

1. Introduction

what is a build tool ?

2. Maven

- A. Build Lifecycle.
 - B. Plugins in Maven.
 - C. Dependency management.
 - D. Project Object Model(POM).
-

3. Gradle

- A. Build script.
First and foremost, what is groovy?
 - B. Build process
 - C. Plugins
 - D. dependency management
-

4. Conclusion

Build Tools


1. Introduction

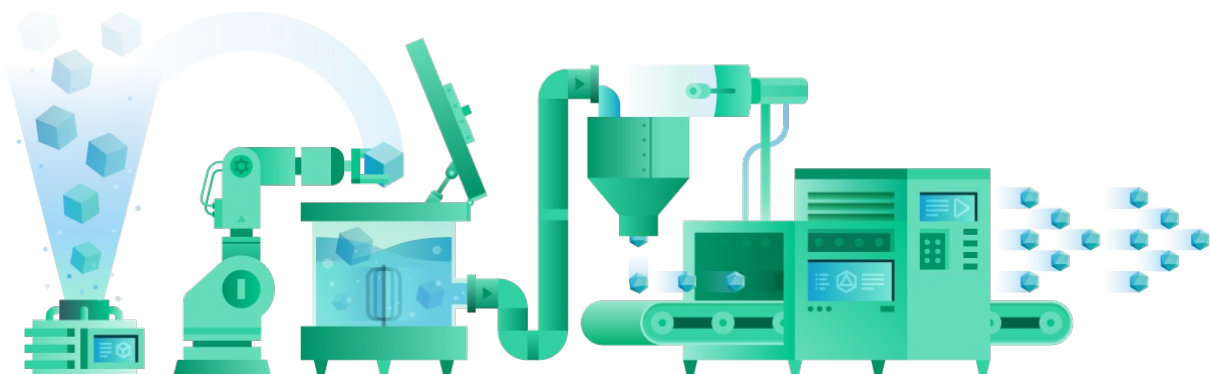
what is a build tool ?

A software application goes through a precise build process made of distinct automated steps before being commercialized as any other commercial product that needs assembling such as cars, planes or toys.

The build process of a software is composed of multiple automated and successive steps, each performing a specific task, that together in the end deliver a ready-to-use product, or in software engineering's lexical field, an artifact.

In its essence, a build tool is a software that automates the build process of one's applications and provides a reusable and easy to maintain build process.

In this redaction, we are going to be focusing on two popular tools : Apache **Maven**™ and  **gradle** from Gradle.Inc.



2. Maven™

Maven is a multi-module supporting build tool, a dependency management tool and a documentation tool. It is mainly used for java applications but supports a multitude of other languages.

Maven uses **Convention** over **Configuration**, meaning for purposes of increasing reusability, it provides a default build process and takes care of most of the build related tasks such as compilation, distribution, documentation, team collaboration and others seamlessly.

Developers do not have to spend unnecessary time on every single configuration detail, as maven provides a conventional default project structure at the project's creation. One is only required to place files accordingly and does not need to worry about detailing the structure while configuring the project.

A. Build Lifecycle.

When building an application, Maven equips the developer with a build process called a *Lifecycle* composed of multiple *phases*. One can either perform each phase of the Lifecycle alone or together with all the preceding phases.

Another option in the build process is using something called *goals*. A **goal** represents a specific task which contributes to the building and managing of a project. Goals are phase bound, meaning they are executed in phases which helps (the phases help or the whole thing ?) determine the order goals get executed in. The preceding phase's goals are always going to be executed before the succeeding phase's goals.

Goals are a bit peculiar in Maven due to their ability of being specified **pre** or **post** phase as well as being invoked directly through the command line independently of the build process even when they are bound to no phase. Goals are executed by plugins, which are at the heart of Maven's core functionalities(**section B**).

Phases in a typical maven build Lifecycle :

Phase:**Description:**

<u>prepare-resources</u>	<u>resource copying</u>	<u>Resource copying can be customized in this phase.</u>
<u>validate</u>	<u>Validating</u>	<u>the information</u> <u>Validates if the project is correct and if all necessary information is available.</u>
<u>compile</u>	<u>compilation</u>	<u>Source code compilation is done in this phase.</u>
<u>Test</u>	<u>Testing</u>	<u>Tests the compiled source code suitable for testing framework.</u>
<u>package</u>	<u>packaging</u>	<u>This phase creates the JAR/WAR package as mentioned in the packaging in POM.xml.</u>
<u>Install</u>	<u>installation</u>	<u>This phase installs the package in local/remote maven repository.</u>
<u>Deploy</u>	<u>Deploying</u>	<u>Copies the final package to the remote repository.</u>

B. Plugins in Maven.

Maven, in its core, is a plugin framework due to the fact that almost any action that one can think of performing on a project is implemented as a Maven plugin.

Fundamentally, a plugin provides extra features performed either as part of the build process, hence through phase bound goals, or invoked individually. In Maven, plugins are composed of a Plugin descriptor that specifies the plugin metadata such as the name of the plugin, the description of the plugin, the group id of the plugin and so forth, as well as one or more MOJOs.

A MOJO (**M**aven **O**ld **J**ava **O**bject) can be defined as an annotated Java class. It can be thought of as equivalent to a goal in Maven. A MOJO specifies a goal's name, which phase of the lifecycle it fits into, the parameters it is expecting, and more.

One can also make one's own custom maven plugin, but maven already provides a plethora of plugins and a massive community support, so the probability one might need to make plugins from scratch is low.

Maven plugins are usually used to :

- ❖ create jar file
- ❖ create war file
- ❖ compile code files
- ❖ unit testing of code
- ❖ create project documentation
- ❖ create project reports

Following is an example of a simple Custom maven plugin (**GreetingMojo.java**) with no parameters and its corresponding configuration in the POM file :

- The plugin has one MOJO and simply displays a greeting message in the console.

```
package com.uni;

import org.apache.maven.plugin.AbstractMojo;
import org.apache.maven.plugin.MojoExecutionException;
import org.apache.maven.plugins.annotations.Mojo;

/** Says "Hi" to the user. */
@Mojo( name = "sayhi" )
public class GreetingMojo extends AbstractMojo
{
    public void execute() throws MojoExecutionException
    {
        getLog().info( "Hello, world." );
    }
}
```

- The plugin's goal "sayhi" is executed in the compile phase.

```
<plugin>
  <groupId>com.uni</groupId>
  <artifactId>plugin-maven</artifactId>
  <version>1.0-SNAPSHOT</version>
  <executions>
    <execution>
      <phase>compile</phase>
      <goals>
        <goal>sayhi</goal>
      </goals>
    </execution>
  </executions>
</plugin>
```

C. Dependency management.

In Maven's lexical field, a *repository* is a directory in which all the project jars, jar libraries, plugins or any other project specific artifacts are stored and can be used by Maven easily.

Maven is used in large projects for its dependency management feature. Maven helps maintain a high degree of control and stability by providing dependency management support for, not only single-module projects, but also multi-module projects, with ease.

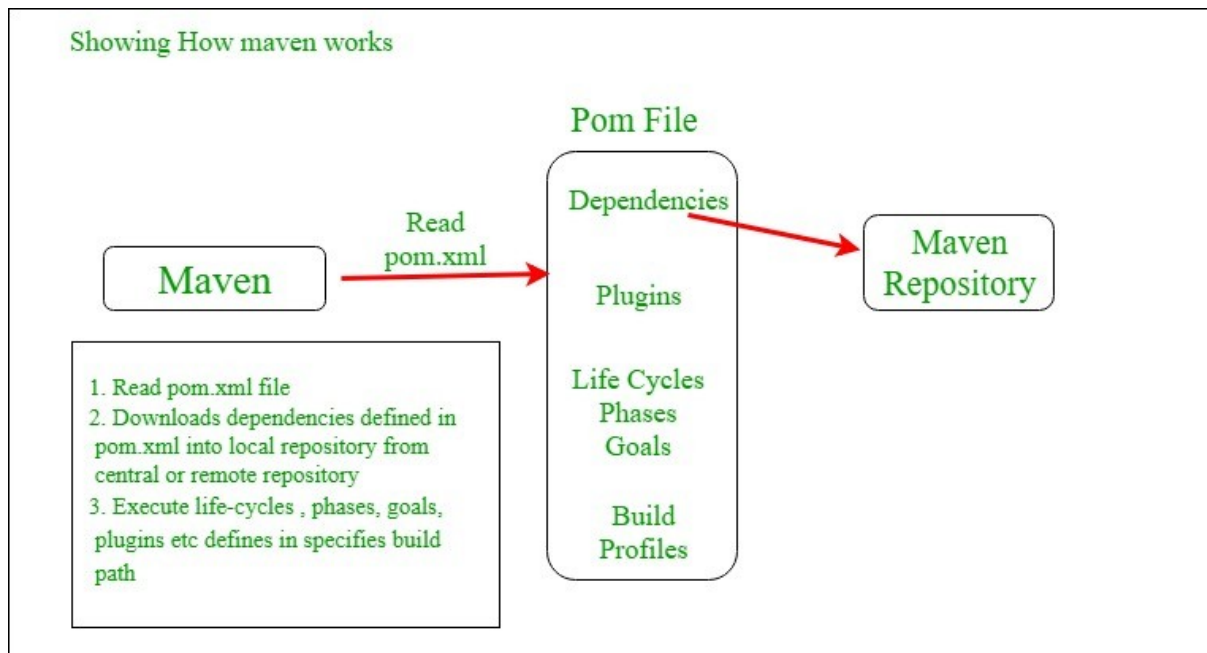
During the project's configuration, one could for instance specify which jar libraries one needs and the ensuing is Maven simply fetches those dependencies from remote repositories or the Maven central repository *et voila*.

Maven provides transitive dependency management as well, meaning if one's project is dependent on some libraries and those libraries are themselves dependent on other libraries, one only needs to specify their project dependencies and can rest assured that Maven will manage the transitive dependencies as well.

The sky is the limit when it comes to the levels that dependencies can be fetched from. The only occurrence of a problem would be in the case of cyclic dependency. In other words, if your project depends on A and A depends on C and C depends on B and B depends on A.

D. Project Object Model(POM).

POM (Project Object Model) is an XML file that contains information about the project and configuration details used by Maven to build the project i.e. sourcecode location, project dependencies etc. This file must be named as pom.xml and placed under root folder of project. When executing a task or a goal, maven reads the POM, gets the needed configuration information, then executes accordingly.



Simple POM Configuration sections :

The POM file consists of different sections each configured accordingly. For simplification purposes, this paper will focus on the main sections, namely, the basic project information section, the build section, and the reporting section.

Following is a simplified example of pom.xml file in which the developer configures the aforementioned sections for a simple “Hello world” Java application:

In The Basic information section they specified :

- The group ID, the artifact ID and the version.
- The dependencies required by this project :
 - **JUnit unit testing framework dependencies**
 - **Maven plugin API dependencies.**
 - **JAVA annotations dependencies.**

```
<!-- -Basic project information :-->

<groupId>com.uni.lu</groupId>
<artifactId>Mavenproject</artifactId>
<version>1.0-SNAPSHOT</version>

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

  <dependency>
    <groupId>org.apache.maven</groupId>
    <artifactId>maven-plugin-api</artifactId>
    <version>3.0</version>
  </dependency>

  <dependency>
    <groupId>org.apache.maven.plugin-tools</groupId>
    <artifactId>maven-plugin-annotations</artifactId>
    <version>3.6.0</version>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

In The Build section they specified:

- The default compiler plugin in order to compile java classes for the *compile* phase.
- The default reporting plugin in order to generate reports for the project's *site* phase.

```
<!-- -Build section :-->
```

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <version>3.5.1</version>
      <configuration>
        <source>1.8</source>
        <target>1.8</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-project-info-reports-plugin</artifactId>
      <version>2.7</version>
    </plugin>
  </plugins>
</build>
```

In The reporting section they specified :

- The checkstyle plugin that analyses **their** source code and generates a report with suggestions for the default *site* phase

```
<!-- Reporting section :-->
```

```
<reporting>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-checkstyle-plugin</artifactId>
      <version>3.1.0</version>
      <reportSets>
        <reportSet>
          <reports>
            <report>checkstyle</report>
          </reports>
        </reportSet>
      </reportSets>
    </plugin>
  </plugins>
</reporting>
```

3. gradle

Gradle is a Groovy-based build management system designed specifically for building Java-based projects.

It's well equipped, even better equipped than Maven, as the majority of the software developer community would agree.

It's also a multi-module supporting build tool, a dependency management tool, and a documentation tool. It has greater flexibility over Maven in terms of resolving version conflicts and managing transitive dependencies. Gradle is more recent than Maven. Although Maven has a broader support community due to its longer existence, Gradle is the official Android build tool.

A. Build script.

First and foremost, what is groovy?

Groovy is a domain specific language, or DSL, for describing builds in Gradle. Similar to Maven's `pom.xml` file, Gradle's Build script, *build.gradle*, describes one's build in Groovy and allows them to configure their project.

Groovy has a concise, familiar, and easy to learn syntax and is intended to be as natural as possible for Java developers.

The best things about groovy is that because it extends JDK, it subsequently extends Java's capabilities and allows one to achieve similar results with less lines of code.

In JetBrains's IntelliJ IDE, one can find a "groovy console" under the "tools" tab and test out Groovy syntax.

Following is simple examples demonstrating, respectively, A java application and a groovy line, that achieve the same result :

- In this Java code , the developer is executing the "ls -l" linux command, reading the output using `BufferedReader` before printing it on the console.

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class Sup {

    public static void main(String[] args) throws IOException {

        Process proc = Runtime.getRuntime().exec( command: "ls -l" );
        BufferedReader result = new BufferedReader(
            new InputStreamReader(proc.getInputStream()));

        String line;

        while((line = result.readLine()) != null) {

            System.out.println(line);

        }

    }

}

```

output :

```

total 12
drwxr-xr-x 3 mahmud mahmud 4096 Nov 30 22:49 out
drwxr-xr-x 2 mahmud mahmud 4096 Nov 30 22:48 src
-rw-r--r-- 1 mahmud mahmud  423 Nov 30 22:45 sup.iml

```

- In order to achieve the same in Groovy, they simply execute this line of code in the groovy console :

```
println "ls -l".execute().text
```

output :

```

total 12
drwxr-xr-x 3 mahmud mahmud 4096 Nov 30 22:49 out
drwxr-xr-x 2 mahmud mahmud 4096 Nov 30 22:48 src
-rw-r--r-- 1 mahmud mahmud  423 Nov 30 22:45 sup.iml

```

B. Build process

In Gradle , the Build process consists of one or more projects and each project consists of one or more tasks. Similar to a phase in Maven, a task is a piece of work performed in the build process. A Task could be anything from simply compiling classes to creating JARs or publishing proprietary archives to a repository.

Where Maven follows a pre-defined build lifecycle that may or may not fit one's project's needs, Gradle tasks are highly customizable and offer much more flexibility to Gradle's build process. It is represented internally in Gradle as an object and can be defined anywhere in the build script.

If the developer does not specify any task to be executed to the "gradle" command, Gradle will execute the default tasks listed in the Gradle build script for that project. Gradle provides a very scalable and high-performance builds for complex multi-module projects.

Gradle offers the incremental build feature that consists of only executing a task if it has been modified, if it has not, one will see an '*up-to-date*' tag in the command line interface when the application is run, thus making Gradle builds subsequently faster.

Furthermore, for multi-module projects, you can declare all the participating projects in the '*settings.gradle*' file placed near the build script, then proceed to configure the build script accordingly.

Following is examples of a simple task and a more "complex" task with dependencies:

- This is a simple task named "whoAreYou", that outputs a string to the console.

```
task whoAreYou {  
    doLast {  
        println "My name is jeff"  
    }  
}
```

- This task is named “multipleTasks” and is composed of different intra-dependant tasks called in a specified desired order of execution.

```
task multipleTasks {

    dependsOn 'clean'
    dependsOn 'build'
    dependsOn 'run'
    dependsOn "whoAreYou"
    dependsOn "Aname"
    dependsOn "task3"

    tasks.findByName('build').mustRunAfter 'clean'
    tasks.findByName("run").mustRunAfter("build")
    tasks.findByName("whoAreYou").mustRunAfter("run")
    tasks.findByName("Aname").mustRunAfter("whoAreYou")
    tasks.findByName("task3").mustRunAfter("Aname")

}
```

C. Plugins

As in Maven, Plugins extend the project’s capabilities. Plugins in Gradle are essentially a set of tasks; almost all useful tasks such as compiling, setting domain objects, setting up source files and so forth, are handled by plugins.

There are two types of plugins in Gradle - script and binary :

- A Script plugin is an additional build script that gives a declarative approach to manipulating the build :

For instance one could specify a task in the *aplugin.gradle* file in this way:

```
task Aplugin1 {
    doLast {
        println "I'm from a plugin"
    }
}
```

Then apply the plugin in the build.gradle script such as this :

```
apply from: 'aplugin.gradle'
```

- Binary plugins are the classes that implement the plugin interface and adopt a programmatic approach to manipulating the build :

The Java plugin adds Java compilation along with testing and bundling capabilities to a project.

```
apply plugin: JavaPlugin
```

Furthermore, One can also make custom plugins in a much easier way than maven.

Here is how one can make a simple custom plugin in Gradle build file :

```
apply plugin: HelloPlugin
class HelloPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.task('hello') << {
            println "Hello from the HelloPlugin"
        }
    }
}
```

D. Dependency management

Gradle offers a very flexible dependency management system and better manages version conflicts and transitive dependencies.

Dependencies are grouped into a set of different configurations. Configurations have names and can extend each other.

If one applies the Java plugin, they'll have *compile*, *testCompile*, *runtime* configurations available for grouping their dependencies. The *default* configuration extends "*runtime*".

In the following example, the developer is setting the repository, switching off transitive dependencies as well as declaring dependencies in various configurations: *testCompile*, *compile*, *runtime*.

They can disable transitive dependencies, either at configuration level or at dependency level such as this :

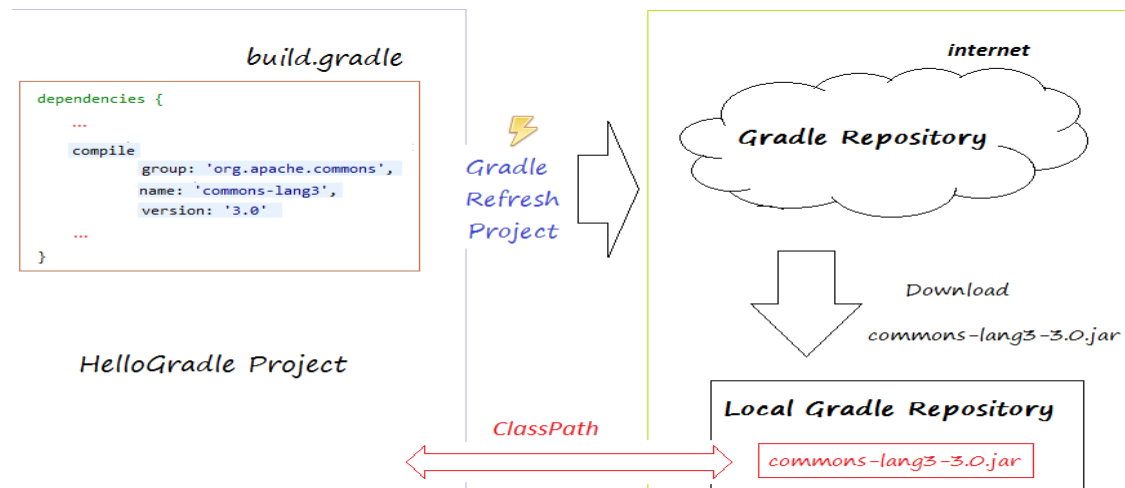
```

configurations {
    testCompile.exclude module: 'junit'
}

testCompile("org.springframework.batch:spring-batch-test:3.0.7.RELEASE"){
    exclude module: 'junit'
}

```

In usual scenarios, when one adds dependencies, the required library is seamlessly included in the build :



Nevertheless, when two dependencies have a common dependency but of different versions of the same library, a conflict arises. Consider the following example in which both mentioned libraries in the below code have a common internal dependency :

```

testImplementation 'junit:junit:4.12'
//(Depends on version 1.3)

androidTestImplementation 'androidx.test.ext:junit:1.1.0'
//(Depends on version 1.1)

```

Since both dependencies are internally using different versions of the same library, Gradle will include the highest version in the build to solve that issue.

Gradle offers more features in terms of handling dependency conflicts, such as excluding the conflicted module/library from one of the dependencies, explicitly defining the conflicted library in build.gradle or forcing resolution of the library. One should keep in mind that Gradle is still incubating and will further improve in its upcoming releases.

4. Conclusion

In Conclusion, Maven gives developers a conventional project structure, a rigid default build process as well as a tough to maintain xml build script. Even though it is possible to write custom maven lifecycle methods, it tends to quickly send the developer down a rabbit hole. Gradle's groovy build scripts are more readable, shorter, and very intuitive, especially for Java developers. Using Gradle, one can reduce project development time and increase productivity with the multitude of solutions Gradle offers for all the pre-existing recurrent issues encountered by its predecessors.

In essence, Maven solved Ant's issues and Gradle solved Maven's issues as well as all the other preceding tools such as Ivy, or Gant. One can alternatively define Gradle as a mutant that combines all the advantages of the older popular build tools.

