Les Principes de Maven

Présentation

- MAVEN est une « framework » de gestion de projets
- Plus précisément, MAVEN est :
  - Un ensemble de standards
  - Une « repository » d'un format particulier
  - Du logiciel pour gérer et décrire un projet
- Il fournit un cycle de vie standard pour
  - Construire, Tester, Déployer des projets
    - Selon une logique commune
- Il s'articule autour d'une déclaration commune de projet que l'on appelle le POM
  - Project Object Model
- MAVEN, c'est une façon de voir un produit comme une collection de composants interdépendants qui peuvent être décrits sous un format standard

Origines de MAVEN

- Jusque très récemment, les processus de construction de chaque produit pouvaient être différents
  - Ex : le processus de construction pour le projet Tomcat est différent de celui de STRUTS
    - Les développeurs au sein d'un même projet à « l'Apache Software Foundation » utilisaient chacun une approche différente.
  - Le manque de vue globale entraîne l'approche
    - « copy » and « paste » pour construire un projet à partir d'un autre.
      - Génère rapidement des incohérences
  - Finalement, les développeurs passent beaucoup de temps à la construction de leur application plutôt qu'à leur contenu.
- MAVEN a permis aussi de faciliter les tâches de documentation, tests, rapports d'exécution (metrics) et de déploiement
- MAVEN est très différent de ANT
  - ANT s'attarde à la notion de tâche
  - MAVEN n'est pas qu'un outil de construction (build)
    - Il voit le projet dans son ensemble
  - Ceci dit, des plugins MAVEN permettent de maintenir les scripts ANT depuis MAVEN

Principes

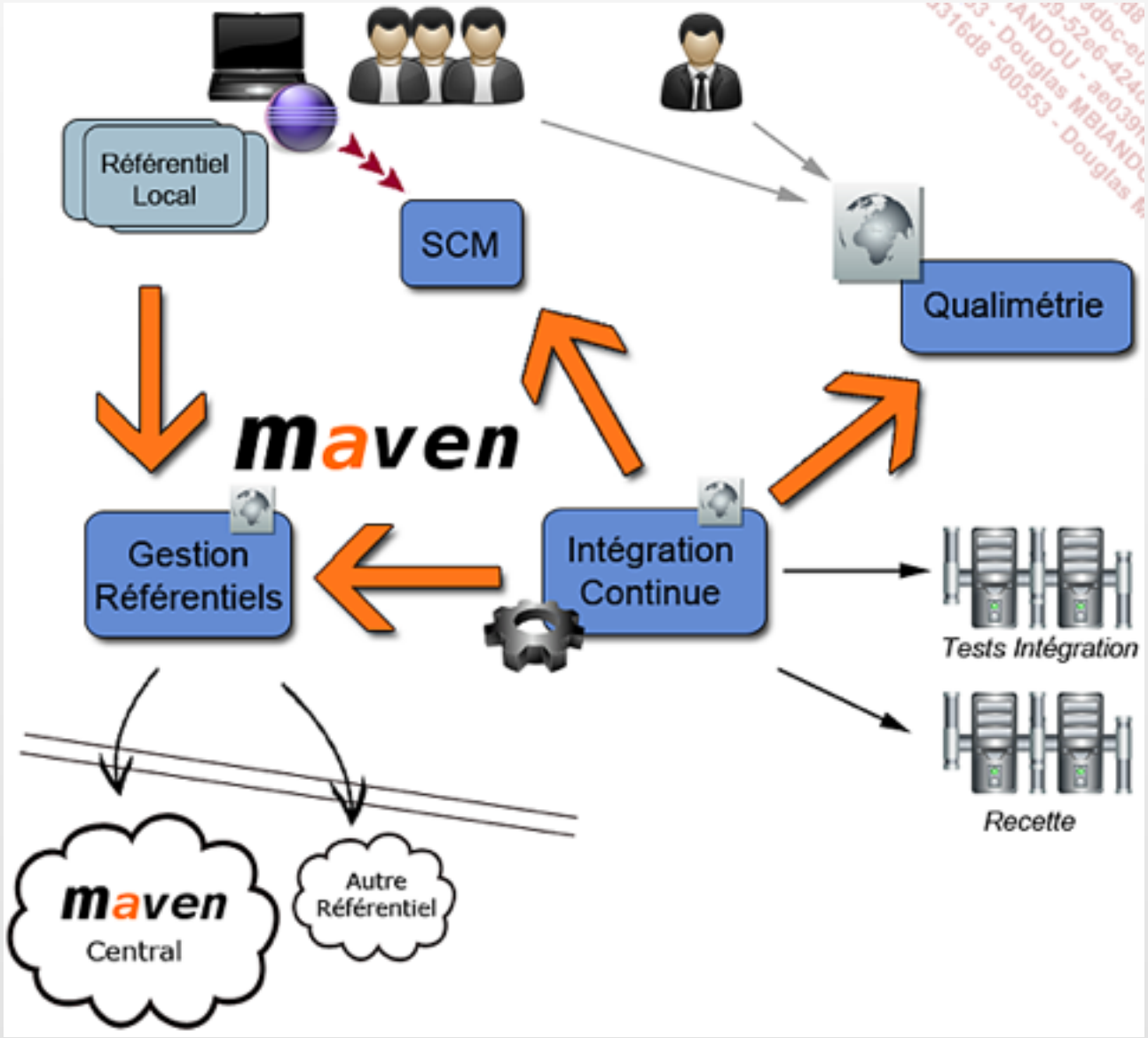
- MAVEN fournit un modèle détaillé applicable à n'importe quel projet
  - Il fournit un langage commun
  - Il fournit une abstraction comme celle de l'interface en JAVA ou autre :
    - Un camion et une voiture de tourisme ont des « implémentations » différentes ; ils se conduisent pourtant de la même façon…
- La structure d'un projet MAVEN et son contenu sont déclarés dans un POM
  - Ce POM est purement déclaratif
  - Le développeur va déclarer des objectifs dans ce POM et des dépendances
    - L'orchestration des tâches qui suivront (compile, test, assemblage, installation) sera gérée par des plugins appropriés et le POM
- MAVEN utilise beaucoup de valeurs par défaut, rendant la génération plus facile

Principes

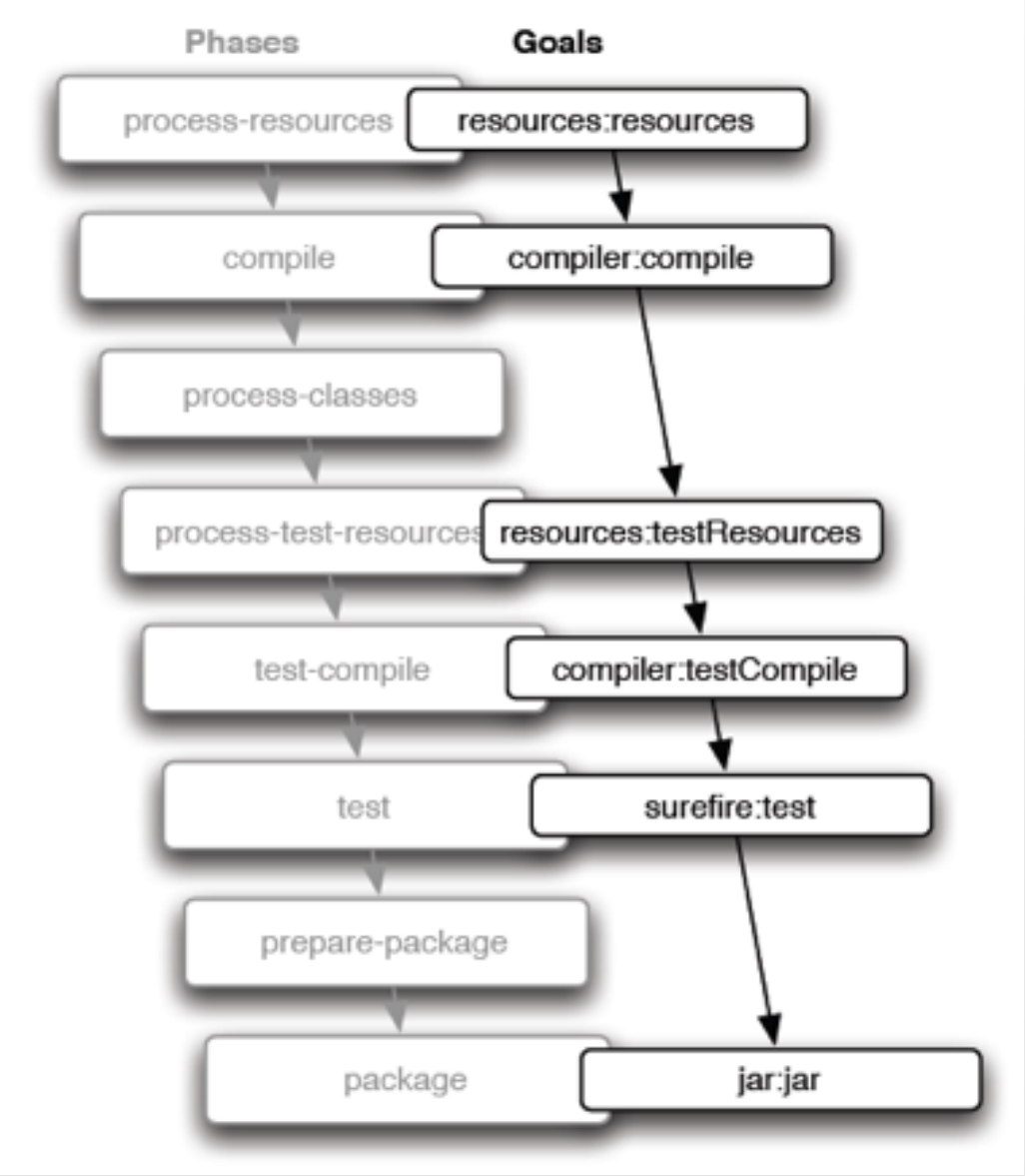
- **En résumé, MAVEN apporte :**
  - **Cohérence, réutilisabilité, agilité, maintenabilité**

- **3 éléments importants dans MAVEN sont :**
  - **Directory standard pour les projets**
    - **Facilite la lecture de tout nouveau projet, structure connue**
      - **Peut être modifiée avec des nouveaux répertoires mais impacte la complexité du POM**
  - **Un projet MAVEN fournit une seule sortie (output)**
    - **MAVEN par contre incite à découper un super projet (ex : client/serveur) en autant de projets individuels facilement réutilisables, donc une sortie par projet réutilisable, c'est le SoC (Separation of Concern)**
  - **Une convention de nommage standard**
    - **Directories, fichiers**

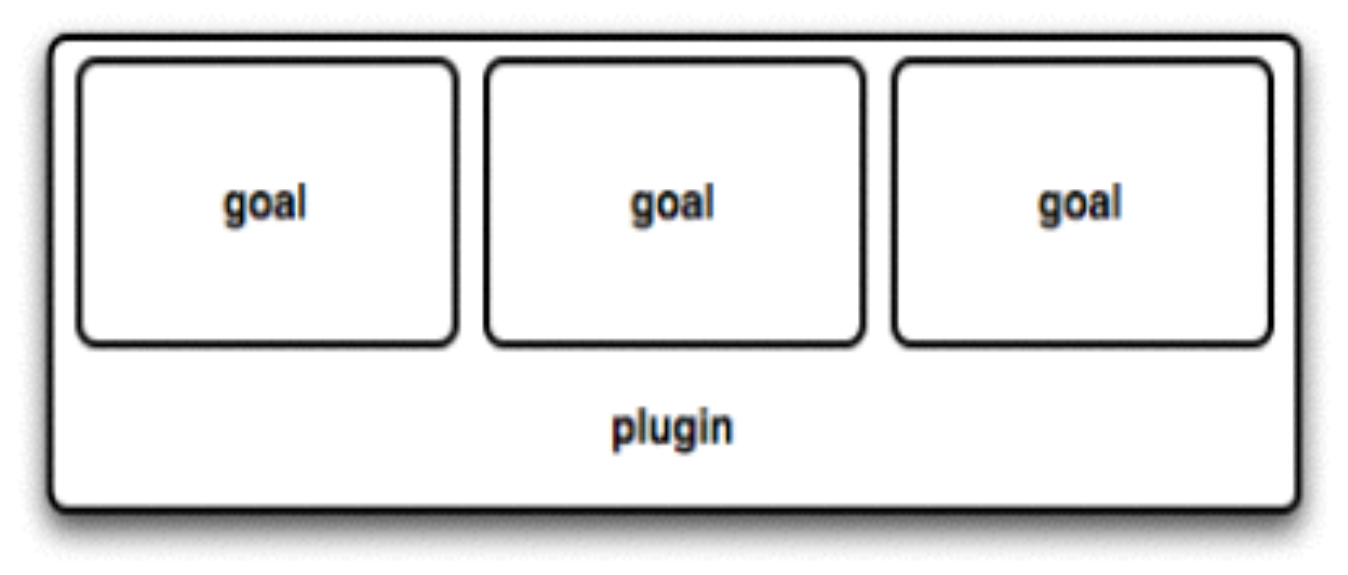
Architecture Maven



Phases , plugins et goals



Plugins et goals



Le POM

- **Le POM**
  - Maven introduit le concept d'un descripteur de projet, ou bien POM.
  - Le POM (Project Object Model) décrit le projet, fournissant toutes les informations nécessaires pour accomplir une série de tâches préconstruites.
  - Ces tâches ou *goals,* utilisent le POM pour s'exécuter correctement.
  - Des plugins peuvent être développés et utilisés dans de multiples projets de la même manière que les tâches préconstruites.
- **Les Goal**
  - En introduisant un descripteur de projet, Maven nécessite qu'un développeur fournisse des informations *sur ce qui est construit*.
  - Cela diffère des systèmes de construction traditionnels où les développeurs informent le système sur la *manière de construire*.
  - Les développeurs peuvent se concentrer sur la logique de développement.

Les Goals

- ***Goals Communs :***
  L'introduction des cycles de vies dans Maven **a considérablement réduit le nombre de goals qu'un développeur doit absolument savoir maîtriser.**
- **Les goals suivants seront toujours considérés comme utiles (et peuvent être utilisés dès que le descripteur de projet a été généré):**
- **Il sont à fournir au lancement de la commande Maven mvn**
  - clean:clean **Supprime tous les artéfacts et les fichiers intermédiaires crées par** Maven
  - eclipse:eclipse **Génère des fichiers projets pour Eclipse**
  - javadoc:javadoc **Génère la documentation Java pour le projet**
  - antrun:run **Exécute une cible ANT**
  - clover:check **Génère un rapport d'analyse du code**
  - checkstyle:checkstyle **Génère un rapport sur le style de codage du projet**
  - site:site **Crée un site web de documentation pour le projet. Ce site inclura beaucoup de rapports d'informations concernant le projet.**

Principes

- **MAVEN s'est appliqué Le SoC (Separation of Concern) aux phases mêmes de construction.**
  - **Les phases de construction par MAVEN sont fournies dans des Plugins, c'est de la responsabilité de MAVEN d'exécuter ces plugins dans l'ordre adéquat.**
  - **Il y a des plugins pour :**
    - **Compiler, lancer les test, créer les JARs, créer les JAVADOCS, etc…**
    - **L'exécution de ces plugins est coordonnée par le POM**
  - **Exemple de POM qui permettra de compiler, tester, documenter l'appli.**
    - **Comment est-ce possible ?**
      - **Grâce au « SUPER POM »**

```
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.mycompany.app</groupId>
  <artifactId>my-app</artifactId>
  <packaging>jar</packaging>
  <version>1.0-SNAPSHOT</version>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

- **Le super POM est à MAVEN ce que java.lang.Object est à JAVA**
  - **Vous vous évitez ainsi à réécrire tout ce qui est décrit dans le SUPER POM, et qui se répétra pour chaque nouveau POM**

Principes

- L ’exemple précédent est simple mais possède les essentiels :
  - modelVersion : version du modèle objet que ce POM utilise .
  - groupId : nom qualifié de l’organisation qui a créé ce projet (un peu le namespace de xml)
  - artifactId : nom de l’artefact (sortie) excepté les versions.
    - MAVEN produira artifactid-version.extension
      - Ex : myapp-1.0.jar
  - packaging : ce qui est indiqué ici générera des cycles de fabrication par MAVEN très différents selon ce type
  - version : en liaison avec l’artifactId
  - name : pour la documentation générée par MAVEN
  - url : où le site du projet peut être trouvé
  - description : simple description du projet
- Pour une complète description des éléments, voir
  - http://maven.apache.org/maven-model/maven.html

Principes

- **Les snapshots**
  - Par convention, une version en cours de développement d'un projet voit son numéro de version suivi d'un -SNAPSHOT.
  - Ainsi un projet en version 2.0-SNAPSHOT signifie que cette version est une pré-version de la version 2.0, en cours de développement.
  - Ce concept de SNAPSHOT est particulièrement important pour Maven.
    - En effet, dans la gestion des dépendances, Maven va chercher à mettre à jour les versions SNAPSHOT régulièrement pour prendre en compte les derniers développements.
    - Utiliser une version SNAPSHOT permet de bénéficier des dernières fonctionnalités d'un projet, mais en contre-partie, cette version peut être (et est) appelée à être modifiée de façon importante, sans aucun préavis.

Principes

- **Pour construire une application, MAVEN s'appuie sur un cycle de construction, constitué de plusieurs phases.**
  - **Si vous demandez d'exécuter une phase, alors toutes les phases précédentes seront forcément d'abord appelées, mais on peut changer ce mode grâce aux conventions**
    - **Ex : demander à MAVEN de compiler entraînera forcément les phases précédentes de :**
      - **Validate, initialize, generate-sources, process-sources, generate-resources**
- **Si vous devez ajouter des phases, vous passez par un plugin, soit livré, soit que vous développez**
- **Le plugin archetype va permettre de proposer des « modèles » tout faits dans des domaines connus (web, jar, etc), mais vous pourrez aussi créer vos archetypes à vous.**

Principes

- **Voilà quelques-unes unes des phases les plus utiles du cycle de vie :**
  - ***generate-sources*: Génère le code source supplémentaire nécessité par l'application, ce qui est généralement accompli par les plug-ins appropriés.**
  - ***compile*: Compile le code source du projet**
  - ***test-compile*: Compile les tests unitaires du projet**
  - ***test*: Exécute les tests unitaires (typiquement avec Junit) dans le répertoire src/test**
  - ***package*: Mets en forme le code compilé dans son format de diffusion (JAR, WAR, etc.)**
  - ***integration-test*: Réalise et déploie le package si nécessaire dans un environnement dans lequel les tests d'intégration peuvent être effectués.**
  - ***install*: Installe les produits dans l'entrepôt local, pour être utilisé comme dépendance des autres projets sur votre machine locale.**
  - ***deploy*: Réalisé dans un environnement d'intégration ou de production, copie le produit final dans un entrepôt distant pour être partagé avec d'autres développeurs ou projets.**

Les plugins

- Maven est construit sur un noyau fournissant les fonctionnalités de base pour gérer un projet, et un ensemble de plug-in qui implémentent les taches de construction d'un projet.
  On peut considérer Maven comme un socle gérant une collection de plug-ins.
- En d'autres termes, les plug-ins sont là pour implémenter les actions à exécuter.
- Ils sont utilisés pour : créer des fichiers JAR, des fichiers WAR, compiler le code, créer les tests, créer la documentation du projet, et bien plus encore.
- Toute nouvelle tache non encore présente dans la distribution des plug-ins de Maven, que l'ont aimerait utiliser, peut être développée comme un plug-in.
- Plug-in est une fonctionnalité importante de Maven puisqu'elle permet la réutilisation de code dans un multiple projet.
- Un plug-in peut être considéré comme une action telle la création d'un WAR que l'on déclare dans un fichier de description de projet POM.

Les plugins

- **Le comportement d'un plug-in est paramétrable à l'aide des paramètres ajoutés au fichier POM.**
- **Un des plug-ins les plus simples de Maven est le plug-in *Clean.***
- **Le plug-in Clean a pour fonction de supprimer les objets qui ont été crées lors du lancement du fichier POM.**
- **Suivant le standard Maven, ces objets sont placés sous le répertoire target.**
- **En lançant mvn clean:clean les goals du plug-in correspondant sont exécutés et les répertoires target sont supprimés.**
- **Un des avantages des plug-in de Maven est le fait qu'ils peuvent être entièrement développés en Java, laissant ainsi l'accès à un large ensemble d'outils et composants réutilisables pour en faciliter le développement.**

Les plugins

- **Les principaux plugins de Maven :**
  - Ant **: Produit un fichier** Ant **pour le projet.**
  - Antrun **: Lance un ensemble de tâches** Ant **à partir d'une phase de la construction.**
  - Clean **: Nettoie après la construction.**
  - Compiler **: compile des sources Java.**
  - Deploy **: Construit et Déploie les artéfacts dans un** repository**.**
  - Ear **: Génère un fichier EAR à partir du projet.**
  - Eclipse **: Génère un fichier projet Eclipse du projet courant.**
  - Install **: Installe les artefacts construits dans un** repository**.**
  - Jar **: Génère un fichier Jar à partir du projet.**

Les plugins

- Javadoc **: Crée la documentation Java du projet.**
- Projecthelp **: fournit des informations sur l'environnement de travail du projet.**
- Project-infos-reports **: Génère un rapport standard de projet.**
- Rar **: construit un RAR à partir du projet.**
- Release **: Libère le projet en cours – mise à jour du POM et étiquetage dans le SCM (Source Control Management).**
- Resources **: Copie les fichiers sous le répertoire** "resources" **dans un fichier JAR**
- Site **: Génère un site pour le projet courant.**
- Source **: Génère un JAR contenant les sources du projet.**
- Surefire **: Exécute les jeux de test.**
- War **: construit un fichier WAR du projet courant.**

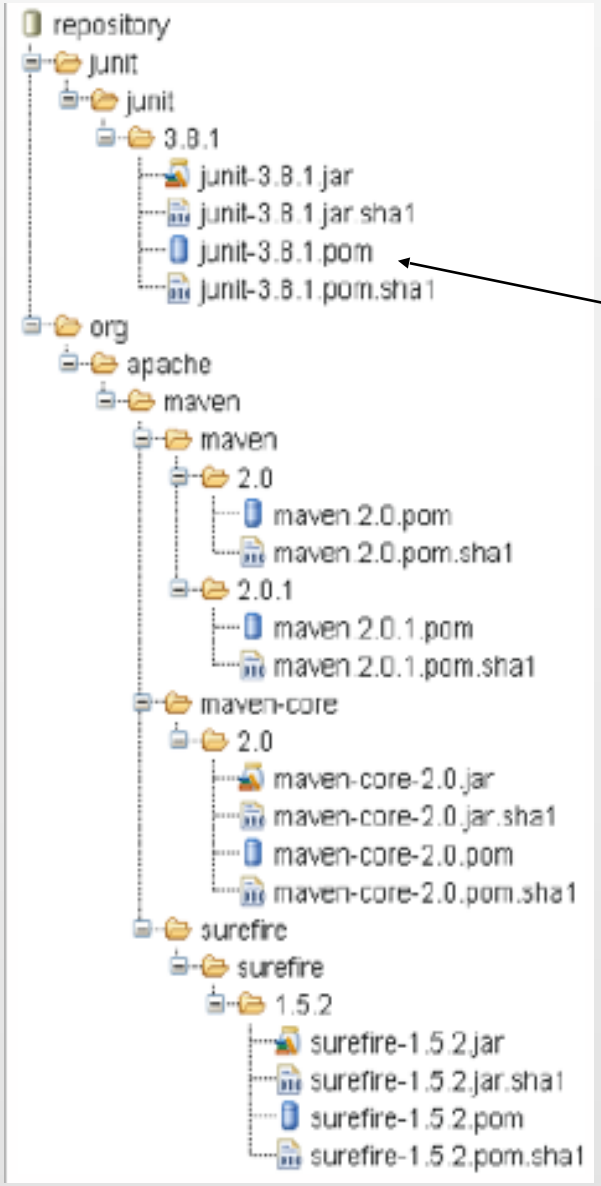
Exemple de lancement de tâche ANT

- **Exécuter un script Ant sur une phase**
- **Ajouter dans le fichier pom.xml les balises suivantes :**
- **Exemple : associer un script Ant à la phase validate**

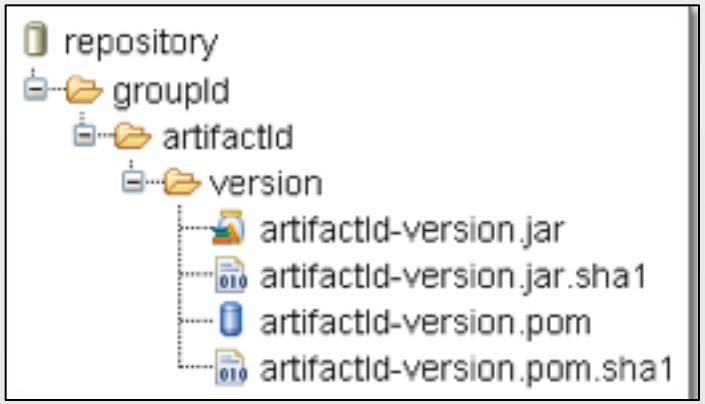
```
<project>
   ...
   <build>
      <plugins>
         <plugin>
            <artifactId>maven-antrun-plugin</artifactId>
            <executions>
               <execution>
                  <phase>validate</phase>
                  <configuration>
                     <tasks>
                        <echo file="${basedir}/hello.txt">hello world</echo>
                     </tasks>
                  </configuration>
                  <goals>
                     <goal>run</goal>
                  </goals>
               </execution>
            </executions>
         </plugin>
      </plugins>
   </build>
   ...
</project>
```

- **Dans une console de commandes, accéder au répertoire du projet, et exécuter la commande suivante :**
- **mvn validate**
- **Le script Ant redirigeant "hello world" dans un fichier hello.txt à la racine du projet, a été traité en même temps que la phase validate**

principes



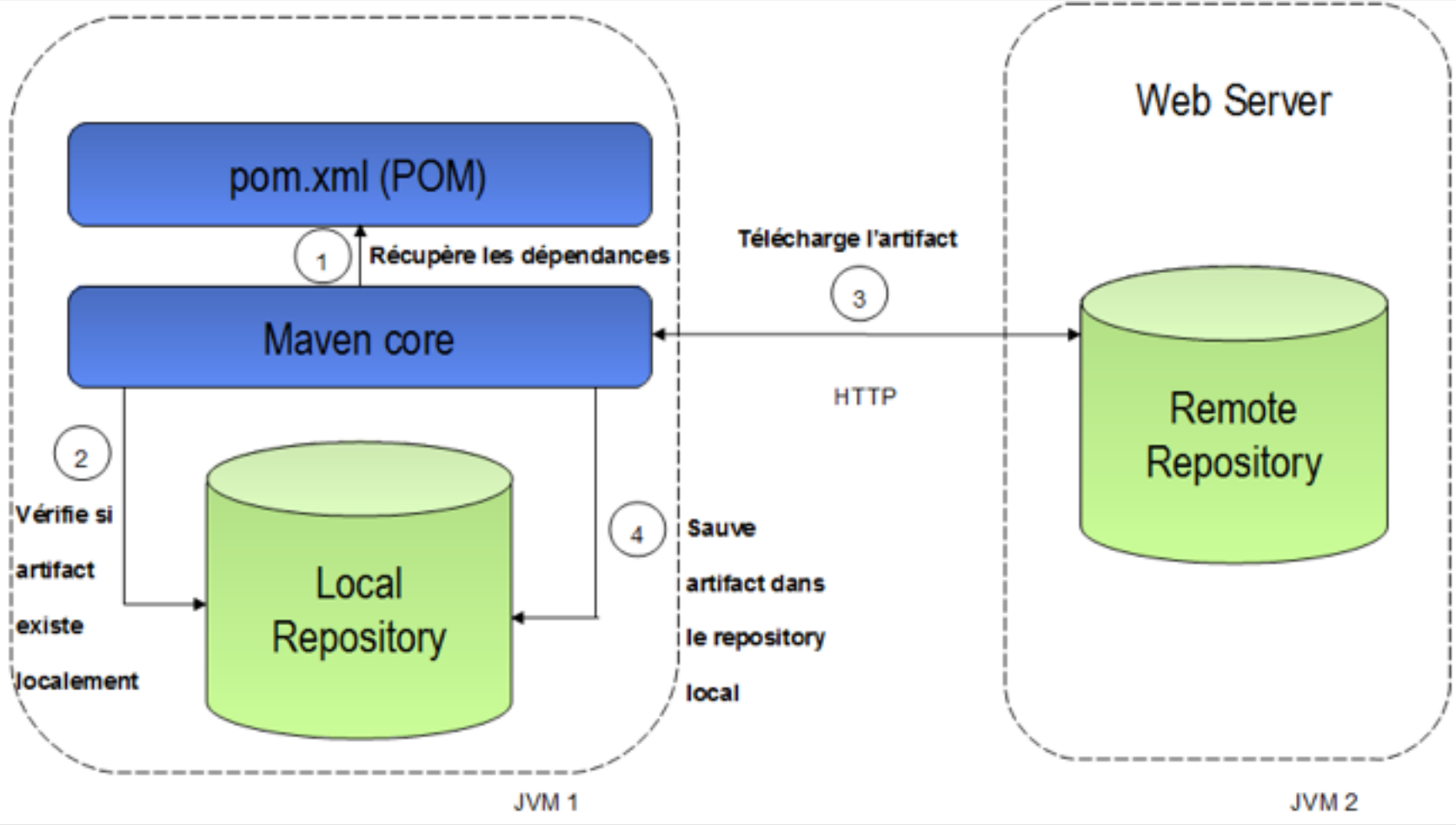
- **Les repositories MAVEN**
  - **A l'installation du produit, une repository locale est créée et remplie de dépendances par défaut.**
  - **La repository est créée par défaut sur :**
    - **/.m2/repository**
      - **Elle doit exister pour que MAVEN fonctionne**
    - **Ci-contre, exemple de repository avec junit-3.8.1.jar comme dépendance**
    - **Ci-dessous, le pattern général dans le repository :**



- **Ci-dessus :**
  - **groupid est le nom qualifié du domaine (a.b.c)**
  - **Artfifactid**
- **Ci-contre, le groupid de maven est org.apache.maven**

- **Pour satisfaire aux dépendances, MAVEN essaie d'abord de trouver l'artefact en faisant de la sorte :**
  - **Génération d'un path de recherche sur la repository locale :**
    - **Ex : pour le groupId junit artifactId junit version 3.8.1, il cherchera :**
      - **/.m2/repository/junit/junit/3.8.1/junit-3.8.1.jar**
        - **Trouvé dans X:\documents and settings\user**
    - **Si il n'est pas trouvé, il sera recherché dans une repository remote**
      - **D'abord http://repo1.maven.org/maven2**
      - **puis les autres remote définies et dans l'ordre dans le POM ou le fichier settings.xml de la directory conf de l'installation maven**

Repository Remote et Locale



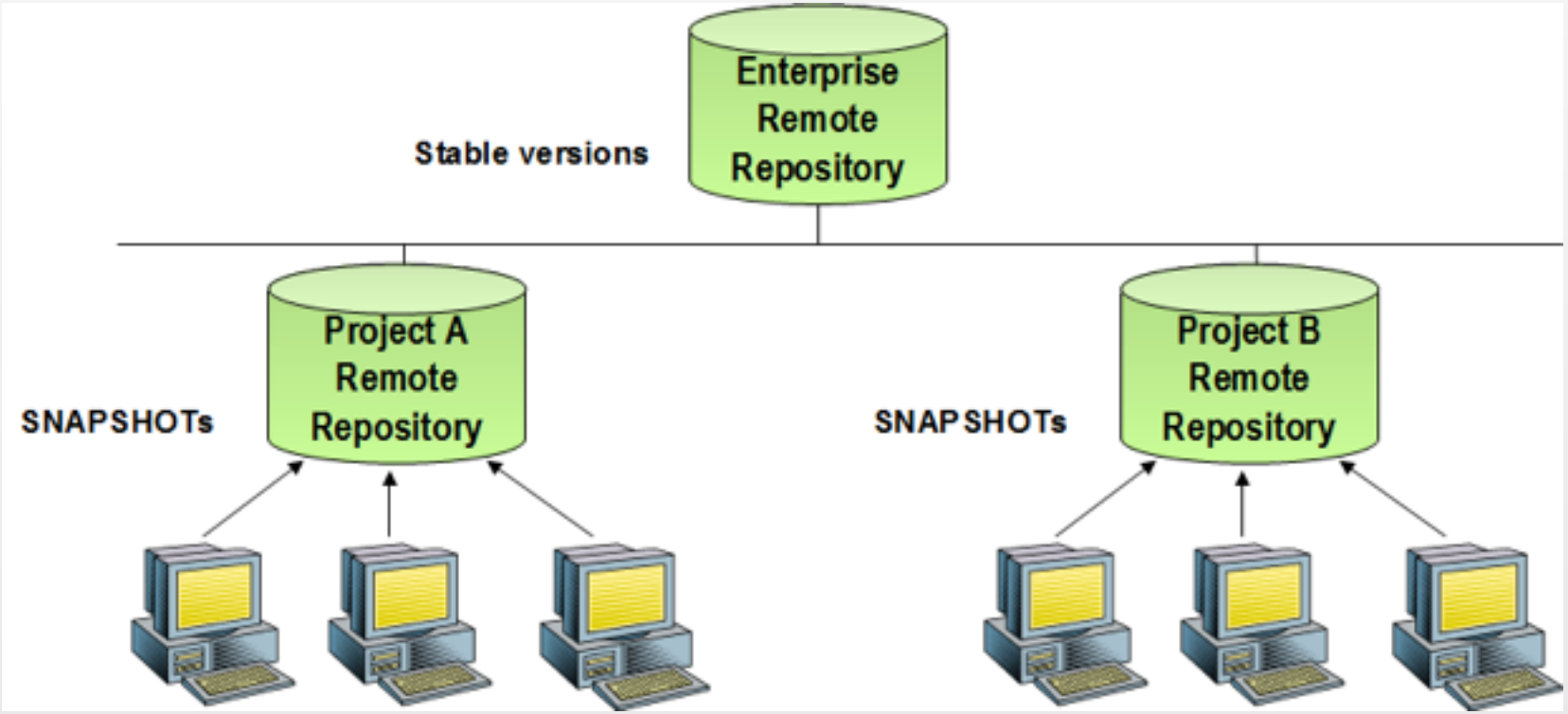
Emplacement du repository local

- **Il est possible de modifier l'emplacement du repository local**
  - **La valeur par défaut indiquant le repository local est %USER_HOME%/.m2/repository**
    - **mais elle peut être modifiée.**
    - **Dans le fichier settings.xml, situé dans %MVN_HOME%/conf ou %USER_HOME%/.m2/settings.xml :**
      - **<settings>**
      - **...**
      - **<localRepository>chemin du répertoire</localRepository>**
        - **...**
      - **</settings>**

Les dépendances

- *La Portée des dépendances*
  - Les descripteurs de projets de Maven tiennent compte de la portée des dépendances des projets.
  - Les portées tiennent compte des dépendances qui ne sont pas nécessaires dans chaque environnement.
- Les quatre portées disponibles sont:
  - Compile
    - indique que la dépendance est nécessaire pour la compilation et l'exécution. Ces dépendances seront souvent fournies avec les distributions et empaquetées dans les déploiements.
  - Provided
    - Indique que la dépendance est nécessaire pour la compilation mais ne devrait pas être empaquetée dans les déploiements et distributions.
    - Ces dépendances devraient être fournies par une source externe (typiquement un container) durant le runtime.
  - Runtime
    - Indique qu'une dépendance n'est pas nécessaire pour la compilation, mais l'est pour le runtime.
    - Souvent les dépendances de runtime sont des implémentations d'API externes et sont injectées au moment de l'exécution.
    - Un exemple de dépendance d'exécution est le pilote JDBC.
  - Test
    - Ces dépendances sont requises pour l'exécution des tests unitaires et ne sont pas rendues disponibles dans les distributions et les déploiements.

Gestion Multi-Projets



- **Parmi les bonnes pratiques de l'utilisation de Maven:**
  - ***à un projet ne correspond qu'un seul artefact comme JAR, WAR, EAR.***
  - **Ceci conduit à une simplification de la structure de projet et en facilite la construction.**
  - **En fournissant une structure de projet fortement hiérarchisée, Maven permet donc de simplifier le développement des applications de l'entreprise.**
  - **Grâce à la gestion multi-projets que procure Maven, il est possible de gérer, à l'aide d'un *descripteur Parent* de type POM toutes les dépendances des sous projets de façon transparentes.**

Les autres Sections du POM

Les autres Sections du POM: developers

- **La section developers permet de recenser tous les développeur qui travail sur un projet Maven.**

```
<developers>
    <developer>
        <id>eric</id>
        <name>Eric</name>
        <email>eredmond@codehaus.org</email>
        <url>http://eric.propellors.net</url>
        <organization>Codehaus</organization>
        <organizationUrl>http://mojo.codehaus.org</organizationUrl>
        <roles>
            <role>architect</role>
            <role>developer</role>
        </roles>
        <timezone>-6</timezone>
        <properties>
            <picUrl>http://tinyurl.com/prv4t</picUrl>
        </properties>
    </developer>
</developers>
```

- **Il existe aussi la balise `contributors` pour ceux qui participent au projet sans être développeur (les testeur par expl.). Il s'y trouve les même sous-balises sauf pour `id` qui est en moins.**

- **id, name, email: Ils correspondent aux ID du développeur, le nom du développeur et à son adresse e-mail.**
- **organization, organizationUrl: Nom de l'organisation du développeur et son URL.**
- **roles: Précise les rôles joués par une personne. Une personne peut avoir plusieurs rôles.**
- **timezone: Une numériques en heures GMT correspondant aux pays des développeurs**
- **properties: Cet élément est l'endroit où toutes autres propriétés peuvent être rajoutés (lien vers une page perso, vers une messagerie, …). Différents plugins peuvent utiliser ces propriétés, ou ils peuvent simplement être lus par les autres développeurs.**

Les autres Sections du POM: licenses

- **Les licences sont des documents légaux qui définissent comment et quand un projet (ou une partie d'un projet) peut être utilisé. Notez qu'un projet devrait répertorier uniquement les licences qui s'appliquent directement à ce projet, et non des licences qui s'appliquent à la liste des dépendances de ce projet.**

```
<licenses>
  <license>
    <name>The Apache Software License, Version 2.0</name>
    <url>http://www.apache.org/licenses/LICENSE-2.0.txt</url>
    <distribution>repo</distribution>
    <comments>A business-friendly OSS license</comments>
  </license>
</licenses>
```

- **name, url et comments: précise le nom de la license, son url (son contenu) et un commentaire.**
- **distribution: Cette section explique comment le projet peut être légalement distribué. Les deux méthodes sont repo (ils peuvent être téléchargés à partir d'un repo Maven) ou manual (ils doivent être installés manuellement).**

Les autres Sections du POM: distributionManagement

- **Cette balise précise où doivent être déployé nos projets packagés et nos sites générés. Si la destination nécessite une authentification alors il faut une balise `<server>` dans le fichier setting.xml**

```
<distributionManagement>
    <repository>
        <id>nexus</id>
        <name>My releases</name>
        <url>http://localhost:8081/nexus/content/repositories/releases/</url>
    </repository>

    <snapshotRepository>
        <id>nexus</id>
        <name>My snapshots</name>
        <url>http://localhost:8081/nexus/content/repositories/snapshots/</url>
    </snapshotRepository>

    <site>
        <id>monappli-repo-site</id>
        <name>repo du site de mon appli</name>
        <url>file://C:/objis/forma_Maven/site</url>
    </site>
</distributionManagement>
```

- **La balise `<repository>` concerne les releases, `<snapshotRepository>` les snapshots et `<site>` le site généré.**
- **La balise `<url>` indique la destination du déploiement. Cela peut être un répertoire, un site ftp ou bien encore un repo distant comme nexus ou archiva.**

Les autres Sections du POM: scm

- La balise scm veut dire Source Control Management. Elle indique où sont déposé les différentes versions du code source des applications. Il existe plusieurs outils pour ces dépôts: CVS, SVN, GIT, …

```
<scm>
  <connection>scm:svn:http://localhost/svn/demomaven/trunk/monappli</connection>
  <developerConnection>scm:svn:https://localhost/svn/demomaven/trunk/monappli</developerConnection>
  <tag>HEAD</tag>
  <url>http://localhost/viewvc/demomaven/trunk/monappli/</url>
</scm>
```

- connection: pour les besoins en lecture seul des plugins Maven.
- developerConnection: pour les besoins en écriture des plugins Maven.
- tag: Indique sous quel tag le projet vit. HEAD (la racine du SMC) est la valeur par défaut.
- url: URL du référentiel consultable avec un navigateur publiquement. Par exemple, par l'intermédiaire ViewVC. Le code source est présenté sous la forme de pages html.

Les autres Sections du POM: repositories

- **Précise un où plusieurs repo où Maven peut télécharger des releases ou des snapshots d'artifacts.**

```
<repositories>
    <repository>
        <releases>
            <enabled>false</enabled>
            <updatePolicy>always</updatePolicy>
            <checksumPolicy>warn</checksumPolicy>
        </releases>
        <snapshots>
            <enabled>true</enabled>
            <updatePolicy>never</updatePolicy>
            <checksumPolicy>fail</checksumPolicy>
        </snapshots>
        <id>codehausSnapshots</id>
        <name>Codehaus Snapshots</name>
        <url>http://snapshots.maven.codehaus.org/maven2</url>
    </repository>
</repositories>
```

- **Il existe aussi pour le téléchargement de releases ou de snapshots de plugins la balise pluginRepositories avec les même sous-balises.**

- **enabled: true ou false pour savoir si ce référentiel est activé**
- **updatePolicy: Cet élément spécifie combien de fois les mises à jour seront tentées. Les choix sont:** `always`, `daily` (valeur par défaut), `interval:X` (où X est un entier en minutes) **ou** `never`
- **checksumPolicy: Quand Maven déploie des artefacts dans un référentiel, il déploie aussi des fichiers de sommes de contrôle (checksum) qui permettent de vérifier l'identité d'un artefact. Les options sont:** `ignore`, `fail` **ou** `warn` **(pour les checksums manquant ou incorrect).**
- **url: URL d'un repo distant accessible publiquement sur internet ou d'un repo distant d'entreprise**

Les autres Sections du POM: mailingList et issueManagement

- **Les listes de diffusion sont un excellent outil pour garder le contact avec des personnes sur un projet. La plupart des listes de diffusion sont pour des développeurs ou des utilisateurs.**

```
<mailingLists>
    <mailingList>
        <name>User List</name>
        <subscribe>user-subscribe@127.0.0.1</subscribe>
        <unsubscribe>user-unsubscribe@127.0.0.1</unsubscribe>
    </mailingList>
</mailingLists>
```

- **subscribe, unsubscribe: Ces deux éléments spécifient les adresses e-mail à utiliser pour s'abonner ou se désabonner à cette liste de diffusion.**

- **La balise `<issueManagement>` définit le système utilisé pour suivre les anomalies (Bugzilla, TestTrack, ClearQuest, etc). Cette information se retrouve dans le site généré pour la documentation et peut être accessible par un plugin.**

```
<issueManagement>
    <system>Bugzilla</system>
    <url>http://127.0.0.1/bugzilla/</url>
</issueManagement>
```

Les autres Sections du POM: ciManagement

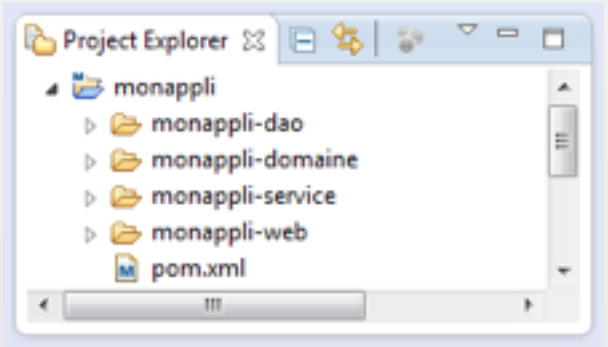
- **Cette balise précise quel est le serveur d'intégration Continue. Il en existe plusieurs: Hudson, Jenkins, Continuum, CruiseControl, … Les informations communes à ces outils sont la notification par e-mail. Le server d'IC pourra lire ces informations dans le pom du projet afin de savoir s'il doit notifier d'une erreur ou d'un succès et à quel adresse e-mail.**

```
<ciManagement>
    <system>continuum</system>
    <url>http://127.0.0.1:8080/continuum</url>
    <notifiers>
        <notifier>
            <type>mail</type>
            <sendOnError>true</sendOnError>
            <sendOnFailure>true</sendOnFailure>
            <sendOnSuccess>false</sendOnSuccess>
            <sendOnWarning>false</sendOnWarning>
            <configuration><address>continuum@127.0.0.1</address></configuration>
        </notifier>
    </notifiers>
</ciManagement>
```

Les Projets Multi-Modules

Création d'applications avec Maven

- Nous allons maintenant regarder une application beaucoup plus lourde.
- Important : Maven valorise la notion de « Separation of concern », ou SoC
  - L'objectif est d'encapsuler des morceaux de soft d'un domaine bien identifié, d'où la notion de module, dans ce que l'on va voir maintenant.
- L'exemple que l'on va voir ici se nomme monappli.
  - Il est constitué de plusieurs modules :
    - domaine: modèle de données
    - Service: l'ensemble des services métier
    - dao: couche d'accès aux données
    - web: l'interface graphique



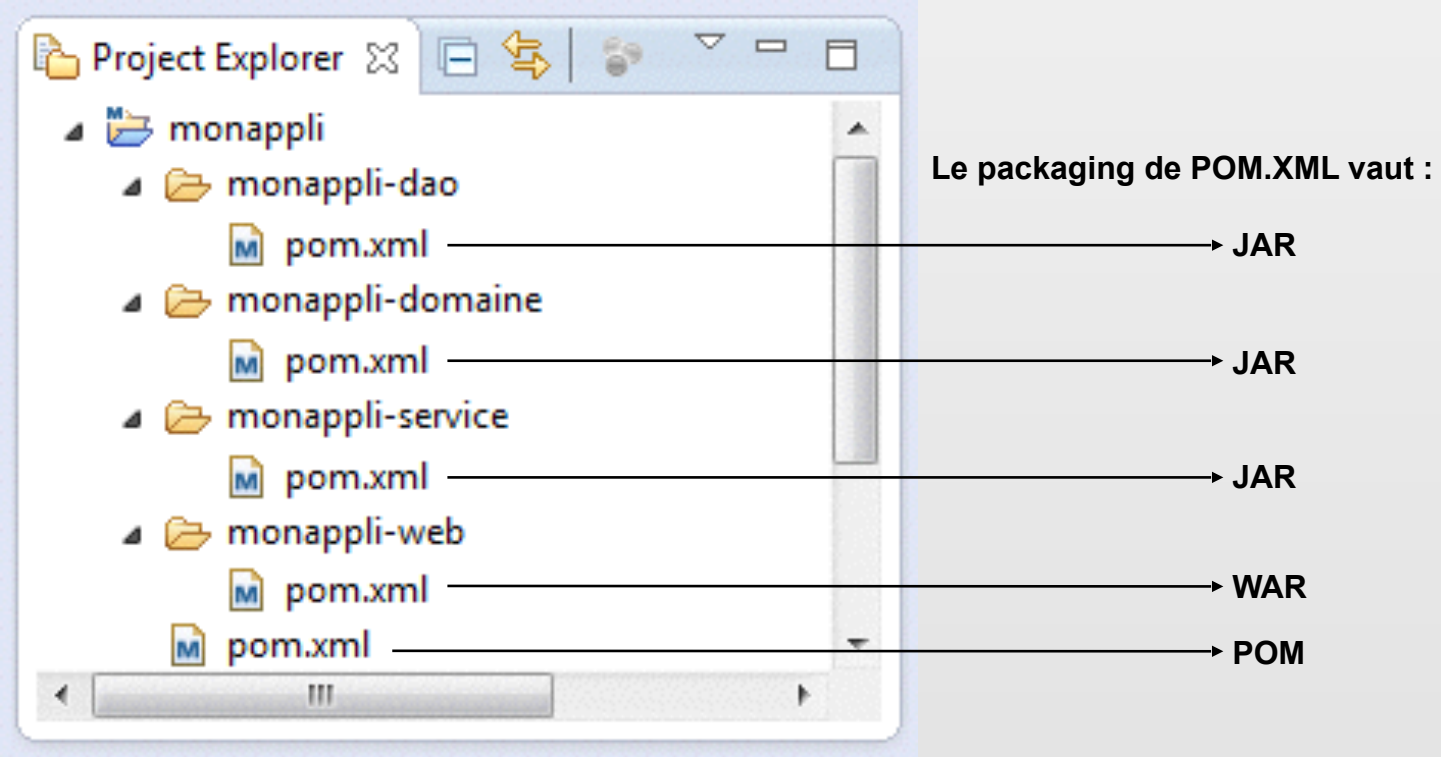

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>com.objis.demomaven</groupId>
  <artifactId>monappli</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>pom</packaging>
  <name>monappli</name>
  <url>http://maven.apache.org</url>
  ...
  <modules>
    <module>monappli-domaine</module>
    <module>monappli-service</module>
    <module>monappli-dao</module>
    <module>monappli-web</module>
  </modules>
  ...
</project>
```

Nom physique des modules

- Ci-contre, quelques lignes du pom.xml principal, qui montre la référence à différents modules livrés.
- Notez aussi la version, qui sera commune à tous les modules.
- Notez la valeur pom pour le packaging (et non jar, war,ect…), cela spécifie à Maven qu'il doit consulter les modules inférieurs pour connaître le type de packaging désiré.

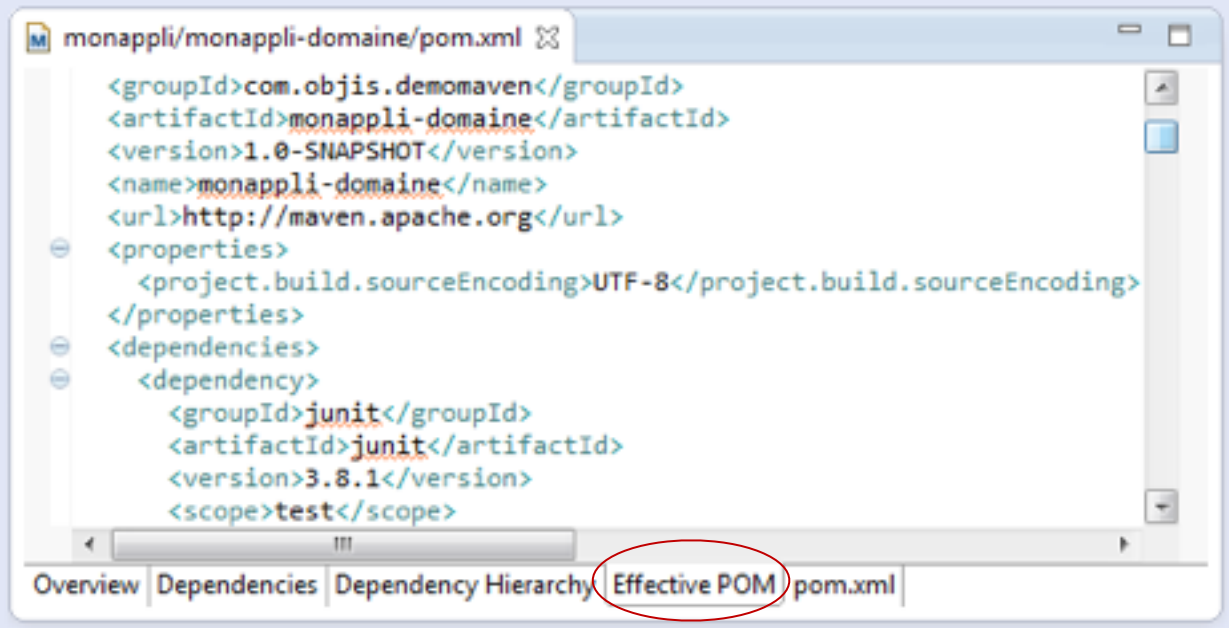
POM ET SOUS POM

- En résumé, dès que la valeur du packaging vaut pom, cela vous permet d'insérer un ensemble de projets différents qui auront leur propre vie
- Exemple avec monappli
  - Voyez les valeurs de packaging pour chaque niveau :
    - Ouvrez la directory et observez



Héritage des POM au niveau projet

- **Une valeur donnée à un élément de POM est donnée en héritage au niveau inférieur.**
  - **C'est le cas par exemple de la dépendance de la librairie junit, qui n'est pas répétée aux niveaux inférieurs, mais qui est pourtant bien déclarée, car héritée.**
- **Il existe une commande très utile de Maven, qui permet de connaître, à chaque niveau de projet, la résultante de l'application de tous les POMs jusqu'à ce niveau.**
  - **Cette commande est `mvn help:effective-pom > help-pom.txt`**
    - **Etant placé à un niveau donné**
    - **Avec le caractère > vous créer un fichier résultat**
  - **Vous pouvez aussi utiliser l'onglet [Effective pom] d'eclipse:**



Le Filtrage et les Profils

Le filtrage

- Le Filtrage permet de faire des variables dynamiques dans les fichiers textes de notre application qui seront pris en charge par Maven
- Ces fichiers texte seront dans le répertoire src/main/resources selon la convention Maven mais peuvent aussi être à d'autres endroits
- Exemples de fichier qui peuvent avoir des variables dynamiques:
  - log4j.properties, db.properties, hibernate.cfg.xml, context.xml
- Les variables dynamiques sont de la forme: `${nomDeLaVariable}`
- Les valeurs de ces variables seront passés au moment de l'exécution de Maven
- Voici un exemple de contenu pour le fichier log4j.properties:

```
log4j.rootLogger=${log.level}, CONSOLE_APP
log4j.appender.CONSOLE_APP=org.apache.log4j.ConsoleAppender
log4j.appender.CONSOLE_APP.layout=org.apache.log4j.PatternLayout
log4j.appender.CONSOLE_APP.layout.ConversionPattern= %d{dd-MM-yyyy
  HH:mm:ss:SSS} %-4r %-5p %c %x - %m%n
```

- Dans ce fichier nous avons une seul variable dynamique: le niveau de log pour la sortie des logs dans la console. Cette variable pourra prendre les valeurs suivantes: DEBUG, INFO, WARN, ERROR, ...

Le filtrage

- La fonction de filtrage n'est pas activé par défaut. Elle peut être activé au niveau du fichier pom.xml de cette manière pour chaque dossier contenant des fichiers texte avec des variables dynamiques:

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

- Cela étant en place, on pourra passé en ligne de commande la valeur de la variable `log.level` avec le paramètre `-D`, exemple:

```
mvn package "-Dlog.level=DEBUG"
```

- En décompressant le package généré dans le répertoire target on pourra alors y retrouvé le fichier log4j.properties généré où l'on pourra voir la ligne suivante:

```
log4j.rootLogger=DEBUG, CONSOLE_APP
```

- La variable dynamique a bien était remplacé par la valeur que nous avons fourni en paramètre

Le filtrage

- Il est possible d'avoir un fichier de filtrage contenant un ensemble de valeurs pour nos variables dynamiques, `LOCAL.properties` par exemple, qui se trouvera par convention dans le répertoire `src/main/filters`. Celui-ci pourra avoir, par exemple, le contenue suivant:
  - `log.level=DEBUG`

- Pour que ce fichier soit pris en compte il faudra rajouter dans le pom.xml les balises xml suivantes (à coté de `resources`) dans la section `build`:

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

- En lançant la commande `mvn package` sans paramètre on pourra alors observé que le fichier log4j.properties dans notre package généré (monappli-dao-1.0.jar par exemple) aura bien la ligne suivante:
  - `log4j.rootLogger=DEBUG, CONSOLE_APP`

Le filtrage

- Une bonne pratique est de variabiliser le nom du fichier de filtrage de la sorte:

```
<build>
    . . .
    <filters>
        <filter>src/main/filters/${env}.properties</filter>
    </filters>
    . . .
</build>
```

- On pourra donner une valeur par défaut à la variable env en rajoutant les balises suivantes dans la section project:

```
<properties>
    <env>LOCAL</env>
</properties>
```

- Avec cette technique on pourra donc avoir un fichier de filtrage par environnement: LOCAL, DEV, RECETTE, PREX, PROD, …

- Pour générer un package qui prendra en compte le fichier de filtrage PROD.properties on pourra lancer la ligne de commande suivante:

```
mvn package "-Denv=PROD"
```

Les profiles

- Les profiles permettent de définir un ensemble de valeurs pour les variables dynamiques et aussi de jouer sur la configuration du pom
- Les profiles permettent de créer des variations dans le cycle de construction, afin de s'adapter à des particularités, comme
  - Des construction pour des plateformes différentes
  - Tester en utilisant différentes DB
  - Référencer un Système de fichiers local
  - Etc…
- Les profiles sont déclarés dans des entrées du POM
  - Ils peuvent être « activés » de différentes manières
  - Ils modifient le POM au moment du build, prenant en considération les paramètres passés
- Les profiles pourront être définis à deux endroits :
  - Le fichier settings.xml de la directory .m2 (repository)
  - Le POM lui-même
- Dans un de ces fichiers, vous pouvez définir les éléments suivants :
  - Repositories, pluginRepositories, dependencies, plugins, modules, reporting
  - dependencyManagement, distributionManagement

Les profiles

- **Vous pouvez activer un ou plusieurs profiles de 3 façons:**
  - **Avec le sélecteur –P de la commande mvn**
    - **qui prendra en argument la liste des ids de profile que vous voulez activer, ex :**
      - `Mvn -Pprofile1,profile2 install`
  - **Via la section activeProfile, qui référence les profiles définis dans settings.xml, voir ci-dessous à droite**
  - **Via un «déclencheur » ou trigger, qui, si vérifié, activera le profile déclaré. Ci-dessous à gauche, le profile1 est activé si la variable système nommé environment vaut test**

```
<profile>
  <id>profile1</id>
 ...
  <activation>
    <property>
      <name>environment</name>
      <value>test</value>
    </property>
  </activation>
</profile>
```

```
<settings>
...
  <profiles>
    <profile>
      <id>profile1</id>
      ...
    </profile>
  </profiles>
  <activeProfiles>
        <activeProfile>profile1<
  </activeProfiles>
...
</settings>
```

Les profiles

- **Observez ces 2 profiles du POM de monappli-dao**

```
<profiles>
	<profile>
		<id>test_pom</id>
		<properties>
			<db.url>jdbc:mysql://mon_serveur_de_test/db_test</db.url>
			<db.driver>com.mysql.jdbc.Driver</db.driver>
			<db.username>user1</db.username>
			<db.password>user1pwd</db.password>
		</properties>
		<activation>
			<property>
				<name>test_pom</name>
			</property>
		</activation>
	</profile>
	<profile>
		<id>dev_pom</id>
		<properties>
			<db.url>jdbc:mysql://localhost/db_dev</db.url>
			<db.driver>com.mysql.jdbc.Driver</db.driver>
			<db.username>root</db.username>
			<db.password />
		</properties>
		<activation>
			<property>
				<name>dev_pom</name>
			</property>
		</activation>
	</profile>
</profiles>
```

Les profiles

- **Deux profiles existent, l'un avec test_pom et l'autre avec dev_pom**
  - **Chacun de ces profiles ont des propriétés pour la base de donnée qui corresponde à un environnement donné.**
    - **Observez aussi le déclenchement par la présence d'une propriété système**
  - **Le lancement dans une configuration donnée se fera par :**
    - `mvn compile –Dtest_pom`
    - `mvn compile –Ddev_pom`