

Les Formations au dojo 道場

MAVEN

Tout ce qu'il faut savoir

Durée : 1 journée

# Sommaire :

- 1 – Maven : Qu'est-ce que c'est ?
- 2 – Configuration du poste de travail
- 3 – Plugins & Lifecycle
- 4 – pom.xml : Le strict nécessaire
  - Dépendances
  - Héritage
  - Properties & placeholders
- 5 – pom.xml : Un peu plus avancé
  - Modules (orchestration)
  - Profiles
  - Insérer un plugin dans un lifecycle

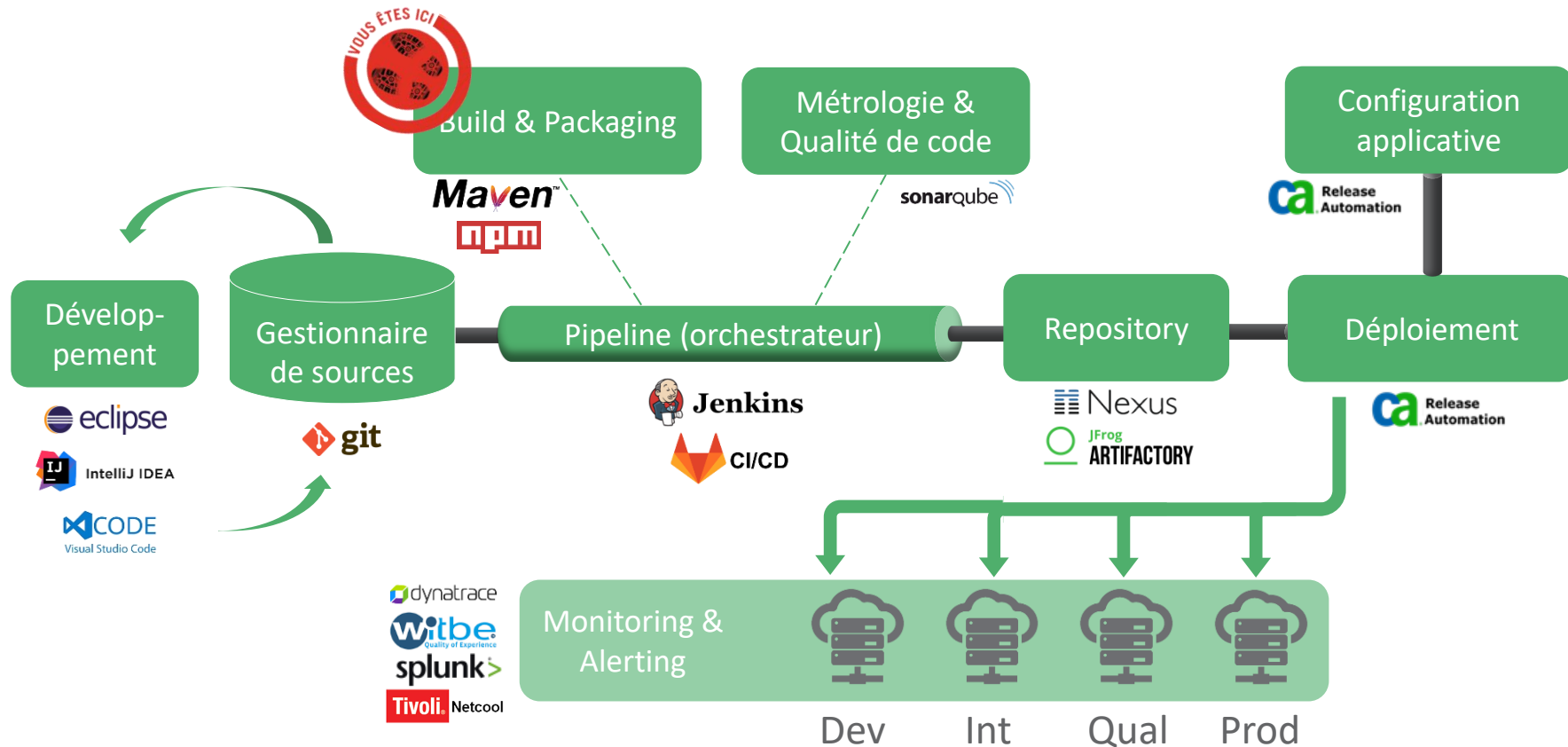
Conclusion

Partie 1 :

Maven : Qu'est-ce que c'est?



# Maven : Qu'est-ce que c'est?



# Maven : Qu'est-ce que c'est?

Maven est un terme Yiddish qui signifie : “Accumulateur de connaissances”

Brillez en soirée grâce à votre formation !

Il est développé en 2003 par Apache Software Foundation

- Organisation à but non lucratif
- Communauté décentralisée de développeurs

Quelques projets de la fondation Apache :

Ant  
Camel  
Cassandra  
Commons  
Hadoop  
Mahout

Apache HTTP Server  
Jmeter  
Kafka  
Apache Logging Services  
Lucene

Maven  
Spark  
Struts  
SVN  
Tomcat



# Maven : Qu'est-ce que c'est?

Depuis, Maven continue à évoluer :

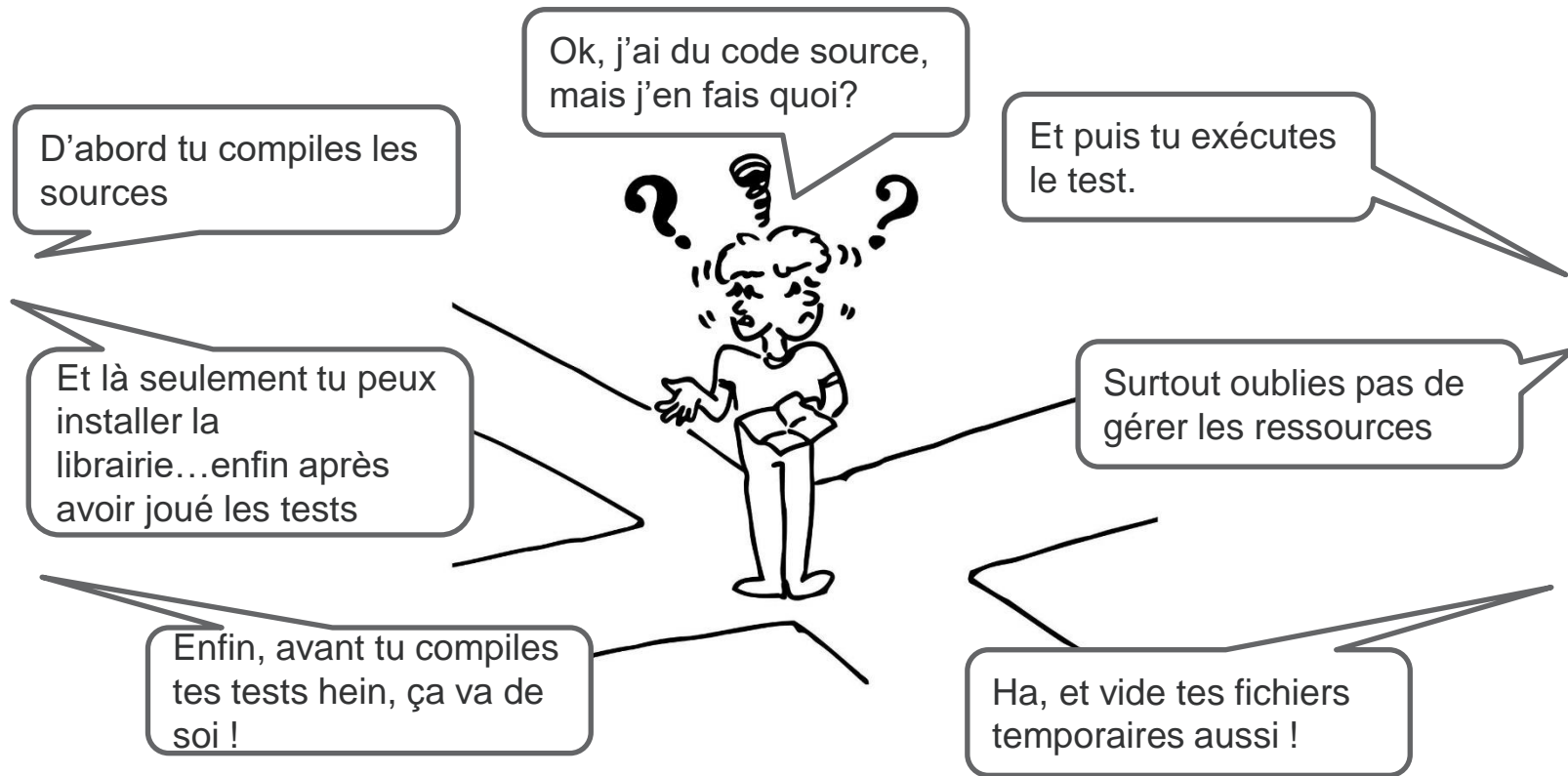
- La dernière version est la 3.6.2
- Maven 5 en cours de développement



Hervé BOUTEMY

Apache Member, Apache Maven  
Committer and PMC member

# Maven : Qu'est-ce que c'est?



A vertical line on the left side of the slide, featuring a solid green circle at the top and four white circles with dark gray outlines below it.

# Maven : Qu'est-ce que c'est?

Problématiques de **build** :

- Comment transformer mes \*.java en \*.class, puis en \*.jar/\*.war/\*.ear ?
- Dans quel ordre je compile mes sous projets ?
- Quel compilateur ?
- Quel package va dans quel sous-dossier ?

Problématiques d'**uniformisation** :

- Build complètement différents d'un projet à l'autre
- Emplacements des fichiers différents
- 3pp : third party product

Problématiques de **dépendances** :

- De où provient ma dépendance ?
- Quelle version utiliser ?
- J'ai besoin de ma dépendance pour faire les tests, mais pas pour la prod...





# Maven : Qu'est-ce que c'est?

Définition :

*Maven est un **outil de management de projet** qui va chercher à **produire un logiciel** à partir de ses sources, en **optimisant** les tâches (...) et en garantissant le bon **ordre** de fabrication.*

*Il apporte également un lot de **conventions** permettant de mettre un place (très) **rapidement** un **cycle de vie** par défaut*

A vertical line on the left side of the slide, featuring a series of circles. The top circle is solid green, while the others are white with a dark outline. The line itself is dark gray.

## Maven : Qu'est-ce que c'est?

Permet de :

- Créer des builds customisables
- Gérer l'utilisation des dépendances vers des librairies externes
- D'accéder facilement à un large (et évolutif) répertoire de librairies
- D'établir une convention uniforme
- Faire du templating de code archetype
- Faire du source code/release management
- Faire de la gestion documentaire (distribution, mailing lists ...)
- ...

En bref...ça simplifie la vie du développeur

A vertical line on the left side of the slide, featuring a solid green circle at the top and five white circles with dark gray outlines below it.

# Maven : Qu'est-ce que c'est?

Convention over configuration

*“Tant que l'on suit la convention, pas besoin de préciser quoi que ce soit”*

Item	Default
Source code	<code>\${basedir}/src/main/java</code>
Resources	<code>\${basedir}/src/main/resources</code>
Tests	<code>\${basedir}/src/test</code>
Compiled Code	<code>\${basedir}/target</code>
Class Files	<code>\${basedir}/target/classes</code>

# Maven : Qu'est-ce que c'est?

Et concrètement ? Qu'est-ce que c'est ?



# Partie 2 :

# Configuration

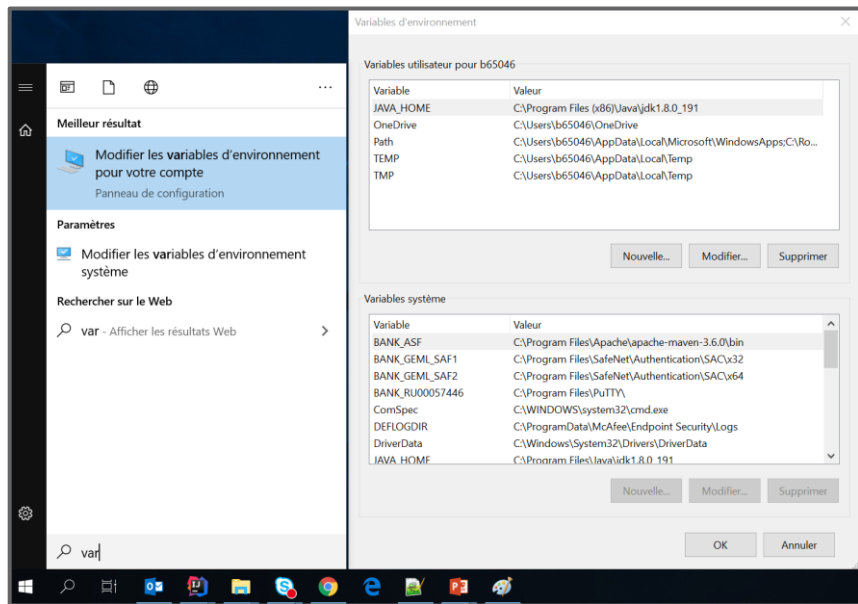
# du poste de travail



# Configuration du poste de travail

Prérequis pour installer un Maven :

- Java 7+
- Une variable d'environnement « JAVA\_HOME » pointant sur votre dossier Java





## Configuration du poste de travail

Installer Maven (chez vous) :

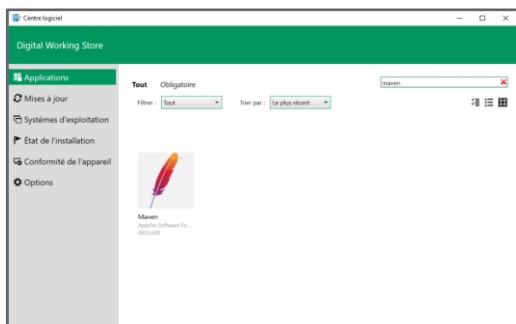
- Télécharger Maven [depuis le site officiel](#)
- Vérifier ou faire en sorte que la variable d'environnement « Path » contienne le chemin vers votre répertoire Maven
- Vérifier que votre Maven fonctionne en entrant la commande suivante :

```
>mvn --version
```

# Configuration du poste de travail

Installer Maven sur un poste BNPP :

- Télécharger Maven depuis le Digital Working Store
- Vérifier ou faire en sorte que la variable d'environnement « Path » contienne le chemin vers votre répertoire Maven
- Vérifier que votre Maven fonctionne...



```
>mvn --version
```





# Configuration du poste de travail

Configurer Maven sur un poste BNPP :

La configuration Maven du poste de travail se trouve dans le fichier :

`${user.home}/.m2/settings.xml`

1. Par défaut, Maven utilise votre profil windows pour stocker les librairies qu'il utilise. Chez BNPP, cet espace est limité. Il faut donc modifier ce répertoire

```
<localRepository>C:\mavenRepo</localRepository>
```

2. BNPP utilise un proxy avec authentification. C'est dans ce fichier que l'on configure ce dernier :

```
<proxies>
  <proxy>
    <id>ncproxy</id>
    <active>true</active>
    <protocol>http</protocol>
    <host>ncproxy</host>
    <port>8080</port>
    <username>VOTRE_UID</username>
    <password>${PASSWORD_JAMEL}</password>
  </proxy>
</proxies>
```

# Configuration du poste de travail

Configurer Maven sur un poste BNPP (suite) :

## 3. Configurer les différents repos avec leurs accès :

```
<servers>
  <server>
    <id>nexus</id>
    <username>VOTRE_UID</username>
    <password>VOTRE_PASSWORD</password>
  </server>
</servers>
```

```
<activeProfiles>
  <activeProfile>ProfilNexus</activeProfile>
</activeProfiles>
```

```
<profiles>
  <profile>
    <id>ProfilNexus</id>
    <repositories>
      <repository>
        <id>nexus</id>
        <url>URL_DE_VOTRE_NEXUS</url>
      </repository>
    </repositories>

    <pluginRepositories>
      <pluginRepository>
        <id>nexus</id>
        <url>URL_DE_VOTRE_NEXUS</url>
      </pluginRepository>
    </pluginRepositories>
  </profile>
</profiles>
```



# Configuration du poste de travail

Configurer Maven sur un poste BNPP (suite/facultatif) :

4. Pour éviter d'avoir votre mot de passe en clair :

a. Créez un master-password

```
>mvn --encrypt-master-password <superpwd>
```

b. Copiez le contenu du résultat dans un fichier  
`${user.home}/.m2/settings-security.xml`

```
<settingsSecurity>  
  <master>{jSM0WnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}</master>  
</settingsSecurity>
```

c. Encryptez votre mot de passe

```
>mvn --encrypt-password <password>
```

d. Utilisez le résultat comme mot de passe dans vos fichiers de configuration !

```
<password>{jSM0WnoPFgsHVpMvz5VrIt5kRbzGpI8u+9EF1iFQyJQ=}</password>
```

# Configuration du poste de travail

Configurer Maven sur un poste BNPP (suite/facultatif) :

5. Lors de vos premiers téléchargements, vous allez certainement rencontrer le message d'erreur suivant :

```
sun.security.validator.ValidatorException: PKIX path building failed:  
sun.security.provider.certpath.SunCertPathBuilderException: unable to find valid  
certification path to requested target -> [Help 1]
```

Pour palier à ce problème, il existe 2 solutions :



certificats\_java.zip

- A. Mettre les certificats Java votre dossier \$JAVA\_HOME/jre/lib/security
- B. Ajouter une variable d'environnement MAVEN\_OPTS ayant la valeur suivante :

```
-Dmaven.wagon.http.ssl.insecure=true  
-Dmaven.wagon.http.ssl.allowall=true  
-Dmaven.wagon.http.ssl.ignore.validity.dates=true
```



# Configuration du poste de travail

**Exercice 1** : Le but est de créer un projet depuis un squelette d'application Maven :

- Ouvrir une invite de commande, se placer dans un dossier créé pour l'exercice et taper `>mvn archetype:generate`
- Maven vous affiche la liste des squelettes disponibles
- Entrer `maven-archetype-quickstart`
- Choisir l'archetype `org.apache.maven.archetypes:maven-archetype-quickstart`

```
Choose archetype:
1: remote -> com.haoxuer.maven.archetype:maven-archetype-quickstart (a simple maven archetype)
2: remote -> org.apache.maven.archetypes:maven-archetype-quickstart (An archetype which contains a sample Maven project.)
Choose a number or apply filter (format: [groupId:]artifactId, case sensitive contains): 2:
```

- Remplir les champs demandés
- Importer le projet dans votre IDE (fichier → ouvrir)



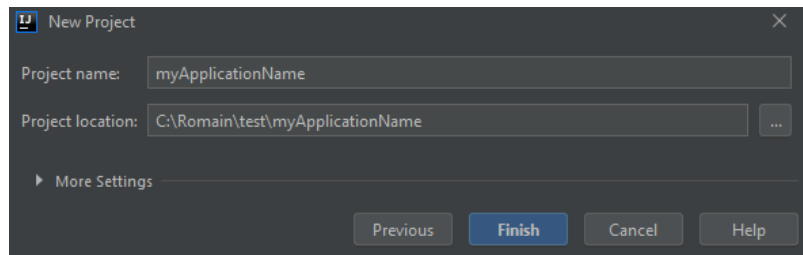
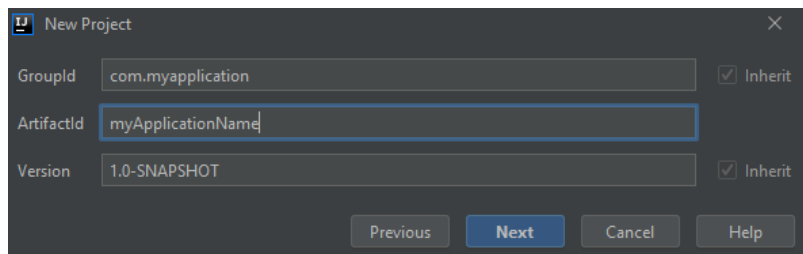
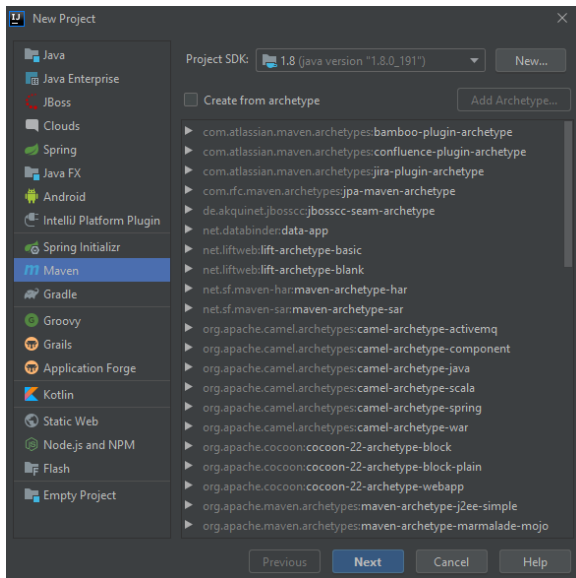
# Configuration du poste de travail

Exercice 2 : Le but est de créer un projet Maven vierge depuis IntelliJ :

- Faire NEW → Project
- Sélectionner « Maven » dans la colonne de gauche

*NB : Remarquez que vous pouvez choisir de créer un projet à partir d'un archétype*

- Suivre le wizard de création en remplissant les champs demandés



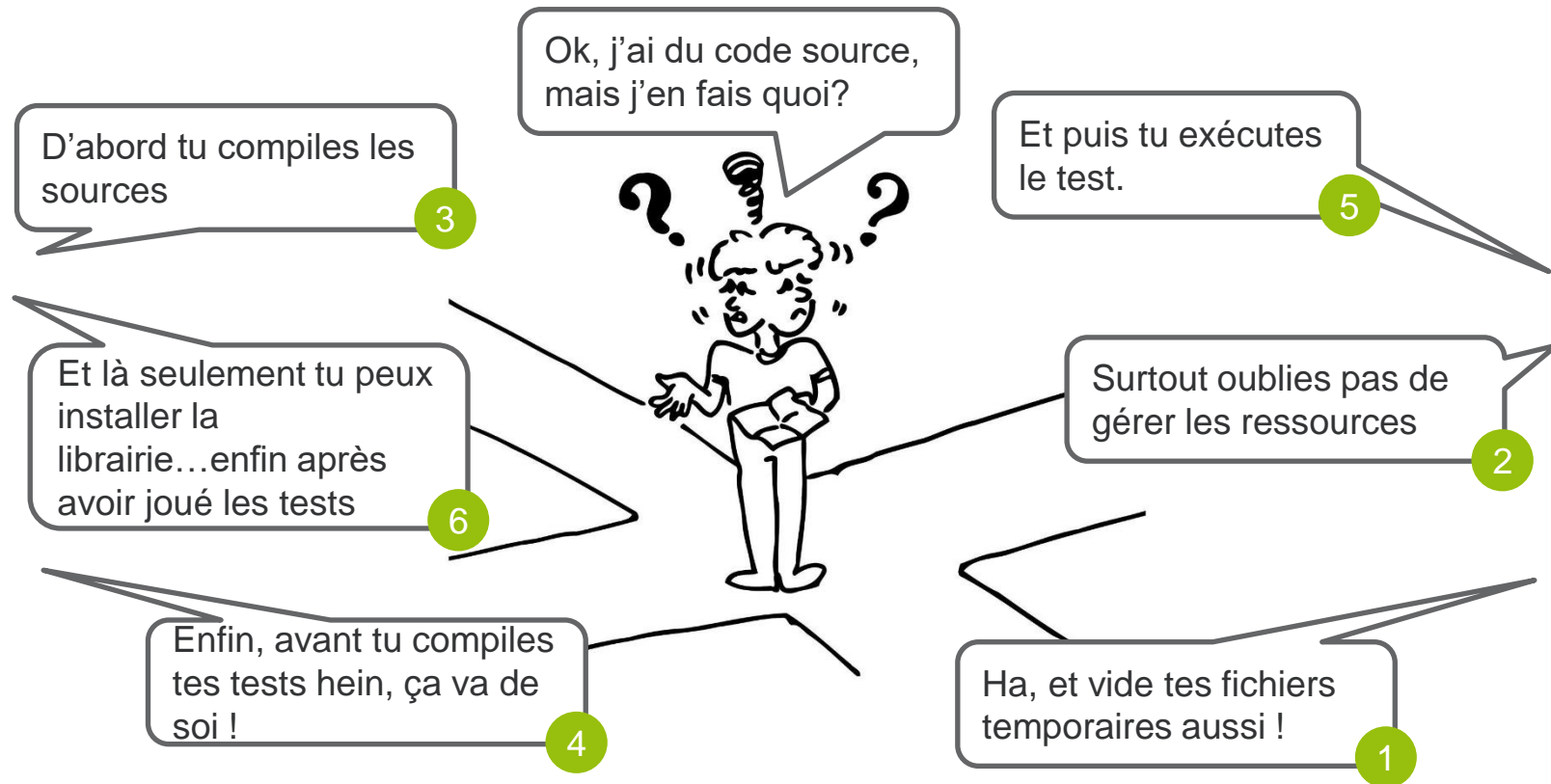
# Partie 3 :

# Plugins & Lifecycle\*



\* Plugiciels et cycle de vie

# Plugins & Lifecycle

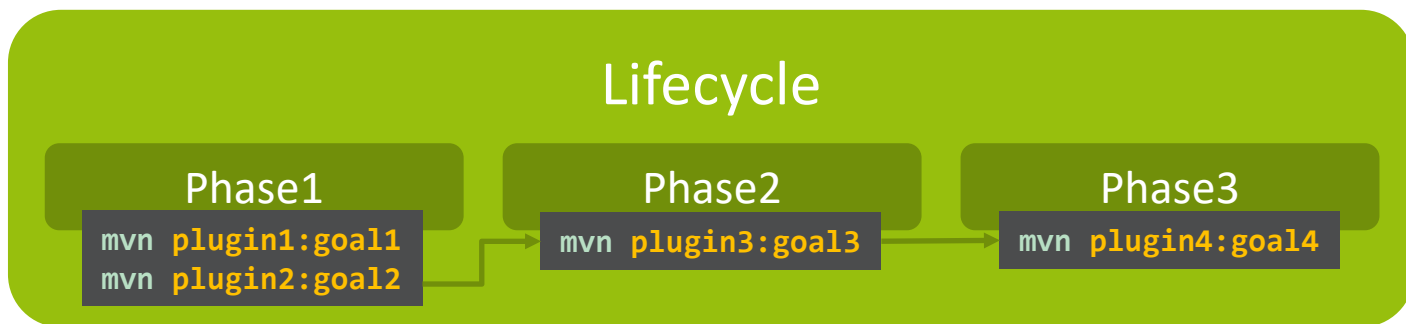




# Plugins & Lifecycle

Un **lifecycle** (cycle de vie) est un enchainement de **phases** dans un ordre bien précis.

Une **phase** peut contenir zéro à plusieurs **plugin goals** (une phase sans plugin n'est pas exécutée)



*NB : Il n'est pas possible de créer ses propres phases/lifecycles*



## Plugins & Lifecycle

Une « action » en Maven est appelé un **goal**

Un **goal** fait partie d'un groupe appelé **plugin**

Pour exécuter un goal, on utilise la notation suivante :

```
>mvn plugin:goal
```

```
>mvn archetype:generate
```

```
>mvn help:effective-pom
```

```
>mvn dependency:tree
```

*Quelques exemples*

# Plugins & Lifecycle

Il existe 30 phases réparties sur 3 lifecycles différents :

Clean (*suppression des fichiers temporaires*)

pre-clean	
clean	clean:clean
post-clean	

Site (*documentation*)

pre-site	
site	site:site
post-site	
site-deploy	site:deploy

Default (*création d'une archive*)

validate	
initialize	
generate-sources	
process-sources	
generate-resources	
process-resources	resources:resources
compile	compiler:compile
process-classes	
generate-test-sources	
process-test-sources	resources:testResources
generate-test-resources	
process-test-resources	
test-compile	compiler:testCompile
process-test-classes	
test	surefire:test
prepare-package	
package	war:war/jar:jar/rar:rar
pre-integration-test	
integration-test	
post-integration-test	
verify	
install	install:install
deploy	deploy:deploy

# Plugins & Lifecycle

Mais pourquoi y'a des phases sans goal ?



Comme ça, tu veux mettre un goal...et ben **PAF !**  
T'as déjà une phase !

# Plugins & Lifecycle

L'exécution d'une phase se fait directement par son nom :

```
>mvn phase
```

WAR/JAR Lifecycle
process-resources
compile
process-test-resources
test-compile
test
package
Install
deploy



Attention ! L'appel d'une phase déclenche le lifecycle auquel il est attaché depuis sa première phase jusqu'à la phase demand



*NB : Il n'est pas possible d'appeler une phase de manière unitaire*

# Plugins & Lifecycle

process-resources	resources:resources	Traite les ressources, remplace les variables si nécessaire, et place les fichiers résultants à la racine du dossier <i>target/classes</i>
compile	compiler:compile	Compile les sources, et place les classes dans le dossier <i>target</i>
process-test-sources	resources:testResources	Traite les ressources des tests, et les place dans le dossier <i>target/test-classes</i>
test-compile	compiler:testCompile	Compile le code des tests, et le place dans le dossier <i>target/test-classes</i>
test	surefire:test	Exécute les tests présents dans les classes se terminant par <i>*Test.java</i> ou <i>*IT.java</i> Place les rapports de test dans le dossier <i>target/surefire-report</i>
package	war:war/jar:jar/rar:rar	Crée une archive (jar/war/ear) et la place à la racine du dossier <i>target</i>
install	install:install	Installe la librairie dans le repo local
deploy	deploy:deploy	Installe la librairie sur le repo distant
clean	clean:clean	Supprime le dossier <i>target</i>



## Plugins & Lifecycle

Il est possible d'appeler plusieurs cycles en une même commande

```
>mvn clean package
```

Maven va alors les exécuter dans l'ordre indiqué



# Plugins & Lifecycle

Question 1 : Que fait la commande

```
>mvn deploy
```

Question 2 : La notation suivante est-elle correcte ?

```
>mvn clean dependency:copy-dependencies package
```

Question 3 : Que réalise la commande suivante ?

```
>mvn package clean
```

Question 4 : Peut-on la simplifier?





# Plugins & Lifecycle

Question 1 : Que fait la commande

```
>mvn deploy
```

Elle exécute toutes les phases jusque deploy. La phase deploy déploie l'archive sur le repo distant

Question 2 : La notation suivante est-elle correcte ?

```
>mvn clean dependency:copy-dependencies package
```

Oui, on peut très bien mixer des phases et des goals

Question 3 : Que réalise la commande suivante ?

```
>mvn package clean
```

Tout jusque package (compile, test etc...), puis elle va supprimer le dossier target...donc elle fait plein de choses mais au final, on aura rien

Question 4 : Peut-on la simplifier?

Non, même si au final on a rien, elle passe exécute des choses quand même

Partie 4 :

`pom.xml` :

Le strict nécessaire



## **pom.xml : Le strict nécessaire**

Maîtriser le POM, c'est maîtriser Maven





## pom.xml : Le strict nécessaire

**P**roject **O**bject **M**odel (*eXtensible Markup Language*)

C'est un fichier qui contient la totalité des informations utiles au build d'un projet

NB : Il peut être verbeux. Il faudra de la discipline dans l'indentation de votre POM (Aidez-vous des IDE pour l'auto-complétion !)

A vertical line on the left side of the slide, featuring a series of circles. The fourth circle from the top is filled with orange, while the others are white with grey outlines.

## pom.xml : Le strict nécessaire

Un pom se présente de la manière suivante :

```
<?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>

  <!-- Votre configuration ici -->

</project>
```

# pom.xml : Le strict nécessaire

Il peut contenir de nombreuses informations :

Descriptif	Dépendances	Variables	Chargement conditionnel	Documentation	Référentiels	Infos CI/CD
groupId <b>1</b> artifactId version packaging  parent <b>4</b>  modules <b>1</b>	dependencies dependency- Management <b>3</b>	properties <b>5</b>	profiles <b>2</b>	name <b>2</b> description url inceptionYear organization licenses developers contributors	repositories pluginRepositories	scm issueManagement ciManagement mailingLists reporting prerequisites  build <b>3</b>



## pom.xml : Le strict nécessaire (identifiants)

- **groupId** : Nom du projet/de l'application.  
En général, il définit le package de base de vos classes
- **artifactId** : Nom de votre module
- **version** : Numéro de version  
- *SNAPSHOT* signifie « en cours de développement »  
On peut y mettre autre chose que des numéros
- **packaging** : type de l'application (jar/war/ear/pom etc...)

« jar » par défaut

```
<groupId>[groupId]</groupId>
<artifactId>[artifactID]</artifactId>
<version>[version]</version>
<packaging>jar</packaging>
```

A vertical line on the left side of the slide, featuring a series of circles. The fifth circle from the top is filled with orange, while the others are white with black outlines.

## pom.xml : Le strict nécessaire (documentation)

Certaines informations ne sont pas nécessaires au build. Elles sont uniquement présentes à titre documentaire. Elles sont facultatives.

- **name** : Nom (fonctionnel) du projet
- **description** : Descriptif du projet
- **url** : URL du projet (s'il en possède une)
- **inceptionYear** : Année de création du projet
- **organization** : Nom de
- **licenses** : Type de licence
- **developers** : Nom des développeurs du projet
- **contributors** : Nom des contributeurs du projet

Bien que facultatives, ces informations peuvent être utilisées par certains plugins pour générer des fichiers de licences.





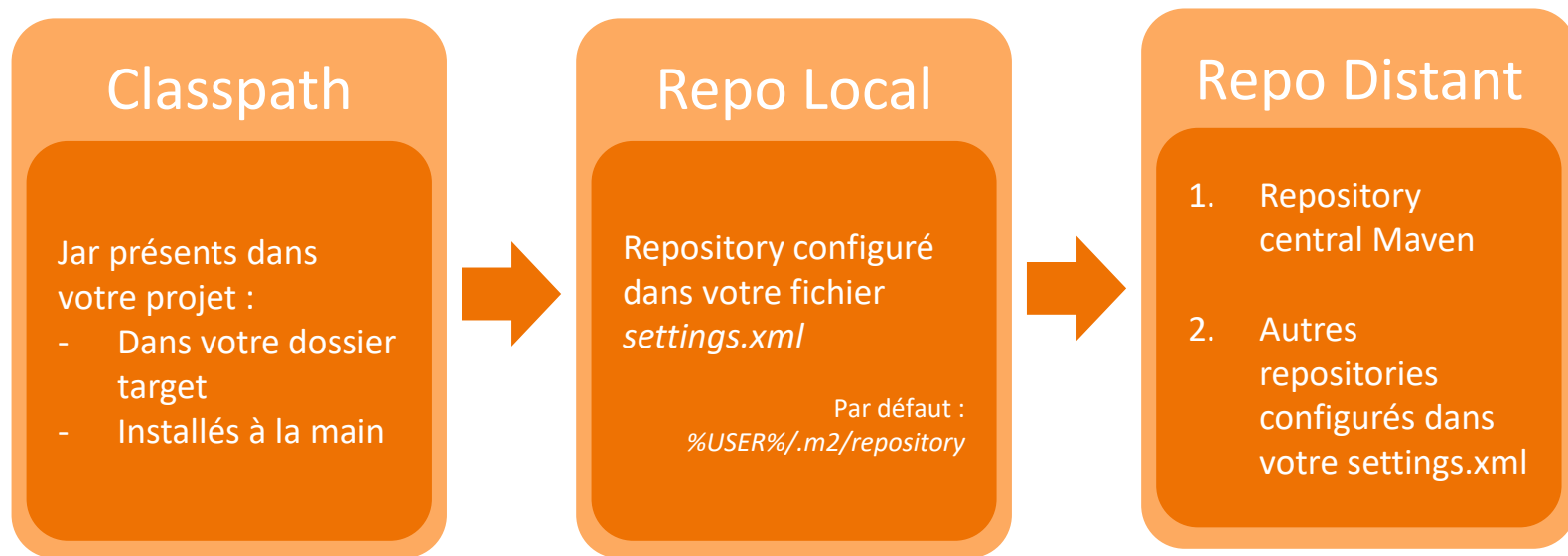
## pom.xml : Le strict nécessaire (dépendances)

Afin d'utiliser du code externe sans avoir à le dupliquer, on utilise les dépendances :

```
<dependencies>
  <dependency>
    <groupId>[groupId]</groupId>
    <artifactId>[artifactId]</artifactId>
    <version>[version]</version>
    <scope>[scope]</scope>
  </dependency>
  (...)
</dependencies>
```

# **pom.xml : Le strict nécessaire (dépendances)**

Maven va alors rechercher la librairie dans 3 zones différentes :



## pom.xml : Le strict nécessaire



**Exercice 1 :** Imaginer un algorithme qui renvoie *true* si tous les caractères sont en majuscule ou si tous les caractères sont en minuscule. *false* sinon

La librairie commons lang3 met à disposition les méthodes :

- `StringUtils.isAllLowerCase(String str)`
- `StringUtils.isAllUpperCase(String str)`

**Exercice 2 :** Ajouter la librairie apache commons lang3

<https://mvnrepository.com/>

**Exercice 3 :** Ecrire une méthode qui exécute cet algorithme

# pom.xml : Le strict nécessaire



**Exercice 1 :** Imaginer un algorithme qui renvoie *true* si tous les caractères sont en majuscule ou si tous les caractères sont en minuscule. *false* sinon

On fait une boucle et on regarde chaque caractère etc... (compliqué)

La librairie commons lang3 met à disposition les méthodes :

- `StringUtils.isAllLowerCase(String str)`
- `StringUtils.isAllUpperCase(String str)`

**Exercice 2 :** Ajouter la librairie apache commons lang3

```
<dependency>
  <groupId>org.apache.commons</groupId>
  <artifactId>commons-lang3</artifactId>
  <version>3.9</version>
</dependency>
```

**Exercice 3 :** Ecrire une méthode qui exécute cet algorithme  
`StringUtils.isAllLowerCase(text) || StringUtils.isAllUpperCase(text)`

## **pom.xml : Le strict nécessaire (dépendances)**

Il se passe quoi si une librairie utilise d'autres librairies ?



# pom.xml : Le strict nécessaire (dépendances)

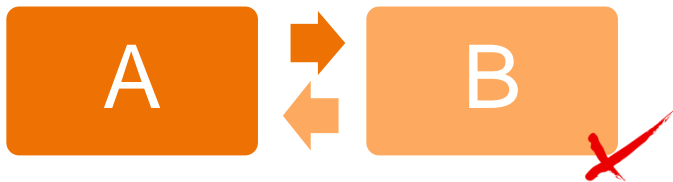
Depuis Maven 2.0, les dépendances sont transitives par défaut



Il n'y a pas de limites au nombre de niveaux

*NB : Il est possible de couper la transitivité grâce au tag `<optional>true</optional>`*

Attention aux dépendances cycliques !



# pom.xml : Le strict nécessaire (dépendances)

One command to rule them all !

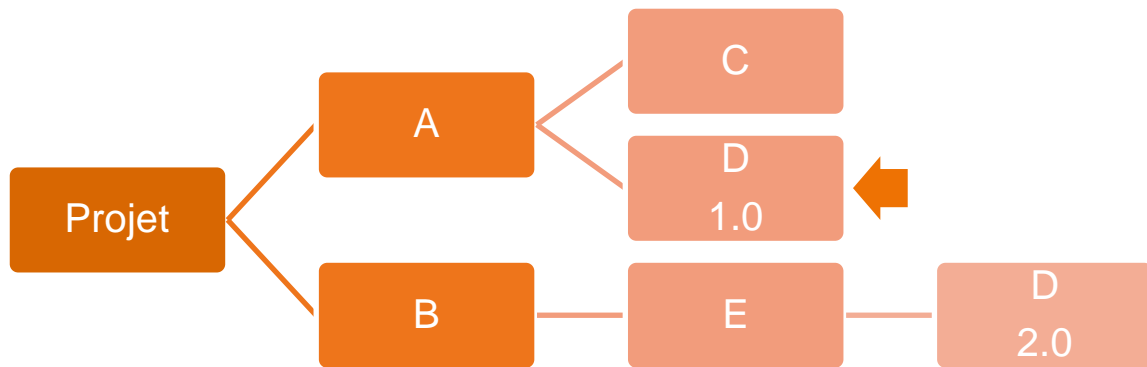
```
>mvn dependency:tree
```

```
[INFO] \- org.springframework.boot:spring-boot-starter:jar:2.0.0.RELEASE:compile
[INFO]   +- org.springframework.boot:spring-boot:jar:2.0.0.RELEASE:compile
[INFO]   | \- org.springframework:spring-context:jar:5.0.4.RELEASE:compile
[INFO]   |   +- org.springframework:spring-aop:jar:5.0.4.RELEASE:compile
[INFO]   |   +- org.springframework:spring-beans:jar:5.0.4.RELEASE:compile
[INFO]   |   \- org.springframework:spring-expression:jar:5.0.4.RELEASE:compile
[INFO] +- org.springframework.boot:spring-boot-autoconfigure:jar:2.0.0.RELEASE:compile
[INFO] +- org.springframework.boot:spring-boot-starter-logging:jar:2.0.0.RELEASE:compile
[INFO]   | +- ch.qos.logback:logback-classic:jar:1.2.3:compile
[INFO]   | | +- ch.qos.logback:logback-core:jar:1.2.3:compile
[INFO]   | | \- org.slf4j:slf4j-api:jar:1.7.25:compile
[INFO]   | +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.10.0:compile
[INFO]   | | \- org.apache.logging.log4j:log4j-api:jar:2.10.0:compile
[INFO]   | \- org.slf4j:jul-to-slf4j:jar:1.7.25:compile
[INFO] +- javax.annotation:javax.annotation-api:jar:1.3.2:compile
[INFO] +- org.springframework:spring-core:jar:5.0.4.RELEASE:compile
[INFO]   | \- org.springframework:spring-jcl:jar:5.0.4.RELEASE:compile
[INFO] \- org.yaml:snakeyaml:jar:1.19:runtime
```



## **pom.xml : Le strict nécessaire (dépendances)**

Si deux librairies utilisent (transitivement) des versions différentes d'une librairie, Maven prend la version la plus proche dans l'arbre de dépendance



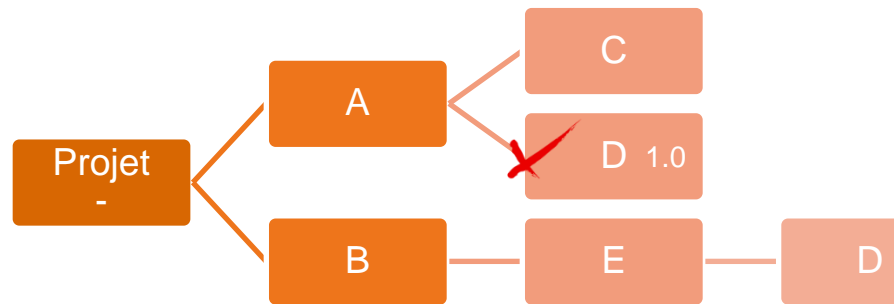
Si la distance est la même, Maven prend en compte l'ordre des déclarations



## pom.xml : Le strict nécessaire (dépendances)

Dans le cas où l'on ne veut pas importer une dépendance par transitivité, il faut le spécifier grâce au tag `<exclusions>`

```
<dependencies>
  <dependency>
    <groupId>com.groupid</groupId>
    <artifactId>A</artifactId>
    <version>1.0-SNAPSHOT</version>
    <exclusions>
      <exclusion>
        <groupId>com.groupid</groupId>
        <artifactId>D</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
</dependencies>
```





## pom.xml : Le strict nécessaire (dépendances)

Il est également possible de forcer Maven à utiliser une version spécifique grâce au « dependency management ».

- Si un version est précisée, elle sera écrasée
- Permet de ne plus avoir à spécifier la version de la librairie à utiliser

```
<dependencyManagement>  
  <dependencies>  
    <dependency>  
      <groupId>org.slf4j</groupId>  
      <artifactId>slf4j-api</artifactId>  
      <version>1.8.0-beta2</version>  
    </dependency>  
  </dependencies>  
</dependencyManagement>
```

# pom.xml : Le strict nécessaire (dépendances)

## Scopes de dépendance :

- **Compile**
  - Scope par défaut
  - Rend disponible la librairie sur la totalité du classpath
  - Permet la transitivité
- **Test**
  - Rend disponible la librairie uniquement en classpath de test
  - Ne permet pas la transitivité
- **Provided**
  - Indique que le conteneur cible (JDK/Serveur d'appli...) contient la librairie
  - Rend disponible la librairie sur la totalité du classpath, mais ne sera pas embarquée dans l'archive
  - Ne permet pas la transitivité
- **Runtime**
  - Nécessaire au runtime mais pas à la compilation (code chargé dynamiquement, JDBC drivers etc...)
  - Rend disponible la librairie sur la totalité du classpath
  - Permet la transitivité
- **System**
  - Spécifie l'emplacement de la librairie sur le file system avec une propriété <systemPath>
  - Mêmes propriétés que le scope compile
- **Import**
  - Uniquement pour le <dependencyManagement>, importe le dependencyManagement d'un autre pom

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

# pom.xml : Le strict nécessaire (dépendances)



## Exercice 1 :

- Importer la dépendance  
org.springframework.boot:spring-boot-starter-web:2.1.0.RELEASE:compile
- Regarder son arbre de dépendance
- Quels sont les scope utilisés ? Pourquoi ?

## Exercice 2 : Puis-je ajouter les dépendances suivantes sans danger ?

- org.apache.commons:commons-lang3:jar:3.9:compile
- org.yaml:snakeyaml:jar:1.25:test
- org.springframework.cloud:spring-cloud-starter-zuul:jar:1.4.7.RELEASE:compile

## Exercice 3 : Même question mais en les ajoutant en dependency management

## Exercice 4 : J'ai besoin d'utiliser la librairie Guava, est-ce cohérent d'écrire ceci ?

```
<dependency>  
  <groupId>com.google.guava</groupId>  
  <artifactId>guava</artifactId>  
</dependency>
```

# pom.xml : Le strict nécessaire (dépendances)



## Exercice 1 :

- Importer la dépendance  
org.springframework.boot:spring-boot-starter-web:2.1.0.RELEASE:compile
- Regarder son arbre de dépendance
- Quels sont les scope utilisés ? Pourquoi ?

```
>mvn dependency:tree
```

compile & runtime ce sont les seuls scopes transitifs

## Exercice 2 : Puis-je ajouter les dépendances suivantes sans danger ?

- org.apache.commons:commons-lang3:jar:3.9:compile **Oui**
- org.yaml:snakeyaml:jar:1.25:test **Non, peut provoquer un conflit de version**
- org.springframework.cloud:spring-cloud-starter-zuul:jar:1.4.7.RELEASE:compile

**Non, peut provoquer un conflit de version de dépendances tirés par zuul**

## Exercice 3 : Même question mais en les ajoutant en dependency management

Mêmes réponses

## Exercice 4 : J'ai besoin d'utiliser la librairie Guava, est-ce cohérent d'écrire ceci ?

```
<dependency>  
  <groupId>com.google.guava</groupId>  
  <artifactId>guava</artifactId>  
</dependency>
```

Il manque la version, c'est cohérent uniquement si j'ai un dependency management qui m'indique la version

# pom.xml : Le strict nécessaire (héritage)

Comme en Java, Maven implémente un mécanisme d'héritage :

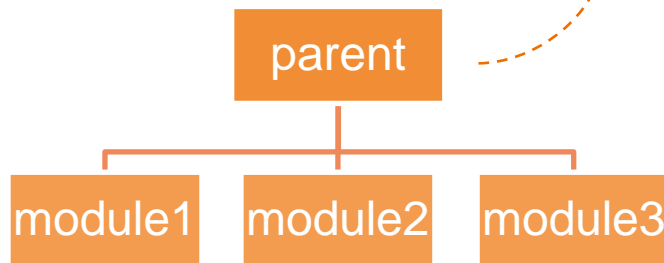
- Un enfant ne peut avoir qu'un seul parent
- Un parent peut avoir plusieurs enfants

Les enfants référencent leur parent de la manière suivante :

```
<parent>
  <groupId>groupIdParent</groupId>
  <artifactId>artifactIdParent</artifactId>
  <version>versionParent</version>
</parent>
```

Attention ! Pour pouvoir être hérité, le type de package doit être défini en « pom »

```
<packaging>pom</packaging>
```



A vertical line on the left side of the slide with six circles. The fourth circle from the top is filled with orange, while the others are white with grey outlines.

## **pom.xml : Le strict nécessaire (héritage)**

L'héritage permet de mutualiser les propriétés et donc de simplifier les POM

Sauf surcharge, le POM enfant va hériter de la quasi-totalité des propriétés de son père

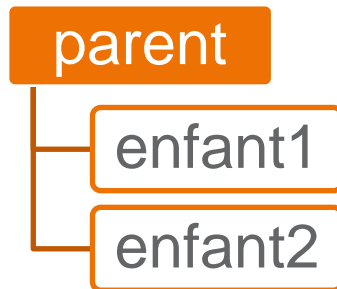
Ne sont pas hérités :

- artifactId
- name
- packaging
- modules
- prerequisites

# **pom.xml : Le strict nécessaire (héritage)**



**Exercice 1** : Créer 3 projets selon l'exemple suivant :



**Exercice 2** : Faire en sorte que les enfants aient le même numéro de version que leur parent

**Exercice 3** : Le parent peut-il contenir des classes Java ?

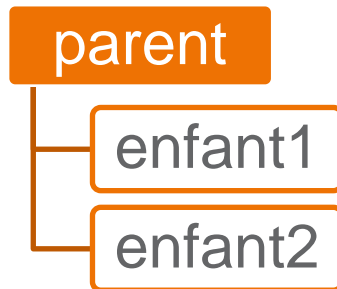
**Exercice 4** : enfant2 peut-il dépendre de enfant1 ?



# pom.xml : Le strict nécessaire (héritage)



Exercice 1 : Créer 3 projets selon l'exemple suivant :



Exercice 2 : Faire en sorte que les enfants aient le même numéro de version que leur parent

C'est déjà le cas, les enfants héritent de la version du parent

Exercice 3 : Le parent peut-il contenir des classes Java ?

Syntaxiquement oui, mais ça ne sert à rien !

Exercice 4 : enfant2 peut-il dépendre de enfant1 ?

Oui, et ça sera même souvent le cas (il faut juste veiller à l'ordre dans lequel ils vont être buildés)

# pom.xml : Le strict nécessaire (héritage)



**Exercice 5** : Dans le but d'accélérer les développements et d'avoir un framework facile d'accès, Spring propose un pom-parent :

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.5.RELEASE</version>
</parent>
```

Pourquoi? Et d'après-vous, quelle est la limite de cette solution ?

# pom.xml : Le strict nécessaire (héritage)



**Exercice 5** : Dans le but d'accélérer les développements et d'avoir un framework facile d'accès, Spring propose un pom-parent :

```
<parent>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-parent</artifactId>
  <version>2.1.5.RELEASE</version>
</parent>
```

Pourquoi? Et d'après-vous, quelle est la limite de cette solution ?

- Cela permet d'hériter de beaucoup de choses rapidement ! (plugins, dependency management etc...)
- Si votre entreprise vous force à utiliser un pom parent « corporate », vous ne pourrez pas utiliser le pom parent de Spring

# pom.xml : Le strict nécessaire (héritage)



**Exercice 6** : Dans le cas où il ne serait pas possible d'utiliser le parent, Spring propose d'ajouter ceci à son POM :

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>2.1.1.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Pourquoi ? Quelles sont les limites de cette solution ?

# pom.xml : Le strict nécessaire (héritage)



**Exercice 6** : Dans le cas où il ne serait pas possible d'utiliser le parent, Spring propose d'ajouter ceci à son POM :

```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-dependencies</artifactId>
      <version>2.1.1.RELEASE</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

Pourquoi ? Quelles sont les limites de cette solution ?

- Cela permet d'importer le dependency management de Spring
- Cette solution couvre **UNIQUEMENT** le dependency management (et pas les plugins par exemple)



# pom.xml : Le strict nécessaire (properties)

Les **properties** sont les équivalents des variables pour Maven

- On cherche à mutualiser une valeur
- On cherche à utiliser une valeur que l'on ne connaît pas explicitement
- On cherche à utiliser une variable dans une ressource

On les résout grâce à des **placeholders**

```
${nom.de.la.variable}
```

On peut les déclarer dans le pom  
(côté dev)

```
<properties>  
<log4j.version>1.8.0</log4j.version>  
<temp>${project.build.directory}/temp/</temp>  
(...)  
</properties>
```

On peut aussi les faire passer  
en ligne de commande  
(côté ops/pipeline)

```
>mvn install -Dlog4j.version=1.8.0  
-Dtemp=${project.build.directory}/temp/
```



# pom.xml : Le strict nécessaire (properties)

On peut utiliser des variables déjà existantes :

- Les variables d'environnement
  - `${env.PATH}` pour récupérer la variable `%PATH%`
- Les variables projets, préfixées par « `project.*` »
  - `${project.organization.name}` pour récupérer `<organization><name>1.0</name></organization>`
- Les variables déclarées dans un fichier `settings.xml`, préfixées par « `settings.*` »
  - `${settings.offline}` pour récupérer `<settings><offline>>false</offline></settings>`
- Les variables Java, préfixées par « `java.*` »
  - `${java.home}`
  - Permet de récupérer les variables accessibles via `java.lang.System.getProperties()`
- Les variables Maven
  - `${maven.build.timestamp}`

*NB : On peut surcharger les variables existantes*

# **pom.xml : Le strict nécessaire (properties)**

Il est possible d'injecter des propriétés dans les ressources :

1. Placer un placeholder dans une ressource
2. Déclarer le dossier de ressources concerné dans la balise build.resources du POM
3. Activer le filtering

*NB : L'injection se fait lors de la phase process-resource (resources:resources)*

Ne filtrez pas de ressources binaires !

```
<build>
  <resources>
    <resource>
      <directory>src/main/resources</directory>
      <filtering>true</filtering>
    </resource>
  </resources>
  <filters>
    <filter>fichier-cle-valeur.properties</filter>
  </filters>
  (...)
</build>
```

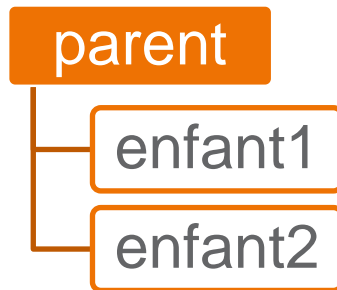
On peut également ajouter un fichier spécifique clef/valeur grâce au tag build.filters.filter



# **pom.xml : Le strict nécessaire (properties)**



Reprenons l'exemple précédent :



**Exercice 1** : Sachant que les enfants ont le même numéro de version que le parent, faire en sorte que enfant1 dépende de enfant2

**Exercice 2** : Importer la librairie org.yaml:snakeyaml:jar:1.25:compile et variabiliser son numéro de version

# pom.xml : Le strict nécessaire (properties)



**Exercice 1** : Sachant que les enfants ont le même numéro de version que le parent, faire en sorte que enfant1 dépende de enfant2

```
<dependency>
  <groupId>com.mygroupid</groupId>
  <artifactId>enfant2</artifactId>
  <version>${project.parent.version}</version>
</dependency>
```

**Exercice 2** : Importer la librairie org.yaml:snakeyaml:jar:1.25:compile et variabiliser son numéro de version

```
<dependency>
  <groupId>org.yaml</groupId>
  <artifactId>snakeyaml</artifactId>
  <version>${snakeyaml.version}</version>
</dependency>
```

```
<properties>
  <snakeyaml.version>1.25</snakeyaml.version>
</properties>
```

## **pom.xml : Le strict nécessaire (properties)**



**Exercice 4** : Dans enfant1, créer un fichier nomDuProjet.txt, dans lequel on va chercher à injecter une variable

- Ecrire le `${placeholder}` associé à la variable dans le fichier
- Adapter le POM pour prendre en compte le remplacement de placeholders dans les ressources
- Exécuter le goal adéquat

**Exercice 5** : Effectuer le même exercice avec le fichier heureDuCrime.txt dans lequel on va chercher à afficher l'heure du build

- Encapsuler la variable `maven.build.timestamp` dans une autre variable de votre POM
- Afficher cette nouvelle variable

# pom.xml : Le strict nécessaire (properties)



**Exercice 4** : Dans enfant1, créer un fichier nomDuProjet.txt, dans lequel on va chercher à injecter une variable

- Ecrire le `${placeholder}` associé à la variable dans le fichier
- Adapter le POM pour prendre en compte le remplacement de placeholders dans les ressources
- Exécuter le goal adéquat

```
<properties>
  <mavariab>Ma super variable</mavariab>
</properties>

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

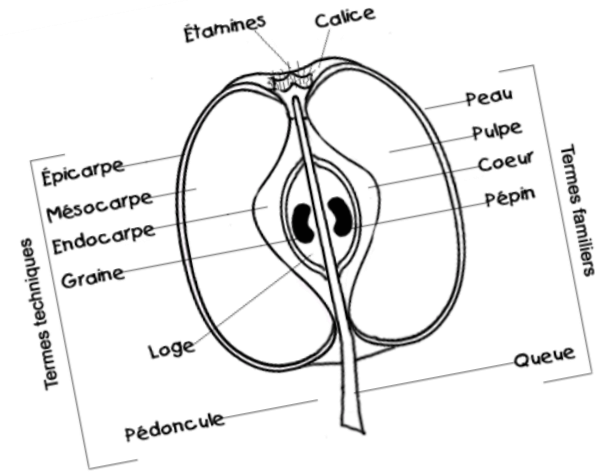
nomDuProjet.txt

```
${mavariab}
```

```
>mvn resources:resources
(cf slide 29)
```

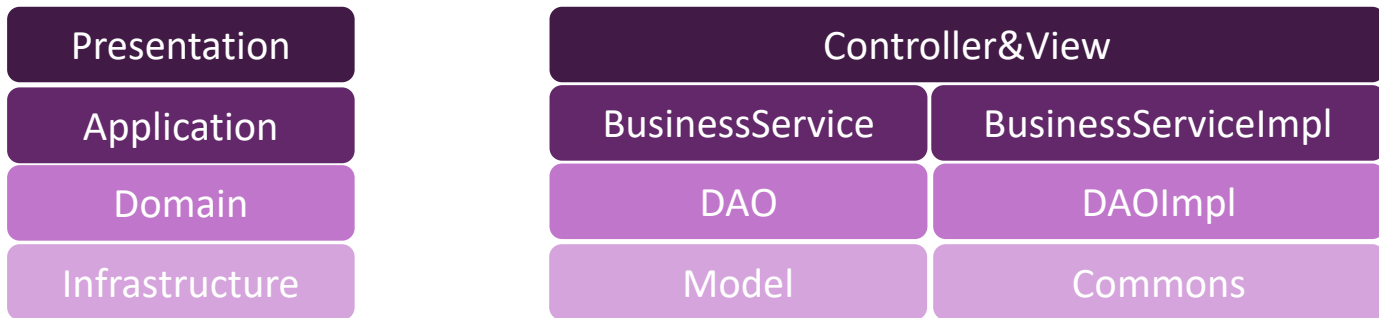
# Partie 5 : pom.xml

Un peu plus avancé



## pom.xml : Un peu plus avancé (modules)

Il arrive souvent qu'un projet soit composé en différents (sous)modules techniques



Afin d'éviter d'avoir à lancer une commande sur chaque modules, Maven implémente un mécanisme d'**orchestration** (ou de **composition**)

A vertical line on the left side of the slide, featuring a series of circles. The fifth circle from the top is filled with a purple color, while the others are white with black outlines.

## pom.xml : Un peu plus avancé (modules)

Pour cela, on peut utiliser un POM comme orchestrateur

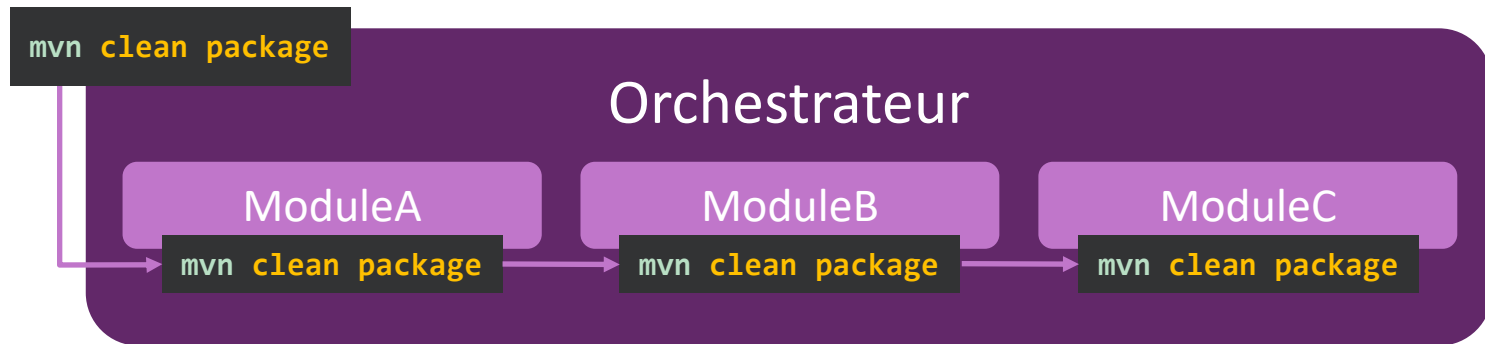
Son rôle est alors de déclarer les modules sur lesquels vont se répercuter ses commandes

```
<modules>
  <module>moduleA</module>
  <module>moduleB</module>
  (...)
</modules>
```

*NB : Les noms à donner sont les <artifactId> et peuvent être référencés de manière relative*

## pom.xml : Un peu plus avancé (modules)

Un build sur l'orchestrateur va déclencher les builds successifs des modules dans l'ordre dans lequel ils sont décrits



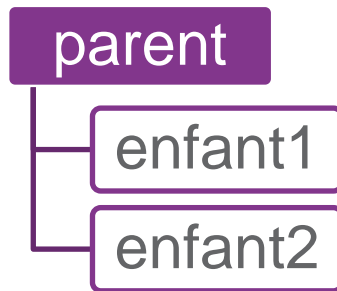
*NB : Un POM peut à la fois jouer le rôle de parent et d'orchestrateur*



## pom.xml : Un peu plus avancé (modules)



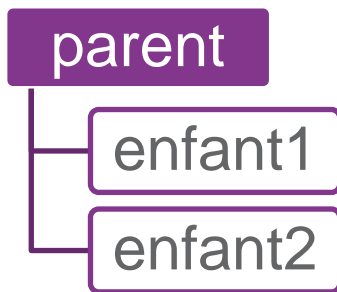
**Exercice 1** : Reprendre l'exercice précédent et faire en sorte que le parent soit également l'orchestrateur de ses enfants



## pom.xml : Un peu plus avancé (modules)



**Exercice 1** : Reprendre l'exercice précédent et faire en sorte que le parent soit également l'orchestrateur de ses enfants



```
<modules>
  <module>enfant1</module>
  <module>enfant2</module>
</modules>
```

A vertical line on the left side of the slide with seven circles. The fourth circle from the top is filled with a purple color, while the others are white with black outlines.

## pom.xml : Un peu plus avancé (profile)

La principale plus-value de Maven est d'avoir un fonctionnement portable :

- Il évite toutes les références vers le file system
- Il utilise un repository local

Sous certaine condition, cette portabilité est impossible :

- Différence dans les librairies selon l'environnement
- Besoin référencer un fichier sur un autre emplacement
- ...

Les **profiles** sont là pour ça !




## pom.xml : Un peu plus avancé (profile)

Les profiles peuvent être déclarés dans les 3 fichiers de configuration :

- Dans la configuration projet (*pom.xml*)
- Dans le profil utilisateur (*%USER\_HOME%/.m2/settings.xml*)
- Dans la configuration globale (*\${maven.home}/conf/settings.xml*)

Un profil est composé de :

- Une condition d'activation
- Son effet



```
<profile>
  <id>dev</id>
  <activation>
    <os>
      <family>Windows</family>
    </os>
  </activation>
  <dependencies>
    <dependency>
      <groupId>org.demo</groupId>
      <artifactId>adependency</artifactId>
      <version>1.0.0-SNAPSHOT</version>
    </dependency>
  </dependencies>
</profile>
```



## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml

```
>mvn clean install -P profile-1,profile-2
```

A vertical line on the left side of the slide, featuring a series of circles. The fourth circle from the top is filled with a purple color, while the others are white with black outlines.

## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml
- Dans le pom.xml avec

```
<settings>
(...)
  <activeProfiles>
    <activeProfile>profile-1</activeProfile>
  </activeProfiles>
(...)
</settings>
```

A vertical line on the left side of the slide, consisting of a series of circles. The fifth circle from the top is filled with a purple color, while the others are white with black outlines.

## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml
- Dans le pom.xml avec
  - La version du JDK
  - Des conditions d'OS

```
<profiles>
  <profile>
    <activation>
      <jdk>[1.3,1.6)</jdk>
    </activation>
    (...)
  </profile>
</profiles>
```



## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml
- Dans le pom.xml avec
  - La version du JDK
  - Des conditions d'OS
  - Des variables

```
<profiles>
  <profile>
    <activation>
      <os>
        <name>Windows XP</name>
      </os>
    </activation>
    (...)
  </profile>
</profiles>
```





## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml
- Dans le pom.xml avec
  - La version du JDK
  - Des conditions d'OS
  - Présence/absence de variables
  - La présence/Absence de fichiers

```
<profiles>
  <profile>
    <activation>
      <property>
        <name>debug</name>
      </property>
    </activation>
    (...)
  </profile>
</profiles>
```

```
>mvn clean install -Ddebug=chat
```



## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml
- Dans le pom.xml avec
  - La version du JDK
  - Des conditions d'OS
  - Présence/absence de variables
  - La présence/Absence de fichiers

```
<profiles>
  <profile>
    <activation>
      <property>
        <name>!debug</name>
      </property>
    </activation>
    (...)
  </profile>
</profiles>
```



## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml
- Dans le pom.xml avec
  - La version du JDK
  - Des conditions d'OS
  - Présence/absence de variables
  - La présence/Absence de fichiers

```
<profiles>
  <profile>
    <activation>
      <property>
        <name>env</name>
        <value>!prod</value>
      </property>
    </activation>
    (...)
  </profile>
</profiles>
```



## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml
- Dans le pom.xml avec
  - La version du JDK
  - Des conditions d'OS
  - Présence/absence de variables
  - Présence/absence de fichiers
  - Par défaut

```
<profiles>
  <profile>
    <activation>
      <file>
        <missing>mon/fichier.xml</missing>
      </file>
    </activation>
    (...)
  </profile>
</profiles>
```



## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml
- Dans le pom.xml avec
  - La version du JDK
  - Des conditions d'OS
  - Présence/absence de variables
  - Présence/absence de fichiers
  - Par défaut

Il est également possible de désactiver

```
<profiles>
  <profile>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    (...)
  </profile>
</profiles>
```



## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml
- Dans le pom.xml avec
  - La version du JDK
  - Des conditions d'OS
  - Présence/absence de variables
  - Présence/absence de fichiers
  - Par défaut

```
>mvn clean install -P !profile-1
```

Il est également possible de désactiver explicitement les profils



## pom.xml : Un peu plus avancé (profile)

L'activation des profils peut se faire de plusieurs manières différentes :

- En ligne de commande
- A travers les settings.xml
- Dans le pom.xml avec
  - La version du JDK
  - Des conditions d'OS
  - Présence/absence de variables
  - Présence/absence de fichiers
  - Par défaut

```
>mvn help:active-profiles
```

```
>mvn help:active-profiles -Denv=dev
```

Il est également possible de désactiver explicitement les profils

Et de savoir à tout moment quel profil est actif



# pom.xml : Un peu plus avancé (profile)

Que mettre dans un profil ?

Dans un pom, un profil peut contenir :

dependencies	modules	reporting
dependencyManagement	repositories	deploy
plugins	pluginRepositories	dependencyManagement
properties	build	distributionManagement

Les profils liés au settings ne sont pas portables !

Par conséquent : peuvent contenir uniquement :

- repositories
- pluginRepositories
- properties



## **pom.xml : Un peu plus avancé (profile)**



**Exercice 1** : Imaginez un cas d'utilisation de profils

**Exercice 2** : Spring met à disposition des développeurs une librairie (`org.springframework.boot:spring-boot-devtools`) permettant de faciliter les développements. Cette librairie permet de bypasser certaines sécurités et ne doit pas être déployé en production !

Créer un profil permettant d'embarquer (ou pas) cette librairie en fonction de l'environnement cible

# pom.xml : Un peu plus avancé (profile)



Exercice 1 : Imaginez un cas d'utilisation de profils

Des plugins qui s'exécutent en fonction de l'environnement

Exercice 2 : Spring met à disposition des développeurs une librairie (org.springframework.boot:spring-boot-devtools) permettant de faciliter les développements.

Cette librairie permet de bypasser certaines sécurités et ne doit pas être déployé en production !

Créer un profil permettant d'embarquer (ou pas) cette librairie en fonction de l'environnement cible

```
<profiles>
  <profile>
    <id>peuimporte</id>
    <activation>
      <property>
        <name>env</name>
        <value>!prod</value>
      </property>
    </activation>
    <dependencies>
      <dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-devtools</artifactId>
        <version>2.2.2.RELEASE</version>
      </dependency>
    </dependencies>
  </profile>
</profiles>
```



## pom.xml : Un peu plus avancé (plugins)

On peut déclarer des plugins dans un POM :

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-[plugin]-plugin</artifactId>
      <version>1.0</version>
    </plugin>
    (...)
  </plugins>
</build>
```

Comme pour les dépendances, la gestion des versions des plugins peut se faire via la section `<pluginsManagement>`

Deux grandes raisons pour les déclarer

- Changement de la configuration par défaut (version, paramètres etc...)
- Insérer un goal dans le cycle de vie



## pom.xml : Un peu plus avancé (plugins)

Exemple : Changement de configuration du plugin « Clean »

```
<plugin>
  <artifactId>maven-clean-plugin</artifactId>
  <version>3.0.0</version>
  <configuration>
    <excludeDefaultDirectories>true</excludeDefaultDirectories>
    <filesets>
      <fileset>
        <directory>${basedir}/temp</directory>
      </fileset>
    </filesets>
  </configuration>
</plugin>
```

A vertical line on the left side of the slide with seven circles. The fourth circle from the top is filled with a dark purple color, while the others are white with dark outlines.

## pom.xml : Un peu plus avancé (plugins)

On peut ajouter la balise `<executions>` au plugin pour lui indiquer un déclenchement automatique :

- Une exécution hérite de la configuration du plugin
- Une exécution peut surcharger/ajouter sa propre configuration

Il faut également préciser dans la balise `<execution>` à quelle phase le plugin doit se déclencher avec la balise `<phase>`

# pom.xml : Un peu plus avancé (plugins)

On parle du plugin « dependency » et on utilise la version 3.0.2

On modifie la configuration par défaut (pour tout goal du plugin) pour :

- Afficher uniquement le scope compile
- L'imprimer dans un fichier tree.txt

Lors de la phase d'installation...

...on exécute le goal dependency:tree...

...en surchargeant la configuration par défaut :

- On prendra uniquement le scope provided (et non compile)
- On continue d'imprimer le résultat dans un fichier tree.txt

```
<plugin>
  <artifactId>maven-dependency-plugin</artifactId>
  <version>3.0.2</version>
  <configuration>
    <scope>compile</scope>
    <outputFile>tree.txt</outputFile>
  </configuration>
  <executions>
    <execution>
      <phase>install</phase>
      <goals><goal>tree</goal></goals>
      <configuration>
        <scope>provided</scope>
      </configuration>
    </execution>
  </executions>
</plugin>
```

## pom.xml : Un peu plus avancé



**Exercice 1** : Entrez la commande : `>mvn help:effective-pom`  
D'après-vous, que représente ce pom ?

**Exercice 2** : Faire en sorte que le dossier target de vos projets se « clean » après une installation

**Exercice 3** : Grâce au plugin maven-resources-plugin:3.1.0 et au goal copy-resources, copier le fichier pom.xml dans le dossier *target* systématiquement lors de la phase *validate*

## pom.xml : Un peu plus avancé



Exercice 1 : Entrez la commande : `>mvn help:effective-pom`

D'après-vous, que représente ce pom ?

Le pom « à plat » avec toutes les dépendances et les plugins utilisés

Exercice 2 : Faire en sorte que le dossier target de vos projets se « clean » après une installation

Exercice 3 : Grâce au plugin maven-resources-plugin:3.1.0 et au goal copy-resources, copier le fichier pom.xml dans le dossier *target* systématiquement lors de la phase *validate*



# pom.xml : Un peu plus avancé



```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-clean-plugin</artifactId>
      <executions>
        <execution>
          <phase>install</phase>
          <goals>
            <goal>clean</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

# pom.xml : Un peu plus avancé



```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-resources-plugin</artifactId>
      <version>3.1.0</version>
      <executions>
        <execution>
          <id>copy-resources</id>
          <phase>validate</phase>
          <goals>
            <goal>copy-resources</goal>
          </goals>
          <configuration>
            <resources>
              <resource>
                <directory>${basedir}</directory>
                <includes>pom.xml</includes>
              </resource>
            </resources>
            <outputDirectory>${basedir}/src/main/resources/pom</outputDirectory>
          </configuration>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

# Conclusion



A vertical line on the left side of the slide, consisting of a series of circles. The top five circles are white with a dark gray outline, and the bottom circle is solid blue with a dark gray outline.

# Conclusion

Félicitations !  
Vous avez survécu aux basiques de Maven !

A ce stade, vous savez :

- Installer Maven et créer un projet
- Exécuter des commandes et comprendre les étapes exécutées
- Déclarer des dépendances et comprendre la transitivité
- Faire de l'héritage entre les POMs
- Utiliser des propriétés et le injecter dans les ressources
- Orchestrer les builds de différents modules
- Gérer différents profils
- Insérer des plugins dans un cycle de vie



## Conclusion

Les limites de Maven :

- Verbeux
- Limité par les fonctionnalités des goals
- Télécharge des dépendances à chaque utilisation
  - Difficile d'utilisation en mode hors-ligne
- Un concurrent : Gradle

# Pour aller plus loin...

Voici les points qui n'ont pas été abordés :

Descriptif	Dépendances	Variables	Chargement conditionnel	Documentation	Référentiels	Infos CI/CD
groupId <span>1</span> artifactId version packaging  parent <span>4</span>  modules <span>1</span>	dependencies <span>3</span> dependency-Management	properties <span>5</span>	profiles <span>2</span>	name <span>2</span> description url inceptionYear organization licenses developers contributors	repositories pluginRepositories	scm issueManagement ciManagement mailingLists reporting prerequisites  build <span>3</span>

Pour aller plus loin :

- Le plugin de release
- Créer son archetype (avancé)

A vertical line on the left side of the slide, composed of a thin grey line with six colored circles (green, cyan, orange, purple, lime green, blue) spaced evenly along it.

Pour réviser (ou aller plus loin) :

<https://maven.apache.org/what-is-maven.html>

Repo Maven :

<https://mvnrepository.com/>

*That's all Folks!*