous les résultats de build d'une page, ou `/rssFailed` pour n'obtenir que les résultats des builds échoués. Enfin, `/rssLatest` vous fournira un flux RSS contenant uniquement les derniers résultats de build. Mais la façon la plus simple de récupérer l'URL est de cliquer simplement sur l'icône RSS sur la page Jenkins correspondante.

Il y a pléthore de lecteurs RSS, à la fois commerciaux et open source, disponibles pour pratiquement toutes les plates-formes et périphériques. Ce peut être un excellent moyen pour garder un œil sur les résultats de build. La plupart des navigateurs (Firefox en particulier) et des clients email supportent les flux RSS. Certains lecteurs ont des problèmes avec l'authentification. Si votre instance Jenkins est sécurisée, il vous faudra peut-être faire un peu de configuration supplémentaire pour voir les résultats de votre build.

Les flux RSS peuvent être une source d'information sur l'ensemble des résultats de build, et vous permettent de voir l'état de vos builds en un coup d'œil sans avoir à se connecter au serveur. Néanmoins, la plupart des lecteurs RSS sont par nature passifs - vous pouvez consulter l'état de vos builds, mais le lecteur RSS ne sera généralement pas en mesure de vous notifier si un nouveau build en échec apparaît.

8.6. Radars de build

Le concept de radar d'informations est couramment utilisé dans les cercles agiles. Selon le gourou agile Alistair Cockburn:

Un radar d'information est un écran affiché dans un endroit que les gens peuvent voir quand ils travaillent ou passent à proximité. Il présente aux lecteurs les informations dont ils se soucient, sans avoir à poser de question à quelqu'un. Cela signifie plus de communication avec moins d'interruptions.

Dans le contexte d'un serveur d'intégration continue, un radar d'informations est un dispositif ou affichage important qui permet aux membres de l'équipe ou à d'autres de facilement voir si l'un des builds est actuellement cassé. Il montre généralement soit un résumé de tous les résultats du build courant, soit seulement ceux en échec, et est affiché sur un grand écran plat situé bien en vue sur un mur. Cette sorte de radar d'informations spécialisé est souvent connu comme un radar de build.

Utilisés correctement, les radars de build sont parmi les stratégies de notification passive les plus efficaces pour que tout le monde soit conscient des build échoués. En outre, contrairement à certains des périphériques de retours extrêmes dont nous discuterons plus tard dans ce chapitre, un radar de build peut contenir plusieurs tâches de build, y compris plusieurs tâches de build échouées, et peut donc encore être efficacement utilisé dans un contexte d'équipes multiples.

Il y a plusieurs solutions de radars de build pour Jenkins. Une des plus simples est d'utiliser le plugin Jenkins Radiator View. Ce plugin ajoute un nouveau type de tâche que vous pouvez créer: le (voir Figure 8.7, "Créer une vue radar").

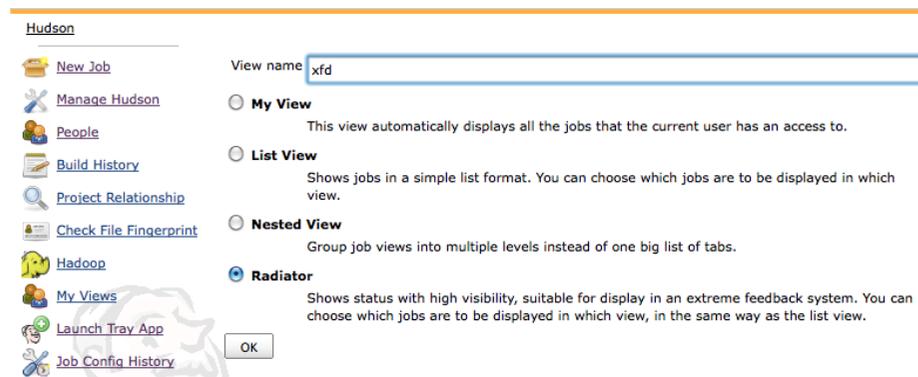


Figure 8.7. Créer une vue radar

Configurer la vue radar est similaire à la configuration d'une vue liste plus conventionnelle - vous devez simplement spécifier les tâches de build que vous voulez inclure dans la vue, en les choisissant une à une ou en utilisant une expression régulière.

Comme la vue radar occupe tout l'écran, modifier ou supprimer une vue radar est un peu délicat. En fait, la seule façon d'ouvrir l'écran de configuration de la vue est d'ajouter `/configure` à l'URL de la vue : si votre radar est nommé "build-radiator," vous pouvez éditer la configuration de la vue en ouvrant `http://my.hudson.server/view/build-radiator/configure`.

La vue radar (voir Figure 8.8, "Afficher une vue radar") affiche une grande boîte rouge ou jaune pour chaque build échoué ou instable. Ces boîtes contiennent le nom de la tâche de build en lettres capitales ainsi que d'autres détails. Vous pouvez configurer la vue radar pour afficher les builds en succès et les builds en échec (ils seront affichés dans de petites boîtes vertes). Cependant, un bon radar ne devrait afficher que les builds échoués, à moins que tous les builds soient en succès.



Figure 8.8. Afficher une vue radar

8.7. Messagerie instantanée

La messagerie instantanée (ou IM) est aujourd'hui largement utilisée comme un moyen rapide et léger de communication, aussi bien professionnelle que personnelle. La messagerie instantanée est, par définition, instantanée, ce qui lui donne un avantage sur l'email quand il s'agit de notification rapide. Elle est également «push» plutôt que «pull». Lorsque vous recevez un message, il apparaîtra sur votre écran et attirera votre attention. Il est un peu plus difficile de l'ignorer ou de reporter qu'un simple message email.

Jenkins offre un bon support pour les notifications via messagerie instantanée. Le plugin Instant Messaging fournit un support générique pour communiquer avec Jenkins via la messagerie instantanée. Des plugins spécifiques peuvent ensuite être ajoutés pour différents protocoles de messagerie instantanée tels que Jabber ou IRC.

8.7.1. Notification par IM avec Jabber

De nombreux serveurs de messagerie instantanée sont basés sur Jabber, un protocole de messagerie instantanée open source et basé sur XML. Jenkins fournit un bon support pour la messagerie instantanée Jabber, de telle sorte que les développeurs peuvent recevoir des notifications en temps réel des échecs de build. De plus, le plugin exécute un robot de messagerie instantanée qui écoute les canaux de communication et permet aux développeurs d'exécuter des commandes sur le serveur Jenkins via des messages instantanés.

La mise en place du support de messagerie instantanée dans Jenkins est simple. D'abord, vous devez installer à la fois le plugin Jenkins instant-messaging et le plugin Jenkins Jabber Notifier en utilisant la page standard du gestionnaire de plugins et en redémarrant Jenkins (voir Figure 8.9, "Installation des plugins Jenkins de messagerie instantanée").

✓	Hudson instant-messaging plugin This plugin provides abstract support for build notification via instant-messaging.	1.13
✓	Hudson Jabber notifier plugin Sends build notifications to jabber contacts and/or chatrooms. Also allows control of builds via a jabber 'bot'. Note that the instant-messaging plugin 1.11 is a requirement for this plugin. <i>Please make sure that it is installed, too!</i>	1.13

Figure 8.9. Installation des plugins Jenkins de messagerie instantanée

Une fois cela fait, vous devez configurer votre serveur de messagerie instantanée. N'importe quel serveur Jabber peut faire l'affaire. Vous pouvez utiliser un service public comme Google Chat, ou configurer votre propre serveur de messagerie instantanée localement (le serveur de messagerie instantanée open source Java OpenFire¹ est un bon choix). Utiliser un service public pour les communications internes peut être mal vu par les administrateurs système, et vous pouvez avoir des difficultés pour passer les firewalls de l'entreprise. D'autre part, configurer votre propre service de messagerie interne a du sens pour une équipe de développement ou toute autre organisation en général, car il fournit un autre canal de communication qui fonctionne bien pour partager des questions techniques ou commentaires entre développeurs. Les exemples suivants utiliseront un serveur OpenFire local, mais l'approche générale fonctionne pour tout serveur Jabber- compatible .

La première étape consiste à créer un compte dédié sur votre serveur Jabber pour Jenkins. Celui-ci est juste un compte de messagerie instantanée ordinaire, mais il doit être distinct des comptes de vos développeurs (voir Figure 8.10, “Jenkins nécessite son propre compte de messagerie instantanée”).

User Summary

Total Users: 7 -- Sorted by Username -- Users per page:

	Online	Username	Name	Created	Last Logout	Edit	Delete
1		admin ★	Administrator	Jan 13, 2011			
2		bill	Bill Smith	Jan 14, 2011			
3		hudson	hudson	Jan 13, 2011	31 minutes		
4		joe	Joe Black	Jan 13, 2011			
5		johnsmart ★	John Smart	Jan 13, 2011			
6		kate	Kate Brown	Jan 14, 2011			
7		pete	Pete Best	Jan 14, 2011			

Figure 8.10. Jenkins nécessite son propre compte de messagerie instantanée

Une fois que vous avez configuré un compte de messagerie instantanée, vous devez configurer Jenkins pour envoyer des notifications par message instantané via ce compte. Allez à la page de configuration

¹ <http://www.igniterealtime.org/projects/openfire/index.jsp>

principale et cochez la case Enable Jabber Notification (voir Figure 8.11, “Mise en place de notifications de base Jabber dans Jenkins”). Ici, vous fournissez l'identifiant Jabber et le mot de passe pour votre compte. Jenkins peut habituellement retrouver le serveur de messagerie à partir de l'identifiant Jabber (s'il est différent, vous pouvez le remplacer dans les options avancées). Si vous utilisez les canaux de communication en groupe (une autre stratégie de communication utile pour les équipes de développement), vous pouvez aussi renseigner ici le nom de ces salons de discussion. De cette façon, Jenkins sera en mesure de traiter les instructions envoyées dans les salons de chat, ainsi que ceux reçus comme des messages directs.

Jabber Notification

Enable Jabber Notification

Jabber ID 

Password

Initial group chats

Group chats to automatically join on startup with a bot (whitespace separated)

Figure 8.11. Mise en place de notifications de base Jabber dans Jenkins

C'est tout ce dont vous avez besoin pour une configuration de base. Cependant, vous devrez peut-être donner quelques informations supplémentaires dans la section avancée pour des détails qui sont spécifiques à votre installation (voir Figure 8.12, “Configuration avancée Jabber”). Ici, vous pouvez spécifier le nom et le port de votre serveur Jabber s'ils ne peuvent être dérivés de l'identifiant Jenkins Jabber. Vous pouvez également fournir un suffixe par défaut qui peut être appliqué à l'utilisateur Jenkins pour générer l'identifiant Jabber correspondant. Surtout, si vous avez sécurisé votre serveur Jenkins, vous devrez fournir un nom d'utilisateur et un mot de passe Jenkins valides afin que le robot de messagerie instantanée puisse réagir correctement aux instructions.

Jabber Notification

Enable Jabber Notification

Jabber ID ?

Password

Initial group chats
Group chats to automatically join on startup with a bot (whitespace separated)

Server ?

Port ?

Default ID suffix
This suffix will be used to determine the Jabber ID from the Hudson ID, if no Jabber ID is specified in the user settings

Enable SASL authentication ?

Expose presence ?

Acceptance mode for subscription requests ?

Bot command prefix ?

Group chat nickname ?

Hudson Username ?

Hudson Password

Figure 8.12. Configuration avancée Jabber

Une fois configuré, vous devez définir une stratégie de notification Jabber pour chacune de vos tâches de build. Ouvrez la page de configuration de tâche de build et cliquez sur l'option Jabber Notification.

D'abord, vous devez définir une liste de destinataires pour les messages. Vous pouvez envoyer des messages à des individus (utilisez simplement l'identifiant Jabber correspondant, tel que `joe@jabber.acme.com`) ou à des salons de messagerie instantanée que vous avez créé. Pour les salons, vous devez normalement ajouter une “*” au début de son identifiant (ex : “*gameoflife@conference.jabber.acme.org”). Cependant, si cet identifiant contient “@conference.”, Jenkins comprendra qu'il s'agit d'un salon de messagerie instantanée et ajoutera automatiquement l’“*”. L'approche basée sur un salon est la plus souple bien que pour que cette stratégie soit vraiment efficace, vous devez être sûr que les développeurs y sont connectés en permanence.

Vous devez aussi définir une stratégie de notification. Celle-ci détermine lesquels de vos résultats de build provoqueront un envoi de message. Les options sont:

all

Envoie une notification pour chaque build.

failure

Envoie une notification uniquement pour les builds échoués et instables.

failure and fixed

Envoie une notification pour les builds échoués et instables, et le premier build en succès après un build échoué ou instable.

change

Envoie une notification quand le résultat de build change.

Si vous utilisez les salons de messagerie instantanée, vous pouvez demander à Jenkins d'envoyer une notification au salon lorsqu'un build démarre (en utilisant l'option “Notify on build starts”).

En ce qui concerne les builds démarrés par les systèmes de gestion de version, Jenkins peut aussi notifier des destinataires supplémentaires. Pour cela, il faut utiliser le suffixe par défaut décrit précédemment afin de générer l'identifiant Jabber à partir de l'utilisateur du système de gestion de version. Vous pouvez choisir de notifier :

SCM committers

Tous les utilisateurs qui ont committé des changements pour le build courant, et donc soupçonnés d'avoir cassé le build.

SCM culprits

Tous les utilisateurs qui ont committé des changements depuis le dernier build avec succès.

SCM fixers

Tous les utilisateurs qui ont committé des changements du premier build avec succès après un build échoué ou instable.

Upstream committers

Envoie une notification aux utilisateurs qui ont committé des changements pour les builds en amont et courant. Cela fonctionne automatiquement pour les tâches de build Maven, mais nécessite d'activer l'empreinte numérique (fingerprint) des autres types de systèmes de build.

Au moment de la rédaction, vous ne pouvez définir qu'une stratégie de notification. Ainsi, certaines des options avancées que nous avons vu dans Section 8.3, “Notification par email avancée” ne sont pas encore disponibles pour la messagerie instantanée.

Les développeurs seront notifiés via leur client de messagerie instantanée favori (voir Figure 8.13, “Messages Jenkins Jabber en action”). Ils peuvent aussi interagir avec le serveur de build via une session de messagerie en utilisant un ensemble de commandes simples dont voici les plus utiles :

- `!build game-of-life`—Démarre le build game-of-life immédiatement.
- `!build game-of-life 15m`—Démarre le build game-of-life dans 15 minutes.
- `!comment game-of-life 207 'oops'`—Ajouter une description au build défini.
- `!status game-of-life`—Affiche le status du dernier build de cette tâche de build.
- `!testresult game-of-life`—Affiche le résultat complet du dernier build.

- `!health game-of-life`—Affiche un résumé plus complet de l'état de santé du dernier build.

Vous pouvez obtenir une liste complète des commandes en envoyant le message `!help` à l'utilisateur Jenkins.



Figure 8.13. Messages Jenkins Jabber en action

8.7.2. Notification avec IRC

Une autre forme de messagerie instantanée Internet populaire est Internet Relay Chat, ou IRC. IRC est traditionnellement centré sur les groupes de discussions (même si la messagerie directe est également supportée). Elle est une forme de communication très populaire parmi les développeurs, en particulier dans le monde open source.

Le plugin Jenkins IRC vous permet d'interagir avec votre serveur Jenkins via un canal IRC, à la fois pour recevoir des messages de notification, mais aussi pour envoyer des commandes au serveur. Comme le plugin Jabber, vous devez installer le plugin Instant Messaging pour que le plugin Jenkins IRC fonctionne.

8.8. Notification par IRC

Rédigé par Juven Xu

Internet Relay Chat (ou IRC) est une forme populaire de messagerie instantanée, conçue principalement pour la communication de groupes par canaux. Par exemple, Jenkins a un canal sur Freenode² de telle

² <http://jenkins-ci.org/content/chat>

façon que les utilisateurs et les développeurs peuvent échanger sur des sujets liés à Jenkins. Vous verrez de nombreux utilisateurs poser des questions et la plupart du temps des utilisateurs plus expérimentés fournir des réponses rapidement.

Comme avec la messagerie instantanée Jabber, vous pouvez configurer Jenkins pour “pousser” des notifications via IRC. Quelques clients IRC tels que xchat³ supportent une configuration d'alerte de telle manière que lorsqu'un message arrive, il peut faire clignoter l'icône du panneau ou émettre un bip sonore. Pour mettre en place le support IRC dans Jenkins, vous devez d'abord installer le plugin IRC⁴ et le plugin Instant Messaging⁵. Allez simplement dans le gestionnaire de plugins par défaut, cochez les cases correspondantes et redémarrez ensuite Jenkins (voir Figure 8.14, “Installation des plugins Jenkins IRC”).

<input checked="" type="checkbox"/>	Instant Messaging Plugin This plugin provides generic support for build notifications and a bot via instant messaging protocols.	1.13
<input checked="" type="checkbox"/>	IRC Plugin This plugin installs Hudson IRC bot on your choice of IRC channels. You can get notifications via IRC and interact with Hudson via IRC. Note that you also need to install the instant-messaging plugin .	2.8

Figure 8.14. Installation des plugins Jenkins IRC

Une fois cela fait, vous devez activer le plugin IRC, et le configurer pour intégrer votre propre environnement. Fondamentalement, cela consiste à fournir le nom d'hôte et le port du serveur IRC que vous utilisez, un canal IRC dédié, et un surnom pour le plugin IRC. Une bonne pratique consiste à mettre en place un canal dédié pour les notifications provenant de l'IC. Ainsi, si les gens bavardent sur d'autres canaux, ils ne seront pas perturbés. Vous pouvez également configurer des détails supplémentaires dans la section Avancé. Tous ces éléments sont disponibles sur la page Configurer le système (voir Figure 8.15, “Configuration avancée des notifications par IRC”).

³ <http://xchat.org/>

⁴ <http://wiki.jenkins-ci.org/display/JENKINS/IRC+Plugin>

⁵ <http://wiki.jenkins-ci.org/display/JENKINS/Instant+Messaging+Plugin>

IRC Notification

Enable IRC Notification ?

Hostname ?
Hostname of the IRC server

Port ?
Port of the IRC server

Channels ?

Channels the bot should join

Nickname ?
Nickname of the bot

Password

NickServ Password ?
On connection, try to identify with NickServ with this password

Command prefix ?
The prefix for the commands

Hudson Username ?

Hudson Password

Use /notice command
Use /notice command instead of /msg (default in ircbot <= 2.0)

Figure 8.15. Configuration avancée des notifications par IRC

En plus du nom d'hôte, du port, du canal, et du surnom que nous avons mentionnés précédemment, vous pouvez également configurer le mot de passe du serveur IRC ou celui du NickServ si votre environnement les nécessite. Les commandes doivent être préfixées essentiellement de la même façon que pour Jabber (voir Section 8.7, “Messagerie instantanée”) si vous souhaitez interagir avec le serveur via des messages IRC. Enfin, vous voudrez peut-être configurer le plugin IRC pour utiliser la commande `/notice` en lieu et place de la commande par défaut `/msg`. `/notice` est identique à `/msg` excepté que le message sera encadré de tirets, ce qui évitera une réponse de la plupart des robots.

Une fois que la configuration globale est prête, vous pouvez activer la notification par IRC pour chaque tâche de build et mettre en place une stratégie de notification. Ouvrez la page de configuration de tâche de build, allez à la section Actions à la suite du build et cliquez sur l'option IRC Notification. Si vous souhaitez configurer une stratégie de notification plutôt que d'utiliser celle par défaut, cliquez sur le bouton "Avancé..." (voir Figure 8.16, “Configuration avancée de notifications par IRC pour une tâche de build”).

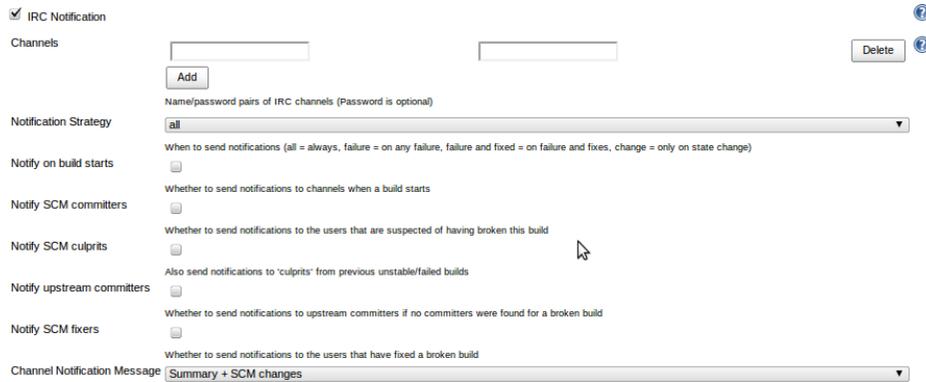


Figure 8.16. Configuration avancée de notifications par IRC pour une tâche de build

Les stratégies de notification (quand et à qui envoyer des messages de notification) sont décrites dans Section 8.7, “Messagerie instantanée”. Le plugin Jabber ainsi que le plugin IRC dépendent du plugin Instant Messaging. Ils partagent donc un certain nombre de caractéristiques fondamentales communes. Certaines options sont toutefois spécifiques à l'extension IRC. Par exemple, vous pouvez définir un canal personnalisé si vous n'aimez pas la valeur globale par défaut. De plus, pour un message de notification envoyé à un canal, vous pouvez choisir les informations à transmettre dans les messages de notification. Vos options ici sont le résumé du build, les changements effectués via le système de gestion de version, et les tests échoués.

Une fois que vous enregistrez la configuration, tout est prêt. Basé sur ce que vous avez configuré, ce plugin va rejoindre les canaux IRC appropriés et envoyer des messages de notification pour les tâches de build.

Par exemple, dans Figure 8.17, “Messages de notification par IRC en action”, le plugin IRC rejoint le canal #ci-book sur freenode. Tout d'abord, l'utilisateur juven a committé quelques changements avec le message "feature x added" et le plugin IRC notifie tous les connectés au canal que le build a été un succès. Ensuite, juven committe un autre changement pour la fonctionnalité y, mais cette fois le build a échoué. John a remarqué et corrigé l'erreur de build. Le plugin IRC déclare maintenant "Yippie, build fixed!" Notez que certaines lignes de cet écran sont soulignées, c'est parce que je me suis connecté en tant qu'utilisateur "juven" et j'ai configuré mon client IRC XChat pour mettre en évidence les messages contenant mon surnom.

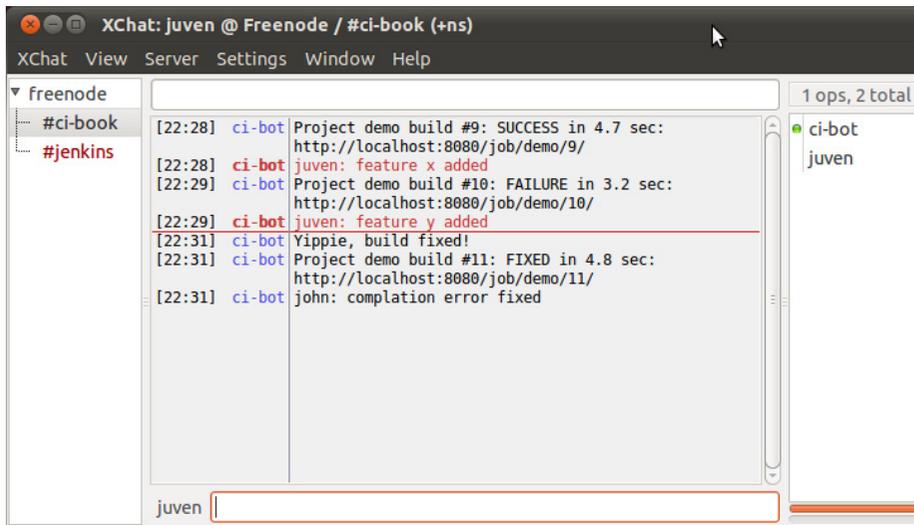


Figure 8.17. Messages de notification par IRC en action

8.9. Notificateurs de bureau

Les meilleures stratégies de notification push s'intègrent en douceur dans l'environnement de travail quotidien du développeur. C'est pourquoi la messagerie instantanée peut être une stratégie efficace si les développeurs ont déjà l'habitude d'utiliser des messageries instantanées pour les autres activités liées au travail.

Les outils de notification de bureau entrent aussi dans cette catégorie. Les outils de notification bureau sont des outils qui s'exécutent localement sur l'ordinateur du développeur, soit comme une application indépendante ou un widget, soit dans le cadre de l'outil de développement du développeur (IDE).

Si vous utilisez Eclipse, le plugin Eclipse Jenkins ⁶ affiche une icône de santé au bas de la fenêtre Eclipse. Si vous cliquez sur cette icône, vous pouvez voir une vue détaillée des projets Jenkins (voir Figure 8.18, "Notifications Jenkins dans Eclipse"). Dans les préférences d'Eclipse, vous indiquez l'URL de votre serveur Jenkins avec tous les détails d'authentification requis. La configuration est assez simple, cependant, vous ne pouvez vous connecter à une instance unique Jenkins pour un espace de travail Eclipse donné.

⁶ <http://code.google.com/p/hudson-eclipse/>

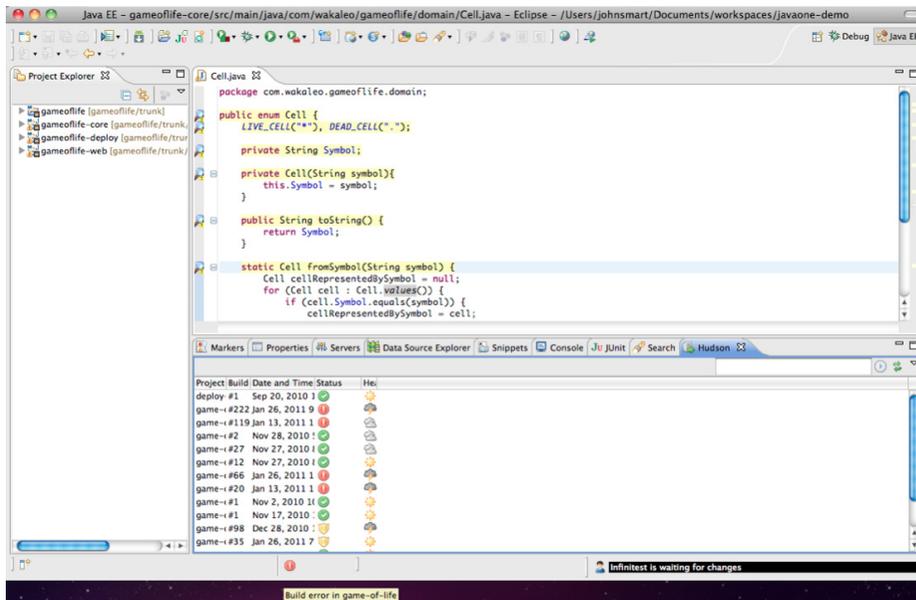


Figure 8.18. Notifications Jenkins dans Eclipse

Si vous utilisez l'IDE NetBeans, vous avez déjà l'intégration avec Hudson et Jenkins. Ouvrez la fenêtre Services et ajoutez des serveurs sous Hudson Builders. (Si vous ouvrez un projet Maven dont la section `ciManagement` indique `hudson` ou `jenkins` sous `system`, le serveur correspondant sera enregistré automatiquement.) Cette intégration a des caractéristiques différentes au-delà des notifications de build dans la barre d'état, telles que l'intégration dans la fenêtre Tests Results, l'affichage des journaux de build et des journaux de changement, la navigation dans l'espace de travail, et un assistant de configuration de tâche de build.

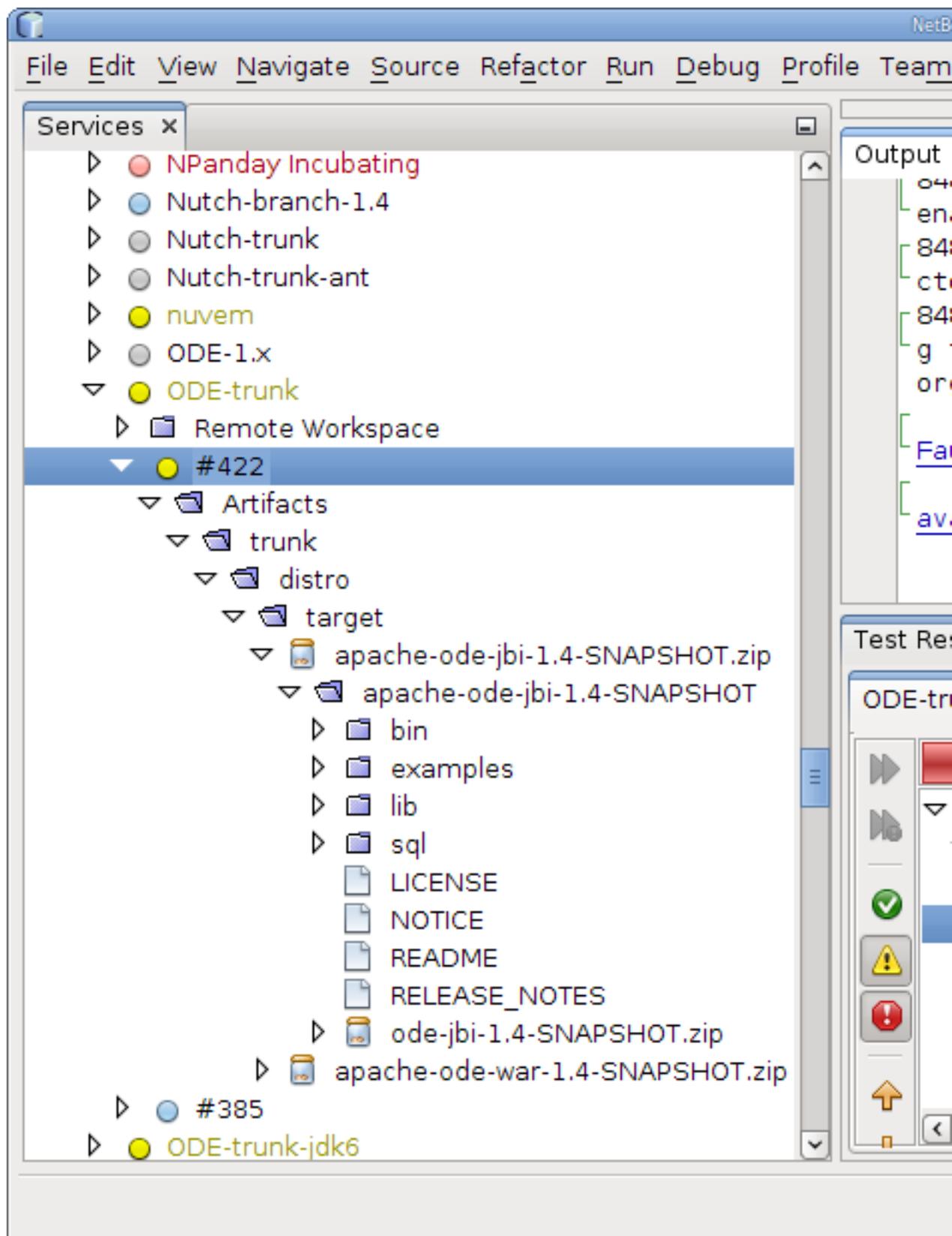


Figure 8.19. Connexion de Jenkins dans NetBeans

Le plugin Jenkins Tray Application (voir Figure 8.20, “Lancement de Jenkins Tray Application”) vous permet de démarrer une petite application cliente Java à l'aide de Java Web Start à partir de votre tableau de bord de Jenkins.



Figure 8.20. Lancement de Jenkins Tray Application

Cette application se trouve dans votre barre d'état du système et vous permet de visualiser l'état courant de vos builds en un coup d'œil. Elle apporte également des fenêtres pop-up vous informant des nouveaux échecs de build (voir Figure 8.21, “Exécution de Jenkins Tray Application”).

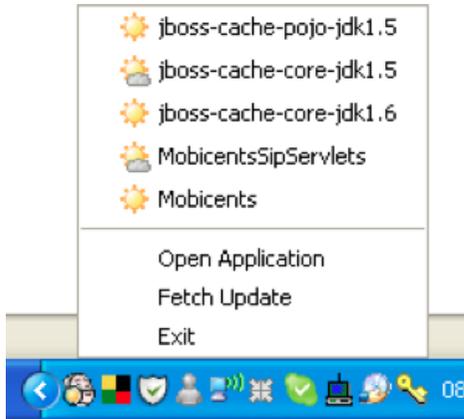


Figure 8.21. Exécution de Jenkins Tray Application

C'est certainement une application utile mais elle souffre de quelques limitations. Au moment de la rédaction, Jenkins Tray Application ne supporte pas l'accès aux serveurs sécurisés Jenkins. En outre, le développeur doit se souvenir de le redémarrer chaque matin. Cela peut sembler un problème mineur, mais en général, quand il s'agit de stratégies de notification, moins vous en demandez à vos développeurs meilleure est la solution.

Une des meilleures options pour les notifications Jenkins de bureau est d'utiliser un service comme Notifo (voir Section 8.10, "Notifications via Notifo") qui fournit des clients pour bureaux et mobiles. Nous allons voir comment cela fonctionne en détail dans la prochaine section.

8.10. Notifications via Notifo

Notifo⁷ est un service rapide et économique pour envoyer en temps-réel des notifications vers votre smartphone ou votre bureau. Dans le contexte d'un serveur Jenkins, vous pouvez l'utiliser pour mettre en place gratuitement ou à faible coût des notifications en temps réel pour vos résultats de builds Jenkins. Les comptes individuels (dont vous avez besoin pour être capable de recevoir des notifications) sont gratuits. Vous avez besoin de mettre en place un compte service pour envoyer des messages de notification de votre serveur Jenkins. C'est ici que Notifo devient payant, même si lors de la rédaction un compte service peut envoyer jusqu'à 10 000 notifications par mois gratuitement, ce qui est habituellement largement suffisant pour une instance moyenne Jenkins. Un des points forts d'un service de notification en temps réel comme Notifo est que les messages de notification peuvent être envoyés à ces mêmes utilisateurs sur différents dispositifs ; en particulier les smartphones et les clients de bureau.

La mise en place des notifications Jenkins avec Notifo est relativement simple. Tout d'abord, allez sur le site Notifio et inscrivez vous pour créer un compte. Chaque membre de l'équipe qui veut être notifié aura besoin de son propre compte Notifo. Ils auront également besoin d'installer le client Notifo sur

⁷ <http://www.notifo.com>

chaque appareil sur lequel ils ont besoin de recevoir des notifications. Au moment de l'écriture de ce livre, les clients Notifo étaient disponibles pour Windows et Mac OS X, ainsi que pour les iPhones. Le support pour les smartphones Linux et autres est en cours.

Ensuite, vous devez configurer un compte de service Notifo pour votre serveur Jenkins. Vous pouvez faire cela avec un de vos comptes développeur, ou créer un nouveau compte à cet effet. Connectez-vous au site Notifo, et allez au menu My Services. Ici, cliquez sur Create Service (voir Figure 8.22, “Créer un service Notifo pour votre instance Jenkins”) et remplissez les champs. Le plus important est que le nom d'utilisateur du service doit être unique. Vous pouvez également spécifier l'URL du site et l'URL de notification par défaut pour pointer vers votre instance Jenkins pour que les utilisateurs puissent ouvrir la console Jenkins en cliquant sur le message de notification.

The screenshot shows the 'Create Service' page on the Notifo website. The page has a blue header with the Notifo logo and navigation links: Notifications, Settings, Services, Learn More, and Mobile Apps. A user greeting 'Hey there, wakaleo! Log out?' is visible in the top right. The main content area is titled 'Create Service' and includes a sub-header 'Services My Services'. Below this, there is a brief explanation: 'Creating a Notifo service provides you with separate API credentials in addition to a service icon and URL shown throughout Notifo and our mobile applications.' A note says 'All fields required.' The form contains five input fields: 'Service Username' with the value 'wakaleo_labs_hudson_serve', 'Service Name' with 'Wakaleo Labs Hudson Serve', 'Email Address' with 'john.smart@wakaleo.com', 'Site URL' with 'http://www.wakaleo-labs.c', and 'Default Notification URL' with 'http://www.wakaleo-labs.c'. At the bottom right is an orange 'Create Service' button. At the bottom left, a small note reads: 'Notifo service pricing is not yet final but 10,000 notifications per month are free. Tinker away!'.

Figure 8.22. Créer un service Notifo pour votre instance Jenkins

Pour recevoir des messages de notification à partir du serveur Jenkins, les développeurs ont maintenant besoin de souscrire à ce service. Vous pouvez ensuite ajouter les développeurs à la liste des abonnés sur la page Subscribers du service en leur envoyant des demandes de souscription. Une fois que le service a été créé et que les utilisateurs sont tous abonnés, vous pouvez configurer votre projet pour envoyer des notifications Notifo (voir Figure 8.23, “Configurer les notifications via Notifo dans votre tâche de build Jenkins”). Vous avez besoin de fournir le nom d'utilisateur de l'API du service de Jenkins que vous avez mis en place, ainsi que l'API Secret. Vous pouvez les voir tous les deux dans le tableau de bord du service Notifo.

Notifo

Service User Name

API Token

Usernames to receive notifications (comma separated)

Send notifications for successful builds

Figure 8.23. Configurer les notifications via Notifo dans votre tâche de build Jenkins

Une fois que cela est mis en place, Jenkins enverra en quasi temps-réel les notifications d'échecs de build à tous les clients Notifo que le développeur a lancé, que ce soit sur un bureau ou sur un appareil mobile (voir Figure 8.24, “Recevoir une notification via Notifo sur un iPhone”).

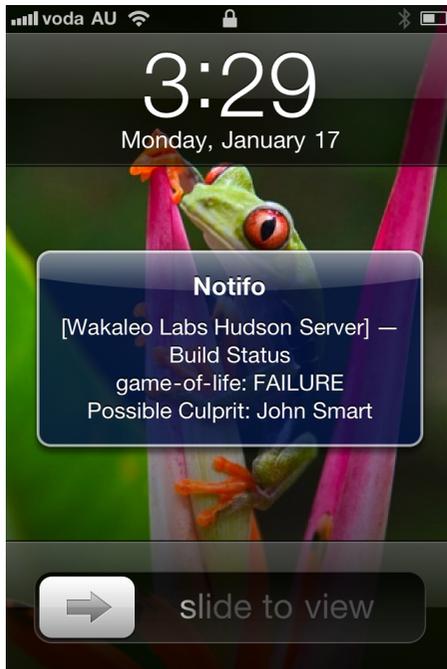


Figure 8.24. Recevoir une notification via Notifo sur un iPhone

Au moment de l'écriture de ce livre, les stratégies de notification sophistiquées ne sont pas prises en charge - vous ne pouvez fournir qu'une liste de noms d'utilisateurs Notifo qui doivent être notifiés. Néanmoins, cela reste un outil de notification très efficace pour les développeurs en ligne de front.

8.11. Notifications vers mobiles

Si votre serveur Jenkins est visible sur Internet (même si vous avez mis en place une authentification sur votre serveur Jenkins), vous pouvez aussi surveiller vos builds via votre appareil mobile iPhone ou Android. L'application gratuite Hudson Helper (voir Figure 8.25, “Utiliser l'application iPhone Hudson Helper”) par exemple, vous permet de lister vos tâches de builds actuelles (soit l'ensemble des tâches de builds sur le serveur, ou seulement les tâches de build d'une certaine vue). Vous pouvez également

afficher les détails d'une tâche de build particulière, y compris son statut actuel, les tests en échec et le temps de build, et même démarrer et arrêter les builds.

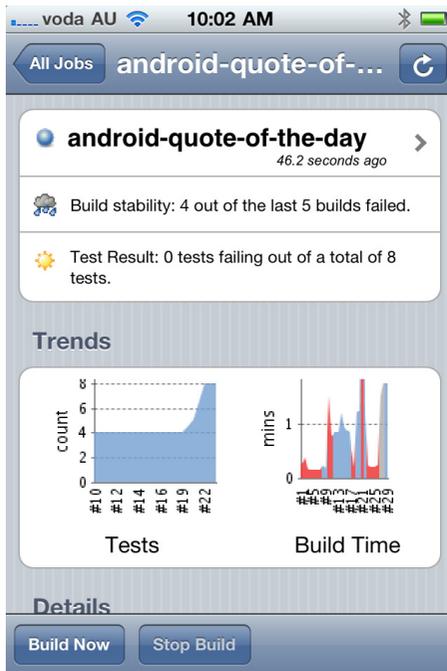


Figure 8.25. Utiliser l'application iPhone Hudson Helper

Pour les téléphones Android, vous pouvez également installer le widget Hudson Mood qui fournit également des mises à jour et des alertes sur les échecs de build.

Notez que ces applications mobiles s'appuient sur une connexion de données, de sorte qu'ils travaillent généralement bien localement, mais vous ne devriez pas compter sur eux si le développeur est à l'extérieur.

8.12. Notifications via SMS

Ces temps-ci, le SMS est un autre canal de communication universel qui a l'avantage supplémentaire d'atteindre les personnes même quand elles ne sont pas au bureau. Pour un ingénieur de build, ce peut être un excellent moyen de surveiller des builds critiques, même si les développeurs ou chefs d'équipe sont loin de leur bureau.

Les passerelles SMS⁸ sont des services qui permettent d'envoyer des notifications SMS via des adresses emails formatées spécialement (par exemple, 123456789@mysmsgateway.com pourrait envoyer un message SMS à 123456789). Beaucoup de vendeurs mobiles offrent ce service, tout comme beaucoup

⁸ http://en.wikipedia.org/wiki/SMS_gateway

de prestataires de services tiers. Il n'y a aucune prise en charge intégrée pour les passerelles SMS dans Jenkins, mais la fonctionnalité de base de ces passerelles rend l'intégration relativement simple : il vous suffit d'ajouter les adresses emails spéciales à la liste de notification normale. Sinon, en utilisant la configuration email avancée, vous pouvez configurer une règle distincte contenant uniquement les adresses email SMS (voir Figure 8.26, “Envoyer des notifications SMS via une passerelle SMS”). Procéder ainsi rend plus facile d'affiner le contenu du message pour adhérer au format des messages SMS.

The screenshot shows the 'Editable Email Notification' configuration page in Jenkins. It includes the following elements:

- Global Recipient List:** A text input field with a help icon.
- Content Type:** A dropdown menu set to 'Default Content Type' with a help icon.
- Default Subject:** A text input field containing '\$DEFAULT_SUBJECT' with a help icon.
- Default Content:** A large text area containing '\$DEFAULT_CONTENT' with a help icon.
- Content Token Reference:** A table with columns: Trigger, Send To Recipient List, Send To Committers, Include Culprits, More Configuration, and Remove.

Trigger	Send To Recipient List	Send To Committers	Include Culprits	More Configuration	Remove
Failure ?	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	collapse	Delete
	Recipient List: 6421032 @email.sms .com ?				
	Subject: ?				
	Content: \$PROJECT_DEFAULT_CONTENT ?				

At the bottom, there is an 'Add a Trigger:' dropdown menu set to 'select' with a help icon.

Figure 8.26. Envoyer des notifications SMS via une passerelle SMS

Une fois que vous avez fait cela, vos utilisateurs recevront une notification rapide des résultats de build sous forme de messages SMS (voir Figure 8.27, “Recevoir des notifications via SMS”). Le principal inconvénient de cette approche est sans doute que ce n'est pas gratuit, et nécessite l'utilisation d'un service commercial tiers. Cela dit, c'est vraiment la seule technique de notification capable d'atteindre les développeurs quand ils sont hors de portée d'Internet ou qu'ils n'ont pas de smartphone activé. En effet, cette technique est populaire parmi les administrateurs système, et peut être très utile pour certaines tâches de build.

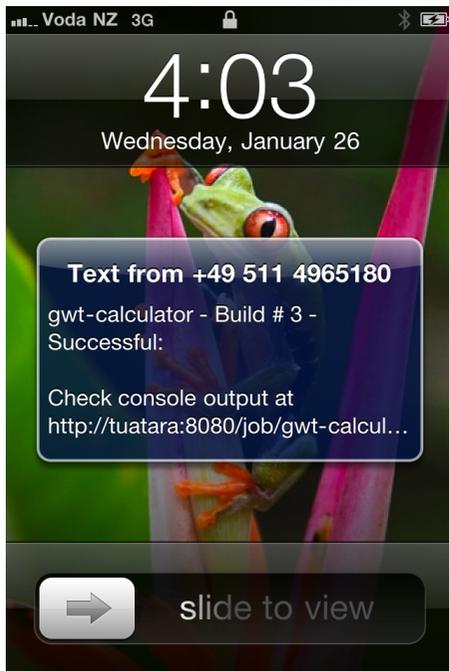


Figure 8.27. Recevoir des notifications via SMS

8.13. Faire du bruit

Si votre instance Jenkins s'exécute sur une machine qui est physiquement située à proximité de l'équipe de développement, vous pouvez aussi vouloir ajouter les sons dans l'ensemble des stratégies de notification. Cela peut être une stratégie efficace pour les petites équipes co-localisées, mais cela devient plus difficile si le serveur de build est mis en place sur une machine virtuelle ou ailleurs dans le bâtiment.

Il y a deux façons d'intégrer des retours audio dans votre processus de build Jenkins : le plugin Jenkins Sounds et le plugin Jenkins Speaks. Les deux peuvent être installés via la page du gestionnaire de plugins de la manière habituelle .

Le plugin Jenkins Sounds est le plus flexible des deux. Il vous permet de construire une stratégie de communication détaillée basée sur le dernier résultat de build et aussi (facultatif) sur le résultat du précédent build (voir Figure 8.28, “Configurer les règles de Jenkins Sounds dans une tâche de build”). Par exemple, vous pouvez configurer Jenkins pour jouer un son la première fois qu'un build échoue, un son différent si le build échoue une seconde fois, et encore un autre son lorsque le build est corrigé.

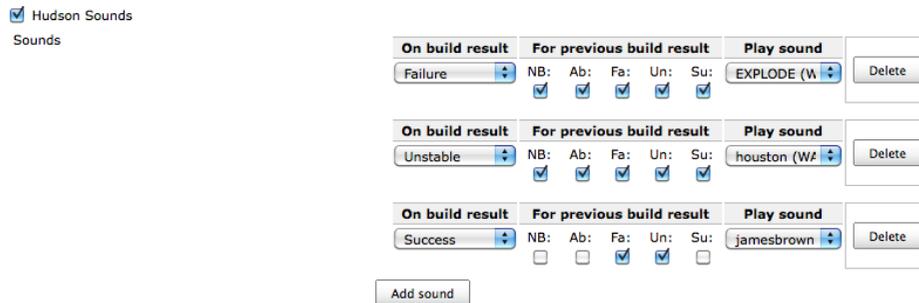


Figure 8.28. Configurer les règles de Jenkins Sounds dans une tâche de build

Pour le mettre en place, vous devez cocher Jenkins Sounds à la section des actions post-build dans la page de configuration de la tâche de build. Vous pouvez ajouter autant de règles de configuration audio que vous le souhaitez. L'ajout d'une règle est assez simple. Vous devez tout d'abord choisir quel résultat de build va déclencher le son. Vous devez également spécifier les résultats de build précédents pour lequel la présente règle est applicable: Non Construit (NB), Avorté (Ab), Échec (Fa), Non-positif (ONU) ou Réussi (Su).

Le plugin Jenkins Sounds propose une grande liste de sons prédéfinis qui offrent généralement beaucoup de choix pour les administrateurs de build les plus exigeants. Vous pouvez cependant ajouter votre propre son à la liste si vous le voulez. Les fichiers sonores des sons sont stockés dans un fichier ZIP ou JAR sous une structure plate (pas de sous-répertoire). La liste des sons proposés par le plugin est tout simplement la liste des noms de fichiers auxquels on retire l'extension. Le plugin supporte les formats AIFF, AU, et WAV.

Dans la page de configuration système, vous pouvez indiquer à Jenkins un nouveau fichier d'archive de sons en utilisant la notation `http://` si votre archive de sons est disponible sur un serveur Web local, ou la notation `file://` s'il est disponible en local (voir Figure 8.29, “Configurer Jenkins Sounds”). Une fois que vous avez sauvé la configuration, vous pouvez tester les sons de votre archive via le bouton Test Sound dans la section Avancé.



Figure 8.29. Configurer Jenkins Sounds

Le plugin Jenkins Sounds est un excellent choix si vous voulez compléter vos techniques de notification plus classiques. Les sons courts et reconnaissables sont un excellent moyen pour attirer l'attention d'un développeur et permettre à l'équipe de savoir que quelque chose doit être réparé. Ils seront alors un peu plus attentifs lorsque les notifications plus détaillées suivront.

Une autre option est le plugin Jenkins Speaks. Avec ce plugin, vous pouvez demander à Jenkins de diffuser une annonce personnalisée (avec une voix très robotique) lorsque votre build échoue (voir Figure 8.30, “Configurer Jenkins Speaks”). Vous pouvez configurer le message exact à l'aide de Jelly. Jelly est un langage de script basé sur XML largement utilisé dans les niveaux inférieurs de Jenkins.

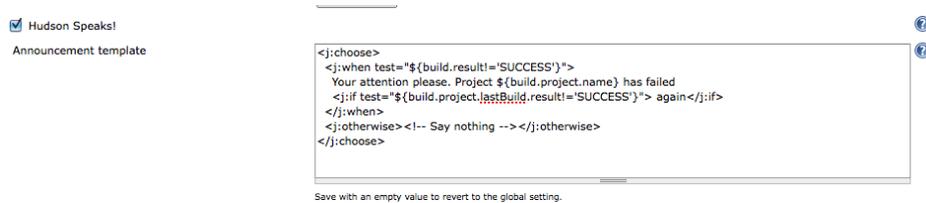


Figure 8.30. Configurer Jenkins Speaks

L'avantage de cette approche réside dans sa précision : puisque vous pouvez utiliser des variables Jenkins dans le script Jelly, vous pouvez demander à Jenkins de dire ce que vous voulez sur l'état de la construction. Voici un simple exemple :

```
<j:choose>
  <j:when test="${build.result!='SUCCESS'}">
    Votre attention, s'il vous plait. Le projet ${build.project.name} a échoué
  <j:if test="${build.project.lastBuild.result!='SUCCESS'}"> encore</j:if>
</j:when>
<j:otherwise><!-- Ne rien dire --></j:otherwise>
</j:choose>
```

Si vous laissez ce champ vide, le plugin va utiliser un modèle par défaut que vous pouvez configurer sur la page de configuration du système. En fait, c'est généralement une bonne idée de faire cela, et seulement d'utiliser script spécifique à un projet si c'est nécessaire.

L'inconvénient est que la voix robotique est parfois peut être un peu difficile à comprendre. Pour cette raison, c'est une bonne idée de commencer votre annonce par une phrase générique telle que "Votre attention s'il vous plaît», ou à combiner Jenkins Speak avec le plugin Jenkins Sounds, de façon à attirer l'attention des développeurs avant que le message réel ne soit diffusé. L'utilisation de traits d'union dans les noms de projets (par exemple, jeu-de-vie plutôt que jeudevie) aidera aussi le plugin à savoir comment prononcer vos noms de projet.

Ces deux approches sont utiles pour les petites équipes, mais peuvent être limitées pour les plus grosses, lorsque le serveur n'est pas physiquement situé à proximité de l'équipe de développement. Les futures versions pourront supporter la lecture de sons sur une machine séparée, mais au moment de la rédaction de ce livre ce n'est pas encore disponible.

8.14. Appareils de retour extrêmes

De nombreux outils de notification et stratégies plus imaginatives existent. Il y a de la place pour l'improvisation si vous êtes prêt à improviser un peu avec l'électronique. Cela inclut des appareils tels

que les orbes d'ambiance, les lampes de lave, les feux de circulation, ou d'autres appareils USB plus exotiques. Le radar de build (voir Section 8.6, “Radars de build”) tombe aussi dans cette catégorie si on le projette sur un écran assez grand.

Un appareil qui s'intègre très bien avec avec Jenkins est le Nabaztag. Le Nabaztag (voir Figure 8.31, “Un Nabaztag”) est un lapin robot WiFi très populaire qui peut faire clignoter des lumières colorées, jouer de la musique, ou même de parler. Un des avantages du Nabaztag, c'est qu'étant donné qu'il fonctionne en Wifi, il n'est pas contraint à être situé à proximité du serveur de build et fonctionnera même si votre instance Jenkins est dans une salle de serveurs ou sur une machine virtuelle. En ce qui concerne les appareils de retour extrêmes, ces petits compagnons sont difficiles à battre.



Figure 8.31. Un Nabaztag

Et encore mieux, il existe un plugin Jenkins pour le Nabaztag. Une fois que vous avez installé le plugin Nabaztag et redémarré Jenkins, il est facile à configurer. Sur la page de configuration principale de Jenkins, rendez-vous à la section Paramètres globaux de Nabaztag et entrez le numéro de série et jeton secret pour votre lapin électronique (voir Figure 8.32, “Configurer votre Nabaztag”). Vous pouvez également fournir des réglages par défaut sur la façon dont votre lapin de build devrait réagir aux changements dans l'état de build (doit-il signaler sur les départs ou succès de build, par exemple), quelle voix utiliser, et quel message dire quand un build échoue, réussit, est fixé, ou échoue à nouveau. Puis, pour activer les notifications Nabaztag pour une tâche de build particulière, vous devez cocher l'option de publication Nabaztag dans la configuration de votre tâche de build. Selon votre environnement par exemple, vous voulez ou non que tous vos builds envoient des notifications à votre Nabaztag.

Global Nabaztag Settings

Serial Number	0014F4B59435	?
Nabaztag API Token	*****	?
		Test credentials
Report On Build Start	<input type="checkbox"/> Report On Build Start	?
Report On Success	<input type="checkbox"/> Report On Success	?
Nabaztag API URL	http://api.nabaztag.com/v1/FR/api.jsp	?
Nabaztag Voice	UK-Penelope	?
Nabaztag Text for Starting Build	Build "\${buildNumber}" of project "\${projectName}" has started.	?
Nabaztag Text for Failure	Failure of build "\${buildNumber}" in project "\${projectName}".	?
Nabaztag Text for Success	Success of build "\${buildNumber}" in project "\${projectName}".	?
Nabaztag Text for Recover	Project "\${projectName}" recovered at build "\${buildNumber}".	?

Figure 8.32. Configurer votre Nabaztag

À l'exception notable du radar de build, nombre de ces dispositifs ont des limitations semblables aux plugins Jenkins Speaks et Jenkins Sounds (voir Section 8.13, "Faire du bruit") - ils sont mieux adaptés pour les petites équipes, co-localisées, qui travaillent sur un nombre limité de projets. Néanmoins, quand ils fonctionnent, ils peuvent être un complément utile à votre stratégie de notification générale.

8.15. Conclusion

La notification est une partie essentielle de votre stratégie globale d'intégration continue. Après tout, un build échoué a peu d'utilité s'il n'y a personne à l'écoute. Ce n'est pas non plus un problème universel. Vous devez penser à votre organisation et adapter votre stratégie pour répondre à la culture locale de l'entreprise et de l'ensemble des outils utilisés.

En effet, il est important de définir et de mettre en œuvre une stratégie de notification bien pensée qui convient à votre environnement. L'email, par exemple, est omniprésent et sera donc l'épine dorsale de nombreuses stratégies de notification. Si vous travaillez dans une grande équipe ou avec un responsable technique occupé, vous devrez peut-être envisager de mettre en place une stratégie progressive basée sur les options de messagerie avancées (voir Section 8.3, "Notification par email avancée"). Vous devrez compléter cette dernière avec une des stratégies les plus actives, telles que la messagerie instantanée ou la notification de bureau. Si votre équipe utilise déjà un canal de discussion IRC pour communiquer, essayez de l'intégrer dans votre stratégie de notification aussi. Enfin, la notification par SMS est une grande stratégie pour les tâches de build vraiment critiques.

Vous devez également veiller à ce que vous ayez à la fois des stratégies de notification actives et passives. Par exemple, un radar de build visible ou un appareil de retour extrême, envoient le message important à l'équipe que la correction des builds est une tâche prioritaire et peut aider à installer une culture plus agile à l'équipe.

Chapter 9. Qualité du Code

9.1. Introduction

Rares sont ceux qui nient l'importance de l'écriture d'un code de qualité. Un code de grande qualité contient moins de bugs, est plus facile à comprendre et plus facile à maintenir. Toutefois, les définitions précises de la qualité d'un code peuvent être plus subjectives, variant entre organisations, équipes, et même entre individus d'une même équipe.

C'est ici que les normes de codage entrent en jeu. Les normes de codage sont des règles, parfois relativement arbitraires, qui définissent les styles et les conventions de codage qui sont considérées comme acceptables au sein d'une équipe ou d'une organisation. Dans beaucoup de cas, se mettre d'accord sur un ensemble de normes et les appliquer est plus important que les normes elles-mêmes. En effet, un des aspects les plus importants d'un code de qualité est qu'il soit facile à lire et à comprendre. Si tous les développeurs d'une équipe appliquent les mêmes normes et les mêmes pratiques, le code sera ainsi plus lisible, au moins pour les membres de cette équipe. Et, si les normes sont communément utilisées dans l'industrie, le code sera d'autant plus lisible pour les nouveaux développeurs arrivant dans l'équipe.

Les normes de codage incluent à la fois des aspects esthétiques comme la disposition et la mise en forme du code, les conventions de nommage et ainsi de suite, ainsi que les potentielles mauvaises pratiques comme les oublis d'accolades après une condition en Java. Un style de codage cohérent réduit les coûts de maintenance, rend le code plus propre et plus lisible et permet de travailler plus facilement avec du code écrit par d'autres membres de l'équipe.

Seul un développeur expérimenté peut réellement juger de la qualité d'un code dans tous ses aspects. C'est le rôle des revues de code et, entre autres, des pratiques comme la programmation en binôme. En particulier, seul un œil humain peut décider si un bout de code est réellement bien écrit et s'il fait réellement ce que les exigences lui demandent de faire. Néanmoins, les outils de mesure de qualité de code peuvent être d'une grande aide. En effet, il n'est pas réaliste d'essayer de forcer des normes de codage sans ce type d'outil.

Ces outils analysent le code source ou le byte code de votre application et vérifient si le code respecte certaines règles. Les mesures de qualité de code peuvent englober beaucoup d'aspects de la qualité d'un code, des normes de codage et des meilleures pratiques jusqu'à la couverture du code, en passant par les avertissements du compilateur jusqu'aux commentaires TODO. Certaines mesures se concentrent sur les caractéristiques mesurables de votre base de code, comme le nombre de lignes de code (NLOC), la moyenne de complexité du code ou le nombre de lignes par classe. D'autres se focalisent sur des analyses statiques plus sophistiquées, ou sur la recherche de bugs potentiels ou de mauvaises pratiques dans votre code.

Il y a une large gamme de plugins établissant des rapports sur la qualité du code disponibles pour Jenkins. Beaucoup sont des outils d'analyse statique Java, comme Checkstyle, PMD, FindBugs, Cobertura et JDepends. D'autres, comme fxcop et NCover, se focalisent sur les applications .NET.

Pour tous ces outils, vous devez configurer votre tâche de build pour générer les données de mesure de qualité de code avant que Jenkins puisse produire le moindre rapport.

L'exception notable de cette règle est Sonar. Sonar peut extraire des mesures de qualité de code depuis n'importe quel projet Maven, sans aucune configuration supplémentaire dans votre projet. Ce qui est excellent lorsque vous avez un nombre important de projets Maven qui doivent être intégrés à Jenkins et que vous voulez configurer des rapports de qualité de code cohérents sur tous les projets.

Dans le reste de ce chapitre, vous verrez comment configurer des rapports de qualité de code dans vos builds Jenkins et également comment les utiliser comme un élément efficace dans votre processus de build.

9.2. La qualité du code dans votre processus de build

Avant de voir comment rapporter les mesures de qualités dans Jenkins, cela peut être utile de revenir en arrière pour avoir une vue plus large. Les mesures de la qualité du code sont d'une valeur limitée si elles sont isolées, elles doivent faire partie d'une stratégie plus large d'amélioration des processus.

Le premier niveau d'intégration de la qualité du code devrait être l'EDI. Les EDI modernes ont un excellent support de beaucoup d'outils de qualité de code — Checkstyle, PMD et FindBugs ont tous des plugins pour Eclipse, NetBeans et IntelliJ, qui fournissent un retour d'information rapide aux développeurs sur les problèmes de qualité du code. C'est le moyen le plus rapide et le plus efficace pour fournir un retour d'information à des développeurs et pour leur apprendre les normes de codage de l'organisation ou du projet.

Le second niveau est votre serveur de build. En plus de vos tâches régulières de tests unitaires et d'intégration, mettez en place un build de qualité de code dédié, qui démarrera après le build régulier et les tests. Le but de ce processus est de fournir des mesures de qualité de code à l'échelle du projet, de garder un œil sur la façon dont le projet est fait dans son ensemble et de traiter n'importe quels problèmes d'un niveau élevé. L'efficacité des ces rapports peut être augmentée par une revue hebdomadaire de la qualité du code, dans laquelle les problèmes et les tendances sur la qualité du code sont discutés au sein de l'équipe.

Il est important d'exécuter cette tâche séparément parce que les outils d'analyse de couverture de code et d'analyse statique peuvent être très longs à exécuter. Il est également important de garder éloigné des builds n'importe quels tests de couverture du code, puisque le processus de couverture du code produit du code instrumenté qui ne devrait jamais être déployé vers un dépôt pour une utilisation en production.

L'établissement de rapports sur la de qualité de code est, par défaut, un processus relativement passif. Personne ne connaîtra l'état du projet s'il ne cherche pas l'information sur le serveur de build. Même si

c'est mieux que rien, vous êtes peut-être exigeant sur la qualité du code, il y a alors un meilleur moyen. Au lieu de simplement établir des rapports sur la qualité du code, mettez en place un build dédié à la qualité du code, qui démarre après le build normal et les tests et configurez le build pour échouer si les mesures de la qualité du code ne sont pas à un niveau acceptable. Vous pouvez faire cela dans Jenkins ou dans votre script de build, bien qu'il soit avantageux de le configurer en dehors de votre script de build car vous pourrez changer les critères de défaillance du build de qualité du code sans changer le code source du projet.

Pour finir, souvenez-vous que les normes de codage sont des lignes directrices et des recommandations, pas des règles absolues. Utilisez la défaillance des builds et les rapports de qualité de code comme des indicateurs d'une zone possible d'amélioration, non pas comme des mesures de valeur absolue.

9.3. Les outils d'analyse de qualité du code populaires pour Java et Groovy

Il y a beaucoup d'outils open source qui peuvent aider à identifier les mauvaises pratiques de codage.

Dans le monde Java, trois outils d'analyses statiques ont résisté à l'épreuve du temps, et sont largement utilisés de manière très complémentaire. Checkstyle excelle dans la vérification des conventions et normes de codage, les pratiques de codage, ainsi que d'autres mesures telles que la complexité du code. PMD est un outil d'analyse statique similaire à Checkstyle, plus focalisé sur les pratiques de codage et de conception. Et FindBugs est un outil innovant, issu des travaux de recherche de Bill Pugh et de son équipe de l'université du Maryland, qui se focalise sur l'identification du code dangereux et bogué. Et si vous êtes en train de travailler avec Groovy ou Grails, vous pouvez utiliser CodeNarc, qui vérifie la norme et les pratiques de codage de Groovy.

Tous ces outils peuvent être facilement intégrés dans votre processus de build. Dans les sections suivantes, nous verrons comment configurer ces outils pour générer des rapports XML que Jenkins peut ensuite utiliser dans ses propres rapports.

9.3.1. Checkstyle

Checkstyle¹ est un outil d'analyse statique pour Java. A l'origine conçu pour faire respecter un ensemble de normes de codage hautement configurable, Checkstyle permet aussi maintenant de vérifier les mauvaises pratiques de codage, ainsi que le code trop complexe ou dupliqué. Checkstyle est un outil polyvalent et flexible qui devrait avoir sa place dans n'importe quelle stratégie d'analyse de code basé sur Java.

Checkstyle supporte un très grand nombre de règles, incluant celles liées aux normes de nommage, annotations, commentaires javadoc, taille de méthode et de classe, mesures de complexité de code, mauvaises pratiques de codage, et beaucoup d'autres.

¹ <http://checkstyle.sourceforge.net>

Le code dupliqué est un autre problème important de la qualité de code — le code dupliqué ou quasi-dupliqué est plus difficile à maintenir et à déboguer. Checkstyle fournit un certain soutien pour la détection de code dupliqué, mais des outils plus spécialisés comme CPD font un meilleur travail dans ce domaine.

Une des choses intéressante au sujet de Checkstyle est la facilité avec laquelle on peut le configurer. Vous pouvez commencer avec les normes de codage de Sun et les adapter selon vos besoins, ou démarrer de zéro. En utilisant le plugin Eclipse (ou même directement en XML), vous pouvez choisir parmi plusieurs centaines de règles, et affiner les options des règles que vous choisissez (voir Figure 9.1, “C’est facile de configurer les règles Checkstyle avec Eclipse”). C’est important, car les organisations, les équipes ou même les projets ont des attentes et des préférences différentes au regard des normes de codage, et c’est mieux d’avoir un ensemble de règles précises qui peuvent être adoptées, plutôt qu’un large éventail de règles qui seront ignorées. C’est encore plus important lorsque de grandes bases de code existant sont impliquées — dans ces cas, il est souvent préférable de commencer avec un ensemble plus limité de règles que d’être submergé par un grand nombre de problèmes de formatage relativement mineurs.

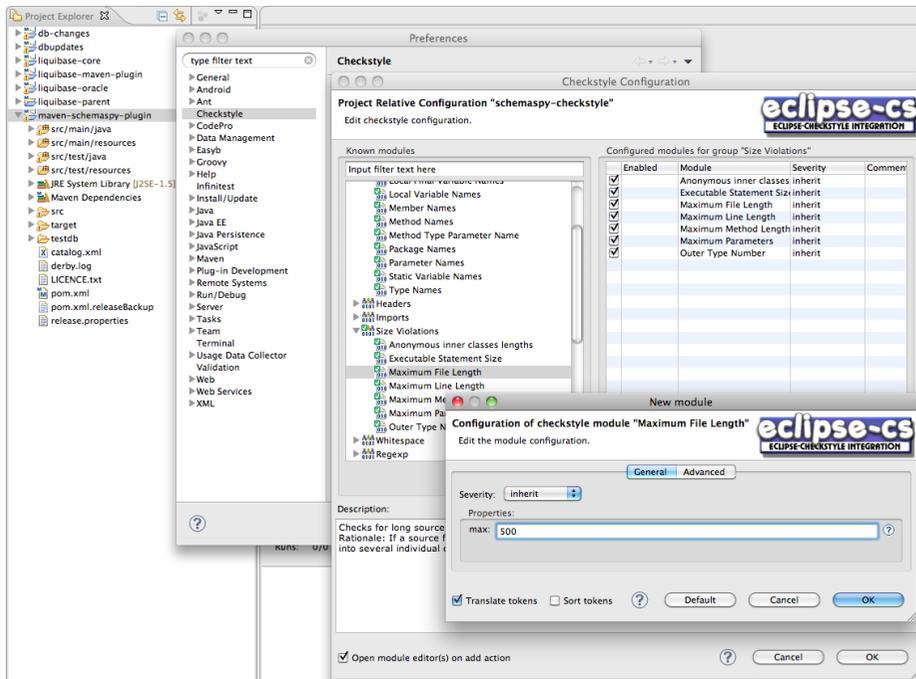


Figure 9.1. C’est facile de configurer les règles Checkstyle avec Eclipse

Configurer Checkstyle dans votre build est généralement simple. Si vous utilisez Ant, vous avez besoin de télécharger le fichier JAR de Checkstyle depuis le site web et de le rendre disponible à Ant. Vous pourriez le placer dans votre répertoire `lib` de Ant, mais cela signifierait de devoir personnaliser l’installation de Ant sur votre serveur de build (et tout les nœuds esclaves), ce n’est donc pas une

solution très portable. Une meilleure approche serait de placer le fichier JAR de Checkstyle dans l'un des répertoires de votre projet, ou d'utiliser Ivy ou la librairie Maven Ant Task pour déclarer une dépendance à Checkstyle. Si vous choisissez de garder le fichier JAR de Checkstyle dans les répertoires du projet, vous pourrez déclarer la tâche Checkstyle comme indiqué ici :

```
<taskdef resource="checkstyletask.properties"
  classpath="lib/checkstyle-5.3-all.jar"/>
```

Ensuite, pour générer les rapports Checkstyle dans le format XML exploitable par Jenkins, vous pourrez procéder comme suit :

```
<target name="checkstyle">
  <checkstyle config="src/main/config/company-checks.xml">
    <fileset dir="src/main/java" includes="**/*.java"/>
    <formatter type="plain"/>
    <formatter type="xml"/>
  </checkstyle>
</target>
```

Maintenant, invoquez juste cette tâche (cf., `ant checkstyle`) afin de générer les rapports Checkstyle.

Dans Maven 2, vous pouvez ajouter quelque chose comme la section `<reporting>` qui suit :

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>2.4</version>
      <configuration>
        <configLocation>
          src/main/config/company-checks.xml
        </configLocation>
      </configuration>
    </plugin>
  </plugins>
</reporting>
```

Pour un projet Maven 3, vous avez besoin d'ajouter un plugin sur l'élément `<reportPlugins>` de la section `<configuration>` du `maven-site-plugin` :

```
<project>
  <properties>
    <sonar.url>http://buildserver.acme.org:9000</sonar.url>
  </properties>
  <build>
    ...
    <plugins>
      ...
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
```

```

<artifactId>maven-site-plugin</artifactId>
<version>3.0-beta-2</version>
<configuration>
  <reportPlugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>2.4</version>
      <configuration>
        <configLocation>
          ${sonar.url}/rules_configuration/export/java/My_Rules/checkstyle.xml
        </configLocation>
      </configuration>
    </plugin>
  </reportPlugins>
</configuration>
</plugin>
</plugins>
</build>
</project>

```

Maintenant, l'exécution de `mvn checkstyle:checkstyle` ou `mvn site` analysera votre code source et générera des rapports XML que Jenkins peut utiliser.

A noter que dans le dernier exemple, nous utilisons un ensemble de règles de Checkstyle que nous avons transféré sur un serveur Sonar (définie par la propriété `${sonar.url}`). Cette stratégie rend facile l'utilisation du même ensemble des règles Checkstyle pour Eclipse, Maven, Jenkins, et Sonar.

Les versions récentes de Gradle offrent aussi une certaine prise en charge intégrée de Checkstyle. Vous pouvez le configurer pour vos builds comme il suit :

```
apply plugin: 'code-quality'
```

Cela utilisera par défaut l'ensemble de règles de Checkstyle définies dans `config/checkstyle/checkstyle.xml`. Vous pouvez redéfinir cela avec la propriété `checkstyleConfigFileName` : au moment de l'écriture de ce livre, néanmoins, il n'est pas possible de télécharger le plugin de qualité de code pour Gradle afin d'obtenir les règles Checkstyle depuis une URL.

Vous pouvez générer les rapports Checkstyle ici en exécutant `gradle checkstyleMain` or `gradle check`.

9.3.2. PMD/CPD

PMD² est un autre outil populaire d'analyse statique. Il se focalise sur les problèmes potentiels de codage comme le code non utilisé ou sous-optimisé, la taille et la complexité du code, et les bonnes pratiques de codage. Certaines règles typiques intègrent « Empty If Statement », « Broken Null Check », « Avoid Deeply Nested If Statements », « Switch Statements Should Have Default », et « Logger Is

² <http://pmd.sourceforge.net>

Not Static Final ». Il y a une bonne quantité de ressemblances avec certaines règles de Checkstyle, bien que PMD ait quelques règles plus techniques, et d'autres plus spécialisées tels que les règles relatives à JSF et Android.

PMD est aussi livré avec CPD, un détecteur open source robuste de code dupliqué ou quasi-dupliqué.

PMD est un peu moins flexible que Checkstyle, bien que vous puissiez choisir les règles que vous voulez utiliser dans Eclipse, et les exporter ensuite dans un fichier XML (voir Figure 9.2, "Configurer les règles PMD dans Eclipse"). Vous pouvez alors importer ces règles dans les autres projets Eclipse, dans Sonar, ou les utiliser dans vos builds Ant ou Maven.

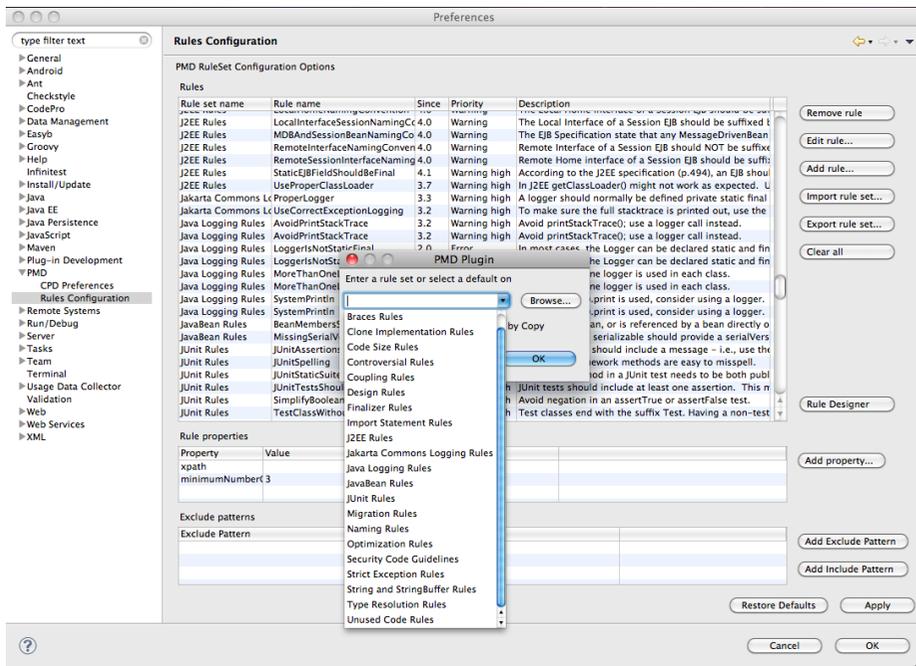


Figure 9.2. Configurer les règles PMD dans Eclipse

PMD est livré avec une tâche Ant que vous pouvez utiliser pour générer les rapports PMD et CPD. Tout d'abord, toutefois, vous devez définir ces tâches, comme le montre l'exemple suivant :

```
<path id="pmd.classpath">
  <pathelement location="org.apache.maven.model.Build@1460c81d"/>
  <fileset dir="lib/pmd">
    <include name="*.jar"/>
  </fileset>
</path>

<taskdef name="pmd" classname="net.sourceforge.pmd.ant.PMDTask"
```

```

classpathref="pmd.classpath"/>

<taskdef name="cpd" classname="net.sourceforge.pmd.cpd.CPDTask"
classpathref="pmd.classpath"/>

```

Ensuite, vous pouvez générer le rapport PMD XML en invoquant la tâche PMD comme illustré ici :

```

<target name="pmd">
  <taskdef name="pmd" classname="net.sourceforge.pmd.ant.PMDTask"
classpathref="pmd.classpath"/>

  <pmd rulesetfiles="basic" shortFileNames="true">
    <formatter type="xml" toFile="target/pmd.xml" />
    <fileset dir="src/main/java" includes="**/*.java"/>
  </pmd>
</target>

```

Et, pour générer le rapport CPD XML, vous pouvez faire quelque chose comme ça :

```

<target name="cpd">
  <cpd minimumTokenCount="100" format="xml" outputFile="/target/cpd.xml">
    <fileset dir="src/main/java" includes="**/*.java"/>
  </cpd>
</target>

```

Vous pouvez placer l'ensemble de ces règles XML dans le classpath de votre projet (par exemple, dans `src/main/resources` pour un projet Maven), ou dans un module séparé (si vous voulez partager la configuration entre les projets). Un exemple sur la façon de configurer Maven 2 pour générer des rapports PMD et CPD en utilisant un ensemble de règles XML exporté est indiqué ici :

```

<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-pmd-plugin</artifactId>
      <version>2.5</version>
      <configuration>
        <!-- PMD options -->
        <targetJdk>1.6</targetJdk>
        <aggregate>true</aggregate>
        <format>xml</format>
        <rulesets>
          <ruleset>/pmd-rules.xml</ruleset>
        </rulesets>

        <!-- CPD options -->
        <minimumTokens>20</minimumTokens>
        <ignoreIdentifiers>true</ignoreIdentifiers>
      </configuration>
    </plugin>
  </plugins>
</reporting>

```

Si vous utilisez Maven 3, vous devez placer la définition du plugin dans la section de configuration `<maven-site-plugin>`. Cet exemple montre aussi comment utiliser un ensemble de règles dans une autre dépendance (dans ce cas, le fichier `pmd-rules.jar`):

```
<project>
...
<build>
...
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-site-plugin</artifactId>
      <version>3.0-beta-2</version>
      <configuration>
        <reportPlugins>
          <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-pmd-plugin</artifactId>
            <version>2.5</version>
            <configuration>
              <!-- PMD options -->
              <targetJdk>1.6</targetJdk>
              <aggregate>true</aggregate>
              <format>xml</format>
              <rulesets>
                <ruleset>/pmd-rules.xml</ruleset>
              </rulesets>

              <!-- CPD options -->
              <minimumTokens>50</minimumTokens>
              <ignoreIdentifiers>true</ignoreIdentifiers>
            </configuration>
          </plugin>
        </reportPlugins>
      </configuration>
    <dependencies>
      <dependency>
        <groupId>com.wakaleo.code-quality</groupId>
        <artifactId>pmd-rules</artifactId>
        <version>1.0.1</version>
      </dependency>
    </dependencies>
  </plugin>
</plugins>
</build>
</project>
```

Maintenant, vous pouvez exécuter soit `mvn site` soit `mvn pmd:pmd pmd:cpd` pour générer les rapports PMD et CPD.

Malheureusement, il n'existe actuellement aucune prise en charge de Gradle pour PMD ou CPD, vous devez donc vous contenter d'appeler directement le plugin PMD pour Ant, comme montré ici :

```

configurations {
    pmdConf
}

dependencies {
    pmdConf 'pmd:pmd:4.2.5'
}

task pmd << {
    println 'Running PMD static code analysis'
    ant {
        taskdef(name:'pmd', classname:'net.sourceforge.pmd.ant.PMDTask',
            classpath: configurations.pmdConf.asPath)

        taskdef(name:'cpd', classname:'net.sourceforge.pmd.cpd.CPDTask',
            classpath: configurations.pmdConf.asPath)

        pmd(shortFileNames:'true', failonruleviolation:'false',
            rulesetfiles:'conf/pmd-rules.xml') {
            formatter(type:'xml', toFile:'build/pmd.xml')
            fileset(dir: "src/main/java") {
                include(name: '**/*.java')
            }
            fileset(dir: "src/test/java") {
                include(name: '**/*.java')
            }
        }

        cpd(minimumTokenCount:'50', format: 'xml',
            ignoreIdentifiers: 'true',
            outputFile:'build/cpd.xml') {
            fileset(dir: "src/main/java") {
                include(name: '**/*.java')
            }
            fileset(dir: "src/test/java") {
                include(name: '**/*.java')
            }
        }
    }
}

```

Cette configuration utilisera la règle PMD configuré dans le répertoire `src/config`, et générera un rapport PMD XML appelé `pmd.xml` dans le répertoire `build`. Il lancera aussi CPD et générera un rapport CPD XML appelé `cpd.xml` dans le répertoire `build`.

9.3.3. FindBugs

FindBugs est un puissant outil d'analyse de code qui vérifie le byte code de votre application afin de trouver des bogues potentiels, des problèmes de performances, ou des mauvaises habitudes de codage. FindBugs est le résultat d'une recherche menée par Bill Pugh à l'université du Maryland, et qui étudie les modèles de byte code venant de bogues dans de réels grands projets, comme les JDKs, Eclipse, ou le code source d'applications Google. FindBugs peut détecter des problèmes assez importants tels que

des exceptions de pointeurs nuls, des boucles infinies, et un accès non intentionnel de l'état interne d'un objet. Contrairement à beaucoup d'autres outils d'analyse statique, FindBugs tend à trouver un plus petit nombre de problèmes, mais de ces problèmes, une grande partie sera importante.

FindBugs est moins configurable que les autres outils que nous avons vu, mais en pratique vous n'avez généralement pas besoin d'affiner autant les règles qu'avec les autres outils dont nous avons discuté. Vous pouvez lister les règles individuelles que vous voulez appliquer, mais vous ne pouvez pas configurer un fichier XML partagé entre vos builds Maven et votre IDE, par exemple.

FindBugs est livrée empaqueté avec une tâche Ant. Vous pouvez définir la tâche FindBugs dans Ant comme montré en dessous. FindBugs a besoin de référencer le répertoire home de FindBugs, qui est où la distribution binaire a été décompressée. Pour rendre le build plus portable, nous stockons l'installation de FindBugs dans la structure de répertoire de notre projet, dans le répertoire `tools/findbugs` :

```
<property name="findbugs.home" value="tools/findbugs" />

<taskdef name="findbugs" classname="edu.umd.cs.findbugs.anttask.FindBugsTask" >
  <classpath>
    <fileset dir="${findbugs.home}/lib" includes="**/*.jar"/>
  </classpath>
</taskdef>
```

Ensuite, pour exécuter FindBugs, vous pourrez configurer une cible 'findbugs' comme montré dans l'exemple suivant. A noter que FindBugs s'exécute sur le byte code de votre application, et non sur le code source, donc vous devez compiler votre code source en premier :

```
<target name="findbugs" depends="compile">
  <findbugs home="${findbugs.home}" output="xml" outputFile="target/findbugs.xml">
    <class location="${classes.dir}" />
    <auxClasspath refId="dependency.classpath" />
    <sourcePath path="src/main/java" />
  </findbugs>
</target>
```

Si vous utilisez Maven 2, vous n'avez pas besoin de garder une copie locale de l'installation de FindBugs. Vous avez juste besoin de configurer FindBugs dans la section reporting comme montré ici :

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>findbugs-maven-plugin</artifactId>
      <version>2.3.1</version>
      <configuration>
        <effort>Max</effort>
        <xmlOutput>true</xmlOutput>
      </configuration>
    </plugin>
  </plugins>
</reporting>
```

Ou pour un projet Maven 3 :

```
<project>
...
<build>
...
  <plugins>
    ...
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-site-plugin</artifactId>
      <version>3.0-beta-2</version>
      <configuration>
        <reportPlugins>
          <plugin>
            <groupId>org.codehaus.mojo</groupId>
            <artifactId>findbugs-maven-plugin</artifactId>
            <version>2.3.1</version>
            <configuration>
              <effort>Max</effort>
              <xmlOutput>>true</xmlOutput>
            </configuration>
          </plugin>
        </reportPlugins>
      </configuration>
    </plugin>
  </plugins>
</build>
</project>
```

Dans les deux cas, vous pouvez générer vos rapports XML en lançant soit `mvn site` soit `mvn findbugs:findbugs`. Les rapports XML seront placés dans le répertoire `target`.

Au moment de la rédaction, il n'y a pas de prise en charge de FindBugs dans Gradle, donc vous devez invoquer le plugin Ant de FindBugs.

9.3.4. CodeNarc

CodeNarc est un outil d'analyse statique de code Groovy, similaire à PMD pour Java. Il vérifie le code source Groovy afin de trouver des défauts potentiels, des mauvais styles et pratiques de codage, du code trop complexe, et ainsi de suite. Des règles typiques incluent « Constant If Expression », « Empty Else Block », « GString As Map Key », et « Grails Stateless Service ».

Pour des projets basés sur Maven ou Ant, le plus simple est d'utiliser le plugin Ant de CodeNarc (un plugin Maven est en cours de développement au moment de l'écriture du livre). Une configuration typique de Ant pour l'utiliser avec Jenkins ressemblerait à ceci :

```
<taskdef name="codenarc" classname="org.codenarc.ant.CodeNarcTask"/>
<target name="runCodeNarc">
  <codenarc ruleSetFiles="rulesets/basic.xml,rulesets/imports.xml"
    maxPriority1Violations="0">
```

```
<report type="xml">
  <option name="outputFile" value="reports/CodeNarc.xml" />
</report>

<fileset dir="src">
  <include name="**/*.groovy"/>
</fileset>
</codenarc>
</target>
```

Vous pouvez intégrer CodeNarc dans un projet Grails simplement en installant le plugin CodeNarc :

```
$ grails install-plugin codenarc
```

Cela configurera CodeNarc pour analyser les fichiers Groovy dans le code de votre application Grails, aussi bien que dans les répertoires `src/groovy` et `test`.

Gradle 0.8 fournit aussi une prise en charge de CodeNarc dans le plugin de qualité de code, que vous pouvez le configurer dans vos builds comme montré ici :

```
apply plugin: 'code-quality'
```

Cela utilisera par défaut le fichier de configuration de CodeNarc suivant `config/codenarc/codenarc.xml`. Vous pouvez redéfinir cela avec la propriété `codeNarcConfigFileName`.

Vous pouvez générer les rapports CodeNarc en exécutant `gradle codenarcMain` ou, plus simplement, `gradle check`.

9.4. Rapports de problèmes de qualité de code avec le plugin Violations

Un des plugins de qualité de code les plus utiles est le plugin Violations. Ce plugin n'analysera pas le code source de votre projet (vous devez configurer le build pour faire cela), mais il fait un excellent travail en élaborant des rapports sur les mesures de la qualité du code pour les builds individuels et les tendances au fil du temps. Le plugin s'adresse aux rapports sur les mesures de qualité de code venant d'une large gamme d'outils d'analyse statique, comprenant :

Pour Java

Checkstyle, CPD, PMD, FindBugs, and jcreport

Pour Groovy

codenarc

Pour JavaScript

jslint

Pour .Net
gendarme and stylecop

Installer ce plugin est d'une simplicité. Il suffit d'aller sur l'écran du Plugin Manager et de sélectionner le plugin Violations de Jenkins. Une fois que vous installé le plugin et redémarré Jenkins, vous serez capable de l'utiliser pour vos projets.

Le plugin Violations ne génère pas de mesures de qualité de code lui-même — vous devez configurer votre build pour faire cela, comme il est montré dans la section précédente. Un exemple pour faire cela pour une tâche de build Maven est illustré dans la Figure 9.3, “Générer les rapports de qualité de code dans un build Maven”. Notez que nous invoquons ici les goals du plugin Maven directement. Nous aurions pu aussi simplement exécuter `mvn site`, mais si nous sommes uniquement intéressés par les mesures de qualité de code, et pas par les autres éléments du site généré par Maven, appeler le plugin directement se traduira par des builds plus rapides.

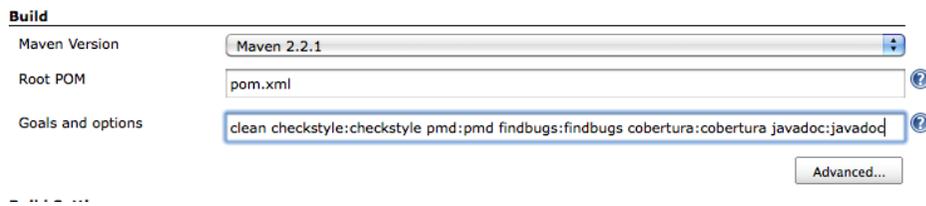


Figure 9.3. Générer les rapports de qualité de code dans un build Maven

Une fois que vous l’avez paramétré, vous pouvez configurer le plugin de violations pour générer des rapports, et si besoin est, des notifications de déclenchement, basés sur les résultats de rapport. Il suffit d’aller dans ‘Actions à la suite du build’ et de cocher la case Report Violations. Les détails de la configuration varient en fonction du type de projet. Penchons-nous sur les tâches de build Freestyle dans un premier temps.

9.4.1. Travailler avec des tâches de build free-style

Les tâches de build free-style vous permettent la configuration la plus flexible, et sont votre unique option pour des projets non Java.

Lorsque vous utilisez le plugin Violations avec une tâche de build free-style, vous devez spécifier les chemins de chaque rapport XML généré par les outils d’analyse de code statiques que vous avez utilisé (voir Figure 9.4, “Configurer le plugin violation pour un projet free-style”). Le plugin peut répondre à plusieurs rapports depuis le même outil, ce qui est utile pour des projets Maven multi-module — il suffit alors d’utiliser une expression générique pour identifier les rapports que vous voulez (par exemple, `**/target/checkstyle.xml`).

Report Violations

☀ ☁ 🟡 XML filename pattern

checkstyle	10	200	500	**/target/checkstyle-result.xml
codenarc	10	999	999	
cpd	0	10	15	**/target/cpd.xml
findbugs	0	50	50	**/target/findbugs.xml
fxcop	10	999	999	
gendarme	10	999	999	
jcreport	10	999	999	
jslint	10	999	999	
pmd	0	100	200	**/target/pmd.xml
pylint	10	999	999	
simian	10	999	999	
stylecop	10	999	999	
Per file limit	100			
Source Path Pattern				
Faux Project Path				
Source encoding	default			

Figure 9.4. Configurer le plugin violation pour un projet free-style

Le plugin Violations générera un graphique de suivi pour chaque type de problème au cours du temps (voir Figure 9.5, “Les violations au cours du temps”). Le graphique affiche une ligne de différente couleur pour chaque type de violations que vous suivez, ainsi qu’un résumé des derniers résultats.

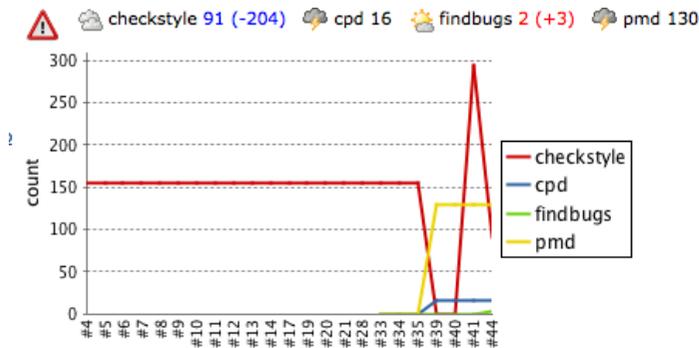


Figure 9.5. Les violations au cours du temps

Vous pouvez aussi cliquer sur le graphique pour vous rendre à un build particulier. Ici, vous pouvez voir le nombre de problèmes soulevés pour un build particulier (voir Figure 9.6, “Les violations pour un build particulier”), avec diverses ventilations par type de violation, sévérité et fichier.

Violations Report for build 45

Type	Violations	Files in violation
checkstyle	91	6
cpd	16	3
findbugs	2	2
pmd	130	6

checkstyle

filename	l	m	h	number
gameoflife-core/src/main/java/com/wakaleo/gameoflife/domain/Grid.java	0	30	0	30
gameoflife-web/src/main/java/com/wakaleo/gameoflife/web/controllers/GameController.java	0	27	0	27
gameoflife-core/src/main/java/com/wakaleo/gameoflife/domain/Universe.java	0	18	0	18
gameoflife-core/src/main/java/com/wakaleo/gameoflife/domain/Cell.java	0	8	0	8
gameoflife-core/src/main/java/com/wakaleo/gameoflife/domain/GridReader.java	0	5	0	5
gameoflife-core/src/main/java/com/wakaleo/gameoflife/domain/GridWriter.java	0	3	0	3

Figure 9.6. Les violations pour un build particulier

Enfin, vous pouvez descendre vers une classe particulière, pour afficher la liste détaillée des problèmes avec leurs emplacements dans le code source.

Mais le plugin Violations permet aussi une gestion plus proactive de la qualité du code. Vous pouvez utiliser les résultats des rapports d’analyse de la qualité du code pour influencer l’icône météorologique du tableau de bord de Jenkins. Cette icône météorologique est normalement liée au nombre de builds défaillants parmi les cinq derniers, mais Jenkins peut aussi prendre en compte d’autres facteurs, comme les résultats de la qualité du code. Afficher une icône pluvieuse ou orageuse pour un projet sur le tableau de bord est une meilleure façon de sensibiliser sur les problèmes de qualité du code que de simplement s’appuyer sur des graphiques et des rapports sur la page de la tâche de build.

Pour le configurer, vous devez revenir dans la section Report Violations dans Actions à la suite du build. Les trois premières colonnes dans Figure 9.4, “Configurer le plugin violation pour un projet free-style” montre une icône ensoleillée, une icône orageuse et une balle jaune. Celle avec l’icône de temps ensoleillé est le nombre maximum de violations tolérées pour garder l’icône ensoleillée sur la page du tableau de bord. La deuxième colonne, avec l’icône de temps orageux, est le nombre de violations qui causera l’affichage de l’icône orageuse sur le tableau de bord. Si vous avez un nombre de violations entre ces deux extrêmes, vous aurez l’une des icônes nuageuses.

Vous pouvez spécifier différentes valeurs pour différents outils. Les seuils exacts varieront entre les équipes et entre les projets, et aussi entre les outils. Par exemple, Checkstyle soulèvera généralement beaucoup plus de problèmes que FindBugs ou CPD, avec PMD quelque part entre. Vous devez ajuster les valeurs utilisées pour refléter comment ces outils travaillent avec votre code de base, et vos attentes.

Vous pouvez aller encore plus loin avec la troisième colonne (celle avec la balle jaune). Cette colonne vous permet de spécifier le nombre de violations qui déclarera le build comme instable. Souvenez-vous, lorsqu'un build devient instable, Jenkins enverra des messages de notifications, donc c'est une stratégie encore plus proactive.

Par exemple, dans Figure 9.4, "Configurer le plugin violation pour un projet free-style", nous avons configuré le nombre minimum des violations Checkstyle à 10, ce qui signifie que l'icône du temps ensoleillé apparaîtra uniquement s'il y a au maximum 10 violations Checkstyle. S'il y en a plus de 10, le temps se dégradera progressivement, jusqu'à 200 violations marquées, où il deviendra orageux. Et s'il y a 500 violations Checkstyle ou plus, le projet sera signalé instable.

Maintenant regardez la configuration de CPD, le détecteur de code dupliqué qui vient avec PMD. Dans ce projet, nous avons adopté une politique zéro tolérance pour le code dupliqué, donc l'icône ensoleillée est spécifiée à zéro. L'icône orageuse est spécifiée à 10, donc s'il y a 10 violations de copié/collé ou plus, il sera déclaré instable.

Maintenant, sur la page du tableau de bord, le projet apparaîtra avec à la fois un icône de temps orageux et comme instable, même s'il n'y a pas d'échecs de tests (voir Figure 9.7, "Configurer le plugin de violations pour un projet free-style"). Ce build particulier est instable parce qu'il y a 16 violations CPD. En complément, si vous placez votre souris sur l'icône du temps, Jenkins affichera quelques détails supplémentaires sur comment il a calculé ce statut particulier.

W	Description	%
	Number of checkstyle violations is 295	0
	Build stability: 2 out of the last 5 builds failed.	60
	Test Result: 0 tests failing out of a total of 71 tests.	100

Figure 9.7. Configurer le plugin de violations pour un projet free-style

9.4.2. Travailler avec des tâches de build Maven

Les tâches de build Maven dans Jenkins utilisent les conventions de Maven et les informations du fichier `pom.xml` du projet pour rendre la configuration plus facile et plus légère. Lorsque vous utilisez le plugin Violations avec une tâche de build Maven, Jenkins utilise ces conventions pour réduire la quantité de travail nécessaire pour configurer le plugin. Vous n'avez pas besoin de dire à Jenkins où trouver les rapports XML pour la plupart des outils d'analyse de code statique (par exemple, Checkstyle, PMD, FindBugs, et CPD), puisque Jenkins peut l'interpréter depuis les conventions de Maven et les configurations du plugin (voir Figure 9.8, "Configurer le plugin Violations pour un projet Maven."). Si vous devez redéfinir ces conventions, vous pouvez choisir l'option Pattern dans la liste déroulante « XML filename pattern », et entrer le chemin comme vous pouvez le faire pour les tâches de build free-style.

Build Settings

E-mail Notification ?
 Publish documents ?
 violations ?

☀ ☁ 🔴 XML filename pattern ?

checkstyle	10	999	999	auto
codenarc	10	999	999	auto
cpd	10	999	999	auto
findbugs	10	999	999	auto pattern
fxcop	10	999	999	
gendarme	10	999	999	
jcreport	10	999	999	auto
jslint	10	999	999	auto
pm�	10	999	999	auto
pylint	10	999	999	
simian	10	999	999	
stylecop	10	999	999	auto
Per file limit	100			?
Source encoding	default			?

Figure 9.8. Configurer le plugin Violations pour un projet Maven.

Le plugin Violations fonctionne bien avec des projets multi-module Maven, mais au moment de la rédaction (de ce livre), il a besoin d'un peu de peaufinage pour obtenir de meilleurs résultats. Les tâches de build Maven comprennent la structure des projets multi-module (voir Figure 9.9, "Les tâches de build Maven de Jenkins comprennent les structures multi-modules de Maven") ; de plus, vous pouvez descendre dans n'importe quel module et obtenir une vue détaillée des résultats de construction pour cette tâche de build.

S	W	Job ↓	Last Success	Last Failure	Last Duration
🌐	☀	gameoflife	1 day 4 hr (#5)	N/A	11 sec
🌐	☀	gameoflife-build	1 day 4 hr (#5)	N/A	1.6 sec
🌐	☁	gameoflife-core	1 day 4 hr (#5)	N/A	33 sec
🌐	☁	gameoflife-web	1 day 4 hr (#5)	N/A	19 sec
🌐	☀	gameoflife-webservice	1 day 4 hr (#5)	N/A	1.5 sec
🌐	☀	gameoflife-cli	1 day 4 hr (#5)	N/A	1.7 sec

Icon: S M L

Figure 9.9. Les tâches de build Maven de Jenkins comprennent les structures multi-modules de Maven

C'est une fonctionnalité très utile, mais cela signifie que vous devez faire un peu de travail supplémentaire pour obtenir tous les avantages des plugins de violations des modules individuels. Par défaut, le plugin Violations affichera une vue agrégée des mesures de qualité de code comme celle dans Figure 9.5, "Les violations au cours du temps". Vous pouvez aussi cliquer sur le graphique des violations, et voir les rapports détaillés de chaque module.

Cependant, pour que ceci puisse fonctionner correctement, vous devez activer le plugin Violations individuellement pour chaque module en complément du projet principal. Pour ce faire, cliquez sur le module que vous voulez configurer dans l'écran Modules, et ensuite cliquez sur le menu 'Configurer'. Ici, vous verrez un petit groupe des options de configuration habituelles (voir Figure 9.10, "Activer le plugin Violations pour un module individuel"). Ici, il vous suffit d'activer l'option Violations, et de configurer les seuils si besoin est. Le côté positif de cela est que vous pouvez définir différentes valeurs de seuils pour des modules différents.

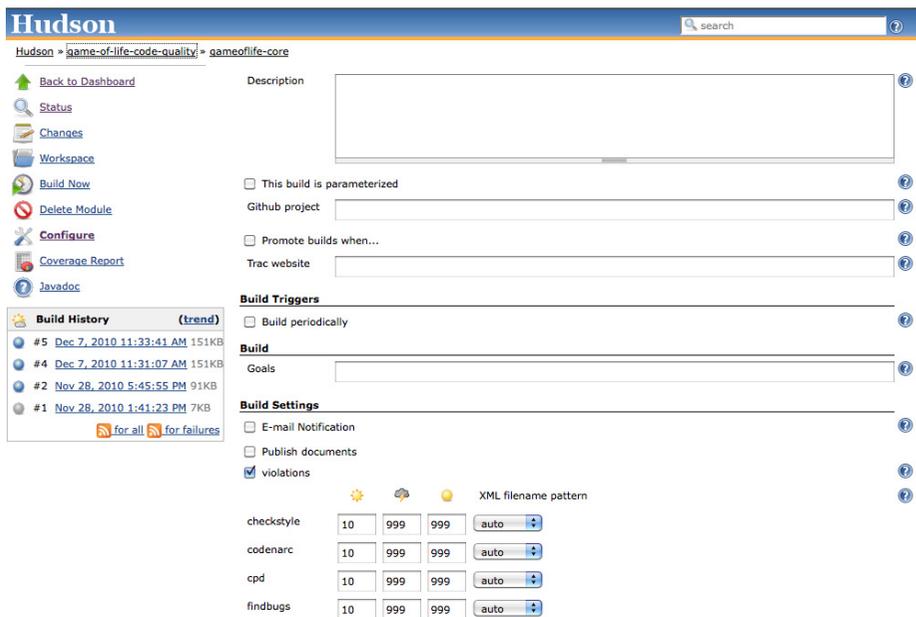


Figure 9.10. Activer le plugin Violations pour un module individuel

Une fois que cela est fait, lorsque vous cliquerez sur le graphique de violations agrégées sur la page d'accueil de la tâche de build du projet, Jenkins listera les graphiques individuels de violations pour chaque module.

9.5. Utiliser les rapports Checkstyle, PMD, et FindBugs

Vous pouvez aussi effectuer des rapports individuellement sur les résultats de Checkstyle, PMD, et FindBugs. En complément du plugin Violations, il y a des plugins Jenkins qui produisent des graphiques de tendance et des rapports détaillés pour chacun de ces outils de façon individuelle. Nous allons voir

comment le faire avec Checkstyle, mais cette approche s'applique aussi pour PMD et FindBugs. Vous pouvez aussi utiliser le plugin Analysis Collector pour afficher les résultats combinés dans un graphique similaire à celui produit par le plugin Violations.

Vous pouvez installer ces plugins au travers du gestionnaire de plugin comme vous le faites habituellement. Les plugins en question sont appelés, sans surprise, plugin Checkstyle, plugin PMD, et plugin FindBugs. Tous ces plugins utilisent le plugin Static Analysis Utilities, qu'il vous faut aussi installer (voir Figure 9.11, "Installer les plugins Checkstyle et Static Analysis Utilities.").

Build Reports		
<input checked="" type="checkbox"/>	Analysis Collector Plugin This plug-in is an add-on for the plug-ins Checkstyle , Drv , FindBugs , PMD , Task Scanner , and Warnings : the plug-in collects the different analysis results and shows the results in a combined trend graph. Additionally, the plug-in provides health reporting and build stability based on these combined results.	1.8
<input checked="" type="checkbox"/>	Static Code Analysis Plug-ins This plug-in provides utilities for the static code analysis plug-ins.	1.14
<input type="checkbox"/>	CCCC Plugin This plugin generates the trend report for CCCC (C and C++ Code Counter).	0.3
<input type="checkbox"/>	CCM Plugin This plug-in generates reports on cyclomatic complexity for .NET code.	1.1
<input checked="" type="checkbox"/>	Checkstyle Plugin This plugin generates the trend report for Checkstyle , an open source static code analysis program.	3.10

Figure 9.11. Installer les plugins Checkstyle et Static Analysis Utilities.

Une fois que vous avez installé ces plugins, vous pouvez paramétrer la création des rapports dans la configuration de votre projet. Cochez la case "Publiez les analyses de résultat Checkstyle". Dans un build free-style, vous aurez besoin de spécifier un patron de chemin pour trouver les rapports XML de Checkstyle ; dans un build Maven 2, Jenkins saurait où les chercher par lui-même.

Ceci fournira des rapports Checkstyle basiques, mais comme d'habitude, vous pouvez personnaliser ceci en cliquant sur le bouton Avancé. Dans un build Maven 2, vous pouvez configurer les valeurs de seuils de la santé (combien de violations passeront le build d'ensoleillé à orageux), et aussi filtrer les priorités de violations que vous voulez inclure dans le calcul. Par exemple, vous pourriez ne vouloir prendre en compte que les problèmes de haute priorité pour le statut de l'icône météorologique.

Les builds free-style ont quelques options supplémentaires que vous pouvez configurer ; en particulier, vous pouvez rendre le build instable (balle jaune) ou même en échec (balle rouge) si vous avez un nombre de violations supérieur à celui défini, ou s'il y a plus de nouvelles violations que le nombre donné (voir Figure 9.12, "Configurer le plugin Checkstyle"). Ainsi, dans la configuration de l'illustration, s'il y a plus de 50 nouvelles violations Checkstyle de n'importe quelle priorité dans un build, le build sera signalé comme instable. Cela a certainement son utilité pour Checkstyle, mais cela peut aussi se révéler très utile avec FindBugs, où les problèmes de haute priorité représentent souvent des bugs dangereux et potentiellement bloquants.

Post-build Actions

Publish Checkstyle analysis results ?

Checkstyle results

Fileset includes setting that specifies the generated raw CheckStyle XML report files, such as `**/checkstyle-result.xml`. Basedir of the fileset is `the workspace root`. If no value is set, then the default `**/checkstyle-result.xml` is used. Be sure not to include any non-report files into this pattern.

Run always

By default, this plug-in runs only for stable or unstable builds, but not for failed builds. If this plug-in should run even for failed builds then activate this check box.

Health thresholds

100% 0%

Configure the thresholds for the build health. If the actual number of warnings is between the provided thresholds, then the build health is interpolated.

Health priorities

Only priority high Priorities high and normal All priorities

Determines which warning priorities should be considered when evaluating the build health.

Status thresholds

	All priorities	Priority high	Priority normal	Priority low
<input type="radio"/> Total	<input type="text" value="200"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="radio"/> New	<input type="text" value="50"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="radio"/> Total	<input type="text" value="500"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>
<input type="radio"/> New	<input type="text" value="100"/>	<input type="text"/>	<input type="text"/>	<input type="text"/>

If the specified number of warnings exceeds one of these thresholds then a build is considered as unstable or failed, respectively.

Use delta for new warnings

If set then the number of new warnings is calculated by subtracting the total number of warnings of the current build from the reference build. This may lead to wrong results if you have both fixed and new warnings in a build. If the checkbox is not set, then the number of new warnings is calculated by an asymmetric set difference of the warnings in the current and reference build. This will find all new warnings even if the number of total warnings is decreasing. However, sometimes false positives will be reported due to minor changes in a warning (refactoring of variable or method names, etc.)

Default Encoding

Default encoding when parsing or showing files. Leave this field empty to use the default encoding of the platform.

Trend graph [You can define the default values for the trend graph in a separate view.](#)

Figure 9.12. Configurer le plugin Checkstyle

Maintenant, lorsque le build s'exécute, Jenkins générera un graphique de tendance et des rapports détaillés pour les violations Checkstyle (voir Figure 9.13, "Afficher les tendances Checkstyle"). De là, vous pouvez descendre pour voir les violations par priorité, par catégorie, par type de démarrage, par package, etc.

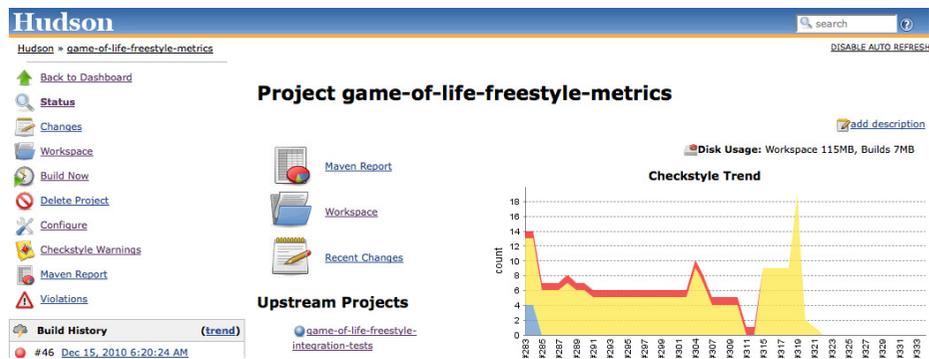


Figure 9.13. Afficher les tendances Checkstyle

Comme nous l'avons mentionné précédemment, la même approche marche aussi avec le plugin PMD et le plugin FindBugs. Ces plugins sont un excellent moyen de fournir des rapports plus concentrés sur les résultats d'un outil particulier, et vous donnent aussi plus de contrôle sur l'impact que ces violations auront sur les résultats du build.

9.6. Les rapports sur la complexité du code

La complexité du code est un autre aspect important de la qualité du code. La complexité du code est mesurée avec un certain nombre de moyens, mais une mesure de complexité généralement utilisée (et facile à comprendre) est la complexité cyclomatique, qui consiste à mesurer le nombre de chemins différents à travers une méthode. En utilisant cette métrique, le code complexe a généralement un grand nombre d'instructions conditionnelles et de boucles imbriquées, qui rendent le code plus difficile à comprendre et à déboguer.

Il y a aussi une théorie de qualité de code mettant en corrélation la complexité du code et la couverture du code qui donne une idée générale de la fiabilité d'une portion de code. Celle-ci est basée sur l'idée (très compréhensible) que le code qui est à la fois complexe et peu testé est plus susceptible de contenir des bugs que du code simple et bien testé.

Le plugin Coverage Complexity Scatter Plot est conçu pour vous laisser visualiser cette information dans vos builds Jenkins (voir Figure 9.14, "Un nuage de point couverture/complexité."). Les méthodes dangereusement complexes et/ou non testées apparaîtront hautes sur le graphique, alors que les méthodes bien écrites et bien testées apparaîtront plus basses.

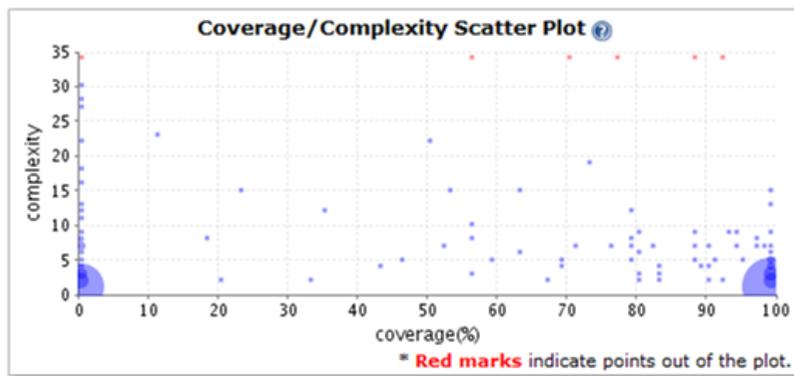


Figure 9.14. Un nuage de point couverture/complexité.

Le graphique de dispersion vous donne un bon aperçu de l'état de votre code en termes de complexité et de couverture de test, mais vous pouvez aussi descendre pour poursuivre l'enquête. Si vous cliquez sur n'importe quel point dans le graphique, vous pouvez voir les méthodes correspondantes, avec leur couverture de test et leur complexité (voir Figure 9.15, "Vous pouvez cliquer sur n'importe quel point du graphique pour poursuivre l'enquête").

Coverage / Complexity Scatter Plot

2 method(s) in the range of coverage (90%~100%) and complexity (5~9)

Method	Complexity	Coverage(%)	Total	Covered
createNextGeneration() : void	7	100	19	19
cellIsOutsideBorders(int,int) : boolean	5	100	3	3

Figure 9.15. Vous pouvez cliquer sur n'importe quel point du graphique pour poursuivre l'enquête

Au moment de la rédaction de ce livre, le plugin nécessite Clover, donc votre build a besoin d'avoir généré un rapport XML de couverture de Clover, et vous avez besoin d'avoir installé et configuré le plugin Clover de Jenkins (voir Section 6.6.2, "Mesurer la couverture de code avec Clover"). Cependant, un support pour Cobertura et d'autres outils est prévu.

9.7. Les rapports sur les tâches ouvertes

Quand il s'agit de qualité de code, l'analyse statique n'est pas le seul outil que vous pouvez utiliser. Un autre indicateur de la santé générale de votre projet peut être trouvé avec le nombre de `FIXME`, `TODO`, `@deprecated`, et autres balises similaires dispersées dans le code source. S'il y en a beaucoup, cela peut être un signe que le code de base a beaucoup de travail inachevé, et n'est donc pas dans un état très finalisé.

Le plugin Task Scanners de Jenkins permet de garder une trace de ces sortes de balises dans votre code source, et optionnellement de signaler un build avec une mauvaise icône de temps sur le tableau de bord s'il y a beaucoup trop de tâches ouvertes.

Pour le configurer, vous devez installer à la fois le plugin Static Analysis Utilities et le plugin Task Scanner. Une fois qu'ils sont installés, vous pouvez activer le plugin dans votre projet en cochant la case « Recherche des tâches ouvertes dans le workspace » dans la section Configuration du Build dans la configuration de la tâche de votre projet.

Configurer le plugin Task Scanner est assez simple (voir Figure 9.16, "Configurer le plugin Task Scanner est simple"). Vous entrez simplement les balises que vous souhaitez suivre, avec différentes priorités si vous considérez certaines balises comme étant plus importantes que d'autres. Par défaut, le plugin scrutera tout le code source java dans le projet, mais vous pouvez redéfinir ce comportement en complétant le champ Files to scan. Dans Figure 9.16, "Configurer le plugin Task Scanner est simple", par exemple, nous vérifions les balises pour les fichiers XML et JSP.

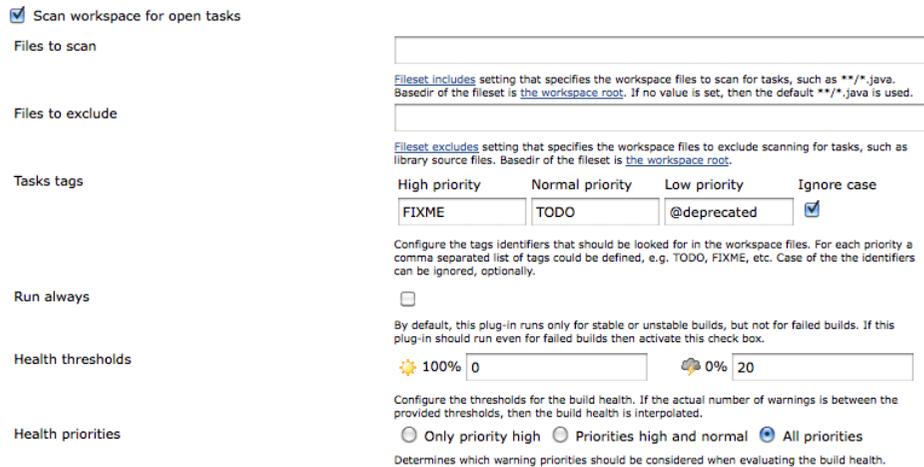


Figure 9.16. Configurer le plugin Task Scanner est simple

Le bouton Avancé vous donne accès à des options plus sophistiqués. Probablement, les plus utiles sont les seuils de santé, qui vous laissent définir le nombre maximum de problèmes tolérés avant que le build ne soit plus considéré « ensoleillé », et le nombre minimum de problèmes requis pour le statut de « temps orageux ».

Le plugin génère un graphique qui montre les tendances de balises par ordre de priorité (voir Figure 9.17, “Le graphique de tendances des tâches ouvertes”). Si vous cliquez sur le rapport des tâches ouvertes, vous pouvez aussi voir le détail des tâches par module Maven, package ou fichier, ou encore une liste des tâches ouvertes.

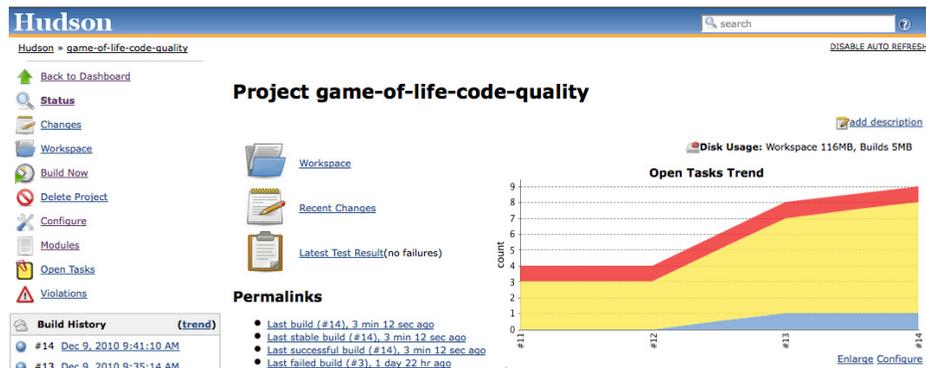


Figure 9.17. Le graphique de tendances des tâches ouvertes

9.8. Intégration avec Sonar

Sonar³ est un outil qui centralise toute une gamme de mesures de qualité de code en un seul site web (voir Figure 9.18, “Rapport de qualité de code par Sonar.”). Il utilise un certain nombre de plugins Maven (Checkstyle, PMD, FindBugs, Cobertura ou Clover, et d’autres) pour analyser des projets Maven et générer un ensemble complet de rapports sur la qualité du code. Les rapports Sonar sur la couverture du code, le respect des règles, et la documentation, mais aussi sur des mesures de plus haut niveau comme la complexité, la maintenabilité et même la dette technique. Vous pouvez utiliser des plugins pour étendre ses fonctionnalités et ajouter le support pour d’autres langages (comme le support de CodeNarc pour du code source Groovy). Les règles utilisées par divers outils sont gérées et configurées de manière centralisée sur le site web de Sonar, et les projets Maven en cours d’analyse ne nécessitent aucune configuration particulière. Cela fait de Sonar l’outil parfait pour travailler sur des projets Maven où vous avez un contrôle limité sur les fichiers POM.

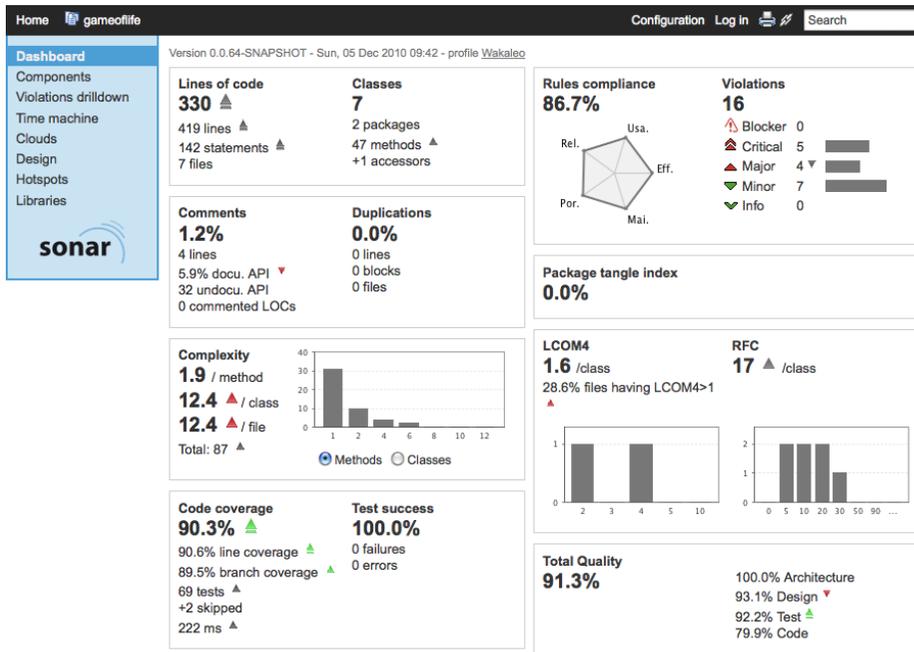


Figure 9.18. Rapport de qualité de code par Sonar.

Dans l’une des utilisations les plus courantes de Sonar, Sonar exécute automatiquement un ensemble de plugins Maven liés à la qualité de code sur votre projet Maven, et stocke les résultats dans une base de données relationnelle. Le serveur Sonar, que vous exécutez séparément, analyse alors les résultats et les affiche comme indiqué dans Figure 9.18, “Rapport de qualité de code par Sonar.”.

³ <http://www.sonarsource.org>

Jenkins s'intègre bien avec Sonar. Le plugin Sonar de Jenkins vous laisse définir les instances Sonar pour tous vos projets, et active ensuite Sonar pour des builds particuliers. Vous pouvez exécuter votre serveur Sonar sur une machine différente de votre instance Jenkins, ou sur la même. La seule contrainte est que l'instance Jenkins doit avoir un accès JDBC à la base de données de Sonar, puisqu'il injecte des mesures de qualité de code directement dans la base de données, sans passer par le site web de Sonar (voir Figure 9.19, "Jenkins et Sonar").

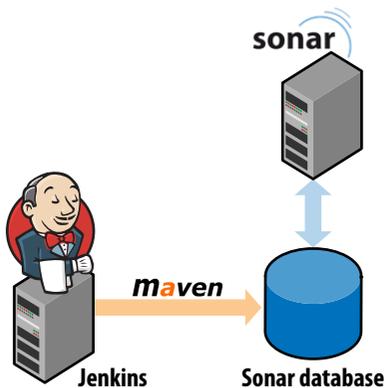


Figure 9.19. Jenkins et Sonar

Sonar a aussi une amorce Ant (avec une amorce Gradle en cours de développement au moment de la rédaction de ce livre) pour les utilisateurs non-Maven.

Vous installez le plugin comme d'habitude, via le gestionnaire de plugin. Une fois installé, vous configurez le plugin Sonar de Jenkins dans l'écran Configurer le système, dans la section Sonar. Il s'agit de définir vos instances Sonar – vous pouvez configurer autant d'instances que vous avez besoin. La configuration par défaut suppose que vous exécutez une instance locale de Sonar avec la base de données embarquée par défaut. Ceci est utile à des fins de test, mais n'est pas très évolutif. Pour un environnement de production, vous exécuterez alors Sonar sur une vraie base de données comme MySQL ou Postgres, et vous aurez besoin de configurer la connexion JDBC sur la base de données de production de Sonar dans Jenkins. Vous pouvez faire ceci en cliquant le bouton Avancé et en remplissant les champs appropriés (voir Figure 9.20, "Configurer Sonar dans Jenkins").

Sonar

Sonar installations	
Name	<input type="text" value="sonar-enterprise"/>
Disable	<input type="checkbox"/>
	<small>Check to quickly disable Sonar on all jobs.</small>
Server URL	<input type="text" value="http://www.acme.com/sonar"/>
	<small>Default is http://localhost:9000</small>
Server Public URL	<input type="text"/>
	<small>If not specified, then Server URL will be used</small>
Database URL	<input type="text" value="jdbc:mysql://localhost:3306/sonar?useUnicode=true&characterEi"/> ?
	<small>Do not set if default embedded database.</small>
Database login	<input type="text" value="sonar"/>
	<small>Default is sonar.</small>
Database password	<input type="text" value="secret"/>
	<small>Default is sonar.</small>
Database driver	<input type="text" value="com.mysql.jdbc.Driver"/> ?
	<small>Do not set if you use the default embedded database on localhost.</small>
Additional properties	<input type="text"/>
	<small>Additional properties to be passed to the mvn executable (example : -Dsome.property=some.value)</small>
Triggers	
	<input type="checkbox"/> Poll SCM ?
	<input checked="" type="checkbox"/> Build periodically ?
	<input checked="" type="checkbox"/> Manually started by user ?
	<input type="checkbox"/> Build whenever a SNAPSHOT dependency is built ?
	<input type="checkbox"/> Skip analysis on build failure ?
	<input type="button" value="Delete Sonar"/>
	<input type="button" value="Add Sonar"/>
	<small>List of Sonar installations</small>

Figure 9.20. Configurer Sonar dans Jenkins

L'autre chose que vous devez configurer est le moment où le build de Sonar débutera dans une tâche de build où Sonar est activé. Vous configurez généralement Sonar pour qu'il s'exécute avec l'une des longues tâches de build Jenkins, comme le build de mesure de qualité de code. Ce n'est pas très utile d'exécuter le build Sonar plus d'une fois par jour, puisque Sonar stocke les mesures sur des tranches de 24 heures. La configuration par défaut exécutera le build de Sonar, dans une tâche de build où Sonar est activé, chaque fois qu'une tâche est déclenchée par un build périodiquement planifié ou par un build manuel.

Pour activer Sonar dans votre tâche de build, avec des options de configuration à l'échelle du système, il suffit de cocher l'option Sonar dans Actions à la suite du build (voir Figure 9.21, "Configurer Sonar dans une tâche de build"). Sonar s'exécutera à chaque fois que votre build est lancé par l'un des mécanismes de déclenchements définis ci-dessus.

Sonar ?

Branch ?

MAVEN_OPTS ?

Additional properties ?

Optional sonar.branch property.

MAVEN_OPTS env var to provide, if not set the plugin will use the MAVEN_OPTS defined by the maven builder config.

Additional properties to be passed to the mvn executable (example: -Dsome.property=some.value).

Dont use global triggers configuration ?

Triggers

Poll SCM ?

Build periodically ?

Manually started by user ?

Build whenever a SNAPSHOT dependency is built ?

Skip analysis on build failure ?

Figure 9.21. Configurer Sonar dans une tâche de build

Vous configurez généralement Sonar pour qu'il s'exécute sur une base régulière, par exemple tous les soirs ou une fois par semaine. Vous pouvez donc activer Sonar sur votre tâche de build de test unitaire/d'intégration normal, simplement en ajoutant un ordonnanceur (voir Figure 9.22, "Planifier les builds Sonar"). Cela évite les détails de configuration en double entre les tâches. Ou, si vous avez déjà une tâche de build ordonnancée qui démarre avec une fréquence appropriée (comme un build dédié aux mesures de qualité de code), vous pouvez activer Sonar sur cette tâche de build.

Build Triggers

Build whenever a SNAPSHOT dependency is built ?

Build after other projects are built ?

Build periodically ?

Schedule ?

Poll SCM ?

Schedule ?

Figure 9.22. Planifier les builds Sonar

Si vous cliquez sur le bouton Avancé, vous pouvez spécifier d'autres options plus sophistiquées, comme lancer votre build Sonar sur une branche séparée, passer des options de ligne de commande Maven supplémentaires (comme de la mémoire supplémentaire), ou redéfinir la configuration par défaut des déclencheurs.

Par défaut, Sonar s'exécutera même si le build normal échoue. C'est généralement ce que l'on souhaite, puisque Sonar devrait enregistrer les builds et les tests défaillants aussi bien que les résultats réussis. Cependant, si c'est nécessaire, vous pouvez aussi désactiver cette option dans les options avancées.

9.9. Conclusion

La qualité de code est une partie importante du processus de build, et Jenkins fournit un excellent support pour la vaste gamme d'outils liés à la qualité de code qui existent. En conséquence, Jenkins devrait être un élément clé de votre stratégie sur la qualité du code.

Chapter 10. Builds avancés

10.1. Introduction

Dans ce chapitre, nous regarderons quelques configurations avancées de tâches de build. Nous traiterons des builds paramétrés, qui permettent à Jenkins de demander à l'utilisateur des paramètres supplémentaires qui seront passés à la tâche de build, et des tâches de build multiconfigurations, qui vous permettent d'exécuter une simple tâche selon de nombreuses variations. Nous verrons comment exécuter les tâches en parallèle, et comment attendre le résultat d'une ou plusieurs tâches avant de continuer. Nous verrons enfin comment implémenter des stratégies de promotion de build et de pipelines de build afin de pouvoir utiliser Jenkins non seulement comme un serveur de build, mais aussi comme un serveur de déploiement.

10.2. Tâches de build paramétrées

Les builds paramétrés sont un concept puissant vous permettant d'ajouter une nouvelle dimension à vos tâches de build.

Le plugin `Parameterized Build` vous permet de configurer des paramètres pour vos tâches de build, qui peuvent être entrés par l'utilisateur lorsque le build est déclenché, ou (comme nous le verrons plus tard) depuis une autre tâche.

Par exemple, vous pourriez avoir une tâche de déploiement, où vous choisiriez un environnement cible dans une liste déroulante quand vous démarrez le build. Vous pourriez aussi vouloir spécifier la version de l'application que vous souhaitez déployer. Ou, en exécutant une tâche de build incluant des tests web, vous pourriez spécifier le navigateur dans lequel vous voulez faire tourner vos tests Selenium ou WebDriver. Vous pouvez même télétransférer un fichier nécessaire à l'exécution de la tâche de build.

Notez que c'est le rôle du script de build d'analyser et de traiter correctement les valeurs de paramètres — Jenkins fournit simplement une interface utilisateur permettant d'entrer les valeurs des paramètres, puis de passer ces paramètres au script de build.

10.2.1. Créer des tâches de build paramétrées

Vous installez le plugin `Parameterized Build` comme d'habitude, via l'écran de gestion des plugins. Une fois que vous avez fait cela, configurer une tâche de build paramétrée est simple. Cochez simplement l'option “Ce build a des paramètres” et cliquez sur `Ajouter un paramètre` pour ajouter un nouveau paramètre de tâche de build (voir Figure 10.1, “Créer une tâche de build paramétrée”). Vous pouvez ajouter des paramètres à n'importe quelle sorte de build, et vous pouvez ajouter autant de paramètres que vous voulez à une tâche de build donnée.

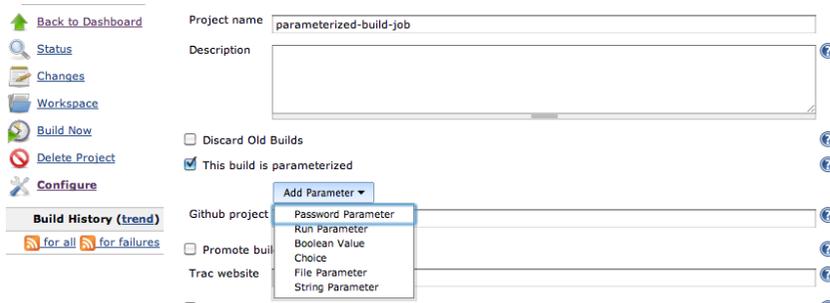


Figure 10.1. Créer une tâche de build paramétrée

Pour ajouter un paramètre à votre tâche de build, sélectionnez simplement le type de paramètre dans la liste déroulante. Cela vous permettra de configurer les détails de votre paramètre (voir Figure 10.2, “Ajouter un paramètre à la tâche de build”). Vous pouvez choisir différents types de paramètres, comme des chaînes de caractères, des booléens, et des listes déroulantes. En fonction du type que vous choisirez, vous devrez entrer des valeurs de configuration légèrement différentes, mais le processus de base est le même. Tous les types de paramètres, à l'exception du paramètre Fichier, ont un nom et une description, et le plus souvent une valeur par défaut.

Dans Figure 10.3, “Ajouter un paramètre à la tâche de build”, par exemple, nous ajoutons un paramètre appelé *version* à une tâche de déploiement. La valeur par défaut (`RELEASE`) sera initialement affichée lorsque Jenkins demandera à l'utilisateur de valoriser ce paramètre, donc si l'utilisateur ne change rien, cette valeur sera utilisée.



Figure 10.2. Ajouter un paramètre à la tâche de build

Quand l'utilisateur démarre un build paramétré (les builds paramétrés sont très souvent démarrés manuellement), Jenkins propose une page dans laquelle l'utilisateur peut entrer une valeur pour chacun des paramètres de la tâche de build (voir Figure 10.3, “Ajouter un paramètre à la tâche de build”).

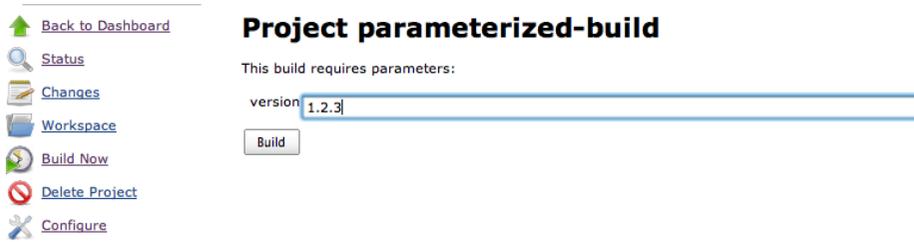


Figure 10.3. Ajouter un paramètre à la tâche de build

10.2.2. Adapter vos build pour travailler avec des scripts de builds paramétrés

Une fois que vous avez ajouté un paramètre, vous devez configurer vos scripts de build pour l'utiliser. Bien choisir le nom du paramètre est important, parce que c'est aussi le nom de la variable que Jenkins passera comme variable d'environnement lorsqu'il lance la tâche de build. Pour illustrer cela, considérons la configuration de build très basique de Figure 10.4, “Démonstration d'un paramètre de build”, où on affiche simplement le paramètre de build en retour dans la console. Notez que, pour rendre les variables plus portables à travers différents systèmes d'exploitation, il est préférable de les mettre en majuscules.

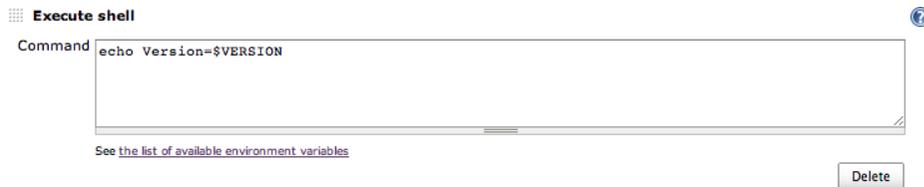


Figure 10.4. Démonstration d'un paramètre de build

Quand on exécute cela, on obtient les lignes suivantes dans la console :

```
Started by user anonymous
Building on master
[workspace] $ /bin/sh -xe /var/folders/y+/y+a+wZ-jG6WKHEm9KwnSvE+++TI/-Tmp-/
jenkins5862957776458050998.sh
+ echo Version=1.2.3
Version=1.2.3
Notifying upstream projects of job completion
Finished: SUCCESS
```

Vous pouvez aussi utiliser ces variables d'environnement au sein de vos scripts de build. Par exemple, dans un build Ant ou Maven, vous pouvez utiliser la propriété spéciale `env` pour accéder aux variables d'environnement courantes :

```
<target name="printversion">
```

```
<property environment="env" />
<echo message="${env.VERSION}" />
</target>
```

Une autre option consiste à passer les paramètres au script de build comme une valeur de propriété. Ce qui suit est un exemple plus pratique d'un fichier POM de Maven. Dans cet exemple, Maven est configuré pour déployer un fichier WAR spécifique. Nous fournissons la version du fichier WAR à déployer dans la propriété `target.version`, qui est utilisé dans la déclaration de la dépendance, comme montré ci-dessous :

```
...
<dependencies>
  <dependency>
    <groupId>com.wakaleo.gameoflife</groupId>
    <artifactId>gameoflife-web</artifactId>
    <type>war</type>
    <version>${target.version}</version>
  </dependency>
</dependencies>
<properties>
  <target.version>RELEASE</target.version>
  ...
</properties>
```

Quand on invoque Maven, on passe le paramètre comme l'une des propriétés du build (voir Figure 10.5, “Ajouter un paramètre à la tâche de build Maven”). On peut ensuite utiliser un outil comme Cargo pour faire le déploiement réel — Maven téléchargera la version demandée du WAR depuis le gestionnaire de dépôt d'entreprise, et la déploiera sur un serveur d'application.



Figure 10.5. Ajouter un paramètre à la tâche de build Maven

En résumé, cela montre comment il vous est possible d'intégrer des paramètres de tâches de build dans vos builds. Toutefois, en plus des bons vieux paramètres de type chaîne de caractères, il existe quelques types de paramètres plus sophistiqués, que nous regarderons dans les paragraphes suivants (voir Figure 10.6, “Différents types de paramètres sont disponibles”).

Project parameterized-build

This build requires parameters:

The screenshot shows a build configuration interface with the following parameters:

- VERSION**: Text input field containing "1.2.3".
- PASSWORD**: Password input field with masked characters "*****".
- COLOR**: Dropdown menu with "red" selected.
- Choose File**: File selection button with "No file chosen" text.
- RUN_FULL_TESTS**: Checked checkbox.
- GAME_OF_LIFE_JOB**: Dropdown menu with "game-of-life #197" selected.

A "Build" button is located at the bottom left of the configuration area.

Figure 10.6. Différents types de paramètres sont disponibles

10.2.3. Types de paramètres plus avancés

Les paramètres Mot de passe sont, comme vous pourriez vous y attendre, très similaires aux paramètres String, mis à part qu'ils sont affichés comme des champs mot de passe.

Il existe plusieurs cas où vous souhaiteriez présenter un ensemble limité d'options de paramètres. Dans un build de déploiement, vous pourriez permettre à l'utilisateur de choisir parmi un ensemble de serveurs cibles. Ou vous pourriez présenter une liste de navigateurs supportés pour une suite de tests d'acceptation. Les paramètres Choix vous permettent de définir un ensemble de valeurs qui seront affichées dans une liste déroulante (voir Figure 10.7, "Configurer un paramètre Choix"). Vous devez fournir une liste de valeurs possibles, une par ligne, en commençant par la valeur par défaut.

The screenshot shows the configuration interface for a "Choice" parameter:

- Name**: Text input field containing "COLOR".
- Choices**: Text area containing a list of values: "red", "green", and "blue".
- Description**: Empty text area.

Each input field has a help icon (question mark) to its right.

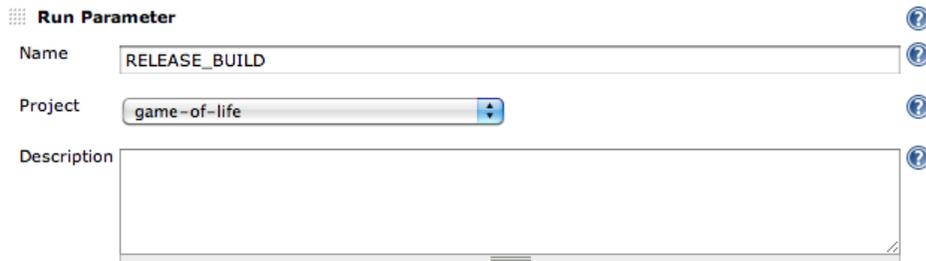
Figure 10.7. Configurer un paramètre Choix

Les paramètres Booléen sont, comme vous vous y attendez, des paramètres qui prennent comme valeur `true` ou `false`. Ils sont présentés en tant que case à cocher.

Deux types de paramètres plus exotiques, qui se comportent un peu différemment des autres, sont les paramètres Run et les paramètres Fichier.

Les paramètres Run vous permettent de sélectionner une exécution particulière (ou un build) d'un build donné (voir Figure 10.8, "Configurer un paramètre Run"). L'utilisateur effectue une sélection à partir

d'une liste de numéros d'exécution de build. L'URL correspondant à l'exécution est stockée dans le paramètre spécifié.



Run Parameter

Name:

Project:

Description:

Figure 10.8. Configurer un paramètre Run

L'URL (qui devrait ressembler à `http://jenkins.myorg.com/job/game-of-life/197/`) peut être utilisée pour obtenir une information ou des artefacts d'une exécution de build. Par exemple, vous pourriez récupérer le fichier JAR ou WAR archivé lors d'un build précédent et exécuter des tests plus poussés sur celui-ci dans une tâche de build séparée. Ainsi, pour accéder au fichier WAR d'un build précédent dans un projet Maven multimodules, l'URL ressemblerait à celle-ci :

```
http://buildserver/job/game-of-life/197/artifact/gameoflife-web/target/  
gameoflife.war
```

Donc, en utilisant le paramètre configuré dans Figure 10.8, “Configurer un paramètre Run”, vous pourriez accéder au fichier WAR en utilisant l'expression suivante :

```
${RELEASE_BUILD}gameoflife-web/target/gameoflife.war
```

Les paramètres Fichier vous permettent de télétransférer un fichier dans l'espace de travail de la tâche de build, afin qu'il puisse être utilisé dans le script de build (voir Figure 10.9, “Configurer un paramètre Fichier”). Jenkins stockera le fichier à l'emplacement spécifié dans l'espace de travail du projet, où vous pouvez y accéder dans vos scripts de build. Vous pouvez utiliser la variable `WORKSPACE` pour faire référence au répertoire de l'espace de travail courant, afin que vous puissiez manipuler le fichier télétransféré dans Figure 10.9, “Configurer un paramètre Fichier” en utilisant l'expression `${WORKSPACE}/deploy/app.war`.



File Parameter

File location:

Description:

Figure 10.9. Configurer un paramètre Fichier

10.2.4. Construire à partir d'un tag Subversion

Le déclenchement paramétré possède un support spécial pour Subversion, vous permettant ainsi de réaliser un build à partir d'un tag Subversion spécifique. C'est utile si vous voulez lancer un build de release en utilisant un tag généré par un build précédent. Par exemple, une tâche de build amont pourrait créer un tag d'une révision particulière. Vous pourriez aussi utiliser le processus standard de release Maven (voir Section 10.7.1, “Gestion des releases Maven avec le plugin M2Release”) pour générer une nouvelle release. Dans ce cas, un tag avec le numéro de release Maven sera automatiquement généré dans Subversion.

Cette approche est utile pour des projets qui ont besoin d'être partiellement ou entièrement reconstruits avant de pouvoir être déployés sur une plateforme donnée. Par exemple, vous pourriez avoir besoin d'exécuter le build Ant ou Maven en utilisant différentes propriétés ou profils pour différentes plateformes, afin que les fichiers de configuration spécifiques puissent être embarqués dans les WAR ou EAR déployés.

Vous pouvez configurer un build Jenkins pour qu'il s'exécute sur un tag sélectionné en utilisant le type de paramètre “List Subversion Tag” (voir Figure 10.10, “Ajouter des paramètres pour réaliser un build à partir d'un tag Subversion”). Vous devez simplement fournir l'URL du dépôt Subversion pointant sur le répertoire des tags de votre projet.



The screenshot shows the Jenkins configuration page for a parameter. At the top, there is a checkbox labeled "This build is parameterized" which is checked. Below it, the parameter is titled "List Subversion tags". The "Name" field contains the text "Release". The "Repository URL" field contains the text "svn://localhost/gameoflife/tags". There are help icons (question marks) next to the title and the repository URL field. At the bottom right of the parameter configuration area is a "Delete" button. At the bottom left is an "Add Parameter" button with a dropdown arrow.

Figure 10.10. Ajouter des paramètres pour réaliser un build à partir d'un tag Subversion

Quand vous exécuterez ce build, Jenkins proposera une liste de tags dans laquelle choisir (voir Figure 10.11, “Réaliser un build à partir d'un tag Subversion”).

Project gameoflife-release-build

This build requires parameters:

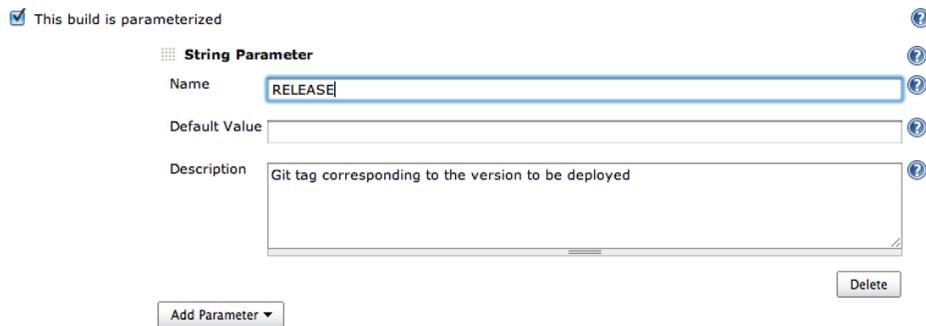
Release
 Select a Subversion tag

Figure 10.11. Réaliser un build à partir d'un tag Subversion

10.2.5. Réaliser un build à partir d'un tag Git

Réaliser un build à partir d'un tag Git n'est pas aussi simple que de le faire à partir d'un tag Subversion, bien que vous puissiez toujours utiliser un paramètre pour indiquer quel tag utiliser. En effet, à cause de la nature même de Git, quand Jenkins obtient une copie du code source depuis Git, il clone le dépôt Git, en incluant tous les tags. Une fois que vous avez la dernière version du dépôt sur votre serveur Jenkins, vous pouvez ensuite procéder à la récupération d'une version en utilisant `git checkout <tagname>`.

Pour configurer cela avec Jenkins, vous devez commencer par ajouter un paramètre String à votre tâche de build (appelée `RELEASE` dans cet exemple — voir Figure 10.12, “Configurer un paramètre pour un tag Git”). Contrairement au support Subversion, il n'est pas possible de lister les tags Git disponibles dans une liste déroulante, les utilisateur devront donc connaître le nom du tag qu'ils veulent livrer.



The image shows the Jenkins configuration page for a String Parameter. At the top, there is a checkbox labeled "This build is parameterized" which is checked. Below it, the parameter configuration is shown for a "String Parameter". The "Name" field contains the text "RELEASE". The "Default Value" field is empty. The "Description" field contains the text "Git tag corresponding to the version to be deployed". There are help icons (question marks) next to the "Name", "Default Value", and "Description" fields. At the bottom left, there is an "Add Parameter" button with a dropdown arrow. At the bottom right, there is a "Delete" button.

Figure 10.12. Configurer un paramètre pour un tag Git

Une fois que vous avez ajouté ce paramètre, vous devez faire un checkout du tag correspondant une fois que le dépôt a été cloné localement. Ainsi, si vous avez un build freestyle, la première étape du build sera un appel en ligne de commande à Git pour faire un checkout du tag référencé par le paramètre `RELEASE` (voir Figure 10.13, “Réaliser un build à partir d'un tag Git”). Bien sûr, un moyen plus portable de faire cela serait d'écrire un simple script Ant ou Groovy pour faire l'équivalent d'une façon plus indépendante du système d'exploitation.

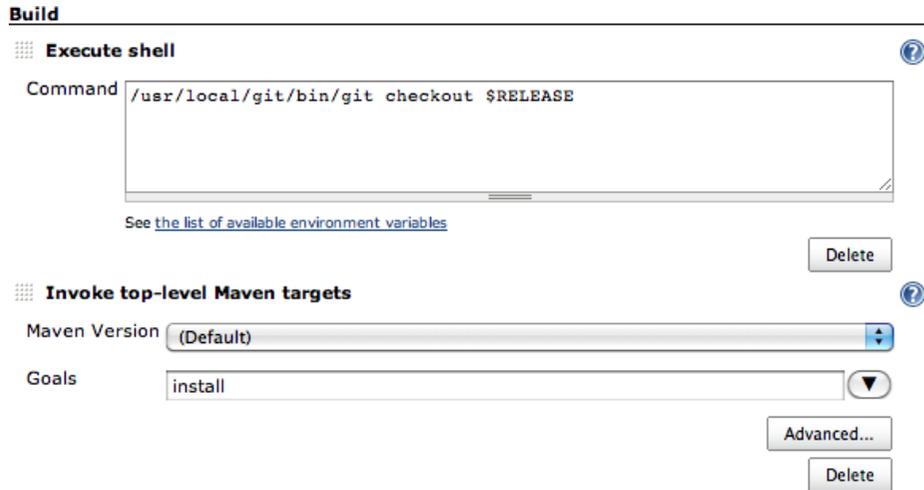


Figure 10.13. Réaliser un build à partir d'un tag Git

10.2.6. Démarrer une tâche de build paramétrée à distance

Vous pouvez aussi démarrer une tâche de build paramétrée à distance, en invoquant l'URL de la tâche de build. La forme typique d'une URL de tâche de build est illustrée ci-après :

```
http://jenkins.acme.org/job/myjob/buildWithParameters?PARAMETER=Value
```

Ainsi, dans l'exemple ci-dessus, vous pourriez déclencher un build de la façon suivante :

```
http://jenkins.acme.org/job/parameterized-build/buildWithParameters?VERSION=1.2.3
```

Quand vous utilisez une URL pour démarrer une tâche de build de cette façon, rappelez-vous que les noms des paramètres sont sensibles à la casse, et que les valeurs doivent être échappées (comme n'importe quel autre paramètre HTTP). Et si vous utilisez un paramètre Run, vous devez fournir le nom de la tâche de build et le numéro d'exécution (e.g., `game-of-life#197`) et pas seulement le numéro d'exécution.

10.2.7. Historique des tâches de build paramétrées

Enfin, il est indispensable de savoir quels paramètres ont été utilisés pour lancer un build paramétré particulier. Par exemple, dans une tâche de build de déploiement automatisé, il est utile de savoir exactement quelle version a réellement été déployée. Heureusement, Jenkins stocke ces valeurs dans l'historique de build (voir Figure 10.14, “Jenkins stocke les valeurs des paramètres utilisées pour chaque build”), afin que vous puissiez toujours retrouver un ancien build et en vérifier les paramètres.



Figure 10.14. Jenkins stocke les valeurs des paramètres utilisées pour chaque build

10.3. Déclencheurs paramétrés

Quand vous déclenchez une autre tâche de build depuis une tâche de build paramétrée, il est souvent utile de pouvoir passer les paramètres du build courant au nouveau. Supposons, par exemple, que vous ayez une application qui doit être testée sur plusieurs bases de données différentes. Comme nous l'avons vu, vous pourriez faire cela en configurant une tâche de build paramétré qui accepte la base de données cible comme un paramètre. Vous sélectionneriez une série de builds, et tous auraient besoin de ce paramètre.

Si vous essayez de faire cela en utiliser l'option conventionnelle "Construire d'autres projets" dans la section Actions Post-Build, cela ne marchera pas. En effet, vous ne pouvez pas déclencher un build paramétré de cette façon.

Toutefois, vous pouvez le faire en utilisant le plugin Jenkins Parameterized Trigger. Ce plugin vous permet de configurer vos tâches de build à la fois pour déclencher des builds paramétrés et pour passer des paramètres arbitraires à ces builds.

Une fois que vous avez installé ce plugin, vous trouverez l'option "Déclencher des builds paramétrés sur d'autres projets" dans la page de configuration de votre tâche de build (voir Figure 10.16, "Ajouter un déclencheur paramétré à une tâche de build"). Ceci vous permet de démarrer une autre tâche de build de différentes façons. En particulier, vous pouvez lancer une tâche de build ultérieure, en passant les paramètres courant à cette nouvelle tâche, ce qui est impossible à faire avec un build paramétré normal. La meilleure façon de voir comment cela fonctionne est au travers d'un exemple.

Dans Figure 10.15, "Tâche de build paramétré pour des tests unitaires" nous avons une tâche de build initiale. Cette tâche prend un unique paramètre, `DATABASE`, qui spécifie la base de données à utiliser pour les tests. Comme nous l'avons vu, Jenkins demandera à l'utilisateur de saisir cette valeur à chaque fois qu'un build sera lancé.

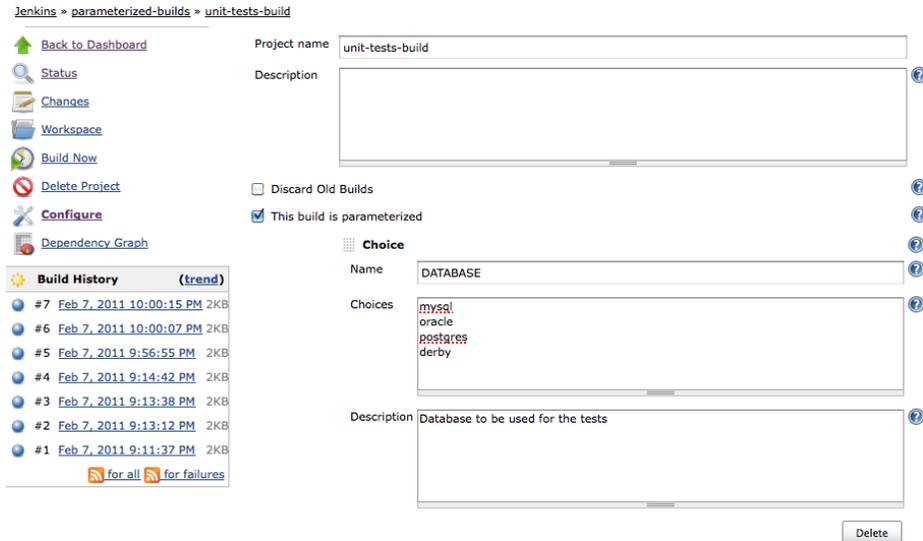


Figure 10.15. Tâche de build paramétré pour des tests unitaires

Supposons maintenant que nous voulons déclencher une seconde tâche de build pour exécuter des tests d'intégration plus complets une fois que la première tâche a terminé. Cependant nous avons besoin d'exécuter les tests sur la même base de données. On peut faire cela en configurant un déclencheur paramétré pour démarrer cette seconde tâche (voir Figure 10.16, "Ajouter un déclencheur paramétré à une tâche de build").



Figure 10.16. Ajouter un déclencheur paramétré à une tâche de build

Dans ce cas, nous passons simplement les paramètres du build courant. Cette seconde tâche de build démarrera automatiquement après la première, avec la valeur du paramètre `DATABASE` fournie par l'utilisateur. Vous pouvez aussi configurer finement la politique de déclenchement en indiquant à Jenkins quand le build doit être lancé. Typiquement, vous déclenchiez seulement un build aval après que votre build ait réussi, mais avec le plugin Parameterized Trigger vous pouvez aussi configurer les builds pour qu'ils déclenchent même si le build est instable, ou seulement quand le build échoue ou encore demander à ce qu'il soit déclenché quoi qu'il arrive au premier build. Vous pouvez même configurer plusieurs déclencheurs pour la même tâche de build.

Naturellement, la tâche de build que vous déclenchez doit être une tâche de build paramétré (comme illustré dans Figure 10.17, “La tâche de build que vous déclenchez doit aussi être une tâche paramétrée”), et vous devez transmettre tous les paramètres qu'elle requiert.

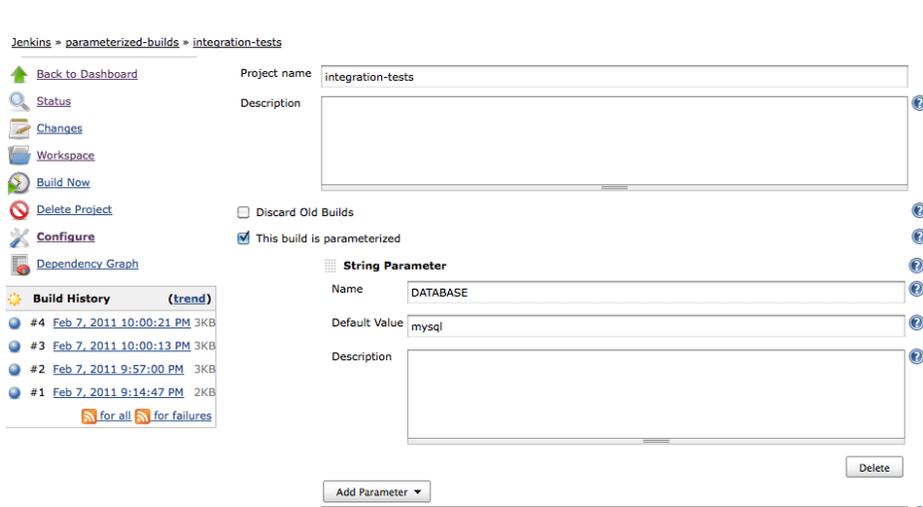


Figure 10.17. La tâche de build que vous déclenchez doit aussi être une tâche paramétrée

Cette fonctionnalité possède en fait des applications bien plus larges que de simplement transmettre les paramètres du build courant. Vous pouvez aussi déclencher une tâche de build paramétré avec un ensemble arbitraire de paramètres, ou utiliser une combinaison de paramètres passés au build courant, et vos propres paramètres traditionnels. Ou alors, si vous avez beaucoup de paramètres, vous pouvez les charger à partir d'un fichier de propriétés. Dans Figure 10.18, “Passer un paramètre prédéfini à une tâche de build paramétré”, nous passons à la fois les paramètres du build courant (la variable `DATABASE` dans ce cas), et un paramètre additionnel appelé `TARGET_PLATFORM`.

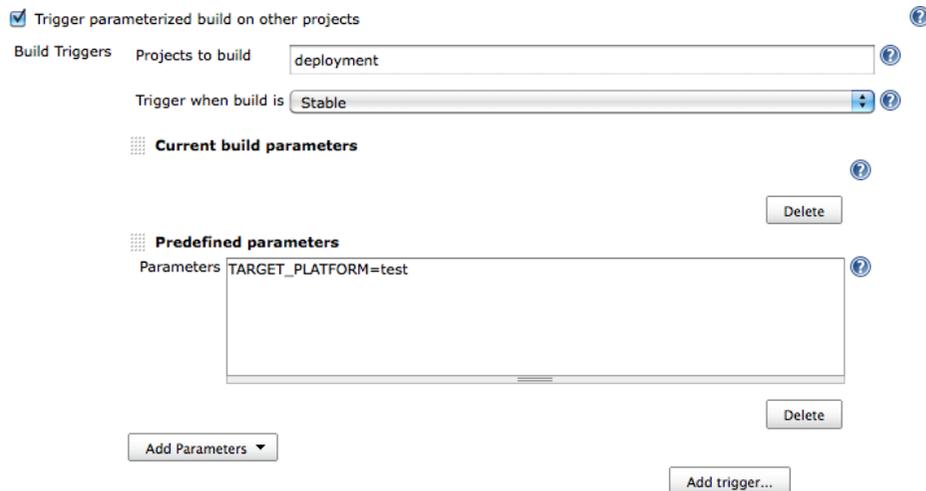


Figure 10.18. Passer un paramètre prédéfini à une tâche de build paramétré

10.4. Tâches de build multiconfiguration

Les tâches de build multiconfiguration sont une fonctionnalité extrêmement puissante de Jenkins. Une tâche de build multiconfiguration peut être vue comme une tâche de build paramétré qui peut être automatiquement lancée avec toutes les combinaisons possibles de paramètres qu'elle puisse accepter. Elles sont particulièrement utiles pour les tests, vous permettant ainsi de tester votre application avec une seule tâche de build, mais avec une grande variété de conditions (navigateurs, bases de données, et ainsi de suite).

10.4.1. Configurer un build multiconfiguration

Pour créer une nouvelle tâche de build multiconfiguration, choisissez simplement l'option suivante sur la page Nouvelle Tâche (voir Figure 10.19, “Créer une tâche de build multiconfiguration”).

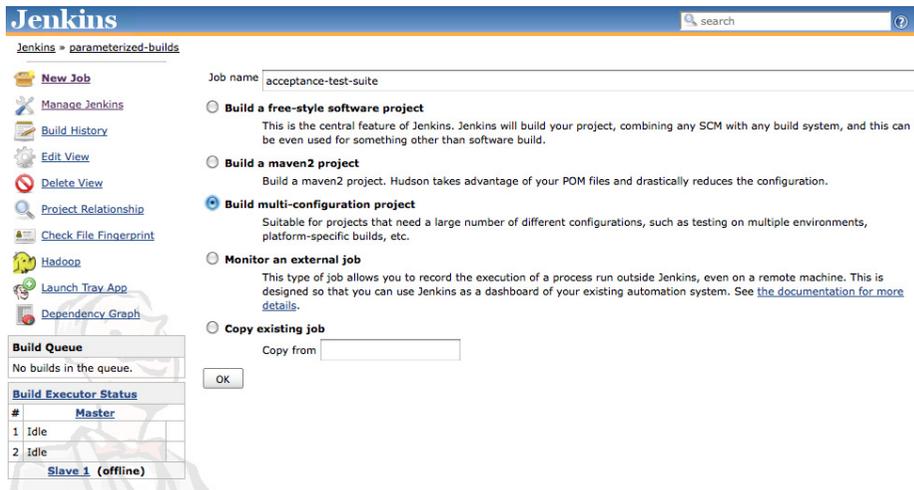


Figure 10.19. Créer une tâche de build multiconfiguration

Une tâche de build multiconfiguration ressemble à n'importe quelle autre tâche, mais avec un élément supplémentaire très important : la Matrice de Configuration (voir Figure 10.20, “Ajouter un axe à un build multiconfiguration”). C'est là que vous définissez les différentes configurations qui seront utilisées pour exécuter vos builds.



Figure 10.20. Ajouter un axe à un build multiconfiguration

Vous pouvez définir différents axes d'options de configuration, incluant l'exécution de la tâche de build sur différents esclaves ou sur différents JDKs, ou en fournissant vos propres propriétés personnalisées au build. Par exemple, dans les tâches de build discutées précédemment, nous pourrions vouloir tester notre application pour différentes bases de données et différents systèmes d'exploitation. Nous pourrions définir un axe définissant les machines esclaves avec différents systèmes d'exploitation sur lesquels nous voudrions faire tourner nos builds, et un autre axe définissant toutes les valeurs possibles de bases de données. Jenkins exécutera ensuite la tâche de build pour chaque base de données et chaque système d'exploitation possibles.

Regardons à présent les types d'axes que nous pouvons définir.

10.4.2. Configurer un axe Esclave

La première option consiste à configurer votre build pour exécuter simultanément le build sur différentes machines esclaves (voir Chapter 11, Builds distribués). Evidemment, l'idée d'avoir un ensemble de

machines esclaves est généralement pour que vous puissiez exécuter votre build sur n'importe laquelle. Mais il y a des cas où il est normal d'être plus sélectif. Par exemple, vous pourriez vouloir lancer vos tests sur Windows, Mac OS X, et Linux. Dans ce cas, vous créez un nouvel axe pour vos noeuds esclaves, comme montré dans Figure 10.21, "Définir un axe de noeuds esclave". Vous pouvez choisir les noeuds que vous voulez utiliser de deux façons : par label ou par noeud individuel. Utiliser des labels vous permet d'identifier des catégories de noeuds de construction (par exemple, des machines Windows), sans lier le build à aucune machine en particulier. C'est une option plus flexible, et elle rend plus facile l'extension de votre capacité de build si nécessaire. Parfois, cependant, vous pouvez réellement vouloir exécuter un build sur une machine spécifique. Dans ce cas, vous pouvez utiliser l'option "Noeuds individuels" et choisir la machine dans la liste.



Figure 10.21. Définir un axe de noeuds esclave

Si vous avez besoin de plus de flexibilité, vous pouvez aussi utiliser une Label Expression, ce qui vous permet de définir quels noeuds esclaves peuvent être utilisés pour un axe particulier en utilisant des expressions booléennes et des opérateurs logiques pour combiner les labels. Par exemple, supposons que vous avez défini des labels pour les machines esclaves en fonction du système d'exploitation ("windows", "linux") et des bases de données installées ("oracle", "mysql", "db2"). Pour définir un axe n'exécutant les tests que pour les machines Windows sur lesquelles est installé MySQL, vous pouvez utiliser une expression comme `windows && mysql`.

Nous traitons du travail avec des noeuds esclaves et les builds distribués plus en détail dans Chapter 11, Builds distribués.

10.4.3. Configurer un axe JDK

Si vous déployez votre application sur une large base client sur laquelle vous avez un contrôle limité de l'environnement cible, vous pouvez avoir besoin de tester votre application en testant différentes versions de Java. Dans ce genre de cas, il est utile de pouvoir mettre en place un axe JDK dans un build multiconfiguration. Quand vous ajoutez un axe JDK, Jenkins propose automatiquement la liste des versions de JDK dont il a connaissance (voir Figure 10.22, "Définir un axe de versions de JDK"). Si vous avez besoin d'utiliser des JDKs additionnels, ajoutez les simplement à votre page de configuration Jenkins.



Figure 10.22. Définir un axe de versions de JDK

10.4.4. Axe personnalisé

Le troisième type d'axe vous permet de définir différentes façons d'exécuter votre tâche de build, basées sur des variables arbitraires que vous définissez. Par exemple, vous pouvez fournir une liste de bases de données que vous avez besoin de tester, ou une liste de navigateurs à utiliser dans vos tests web. Ceci est très similaire aux paramètres pour une tâche de build paramétré, excepté que vous fournissez la liste complète des valeurs possibles, et que plutôt que d'interroger l'utilisateur pour qu'il entre une valeur, Jenkins lance le build avec toutes les valeurs fournies (Figure 10.23, "Définir un axe spécifique à l'utilisateur").



Figure 10.23. Définir un axe spécifique à l'utilisateur

10.4.5. Exécuter un Build Multiconfiguration

Une fois que vous avez configuré les axes, vous pouvez exécuter votre build multiconfiguration comme n'importe quel autre. Toutefois, Jenkins traitera chaque combinaison de variables comme un build séparé. Jenkins affiche les résultats agrégés dans un tableau, où toutes les combinaisons sont montrées (voir Figure 10.24, "Résultats de build multiconfiguration"). Si vous cliquez sur les boules, Jenkins vous emmènera aux résultats détaillés pour un build particulier.

[Back to Dashboard](#)

[Status](#)

[Changes](#)

[Workspace](#)

[Build Now](#)

[Delete Project](#)

[Configure](#)

[Dependency Graph](#)

Build History		(trend)
#3	Feb 12, 2011 1:18:52 PM	
#2	Feb 10, 2011 11:36:12 PM	50KB
#1	Feb 10, 2011 11:34:54 PM	25KB

for all for failures

Project acceptance-test-suite

Configuration Matrix

		OSX	linux	windows
mysql	Java 1.6	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Java 1.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
oracle	Java 1.6	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Java 1.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
postgres	Java 1.6	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Java 1.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
derby	Java 1.6	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
	Java 1.5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Figure 10.24. Résultats de build multiconfiguration

Par défaut, Jenkins exécutera les tâches de build en parallèle. Toutefois, il y a quelques cas où cela n'est pas une bonne idée. Par exemple, de nombreuses applications web Java utilisent des tests Selenium ou WebDriver s'exécutant sur une instance locale de Jetty automatiquement démarrée par la tâche. Les scripts de build de ce genre doivent être spécialement configurés pour pouvoir s'exécuter en parallèle sur la même machine, pour éviter les conflits de ports. L'accès concurrent à la base de données pendant les tests peut être une autre source de problèmes si la gestion de la concurrence n'est pas intégrée à la conception des tests. Si vos builds ne sont pas conçus pour fonctionner en parallèle, vous pouvez forcer Jenkins à exécuter les tests de manière séquentielle en cochant la case Exécuter chaque configuration séquentiellement en bas de la section Configuration de la matrice.

Par défaut, Jenkins exécutera toutes les combinaisons possibles des différents axes. Donc, dans l'exemple ci-dessus, nous avons trois environnements, deux JDKs et quatre bases de données. Ceci résulte en un total de 24 builds. Toutefois, dans certains cas, exécuter certaines combinaisons ne pourrait avoir aucun sens (ou n'être pas possible). Par exemple, supposons que vous ayez une tâche de build qui exécute des tests web automatisés. Si un axe contient les navigateurs web à tester (Firefox, Internet Explorer, Chrome, etc.) et un autre les systèmes d'exploitation (Linux, Windows, Mac OS), cela n'aurait pas beaucoup de sens d'exécuter Internet Explorer avec Linux ou Mac OS.

L'option de Filtre de Combinaison vous permet de mettre en place des règles définissant quelles combinaisons de variables sont valides. Ce champ est une expression booléenne Groovy qui utilise les

noms des variables que vous définissez pour chaque axe. L'expression doit valoir true pour que le build s'exécute. Par exemple, supposez que vous ayez une tâche de build exécutant des tests web dans différents navigateurs sur différents systèmes d'exploitation (voir Figure 10.25, “Mettre en place un filtre de combinaison”). Les tests nécessitent d'exécuter Firefox, Internet Explorer et Chrome, sur Windows, Mac OS X, et Linux. Toutefois Internet Explorer ne fonctionne que sous Windows, et Chrome ne fonctionne pas sous Linux.

The screenshot shows the 'Configuration Matrix' configuration interface. It features two main sections: 'User-defined Axis' and 'Slaves'.
1. 'User-defined Axis':

- Name: browser
- Values: iexplorer firefox chrome
- Buttons: Delete

- 'Slaves':
- Name: os
- Node/Label: os
 - Labels:
 - OSX (iMac Slave)
 - linux (Ubuntu Slave)
 - windows (Windows Slave)
 - Individual nodes: +
- Buttons: Delete
- Global options:
- Add axis: dropdown
- Run each configuration sequentially
- Combination Filter
- Filter: (browser=="firefox") || (browser=="iexplorer" && os=="windows") || (browser=="chrome" && os != "linux")
- Execute touchstone builds first

Figure 10.25. Mettre en place un filtre de combinaison

Pour configurer un Filtre de Combinaison, nous pourrions utiliser une expression comme la suivante :

```
(browser=="firefox")  
|| (browser=="iexplorer" && os=="windows")  
|| (browser=="chrome" && os != "linux")
```

Ceci résulterait dans le fait que seules les combinaisons correctes navigateur/système d'exploitation seraient exécutées (voir Figure 10.26, “Résultats de build utilisant un filtre de combinaison”). Les builds exécutés sont affichés dans les couleurs habituelles, alors que les builds non-exécutés sont grisés.

Jenkins » parameterized-builds » acceptance-test-suite

[Back to Dashboard](#)

[Status](#)

[Changes](#)

[Workspace](#)

[Build Now](#)

[Delete Project](#)

[Configure](#)

[Dependency Graph](#)

Build History [\(trend\)](#)

#15 Feb 12, 2011 3:05:12 PM

Project acceptance-test-suite

Configuration Matrix

	iexplorer	firefox	chrome
OSX	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
linux	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
windows	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 10.26. Résultats de build utilisant un filtre de combinaison

Une autre raison d'utiliser un filtre de build est qu'il y a simplement trop de combinaisons valides pour s'exécuter dans un temps raisonnable. Dans ce cas, la meilleure solution pourrait être d'augmenter la capacité de votre serveur de build. La deuxième meilleure solution, d'un autre côté, serait d'exécuter uniquement un sous-ensemble des combinaisons, éventuellement exécutant l'ensemble complet de combinaison pendant la nuit. Vous pouvez faire cela en utilisant la variable spéciale `index`. Si vous incluez l'expression `(index%2 == 0)`, par exemple, cela assurera que seulement un build sur deux est en fait exécuté.

Vous pourriez aussi vouloir que certains builds s'exécutent avant les autres, comme tests de cohérence. Par exemple, vous pouvez vouloir exécuter la configuration par défaut (et, théoriquement, la plus fiable) pour votre application en premier, avant de continuer avec des combinaisons plus exotiques. Pour faire cela, vous pouvez utiliser l'option "Execute touchstone builds first". Ici, vous entrez une valeur de filtre (comme celle montrée ci-dessus) pour définir le ou les premiers builds à exécuter. Vous pouvez aussi spécifier si le build devrait continuer seulement si ces builds sont réussis, ou même s'ils échouent. Une fois que ces builds se sont terminés comme prévu, Jenkins procédera au lancement des autres combinaisons.

10.5. Générer vos tâches de build Maven automatiquement

Rédigé par Evgeny Goldin

Comme mentionné dans la section précédente, le nombre de tâches de build que votre serveur Jenkins héberge peut varier. Lorsque ce nombre grossira, il deviendra plus difficile non seulement de les voir dans le tableau de bord Jenkins, mais aussi de les configurer. Imaginez ce que nécessiterait la

configuration de 20 à 50 tâches une par une ! De plus, plusieurs de ces tâches pourraient avoir en commun des éléments de configuration, comme des goals Maven ou des paramètres de configuration mémoire, ce qui résulte en une configuration dupliquée et un surplus de maintenance.

Par exemple, si vous décidez d'exécuter `mvn clean install` au lieu de `mvn clean deploy` pour vos tâches de release et de passer à des méthodes de déploiement alternatives, comme celles fournies par le plugin `Artifactory`¹, vous n'aurez plus d'autre choix que d'ouvrir toutes les tâches concernées et de les mettre à jour manuellement.

Sinon, vous pourriez tirer parti du fait que Jenkins est un outil simple et direct qui garde trace de toutes ses définitions sur le disque dans des fichiers bruts. En effet, vous pouvez mettre à jour les fichiers `config.xml` de vos tâches directement dans le répertoire `.jenkins/jobs` où ils sont conservés. Bien que cette approche fonctionnerait, elle est loin d'être idéale parce qu'elle implique un nombre assez important de sélections manuelles et de remplacements délicats dans des fichiers XML Jenkins.

Il y a une troisième façon d'atteindre le nirvana des mises à jour massives de tâches : générer vos fichiers de configuration automatiquement en utilisant une sorte de fichier de définition. Le `Maven Jenkins Plugin`² fait exactement cela : générer les fichiers `config.xml` pour toutes les tâches en utilisant des définitions standards conservées dans un unique fichier `pom.xml`.

10.5.1. Configurer une tâche

Quand vous configurez une tâche avec le `Maven Jenkins Plugin`, vous pouvez définir tous les éléments habituels de configuration, comme les goals Maven, l'emplacement du POM, les URL de dépôts, les adresses e-mail, le nombre de jours pendant lesquels conserver les logs, et ainsi de suite. Le plugin essaie de vous rapprocher au plus près de la configuration classique d'une tâche dans Jenkins.

Jetons un oeil à la tâche de build de `Google Guice`³ :

```
<job>
  <id>google-guice-trunk</id>
  <description>Building Google Guice trunk.</description>
  <descriptionTable>
    <row>
      <key>Project Page</key>
      <value>
        <a href="http://code.google.com/p/google-guice/">
          <b><code>code.google.com/p/google-guice</code></b>
        </a>
      </value>
      <escapeHTML>false</escapeHTML>
      <bottom>false</bottom>
    </row>
  </descriptionTable>
  <jdkName>jdk1.6.0</jdkName>
```

¹ <http://wiki.jenkins-ci.org/display/JENKINS/Artifactory+Plugin>

² <http://evgeny-goldin.com/wiki/Maven-jenkins-plugin>

³ <http://code.google.com/p/google-guice/>

```

<mavenName>apache-maven-3</mavenName>
<mavenOpts>-Xmx256m -XX:MaxPermSize=128m</mavenOpts>
<daysToKeep>5</daysToKeep>
<useUpdate>>false</useUpdate>
<mavenGoals>-e clean install</mavenGoals>
<trigger>
  <type>timer</type>
  <expression>0 0 * * *</expression>
</trigger>
<repository>
  <remote>http://google-guice.googlecode.com/svn/trunk/</remote>
</repository>
<mail>
  <recipients>jenkins@evgeny-goldin.org</recipients>
</mail>
</job>

```

Cette tâche utilise un certain nombre de configurations standards comme `<jdkName>`, `<mavenName>` et `<mavenOpts>`. Le code est récupéré à partir d'un dépôt Subversion (défini dans l'élément `<repository>`), et un `<trigger>` cron qui exécute la tâche pendant la nuit à 00:00. Les notifications Email sont envoyées aux personnes spécifiées avec l'élément `<mail>`. Cette configuration ajoute aussi un lien vers la page du projet dans le tableau de description généré automatiquement pour chaque tâche.

Cette tâche générée est affichée dans votre serveur Jenkins comme illustré dans Figure 10.27, "Une tâche générée avec le Maven Jenkins plugin".

The screenshot shows the Jenkins web interface for a job named "Project google-guice-trunk". The job definition is generated by Maven. A warning message states: "If you configure this project manually - it will probably be overwritten!". The job is currently building "Building Google Guice trunk".

Project Page	code.google.com/p/google-guice
Job	google-guice-trunk
Job type	Maven2
Maven goals	-e clean install
Maven repository	"cuser-home"/.m2/repository"
Maven options	-Xmx256m -XX:MaxPermSize=128m
Mail recipients	jenkins@evgeny-goldin.org
SVN update policy	Revert - [false], update - [false], checkout - [true]
Node	master
Triggers	• timer: "0 0 * * *"
Repositories	• http://google-guice.googlecode.com/svn/trunk
POM	• http://google-guice.googlecode.com/svn/trunk/pom.xml

Figure 10.27. Une tâche générée avec le Maven Jenkins plugin

Voici une autre tâche réalisant la build de la branche master du projet Jenkins hébergé chez GitHub :

```

<job>
  <id>jenkins-master</id>
  <jdkName>jdk1.6.0</jdkName>
  <numToKeep>5</numToKeep>

```

```

<mavenName>apache-maven-3</mavenName>
<trigger>
  <type>timer</type>
  <expression>0 1 * * *</expression>
</trigger>
<scmType>git</scmType>
<repository>
  <remote>git://github.com/jenkinsci/jenkins.git</remote>
</repository>
<mail>
  <recipients>jenkins@evgeny-goldin.org</recipients>
  <sendForUnstable>>false</sendForUnstable>
</mail>
</job>

```

Elle génère la tâche montrée dans Figure 10.28, “Tâche générée jenkins-master”.

Job	jenkins-master
Job type	Maven2
Maven goals	-e clean install
Maven repository	"~user-home/.m2/repository"
Mail recipients	jenkins@evgeny-goldin.org
Node	master
Triggers	• timer: "0 1 * * *"
Repositories	• git://github.com/jenkinsci/jenkins.git : master
POM	• http://github.com/jenkinsci/jenkins/blob/master/pom.xml

Figure 10.28. Tâche générée jenkins-master

La documentation⁴ du plugin fournit une référence détaillée de tous les paramètres qui peuvent être configurés.

10.5.2. Réutiliser une configuration de tâche par héritage

Être capable de générer des tâches Jenkins jobs en utilisant une configuration centralisée, comme un POM Maven, résout le problème de la création et de la mise à jour de plusieurs tâches à la fois. Tout ce que vous avez à faire est de modifier les définitions de job, relancer le plugin et charger les définitions mises à jour avec Administrer Jenkins#“Recharger la configuration à partir du disque”. Cette approche a aussi l’avantage de rendre facile le stockage de vos configurations de tâche dans un système de gestion de versions, ce qui rend par la même plus facile le suivi des changements faits aux configurations de build.

Cela ne résout toutefois pas le problème consistant à maintenir des tâches qui partagent un certain nombre de propriétés identiques, comme les goals Maven, les destinataires email ou l'URL du dépôt

⁴ <http://evgeny-goldin.com/wiki/Maven-jenkins-plugin#.3Cjob.3E>

de code. Pour cela, le Maven Jenkins Plugin fournit de l'héritage de tâches, démontré dans l'exemple suivant:

```
<jobs>
  <job>
    <id>google-guice-inheritance-base</id>
    <abstract>true</abstract>
    <jdkName>jdk1.6.0</jdkName>
    <mavenName>apache-maven-3</mavenName>
    <daysToKeep>5</daysToKeep>
    <useUpdate>true</useUpdate>
    <mavenGoals>-B -e -U clean install</mavenGoals>
    <mail><recipients>jenkins@evgeny-goldin.org</recipients></mail>
  </job>

  <job>
    <id>google-guice-inheritance-trunk</id>
    <parent>google-guice-inheritance-base</parent>
    <repository>
      <remote>http://google-guice.googlecode.com/svn/trunk/</remote>
    </repository>
  </job>

  <job>
    <id>google-guice-inheritance-3.0-rc3</id>
    <parent>google-guice-inheritance-base</parent>
    <repository>
      <remote>http://google-guice.googlecode.com/svn/tags/3.0-rc3/</remote>
    </repository>
  </job>

  <job>
    <id>google-guice-inheritance-2.0-maven</id>
    <parent>google-guice-inheritance-base</parent>
    <mavenName>apache-maven-2</mavenName>
    <repository>
      <remote>http://google-guice.googlecode.com/svn/branches/2.0-maven/
      </remote>
    </repository>
  </job>
</jobs>
```

Dans cette configuration, `google-guice-inheritance-base` est une tâche parent abstraite contenant toutes les propriétés communes : le nom du JDK, le nom de Maven, le nombre de jours de conservation des logs, la politique de mise à jour SVN, les goals Maven et les destinataires email. Les trois tâches suivantes sont très courtes, spécifiant simplement qu'elles étendent une tâche `<parent>` et ajoutent les configurations manquantes (URLs de dépôt dans ce cas). Une fois générées, elles héritent de toutes les propriétés de la tâche parente automatiquement.

Toute propriété héritée peut être rédéfinie, comme démontré dans la tâche `google-guice-inheritance-2.0-maven` où Maven 2 est utilisé à la place de Maven 3. Si vous voulez "annuler" une propriété héritée, vous devrez la redéfinir avec une valeur vide.

L'héritage de tâches est un concept très puissant qui permet aux tâches de former des groupes hiérarchiques de n'importe quel type et dans n'importe quel but. Vous pouvez grouper vos tâches d'IC, nocturnes ou de release de cette façon, en centralisant les déclencheurs d'exécution partagés, les goals Maven ou les destinataires email dans des tâches parentes. Cette approche emprunté au monde orienté object permet de résoudre le problème de maintenance de tâches partageant un certain nombre de propriétés identiques.

10.5.3. Le support des plugins

En plus de configurer une tâche et de réutiliser ses définitions, vous pouvez bénéficier d'un support spécial pour un certain nombre de plugins Jenkins. À l'heure actuelle, une utilisation simplifiée des plugins Parameterized Trigger et Artifactory est fournie, un support pour d'autres plugins populaires est prévu dans de futures versions.

Ci-dessous se trouve un exemple d'invocation de tâches avec le plugin Parameterized Trigger. Utiliser cette option suppose que vous avez déjà ce plugin installé :

```
<job>
  <id>google-guice-inheritance-trunk</id>
  ...
  <invoke>
    <jobs>
      google-guice-inheritance-3.0-rc3,
      google-guice-inheritance-2.0-maven
    </jobs>
  </invoke>
</job>

<job>
  <id>google-guice-inheritance-3.0-rc3</id>
  ...
</job>

<job>
  <id>google-guice-inheritance-2.0-maven</id>
  ...
</job>
```

L'élément `<invoke>` vous permet d'invoquer d'autres tâches chaque fois que la tâche courante se termine correctement. Vous pouvez créer un pipeline de tâches de cette façon, vous assurant que chaque tâche du pipeline invoque la suivante. Notez que s'il y a plus d'un exécuteur Jenkins disponible au moment de l'invocation, les tâches spécifiées démarreront en parallèle. Pour une exécution en série, vous devrez connecter chaque tâche amont à une tâche aval avec `<invoke>`.

Par défaut, l'invocation ne se fait que quand la tâche courante est stable. Ceci peut être modifié, comme montré dans les exemples suivants :

```
<invoke>
```

```
<jobs>jobA, jobB, jobC</jobs>
<always>true</always>
</invoke>

<invoke>
  <jobs>jobA, jobB, jobC</jobs>
  <unstable>true</unstable>
</invoke>

<invoke>
  <jobs>jobA, jobB, jobC</jobs>
  <stable>>false</stable>
  <unstable>>false</unstable>
  <failed>true</failed>
</invoke>
```

La première invocation dans l'exemple ci-dessus invoque toujours les tâches aval. Ceci peut être utilisé pour un pipeline de tâches qui devraient toujours être exécutées même si certaines, ou leurs tests, échouent.

La seconde invocation dans l'exemple ci-dessus invoque les tâches aval même si une tâche amont est instable : l'invocation prend place quels que soient les résultats des tests. Cela peut être utilisé pour un pipeline de tâches moins sensibles aux tests et à leurs échecs.

La troisième invocation ci-dessus invoque les tâches aval seulement quand une tâche amont échoue mais pas lorsqu'elle est stable ou instable. Cette configuration peut vous être utile si une tâche en échec doit effectuer des actions additionnelles autres que les notifications email traditionnelles.

Artifactory⁵ est un dépôt de binaires à usage général qui peut être utilisé comme gestionnaire de dépôt Maven. Le plugin Jenkins Artifactory⁶, montré dans Figure 10.29, “Configuration du plugin Jenkins pour Artifactory”, fournit un certain nombre d'avantages pour les tâches de build Jenkins. Nous avons déjà passé en revue quelques-unes d'entre elles dans Section 5.9.4, “Déployer vers un gestionnaire de dépôt d'entreprise”, notamment la capacité à déployer des artefacts à l'achèvement de la tâche ou d'envoyer avec des informations de l'environnement de build avec les artefacts pour une meilleure traçabilité.

⁵ <http://jfrog.org>

⁶ <http://wiki.jenkins-ci.org/display/JENKINS/Artifactory+Plugin>

Post-build Actions

- Build other projects
- Archive the artifacts
- Aggregate downstream test results
- Deploy artifacts to Maven repository
- Deploy artifacts to Artifactory

Artifactory Configuration

Artifactory server

Target releases repository

Target snapshots repository

Override default deployer credentials

Deploy even if the build is unstable

Deploy maven artifacts

Check if you wish to publish produced build artifacts to Artifactory.

Include Patterns

Exclude Patterns

Capture and publish build info

Include All Environment Variables

Figure 10.29. Configuration du plugin Jenkins pour Artifactory

Vous pouvez aussi utiliser le plugin Jenkins Artifactory conjointement au Maven Jenkins Plugin pour déployer dans Artifactory, comme montré dans l'exemple suivant :

```
<job>
  ...
  <artifactory>
    <name>http://artifactory-server/</name>
    <deployArtifacts>true</deployArtifacts>
    <includeEnvVars>true</includeEnvVars>
    <evenIfUnstable>true</evenIfUnstable>
  </artifactory>
</job>
```

Les informations d'identité pour le déploiement sont spécifiées dans la configuration de Jenkins dans l'écran Administrer Jenkins#Configurer le système. Elles peuvent aussi être spécifiées pour chaque tâche Jenkins. Les dépôts Maven par défaut sont `libs-releases-local` et `libs-snapshots-local`. Vous trouverez plus de détails dans la documentation du plugin à l'adresse <http://wiki.jenkins-ci.org/display/JENKINS/Artifactory+Plugin>.

10.5.4. Les tâches Freestyle

En supplément des tâches Maven, le Maven Jenkins Plugin vous permet de configurer des tâches freestyle Jenkins. Un exemple est montré ici :

```
<job>
  <id>free-style</id>
  <jobType>free</jobType>
  <scmType>git</scmType>
  <repository>
    <remote>git://github.com/evgeny-goldin/maven-plugins-test.git</remote>
  </repository>
  <tasks>
    <maven>
      <mavenName>apache-maven-3</mavenName>
      <jvmOptions>-Xmx128m -XX:MaxPermSize=128m -ea</jvmOptions>
      <properties>plugins-version = 0.2.2</properties>
    </maven>
    <shell><command>pwd; ls -al; du -hs .</command></shell>
  </tasks>
</job>
```

Les tâches Freestyle vous permettent d'exécuter un shell ou une commande batch, exécuter Maven ou Ant, et invoquer d'autres tâches. Elles fournissent un environnement d'exécution bien pratique pour les scripts systèmes ou tout autre type d'activité qui n'est pas directement implémentée dans Jenkins ou l'un de ses plugins. En utilisant cette approche, vous pouvez générer des fichiers de configuration de tâche de build Freestyle de façon similaire à l'approche que nous avons vue pour les tâches de build Maven, ce qui peut aider à rendre votre environnement de construction plus cohérent et maintenable.

10.6. Coordonner vos builds

Déclencher des tâches avals est assez facile. Toutefois, quand on met en place des configurations de tâches de build plus importantes et plus compliquées, on aimerait parfois être capable de lancer des exécutions simultanées, ou éventuellement attendre la fin de certaines tâches de build afin de continuer. Dans cette section, nous allons regarder les techniques et les plugins qui peuvent nous aider à faire cela.

10.6.1. Les builds parallèles dans Jenkins

Jenkins possède un support intégré pour les build parallèles — quand une tâche démarre, Jenkins va lui assigner le premier noeud de build disponible. Vous pouvez donc avoir potentiellement autant de builds parallèles en exécution que vous avez de noeuds disponibles.

Si vous avez besoin d'exécuter une légère variations de la même tâche de build en parallèle, les tâches de build multiconfiguration (voir Section 10.4, “Tâches de build multiconfiguration”) sont une excellente option. Ceci peut s'avérer très pratique comme moyen d'accélérer votre processus de build. Une application typique des tâches de build multiconfiguration dans ce contexte est d'exécuter des tests d'intégration en parallèle. Vous pourriez définir des profils Maven par exemple, ou configurer votre

build pour utiliser des paramètres de ligne de commande pour décider quels tests exécuter. Une fois que vous avez configuré vos scripts de build de cette façon, il est aisé de configurer une tâche de build multiconfiguration pour exécuter un sous ensemble de vos tests d'intégration en parallèle.

Vous pouvez aussi faire que Jenkins déclenche plusieurs tâches avales en parallèle, en les listant simplement dans le champ "Construire d'autres projets" (voir Figure 10.30, "Déclencher plusieurs autres builds après une tâche de build"). Les tâches de build suivantes seront exécutées en parallèle autant que possible. Toutefois, comme nous le verrons plus loin, cela peut ne pas toujours être exactement ce dont vous avez besoin.

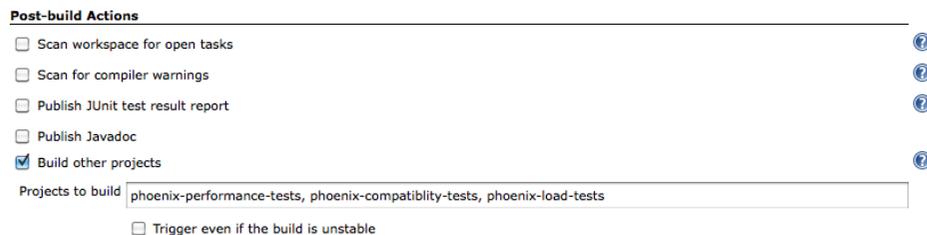


Figure 10.30. Déclencher plusieurs autres builds après une tâche de build

10.6.2. Graphes de dépendance

Avant d'étudier les points les plus fins des builds parallèles, il est utile de pouvoir visualiser les relations entre vos tâches de build. Le plugin Dependency Graph View analyse vos tâches de build et affiche un graphe décrivant les connexions amont et aval entre vos tâches. Ce plugin utilise graphviz⁷, que vous aurez besoin d'installer sur votre serveur si vous ne l'avez pas déjà.

Ce plugin ajoute une icône Graphe de dépendance dans le menu principal, qui affiche un graphe montrant les relations entre toutes les tâches de build dans votre projet (au niveau tableau de bord), ou toutes les tâches de build liées à la tâche de build courante (quand vous êtes à l'intérieur d'un projet particulier [voir Figure 10.31, "Un graphe de dépendance de tâche de build"]). De plus, si vous cliquez sur une tâche de build dans le graphe, Jenkins vous emmènera directement vers la page projet de cette tâche de build.

⁷ <http://www.graphviz.org>

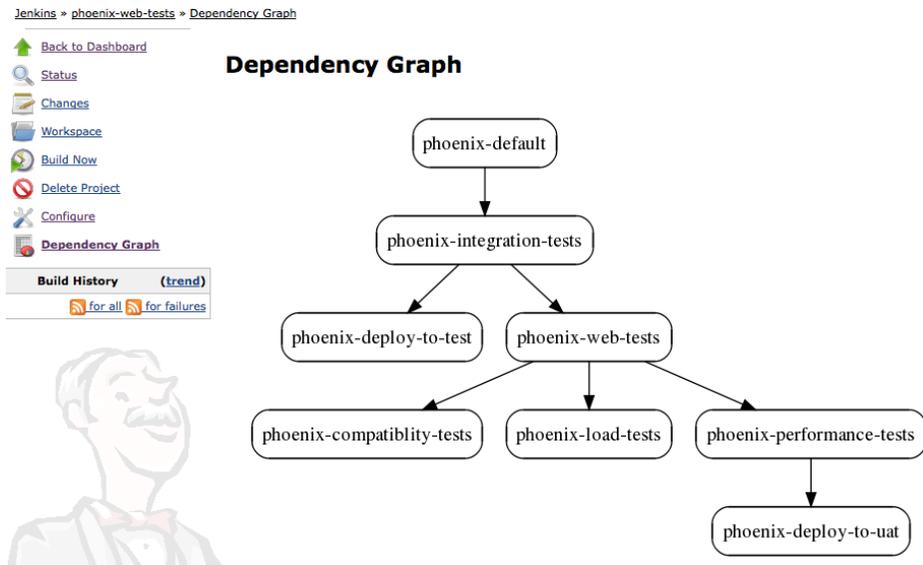


Figure 10.31. Un graphe de dépendance de tâche de build

10.6.3. Jonctions

Lors de la configuration de pipelines de builds plus compliqués, vous rencontrerez fréquemment des situations où une tâche de build ne peut démarrer tant qu'un certain nombre d'autres tâches de build ne sont pas terminées, mais que ces tâches amont ne nécessitent pas d'être exécutées séquentiellement. Par exemple, dans Figure 10.31, “Un graphe de dépendance de tâche de build”, imaginez que la tâche de build **phoenix-deploy-to-uat** ait en fait besoin que trois tâches réussissent avant qu'elle puisse être exécutée : **phoenix-compatibility-tests**, **phoenix-load-tests**, et **phoenix-performance-tests**.

On peut configurer cela en utilisant le plugin Joins, que vous devez installer de la façon habituelle via le centre de mise à jour. Une fois qu'il est installé, vous configurez une jonction dans la tâche de build qui initie le processus de jonction (ici, ce serait **phoenix-web-tests**). Dans notre exemple, nous devons modifier la tâche de build **phoenix-web-tests** afin qu'elle déclenche en premier **phoenix-compatibility-tests**, **phoenix-load-tests**, et **phoenix-performance-tests**, et ensuite, si ces trois réussissent, la tâche de build **phoenix-deploy-to-uat**.

Nous le faisons en configurant simplement le champ déclencheur de jonction avec le nom de la tâche de build **phoenix-deploy-to-uat** (voir Figure 10.32, “Configurer une jonction dans la tâche de build phoenix-web-tests”). Le champ “Construire d'autres projets” n'est pas modifié, et liste encore les tâches de build à déclencher immédiatement après la tâche courante. Le champ déclencheur de jonction contient les tâches de build à lancer une fois que toutes les tâches avals immédiates se sont terminées.

Post-build Actions

- Scan workspace for open tasks
- Publish JUnit test result report
- Publish Javadoc
- Build other projects

Projects to build

- Trigger even if the build is unstable

- Archive the artifacts
- Aggregate downstream test results
- Record fingerprints of files to track usage
- Publish Clover Coverage Report
- Publish Cobertura Coverage Report
- Set build description
- Publish documents
- Join Trigger

Projects to build once, after all downstream projects have finished

- Trigger even if some downstream projects are unstable
- Run post-build actions at join

Figure 10.32. Configurer une jonction dans la tâche de build phoenix-web-tests

Résultat, vous n'avez plus besoin du déclencheur de build original pour la tâche de build final, puisque c'est à présent redondant.

Ce nouveau déroulement apparaît bien dans les graphes de dépendance illustrés dans Figure 10.33, “Un graphe de dépendance de tâche de build plus compliqué”.

Dependency Graph

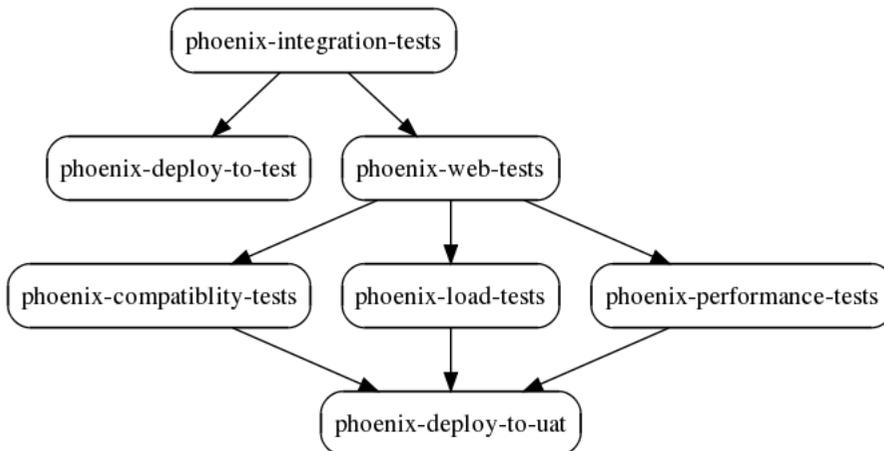


Figure 10.33. Un graphe de dépendance de tâche de build plus compliqué

10.6.4. Plugin Locks and Latches

Dans d'autres situations, vous pourriez être capable de lancer une série de builds en parallèle jusqu'à un certain point, mais certaines tâches de build pourraient ne pas pouvoir être lancées en parallèle parce qu'elles accèdent à des ressources en concurrence. Bien sûr, des tâches de build bien conçues devraient s'efforcer d'être aussi indépendantes que possible, mais cela peut parfois être difficile. Par exemple, différentes tâches de build peuvent accéder à la même base de données de test, ou à des fichiers sur le disque dur, et faire cela simultanément pourrait potentiellement compromettre le résultat des tests. Une tâche de build de performance pourrait avoir besoin d'un accès exclusif au serveur de test, afin d'avoir des résultats cohérents à chaque fois.

Le plugin Locks and Latches vous permet d'une certaine façon de contourner ce problème. Ce plugin permet de configuration des “verrous” (ndt: locks) pour certaines ressources, de façon similaire aux verrous en programmation multithreadée. Supposez, par exemple, dans les tâches de build dépeintes dans Figure 10.33, “Un graphe de dépendance de tâche de build plus compliqué”, que les tests de charge et les tests de performance soient exécutés sur un serveur dédié, mais qu'une seule tâche de build puisse être exécutée à la fois sur ce serveur. Imaginez de plus que les tests de performance pour les autres projets soient aussi exécutés sur ce serveur.

Pour éviter la contention sur le serveur de performance, vous pourriez utiliser le plugin Locks and Latches pour mettre en place un accès par réservation de “verrou” à ce serveur pour une tâche à un instant donné. Premièrement, dans la page de configuration du système, vous devez ajouter un nouveau verrou dans la section Verrous (voir Figure 10.34, “Ajouter un nouveau verrou”). Ce verrou sera ensuite disponible à toutes les tâches de build sur le serveur.



Figure 10.34. Ajouter un nouveau verrou

Ensuite, vous devez configurer chaque tâche de build qui utilisera la ressource en contention. Dans la section Environnement de build, vous trouverez un champ Verrous. Cochez la case et sélectionnez le verrous que vous venez juste de créer (voir Figure 10.35, “Configurer une tâche de build pour utiliser un verrou”). Une fois que vous avez fait cela pour chacune des tâches de build qui ont besoin d'accéder à la ressource en question, seule une des tâches de build pourra s'exécuter à un instant donné.

Build Environment

Maven3-Artifactory Integration (deprecated)
 Gradle-Artifactory Integration
 Locks

Name:

Set environment variables

Figure 10.35. Configurer une tâche de build pour utiliser un verrou

10.7. Pipelines de build et promotions

L'intégration continue ne consiste pas simplement à construire et tester automatiquement un logiciel, elle peut aussi apporter une aide dans un contexte plus large de développement de produit logiciel et de cycle de vie de release. Dans de nombreuses organisation, la vie d'une version particulière d'une application ou d'un produit démarre en développement. Lorsqu'on l'estime prête, elle est passée à l'équipe d'assurance qualité pour la tester. S'ils considèrent la version acceptable, ils la transmettent à des utilisateurs sélectionnés pour davantage de tests dans un environnement de tests d'acceptation. Si les utilisateurs sont contents, elle est envoyée en production. Bien sûr, il y a presque autant de variations de cela qu'il y a d'équipes de développement, mais un principe commun est que des versions spécifiques sont sélectionnées, selon certains critères de qualité, afin d'être "promues" à l'étape suivante du cycle de vie. Ceci est connu sous l'appellation promotion de build, et le processus plus global est connu sous le nom de pipeline de build. Dans cette section, nous regarderons comment implémenter des pipelines de build en utilisant Jenkins.

10.7.1. Gestion des releases Maven avec le plugin M2Release

Une part importante de tout pipeline de build est d'avoir une stratégie de release bien définie. Ceci implique, entre autres choses, de décider comment et quand lancer une nouvelle release, et comment l'identifier avec un libellé unique ou numéro de version. Si vous travaillez avec des projets Maven, utiliser le plugin Maven Release pour gérer les numéros de versions est une pratique hautement recommandée.

Les projets Maven utilisent des numéros de version bien définis et bien structurés. Un numéro de version typique est composé de trois digits (e.g., "1.0.1"). Les développeurs travaillent sur des versions SNAPSHOT (e.g., "1.0.1-SNAPSHOT"), qui, comme son nom l'indique, n'est pas conçu pour être définitif. Les releases définitives (e.g., "1.0.1") sont construites une seule fois et déployées dans le dépôt local d'entreprise (ou le dépôt central Maven pour les bibliothèques opensource), où elles peuvent à leur tour être utilisées par d'autres projets. Les numéros de version utilisés dans des artefacts Maven sont une part critique du système de gestion de dépendances Maven, et il est fortement conseillé de respecter les conventions Maven.

Le plugin Maven Release aide à automatiser le processus de mise à jour des numéros de version Maven de vos projets. En résumé, cela vérifie, construit et teste votre application, monte les numéros de version,

met à jour votre système de contrôle de versions avec les tags appropriés, et déploie les versions de release de vos artefacts dans votre dépôt Maven. C'est une tâche fastidieuse à faire manuellement, le plugin Maven Release est donc un excellent moyen d'automatiser les choses.

Toutefois, le plugin Maven Release peut aussi être capricieux. Des fichiers locaux non-archivés ou modifiés peuvent faire échouer le processus, par exemple. Le processus est aussi consommateur en temps et consomme intensivement le CPU, plus spécialement pour les gros projets : cela construit et exécute entièrement l'ensemble de tests unitaires et d'intégration plusieurs fois, récupère une copie propre du code depuis le dépôt, et envoie plusieurs artefacts au dépôt d'entreprise. Concrètement, ce n'est pas le genre de chose que vous voulez exécuter sur une machine de développeur.

Il est donc de bon ton d'exécuter ce processus sur votre serveur de build.

Une façon de faire cela est de configurer une tâche manuelle de build spéciale invoquant le plugin Maven Release. Toutefois, le plugin M2Release propose une approche plus simple. En utilisant ce plugin, vous pouvez ajouter la possibilité de construire une version de release Maven à une tâche existante. Vous pouvez ainsi éviter de dupliquer des tâches de builds inutilement, facilitant par la même la maintenance du serveur.

Une fois ce plugin installé, vous pouvez configurer toute tâche de build pour qu'elle propose une étape manuelle de release Maven. Ceci s'effectue en cochant la case “Maven release build” dans la section Environnement de Build (voir Figure 10.36, “Configurer une release Maven en utilisant le plugin M2Release”). Ici, vous définissez les goals que vous voulez exécuter pour le build (typiquement `release:prepare release:perform`).

Build Environment

Maven release build

Release goals and options ?

Preselect versioning mode ?

Preselect custom SCM comment prefix ?

Preselect append Hudson username ?

Figure 10.36. Configurer une release Maven en utilisant le plugin M2Release

Lorsque ceci est configuré, vous pouvez déclencher manuellement une release Maven en utilisant une nouvelle option de menu appelée “Perform Maven Release” (voir Figure 10.37, “L'option de menu Perform Maven Release”).

Jenkins

Jenkins » [game-of-life](#)

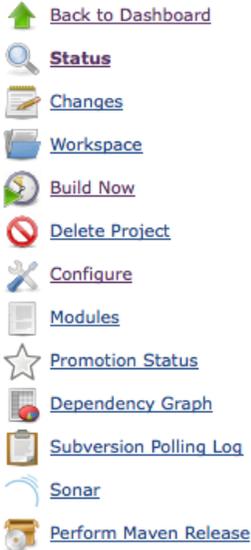


Figure 10.37. L'option de menu Perform Maven Release

Cela déclenchera une tâche de build spéciale utilisant les goals que vous avez fournis dans la configuration du plugin (voir Figure 10.38, “Effectuer une release Maven dans Jenkins”). Jenkins vous offre le choix d'utiliser soit les numéros de version par défaut fournis par Maven (par exemple, la version 1.0.1-SNAPSHOT sera livrée avec la version 1.0.1, et le numéro de version en développement sera positionnée à 1.0.2-SNAPSHOT), soit de fournir vos propres numéros de version personnalisés. Si, par exemple, vous voulez livrer une version majeure vous pourriez décider de spécifier manuellement 1.1.0 comme numéro de version et 1.1.1-SNAPSHOT comme prochain numéro de version de développement.

Si vous avez un projet multimodule Maven, vous pouvez choisir une configuration de numéro de version unique pour tous les modules, ou de fournir une mise à jour de numéro de version différente pour chaque module. Notez que ce n'est généralement pas une pratique recommandée que de fournir des numéros de version différents pour différents modules dans un projet multimodule.

Perform Maven Release

Versioning mode

Maven will decide the version

Specify version(s)

Specify one version for all modules

Release Version

Development version

Append Hudson Build Number

Specify SCM login/password

Specify custom SCM comment prefix

Figure 10.38. Effectuer une release Maven dans Jenkins

En fonction de votre configuration SCM, vous pourriez aussi avoir besoin de fournir un nom d'utilisateur et un mot de passe valide pour permettre à Maven de créer les tags dans votre dépôt de code source.

L'édition professionnelle du Dépôt d'Entreprise Nexus fournit une fonctionnalité appelée Staging Repositories, qui permet de déployer des artefacts dans un espace spécial de staging afin de faire davantage de tests avant de les livrer officiellement. Si vous utilisez cette fonctionnalité, vous devez paramétrer plus finement votre configuration de serveur de build pour de meilleurs résultats.

Nexus Professional travaille en créant un nouvel espace de staging pour chaque adresse IP unique, utilisateur de déploiement et User-Agent HTTP. Une machine de build Jenkins donnée aura toujours la même adresse IP et le même utilisateur. Toutefois, vous voudrez typiquement avoir un espace de staging séparé pour chaque build. L'astuce est alors de configurer Maven pour qu'il utilise une chaîne de User-Agent HTTP unique pour le processus de déploiement. Vous pouvez le faire en configurant le fichier `settings.xml` sur votre serveur de build afin qu'il contienne quelque chose dans le genre des lignes suivantes (l'ID doit correspondre à l'ID du dépôt de release de la section deployment de votre projet) :

```
<server>
  <id>nexus</id>
  <username>my_login</username>
  <password>my_password</password>
  <configuration>
    <httpHeaders>
      <property>
        <name>User-Agent</name>
        <value>Maven m2Release (java:25.66-b01 ${env.BUILD_TAG }</value>
      </property>
    </httpHeaders>
  </configuration>
</server>
```

10.7.2. Copier des artefacts

Pendant un processus de build impliquant plusieurs tâches de build, comme celle illustrée dans Figure 10.33, “Un graphe de dépendance de tâche de build plus compliqué”, il peut parfois être utile

de réutiliser des artefacts produits par un build dans une tâche de build ultérieure. Par exemple, vous pourriez vouloir exécuter une série de tests web en parallèles sur des machines séparées, en utilisant des serveurs d'application locaux pour améliorer les performances. Dans ce cas, il est normal de récupérer le binaire exact qui a été produit dans le build précédent, plutôt que de le reconstruire chaque fois ou, si vous utilisez Maven, de reposer sur un build SNAPSHOT déployé dans le dépôt d'entreprise. En effet, ces deux approches pourraient vous faire courir le risque de résultats de build incohérent : si vous utilisez une SNAPSHOT d'un dépôt d'entreprise, par exemple, vous utiliserez le dernier build SNAPSHOT, qui pourrait ne pas nécessairement être celui construit dans la tâche de build amont.

Le plugin Copy Artifact vous permet de copier des artefacts d'un build amont et de les réutiliser dans votre build courant. Une fois que vous avez installé ce plugin et redémarré Jenkins, vous pourrez ajouter une nouvelle étape de build appelée “Copier des artefacts d'un autre projet” à vos tâches de build freestyle (voir Figure 10.39, “Ajouter une étape de build “Copier des artefacts d'un autre projet””).

The image shows the configuration page for the 'Copy artifacts from another project' build step in Jenkins. The page is titled 'Build' and contains the following fields and options:

- Project name:** A text input field containing 'game-of-life'.
- Warning:** A yellow warning icon and text: 'Artifacts will be copied from all modules of this Maven project; click the help icon to learn about selecting a particular module.'
- Which build:** A dropdown menu with 'Latest successful build' selected.
- Stable build only:** An unchecked checkbox.
- Artifacts to copy:** A text input field containing '**/*.war'.
- Target directory:** An empty text input field.
- Flatten directories:** A checked checkbox.
- Optional:** An unchecked checkbox.
- Delete:** A button at the bottom right.

Figure 10.39. Ajouter une étape de build “Copier des artefacts d'un autre projet”

Cette nouvelle étape de build vous permet de copier des artefacts d'un projet dans l'espace de travail du projet courant. Vous pouvez spécifier n'importe quel autre projet, bien que ce sera typiquement l'une des tâches de build amont. Et bien sûr vous pouvez spécifier, avec une grande flexibilité et précision, les artefacts exacts que vous souhaitez copier.

Vous devez spécifier où trouver les fichiers que vous voulez dans l'espace de travail de l'autre tâche de build, et où Jenkins doit les mettre dans votre espace de travail courant. Ceci peut être une expression régulière flexible (comme `**/*.war`, pour tout fichier WAR produit par la tâche de build), ou cela peut être beaucoup plus précis (comme `gameoflife-web/target/gameoflife.war`). Notez que par défaut, Jenkins copiera la structure de répertoire en même temps que le fichier que vous récupérez, ainsi si le fichier WAR se trouve dans dans le répertoire `target` du module `gameoflife-web`, Jenkins le placera dans le répertoire `gameoflife-web/target` de votre espace de travail courant. Si cela ne vous convient pas, vous pouvez cocher l'option “Aplatir l'arborescence” pour dire à Jenkins de mettre tous les artefacts à la racine du répertoire que vous spécifiez (ou, par défaut, dans l'espace de travail de votre projet).

Souvent, vous voudrez simplement récupérer des artefacts depuis le build réussi le plus récent. Toutefois, vous voudrez parfois plus de précision. Le champ "Quel build" vous permet de spécifier où chercher des artefacts d'un bon nombre d'autres façons, incluant le dernier build sauvé (builds qui ont été marqués à "toujours conserver"), le dernier build réussi, ou même un numéro spécifique de build.

Si vous avez installé le plugin Build Promotion (voir Section 10.7.3, "Promotions de build"), vous pouvez aussi sélectionner le dernier artefact promu dans un processus de promotion en particulier. Pour faire cela, choisissez "Spécifier par permalien", puis choisissez le processus de promotion de build approprié. C'est un excellent moyen de s'assurer d'un pipeline de build fiable et cohérent. Par exemple, vous pouvez configurer un processus de promotion de build pour déclencher un build qui copie un fichier WAR généré depuis le dernier build promu et le déploie sur un serveur particulier. Ceci vous assure de déployer le bon fichier binaire, même si d'autres builds se sont produits depuis.

Si vous copiez des artefacts d'une tâche de build multimodule Maven, Jenkins copiera, par défaut, tous les artefacts de ce build. Toutefois vous êtes souvent intéressé uniquement par un artefact spécifique (comme l'artefact WAR pour une application web, par exemple).

Ce plugin est particulièrement utile quand vous avez besoin d'exécuter des tests fonctionnels ou de performance sur votre application web. Il est souvent stratégiquement utile de placer ces tests dans un projet séparé, et non comme une partie de votre processus de build principal. Cela facilite l'exécution de ces tests sur différents serveurs ou d'exécuter le sous-ensemble des tests en parallèle, tout en utilisant le même artefact binaire pour déployer et tester.

Par exemple, imaginez que vous avez une tâche de build par défaut appelée `gameoflife` qui génère un fichier WAR, et que vous vouliez déployer ce WAR sur un serveur d'application local et exécuter une série de tests fonctionnels. De plus, vous voulez pouvoir faire cela en parallèle sur plusieurs machines distribuées.

Une façon de faire cela serait de créer un projet Maven dédié pour lancer les tests fonctionnels sur un serveur arbitraire. Ensuite, vous mettriez en place une tâche de build pour exécuter ces tests fonctionnels. La tâche de build utiliserait le plugin Copy Artifact pour récupérer le dernier fichier WAR (ou même le dernier fichier WAR promu, pour plus de précision), et le déploierait sur une instance Tomcat locale en utilisant Cargo. Cette tâche de build pourrait ensuite être configurée en tant que tâche de build ("matrix") configurable, et exécutée en parallèle sur plusieurs machines, éventuellement avec des paramètres de configuration supplémentaires pour filtrer les exécutions de test de chaque build. Chaque exécution de build utiliserait ensuite sa propre copie du fichier WAR original. Un exemple d'une configuration comme celle-ci est illustré dans Figure 10.40, "Exécuter des tests web sur un fichier WAR copié".

Build

Copy artifacts from another project

Project name ?

⚠ Artifacts will be copied from all modules of this Maven project; click the help icon to learn about selecting a particular module.

Which build ?

Stable build only

Artifacts to copy ?

Target directory ?

Flatten directories Optional ?

Delete

Invoke top-level Maven targets ?

Maven Version ?

Goals ▼

Advanced...
Delete

Invoke top-level Maven targets ?

Maven Version ?

Goals ▼

Advanced...
Delete

Figure 10.40. Exécuter des tests web sur un fichier WAR copié

Le plugin Copy Artifact n'est pas restreint à la récupération de fichiers depuis des tâches de build conventionnelles. Vous pouvez aussi copier des artefacts à partir de tâches de build multiconfiguration (voir Section 10.4, “Tâches de build multiconfiguration”). Les artefacts de chaque configuration exécutée seront copiés dans l'espace de travail courant, chacun dans son propre répertoire. Jenkins construira une structure de répertoire en se basant sur les axes utilisés dans le build multiconfiguration. Par exemple, imaginez que nous ayons besoin de produire une version hautement optimisée de notre produit pour un certain nombre de bases de données et de serveurs d'application cibles. Nous pourrions faire cela avec une tâche de build multiconfiguration comme celle illustrée dans Figure 10.41, “Copier à partir d'un build multiconfiguration”.

Project phoenix-multi-config-build

Configuration Matrix

	tomcat	resin	websphere	weblogic
oracle	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
mysql	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
sqlserver	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
db2	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>

Figure 10.41. Copier à partir d'un build multiconfiguration

Le plugin Copy Artifacts peut dupliquer n'importe lequel ou même tous les artefacts produits par cette tâche de build. Si vous spécifiez un build multiconfiguration comme source de vos artefacts, le plugin copiera les artefacts de toutes les configurations dans l'espace de travail de la tâche de build cible, en utilisant une structure de répertoire imbriquée basée sur les axes du build multiconfiguration. Par exemple, si vous définissez le répertoire cible comme `multi-config-artifacts`, Jenkins copiera les artefacts dans un certain nombre de sous-répertoires dans le répertoire cible, chacun avec un nom correspondant à un ensemble particulier de paramètres. Ainsi, en utilisant la tâche de build illustrée dans Figure 10.41, “Copier à partir d'un build multiconfiguration”, le fichier JAR personnalisé pour Tomcat et MySQL serait copié dans le répertoire `$WORKSPACE/multi-config-artifacts/APP_SERVER/tomcat/DATABASE/mysql`.

10.7.3. Promotions de build

Dans le monde de l'Intégration Continue, tous les builds créés ne sont pas égaux. Par exemple, vous pourriez vouloir déployer la dernière version de votre application web sur un serveur de test, mais seulement après avoir réussi un certain nombre de tests fonctionnels automatisés ou de charge. Ou vous pourriez vouloir que les testeurs puissent marquer certains builds comme étant prêts pour un déploiement pour les tests d'acceptation utilisateur, une fois qu'ils ont terminé leurs propres tests.

Le plugin Promoted Builds vous permet d'identifier des builds spécifiques ayant atteint des critères additionnels de qualité, et de déclencher des actions sur ces builds. Par exemple, vous pourriez construire une application web dans une tâche de build, exécuter une série de tests automatisés dans un build ultérieur, puis déployer le fichier WAR généré sur le serveur de tests d'acceptation utilisateur pour effectuer davantage de tests.

Voyons comment cela fonctionne en pratique. Dans le projet illustré ci-dessus, une tâche de build par défaut (`phoenix-default`) exécute des tests unitaires et d'intégration, puis produit un fichier WAR. Ce fichier WAR est ensuite réutilisé pour des tests plus étendus (dans la tâche de build `phoenix-integration-tests`) et ensuite pour une série de tests web automatisés (dans la tâche

de build `phoenix-web-test`). Si le build réussit les tests web automatisés, nous aimerions déployer l'application dans un environnement de tests fonctionnels où elle pourrait être testée par des testeurs humains. Le déploiement dans cet environnement est effectué avec la tâche de build `phoenix-test-deploy`. Une fois que les testeurs ont validé la version, elle peut être promue vers l'environnement de tests d'acceptation utilisateur, et enfin en production. La stratégie complète de promotion est illustrée dans Figure 10.42, "Tâches de build dans le processus de promotion".

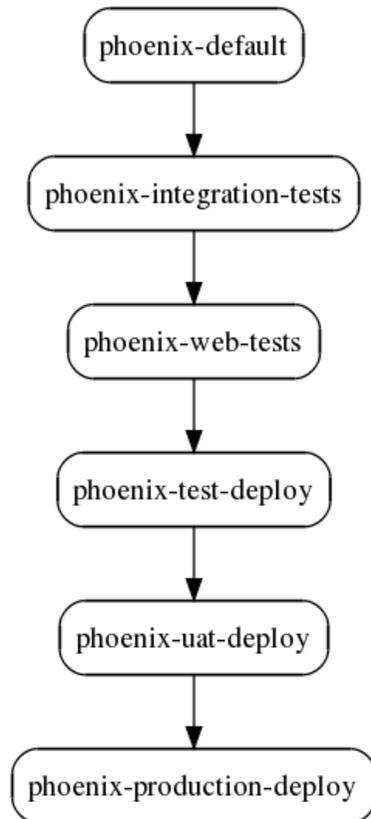


Figure 10.42. Tâches de build dans le processus de promotion

Cette stratégie est facile à implémenter en utilisant le plugin Promoted Builds. Une fois que vous l'avez installé de la façon habituelle, vous trouverez une nouvelle case "Promouvoir builds quand" dans la page de configuration de la tâche. Cette option est utilisée pour configurer les processus de promotion de build. Vous définissez un ou plusieurs processus de promotion de build dans la tâche de build initiale du processus (`phoenix-default` dans cet exemple), comme illustré dans Figure 10.43, "Configurer un processus de promotion de build". Une tâche de build peut être le point de départ de plusieurs processus de promotion de build, certains automatisés, certains manuels. Dans Figure 10.43, "Configurer un processus de promotion de build", par exemple, il y a un processus de promotion de build automatisé appelé `promote-to-test` et un manuel appelé `promote-to-uat`. Les processus de promotion de build automatisés sont déclenchés par les résultats de tâches de build aval. Les processus de promotion

manuels (indiqués en cochant la case ‘Seulement si approuvé manuellement’) peuvent uniquement être déclenchés par une intervention utilisateur.

Promote builds when...

Promotion process

Name

Icon

Criteria

When the following downstream projects build successfully ?

Job names

Trigger even if the build is unstable

Only when manually approved ?

When the following upstream promotions are promoted ?

Actions

Build other projects ?

Projects to build

Figure 10.43. Configurer un processus de promotion de build

Regardons à présent comment configurer le processus de build automatisé promote-to-test.

Vous devez commencer par définir comment le processus de promotion de build sera déclenché. La promotion de build peut être soit automatique, basée sur le résultat d'une tâche de build aval, soit activée manuellement par un utilisateur. Dans Figure 10.43, “Configurer un processus de promotion de build”, la promotion de build pour cette tâche de build sera automatiquement déclenchée lorsque les tests web automatisés (exécuté par la tâche de build **phoenix-web-tests**) auront réussi.

Vous pouvez aussi faire que certaines tâches de build ne puissent être promues que manuellement, comme illustré dans Figure 10.44, “Configurer un processus manuel de promotion de build”. La promotion de build manuelle est utilisée pour les cas où une intervention humaine est requise pour approuver une promotion de build. Le déploiement dans l'environnement de test d'acceptation utilisateur ou de production en sont des exemples courants. Autre exemple, lorsque vous voulez suspendre temporairement les promotions de build pour une courte période, comme à l'approche d'une release.

Les builds manuels, comme leur nom le suggère, nécessite d'être approuvés manuellement avant de pouvoir être exécutés. Si le processus de promotion consiste à déclencher une tâche de build paramétrée, vous pouvez aussi fournir des paramètres que l'approbateur devra entrer lors de l'approbation. Dans certains cas, il peut être utile de désigner certains utilisateurs autorisés à activer la promotion manuelle. Vous pouvez faire cela en spécifiant une liste d'utilisateurs ou de groupes dans la liste d'approbateurs.

Promotion process

Name

Icon

Criteria

When the following downstream projects build successfully ?

Job names

Trigger even if the build is unstable

Only when manually approved ?

Approvers

Approval Parameters

When the following upstream promotions are promoted ?

Actions

?

Projects to build

Figure 10.44. Configurer un processus manuel de promotion de build

Parfois, il est utile de donner un peu de contexte à une personne approuvant une promotion. Quand vous configurez un processus de promotion manuel, vous pouvez aussi spécifier d'autres conditions devant être remplies, en particulier des tâches de build aval (ou amonts) qui doivent avoir été construites avec succès (voir Figure 10.45, “Voir les détails d'une promotion de build”). Celles-ci apparaîtront dans les “Met Qualifications” (pour les tâches de build en succès) et dans les “Unmet Qualifications” (pour les tâches de builds qui ont échoué ou n'ont pas encore été exécutées).

The screenshot shows the Jenkins 'Promotions' page for a build named 'promote-to-test'. The page is divided into several sections:

- Promotions:** A star icon and the name 'promote-to-test'.
- Promotion History:** A table showing a single entry: 'phoenix-default » promotion » promote-to-test #21'.
- Qualification:** Status: 'promoted 2 min 25 sec ago — 41 sec after build'.
- Downstream builds succeeded:** A table showing 'phoenix-web-tests #34'.
- Status:** 'Successfully promoted (log)'. A button 'Re-execute promotion' is visible.
- promote-to-uat:** A star icon and the name 'promote-to-uat'. Below it, the text 'This promotion has not happened.' and a button 'Force promotion'.
- Met Qualification:** Status: 'Downstream builds succeeded'. A table showing 'phoenix-deploy-to-test #28'.
- Unmet Qualification:** No entries.
- Manual Approval:** A section with 'Approvers' set to 'john' and a button 'Approve'.

Figure 10.45. Voir les détails d'une promotion de build

Vous devez ensuite dire à Jenkins ce qu'il doit faire lorsque le build est promu. Cela se fait en ajoutant des actions, tout comme dans une tâche de build freestyle. Ceci rend les promotions de build extrêmement flexible, parce que vous pouvez ajouter pratiquement n'importe quelle action disponible dans une tâche de build freestyle normale, incluant n'importe quelles étapes additionnelles offertes par le plugins installés sur votre instance Jenkins. Les actions courantes incluent l'invocation de script Maven ou Ant, le déploiement d'artefacts dans un dépôt Maven, ou le déclenchement d'un autre build.

Une chose importante à garder à l'esprit ici est que vous ne pouvez pas vous reposer sur des fichiers de l'espace de travail lors de la promotion de votre build. En effet, au moment où vous promouvez votre build, automatiquement ou manuellement, d'autres tâches de build pourraient avoir supprimé ou réécrit les fichiers que vous avez besoin d'utiliser. Pour cette raison, il est imprudent, par exemple, de déployer un fichier WAR directement à partir de l'espace de travail vers un serveur d'application pendant un processus de promotion de build. Une solution plus robuste consiste à déclencher une tâche de build séparée et d'utiliser le plugin Copy Artifacts (voir Section 10.7.2, "Copier des artefacts") pour récupérer précisément le bon fichier. Dans ce cas, vous copierez des artefacts que vous avez demandé à Jenkins de conserver, plutôt que de copier directement des fichiers de l'espace de travail.

Pour que la promotion de build fonctionne correctement, Jenkins doit pouvoir lier précisément les tâches de build aval à celles en amont. La façon la plus précise de faire cela est d'utiliser les fingerprints. Dans

Jenkins, un fingerprint est le somme de contrôle MD5 d'un fichier produit ou utilisé dans une tâche de build. En faisant correspondre les fingerprints, Jenkins est capable d'identifier tous les builds utilisant un fichier particulier.

Dans le contexte de la promotion de build, une stratégie courante est de construire votre application une seule fois, puis d'exécuter des tests sur les fichiers binaires générés dans une série de tâches de builds aval. Cette approche fonctionne bien avec la promotion de build, mais vous devez vous assurer que Jenkins créer un fingerprint des fichiers partagés ou copiés entre tâches de build. Dans l'exemple montré dans Figure 10.43, “Configurer un processus de promotion de build”, notamment, nous avons besoin de faire deux choses (Figure 10.46, “Utiliser fingerprints dans le processus de promotion de build”). Premièrement, nous devons archiver le fichier WAR généré afin qu'il puisse être utilisé dans le projet aval. Deuxièmement, nous devons enregistrer un fingerprint des artefacts archivés. Vous faites cela en cochant l'option “Enregistrer les fingerprints de fichiers pour tracer leur utilisation”, et en spécifiant les fichiers pour lesquels vous voulez créer un fingerprint. Un raccourci utile consiste simplement à créer un fingerprint pour tous les fichiers archivés, puisque ce sont les fichiers qui vont typiquement être récupérés et réutilisés par les tâches de build aval.

The image shows a Jenkins configuration page with several sections:

- Build other projects (with a help icon ?)
- Projects to build:
- Trigger even if the build is unstable
- Archive the artifacts (with a help icon ?)
- Files to archive: (with a help icon ?)
- Aggregate downstream test results (with a help icon ?)
- Record fingerprints of files to track usage (with a help icon ?)
- Files to fingerprint:
- Can use wildcards like 'module/dist/**/*.*.zip'. See the [@includes of Ant fileset](#) for the exact format. The base directory is [the workspace](#).
- Fingerprint all archived artifacts
- Keep the build logs of dependencies (with a help icon ?)

An "Advanced..." button is visible to the right of the "Files to archive" field.

Figure 10.46. Utiliser fingerprints dans le processus de promotion de build

C'est tout ce que vous avez besoin de faire pour configurer le processus initial de build. L'étape suivante consiste à configurer les tests d'intégration exécutés dans la tâche de build `phoenix-integration`. Ici, nous utilisons le plugin Copy Artifact pour récupérer le WAR généré par la tâche de build `phoenix-default` (voir Figure 10.47, “Récupérer le fichier WAR depuis la tâche de build amont”). Comme cette tâche de build est déclenchée immédiatement après la tâche de build `phoenix-default`, on peut simplement récupérer le fichier WAR depuis le dernier build réussi.

Build

Copy artifacts from another project

Project name ?

Which build ?
 Stable build only

Artifacts to copy ?

Target directory ?

Flatten directories Optional ?

Figure 10.47. Récupérer le fichier WAR depuis la tâche de build amont

Cependant, ce n'est pas encore tout à fait tout ce que nous devons faire pour les tests d'intégration. La tâche de build **phoenix-integration** est suivie de la tâche de build **phoenix-web**, qui exécute les tests web automatisés. Pour s'assurer que le même fichier WAR est utilisé à chaque étape du processus de build, nous devons le récupérer dans la tâche de build amont **phoenix-integration**, et non depuis la tâche originale **phoenix-default** (qui pourrait avoir été exécutée à nouveau dans l'intervalle). Nous avons donc aussi besoin d'archiver le fichier WAR dans la tâche de build **phoenix-integration** (voir Figure 10.48, “Archiver le fichier WAR dans la tâche aval”).

Build other projects ?

Projects to build
 Trigger even if the build is unstable

Archive the artifacts ?

Files to archive ?

Figure 10.48. Archiver le fichier WAR dans la tâche aval

Dans la tâche de build **phoenix-web**, nous récupérons ensuite le WAR depuis la tâche **phoenix-integration**, en utilisant une configuration très similaire à celle montrée ci-dessus (voir Figure 10.49, “Récupérer le fichier WAR depuis la tâche d'intégration”).

Build

Copy artifacts from another project

Project name ?

Which build ?
 Stable build only

Artifacts to copy ?

Target directory ?

Flatten directories Optional ?

Figure 10.49. Récupérer le fichier WAR depuis la tâche d'intégration

Pour que le processus de promotion de build fonctionne correctement, il y a une chose importante de plus que nous devons configurer dans la tâche de build **phoenix-web**. Comme nous l'avons évoqué précédemment, Jenkins a besoin de pouvoir être sûr que le fichier WAR utilisé dans ces tests est le même que celui généré dans le build original. Nous faisons cela en activant la création de fingerprint sur le fichier WAR que nous avons récupéré depuis la tâche de build **phoenix-integration** (qui, rappelez-vous, a originellement été construit par la tâche **phoenix-default**). Comme nous avons copié ce fichier WAR dans l'espace de travail, une configuration comme celle de Figure 10.50, "Nous avons besoin de déterminer le fingerprint du fichier WAR que nous utilisons" fonctionnera très bien.

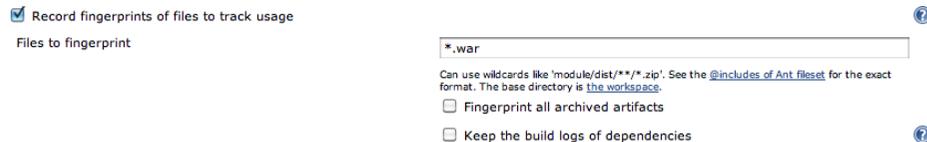


Figure 10.50. Nous avons besoin de déterminer le fingerprint du fichier WAR que nous utilisons

L'étape final consiste à configurer la tâche de build **phoenix-deploy-to-test** pour récupérer le dernier WAR promu (plutôt que le dernier réussi). Pour faire cela, nous utilisons à nouveau le plugin Copy Artifact, mais cette fois nous choisissons l'option "Spécifier par permalien". Ici, Jenkins proposera, entre autres choses, les processus de promotion de build configurés pour la tâche de build à partir de laquelle vous êtes en train de copier. Donc, dans Figure 10.51, "Récupérer le dernier fichier WAR promu", nous récupérerons le dernier fichier WAR promu par la tâche **phoenix-default**, ce qui est précisément ce que nous voulons.

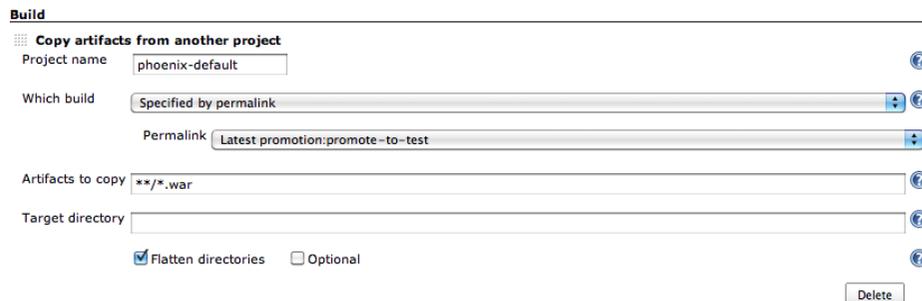


Figure 10.51. Récupérer le dernier fichier WAR promu

Notre processus de promotion est maintenant prêt pour l'action. Quand les tests web automatisés réussiront lors d'un build particulier, la tâche de build originale sera promu et le fichier WAR correspondant déployé dans l'environnement de test. Les builds promus sont indiqués par une étoile dans l'historique de build (voir Figure 10.52, "Les builds promus sont indiqués par une étoile dans l'historique de build"). Par défaut, les étoiles sont jaunes, mais vous pouvez configurer la couleur de l'étoile dans la configuration de la promotion de build.

Figure 10.52. Les builds promus sont indiqués par une étoile dans l'historique de build

Vous pouvez aussi utiliser l'entrée de menu "Etat de Promotion" (ou cliquez sur l'étoile colorée dans l'historique de build) pour voir les détails d'une promotion de build particulière, et même de réexécuter manuellement une promotion (voir Figure 10.45, "Voir les détails d'une promotion de build"). Toute promotion de build peut être déclenchée manuellement, en cliquant sur "Forcer la promotion" (si cette tâche de build n'a jamais été promue) ou "Ré-exécuter la promotion" (si elle l'a été).

10.7.4. Agréger des résultats de tests

Lorsqu'on répartit différents types de tests dans différentes tâches de build, il est facile de perdre la vision globale des résultats de tests de l'ensemble. Ces résultats sont dispersés parmi les diverses tâches de build, sans un endroit central où voir le nombre total de tests exécutés et échoués.

Un bon moyen d'éviter ce problème est d'utiliser la fonctionnalité d'agrégation de résultats de tests de Jenkins. Ceci récupérera tout résultat de test depuis les tâches de build avals, et les agrégera dans la tâche de build amont. Vous pouvez configurer cela dans la tâche de build initiale (amont) en cochant l'option "Agréger les résultat de test avals" (voir Figure 10.53, "Rapport sur l'agrégation des résultats de test").

Figure 10.53. Rapport sur l'agrégation des résultats de test

Les résultats de test agrégés peuvent être consultés dans la page de détails du build (voir Figure 10.54, “Visualisation des résultats de tests agrégés”). Malheureusement, ces résultats de test agrégés n'apparaissent pas dans les résultats de test globaux, mais vous pouvez afficher la liste complète des tests exécutés en cliquant sur le lien Résultats de Test Agrégés sur la page du build particulier.

Jenkins » Phoenix » phoenix-default » #22

[Back to Project](#)

[Status](#)

[Changes](#)

[Console Output](#)

[Configure](#)

[Tag this build](#)

[Promotion Status](#)

[Test Result](#)

[Aggregated Test Result](#)

[See Fingerprints](#)

[Downstream build view](#)

[Previous Build](#)

Build #22 (Mar 4, 2011)

[Build Artifacts](#)

- [gameoflife.war](#)

Revision: 394
No changes.

Started by user [anonymous](#)

[Test Result \(no failures\)](#)

[Aggregated Test Result \(no failures\)](#)

Figure 10.54. Visualisation des résultats de tests agrégés

Pour que cela fonctionne correctement, vous devez vous assurer d'avoir configuré la création de fingerprint pour les fichiers binaires utilisés à chaque étape. Jenkins agrégera seulement les résultats de test aval de builds contenant un artefact avec le même fingerprint.

10.7.5. Pipelines de Build

Le dernier plugin que nous allons regarder dans cette section est le plugin Build Pipeline. Le plugin Build Pipelines emmène l'idée de la promotion de build encore plus loin, et vous aide à concevoir et superviser des pipelines de déploiement. Un pipeline de déploiement est une façon d'orchestrer vos builds au travers d'une série de passages garantissant la qualité, avec des approbations automatisées ou manuelles à chaque étape, culminant avec le déploiement en production.

Le plugin Build Pipeline fournit une autre façon de définir des tâches de build aval. Un pipeline de build, contrairement aux dépendances aval conventionnelles, est considéré comme un processus linéaire, une série de tâches de build exécutées en séquence.

Pour utiliser ce plugin, commencez par configurer les tâches de build aval pour chaque tâche de build dans le pipeline, en utilisant le champ “Construire d'autres projets” comme vous le feriez habituellement. Le plugin Build Pipeline utilise les configurations de build amont ou aval standards, et pour les étapes automatiques c'est tout ce que vous avez à faire. Toutefois, le plugin Build Pipeline supporte aussi les étapes de build manuelles, où un utilisateur doit manuellement approuver l'étape suivante. Pour les étapes manuelles, vous devez aussi configurer les **Post-build Actions** de votre tâche de build amont : cochez simplement la case “Build Pipeline Plugin -> Spécifier Projet Aval”, sélectionnez l'étape suivante dans votre projet, et cochez l'option “Require manual build executor” (voir Figure 10.55, “Configurer une étape manuelle dans le pipeline de build”).



Figure 10.55. Configurer une étape manuelle dans le pipeline de build

Une fois que vous avez configuré votre processus de build à votre convenance, vous pouvez configurer la vue build pipeline. Vous pouvez créer cette vue comme n'importe quelle autre (voir Figure 10.56, “Créer une vue Build Pipeline”).

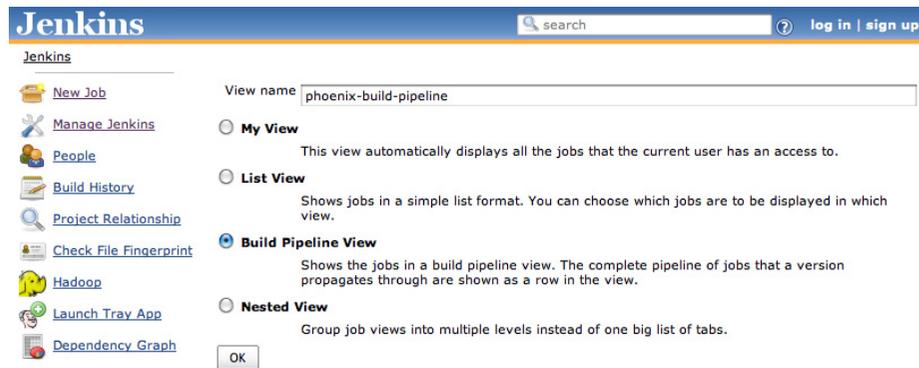


Figure 10.56. Créer une vue Build Pipeline

Il y a une astuce à connaître lors de la configuration de la vue, cependant. Au moment de l'écriture de ces lignes, il n'y a pas d'option de menu ou de bouton vous permettant de configurer la vue directement. En fait, vous devez entrer l'URL manuellement. Heureusement, ce n'est pas difficile : ajoutez juste /configure à la fin de l'URL lorsque vous affichez cette vue. par exemple, si vous avez appelé cette vue “phoenix-build-pipeline”, comme montré ici, l'URL pour configurer cette vue serait `http://my_jenkins_server/view/phoenix-build-pipeline`. (voir Figure 10.57, “Configurer une vue Build Pipeline”).

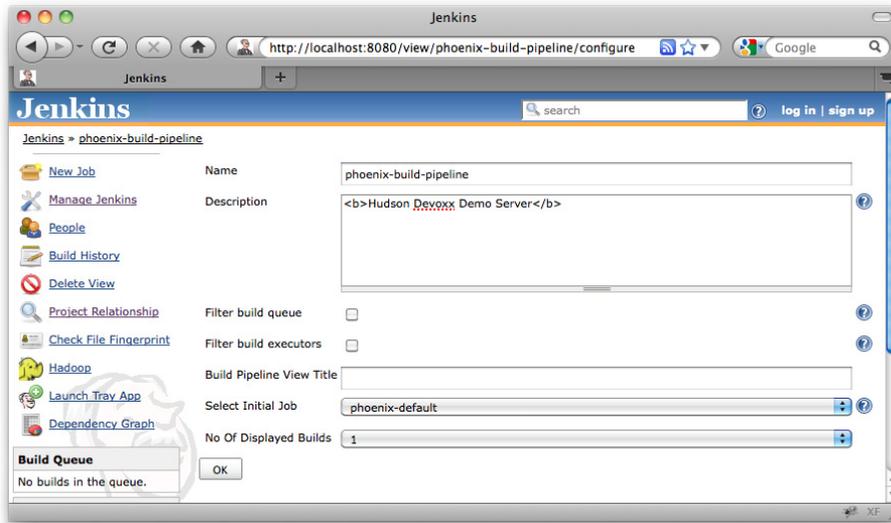


Figure 10.57. Configurer une vue Build Pipeline

La chose la plus importante à configurer dans cet écran est la tâche initiale. Ceci marque le point d'entrée de votre pipeline de build. Vous pouvez définir de multiples vues de pipeline de build, chacune avec une tâche initiale différente. Vous pouvez aussi configurer le nombre maximum de séquences de build à faire apparaître à la fois sur l'écran.

Une fois que vous avez configuré le point de départ, vous pouvez retourner à la vue pour voir l'état courant de votre pipeline de build. Jenkins affiche les tâches de build successives horizontalement, en utilisant une couleur pour indiquer le résultat de chaque build (Figure 10.58, "Un Pipeline de Build en action"). Il y a une colonne pour chaque tâche de build dans le pipeline. Dès lors que la tâche de build initiale démarre, une nouvelle ligne apparaît sur cette page. Alors que le build progresse parmi les tâches de build successives, Jenkins ajoute une boîte colorée dans les colonnes successives, indiquant le résultat de chaque étape. Vous pouvez cliquer sur la boîte pour descendre dans un résultat de build particulier pour plus de détails. Enfin, si une exécution manuelle est requise, un bouton sera affiché afin que l'utilisateur puisse déclencher la tâche.



Figure 10.58. Un Pipeline de Build en action

Ce plugin est encore relativement nouveau, et ne s'intègre pas avec les autres plugins que nous avons vus ici. En particulier, il est vraiment conçu pour un pipeline de build linéaire, et ne s'en sort pas très bien avec des branches ou des tâches de build parallèles. Néanmoins, il donne une excellente vision globale d'un pipeline de build.

10.8. Conclusion

Les tâches de build d'Intégration Continue sont beaucoup plus que de simples exécutions planifiées de scripts de build. Dans ce chapitre, nous avons revu un certain nombre d'outils et de techniques vous permettant d'aller au delà de vos tâches de build typiques, en les combinant afin qu'elles travaillent ensemble comme partie d'un processus plus large. Nous avons à présent vu comment les tâches de build paramétrées et multiconfiguration ajoutent un élément de flexibilité aux tâches de build ordinaires en vous permettant d'exécuter la même tâche de build avec différents ensembles de paramètres. D'autres outils aident à coordonner et à orchestrer des groupes de tâches de build reliées. Les plugins Joins et Locks and Latches vous aident à coordonner des tâches de build s'exécutant en parallèle. Et les plugins Build Promotions et Build Pipelines, avec l'aide du plugin Copy Artifacts, rendent relativement facile la conception et la configuration de stratégies de promotion de build complexes pour vos projets.

Chapter 11. Builds distribués

11.1. Introduction

L'une des fonctionnalités les plus puissantes de Jenkins est sans aucun doute sa capacité à répartir les tâches de build sur un grand nombre de machines. Il est assez simple de configurer une ferme de serveurs de build, soit pour répartir la charge sur de multiples machines, soit pour exécuter des tâches de build dans différents environnements. C'est une stratégie très efficace qui peut potentiellement accroître de façon considérable la capacité de votre infrastructure d'IC.

Les builds distribués sont généralement utilisés soit pour absorber une charge additionnelle, par exemple pour absorber les pics d'activité dans les builds en ajoutant dynamiquement des machines supplémentaires selon les besoins, soit pour exécuter des tâches de build spécialisées sur des systèmes d'exploitation ou des environnements spécifiques. Par exemple, vous pourriez avoir besoin d'exécuter une tâche de build spéciale sur une machine ou un système d'exploitation particulier. Si vous avez besoin d'exécuter des tests web en utilisant Internet Explorer, vous devrez utiliser une machine Windows. Ou alors une de vos tâches de build pourrait être particulièrement consommatrice en ressources, et nécessite d'être exécutée sur une machine dédiée afin de ne pas pénaliser d'autres tâches de build.

La demande pour des serveurs de build peut aussi fluctuer dans le temps. Si vous travaillez avec des cycles de release de produit, vous pouvez avoir besoin de beaucoup plus de tâches de build en fin de cycle, par exemple, lorsque des tests fonctionnels ou de régression plus complets deviennent plus fréquents.

Dans ce chapitre, nous verrons comment configurer et gérer une ferme de serveurs de build en utilisant Jenkins.

11.2. L'Architecture de build distribuée de Jenkins

Jenkins utilise une architecture maître/esclave pour gérer les builds distribués. Votre serveur Jenkins principal (celui que nous avons utilisé jusqu'à présent) est le maître. En un mot, le rôle du maître est de gérer l'ordonnancement des tâches de build, de répartir les builds sur les esclaves pour leur exécution réelle, surveiller les esclaves (en les mettant éventuellement hors-ligne si nécessaire) et enfin enregistrer et présenter les résultats de build. Même dans une architecture distribuée, une instance maître de Jenkins peut aussi exécuter des tâches de build directement.

Le rôle des esclaves est de faire ce qu'on leur dit, ce qui inclut l'exécution de tâches de build envoyées par le maître. Vous pouvez configurer un projet pour qu'il s'exécute toujours sur un esclave particulier, sur un type particulier de machine, ou simplement laisser Jenkins sélectionner le prochain esclave disponible.

Un esclave est un petit exécutable Java qui fonctionne sur une machine distante et se met en écoute de requêtes de la part de l'instance maître Jenkins. Les esclaves peuvent (et c'est généralement le cas) s'exécuter sur différents systèmes d'exploitation. L'instance esclave peut être démarrée de façons

diverses, en fonction du système d'exploitation et de l'architecture réseau. Une fois que l'instance esclave est en marche, elle communique avec l'instance maître au travers d'une connexion TCP/IP. Nous regarderons différentes configurations dans le reste de ce chapitre.

11.3. Stratégies Maître/Esclave dans Jenkins

Il existe différentes façons de configurer une ferme de build distribuée avec Jenkins, en fonction de votre système d'exploitation et de votre architecture réseau. Dans tous les cas, le fait qu'une tâche de build s'exécute sur un esclave, et comment cet esclave s'exécute, est transparent pour l'utilisateur final : les résultats du build et les artefacts finiront toujours sur le serveur maître.

Créer un nouveau noeud esclave Jenkins est un processus simple. Premièrement, rendez vous dans l'écran Administrer Jenkins et cliquez sur Gérer les noeuds. Cet écran affiche la liste des agents esclaves (aussi connus en tant que “Noeuds” en termes plus politiquement corrects), comme montré dans Figure 11.1, “Gérer les noeuds esclaves”. A partir de là, vous pouvez configurer de nouveaux noeuds en cliquant sur Nouveau noeud. Vous pouvez aussi configurer quelques-uns des paramètres liés à votre installation de build distribuée (see Section 11.5, “Surveillance des noeuds”).

S	Name	Free Disk Space	Free Temp Space	Architecture	Clock Difference	Response Time	Free Swap Space
	master	185GB	27GB	Linux (amd64)	In sync	0ms	15217MB

#	Status
1	Idle
2	Idle

Figure 11.1. Gérer les noeuds esclaves

Il y a plusieurs stratégies différentes lorsqu'il s'agit de gérer les noeuds esclaves Jenkins, en fonction de vos systèmes d'exploitation cibles et d'autres considérations architecturales. Ces stratégies affectent la façon dont vous configurez vos noeuds esclaves, nous devons donc les considérer séparément. Dans les sections suivantes, nous regarderons les façons les plus fréquemment utilisées pour installer et configurer des esclaves Jenkins :

- Le maître démarre l'agent esclave via SSH
- Démarrage manuel de l'agent esclave en utilisant Java Web Start
- Installation de l'agent esclave en tant que service Windows
- Démarrage de l'agent esclave directement depuis la ligne de commande sur la machine esclave

Chacune de ces stratégies possède ses utilisations, ses avantages et ses inconvénients. Regardons chacune d'entre elles.

11.3.1. Le maître démarre l'agent esclave en utilisant SSH

Si vous travaillez dans un environnement Unix, la façon la plus pratique de démarrer un esclave Jenkins est sans aucun doute d'utiliser SSH. Jenkins possède son propre client SSH intégré, et presque tous les environnements Unix supportent SSH (habituellement `sshd`) de base.

Pour créer un esclave de type Unix, cliquez sur le bouton Nouveau noeud comme nous l'avons mentionné ci-dessus. Cela vous demande d'entrer le nom de votre esclave, et son type (voir Figure 11.2, “Créer un nouveau noeud esclave”). Au moment de l'écriture de ces lignes, seuls les “esclaves passifs” sont supportés en standard; ces esclaves répondent simplement aux requêtes de build en provenance du noeud maître. C'est la façon la plus commune de mettre en place une architecture de build distribuée, et c'est la seule option disponible dans une installation par défaut.

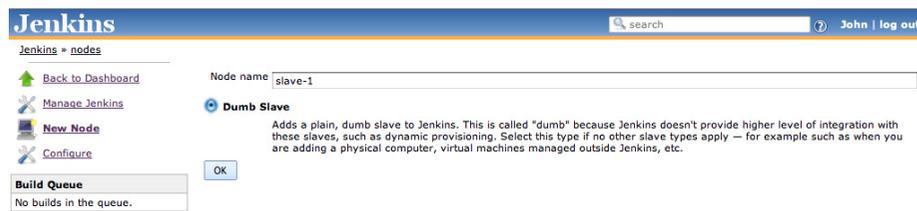


Figure 11.2. Créer un nouveau noeud esclave

Dans cet écran, vous devez simplement fournir un nom pour votre esclave. Lorsque vous cliquez sur OK, Jenkins vous permet de fournir plus de détails sur vos machines esclaves (voir Figure 11.3, “Créer un noeud esclave Unix”).

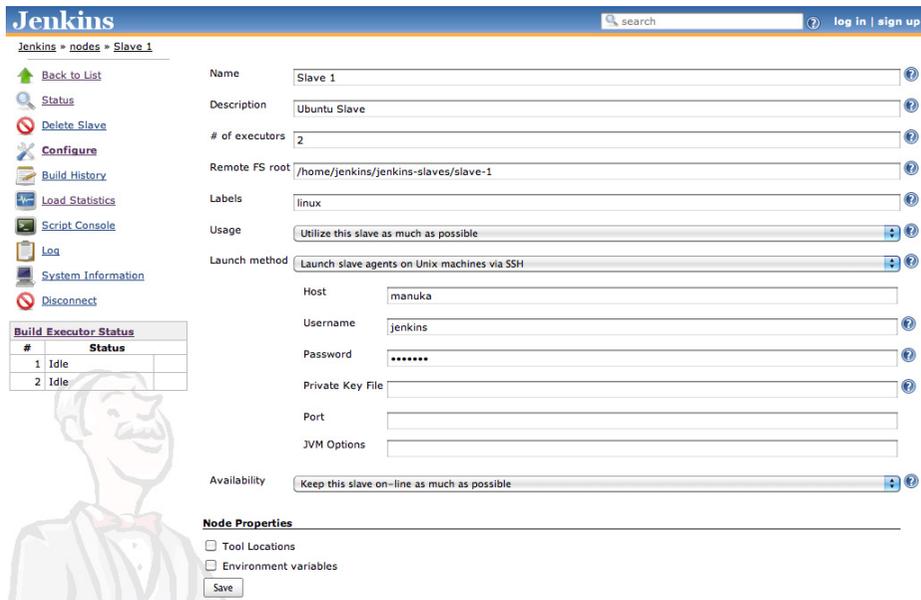


Figure 11.3. Créer un noeud esclave Unix

Ce nom est simplement un moyen unique pour identifier votre machine esclave. Ce peut être n'importe quoi, mais il pourrait être utile que ce nom vous rappelle la machine physique sur laquelle cela fonctionne. Il est aussi utile que ce nom soit compatible avec le système de fichiers et un format URL. Les espaces sont autorisés, mais cela vous facilitera la vie de les éviter. Ainsi, “Slave-1” est meilleur que “Slave 1”.

La description est aussi purement destinée à la lecture humaine, et peut être utilisée pour indiquer pourquoi utiliser cet esclave plutôt qu'un autre.

Comme sur l'écran principal de configuration Jenkins, le nombre d'exécuteurs vous permet de définir le nombre de tâches de build concurrentes que ce noeud peut exécuter.

Tout noeud esclave Jenkins nécessite aussi un emplacement qu'il puisse utiliser comme racine, ou, plus précisément un répertoire dédié sur la machine esclave que l'agent esclave puisse utiliser pour exécuter des tâches de build. Vous définissez ce répertoire dans le champ racine du disque distant. Vous devez fournir un chemin local, spécifique à l'OS, tel que `/var/jenkins` pour une machine Unix ou `C:\jenkins` sur Windows. Rien d'essentiel n'est stocké dans ce répertoire — tout ce qui est important est renvoyé à la machine maître une fois que le build est effectué. Vous n'avez donc généralement pas besoin de vous inquiéter de sauvegarder ces répertoires comme c'est le cas avec ceux du maître.

Les libellés sont un concept particulièrement utile quand votre architecture distribuée commence à grossir. Vous définissez des libellés, des tags, pour chaque noeud de build, et configurez ensuite une tâche de build afin qu'elle s'exécute avec un libellé particulier. Les libellés peuvent avoir trait au système

d'exploitation (unix, windows, macosx, etc.), aux environnements (staging, recette, développement, etc.) ou n'importe quel critère que vous trouveriez utile. Par exemple, vous pouvez configurer vos tests automatisés WebDriver/Selenium pour qu'ils s'exécutent avec Internet Explorer, mais seulement sur des noeuds esclaves avec le libellé “windows”.

Le champ Utilisation vous permet de configurer l'intensité avec laquelle Jenkins utilisera cet esclave. Vous avez le choix parmi trois options : utiliser cet esclave autant que possible, réserver pour les tâches de build dédiées, ou le mettre en ligne quand c'est nécessaire.

La première option “Utiliser cet esclave autant que possible”, indique à Jenkins d'utiliser librement cet esclave dès qu'il devient disponible, pour toute tâche qu'il peut exécuter. C'est de loin l'option la plus utilisée, et généralement celle que vous voulez.

Quelques fois, cependant, la seconde option peut s'avérer utile. Dans la configuration du projet, vous pouvez lier une tâche de build à un noeud spécifique — c'est utile quand une tâche particulière, comme un déploiement automatisé ou une suite de tests de performance, nécessite d'être exécutée sur une machine spécifique. Dans ce cas, l'option “Réserver cette machine pour les tâches associées uniquement” peut avoir du sens. Vous pouvez aller encore plus loin en positionnant le nombre maximum d'Exécuteurs à 1. Dans ce cas, non seulement cet esclave sera réservé pour un type particulier de tâche, mais il sera uniquement capable d'exécuter une seule de ces tâches de build à tout instant. C'est une configuration très utile pour les tests de performance ou de charge, où vous avez besoin de réserver la machine pour qu'elle exécute ses tests sans interférence.

La troisième option est “Mettre cet esclave en ligne lors de demande et hors-ligne sinon” (voir Figure 11.4, “Mettre un esclave hors-ligne lorsqu'il est inactif”). Comme le nom l'indique, cette option indique à Jenkins de mettre cet esclave en ligne lorsque la demande est élevée et de le mettre hors-ligne lorsque la demande faiblit. Ceci permet de garder quelques esclaves de build pour les périodes d'utilisation importante, sans avoir à maintenir dessus un agent esclave fonctionnant en permanence. Quand vous choisissez cette option, vous devez aussi fournir quelques détails supplémentaires. Le “Délai de la demande” indique combien de minutes les tâches doivent avoir attendu dans la file d'attente avant que cet esclave ne soit mis en ligne. Le champ Délai d'inactivité indique combien de temps l'esclave doit avoir été inactif avant que Jenkins ne le mette hors-ligne.

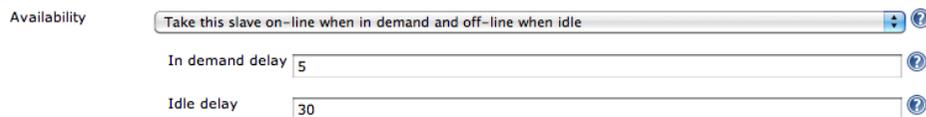


Figure 11.4. Mettre un esclave hors-ligne lorsqu'il est inactif

La méthode de lancement décide de comment Jenkins lancera le noeud, comme nous l'avons mentionné précédemment. Pour la configuration dont nous parlons ici, vous choisiriez “Lancer les agents esclaves sur machines Unix via SSH”. Le bouton Avancé vous permet d'entrer des détails additionnels dont Jenkins a besoin pour se connecter à la machine esclave Unix : un nom d'hôte, un login et mot de passe

et un numéro de port. Vous pouvez aussi fournir un chemin vers un fichier de clé privée SSH sur la machine maître (e.g., `id_dsa` ou `id_rsa`) à utiliser pour une authentification “sans mot de passe” par clés Publique/Privée.

Vous pouvez aussi configurer quand Jenkins démarre ou arrête l'esclave. Par défaut, Jenkins gardera simplement l'esclave en fonctionnement et l'utilisera chaque fois qu'il en aura besoin (option “Garder cet esclave en ligne autant que possible”). Si Jenkins remarque que l'esclave est déconnecté (par exemple à cause d'un redémarrage serveur), il essaiera de le redémarrer s'il le peut. Sinon, Jenkins peut être plus conservateur avec vos ressources systèmes, et mettre l'esclave hors-ligne lorsqu'il n'en a pas besoin. Pour faire cela, choisissez simplement l'option “Mettre cet esclave en ligne si nécessaire et hors-ligne en cas d'inactivité”. C'est utile si vous avez régulièrement des pics et accalmies de l'activité de build, car un esclave peut être mis hors-ligne pour conserver les ressources systèmes pour d'autres tâches, et remis en ligne lorsque c'est nécessaire.

Jenkins a aussi besoin de savoir où il peut trouver les outils de build dont il a besoin pour vos tâches de build sur les machines esclaves. Ceci inclut aussi bien les JDKs que les outils de build comme Maven, Ant, et Gradle. Si vous avez configuré vos outils de build pour être automatiquement installés, vous n'aurez généralement pas de configuration supplémentaire à effectuer pour vos machines esclaves; Jenkins téléchargera et installera les outils au besoin. D'un autre côté, si vos outils de build sont installés localement sur la machine esclave, vous aurez besoin d'indiquer à Jenkins où il peut les trouver. Ceci se fait en cochant la case Emplacement des outils, et en fournissant les chemins locaux pour chaque outil nécessaire à vos tâches de build (voir Figure 11.5, “Configurer l'emplacement des outils”).

Node Properties

Tool Locations

List of tool locations

Name	(Maven) Maven 3.0	?
Home	/opt/maven/apache-maven-2.2.1	Delete
Name	(JDK) Java 1.6	?
Home	/opt/maven/apache-maven-3.0-beta-2	Delete
Name	(JDK) Java 1.5	?
Home	/usr/lib/jvm/java-6-sun	Delete
Name	(JDK) 1.4.2	?
Home	/usr/lib/jvm/java-6-sun	Delete
Name	(JDK) 1.4.2	?
Home	/usr/lib/jvm/java-6-sun	Delete

Environment variables

Figure 11.5. Configurer l'emplacement des outils

Vous pouvez aussi spécifier des variables d'environnement. Celles-ci seront passées à vos tâches de build, et ce peut être un moyen de permettre à vos tâches de se comporter différemment en fonction de l'endroit où elles s'exécutent.

Une fois que vous avez fait cela, votre nouveau noeud esclave apparaîtra dans la liste des ordinateurs sur la page des Noeuds Jenkins (voir Figure 11.6, “Votre nouveau noeud esclave en action”).



Figure 11.6. Votre nouveau noeud esclave en action

11.3.2. Démarrer l'agent esclave manuellement via Java Web Start

Une autre option est de démarrer l'agent esclave depuis la machine esclave elle-même en utilisant Java Web Start (JNLP). Cette approche est utile si le serveur ne peut pas se connecter à l'esclave, par exemple si la machine esclave s'exécute de l'autre côté d'un firewall. Cela fonctionne quel que soit le système d'exploitation de votre esclave, toutefois c'est l'option la plus souvent utilisée pour les esclaves Windows. Cela présente quelques inconvénients majeurs : le noeud esclave ne peut pas être démarré, ou redémarré, automatiquement par Jenkins. Ainsi, si l'esclave tombe, l'instance maître ne peut pas le redémarrer.

Quand vous faites cela sur une machine Windows, vous devez démarrer l'esclave Jenkins manuellement au moins une fois. Ceci implique d'ouvrir un navigateur sur la machine, ouvrir la page du noeud esclave sur le maître Jenkins et de lancer l'esclave en utilisant l'icône JNLP bien visible. Une fois que vous avez lancé l'esclave, vous pouvez l'installer comme un service Windows.

Il y a aussi des moments où vous avez besoin de faire cela depuis la ligne de commande, dans un environnement Unix. Vous pourriez avoir besoin de ça à cause d'un firewall ou d'autres problèmes réseau, ou parce que SSH n'est pas disponible dans votre environnement.

Détaillons à présent les deux processus.

La première chose que vous devez faire dans tous les cas est de créer un nouvel esclave. Comme pour tout noeud esclave, vous faites cela en cliquant sur l'entrée Nouveau noeud dans l'écran Noeuds. Lors de la saisie des détails concernant votre noeud esclave, assurez-vous de choisir “Lancer les agents esclave via JNLP” dans le champ Méthode de lancement (voir Figure 11.7, “Créer un noeud esclave pour JNLP”). Rappelez-vous aussi que c'est un noeud esclave Windows, la racine du système de fichiers distant doit

être un chemin Windows (comme C:\jenkins-slave). Ce répertoire n'a pas à exister : Jenkins le créera automatiquement s'il manque.

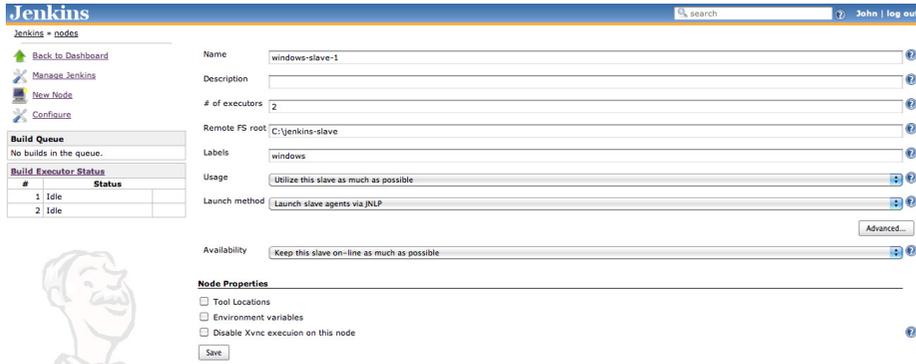


Figure 11.7. Créer un noeud esclave pour JNLP

Une fois que vous avez sauvé cette configuration, connectez-vous, ensuite, sur la machine esclave et ouvrez l'écran du noeud esclave dans un navigateur, comme montré sur Figure 11.8, “Lancer un esclave via Java Web Start”. Vous verrez un large bouton orange Lancer — si vous cliquez sur ce bouton, vous devriez être capable de lancer un agent esclave directement depuis votre navigateur.



Figure 11.8. Lancer un esclave via Java Web Start

Si tout va bien, ceci ouvrira une petite fenêtre indiquant que votre esclave est à présent en fonctionnement (voir Figure 11.9, “L'agent esclave Jenkins en action”).



Figure 11.9. L'agent esclave Jenkins en action

Les navigateurs sont inconstants, toutefois, et Java Web Start n'est pas toujours simple à utiliser. Cette approche fonctionne habituellement avec Firefox, bien que vous deviez auparavant avoir installé le JRE Java pour que Firefox comprenne Java. Utiliser JNLP avec Internet Explorer requiert un ensemble (non négligeable) de bricolages pour associer les fichiers *.jnlp avec l'exécutable Java Web Start, un fichier appelé `javaws`, que vous trouverez dans le répertoire `bin` de Java. Il est en fait probablement plus simple de le lancer depuis la ligne de commande comme discuté ci-dessous.

Une approche plus fiable, quoique bas-niveau, est de démarrer l'esclave depuis la ligne de commande. Pour faire ça, invoquez simplement l'exécutable `javaws` depuis une fenêtre de commande comme suit :

```
C:> javaws http://build.myorg.com/jenkins/computer/windows-slave-1/slave-agent.jnlp
```

La commande exacte que vous devez exécuter, notamment avec l'URL correcte, est idéalement affichée dans la fenêtre du noeud esclave Jenkins juste en dessous du bouton de lancement JNLP (voir Figure 11.8, "Lancer un esclave via Java Web Start").

Si la sécurité est activée sur votre serveur Jenkins, Jenkins communiquera avec l'esclave sur un port spécifique non standard. Si pour une raison quelconque ce port est inaccessible, le noeud esclave échouera au lancement et affichera un message d'erreur similaire à celui montré dans Figure 11.10, "L'esclave Jenkins échouant à la connexion au maître".

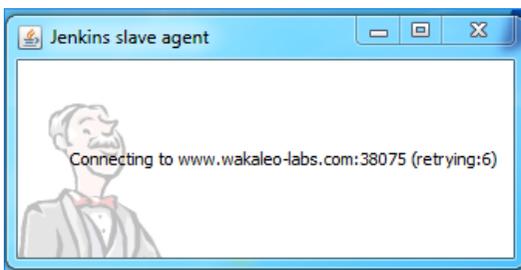


Figure 11.10. L'esclave Jenkins échouant à la connexion au maître

Ceci est habituellement le signe qu'un firewall bloque un port. Par défaut, Jenkins choisit aléatoirement un port pour la communication TCP avec ses esclaves. Cependant si vous devez avoir un port spécifique que votre firewall autorise, vous pouvez forcer Jenkins à utiliser un port fixe dans l'écran de configuration

système en sélectionnant l'option Fixe dans “Port TCP pour les agents esclaves JNLP”, comme montré dans Figure 11.11, “Configurer le port de l'esclave Jenkins”.

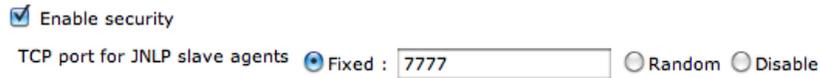


Figure 11.11. Configurer le port de l'esclave Jenkins

11.3.3. Installer un esclave Jenkins en tant que service Windows

Une fois que vous avez démarré votre esclave sur votre machine Windows, vous pouvez vous épargner la peine d'avoir à le redémarrer manuellement chaque fois que votre machine redémarre en l'installant comme un service Windows. Pour faire cela, sélectionnez l'option de menu “Installer comme Service Windows” dans le menu Fichier de la fenêtre de l'agent esclave (voir Figure 11.12, “Installer l'esclave Jenkins en tant que service Windows”).

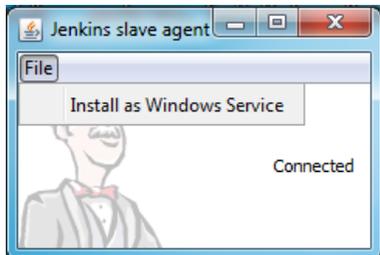


Figure 11.12. Installer l'esclave Jenkins en tant que service Windows

Une fois que c'est fait, votre noeud esclave Jenkins démarrera automatiquement chaque fois que la machine démarre, et peut être administré comme n'importe quel autre service Windows (voir Figure 11.13, “Gérer le service Windows Jenkins”).

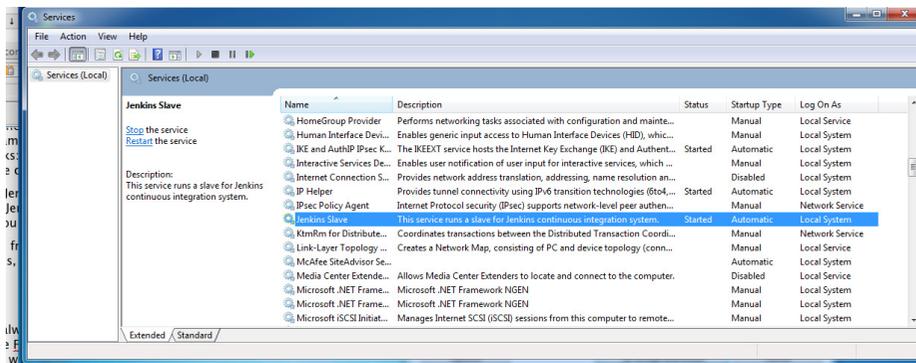


Figure 11.13. Gérer le service Windows Jenkins

11.3.4. Démarrer le noeud esclave en mode Headless

Vous pouvez aussi démarrer un agent esclave en mode headless, directement depuis la ligne de commande. C'est utile si vous n'avez pas d'interface utilisateur disponible, par exemple si vous démarrez un noeud esclave JNLP sur une machine Unix. Si vous travaillez avec des machines Unix, il est généralement plus facile et plus flexible d'utiliser simplement une connexion SSH, mais il y a parfois des contraintes de réseau ou d'architecture qui vous empêchent d'utiliser SSH. Dans ce genre de cas, il est encore possible d'exécuter un noeud esclave depuis la ligne de commande.

Pour démarrer le noeud esclave de cette façon, vous devez utiliser le fichier `slave.jar` de Jenkins. Vous pouvez le trouver dans `JENKINS_HOME/war/WEB-INF/slave.jar`. Une fois ce fichier localisé et copié sur la machine esclave Windows, vous pouvez l'exécuter comme suit :

```
java -jar slave.jar \  
-jnlpUrl http://build.myorg.com/jenkins/computer/windows-slave-1/slave-agent.jnlp
```

Et si votre serveur Jenkins nécessite une authentification, passez simplement l'option `-auth username:password` :

```
java -jar slave.jar \  
-jnlpUrl http://build.myorg.com/jenkins/computer/windows-slave-1/slave-agent.jnlp  
-auth scott:tiger
```

Une fois que vous avez démarré l'agent esclave, assurez de l'installer en tant que service Windows, comme indiqué dans la section précédente.

11.3.5. Démarrer un esclave Windows en tant que service distant

Jenkins peut aussi gérer un esclave Windows distant comme un service Windows, en utilisant le service Windows Management Instrumentation (WMI) qui est installé par défaut sur Windows 2000 ou supérieur (voir Figure 11.14, “Permettre à Jenkins de contrôler un esclave Windows comme un service Windows”). Quand vous choisissez cette option, vous avez seulement besoin de fournir un nom d'utilisateur et un mot de passe Windows. Le nom du noeud doit être le nom de machine de la machine esclave.

Ceci est certainement pratique, parce que cela ne requiert pas de se connecter à la machine Windows pour la configurer. Toutefois, cette méthode possède quelques limitations — en particulier, vous ne pouvez pas exécuter d'applications nécessitant une interface graphique, vous ne pouvez donc pas mettre en place un esclave de cette façon pour faire du test Web, par exemple. En pratique, cela peut se révéler un peu compliqué à paramétrer, parce que vous pourriez avoir besoin de configurer le firewall Windows pour ouvrir les ports et services appropriés. Si vous rencontrez des problèmes, assurez-vous que votre configuration réseau autorise les connexions TCP aux ports 135, 139, et 445, ainsi que les connexions UDP aux ports 137 et 138 (voir <https://wiki.jenkins-ci.org/display/JENKINS/Windows+slaves+fail+to+start+via+DCOM> pour plus de détails).

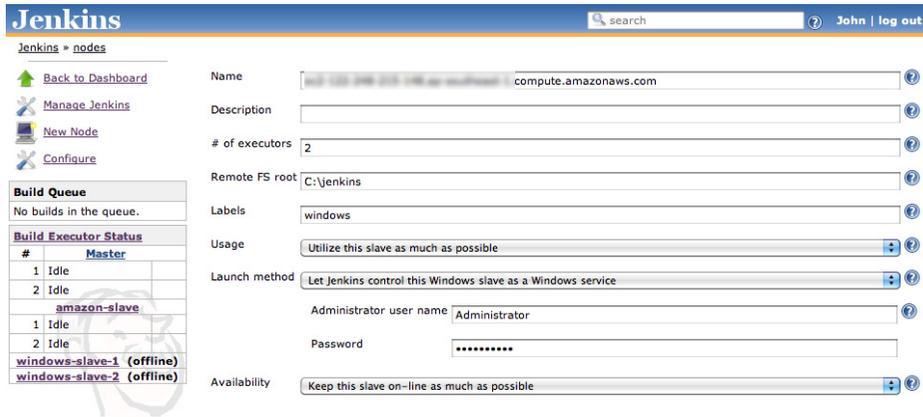


Figure 11.14. Permettre à Jenkins de contrôler un esclave Windows comme un service Windows

11.4. Associer une tâche de build avec un esclave ou un groupe d'esclaves

Dans la section précédente, nous avons vu comment attribuer des libellés à vos noeuds esclaves. C'est un moyen commode pour grouper vos esclaves en fonction de caractéristiques telles que le système d'exploitation, l'environnement cible, le type de base de données, ou tout autre critère pertinent dans votre processus de build. Une application commune de cette pratique est d'exécuter des tests fonctionnels spécifiques à un OS sur des noeuds esclaves dédiés, ou de réserver une machine particulière aux tests de performance).

Une fois que vous avez affecté vos libellés à vos noeuds esclaves, vous devez aussi dire à Jenkins où il peut exécuter les tâches de build. Par défaut, Jenkins utilisera simplement le premier noeud esclave disponible, ce qui offre généralement le meilleur temps de traitement global. Si vous avez besoin d'attacher une tâche de build à une machine ou un groupe de machines particulier, vous devez cocher la case "Restreindre les emplacements où ce projet peut s'exécuter" dans la page de configuration du build (voir Figure 11.15, "Exécuter une tâche de build sur un noeud esclave particulier"). Ensuite, entrez le nom de la machine, ou un libellé identifiant un groupe de machines, dans le champ Expression de libellé. Jenkins fournira une liste déroulante dynamique montrant les noms de machines et libellés de au fur et à mesure que vous tapez.

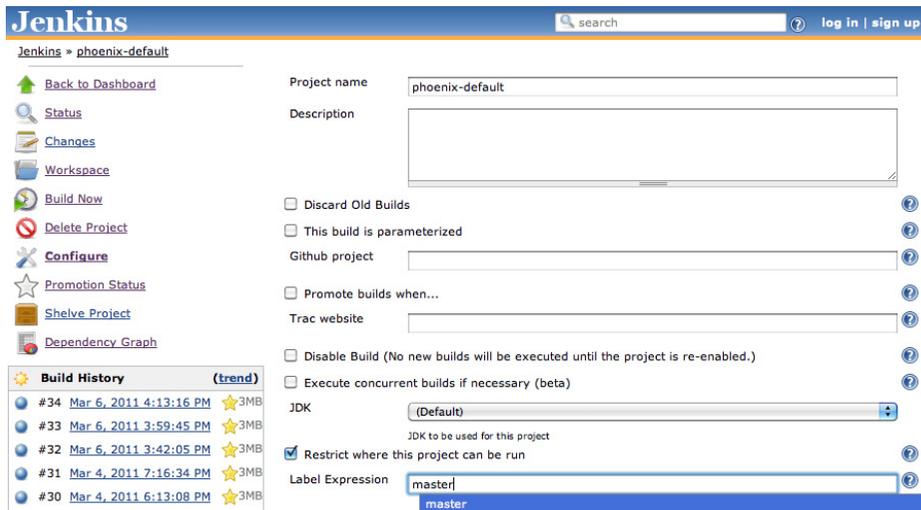


Figure 11.15. Exécuter une tâche de build sur un noeud esclave particulier

Ce champ admet aussi des expressions booléennes, ce qui vous permet de définir des contraintes plus compliquées spécifiant où votre build devrait s'exécuter. Le plus simple pour expliquer comment utiliser ces expressions est de montrer des exemples. Supposez que vous avez une ferme de construction avec des noeuds esclaves Windows et Linux (identifiés par les libellés "windows" et "linux"), distribués sur trois sites ("sydney", "sanfrancisco", et "london"). Votre application nécessite aussi d'être testée sur différentes bases de données ("oracle", "db2", "mysql", et "postgres"). Vous pouvez aussi utiliser des libellés pour distinguer les noeuds esclaves utilisés pour déployer vers différents environnements (test, test d'acceptation, production).

L'utilisation la plus simple des expressions de libellés est de définir où une tâche de build peut ou ne peut pas être exécutée. Si vos tests web nécessitent Internet Explorer, par exemple, vous aurez besoin de les exécuter sur une machine Windows. Vous pourriez exprimer cela en indiquant simplement le libellé suivant :

```
windows
```

Sinon, vous pourriez vouloir exécuter vos tests sur Firefox, mais seulement sur des machines Linux. Vous pourriez exclure les machines Windows de l'éventails des noeuds candidats en utilisant l'opérateur ! :

```
!windows
```

Vous pouvez aussi utiliser les opérateurs **et** (&&) et **ou** (!!) pour combiner les expressions. Par exemple, supposez que la base Postgres soit uniquement testée pour Linux. Vous pourriez dire à Jenkins d'exécuter une tâche de build particulière seulement sur les machines Linux sur lesquelles est installé Postgres en utilisant l'expression suivante :

```
linux && postgres
```

Où vous pourriez spécifier qu'une tâche de build particulière doit uniquement tourner sur l'environnement de test d'acceptation utilisateur de Sydney ou Londres :

```
uat && (sydney || london)
```

Si votre nom de machine contient des espaces, vous devrez les entourer de double quotes :

```
"Windows 7" || "Windows XP"
```

Il y a aussi deux opérateurs logiques plus avancés que vous pourriez trouver utile. L'opérateur **implique** (\Rightarrow) vous permet de définir une contrainte logique de la forme "si A est vrai, alors B doit aussi être vrai." Par exemple, supposez que vous ayez une tâche de build qui peut s'exécuter sur n'importe quelle distribution Linux, mais que si c'est une machine Windows, ce doit être Windows 7. Vous pourriez exprimer cette contrainte comme suit :

```
windows -> "Windows 7"
```

L'autre opérateur logique est l'opérateur **si-et-seulement-si** (\Leftrightarrow). Cette opération vous permet de définir des contraintes plus fortes de la forme "Si A est vrai, alors B doit être vrai, mais si A est faux, alors B doit être faux". Par exemple, supposez que les tests Windows 7 doivent uniquement être exécutés sur l'environnement de tests d'acceptation utilisateur, et que seuls les tests Windows 7 doivent être exécutés dans l'environnement de tests d'acceptation. Vous pourriez exprimer cela comme montré ici :

```
"Windows 7" <-> uat
```

11.5. Surveillance des noeuds

Jenkins ne distribue pas les tâches de build aux agents esclaves et advienne que pourra : il surveille proactivement les machines esclaves, et mettra un noeud offline s'il considère que celui-ci est incapable d'effectuer un build sans danger. Vous pouvez définir exactement ce que Jenkins surveille dans l'écran Gérer les noeuds (voir Figure 11.16, "Jenkins surveille proactivement vos agents de build"). Jenkins surveille les agents esclave de plusieurs façons. Il surveille le temps de réponse : un temps de réponse excessif peut indiquer soit un problème réseau soit que la machine est tombée. Il surveille aussi la quantité d'espace disque, l'espace disque temporaire et l'espace de swap disponible à l'utilisateur Jenkins sur la machine esclave, puisque les tâches de build peuvent notamment être consommatrices en espace disque. Il garde aussi un oeil sur les horloges systèmes, parce que si les horloges ne sont pas correctement synchronisées, des erreurs bizarres peuvent apparaître. Si l'un de ces critères n'est pas rempli, Jenkins mettra automatiquement le serveur hors-ligne.

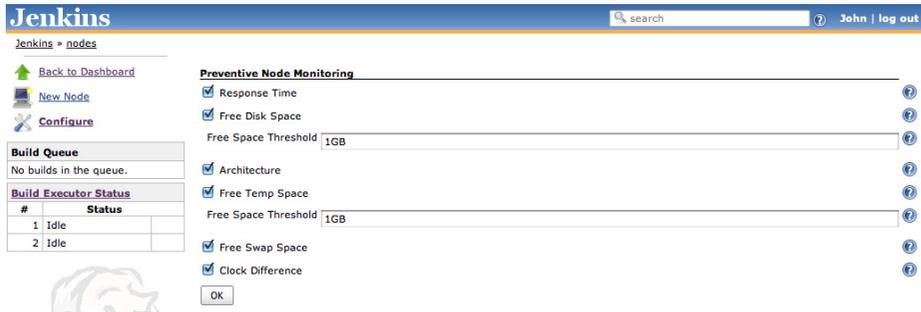


Figure 11.16. Jenkins surveille proactivement vos agents de build

11.6. Cloud computing

Le cloud computing consiste à utiliser des ressources matérielles sur Internet comme extension et/ou en remplacement de votre architecture informatique locale. Le cloud computing est en expansion dans plusieurs domaines de l'entreprise, incluant l'email et le partage de document (Gmail et Google Apps sont des exemples particulièrement connus, mais il y en a d'autres), stockage de données hors-site (comme Amazon S3), aussi bien que des services techniques comme des dépôts de code source (comme GitHub, Bitbucket, etc.) et de nombreux autres.

Bien sûr les solutions d'architecture matérielle externalisée existent depuis longtemps. Le point principal qui distingue le cloud computing des services plus traditionnels est la vitesse et la flexibilité avec laquelle un service peut être monté, et démonté quand il n'est plus nécessaire. Dans un environnement de cloud computing, une nouvelle machine peut fonctionner et être disponible en quelques secondes.

Cependant, le cloud computing dans le contexte de l'Intégration Continue n'est pas toujours aussi simple qu'il n'y paraît. Pour qu'une approche basée sur le cloud fonctionne, certaines de vos ressources internes pourraient devoir être disponibles au monde extérieur. Ceci peut inclure l'ouverture d'accès à votre système de contrôle de version, votre base de données de test, et à n'importe quelle autre ressource que vos builds et vos tests requièrent. Tous ces aspects doivent être examinés attentivement lors du choix d'une architecture d'IC basée sur le cloud, et pourrait limiter vos options si certaines ressources ne peuvent tout simplement pas être accédées depuis Internet. Malgré tout, l'IC basée sur le cloud a le potentiel pour vous offrir d'énormes bénéfices sur l'évolutivité de votre infrastructure.

Dans les sections suivantes, nous regarderons comment utiliser les services de cloud computing d'Amazon EC2 pour mettre en place une ferme de build basée sur le cloud.

11.6.1. Utiliser Amazon EC2

En plus de vendre des livres, Amazon est l'un des fournisseurs les plus connus de services cloud computing. Si vous êtes prêt à payer pour le service, Amazon peut vous fournir des machines de build qui peuvent soit être utilisées de façon permanente comme partie de votre ferme de build, soit mis en ligne au besoin lorsque vos machines de build existantes deviennent surchargées. C'est un moyen excellent

et raisonnablement coûteux pour absorber la charge exceptionnel de build en fonction de vos besoins, et sans le mal de tête associé à des machines physiques supplémentaires à maintenir.

Si vous voulez la flexibilité d'une architecture d'IC basée sur le cloud, mais que vous ne voulez pas externaliser votre matériel, une autre option est de mettre en place un cloud Eucalyptus. Eucalyptus est un outil open source qui vous permet de créer localement un cloud privé sur du matériel existant. Eucalyptus utilise une API compatible avec Amazon EC2 et S3, et fonctionne bien avec Jenkins.

11.6.1.1. Mettre en place votre ferme de build Amazon EC2

Amazon EC2 est probablement le service de cloud computing commercial le plus populaire et le plus connu. Pour utiliser ces services, vous devrez créer un compte EC2 avec Amazon si vous n'en avez pas déjà un. Le processus requis pour faire cela est bien documenté sur le site web d'Amazon, nous n'insisterons donc pas ici sur le sujet. Une fois que vous avez créé votre compte, vous pouvez créer des machines virtuelles et des images de machines qui formeront votre ferme de build basée sur EC2.

Quand vous utilisez Amazon EC2, vous créez des machines virtuelles, appelées instances, en utilisant la console de gestion Amazon Web Services (AWS) (voir Figure 11.17, “Vous gérez vos instances EC2 en utilisant la console de gestion Amazon AWS”). Ce site web vous permet de gérer vos instances en fonctionnement et d'en créer de nouvelles. Vous créez ces instances à partir d'images prédéfinies, appelées Amazon Machine Images (AMIs). Il y a plusieurs images AMI, à la fois d'Amazon et dans le domaine public, que vous pouvez utiliser comme point de départ, couvrant la plupart des systèmes d'exploitation populaires. Une fois que vous avez créé une nouvelle instance, vous pouvez vous y connecter soit via SSH (pour les machines unix) soit via une connexion à distance Windows, pour la configurer en fonction de ce que vous voulez en faire.

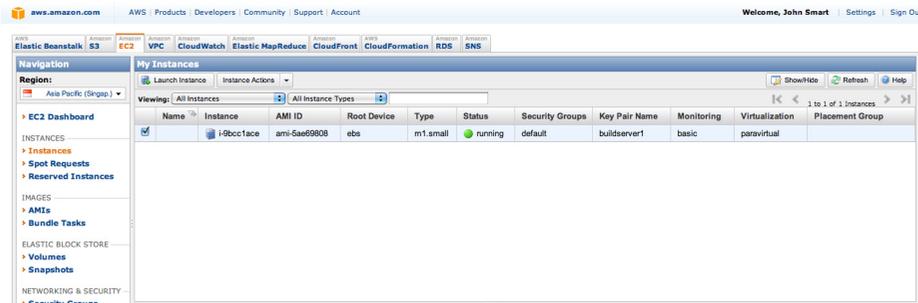


Figure 11.17. Vous gérez vos instances EC2 en utilisant la console de gestion Amazon AWS

Pour mettre en place une ferme de build, vous aurez également besoin de configurer la votre, rendez-vous simplement dans le menu Paires de clés dans la Sécurité du serveur de build afin d'être en mesure d'accéder à vos instances EC2. En particulier, vous aurez besoin d'installer les outils de l'API Amazon EC2, de configurer les clés privées/publiques appropriées, et d'autoriser les connexions SSH depuis votre serveur ou votre réseau vers vos instances Amazon. Encore une fois, les détails indiquant comment faire cela sont bien documentés pour tous les systèmes d'exploitation principaux sur le site web EC2.

Vous pouvez utiliser les instances Amazon EC2 de deux façons — soit vous créez les machines esclaves sur Amazon EC2 et les utilisez comme machines distantes, soit vous demandez à Jenkins de les créer dynamiquement pour vous à la demande. Ou vous pouvez avoir une combinaison des deux. Les approches ont leur utilité, et nous discuterons de chacune d'elles dans les sections suivantes.

11.6.1.2. Utiliser des instances EC2 comme partie de votre ferme de build

Créer une nouvelle instance EC2 revient tout simplement à choisir une image de base que vous voulez utiliser. Vous devrez juste fournir quelques détails de base à propos de l'instance, comme sa taille et sa capacité, et la clé privée que vous voulez utiliser pour accéder à la machine. Amazon créera ensuite une nouvelle machine virtuelle basée sur cette image. Une fois que vous avez configuré cela, une instance EC2 est en substance une machine comme n'importe quelle autre, et il est simple et commode de configurer des machines EC2 permanentes ou semi-permanentes comme part de votre infrastructure de build. Vous pourriez même opter pour utiliser une image EC2 pour votre serveur maître.

Configurer une instance EC2 existante comme un esclave Jenkins est peu différent que de configurer n'importe quel autre esclave distant. Si vous mettez en place un esclave EC2 Unix ou Linux, vous devrez référencer le fichier de clé privée (voir Figure 11.18, “Configurer un esclave Amazon EC2”) que vous avez utilisé pour créer l'instance EC2 sur la console de gestion AWS. En fonction du type de Linux que vous utilisez, vous pourriez aussi avoir besoin de fournir un nom d'utilisateur. La plupart des distributions se connectent en tant que root, mais certaines, comme Ubuntu, nécessitent un nom d'utilisateur différent.

Launch method: Launch slave agents on Unix machines via SSH

Host: ap-southeast-1.compute.amazonaws.com

Username: ubuntu

Password: [Redacted]

Private Key File: /var/lib/jenkins/ec2/buildserver1.pem

Port: 22

JavaPath: [Redacted]

JVM Options: [Redacted]

Availability: Take this slave on-line when in demand and off-line when idle

In demand delay: 5

Idle delay: 30

Figure 11.18. Configurer un esclave Amazon EC2

11.6.1.3. Utiliser des instances dynamiques

La seconde approche implique de créer de nouvelles machines Amazon EC2 dynamiquement, lorsqu'elles sont nécessaires. Configurer des instances dédiées n'est pas difficile, mais cela ne s'adapte pas très bien à la charge. Une meilleure approche est de laisser Jenkins créer de nouvelles instances en fonction du besoin. Pour faire cela, vous aurez besoin d'installer le plugin Jenkins Amazon EC2. Ce plugin permet à Jenkins de démarrer des esclaves EC2 sur le cloud à la demande, et de les éteindre ensuite lorsqu'ils ne sont plus nécessaires. Le plugin fonctionne à la fois avec Amazon EC2, et Ubuntu

Enterprise Cloud. Nous nous concentrerons ici sur Amazon EC2. Notez qu'à l'écriture de ces lignes le plugin supportait seulement la gestion des images EC2 Unix.

Une fois que vous avez installé le plugin et redémarré Jenkins, allez dans l'écran principal de configuration et cliquez sur Ajouter un Nouveau Cloud (voir Figure 11.19, "Configurer un esclave Amazon EC2"). Choisissez Amazon EC2. Vous devez fournir votre clé d'identification d'accès Amazon (NdT : Amazon Access Key ID) et votre clé d'accès secrète afin que Jenkins puisse communiquer avec votre compte Amazon EC2. Vous pouvez accéder à ceux-ci dans l'écran Paires de clés de votre tableau de bord EC2.

The screenshot shows the Jenkins configuration interface for Amazon EC2. The 'Cloud' section is expanded to show 'Amazon EC2' configuration. The 'Region' is set to 'Asia Pacific (Singapore)'. The 'Access Key ID' is 'AKIAIRI...'. The 'Secret Access Key' is masked with dots. The 'EC2 RSA Private Key' is a long alphanumeric string. Below these are the 'AMIs' configuration fields: 'AMI ID' is 'ami-7c0f762e', 'Instance Type' is 'DEFAULT', 'Description' is 'ubuntu', 'Remote FS root' is '/var/jenkins', 'Remote user' is empty, and 'Labels' are 'ubuntu, linux, amazon'. There is an empty 'Init Script' field. Buttons for 'Advanced...', 'Generate Key', 'Test Connection', 'Check AMI', 'Add', 'Delete', and 'Delete cloud' are visible.

Figure 11.19. Configurer un esclave Amazon EC2

Vous devrez aussi fournir votre clé privée RSA. Si vous n'en avez pas, rendez vous simplement dans le menu Paires de clé dans l'écran Security Credential et créez en une. Cela créera une nouvelle paire de clé pour vous et téléchargera la clé privée. Conservez la clé privée dans un endroit sûr (vous en aurez besoin si vous voulez vous connecter à vos instances EC3 via SSH).

Dans les options avancées, vous pouvez utiliser le champ Limite d'Instances pour limiter le nombre d'instances EC2 que Jenkins lancera. Cette limite est liée au nombre total d'instances en exécution, pas seulement celles que Jenkins exécute en ce moment. C'est une mesure de sécurité utile, parce que vous payez pour le temps pendant lequel vos instances restent actives.

Une fois que vous avez configuré votre connexion EC2 globale, vous devez définir les machines avec lesquelles vous travaillerez. Vous faites cela en spécifiant l'identifiant d'Image Miroir Amazon (AMI) de l'image serveur avec laquelle vous aimeriez démarrer. Amazon fournit quelques images de démarrage, et plusieurs autres le sont par la communauté, toutefois elles ne fonctionneront pas toutes avec EC2. A

l'écriture de ces lignes, seules certaines images basées sur des distributions Linux 32-bit fonctionnent correctement.

Les images AMI prédéfinies par Amazon et publiques sont des points de départ utiles pour vos machines virtuelles permanentes, mais pour les besoins inhérents à la mise en oeuvre d'un cloud dynamique basé sur EC2, vous devez définir vos propres AMI avec les outils essentiels (Java, outils de build, configuration SCM etc.) préinstallés. Heureusement, c'est un processus simple : démarrez simplement avec une AMI générique (de préférence une compatible avec le plugin Jenkins EC2), et installez tout ce dont vous avez besoin. Assurez-vous d'utiliser une image EBS. De cette façon, les changements que vous faites sur votre instance serveur sont sauvegardés sur un volume EBS afin que vous ne les perdiez pas lorsque le serveur s'éteint. Créez alors une nouvelle image en sélectionnant l'option Créer une image dans l'écran Instances de la console de gestion EC2 (voir Figure 11.20, "Créer une nouvelle image Amazon EC2"). Vérifiez que SSH est ouvert depuis l'adresse IP de votre serveur de build dans le groupe de sécurité par défaut sur Amazon EC2. Si vous ne faites pas cela, Jenkins échouera après avoir attendu trop longtemps le démarrage du nouveau noeud esclave.

Une fois que vous aurez préparé votre image, vous serez capable de l'utiliser pour votre configuration EC2.

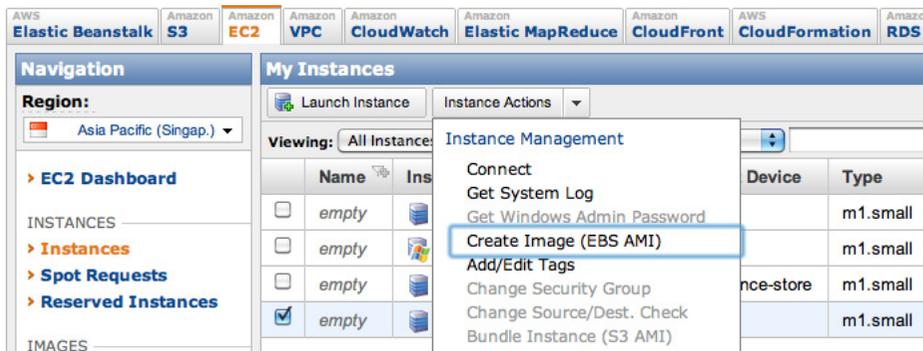


Figure 11.20. Créer une nouvelle image Amazon EC2

A présent, Jenkins créera automatiquement une nouvelle instance EC2 en utilisant cette image lorsque c'est nécessaire, et supprimera (ou la "terminera," en termes Amazon) l'instance une fois qu'il n'en aura plus besoin. Sinon, vous pouvez ajouter un nouvel esclave EC2 manuellement depuis l'écran Noeuds en utilisant le bouton Provisionner via EC2 (voir Figure 11.21, "Ajouter un nouvel esclave Amazon EC2 manuellement"). C'est un moyen utile pour tester votre configuration.

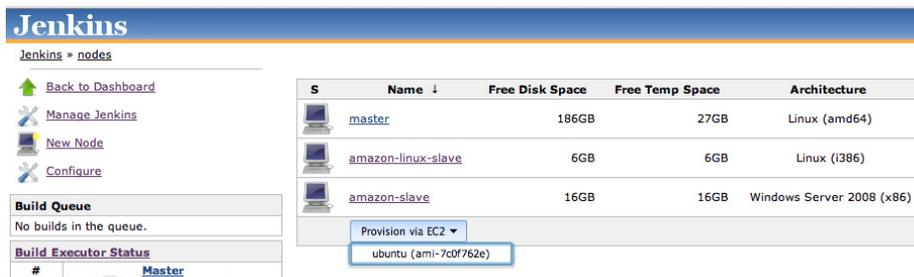


Figure 11.21. Ajouter un nouvel esclave Amazon EC2 manuellement

11.7. Utiliser le service CloudBees DEV@cloud

Un autre option que vous pourriez considérer est d'exécuter votre instance Jenkins en utilisant une architecture basée sur le cloud dédiée à Jenkins, comme le service DEV@cloud offert par CloudBees. CloudBees fournit du "Jenkins as a service" et d'autres services de développement variés (comme Sonar) autour de Jenkins. En utilisant un service spécifique dédié à Jenkins, il n'est pas nécessaire d'installer (ou de gérer) des maîtres ou esclaves Jenkins sur vos machines. Une instance maître est automatiquement configurée pour vous, et quand vous donnez une tâche à construire, CloudBees met à disposition un esclave pour vous et le reprend quand la tâche est effectuée.

Comment cette approche se compare-t-elle à l'architecture basée sur Amazon EC2 que nous avons présenté dans la section précédente ? L'avantage principal est qu'il y a beaucoup moins de travail à effectuer dans la gestion du matériel de votre architecture d'IC. Utiliser l'infrastructure Amazon EC2 vous évite d'avoir à vous soucier du matériel, mais vous avez encore à configurer et gérer vos images de serveur par vous-même. L'architecture CloudBees DEV@cloud est de plus haut-niveau, un service centré sur l'IC, qui fournit non seulement un serveur Jenkins mais aussi d'autres outils en relation comme des dépôts SVN ou Git, de la gestion utilisateur, et Sonar. En plus, le modèle de tarification (paiement à la minute) est sans doute mieux adapté à une architecture d'IC basée sur le cloud que l'approche par paiement à l'heure utilisée par Amazon.

Les services basés sur Amazon EC2 sont souvent, mais pas toujours, utilisés dans un environnement de "cloud hybride" où vous déchargez vos tâches sur le cloud, mais une certaine quantité de vos build restent chez vous. Le service DEV@cloud de CloudBees est une solution de cloud public où le build complet se déroule dans le cloud (bien que CloudBees offre une solution similaire fonctionnant sur un cloud privé).

Créer un compte CloudBees DEV@cloud est facile, et vous pouvez en utiliser un gratuit pour expérimenter le service (notez que le service gratuit CloudBees a seulement un nombre limité de plugins disponibles ; vous devrez vous inscrire à la version professionnelle pour pouvoir utiliser la gamme complète de plugins). Pour créer un compte CloudBees, allez sur la page d'enregistrement¹. Vous devrez entrer quelques informations telles que le nom d'utilisateur, des informations email, et un nom de compte.

¹ <https://grandcentral.cloudbees.com/account/signup>

Une fois enregistré, vous aurez accès à la fois aux services DEV@cloud et RUN@cloud (en résumé la plateforme entière CloudBees).

A ce stade, vous devrez souscrire au service DEV@cloud. Pour nos besoins, vous pouvez vous en sortir en choisissant simplement l'option gratuite. Vous devrez attendre quelques minutes que CloudBees mette à disposition un maître Jenkins pour vous. L'étape suivante est de valider votre compte (ceci aide CloudBees à éviter la création de comptes factices de faire tourner des tâches fallacieuses sur le service). Cliquez sur le lien de validation, et entrez votre numéro de téléphone. Un appel entrant automatique vous donnera un code ; entrez le code sur le formulaire. Une fois ceci fait, vous pouvez commencer à exécuter des builds.

Votre première escale après connexion sera la console de gestion (appelée GrandCentral). Cliquez sur le bouton “Jenkins Builds” pour vous rendre sur votre maître Jenkins flambant neuf.

A partir de là, votre interaction avec la plateforme DEV@cloud est exactement comme un Jenkins autonome. Quand vous créez une nouvelle tâche de build, pointez simplement sur votre dépôt de code source existant et appuyez sur build repository. DEV@cloud mettra à disposition un esclave pour vous et lancera un build (cela pourrait prendre une minute ou deux pour préparer l'esclave).

11.8. Conclusion

Dans le domaine de l'Intégration Continue, les builds distribués sont la clé d'une architecture vraiment évolutive. Que vous ayez besoin de capacité de build supplémentaire en un claquement de doigt, ou si vos modèles de build sont sujet à des périodes de pics de demande, une architecture de build distribuée est un excellent moyen d'absorber une charge additionnelle. Les builds distribués sont aussi un bon moyen pour déléguer des tâches spécialisées, comme du test web spécifique à un OS, à certaines machines dédiées.

Une fois que vous commencez à suivre le chemin des builds distribués, les fermes distribuées basées sur le cloud sont une extension très logique. Mettre vos serveurs de build dans le cloud simplifie l'évolutivité de votre infrastructure de build lorsque cela devient nécessaire, et autant que nécessaire.

Chapter 12. Déploiement automatisé et livraison continue

12.1. Introduction

L'Intégration Continue ne devrait pas s'arrêter une fois que votre application compile correctement. Elle ne devrait pas non plus s'interrompre une fois que des tests, contrôles automatiques ou audit de la qualité du code puissent être lancés. L'enchaînement naturel, une fois toutes ces étapes mises en oeuvre, est d'étendre votre processus de construction automatisé au déploiement. Cette pratique est connue globalement sous le nom de Déploiement Automatisé ou Déploiement Continu.

Dans sa forme la plus élaborée, le Déploiement Continu est le processus où toute modification du code, dotée des tests automatisés et autres vérifications appropriées, est immédiatement déployée en production. Le but est de réduire la durée du cycle ainsi que le temps et l'effort du processus de déploiement. Cela aide également les équipes de développement à réduire le temps nécessaire pour livrer des fonctionnalités individuelles ou des corrections de bogue, et par conséquent augmente significativement la productivité des équipes. Réduire ou éliminer ces périodes d'intenses activités menant à une livraison traditionnelle ou à un déploiement libère également du temps et des ressources pour améliorer les processus et pour l'innovation. Cette approche est comparable à la philosophie de l'amélioration continue promue par des processus Lean tel que Kanban.

Cependant, déployer systématiquement le dernier code en production n'est pas toujours approprié, quelque soit la qualité de vos tests automatisés. De nombreuses organisations ne sont pas préparées à ce que de nouvelles versions apparaissent sans annonce chaque semaine. Des utilisateurs ont peut être besoin d'être formés, du marketing peut être nécessaire autour du produit et ainsi de suite. Une variante plus conservatrice de ce principe, souvent vue dans les organisations de grande taille, est d'avoir le processus de déploiement entièrement automatisé mais de déclencher le déploiement effectif via un mécanisme à un clic. Cela est connu sous le nom de Livraison Continue, et a tous les avantages du Déploiement Continu sans ses inconvénients. Des variantes au processus de Livraison Continue peuvent aussi impliquer des déploiements automatisés vers certains environnements (tels que test et AQ) tout en utilisant un déploiement manuel à un clic pour d'autres environnements (tels que recette et production). La caractéristique la plus importante de la Livraison Continue est que chaque build ayant passé avec succès tous les tests automatisés et vérifications de qualités appropriés puisse être potentiellement déployé en production au moyen d'un processus entièrement automatisé et déclenché via un clic, au choix de l'utilisateur final et en quelques minutes. Le processus n'est cependant pas automatique : c'est l'équipe métier et non informatique qui décide du meilleur moment pour livrer les dernières modifications.

Le Déploiement Continu et la Livraison Continue sont tous deux considérés, à juste titre, comme signes d'un haut niveau de maturité en termes de processus de build et de pratiques SDLC. Ces techniques ne peuvent exister sans un ensemble très solide de tests automatisés. Pas plus qu'ils ne peuvent exister sans

un environnement d'Intégration Continue et un séquenceur de build robuste. En effet, le Déploiement Continu ou la Livraison Continue représente généralement la dernière étape et le but d'un pipeline de build. Cependant, vu les avantages significatifs que ces pratiques apportent, elles sont un objectif pertinent. Dans la suite de ce chapitre nous allons utiliser le terme "Déploiement Continu" pour parler tant de Déploiement Continu que de Livraison Continue. En effet, le Déploiement Continu peut être vu comme la Livraison Continue avec une étape finale (le déploiement en production) dictée par la partie métier plutôt que l'équipe de développement.

12.2. Mise en oeuvre du déploiement automatisé et continu

Dans sa forme la plus élémentaire, le Déploiement Automatisé peut être aussi simple que l'écriture de vos propres scripts afin de déployer votre application vers un serveur particulier. L'avantage principal de la solution scriptée est la simplicité et l'aisance en termes de configuration. Cependant, une telle approche peut atteindre ses limites si vous avez besoin de mettre en oeuvre des actions de déploiement plus élaborées, telles qu'installer un logiciel sur une machine ou redémarrer le serveur. Pour des scénarios plus avancés, vous pourriez avoir besoin d'un outil de déploiement/configuration plus sophistiqué tel que Puppet ou Chef.

12.2.1. Le script de déploiement

Une part essentielle de toute initiative de Déploiement Automatisé est un processus de déploiement scripté. Bien que celui puisse sembler évident, il y a encore de nombreuses organisations où le déploiement demeure un processus consommateur de ressources, compliqué et lourd, incluant copie manuelle de fichiers, exécution manuelle de script, des notes écrites à la main et ainsi de suite. La bonne nouvelle est qu'en général cela n'a vocation pas à rester ainsi. Avec un peu de travail, il est généralement possible d'écrire un script pour automatiser la plupart, voir l'intégralité, du processus.

La complexité d'un script de déploiement varie énormément d'une application à une autre. Pour un simple site web, un déploiement de script peut se limiter à resynchroniser un dossier sur le serveur cible. De nombreuses applications Java ont Ant ou des plugins Maven qui peuvent être utilisés pour le déploiement. Pour une architecture plus compliquée, le déploiement peut impliquer plusieurs applications et services sur de multiples serveurs en grappe, le tout coordonné de façon extrêmement précise. La plupart des processus de déploiement sont entre ces deux extrêmes.

12.2.2. Mises à jour de base de données

Déployer votre application vers le serveur d'application est généralement seulement une partie du puzzle. Les bases de données, relationnelles ou autres, jouent presque toujours un rôle central dans l'architecture logiciel. Bien sûr, idéalement, votre base de données devrait être parfaite dès le début, mais cela est rarement le cas dans le monde réel. En effet, lorsque vous mettez à jour votre application, vous avez généralement également besoin de mettre une ou plusieurs bases de données à jour.

Les mises à jour de base de données sont généralement plus difficile à mettre en oeuvre que les mises à jour applicatives, vu que tant la structure que les données sont susceptibles d'être modifiées. Cependant, les mises à jour de base de données sont critiques pour le développement et le déploiement. Elles méritent donc de l'attention et de la planification.

Certains frameworks, tels que Ruby on Rails et Hibernate, peuvent supporter automatiquement des changements structurels d'une base de données. En utilisant ces frameworks, vous pouvez usuellement spécifier si vous voulez créer un nouveau schéma de la base à chaque mise à jour ou si vous voulez mettre à jour le schéma tout en conservant les données. Bien que cela semble utile à première vue, ces fonctionnalités sont en fait juste suffisantes pour des environnements non critiques. Ces outils ne gèrent pas bien les migrations de données, point pourtant essentiel. Par exemple, si vous créez une colonne dans votre base de données, le processus de mise à jour va simplement créer une nouvelle colonne: il ne va pas copier les données de l'ancienne colonne vers la nouvelle, pas plus qu'il ne va supprimer l'ancienne colonne de la table mise à jour.

Heureusement, cela n'est pas la seule approche disponible. Un autre outil qui tente de résoudre le problème complexe des mises à jour de base de données est Liquibase¹. Liquibase est un outil open source qui permet d'organiser des processus de mises à jour entre versions d'une base de données à travers une approche de haut niveau.

Liquibase garde un historique des mises à jour appliquées dans une table de la base de données. Il peut ainsi aisément amener n'importe quelle base vers l'état souhaité. Par conséquent, pas de risque d'appliquer deux fois le même script de mise à jour : Liquibase applique seulement les scripts qui n'ont pas encore été appliqués. Liquibase est aussi capable de défaire des changements, du moins pour certains types de changements. Vu que cela ne fonctionne pas pour tous les changements (les données d'une table supprimée, par exemple, ne peuvent pas être restaurées) il est préférable de ne pas trop compter sur cette fonctionnalité.

Dans Liquibase, les changements de la base de données sont agrégés sous la forme "d'ensembles de changement", chacun représentant la mise à jour dans un format XML indépendant de la base de données. Ces ensembles de changement peuvent inclure tous les changements que vous pourriez appliquer à une base de données, de la création ou suppression de table à la création ou mise à jour de colonne, index ou clés étrangères :

```
<databaseChangeLog
xmlns="http://www.liquibase.org/xml/ns/dbchangelog/1.6"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.liquibase.org/xml/ns/dbchangelog/1.6
http://www.liquibase.org/xml/ns/dbchangelog/dbchangelog-1.6.xsd">
  <changeSet id="1" author="john">
    <createTable tableName="department">
      <column name="id" type="int">
        <constraints primaryKey="true" nullable="false"/>
      </column>
      <column name="name" type="varchar(50)">
        <constraints nullable="false"/>
      </column>
    </createTable>
  </changeSet>
</databaseChangeLog>
```

¹ <http://www.liquibase.org/>

```

    </column>
    <column name="active" type="boolean" defaultValue="1"/>
  </createTable>
</changeSet>
</databaseChangeLog>

```

Les ensembles de changement peuvent aussi refléter des modifications à une table existante. Par exemple, l'ensemble de changement suivant représente le renommage d'une colonne :

```

<changeSet id="1" author="bob">
  <renameColumn tableName="person" oldColumnName="fname" newColumnName="firstName"/>
</changeSet>

```

Etant donné que cette représentation concerne la nature sémantique du changement, Liquibase est capable de réaliser correctement tant la mise à jour du schéma que la migration de données correspondante.

Liquibase peut aussi bien gérer les changements de données que de structure. Par exemple, l'ensemble de changement suivant insère une nouvelle ligne de données dans une table :

```

<changeSet id="326" author="simon">
  <insert tableName="country">
    <column name="id" valueNumeric="1"/>
    <column name="code" value="AL"/>
    <column name="name" value="Albania"/>
  </addColumn>
</changeSet>

```

Chaque ensemble de changements a un identifiant et un auteur, ce qui facilite la traçabilité et réduit le risque de conflit. Les développeurs peuvent tester leurs ensembles de changements à leur propre base de données et les archiver une fois qu'ils sont prêts. Bien sûr, l'étape suivante est de configurer Jenkins pour qu'il applique les mises à jour Liquibase à la base de données appropriée avant que les tests d'intégration ou que les déploiements d'applications ne soient réalisés, généralement en tant que partie du script de build ordinaire du projet.

Liquibase s'intègre bien dans le processus de build. Il peut être exécuté depuis la ligne de commande ou être intégré dans Ant ou Maven. Pour ce dernier, par exemple, vous pouvez configurer le "Maven Liquibase Plugin" comme suit :

```

<project>
  <build>
    <plugins>
      <plugin>
        <groupId>org.liquibase</groupId>
        <artifactId>liquibase-plugin</artifactId>
        <version>1.9.3.0</version>
        <configuration>
          <propertyFileWillOverride>true</propertyFileWillOverride>
          <propertyFile>src/main/resources/liquibase.properties</propertyFile>
        </configuration>
      </plugin>
    </plugins>

```

```
</build>
...
</project>
```

En utilisant ainsi Liquibase avec Maven, vous pourriez mettre à jour une base de données cible vers le schéma courant en utilisant ce plugin:

```
$ mvn liquibase:update
```

Les informations de connexion par défaut à la base de données sont spécifiées dans le fichier `src/main/resources/liquibase.properties`, et ont l'aspect suivant :

```
changeLogFile = changelog.xml
driver = com.mysql.jdbc.Driver
url = jdbc:mysql://localhost/ebank
username = scott
password = tiger
verbose = true
dropFirst = false
```

Vous pouvez toutefois surcharger n'importe laquelle de ces propriétés depuis la ligne de commande, ce qui rend facile de configurer un build Jenkins pour mettre à jour différentes bases de données.

D'autres commandes similaires vous permettent de générer un script SQL (si vous avez besoin de le soumettre à votre administrateur de base de données par exemple), ou de revenir à une version précédente du schéma.

Cela n'est bien sûr qu'un exemple d'une solution possible. D'autres équipes préfèrent maintenir manuellement une série de scripts SQL de mises à jour, voir écrivent leur propre solution. L'important est d'avoir un outillage vous permettant de mettre à jour différentes bases de données de façon fiable et reproductible lors de vos déploiements applicatifs.

12.2.3. Tests fumigatoires

N'importe quel déploiement automatisé sérieux a besoin d'être suivi par une série de tests fumigatoires automatisés. Un sous ensemble des tests d'acceptance automatisés peut faire un bon candidat. Les tests fumigatoires devraient être non intrusifs et relativement rapides. Ils doivent pouvoir être exécutés en production, ce qui est susceptible de restreindre le nombre d'actions possibles durant les tests.

12.2.4. Revenir sur des changements

Un autre aspect important à considérer lors de la mise en place du Déploiement Automatisé est comment revenir en arrière si quelque chose se passe mal. Cela est encore plus important si vous voulez implémenter du Déploiement Continu. Il est en effet critique de pouvoir revenir en arrière si besoin.

La mise en oeuvre varie grandement en fonction de l'application. Bien qu'il soit relativement simple de redéployer une version précédente d'une application en utilisant Jenkins (nous verrons cela plus loin dans ce chapitre), l'application n'est généralement pas le seul acteur en jeu. Un retour en arrière de la base de données à un état précédent est souvent à mettre en oeuvre.

Nous avons vu comment il est possible d'utiliser Liquibase pour les mises à jour de base de données, et bien sûr de nombreuses autres stratégies sont également possibles. Cependant, revenir à un état précédent de la base de données présente des défis particuliers. Liquibase, par exemple, permet de revenir sur certains changements de structure de la base de données, mais pas de tous. De plus, des données perdues (lors de la suppression de table par exemple) ne peuvent être restaurées en utilisant que Liquibase.

La façon la plus fiable de retourner à un état précédent est probablement de prendre une image de la base juste avant la mise à jour. Cette image peut ensuite être utilisée lors de la restauration. Une méthode efficace est d'automatiser ce processus dans Jenkins dans la tâche de déploiement, et de sauver l'image de la base et le fichier binaire déployable en tant qu'artéfacts. De cette façon, vous pouvez aisément restaurer la base de données en utilisant l'image sauvegardée et ensuite de redéployer l'application en utilisant le fichier déployable sauvé. Nous allons regarder un exemple de cette stratégie plus loin dans ce chapitre.

12.3. Déployer vers un serveur d'application

Jenkins fournit des plugins pour aider à déployer votre application vers un certain nombre de serveurs d'applications fréquemment utilisés. Le plugin Deploy vous permet de déployer vers Tomcat, JBoss, et GlassFish. Et le plugin Deploy Websphere tente de s'occuper des particularités du serveur d'application IBM WebSphere.

Pour d'autres serveurs d'application, vous devez généralement intégrer le processus de déploiement dans vos scripts de build, ou de recourir à des scripts personnalisés, pour déployer votre application. De même, pour les autres langages, votre processus de déploiement variera, mais il impliquera souvent d'utiliser des scripts shell. Par exemple, pour une application Ruby on Rails, vous pouvez utiliser un outil tel que Capistrano or Chef, ou simplement un script shell. Pour une application PHP, un transfert FTP ou via scp peut suffire.

Regardons en premier certaines stratégies possibles pour déployer votre application Java vers un serveur d'application.

Lorsque l'application est déployée sur un serveur d'application en fonctionnement, cela est désigné sous le terme déploiement à chaud. Cela est généralement une façon rapide et efficace de mettre votre application en ligne. Cependant, en fonction de votre application et de votre serveur d'application, cette approche est connue pour causer des fuites de mémoire ou des problèmes de verrous sur des ressources. Les anciennes versions de Tomcat, par exemple, étaient particulièrement connues pour cela. Si vous rencontrez ce type de problème, vous pourriez avoir à forcer un redémarrage de l'application à chaque déploiement, ou à planifier un redémarrage de l'application chaque nuit sur votre serveur de test.

12.3.1. Déployer une application Java

Dans cette section, nous allons regarder un exemple montrant comment déployer votre application web Java ou JEE vers un serveur d'application tel que Tomcat, JBoss, ou GlassFish.

L'un des principes fondamentaux du déploiement automatisé est de réutiliser vos fichiers binaires déployables. Il est inefficace et potentiellement dangereux de réaliser un nouveau build pendant le processus de déploiement. En effet, imaginer que vous exécutez une série de tests unitaire et d'intégration sur une version donnée de votre application, avant de la déployer dans un environnement de test. Si vous reconstruisez le binaire avant de le déployer, le code source peut avoir changé depuis la version testée, et donc vous ne savez pas exactement ce que vous déployez.

Un processus plus efficace est de réutiliser des binaires générés par un build précédent. Par exemple, vous pourriez configurer une tâche de build de façon à exécuter les tests unitaires et d'intégration avant de déployer un fichier binaire (généralement un fichier WAR ou EAR). Vous pouvez faire cela très efficacement en utilisant le plugin Copy Artifact (voir Section 10.7.2, “Copier des artefacts”). Ce plugin vous permet de copier un artefact d'un autre espace de travail dans l'espace de travail de la tâche de build courante. Cela, lorsque combiné avec un trigger de build normal ou avec le plugin Build Promotion, vous permet de déployer précisément le fichier binaire que vous avez construit et testé dans la phase précédente.

Cette approche impose certaines contraintes dans la façon de construire votre application. En particulier, toute configuration spécifique à un environnement doit être externalisée hors de l'application; les connections JDBC et autres éléments de configuration ne doivent pas être inclus dans votre fichier WAR, mais plutôt, par exemple, être définis au moyen de JDNI ou dans un fichier de propriétés externe. Si ce n'est pas le cas, vous pourriez devoir construire à partir d'une révision données du gestionnaire de source, comme discuté pour Subversion dans Section 10.2.4, “Construire à partir d'un tag Subversion”.

12.3.1.1. Utiliser le plugin Deploy

Si vous déployez sur serveur Tomcat, JBoss ou GlassFish, l'outil le plus utile à votre disposition sera probablement le plugin Deploy. Ce plugin rend relativement simple l'intégration de ces plateformes dans votre processus de build Jenkins. Si vous déployez sur IBM Websphere, vous pouvez utiliser le plugin Websphere Deploy dans le même but.

Voyons ce plugin en action, en utilisant le séquenceur de build illustré dans Figure 12.1, “Une chaîne simple de déploiement automatisé”.



Figure 12.1. Une chaîne simple de déploiement automatisé

Ici, le build par défaut (**gameoflife-default**) exécute les tests unitaires et d'intégration, puis crée un déployable binaire sous la forme d'un fichier WAR. Le build des métriques (**gameoflife-metrics**) lance des contrôles supplémentaires sur les standards de codage et la couverture des tests. Si ces deux builds réussissent, l'application sera automatiquement déployée vers l'environnement de test par la tâche de build **gameoflife-deploy-to-test**.

Dans la tâche de build **gameoflife-deploy-to-test**, nous utilisons le plugin Copy Artifact plugin pour récupérer le fichier WAR généré dans la tâche **gameoflife-default** puis nous le copions dans l'espace de travail de la tâche courante (voir Figure 12.2, “Copier l'artefact binaire à déployer”).

The screenshot shows the 'Build' section of a Jenkins job configuration. Under the heading 'Copy artifacts from another project', there are several fields and options: 'Project name' is set to 'gameoflife-default/com.wakaleo.gameoflife\$gameoflife-web'; 'Which build' is set to 'Latest successful build' with a dropdown arrow; there is an unchecked checkbox for 'Stable build only'; 'Artifacts to copy' is set to '**/*.war'; 'Target directory' is empty; there are checkboxes for 'Flatten directories' (checked) and 'Optional' (unchecked); and a 'Delete' button is located at the bottom right.

Figure 12.2. Copier l'artefact binaire à déployer

Ensuite, nous utilisons le plugin Deploy pour déployer le fichier WAR sur le serveur de test. Bien sûr, il est généralement possible, et pas trop difficile, d'écrire à la main un tel script de déploiement. Dans certains cas, cela peut être votre seul recours. Toutefois, si un plugin Jenkins existe pour votre serveur d'application, cela peut simplifier considérablement les choses de l'utiliser. Si vous déployez sur Tomcat, JBoss, ou GlassFish, le plugin Deploy est susceptible de vous aider. Ce plugin utilise Cargo pour se connecter à votre serveur d'application et y déployer (ou redéployer) votre application. Il suffit sélectionner le type de serveur visé, spécifier son URL ainsi qu'un nom et mot de passe d'utilisateur ayant les droits de déploiement (voir Figure 12.3, “Déployer sur Tomcat avec le plugin Deploy”).

The screenshot shows the 'Deploy war/ear to a container' configuration page. It includes a checked checkbox for 'Deploy war/ear to a container', a 'WAR/EAR files' field set to '**/*.war', a 'Container' dropdown menu set to 'Tomcat 6.x', and three input fields: 'Manager user name' with 'admin', 'Manager password' with masked characters, and 'Tomcat URL' with 'http://localhost:8888'.

Figure 12.3. Déployer sur Tomcat avec le plugin Deploy

Cela est connu comme le déploiement à chaud, c'est à dire que l'application est déployée sur un serveur en cours de fonctionnement. Cela est généralement une façon rapide et efficace d'avoir votre application

en ligne, et devrait être votre solution préférée pour cela. Cependant, en fonction de votre application et de votre serveur, cette approche fut parfois source de fuites de mémoire ou de verrouillage de ressources. Les anciennes versions de Tomcat, par exemple, étaient bien connues pour cela. Si cela vous arrive, vous pourriez avoir à planifier des redémarrages après chaque déploiement, ou éventuellement planifier des redémarrages nocturnes du serveur d'application de votre environnement de test.

12.3.1.2. Redéployer une version spécifique

Lorsque vous déployez votre application de façon automatisée ou continuellement, il devient critique de bien identifier la version de l'application actuellement déployée. Il y a plusieurs façons de faire cela, qui varient essentiellement en fonction du rôle de Jenkins dans l'architecture de build et de déploiement.

Certaines équipes utilisent Jenkins comme la source de vérité, où les artefacts sont à la fois construits et gardés pour référence. Si vous stockez vos artefacts déployables dans Jenkins, il est alors parfaitement logique de déployer vos artefacts directement depuis votre instance Jenkins. Cela est facile à faire : dans la prochaine section nous verrons comment faire cela au moyen des plugins Copy Artifacts, Deploy, et Parameterized Trigger.

Alternativement, si vous gardez vos artefacts dans un dépôt d'entreprise tel que Nexus ou Artifactory, alors ce dépôt devrait être le point central de référence : les artefacts devraient être construits et déployés par Jenkins, et les déployer ensuite depuis ici. C'est typiquement le cas lorsque vous utilisez Maven comme votre outil de build. Des équipes utilisant Gradle ou Ivy sont aussi susceptibles d'utiliser cette approche. Les gestionnaires de dépôt tels que Nexus et Artifactory, particulièrement dans leur éditions commerciales, rendent cette stratégie plus aisée en offrant des fonctionnalités telles que la promotion de build et des dépôts intermédiaires qui aident à contrôler les versions des artefacts.

Voyons à présent comment vous pourriez implémenter chacune de ces stratégies en utilisant Jenkins.

12.3.1.3. Déployer une version depuis un build Jenkins précédent

Redéployer un artefact précédemment déployé dans Jenkins est relativement simple. Dans Section 12.3.1.1, “Utiliser le plugin Deploy”, nous avons vu comment utiliser les plugins Copy Artifacts et Deploy pour déployer un fichier WAR produit par une tâche de build précédente vers un serveur d'application. Ce dont nous avons besoin à présent est de laisser l'utilisateur spécifier la version à déployer, plutôt que de se contenter de déployer le dernier build.

Nous pouvons faire cela en utilisant le plugin Parameterized Trigger (voir Section 10.2, “Tâches de build paramétrées”). En premier lieu, nous ajoutons un paramètre à la tâche de construction, en utilisant le type de paramètre “Build selector for Copy Artifact” (voir Figure 12.4, “Ajouter un paramètre “Build selector for Copy Artifact””).

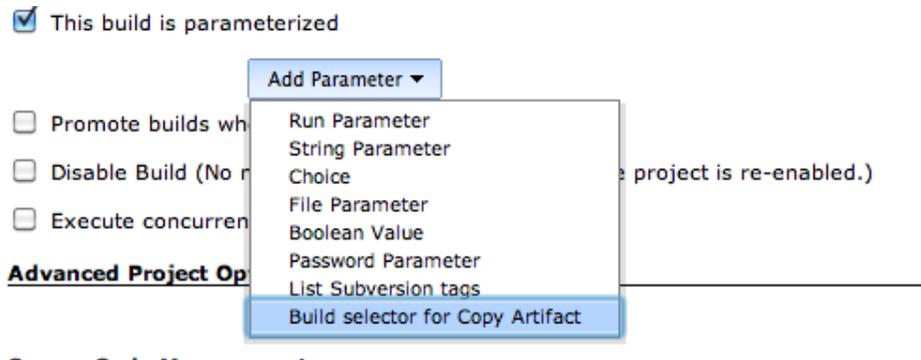


Figure 12.4. Ajouter un paramètre “Build selector for Copy Artifact”

Cela ajoute un nouveau paramètre à votre tâche de build (voir Figure 12.5, “Configurer le sélecteur de paramétrage de build”). Vous devez entrer un nom et une description courte. Le nom fourni sera utilisé comme une variable d’environnement pour les étapes suivantes du build.

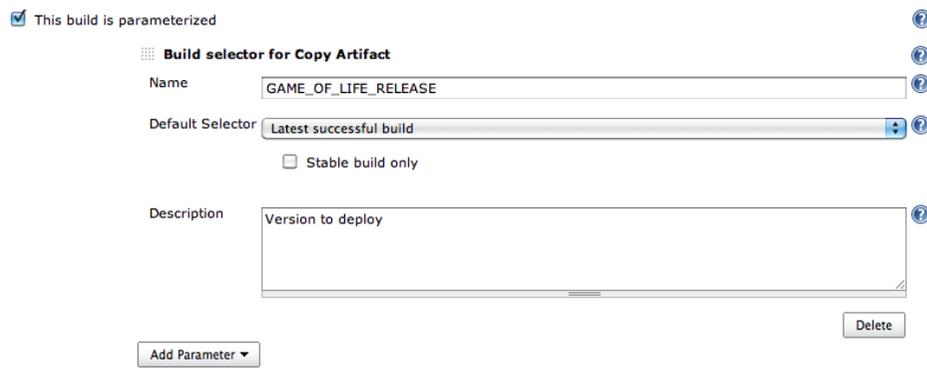


Figure 12.5. Configurer le sélecteur de paramétrage de build

Le sélecteur de paramétrage de build vous permet de choisir un build précédent de diverses façons, y compris le dernier build réussi, le build en amont ayant déclenché cette tâche de build ou un build spécifique. Toutes ces options seront offertes à l'utilisateur lorsqu'il ou elle déclenchera un build. Le sélecteur par défaut vous permet de spécifier laquelle de ces options sera proposé par défaut.

Lorsque l'utilisateur sélectionne une tâche de build particulière, le numéro de build sera également stocké dans la variable d’environnement pour l'utiliser dans les étapes du build. La variable d’environnement est appelé `COPYARTIFACT_BUILD_NUMBER_MY_BUILD_JOB`, où `MY_BUILD_JOB` est le nom de la tâche de build originelle (en lettres majuscules et avec les caractères autres que A–Z convertis en blanc soulignés). Par exemple, si nous copions un artefact avec le numéro de build 4 vers le project `gameoflife-default`, la variable d’environnement `COPYARTIFACT_BUILD_NUMBER_GAMEOFLIFE_DEFAULT` aura la valeur de 4.

La seconde partie de la configuration est de dire à Jenkins ce qu'il doit récupérer, et en provenance de quelle tâche de build. Dans la section Build de la configuration de notre projet, nous ajoutons une étape “Copy artifacts from another project”. Là nous spécifions le projet où l'artefact a été construit et archivé (gameoflife-default dans notre exemple). Vous devez aussi faire en sorte que Jenkins utilise le build spécifié dans le paramètre défini précédemment. Cela se fait en choisissant “Specified by a build parameter” dans l'option “Which build” et en fournissant le nom de variable donné plus tôt dans le sélecteur de paramétrage de build (voir Figure 12.6, “Spécifier où trouver les artefacts à déployer”). Ensuite, il suffit de configurer les artefacts à copier comme nous l'avons fait dans l'exemple précédent.

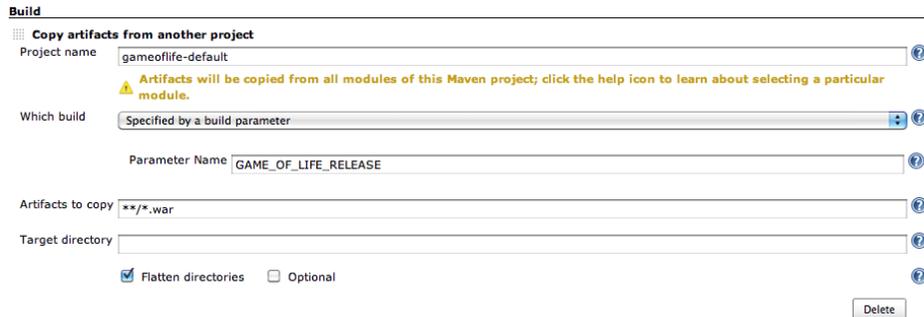


Figure 12.6. Spécifier où trouver les artefacts à déployer

Enfin, nous déployons l'artefact copié en utilisant le plugin Deploy, comme illustré dans Figure 12.3, “Déployer sur Tomcat avec le plugin Deploy”.

Voyons comment ce build fonctionne en pratique. Lorsque nous déclenchons un build manuellement, Jenkins va proposer une liste d'options vous permettant de choisir le build à redéployer (voir Figure 12.7, “Choix du build à redéployer”).

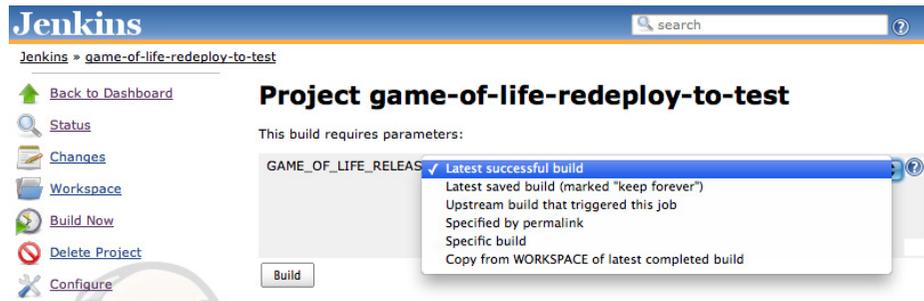


Figure 12.7. Choix du build à redéployer

La plupart de ces options sont explicites.

L'option “latest successful build” correspond au build le plus récent sans prendre en compte les builds échoués. Cette option va donc se contenter de déployer à nouveau la dernière version. Si vous utilisez

cette option, vous voudrez probablement cocher la case “Stable builds only”, qui exclura également tout build instable.

Si vous avez choisi de supprimer les anciens builds, vous serez capable d'indiquer que certaines tâches de build doivent être éternellement conservées (voir Section 5.3.1, “Options Générales”). Dans ce cas, vous pouvez choisir de déployer le “Latest saved build”.

Une option raisonnable pour une tâche de build automatisée à la fin d'un séquenceur de build est “Upstream build that triggered this job”. Ainsi, vous pouvez être sûr que vous allez déployer l'artefact qui a été généré (ou promu) par la précédente tâche de build, même si d'autres builds ont eu lieu depuis. Il est important de noter que, bien que ce type de build paramétré est souvent utilisé pour déployer manuellement un artefact spécifique, il peut aussi être utilisé efficacement au sein d'un build automatisé. S'il n'est pas déclenché manuellement, le build va simplement utiliser la valeur définie dans le champ “default selector”.

Vous pouvez aussi choisir l'option “Specified by permalink” (voir Figure 12.8, “Utiliser l'option “Specified by permalink””). Cela vous permet de choisir parmi un certain nombre de valeurs, telles que le dernier build, le dernier build stable, le dernier build réussi et ainsi de suite.

Project game-of-life-redeploy-to-test

This build requires parameters:

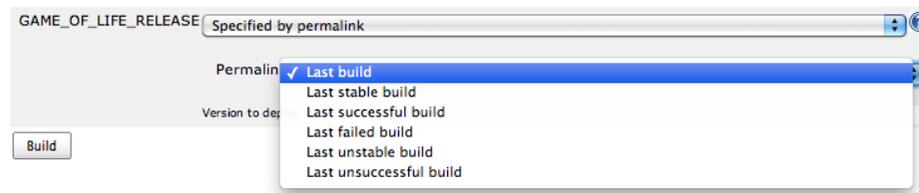


Figure 12.8. Utiliser l'option “Specified by permalink”

Cependant, si vous voulez redéployer une version particulière de votre application, une option plus utile est “Specific build” (voir Figure 12.9, “Utiliser un build spécifique”). Cette option vous demande de fournir un numéro de build particulier indiquant le build à déployer. Cela est la façon la plus flexible de redéployer une application, vous avez seulement besoin de connaître le numéro du build à redéployer, mais cela n'est généralement pas difficile à trouver en regardant l'historique du build de la tâche de build originelle.

Project game-of-life-redeploy-to-test

This build requires parameters:

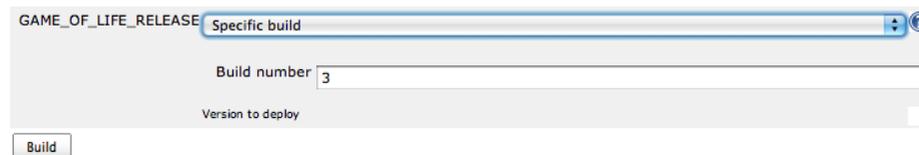


Figure 12.9. Utiliser un build spécifique

Cela est une façon pratique de déployer ou redéployer des artefacts de build Jenkins précédents. Cependant, dans certains cas, vous pourriez préférer utiliser un artefact stocké dans un dépôt d'entreprise comme Nexus ou Artifactory. Nous allons voir un exemple de cela dans la prochaine section.

12.3.1.4. Déployer une version depuis un dépôt Maven

De nombreuses organisations utilisent un gestionnaire de dépôt tel que Nexus ou Artifactory afin de stocker et partager des artefacts binaires tels que des fichiers JAR. Cette stratégie est communément utilisée avec Maven, mais également avec d'autres outils de build tels que Ant (couplé à Ivy ou aux Maven Ant Tasks) et Gradle. Dans un environnement d'Intégration Continue, les versions snapshots et délivrables sont construites sur votre serveur Jenkins, puis déployées sur votre gestionnaire de dépôt (voir Figure 12.10, "Utiliser un dépôt d'entreprise Maven"). A chaque fois qu'un développeur enregistre un changement dans le code source sur le système de contrôle des versions, Jenkins va prendre ces changements et construire de nouvelles versions snapshot des artefacts correspondant. Jenkins déploie alors ces artefacts snapshots sur le gestionnaire de dépôt d'entreprise, où ils peuvent être mis à disposition des autres développeurs de l'équipe ou d'autres équipes au sein de l'organisation. Nous avons discuté sur la façon de faire pour que Jenkins déploie automatiquement les artefacts Maven dans Figure 12.10, "Utiliser un dépôt d'entreprise Maven". Une approche similaire peut être suivie avec Gradle ou Ivy.

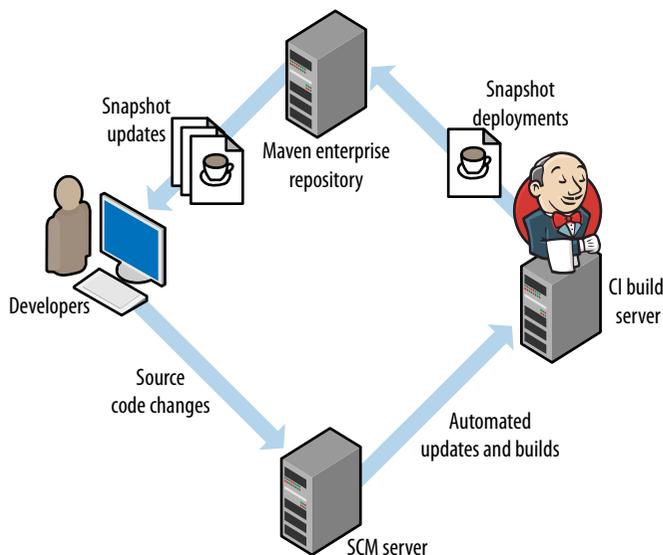


Figure 12.10. Utiliser un dépôt d'entreprise Maven

Les conventions Maven se basent sur un système bien défini d'identifiants de version, faisant la distinction entre des versions SNAPSHOTS et RELEASE. Les versions SNAPSHOT sont considérées comme des builds potentiellement instables des dernières sources, tandis que les versions RELEASE sont des versions officielles étant passées au travers d'un processus de livraison plus formel.

Typiquement, les artefacts SNAPSHOT sont réservés à un usage interne à l'équipe de développement, tandis que les versions RELEASE sont considérées comme prêtes pour des tests plus avancés.

Une approche très similaire peut être utilisée pour des artefacts déployables tels que les fichiers WAR ou EAR, qui sont construits et testés sur le serveur d'intégration continue, puis déployés automatiquement vers le dépôt d'entreprise, souvent au sein d'un séquenceur de build impliquant des tests automatisés et des contrôles qualité (voir Section 10.7, "Pipelines de build et promotions"). Les versions SNAPSHOT sont typiquement déployées sur un serveur de test pour des tests automatisés ou manuels, afin de déterminer si une version est prête pour être officiellement publiées.

La stratégie exacte utilisée pour déterminer quand une version est à créer et comment elle est déployée varie grandement d'une organisation à une autre. Par exemple, certaines équipes préfèrent un processus formel à la fin de chaque itération, avec un numéro de version bien défini et des notes de livraison correspondantes distribuées aux équipes d'assurance qualité pour des tests supplémentaires. Quand une version particulière obtient le feu vert de l'assurance qualité, elle peut être déployée en production. D'autres préfèrent utiliser un processus lean, déployant une nouvelle version lorsqu'un correctif ou une nouvelle fonctionnalité est prêt à être déployé. Si une équipe est particulièrement confiante dans ses tests automatisés et ses contrôles qualité, il est même possible d'automatiser complètement le processus, en générant et livrant une nouvelle version périodiquement (par exemple chaque nuit) ou à chaque fois qu'un changement a été réalisé.

Il y a de nombreuses façons d'implémenter ce type de stratégie. Dans le reste de cette section, nous verrons comment la réaliser via un projet Maven multimodule conventionnel. Notre projet de démonstration est une application web nommée `gameoflife`, constituée de trois modules : `gameoflife-core`, `gameoflife-services` et `gameoflife-web`. Le module `gameoflife-web` génère un fichier WAR incluant les JAR des deux autres modules. Le WAR est ce que nous voulons déployer :

```
tuatara:gameoflife johnsmart$ ls -l
total 32
drwxr-xr-x  16 johnsmart  staff   544 16 May 09:58 gameoflife-core
drwxr-xr-x   8 johnsmart  staff   272  4 May 18:12 gameoflife-deploy
drwxr-xr-x   8 johnsmart  staff   272 16 May 09:58 gameoflife-services
drwxr-xr-x  15 johnsmart  staff   510 16 May 09:58 gameoflife-web
-rw-r--r--@  1 johnsmart  staff 12182  4 May 18:07 pom.xml
```

Précédemment dans ce chapitre, nous avons vu comment utiliser le plugin Deploy pour déployer un fichier WAR généré par la tâche de build courante vers un serveur d'application. Ce que nous voulons maintenant c'est déployer une version arbitraire du fichier WAR vers un serveur d'application.

Dans Section 10.7.1, "Gestion des releases Maven avec le plugin M2Release", nous avons discuté de comment configurer Jenkins pour invoquer le plugin Maven Release afin de générer une version release formelle de l'application. La première étape du déploiement débute ici, nous supposons donc que cela a été configuré et que plusieurs versions releases ont déjà été déployées sur notre gestionnaire de dépôt d'entreprise.

La prochaine étape implique de créer un projet dédié pour le processus de déploiement. Ce projet sera un projet Maven standard.

La première chose à faire est de mettre en place un projet de déploiement dédié. Dans sa forme la plus simple, ce projet va simplement récupérer la version requise de votre fichier WAR depuis le gestionnaire de dépôt pour le déployer avec Jenkins. Dans le fichier `pom.xml` qui suit, nous utilisons le module `maven-war-plugin` pour récupérer la version spécifiée du fichier WAR `gameoflife-web`. Cette version est spécifiée dans la propriété `target.version` :

```
<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>com.wakaleo.gameoflife</groupId>
  <artifactId>gameoflife-deploy-with-jenkins</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>war</packaging>
  <dependencies>
    <dependency>
      <groupId>com.wakaleo.gameoflife</groupId>
      <artifactId>gameoflife-web</artifactId>
      <type>war</type>d
      <version>${target.version}</version>
    </dependency>
  </dependencies>
  <build>
    <plugins>
      <plugin>
        <artifactId>maven-war-plugin</artifactId>
        <configuration>
          <warName>gameoflife</warName>
          <overlays>
            <overlay>
              <groupId>com.wakaleo.gameoflife</groupId>
              <artifactId>gameoflife-web</artifactId>
            </overlay>
          </overlays>
        </configuration>
      </plugin>
    </plugins>
  </build>
  <properties>
    <target.version>RELEASE</target.version>
  </properties>
</project>
```

Ensuite, nous configurons un build Jenkins pour invoquer le fichier `pom.xml` en utilisant une valeur pour la propriété définie par l'utilisateur (voir Figure 12.11, “Déployer un artefact depuis un dépôt Maven”). Notez que nous avons défini la valeur par défaut à `RELEASE` pour qu'ainsi, par défaut, la version release la plus récente soit déployée. Sinon, l'utilisateur peut fournir le numéro de version de la version à déployer ou à redéployer.

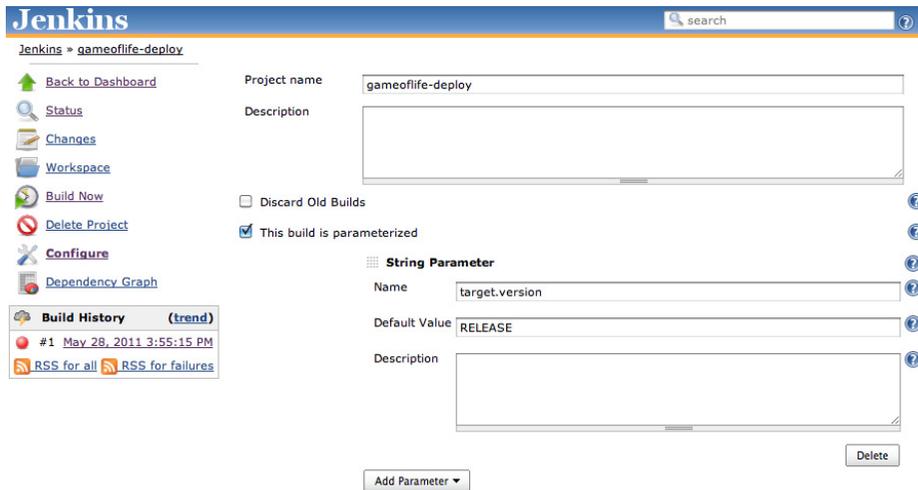


Figure 12.11. Déployer un artefact depuis un dépôt Maven

La suite de cette tâche de build récupère simplement le projet à déployer et appelle le but `mvn package`, puis déploie le fichier WAR en utilisant le plugin Deploy (voir Figure 12.12, “Préparer le WAR à déployer”). La propriété `target.version` sera automatiquement passée dans la tâche de build et utilisée pour déployer la bonne version.



Figure 12.12. Préparer le WAR à déployer

Des techniques similaires peuvent être utilisées pour d'autres types de projet. Si vous déployez sur un serveur d'application non supporté par le plugin Deploy, vous avez aussi la possibilité d'écrire un script spécifique dans le langage de votre choix et en faisant en sorte que Jenkins passe le numéro de version demandé comme décrit plus haut.

12.3.2. Déployer des applications à base de scripts telles Ruby et PHP

Déployer des projets utilisant des langages de script tels que PHP et Ruby est généralement plus simple que de déployer des applications Java, bien que les problématiques liées aux mises à jour de base de données soient similaires. En effet, bien souvent ces déploiement impliquent essentiellement de copier des fichiers vers un serveur distant. Pour obtenir les fichiers en premier lieu, vous avez le choix entre les copiers depuis l'espace de travail d'une autre tâche de build via l'option Copy Artifacts, ou de récupérer

directement le code source depuis le dépôt de code source, en utilisant au besoin une révision spécifique ou un tag comme décrit pour Subversion dans Section 10.2.4, “Construire à partir d'un tag Subversion” et pour Git dans Section 10.2.5, “Réaliser un build à partir d'un tag Git”. Une fois le code source présent dans votre espace de travail Jenkins, vous n'avez simplement qu'à le copier vers le serveur cible.

Un outil utile pour ce type de déploiement est la série de plugins Publish Over pour Jenkins (Publish Over FTP, Publish Over SSH, et Publish Over CIFS). Ces plugins fournissent une façon uniforme et flexible de déployer vos artefacts applicatifs vers d'autres serveurs via différents protocoles, incluant CIFS (pour les disques Windows partagés), FTP, et SSH/SFTP.

La configuration de chacun de ces plugins est similaire. Une fois les plugins installés, vous devez définir les configurations des hôtes, qui sont gérées ensemble dans l'écran principal de configuration. Vous pouvez créer autant de configurations d'hôtes que désiré. Elles apparaîtront dans une liste déroulante dans la page de configuration de la tâche.

La configuration des hôtes a des libellés explicites (voir Figure 12.13, “Configurer un hôte distant”). Le nom apparaîtra dans la liste déroulante des configurations de tâches de build. Vous pouvez configurer une authentification FTP au moyen d'un identifiant et mot de passe, ou bien, pour le SSH, une clé SSH ou un identifiant/mot de passe. Vous devez également fournir un répertoire existant sur le serveur distant qui servira de répertoire racine pour cette configuration. Dans les Options avancées, vous pouvez également configurer le port SSH et la durée d'expiration.

SSH

Jenkins SSH Key

Passphrase

Path to key

Key

Disable exec

SSH Servers

SSH Server

Name: Manuka

Hostname: 192.168.1.200

Username: john

Remote directory: /home/john/jenkins

Use password authentication, or use a different key

Port: 8922

Timeout (ms): 300000

Disable exec

Test Configuration

Delete

Add

Figure 12.13. Configurer un hôte distant

Une fois vos hôtes configurés, vous pouvez configurer vos tâches de build afin qu'elles déploient des artefacts vers ces hôtes. Vous pouvez faire cela en tant qu'étape de build (voir Figure 12.14, “Déployer des fichiers vers un hôte distant dans la section build”) ou en tant qu'action faisant suite au build (voir Figure 12.15, “Déployer des fichiers vers un hôte distant depuis les actions réalisées après le build”). Dans les deux cas, les options sont similaires.

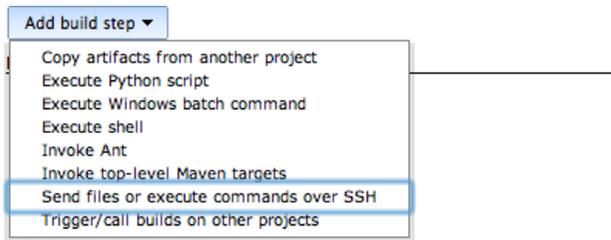


Figure 12.14. Déployer des fichiers vers un hôte distant dans la section build

En premier lieu, vous devez sélectionner l'hôte cible depuis la liste des hôtes que vous avez configuré dans la section précédente. Ensuite, vous indiquez les fichiers que vous voulez transférer. Cela se fait en définissant un ou plusieurs “Transfer sets.” Un Transfer set est un ensemble de fichiers (définis par une expression Ant) que vous déployez vers le dossier spécifié sur le serveur distant. Vous pouvez aussi fournir un préfix à supprimer, cela vous permet de laisser de côté des dossiers non nécessaires que vous ne voulez pas voir apparaître sur le serveur (tel que le chemin `target/site` dans l'exemple). Vous pouvez ajouter autant d'ensemble de transfert que requis, de façon à avoir les fichiers voulus sur le serveur distant. Le plugin propose également plusieurs options pour exécuter des commandes sur le serveur distant une fois le transfert fini ou pour exclure certains fichiers ou apllatir les dossiers.

Send build artifacts over SSH

SSH Publishers SSH Server

Name ?

Verbose output in console ?

Transfers Transfer Set

Source files ?

Remove prefix ?

Remote directory ?

Exec command ?

All of the transfer fields (except for Exec timeout) support substitution of [Jenkins environment variables](#)

Exclude files ?

Flatten files ?

Remote directory is a date format ?

Exec timeout (ms) ?

Figure 12.15. Déployer des fichiers vers un hôte distant depuis les actions réalisées après le build

12.4. Conclusion

Le Déploiement Automatisé, et ses formes plus avancées, le Déploiement Continu ou la Livraison Continue, peuvent être considérés comme le point culminant d'une infrastructure moderne d'Intégration Continue.

Dans ce chapitre, nous avons vu plusieurs techniques de Déploiement Automatisé, essentiellement axées autour de déploiements Java. Cependant, les principes généraux discutés ici sont valables pour n'importe quelle technologie. En effet, le processus de déploiement dans de nombreuses autres technologies, spécialement pour les langages de scripts tels Ruby et PHP, sont considérablement plus simple que leur équivalent Java, impliquant essentiellement de copier des fichiers vers le serveur de production. Ruby bénéficie aussi d'outils tels que Heroku et Capistrano pour aider à accomplir cette tâche.

Il y a plusieurs aspects importants à prendre en compte lors de la mise en place d'un Déploiement Automatisé. En premier lieu, le Déploiement Automatisé est le point final de votre architecture d'Intégration Continue : vous devez définir un séquenceur de build, de la compilation initiale et des tests unitaires vers des tests fonctionnels et d'acceptation automatisés ainsi que des contrôles de qualité du code, culminant au déploiement vers une ou plusieurs plate formes. Le degré de confiance dans votre séquenceur de build dépend largement du degré de confiance que vous avez en vos tests. En d'autres

termes, plus les tests sont douteux et limités, le plus tôt vous aurez à recourir à des tests manuels et à une intervention humaine.

Finallement, si possible, il est important de construire vos artefacts déployables en une fois seulement, et ensuite de les réutiliser dans les étapes suivantes, que ce soit pour les tests fonctionnels ou des déploiements vers différentes plateformes.

Chapter 13. Maintenir Jenkins

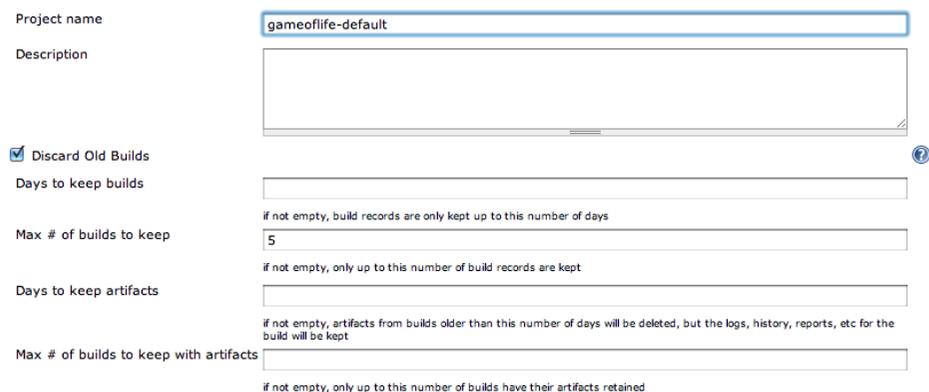
13.1. Introduction

Dans ce chapitre, nous allons discuter de quelques trucs et astuces que vous pourriez trouver utile lors de la maintenance d'une instance Jenkins conséquente. Nous regarderons des choses comme limiter et surveiller l'espace disque, comment donner assez de mémoire à Jenkins et comment archiver les tâches de build ou les migrer d'un serveur à un autre. Certains de ces sujets sont abordés ailleurs dans le livre, mais ici nous allons regarder ces éléments du point de vue d'un administrateur système.

13.2. Surveillance de l'espace disque

L'historique des builds prend de l'espace disque. De plus, Jenkins analyse les builds précédents lorsqu'il charge la configuration d'un projet. Ainsi, le chargement d'une tâche avec un millier de builds archivés prendra bien plus de temps qu'une tâche n'en n'ayant que 50. Si vous avez un gros serveur Jenkins avec des dizaines ou des milliers de tâches, le temps total est proportionnellement multiplié.

La façon la plus simple de plafonner l'utilisation de l'espace disque est probablement de limiter le nombre de builds qu'un projet conserve dans son historique. Cela se configure en cochant la case "Supprimer les anciens builds" en haut de la page de configuration d'un projet (voir Figure 13.1, "Suppression des anciens builds"). Si vous dites à Jenkins de ne garder que les 20 derniers builds, il commencera à effacer les plus vieux builds une fois ce nombre atteint. Vous pouvez limiter le nombre d'anciens builds conservés par un nombre de builds ou par date (par exemple les builds de moins de 30 jours). Jenkins fait cela intelligemment: il gardera toujours le dernier build réussi au sein de son historique, ainsi vous ne perdrez jamais votre dernier build réussi.



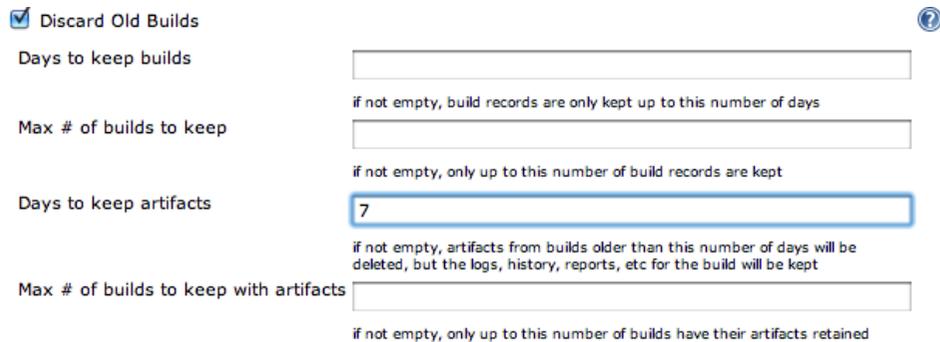
The image shows a screenshot of the Jenkins project configuration page. The 'Project name' field contains 'gameoflife-default'. The 'Description' field is empty. The 'Discard Old Builds' checkbox is checked. Below it, there are several input fields: 'Days to keep builds' (empty), 'Max # of builds to keep' (set to 5), 'Days to keep artifacts' (empty), and 'Max # of builds to keep with artifacts' (empty). Each of these fields has a small text label below it: 'if not empty, build records are only kept up to this number of days', 'if not empty, only up to this number of build records are kept', 'if not empty, artifacts from builds older than this number of days will be deleted, but the logs, history, reports, etc for the build will be kept', and 'if not empty, only up to this number of builds have their artifacts retained'. There is a help icon (question mark in a circle) to the right of the 'Discard Old Builds' checkbox.

Figure 13.1. Suppression des anciens builds

Le problème avec la suppression des anciens builds est que vous perdez l'historique des builds par la même occasion. Pourtant, Jenkins utilise cet historique pour réaliser différents graphiques sur les

résultats des tests et les métriques de build. Limiter le nombre de build conservé à 20, par exemple, implique que Jenkins affichera des graphiques contenant seulement 20 points. Cela peut être un peu limité. Cette sorte d'information peut être très utile aux développeurs. Il est souvent intéressant de pouvoir afficher l'évolution des métriques sur l'ensemble de la vie du projet, et pas seulement sur les 2 dernières semaines.

Heureusement, Jenkins a un mécanisme à même de rendre les développeurs et les administrateurs systèmes heureux. En général, les éléments prenant le plus de place sont les artefacts de build : fichiers JAR, WAR et ainsi de suite. L'historique de build en elle-même est principalement constituée de fichiers de log XML, qui ne prennent pas trop de place. Si vous cliquez sur le bouton "Avancé...", Jenkins vous offre la possibilité de supprimer les artefacts mais pas les données du build. Dans Figure 13.2, "Supprimer les anciens builds — options avancées", par exemple, nous avons configuré Jenkins pour qu'il garde les artefacts 7 jours au maximum. Cette option est vraiment pratique si vous avez besoin de limiter l'utilisation du disque tout en désirant fournir l'ensemble des métriques pour les équipes de développement.



The image shows a configuration interface for Jenkins with the following elements:

- Discard Old Builds (with a help icon to the right)
- Days to keep builds: [Empty text input field]
- if not empty, build records are only kept up to this number of days
- Max # of builds to keep: [Empty text input field]
- if not empty, only up to this number of build records are kept
- Days to keep artifacts: [7] (highlighted with a blue border)
- if not empty, artifacts from builds older than this number of days will be deleted, but the logs, history, reports, etc for the build will be kept
- Max # of builds to keep with artifacts: [Empty text input field]
- if not empty, only up to this number of builds have their artifacts retained

Figure 13.2. Supprimer les anciens builds — options avancées

N'hésitez pas à être drastique, en gardant un nombre maximal d'artefact assez faible. Souvenez vous que Jenkins gardera toujours le dernier build stable et le dernier réussi, quelque soit sa configuration. Ainsi, vous aurez toujours au moins le dernier build réussi (à moins bien sûr qu'il n'y ait pas encore eu de build réussi). Jenkins offre également de marquer un build particulier à "Conserver ce build sans limite de temps", afin que certains builds importants ne puissent être supprimés automatiquement.

13.2.1. Utiliser le plugin "Disk Usage"

Le plugin Disk Usage est un des plus utiles pour un administrateur Jenkins. Ce plugin conserve et reporte l'espace disque utilisé par vos projets. Il vous permet de repérer et corriger les projets qui utilisent trop d'espace.

Vous pouvez installer le plugin Disk Usage de la façon habituelle, depuis l'écran "Gestion des plugins". Après installation du plugin et redémarrage de Jenkins, le plugin Disk Usage enregistre la quantité d'espace disque utilisée par chaque projet. Il ajoute également un lien "Disk usage" sur la page

"Administrer Jenkins". Ce lien vous permet d'afficher la quantité totale d'espace utilisé par vos projets (voir Figure 13.3, "Voir l'utilisation d'espace disque").

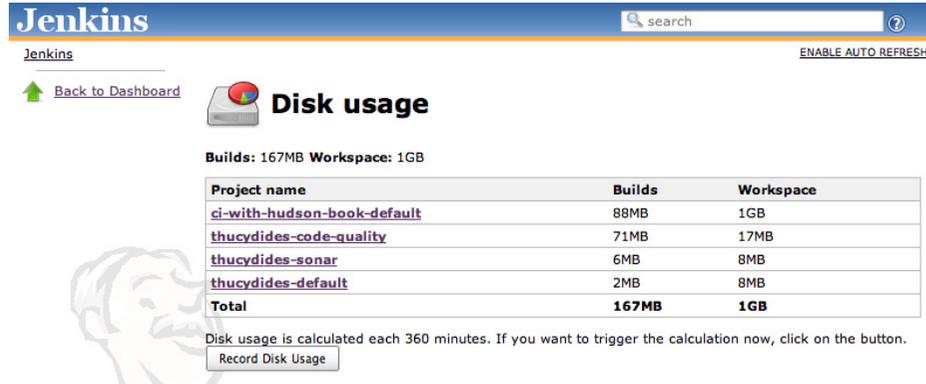


Figure 13.3. Voir l'utilisation d'espace disque

La liste est triée par utilisation totale d'espace disque, ainsi les projets utilisant le plus d'espace apparaissent en haut. La liste fournit deux valeurs par projet. La colonne "Builds" indique l'espace disque total utilisé par l'historique des builds, tandis que la colonne "Workspace" est l'espace disque utilisé pour construire le projet. Pour les projets en cours, l'espace utilisé par l'espace de travail tend à être relativement stable, tandis que la valeur pour l'historique des builds croît au cours du temps, parfois à une vitesse excessivement rapide, à moins que vous ne fassiez quelque chose. Vous pouvez garder sous contrôle l'espace disque utilisé par l'historique d'un projet en limitant le nombre de builds conservés et en faisant attention à quels artefacts sont conservés.

Pour se faire une idée sur la vitesse à laquelle l'espace disque est utilisé, vous pouvez aussi afficher l'espace disque utilisé par chaque projet au cours du temps. Pour faire cela, vous devez configurer le plugin sur la page "Configurer le système" (voir Figure 13.4, "Affichage de l'utilisation disque d'un projet").



Figure 13.4. Affichage de l'utilisation disque d'un projet

Cela va enregistrer et afficher combien d'espace disque vos projets consomment au cours du temps. Le plugin "Disk Usage" affiche un graphique de l'utilisation du disque au cours du temps (voir Figure 13.5, "Affichage de l'espace disque d'un projet au cours du temps") qui donne un bon aperçu de la vitesse à laquelle votre projet consomme l'espace disque, ou au contraire si l'espace utilisé est stable au cours du temps.

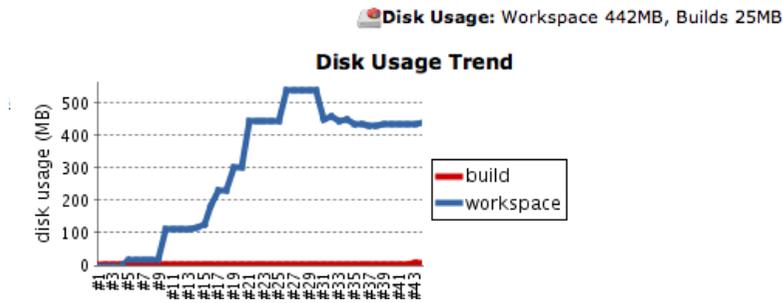


Figure 13.5. Affichage de l'espace disque d'un projet au cours du temps

13.2.2. Disk Usage et les projets Jenkins de type Apache Maven

Si vous utilisez les tâches de build Maven, il y a des détails supplémentaires que vous devriez connaître. Dans Jenkins, les tâches de build Maven archivent automatiquement, par défaut, les artefacts du build. Cela peut être différent de vos attentes.

Le problème est que ces artefacts SNAPSHOT prennent de la place, beaucoup même. Sur un projet actif, Jenkins est susceptible de réaliser plusieurs builds par heure. Stocker de façon permanente chacun des fichiers JAR générés pour chaque build peut être vraiment couteux. Le problème s'amplifie si vous avez des projets multimodules. En effet, Jenkins archive les artefacts générés pour chaque module.

En fait, si vous avez besoin d'archiver vos artefacts SNAPSHOT Maven, il est probablement plus avisé de les déployer directement dans votre gestionnaire de dépôt local. Nexus Pro, par exemple, peut être configuré pour faire cela. Artifactory peut être configuré pour supprimer les vieux artefacts SNAPSHOT.

Heureusement, vous pouvez configurer Jenkins pour réaliser cela. Allez dans la section "Build" de l'écran de configuration de votre tâche et cliquez sur le bouton "Avancé...". Des champs supplémentaires sont alors affichés, comme montré dans Figure 13.6, "Tâches de build Maven—options avancées".

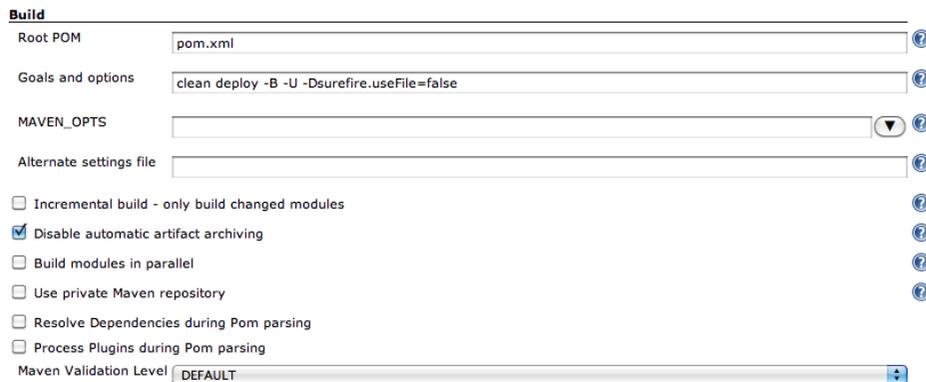


Figure 13.6. Tâches de build Maven—options avancées

Si vous cochez l'option "Désactive l'archivage automatique des artefacts", Jenkins ne stockera pas les fichiers Jar généré par les builds de votre projet. C'est une bonne façon de rendre heureux votre administrateur système.

Notez que parfois vous avez vraiment besoin d'archiver les artefacts Maven. Par exemple, cela s'avère souvent utiles quand vous implémentez un séquenceur de build (voir Section 10.7, "Pipelines de build et promotions"). Dans ce cas, vous pouvez toujours choisir, manuellement, les artefacts nécessaires, et alors utiliser l'option "Supprimer les anciens builds" pour définir la durée de conservation.

13.3. Surveiller la charge serveur

Jenkins inclut une surveillance des activités serveur. Sur l'écran "Administrer Jenkins", cliquez sur l'icône "Statistiques d'utilisation". Cela affiche un graphique de la charge serveur au cours du temps pour le noeud maître (voir Figure 13.7, "Statistiques de charge Jenkins"). Ce graphique contient trois métriques: nombre total d'exécuteurs, nombre d'exécuteurs occupés et longueur de la queue.

Le **nombre total d'exécuteurs** (la ligne bleue) inclut les exécuteurs sur les noeuds maître et esclaves. Ce chiffre peut varier quand les esclaves sont mis allumés ou éteints, et est un indicateur utile pour déterminer si la gestion dynamique des esclaves fonctionnent.

Le **nombre d'exécuteurs occupés** (la ligne rouge) indique le nombre d'exécuteurs en train de réaliser des builds. Vous devriez vous assurer que vous avez une capacité suffisante en réserve afin de supporter les pics de builds. Si tous vos exécuteurs sont occupés de façon permanente, vous devriez ajouter plus d'exécuteurs et/ou de noeuds esclaves.

La **longueur de la queue** (la ligne grise) est le nombre de tâches de build attendant d'être exécutées. Les tâches de build sont mises en attente lorsque tous les exécuteurs sont occupés. Cette métrique n'inclut pas les tâches en attente qu'un build en amont soit fini. Ainsi, elle donne une idée raisonnable du moment où votre serveur pourrait bénéficier de plus de capacités.

Jenkins

-  [New Job](#)
-  [Manage Jenkins](#)
-  [People](#)
-  [Build History](#)
-  [Project Relationship](#)
-  [Check File Fingerprint](#)
-  [Dependency Graph](#)

Build Queue
No builds in the queue.

Build Executor Status	
#	Status
1	Idle
2	Idle



Load statistics: Jenkins

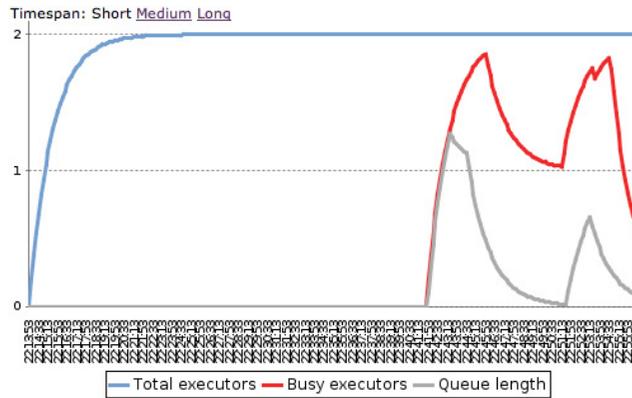


Figure 13.7. Statistiques de charge Jenkins

Vous pouvez obtenir un graphique similaire pour les noeuds esclaves, en utilisant le bouton "Statistiques d'utilisation" dans la page de détails du noeud esclave.

Une autre possibilité est d'installer le plugin Monitoring. Ce plugin utilise JavaMelody afin de réaliser des rapports HTML complets sur l'état de votre serveur de build. Les rapports incluent la charge système et processeur, les temps moyen de réponse et l'utilisation de la mémoire (voir Figure 13.8, "Le plugin Jenkins Monitoring"). Une fois ce plugin installé, vous pouvez accéder aux graphiques JavaMelody depuis la page "Administrer Jenkins", en utilisant les entrées du menu "Monitoring of Hudson/Jenkins master" ou "Hudson/Jenkins nodes".

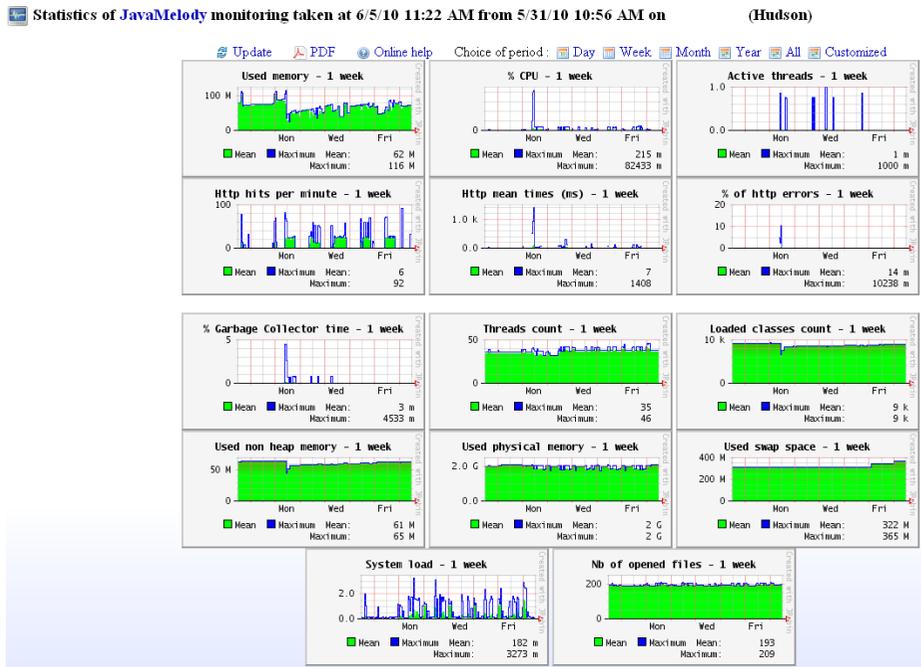


Figure 13.8. Le plugin Jenkins Monitoring

13.4. Sauvegarde de votre configuration

Sauvegarder vos données est une pratique universellement recommandée, et vos serveurs Jenkins ne devraient pas y faire exception. Par chance, sauvegarder Jenkins est relativement aisé. Dans cette section nous allons regarder plusieurs façons de réaliser cela.

13.4.1. Fondamentaux de la sauvegarde Jenkins

Dans la plus simple des configurations, il suffit de sauvegarder périodiquement votre dossier `JENKINS_HOME`. Il contient la configuration de toutes vos tâches de build, les configurations de vos nœuds esclaves et l'historique des builds. La sauvegarde peut se faire pendant que Jenkins tourne. Il n'y a pas besoin de couper votre serveur pendant la sauvegarde.

L'inconvénient de cette approche est que le dossier `JENKINS_HOME` peut contenir un volume très important de données (voir Section 3.13, “What’s in the Jenkins Home Directory”). Si cela devient un problème, vous pouvez en gagner un peu en ne sauvegardant pas les dossiers suivants, qui contiennent des données aisément recrées à la demande par Jenkins :

```
$JENKINS_HOME/war
Le fichier WAR éclaté
```

`$JENKINS_HOME/cache`

Outils téléchargés

`$JENKINS_HOME/tools`

Outils décompressés

Vous pouvez aussi être sélectif concernant ce que vous sauvegarder dans vos tâches de build. Le dossier `$JENKINS_HOME/jobs` contient la configuration de la tâche, l'historique des builds et les fichiers archivés pour chacun de vos builds. La structure d'un dossier de tâche de build est présentée dans Figure 13.9, “Le dossier des builds”.

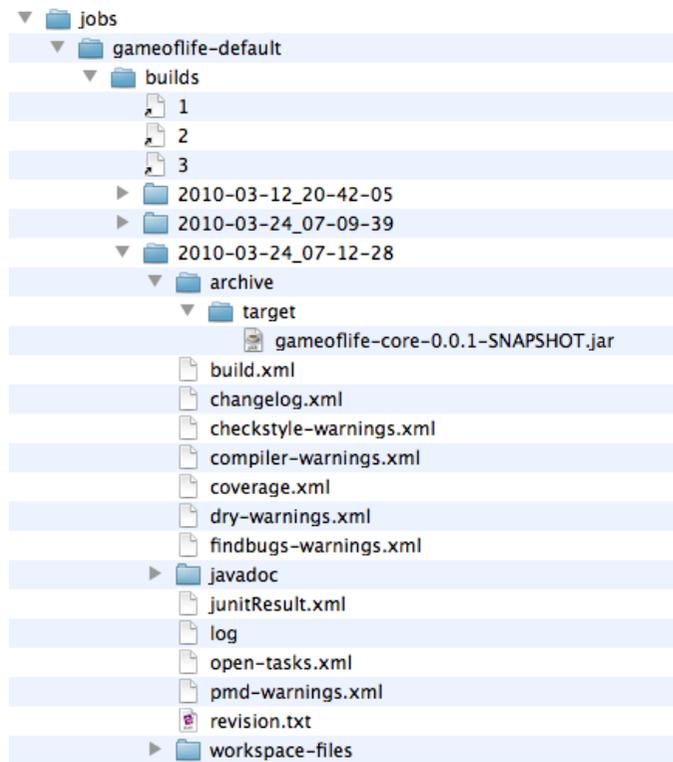


Figure 13.9. Le dossier des builds

Pour savoir comment optimiser vos sauvegardes Jenkins, vous devez comprendre comment sont organisés les dossiers de tâche de build. Au sein du dossier `jobs`, il y a un dossier pour chaque tâche de build. Ce dossier contient deux dossiers : `builds` et `workspace`. Il n'y a pas besoin de sauvegarder le dossier `workspace`, vu qu'il sera simplement restauré via une simple récupération si Jenkins constate son absence.

Au contraire, le dossier `builds` requiert plus d'attention. Il contient l'historique de vos résultats de build et de vos artefacts générés précédemment, avec un dossier horodaté pour chaque build précédent. Si vous

n'êtes pas intéressés par la restauration de votre historique des builds ou d'anciens artefacts, vous n'avez pas besoin de sauver ce dossier. Si vous l'êtes, continuez à lire! Dans chacun de ces dossiers, vous trouverez l'historique des builds (stockés sous la forme de fichiers XML, par exemple les résultats des tests JUnit) et les artefacts archivés. Jenkins utilise les fichiers texte et XML pour réaliser les graphiques affichés sur le tableau de bord des tâches de build. Le dossier `archive` contient les fichiers binaires ayant été générés et stockés par les builds précédents. Les binaires peuvent vous être importants ou non, mais ils peuvent prendre beaucoup de place. Aussi, si vous les excluez de vos sauvegardes, vous pourriez économiser beaucoup d'espace.

De même qu'il est sage de réaliser des sauvegardes fréquentes, il est également sage de tester votre procédure de sauvegarde. Avec Jenkins, cela est facile à faire. Les répertoires racine de Jenkins sont totalement portables, pour tester votre sauvegarde, il suffit donc de l'extraire dans un dossier temporaire et de lancer une instance Jenkins. Par exemple, imaginons que vous ayez extrait votre sauvegarde dans un dossier temporaire nommé `/tmp/jenkins-backup`. Pour tester cette sauvegarde, assigner le chemin du dossier temporaire à la variable `JENKINS_HOME` :

```
$  
export  
  JENKINS_HOME=/tmp/jenkins-backup
```

Puis démarrer Jenkins sur un port différent et regardez s'il fonctionne :

```
$  
java -jar jenkins.war --httpPort=8888
```

Vous pouvez maintenant voir Jenkins tourner sur ce port et vérifier que votre sauvegarde fonctionne correctement.

13.4.2. Utilisation du Backup Plugin

L'approche décrite dans la section précédente est suffisamment simple pour s'intégrer dans vos procédures normales de sauvegardes, mais vous pourriez préférer quelque chose de plus spécifique à Jenkins. Le plugin Backup (voir Figure 13.10, "Le plugin Jenkins Backup Manager") fournit une interface utilisateur simple que vous pouvez utiliser pour sauvegarder et restaurer vos configurations et données Jenkins.

Jenkins

Jenkins » Backup manager

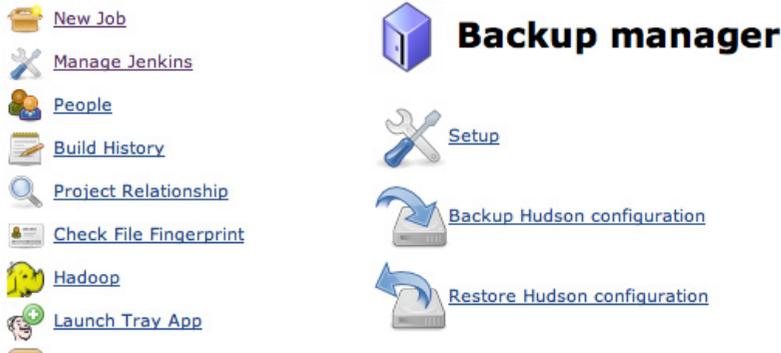


Figure 13.10. Le plugin Jenkins Backup Manager

Ce plugin vous permet de configurer et de lancer des sauvegardes tant des configurations de vos tâches de build que de votre historique des builds. L'écran "Configuration" vous donne un important contrôle sur les éléments à sauvegarder (voir Figure 13.11, "Configurer Jenkins Backup Manager"). Vous pouvez choisir de seulement sauvegarder les fichiers XML de configuration, ou de sauvegarder avec l'historique des builds. Vous pouvez aussi choisir de sauvegarder (ou non) les artefacts Maven automatiquement générés (dans de nombreux processus de build, ces artefacts sont disponibles dans votre Entreprise Repository Manager local). Vous pouvez aussi sauvegarder les espaces de travail des tâches (généralement non nécessaire, comme discuté plus haut) et toutes empreintes numériques générées.

Backup config files

Backup configuration

Hudson root directory `/Users/johnsmart/Projects/Demos/hudson-demo/jenkins-data`

Backup directory

Format

File name template

Custom exclusions

Verbose mode

Configuration files (.xml) only

No shutdown

Backup content

Backup job workspace

Backup builds history

Backup maven artifacts archives

Backup fingerprints

Figure 13.11. Configurer Jenkins Backup Manager

Vous pouvez déclencher une sauvegarde manuellement, depuis l'écran "Gestionnaire de backup" (accessible depuis l'écran "Administrer Jenkins"). La sauvegarde prend du temps et stoppe Jenkins durant cette période (à moins de désactiver cette option dans la configuration de la sauvegarde).

A l'heure où ces lignes sont écrites, il n'est pas possible de planifier cette opération depuis Jenkins, mais vous pouvez démarrer la sauvegarde en invoquant l'adresse correspondante (par exemple `http://localhost:8080/backup/backup` si votre instance Jenkins tourne localement sur le port 8080). Dans un environnement unix, par exemple, cela serait généralement fait via une tâche cron en utilisant un outil tel que `wget` ou `curl` pour démarrer la sauvegarde.

13.4.3. Des sauvegardes automatisées plus légères

Si tout ce que vous voulez sauvegarder est votre configuration des tâches de build, le plugin Backup Manager peut être considéré excessif. Une autre option est d'utiliser le plugin "Thin Backup", qui permet de planifier des sauvegardes complètes et incrémentales de vos fichiers de configuration. Vu qu'ils ne sauvegardent pas votre historique des builds ou vos artefacts, ces sauvegardes sont très rapides et peuvent ainsi être réalisées sans stopper le serveur pour les réaliser.

Tout comme le plugin Backup, ce plugin ajoute une icône dans la page "Administrer Jenkins". De là, vous pouvez configurer et planifier les sauvegardes de votre configuration, déclencher une sauvegarde immédiate ou restaurer une sauvegarde précédente. La configuration est sans détours (voir Figure 13.12, "Configurer le plugin Thin Backup"). Elle implique simplement de configurer des sauvegardes complètes et incrémentales en utilisant une syntaxe similaire à celle de cron et de fournir un dossier où stocker les sauvegardes.

Backup Configuration

Backup settings

Backup directory:

Backup schedule for full backups:

Backup schedule for differential backups:

Max number of stored full backups:

Clean up differential backups

Save

Figure 13.12. Configurer le plugin Thin Backup

Pour restaurer une configuration précédente, aller simplement à la page "Restore" et choisissez la date de la configuration que vous voulez réappliquer (voir Figure 13.13, "Restaurer une configuration précédente"). Une fois la configuration précédente restaurée, vous devez recharger la configuration Jenkins depuis le disque ou redémarrer Jenkins.

Restore Configuration



Figure 13.13. Restaurer une configuration précédente

13.5. Archiver les tâches de build

Une autre façon d'aborder la problématique de l'espace disque est de supprimer ou d'archiver les projets qui ne sont plus actifs. Archiver un projet vous permet de le restaurer aisément ultérieurement afin de consulter ses données ou artefacts. Archiver un projet est facile : il suffit de déplacer le répertoire de celui-ci en dehors du répertoire des tâches. Bien sûr, généralement, le répertoire de la tâche est en premier lieu compressé dans un fichier ZIP ou une tarball.

Dans l'exemple qui suit, nous voulons archiver le projet `tweeter-default`. En premier lieu, nous nous rendons dans le répertoire `jobs` de Jenkins et y créons une "tarball" (archive compressée) du répertoire `tweeter-default` se trouvant dans le répertoire des tâches.

```
$
cd $JENKINS_HOME/jobs
$
ls
gameoflife-default tweeter-default
$
tar czf tweeter-default.tgz
tweeter-default
$
ls
gameoflife-default tweeter-default tweeter-default.tgz
```

Si le projet n'est pas en cours de construction par Jenkins, vous pouvez alors le supprimer en toute sécurité et déplacer l'archive vers son lieu de stockage :

```
$
rm -Rf tweeter-default
$
mv tweeter-default.tgz
/data/archives/jenkins
```

Une fois cela réalisé, vous pouvez tout simplement recharger la configuration depuis le disque dans l'écran "Administrer Jenkins" (voir Figure 13.14, "Recharger la configuration à partir du disque"). Le projet archivé va disparaître instantanément de votre tableau de bord.

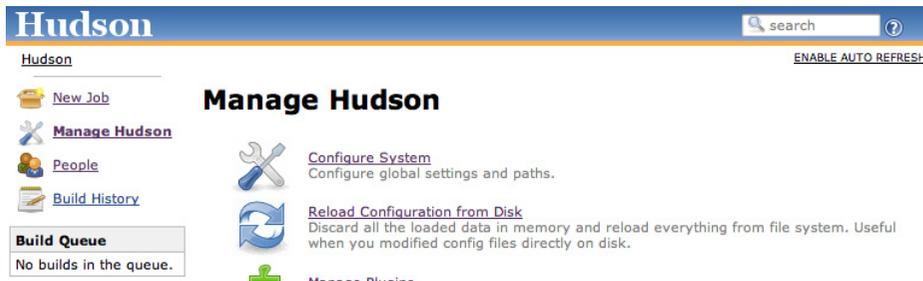


Figure 13.14. Recharger la configuration à partir du disque

Sur une machine Windows, vous pouvez faire exactement la même chose en créant un fichier ZIP du répertoire du projet.

13.6. Migrer les tâches de build

Il arrive que vous ayez à déplacer ou copier des tâches de build Jenkins d'une instance Jenkins à une autre sans copier toute la configuration de Jenkins. Par exemple, vous pourriez migrer vos tâches de build vers une instance Jenkins sur une machine flambant neuve, avec une configuration système différente de celle de la machine initiale. Ou vous pourriez être en train de restaurer une vieille archive de tâche de build.

Comme nous l'avons vu, Jenkins stocke toutes les données nécessaires pour un projet au sein d'un sous-répertoire du répertoire `jobs` dans votre répertoire racine de Jenkins. Ce sous-répertoire est aisé à identifier — il a le même nom que votre projet. D'ailleurs, cela explique pourquoi vos noms de projets ne doivent pas contenir d'espaces, spécialement si Jenkins tourne sous Unix ou Linux — cela rend la maintenance et les tâches d'administrations bien plus aisées si les noms de projet sont également des fichiers Unix correctement nommés.

Vous pouvez copier ou déplacer des tâches de build entre instances de projets assez simplement en copiant ou déplaçant les répertoires des tâches de build dans la nouvelle instance Jenkins. Le répertoire de la tâche du projet est autonome — il contient tant la configuration complète du projet que son historique des builds. Il est également possible de copier des répertoires de tâches de build dans une instance Jenkins en cours de fonctionnement. Toutefois, si vous effacez également ces répertoires du serveur originel, vous devriez en premier stopper ce dernier. Vous n'avez même pas besoin de redémarrer la nouvelle instance Jenkins pour voir le résultat de votre import, allez simplement à l'écran "Administrer Jenkins" et cliquez sur "Recharger la configuration à partir du disque". Cela chargera les nouvelles tâches de build et les rendra immédiatement visible sur le tableau de bord Jenkins.

Il y a quelques précautions à prendre toutefois. Si vous migrez vos tâches vers une installation toute fraîche de Jenkins, souvenez-vous d'installer ou de migrer les plugins de votre précédent serveur. Les plugins se trouvent dans le répertoire `plugins`, il suffit donc de simplement copier tout le contenu de ce répertoire dans le répertoire correspondant à votre nouvelle instance.

Bien sûr, vous pourriez être en train de migrer les tâches de build vers une nouvelle instance précisément parce que la configuration des plugins est problématique. Certains plugins peuvent parfois être un peu bogués, et vous pourriez vouloir une installation propre avec des plugins bien précis. Dans ce cas, vous pourriez avoir à retravailler certaines configurations de projets une fois ceux-ci importés.

L'explication de cela est simple. Quand vous utilisez un plugin dans un projet, le fichier `config.xml` du projet est mis à jour avec des champs de configuration spécifiques au plugin. Si, pour quelques raisons que ce soit, vous devez migrer des projets vers une installation Jenkins sans ces plugins, Jenkins ne comprendra pas les parties correspondantes de la configuration de ces projets. La même chose peut également arriver si les versions des plugins sont très différentes et que le format des données de configuration a changé.

Si vous migrez des tâches vers une instance Jenkins avec une configuration différente, il est également intéressant de garder un oeil sur les logs systèmes. Les configurations de plugin invalides sont généralement visibles via des alertes ou des exceptions. Bien que non fatal, ces messages d'erreurs indiquent souvent que le plugin ne fonctionnera pas comme attendu, voir pas du tout.

Jenkins offre des fonctionnalités utiles pour vous aider à migrer les configurations de vos projets. Si Jenkins trouve des données qu'il considère invalides, il vous le fera savoir. Sur l'écran "Administrer Jenkins", vous aurez des messages comme celui dans Figure 13.15, "Jenkins vous informe si vos données ne sont pas compatibles avec la version actuelle".

Manage Hudson



Figure 13.15. Jenkins vous informe si vos données ne sont pas compatibles avec la version actuelle

Plusieurs options s'offrent alors à vous. Vous pouvez laisser la configuration en l'état, par exemple si vous souhaitez revenir à une version précédente de votre instance Jenkins. Vous pouvez aussi laisser Jenkins ignorer les champs qu'il ne peut lire. Si vous retenez cette option, Jenkins affichera un écran avec plus de détails sur l'erreur et il pourra même, si vous le souhaitez, vous aider à nettoyer votre fichier de configuration (voir Figure 13.16, "Gestion de configuration périmée").

Manage Old Data

When there are changes in how data is stored on disk, Hudson uses the following strategy: data is migrated to the new structure when it is loaded, but the file is not resaved in the new format. This allows for downgrading Hudson if needed. However, it can also leave data on disk in the old format indefinitely. The table below lists files containing such data, and the Hudson version(s) where the data structure was changed. Sometimes errors occur while reading data (if a plugin adds some data and that plugin is later disabled, if migration code is not written for structure changes, or if Hudson is downgraded after it has already written data not readable by the older version). These errors are logged, but the unreadable data is then skipped over, allowing Hudson to startup and function properly.

Type **Name** **Version**

The form below may be used to resave these files in the current format. Doing so means a downgrade to a Hudson release older than the selected version will not be able to read the data stored in the new format. Note that simply using Hudson to create and configure jobs and run builds can save data that may not be readable by older Hudson releases, even when this form is not used. Also if any unreadable data errors are reported in the right side of the table above, note that this data will be lost when the file is resaved.

Eventually the code supporting these data migrations may be removed. Compatibility will be retained for at least 150 releases since the structure change. Versions older than this are in bold above, and it is recommended to resave these files.

No old data was found.

Unreadable Data

It is acceptable to leave unreadable data in these files, as Hudson will safely ignore it. To avoid the log messages at Hudson startup you can permanently delete the unreadable data by resaving these files using the button below.

Type	Name	Error
hudson.matrix.MatrixConfiguration	phoenix-multi-config-build/APP_SERVER=resin,DATABASE=mysql	NonExistentFieldException: No such field hudson.matrix.MatrixConfiguration.blockBuildWhenDownstreamBuilding
hudson.plugins.promoted_builds.PromotionProcess	phoenix-default/promotion/promote-to-test	NonExistentFieldException: No such field hudson.plugins.promoted_builds.PromotionProcess.blockBuildWhenDownstreamBuilding
hudson.matrix.MatrixConfiguration	phoenix-multi-config-build/APP_SERVER=tomcat,DATABASE=db2	NonExistentFieldException: No such field hudson.matrix.MatrixConfiguration.blockBuildWhenDownstreamBuilding
hudson.matrix.MatrixConfiguration	acceptance-test-suite/browser=chrome,os=OSX	NonExistentFieldException: No such field hudson.matrix.MatrixConfiguration.blockBuildWhenDownstreamBuilding
hudson.plugins.promoted_builds.PromotionProcess	gameoffie-deploy-to-uat/promotion/Deploy to Production	NonExistentFieldException: No such field hudson.plugins.promoted_builds.PromotionProcess.blockBuildWhenDownstreamBuilding
hudson.model.FreeStyleProject	game-of-life-freestyle-metrics	NonExistentFieldException: No such field hudson.model.FreeStyleProject.blockBuildWhenDownstreamBuilding
hudson.matrix.MatrixConfiguration	phoenix-multi-config-build/APP_SERVER=resin,D=mysql	NonExistentFieldException: No such field hudson.matrix.MatrixConfiguration.blockBuildWhenDownstreamBuilding

Figure 13.16. Gestion de configuration périmée

Cet écran donne plus de détails sur le projet contenant les données périmées ainsi que le message d'erreur obtenu. Cela offre plusieurs possibilités. Si vous êtes sûr de ne plus avoir besoin du plugin ayant originellement créé ces données, vous pouvez supprimer les champs fautifs en toute sécurité en cliquant sur le bouton "Discard Unreadable Data". Il est aussi possible que ces données appartiennent à un plugin qui n'a pas encore été installé sur cette instance Jenkins. Dans ce cas, installez le plugin et tout devrait aller bien. Enfin, vous pouvez toujours choisir de laisser les données redondantes et de vivre avec le message d'erreur, au moins jusqu'à ce que vous soyez sûr de ne plus avoir à migrer ces données vers l'ancien serveur.

Cependant, Jenkins ne détecte pas toujours toutes les erreurs et inconsistences. Il est toujours utile de garder un oeil sur les fichiers de log lorsque que vous migrez vos tâches de build. Par exemple, ce qui suit est un exemple réel d'un fichier de log Jenkins montrant ce qu'il peut arriver pendant une migration :

```
Mar 16, 2010 2:05:06 PM
  hudson.util.CopyOnWriteList$ConverterImpl unmarshal
WARNING: Failed to resolve class com.thoughtworks.xstream.mapper.CannotResolveClassException:
  hudson.plugins.cigame.GamePublisher :
  hudson.plugins.cigame.GamePublisher
  at
  com.thoughtworks.xstream.mapper.DefaultMapper.realClass(DefaultMapper.java:68)
  at
  com.thoughtworks.xstream.mapper.MapperWrapper.realClass(MapperWrapper.java:38)
  at
  com.thoughtworks.xstream.mapper.DynamicProxyMapper.realClass(DynamicProxyMapper.java:71)
  at
  com.thoughtworks.xstream.mapper.MapperWrapper.realClass(MapperWrapper.java:38)
```

Cette erreur nous informe que Jenkins ne peut trouver une classe appelée `hudson.plugins.cigame.GamePublisher`. En fait, l'installation cible n'a pas le plugin "CI Game". Et dans ce cas (comme cela arrive parfois), aucun message d'alerte n'est apparu sur la page

"Administrer Jenkins". Par conséquent, Jenkins n'a pas été capable de corriger le fichier de configuration par lui-même.

La solution la plus simple, dans ce cas, serait d'installer le plugin "CI Game". Mais que faire si on ne veut pas installer ce plugin? Nous pourrions laisser les fichiers de configuration en l'état, mais cela pourrait cacher des erreurs plus importantes ultérieurement. Il serait mieux de les nettoyer.

Dans ce cas, il faut inspecter et modifier les fichiers de configuration du projet à la main. Sur cette machine Unix, j'ai juste utilisé `grep` pour trouver tous les fichiers de configuration contenant "cigame" :

```
$  
cd $JENKINS_HOME/jobs  
$  
grep cigame */config.xml  
project-a/config.xml: <hudson.plugins.cigame.GamePublisher/>  
project-b/config.xml: <hudson.plugins.cigame.GamePublisher/>  
project-c/config.xml: <hudson.plugins.cigame.GamePublisher/>
```

Dans ces fichiers `config.xml`, j'ai trouvé une référence au plugin CI Game dans la section `<publishers>`, là où se trouve généralement les configurations des plugins réalisant des rapports :

```
<maven2-moduleset>  
...  
<publishers>  
<hudson.plugins.cigame.GamePublisher/>  
<hudson.plugins.claim.ClaimPublisher/>  
</publishers>  
...  
</maven2-moduleset>
```

Pour résoudre le problème, il suffit de supprimer la ligne incriminée:

```
<maven2-moduleset>  
...  
<publishers>  
<hudson.plugins.claim.ClaimPublisher/>  
</publishers>  
...  
</maven2-moduleset>
```

L'emplacement précis des données de configuration du plugin varie en fonction du plugin, mais en général les fichiers `config.xml` sont relativement lisibles. Les mettre à jour manuellement n'est pas trop compliqué.

Au final, migrer des tâches de build entre des instances Jenkins n'est pas si dur. Vous avez juste besoin de connaître quelques astuces pour les cas particuliers, et, si vous savez où regarder, Jenkins fournit de beaux outils pour rendre le processus aisé.

13.7. Conclusion

Dans ce chapitre, nous avons abordé certains éléments à connaître afin d'administrer votre serveur Jenkins, notamment la surveillance de l'espace disque et de la charge du serveur, comment sauvegarder vos tâches de build et les fichiers de configuration, ainsi que comment migrer vos tâches de build en toute sécurité.

Appendix A. Automatiser vos tests unitaires et d'intégration

A.1. Automatiser vos tests avec Maven

Maven est un outil de build open source populaire dans le monde Java, qui fait usage de pratiques telles que les déclarations de dépendances, les répertoires et cycles de vie de build standards, et la convention préférée à la déclaration pour encourager des scripts de build propres, maintenables et de bonne qualité. L'automatisation des tests est fortement prise en charge par Maven. Les projets Maven utilisent une structure de dossiers standard : les tests unitaires seront automatiquement recherchés dans un dossier nommé (par défaut) `src/test/java`. Il y a quelques petits réglages supplémentaires : en ajoutant juste une dépendance à un (ou plusieurs) framework(s) de test utilisé(s) par vos tests, Maven détectera et exécutera automatiquement les tests JUnit, TestNG, ou même Plain Old Java Objects (POJO) contenus dans cette structure de dossiers.

Avec Maven, vous lancez vos tests unitaires en invocalant la phase `test` du cycle de vie, comme présenté ici :

```
$ mvn test
[INFO] Scanning for projects...
[INFO] -----
[INFO] Building Tweeter domain model
[INFO]    task-segment: [test]
[INFO] -----
...
-----
T E S T S
-----
Running com.wakaleo.training.tweeter.domain.TagTest
Tests run: 13, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.093 sec
Running com.wakaleo.training.tweeter.domain.TweeterTest
Tests run: 3, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.021 sec
Running com.wakaleo.training.tweeter.domain.TweeterUserTest
Tests run: 4, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.055 sec
Running com.wakaleo.training.tweeter.domain.TweetFeeRangeTest
Tests run: 10, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.051 sec
Running com.wakaleo.training.tweeter.domain.HamcrestTest
Tests run: 8, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.023 sec

Results :

Tests run: 38, Failures: 0, Errors: 0, Skipped: 0
```

En plus d'exécuter vos tests, et de mettre le build en échec dès qu'un test échoue, Maven va produire un ensemble de rapports de tests (encore, par défaut) dans le dossier `target/surefire-reports`, dans les formats XML et texte. Pour nos objectifs d'intégration continue, ce sont les fichiers XML qui nous

intéressent puisque Jenkins est en mesure de comprendre et d'analyser ces fichiers pour ses rapports d'intégration :

```
$ ls target/surefire-reports/*.xml
target/surefire-reports/TEST-com.wakaleo.training.tweeter.domain.HamcrestTest.xml
target/surefire-reports/TEST-com.wakaleo.training.tweeter.domain.TagTest.xml
target/surefire-reports/TEST-com.wakaleo.training.tweeter.domain.TweetFeeRangeTest.xml
target/surefire-reports/TEST-com.wakaleo.training.tweeter.domain.TweeterTest.xml
target/surefire-reports/TEST-com.wakaleo.training.tweeter.domain.TweeterUserTest.xml
```

Maven définit deux phases de tests distinctes : les tests unitaires et les tests d'intégration. Les tests unitaires doivent être rapides et légers, en fournissant une quantité importante de retours de test en un minimum de temps. Les tests d'intégration sont plus lents et lourds, et requièrent souvent que l'application soit construite et déployée sur un serveur (potentiellement embarqué) pour supporter des tests plus complets. Ces deux types de tests sont importants, et pour un environnement d'Intégration Continue bien conçu, il est nécessaire de bien les distinguer. Le build doit assurer que tous les tests unitaires sont lancés en premier - si un test unitaire échoue, les développeurs devraient en être notifiés très rapidement. Le lancement des tests d'intégration, lents et plus lourds, ne vaut le coup que si tous les tests unitaires passent.

Avec Maven, les tests d'intégration sont exécutés pendant la phase `integration-test` du cycle de vie, que vous pouvez invoquer en lançant `mvn integration-test` ou (plus simplement) `mvn verify`. Pendant cette phase, il est facile de configurer Maven pour démarrer votre application web sur un serveur Jetty embarqué, ou pour packager et déployer votre application sur un serveur de test, par exemple. Vos tests d'intégration peuvent ensuite être exécutés sur l'application en marche. Cependant, la partie délicate, est de dire à Maven comment distinguer les tests unitaires des tests d'intégration, afin qu'ils ne soient exécutés que si une version fonctionnelle de l'application est disponible.

Il y a plusieurs manières d'y parvenir, mais au moment de l'écriture il n'existe pas d'approche standard officielle utilisée à travers tous les projets Maven. Une stratégie simple est d'utiliser les conventions de nommage : tous les tests d'intégration peuvent se terminer par "IntegrationTest", ou être placés dans un package particulier. La classe suivante utilise une convention de la sorte :

```
public class AccountIntegrationTest {

    @Test
    public void cashWithdrawalShouldDeductSumFromBalance() throws Exception {
        Account account = new Account();
        account.makeDeposit(100);
        account.makeCashWithdraw(60);
        assertThat(account.getBalance(), is(40));
    }
}
```

Avec Maven, les tests sont configurés via le plugin `maven-surefire-plugin`. Pour assurer que Maven lance ces tests seulement pendant la phase `integration-test`, vous pouvez configurer ce plugin comme présenté ici :

```
<project>
```

```

...
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <skip>true</skip>❶
      </configuration>
      <executions>
        <execution>❷
          <id>unit-tests</id>
          <phase>test</phase>
          <goals>
            <goal>test</goal>
          </goals>
          <configuration>
            <skip>false</skip>
            <excludes>
              <exclude>**/*IntegrationTest.java</exclude>
            </excludes>
          </configuration>
        </execution>
        <execution>❸
          <id>integration-tests</id>
          <phase>integration-test</phase>
          <goals>
            <goal>test</goal>
          </goals>
          <configuration>
            <skip>false</skip>
            <includes>
              <include>**/*IntegrationTest.java</include>
            </includes>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </build>
...

```

- ❶ Saute tous les tests par défaut — ceci désactive la configuration par défaut des tests pour Maven.
- ❷ Pendant la phase des tests unitaires, lance les tests en excluant les tests d’intégration.
- ❸ Pendant la phase des tests d’intégration, lance les tests mais en incluant seulement les tests d’intégration.

Ceci assure que les tests d’intégration seront ignorés pendant la phase des tests unitaires, et exécutés seulement pendant la phase des tests d’intégration.

Si vous ne voulez pas ajouter de contrainte non souhaitée sur les noms de vos classes de test, vous pouvez utiliser les noms de package plutôt. Dans le projet illustré dans Figure A.1, “Un projet contenant des classes de tests nommées librement”, tous les tests fonctionnels ont été placés dans un package nommé **webtests**. Il n’y a aucune contrainte sur les noms des tests, mais nous utilisons des Page Objects pour

modéliser l'interface de notre application, donc nous nous assurons aussi qu'aucune classe du package **pages** (à l'intérieur du package **webtests**) ne soit considéré comme un test.

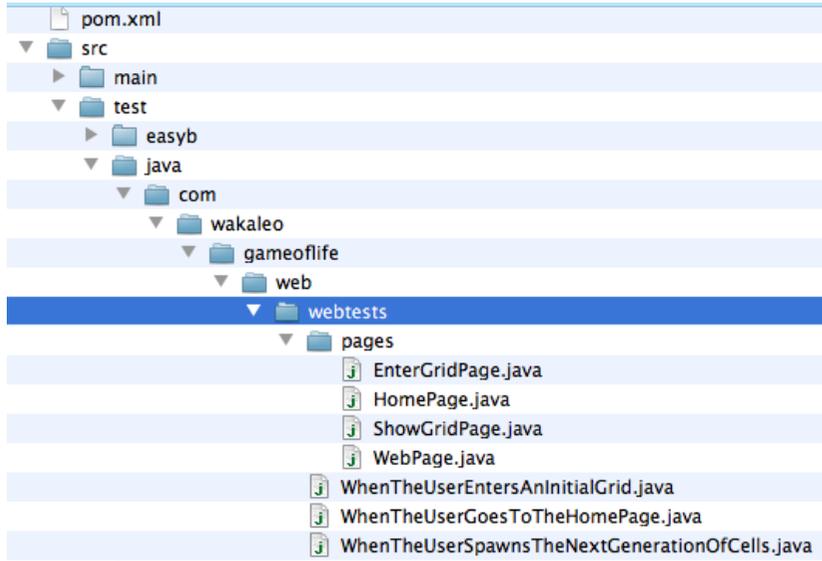


Figure A.1. Un projet contenant des classes de tests nommées librement

Avec Maven, nous pouvons faire cela avec la configuration suivante :

```
<plugin>
  <artifactId>maven-surefire-plugin</artifactId>
  <configuration>
    <skip>>true</skip>
  </configuration>
</plugin>
<executions>
  <execution>
    <id>unit-tests</id>
    <phase>test</phase>
    <goals>
      <goal>test</goal>
    </goals>
    <configuration>
      <skip>>false</skip>
      <excludes>
        <exclude>**/webtests/*.java</exclude>
      </excludes>
    </configuration>
  </execution>
  <execution>
    <id>integration-tests</id>
    <phase>integration-test</phase>
    <goals>
      <goal>test</goal>
    </goals>
    <configuration>
```

```

        <skip>false</skip>
        <includes>
            <include>**/webtests/*.java</include>
        </includes>
        <excludes>
            <exclude>**/pages/*.java</exclude>
        </excludes>
    </configuration>
</execution>
</executions>
</plugin>

```

TestNG a actuellement un support plus flexible des groupes de test que JUnit. Si vous utilisez TestNG, vous pouvez identifier vos tests d'intégration en utilisant les TestNG Groups. Avec TestNG, les classes et les méthodes de test peuvent être étiquetées en utilisant l'attribut `groups` de l'annotation `@Test`, comme présenté ici :

```

@Test(groups = { "integration-test" })
public void cashWithdrawalShouldDeductSumFromBalance() throws Exception {
    Account account = new Account();
    account.makeDeposit(100);
    account.makeCashWithdraw(60);
    assertThat(account.getBalance(), is(40));
}

```

En utilisant Maven, vous pouvez vous assurer que ces tests sont seulement lancés pendant la phase des tests d'intégration avec la configuration suivante :

```

<project>
...
<build>
  <plugins>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <skip>true</skip>
      </configuration>
      <executions>
        <execution>
          <id>unit-tests</id>
          <phase>test</phase>
          <goals>
            <goal>test</goal>
          </goals>
          <configuration>
            <skip>false</skip>
            <excludedGroups>integration-tests</excludedGroups>❶
          </configuration>
        </execution>
      </executions>
    </plugin>
    <plugin>
      <artifactId>maven-surefire-plugin</artifactId>
      <configuration>
        <skip>false</skip>
        <excludedGroups>integration-tests</excludedGroups>❷
      </configuration>
      <executions>
        <execution>
          <id>integration-tests</id>
          <phase>integration-test</phase>
          <goals>

```

```

    <goal>test</goal>
  </goals>
  <configuration>
    <skip>false</skip>
    <groups>integration-tests</groups>❶
  </configuration>
</execution>
</executions>
</plugin>
...

```

- ❶ Ne lance pas le groupe `integration-tests` pendant la phase `test`.
- ❷ Lance seulement les tests du groupe `integration-tests` pendant la phase `integration-test`.

Il est souvent intéressant de lancer les tests en parallèle dès que possible, puisque cela peut accélérer vos tests de façon significative (voir Section 6.9, “A l'aide ! Mes tests sont trop lents !”). Les tests en parallèle sont particulièrement efficaces avec des tests lents qui utilisent beaucoup d'accès E/S, disque ou réseau (comme les tests web), ce qui est pratique, puisque ce sont précisément les types de tests que nous voulons généralement accélérer.

TestNG propose un bon support des tests en parallèle. Par exemple, avec TestNG, vous pouvez configurer vos méthodes de tests pour qu'elles se lancent en parallèle sur dix threads concurrents comme ceci :

```

<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-surefire-plugin</artifactId>
  <version>2.5</version>
  <configuration>
    <parallel>methods</parallel>
    <threadCount>10</threadCount>
  </configuration>
</plugin>

```

Depuis JUnit 4.7, vous pouvez aussi lancer vos tests JUnit en parallèle en utilisant une configuration similaire. En fait, la configuration présentée ci-dessus fonctionnera pour JUnit 4.7 et suivant.

Vous pouvez aussi régler le paramètre de configuration `<parallel>` à la valeur `classes` au lieu de `methods`, ce qui tentera de lancer les classes de test en parallèle, plutôt que pour chaque méthode. Cela peut être plus ou moins rapide, en fonction du nombre de classes de test que vous avez, mais peut être plus sûr pour certains cas de test non conçus avec la concurrence à l'esprit.

Les résultats peuvent varier, donc vous feriez bien d'expérimenter les nombres pour obtenir les meilleurs résultats.

A.2. Automatiser vos tests avec Ant

Mettre en place des tests automatisés avec Ant est aussi relativement facile, bien que cela requiert un peu plus de plomberie qu'avec Maven. En particulier, Ant ne fournit pas directement les librairies JUnit

et les tâches Ant adaptées, donc il faut les installer soi-même quelque part. L'approche la plus portable est d'utiliser un outil de Gestion de dépendances tel qu'Ivy, ou de placer les fichiers JAR correspondants dans un répertoire à l'intérieur de la structure de votre projet.

Pour lancer les tests avec Ant, vous appelez la tâche `<junit>`. Une configuration typique adaptée à Jenkins est présentée dans cet exemple :

```
<property name="build.dir" value="target" />
<property name="java.classes" value="${build.dir}/classes" />
<property name="test.classes" value="${build.dir}/test-classes" />
<property name="test.reports" value="${build.dir}/test-reports" />
<property name="lib" value="${build.dir}/lib" />

<path id="test.classpath">❶
  <pathelement location="/jenkins-guide-complet/hudsonbook-content-fr/tools/junit/*.jar" />
  <pathelement location="${java.classes}" />
  <pathelement location="${lib}" />
</path>

<target name="test" depends="test-compile">
  <junit haltonfailure="no" failureproperty="failed">❷
    <classpath>❸
      <path refid="test.classpath" />
      <pathelement location="${test.classes}" />
    </classpath>
    <formatter type="xml" />❹
    <batchtest fork="yes" forkmode="perBatch"❺ todir="${test.reports}">
      <fileset dir="${test.src}">❻
        <include name="**/*Test*.java" />
      </fileset>
    </batchtest>
  </junit>
  <fail message="TEST FAILURE" if="failed" />❼
</target>
```

- ❶ Nous devons mettre en place un classpath contenant les fichiers JAR `junit` et `junit-ant`, sans oublier les classes de l'application et toute autre dépendance de l'application pour la compilation et le lancement.
- ❷ Les tests en eux-mêmes sont lancés ici. L'option `haltonfailure` est utilisée pour faire échouer le build immédiatement dès qu'un test échoue. Dans un environnement d'Intégration Continue, ce n'est pas exactement ce qu'on veut, puisque nous devons également avoir les résultats des tests suivants. Nous avons donc positionné cette valeur à `no` et utilisé l'option `failureproperty` pour forcer l'échec du build dès que tous les tests sont terminés.
- ❸ Le classpath doit contenir les bibliothèques JUnit, les classes de votre application et leurs dépendances, et vos classes de test compilées.
- ❹ La tâche Junit de Ant peut produire des rapports aux formats texte et XML, mais pour Jenkins, nous avons seulement besoin de ceux en XML.
- ❺ L'option `fork` lance vos test dans une JVM séparée. C'est généralement une bonne idée, puisque cela peut éviter les problèmes de type classloader liés à des conflits avec les bibliothèques propres à

Ant. Néanmoins, le comportement par défaut de la tâche Junit de Ant est de créer une nouvelle JVM pour chaque test, ce qui ralentit significativement les tests. L'option `perBatch` est plus intéressante, puisqu'elle crée une seule nouvelle JVM pour chaque batch de tests.

- ⑥ Vous définissez les tests que vous voulez lancer au sein d'un élément `fileset`. Cela permet une grande souplesse, et rend simple la définition d'autres objectifs pour différents sous-ensembles de tests (intégration, web, et autres).
- ⑦ Force l'échec du build après la fin des tests, si l'un d'entre eux a échoué.

Si vous préférez TestNG, Ant est évidemment bien supporté également. En utilisant TestNG pour l'exemple précédent, vous pourriez faire quelque chose comme ceci :

```
<property name="build.dir" value="target" />
<property name="java.classes" value="${build.dir}/classes" />
<property name="test.classes" value="${build.dir}/test-classes" />
<property name="test.reports" value="${build.dir}/test-reports" />
<property name="lib" value="${build.dir}/lib" />

<path id="test.classpath">
  <pathelement location="${java.classes}" />
  <pathelement location="${lib}" />
</path>

<taskdef resource="testngtasks" classpath="lib/testng.jar"/>

<target name="test" depends="test-compile">
  <testng classpathref="test.classpath"
    outputDir="${testng.report.dir}"
    haltonfailure="no"
    failureproperty="failed">
    <classfileset dir="${test.classes}">
      <include name="**/*Test*.class" />
    </classfileset>
  </testng>
  <fail message="TEST FAILURE" if="failed" />
</target>
```

TestNG est une librairie de test très flexible, et la tâche TestNG a beaucoup plus d'options que ça. Par exemple, pour lancer seulement les tests définis comme faisant partie du groupe "integration-test" que nous avons vu précédemment, nous pourrions faire ça :

```
<target name="integration-test" depends="test-compile">
  <testng classpathref="test.classpath"
    groups="integration-test"
    outputDir="${testng.report.dir}"
    haltonfailure="no"
    failureproperty="failed">
    <classfileset dir="${test.classes}">
      <include name="**/*Test*.class" />
    </classfileset>
  </testng>
  <fail message="TEST FAILURE" if="failed" />
</target>
```

Ou pour lancer les tests en parallèle, en utilisant quatre threads concurrents, vous pourriez faire ça :

```
<target name="integration-test" depends="test-compile">
  <testng classpathref="test.classpath"
    parallel="true"
    threadCount=4
    outputDir="${testng.report.dir}"
    haltonfailure="no"
    failureproperty="failed">
    <classfileset dir="${test.classes}">
      <include name="**/*Test*.class" />
    </classfileset>
  </testng>
  <fail message="TEST FAILURE" if="failed" />
</target>
```


Index

A

- Active Directory, Microsoft, comme domaine de sécurité, 186
 - administrateur
 - pour la base de données utilisateurs interne de Jenkins, 181
 - pour la sécurité basée sur une matrice, 192
 - agents de build
 - surveillance, 332
 - agents de constructions
 - configurer de multiples versions du JDK, 76
 - agréger résultats tests, 313-314
 - Amazon EC2 plugin, 335
 - Amazon Machine Image (AMI), 334
 - Amazon Web Services (AWS), 334
 - AMI (Amazon Machine Image), 334
 - analyse (see métriques de couverture de code; métriques de qualité de code; tests)
 - annuaire LDAP, comme domaine de sécurité, 185-186
 - Ant, 78-79
 - automatiser les tests, 384-387
 - code quality metrics
 - with Checkstyle, 240
 - configurer, 78-79
 - dans les étapes de build Freestyle, 111-111
 - installer, 79
 - les variables d'environnement, accédées depuis, 114
 - mesures de qualité de code
 - avec CodeNarc, 248
 - avec FindBugs, 247
 - avec PMD et CPD, 243
 - métriques de couverture de code avec Cobertura, 156-158
 - ANT_OPTS environment variable, 58
 - Apache Maven
 - versions SNAPSHOT, 90
 - appareils mobiles, notifications vers , 229
 - application autonome
 - exécuter Jenkins comme, 52-56
 - application server
 - deploying Jenkins to, 57-58
 - upgrading Jenkins on, 68
 - applications à base de scripts, déployer vers un serveur d'application, 356-358
 - applications Java
 - déployer depuis un dépôt Maven, 353-356
 - déployer vers un serveur d'application, 346-356
 - rapports de test de, 146
 - redéployer depuis un build précédent, 349-353
 - Redéployer une version spécifique, 349
 - applications JEE (see applications Java)
 - applications PHP, déployer vers un serveur d'application, 356-358
 - applications Ruby, 139-140, 356-358
 - architecture maître/esclave pour les builds distribués, 319
 - archiver les binaires d'artefact
 - déployer vers un gestionnaire de dépôt d'entreprise, 127-131
 - archiver les tâches de build, 372-373
 - archives d'artefact de binaire
 - désactiver, 125
 - archives of binary artifacts, 27
 - artefacts binaires
 - réutiliser dans des pipelines de build, 301-305
 - Artifactory
 - Gestionnaire de dépôt d'entreprise, 129, 130
 - support de Jenkins pour, 5
 - Artifactory plugin, 291
 - artifacts (see artefacts binaires)
 - Atlassian Crowd, en tant que domaine de sécurité, 188-189
 - auditer les actions utilisateurs, 201-204
 - autorisation, 179, 179
 - (see also sécurité)
 - pas de restrictions sur, 180-181
 - sécurité basée sur le projet, 196-198
 - sécurité basée sur les rôles, 198-200
 - sécurité basée sur une matrice, 192-196
 - AWS (Amazon Web Services), 334
-

B

Backup plugin, 369

backups, 67

base de données, 180

(see also sécurité, domaines de sécurité)

base de données utilisateurs, 180, 181-184

mettre à jour avec le déploiement automatisé, 342-345

revenir sur des changements pour, 346

base de données utilisateurs, 180, 181-184

BDD (Behavior-Driven Development), 143

BDD (Behaviour Driven Development), 164

binaire d'artefacts

archiver

désactiver, 125

dans les tâches de build Freestyle, 118-121

binaire d'artifacts

archiver

déployer vers un gestionnaire de dépôt d'entreprise, 127-131

binary artifacts

archiving, 27

build history

in builds directory, 65-67

details regarding, 31-33

results summary for, 28, 31

build jobs

binary artifacts from (see binary artifacts)

build instable depuis

notifications pour, 122

code coverage metrics in (see code coverage metrics)

creating, 22-27

failed

example of, 29-33

indicator for, 29, 31

history of (see build history)

Javadocs generation in, 34-35

naming, 23

reports resulting from (see reporting)

scheduling (see build triggers)

source code location for, 24

status of, while running, 27

steps in, adding, 26-27, 34, 37

success of, indicator for, 29

triggering manually, 27, 28

types of, 23

unstable build from

indicator for, 38

build triggers

configuring, 25-26

manual, 27, 28

builds directory, 65-67

builds distribués, 47, 319-320

architecture maître/esclave pour, 319

avec une ferme de build basée sur le cloud, 333-337

noeuds esclaves pour

associer avec des tâches de build, 330-332

créer, 320

démarrer en mode headless, 329

démarrer en tant que service distant, 329

démarrer en utilisant SSH, 321-325

installer comme service Windows, 328-328

lancés avec Java Web Start, 325-328

surveillance, 332

builds instables, 147

critère de, 256

critère pour, 120, 161

déclencher d'autre build après, 122

déclencher une autre tâche de build à la suite de, 104

notifications pour, 122, 205, 208

builds quotidiens (see builds quotidiens automatisés)

builds quotidiens automatisés, 6

C

CAS (Central Authentication Service), 190

Checkstyle, 239-242, 255

clefs SSH, 12

clés SSH, 92

cloud computing, pour les builds, 176, 333-337

cloud Eucalyptus, 334

CloudBees (sponsor), xxix

- Clover, 162-163
- Cobertura, 36-42, 153-162
 - avec Ant, 156-158
 - configuration dans vos tâches de build, 158-161
 - avec Maven, 153-156
 - rappports de, 161-162
- code coverage metrics
 - with Cobertura, 36-42
- CodeNarc, 248-249
- complexité du code, 258-259
- config.xml file, 65
- configuration, 18-22, 69-72
 - Console de script, 71
 - CVS, 80
 - écran Configurer le système, 72-73
 - écran configurer système, 70
 - écran de configuration du rechargement à partir du disque, 70
 - écran de gestion des plugins, 71
 - écran Gérer les nœuds, 71
 - écran Information Système, 71
 - écran Log système, 71
 - écran Préparer à la fermeture, 72
 - écran Statistiques d'utilisation, 71
 - Git, 21-22
 - JDK, 20, 74-76
 - Maven, 19-20
 - message système sur la page d'accueil, 73
 - notifications, 21
 - outils de build, 77-79
 - période d'attente avant que le build ne démarre, 73
 - propriétés globales, 73-74
 - proxy, 81-82
 - serveur de messagerie, 80-81
 - Subversion, 80
 - systèmes de gestion de version, 79-80
- configurer
 - Ant, 78-79
 - Maven, 77-78
- Console Jenkins, 16
- conteneur de Servlet
 - en tant que domaine de sécurité, 187
- conteneur de Servlet GlassFish, 187
- conteneur de Servlet Tomcat, 187
- conteneur de servlets
 - exécuter Jenkins de façon autonome en utilisant, 52
- contributeurs pour ce livre, xxvii
- conventions utilisées dans ce livre, xxvi
- Coverage Complexity Scatter Plot plugin, 258
- CPD, 243-246
- CppUnit, 144
- CPUs, les besoins d'un serveur de build en, 46
- Crowd, Atlassian, en tant que domaine de sécurité, 188
- CVS
 - configurer, 80
 - Jenkins supporting, 21
 - retarder la construction d'un job, 86
 - retarder les tâches de build, 73
 - scruter avec, 106

D

- déclencheurs de build
 - pour les tâches de build free-style, 103-108
 - manuel, 108
 - paramétrés, 276-278
 - quand un autre build se termine, 104
 - scruter le SCM pour des changements dans le contrôle de version, 105
- déclencheurs de builds
 - déclencher à distance depuis le système de gestion de version, 106-108
 - à intervalles régulières, 104-105
- déclencheurs paramétrés, 276-278
- dépendances SNAPSHOT, 109, 124-124
- déploiement (see déploiement automatisé; déploiement continu)
- déploiement automatisé, 341-346
 - vers un serveur d'application, 346-358
 - mettre à jour de base de données avec, 342-345
 - revenir sur des changements dans, 345
 - script de déploiement pour, 342
 - tests fumigatoires pour, 345
- déploiement continu, 2, 7, 341-346

- vers un serveur d'application, 346-358
- mises à jour de base de données avec, 342-345
- revenir sur des changements dans, 345
- script de déploiement pour, 342
- tests fumigatoires pour, 345

dépôt

- cloner une copie locale de , 12

dépôt GitHub

- compte, configuration, 11
- forker, 12-13

Dépôt GitHub, 98, 102

développement piloté par les tests, 6

développement piloté par les tests d'acceptance, 6

disk space

- for build directory, 66

documentation (see Javadocs)

E

Eclipse

- mesures de qualité de code avec PMD, 243
- métriques de la qualité du code avec Checkstyle, 240
- notificateurs de bureau avec, 223

écran Administrer Jenkins, 69-72

écran Configurer le système, 72-73

écran configurer système, 70

écran Console de script, 71

écran de configuration du rechargement à partir du disque, 70

écran de gestion des plugins, 71

écran Gérer les nœuds, 71

écran Information Système, 71

écran Log système, 71

écran Préparer à la fermeture, 72

écran Statistiques d'utilisation, 71

EDI, mesures de qualité de code avec, 238

email notifications, 21

- (see also notifications)

espace disque

- surveillance, 361-365

exemples de code, utilisation, xxx

Extended Read Permission plugin, 198

F

fichier WAR, installer Jenkins à partir de , 17

FindBugs, 246-248, 255

fingerprints, 309, 314

fingerprints directory, 63

flux RSS, des résultats de build, 211-212

freestyle build jobs, 23

- build steps in
 - Maven build steps, 26
- creating, 23-27
- Git used with
 - post-build merging and pushing actions, 100-102

G

Game of Life example application, 23-40

Gerrit Trigger plugin, 99

gestionnaire de dépôt d'artefact, 127-131

Git, 9

- adresse du dépôt, 92
- auteur du commit, inclure dans le changelog, 96
- branches à construire, 93, 97
- clés SSH, 92
- déclencheurs de build, 98-100
- espace de travail, effacer avant le build, 97
- exclure certaines régions du déclenchement de builds, 94
- exclure des utilisateurs du déclenchement de builds, 95
 - avec des tâches de build free-style, 91-103
- fusionner avant le build, 96
- installation, 11
- localisation de l'espace de travail, écraser, 96
- mise à jour récursive des sous-modules, 96
- navigateurs de code source pour, 98
- nettoyer après récupération, 96
- post-build merging and pushing actions, 100-102
- récupérer sur une branche locale, 95
- tags, exécuter sur, 274-274
- tailler les branches distantes avant le build, 96

Git plugin, 21-22

GitHub repository, 9
Gmail, configurer, 81
Goldin, Evgeny (contributeur), xxvii
Gradle

- code quality metrics
 - with Checkstyle, 242
- construit dans, démarrer avec Jenkins, 134-137
- mesures de qualité de code
 - avec CodeNarc, 249
- support de Jenkins pour, 5

Grails

- construit dans, démarrer dans Jenkins, 133-134
- mesures de qualité de code avec CodeNarc, 249

Groeschke, Rene (contributeur), xxviii
Groovy scripts

- mesures de qualité de code avec CodeNarc, 248-249
- script d'authentification, 191-191

groupes

- Active Directory, 187
- Atlassian Crowd, 189
- LDAP, 186
- Unix, 187

groups

- Active Directory, 186

H
Hibernate, mises à jour de base de données avec, 343
historique de build

- nombre de builds à conserver, 85
- paramétré, 275
- permissions pour, 194

historique des builds

- utilisation disque de, 361-365

home directory for Jenkins, 63-67
hot-deploy, 346, 348
Hudson, xxv, 3, 3, 5

- (see also Jenkins)

I

IC (Intégration Continue), 1-3, 5-7

IM (see messagerie instantanée)
informations de contact pour ce livre, xxxi
installation

Ant, 79

Git, 11

JDK, 75

Jenkins, 43-45

from binary distribution, 44

sur un serveur de build, 46-47

sur CentOS, 50-51

sur Debian, 49-50

sur Fedora, 50-51

with Java Web start, 14-16

sur Linux, 44

sur OpenSUSE, 51-52

sur Redhat, 50-51

sur SUSE, 51-52

sur Ubuntu, 49-50

sur Unix, 44

à partir du fichier WAR, 17, 43

Windows, 43, 44

as Windows service, 59-62

JRE, 10

Maven, 19-20, 77

plugins, 36, 36

(see also specific plugins)

upgrading, 67-68

Intégration Continue (see IC)

IRC (Internet Relay Chat), 219-222

J

Java Development Kit (see JDK)

Java Runtime Environment (JRE), installation, 10

Java Web Start

installer et démarrer Jenkins en utilisant, 14-16

lancer des noeuds esclaves avec, 325-328

JAVA_OPTS environment variable, 58

Javadocs, 34-35

JDK (Java Development Kit), 9

configurer de multiples versions du, 74-76

configuring, 20

installer, 75

pré-requis, 43

- versions du, pour tâches de build multiconfiguration, 281
 - Jenkins, 3-4
 - arrêter, 16
 - communauté de, 4
 - configurer (see configuration)
 - CVS supported by, 21
 - cycle de livraisons rapide de, 5
 - disponible sur le port, 45
 - environment, requirements for, 9-13
 - exécuter
 - dans un serveur d'application, 17
 - depuis la ligne de commande, 17
 - avec Java Web Start, 14
 - comme une application autonome, 52-56
 - exécution
 - ligne de commande, 45
 - help icons in, 19
 - historique, 3-4
 - historique de, xxv
 - home directory for, 63-67
 - installer (see installation)
 - maintenance de, 361-376
 - archiver les tâches de build, 372-373
 - migrer les tâches de build, 373-376
 - sauvegardes, 367-371
 - Surveillance de l'espace disque, 361-365
 - surveiller la charge serveur, 365-366
 - maintenance of
 - backups, 67
 - mémoire requise pour, 46
 - memory requirements for, 58-58
 - comme un projet Open Source, 4
 - page d'accueil pour, 17, 73
 - port d'écoute, 44
 - pré-requis, 43
 - raisons pour utiliser, 4
 - répertoire de travail de, 73
 - répertoire de travail pour, 48-49
 - running
 - on Apache server, 56-57
 - on application server, 57-58
 - systèmes de gestion de version supportés par, 88
 - upgrading, 67-68
 - utilisateur dédié pour, 47
 - version control systems supported by, 21, 24
 - Jenkins M2 Extra Steps plugin, 132
 - JMeter, 167-174
 - jobs directory, 63-67
 - jobs externes, contrôler, 84
 - joins, in build jobs, 295-296
 - JRE (Java Runtime Environment), installation, 10
 - JUnit reports
 - configuring in freestyle build job, 27
 - format for, 26
- ## K
- Kawaguchi, Kohsuke (développeur d'Hudson), 3
- ## L
- la variable d'environnement BUILD_ID, 113
 - la variable d'environnement BUILD_NUMBER, 113
 - la variable d'environnement BUILD_TAG, 113
 - la variable d'environnement BUILD_URL, 114
 - la variable d'environnement CVS_BRANCH, 114
 - la variable d'environnement EXECUTOR_NUMBER, 113
 - la variable d'environnement HUDSON_URL, 114
 - la variable d'environnement JAVA_HOME, 113
 - la variable d'environnement JOB_NAME, 113
 - la variable d'environnement JOB_URL, 114
 - la variable d'environnement NODE_LABELS, 113
 - la variable d'environnement NODE_NAME, 113
 - la variable d'environnement SVN_REVISION, 114
 - la variable d'environnement WORKSPACE, 113
 - LDAP/Active Directory, 5
 - le plugin Checkstyle, 256
 - le plugin FindBugs, 256
 - le plugin Parameterized Build, 267
 - le plugin Parameterized Trigger, 276

- le plugin PMD, 256
- les archives de binaire d'artefacts
 - dans les tâches de build Freestyle, 118-121
- les builds instables
 - critère pour, 253
- les projets Ruby on Rails, 343
- les scripts batch, 111-113
- Les scripts de build NAnt, 138
- les scripts Groovy
 - démarrer dans des tâches de build, 115-117
 - les variables d'environnement dans, 115
- les scripts shell, 111-113
- les tâches de build
 - build instable
 - critère pour, 120
 - les mesures de qualité de code dans (see les mesures de qualité de code)
 - les tâches de build free-style
 - mesures de qualité de code dans, avec Violations, 250-253
 - les tâches de build Freestyle
 - Actions à la suite du build, 117-122
 - les tâches de build Maven
 - mesures de qualité de code intégrés, avec Violations, 253-255
 - les variables d'environnement, 113
 - (see also les variables d'environnement spécifiques)
 - utilisées dans les étapes de build, 113-115
- Linux, 49
 - (see also plates-formes Linux spécifiques)
 - upgrading Jenkins on, 67
- Liquibase, 343-345
- livraison continue, 2
- livraisons LTS (Long-Term Support), 3
- lmachine esclave
 - pour les builds distribués, 319

M

- M2Eclipse, 5
- machine virtuelle, pour serveur de build, 46, 176
- machines esclaves
 - pour builds distribués , 47
 - pour tâches de build multiconfiguration, 280-281
- maintenance, 361-376
 - archiver les tâches de build, 372-373
 - backups, 67
 - migrer les tâches de build, 373-376
 - sauvegardes, 367-371
 - Surveillance de l'espace disque, 361-365
 - surveiller la charge serveur, 365-366
- Manage Jenkins screen, 18
- Maven, 9
 - automatiser les tests, 379-384
 - build steps in freestyle build jobs, 26, 109-111
 - Cobertura avec, 153-156
 - code quality metrics
 - with Checkstyle, 241
 - configurer, 77-78
 - configuring, 19-20
 - dépendances SNAPSHOT, 109, 124-124
 - installer, 77
 - installing, 19-20
 - les variables d'environnement dans, 114
 - mesures de qualité de code
 - avec CodeNarc, 248
 - avec FindBugs, 247
 - métriques de la qualité du code
 - avec PMD et CPD, 245
 - numéro de version pour, 298-301
 - support d'Hudson pour, 5
- Maven build jobs, 23
- Maven Jenkins plugin, 286, 293
- Maven Release plugin, 298
- MAVEN_OPTS environment variable, 58
- McCullough, Matthew (contributeur), xxvii
- mémoire, requise pour, 46
- memory, requirements for, 58-58
- messagerie instantanée (IM), 214-219
 - IRC pour, 219-222
 - protocole Jabber pour, 214-219
- messages de commit, exclusion du déclenchement de tâches de build, 90
- messages SMS, notifications via, 230-231
- mesures de qualité de code, 237-239, 258-259

- avec Checkstyle, 239-242
- avec CodeNarc, 248-249
- avec CPD, 243-246
- avec FindBugs, 246-248, 255
- logiciel pour, 239
- avec PMD, 242-246
- avec Sonar, 238
- tâches ouvertes, 259-260
- avec le plugin Violations, 249-255
- mesures de qualité du code
 - dans les tâches de build, 238
 - avec Checkstyle, 255
 - avec l'EDI, 238
 - plugins pour, 238
 - avec PMD, 255
 - with Sonar, 261-264
- métriques (see rapports)
- métriques de couverture de code, 6, 152-163
 - avec Clover, 162-163
 - avec Cobertura, 153-162
 - logiciels pour, 153
- métriques de qualité de code, 6
- Microsoft Active Directory, comme domaine de sécurité, 186-187
- migrer les tâches de build, 373-376
- mode headless, démarrer des noeuds esclaves en, 329
- MSTest plugin, 138

N

- NAnt plugin, 139
- navigateurs de code source
 - avec Git, 98
 - avec Subversion, 89
- projets .NET, 137-138
- NetBeans, 224
- Nexus
 - Gestionnaire de dépôt d'entreprise, 130
 - Gestionnaire de Dépôts d'Entreprise, 301
 - support d'Hudson pour, 5
- normes de codage, 237
- notificateurs de bureau, 223-229
- notifications, 205

- configuring, 21
- email, 183, 205-210
- flux RSS, 211-212
- depuis une tâche de build Freestyle, 122-122
- messaging instantanée, 214-219
- messages SMS, 230-231
- vers les appareils mobiles, 229
- utilisant Nabaztag, 235
- notificateurs de bureau, 223-229
- notifications actives (push), 205
- parlées, 234
- passive (pull), 205
- radars de build, 212-213
- vers les smartphones, 227-230
- sons dans, 232
- notifications actives (push), 205
- notifications email, 205-210
- notifications par email, 183, 210
- notifications passives (pull), 205
- Notifo, 227-229
- NTLM proxy authentication, 82
- numéros de version, Maven, 298-301
- NUnit, 144

O

- Odd-e (sponsor), xxx
- outils de build, configurer, 77-79
- outils Sonatype, 4, 5

P

- page d'accueil, 17, 73
- page de démarrage (see page d'accueil)
- paramètre JAVA_ARGS, 50
- paramètre JENKINS_JAVA_CMD, 51
- paramètre JENKINS_JAVA_OPTIONS, 51
- paramètre JENKINS_PORT, 51
- paramètres Booléen, 271
- paramètres chaînes, 268
- paramètres Fichier, 272
- paramètres Mot de passe, 271
- paramètres Run, 271
- performance

- de l'analyse de la couverture de code, 152
- des applications, 167-174
- des tests, 149-150, 175-177
- période d'attente avant que le build ne démarre, 73
- période d'attente avant que le build ne soit lancé, 86
- permissions (see autorisation)
- permissions de niveau projet, dans la sécurité basée sur les rôles, 199
- PHPUnit, 144
- pipelines (see pipelines de build)
- pipelines de build, 298-317
 - agréger les résultats de tests d'un, 313-314
 - numéros de versions Maven pour, 298-301
 - pipelines de déploiement, 314-317
 - promotions dans, 298, 305-313
 - réutiliser des artefacts dans, 301-305
- pipelines de déploiement, 314-317
- plugin Audit Trail, 201-202
- plugin Build Pipeline, 314
- plugin Build Promotion, 347
- plugin Clover, 163
- plugin Cobertura, 158
- plugin Copy Artifact, 302, 347, 349
- plugin Dependency Graph View, 294
- plugin Deploy, 346, 347-349, 349
- plugin Deploy Websphere, 346, 347
- plugin Disk Usage, 362-363
- plugin DocLinks, 166
- plugin Eclipse, 223
- plugin Email-ext, 207-210
- plugin Git, 91-92
- Plugin GitHub, 102
- plugin HTML Publisher, 165-166
- plugin Instant Messaging, 214
- plugin IRC, 219, 220
- plugin Jabber Notifier, 214
- plugin JobConfigHistory, 202-204
- plugin Locks and Latches, 297
- plugin MSBuild, 137
- plugin Nabaztag, 235
- plugin Parameterized Trigger, 290, 349
- plugin Promoted Builds, 305
- plugin Role Strategy, 198
- plugin Script Security Realm, 190-191
- plugin Sounds, 232
- plugin Speaks, 234
- plugin Thin Backup, 371
- plugin Tray Application, 226-227
- plugin Violations , 249-255
- plugin xUnit, 146
- plugins
 - Active Directory, 186
 - Amazon EC2, 335
 - architecture de, Jenkins comparé à Hudson, 5
 - Artifactory, 129, 291
 - Audit Trail, 201-202
 - Backup, 369
 - Build Pipeline, 314
 - Build Promotion, 347
 - CAS, 190
 - Checkstyle, 256
 - Clover, 163
 - Cobertura, 158
 - Copy Artifact, 302, 347, 349
 - Coverage Complexity Scatter Plot, 258
 - Crowd, pour Atlassian Crowd, 188
 - Dependency Graph View, 294
 - Deploy, 346, 347-349, 349
 - Deploy Websphere, 346, 347
 - Disk Usage, 362-363
 - DocLinks, 166
 - Eclipse, 223
 - Email-ext, 207-210
 - Extended Read Permission, 198
 - FindBugs, 256
 - Gerrit Trigger, 99
 - gestion, 71
 - Git, 21-22, 91-92
 - GitHub, 102
 - HTML Publisher, 165-166
 - installing, 36-37
 - IRC, 219, 220
 - Jabber Notifier, 214
 - Jenkins M2 Extra Steps, 132
 - JobConfigHistory, 202-204

- Locks and Latches, 297
- Maven Jenkins, 286, 293
- Maven Release, 298
- messagerie instantanée, 214
- MSBuild, 137
- MSTest, 138
- Nabaztag, 235
- NAnt, 139
- Parameterized Build, 267
- Parameterized Trigger, 276, 290, 349
- plugin Tray Application, 226-227
- PMD, 256
- Promoted Builds, 305
- Publish Over, 357
- Role Strategy, 198
- Script Security Realm, 190
- SFEE, 190
- Sounds, 232
- Speaks, 234
- Task Scanners, 259
- Thin Backup, 371
- upgrading, 68
- Violations, 249-255
 - xUnit, 146
- plugins directory, 63
- plugins Publish Over, 357
- PMD, 242-246, 255
- polices utilisées dans ce livre, xxvi
- processeurs, les besoins d'un serveur de build en, 46
- projet Github, 4
- projets Ruby on Rails, 139-140
- promotions, 298, 305-313
- propriétés
 - globales, 73-74
 - paramètres de build comme, 270
- propriétés globales, 73-74
- protocole Jabber, 214-219
- proxy, configurer, 81-82

R

- radars d'informations, 212-213
- radars de build, 212-213

- radars, informations, 212-213
- rapport
 - les résultats de test
 - les rapports JUnit, 117-118
 - mesures de qualité de code
 - avec Checkstyle, 255
 - tâches ouvertes, 259-260
- rapporter
 - métriques de couverture de code
 - de Cobertura, 161-162
 - résultats de test
 - afficher, 147-150
 - configurer, 145-146
 - dans les flux RSS, 211-212
- rapports
 - mesures de qualité de code
 - avec PMD, 255
 - mesures de qualité du code
 - avec FindBugs, 255
 - métriques de couverture de code, 6
 - de Clover, 163
 - métriques de qualité de code, 6
 - résultats de test d'acceptation, 164-166
 - résultats de test de performance, 172-174
- rapports JUnit, 144
 - pour tests d'acceptation, 164
 - configurer dans une tâche de build free-style, 146
- répertoire de travail de Jenkins, 73
- répertoire de travail pour Jenkins, 48-49
- reporting
 - code coverage metrics, 36-42
 - Javadocs API documentation, 34-35
 - mesures de qualité de code
 - plugin Violations pour , 249-255
 - résultats de test
 - agréger, 313-314
 - test results, 31-33
 - JUnit reports, 26
- revendiquer des builds échoués, 210

S

- sauvegardes, 367-371

- sauvegardes plus légères, 371
- SCM (Gestion du Code Source), 88-103
- SCM (Source Code Management), 24
 - (see also version control systems)
- script de déploiement, 342
- scripts, 111
 - (see also Ant; Maven)
 - les langages supportées, 117
 - les scripts batch, 111-113
 - les scripts Groovy, 115-117
 - les scripts shell, 111-113
 - paramétrés, 269-270
 - script de déploiement, 342
 - scripts batch, 79
 - scripts d'authentification personnalisés, 190
 - scripts shell, 79
- scripts batch, 79
- scripts de build (see scripts)
- scripts Groovy
 - fonctionnant dans la console de script, 71
- scripts shell, 79
- securité, 179-181
 - activer, 179
 - autorisation, 179
 - pas de restrictions sur, 180-181
 - domaine de sécurité, 179
- sécurité
 - autorisation
 - sécurité basée sur le projet, 196-198
 - sécurité basée sur les rôles, 198-200
 - sécurité basée sur une matrice, 192-196
 - domaine de sécurité
 - Atlassian Crowd, 188-189
 - domaines de sécurité
 - activer l'enregistrement utilisateur, 183
 - activer les connexions, 180
 - annuaire LDAP, 185-186
 - base de données utilisateurs interne à Jenkins, 180, 181-184
 - CAS, 190
 - conteneur de Servlet, 187
 - Microsoft Active Directory, 186-187
 - personnaliser, 190-191
 - SFEE, 190
 - utilisateurs et groupes Unix, 187
 - sécurité basée sur le projet, 196-198
 - sécurité basée sur les rôles, 198-200
 - sécurité basée sur une matrice, 192-196
 - serveur d'application
 - déploiement automatisé vers, 346-358
 - applications à base de scripts, 356-358
 - applications Java, 346-356
 - déployer Jenkins dans, 17
 - serveur d'application GlassFish, déployer des applications Java vers, 346-356
 - serveur d'application JBoss, déployer des applications Java vers, 346-356
 - serveur d'application Tomcat
 - déployer des applications Java vers, 346-356
 - déployer Jenkins en utilisant, 17
 - serveur de build, 5
 - besoins en processeur pour, 46
 - installation de Jenkins sur, 46-47
 - machine virtuelle pour, 46, 176
 - mémoire requise pour, 46
 - mise à jour, 175
 - multiplés, exécution des builds sur (see builds distribués)
 - surveiller la charge du, 365-366
 - serveur de messagerie électronique, configurer, 80-81, 80-81
 - serveur proxy HTTP, 81
 - service de cloud computing Amazon EC2, 333-337
 - service distant, démarrer des noeuds esclaves en tant que, 329
 - services services
 - démarrer des noeuds esclaves en tant que, 329
 - services Windows
 - installer des noeuds esclaves comme, 328-328
 - SFEE (Source Forge Enterprise Edition), 190
 - smartphones, notifications vers, 227-229
 - Sonar
 - fréquence des builds, 104
 - les mesures de qualité de code avec, 261-264
 - mesures de qualité de code avec, 238

- sons, dans les notifications, 232
- Source Code Management (see SCM; version control systems)
- Source Forge Enterprise Edition (see SFEE)
- sponsors pour ce livre, xxix
- SSH, démarrer un noeud esclave en utilisant, 321-325
- stand-alone application
 - upgrading Jenkins as, 67
- Subversion
 - configurer, 80
 - exclure des messages de commit du déclenchement de builds, 90
 - exclure des régions du déclenchement de builds, 90
 - exclure des utilisateurs du déclenchement de builds, 90
 - avec les tâches de build free-style, 88-91
 - Jenkins supporting, 21
 - navigateurs de code source pour, 89
 - tags, build à partir de, 273-273
- systèmes de contrôle de version
 - déclencher des builds à distance depuis, 106-108
 - scruter les changements pour déclencher un build, 105
- systèmes de gestion de version, 79
 - (see also CVS; Git; Subversion)
 - configurer, 79-80
 - supporté par Jenkins, 88
 - supportés par Jenkins, 79-80

T

- tâche de build
 - build instable de
 - notifications pour, 208
 - build instable depuis
 - déclencher d'autre build après, 122
- tâche de build free-style
 - description de, pour la page de démarrage du projet, 85
- tâche de build multiconfiguration, 279-285
- tâches cron (see jobs externes)

- tâches de build
 - build instable de, 147
- tâches de build, 83, 83
 - (see also tâches de build free-style; tâches de build Apache Maven)
 - archiver, 372-373
 - axe JDK pour, 281
 - build instable de
 - critère pour, 161
 - déclencher un autre build à la suite de, 104
 - notifications pour, 205
 - build instable depuis
 - critère de, 256
 - critère pour, 253
 - copier, 84
 - créer, 83-84
 - déclencher manuellement, 108
 - dépendances entre, 294
 - distribuées parmi des serveurs de build, 319-320
 - créer des noeuds esclaves, 320
 - démarrer des noeuds esclaves, 321
 - distribuées sur des serveurs de build
 - ferme de build basée sur le cloud pour, 333-337
 - distribuées sur les serveurs de build
 - associer les noeuds esclaves aux tâches, 330-332
 - distribués sur les serveurs de build
 - surveillance de noeuds esclaves, 332
 - échouées
 - détails à propos de, 147-149
 - notifications pour, 205, 208
 - revendiquer, 210
 - exécution en parallèle, 293-297
 - externe, contrôler, 84
 - jonctions dans, 295-296
 - migrer, 373-376
 - multiconfiguration, 279-285
 - axe esclave pour, 280-281
 - axe JDK pour, 281
 - axe personnalisé pour, 282
 - créer, 279-280

- exécuter, 282-285
- filtre de combinaison pour, 283
- matrice de configuration, 283
- numéro d'exécution pour, en tant que paramètres, 271
- paramétré
 - types de paramètres, 268
- paramétrées, 267-275
 - créer, 267
 - démarrées à distance, 275-275
 - exécutées sur un tag Git, 274-274
 - exécutées sur un tag subversion, 273-273
 - historique de, 275
 - tâches de build paramétrées, 269-270
 - types des paramètres, 271-272
- propriétés globales pour, 73-74
- retarder le démarrage, 73
- serveurs de build distribué parmi
 - architecture maître/esclave pour, 319
- tests dans (see tests)
- types de, 83
- verrouiller les ressources pour, 297-297
- tâches de build Apache Maven, 83
- tâches de build free-style, 83, 84-88
 - Actions à la suite du build, 145
 - bloquer pour un projet en amont, 86
 - déclencheurs de build pour, 103-108
 - désactiver, 86
 - échouées, 148
 - espace de travail pour, surcharger, 87
- Git utilisé avec, 91-103
 - adresse du dépôt, 92
 - auteur du commit, inclure dans le changelog, 96
 - branches à construire, 93, 97
 - clés SSH, 92
 - déclencheurs de build, 98-100
 - espace de travail, effacer avant le build, 97
 - exclure certaines régions du déclenchement, 94
 - exclure des utilisateurs du déclenchement, 95
 - exécutable Git, spécifier, 97
 - fusionner avant le build, 96
 - localisation de l'espace de travail, écraser, 96
 - mise à jour récursive des sous-modules, 96
 - navigateurs de code source pour, 98
 - nettoyer après récupération, 96
 - récupérer sur une branche locale, 95
 - tailler les branches distantes avant le build, 96
- historique de build pour, nombre de builds à conserver, 85
- nommage, 85
- période d'attente, 86
- rapport sur des résultats de test, 145
- Subversion utilisé avec, 88-91
 - exclure des messages de commit du déclenchement, 90
 - exclure des régions du déclenchement, 90
 - exclure des utilisateurs du déclenchement, 90
 - navigateurs de code source pour, 89
- tâches de build Freestyle
 - Archiver les binaires d'artefacts, 118-121
 - démarrer, 123
 - démarrer d'autres tâches de build depuis, 122
 - les étapes de build dans, 108-117
 - Les étapes de build Maven, 109-111
 - les scripts batch, 111-113
 - Les scripts de build Ant, 111-111
 - les scripts Groovy, 115-117
 - les scripts shell, 111-113
 - les variables d'environnement dans, 113-115
 - Les scripts de build NAnt dans, 138
 - projets .NET dans, 137-138
 - notifications envoyées après, 122-122
 - projets Gradle dans, 134-137
 - projets Grails dans, 133-134
 - projets Ruby et Ruby on Rails dans, 139-140
 - rapport sur les résultats de test, 117-118
- tâches de build freestyle
 - générer automatiquement, 293
- tâches de build matrix (see tâches de build multi-configuration)
- tâches de build Maven, 123-132

- Actions à la suite du build, 126
- archiver les binaires d'artefact, désactiver, 125
- builds incrémentals, 125
- créer, 123
- démarrer des modules en parallèle, 126
- déployer les artefacts vers un gestionnaire de dépôt d'artefact, 127-131
- dépôt privé pour, 126
- générer automatiquement, 285-293
 - configurer, 286-288
- les étapes de build dans, 124, 132
- modules for, managing, 131
- rapport sur des résultats de test, 145
- résultats de test de, 147
- utilisation du disque des, 364-365
- Tâches de build Maven
 - générer automatiquement
 - Artifactory plugin avec, 291
 - héritage de configuration, 288-290
 - plugin Parameterized Trigger avec, 290
- tâches de build multi-configuration, 84
- tâches de build multiconfiguration
 - axe esclave pour, 280-281
 - axe personnalisé, 282
 - créer, 279-280
 - exécuter, 282-285
 - filtre de combinaison pour, 283
 - matrice de configuration, 283
- tâches de build paramétré, 279
 - (see also tâches de build multiconfiguration)
- tâches de build paramétrées, 267-275
 - créer, 267
 - démarrage à distance, 275-275
 - exécutées sur un tag Git, 274-274
 - exécutées sur un tag Subversion, 273-273
 - historique de, 275
 - scripts de build pour, 269-270
 - types de paramètres, 268
 - types des paramètres, 271-272
- tâches ouvertes, rapport sur , 259-260
- Task Scanners plugin, 259
- TDD (Test Driven Development), 143
- Test Result Trend graph, 32
- Test::Unit, 144
- TestNG, 144, 146, 151
- tests
 - automatisation, 6
 - automatiser, 6
 - avec Ant, 384-387
 - avec Maven, 379-384
 - automatisés, 143-145
 - développement piloté par les tests, 6
 - dans des tâches de build free-style, 145
 - ignorer, 150-152
 - dans des tâches de build Maven, 145
 - performance de, 149-150, 175-177
 - rapport de
 - agrégation, 313-314
 - rapports de
 - afficher, 147-150
 - configurer, 145-146
 - rapports depuis
 - les rapports JUnit, 117-118
 - reports from, 31-33
 - JUnit reports, 26
 - tests d'acceptance, 6
 - tests d'acceptation, 143, 164-166
 - tests d'intégration, 143, 144
 - tests de performance, 167-174
 - tests fonctionnels (régression), 144
 - tests fonctionnels (regression), 144
 - tests fumigatoires, 345
 - tests unitaires, 143, 144
 - tests web, 144, 144
- tests automatisés (see tests)
- tests d'acceptance tests, automatisés, 6
- tests d'acceptation, automatisé, 143
- tests d'acceptation, automatisés, 164-166
- tests d'intégration, 143, 144
 - nombre de, 176
 - performance de, 175
- tests de régression (see tests fonctionnels (régression))
- tests fonctionnels (régression), 144
 - exécution en parallèle, 176
 - nombre de, 176

- performance de, 175
- tests fonctionnels (regression), 144
- tests fumigatoires, 345
- tests unitaires, 143, 144
- tests web, 144, 144

U

- Ubuntu Enterprise Cloud, 335
- Unix, 44
 - (see also plates-formes Unix spécifiques)
 - utilisateurs et groupes, en tant que domaine de sécurité, 187
- unstable builds
 - indicator for, 38
- updates directory, 63
- upgrades, 67-68
- userContent directory, 63
- users directory, 64
- utilisateurs
 - administrateur
 - pour la base de données utilisateurs interne de Jenkins, 181
 - pour la sécurité basée sur une matrice, 192
 - auditer les actions des, 201-204
 - autorisation pour (see autorisation)
 - exclure du déclenchement de builds, 95
 - exclure pour le déclenchement de build, 90
 - pour Jenkins, sur un serveur de build, 47
 - revendiquer des builds échoués, 210

V

- variable d'environnement HUDSON_HOME, 48
- variable d'environnement JAVA_HOME, 75
- variable d'environnement JENKINS_HOME, 48-49, 73
- variable d'environnement P, 190
- variable d'environnement U, 190
- variables d'environnement
 - paramètres de build comme, 269
- verrouiller des ressources pour des tâches de build, 297-297
- version control systems, 9

- configuring, 24
- versions SNAPSHOT, 90
- Visual Studio MSBuild, 137-138
- votre version de Java , 43

W

- Wakaleo Consulting (sponsor), xxix
- war directory, 64
- WebSphere Application Server, 346, 347
- Windows
 - package d'installation de Jenkins , 43
- Windows services
 - installing Jenkins as, 59-62
- WMI (Windows Management Instrumentation), 329
- workspace directory, 65

X

- XML format for test reports (see JUnit reports)
- Xu, Juven (contributeur), xxviii
- xUnit, 144, 146

Colophon

L'animal sur la couverture de Jenkins : Le guide complet est une grenouille faux-grillon ornée (*Pseudacris ornata*). On peut trouver ces petits amphibiens, de seulement 2,5 à 3,5 centimètres de long, sur les plaines côtières d'Amérique du Nord de la Caroline du Nord jusqu'au milieu de la Floride et dans l'est de la Louisiane. Ils préfèrent les zones d'eau peu profonde sans végétation dense, telles que les mares, les fossés de bord de route, et les prés inondés.

La couleur des grenouilles faux-grillons ornées varie en fonction des endroits, et les individus peuvent être principalement noir, blanc, marron, rouge, vert, ou toute autre variation proche. Quand bien même, tous les spécimens, possèdent une bande sombre ou un ensemble de taches allant des narines jusqu'en haut du dos en passant par l'oeil, et la plupart possèdent aussi d'autres taches ou bandes. Les espèces se reproduisent de novembre à mars, et l'appel des mâles peut être entendu dans ou à proximité des zones d'eau peu profondes.

Les grenouilles faux-grillons ornées doivent également leur nom au son de leur champ nuptial : *Pseudacris* vient du grec ancien signifiant "faux grillon". Ce nom fut donné en 1836 par le naturaliste américain John Edwards Holbrook après qu'il ait observé que le son aigu ressemblait à celui produit par ce non moins célèbre insecte.

L'image de couverture est tirée de l'ouvrage Cassell's Natural History. La police d'écriture de la couverture est Adobe ITC Garamond. La police utilisée pour le texte est Linotype Birka ; la police des titres est Adobe Myriad Condensed ; et la police utilisée pour le code est LucasFont's TheSansMonoCondensed.
