SE 233 Advanced Programming

Chapter VI Build Management System

College of Arts, Media and Technology
Chiang Mai University

Lect Passakorn Phannachitta, D.Eng.

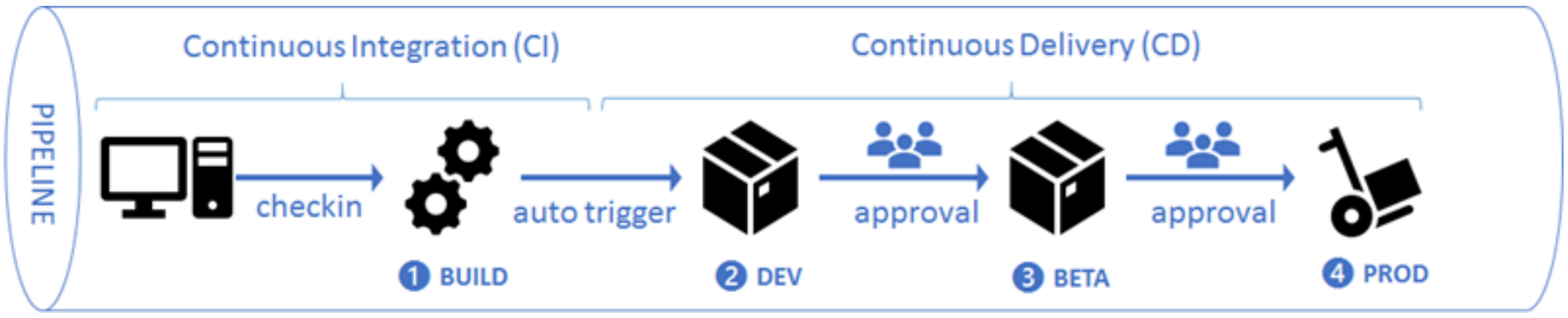
passakorn.p@cmu.ac.th

College of Arts, Media and Technology

Chiang Mai University, Chiangmai, Thailand

Ultimate goal of software development

- Always deployable and the release is always stable
- Software can be built in any system with zero or minimum additional effort



CI & CD will be discussed in SE234

- Many fundamentals are required before we are able to practice CI & CD with ease.
- In this course, we will discuss its very first essential components which is **build management**

Software build management

- An essential of software configuration management (SCM)

- SCM primary responsibilities are to **handle changes**, **corrections**, **extensions**, and **adaptations** made to the software application during its **developments**.

Software build management

- SCM key functionalities are those related to build and release management, including **workspace**, **versioning**, **building**, **release management**, **dependency management**, **repository management**, and **change management**.

Build tool

- A composite product primarily used to compile and make a runnable software application from source codes.
  - configuration component, e.g., a build script
  - construction components, e.g., an executable

Build tool — Main tasks

- recompile **only the portions** of software application that have been modified;
- properly handle **dependencies** of the software application;
- allow the user to select compiling profiles, e.g., development or production;
- minimize the effort required to adapt to changes regarding new product requirements;
- make everything **automated** and remove all manual efforts as possible.

Dependency

- E.g., we have imported many jar files to our project to use one or more method they provided.
- We call this situation as that the imported jar files are the dependencies of our program.
- When we work with more components, more and more jar files will be required.
- In most cases, these jar files may also have their dependencies as well.

Dependency — without carefully management

- They are easily conflicted in production.

- E.g., the libraries we manually included in our application may cause the other dependencies of dependencies to fail to compile.
    - Versioning
    - Etc.

Dependency management tool

- Its key responsibility is to identify software components' dependencies and resolve dependency conflicts if they exist:
  - automatically search and pinpoint a unique set of dependencies accepted by all the modules or components of the software application,
  - make the application compiled and run successfully.

Java build management system

- For example
  - ANT (& Ivy)
  - Maven
  - Gradle

ANT + Ivy

- Ant was the first build tool for JAVA. It was created in 1999
  - **Good points**
    - The very first build tool that allows us to have a complete control over the build process.
    - Easy to learn — XML based scripts
  - **Drawback**
    - It is difficult to manage ANT's XML used with all but very small projects
    - ANT does not have the predefined set of build cycle — we need to write the scripts for everything.

ANT + Ivy

- Example — ANT part

```
<project xmlns:ivy="antlib:org.apache.ivy.ant" name="java-build-tools" default="jar">
    <property name="src.dir" value="src" />
    <property name="build.dir" value="build" />
    <property name="classes.dir" value="${build.dir}/classes" />
    <property name="jar.dir" value="${build.dir}/jar" />
    <property name="lib.dir" value="lib" />
    <path id="lib.path.id">
        <fileset dir="${lib.dir}" />
    </path>

    <target name="resolve">
        <ivy:retrieve />
    </target>

    <target name="clean">
        <delete dir="${build.dir}" />
    </target>

    <target name="compile" depends="resolve">
        <mkdir dir="${classes.dir}" />
        <javac srcdir="${src.dir}" destdir="${classes.dir}" classpathref="lib.path.id" />
    </target>

    <target name="jar" depends="compile">
        <mkdir dir="${jar.dir}" />
        <jar destfile="${jar.dir}/${ant.project.name}.jar" basedir="${classes.dir}" />
    </target>
</project>
```

ANT + Ivy

- Example — Ivy part

```xml
<ivy-module version="2.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="http://ant.apache.org/ivy/schemas/ivy.xsd">
    <info organisation="org.apache" module="java-build-tools" />
    <dependencies>
        <dependency org="junit" name="junit" rev="4.11"></dependency>
    </dependencies>
</ivy-module>
```

Maven

- Maven was released in 2004.
- Good points
  - Able to download dependencies over the network.
  - Based on cycled processes.
- Drawback
  - Conflicts between different versions of the same library cannot be simply handled by Maven.
  - It is more difficult to write **custom** build scripts in MAVEN than ANT.
  - MAVEN configuration is based on XML and is said to be cumbersome.

Maven

- Example

```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>com.programming.mitra</groupId>
    <artifactId>java-build-tools</artifactId>
    <packaging>jar</packaging>
    <version>1.0</version>

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

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>2.3.2</version>
            </plugin>
        </plugins>
    </build>
</project>
```

Gradle

- Gradle was released in 2007.
- **Good points**
  - The build scripts is generally much shorter and clearer than those written based on ANT or MAVEN syntax.
  - Has its own native dependency resolution engine
- **Drawback**
  - There are some critics upon its dependency management in which some difficult case, using MAVEN is more pre

Let’s scrutinize Maven in this course

- Main areas
  - complete build lifecycle
  - dependency management

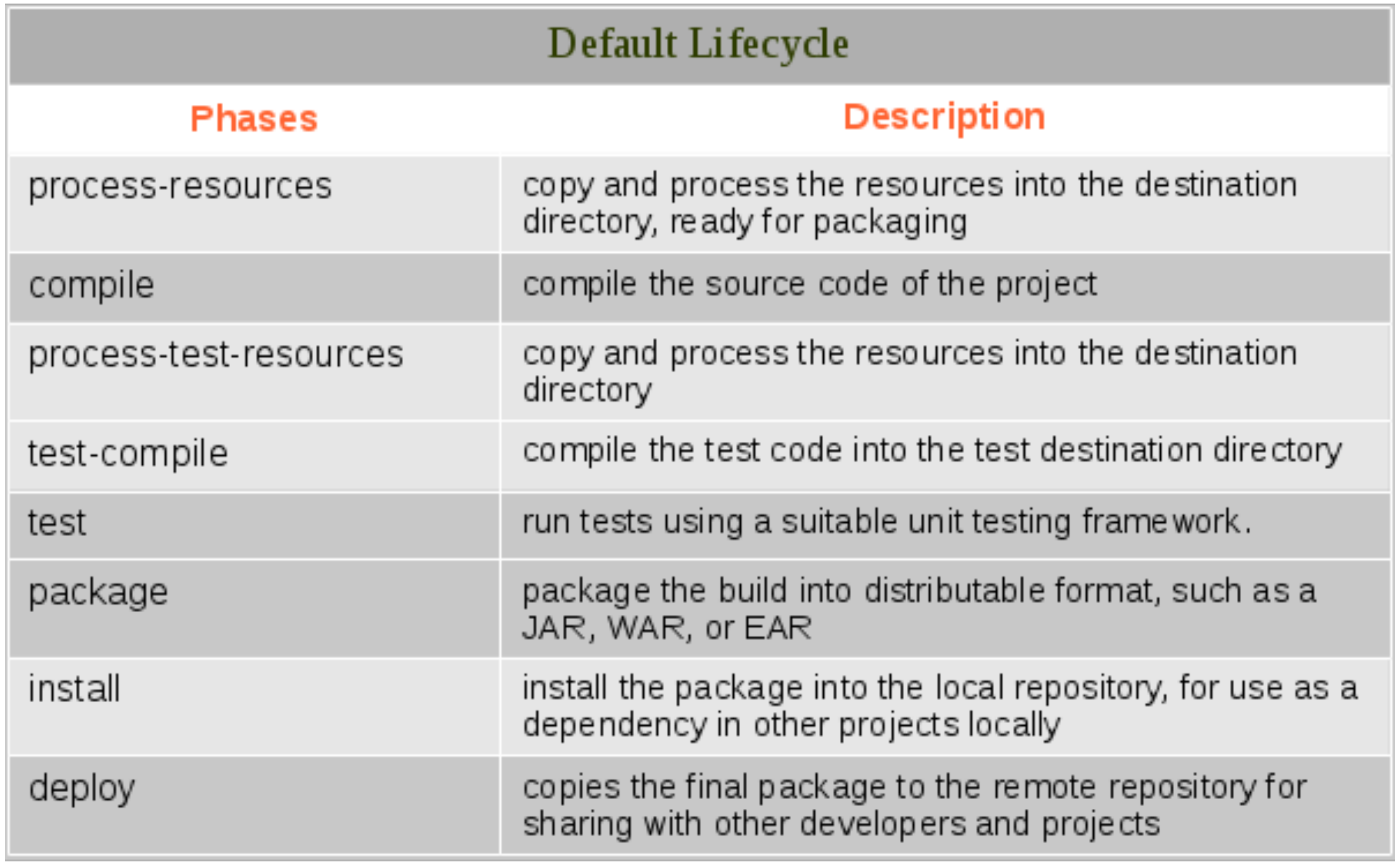
Build life cycle

- sequential phases representing stages in building a software.
  - Each phase contains goals it has to fulfill as to say the task carried out at the particular build stage is completed.
  - Maven provides 23 default build lifecycle phases in total; however, only a subset is needed for a particular software project.

Build life cycle

- E.g., process-resources -> compile, -> process-test-resources -> test-compile -> test -> package -> install -> deploy
- With the main principle saying that “All build systems are essentially the same”

Build life cycle — Typical phases



Default Lifecycle	
Phases	**Description**
process-resources	copy and process the resources into the destination directory, ready for packaging
compile	compile the source code of the project
process-test-resources	copy and process the resources into the destination directory
test-compile	compile the test code into the test destination directory
test	run tests using a suitable unit testing framework.
package	package the build into distributable format, such as a JAR, WAR, or EAR
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

Ref: https://www.codetab.org/tutorial/apache-maven/lifecycle-phases/ (access Aug 2020)

Standard directory

```
src
└── main
│   ├── java
│   └── resources
└── test
    ├── java
    └── resources
target
pom.xml
```

- When joining a new project, a Maven user who has already familiar with this standardized structure can easily understand the project layout and know exactly where to find each particular file.
- can be shared easily between IDEs without compatibility issues.

Configuration — Project object model (POM)

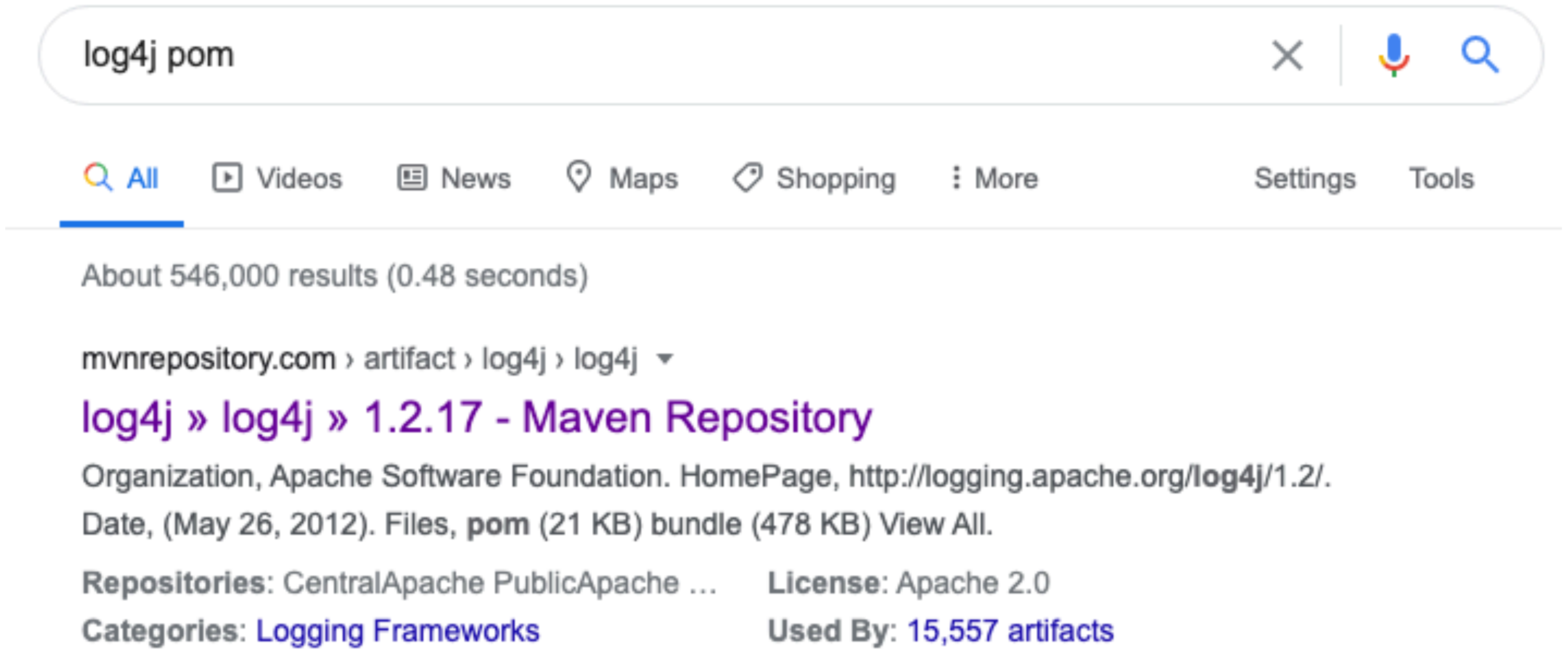
- POM is an xml file located at the root directory of a Maven Java project.
- A POM file contains information about the project and configuration used for building the project.
- Maven will read the POM file of the project at the project root directory to properly configure the project building following the developers' intent.

Project object model (POM)

Project object model (POM)

- GAV Syntax: groupId:artifactId:version
- Maven uniquely identifies a project using:
  - **groupID:** Arbitrary project grouping identifier — Usually loosely based on Java package
  - **artfiactId:** Arbitrary name of project
  - **version:** Version of project
  - Format {Major}.{Minor}.{Maintanence}
  - Add '-SNAPSHOT' to identify in development

POM — Adding a dependency



About 546,000 results (0.48 seconds)

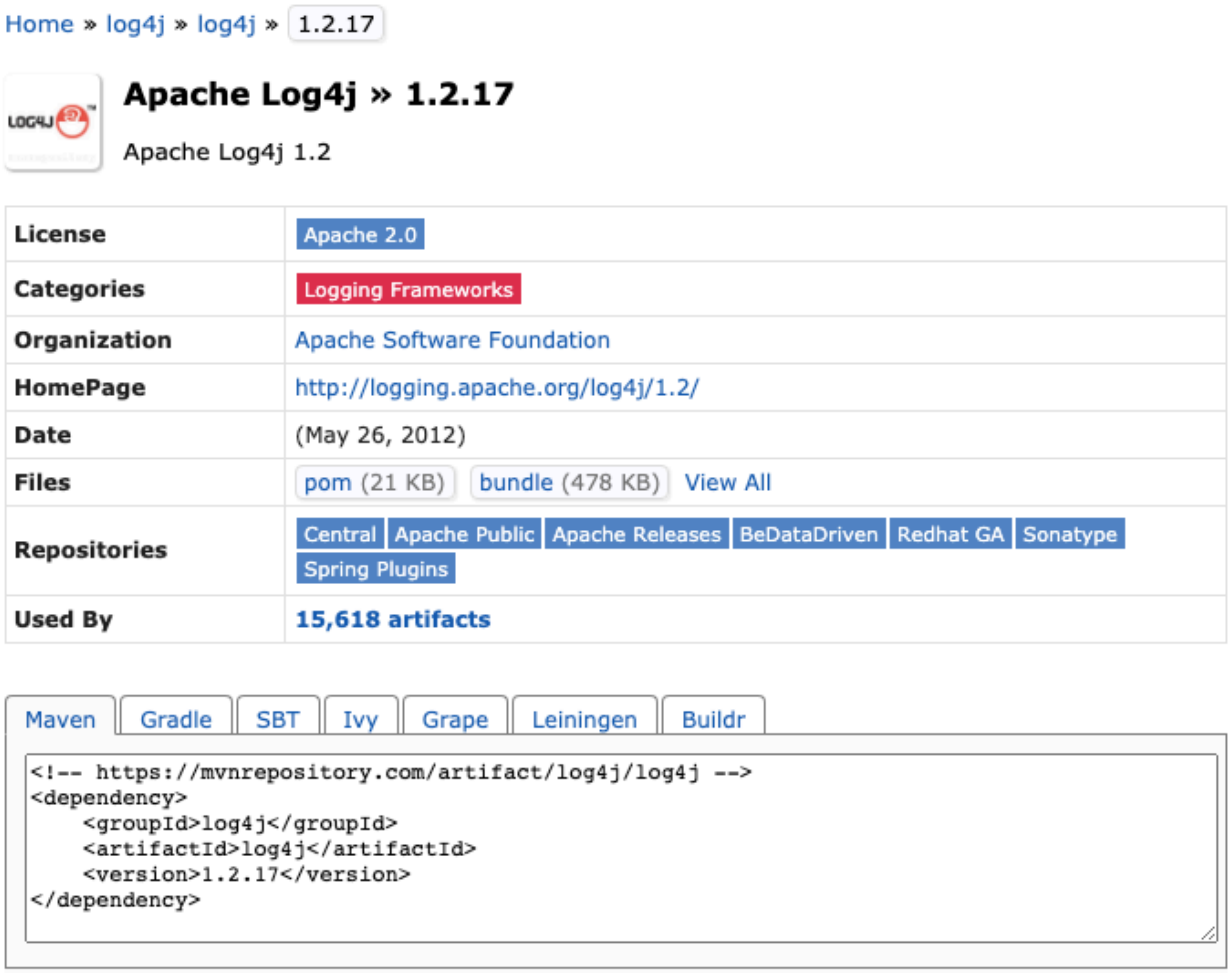
mvnrepository.com › artifact › log4j › log4j

log4j » log4j » 1.2.17 - Maven Repository

Organization, Apache Software Foundation. HomePage, http://logging.apache.org/**log4j**/1.2/. Date, (May 26, 2012). Files, **pom** (21 KB) bundle (478 KB) View All.

Repositories: CentralApache PublicApache ... **License**: Apache 2.0
Categories: Logging Frameworks **Used By**: 15,557 artifacts

POM — Adding a dependency



Home » log4j » log4j » 1.2.17

Apache Log4j » 1.2.17

Apache Log4j 1.2

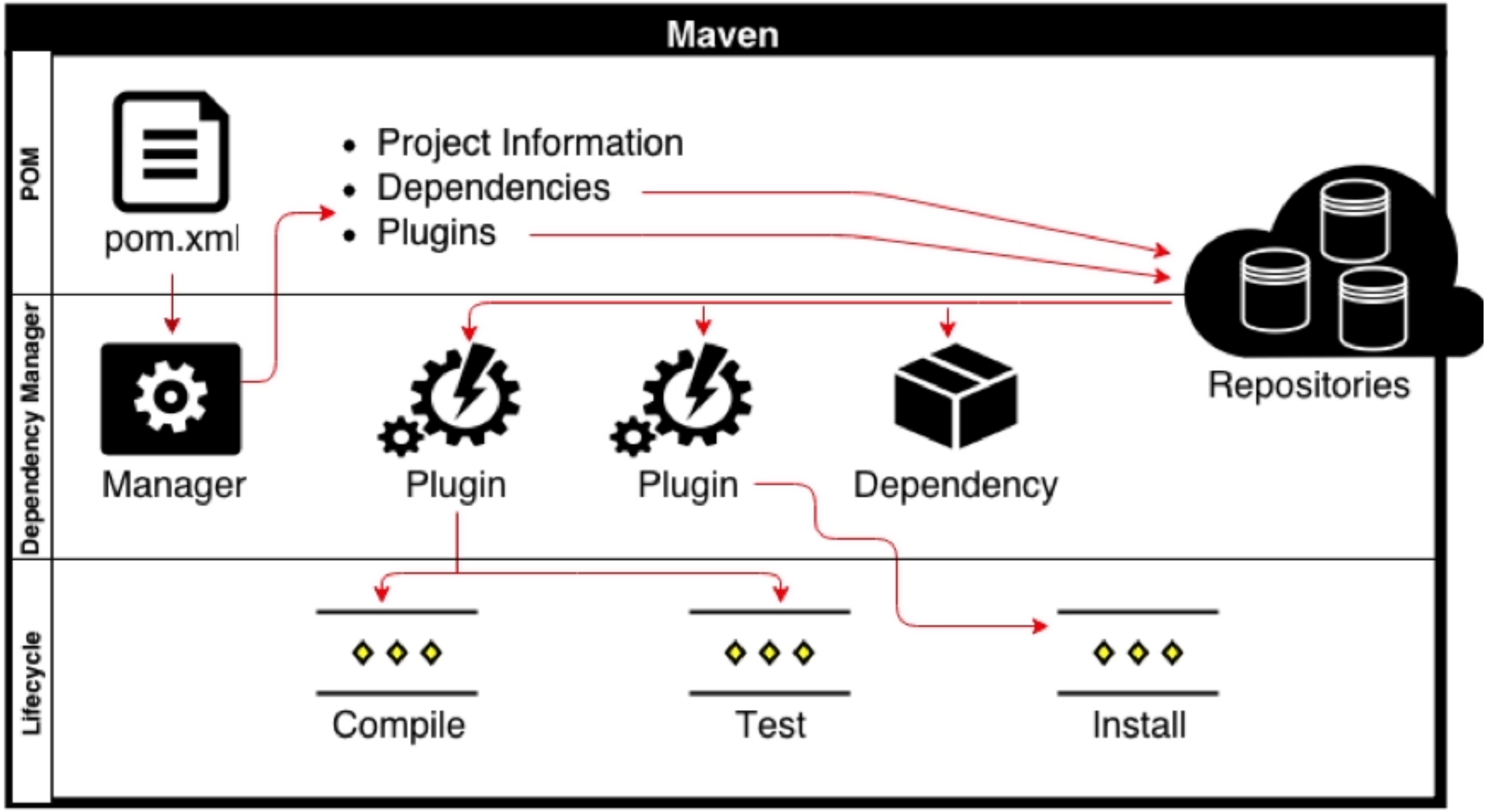
License	Apache 2.0
Categories	Logging Frameworks
Organization	Apache Software Foundation
HomePage	http://logging.apache.org/log4j/1.2/
Date	(May 26, 2012)
Files	pom (21 KB) bundle (478 KB) View All
Repositories	Central Apache Public Apache Releases BeDataDriven Redhat GA Sonatype Spring Plugins
Used By	**15,618 artifacts**

Maven | Gradle | SBT | Ivy | Grape | Leiningen | Buildr

```
<!-- https://mvnrepository.com/artifact/log4j/log4j -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

Project object model (POM)

Background work



Summary

- Software build management is an essential of software configuration management (SCM)
- A composite product primarily used to compile and make a runnable software application from source codes.

Summary

- With Maven, without the hassle of downloading and importing all the dependencies to our project classpath, we only need to copy the dependency coordinates from the Maven repository and paste it in the Maven configuration file in our project and let Maven automatically handle rest for us.

Questions