Pour aller plus loin :
Programmation outillée

Denis Conan

CSC4102
Table des matières

Pour aller plus loin : Programmation outillée

Denis Conan, Télécom SudParis, CSC4102
Janvier 2022

<table>
<thead>
<tr>
<th>1 Construction de logiciel avec Maven</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Main principles</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Artifact names and project relationships</td>
<td>3</td>
</tr>
<tr>
<td>1.2.1 Local repository and snapshot versions</td>
<td>5</td>
</tr>
<tr>
<td>1.3 Life-cycle for jar packaging — simplified</td>
<td>5</td>
</tr>
<tr>
<td>1.4 Build life-cycles, phases and goals</td>
<td>6</td>
</tr>
<tr>
<td>1.5 Standard directory structure — a less simplified view</td>
<td>6</td>
</tr>
<tr>
<td>1.6 More details on the build process</td>
<td>7</td>
</tr>
<tr>
<td>1.6.1 Exemplification with the JAR build life-cycle</td>
<td>7</td>
</tr>
<tr>
<td>1.6.2 Life-cycle for jar packaging</td>
<td>7</td>
</tr>
<tr>
<td>1.6.3 Default life-cycle</td>
<td>8</td>
</tr>
<tr>
<td>1.6.4 Ways to manipulate build life-cycle *</td>
<td>8</td>
</tr>
<tr>
<td>1.7 Multi-module software repository</td>
<td>10</td>
</tr>
<tr>
<td>1.7.1 Aggregation in multi-module projects</td>
<td>10</td>
</tr>
<tr>
<td>1.7.2 POM file of a submodule</td>
<td>10</td>
</tr>
<tr>
<td>1.7.3 Dependency inheritance</td>
<td>11</td>
</tr>
<tr>
<td>1.8 Dependencies conflict resolution</td>
<td>12</td>
</tr>
<tr>
<td>1.9 Plug-ins</td>
<td>13</td>
</tr>
<tr>
<td>1.9.1 Compilation : mvn compile</td>
<td>13</td>
</tr>
<tr>
<td>1.9.2 Unit tests : mvn test</td>
<td>13</td>
</tr>
<tr>
<td>1.9.3 Installation (mvn install) and deployment (mvn deploy)</td>
<td>14</td>
</tr>
<tr>
<td>1.9.4 Execution : mvn exec :java</td>
<td>14</td>
</tr>
<tr>
<td>1.9.5 Eclipse : mvn eclipse :eclipse</td>
<td>clean</td>
</tr>
<tr>
<td>1.10 Properties *</td>
<td>16</td>
</tr>
<tr>
<td>1.11 Other plug-ins *</td>
<td>17</td>
</tr>
<tr>
<td>1.11.1 Filtering resources : mvn process-resources *</td>
<td>17</td>
</tr>
<tr>
<td>1.11.2 Packaging : mvn package</td>
<td>17</td>
</tr>
<tr>
<td>1.12 Default life-cycle for project's web site generation *</td>
<td>19</td>
</tr>
<tr>
<td>1.12.1 Site descriptor *</td>
<td>19</td>
</tr>
<tr>
<td>1.12.2 Web site reports *</td>
<td>19</td>
</tr>
<tr>
<td>1.12.3 Helping *</td>
<td>20</td>
</tr>
<tr>
<td>1.12.4 More on plug-ins *</td>
<td>20</td>
</tr>
<tr>
<td>1.13 Maven Versus Ant and Versus Make</td>
<td>22</td>
</tr>
<tr>
<td>1.13.1 Ant beyond make-like tools</td>
<td>22</td>
</tr>
<tr>
<td>1.13.2 Maven beyond Ant</td>
<td>22</td>
</tr>
</tbody>
</table>
1 Construction de logiciel avec Maven

1.1 Main principles

- Standard life-cycles and naming conventions
- Version management and dependency management
  - groupId: Fully qualified domain name such as eu.telecom-sudparis
  - Artifact name of the form <artifactId>-<version>.<extension>
- Repositories
  - Lots of remote repositories
  - One local repository: ~/.m2/repository
    - Populated from remote ones
- No more externals or lib, and no more jars in developer’s directories
1.2 Artifact names and project relationships

- Coordinates
  - groupId: Group, company, team, organisation, project, or other group
  - artifactId: A unique identification under the groupId representing a single project
  - version: A snapshot in time of the project in question
    - Version number = major.minor.bug fix-qualifier-build number
  - packaging: Type of build life-cycle this project will encounter
  - And sometimes classifier, e.g. linux, windows, etc.

- Three types of relationships between projects
  - Dependency: This project requires another one
  - Transitive dependencies: A dependency of a dependency
    - Maven does not allow circular dependencies
  - Parent: This project inherits some attributes from another project
1.2.1 Local repository and snapshot versions

- Local repository :
  - ~/.m2/repository
    - groupId/artifactId/version/artifactId-version.packaging
    - + a POM file describing meta-data, e.g. mainly dependencies

- A snapshot in Maven is an artifact that has been prepared using the most recent sources available

- Specifying a snapshot version for a dependency = Maven looks for new versions
  - By default, on a daily basis
  - -U command line option to force for updates
  - To avoid a remote access to search for the latest snapshot :
    mvn -o clean install (-o for "offline")

- Specifying a non-snapshot version of a dependency = download that dependency only once

1.3 Life-cycle for jar packaging — simplified

1. process-resources : Copy files and filter variables
   - maven-resources-plugin
2. compile : Compile project source code
   - maven-compiler-plugin
3. process-test-resources : Copy non-source-code for unit tests
   - maven-resources-plugin
4. test-compile : Compile unit-tests source code
   - maven-compiler-pluging
5. test : Execute project unit tests
   - maven-surefire-plugging
6. package : Create the jar of the project
   - maven-jar-plugin
7. install : Install the jar into local Maven repository ~/.m2/repository
   - maven-install-plugin
8. deploy : Deploy the jar into a remote Maven repository for web access
   - maven-deploy-plugin
1.4 Build life-cycles, phases and goals

- Build life cycle: Ordered sequence of phases
  - One life cycle per package type
    - Default, jar, pom, ejb, war, ear, etc., your technology-dependent ones

- Phase: A step in a build life cycle
  - May have zero or more goals bound to it
  - Executing a phase first execute all the preceding phases
  - E.g., $ mvn install

- Goal: Maven unit of work bound to a phase
  - Bound goals are run when their phases execute

- Plug-in: May contain several goals
  - Executing a goal of a plug-in does not execute preceding phases’ goals
  - $ mvn exec:java

1.5 Standard directory structure — a less simplified view

- pom.xml: Maven POM (Project Object Model) file always at the top-level
- LICENSE.txt: License file is encouraged
- README.txt: Simple note to start with the project
- src/main/java: Source code of the project
- src/main/filters: Filters (properties files) of the project for the build
- src/main/resources: Project non-source code (resources)
- src/main/config: Configuration for compiling the sources
- src/main/assembly: Project assembly filters
- src/test/java: Source code of the unit tests
- src/test/resources: Non-source code (resources) of the unit tests
- src/test/filters: Resource filters of the unit tests
- src/site: Resources used to generate the web site
- target: Target for all generated output: classes, web site...
1.6 More details on the build process

1.6.1 Exemplification with the JAR build life-cycle

PHASES
- process-resources
  - compile
  - process-test-resources
  - test-compile
  - test

PLUGINS:GOALS
- resources:resources
- compiler:compile
- resources:testResources
- compiler:test-compile
- surefire:test
- jar:jar
- install:install
- deploy:deploy
1.6 More details on the build process

1.6.2 Life-cycle for jar packaging

1. process-resources : Copy files and filter variables
   ■ maven-resources-plugin
2. compile : Compile project source code
   ■ maven-compiler-plugin
3. process-test-resources : Copy non-source-code for unit tests
   ■ maven-resources-plugin
4. test-compile : Compile unit-tests source code
   ■ maven-compiler-plugin
5. test : Execute project unit tests
   ■ maven-surefire-pluging
6. package : Create the jar of the project
   ■ maven-jar-plugin
7. install : Install the jar into local Maven repository ~/.m2/repository
   ■ maven-install-plugin
8. deploy : Deploy the jar into a remote Maven repository for web access
   ■ maven-deploy-plugin

1.6.3 Default life-cycle

1. Validate
2. Generate-sources
3. Process-sources
4. Generate-resources
5. Process-resources
6. Compile
7. Process-classes
8. Generate-test-sources
9. Process-test-sources
10. Generate-test-resources
11. Process-test-resources
12. Test-compile
13. Test
14. Prepare-package
15. Package
16. Pre-integration-test
17. Integration-test
18. Post-integration-test
19. Verify
20. Install
21. Deploy
1.6.4 Ways to manipulate build life-cycle *

- Five ways (not studied in this presentation, but cited here for the sake of completeness)
  1. Choose project package: pom, jar, ejb, etc.
  2. Manually bind a goal to a phase in the POM
     - E.g., choose surefire plug-in for executing tests
  3. Write a plug-in and bind in the goal definition in the plug-in Java class
  4. Create a forked life-cycle and execute it in a plug-in
  5. Create your own packaging type with a custom default life-cycle
1.7 Multi-module software repository

1.7.1 Aggregation in multi-module projects

- Similar to virtual packages in GNU/Linux packaging
- Declaring a parent project = packaging pom

```xml
<helloworld/pom.xml
<project>
  <groupId>eu.it-sudparis</groupId>
  <artifactId>helloworld</artifactId>
  <packaging>pom</packaging>
  <version>0.1-SNAPSHOT</version>
  <name>Hello world example</name>
  <modules>
    <module>first-sub-module</module>
    <module>second-sub-module</module>
  </modules>
  ...
</project>
```

- mvn install copies pom.xml to ~/.m2/repository/helloworld/pom.xml

1.7.2 POM file of a submodule

1.7.3 Dependency inheritance
1.7.2 POM file of a submodule

- Declaring a parent module = inheriting parent configuration
  
  helloworld/first-sub-module/pom.xml
  
  `<project>`
  
  `<parent>`
  
  `<groupId>eu.it-sudparis</groupId>`
  
  `<artifactId>helloworld</artifactId>`
  
  `<version>0.1-SNAPSHOT</version>`
  
  `</parent>`
  
  `<packaging>jar</packaging>`
  
  `<artifactId>first-sub-module</artifactId>`
  
  `<name>The first sub-module</name>`
  
  `...`
  
  `</project>`

- `mvn help:effective-pom` displays the effective POM of a sub-module

1.7.3 Dependency inheritance

- Submodules inherit dependencies of parent modules + transitive dependencies

  1. Inherit a version of a dependency from parent
  2. Declare a version dependency and use it

     - Use the dependency management section
     - State only the preference for a version, do not affect the dependency graph

       `<dependencyManagement>`
       
       `<dependencies>`
       
       `<dependency>`
       
       `<groupId>otherorg.proj</groupId>`
       
       `<artifactId>artifactApp</artifactId>`
       
       `<version>1.0.9</version>`
       
       `</dependency>`
       
       `</dependencies>`
       
       `</dependencyManagement>`

     - Declare that the dependency’s usage, without any version number

       `<dependency>`
       
       `<groupId>otherorg.proj</groupId>`
       
       `<artifactId>artifactApp</artifactId>`
       
       `</dependency>`
1.8 Dependencies conflict resolution

- Reasons for excluding transitive dependencies
  - Current project requires an alternately named version
    \[\implies\] Two copies of a different versioned project in the classpath
  - Artifact unused
  - Artifact provided by your runtime container
  - An API which might have multiple vendors

```xml
<dependency>
  <groupId>otherorg.proj</groupId>
  <artifactId>artifactApp</artifactId>
  <version>1.0.9</version>
  <exclusions>
    <exclusion>
      <groupId>otherorg.proj</groupId>
      <artifactId>otherArtifactApp</artifactId>
    </exclusion>
  </exclusions>
</dependency>
```
1.9 Plug-ins

1.9.1 Compilation : mvn compile

- Allow JDK 5.0 source code
  `<build><plugin><plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>2.0</version>
  <configuration>
    <source>1.5</source>
    <target>1.5</target>
  </configuration>
  </plugin></plugins></build>

- Another JRE with a fork
  `<configuration>
    <executable>/usr/.../javac</executable>
    <compilerVersion>1.5</compilerVersion>
    <fork>true</fork>
  </configuration>

- Other configuration elements, e.g. compiler arguments
1.9.2 Unit tests: mvn test

- Surefire plug-in
  ```xml
  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.0</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  ```

  By default, the following tests are included
  - **/*Test.java
  - **/Test*.java
  - **/*TestCase.java

  The following tests are excluded
  - **/Abstract*Test.java
  - **/Abstract*TestCase.java

- Skip the tests using the command line option 
  `-Dmaven.test.skip=true`

1.9.3 Installation (mvn install) and deployment (mvn deploy)

- Installation = local deployment in your local repository
  - In the directory `~/.m2/repository`

- Deployment = Internet deployment
  - See for instance [http://maven.objectweb.org/maven2](http://maven.objectweb.org/maven2)
1.9.4 Execution : mvn exec :java

- Plug-in org.codehaus.mojo:exec-maven-plugin in a profile
  ```xml
  <profiles>
  <profile>
    <id>run</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <build>
      <plugins>
        <plugin>
          <groupId>org.codehaus.mojo</groupId>
          <artifactId>exec-maven-plugin</artifactId>
          <executions>
            <execution>
              <goals>
                <goal>java</goal>
              </goals>
              <phase>runtime</phase>
            </execution>
          </executions>
          <configuration>
            <mainClass>ClassWithTheMain</mainClass>
          </configuration>
        </plugin>
      </plugins>
    </build>
  </profile>
  </profiles>
  ```

- $ mvn exec:java

1.9.5 Eclipse : mvn eclipse :eclipse|clean

- Maven plug-in for Eclipse
  - From the POM file pom.xml
    - Generation of .classpath and .project files
      - $ mvn eclipse:eclipse
    - To remove .classpath and .project files
      - $ mvn eclipse:clean
  - One Eclipse project per Maven module

- There exists an Eclipse plug-in for Maven
  - http://m2eclipse.sonatype.org/sites/m2e
1.10 Properties *

- env.X : Environment variable of the shell, e.g. ${env.PATH}
- project.x : Project element value, e.g. ${project.groupId}
- settings.x : Variables in configuration file settings.xml
- Java System Properties accessible via java.lang.System.getProperties()
- x : Property sets with POM element <properties>
1.11 Other plug-ins *

1.11.2 Filtering resources: `mvn process-resources` *

Filtering resources = Parse some files to fill value to be supplied at build time

- In `pom.xml` of the resource directory
  ```xml
  <properties>
    <my.filter.value>hello</my.filter.value>
  </properties>
  <build>
    <filters>
      <filter>src/main/filters/build.properties</filter>
    </filters>
    <resources>
      <resource>
        <directory>src/main/resources</directory>
        <filtering>true</filtering>
      </resource>
    </resources>
  </build>
  ```

- In an external file (e.g., `build.properties`) or in the properties section of `pom.xml`
  - Syntax `${property name}`, e.g., `application.name = ${project.name}`

- In command line parameters (`-Dmy.prop=hello world!`)

- It is possible to prevent some files (e.g., binary files) from filtering
1.11.2 Packaging : mvn package

- Include project classes in target/classes
- Include project resources in src/main/resources
- Insert a manifest automatically generated by Maven
- Insert a pom.xml and a pom.properties files
  - To be self-describing and to utilise meta-data during project execution
- To override the manifest file, in pom.xml of the project do the following
  
  ```xml
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-jar-plugin</artifactId>
    <configuration>
      <archive>
        <manifestFile>src/meta/META-INF/MANIFEST.MF</manifestFile>
      </archive>
    </configuration>
  </plugin>
  ```
1.12 Default life-cycle for project’s web site generation

1. `mvn site`: Generate all configured project reports
   - `maven-site-plugin`

2. `site-deploy`: Deploy the generated web site to the web server path specified in the POM distribution Management section
   - `maven-site-plugin`

- Lots of web site reports: Javadoc, JXR (source code cross reference), Checkstyle/PMD (source code formatting rules), Tag list (track TODO items), Cobertura (unit test’s and execution’s coverage), Surefire Report (visual result of unit tests) Clirr (compare jars for binary compatibility), Changes, etc.

1.12.1 Site descriptor

- In file `src/site/site.xml`
  - `bannerLeft` and `bannerRight`: Appearance of the banner
  - `skin`: Skin used for the site
  - `publishedDate`: Format of the published date
  - `body/links`: Links displayed below the banner
  - `body/head`: Meta-data to be fed into the `<head>` element
  - `body/menu`: Menu items displayed in the navigation column
  - `body/${reports}`: To control which project reports are displayed in the web site
1 Construction de logiciel avec Maven

1.12 Default life-cycle for project’s web site generation

1.12.2 Web site reports

- Javadoc : API reference from Javadoc
- JXR : Source code cross reference for any Java code
- Checkstyle : Check source code against formatting rules
- PMD : Checks source code against known rules for code smells (overlap with Checkstyle)
- CPD : Checks for duplicate source code blocks (copy/paste) (part of PMD)
- Tag list : Outstanding tasks or other markers (track TODO items)
- Cobertura : Analyse code statement coverage during unit tests or code execution
  ♦ Help in identifying untested or unused source code blocks
- Surefire Report : Show the result of unit tests visually
- Clirr : Compare two versions of a jar for binary compatibility
- Changes : Produce release notes and road maps from issue tracking systems

1.12.3 Helping

- help:active-profiles lists the profiles which are currently active for the build
- help:effective-pom displays the effective POM for the current build, with the active profiles factored in
- help:effective-settings prints out the calculated settings for the project
  ♦ Any profile enhancement
  ♦ The inheritance of the global settings into the user-level settings
- help:describe describes the attributes of a plug-in
  ♦ help:describe -Dplugin=compiler
  ♦ help:describe -Dplugin=compiler -Dfull
    ▶ For discovering all of the goals of a plug-in as well as their parameters
1.12.4 More on plug-ins *

- The Maven command `mvn aname:goal` executes the plug-in named:
  - Either `maven-aname-plugin` or `aname-maven-plugin`

- Plug-ins are configured in POM files under the `<build>` element:

  1. Configuring a plug-in or a goal already in the life-cycle:
     ```xml
     <build><plugins><plugin>
     <groupId>org.apache.maven.plugins</groupId>
     <artifactId>maven-jar-plugin</artifactId>
     <configuration>
     ...use properties declared by the plug-in...
     </configuration>
     </plugins></build>
     ```

  2. Plug-in or goal not in the life-cycle, e.g. `mvn antrun:run`:
     ```xml
     <build><plugins><plugin>
     <artifactId>maven-antrun-plugin</artifactId>
     <configuration><tasks><echo>HOME: $env.JAVA_HOME</echo></tasks></configuration>
     </plugins></build>
     ```

  3. The same as before, but bound to a phase: `<execution><goal>elements`

  4. The same as before + conditional execution
1.13 Maven Versus Ant and Versus Make

1.13.1 Ant beyond make-like tools

- **MAKE** was a marked improvement over shell script
  - Abstraction of specific shell commands into a more general and consistent syntax
  - Determines automatically which pieces of a large program need to be rebuilt and issues the commands

- But...
  - **MAKE** mainly/historically targets the language C et C++
    - Implicit rules and pattern rules
      - Execute make -p in a directory with no Makefile
      - Default suffixes: .out .a .ln .o .c .cc .C .cpp .p .f .F .r .y .l .s .S .mod .sym .def .h .info .dvi .tex .texinfo .txi .txinfo, etc.
  - **MAKE** mainly/historically targets Unix-like operating systems
    - Machine-specific aspects of Make $\Rightarrow$ less portable than the code to build

- Thus...
  - **ANT** builds software across multiple platforms with plug-ins for Java
  - **ANT** is natively-integrated to Eclipse IDE
1.13.2 Maven beyond Ant

- Encompass build activities: compilation, documentation, distribution, deployment
  - Declarative model for software projects: Project Object Model (POM)
    - Reuse of build logic and default strategies for the most common tasks
  - Tools/plug-ins that interact with the declarative model
    - There already exists a very long list of such plug-ins
    - Access to all the Ant tasks through the plug-ins maven-ant-plugin and maven-antrun-plugin
    - You can provide your own plugins for your own technology
  - Organisation of dependencies
    - Management the of JAR hell
    - Management the of plug-in hell

- Written in Java, dedicated for writing in Java
- Maven is integrated to IDEs like Eclipse
- This presentation is for Maven 2
  - New functionalities in Maven 3: parallel build for multicores, decoupling, etc.