Documentation Utilisateur

Développer en Java avec Maven

11 octobre 2011

Versions

Évolution du document			
Version	Date	Auteur	Remarques
1.0	05/07/2010	Émilie Oudot	-
2.0	06/06/2011	Pierre-Christophe	-

Table des matières

Pré-requis

Dans ce document, il est supposé que les technologies utilisées sont celles répertoriées dans le tableau suivant. Toutes les explications/captures d'écran sont basées sur ces logiciels et ces versions.

Outil de build	Maven 2.*
Repository Maven	Nexus 1.6
Intégration Continue	Hudson 1.363

Urls d'accès aux serveurs étudiants

Serveur	Url d'accès
Hudson	`http://prjmng.deptinfo-st.univ-fcomte.fr:8090/`
Nexus	`http://prjmng.deptinfo-st.univ-fcomte.fr:8081/nexus/`

1 Maven par la théorie

1.1 Architecture d'un projet

Maven impose certaines contraintes pour garantir son fonctionnement. Il est notamment nécessaire de décrire l'ensemble du projet dans un unique fichier, le `POM`, ou *Project Object Model*.

1.1.1 Le `POM`

Chaque projet et sous-projet Maven est configuré par un `POM` qui contient les informations nécessaires à Maven pour traiter le projet, telles que son nom, sa version, les noms des contributeurs, ...). Ce `POM` se matérialise par un fichier `pom.xml` à la racine du projet.

1.1.1.a Super `POM`

Chaque `POM` de projet hérite du `Super POM` implicitement. C'est ce fichier qui indique les informations minimales du projet sur sa structuration, mais également sur le dépôt principal de Maven.

1.1.1.b `POM` minimal

Le contenu minimal du `POM` est le suivant :

```
<project>
       <modelVersion>4.0.0</modelVersion>
       <groupId>such.a.group</groupId>
       <artifactId>mon-artefact</artifactId>
       <version>1.0.0</version>
</project>
```

Un projet est identifié par son groupe, son nom d'artefact et sa version. Ces trois champs forment le nom unique de chaque projet, sous la forme suivante :

<groupId> :<artifactId> :<version>

Comme mentionné précédemment, si des informations nécessaires au projet ne sont pas indiquées dans le `POM` du projet, Maven ira trouver ces informations dans le Super `POM`.

Notons que les versions peuvent être indiquées par n'importe quoi. Toutefois, par convention, on s'attachera à utiliser la notion `<majeur>.<mineur>.<micro>`. Ainsi, le passage de 3.1.4 à 3.1.5 indique la correction de bugs dans la version 3.1 du logiciel, n'impliquant que peu de choses sur les fonctionnalités en elle-même. Le passage de 3.1 à 3.2 indique l'ajout de fonctionnalités, mais sans régression des autres fonctionnalités disponibles précédemment. Le passage de la version 3 à la version 4 indique un changement majeur dans l'architecture du logiciel, induisant potentiellement un changement dans l'utilisation du logiciel.

De plus, Maven est capable de faire la différence entre deux phases de développement d'un projet : la phase `release` et la phase `snapshot`. Les versions de projet se terminant par `-SNAPSHOT` sont considérés comme étant des snapshots du projet, des instantanées n'ayant pas pour but d'être utilisé réellement, mais plutôt livré à des fins de test de mise en situation d'utilisation. Les autres versions de projet sont considérés par Maven comme des releases.

1.1.1.c Héritage de `POM`

De la même façon que chaque projet hérite par défaut du Super `POM`, il est possible pour un projet d'hériter de la configuration d'un autre projet. Les informations héritées sont les suivantes :

- dépendances,
- développeurs et contributeurs,
- liste et configuration des plugins,
- exécution des plugins ayant un `id` correspondant,
- ressources

L'héritage entre projets s'effectue de la manière suivante. Si un projet *pere* est créé, et qu'un projet *fils* souhaite en hériter, alors le `POM` minimal du projet fils est le suivant :

```
<project>
       <modelVersion>4.0.0</modelVersion>
       <artifactId>fils</artifactId>
       <parent>
              <groupId>such.a.father.group</groupId>
              <artifactId>pere</artifactId>
              <version>1.0.0</version>
       </parent>
</project>
```

Dans cet exemple, les groupes et versions du projet *fils* sont hérités du projet *pere*. Il est bien sûr possible de donner une valeur aux groupes et versions du projet *fils* indépendamment du projet *pere*, comme dans l'exemple suivant :

```
<project>
       <modelVersion>4.0.0</modelVersion>
       <artifactId>fils</artifactId>
       <version>2.0.0</version>
       <parent>
              <groupId>such.a.father.group</groupId>
              <artifactId>pere</artifactId>
              <version>1.0.0</version>
       </parent>
</project>
```

Dans cet exemple, le projet *fils* ne spécifie aucun groupe. Il hérite donc du groupe du projet *pere*, à savoir *such.a.father.group*. Par contre, le projet *fils* spécifie une version (*2.0.0*). Celle-ci est donc utilisée, en lieu et place de celle du projet *pere*.

En résumé, deux projets sont considérés, avec les informations suivantes :

Groupe	such.a.father.group	such.a.father.group
Artefact	pere	fils
Version	1.0.0	2.0.0

La configuration précédente est valable uniquement dans le cas où le projet *pere* existe déjà dans le dépôt Maven, où dans le cas où le projet *pere* se situe en amont dans l'arborescence du projet *fils*. Dans le cas où le développement des projets *pere* et *fils* se feraient conjointement, et sur des arborescences séparées, un champ permet d'indiquer le chemin vers le `POM` du projet *pere*. Considérons l'architecture suivante :

dossier/
pere/
pom.xml
fils/
pom.xml

Le `POM` minimal du projet *fils* sera :

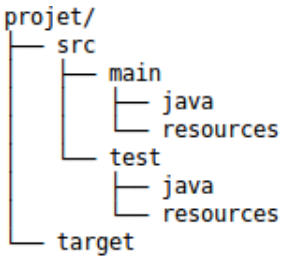
```
<project>
	<modelVersion>4.0.0</modelVersion>
	<artifactId>fils</artifactId>
	<parent>
		<groupId>such.a.father.group</groupId>
		<artifactId>pere</artifactId>
		<version>1.0.0</version>
		<relativePath>../pere/pom.xml</relativePath>
	</parent>
</project>
```

1.1.1.d Création de projet

Il est possible de créer un projet "à la main", en créant soit-même le `POM` et l'arborescence du projet, mais un plugin Maven, `maven-archetype-plugin`, permet de construire un ensemble assez conséquent de projets par défaut, selon l'application finale du projet (401 projets prédéfinis). Les IDE sont également capables de créer des projets Maven, mais avec moins de choix que le plugin dédié.

1.1.2 Arborescence d'un projet

Tout projet Maven induit une architecture bien spécifique afin de garantir l'utilisation de Maven. Par défaut, l'architecture est la suivante :



Chaque projet est constitué par un fichier `pom.xml`, ainsi que deux répertoires. Le répertoire `src` est destiné à contenir tous les fichiers source nécessaire à la compilation et au fonctionnement du projet. Le répertoire `target` est destiné à recevoir tous les fichiers issus de la construction de votre projet. Ce répertoire n'est pas destiné à être conservé, son contenu doit pouvoir être reconstruit à chaque construction du projet.

Le dossier `src` contient, par défaut, un répertoire `main` et un répertoire `test`. Le répertoire `main` doit contenir l'ensemble des fichiers source du projet, tandis que le répertoire `test` doit contenir l'ensemble des tests du projet.

Dans chacun de ces deux dossiers se trouvent, par défaut, deux répertoires, `java` et `resources`. Le dossier `java` est destiné à contenir les fichiers sources Java, tandis que le répertoire `resources`

contient l'ensemble des fichiers annexes nécessaires, soit à l'exécution du projet (par exemple des fichiers d'image pour une interface graphique) si on se trouve dans le dossier `main`, soit à l'exécution des tests (par exemple des fichiers d'entrée pour vérifier la conformité) si on se trouve dans le dossier `test`.

Il est bien sûr possible de configurer, pour chaque projet, les dossiers indiquant les fichiers sources, les ressources, les tests, ...Toutefois, il est conseillé de conserver ce type d'architecture afin de faciliter la lecture d'un projet.

1.2 Architecture modulaire

L'intérêt de l'utilisation de Maven réside dans son architecture modulaire. En effet, chaque projet peut définir un ensemble de fonctionnalités, qui peuvent être ensuite utilisées par d'autres projets Maven, de la même façon qu'un projet Java utilise des librairies (fichiers .jar) pour fonctionner. Cette architecture modulaire s'exprime par un ensemble de dépendances entre les projets.

1.2.1 Notion de dépendances

Une dépendance désigne, dans l'absolu, un fichier jar apportant un ensemble de fonctionnalités que l'on souhaite utiliser au sein du projet souhaité. Dans le cas de Maven, chaque projet est identifié de façon unique par son groupe, son identifiant et sa version. Il est alors très simple de définir une dépendance à un projet.

L'ajout d'une dépendance à un projet s'effectue dans le `POM` du projet. Dans le cas suivant, nous ajoutons une dépendance à Junit dans notre projet, qui est nécessaire pour compiler et exécuter les tests :

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

Une dépendance est donc identifiée comme un projet, par son groupe, son identifiant et sa version. Il est possible de restreindre la validité d'une dépendance à chaque étape de la vie du projet, par le `scope`. Les scopes possibles sont :

- `compile` : c'est le `scope` par défaut, la dépendance est toujours considéré, que ce soit à la compilation, ou à l'exécution du projet ou des tests,
- `test` : la dépendance est considérée uniquement dans les phases de test,
- `runtime` : la dépendance est nécessaire à l'exécution, mais pas à la compilation
- `provided` : la dépendance est nécessaire à la compilation, mais ne sera pas packagée avec le projet. Typiquement, un container lui fournira cette dépendance
- `system` : la dépendance est toujours considérée, mais n'est pas un projet Maven. Dans ce cas, un champ supplémentaire, `<systemPath>` permet d'indiquer le chemin vers le fichier jar souhaité (typiquement, les librairies Smartesting).

Lors de l'ajout d'une dépendance en version `snapshot` au projet, à chaque fois que le projet sera compilé ou exécuté, Maven va chercher à mettre à jour la dépendance.

1.2.2 Notion de dépôt

Lors de la construction d'un projet Maven, si celle-ci se termine correctement et si cela est demandé par la construction du projet, l'artefact résultant (le fichier jar) est stocké dans un dépôt Maven. Un dépôt Maven local est un répertoire sur le poste du développeur, permettant de stocker, suivant l'arborescence des projets, tous les artefacts téléchargés depuis un dépôt distant ou construit par les projets du développeur.

Par défaut, le dépôt sur la machine locale de l'utilisateur est indiqué par $(user.home)/.m2/. Ce répertoire est modifiable dans la configuration de Maven.

Un projet ayant pour `POM` l'exemple suivant, sera stocké, dans le dépôt, suivant le chemin $(repository_home)/such/a/group/mon-artefact/1.0.0/.

```
<project>
       <modelVersion>4.0.0</modelVersion>
       <artifactId>mon-artefact</artifactId>
       <groupId>such.a.group</groupId>
       <version>1.0.0</version>
</project>
```

Un dépôt distant est un dépôt situé sur un serveur accessible par Maven. Certains outils permettent de définir et d'utiliser des dépôts distants, tel que Nexus.

1.3 Cycles de vie associés à un projet Maven

1.3.1 Cycle de vie par défaut de construction d'un projet

Chaque projet Maven est caractérisé par un cycle de vie de construction. Par défaut, ce cycle de vie se décompose en plusieurs étapes :

1. `compile` : compile le code source Java associé au projet,
2. `test` : teste le code source compilé en utilisant les tests unitaires présents dans le projet. Cette phase ne requiert pas le code soit packagé ou déployé,
3. `package` : package le code source compilé, par exemple en un fichier jar,
4. `install` : installe le package dans le dépôt local, pour pouvoir être utilisé en tant que dépendance par un autre projet par exemple,
5. `deploy` : effectuée dans un environnement adapté, cette phase déploie le package final pour pouvoir être utilisé par d'autres développements.

À chaque exécution de Maven sur l'une des étapes du cycle de vie du projet, l'ensemble des étapes en amont sont exécutées séquentiellement. Autrement dit, lors de l'utilisation du cycle de vie, Maven va premièrement valider le projet, puis compiler les fichiers sources associées au projet, exécuter les tests, packager les fichiers, exécuter les tests d'intégration sur ce paquet, vérifier le paquet, installer le paquet dans le dépôt local, et enfin déployer le paquet dans l'environnement spécifié.

Pour effectuer toutes ces phases, seule la dernière phase de cycle de vie a besoin d'être exécutée, via :

```
mvn deploy
```

1.3.2 Extensions et adaptations du cycle de construction par défaut

Le cycle de vie par défaut de Maven est extensible, par l'ajout de plugins et leur configuration. Par exemple, si l'on souhaite indiquer qu'il est nécessaire de compiler le projet en Java 1.6, alors il suffit d'ajouter des informations dans le `POM` permettant de modifier le plugin associé, comme suivant pour cet exemple :

```
<project>
	...
	<build>
		<plugins>
			<plugin>
				<groupId>org.apache.maven.plugins</groupId>
				<artifactId>maven-compiler-plugin</artifactId>
				<configuration>
					<source>1.6</source>
					<target>1.6</target>
				</configuration>
			</plugin>
		</plugins>
	</build>
</project>
```

Notons que pour tous les plugins du groupe `org.apache.maven.plugins` (plugins de Maven fournis par Apache), il n'est pas nécessaire d'indiquer la version, celle-ci étant généré à la volée par Maven.

1.3.3 Autres cycles de vie

Il existe deux autres cibles valides fournis par Maven pour un projet. La première, `clean`, permettant de nettoyer un projet, en effaçant tous les fichiers temporaires, créés par la construction du projet. En pratique, cette cible efface le dossier `target`. Si l'utilisateur souhaite que cette cible supprime d'autres fichiers ne se situant pas dans le dossier `target`, il est nécessaire de configurer le plugin.

Le seconde cible *site*, permet de construire le site web consacré au projet, répertoriant l'ensemble des informations associées au projet, telles que les dépendances, les développeurs, la licence de diffusion, ..., à partir du `POM` du projet. Il est possible de configurer cette cible pour adapter la feuille de style, la disposition, les informations recueillies, ...

2 Maven par la pratique

2.1 Installation de Maven

Télécharger l'archive apache-maven-2.2.1-bin.tar.gz située à l'adresse :

`http://maven.apache.org/download.html`

Suivre les installations d'instructions situées à l'adresse :

`http://maven.apache.org/download.html#Installation_Instructions`

2.2 Nexus, un proxy Maven

Nexus (`http://nexus.sonatype.org/`) est utilisé comme dépôt Maven interne à l'équipe. Il est situé à l'adresse suivante :

`http://172.20.65.33:8080/nexus`

Les login et mot de passe sont ceux du LDAP du Laboratoire.

Si vous obtenez un message vous disant que vous n'avez pas les droits pour accéder à Nexus, demander à une personne habilitée à vous les donner de le faire (attention, ce dépôt n'est pas géré par le LIFC mais uniquement par l'équipe ActiTest : voir donc une personne de l'équipe pour ces problèmes de droits).

Nexus gère plusieurs repositories (menu Views/Repositories → Repositories) :

- 6 repositories de type "proxy" : ce sont des repositories distants auquel Nexus accède pour aller chercher des librairies "classiques" (commons-*, junit, mockito...). En particulier, Maven Central est le dépôt Maven où l'on peut trouver la plupart des librairies,
- 3 repositories de type "hosted" : ce sont les repositories locaux :
  - Releases : dépôt dans lequel sont déposées les versions releasées des artefacts produits au LIFC,
  - Snapshots : dépôt dans lequel sont déposées les versions en cours de développement des artefacts produits au LIFC,
  - 3rd party : dépôt dans lequel sont déposées les librairies utilisées par les artefacts produits au LIFC et qui ne sont pas disponibles sur les repositories distants (par exemple, les librairies Smartesting ou encore des librairies disponibles sur les repositories distants mais pas dans la version souhaitée).

Les repositories de type "virtual" ne devraient a priori pas être utilisés.

Concernant les repositories locaux, vous ne devez pas ajouter manuellement vos artefacts sur les repositories releases et snapshots. Hudson se chargera de le faire. En revanche, vous pouvez ajouter manuellement des librairies dans le dépôt 3rd party. Pour cela, sélectionner le dépôt 3rd party dans la liste des repositories, puis dans la vue en bas de la fenêtre, sélectionner l'onglet "Artifact upload".

Dans la partie "Select GAV Definition Source", sélectionner GAV Parameters dans la liste déroulante (si l'artefact à uploader possède un POM, vous pouvez sélectionner "from POM", sinon sélectionner "GAV Parameters"). Renseigner les champs Group, Artifact, Version et Packaging.

Dans la partie "Select Artifact(s) for upload", cliquer sur le bouton "Select Artifact(s) to Upload..." puis sélectionner votre artefact. Cliquer ensuite sur le bouton "Add Artifact" et enfin sur le bouton "Upload Artifacts".

2.3 Configurations pour le LIFC

Afin d'éviter de télécharger les artefacts nécessaires à chaque compilation locale de vos modules, Maven va créer un dépôt local sur votre poste de travail dans lequel il va télécharger ces artefacts une seule fois (pour les artefacts snapshot, il ira consulter tout de même Nexus à chaque fois afin de savoir si une version plus récente de ces artefacts existe, et la télécharger le cas échéant). Par défaut, ce dépôt se trouve dans un dossier .m2 dans votre répertoire home (par exemple sous Windows : C :\Documents and Settings\user\.m2). L'emplacement de ce dossier est paramétrable : consulter la documentation Maven pour ce paramétrage.

Si vous n'avez jamais utilisé Maven auparavant, ce dossier n'existera pas encore. Créez-le. Ce dossier doit contenir la configuration d'accès au serveur Nexus. L'accès au serveur étant sécurisé, il faut créer deux fichiers :

– un fichier `settings.xml`, qui contient la configuration de Maven pour le poste local,
– un fichier `settings-security.xml`, qui gère les mots de passe présents dans le fichier précédent.

Voici un résumé des actions à effectuer pour crypter votre mot de passe. Les informations détaillées sur le cryptage des mots de passe sont disponibles à l'adresse :

`http://maven.apache.org/guides/mini/guide-encryption.html`

Pour pouvoir crypter votre mot de passe d'accès au serveur Nexus, Maven utilise un mot de passe "maître". Ce mot de passe est contenu dans le fichier `settings-security.xml`.

Pour crypter le mot de passe, "monMotDePasse", ouvrez une console et tapez la commande `mvn -encrypt-master-password monMotDePasse`.

```
pcbue@pcbue-laptop:~$ mvn --encrypt-master-password monMotDePasse
{/Ns9idx64SkHJM7jPXwVevz3zq5gU0C5uRGxsewfzsc=}
pcbue@pcbue-laptop:~$
```

Le résultat de cette commande est le mot de passe crypté. Stockez ce mot de passe crypté dans le fichier `settings-security.xml` de la manière suivante :

```
<settingsSecurity>
       <master>{/Ns9idx64SkHJM7jPXwVevz3zq5gU0C5uRGxsewfzsc=}</master>
</settingsSecurity>
```

Ce sont les seules informations que contiendra ce fichier `settings-security.xml`.

Crypter ensuite votre mot de passe pour le serveur Nexus avec la commande `mvn -encrypt-password monMotDePasseNexus`.

```
pcbue@pcbue-laptop:~$ mvn --encrypt-master-password monMotDePasse
{/Ns9idx64SkHJM7jPXwVevz3zq5gU0C5uRGxsewfzsc=}
pcbue@pcbue-laptop:~$ mvn --encrypt-password monMotDePasseNexus
{s2cS7fc/EQQHS8SaD3Sb+nPCr3CepnXZwwZi307N606emKebS8W4C6pOHRDB7Kfr}
pcbue@pcbue-laptop:~$
```

Vous obtenez un nouveau mot de passe crypté que vous placerez dans le fichier `settings.xml`, dans la balise `<password>`.

Ce fichier `settings.xml` contient également d'autres informations de connexion au dépôt Nexus. Le contenu à placer dans ce fichier (modulo le mot de passe qui sera celui que vous aurez généré) est le suivant.

```
<settings>
    <proxies>
        <proxy>
            <host>proxy-web.univ-fcomte.fr</host>
            <port>3128</port>
        </proxy>
    </proxies>

    <localRepository>/home/pcbue/.m2/repository</localRepository>

    <servers>
        <server>
```

```
            <id>Nexus-lifc</id>
            <username>pcbue</username>
            <password>{s2cS7fc/EQQHS8SaD3Sb+nPCr3CepnXZwwZi307N606emKebS8W4C6pOHRDB7Kfr}</password>
        </server>
    </servers>

    <mirrors>
        <mirror>
            <id>Nexus-lifc</id>
            <name>Lifc nexus public mirror</name>
            <url>http://172.20.65.33:8080/nexus/content/groups/public/</url>
            <mirrorOf>*</mirrorOf>
        </mirror>
    </mirrors>
</settings>
```

Une fois ces deux fichiers configurés, Maven pourra alors se connecter au dépôt Nexus depuis votre poste.

2.4 Définir un Job Hudson pour votre projet Maven

Vous pouvez vous connecter à Hudson à l'adresse suivante :

`http://172.20.65.33:8080/hudson/`

Attention, actuellement, les logins et mots de passe ne sont pas ceux du LDAP. Tout comme Nexus, Hudson est géré par l'activité "ActiTest" (voir donc un responsable de l'équipe pour la création de compte sur Hudson).

Voici la marche à suivre pour créer un job sur Hudson permettant de récupérer vos sources sur votre dépôt SVN, les compiler ainsi que compiler les tests, exécuter les tests, packager votre module et enfin le déployer sur le dépôt de partage de modules Nexus.

Dans le menu d'Hudson, cliquer sur "Nouvelle Tâche". Donner un nom à votre job, puis sélectionner "Construire un projet maven2". Cliquer ensuite sur OK.

La page de configuration du job apparaît alors. Quand vous aurez terminé cette configuration, cliquez sur le bouton "Sauver" en bas de la page. Vous pourrez ensuite à tout moment modifier cette configuration en cliquant sur le bouton "Configurer" situé dans le menu à gauche de la page.

Les parties suivantes correspondent aux différentes parties que vous retrouvez dans la page de configuration de votre job Hudson (sauf les deux premières sections qui correspondent à la partie de la configuration située avant la partie "Options avancées du projet").

2.4.1 Gestion des anciens builds et sélection du JDK

Afin d'éviter que la conservation des anciens builds n'utilise une quantité d'espace disque trop importante, il est préférable de cocher la case "Supprimer les anciens builds". Concernant le paramétrage, il est alors possible de spécifier le nombre de jours pendant lesquels un build doit être conservé, ou le nombre maximum de builds à conserver. A noter que le dernier build en succès et le dernier build stable seront de toute façon toujours conservés.

Dans la propriété JDK, vous pouvez également spécifier quel JDK vous souhaitez utiliser pour votre build. Actuellement, seul un JDK 6 est installé sur la machine sur laquelle Hudson est hébergé. Si vous avez besoin d'un autre JDK, demandez à une personne ayant les droits d'administration sur la machine de vous l'installer.

2.4.2 Activer la sécurité

Actuellement, les utilisateurs créés par l'administrateur n'ont que le droit de lecture sur Hudson. C'est le droit minimal afin de pouvoir accéder à l'interface d'Hudson. Cependant, ce droit ne donne pas accès (même en lecture) à aucun des jobs définis. Afin de donner des droits sur votre job à certains utilisateurs, il faut activer la "sécurité basée projet" pour ce job. Pour cela, il faut donc cocher la case correspondante "Activer la sécurité basée projet".

Dans la zone "Utilisateur/groupe à ajouter", tapez le login d'un utilisateur auquel vous souhaitez donner des droits sur votre job, puis cliquez sur le bouton "Ajouter". Dans le tableau des utilisateurs du job, une nouvelle ligne apparaît pour l'utilisateur nouvellement ajouté. Vous pouvez ensuite configurer les droits que vous souhaitez lui donner sur ce job :

- Job / Delete : permet de supprimer le job,
- Job / Configure : permet de configurer le job,
- Job / Read : permet d'accéder en lecture au job,
- Job / Build : permet de lancer un build pour ce job,
- Job / Workspace : permet d'accéder au workspace à partir duquel sont lancés les builds pour ce job (attention, le workspace contient également les sources du projet)
- Lancer / Delete : permet de supprimer un build,
- Lancer / Update : permet de donner des informations sur un build (description ou encore, pour un build en échec, écrire des notes permettant de donner les causes de l'échec du build).

2.4.3 Gestion de code source

Cette partie de la configuration permet de définir quel outil est utilisé pour la gestion du code source (CSV / Subversion), ainsi que l'adresse du dépôt sur lequel se trouve vote code source. Nous supposons ici que vous utilisez le gforge du LIFC.

Cochez donc le bouton "Subversion". Dans la zone "URL du repository", donnez l'adresse de votre dépôt svn (qui devrait donc débuter par ceci : https ://lifc-svn.univ-fcomte.fr/svn/...). Si Hudson ne s'est jamais connecté à l'URL donnée, il devrait vous demander de donner les identifiants permettant de s'y connecter. Entrez ces identifiants puis validez.

Il est préférable de cocher la case "svn update". Ceci permet de configurer votre job pour que, à chaque build, un update soit effectué plutôt qu'un checkout. Ceci permet ainsi de gagner du temps dans la réalisation du build. Attention cependant, avec un update, les anciens artefacts sont conservés.

2.4.4 Ce qui déclenche le build

Cette partie permet de configurer le mode de déclenchement du build. Suivant les dépendances de votre projet, vous pouvez configurer différemment cette partie.

Par défaut, vous pouvez simplement choisir "Scruter l'outil de gestion de version", et donner la fréquence à laquelle Hudson va aller scruter votre dépôt svn. Par exemple, le planning "*/5 * * * *" signifie qu'Hudson va aller scruter votre dépôt svn toutes les cinq minutes, et lancer un build dès qu'une modification aura été apportée sur ce dépôt (la syntaxe utilisée est celle de cron, cliquez sur le bouton en forme de point d'interrogation à droite de la case "planning" pour accéder à l'aide sur la syntaxe à utiliser pour définir ce planning).

Si votre projet possède une dépendance SNAPSHOT vers un projet construit et packagé sur Hudson, vous pouvez également cocher la case "Lancer un build à chaque fois qu'une dépendance SNAPSHOT est construite". Ainsi, après le build de ce projet SNAPSHOT (et son déploiement

sur Nexus), le build de votre projet sera effectué, prenant ainsi en compte la nouvelle version du projet SNAPSHOT.

De la même façon, il est possible de lancer votre job en aval d'un autre job (en particulier pour Hydra). Ainsi, lorsque le job en amont sera lancé, à la fin de celui-ci sera lancé votre job, afin de vérifier que le projet en amont est compatible avec le votre.

2.4.5 Build

Cette partie est essentielle puisqu'elle permet de faire le lien avec le fichier pom.xml de votre projet et définir quels goals Maven vous souhaitez réaliser :

- dans la zone "POM Racine", donnez le chemin de votre fichier pom.xml (relativement à l'URL donnée précédemment dans la zone "URL du repository"),
- dans la zone "Goals and options", définissez les goals Maven que vous souhaitez exécuter. L'un des rôles d'Hudson étant de déployer votre projet (packagé) sur Nexus, il faut donc exécuter le goal "deploy". Tapez donc : `clean deploy`.

Vous pouvez également définir une configuration avancée en cliquant sur le bouton "Avancé...". La partie MAVEN_OPTS permet notamment de donner des options à la JVM. En particulier, si vous notez que votre build est souvent en échec pour cause de "Java heap space", vous pouvez définir l'option Xmx de la JVM dans cette zone "MAVEN_OPTS".

2.4.6 Configuration du build

Si vous le souhaitez, vous pouvez cocher la case "Notification par email". Ceci permettra d'envoyer des mails à chaque build en échec ou instable. Les destinataires de ces mails peuvent être configurés :

- soit en spécifiant une liste de destinataires : dans ce cas, tous ces destinataires recevront un mail à chaque build en échec ou instable,
- soit en cochant la case "Envoyer des emails séparés aux personnes qui ont cassé le buid" : dans ce cas, seules les personnes responsables de l'échec / instabilité du build seront averties par email (Hudson doit a priori faire le lien entre le login "svn" de l'utilisateur qui a commité des modifications faisant échouer le build et l'utilisateur "Hudson". Cette option n'a toutefois jamais été expérimentée et reste donc à tester).

2.4.7 Actions à la suite du build

Dans cette partie, vous pouvez définir ce qui doit être effectué une fois que le build de votre projet a été réalisé.

En particulier, si vous voulez faire afficher les rapports de couverture de code (en utilisant Emma), vous pouvez cocher la case "Record Emma coverage report". Dans ce cas, il faut également penser à rajouter le goal "emma :emma" dans les goals à exécuter dans votre build, juste avant le goal "deploy", ou à rajouter le plugin `emma-maven-plugin` dans les plugins du `POM`. Dans le premier cas, dans la zone "Goals and options" de la partie "build", vous aurez donc :

```
clean emma:emma deploy
```

Si votre projet contient des spécifications Concordion et que vous souhaitez afficher les pages HTML de résultats d'exécution de ces spécifications, utilisez l'option "Publish HTML reports" (Hudson ne possède pour l'instant pas de plugin spécifique pour Condordion). Dans les zones de texte qui apparaissent, donnez le chemin du dossier qui contient les pages résultats ("HTML directory to archive", le nom de la page à afficher (index page[s]), et le nom du lien qui permettra d'accéder à cette page depuis la page de résultats de votre build ("Report title »).

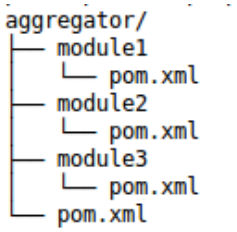
3 Foire à la configuration

Cette partie regroupe des configurations connues du `POM` pour contourner un problème simple d'utilisation ou de configuration d'un plugin particulier. Notons que la majorité des IDE proposent une complétion automatique à la volée des différentes informations possibles pour le `POM`. Cette complétion repose sur XSI. Les informations à rajouter pour la complétion des `POM` sont donc :

```
<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/xsd/maven-4.0.0.xsd" >
```

3.1 Projet multi-modules

Pour créer un projet regroupant plusieurs modules, il est nécessaire de passer soit par l'héritage de `POM`, soit par la création d'un projet multi-modules. Un projet multi-modules, ou `aggregator`, est un projet packagé sous la forme d'un `pom`, et regroupant l'ensemble des modules spécifiés. Le `POM` suivant illustre un `aggregator` en considérant l'arborescence suivante :




```
<project>
    <groupId>such.a.group</groupId>
    <artifactId>mon-artifact</artifactId>
    <version>1.0.0</version>
    <name>Aggregator</name>
    <packaging>pom</packaging>

    <modules>
        <module>module1</module>
        <module>module2</module>
        <module>module3</module>
    </modules>
</project>
```

De cette façon, en appelant n'importe quelle cible sur le projet `aggregator`, l'appel à cette cible sera propagé aux modules. Les projets `module1`, `module2` et `module3` peuvent également être eux-mêmes des `aggregators`. Si des dépendances existent entre les modules, alors Maven va calculer l'ordre de compilation des différents modules en fonction des dépendances déclarées.

Pour aller plus loin, `http://maven.apache.org/pom.html#Aggregation`

3.2 Déploiement d'artefacts

Le déploiement d'artefacts associé aux projets est effectué par Hudson. Toutefois, il est nécessaire de renseigner le `POM` afin qu'Hudson sache où il faut déployer les artefacts. L'extrait suivant montre ce qu'il faut ajouter au `POM` pour gérer le déploiement.

```
<project>
    ...
    <distributionManagement>
        <repository>
            <id>nexus</id>
            <name>Nexus-repository</name>
            <url>http://172.20.65.33:8080/nexus/content/repositories/releases/</url>
        </repository>
        <snapshotRepository>
            <id>nexus</id>
            <name>Nexus-repository-snapshot</name>
            <url>http://172.20.65.33:8080/nexus/content/repositories/snapshots/</url>
        </snapshotRepository>
    </distributionManagement>
    ...
</project>
```

ATTENTION !! : il est très important que le champ `id` des deux repository reste `nexus`. Cet identifiant est celui utilisé par Hudson pour pouvoir déployer les artefacts.

3.3 Encodage des fichiers du projet

Deux types d'encodage peuvent être configurés : encodage des ressources et encodage utilisé pour la compilation. Le premier est indiqué par une propriété du `POM`, le second par la configuration du plugin de compilation.

```
<project>
    ...
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    ...
    <build>
        ...
        <plugins>
            ...
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <encoding>ISO-8859-1</encoding>
                    ...
                </configuration>
            </plugin>
            ...
        </plugins>
    </build>
    ...
</project>
```

3.4 `emma-maven-plugin`

Le plugin `emma` peut être appelé de deux façons : par la ligne de commande avec `mvn emma:emma`, ou en l'ajoutant dans le `POM` du projet souhaité, ou il est alors automatiquement ajouté dans le cycle de vie au moment de la phase `test`.

```
<project>
    ...
    <build>
        ...
        <plugins>
            ...
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>emma-maven-plugin</artifactId>
            </plugin>
            ...
        </plugins>
    </build>
    ...
</project>
```

Pour aller plus loin, `http://emma.sourceforge.net/maven-emma-plugin/`

3.5 exec-maven-plugin

Ce plugin offre la possibilité d'exécuter le programme depuis, au choix, la machine virtuelle de Maven (`exec:java`), ou une machine virtuelle dédiée (`exec:exec`), depuis la ligne de commande ou en configurant le `POM` du projet, et en appelant, selon la configuration, l'une des commandes, :

```
mvn exec:exec
      ou
mvn exec:java
```

Pour aller plus loin, `http://mojo.codehaus.org/exec-maven-plugin/`

3.6 javacc-maven-plugin

Ce plugin permet de compiler des parsers JJTree/JavaCC en utilisant Maven. Pour cela, il est nécessaire de configurer le `POM`, ainsi que de placer les fichiers .jj (JavaCC) dans le répertoire src/main/javacc/ et les fichiers .jjt (JJTree) dans le répertoire src/main/jjtree/.

```
<project>
    ...
    <build>
        ...
        <plugins>
            ...
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>javacc-maven-plugin</artifactId>
                <version>2.6</version>
                <executions>
                    <execution>
                        <id>jjtree-monParser</id>
                        <goals>
                            <goal>jjtree-javacc</goal>
                        </goals>
                        <configuration>
                            <multi>true</multi>
```

```
                        <lookAhead>1</lookAhead>
                        ...
                    </configuration>
                </execution>
            </executions>
        </plugin>
        ...
    </plugins>
</build>
...
</project>
```

Pour aller plus loin, `http://mojo.codehaus.org/javacc-maven-plugin/`

3.7 maven-archetype-plugin

Cette tâche peut être utilisée pour générer l'arborescence et l'ensemble des fichiers nécessaire à un projet, selon l'environnement dans lequel il sera utilisé, ou ses fonctionnalités. Cette création, interactive, s'effectue par la commande :

```
mvn archetype:generate
```

À l'heure où ce document est écrit, un peu plus de 400 archétypes de projets sont disponibles, le plus commun étant `maven-archetype-quickstart`.

Pour aller plus loin, `http://maven.apache.org/archetype/maven-archetype-plugin/`

3.8 maven-assembly-plugin

Cette tâche est dédié à la création de bundles d'applications, permettant ainsi la livraison d'un logiciel complet. Cette tâche nécessite l'ajout d'informations dans le `POM` d'une part, et l'ajout d'un fichier spécifique, décrivant les éléments à ajouter au bundle.

Les informations à ajouter dans le `POM` sont, en général, les suivantes :

```
<project>
    ...
    <build>
        ...
        <plugins>
            ...
            <plugin>
                <artifactId>maven-assembly-plugin</artifactId>
                <executions>
                    <execution>
                        <id>mon-assembly</id>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                        <configuration>
                            <descriptors>
                                <descriptor>mon-assembly.xml</descriptor>
                            </descriptors>
                        </configuration>
                    </execution>
```

```
                </executions>
            </plugin>
            ...
        </plugins>
    </build>
    ...
</project>
```

En considérant la configuration précédente, il est nécessaire de fournir un fichier `mon-assembly.xml`, décrivant le bundle à construire. Dans le cas suivant, il s'agit d'un bundle sous la forme d'un dossier ayant le même nom que le module ayant lancé la création de l'assembly, regroupant une partie des modules utilisés dans le projet sous leur forme binaire (fichiers jar).

```
<assembly>
    <id>mon-assembly-description</id>
    <formats>
        <format>dir</format>
    </formats>
    <includeBaseDirectory>false</includeBaseDirectory>
    <moduleSets>
        <moduleSet>
            <includes>
                <include>group1.subgroup1:artifact1</include>
                <include>group1.subgroup2:artifact2</include>
                <include>group2.subgroup2:artifact3</include>
            </includes>
            <binaries>
                <outputDirectory>${artifactId}</outputDirectory>
                <unpack>false</unpack>
            </binaries>
        </moduleSet>
    </moduleSets>
</assembly>
```

Un schéma XSI est également disponible pour la construction de bundles.

```
<assembly    xmlns="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:schemaLocation="http://maven.apache.org/plugins/maven-assembly-plugin/assembly/1.1.2
                http://maven.apache.org/xsd/assembly-1.1.2.xsd" >
```

Pour aller plus loin, `http://maven.apache.org/plugins/maven-assembly-plugin/`

3.9 `maven-antrun-plugin`

Il est possible d'utiliser, dans les cas où aucune tâche Maven ne peut l'effectuer, un fragment de script Ant, afin de réaliser des tâches spécifiques.

Pour aller plus loin, `http://maven.apache.org/plugins/maven-antrun-plugin/`

3.10 `maven-compiler-plugin`

Il est possible de configurer la compilation du projet, notamment en indiquant les versions de Java à utiliser pour lire et compiler le projet, ainsi que l'encodage des fichiers source.

```
<project>
    ...
    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>
    ...
    <build>
        ...
        <plugins>
            ...
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <configuration>
                    <encoding>ISO-8859-1</encoding>
                    <source>1.6</source>
                    <target>1.6</target>
                </configuration>
            </plugin>
            ...
        </plugins>
    </build>
    ...
</project>
```

Pour aller plus loin, `http://maven.apache.org/plugins/maven-compiler-plugin/`

3.11 maven-install-plugin

Lorsqu'une dépendance d'un projet n'a pas d'équivalent sous Maven, il est possible d'installer le fichier Jar associé dans le dépôt de la machine locale. Cette installation s'effectue, en donnant le groupe, l'artefact, la version et le fichier à installer, par la commande :

```
mvn install:install-file  -Dfile=path-to-file.jar
                          -DgroupId=such.a.group
                          -DartifactId=monArtefact
                          -Dversion=1.0.0
                          -Dpackaging=jar
                          -DgeneratePom=true
```

Pour aller plus loin, `http://maven.apache.org/plugins/maven-install-plugin/`

3.12 maven-jar-plugin

Ce plugin permet, entre autres, d'ajouter des entrées dans le Manifest du fichier Jar (par exemple le Handler-Path nécessaire à Hydra).

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

```
                <configuration>
                    <archive>
                        <manifestEntries>
                            <Handler-Path>lifc.hydra.handler.MonHandler</Handler-Path>
                        </manifestEntries>
                    </archive>
                </configuration>
            </plugin>
            ...
        </plugins>
    </build>
    ...
</project>
```

Pour aller plus loin, `http://maven.apache.org/plugins/maven-jar-plugin/`

3.13 maven-javadoc-plugin

Ce plugin permet d'associer un fichier jar contenant la Javadoc du projet lors du déploiement par Hudson sur Nexus. Ce fichier peut ensuite être utilisé, par exemple par un IDE, pour visualiser la javadoc du projet.

```
<project>
    ...
    <build>
        ...
        <plugins>
            ...
            <plugin>
                <artifactId>maven-javadoc-plugin</artifactId>
                <executions>
                    <execution>
                        <id>ma-javadoc</id>
                        <phase>package</phase>
                        <goals>
                            <goal>jar</goal>
                        <goals>
                    </execution>
                </executions>
            </plugin>
            ...
        </plugins>
    </build>
    ...
</project>
```

Pour aller plus loin, `http://maven.apache.org/plugins/maven-javadoc-plugin/`

3.14 maven-source-plugin

Ce plugin permet d'associer un fichier jar contenant les fichiers source du projet lors du déploiement par Hudson sur Nexus. Ce fichier peut ensuite être utilisé, par exemple par un IDE, pour visualiser les sources du projet.

```
<project>
    ...
    <build>
        ...
        <plugins>
            ...
            <plugin>
                <artifactId>maven-source-plugin</artifactId>
                <executions>
                    <execution>
                        <id>ma-source</id>
                        <phase>package</phase>
                        <goals>
                            <goal>jar</goal>
                        <goals>
                    </execution>
                </executions>
            </plugin>
            ...
        </plugins>
    </build>
    ...
</project>
```

Pour aller plus loin, `http://maven.apache.org/plugins/maven-source-plugin/`