



Getting to Know Java Build Tools

مباحث تکمیلی در برنامه نویسی پیشرفته
دانشگاه صنعتی امیرکبیر - بهار ۱۳۹۷

Why Build Tools?

- Building Java software projects in the industry, typically consist of several tasks such as:
 - downloading dependencies,
 - putting additional JARs on the CLASSPATH,
 - compiling source code into binary code,
 - running tests,
 - packaging into deployable artifacts (JAR, WAR, ...)
 - deploying artifacts to an application server or repository.

Why Build Tools?

- When working with a team on such projects, there are several issues that need to be handled across all team members:
 - varying project structures in different IDEs;
 - integrating IDEs with various tools;
 - managing and sharing project dependencies;
 - enabling all members to build and test the project;
- To solve these issues and facilitate team-work, software build tools were created by developers.

Why Build Tools?

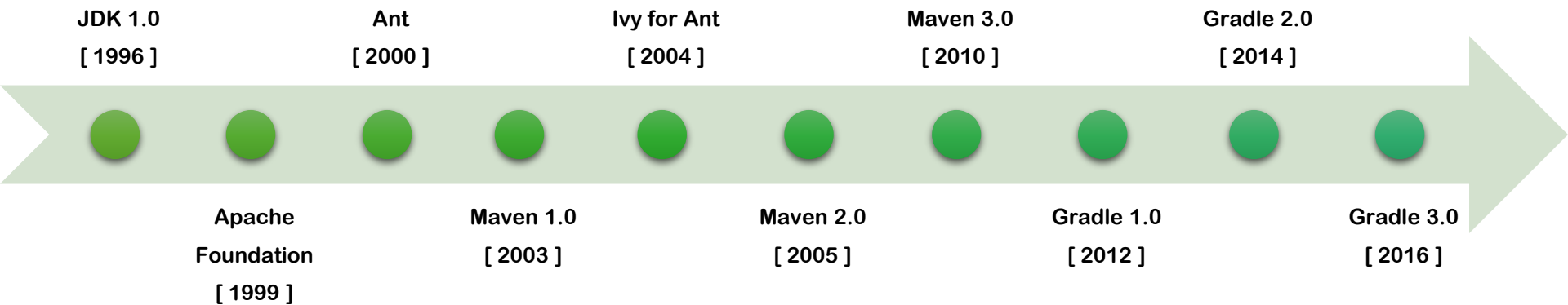
- So, build tools are for team projects, right? Wrong!
- Single-person (solo) projects can also benefit from using build-automation tools :
 1. Easy dependency management
 2. Automating build process (compile, test, deploy, run)
 3. Releasing open-source projects to the public community

Java Build Tools





Three well-established Java build-tools:






History Timeline



Pros / Cons

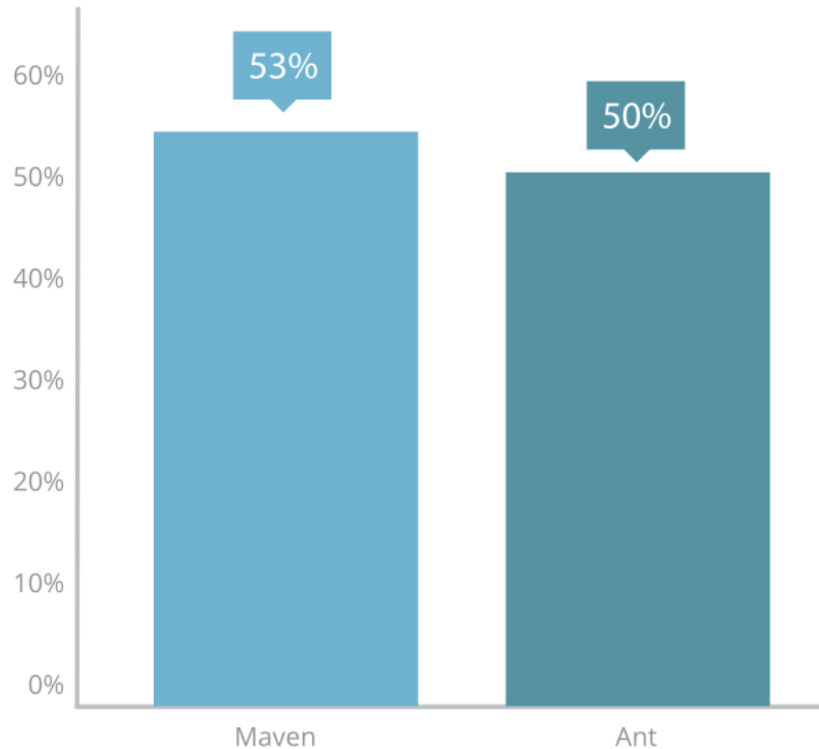
	WHAT ROCKS	WHAT LACKS
	<ul style="list-style-type: none"> • Maven Central was a major game changer and is still the de facto repository for plugins and anything else you want. • Common project structure and build cycles makes you familiar with all Maven projects if you're familiar with one. • Easy artifact sharing lets you upload to an existing service or manage your own repository (i.e. Artifactory or Nexus). 	<ul style="list-style-type: none"> • Little customization available, relies on other tools for that. • "Downloading the internets" at risk if you don't read the special documentation prepared because so many people complained • Long pom.xml-s aren't easily readable, especially with multi-module projects.
	<ul style="list-style-type: none"> • Lets you ditch duplicated folder structures for multiple-build projects, and enables clean and nice configurations for custom builds. • Gets rid of projects with large build scripts, while still being relatively readable without the boilerplate. • DSL + Groovy enable to write custom and complex logic sometimes needed in Android build scripts. 	<ul style="list-style-type: none"> • Requires knowledge of Groovy to be able to work with Gradle, presenting a learning curve. • DSL means that a random developer will have a lot of trouble understanding what's going on with your build • There is no Gradle repository or repo format yet, so familiarity with either Ivy or Maven is still needed when doing coding in Java.
 	<ul style="list-style-type: none"> • You are in complete control of the build process, you decide everything... • No inherent restrictions imposed with regards to directory structure • Customizable in a way that Maven is not, depending on your comfort with plugins 	<ul style="list-style-type: none"> • You are in complete control of the build process, nothing is handed to you... • Build scripts can easily become long and verbose (try over 130 LoC just to start Ant and Ivy!) • No build script is the same; checking out and building unknown software using Ant can require you to look through the defined targets, to figure out what to actually call in order to make everything run.

Comparison

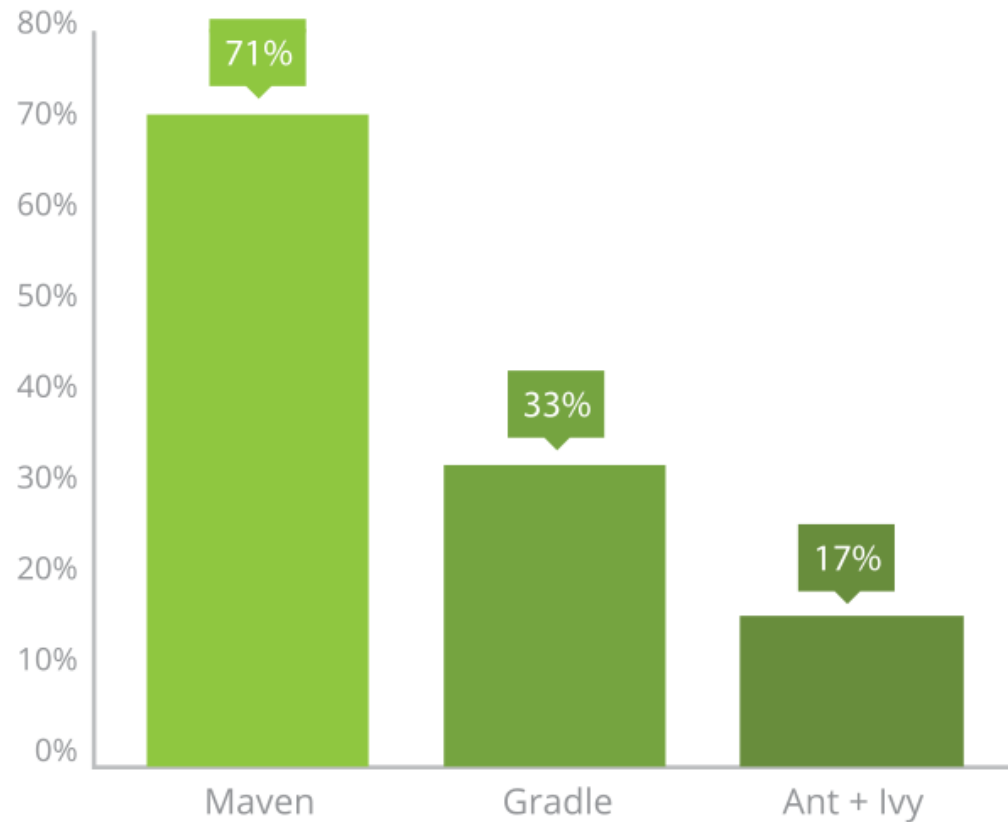
	maven	 gradle	 
Learning Curve	3	4	3
Build Speed	4.5	4.5	3.5
Complexity	1.5	4.5	3
Plugins	4	3	3
Community & Docs	3	5	2
Developer Tools Integration	5	3	4
Total	21	24	18.5

Popularity

Build Tools Popularity - Late 2010



Build Tools Popularity - Mid 2013



Maven is our choice

- **Due to its high popularity in the industry, we shall learn about how to use Maven.**
- **Gradle is clearly the choice of the future, and that's why Google opted it for Android; but still Maven is here to stay for a long time.**

Installing Maven

- Let's begin by installing Maven:
 - On Debian/Ubuntu Linux systems, it's as easy as this:

```
$ sudo apt install maven
```

- On Microsoft Windows and Apple macOS systems, first you need to install Java (JDK), and then download the latest stable version of Maven from the official website:

<https://maven.apache.org/download.html>

Installing Maven

■ Installation on Microsoft Windows :

1. After downloading, unzip it to an appropriate folder, like:

`C:\Program Files\Apache\maven`

2. Add the following environment variables to the system:

```
JAVA_HOME    = C:\Program Files\Java\jdk1.8.0_xx  
MAVEN_HOME   = C:\Program Files\Apache\maven
```

2. Add Maven's bin directory to system's PATH:

```
PATH += ;%MAVEN_HOME%\bin
```

Running Maven

- To test your maven installation, simply run:

```
$ mvn -version  
  
Apache Maven 3.3.9  
Maven home: /usr/share/maven  
Java version: 1.8.0_144, vendor: Oracle Corporation  
Java home: /usr/lib/jvm/jdk1.8.0_144/jre  
Default locale: en_US, platform encoding: UTF-8  
OS name: "linux", version: "4.13.0-41", arch: "amd64", family: "unix"
```

- This command tests whether Maven is properly installed and working.

Project Setup

- To setup a basic project using Maven, we use the following command:

```
$ mvn archetype:generate  
    -DartifactId=my-app  
    -DgroupId=ir.domain.app  
    -DarchetypeArtifactId=maven-archetype-quickstart  
    -DinteractiveMode=false
```

- Maven will start downloading required files from online repositories and setup the project structure.

Project Setup

- After the command completes, a new directory named 'my-app' is created with the following structure:

```
$ tree my-app/  
├── pom.xml  
└── src  
    ├── main  
    │   ├── java  
    │   │   ├── ir  
    │   │   │   ├── domain  
    │   │   │   │   ├── app  
    │   │   │   │   │   App.java  
    │   └── test  
    │       ├── java  
    │       │   ├── ir  
    │       │   │   ├── domain  
    │       │   │   │   ├── app  
    │       │   │   │   │   AppTest.java
```

11 directories, 3 files

Project Setup

- The project setup command, includes:
 1. `archetype:generate`
This part specifies we want Maven to generate a new project
 2. `archetypeArtifactId=maven-archetype-quickstart`
This part specifies the structure of the project
 3. `artifactId=my-app`
This is the name of our project
 4. `groupId=ir.domain.app`
This is the base package for our source codes
 5. `interactiveMode=false`
Maven will not ask for required parameters

Building the Project

- To compile source files of the new project, enter:

```
$ cd my-app  
$ mvn compile
```

- To package the compiled sources in a JAR, enter:

```
$ mvn package
```

- The above commands will add a new directory to the project named 'target' which includes all the compiled class files and the packaged JAR file.

Running the Project

- To run the tests, enter:

```
$ mvn test
```

```
-----
```

```
TESTS
```

```
-----
```

```
Running ir.domain.app.AppTest
```

```
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.003 sec
```

- To execute the packaged program, enter:

```
$ java -cp target/my-app-1.0-SNAPSHOT.jar ir.domain.app.App
```

```
Hello World!
```

Maven Project Lifecycle

- Lifecycle of a Maven project consists of various build phases:
 1. clean: cleans the project directory from build artifacts
 2. validate: checks the correctness of the project
 3. compile: compiles provided source code
 4. test: executes unit tests
 5. package: packages compiled code into a deployable file
 6. integration-test: executes additional integration tests
 7. deploy: deploys the package file

The POM

- The POM (Project Object Model) file named 'pom.xml' is the core of a Maven project's configuration.
- It is a single configuration file that contains the majority of information required to build a project.
- The POM file of large projects can become huge and understanding it can be daunting in its complexity!
- The POM file of our example is listed on the next page.

The POM

```
<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>ir.domain.app</groupId>
```

```
    <artifactId>my-app</artifactId>
```

```
    <packaging>jar</packaging>
```

```
    <version>1.0-SNAPSHOT</version>
```

Basic Project Info

```
    <name>my-app</name>
```

```
    <url>http://app.domain.ir</url>
```

Additional Project Info

```
    <dependencies>
```

```
      <dependency>
```

```
        <groupId>junit</groupId>
```

```
        <artifactId>junit</artifactId>
```

```
        <version>3.8.1</version>
```

```
        <scope>test</scope>
```

```
      </dependency>
```

```
    </dependencies>
```

Project Dependencies

```
</project>
```

The POM

- This is just a simple and small POM file. As mentioned, POM files can become very large and complicated!
- To learn more about all the configurations in a POM, check out the following URL:

<https://maven.apache.org/pom.html>

Dependencies

- Dependencies: External libraries (JAR) that a project uses.
- The dependency management feature in Maven ensures automatic download of those libraries from a central repository.
- To declare a new dependency, you need to provide the 'groupId', 'artifactId', and the 'version'. Example:

```
<dependency>
  <groupId>org.springframework</groupId>
  <artifactId>spring-core</artifactId>
  <version>4.3.5.RELEASE</version>
</dependency>
```

Repositories

- Repository: A location used by Maven to store and fetch build artifacts and dependencies.
- The default local repository is located at “`.m2/repository`” folder under the user home directory.
- If an artifact is available in the local repo, Maven uses it. Otherwise, it is downloaded from remote repositories and stored in the local repository for reuse.
- The default remote repository is “Maven Central”:

<http://central.maven.org/>

Repositories

- Some libraries (for legal, licensing, or other issues), are not available at the central repo but are available at other alternate repositories.
- Or your team / company may want to setup its own repository of Java libraries.
- In any case, to specify additional repos in the POM:

```
<repositories>
  <repository>
    <id>Team repo</id>
    <url>http://repo.domain.ir/</url>
  </repository>
</repositories>
```


Properties

- Properties are like variables defined in the POM file.
- They can be used with the `${name}` notation:

```
<properties>
  <spring.version>4.3.5.RELEASE</spring.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>${spring.version}</version>
  </dependency>
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>${spring.version}</version>
  </dependency>
</dependencies>
```


Java Versions

- To specify the source and target versions of our Java project, we can define these properties in our POM:

```
<properties>  
  <maven.compiler.source>1.7</maven.compiler.source>  
  <maven.compiler.target>1.7</maven.compiler.target>  
</properties>
```

- This is equivalent to specifying the following params on the Java compiler command-line:

```
$ javac -source 1.7 -target 1.7 project/src/*
```

Plugins & Goals

- Plugins are collections of one or more goals.
- Goals are actual tasks that Maven executes.
- Goals can be assigned to phases in the lifecycle.
- Let's configure our POM with some plugins to make Maven build an executable JAR during the 'package' phase.
- We shall demonstrate this in two different ways.

Java Versions

- But before that ...
- Alternative way using Maven plugins to specify the source and target versions of our Java project.

<https://maven.apache.org/plugins/maven-compiler-plugin/examples/set-compiler-source-and-target.html>

```
<project>
  [...]
  <build>
    [...]
    <plugins>
      <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.7.0</version>
        <configuration>
          <source>1.7</source>
          <target>1.7</target>
          <executable>
            /usr/lib/jvm/jdk1.7.0_80/bin/javac
          </executable>
        </configuration>
      </plugin>
    </plugins>
    [...]
  </build>
  [...]
</project>
```

Executable JAR

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-dependency-plugin</artifactId>
      <executions>
        <execution>
          <id>copy-dependencies</id>
          <phase>prepare-package</phase>
          <goals>
            <goal>copy-dependencies</goal>
          </goals>
          <configuration>
            <outputDirectory>
              ${project.build.directory}/libs
            </outputDirectory>
          </configuration>
        </execution>
      </executions>
    </plugin>
    ...
```

Executable JAR

continued from previous slide ...

```
...
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-jar-plugin</artifactId>
  <configuration>
    <archive>
      <manifest>
        <addClasspath>true</addClasspath>
        <classpathPrefix>libs/</classpathPrefix>
        <mainClass>
          ir.domain.app.App
        </mainClass>
      </manifest>
    </archive>
  </configuration>
</plugin>
</plugins>
</build>
```

Executable JAR

- This configuration manually specifies all the steps required to create an executable JAR.
- In the first plugin of the <build> section, we specify a goal named 'copy-dependencies', which tells Maven to copy these dependencies into the specified 'outputDirectory'.
- In the second plugin, the JAR manifest is configured. We add a classpath, with all dependencies (folder 'libs'), and specify the main class.
- Now, execute the package phase:

```
$ mvn clean package
```


Executable JAR

- This will first clean all previously created artifacts, then as before, it will create a new directory named 'target' which includes the 'libs' and JAR file of our project.
- The JAR will only run using ' java -jar ' if the 'libs' folder is placed alongside it.
- A different approach is to create a so called “ fat-JAR ”; which in contrast to the previous approach, will produce a single larger JAR that includes all its dependencies.

Fat JAR

```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-assembly-plugin</artifactId>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>single</goal>
      </goals>
      <configuration>
        <archive>
          <manifest>
            <mainClass>ir.domain.app.App</mainClass>
          </manifest>
        </archive>
        <descriptorRefs>
          <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
      </configuration>
    </execution>
  </executions>
</plugin>
```

Fat JAR

- Now, execute the package phase:

```
$ mvn clean package
```

- Again, this will create JAR files in the 'target' folder. The JAR file which its name ends with 'jar-with-dependencies' is the executable fat-JAR.
- There are even more approaches for creating executable JAR files. For more information refer to the link below:

<http://www.baeldung.com/executable-jar-with-maven>

Build Profiles

- In professional software teams, it is typical to have different environments such as:
 - ✓ Develop
 - ✓ Testing
 - ✓ Production
- These environments have different build & deploy process.
- Maven can support this using profiles.
- A profile is basically a set of configuration values.

Build Profiles

```
<profiles>
```

```
<profile>
```

```
  <id>develop</id>
```

```
  <activation>
```

```
    <activeByDefault>true</activeByDefault>
```

```
  </activation>
```

```
  <build>
```

```
    <plugins> ... </plugins>
```

```
  </build>
```

```
</profile>
```

```
<profile>
```

```
  <id>production</id>
```

```
  <build>
```

```
    <plugins> ... </plugins>
```

```
  </build>
```

```
</profile>
```

```
</profiles>
```



Default Active Profile

Build Profiles

- With the above defined profiles, the below command uses the 'develop' profile:

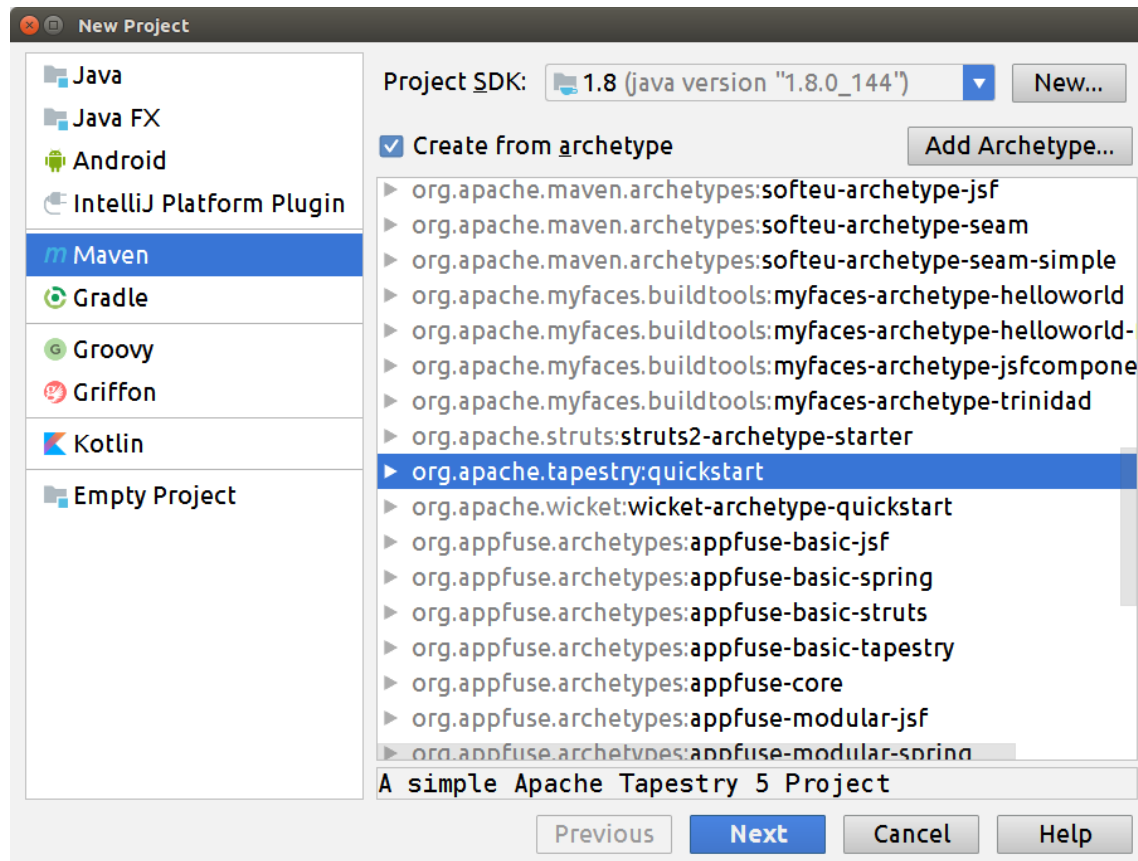
```
$ mvn clean package
```

- If you need to use the 'production' profile, enter:

```
$ mvn clean package -Pproduction
```


Maven & IDEs

Creating a new Maven project in IntelliJ-IDEA:



Questions ?

