



Maven

---

**oxiane**  
Luxembourg



# Agenda

---

## Core Concepts

- The *pom.xml*
- Maven repositories
- Packaging and artefacts formats
- POMs relationships and multi-modules
- Version Management

## Build customisation

- LifeCycle customisation
- Profiles

## Release

### management

- Deploying to private repositories
- Nexus a repositories manager
- Maven and SCMs
- The Release plugin

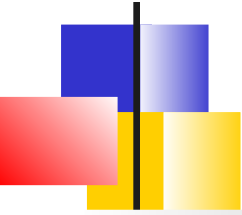


# What is Maven

---

- Official definition (Apache) : Project management tool which provides :
  - A project **model**
  - Set of **standards**
  - A Life Cycle (the phases from source to deployment)
  - **Dependencies management**
  - Logic to execute goals thanks to **plugins**
- To use Maven, we must describe our project in a Project Object Model file : *pom.xml*
- Then, Maven apply transversal logic via plugins shared by the java community

# Convention versus Configuration

- 
- It means reducing the configuration tasks by adopting conventions (default values used by all the community)
  - For example, Maven lie on a default project tree :
    - **`${basedir}/src/main/java`** contains the code source
    - **`${basedir}/src/main/resources`** contains resources
    - **`${basedir}/src/test`** contains test classes
    - **`${basedir}/target/classes`** The compiled classes
    - **`${basedir}/target`** The final jar to deploy
  - All the core plugins of Maven use this default structure to compile, package, generate documentation, etc.  
=> Then everything work if we adopt this default project organization



# Before/After

---

- Before Maven, a team's member was dedicated to develop and maintain build scripts (Ant, Makefile)
- With Maven, every one can :
  - Check-out the source
  - Execute ***mvn install***



# Plugins

---

- The core of Maven is only able to parse some XML files.
- Most tasks are in fact delegated to ***plugins.***
- These plugins are automatically retrieved from the central repository of Maven
- The first time `mvn install` is executed, several required plugins are downloaded



# Conceptual model of a project

---

- Developers must specify the model of their project :
  - Unique Identifier  $\Leftrightarrow$  Maven coordinates
  - License model
  - Developers and Contributors
  - Libraries used
  - Repositories used
  - Plugins configuration
  - ...
- The « Project Object Model » is described in a single XML file : ***pom.xml***



# POM'S Content

---

The POM contains 4 sets of information :

**POM relationships** : Inheritance, submodules, coordinates dependencies..

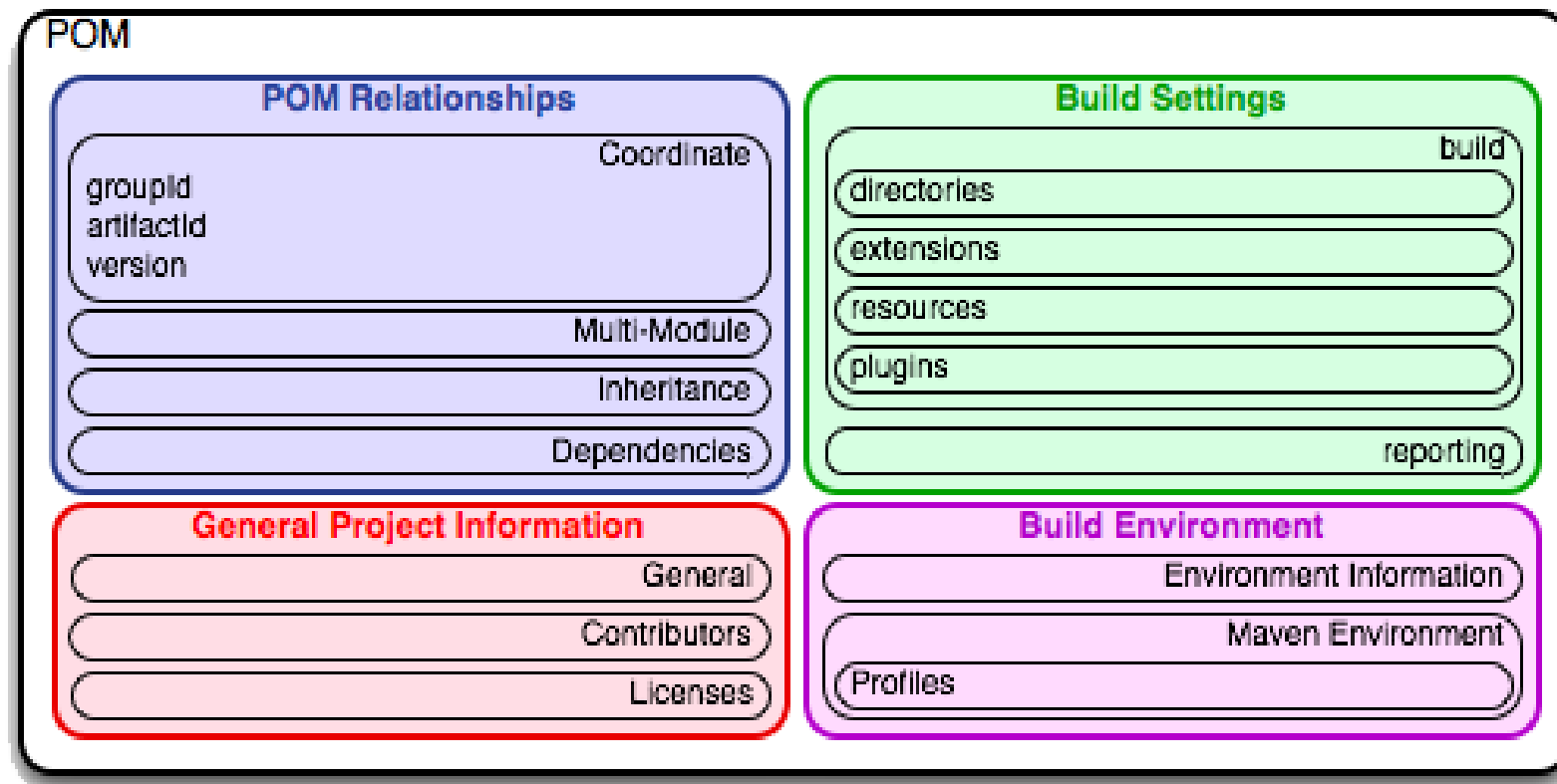
**General information** : Name of the project, web site, organization, developers, contributors, licence

**Build settings** : Customization of the default build : default plugins configuration, integration of other plugins

**Build environments** : Build profiles to adapt the build for a particular environment : (development, integration, QA production)



# POM's content





# Example *pom.xml*

---

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
    <modelVersion>4.0.0</modelVersion>
    <groupId>org.sonatype.mavenbook.ch03</groupId>
    <artifactId>simple</artifactId>
    <packaging>jar</packaging>
    <version>1.0-SNAPSHOT</version>
    <name>simple</name>
    <url>http://maven.apache.org</url>
    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>3.8.1</version>
            <scope>test</scope>
        </dependency>
    </dependencies>
</project>
```



# Maven's coordinates

---

- Maven coordinates are composed :
  - **groupid** : Equivalent to a domain's name.  
Ex : org.apache
  - **artifactId** : The name of the project.  
Ex : lucene
  - **version** : A specific release of a project.  
Projects under development use a special suffix :  
SNAPSHOT.
  - **packaging** : The format of distribution
- These coordinates identify the project in the Maven space. They must be unique around the world
- Coordinates are used to specify a dependency, a plugin, a parent project, ...



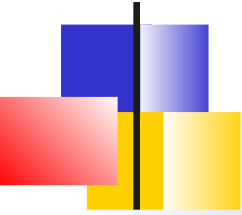
# Repositories

---

- The *groupId:artifactId:version* coordinates make the project unique and allow Maven to locate the final artefact in the configured repositories
- By default, Maven Central is already configured in Maven (<https://mvnrepository.com/>)
- If we want to add other repositories (corporate or third-party), we must define them in `$MAVEN_HOME/settings.xml`

By default, `$MAVEN_HOME = $HOME/.m2`

# Properties

- 
- A POM may include properties evaluated during execution  
Example : `${project.groupId}-${project.artifactId}`
  - Maven provides implicit properties :
    - ***env*** : Environnement variables
    - ***project*** : Project's attributes
    - ***settings*** : Settings attributes defined in *setting.xml*
    - JVM properties
  - Properties can also be defined in pom.xml :  
Example : `<properties> <foo>bar</foo> </properties>`



# Dependencies management

---

- Dependencies are specified via Maven coordinates.

```
<dependency>
  <groupId>log4j</groupId>
  <artifactId>log4j</artifactId>
  <version>1.2.14</version>
</dependency>
```

- Artefact and their associated POM are stored in repositories. Then, Maven is able to resolve **transitive dependencies**.  
This is surely the most valuable functionality of Maven
- Additionally, it is possible to specify the **scope** of a dependency.  
The scope precise where the dependency is needed  
For example, the dependency *junit* is only needed in the test scope



# Transitive dependencies

---

- For example, if your project depends on a library B which itself depends on a library C, it is not necessary to indicate the dependence on C
- When a dependency is uploaded to the local repository, Maven also retrieves the dependency pom and is able to retrieve the other dependent artifacts. These dependencies are then added to the dependencies of the current project.
- If this is not the desirable behavior, it is possible to exclude certain transitive dependencies

# Adding dependencies



Adding dependency is done via a **<dependency>** tag under the **<dependencies>**

Example :

```
<dependencies>
  <dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.14</version>
  </dependency>
</dependencies>
```



# Scopes



---

The `<dependency>` tag can also include a **`<scope>`** tag.

5 scopes are available :

- **`compile`** : The default scope, dependencies are always in the classpath and are packaged in the distribution
- **`provided`** : Dependency is provided by a container (A JavaEE server for example). Needed for compilation but not packaged
- **`runtime`** : Needed for execution and test but not for compilation. Ex : A JDBC implementation
- **`test`** : Needed only for tests
- **`system`** : Provided by the system (not downloaded from a repository)

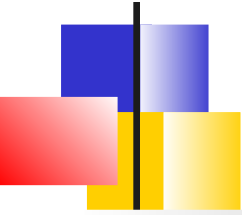
# Example



---

```
<dependency>  
  <groupId>org.apache.commons</groupId>  
  <artifactId>commons-io</artifactId>  
  <version>1.3.2</version>  
  <scope>test</scope>  
</dependency>
```

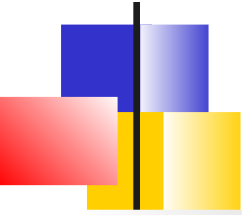
# Dependency exclusion



---

```
<dependencies>
  <dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate</artifactId>
    <version>3.2.5.ga</version>
    <exclusions>
      <exclusion>
        <groupId>javax.transaction</groupId>
        <artifactId>jta</artifactId>
      </exclusion>
    </exclusions>
  </dependency>
  <dependency>
    <groupId>org.apache.geronimo.specs</groupId>
    <artifactId>geronimo-jta_1.1_spec</artifactId>
    <version>1.1</version>
  </dependency>
</dependencies>
```

# Where to find dependencies ?

- 
- 
- The website <http://www.mvnrepository.com> provides a user interface to search for a specific artifact
  - With a keyword, we can retrieve all the artefacts available and their releases.
  - By clicking on a specific release, we can access the `<dependency>` element to include in our POM



# Maven commands

---

***mvn <plugins:goals> <phase>***

The command may specify the execution of :

- A specific **goal** of a plugin
- A maven **phase** ... which leads to the execution of different plugins goals in sequence



# Plugins and goals

---

- A Maven **plugin** provides several atomic tasks named **goals**
- For example :
  - The **Jar** plugin provides goals to create archive
  - The **Compiler** plugin provides goals to compile source code (either main code or test class)
  - The **Surefire** plugin provides goals to execute tests and generate test reports.
  - There are lot of plugins available which integrate a specific tool with Maven (SonarQube, JMeter, Nexus, ...)

You can write your own plugin with Java, Ant, Groovy, beanshell, ...



# Example

---

Execution of the goal *generate* of the *archetype* plugin

```
$ mvn archetype:generate  
  -DgroupId=org.formation.maven.simpleProject \  
-DartifactId=simpleProject \  
-DpackageName=org.formation.maven
```

This execution starts a wizard to create a Maven project (tree and pom.xml) from a template (archetype)



# Maven Life Cycle

---

- A Maven LifeCycle is an **ordered sequence of phases** participating to the build of an artefact.
- Maven supports different LifeCycles.  
The default life cycle is the most used, it starts with a phase validating the integrity of the project and ends with the deploy phases.
- The LifeCycle's phases are intentionally vague : *validation, test or deployment*
- Default goals may be attached to them and project's personalization consists to attach specific goals to phases



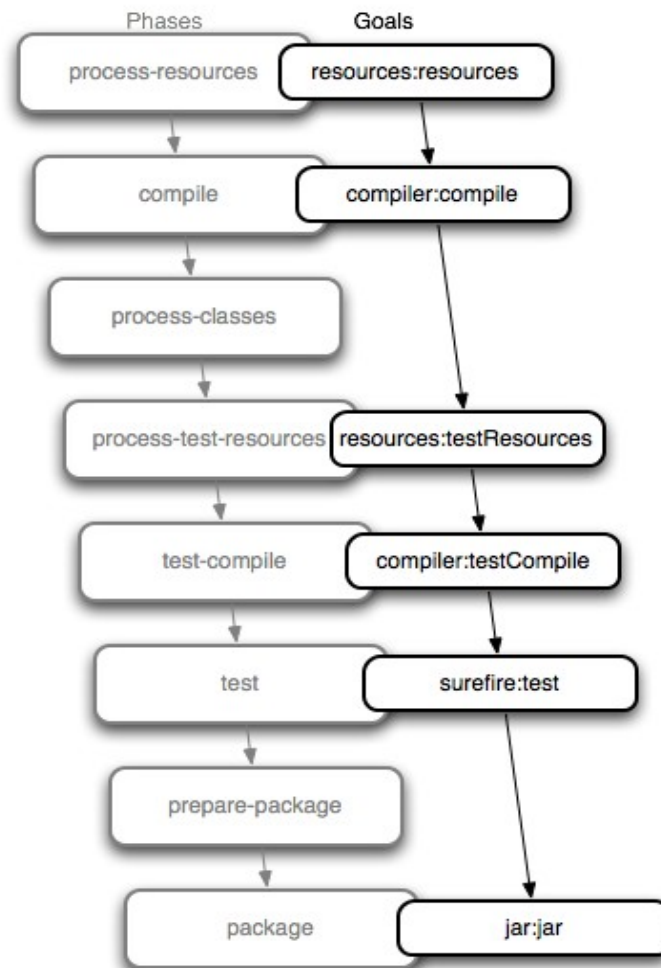


# Phases and goals

---

- Plugin's goals may be **attached** to Maven phases. When maven reach a phas during a build execution, all the attached goals are executed.
- For example, when executing ***mvn install*** *install* represents the phase and it will lead to the execution of all the objectives attached to the precedent phases of *install* and to the objectives attached to *install* itself

# Default life cycle





# Maven repositories



# Maven Repositories

---

- A **repository** designates the storage space for Maven artifacts (plugins, jars, pom.xml)
- A repository can be :
  - Remote
  - Local, acting as a cache of remote repositories



# Adding repositories

---

- Default repositories can be replaced or completed with references to other repositories (Third party or corporate).
- Some tools are dedicated to manage corporate repositories like Nexus, Archiva, Artifactory

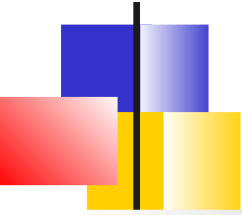


# Repository structure

---

- POMs and artefacts are store in a directory's tree which matches the Maven coordinates
- This structure is easy browsable with a file explorer or a browser  
(ex : <http://repo1.maven.org/maven2/>. )
- The standard directory for storing an stocker un artefact dans un répertoire :  
`/<groupId>/<artifactId>/<version>/<artifactId>-<version>.<packaging>`

# Local repository and specific configuration

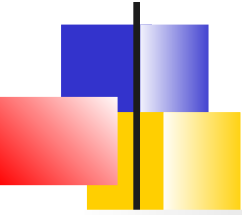
- 
- During its first executions, Maven creates configuration files and a local repository in the home directory of the user :
    - ***~/.m2/settings.xml*** : This file may contains specific configuration, declaration of repositories, source control management, credentials of the user, ...
    - ***~/.m2/repository/*** : The local repository which contains downloaded artefacts (libraries, plugins, poms)



# Packaging and artifacts formats



# Packaging

- 
- The tag **<packaging>** specifies the format of the artefact that the project builds.
  - It has a direct influence on the goals executed during a life cycle.
  - The default packaging is **jar**. But, there are other packaging formats



# Default goals for a jar packaging

---

- ***process-resources / resources:resources*** : Copy all the files from *src/main/resources* to *target*
- ***compile / compiler:compile*** : Compile all the source code located in *src/main/java* to *target*.
- ***process-test-resources / resources:testResources*** : Copy all the files from *src/test/resources* to test target directory
- ***test-compile / compiler:testCompile*** : Compile all the source code located in *src/test/java* to test target directory
- ***test / surefire:test*** : Execute all the tests and create test reports, stop the build if a test fails
- ***package / jar:jar*** : Create a jar of compiled classes and resources in the target repository



# Packaging *pom*

---

- The artefact to build is then a POM
- The build generally depends on other project
- For example, multi-modules project



# Default goals for a *pom* packaging

---

- ***package / site-attach-descriptor*** : Add site descriptor (*site.xml*) which provides documentation
- ***install / install:install*** : Install the artefact (pom.xml) in the local repository
- ***deploy / deploy:deploy*** : Deploy the artefact in a remote repository



# Packaging JavaEE

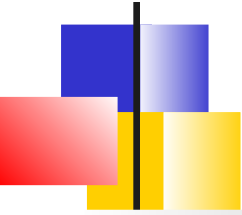
---

- 3 packaging related to the Java EE standard are available :
  - *ejb* : Enterprise Java Beans archive
  - *war* : Web application
  - *ear* : Enterprise application



# POM relationships and multi-modules project

# Effective POM

- 
- There may exist inheritance relationships between POMs :
    - A parent POM to share configuration properties
    - A POM of a multi-modules project
  - Every POM inherits from a root POM named Super POM
  - The effective *pom.xml* used for execution is then a combination of the project's *pom*, of all its parents *pom* and of specific settings of the user
  - To display the effective POM, the help plugin provides :  
**\$ mvn help:effective-pom**



# Parent POM

---

A reference to a parent POM is specified by the tag **<parent>**

```
<parent>
  <groupId>org.formation.maven</groupId>
  <artifactId>parent</artifactId>
  <version>1.0</version>
</parent>
```

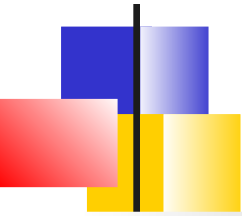
Maven looks for the *pom.xml* in the local repository.

It is possible to point directly to the file system by indicating the **<relativePath/>** tag

```
<parent>
  <groupId>org.formation.maven</groupId>
  <artifactId>parent</artifactId>
  <version>1.0</version>
  <relativePath>../parent-project/pom.xml</relativePath>
</parent>
```



# Super POM



The Super POM is included in the distribution  
(jar *maven-model-builder-x.x.jar*)

- It mainly defines :
  - Maven central.
  - The main plugins and their default versions
  - The default directory tree

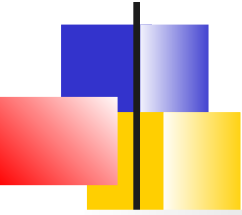
# Multi-modules projects



---

- A multi-modules project contains modules which can be built independently and which may have dependencies on each other
- The project has a parent POM which lists the directories of the sub-modules via the tag ***<modules>***
- Each sub-modules has its own *pom.xml* which can inherit from the parent one

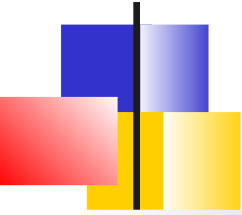
# POM parent



---

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/maven-v4_0_0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>org.sonatype.mavenbook.ch07</groupId>
  <artifactId>simple-parent</artifactId>
  <packaging>pom</packaging>
  <version>1.0</version>
  <name>Chapter 7 Simple Parent Project</name>
  <modules>
    <module>simple-model</module>
    <module>simple-persist</module>
    <module>simple-webapp</module>
  </modules>
</project>
```

# Parent POM of a multi-modules project

- 
- The POM parent defines the *groupId* and the *version* of Maven coordinates
  - It does not build a jar itself, then its packaging is ***pom***.
  - In the ***<modules>*** element, the sub-modules are listed with a sub-directory
  - In each subdirectory, there is a *pom.xml* which fullfill the Maven coordinates with the *artifactId*
  - The parent POM may contain configuration properties that will inherited by submodules

# POM of sub-modules

- Sub-modules reference their parent via its coordinates
- Sub-modules have often dependencies upon each other

```
<parent>
```

```
  <groupId>org.sonatype.mavenbook.ch07</groupId>
```

```
  <artifactId>simple-parent</artifactId>
```

```
  <version>1.0</version>
```

```
</parent>
```

```
...
```

```
<dependencies>
```

```
  <dependency>
```

```
    <groupId>org.sonatype.mavenbook.ch07</groupId>
```

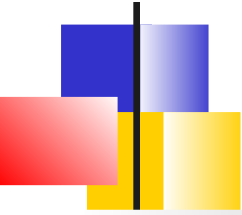
```
    <artifactId>simple-model</artifactId>
```

```
    <version>1.0</version>
```

```
  </dependency>
```

```
...
```

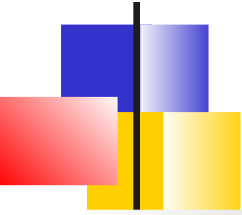
# Maven Execution

- 
- When a Maven command is executed in the parent directory
    - Maven starts with loading the POM parent and locates sub-modules POMs
    - Then, sub-modules dependencies are analyzed and determine the order of compilation and installation



# Versions management

# Project's version

- 
- The version of a Maven project allow to group and order different releases.
  - It is generally composed of 4 parts :

`<major version>.<minor version>.<incremental version>-<qualifier>`

Example : *1.3.5-beta-01*

- By respecting this format, Maven is able to determine the most recent version
- SNAPSHOT suffix indicates that this version is under development .... and means that the artefact cannot be cached in a repository





# Dependencies version

---

To avoid that sub-modules (or different projects of the same company) use different versions of the same library, Maven provides the **<dependencyManagement>** tag

It allows to centralized the version numbers of dependencies in a POM parent

Children projects no longer need to specify the version of a dependency

# POM parent



---

```
<dependencyManagement>
```

```
  <dependencies>
```

```
    <dependency>
```

```
      <groupId>mysql</groupId>
```

```
      <artifactId>mysql-connector-java</artifactId>
```

```
      <version>5.1.2</version>
```

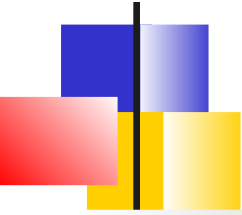
```
    </dependency>
```

```
    . . .
```

```
  </dependencies>
```

```
</dependencyManagement>
```

# Child POM



---

```
<parent>
  <groupId>org.sonatype.mavenbook</groupId>
  <artifactId>a-parent</artifactId>
  <version>1.0.0</version>
</parent>
<artifactId>project-a</artifactId>
...
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
    <!-- No version tag is needed -->
  </dependency>
</dependencies>
```



# Dependencies importation via BOM

---

A **BOM (Bills Of Material)** is a specific POM only used to manage versions of dependencies

It provides a single point of configuration.

If a child project imports a BOM, it does not need anymore to specify versions.



# Example

---

```
<project ...>
  <modelVersion>4.0.0</modelVersion>
  <groupId>baeldung</groupId>
  <artifactId>Baeldung-BOM</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <packaging>pom</packaging>
  <description>parent pom</description>
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>test</groupId>
        <artifactId>a</artifactId>
        <version>1.2</version>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>b</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
      </dependency>
      <dependency>
        <groupId>test</groupId>
        <artifactId>c</artifactId>
        <version>1.0</version>
        <scope>compile</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```



# 2 use cases

---

## Inheritance or better : import

```
<project ...>
  ...
  <dependencyManagement>
    <dependencies>
      <dependency>
        <groupId>baeldung</groupId>
        <artifactId>Baeldung-BOM</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <type>pom</type>
        <scope>import</scope>
      </dependency>
    </dependencies>
  </dependencyManagement>
</project>
```



# Versions of plugins

---

The same mechanism is used to fix the versions of the plugins

In a POM parent, the

**<pluginManagement>** element sets the versions of the plugins potentially used and their configuration properties

=> All the children POMs use the same version of the plugins

# Exemple *pluginManagement*



---

```
<build>
  <pluginManagement>
    <plugins>
      <plugin>
        <groupId>org.codehaus.mojo</groupId>
        <artifactId>hibernate3-maven-plugin</artifactId>
        <version>2.1</version>
        <configuration>
          <components>
            <component>
              <name>hbm2ddl</name>
              <implementation>annotationconfiguration</implementation>
            </component>
          </components>
        </configuration>
      </plugin>
    </plugins>
  </pluginManagement>
```





# Build customisation

---

Customization of the life cycle  
Profiles



# Introduction

---

2 alternatives exist to customize the build

- **Configure** plugins by overriding the default configuration.

*Access the doc. Reference to see all options of a plugin*

- **Attach** goals of existing or new plugins to specific phases of the build



# Example *compiler:compile*

---

Default behaviour is to compile *src/main/java* to *target/classes*

The plugin *Compile* uses *javac* and assumes that sources are in Java 1.6 and targets a 1.6 JVM !

To change, we must specify the java versions in the *pom.xml*




# Example

---

```
<project> ...  
<build> ...  
<plugins>  
  <plugin>  
    <artifactId>maven-compiler-plugin</artifactId>  
    <configuration>  
      <source>1.8</source>  
      <target>1.8</target>  
    </configuration>  
  </plugin>  
</plugins> ...  
</build> ...  
</project>
```

# Generation of an executable

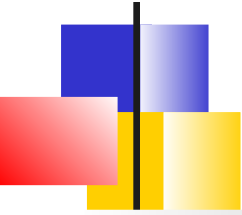
- 
- By default, Maven builds a library but not an executable jar
  - The **Assembly** plugin allows to create different format of executable jar (*.jar*, *.war*, *.ear*)
  - This plugin is not used in the default life cycle
  - If we want to insert this plugin in the build process, we have to :
    - Declare and configure it in the *pom.xml*
    - Attach one of its goal to a Maven phase

# Example, define a newplugin

- Example fat jar (*jar-with-dependencies*)

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
          <manifest><mainClass>org.maven.myapp.Main</mainClass></manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
...
$ mvn install assembly:single
```

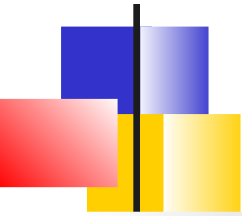
# Attach assembly with package phase



```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
      <executions>
        <execution>
          <id>make-assembly</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

```
...
$ mvn package
```

*Lab : LifeCycle customization*



# Profiles





# Profiles

---

- profiles*** allow to adapt the build to a specific environment (test, integration, production, ...)
- Profiles override the default configuration
  - There can as many profiles as you like
  - Profiles are activated :
    - either manually ,
    - or automatically according to environment variables or OS
  - Each profile, configured in *pom.xml*, has an unique identifier



# Example

---

```
<profiles>
  <profile>
    <id>production</id>
    <build>
      <plugins>
        <!-- Configuration overrides the default one for this profile -->
        <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>
```



# *<profiles>*

---

- The ***<profiles>*** element lists the profiles of the project.  
It is generally placed at the end of the file
- Each *<profile>* element
  - has an ***<id>*** nested element
  - may contain **all the elements of the *<project>*** element
- Activation via the command line is made with **-P<id>**



# Automatic activation

---

Il est possible d'activer automatiquement un profil en fonction de :

- ✓ La version du JDK
- ✓ Les propriétés de l'OS
- ✓ Les propriétés Maven (ou l'absence de propriété)
- ✓ La présence d'un fichier



# Release management

---

Deploying to private repositories  
Nexus  
The Release plugin  
Nexus



# Repositories

---

With Maven, there are different types of repositories used to store dependencies and install artifacts..

- The **local** repository
- **Public** repositories provided by Maven or third-party Redhat, Pivotal, .... ...
- **Mirror** repositories
- **Private** repositories, accessible via credentials and internal to a company



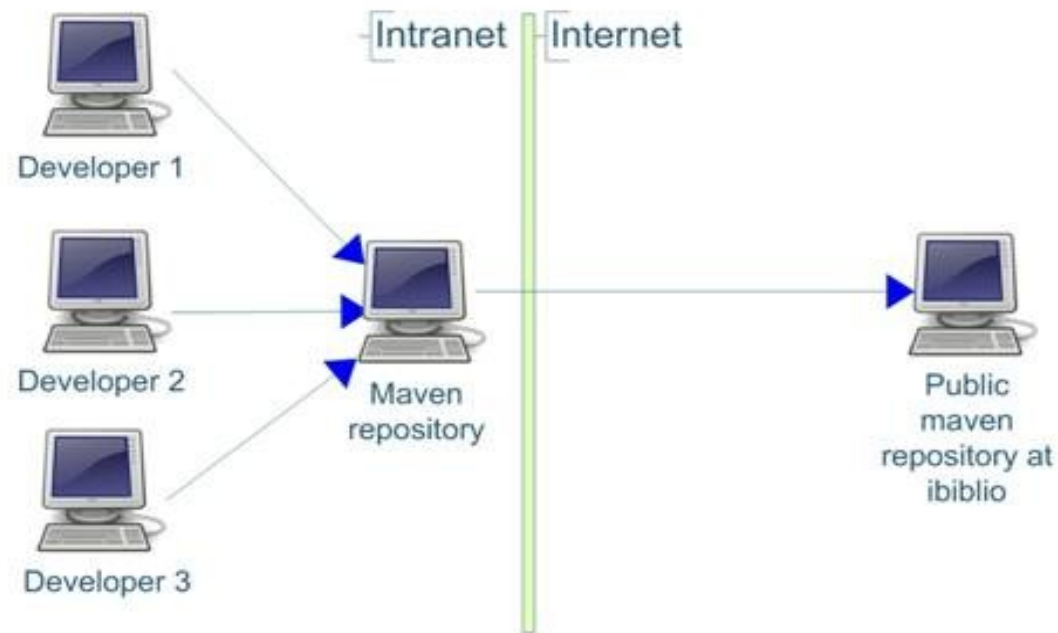
# Private repositories

---

Setting a corporate repository provides several features to a company :

- Security : Internal projects are not public but can be used inside the company. Some libraries can be disallowed
- Bandwith saving : the corporate repository acts as proxy of public repositories
- Archiving : All the produced artefacts of the company.
- Provisionning : Artefacts can be retrieved via different protocols : *http, scp, ftp ...*

# Architecture







# Configuration

---

There are two different ways to specify the use of other repositories.

- Specify in a the pom.xml
- Specify in settins.xml with a profile

In both cases, use the **<repository>** element with 3 sub-elements :

- *<id>* : An identifier
- *<name>* : A human name
- *<url>* : The URL of the root tree



# Sample : *pom.xml*

---

```
<project>
```

```
...
```

```
  <repositories>
```

```
    <repository>
```

```
      <id>my-repo1</id>
```

```
      <name>your custom repo</name>
```

```
      <url>http://jarsm2.dyndns.dk</url>
```

```
    </repository>
```

```
    <repository>
```

```
      <id>my-repo2</id>
```

```
      <name>your custom repo</name>
```

```
      <url>http://jarsm2.dyndns.dk</url>
```

```
    </repository>
```

```
  </repositories>
```

```
...
```

```
</project>
```



# Sample : *settings.xml*

---

```
<settings>
...
<profiles>
...
  <profile>
    <id>myprofile</id>
    <repositories>
      <repository>
        <id>my-repo2</id>
        <name>your custom repo</name>
        <url>http://jarsm2.dyndns.dk</url>
      </repository>
    </repositories>
  </profile>
...
</profiles>

<activeProfiles>
  <activeProfile>myprofile</activeProfile>
</activeProfiles>
...
</settings>
```



# Authentication

---

Generally, the corporate repository requires authentication

The configuration consist to declare in *settings.xml* the credentials needed to access the repository.

This is done with a **<server>** element having the same *<id>* element as the *<repository>* element



# Example

---

```
<settings xmlns="http://maven.apache.org/SETTINGS/1.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/SETTINGS/1.0.0
    http://maven.apache.org/xsd/settings-1.0.0.xsd">
  ...
  <servers>
    <server>
      <id>my-internal-site</id>
      <username>my_login</username>
      <password>my_password</password>
      <privateKey>${user.home}/.ssh/id_dsa</privateKey>
      <passphrase>some_passphrase</passphrase>
      <filePermissions>664</filePermissions>
      <directoryPermissions>775</directoryPermissions>
      <configuration></configuration>
    </server>
  </servers>
  ...
</settings>
```



# *Deploy* plugin

---

The ***deploy*** plugin deploys the artefact to its final repository.

Typically, a corporate repository.

This plugin provides 2 main goals :

***deploy*** : attached to *deploy* phase, it publishes to the final repository

***deploy-file*** : Deploys a file not managed by Maven



# *<distributionManagement>*

---

The use of deploy phase requires the configuration of the final repository via the ***<distributionManagement>*** element

```
<distributionManagement>
  <repository>
    <id>internal.repo</id>
    <name>MyCo Internal Repository</name>
    <url>Host to Company Repository</url>
  </repository>
</distributionManagement>
```



# A repository manager tool : Nexus





# Nexus

---

***Nexus*** is a repository manager which provides 2 main features :

- Proxy of remote repositories (Maven central)
- Hosting of private repository

*Nexus* supports several kinds of repository (Maven, npm, docker registry, ...)



# Installation

---

## Java 8

### Distributions :

- Archive :Dézipper

```
./nexus run
```

- Docker :

```
docker run -d -p 8081:8081 --name  
nexus sonatype/nexus3
```

### Access à

```
http://admin:admin123@localhost:8081
```



# Concepts

---

Nexus manages repositories and components

- **Components** are artefacts identified with their coordinates.  
Nexus allows to add some meta-data to a component.
- **Repositories** are Maven repositories or other kind of repo (npm, RubyGems, ...)



# Maven configuration

---

Maven configuration consists to edit *settings.xml* :

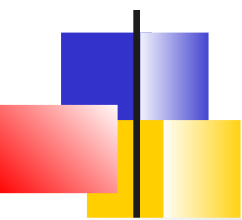
- Adding the mirroring functionality for Maven Central
- Override the central repository for dependencies and plugins



# Example

---

```
<settings>
  <mirrors>
    <mirror>
      <id>nexus</id>
      <mirrorOf>*</mirrorOf>
      <url>http://localhost:8081/repository/maven-public/</url>
    </mirror>
  </mirrors>
  <profiles>
    <profile> <id>nexus</id>
      <repositories>
        <repository>
          <id>central</id> <url>http://central</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </repository>
      </repositories>
      <pluginRepositories>
        <pluginRepository>
          <id>central</id> <url>http://central</url>
          <releases><enabled>true</enabled></releases>
          <snapshots><enabled>true</enabled></snapshots>
        </pluginRepository>
      </pluginRepositories>
    </profile> </profiles>
  <activeProfiles>
    <activeProfile>nexus</activeProfile>
  </activeProfiles>
</settings>
```



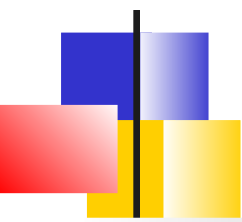
# Repository creation in Nexus

---

The URL of *settings.xml* must match a repository URL of Nexus

The default installation of Nexus has already created the ***maven-public*** repository which provides :

- *maven-releases* : Store internal releases
- *maven-snapshots* : Store internal snapshots
- *maven-central* : Proxy of Maven central



# Deployment configuration

---

To enable deployment to Nexus repository. You have to :

- Use *<distributionManagement>* and define 2 repositories :
  - One for snapshot
  - One for repository
- Specify the credentials in the settings file



# Project configuration

---

```
<project>
...
<distributionManagement>
  <repository>
    <id>nexus</id>
    <name>Releases</name>
    <url>http://localhost:8081/repository/maven-releases</url>
  </repository>
  <snapshotRepository>
    <id>nexus</id>
    <name>Snapshot</name>
    <url>http://localhost:8081/repository/maven-snapshots</url>
  </snapshotRepository>
</distributionManagement>
...
<settings>
....
  <servers>
    <server>
      <id>nexus</id>
      <username>admin</username>
      <password>admin123</password>
    </server>
  </servers>
```



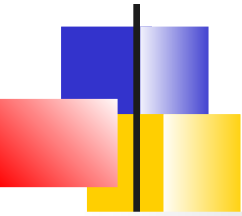


# User Interface

---

*Nexus* provides an anonymous access to search for components with many options

If we are logged in, we can administrate (repository management, security, traces, rest API , ...)



# Maven and SCMs



# Maven and SCM

---

Maven interacts with SCMs.

Then, SCM settings are provided in the *pom.xml*

Several plugins (like *SCM and Release*) use this information to facilitate build actions which involve the SCM



# Configuration

---

Configuration is made via **<scm>** which specifies :

- *<connection>* : Read access to the SCM
- *<developerConnection>* : Write access
- *<url>* : Documentation URL
- Other elements including tag, branch, depending on the SCM technology



# SVN sample

---

```
<scm>  
  <connection>scm:svn:http://127.0.0.1/svn/my-project</connection>  
  <developerConnection>scm:svn:https://127.0.0.1/svn/my-project</developerConnection>  
  <tag>HEAD</tag>  
  <url>http://127.0.0.1/websvn/my-project</url>  
</scm>
```



# Example GIT

---

```
<scm>
<url>https://github.com/kevinsawicki/github-maven-example</url>
<connection>
  scm:git:git://github.com/kevinsawicki/github-maven-example.git
</connection>
<developerConnection>
  scm:git:git@github.com:kevinsawicki/github-maven-example.git
</developerConnection>
</scm>
```



# Le plugin *scm*

---

The first plugin to use this information is ***scm***  
It provides 16 goals including all the usual  
commands of the SCM : *checkin, checkout,*  
*update, update-subprojects, add, diff, export,*  
*edit, unedit, status, changelog, branch, tag,*  
*validate, bootstrap*



# The *Release* plugin





# Plugin *Release*

---

The ***Release*** plugin facilitates the release procedure.

A typical release required to :

- Ensure that all source code is committed
- Ensure that the source code builds and all tests passed
- Tag the version in the SCM
- Increase the version number in source code for further development
- The plugin automates this procedure



# Plugin *Release*

---

A release is done with 2 goals :

***prepare*** : Preparation

***perform*** : Realization

Other goals are provided by the plugin :

*clean* : Cleaning temporary files created by prepare

*prepare-with-pom* : *prepare* + and generate the effective poms

*rollback* : Rebuild an old release

*stage* : Perform a release in a staging repository

*branch* : Create a branch with the version number updated

*update-versions* : Update the versions



# *prepare*

---

*prepare* executes a wizard in several steps, asking some information to the user :

- ✓ Check that all sources are committed
- ✓ Check that there is no dependency SNAPSHOT
- ✓ Change the versions in POMs from *x-SNAPSHOT* to a new version entered by the user
- ✓ Execute tests with the updated POMs
- ✓ Commit updated POMs modified
- ✓ Tag the SCM with the name of the version
- ✓ Change the versions in POMs with a new value : *y-SNAPSHOT*
- ✓ Commit the updated POMs
- ✓ Generates a *release.properties* file used by the *perform* goal



# Samples

---

Resume the command where it stopped

```
mvn release:prepare
```

Restart the whole command

```
mvn release:prepare -Dresume=false
```

Or

```
mvn release:clean release:prepare
```



# *perform*

---

- The ***perform*** goal executes 2 steps from *release.properties* :
- ✓ Perform a check-out from the URL SCM read in *release.properties*
  - ✓ Call the *deploy* phase