

Maven per gestire il ciclo di vita di un'applicazione

Ingegneria del Software
A.A. 2020/2021

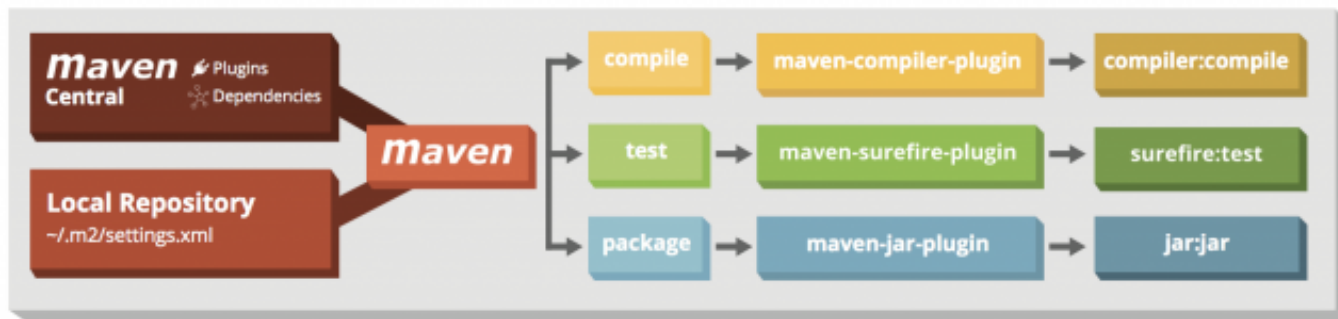
Prof. Andrea Fornaia

Compilare un progetto Java

- Un progetto Java ha spesso diverse **dipendenze** (librerie in formato .jar), e questo può rendere la compilazione difficoltosa:
 - i .jar delle librerie in una data versione devono essere **reperiti e scaricati** in locale
 - il **CLASSPATH** deve essere modificato
 - il progetto può essere composto da **più moduli dipendenti** tra loro da compilare nel giusto ordine
 - La **lista delle dipendenze** dovrebbe essere condiviso assieme al codice sorgente
- La creazione e la gestione di un progetto dovrebbe essere **indipendente dall'IDE**:
 - es. dovrei essere in grado di creare un progetto con Eclipse ed aprirlo facilmente con IntelliJ e viceversa
- La compilazione di un'eseguibile (artefatto) richiede varie fasi da **automatizzare** quanto più possibile (pipeline):
 - compilazione, esecuzione test, creazione jar eseguibile...
- Strumenti utili di **reportistica** dovrebbero poter essere aggiunti facilmente alla pipeline di compilazione:
 - Generazione automatica della documentazione
 - Misura della copertura del codice da parte dei test
- Dovrebbe essere facile **condividere versioni aggiornate** ed eseguibili della nostra applicazione:
 - quando viene creata una nuova release, l'artefatto corrispondente (jar) dovrebbe essere caricato in un repository, permettendo così ad altre applicazioni, o sistemi, di ottenere facilmente la versione aggiornata

Maven

- È un Software Project Management Tool per Java
- Tipicamente detto semplicemente Build Tool
 - Automatizzazione:
 - compilazione (con **gestione dipendenze** tramite file di configurazione)
 - testing (es. eseguiti prima del packaging)
 - packaging (es. creazione jar)
 - altro: (installazione in un repository locale, deploy in un repository remoto...)
 - Descrizione basata su file XML (pom.xml)
 - Basato su plugin per estenderne le funzionalità (es. deploy usando docker)
 - Utilizza repository *remoto* e *locale* (**.m2**) per fornire le dipendenze
 - Condivido il file di configurazione (pom.xml) non le librerie (jar)



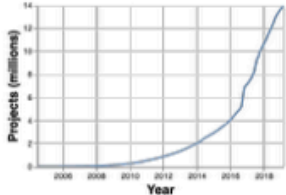
Maven Central Repository

MVNREPOSITORY

Search for groups, artifacts, categories

Search


Indexed Artifacts (17.6M)



Popular Categories

[Aspect Oriented](#)
[Actor Frameworks](#)
[Application Metrics](#)
[Build Tools](#)
[Bytecode Libraries](#)
[Command Line Parsers](#)
[Cache Implementations](#)
[Cloud Computing](#)
[Code Analyzers](#)
[Collections](#)
[Configuration Libraries](#)
[Core Utilities](#)
[Date and Time Utilities](#)
[Dependency Injection](#)
[Embedded SQL Databases](#)
[HTML Parsers](#)
[HTTP Clients](#)

Home » [junit](#) » [junit](#) » **4.11**

**JUnit » 4.11**
JUnit is a unit testing framework for Java, created by Erich Gamma and Kent Beck.

License	CPAL 1.0 CPL 1.0
Categories	Testing Frameworks
Organization	JUnit
HomePage	http://junit.org
Date	(Nov 14, 2012)
Files	pom (2 KB) jar (239 KB) View All
Repositories	Central AdobePublic Aspose Geomajas Redhat GA Sonatype
Used By	101,013 artifacts

Note: There is a new version for this artifact

New Version	4.13
--------------------	----------------------

[Maven](#) [Gradle](#) [SBT](#) [Ivy](#) [Grape](#) [Leiningen](#) [Buildr](#)

```
<!-- https://mvnrepository.com/artifact/junit/junit -->
<dependency>
  <groupId>junit</groupId>
  <artifactId>junit</artifactId>
  <version>4.11</version>
  <scope>test</scope>
</dependency>
```

Creare un progetto

Controllare se maven è installato:

```
$ mvn --version
```

Creare un progetto da un template (**archetype**) mod. interattiva:

```
$ mvn archetype:generate
```

Creare un progetto da un template (**archetype**) mod. semi-interattiva

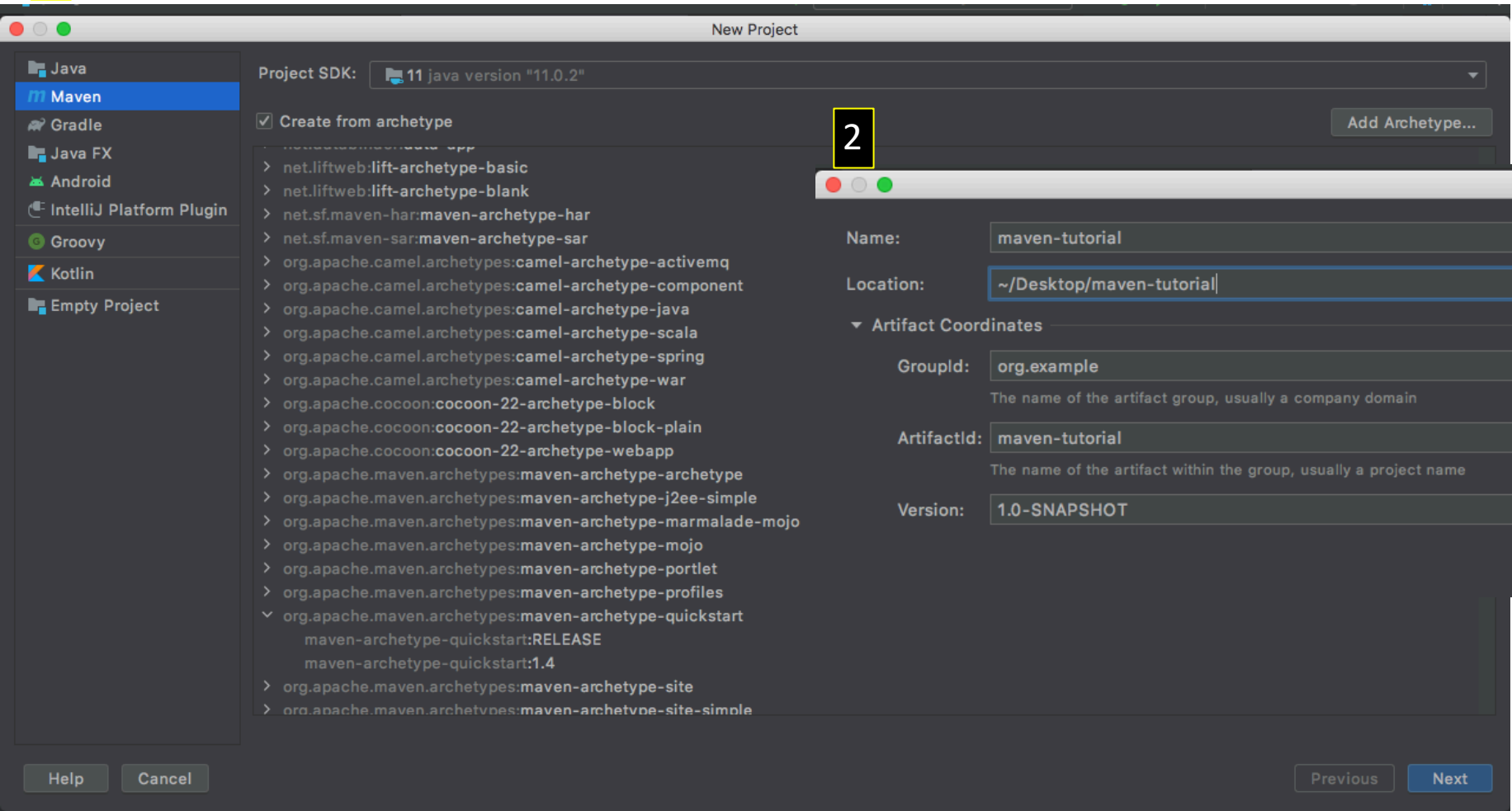
```
$ mvn archetype:generate  
  -DarchetypeArtifactId=maven-archetype-quickstart  
  -DarchetypeVersion=1.4
```

Creare un progetto da un template (**archetype**) mod. non interattiva

```
$ mvn archetype:generate  
  -DarchetypeArtifactId=maven-archetype-quickstart  
  -DarchetypeVersion=1.4  
  -DgroupId=com.mycompany.app  
  -DartifactId=my-app  
  -DinteractiveMode=false
```

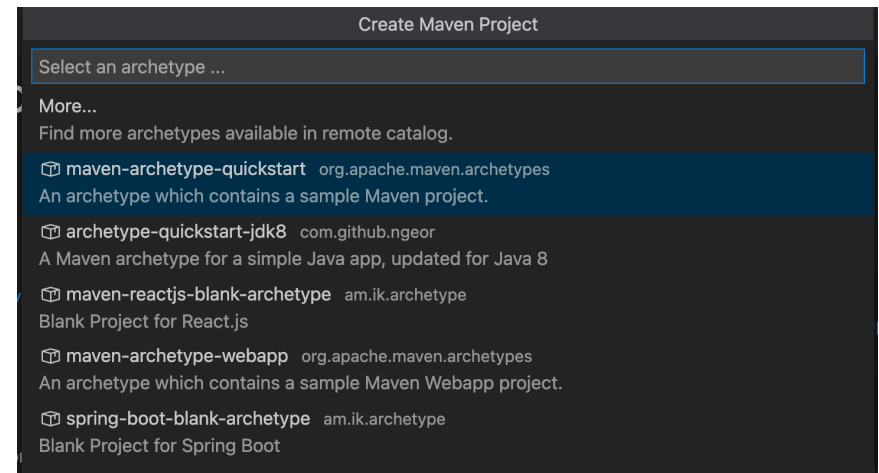
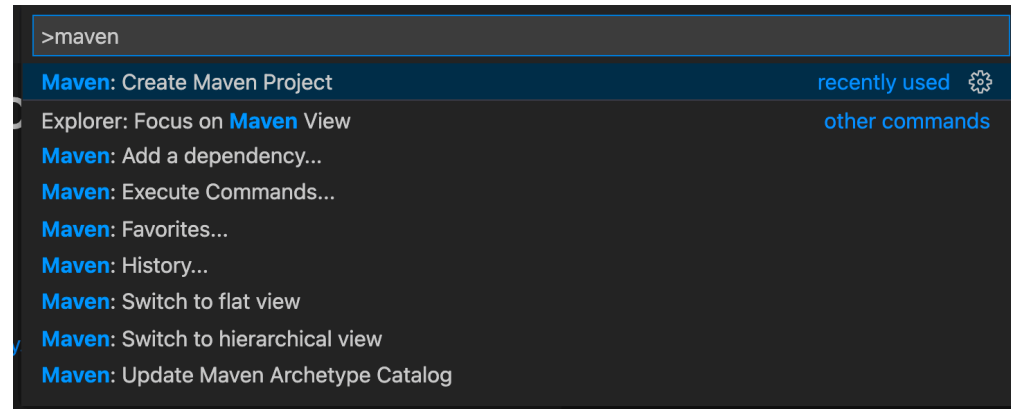
Oppure tramite IDE (es. IntelliJ)

1

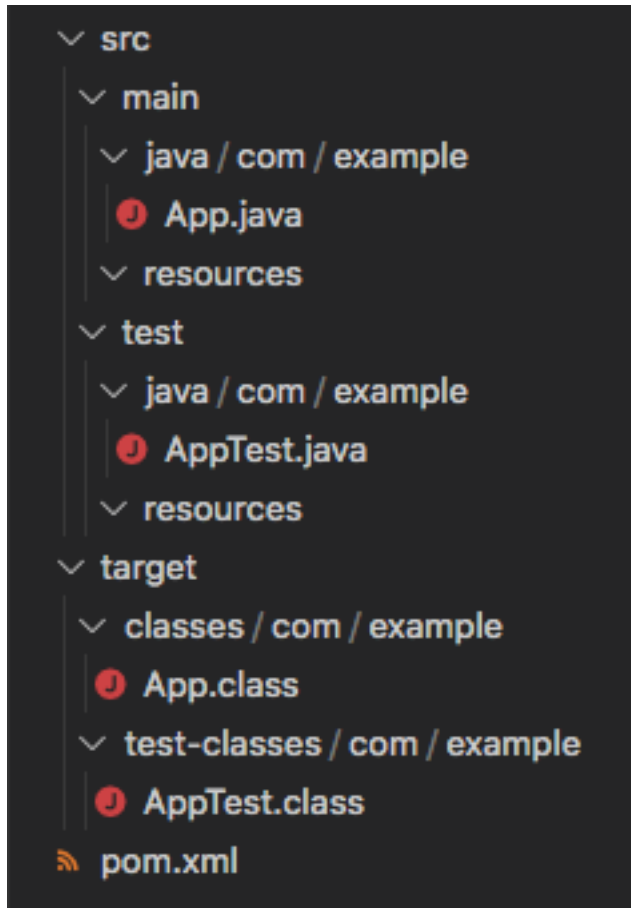


O tramite VS Code (useremo questo)

- CTRL+SHIFT+P (Windows/Linux)
- CMD+SHIFT+P (Mac)
- Scegliere di creare un progetto maven
- Scegliere il template (archetype)
- Scegliere la versione (l'ultima)
- Scegliere il nome del package radice (es. com.example)
- Scegliere la cartella dove creare il progetto
- Nel terminale integrato di VS Code, confermare le scelte presentate durante la generazione dei file



Struttura standardizzata generata automaticamente



- **src:** contiene tutti sorgenti e le risorse
 - **main:** per l'applicazione
 - **java:** sorgenti (packages)
 - **resources:** risorse, es. file .properties
 - **test:** per i test
 - **java:** sorgenti
 - **resources:** risorse per il testing
- **target:** creata dopo la compilazione
 - **classes:** risultato compilazione file in main
 - **test-classes:** risultato compilazione file in test
 - **app-1.0-SNAPSHOT.jar** (dopo \$ mvn package)
- **pom.xml:** descrive dettagli del progetto (es. dipendenze, compilazione, testing)
- Aggiungete la linea `./target/` nel **.gitignore** per ignorare la cartella target e il suo contenuto

pom.xml generato

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

  <groupId>com.example</groupId>
  <artifactId>my-app</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>my-app</name>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>1.8</maven.compiler.source>
    <maven.compiler.target>1.8</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>

</project>
```

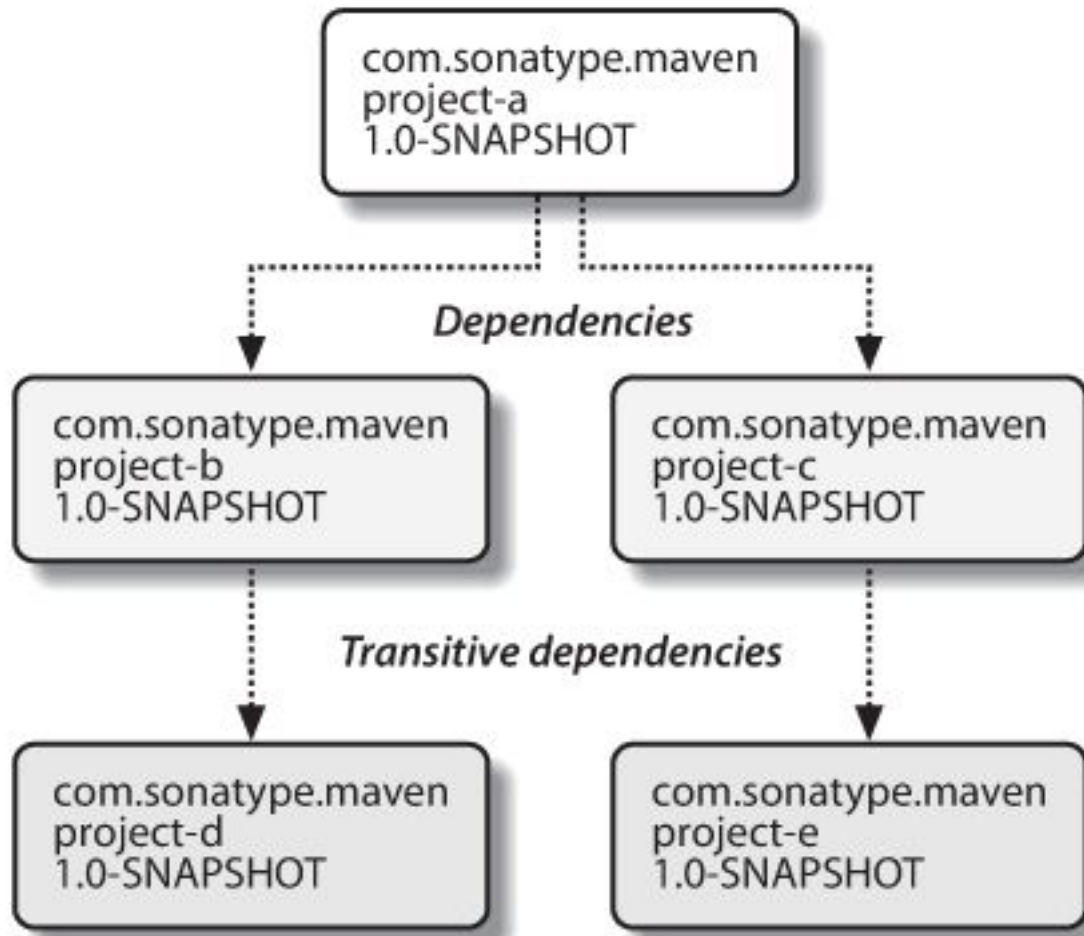
Esempio Dipendenza: JUnit

```
<dependencies>
  <dependency>
    <groupId>junit</groupId>
    <artifactId>junit</artifactId>
    <version>4.11</version>
    <scope>test</scope>
  </dependency>
</dependencies>
```

Repository locale

```
$ cd .m2/repository/junit/junit/
3.8.1/      4.10/      4.12/      4.13/      4.8.2/
3.8.2/      4.11/      4.12-beta-3/ 4.8.1/
$ cd .m2/repository/junit/junit/4.11/
_remote.repositories  junit-4.11-sources.jar.sha1  junit-4.11.pom.sha1
junit-4.11-javadoc.jar  junit-4.11.jar              m2e-lastUpdated.properties
junit-4.11-javadoc.jar.sha1  junit-4.11.jar.sha1
junit-4.11-sources.jar      junit-4.11.pom
```

Maven risolve le dipendenze in maniera transitiva

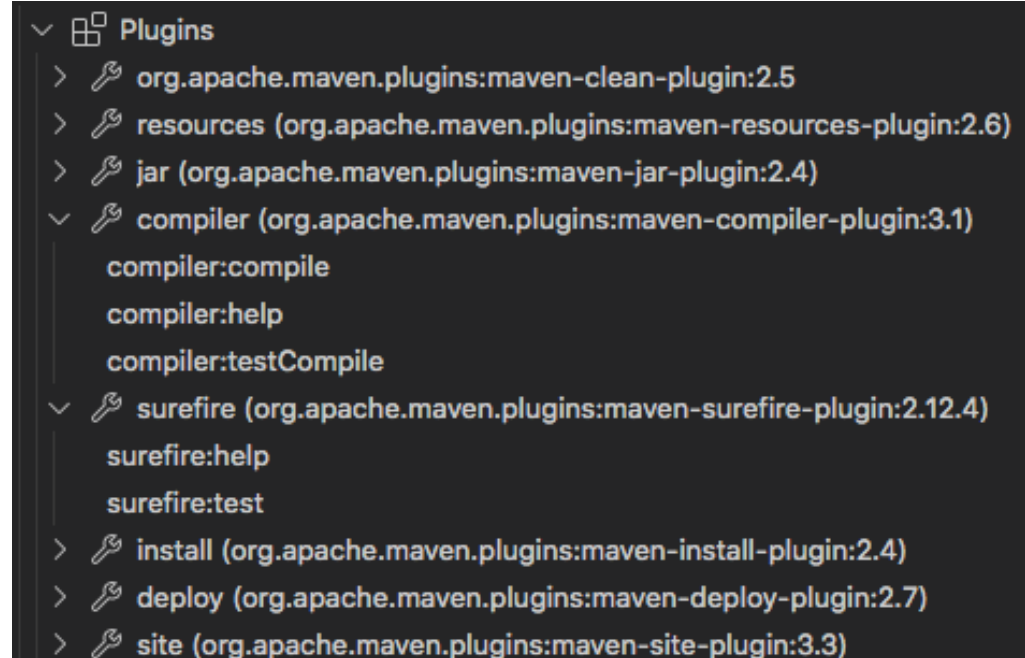


Maven Plugin

- Un plugin è un insieme di uno o più goal
- Un goal è un task eseguibile (MOJO: Maven plain Old Java Object):
 - da command line (mvn plugin-prefix:goal)
 - in automatico, legandolo ad una fase del lifecycle
- Tutti i progetti maven hanno dei plugin di default (core) e vengono messi a disposizione implicitamente dal *Super POM*
- Spesso i goal hanno una **fase di default** a cui legarsi (es. *surefire:test* si lega alla fase *test*)
- La fase a cui legarsi può essere ridefinita nel pom.xml

Lista plugin del Maven Project:
<https://maven.apache.org/plugins/index.html>

(accanto: tab di VS Code)



Esempio di plugin- prefix:goal

\$ mvn dependency:tree

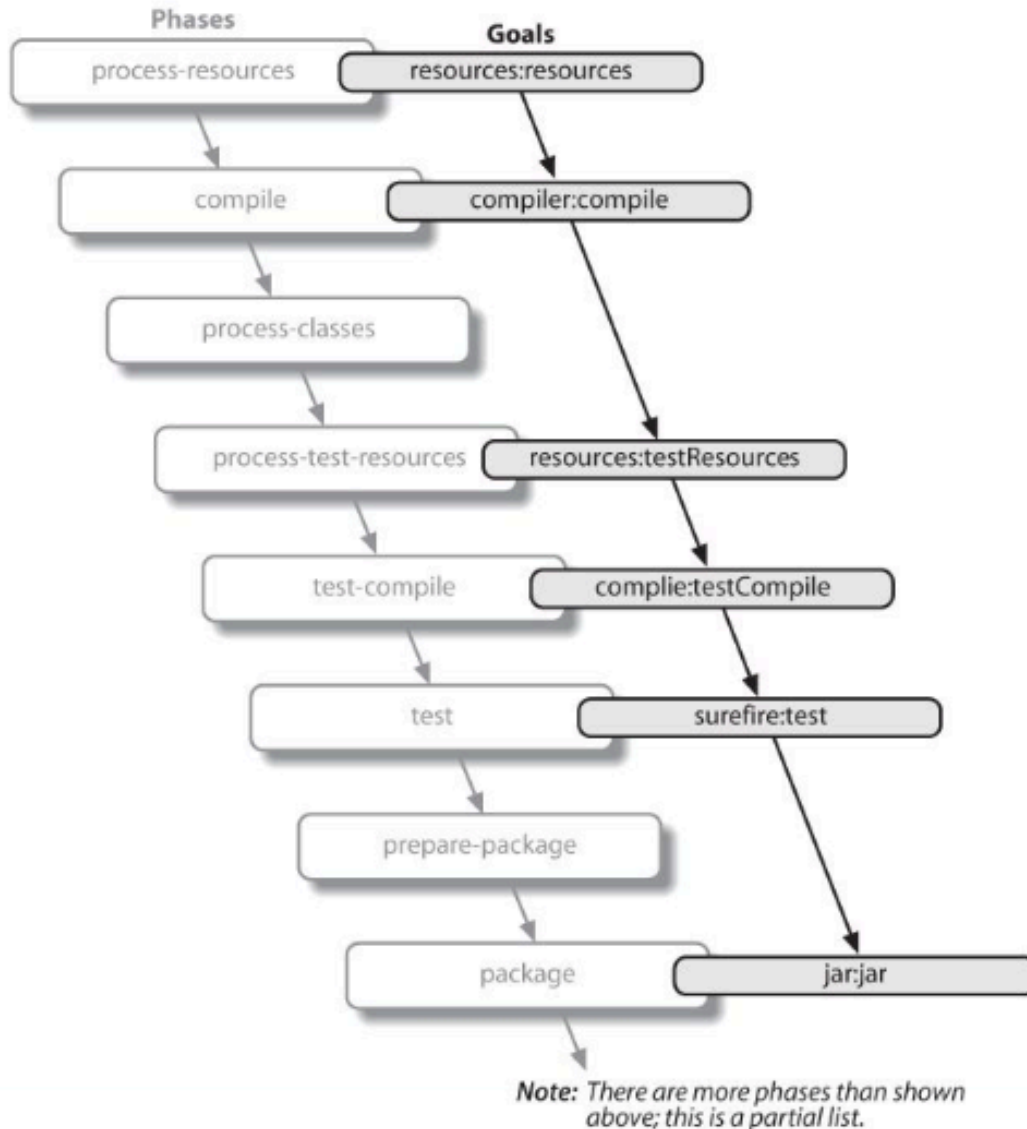


- **dependency** è il prefix del plugin
- **tree** è il goal messo a disposizione dal plugin

*nota: è un progetto
Spring Boot “vuoto”...*

```
--- maven-dependency-plugin:3.1.1:tree (default-cli) @ alien ---
com.space.scanner:alien:jar:0.0.1-SNAPSHOT
+- org.springframework.boot:spring-boot-starter-web:jar:2.1.2.RELEASE:compile
| +- org.springframework.boot:spring-boot-starter:jar:2.1.2.RELEASE:compile
| | +- org.springframework.boot:spring-boot:jar:2.1.2.RELEASE:compile
| | +- org.springframework.boot:spring-boot-autoconfigure:jar:2.1.2.RELEASE:compile
| | +- org.springframework.boot:spring-boot-starter-logging:jar:2.1.2.RELEASE:compile
| | | +- ch.qos.logback:logback-classic:jar:1.2.3:compile
| | | | \- ch.qos.logback:logback-core:jar:1.2.3:compile
| | +- org.apache.logging.log4j:log4j-to-slf4j:jar:2.11.1:compile
| | | \- org.apache.logging.log4j:log4j-api:jar:2.11.1:compile
| | \- org.slf4j:jul-to-slf4j:jar:1.7.25:compile
| +- javax.annotation:javax.annotation-api:jar:1.3.2:compile
| \- org.yaml:snakeyaml:jar:1.23:runtime
+- org.springframework.boot:spring-boot-starter-json:jar:2.1.2.RELEASE:compile
| +- com.fasterxml.jackson.core:jackson-databind:jar:2.9.8:compile
| | +- com.fasterxml.jackson.core:jackson-annotations:jar:2.9.0:compile
| | \- com.fasterxml.jackson.core:jackson-core:jar:2.9.8:compile
| +- com.fasterxml.jackson.datatype:jackson-datatype-jdk8:jar:2.9.8:compile
| +- com.fasterxml.jackson.datatype:jackson-datatype-jsr310:jar:2.9.8:compile
| \- com.fasterxml.jackson.module:jackson-module-parameter-names:jar:2.9.8:compile
+- org.springframework.boot:spring-boot-starter-tomcat:jar:2.1.2.RELEASE:compile
| +- org.apache.tomcat.embed:tomcat-embed-core:jar:9.0.14:compile
| +- org.apache.tomcat.embed:tomcat-embed-el:jar:9.0.14:compile
| \- org.apache.tomcat.embed:tomcat-embed-websocket:jar:9.0.14:compile
+- org.hibernate.validator:hibernate-validator:jar:6.0.14.Final:compile
| +- javax.validation:validation-api:jar:2.0.1.Final:compile
| +- org.jboss.logging:jboss-logging:jar:3.3.2.Final:compile
| \- com.fasterxml:classmate:jar:1.4.0:compile
+- org.springframework:spring-web:jar:5.1.4.RELEASE:compile
| \- org.springframework:spring-beans:jar:5.1.4.RELEASE:compile
|- org.springframework:spring-webmvc:jar:5.1.4.RELEASE:compile
| +- org.springframework:spring-aop:jar:5.1.4.RELEASE:compile
| +- org.springframework:spring-context:jar:5.1.4.RELEASE:compile
| \- org.springframework:spring-expression:jar:5.1.4.RELEASE:compile
\-- org.springframework.boot:spring-boot-starter-test:jar:2.1.2.RELEASE:test
| +- org.springframework.boot:spring-boot-test:jar:2.1.2.RELEASE:test
| +- org.springframework.boot:spring-boot-test-autoconfigure:jar:2.1.2.RELEASE:test
| +- com.jayway.jsonpath:json-path:jar:2.4.0:test
| | +- net.minidev:json-smart:jar:2.3:test
| | | \- net.minidev:accessors-smart:jar:1.2:test
| | | | \- org.ow2.asm:asm:jar:5.0.4:test
| | \- org.slf4j:slf4j-api:jar:1.7.25:compile
| +- junit:junit:jar:4.12:test
| +- org.assertj:assertj-core:jar:3.11.1:test
| +- org.mockito:mockito-core:jar:2.23.4:test
| | +- net.bytebuddy:byte-buddy:jar:1.9.7:test
| | +- net.bytebuddy:byte-buddy-agent:jar:1.9.7:test
| | \- org.objenesis:objenesis:jar:2.6:test
| +- org.hamcrest:hamcrest-core:jar:1.3:test
| +- org.hamcrest:hamcrest-library:jar:1.3:test
| +- org.skyscreamer:jsonassert:jar:1.5.0:test
| | \- com.vaadin.external.google:android-json:jar:0.0.20131108.vaadin1:test
| +- org.springframework:spring-core:jar:5.1.4.RELEASE:compile
| \- org.springframework:spring-jcl:jar:5.1.4.RELEASE:compile
+- org.springframework:spring-test:jar:5.1.4.RELEASE:test
\-- org.xmlunit:xmlunit-core:jar:2.6.2:test
```

Build lifecycle: fasi e goal



- Un lifecycle è una sequenza di fasi
- Ad una fase posso legare (bind) dei goal
- Quando viene richiesta l'esecuzione di una fase (es. *mvn test*) vengono **eseguite in automatico tutte le fasi precedenti nel lifecycle** (es. *compile*)
- Eseguendo una fase vengono eseguiti tutti i goal legati a quella fase (es. *surefire:test*)
- Se invece di una fase richiedo l'esecuzione esplicita di un goal (es. *surefire:test*) viene eseguito solo quel goal (nota: può fallire se non ci sono le classi compilate)

Mvn lifecycle

Clean lifecycle:

- **pre-clean** — operazioni da eseguire prima del clean
- **clean** — elimina la cartella target (risultato della compilazione precedente)
- **post-clean** — operazioni da eseguire dopo il clean

Default lifecycle (build) – riportati solo i più importanti

- **validate** — valida il pom.xml
- **initialize** — es. caricamento agenti nella JVM per l'instrumentazione delle classi a runtime (code coverage)
- **compile** — compila il codice in src, le classi vengono salvate in target/classes
- **test** — esegue i test
- **package** — prende il codice compilato e crea un package di distribuzione (JAR, WAR)
- **verify** — esegue i controlli di verifica sul package, es. controlli di qualità
- **install** — installa (copia) il package (artifact) nel repository locale (\$HOME/.m2)
- **deploy** — copia il package (artifact) nel repository remoto, se configurato (artifacts)

Site lifecycle – riportati solo i più importanti

- **site** — generazione documentazione (es. Javadoc) e reportistica (es. code coverage)
- **site-deploy** — deploy della documentazione su un web server specificato

Esempi di comandi

\$ mvn clean

\$ mvn compile

\$ mvn test

\$ mvn test -Dtest=TestClass#testMethod

\$ mvn package

\$ mvn install

\$ mvn deploy

\$ mvn site

\$ mvn clean dependency:copy-dependencies package

Aggiungere Plugin

- I plugin aggiungono funzionalità richiamabili da maven durante l'esecuzione la gestione del lifecycle
- I plugin aggiuntivi sono a tutti gli effetti delle dipendenze che devono essere specificate all'interno del pom.xml
 - es. per aggiungere un plugin appartenente al lifecycle di build bisogna aggiungere la dipendenza sotto *project.build.plugins*
- L'aggiunta del plugin mette a disposizione i goal richiamabili singolarmente, ma questi goal non vengono agganciati automaticamente al lifecycle
- Il plugin deve infatti essere configurato (sotto *executions*) per indicare quali dei goal aggiungere durante l'esecuzione del lifecycle
- I goal selezionati si agganceranno in automatico ad una fase di default prevista dallo sviluppatore del plugin (es. test) ma questa configurazione può essere ridefinita (sotto *phase*)

pom.xml (esteso) con aggiunta di un plugin

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3   xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
4   <modelVersion>4.0.0</modelVersion>
5   <parent>
6     <groupId>org.springframework.boot</groupId>
7     <artifactId>spring-boot-starter-parent</artifactId>
8     <version>2.1.2.RELEASE</version>
9     <relativePath/> <!-- lookup parent from repository -->
10  </parent>
11  <groupId>com.space.scanner</groupId>
12  <artifactId>alien</artifactId>
13  <version>0.0.1-SNAPSHOT</version>
14  <name>alien</name>
15  <description>Space Scanner Alien Service</description>
16
17  <properties>
18    <java.version>1.8</java.version>
19  </properties>
20
21  <dependencies>
22    <dependency>
23      <groupId>org.springframework.boot</groupId>
24      <artifactId>spring-boot-starter-web</artifactId>
25    </dependency>
26
27    <dependency>
28      <groupId>org.springframework.boot</groupId>
29      <artifactId>spring-boot-starter-test</artifactId>
30      <scope>test</scope>
31    </dependency>
32  </dependencies>
33
34  <build>
35    <plugins>
36      <plugin>
37        <groupId>org.springframework.boot</groupId>
38        <artifactId>spring-boot-maven-plugin</artifactId>
39      </plugin>
40    </plugins>
41  </build>
42
43 </project>
```

← Riferimento ad un parent POM
(implicito: Super POM)

← Artifact Coordinates (GAV)

← Properties

← Dipendenze

← Dipendenze con scope (es. test)

← plugin per aggiungere
funzionalità (goal) eseguibili
su richiesta o in automatico
durante una fase del lifecycle

Esempio di plugin: JaCoCo per la code coverage

```
<build>
  <plugins>
    <plugin>
      <groupId>org.jacoco</groupId>
      <artifactId>jacoco-maven-plugin</artifactId>
      <version>0.8.2</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>
  </plugins>
</build>
```

Nella prima parte viene indicato il GAV

In executions vengono specificati quali goal aggiungere al build lifecycle

È possibile sovrascrivere la fase di default a cui agganciare il goal (es. test)

Nell'esempio, la creazione del report di copertura sarebbe stata creata solo nel lifecycle di report, in questo caso stiamo specificando di volerlo creare alla fine della fase di test

\$ mvn test

- Esegue prepare-agent durante la fase initialize (binding di default)
- Esegue tutte le fasi fino a test (es. compile)
- Esegue i test (surefire:test) e crea il report di copertura (jacoco:report)

Esempio di plugin: creare un jar-with-dependencies

```
<build>
  <plugins>
    <plugin>
      <artifactId>maven-assembly-plugin</artifactId>
      <version>2.6</version>
      <configuration>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
        <archive>
          <manifest>
            <mainClass>com.example.Main</mainClass>
          </manifest>
        </archive>
      </configuration>
      <executions>
        <execution>
          <id>make-assembly</id>
          <phase>package</phase>
          <goals>
            <goal>single</goal>
          </goals>
        </execution>
      </executions>
    </plugin>
  </plugins>
</build>
```

Di base **mvn package** crea un jar con solo le classi del progetto.

Tramite **mvn install** questo jar senza dipendenze può essere aggiunto al repository locale, rendendolo disponibile ad essere richiamato come dipendenza in altri progetti

In alcuni casi però vorremmo poter fornire un **jar autonomo ed eseguibile**, con tutte le dipendenze al suo interno

Per farlo possiamo usare questo plugin

Nota: per rendere il jar eseguibile bisogna indicare la classe con il main

\$ mvn package (crea un jar eseguibile con tutte le dipendenze)

\$ java -jar target/my-app-1.0-SNAPSHOT.jar

Properties

- Le properties sono dei segnaposto per dei valori di configurazione.
- Sono accessibili da altre parti nel POM tramite la notazione **`${X}`**, dove **X** è il **nome della proprietà**.
- Vengono spesso usate per avere consistenza nella versione da usare in più dipendenze
- Vengono anche usate dai plugin come parametri di configurazione: la loro modifica quindi semplifica la configurazione dei plugin

```
<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
  <project.reporting.outputEncoding>UTF-8</project.reporting.outputEncoding>
  <junit.version>4.11</junit.version>
  <my.custom.property>value</my.custom.property>
</properties>
```

Esercizio

- Creare un progetto maven ***producer*** che contiene una classe *Producer*. Questa espone un metodo `getProduct():String` che restituisce una semplice stringa “Prodotto_n” con n intero progressivo incrementato ad ogni richiesta di prodotto
- Creare un altro progetto maven ***consumer*** che contiene una classe *Consumer*, aggiungendo il progetto producer tra le sue dipendenze, in modo che questo, nel main, possa richiamare il metodo `getProduct()` su un'istanza di *Producer*
- Nota: per poter risolvere la dipendenza localmente è necessario:
 - fare **mvn install del progetto producer**, in modo da renderlo disponibile nel repository locale
 - aggiungere il **GAV del producer nel pom.xml** del consumer