

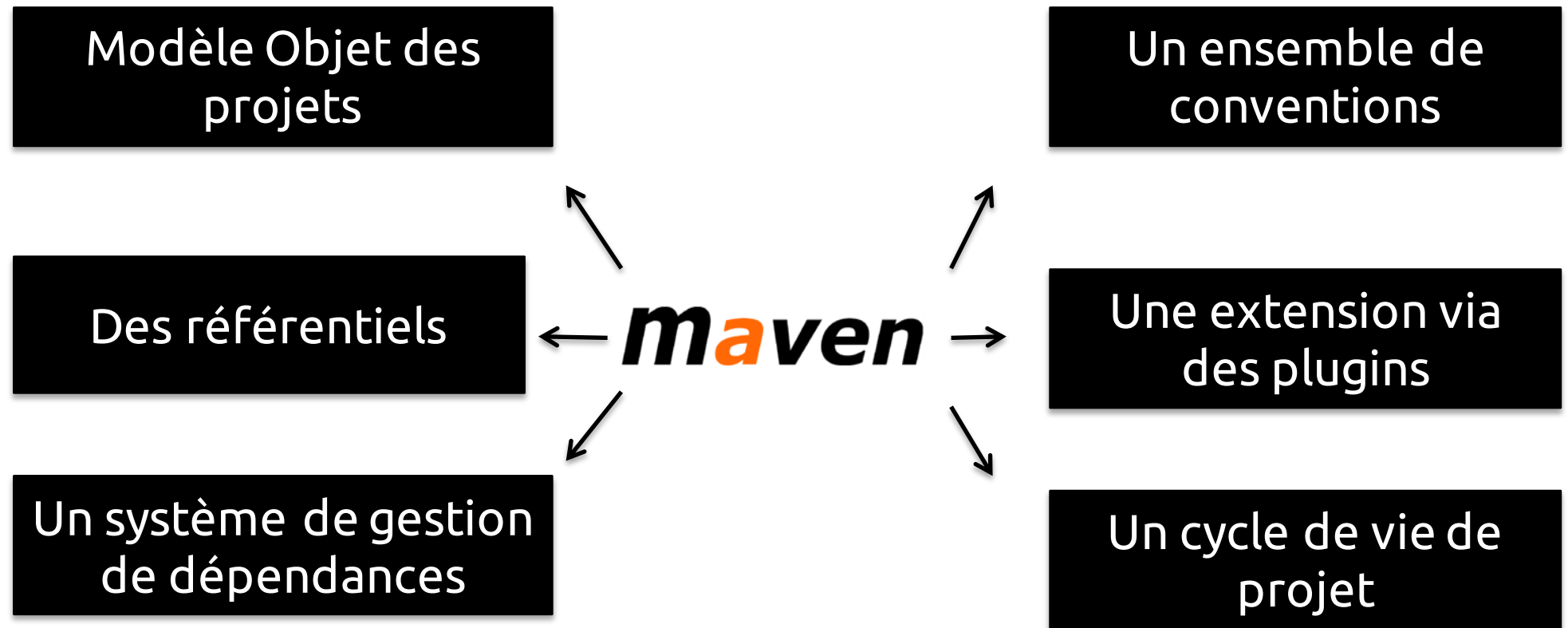


**digitalent**  
FORMATION

# Maven par la pratique

# Introduction

# Maven ?



# Modèle objet de projet

- Définit par un fichier de configuration XML appelé **pom.xml**
  - POM = **P**roject **O**bjets **M**odel
- Configure des informations concernant l'identité du projet (groupId, artifactId, version, packaging)
- Définit de façon déclarative les différentes tâches à exécuter
- etc...

```
<?xml version="1.0" encoding="UTF-8"?>
<project
  xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

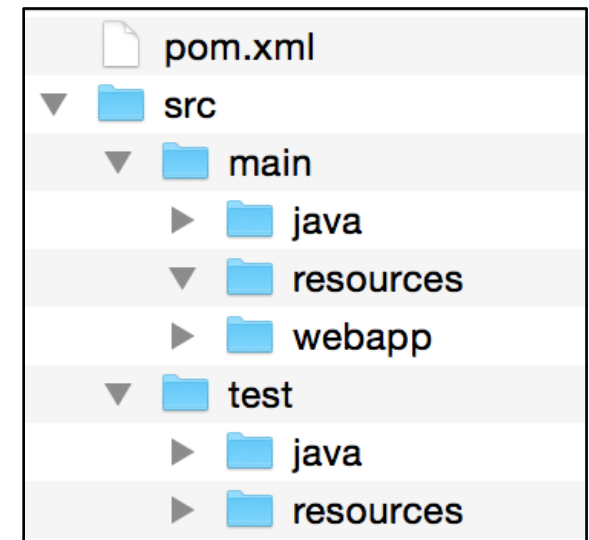
  <modelVersion>4.0.0</modelVersion>

  <groupId>dta</groupId>
  <artifactId>mywebapp</artifactId>
  <version>1.0-SNAPSHOT</version>
  <packaging>war</packaging>

  <name>mywebapp</name>
  <properties>
```

# Un ensemble de conventions

- Maven privilégie une approche "convention over configuration"
  - Les sources java sont supposés être dans le répertoire **src/main/java**
  - Les autres fichiers nécessaire au programme (.properties, .xml) sont supposés être dans le répertoire **src/main/resources**
  - Les sources java des tests sont supposés être dans le répertoire **src/test/java**
  - Les autres fichiers nécessaire aux tests (.properties, .xml) sont supposés être dans le répertoire **src/test/resources**
  - Les fichiers générés par Maven sont dans le répertoire **target**
  - Les fichiers web (html, css, jsp, ...) sont supposés être dans le répertoire **src/main/webapp**



# Gestion de dépendances

- Maven permet de gérer les dépendances d'un projet en configurant le fichier pom.xml
- Les dépendances sont identifiées grâce aux informations : groupId, artifactId, version
- Les dépendances peuvent être de plusieurs types : jar, zip, pom, ...

Important pour les dépendances

```
<dependencies>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-web-api</artifactId>
    <version>7.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Les dépendances se mettent après le build exemple:

```
<?xml version="1.0" encoding="UTF-8" ?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/x
<groupId>com</groupId>
<artifactId>pizzeria-console-objet</artifactId>
<version>1.0.0</version>
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <artifactId>maven-compiler-plugin</artifactId>
        <configuration>
          <source>1.8</source>
          <target>1.8</target>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
</build>
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.12</version>
    <scope>test</scope>
  </dependency>
</dependencies>
</project>
```

# Un cycle de vie de projet

- Maven propose une construction de projet en plusieurs phases :
  - ...
  - generate-sources
  - process-resources
  - compile
  - process test resources
  - test
  - package
  - install
  - deploy
  - ...

generate-sources

process-resources

compile

pro-test-resources

test

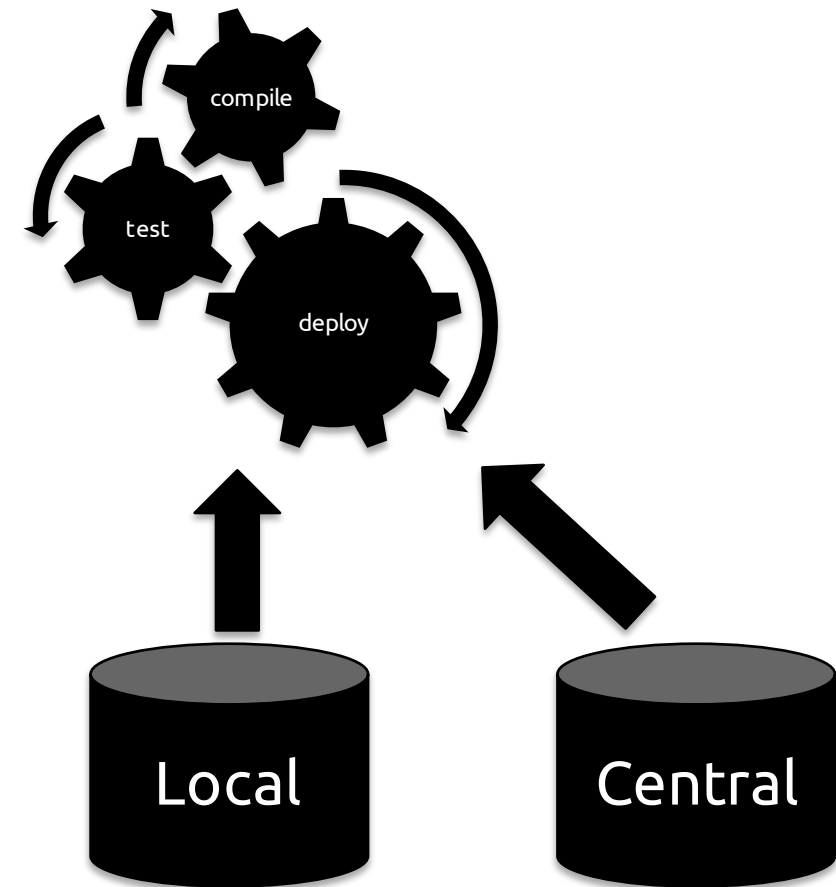
package

install

deploy

# Des référentiels

- Les artefacts Maven peuvent être stockés dans des référentiels locaux ou distants
- Le référentiel central Maven (<http://search.maven.org/>) héberge une grande base d'artefacts réutilisables dans des projets





# Des plugins

- Les plugins permettent d'étendre la construction de projet Maven pour ajouter et configurer des tâches spécifiques

```
<plugins>
  <plugin>
    <artifactId>maven-war-plugin</artifactId>
    <version>2.6</version>
    <configuration>
      <failOnMissingWebXml>>false</failOnMissingWebXml>
    </configuration>
  </plugin>
  <plugin>
    <groupId>org.wildfly.plugins</groupId>
    <artifactId>wildfly-maven-plugin</artifactId>
    <version>3.0</version>
  </plugin>
</plugins>
```

# Que faire avec Maven ?

- Créer un projet à l'aide de modèles
- Compiler un projet avec ses dépendances
- Exécuter des tests unitaires, d'intégrations, ...
- Gérer les versions d'un projet
- Assembler un projet en livrable (JAR, ZIP, WAR, EAR, ...)
- Archiver l'artefact d'un projet dans un référentiel
- Générer un rapport de qualité du code
- Déployer un artefact sur un serveur
- ...

# Installation

# Installer Maven

- Prérequis :
  - Avoir un JDK 1.7+ installé
- Télécharger Maven
  - <https://maven.apache.org/download.cgi>
- Décompresser l'archive
- Mettre à jour la variable PATH
- Tester l'installation de Maven
  - `mvn -v`

POM

# Le modèle objet de projet

## POM

### Relations du POM

Coordonnées d'un projet  
Sous-modules  
Héritage  
Dépendances

### Configuration de construction

Répertoires de sources  
Plugins  
Reporting

### Informations générales

Généralités  
Contributeurs  
Licences

### Configuration d'environnement

Informations sur l'environnement  
Les profils

# Coordonnées d'un projet

- Un projet est identifié à l'aide des informations suivantes :
  - groupId
  - artifactId
  - version
  - packaging (par défaut "jar")

# Super POM

- Tous les projets Maven héritent d'un super POM
- Pour visualiser le super POM
  - Désarchiver le fichier **M2\_HOME/lib/maven-model-builder-3.3.3.jar**
  - Le super POM se trouve dans le fichier **org/apache/maven/model/pom-4.0.0.xml**
- Il contient la configuration Maven par défaut => la convention Maven



# POM Effectif

- Les POMs héritent leur configuration d'autres POMs (à minima du super POM).
- Le modèle objet du projet va au final être la combinaison des configurations (via le lien d'héritage)
- Pour visualiser toute la configuration d'un POM  
**`mvn help:effective-pom`**

# Versions d'un projet

- Maven propose une utilisation des versions d'un projet avec le format suivant :

**<version majeure>.<version mineure>.<version incrémentale>-<qualifieur>**

- Quelques exemples :
  - 3.4.5 → version majeure 3, mineure 4 et incrémentale 5
  - 12 → version majeure 12
  - 2.5-beta-01 → version majeure 2, mineure 5 et qualifieur "beta-01"

# Propriétés (1)

- Un POM peut contenir des références à des propriétés via
  - **`${propriété}`**
- Maven initialise 3 variables :
  - **env**
    - permet d'accéder aux variables d'environnement du système d'exploitation.  
Exemple `${env.PATH}`
  - **project**
    - permet d'accéder au POM. Exemple `${project.version}`
  - **settings**
    - permet d'accéder aux informations de configuration Maven (fichier settings.xml).  
Exemple : `${settings.offline}`

# Propriétés

- Il est possible d'ajouter des propriétés dans un fichier pom.xml via la balise `<properties>`
- Dans l'exemple, ci-contre, les propriétés sont accessibles via :
  - `${junit.version}`
  - `${source.version}`

```
<properties>  
  <junit.version>4.2</junit.version>  
  <source.version>1.8</source.version>  
</properties>
```

# Dépendances d'un projet (1)

- Les dépendances d'un projet se déclarent dans la section **<dependencies>**
- La balise **<scope>** définit la portée de la dépendance

```
<dependencies>
  <dependency>
    <groupId>javax</groupId>
    <artifactId>javaee-web-api</artifactId>
    <version>7.0</version>
    <scope>provided</scope>
  </dependency>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>
```

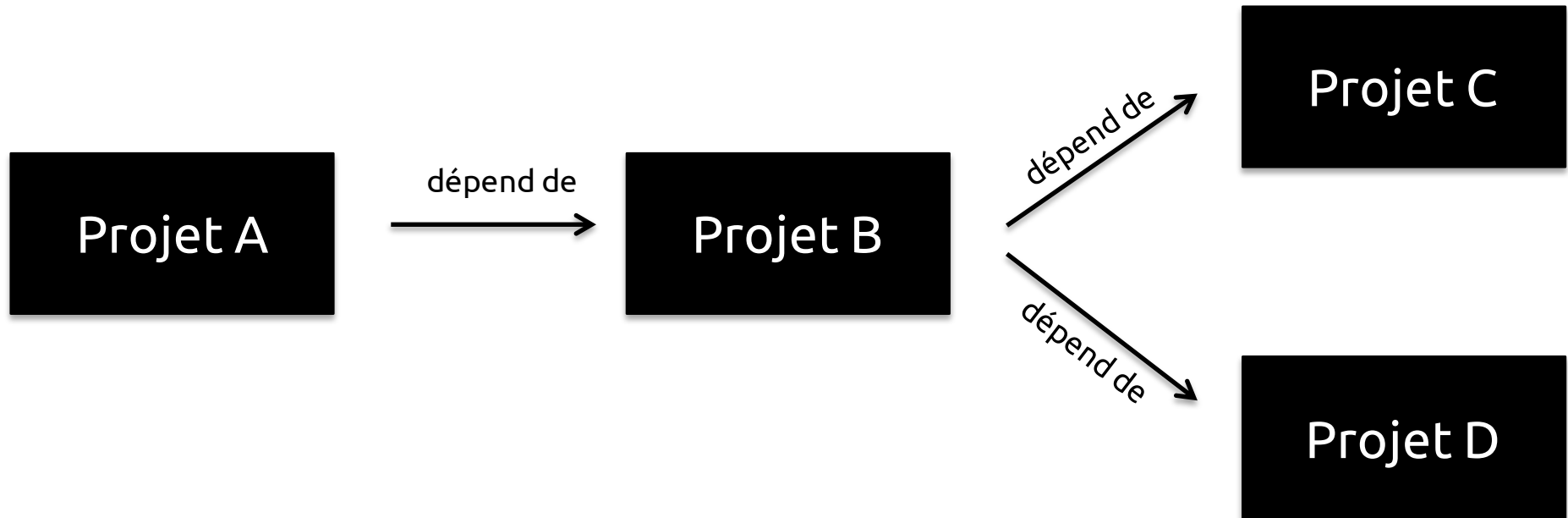
# Portée d'une dépendance

La portée se mets dans le scope, qui se situe juste avant la fin de <dependency>

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>com</groupId>
  <artifactId>pizzeria-console-object</artifactId>
  <version>1.0.0</version>
  <build>
    <pluginManagement>
      <plugins>
        <plugin>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <source>1.8</source>
            <target>1.8</target>
          </configuration>
        </plugin>
      </plugins>
    </pluginManagement>
  </build>
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.12</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

- **compile (par défaut)**
  - indispensable à la compilation et à l'exécution
- **runtime**
  - inutile à la compilation mais indispensable à l'exécution
- **test**
  - utile uniquement à la compilation et l'exécution des tests
- **provided**
  - indispensable à la compilation, dépendance fournie par l'environnement d'exécution (par exemple, un serveur d'application)
- ~~**system**~~
  - dépendance à récupérer en local, hors dépôt Maven => à ne pas utiliser dans la mesure du possible
- **import (Maven version >=2.0.9)**
  - Applicable uniquement à la section <dependencyManagement> à une dépendance de type pom. Il permet de mutualiser une liste de dépendances sans recourir à l'héritage.

# Transitivité des dépendances



=> Le projet A peut utiliser les classes des projets B, C et D

# Exclure une dépendance

- Il est possible d'exclure une dépendance transitive via la balise **<exclusions>**

```
<dependency>
  <groupId>org.hibernate</groupId>
  <artifactId>hibernate-core</artifactId>
  <version>4.1.2</version>
  <exclusions>
    <exclusion>
      <groupId>dom4j</groupId>
      <artifactId>dom4j</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```



# Intervalles de versions

- Il est possible de définir un ensemble de versions acceptables pour une dépendance donnée.

**(,)** → définir un intervalle ouvert.

Exemples:

- (3.8,4.2)
- (,4.2)
- (3.8,)

**[,]** → définir un intervalle fermé.

Exemples:

- [3.8, 4.2]
- [,4.2]
- [3.8,]

```
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>[4.0,4.2]</version>
  <scope>test</scope>
</dependency>
```

# Dépendance optionnelle

- Les dépendances marquées comme optionnelles ne sont pas transitives.
  - Si un projet A dépend d'un projet B qui dépend d'un projet C. Si C est marqué comme optionnel dans B, alors **A ne dépendra pas automatiquement de C.**

```
<dependency>
  <groupId>net.sf.ehcache</groupId>
  <artifactId>ehcache</artifactId>
  <version>1.4.1</version>
  <optional>true</optional>
</dependency>

<dependency>
  <groupId>swarmcache</groupId>
  <artifactId>swarmcache</artifactId>
  <version>1.0RC2</version>
  <optional>true</optional>
</dependency>
```

# TP 01

- Prise en main & travail avec des dépendances
- Travailler avec les dépendances

# Plugins

# Quelques plugins - core

- **maven-clean-plugin**
  - Supprime le répertoire de sortie utilisé pour le projet (par défaut `${basedir}/target`)
- **maven-compiler-plugin**
  - Compile les sources Java
- **maven-deploy-plugin**
  - Déploie un artefact dans un dépôt distant
- **maven-failsafe-plugin**
  - Lance les tests JUnit dans un classloader isolé
- **maven-install-plugin**
  - Installe un artefact dans le dépôt local
- **maven-resources-plugin**
  - Copie des ressources vers le répertoire de sortie (pour un packaging par exemple)
- **maven-site-plugin**
  - Générer un site pour le projet courant
- **maven-surefire-plugin**
  - Exécute des tests unitaires dans un classloader isolé

# Quelques plugins - packaging

- **maven-ear-plugin**
  - construit un EAR pour le projet courant
- **maven-ejb-plugin**
  - construit un EJB pour le projet courant
- **maven-jar-plugin**
  - construit un JAR pour le projet courant
- **maven-rar-plugin**
  - construit un RAR pour le projet courant
- **maven-war-plugin**
  - construit un WAR pour le projet courant
- **maven-shade-plugin**
  - construit un "Uber-JAR" (JAR qui inclut toutes les dépendances) pour le projet courant
- **maven-source-plugin**
  - construit un JAR contenant les sources du projet courant

# Quelques plugins - reporting

- **maven-changelog-plugin**
  - Génère une liste de changement récents depuis le gestionnaire de version du projet
- **maven-changes-plugin**
  - Génère un rapport depuis un gestionnaire de bugs
- **maven-checkstyle-plugin**
  - Génère un rapport checkstyle
- **maven-javadoc-plugin**
  - Génère la documentation java du projet
- **maven-surefire-report**
  - Génère un rapport d'exécution des tests

# Quelques plugins - Outils

- **maven-ant-plugin**
  - Génère un fichier de build ANT pour le projet
- **maven-antrun-plugin**
  - Exécute des tâches ANT
- **maven-archetype-plugin**
  - Génère un modèle de projet
- **maven-assembly-plugin**
  - Construit un livrable personnalisable (zip par exemple) du projet
- **maven-dependency-plugin**
  - Permet de manipuler (copier, désarchiver) et analyser des dépendances
- **maven-enforcer-plugin**
  - Permet de vérifier les contraintes de l'environnement d'exécution (version de maven, version du JDK, ...)



# Quelques plugins - Autres

- **maven-eclipse-plugin**
  - Génère les fichiers de projet Eclipse
- **sql-maven-plugin**
  - Exécute des scripts SQL
- **cargo-maven2-plugin**
  - Démarrer/Arrêter/Configurer un container Java EE et y déployer des packages

# Configurer plugins

```
<build>
  <finalName>${project.groupId}-banko</finalName>
  <plugins>
    <plugin>
      <groupId>org.apache.maven</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.3</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
  </plugins>
</build>
```

# Goal

- Un "goal" dans Maven représente une action (tâche) à exécuter par un plugin.
- Lancer le goal "clean" du plugin "maven-clean-plugin"
  - `mvn clean:clean`
- Générer les fichiers projets Eclipse
  - `mvn eclipse:eclipse`

# TP 02

- Se familiariser à l'utilisation des plugins

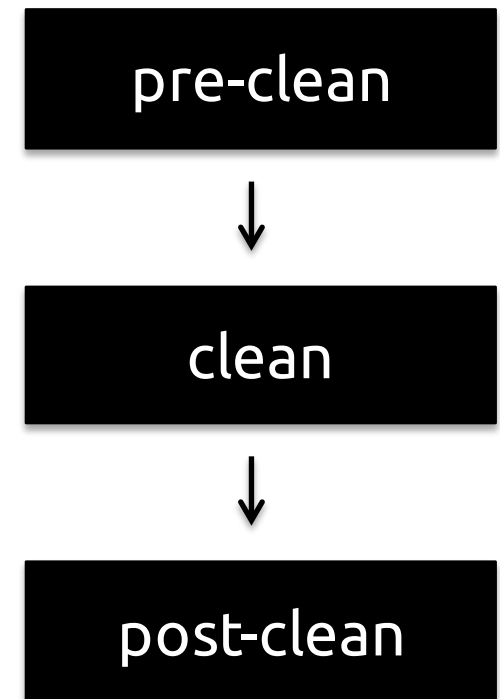
# Cycle de vie du build

# Cycle de vie

- Le cycle de vie d'un projet Maven est composé d'une **séquence de phases** dans lesquelles sont exécutés des **actions** ("goals")
- Maven propose 3 types de cycle de vie :
  - **clean**
  - **default** (build)
  - **site**

# Cycle de vie "clean"

- L'exécution de la commande "mvn clean" invoque le cycle de vie "clean".
- Il est composé de 3 phases : **pre-clean**, **clean** et **post-clean**
- L'action "clean" du plugin "maven-clean-plugin" supprime le répertoire de build (par défaut `${basedir}/target`)



# Plugins, goal et phase

- Il est possible de configurer l'exécution d'une tâche ("goal") d'un plugin dans une phase Maven.

```
<build>
  <plugins>
    <plugin>
      <groupId>org.codehaus.mojo</groupId>
      <artifactId>exec-maven-plugin</artifactId>
      <version>1.4.0</version>
      <executions>
        <execution>
          <phase>pre-clean</phase>
          <goals><goal>exec</goal></goals>
        </execution>
      </executions>
      <configuration>
        <executable>cp</executable>
        <arguments>
          <argument>-R</argument>
          <argument>target</argument>
          <argument>target-backup</argument>
        </arguments>
      </configuration>
    </plugin>
  </plugins>
</build>
```



# Cycle de vie "build" (1)

- Il s'agit du cycle de vie par défaut qui commence par une phase "validate" et se termine par la phase "deploy".

# Cycle de vie "build" (2)

Phase du cycle de vie	Description
<b>validate</b>	Valide le projet et la disponibilité de toutes les informations requises pour le build
<b>generate-sources</b>	Génère le code source nécessaire pour l'inclure à la compilation
<b>process-sources</b>	Traite le code source, pour filtrer certaines valeurs par exemple
<b>generate-resources</b>	Génère les ressources à inclure dans le package
<b>process-resources</b>	Copie et traite les ressources dans leur répertoire destination, afin qu'elles soient prêtes pour le packaging
<b>compile</b>	Compile le code source du projet
<b>process-classes</b>	Traite à posteriori les fichiers générés par la compilation, pour modifier du bytecode par exemple
<b>generate-test-sources</b>	Génère le code source des tests pour l'inclure à la compilation
<b>process-test-sources</b>	Traite le code source des tests, pour filtrer certaines valeurs par exemple

# Cycle de vie "build" (3)

Phase du cycle de vie	Description
<b>generate-test-resources</b>	Génère les ressources pour les tests
<b>process-test-resources</b>	Copie et traite les ressources de test dans le répertoire destination des tests
<b>test-compile</b>	Compile le code source des tests dans le répertoire destination des tests
<b>test</b>	Exécute les tests en utilisant le framework de test approprié. Le code de ces tests ne doit pas être nécessaire au packaging ni au déploiement.
<b>prepare-package</b>	Effectue les opérations nécessaires pour la préparation du package avant que celui-ci ne soit réellement créé. Il en résulte souvent une version dézippée et prête à être packagée du futur package (Maven 2.1+)
<b>package</b>	Package le code compilé dans un format distribuable tel que JAR, WAR ou EAR
<b>pre-integration-test</b>	Effectue les actions nécessaires avant de lancer les tests d'intégration, comme configurer un environnement par exemple.
<b>integration-test</b>	Traite et déploie si nécessaire le package dans l'environnement où les tests pourront être exécutés
<b>post-integration-test</b>	Effectue les actions nécessaires à la fin de l'exécution des tests d'intégration, comme nettoyer l'environnement par exemple

# Cycle de vie "build" (4)

Phase du cycle de vie	Description
<b>verify</b>	Lance des points de contrôle pour vérifier que le package est valide et qu'il passe les critères qualité
<b>install</b>	Installe le package dans le dépôt local, celui-ci pourra ainsi être utilisé comme dépendance par d'autres projets locaux
<b>deploy</b>	Copie le package final sur le dépôt distant. Permet de partager le package avec d'autres utilisateurs et projets (souvent pertinent pour une vraie livraison)

# Cycle de vie "site"

- Le cycle de vie "site" contient 4 phases
  - pre-site
  - site
  - post-site
  - site-deploy
- Exemple d'exécution appliquant ce cycle de vie :
  - mvn site:site
  - mvn site:deploy

# Cycle de vie / type de package

- La balise packaging influe sur les étapes requises pour construire un projet.

# JAR : Cycle de vie

Phase du cycle de vie	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar
install	install:install
deploy	deploy:deploy

# POM: Cycle de vie

Phase du cycle de vie	Goal
package	site:attach-descriptor
install	install:install
deploy	deploy:deploy



# Plugins : Cycle de vie

Phase du cycle de vie	Goal
generate-resources	plugin:descriptor
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	jar:jar, plugin:addPluginArtifactMetadata
install	install:install, plugin:updateRegistry
deploy	deploy:deploy

# EJB : Cycle de vie

Phase du cycle de vie	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	ejb:ejb
install	install:install
deploy	deploy:deploy

# WAR : Cycle de vie

Phase du cycle de vie	Goal
process-resources	resources:resources
compile	compiler:compile
process-test-resources	resources:testResources
test-compile	compiler:testCompile
test	surefire:test
package	war:war
install	install:install
deploy	deploy:deploy

# EAR : Cycle de vie

Phase du cycle de vie	Goal
generate-resources	ear:generate-application-xml
process-resources	resources:resources
package	ear:ear
install	install:install
deploy	deploy:deploy

# Projets Multimodules

# Projet multimodules

- Est composé de plusieurs projets Maven
  - la composition se fait via la notion de module
- Son packaging est "pom"
- Un projet multimodule n'existe que pour regrouper un ensemble de projets dans un même build et mutualiser de la configuration entre les projets

# Le projet parent

- Le projet parent référence les projets enfants via les balises "modules" et "module"
- La construction du module parent va lancer la construction de chacun des modules enfants

```
<project xmlns="...">
  <modelVersion>4.0.0</modelVersion>
  <groupId>dta</groupId>
  <artifactId>bankonet-parent</artifactId>
  <version>1</version>

  <modules>
    <module>bankonet-rest</module>
    <module>bankonet-client</module>
    <module>bankonet-ihm</module>
  </modules>
```

# Un projet enfant

- Un projet enfant référence un projet parent via la balise parent
- Ses coordonnées "groupId" et "version" peuvent être héritées du projet parent

```
<project>

  <modelVersion>4.0.0</modelVersion>

  <parent>
    <groupId>dta</groupId>
    <artifactId>bankonet-parent</artifactId>
    <version>1</version>
    <relativePath>../pom.xml</relativePath>
  </parent>

  <artifactId>bankonet-client</artifactId>
```



# Gestion des dépendances

- Dans un projet multi-modules, il est recommandé d'harmoniser les versions des librairies et des plugins utilisés dans tous les modules.
- Cela peut se faire en configurant les sections **<dependencyManagement>** et **<pluginManagement>** dans le pom parent
  - Les pom enfants n'auront plus à redéfinir, par exemple, les versions des dépendances

```
<dependencyManagement>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>${junit.version}</version>
    <scope>test</scope>
  </dependency>
</dependencyManagement>

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

# TP 03

- Projets Multi-modules

# Profils

# Profils

- Les profils permettent d'adapter un build à plusieurs contextes d'exécution (environnements).
- Il devient possible de configurer un comportement différent entre build sur la machine d'un développeur et un build sur un environnement de test.
- Les profils rendent les projets Maven "portables" (facilité avec laquelle un projet peut être construit sur différents environnements).
- Un profil peut-être défini :
  - dans un fichier pom.xml
  - dans un fichier profiles.xml
  - dans le fichier ~/.m2/settings.xml
  - dans le fichier \${M2\_HOME}/conf/settings.xml

# Configurer un profil

```
<profiles>
  <profile>
    <id>production</id>
    <build>
      <plugins>
        <plugin>
          <groupId>org.apache.maven.plugins</groupId>
          <artifactId>maven-compiler-plugin</artifactId>
          <configuration>
            <debug>false</debug>
            <optimize>true</optimize>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
</profiles>
```

**mvn clean install -Pproduction**

# Balises autorisés dans un profil

```
<project>
  <profiles>
    <profile>
      <build>
        <defaultGoal>...</defaultGoal>
        <finalName>...</finalName>
        <resources>...</resources>
        <testResources>...</testResources>
        <plugins>...</plugins>
      </build>
      <reporting>...</reporting>
      <modules>...</modules>
      <dependencies>...</dependencies>
      <dependencyManagement>...</dependencyManagement>
      <distributionManagement>... </distributionManagement>
      <repositories>...</repositories>
      <pluginRepositories>...</pluginRepositories>
      <properties>... </properties>
    </profile>
  </profiles>
</project>
```

# Activer un profil

- Plusieurs possibilités d'activation d'un profil lors d'un build
  - en ligne de commande via l'option -P
    - mvn clean install **-Pproduction**
  - en configurant des conditions d'activation dans le fichier pom.xml

# Profils – Conditions d'activation

- Un profil est activé si toutes les conditions d'activation sont satisfaites.
- Ci-contre, un exemple de configuration complexe

```
<project>
...
<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <activeByDefault>false</activeByDefault>
      <jdk>1.5</jdk>
      <os>
        <name>Windows XP</name>
        <family>Windows</family>
        <arch>x86</arch>
        <version>5.1.2600</version>
      </os>
      <property>
        <name>mavenVersion</name>
        <value>3.3.5</value>
      </property>
      <file>
        <exists>file2.properties</exists>
        <missing>file1.properties</missing>
      </file>
    </activation>
    ...
  </profile>
</profiles>
</project>
```



# Lister les profils actifs

- Pour lister les profils actifs  
`mvn help:active-profiles`

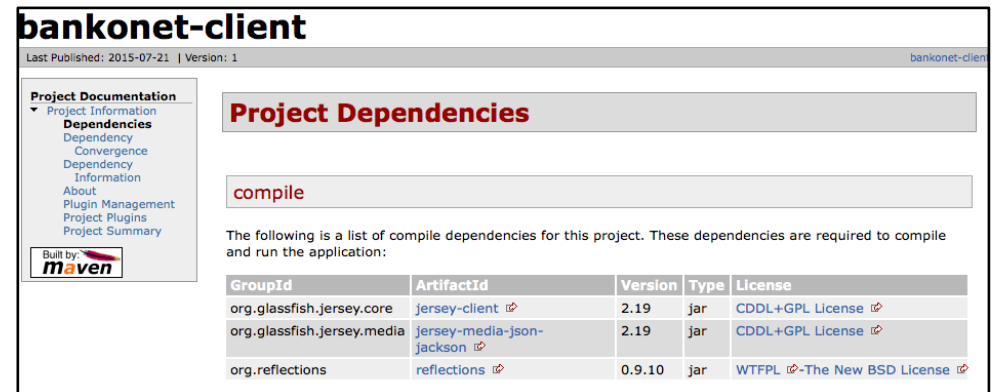
# TP 04

- Utiliser les profils Maven

# Site

# Générer un site


- "maven-site-plugin" permet de générer un site pour un projet Maven
- Quelques commandes :
  - **mvn site:run** → déployer le site dans un serveur
  - **mvn site:site** → génère les fichiers statiques du site dans le répertoire target/site
- Le site généré est personnalisable :
  - <https://maven.apache.org/plugins/maven-site-plugin/>



**bankonet-client**  
Last Published: 2015-07-21 | Version: 1

**Project Documentation**

- Project Information
- Dependencies**
- Dependency
- Convergence
- Dependency
- Information
- About
- Plugin Management
- Project Plugins
- Project Summary

Built by: 

## Project Dependencies

compile

The following is a list of compile dependencies for this project. These dependencies are required to compile and run the application:

GroupId	ArtifactId	Version	Type	License
org.glassfish.jersey.core	<a href="#">jersey-client</a>	2.19	jar	<a href="#">CDDL+GPL License</a>
org.glassfish.jersey.media	<a href="#">jersey-media-json-jackson</a>	2.19	jar	<a href="#">CDDL+GPL License</a>
org.reflections	<a href="#">reflections</a>	0.9.10	jar	<a href="#">WTFPL</a> - <a href="#">The New BSD License</a>