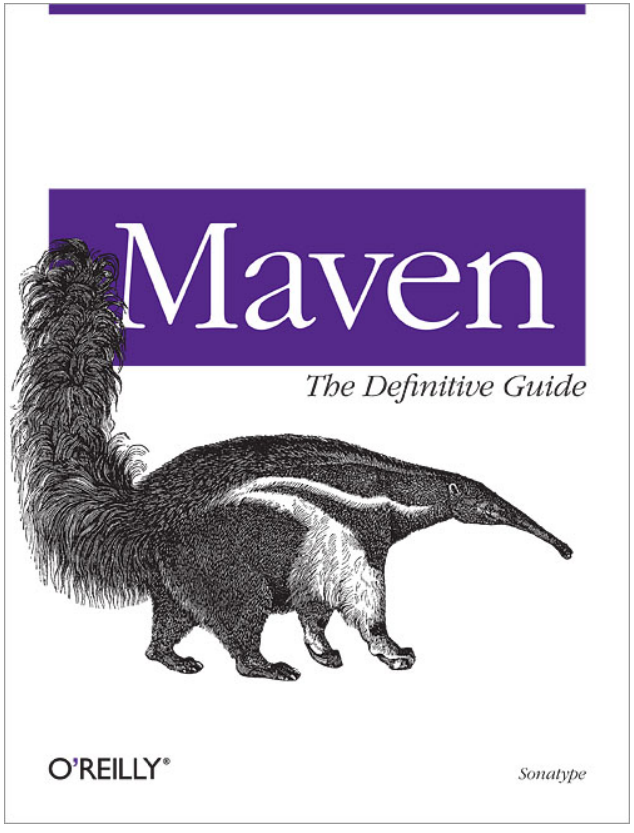
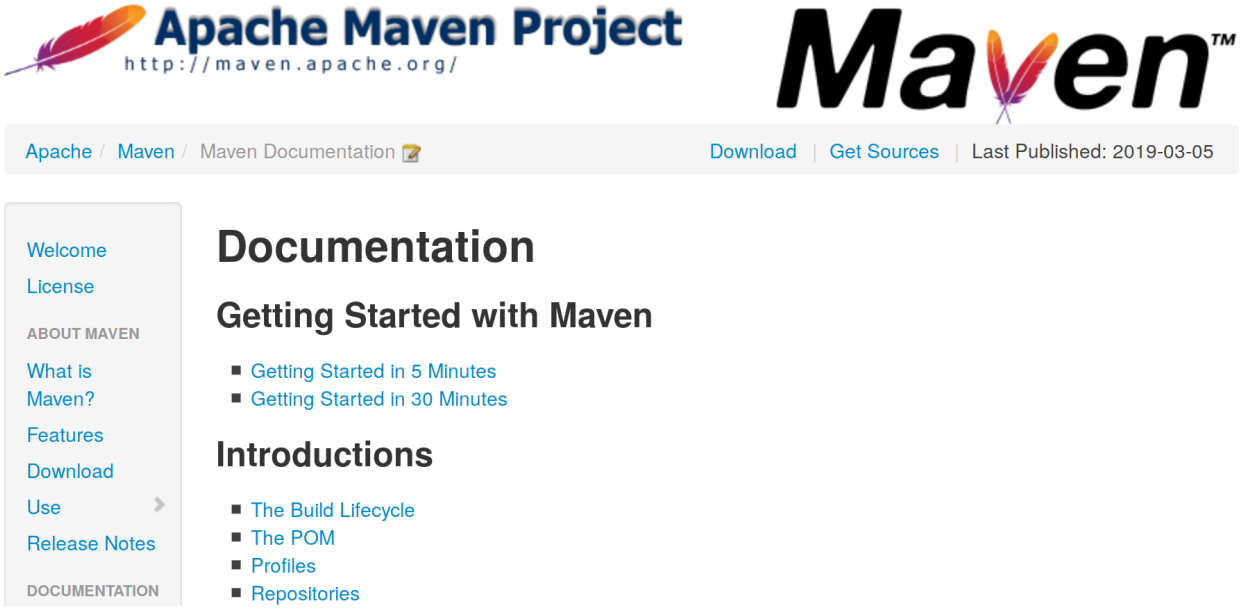
Développement d'applications modulaires en Java

Chouki Tibermacine

Chouki.Tibermacine@umontpellier.fr

Références bibliographiques





https://maven.apache.org/guides/

Plan du cours

1. Introduction : besoins d'automatisation

2. Cycle de vie, phases et goals

3. Gestion des dépendances

Au commencement, il y a avait Make

- Un système de build qui s'appelle *Make*
- Très utilisé, même de nos jours, avec C sous les systèmes Unix notamment
- Make permet d'écrire des *Makefiles* (fichiers texte) avec des règles de build (compilation, clean, ...)
- Un outil (commande `make`) interprète les règles
- Mais problème : pas très adapté à l'éco-système Java

Ensuite, il y a eu Ant

- Ant pour *Another Neat Tool* : un projet Apache
- Outil Java permettant d'interpréter des règles de build
- Les règles (`targets`) sont définies dans des fichiers `build.xml`
- Avantage : flexibilité (aucune structure imposée pour le projet)

Exemple avec Ant – `build.xml`

```
<project>
    <target name="clean">
        <delete dir="classes" />
    </target>

    <target name="compile" depends="clean">
        <mkdir dir="classes" />
        <javac srcdir="src" destdir="classes" />
    </target>

    <target name="jar" depends="compile">
        <mkdir dir="jar" />
        <jar destfile="jar/HelloWorld.jar" basedir="classes">
            <manifest>
                <attribute name="Main-Class"
                  value="HelloWorld" />
            </manifest>
        </jar>
    </target>

    <target name="run" depends="jar">
        <java jar="jar/HelloWorld.jar" fork="true" />
    </target>
</project>
```

Exemple avec Ant – A tester

- Écrire sur un éditeur de texte une classe `HelloWorld` avec une méthode `main`, qui affiche le message “Hello World” et placer le fichier `HelloWorld.java` dans un répertoire `src`
- Télécharger les fichier `build.xml` depuis le dépôt Git et le placer à côté du répertoire `src`
- Taper les commandes :
  1. `ant compile`
  2. `ant jar` et ensuite `java -jar jar/HelloWorld.jar`
  3. `ant run`

Mais Ant a vite montré ses limites

- Son avantage de flexibilité est devenu petit à petit son principal défaut :
  - Des fichiers de build trop verbeux (volumineux)
  - Trop complexes à comprendre et à maintenir
- Gestion des dépendances non fournie au départ :
  - Pourtant très importante, surtout dans les gros projets
  - Solution proposée : *Ivy* – un autre (sous-)projet d'Apache intégré à Ant
  - Mais les développeurs Java étaient petit à petit lassés d'utiliser cette paire d'outils

Et Maven est né ! ! !

- Un système de build, mais aussi de gestion de dépendances
- Il garde la syntaxe XML pour les fichiers de config
- Mais contrairement à Ant, Maven privilégie **la convention sur la configuration** (configurations/règles implicites)
- Ceci permet d'avoir des configurations plus concises
- Le fichier de config s'appelle par convention `pom.xml` (POM pour *Project Object Model*)

Un premier exemple avec Maven – `pom.xml`

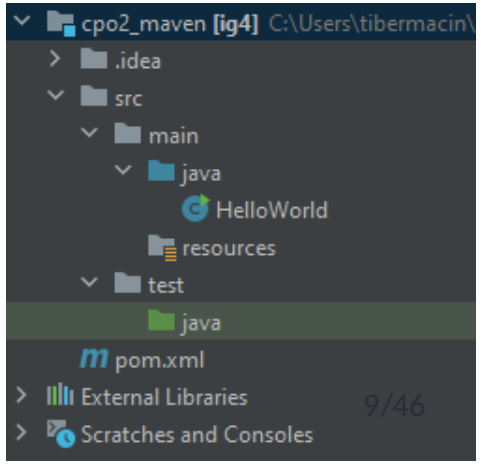
```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/
      xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ig4</groupId>
  <artifactId>HelloWorld</artifactId>
  <version>1.0-SNAPSHOT</version>
  <properties>
    <maven.compiler.source>16</maven.compiler.source>
    <maven.compiler.target>16</maven.compiler.target>
  </properties>
</project>
```

Généré automatiquement sur votre IDE :

Nouveau Projet Maven

Sur certaines IDEs, on doit ajouter :

`<properties> ... </properties>`



Un premier exemple avec Maven – A tester

- En ligne de commande, on peut utiliser `mvn` pour créer un nouveau projet avec un Archétype (*template de projet*) :

```
mvn -B archetype:generate
-DarchetypeGroupId=org.apache.maven.archetypes
-DgroupId=com.mycompany.app  -DartifactId=my-app
```

- Penser à ajouter dans le POM l'élément properties du slide précédent

Le `pom.xml` obtenu (`https://maven.apache.org`)

```
<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.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <name>my-app</name>
  <url>http://maven.apache.org</url>
  <properties>
    <maven.compiler.source>16</maven.compiler.source>
    <maven.compiler.target>16</maven.compiler.target>
  </properties>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

En ligne de commande : `mvn compile`

La structure obtenue (`https://maven.apache.org`)

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

- Notez la structure (générée automatiquement) : *src/main/java* et *src/test/java*
- Structure recommandée si création de projet à la main
- Après le `mvn compile`, qu'est-ce qui se passe ?

D'autres commandes à tester

- Compiler les tests (sans les exécuter) : `mvn test-compile`
- Compiler et exécuter les tests : `mvn test`
- Créer un jar : `mvn package` (par défaut, le packaging est fait dans un JAR)
- Installer le JAR dans l'entrepôt (repository) local (par défaut, c'est le répertoire `${user.home}/.m2/repository`) : `mvn install`
- Effacer le répertoire `target` et partir sur une base propre : `mvn clean`
- Plein d'autres commandes

Inclure des ressources dans le JAR

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- mycompany
    |   |           `-- app
    |   |               `-- App.java
    |   `-- resources
    |       `-- META-INF
    |           `-- application.properties
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

```
|-- META-INF
|   |-- MANIFEST.MF
|   |-- application.properties
|   `-- maven
|       `-- com.mycompany.app
|           `-- my-app
|               |-- pom.properties
|               `-- pom.xml
`-- com
    `-- mycompany
        `-- app
            `-- App.class
```

Inclure des ressources pour les tests dans le JAR

```
my-app
|-- pom.xml
`-- src
    |-- main
    |   |-- java
    |   |   `-- com
    |   |       `-- mycompany
    |   |           `-- app
    |   |               `-- App.java
    |   `-- resources
    |       `-- META-INF
    |           |-- application.properties
    `-- test
        |-- java
        |   `-- com
        |       `-- mycompany
        |           `-- app
        |               `-- AppTest.java
        `-- resources
            `-- test.properties
```

```
...

// Retrieve resource
InputStream is = getClass().getResourceAsStream( "/test.properties" );

// Do something with the resource

...
```

Retour sur la structure de base d'un POM

- **project** : élément racine
- **modelVersion** : version du modèle POM
- **groupId** : identifiant unique de l'organisation ou le groupe qui crée le projet (notation DNS inversée)
- **artifactId** : le nom unique de l'artefact généré par le projet (un JAR, ...). Un artefact type :
  $< artifactId > - < version > . < extension >$
  (ex. : *myapp* — 1.0.*jar*)
- **packaging** : type de packaging (JAR, WAR, ...). JAR par défaut
- **version** : version de l'artefact généré par le projet
- **name** : nom affiché
- **url** : le site du projet
- **description**

ça veut dire quoi *SNAPSHOT* ?

- C'est un suffixe du numéro de version
- Il indique que la version est en cours de développemnt
- Dans la version finale distribuée (la *release*) le mot *SNAPSHOT* est supprimé
- Ex : la version `1.0-SNAPSHOT` est distribuée comme release `1.0`
  Version de développement suivante : `1.1-SNAPSHOT`

Et Gradle ? le concurrent (futur substitut ?) de Maven

- Un autre système de build et de gestion de dépendances
- Objectif : Rendre le système flexible comme Ant, mais tout en ayant des fichiers de config petits comme dans Maven (*Convention over Configuration*)
- Plus de XML, mais un DSL (langage dédié) inspiré de la syntaxe Groovy ou Kotlin
- Multi-langages : Java, Scala, ...
- Les étapes de build s'appellent des tâches (*tasks*)
- Un système minimaliste couplé à un gros éco-système de plugins

Exemple avec Gradle – Fichier de config.

`build.gradle`

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

jar {
    baseName = 'gradleExample'
    version = '0.0.1-SNAPSHOT'
}

dependencies {
    compile 'junit:junit:4.12'
}
```

Plugin `java` pour pouvoir compiler (commande : `gradle classes`)

Plan du cours

1. Introduction : besoins d'automatisation

2. Cycle de vie, phases et goals

3. Gestion des dépendances

Cycle de vie d'un build

- Le processus de build et de distribution d'artefacts (un JAR par exemple) est explicitement défini par son "cycle de vie"
- Pour un développeur, il suffit de taper de simples commandes et dans le POM il y a tout le nécessaire pour obtenir les résultats voulus
- Trois types de cycles de vie :
  - `default` ou `build` : gère le déploiement du projet
  - `clean` : gère le nettoyage du projet
  - `site` : gère la création du site de documentation du projet
- Chaque cycle de vie est constitué d'une liste de *phases*

Phases d'un cycle de vie

- Pour le cycle de vie par défaut (**`default`** ou **`build`**)
  1. **`validate`** : valider que le projet est correct et que toute l'information nécessaire est disponible
  2. **`compile`** : compiler le code source du projet
  3. **`test`** : tester le code compilé en utilisant un framework de tests unitaires (jUnit par ex.)
  4. **`package`** : empaqueter le code compilé dans un format "distribuable", un JAR par ex.
  5. **`verify`** : exécuter n'importe quelles vérifications sur les résultats des tests d'intégration
  6. **`install`** : installer le paquet dans un *repo.* local pour être utilisé comme dépendance dans d'autres projets locaux
  7. **`deploy`** : fait dans l'environnement de build en copiant le package final dans un repo distant pour partager le projet avec d'autres développeurs/projets

Exécuter une phase via une commande

- Le fait d'exécuter une commande avec une phase va provoquer l'exécution de toutes les phases en amont dans le cycle de vie
- Exemple :
  - exécuter "`mvn install`"
  - provoque l'exécution de toutes les phases du cycle de vie par défaut, jusqu'à la phase `install` :
    `validate` > `compile` > `test` > `package` > `verify` > `install`
- On peut exécuter des commandes relevant de différents cycles de vie :
  - Exemple : "`mvn clean deploy`"
  - nettoie (cycle de vie `clean`) et ensuite déploie

Objectifs (*Goals*) des plugins

- En réalité, une phase est constituée d'"*objectifs*" contribués par des plugins
- Exemple :
  `mvn clean dependency:copy-dependencies package`
  - `clean` et `package` sont des phases auxquelles sont associées des objectifs implicites par défaut (`clean:clean` et `jar:jar`)
  - `copy-dependencies` est un objectif d'un plugin `dependency`
  - les objectifs sont exécutés dans l'ordre

Personnaliser le cycle de vie d'un projet

Deux façons de faire :

- personnaliser le packaging
  - Élément POM packaging à éditer (packaging JAR par défaut)
  - Pour le packaging JAR :

phase	**plugin :goal**
...	...
`compile`	`compiler:compile`
`test`	`surefire:test`
`package`	`jar:jar`
...	...

- ajouter des objectifs en configurant des plugins
  - Un plugin : un artefact contribuant avec des objectifs
  - Exemple : Plugin *Compiler* contribue avec deux objectifs `compile` pour compiler le code source du projet et `testCompile` pour compiler les classes de test

Personnaliser le cycle de vie d'un projet -suite-

Configurer un plugin

1. Ajouter un élément `plugin` dans le POM (sous l'élément `plugins`)
2. Indiquer les objectifs/*goals* (l'intégration seule du plugin ne suffit pas)

- Ces objectifs vont s'ajouter aux objectifs déjà affectés à la phase du cyle de vie (indiquée dans le POM sous l'élément `plugin`)
- Ils vont s'exécuter après ceux déjà affectés à la phase
- Il est possible de changer l'ordre (élément `executions`)

Personnaliser le cycle de vie d'un projet -suite-

Ex : rendre le JAR produit par la phase package exécutable. Après l'élément <dependencies>, ajouter :

```
...
<build>
 <plugins>
  <plugin>
   <groupId>org.apache.maven.plugins</groupId>
   <artifactId>maven-jar-plugin</artifactId>
   <configuration>
    <archive>
     <manifest>
      <mainClass>HelloWorld</mainClass>
     </manifest>
    </archive>
   </configuration>
  </plugin>
 </plugins>
</build>
```

Liste de plugins (`https://maven.apache.org/plugins/`)

Plugin	Type*	Version	Release Date	Description
Core plugins				**Plugins corresponding to default core phases (ie. clean, compile).**
`clean`	B	3.1.0	2018-04-14	Clean up after the build.
`compiler`	B	3.8.0	2018-07-26	Compiles Java sources.
`deploy`	B	3.0.0-M1	2018-09-23	Deploy the built artifact to the remote repository.
`failsafe`	B	3.0.0-M3	2018-12-23	Run the JUnit integration tests in an isolated classloader.
`install`	B	3.0.0-M1	2018-09-23	Install the built artifact into the local repository.
`resources`	B	3.1.0	2018-04-23	Copy the resources to the output directory for including in the JAR.
`site`	B	3.7.1	2018-04-27	Generate a site for the current project.
`surefire`	B	3.0.0-M3	2018-12-23	Run the JUnit unit tests in an isolated classloader.
`verifier`	B	1.1	2015-04-14	Useful for integration tests - verifies the existence of certain conditions.

* B pour Build Plugin ou R pour Reporting plugin

Liste de plugins (`https://maven.apache.org/plugins/`)

Packaging types/tools				**These plugins relate to packaging respective artifact types.**
`ear`	B	3.0.1	2018-05-09	Generate an EAR from the current project.
`ejb`	B	3.0.1	2018-05-03	Build an EJB (and optional client) from the current project.
`jar`	B	3.1.1	2018-12-08	Build a JAR from the current project.
`rar`	B	2.4	2014-09-08	Build a RAR from the current project.
`war`	B	3.2.2	2018-06-03	Build a WAR from the current project.
`app-client/acr`	B	3.1.0	2018-06-19	Build a JavaEE application client from the current project.
`shade`	B	3.2.1	2018-11-12	Build an Uber-JAR from the current project, including dependencies.
`source`	B	3.0.1	2016-06-18	Build a source-JAR from the current project.
`jlink`	B	3.0.0-alpha-1	2017-09-09	Build Java Run Time Image.
`jmod`	B	3.0.0-alpha-1	2017-09-17	Build Java JMod files.

Liste de plugins (`https://maven.apache.org/plugins/`)

Reporting plugins				**Plugins which generate reports, are configured as reports in the POM and run under the site generation lifecycle.**
`changelog`	R	2.3	2014-06-24	Generate a list of recent changes from your SCM.
`changes`	B+R	2.12.1	2016-11-01	Generate a report from an issue tracker or a change document.
`checkstyle`	B+R	3.0.0	2018-01-07	Generate a Checkstyle report.
`doap`	B	1.2	2015-03-17	Generate a Description of a Project (DOAP) file from a POM.
`docck`	B	1.1	2015-04-03	Documentation checker plugin.
`javadoc`	B+R	3.1.0	2019-03-04	Generate Javadoc for the project.
`jdeps`	B	3.1.1	2018-02-28	Run JDK's JDeps tool on the project.
`jxr`	R	3.0.0	2018-09-25	Generate a source cross reference.
`linkcheck`	R	1.2	2014-10-08	Generate a Linkcheck report of your project's documentation.
`pmd`	B+R	3.11.0	2018-10-26	Generate a PMD report.
`project-info-reports`	R	3.0.0	2018-06-23	Generate standard project reports.
`surefire-report`	R	3.0.0-M3	2018-12-23	Generate a report based on the results of unit tests.

Liste de plugins (`https://maven.apache.org/plugins/`)

Tools				**These are miscellaneous tools available through Maven by default.**
`ant`	B	2.4	2014-12-15	Generate an Ant build file for the project.
`antrun`	B	1.8	2014-12-26	Run a set of ant tasks from a phase of the build.
`archetype`	B	3.0.1	2017-04-11	Generate a skeleton project structure from an archetype.
`assembly`	B	3.1.1	2019-01-02	Build an assembly (distribution) of sources and/or binaries.
`dependency`	B+R	3.1.1	2018-05-19	Dependency manipulation (copy, unpack) and analysis.
`enforcer`	B	3.0.0-M2	2018-06-16	Environmental constraint checking (Maven Version, JDK etc), User Custom Rule Execution.
`gpg`	B	1.6	2015-01-19	Create signatures for the artifacts and poms.
`help`	B	3.1.1	2018-12-08	Get information about the working environment for the project.
`invoker`	B+R	3.2.0	2019-01-21	Run a set of Maven projects and verify the output.
`jarsigner`	B	3.0.0	2018-11-06	Signs or verifies project artifacts.
`jdeprscan`	B	3.0.0-alpha-1	2017-11-15	Run JDK's JDeprScan tool on the project.

Liste de plugins (`https://maven.apache.org/plugins/`)

`patch`	B	1.2	2015-03-09	Use the gnu patch tool to apply patch files to source code.
`pdf`	B	1.4	2017-12-28	Generate a PDF version of your project's documentation.
`plugin`	B+R	3.6.0	2018-11-01	Create a Maven plugin descriptor for any mojos found in the source tree, to include in the JAR.
`release`	B	2.5.3	2015-10-17	Release the current project - updating the POM and tagging in the SCM.
`remote-resources`	B	1.6.0	2018-10-31	Copy remote resources to the output directory for inclusion in the artifact.
`repository`	B	2.4	2015-02-22	Plugin to help with repository-based tasks.
`scm`	B	1.11.1	2018-09-11	Execute SCM commands for the current project.
`scm-publish`	B	3.0.0	2018-01-29	Publish your Maven website to a scm location.
`stage`	B	1.0	2015-03-03	Assists with release staging and promotion.
`toolchains`	B	1.1	2014-11-12	Allows to share configuration across plugins.

Plan du cours

1. Introduction : besoins d'automatisation

2. Cycle de vie, phases et goals

3. Gestion des dépendances

L'élément `dependencies/dependency` dans le POM

- Dans un exemple donné avant, il y avait une dépendance déclarée vers jUnit dans le POM :

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

  Cette dépendance externe est nécessaire pour le build du projet lors de la compilation, lors des tests ou à l'exécution (scope : *compile*, *test* ou *runtime*)

Où chercher les dépendances ?

- Au moment du build, Maven lit les dépendances dans le POM (il y en a qui sont héritées du "*Super POM*")
- Pour chaque dépendance, il va chercher la dépendance d'abord dans le repo local (par défaut, `${user.home}/.m2/repository`)
- Sinon, il la cherche (la télécharge et l'installe dans le repo local) depuis des repos distants
- Par défaut, le repo distant est *Maven Central* :

  `https://repo.maven.apache.org/maven2/`

  Certaines entreprises ont leur propre repo
- C'est quoi un repo. (entrepôt) : un endroit où sont stockés les artifacts (JAR entre autres) partagés par les projets Maven

Ajouter une dépendance

- On doit d'abord chercher les : *groupId*, *artifactId* et *version*
- Exemple : on veut produire des logs dans notre code et on va utiliser log4j pour ça
- En parcourant le repo Maven Central, on retrouve la description suivante (dans le fichier `maven_metadata.xml` de log4j) :

```
<metadata modelVersion="1.1.0">
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <versioning>
    <latest>1.2.17</latest>
    <release>1.2.17</release>
    <versions>
      <version>1.1.3</version>
      <version>1.2.4</version>
      ...
      <version>1.2.17</version>
    </versions>
    <lastUpdated>20140318154402</lastUpdated>
  </versioning>
</metadata>
```

Ajouter une dépendance -suite-

- On va ajouter ça au POM de notre projet :

```
...
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.17</version>
      <scope>compile</scope>
    </dependency>
...
```

- En faisant ensuite `mvn compile`, Maven va télécharger la dépendance pour nous et la mettre à disposition du compilateur (configure le CLASSPATH)

Déployer des artefacts dans un repo distant

- Il faudrait indiquer l'URL du repo dans le POM et indiquer un moyen d'authentifcation (settings.xml)
- Exemple avec scp :

```
<distributionManagement>
  <repository>
    <id>mycompany-repository</id>
    <name>MyCompany Repository</name>
    <url>scp://repository.mycompany.com/repository/maven2</url>
  </repository>
</distributionManagement>
```

```
1. <settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
2.   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.   xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
4.                       http://maven.apache.org/xsd/settings-1.0.0.xsd">
5.   ...
6.   <servers>
7.     <server>
8.       <id>mycompany-repository</id>
9.       <username>jvanzyl</username>
10.      <!-- Default value is ~/.ssh/id_dsa -->
11.      <privateKey>/path/to/identity</privateKey> (default is ~/.ssh/id_dsa)
12.      <passphrase>my_key_passphrase</passphrase>
13.    </server>
14.  </servers>
15.  ...
16. </settings>
```

Utiliser un repo interne

- Dans certaines entreprises on utilise un repo interne pour déployer des projets privés
- Il suffit d'installer un serveur de fichier ou Web organisé comme Maven Central (`http://repo.maven.apache.org/maven2/`)
- Ne pas scraper ou faire un rsync de Maven Central, sous peine d'être bloqué
- Plus d'explications/outils pour gérer un repo interne : `https://maven.apache.org/repository-management.html`
- Pour utiliser un repo interne :

```
<project>
  ...
  <repositories>
    <repository>
      <id>my-internal-site</id>
      <url>http://myserver/repo</url>
    </repository>
  </repositories>
  ...
</project>
```

Dépendances transitives

- Maven évite aux développeurs d'analyser et spécifier les dépendances requises par les dépendances d'un projet
- Il garantit de façon automatique des dépendances transitives
- Cette résolution des dépendances peut s'étendre assez largement (graphe de dép. peut être très large et profond)
- On peut avoir des problèmes d'ambiguité ou bien des problèmes avec des dépendances cycliques
- Ambiguité : Si dans le graphe de dépendances, on a :
  A → B → C → D[2.0]
  A → E → D[1.0]
  Maven prend D[1.0] (chemin le plus court) sinon il faut mettre la dépendance vers D[2.0] dans A

Exclure des dépendances

- Il est possible dans un projet d'exclure une dépendance transitive
- Ex : X → Y → Z
  Dans X on peut déclarer (dans un élément `exclusion` du POM) qu'on exclut Z
- Un autre moyen d'exclure des dépendances est de les déclarer optionnelles (élément `optional` du POM)
- Ex : Y → Z
  Dans Y, on peut déclarer Z comme dépendance optionnelle
  Si X → Y, il ne sera pas dépendant de Z

Expliciter les dépendances tout de même

- Supposons X ⟶ Y ⟶ Z
- X peut tout à fait utiliser le code dans Z
- Mais il est recommandé de l'expliciter dans les dépendances de X pour ne pas avoir des problèmes de build et pour améliorer la documentation de X
- Maven fournit un plugin `dependency` qui fournit un objectif `dependency:analyze` qui permet de vérifier les dépendances lors du build

Portée (*scope*) des dépendances

- **compile** (par défaut) : dépendances disponibles dans tous les classpaths du projet et sont propagées aux projets dépendants
- **provided** : ressemble beaucoup à *compile*, mais n'est pas transitive
- **runtime** : dépendances non requises pour la compilation
- **test** : dépendances requises pour la compilation et l'exécution des tests. N'est pas transitive
- **system** : comme *provided*, mais le JAR de l'artifact requis doit être fourni (n'est pas recherché dans un repo)
- **import** : dépendance doit être remplacée par ce qui est indiqué dans la section `<dependencyManagement>` d'un POM.

Dépendance entre modules dans un projet Maven

```
+- pom.xml
+- my-app
|  +- pom.xml
|  +- src
|     +- main
|        +- java
+- my-webapp
|  +- pom.xml
|  +- src
|     +- main
|        +- webapp
```

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

  <groupId>com.mycompany.app</groupId>
  <artifactId>app</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>

  <modules>
    <module>my-app</module>
    <module>my-webapp</module>
  </modules>
</project>
```

Dépendance entre modules dans un projet Maven

Dans `my-webapp/pom.xml`

```
1.  ...
2.  <dependencies>
3.    <dependency>
4.      <groupId>com.mycompany.app</groupId>
5.      <artifactId>my-app</artifactId>
6.      <version>1.0-SNAPSHOT</version>
7.    </dependency>
8.    ...
9.  </dependencies>
```

Ajouter `parent` dans le POM de chaque module

```
1.  <project xmlns="http://maven.apache.org/POM/4.0.0"
2.    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
4.                        http://maven.apache.org/xsd/maven-4.0.0.xsd">
5.    <parent>
6.      <groupId>com.mycompany.app</groupId>
7.      <artifactId>app</artifactId>
8.      <version>1.0-SNAPSHOT</version>
9.    </parent>
10.   ...
```