



Leuville Objects

Maven 3

Gérer le cycle de vie d'un projet Java

Plan

- Présentation
- Le POM
- Le cycle de vie d'un projet
- Gestion des dépendances
- Les référentiels
- Les plugins
- Les projets multi-modules
- Utilisation avancée
- Le processus de livraison
- Rapports et mesure de qualité
- Intégration continue

Présentation

Définition

- Logiciel de **gestion de projet** et de construction.
- Élément central dans la gestion d'une **infrastructure** de projet informatique.
- Maven est capable de gérer toute la vie du projet :
 - construction du livrable final,
 - gestion des bibliothèques logicielles de dépendances,
 - génération de la documentation et rapport du projet,
 - déploiement des versions de livraison sur les plateformes cibles

Apport de Maven

- Généralisé sur une grande majorité des projets Java Open Source (Spring, Hibernate, Struts...), Maven uniformise ainsi la gestion d'un projet Java.
- Maven offre la possibilité d'utiliser des outils qui permettent l'industrialisation du développement logiciel via la génération automatique de rapports ou l'utilisation de systèmes d'intégration continue.

Historique

- **2001**
 - Jason van Zyl, son créateur, propose le premier prototype au sein du projet *Apache Jakarta Alexandria*.
 - Usage réel avec le projet *Jakarta Turbine*
- **2003**
 - *Maven* devient *Apache Maven*, un projet principal de l'ASF
- **2004**
 - Sortie de *Apache Maven 1*
- **2005**
 - Sortie de *Apache Maven 2.0*
- **2010**
 - Sortie de *Apache Maven 3.0*

La communauté Maven

- L'équipe du projet Maven est composée de plus de 50 personnes.
- Maven est un projet communautaire de la fondation Apache (ASF).
- Il est sponsorisé en partie sous forme de main d'œuvre par d'autres sociétés comme *Sonatype*.
- *Sonatype* est la société fondée par le créateur de Maven, Jason Van Zyl.
- *Sonatype* intervient sur de nombreux aspect du projet et propose également de la documentation et des services pour étendre l'utilisation de Maven.

Le plugin *m2eclipse*

- Le plugin *m2eclipse* est le plugin officiel de la fondation Eclipse pour intégrer Maven dans son éditeur. Il est développé par la société *Sonatype*.

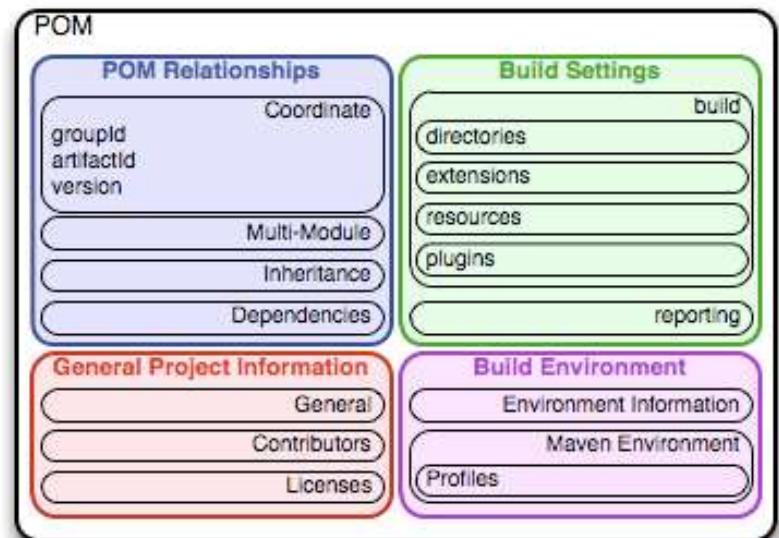
Le POM

Le POM (*Project Object Model*)

- Le POM est le descripteur d'un projet Maven.
- Fichier au format XML.
- Ce fichier, nommé **pom.xml**, définit un projet Maven.
- Il propose un modèle de données pour identifier les éléments indispensables du projet.

Format du fichier POM

- Le POM se compose de quatre catégories de description et de configuration :
 - Informations générales sur le projet,
 - Configuration du build,
 - Environnement du build,
 - Relations entre POM.



Source : <http://www.sonatype.com>

Exemple de POM minimal

- Les éléments obligatoires dans un fichier POM sont les suivants :
 - `modelVersion` : version de l'objet modèle utilisée dans ce POM.
 - `groupId` : indique l'identifiant unique de l'organisation ou groupe qui a créé le projet.
 - `artifactId` : indique le nom de base unique de l'artefact principal généré dans ce projet.
 - `version` : version de l'artefact générée par le projet.

```
<project xmlns="http://maven.apache.org/POM/4.0.0">  
  <modelVersion>4.0.0</modelVersion>  
  <groupId>org.formation</groupId>  
  <artifactId>maven-exemple</artifactId>  
  <version>1.0.0-SNAPSHOT</version>  
</project>
```

Les coordonnées Maven

- Les coordonnées Maven définissent un ensemble d'identifiants qui permet de définir de manière unique un projet, une dépendance ou un plugin.

```
<groupId>:<artifactId>:<packaging>:<version>
```

- Chaque projet Maven est identifié par ses coordonnées GAV :
 - Group (G)
 - Artifact (A)
 - Version (V)

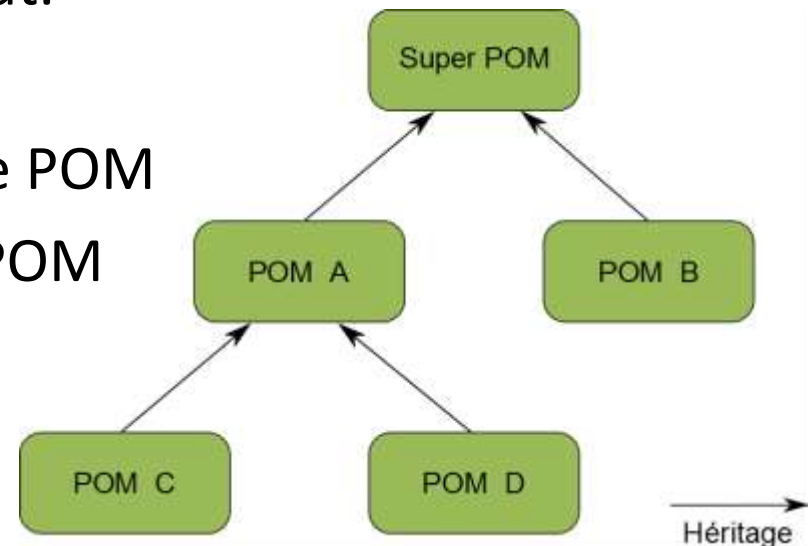
La notion d'artéfact

- Maven définit comme **artéfact** le fichier de sortie que le projet va générer au final.
- Chaque projet ne doit fournir qu'un seul artéfact dont le format est déterminé par la valeur de l'élément `<packaging />` du POM.
- Les recommandations de Maven précisent que la forme de l'artéfact de sortie est :

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

Le Super POM

- A l'instar de toutes les classes Java qui héritent implicitement de la classe `java.lang.Object`, tous les POM héritent d'un POM commun : le **Super POM**.
- Ce POM définit les valeurs par défaut.
- Toutes les valeurs présentes dans ce POM peuvent être surchargées dans les POM qui en héritent.



Le POM effectif

- Un POM Maven est la combinaison du Super POM, de tous les POMs parents intermédiaires et enfin du POM du projet en cours.
- Pour obtenir le POM effectif d'un projet, Maven commence par surcharger la configuration du Super POM avec un ou plusieurs POMs parents. Puis, il surcharge la configuration résultante avec les valeurs du POM du projet en cours.
- Il est possible d'obtenir le POM effectif d'un projet en exécutant la commande suivante :

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

- Le plugin *m2eclipse* propose un onglet pour voir le POM effectif d'un projet.



Le cycle de vie d'un projet

Structure d'un projet

- Maven impose des standards *modifiables* (contrairement à ant) :
 - Structure et organisation des fichiers,
 - Convention.
- Il est conseillé de respecter les standards de Maven autant que possible :
 - le fichier POM est plus court et plus simple grâce aux valeurs par défaut,
 - le projet est plus simple à comprendre, à maintenir lors de changement dans l'équipe,
 - l'intégration de plug-ins est plus simple.

Structure d'un projet (2)

- La structure d'un projet Maven de base est définie comme suit (la variable `${project.basedir}` est utilisée pour définir la racine du projet) :

Répertoire	Contenu
<code>\${project.basedir}/src/main/java</code>	Fichiers du code source Java
<code>\${project.basedir}/src/main/resources</code>	Ressources du projet (.properties, .xml ...)
<code>\${project.basedir}/src/test/java</code>	Fichiers du code source Java des tests
<code>\${project.basedir}/src/test/resources</code>	Fichiers de configuration des tests
<code>\${project.basedir}/target</code>	artéfact du projet
<code>\${project.basedir}/target/classes</code>	Classes compilées et ressources filtrées
<code>\${project.basedir}/target/test-classes</code>	Classes de test compilées et ressources filtrée
<code>\${project.basedir}/target/site</code>	Fichiers générés du site Web associé au projet

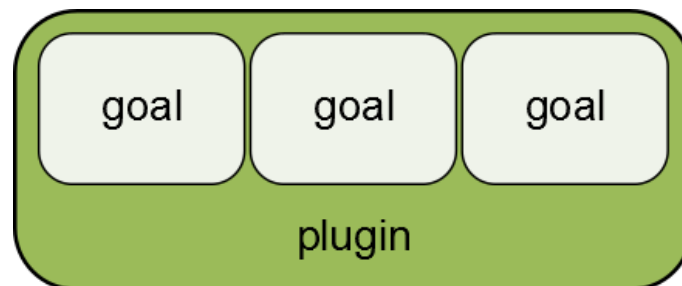
Structure d'un projet (3)

- Pour modifier l'emplacement par défaut de ces répertoires, il faut surcharger les valeurs définies dans le Super POM.

```
<build>
  <sourceDirectory>. . .</sourceDirectory>
  <testSourceDirectory>. . .</testSourceDirectory>
  <outputDirectory>. . .</outputDirectory>
  <testOutputDirectory>. . .</testOutputDirectory>
  <resources>
    <resource>
      <directory>. . .</directory>
    </resource>
  </resources>
  <testResources>
    <testResource>
      <directory>. . .</directory>
    </testResource>
  </testResources>
  . . .
</build>
```

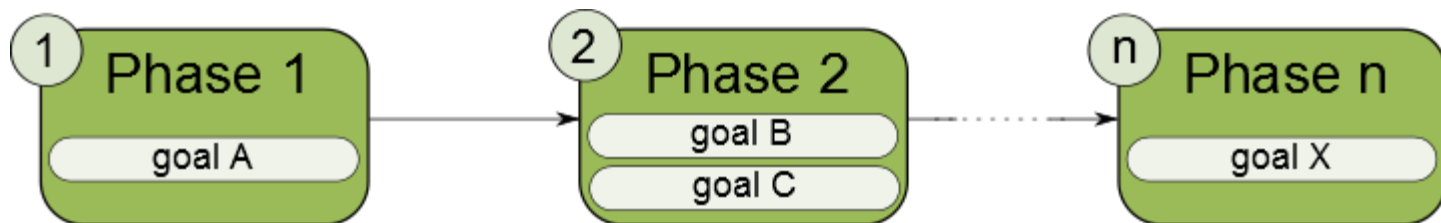
Les plugins

- Maven est principalement géré par des plugins : tous les traitements réalisés au sein d'un projet sont liés aux goals issus des plugins.
- Un **plugin** Maven se compose de plusieurs **goals**.
- Un goal est une tâche spécifique qui peut être exécutée individuellement ou combinée à d'autres goals.



Phases / Goals

- Maven liste toutes les étapes nécessaires à la réussite d'un projet.
- Ces étapes sont définies comme les **phases du projet**.
- Chaque phase peut réaliser des actions (**goals**).
- La succession des phases est définie comme le **cycle de vie du projet**.

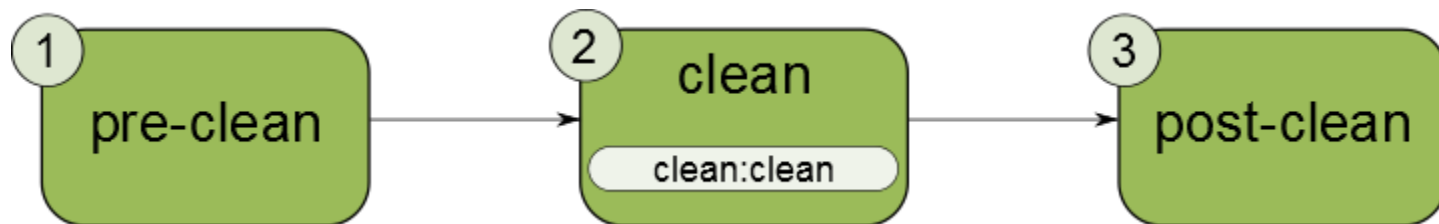


Les différents cycles de vie

- Maven a identifié 3 cycles de vie spécifique dans l'existence d'un projet :
 - Le cycle de vie pour le nettoyage du projet (*clean life cycle*).
 - Le cycle de vie par défaut (*default life cycle*).
 - Le cycle de vie pour le site du projet (*site life cycle*).

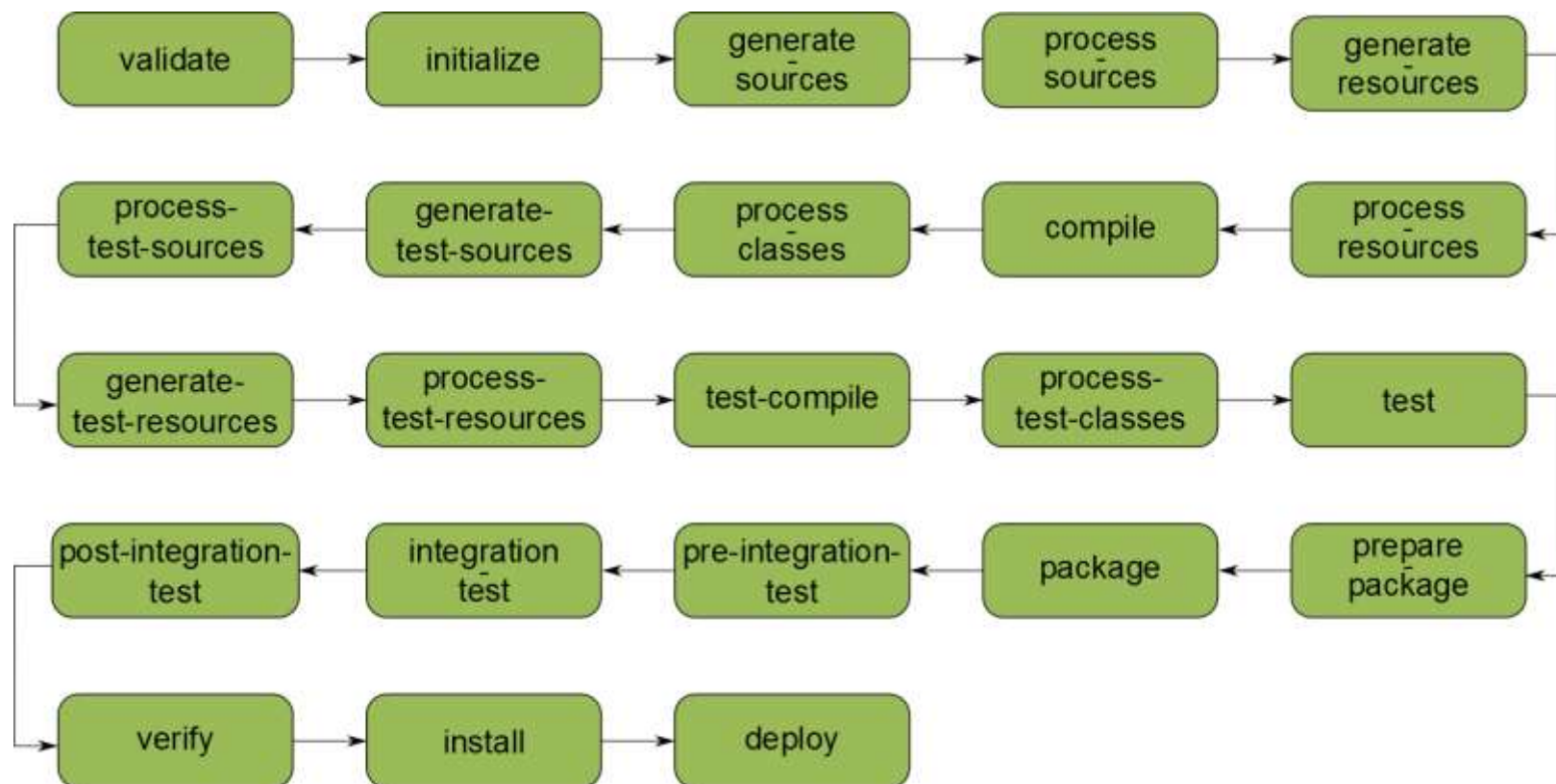
clean life cycle

- Cycle de vie qui assure d'avoir tous les dossiers de travail vides avant la construction d'un nouvel artéfact.
- Il définit **3 phases** :
 - **pre-clean** : initialisation du processus de nettoyage.
 - **clean** : suppression des fichiers générés par un build précédent.
 - **post-clean** : finalisation du processus de nettoyage.



Le cycle de vie par défaut

- Il est composé de **23 phases** :

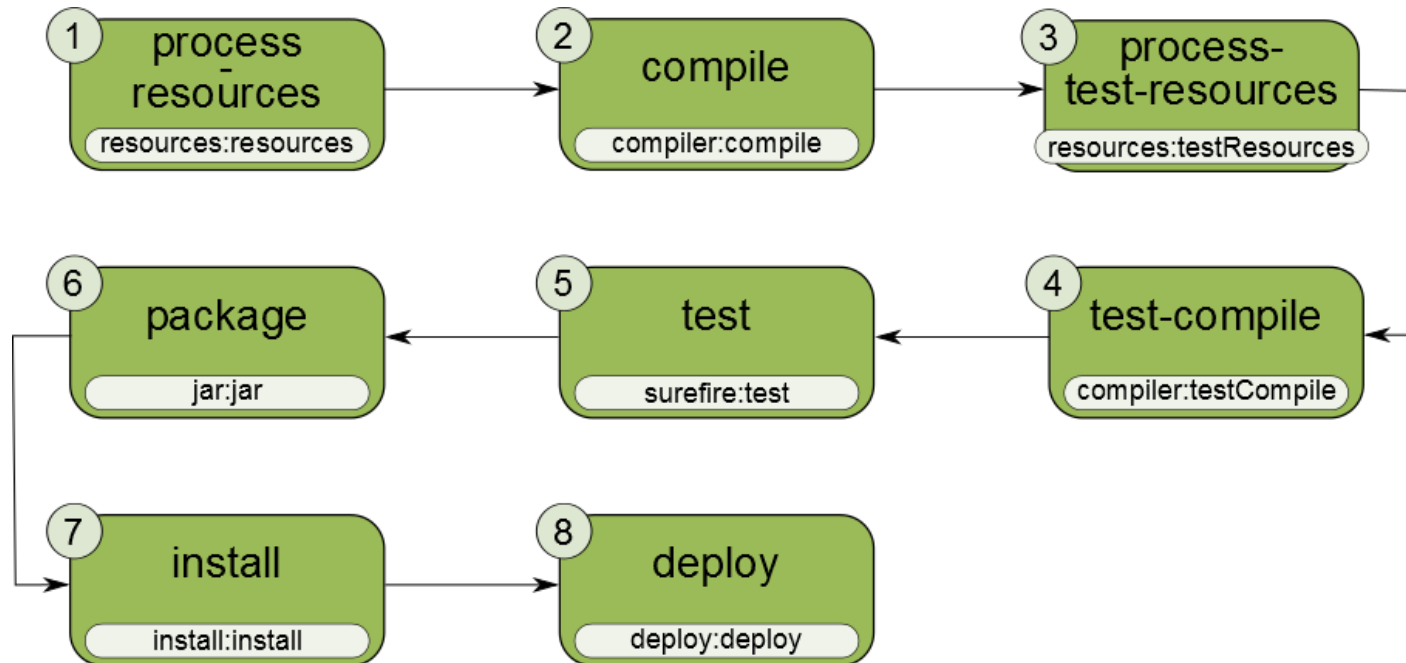


Le cycle de vie par défaut (2)

- Ces phases ne réalisent pas les mêmes opérations selon la configuration du projet.
- Chaque type de projet doit définir les goals à exécuter lors de chaque phase. C'est-à-dire associer ses propres traitements aux phases du cycle de vie par défaut.
- Il n'est pas obligatoire d'associer des goals à toutes les phases du cycle de vie par défaut.

Création d'une archive Java

- Le cycle de vie d'un projet pour la création d'une archive Java (JAR) redéfinit 8 phases du cycle de vie par défaut :



Création d'une archive Java (2)

Phases	Description	Plugins
process-resources	Copie les fichiers, hors code source, dans le répertoire de construction finale en les filtrant si nécessaire.	maven-resources-plugin
compile	Compile le code source du projet dans le dossier de construction du JAR.	maven-compiler-plugin
process-test-resources	Copie les fichiers de test, hors code source, dans le répertoire de construction finale en les filtrant si nécessaire.	maven-resources-plugin
test-compile	Compile les classes de tests.	maven-compiler-plugin
test	Exécute les tests unitaires.	maven-surefire-plugin
package	Crée l'archive JAR à partir du dossier contenant les sources compilées et les fichiers de ressources filtrés.	maven-jar-plugin
install	Copie l'archive JAR dans le référentiel local.	maven-install-plugin
deploy	Déploie l'archive JAR dans le référentiel distant.	maven-deploy-plugin

Le mode commande

- La syntaxe :

```
$ mvn [options] [<goal(s)>] [<phase(s)>]
```

- L'option **-D** permet de passer des paramètres à la JVM utilisés lors du traitement des commandes et notamment par les plugins :

```
$ mvn help:effective-pom -Doutput=pom-complet.xml
```

- Il existe également une option pour que Maven redirige tous les messages liés à la commande dans un fichier texte :

```
$ mvn -l sortie.log clean deploy
```

Gestion des erreurs

- Les options suivantes permettent de contrôler le comportement de Maven lors de l'échec d'un build :

```
$ mvn -X -e deploy
```

- X pour activer le **mode debug**,
 - e pour consulter la **trace Java complète de l'erreur**.
- Maven a mis l'accent sur l'affichage d'information lorsqu'une erreur se produit. Il est important d'étudier attentivement ces messages pour déterminer l'origine de l'erreur.

Exécuter un traitement

- Par exemple, il est possible d'exécuter un traitement de deux manières différentes :

- Par l'appel d'un goal : `$ mvn jar:jar`

Ici le traitement réalise simplement l'archivage dans un fichier jar du contenu du répertoire `target`. Le fichier peut alors être vide si le répertoire `target` du projet ne contient aucun fichier.

- Par l'appel d'une phase : `$ mvn package`

Ici toutes les phases précédant la phase `package` sont également exécutées. À partir du moment où le projet contient des sources Java, le fichier Jar ne peut être vide.

Gestion des dépendances

La gestion des dépendances

- La gestion des dépendances est un élément essentiel dans la configuration d'un projet Maven.
- Les librairies externes utilisées par le projet ne doivent être que des liens vers d'autres artefacts Maven et surtout pas copiées dans les répertoires du projet.
- Il est possible d'ajouter des dépendances dans le POM du projet. Une dépendance est identifiée comme tout artefact par un `groupId`, un `artifactId` et un numéro de version.

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-entitymanager</artifactId>
    <version>4.1.8.Final</version>
    <scope>compile</scope>
  </dependency>
</dependencies>
```

Les champs d'application (*scope*)

- Pour chaque dépendance, il faut définir son champs d'application (***scope***). Il précise la façon dont la dépendance est utilisée lors des différentes phases du projet.
- Cette information a des conséquences sur la présence ou non de la librairie lors de la compilation des classes du projet ou lors de la création de l'artéfact.
- Le champs d'application par défaut est **compile**.

Les différents champs d'application

Scope	Description
compile	La librairie est nécessaire dans le processus de compilation. La librairie est incluse dans le CLASSPATH lors de la compilation et lors de l'exécution.
provided	Similaire à compile à l'exception que la librairie est disponible dans le conteneur de déploiement de l'application. La librairie n'est pas incluse dans le CLASSPATH lors de l'exécution.
runtime	La librairie n'est pas nécessaire lors de la compilation des sources mais uniquement lors de l'exécution du projet.
test	La librairie est incluse dans le CLASSPATH pour la compilation et l'exécution des tests.
system	Permet au projet de dépendre d'une librairie qui ne se trouve pas dans un référentiel mais qui est disponible sur le serveur dans lequel se trouve le projet. Il fonctionne exactement comme provided .
import	Utilisé uniquement pour définir une dépendance vers un POM de type pom. Il permet d'inclure dans un POM la gestion des dépendances d'un autre POM.

Les dépendances transitives

- Maven permet de résoudre les dépendances transitives
- Si un artefact A dépend d'un artefact B qui dépend d'un artefact C, la résolution des dépendances de A trouvera B et C
 - Déclaration de la dépendance de l'artefact A dans le pom.xml
 - Maven se charge des artefacts B et C
- La commande suivante permet d'afficher les dépendances transitives :

```
$ mvn dependency:tree
```

```
[INFO] --- maven-dependency-plugin:2.1:tree (default-cli) @ mabiblio-persistence ---
[INFO] org.formation.mabiblio:mabiblio-persistence:jar:1.0.1-SNAPSHOT
[INFO] \- org.hibernate:hibernate-entitymanager:jar:4.1.8.Final:compile
[INFO]   +- org.jboss.logging:jboss-logging:jar:3.1.0.GA:compile
[INFO]   +- org.jboss.spec.javax.transaction:jboss-transaction-api_1.1_spec:jar:1.0.0.Final:compile
[INFO]   +- dom4j:dom4j:jar:1.6.1:compile
[INFO]   +- org.hibernate:hibernate-core:jar:4.1.8.Final:compile
[INFO]     \- antlr:antlr:jar:2.7.7:compile
[INFO]   +- org.hibernate.javax.persistence:hibernate-jpa-2.0-api:jar:1.0.1.Final:compile
[INFO]   +- org.javassist:javassist:jar:3.15.0-GA:compile
[INFO]   \- org.hibernate.common:hibernate-commons-annotations:jar:4.0.1.Final:compile
```

Gérer les conflits de dépendances

- Lors de la résolution de la transitivité sur un projet, il est possible que le graphe donne lieu à des conflits entre plusieurs dépendances.
- Un conflit intervient lorsque au moins deux dépendances avec les mêmes coordonnées sont disponibles dans l'arborescence dans plusieurs versions différentes.
- Un conflit entre deux dépendances directes se règle en supprimant la dépendance qui pose problème.
- Dans le cas d'un conflit entre deux dépendances directes issues de POM différents avec l'héritage, c'est la version de la dépendance déclarée dans le POM enfant qui a la priorité sur la version déclarée dans le POM parent.

Gérer les conflits de dépendances (2)

- La grande majorité des conflits apparaît avec les dépendances transitives.
- Maven définit des règles afin de gérer les cas de potentiels conflits.
- L'ordre de critères de priorité est le suivant :
 - La dépendance transitive la plus proche du projet en termes de profondeur d'arborescence.
 - La dépendance transitive issue de la dépendance directe déclarée en premier dans l'ordre des dépendance du POM.

Les référentiels

Les référentiels

- Un référentiel est un emplacement sur un système de fichiers qui contient des artefacts organisés selon une arborescence spécifique à Maven (basée sur les notions de `groupId`, `artifactId` et `version`) et indexées à l'aide de fichiers XML (`metadata.xml`).
- Il existe deux types de référentiels :
 - Le référentiel local (*local repository*),
 - Les référentiels distants (*remote repository*).

Le référentiel local

- Le **référentiel local** est l'emplacement sur le système de fichiers local où sont stockées et indexées toutes les bibliothèques et plugins utilisés par Maven et les projets en cours de développement sur la machine locale.
- L'emplacement par défaut du référentiel local est `${user.home}/.m2/repository`
- Il est possible de modifier cet emplacement dans le fichier `settings.xml` :

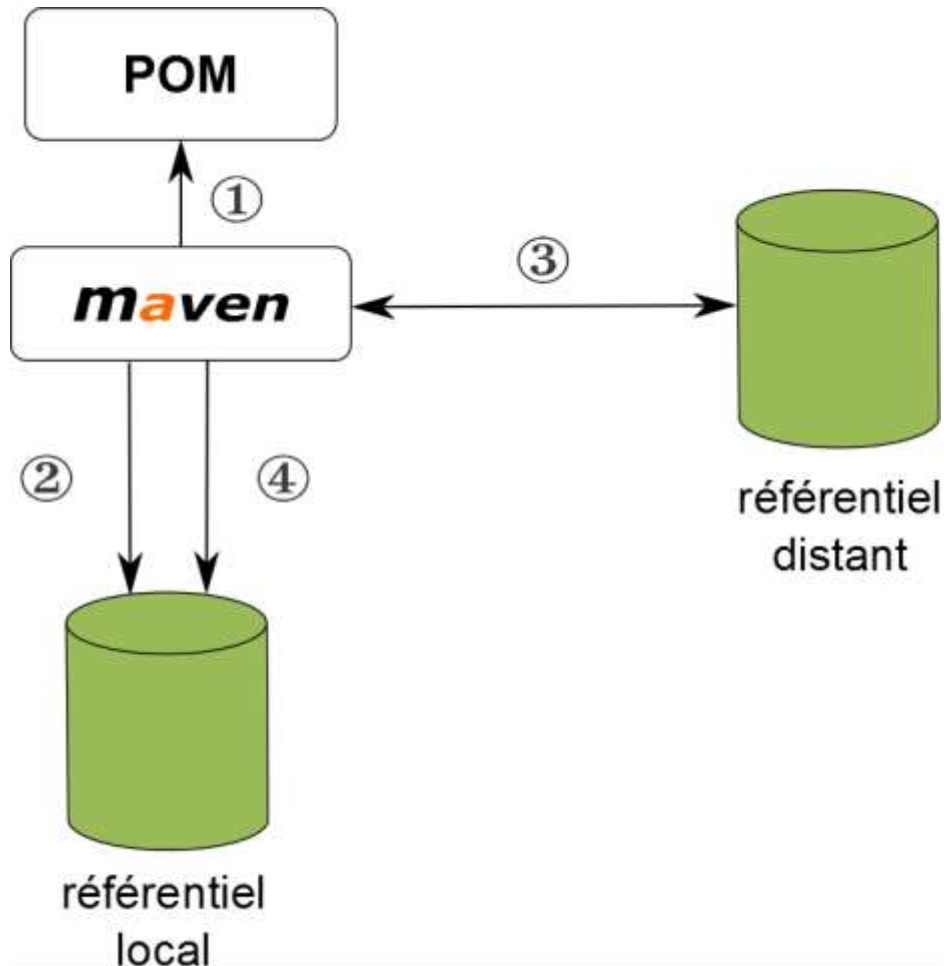
```
<localRepository>C:\Formations\Maven3\repo</localRepository>
```

Les référentiels distants

- Un **référentiel distant** permet d'alimenter le référentiel local.
- Maven différencie les référentiels contenant les plugins de ceux contenant les librairies.
- Le Super POM définit deux référentiels par défaut : un pour les librairies et un pour les plugins (voir ci-contre).
- Il est possible d'ajouter d'autres référentiels distants dans un POM.

```
<repositories>
  <repository>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <id>central</id>
    <name>Central Repository</name>
    <url>http://repo.maven.apache.org/maven2</url>
  </repository>
</repositories>
<pluginRepositories>
  <pluginRepository>
    <releases>
      <updatePolicy>never</updatePolicy>
    </releases>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
    <id>central</id>
    <name>Central Repository</name>
    <url>http://repo.maven.apache.org/maven2</url>
  </pluginRepository>
</pluginRepositories>
```

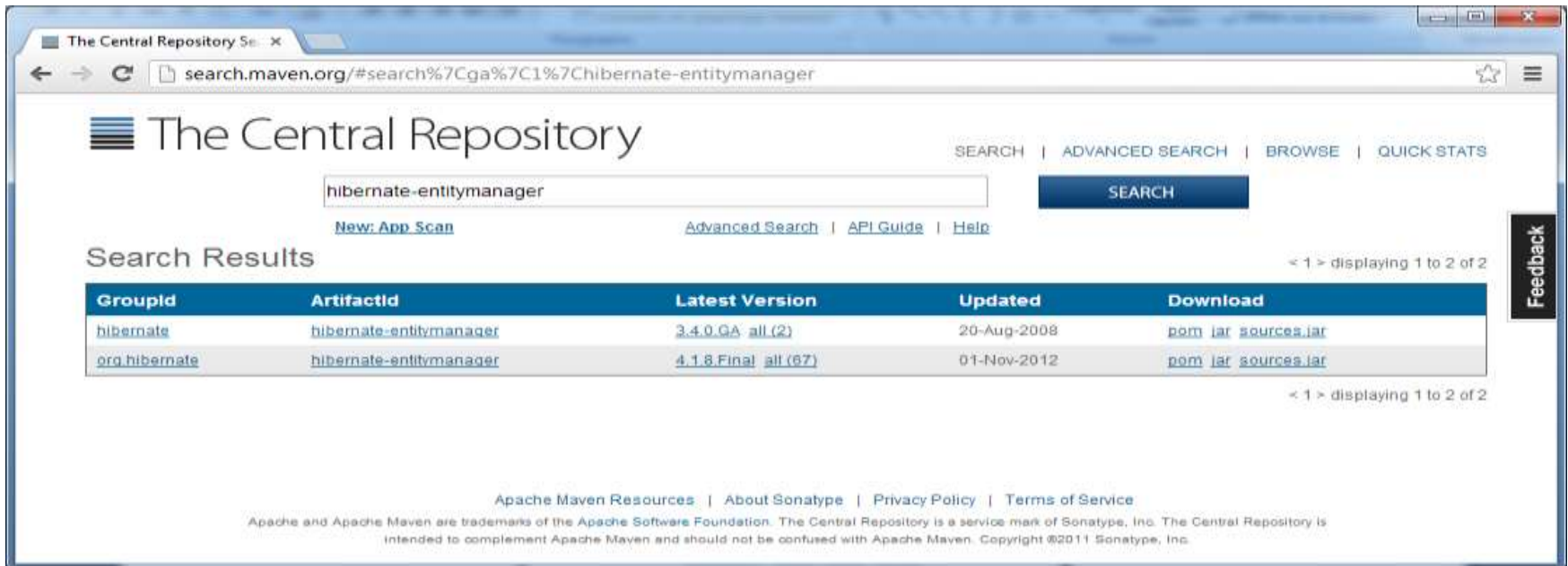
Communications local /distant



- ① Lecture du POM.
- ② Vérification de l'existence des dépendances dans le référentiel local.
- ③ Téléchargement des dépendances non trouvées.
- ④ Copie des dépendances dans le référentiel local.

Le référentiel *Maven Central*

- Le référentiel distant le plus utilisé et le plus complet pour la gestion des dépendances est le référentiel *central*.
- Une application Web est disponible à l'adresse <http://search.maven.org> et propose de rechercher des artéfacts disponibles dans ce référentiel.



The screenshot shows the Maven Central search results for the artifact `hibernate-entitymanager`. The page includes a search bar with the query, navigation links (SEARCH, ADVANCED SEARCH, BROWSE, QUICK STATS), and a table of results. The table lists two versions: 3.4.0.GA and 4.1.8.Final, both with download links for pom, jar, and sources.

GroupId	ArtifactId	Latest Version	Updated	Download
hibernate	hibernate-entitymanager	3.4.0.GA all (2)	20-Aug-2008	pom jar sources.jar
org.hibernate	hibernate-entitymanager	4.1.8.Final all (67)	01-Nov-2012	pom jar sources.jar

Gestion des artefacts

- Lors de l'installation ou du déploiement d'un projet *Maven* dans un référentiel, une certaine quantité de fichiers sont impactés par le processus.
- Ces fichiers permettent à Maven d'identifier précisément les artefacts présents dans un référentiel et de savoir s'il est nécessaire de les télécharger d'un référentiel distant vers le référentiel local
- La gestion des artefacts est différente entre un référentiel local et un référentiel distant.

Installation dans un référentiel local

- L'installation d'un artéfact dans le référentiel local se fait lors la phase `install`.
- le référentiel local doit être considéré comme un emplacement de stockage temporaire qui peut être vidé de son contenu sans conséquence majeure. Il ne doit en aucun cas contenir des données du projet non stockées dans un emplacement de sauvegarde officiel.
- Lors de chaque nouvelle installation en local, l'artéfact précédent est supprimé et remplacé par le nouveau. Les dates sont mises à jour dans les fichiers XML.

Les gestionnaires de référentiels

- Un gestionnaire de référentiels est un logiciel qui permet de centraliser et de masquer la complexité liée aux référentiels distants.
- Il existe sur le marché trois principaux logiciels qui assurent ce rôle : *Apache Archiva*, *Artifactory* et *Sonatype Nexus*.

Les gestionnaires de référentiels (2)

- La mise en place d'un gestionnaire de référentiels offre de multiples avantages pour une équipe projet.
- Les postes de développement ne sont plus dépendants d'une connexion Internet pour réaliser leurs traitements.
- Le gestionnaire gère un cache pour chaque référentiels distants afin de limiter les requêtes HTTP externes.

Déploiement dans un référentiel distant

- Les adresses réseaux cibles pour le déploiement d'artéfacts sont configurés dans l'élément `<distributionManagement />` du POM
- Alors que les données d'authentification pour accéder à ces adresses sont configurées dans l'élément `<servers />` du fichier `settings.xml`.
- Le déploiement d'artéfact dans un référentiel distant ne doit pas être effectué à partir de postes de développement. Un serveur doit être dédié à la mise en œuvre de ces procédure.

Déclaration d'un référentiel distant

```
<projet>
...
<distributionManagement>
  <repository>
    <id>releases</id>
    <name>Internal releases</name>
    <url>http://localhost:8080/nexus/content/repositories/releases/</url>
  </repository>
  <snapshotRepository>
    <id>snapshots</id>
    <name>Internal snapshots</name>
    <url>http://localhost:8080/nexus/content/repositories/snapshots/</url>
  </snapshotRepository>
</distributionManagement>
...
</projet>
```

pom.xml

settings.xml

```
<settings>
...
  <servers>
    <server>
      <id>releases</id>
      <username>cyril</username>
      <password>pass</password>
    </server>
    <server>
      <id>snapshots</id>
      <username>cyril</username>
      <password>pass</password>
    </server>
  </servers>
...
</projet>
```

Les protocoles pour le déploiement

- Pour gérer les artefacts vers les référentiels distants, Maven utilise la bibliothèque abstraite `Maven Wagon`.
- Cette bibliothèque possède plusieurs implémentations selon le protocole de transfert à mettre en œuvre.
 - HTTP/HTTPS
 - SCP (SSH)
 - WebDav
 - FTP
- Maven supporte par défaut uniquement le déploiement avec le protocole HTTP.
- Il est possible de mettre en œuvre d'autres protocoles en configurant le POM du projet avec l'élément `<extensions />`

Sécuriser les déploiements

- Le renforcement de sécurité est disponible dans la gestion des mots de passe.
- Un mot de passe identifié comme le mot de passe principal (*master*) est créé.

```
mvn -emp <password>
```

- Ce mot de passe est chiffré et stocké dans le fichier settings-security.xml au même niveau que le fichier settings.xml.
- Les mots de passe présents dans le fichier settings.xml sont chiffrés en utilisant la valeur chiffrée du mot de pass principal

```
mvn -ep <password>
```

Les plugins

Utilisation d'un plugin

- Les plugins sont configurés par l'intermédiaire de propriétés définies par goals.
- Pour afficher la description d'un plugin, il est possible d'utiliser la commande suivante :

```
mvn help:describe -Dcmd=compiler:compile -Ddetail
```

Utilisation d'un plugin (2)

- Pour chaque plugin, il existe un site de documentation



Codehaus / Mojo / Exec Maven Plugin / Exec Maven Plugin - Introduction

Overview

[Introduction](#)

[Goals](#)

[Usage](#)

[FAQ](#)

Examples

[Running Java programs with exec:exec](#)

[Changing the classpath scope when running Java programs](#)

[Using plugin dependencies with exec:exec](#)

Project Documentation

▼ [Project Information](#)

Exec Maven Plugin

The plugin provides 2 goals to help execute system and Java programs.

Goals Overview

General information about the goals.

- [exec:exec](#) execute programs and Java programs in a separate process.
- [exec:java](#) execute Java programs in the same VM.

Usage

General instructions on how to use the Exec Maven Plugin can be found on t

In case you still have questions regarding the plugin's usage, please feel free the answer to your question as part of an older thread. Hence, it is also worth

If you feel like the plugin is missing a feature or has a defect, you can fill a fe comprehensive description of your concern. Especially for fixing bugs it is an

Configuration d'un plugin

- Configuration des paramètres globaux d'un plugin :

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <configuration>
    <mainClass>org.formation.maven.mabiblio.App</mainClass>
  </configuration>
</plugin>
```

- Modification des paramètres spécifique à une exécution :

```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>exec-maven-plugin</artifactId>
  <executions>
    <execution>
      <phase>test</phase>
      <goals>
        <goal>java</goal>
      </goals>
      <configuration>
        <mainClass>org.formation.maven.mabiblio.App</mainClass>
      </configuration>
    </execution>
  </executions>
</plugin>
```

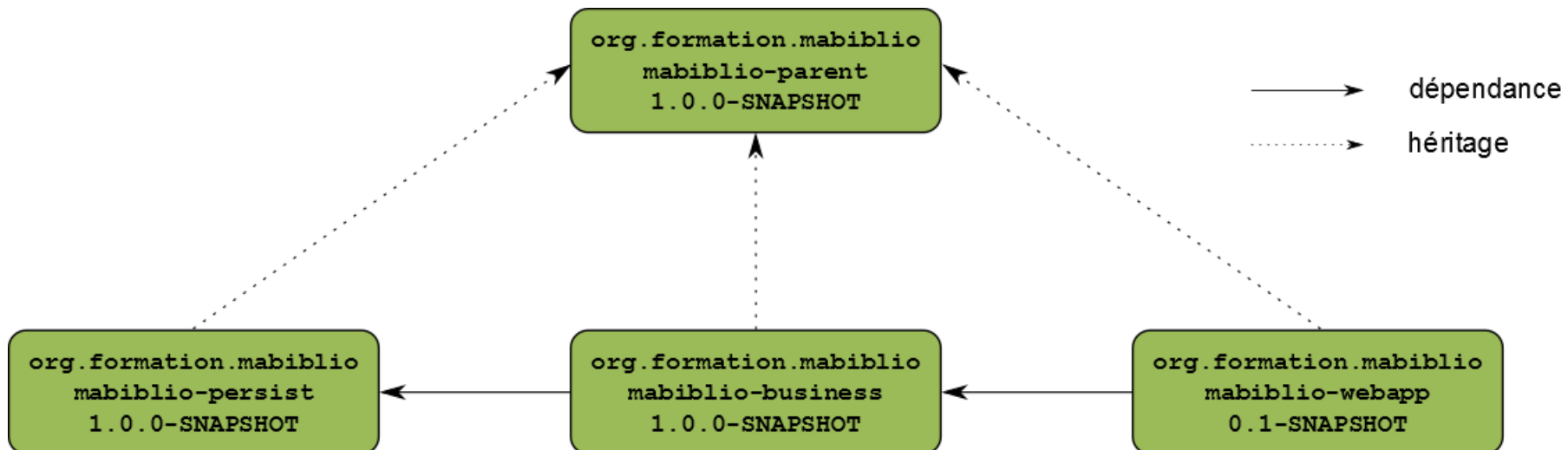

Définition d'un plugin

- Un plugin Maven est un artefact Maven qui contient un descripteur de plugin et un ou plusieurs Mojos.
- Un **Mojo** peut se comparer à un goal Maven.
- Derrière chaque goal se cache un Mojo.
 - Le goal `compiler:compile` correspond à la classe `CompilerMojo` dans le plugin Maven Compiler
 - le goal `jar:jar` correspond à la classe `JarMojo` du plugin Maven Jar
- L'écriture d'un plugin consiste à regrouper un ensemble de Mojos (ou goals) dans un seul artefact.

Héritage et multi-modules

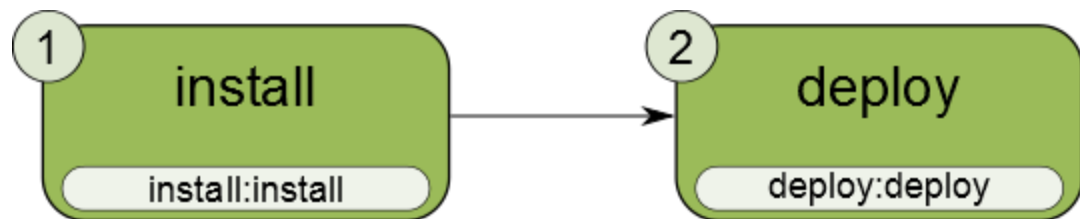
Le besoin d'héritage

- Le développement d'une application peut nécessiter la création de plusieurs projets Maven.
- Plusieurs informations peuvent être redondantes dans les POM des différents projets.
- Au même titre que tous les POM héritent du Super POM, il devient nécessaire de créer un POM parent commun à tous les POM de l'application.



Le type POM

- Pour créer un POM dont le but unique est de factoriser les informations communes à tous les projets, Maven propose le **type de projet pom**.
- Ce type de projet possède uniquement le fichier `pom.xml`.
- Le cycle de vie d'un tel projet est minimal et contient uniquement deux phases :



- L'artéfact de sortie est le fichier `pom.xml`.

Le Corporate POM

- Le POM parent à tous les projets d'une application est appelé le Corporate POM.
- Il est souvent identifié par un `artifactId` contenant le mot clé ***parent***.
- La mise en place d'un Corporate POM est l'occasion de renseigner les informations générales de l'application.

```
<project>
  . . .
  <groupId>org.formation.mabiblio</groupId>
  <artifactId>mabiblio-parent</artifactId>
  <version>1.0.0-SNAPSHOT</version>
  . . .
</project>
```

Héritage d'un POM

- Les POM des projets enfants doivent déclarer en premier lieu les coordonnées du POM parent afin d'activer le processus d'héritage grâce à l'élément `<parent />`.

```
<project>
  <parent>
    <groupId>org.formation.mabiblio</groupId>
    <artifactId>mabiblio-parent</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </parent>
  <artifactId>mabiblio-persist</artifactId>
  .
  .
  .
</project>
```

- Lors d'un héritage, les éléments `<groupId />` et `<version />` ne sont pas obligatoires dans le POM enfant. Les valeurs sont alors initialisées avec celles du POM parent. Néanmoins, Il est possible de les redéfinir.

Factoriser les dépendances

- La mise en place d'un Corporate POM permet de centraliser des informations communes.
- La notion d'héritage permet de déclarer les dépendances communes à tous les POM enfants directement dans le POM parent.
- Les dépendances déclarées dans le POM enfant sont disposées avant celles déclarées dans le POM parent.
- Il est possible de redéfinir la version dans les POM enfants. La version d'une dépendance déclarée dans le POM enfant est prioritaire sur la version du POM parent.

Centraliser les informations

- Il est possible de centraliser toutes les informations concernant les dépendances afin de maîtriser et d'alléger les dépendances des POM enfants.
- L'élément `<dependencyManagement />` permet de centraliser :
 - Les versions des dépendances à utiliser,
 - Les dépendances transitives à exclure,
 - Les champs d'application des dépendances.

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>log4j</groupId>
      <artifactId>log4j</artifactId>
      <version>1.2.13</version>
      <scope>compile</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

```
<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
  </dependency>
</dependencies>
```


Centraliser les informations (2)

- L'élément `<pluginManagement />` est l'équivalent pour la gestion des plugins de l'élément `<dependencyManagement />` pour les dépendances du POM.
- Il permet de centraliser les versions et les configurations des plugins pour l'ensemble des POM enfants.

```
<pluginManagement>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.0</version>
      <configuration>
        <source>1.6</source>
        <target>1.6</target>
      </configuration>
    </plugin>
  </plugins>
</pluginManagement>
```

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugin</groupId>
    <artifactId>maven-compiler-plugin</artifactId>
  </plugin>
</plugins>
```

Les projets multi-modules

- L'organisation des projets et les dépendances qu'ils possèdent les uns envers les autres peut devenir compliquer à gérer.
- De nombreuses manipulations peuvent être nécessaires dès lors qu'un projet est modifié.
- Maven propose une solution pour éviter ces manipulations en **associant des projets** ensemble à travers le **concept de modules**.

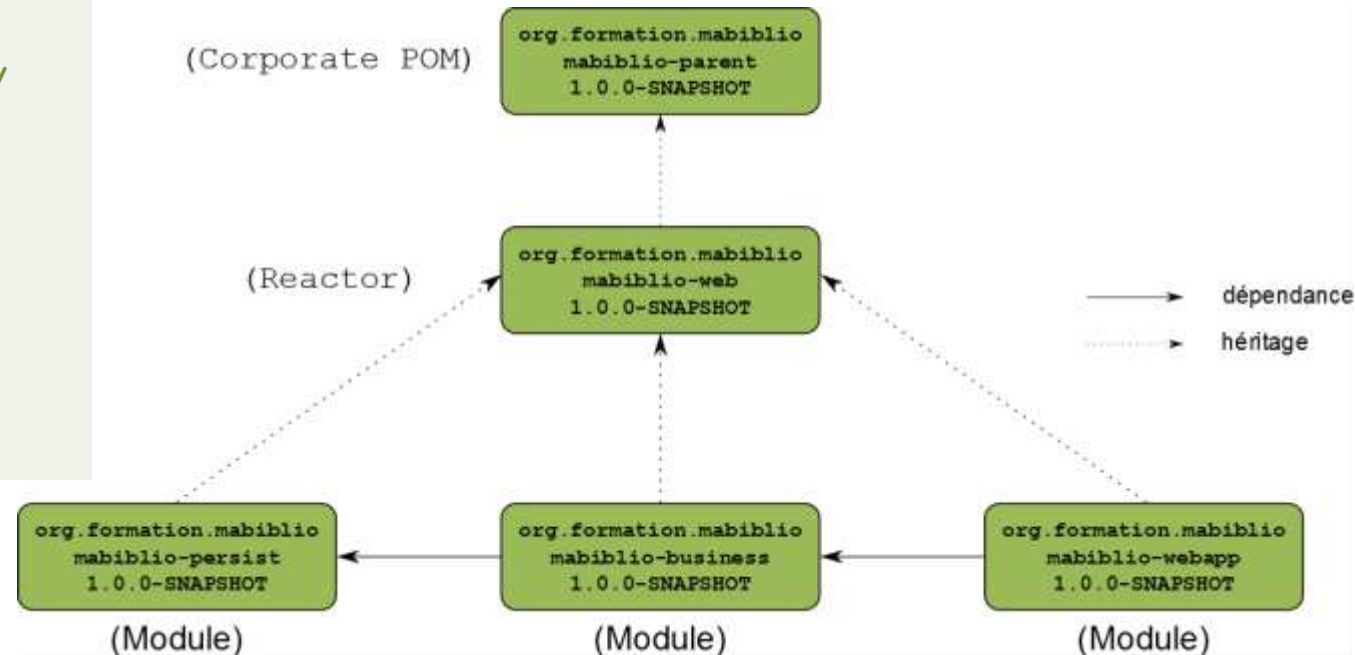
Définition du *reactor*

- Maven permet de créer un projet de **type pom**, appelé `reactor`, dont le but unique est de lister les projets qu'il identifie comme **ses modules**.
- Une seule commande appelée sur ce POM effectue les phases sur chaque projet en respectant leurs dépendances et donc l'ordre correct d'exécution.
- La mise en place d'un `reactor` s'articule autour de deux notions fondamentales :
 - La définition dans le POM du `reactor` des éléments `<module />` pour identifier les projets Maven à associer.
 - Le **respect de l'arborescence** des projets sur le système de fichier en adéquation avec le chemin d'accès défini dans les éléments `<module />` du POM du `reactor`.

Organisation d'un projet multi-modules

```
mabiblio-web
|
+- mabiblio-persist/
| +- src/
| +- pom.xml
|
+- mabiblio-business/
| +- src/
| +- pom.xml
|
+- mabiblio-webapp/
| +- src/
| +- pom.xml
|
+- pom.xml
```

```
<project>
. . .
<modules>
  <module>mabiblio-webapp</module>
  <module>mabiblio-persistence</module>
  <module>mabiblio-business</module>
</modules>
. . .
</project>
```



Traitement d'un projet multi-modules

```
$ mvn install
```

- L'exécution de cette commande sur le projet `reactor` donne le résultat suivant :

```
[INFO] Scanning for projects...
[INFO] -----
[INFO] Reactor Build Order:
[INFO]
[INFO] mabiblio-web
[INFO] mabiblio-persistence
[INFO] mabiblio-business
[INFO] mabiblio-webapp
[INFO]
[INFO] -----
[INFO] Building mabiblio-web 1.0.0-SNAPSHOT
[INFO] -----
```

Ordre d'exécution lié aux dépendances

org.formation.mabiblio
mabiblio-web
1.0.0-SNAPSHOT

org.formation.mabiblio
mabiblio-persist
1.0.0-SNAPSHOT

org.formation.mabiblio
mabiblio-business
1.0.0-SNAPSHOT

org.formation.mabiblio
mabiblio-webapp
1.0.0-SNAPSHOT

Le mode commande adapté au *reactor*

- À partir de la racine d'un projet `reactor`, il est possible de préciser la liste des projets sur lesquels exécuter une commande :

```
$ mvn -pl mabiblio-business test
```

- Cette commande ne traite pas les dépendances différemment d'un projet sans module, c'est-à-dire que les artéfacts des dépendances sont recherchés dans les référentiels.
- Pour profiter de l'organisation des projets en module, il faut utiliser l'option `-am`.

```
$ mvn -pl mabiblio-business -am test
```

- Ainsi, le `reactor` construit les projets dans le bon ordre selon les dépendances entre projets.

Utilisation avancée

Maven et les tests

- La place des tests est fondamentale dans un projet de développement logiciel.
- Il est possible de différencier deux niveaux de tests :
 - Les tests unitaires,
 - Les tests d'intégration.
- Maven ne gère nativement que les tests unitaires. En effet, son arborescence de dossier standard est telle qu'il n'y a seulement qu'un point d'entrée pour les tests :
 - `${project.basedir}/src/test/java`
 - `${project.basedir}/src/test/resources`
- Néanmoins, il est possible de configurer le POM pour séparer les tests unitaires des tests d'intégration dans le cycle de vie d'un projet Maven

Les tests unitaires

- Le plugin utilisé par défaut pour exécuter les tests est le plugin `maven-surefire-plugin` qui propose le Mojo `surefire:test`.
- Les rapports de test sont générés dans le dossier `${project.basedir}/target/surefire-reports`.
- Il est possible de choisir :
 - La (ou les) classe de tests à exécuter :

```
$ mvn -Dtest=LivreDaoImplTest <phase>
```

```
$ mvn -Dtest=*Test <phase>
```

- la (ou les) méthode à exécuter :

```
$ mvn -Dtest=LivreDaoImplTest#testInsert <phase>
```

Les tests d'intégration

- Parmi les phases définies dans le cycle de vie par défaut, il en existe 4 dédiées aux tests d'intégration :



- Le plugin associé à ces phases est le plugin `maven-failsafe-plugin`. Il est basé sur le plugin `maven-surefire-plugin`.
- Si les tests échouent pendant le phase `integration-test`, le plugin `maven-failsafe-plugin` s'assure que la phase `post-integration` soit toujours appelée.
- La phase `verify` valide les résultat des tests.

Inclusions et exclusions de tests

- Comme les classes de tests pour les tests unitaires ou les tests d'intégration se trouvent dans le même répertoire `${project.basedir}/src/test/java`, il est nécessaire de les différencier pour ne pas exécuter deux fois les mêmes tests.
- Le plugin `maven-surefire-plugin` recherche par défaut les classes dont le nom respecte les formats suivants:
 - `**/Test*.java`
 - `**/*Test.java`
 - `**/*TestCase.java`
- Le plugin `maven-failsafe-plugin` recherche par défaut les classes dont le nom respecte les formats suivants :
 - `**/IT*.java`
 - `**/*IT.java`
 - `**/*ITCase.java`
- Il est possible de modifier en configurant chacun des deux plugins à l'aide des éléments `<includes />` et `<excludes />`

Configurer l'exécution des tests

- Que ce soit pour les tests unitaires ou les tests d'intégration, il est possible d'annuler leur exécution :

- Pour tous les tests

```
$ mvn <phase> -DskipTests
```

- Uniquement les tests d'intégration

```
$ mvn <phase> -DskipITs
```

- Il est possible d'annuler la compilation des tests :

```
$ mvn <phase> -Dmaven.test.skip=true
```

- Il est aussi possible d'ignorer les tests unitaires en échec (même si cela fortement déconseillé !)

```
$ mvn <phase> -Dmaven.test.failure.ignore=true
```

Les propriétés

- Il est possible d'utiliser des propriétés existantes ou de déclarer de nouvelles propriétés dans le POM.
- Pour utiliser une propriété, la syntaxe est toujours la même :
`${nom.de.la.propriété}`
- Pour déclarer une propriété, il faut utiliser l'élément `<properties />`

```
<properties>  
  <log4j.version>1.2.13</log4j.version>  
</properties>
```

- Le nom de la balise XML est alors le nom de la propriété.

Les propriétés (2)

- Maven fournit trois variables implicites qui peuvent être utilisées pour accéder aux variables d'environnement, aux informations du POM et à votre configuration de Maven :
 - `project` : permet d'accéder aux différents éléments du POM en utilisant une notation pointée pour référencer la valeur d'un élément du POM.
Ex : `${project.build.filename}`
 - `env` : permet d'accéder aux variables d'environnement du système d'exploitation. Ex : `${env.PATH}`
 - `settings` : permet d'accéder aux informations de la configuration de Maven en utilisant une notation pointée pour référencer la valeur d'un élément du fichier `settings.xml`. Ex : `${settings.offline}`
- Toutes les propriétés accessibles via la méthode `getProperties()` de la classe `java.lang.System` sont visibles comme propriétés du POM
 - Ex : `${user.name}`, `${user.home}`, `${java.home}`,
et `${os.name}`

Les filtrage de ressources

- Les valeurs des propriétés définies dans le POM peuvent être insérées dans les fichiers de configuration du projet.
- Cette fonctionnalité est possible grâce au plugin *maven-resources-plugin*. Pour que la configuration du plugin soit active, il faut autoriser les ressources du projet à être filtrées
- Les propriétés au format Maven présentes dans les fichiers contenus dans les ressources filtrées sont remplacées par leur valeur lors de l'appel du goal `resource:resource`

Les filtrage de ressources (2)

- Exemple :

```
log4j.rootLogger=${logger.level}, A1  
...
```

src/main/resources/log4j.properties

```
<build>  
  ...  
  <resources>  
    <resource>  
      <directory>src/main/resources</directory>  
      <filtering>true</filtering>  
    </resource>  
  </resources>  
  ...  
</build>
```

- Il est possible de configurer dans le POM des fichiers définissant des propriétés utilisées lors du filtrage des ressources à l'aide de l'élément `<filters />`.

```
logger.level=DEBUG  
...
```

src/main/filters/dev.properties

```
<build>  
  ...  
  <filters>  
    <filter>src/main/filters/dev.properties</filter>  
  </filters>  
  ...  
</build>
```


Les profils

- Un profil est un **sous-modèle** du POM.
- Il peut être déclaré dans le fichier `settings.xml` ou dans le POM du projet.
- Contrairement aux autres éléments du POM, les éléments compris dans les profils ne vont pas être obligatoirement pris en compte lors de l'exécution d'une commande Maven sur un projet.

```
<profiles>
  <profile>
    <id>profil-1</id>
    .
    .
    .
  </profile>
  <profile>
    .
    .
    .
  </profile>
  .
  .
  .
</profiles>
```

Configuration des profils

- Il est important d'identifier les éléments mis à disposition selon l'emplacement de la déclaration du profil.
- Éléments du profil dans le POM :
 - Le modèle de données du profil est aussi complet que son élément parent à l'exception des coordonnées qui identifie le projet et des informations générales du projet.
- Éléments du profil dans le `settings.xml` :
 - Il y a moins d'éléments. Il est possible de définir des propriétés et des référentiels.

Activation de profils

- Activation par leurs identifiants à l'aide du paramètre

```
$ mvn -P profil-1 install
```

- Activation par rapport :
 - Au système d'exploitation
 - À la version du JDK
 - À la présence ou non d'une propriété
 - À la présence ou non d'un fichier

```
<profile>
<id>profil-1</id>
<activation>
  <os>
    <name>windows 7</name>
    <arch>x86</arch>
    <family>windows</family>
    <version>6.1</version>
  </os>
  <jdk>1.6</jdk>
  <property>
    <name>...</name>
    <value>...</value>
  </property>
  <file>
    <exists>...</exists>
    <missing>...</missing>
  </file>
</activation>
. . .
</profile>
```

Activation de profils (2)

- Il est possible d'activer un profil par défaut.

```
<profile>
  . . .
  <activation>
    <activeByDefault>true</activeByDefault>
  </activation>
  . . .
</profile>
```

- Le plugin *maven-help-plugin* propose des goals qui permettent :
 - De lister les profils définis pour le projet :
 - De lister les profils actifs

```
$ mvn help:all-profiles
```

```
$ mvn help:active-profiles
```

Les archétypes

- Un archétype est un modèle de projet Maven.
- Les archétypes peuvent également être utilisés dans une société pour encourager la standardisation d'un ensemble de projets.
- Toute personne est capable de démarrer rapidement un nouveau projet en respectant la structure préconisée.

Utilisation d'un archétype

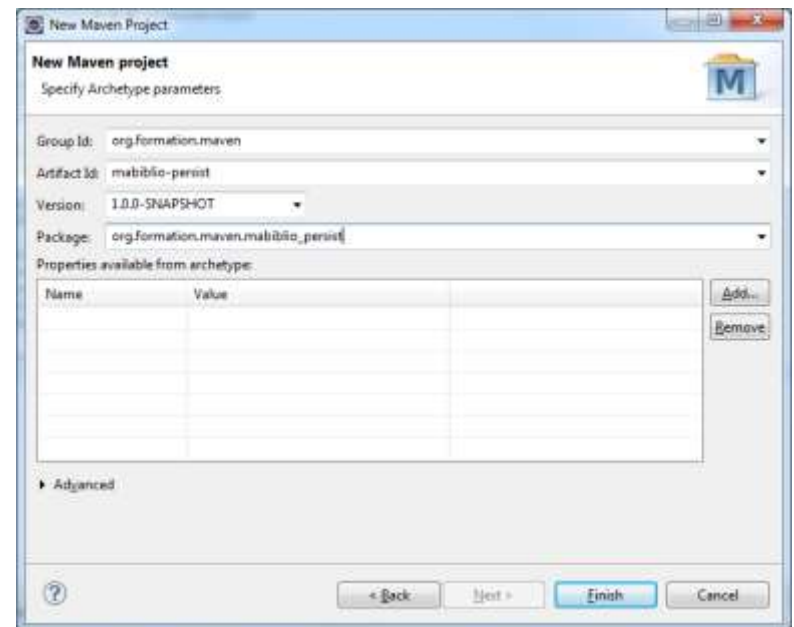
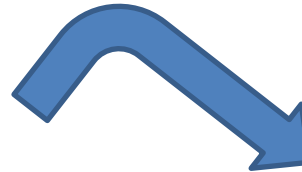
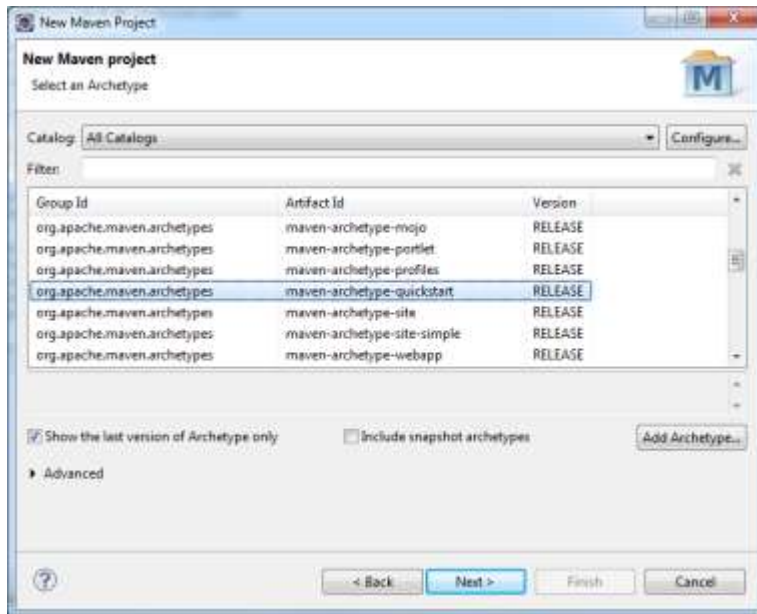
- Le plugin `maven-archetype-plugin` utilise ces archétypes pour créer de nouveaux projets.
- Utilisation de la ligne de commande :

```
$ mvn archetype:generate
```

- Utilisation du mode interactif.
 - Le moyen le plus simple d'utiliser ce plugin pour générer un nouveau projet Maven à partir d'un archétype est d'utiliser le goal `archetype:generate` en mode interactif.

Archétype et *m2eclipse*

- Utilisation du plugin *m2eclipse*



Les archétypes disponibles

- De plus en plus d'archétypes sont publiés et sont disponibles.
- Ils fournissent un moyen simple de créer des projets à partir de modèles existants.
- Les archétypes disponibles les plus simple sont regroupés dans le `groupId org.apache.maven.archetypes`.
- Quelques exemples :
 - **maven-archetype-quickstart** : pour créer un simple projet de type JAR avec une dépendance sur Junit.
 - **maven-archetype-webapp** : pour créer un simple projet de type WAR avec une dépendance sur Junit.
 - **maven-archetype-mojo** : pour créer un simple projet de type maven-plugin.

Création d'un archétype

- A partir d'un projet existant

```
$ mvn archetype:create-from-project
```

- Une fois la commande lancée, Maven créera complètement l'archétype dans le répertoire `target/generated-sources/archetype`
- Pour installer l'archétype dans le **catalogue local** et ainsi l'utiliser sur la machine locale, il faut exécuter la commande suivante :

```
$ mvn install
```

Définition d'un archétype

1. Créez un nouveau projet et un fichier `pom.xml` pour l'artefact archétype

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>my.groupId</groupId>
  <artifactId>my-archetype-id</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>jar</packaging>
</project>
```

2. Créer le descripteur de l'archétype: `archetype.xml`
3. Créez les fichiers du prototype et le `pom.xml` du prototype
4. Installez et exécutez l'archétype du plugin archetype.

Définition d'un archétype (2)

- Le descripteur archétype est un fichier appelé `archetype.xml`.
- Il doit être situé dans le répertoire `src/main/resources/META-INF/maven/`.

```
archetype
|-- pom.xml
'-- src
    '-- main
        '-- resources
            |-- META-INF
            |   '-- maven
            |       '-- archetype.xml
            '-- archetype-resources
                |-- pom.xml
                '-- src
                    |-- main
                    |   '-- java
                    |       '-- App.java
                    '-- test
                        '-- java
                            '-- AppTest.java
```

```
<archetype>
  <id>quickstart</id>
  <sources>
    <source>src/main/java/App.java</source>
  </sources>
  <testSources>
    <source>src/test/java/AppTest.java</source>
  </testSources>
</archetype>
```

Les Assemblies

- Même si Maven supporte les différents formats standards de packaging grâce à ses plugins et ses cycles de vie personnalisés, il arrive parfois le besoin de créer une archive ou un répertoire avec une structure qui lui est propre.
- Ces archives personnalisées sont appelées Assemblies Maven.
- Les Assemblies apportent aux utilisateurs de Maven la flexibilité dont ils ont besoin pour produire des archives personnalisées de tout type.

Les descripteurs d'assembly prédéfinis

- Le plugin Assembly fournit des descripteurs prêts à l'emploi pour plusieurs types d'archives communs :
 - `bin` : utilisé pour packager les fichiers LICENSE, README et NOTICE du projet avec son artefact principal, pour peu que ce dernier soit un jar.
 - `jar-with-dependencies` : construit une archive JAR avec le contenu du jar du projet principal et les contenus des dépendances d'exécution de ce projet.
 - `project` : construit une archive à partir de la structure de répertoire du projet telle qu'elle existe sur votre système de fichier.
 - `src` : produit une archive du code source de votre projet avec les fichiers pom.xml, ainsi que les éventuels fichiers LICENSE, README et NOTICE qui se trouvent dans le répertoire racine du projet.

Le descripteur d'assembly

- Quand les descripteurs d'assembly prédéfinis ne sont pas pertinents pour un projet, il est nécessaire de définir son propre descripteur.
- Un descripteur d'assembly est un document XML qui définit la structure et les contenus d'un assembly.
- Ce descripteur se compose de cinq sections principales de configuration :
 - Configuration de base,
 - Informations concernant les fichiers,
 - Informations concernant les dépendances,
 - Informations concernant les dépôts,
 - Informations concernant les modules.

Le processus de livraison

Introduction

- Le but ultime de tout développement logiciel est de **livrer le projet au client**.
- Cette étape dans la vie d'un projet est généralement conduite par des objectifs ciblés sur des **périmètres techniques et fonctionnels**.
- Le terme de **livraison** peut englober plusieurs éléments :
 - L'application stable sur un périmètre technique et fonctionnel précis,
 - La documentation technique du projet,
 - Le code source du projet,
 - Les rapports techniques sur la qualité du projet,
 - La documentation pour les utilisateurs de l'application,
 - Le guide d'installation de l'application.

Les SCM

- Le **gestionnaire de code source** (*Source Code Management*) du projet est généralement un des premier éléments mis en place lors du démarrage du projet.
- Il existe de nombreux outils capables de réaliser la gestion de code source.
- Quelques SCM reconnus en entreprise :
 - CVS (*Concurrent Versions System*),
 - SVN (*Subversion*),
 - Git

Configuration des SCM

- La syntaxe des URL de connexion à un gestionnaire de code source est la suivante :

```
scm:<identifiant_du_scm>:<url_spécifique_au_scm>
```

- Les éléments du POM qui permettent de gérer les connexions au gestionnaire de code source du projet sont les suivants :

```
<project>
  . . .
  <scm>
    <url>http://websvn.mabiblio.org</url>
    <connection>scm:svn:svn://localhost/mabiblio/</connection>
    <developerConnection>scm:svn:svn://localhost/mabiblio/trunk/</developerConnection>
  </scm>
  . . .
</project>
```

- L'élément `<connection />` est dédiée à l'accès en lecture seule.
- L'élément `<developerConnection />` est dédié à un accès en écriture.

Le plugin *maven-scm-plugin*

- Les éléments présents dans le POM donnent des informations générales de connexion au gestionnaire de code source. Pour réaliser les opérations communes à tous les SCM, Maven propose le plugin `maven-scm-plugin`.
- Ce plugin propose des goals associées aux commandes qui seront exécutées au final par le client du SCM :
 - `scm:bootstrap` : initialise le transfert d'un projet à partir du serveur SCM puis exécute un goal sur le projet en local.
 - `scm:checkin` : commit les modifications locales vers le serveur SCM.
 - `scm:checkout` : récupère les sources à partir du serveur SCM.
 - `scm:add` : ajoute un fichier dans le gestionnaire de code source.
 - `scm:update` : met à jour la version locale du projet avec la dernière version présente sur le serveur SCM.

Gestion des numéros de version

- Le numéro de version est l'**identifiant** qui permet de faire la relation entre le nom d'un livrable et son contenu technique et fonctionnel.
- Le numéro de version doit respecter **des normes précises**.
- Maven considère qu'un projet n'est pas une version de type *release* lorsque son numéro de version contient la chaîne de caractère – **SNAPSHOT**.

```
<majeure>.<mineure>.<incrémental>-<qualificatif>-<construction>
```

- Les délimiteurs sont important car le point (.) définit une version plus importante que le tiret (-). Ex : version 1.0.5 > version 1-1
- Il est fortement conseillé de le format défini par Maven.

Le plugin versions-maven-plugin

- La gestion des numéros de version peut devenir une tâche contraignante lorsque les projets disposent de nombreux modules ou si l'héritage se réalise sur plusieurs niveaux.
- Pour optimiser la gestions des numéros de version, il existe le plugin `versions-maven-plugin`.
- Il propose plusieurs goals très utiles :
 - `versions:set` : modifie la version dans tous les POM du projet.
 - `versions:update-parent` : vérifie si le POM parent du projet est disponible dans une version plus récente et met à jour le POM du projet si nécessaire.
 - `versions:display-dependency-updates` : affiche la liste des dépendances du projet qui sont disponibles dans une version plus récente.
 - `versions:display-plugin-updates` : affiche la liste des plugins utilisés dans le projet qui sont disponibles dans une version plus récente que celle utilisée.

Le processus de *Release*

- Maven gère le processus de *Release* des projets avec le plugin `maven-release-plugin`.
- Les deux goals suivants définissent le processus de *Release* :
 - `release:prepare` : préparation de la Release.
 - `release:perform` : exécution de la Release.

Préparer et effectuer une Release

- **Préparer une Release** passe par les phases suivantes :
 - vérifie que le code local ne diffère pas de ce qui se trouve sur le SCM
 - vérifie qu'il n'existe pas de dépendances en version SNAPSHOT (entraînant la non reproductivité du livrable)
 - demande à l'utilisateur d'indiquer le numéro de version du livrable à fournir, le nom du tag qui sera posé et la version du numéro de projet à utiliser à la suite de la génération du livrable
 - vérifie la conformité du code récupéré du SCM en lançant le goal maven *verify*
 - modifie les informations des pom du projet avec le numéro de version du livrable
 - committe le projet dans le SCM au tag indiqué
 - modifie les informations des pom du projet avec le numéro de version à utiliser à la suite de la génération du livrable
 - committe le projet dans le SCM
 - prépare la phase suivante en générant le fichier *release.properties*
- **Exécuter une Release** passe par les phases suivantes :
 - Checkout de la version taguée à partir du serveur SCM
 - Exécute les goals prédéfinis (par défaut *deploy site-deploy*)

Optimiser le processus de *Release*

- Vérifier les dépendances

- Il est possible d'afficher un rapport sur les dépendances du projet qui sont déclarées dans le POM alors qu'elles ne sont pas utilisées grâce au plugin `maven-dependency-plugin`

```
$ mvn dependency:analyse
```

- Valider les paramètres techniques

- Afin d'éviter des constructions non conformes à l'architecture technique du projet, il est possible de vérifier des règles grâce au plugin `maven-enforcer-plugin`.

```
$ mvn enforcer:enforce
```

- Tester le processus de *Release*

- Avant de réaliser le processus de Release, il est possible de faire une simulation de la procédure de préparation pour valider les aspects techniques. Les traitements d'écriture sur le SCM ne sont pas réalisés.

```
$ mvn -DdryRun=true release:prepare
```

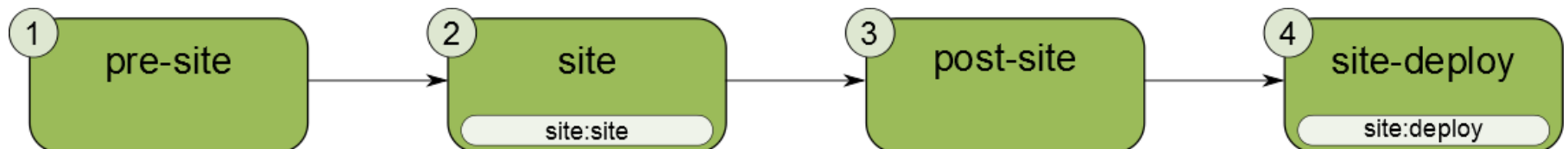

Rapports et mesure de la qualité

Générer le site Web du projet

- Dans le développement logiciel, la gestion de la documentation est primordiale à la maintenance, au suivi et à l'évolution d'un projet.
- Maven gère cette documentation sous forme de rapports au format HTML.
- Elle fait partie intégrante d'un projet Maven, notamment lors du processus de *Release* puisque le goal `release:perform` est configuré par défaut pour déployer le site Web du projet.

site life cycle

- Toute la notion de génération de documentation et de déploiement du site Web est centralisé par Maven dans le cycle de vie pour le site du projet qui est associé avec le plugin *maven-site-plugin*.
- Ce cycle de vie définit **4 phases** :
 - **pre-site** : initialisation avant la génération du site.
 - **site** : exécution des rapports du projet et génération du site en HTML.
 - **post-site** : finalisation de la génération du site.
 - **site-deploy** : déploiement du site sur un serveur Web.



Structure des répertoires d'un site

- Maven récupère les documents pour générer le site dans le répertoire `src/site`.
- Les documents avec des formats similaires sont regroupés dans des sous-répertoires de celui-ci.
- Le descripteur de site se trouve à l'emplacement `src/site/site.xml`
- A la génération du site, Maven copie tout le contenu du répertoire `src/site/resources` à la racine du site

```
+-- src/  
  +- site/  
    +- apt/  
      | +- index.apt  
      |  
    +- xdoc/  
      | +- other.xml  
      |  
    +- fml/  
      | +- general.fml  
      | +- faq.fml  
      |  
      |-- resources/  
        | +- example.html  
        |  
    +- site.xml
```

Formats des fichiers

- Maven utilise en moteur de traitement de documentation appelé **Doxia**.
- *Doxia* peut transformer différents formats de fichiers avec un même modèle de document, manipuler ces modèles et effectuer le rendu dans différents formats de sorties dont PDF ou XHTML.
- À ce jour, *Doxia* supporte 3 formats :
 - Format **Xdoc** : C'est un format XML, ce format est disponible depuis les versions 1.x de Maven.
 - Format **APT** "Almost Plain Text" : Désigné remplaçant du Xdoc, ce format de type Wiki est beaucoup moins verbeux.
 - Format **FML** : C'est le format utilisé pour créer des FAQ.
- À la génération du site, une page écrite au format APT dans le fichier `src/site/apt/xxx/xxx.appt` sera généré dans `target/site/xxx/xxx.html` et c'est ainsi pour tous les types de fichiers.

Génération de rapports

- La configuration de chaque plugin qui génère un rapport s'effectue dans l'élément `<reportPlugins />` du plugin `maven-site-plugin`.

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-site-plugin</artifactId>
  <version>3.2</version>
  <configuration>
    <reportPlugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-project-info-reports-plugin</artifactId>
        <version>2.6</version>
      </plugin>
      . . .
    </reportPlugins>
  </configuration>
</plugin>
```

Génération de rapports (2)

- Rapports liés au code source
 - Code source en HTML
 - Documentation du code Java (Javadoc)
 - Analyse avec PMD
 - Analyse avec Checkstyle
- Rapports sur les tests du projet
 - Rapport d'exécution de tests
 - Rapport de couverture de tests

La mesure de qualité

- La mesure de qualité d'un projet repose sur un **ensemble de métriques**.
- Il est possible de classer les métriques et mesures en **4 catégories** :
 - Le volume et la longueur du code source
 - tailles des classes, le nombre de lignes d'une méthode ...
 - La conception et la structure du code
 - Les normes de codage et le respect de l'architecture,
 - conventions de nommage, noms des classes ...
 - L'exécution du code et des tests unitaires.
 - nombre de tests en erreur, taux de couverture lors des test ...

Présentation de Sonar

- Sonar est une plate-forme open source de gestion de la qualité du code source pour les projets. Il se présente sous la forme d'une application Web.
- Sonar se base sur 7 axes de qualité pour analyser le code source des projets :
 - Les commentaires,
 - Les règles de codage,
 - Les bugs potentiels,
 - La complexité du code,
 - Les tests unitaires,
 - La duplication du code,
 - Les relations entre les composants.

Présentation de Sonar (2)

- Analyse du code de chaque projet à l'aide de différentes métriques.
- Possibilité d'observer l'évolution des valeurs au cours du temps.



Le plugin *maven-sonar-plugin*

- L'un des principes de fonctionnement de Sonar est de réaliser l'analyse complète d'un projet avec le plugin `maven-sonar-plugin`. Ce plugin dispose d'un goal `sonar:sonar` qui analyse le projet et sauvegarde l'analyse dans la base de données utilisée par Sonar.
- Il faut configurer le fichier `settings.xml` du serveur dédiée à l'analyse pour qu'il accède à la base de données.
- Il faut ensuite exécuter les goals suivants sur le projet à analyser :

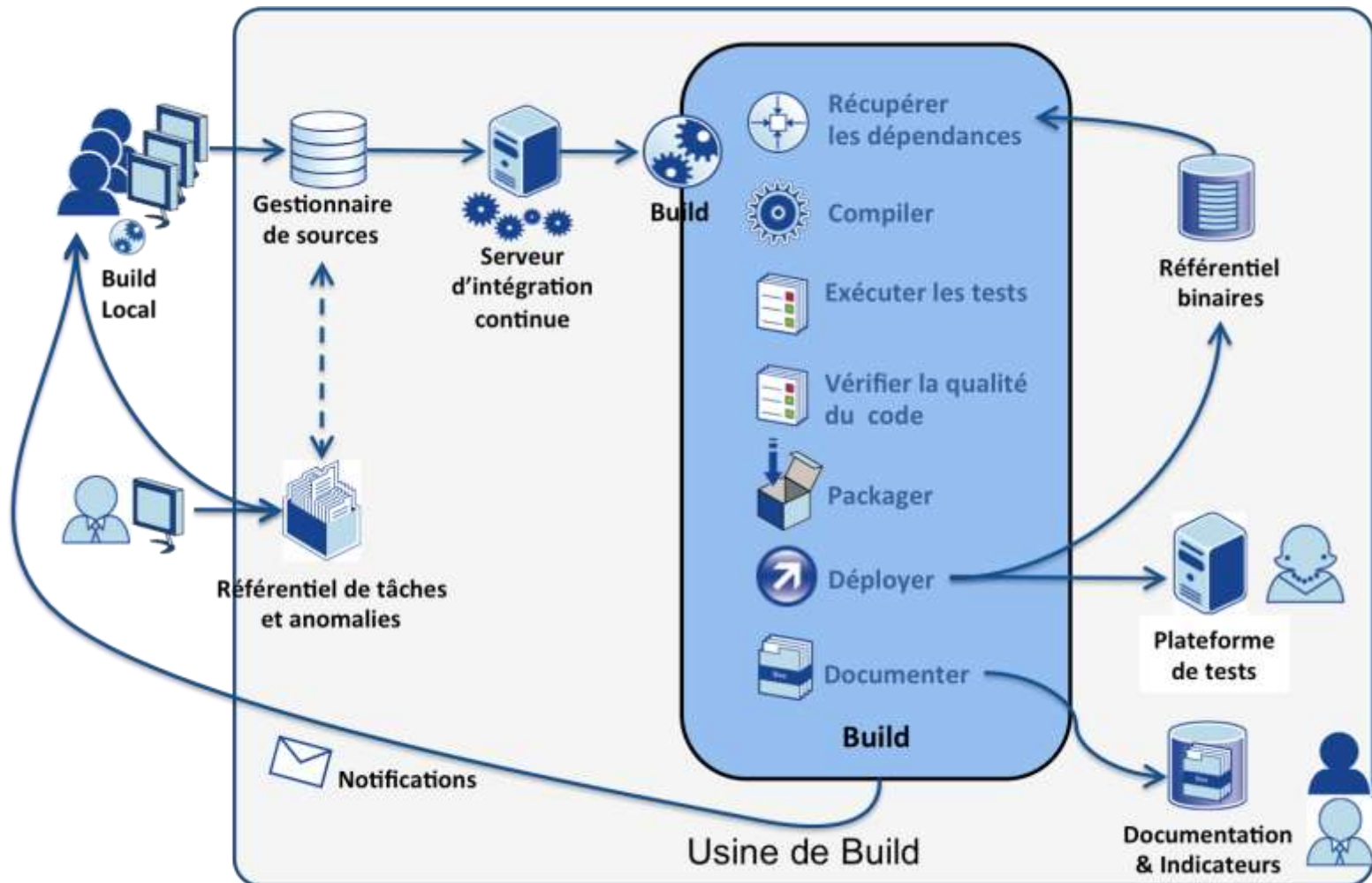
```
$ mvn install  
$ mvn sonar:sonar
```

Intégration continue

Intégration Continue : pourquoi ?

- L'intégration d'un projet est constitué de différentes étapes souvent laborieuses et consommatrices de temps.
- Cette phase d'intégration est d'autant plus critique si celle-ci est faite en fin de projet avant la livraison révélant de nombreux problèmes et mettant alors malheureusement la réussite du projet en péril
- Une anomalie générée par la modification du code source par un développeur peut être seulement découverte lors de la phase d'intégration d'une application.
- **L'intégration continue a un principe simple :**
 - **Surveiller** les modifications de code source
 - Dès qu'un changement est **déecté**, il faut **compiler** et **tester** l'application

Intégration Continue : Principes



Source : <http://blog.octo.com/>

Intégration Continue

Bonnes pratiques

- Les bonnes pratiques sont les suivantes :
 - maintenir un dépôt unique de code source versionné
 - automatiser les compilations
 - rendre les compilations « auto-testantes »
 - tout le monde commit tous les jours
 - tout commit doit compiler le tronc (*trunk*) sur une **machine d'intégration**
 - maintenir une compilation courte
 - tester dans un environnement de production cloné
 - rendre disponible facilement le dernier exécutable
 - toutes les personnes participant au projet doivent pouvoir observer ce qui se passe
 - automatiser le déploiement

Intégration Continue

- L'intégration continue a pour objectif de **vérifier** que chaque **mise à jour du code source** d'un projet en cours de développement **ne génère pas d'anomalies** ou de **régressions** sur le projet.
- L'intégration continue consiste en un **ensemble de pratiques** visant à intégrer, par un processus automatisé, le travail de chaque développeur du projet afin de **détecter les éventuelles erreurs** d'intégration le plus tôt possible.

Jenkins : Présentation

- Jenkins est un outil open source d'intégration continue.

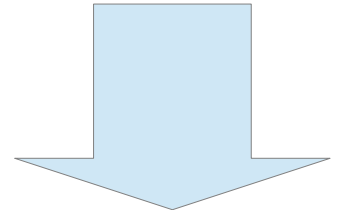


Jenkins

- Écrit en Java !
- Jenkins fonctionne dans un conteneur de servlets tel qu'*Apache Tomcat*, ou en mode autonome avec son propre serveur Web embarqué.
- Jenkins est facile à utiliser. L'interface utilisateur est simple, intuitive et visuellement agréable.

Jenkins : Historique

- Kohsuke Kawaguchi crée **Hudson** en 2004 lorsqu'il travaillait chez **Sun**.
- En 2010, Hudson était devenu la principale solution d'Intégration Continue avec une part de marché d'environ 70%.
- En 2009, **Oracle** racheta **Sun**.
- Fin 2010, de tensions entre la communauté de développeurs d'Hudson et d'Oracle apparaissent → Migration du code originel d'Hudson vers un nouveau projet **Github**.
- En Janvier 2011, la communauté des développeurs Hudson vota pour renommer le projet en **Jenkins**.



Jenkins : Concurrency

- CruiseControl (licence BSD)



- Bamboo – Altassian



- Hudson – Oracle



- Continuum - Apache





Jenkins : Plugins

- Jenkins permet une mise en œuvre de l'intégration continue adaptée à plusieurs environnements de développement grâce aux plugins proposant l'intégration d'une large palette d'outils de développement.
- Jenkins est particulièrement adapté à l'intégration continue de projets Java mais supporte aussi d'autres langages (**C/C++**, **PHP**, **Python**, **ADA**, **Groovy**)
- Jenkins dispose de plugins d'intégration vers les gestionnaires de versions dont **Subversion**, **CVS**, **Git**...
- Parmi les outils de builds, Jenkins propose en particulier des plugins d'intégration avec **Maven**, **Ant**, **Ivy**, **Gradle**, **Make**, **CMake**, ...

Jenkins : Interface Utilisateur





- Interface simple et claire.
- Visualisation de l'ensemble des projets ou d'un sous-ensemble grâce aux onglets.



S	W	Name	Dernier succès	Dernier échec	Dernière durée
		LN2R	2 j 9 h - #12	2 h 12 mn - #14	59 s
		LRI Framework	6 h 20 mn - #14	6 h 20 mn - #13	16 s
		LRI POM Parent	2 j 13 h - #6	6 j 8 h - #1	6,8 s
		Taxomap Alignment	6 h 40 mn - #2	6 h 19 mn - #3	1 mn 19 s
		Taxomap Refinement	6 h 40 mn - #3	s. o.	50 s

Jenkins : Indicateurs

- Il existe plusieurs indicateurs :
 - Statut d'un projet :

-  Aucun build n'a jamais été créé pour ce projet, ou alors ce dernier est désactivé.
-  Le précédent build s'est terminé correctement.
-  Le précédent build était instable (des tests ont échoué).
-  Le précédent build a échoué (généralement due à une erreur de compilation).

- Santé d'un projet : correspond au pourcentage de « builds » récents qui ont réussi.



0% - 20%



20% - 40%



40% - 60%



60% - 80%



80% - 100%

Jenkins : Indicateurs (2)

- Tendence d'un projet : montre l'historique de la durée des builds

