Maven

università di ferrara

DA SEICENTO ANNI GUARDIAMO AVANTI.

Topics covered

- Introduction to Maven
- Maven for Dependency Management
- Maven Lifecycles and Plugins
- Hands on session

Introduction to Maven

What is Maven?

- A Java project management and integration build tool.
- Based on the concept of XML Project Object Model (POM).
- Maven can manage a project's build, testing, reporting, documentation and releases from a central piece of information
- Licensed by Apache
- Stores libraries and plugins in a central repository

Apache maven

Maven's Objectives

- Making the build process easy
- Providing a uniform build system
- Providing quality project information
  - Maven provides plenty of useful project information that is in part taken from your POM and in part generated from your project's sources
- Providing guidelines for best practices development
  - Maven aims to gather current principles for best practices development, and make it easy to guide a project in that direction. For example, specification, execution, and reporting of unit tests are part of the normal build cycle using Maven.
- Allowing transparent migration to new features
  - Maven provides an easy way for the installation of new or updated plugins

How does it work ?

- Build controlled via pom.xml project file
- POM = Project Object Model
- Uses standard build order, directories, plugins
- Identifies dependencies in the pom.xml

Maven for Dependency Management

Maven repositories

- In Maven terminology, a **repository** is a directory where all the project jars, library jar, plugins or any other project specific artifacts are stored and can be used by Maven easily.
- Maven repository are of three types:
  - **local**, Maven local repository is a folder location on your machine. It gets created when you run any maven command for the first time.
  - **central**, Maven central repository is repository provided by Maven community: https://search.maven.org/
  - **remote**, developer's own custom repository containing required libraries or other project jars
- Remote repository are defined within tag `<repositories>`

Handling Dependencies with Maven

- Unless your project is small, your project may need external Java APIs or frameworks which are usually packaged in their own JAR or WAR files. These JAR files are needed on the classpath when you compile your project code.
- Dependency management is one of the features of Maven that is best known.
- In the POM file you specify what external libraries your project depends on, and which version, and then Maven downloads them for you and puts them in your local Maven repository.
  - These external libraries are called **dependencies**.

Transitive dependencies

- If any of these external libraries need other libraries, then these other libraries are also downloaded into your local Maven repository.
- This feature allows you to avoid needing to discover and specify the dependencies that your own dependencies require, and including them automatically.
  - E.g. If my project depends on A and A depends on B. I don’t have to specify B in my POM file
- **Nearest definition**. The version used will be the closest one to your project in the tree of dependencies.
  - e.g. if dependencies for A, B, and C are defined as A -> B -> C -> D 2.0 and A -> E -> D 1.0, then D 1.0 will be used when building A because the path from A to D through E is shorter.
  - You could explicitly add a dependency to D 2.0 in A to force the use of D 2.0

Scope of a Dependency

- Dependency scope is used to limit the transitivity of a dependency, and also **to affect the classpath used for various build tasks**.
- There are 6 scopes available:
  - **compile**, default scope. Compile dependencies are available in all classpaths of a project. Those dependencies are propagated to dependent projects.
  - **provided**, similar to compile, but indicates you expect the JDK or a container to provide the dependency at runtime. This scope is only available on the compilation and test classpath, and is not transitive. In other words, it means that the JAR is added in the classpath by Maven during compilation, but at run time there is already a JAR *provided* by the environment.
  - **runtime**, indicates that the dependency is not required for compilation, but is for execution. It is in the runtime and test classpaths, but not the compile classpath.
  - **test**, indicates that the dependency is required only for the test compilation and execution phases. This scope is not transitive.
  - **system**
  - **import**

Adding a Dependency

- The dependency must be enclosed by the element **`<dependency>`** and added into the element **`<dependencies>`**
- We must provide
  - groupId
  - artifactId
  - version
  - type (default jar)
  - scope (default compile)

Adding a Dependency - Example

- E.g.

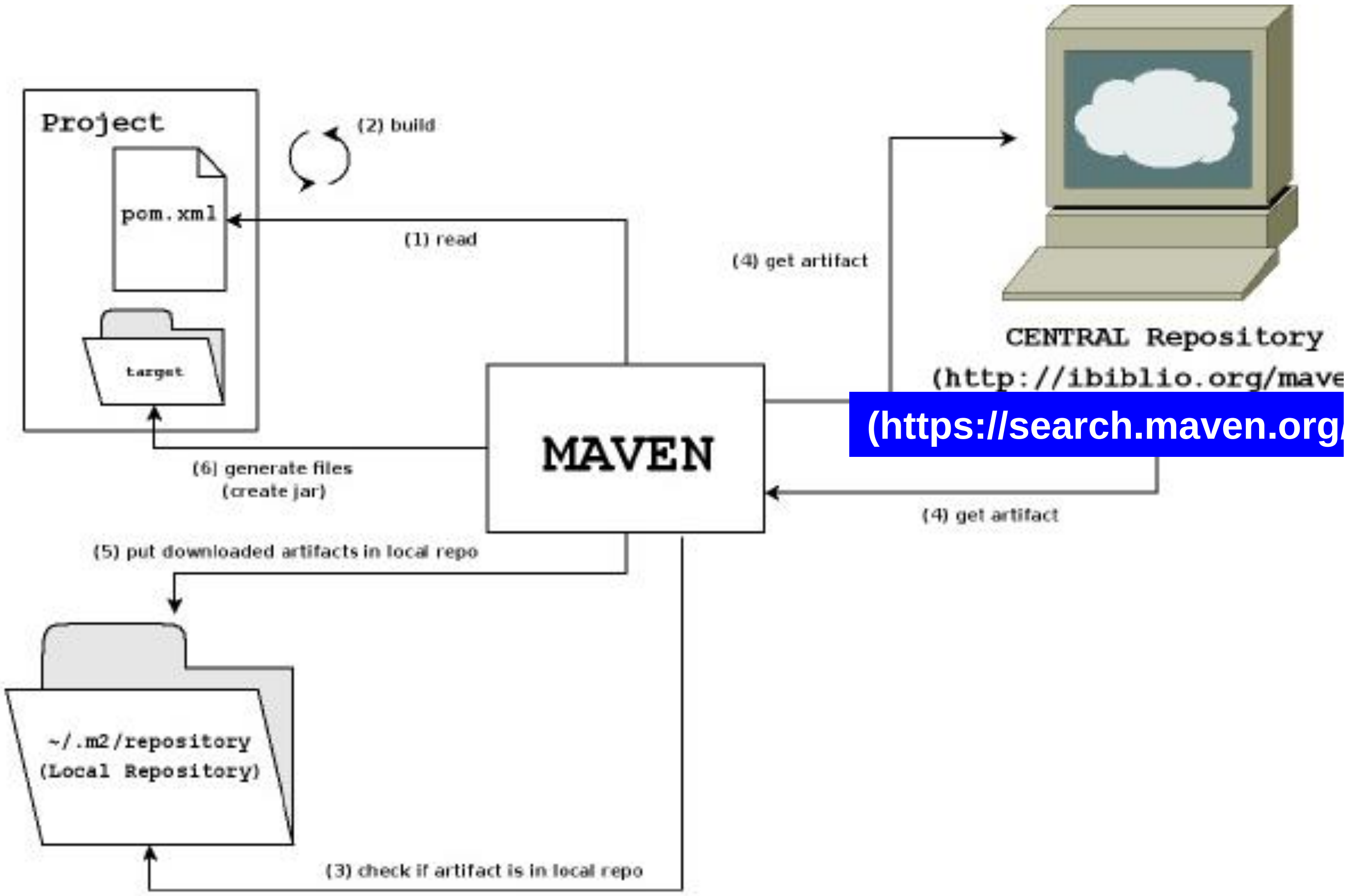
```
<project>
  ...
  <dependencies>
    <dependency>
      <groupId>group-c</groupId>
      <artifactId>artifact-b</artifactId>
      <version>1.0</version>
      <type>war</type>
      <scope>runtime</scope>
    </dependency>
…
</project>
```

- We are adding the dependency group-c:artifact-b:1.0 with scope runtime and the dependency is packaged as war.

Maven dependency mechanism (How Maven finds the dependencies)

1. Search the dependency in Maven local repository (~/.m2/ in Linux)
2. Search the dependency in Maven central repository (https://search.maven.org)
3. Search the dependency among Maven remote repositories (if defined in pom.xml)

Maven dependency mechanism



Maven dependency mechanism - Example

- Aim: we need the log4j library
- We need to know the log4j Maven coordinates, for example

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

- We declare the log4j Maven coordinates into pom.xml file. We enclose the coordinates within the `<dependency>` element and in turn within the `<dependencies>` element
- When Maven is compiling or building, the log4j jar will be downloaded automatically and put it into your Maven local repository (if not already in the local repository)
- All managed by Maven
- The Maven coordinates can be found by visiting the Maven Central Repository

Maven Lifecycles and Plugins

Maven Build Lifecycles

- Maven is based around the central concept of a build lifecycle. The process for building and distributing a particular artifact (project) is clearly defined.
- The POM file contains all the information for the build lifecycle
- The user have to learn a small set of commands
- There are three built-in lifecycles:
  - **clean**, handles project cleaning
  - **default**, handles the project building, testing and deployment
  - **site**, handles the creation of project's site documentation.
- A Maven “build lifecycle” is defined by a list of build **phases**
  - **a build phase represents a stage in the lifecycle**
- **The lifecycle phases are executed sequentially**

Clean Lifecycle

- **pre-clean**, execute processes needed prior to the actual project cleaning
- **clean,** remove all files generated by the previous build
- **post-clean,** execute processes needed to finalize the project cleaning

Default Lifecycle

- **The defaule lifecycle has**
- **validate**, validate the project is correct and all necessary information is available
- **compile**, compile the source code of the project
- **test**, test the compiled source code using a suitable unit testing framework.
- **package**, take the compiled code and package it in its distributable format, such as a JAR.
- **verify**, run integration tests
- **install**, install the package into the local repository, for use as a dependency in other projects locally
- **deploy**, copies the final package to the remote repository for sharing with other developers and projects.

Site Lifecycle

- **pre-site**, execute processes needed prior to the actual project site generation
- **site**, generate the project's site documentation
- **post-site**, execute processes needed to finalize the site generation, and to prepare for site deployment
- **site-deploy**, deploy the generated site documentation to the specified web server

università di ferrara
DA SEICENTO ANNI GUARDIAMO AVANTI.

Maven main commands

- To execute a phase on Maven you can use the command **mvn**

  mvn <phase_name>

- It executes all the previous phases in a lifecycle, before executing the one specified
- Example:

  mvn install

  - This command executes each default life cycle phase in order (validate, compile, test, package, verify), before executing install

Maven Plugins

- Maven is - at its heart - a plugin execution framework; all work is done by plugins.
- Plugins are artifacts that provide goals to Maven.
- **A plugin may have one or more goals.**
  - Each goal represents a capability of that plugin.
- There are two types of plugins:
  - **Build plugins** will be executed during the build and they should be configured in the <build> element from the POM.
  - **Reporting plugins** will be executed during the site generation and they should be configured in the <reporting> element from the POM.

Maven Goals

- A goal is a “unit of work” in Maven. It is possible to execute goals independently or a part of a larger chain of goals.
- A goal can be executed independently using the following syntax:

  ```
  mvn [plugin-name]:[goal-name]
  ```
- You can add goals to lifecycle phases by configuring more Maven plugins and adding them to a life cycle in your POM file.
  - Plugins can contain information that indicates which lifecycle phase to bind a goal to. Note that adding the plugin on its own is not enough information - you need to specify which goal should be executed.
  - If the plugin does not specify the default life cycle it should run, you must also specify the life cycle phase it should run.

Add Goals to Phases - Example

```
<build>
    <plugins>
        <plugin>
           <groupId>org.apache.maven.plugins</groupId>
           <artifactId>maven-javadoc-plugin</artifactId>
           <version>2.10.4</version>
           <executions>
             <execution>
               <phase>site</phase>
               <goals>
                 <goal>javadoc</goal>
               </goals>
             </execution>
           </executions>
         </plugin>
    …
    </plugins>
</build>
```

università di ferrara
DA SEICENTO ANNI GUARDIAMO AVANTI.

References

- Maven official site
  - https://maven.apache.org/
- Maven in 5 minutes
  - https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html
- Getting started
  - https://maven.apache.org/guides/getting-started/index.html

Hands on Session

Install Maven

- Linux (Debian)
  - Command: sudo apt-get install maven
- Linux (Fedora)
  - Command: sudo yum install maven
- Mac
  - https://maven.apache.org/install.html
- Windows
  - https://maven.apache.org/install.html

What we are going to do (a)

- Create a maven project with NetBeans
- Install the JAR of our project into the Maven local repository

```
mvn install
```

- Add a dependecy (log4j)
- Produce the documentation of our project

```
mvn site
```

- Install again the JAR of our project into the Maven local repository

```
mvn install
```

- **Using Git!**

What we are going to do (b)

- Create a test (add JUnit dependency)
- Add the JaCoCo plugin:

```
<plugin>
     <groupId>org.jacoco</groupId>
     <artifactId>jacoco-maven-plugin</artifactId>
     <version>0.7.9</version>
     <executions>
          <execution>
               <goals>
                    <goal>prepare-agent</goal>
               </goals>
          </execution>
          <execution>
               <id>report</id>
               <phase>test</phase>
               <goals>
                    <goal>report</goal>
               </goals>
          </execution>
     </executions>
 </plugin>
```

università di ferrara
DA SEICENTO ANNI GUARDIAMO AVANTI.