Build Tools - Laboratory 8

1. What are build tools?

Build tools are programs that automate the creation of executable applications from source code(e.g. *.apk* for an Android app). Building incorporates compiling, linking and packaging the code into a usable or **executable** form.

Basically build automation is the act of scripting or automating a wide variety of tasks that software developers do in their day-to-day activities, like:

- Downloading dependencies
- Compiling source code into binary code
- Packaging that binary code
- Running tests
- Deployment to production systems

2. Why should we use build tools?

In small projects, developers will often manually invoke the build process. This is not practical for larger projects, where it is very hard to keep track of what needs to be built, in what sequence and what dependencies there are in the building process. Using an automation tool allows the build process to be more **consistent**.

The advantages of build automation to software development projects include:

- A necessary precondition for continuous integration and continuous testing
- Improve product quality

- Accelerate the compile and link processing
- Eliminate redundant tasks
- Minimize "bad builds"
- Eliminate dependencies on key personnel
- Have history of builds and releases in order to investigate issues
- Save time and money - because of the reasons listed above

3. Short history of building software

Developers used build automation to call compilers and linkers from inside a build script versus attempting to make the compiler calls from the command line. It is simple to use the command line to pass a single source module to a compiler and then to a linker to create the final deployable object. However, when attempting to compile and link many source code modules, in a particular order, using the command line process is not a reasonable solution.

The make scripting language offered a better alternative. It allowed a build script to be written to call in a series, the needed compile and link steps to build a software application. GNU Make also offered additional features such as "makedepend" which allowed some source code dependency management as well as incremental build processing. This was the beginning of Build Automation. Its primary focus was on automating the calls to the compilers and linkers.

As the build process grew more complex, developers began adding pre and post actions around the calls to the compilers such as a check-out from version control to the copying of deployable objects to a test location. The term "build automation" now includes managing the pre and post compile and link activities as well as the compile and link activities.

4. Dependencies

With the continuously gaining popularity of modularity, the need for inter-project as well as external dependencies has grown, and as a result all the most commonly used build automation tools have stepped up to that challenge and have dependency management support; either out of the box, or through plugins. And to make things even simpler for the developer, they all use similar syntax for defining dependencies, as well as are all able to pull dependencies from the same public artifact repositories (e.g. *Maven Central*).

The basic syntax most commonly used for defining dependencies is adding a tuple of the group-id, artifact-id and requested version to the dependencies section of the build script. The build tool then tries to resolve these dependencies, by searching for them in its local and remote-defined repositories.

5. Java Build Tools

5.1. Ant

5.1.1. Short description

Apache Ant (“Another Neat Tool”) is a Java library used for automating build processes for Java applications. Additionally, Ant can be used for building non-Java applications. It was initially part of Apache Tomcat codebase and was released as a standalone project in 2000.

In many aspects, Ant is very similar to Make, and it’s simple enough so anyone can start using it without any particular prerequisites. Ant build files are written in XML, and by convention, they’re called *build.xml*.

Different phases of a build process are called “targets”.

5.1.2. Build script example

Here is an example of a build.xml file for a simple Java project with the HelloWorld main class:

```
<project>
    <target name="clean">
        <delete dir="classes" />
    </target>

    <target name="compile" depends="clean">
        <mkdir dir="classes" />
        <javac srcdir="src" destdir="classes" />
    </target>

    <target name="jar" depends="compile">
        <mkdir dir="jar" />
        <jar destfile="jar/HelloWorld.jar" basedir="classes">
            <manifest>
```

```
                <attribute name="Main-Class"
                  value="antExample.HelloWorld" />
            </manifest>
        </jar>
    </target>

    <target name="run" depends="jar">
        <java jar="jar/HelloWorld.jar" fork="true" />
    </target>
</project>
```

This build file defines four targets: `clean`, `compile`, `jar` and `run`. For example, we can compile the code by running:

```
ant compile
```

5.1.3. Ivy

At first, Ant had no built-in support for dependency management. However, as dependency management became a must in the later years, *Apache Ivy* was developed as a sub-project of the Apache Ant project. It's integrated with Apache Ant, and it follows the same design principles.

5.2. Maven

5.2.1. Convention over Configuration

Apache Maven is a software project management and comprehension tool. Based on the concept of a project object model (**POM**), Maven can manage a project's build, reporting and documentation from a central piece of information. Maven is a build automation tool used primarily for Java projects.

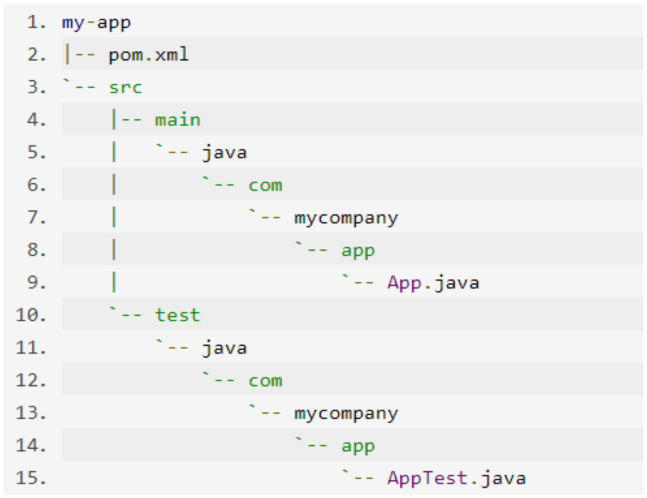
5.2.2. Standard Project Structure

Maven is an universal software project management, in order to get maven users familiar with maven projects, Maven defines some conventions or directory layouts. The complete standard directory layout can be found here.

Through those directory layouts Maven achieves an uniform way to organize projects and files inside of it. This a very good approach because you can work on several projects and you always will have the same project structure, so you will switch between projects and you don't have to expend time in order to learn how the project is organized.

In the image below is a simple project structure that has the 2 most important folders and the POM file:

- `src/main/java`: directory that contains the project source code
- `src/test/java`: directory that contains the test source
- `pom.xml`



```
my-app
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- com
    |           `-- mycompany
    |               `-- app
    |                   `-- App.java
    `-- test
        `-- java
            `-- com
                `-- mycompany
                    `-- app
                        `-- AppTest.java
```

Maven simple project structure

5.2.3. P.O.M.

5.2.3.1. XML

POM stands for Project Object Model. It is fundamental unit of work in Maven. It is an XML file that resides in the base directory of the project as pom.xml.

The POM contains information about the project and various configuration detail used by Maven to build the project(s).

5.2.3.2. Details

POM also contains the goals and plugins. While executing a task or goal, Maven looks for the POM in the current directory. It reads the POM, gets the needed configuration information, and then executes the goal. Some of the configuration that can be specified in the POM are following:

- project dependencies
- plugins
- goals
- build profiles
- project version

- developers
- mailing list

Before creating a POM, you should first decide the project group (groupId), its name (artifactId) and its version as these attributes help in uniquely identifying the project in repository.

```
<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.companyname.project-group</groupId>
   <artifactId>project</artifactId>
   <version>1.0</version>
</project>
```

It should be noted that there should be a single POM file for each project. All POM files require the project element and three mandatory fields: **groupId**, **artifactId**, **version**. Projects notation in repository is **groupId:artifactId:version**.

5.2.3.3. Inheritance

One powerful addition that Maven brings to build management is the concept of project inheritance. Although in build systems such as Ant, inheritance can certainly be simulated, Maven has gone the extra step in making project inheritance explicit to the project object model.

```
<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
                     https://maven.apache.org/xsd/maven-4.0.0.xsd">
 <modelVersion>4.0.0</modelVersion>

 <groupId>org.student.my-group</groupId>
 <artifactId>my-parent</artifactId>
 <version>2.0</version>
 <packaging>pom</packaging>
</project>
```

The *packaging* type required to be pom for *parent* and *aggregation* (multi-module) projects. These types define the goals bound to a set of lifecycle stages. For example, if packaging is *jar*, then the package phase will execute the *jar:jar* goal. Now we may add values to the parent POM, which will be inherited by its children. Most elements from the parent POM are inherited by its children.

5.2.4. Dependencies

5.2.4.1. Artifact coordinates (naming conventions)

Maven coordinates identify uniquely a project, a dependency, or a plugin defined in POM. Each entity is uniquely identified by the combination of a group identifier, an artifact identifier, and the version (and, of course, with the packaging and the classifier).

The group identifier is a way of grouping different Maven artifacts. For example, a set of artifacts produced by a company can be grouped under the same group identifier.

The artifact identifier is the way you identify an artifact, which could be JAR, WAR, or any other type of an artifact uniquely within a given group.

The version element lets you keep the same artifact in different versions in the same repository.

Examples:

- **groupId**: *org.apache.maven*, *org.apache.maven.plugins*, *org.apache.maven.reporting*
- **artifactId**: *maven*, *commons-math*
- **version**: *2.0*, *2.0.1*, *1.3.1*

5.2.4.2. Maven Repositories

A repository in Maven holds build artifacts and dependencies of varying types.

There are two types of repositories: **local** and **remote**. The local repository is a directory on the computer where Maven runs. It caches remote downloads and contains temporary build artifacts that you have not yet released.

Remote repositories refer to any other type of repository, accessed by a variety of protocols such as file:// and http://. These repositories might be a truly remote repository set up by a third party to provide their artifacts for downloading (for example, *repo.maven.apache.org* and *uk.maven.org* house Maven's central repository).

In general, you should not need to do anything with the local repository on a regular basis, except clean it out if you are short on disk space (or erase it completely if you are willing to download everything again).

For the remote repositories, they are used for both downloading and uploading (if you have the permission to do so).

```
<project>
  ...
  <repositories>
    <repository>
      <id>my-internal-site</id>
      <url>http://myserver/repo</url>
    </repository>
  </repositories>
  ...
</project>
```

5.2.4.3. Transitive Dependencies

Maven avoids the need to discover and specify the libraries that your own dependencies require by including transitive dependencies automatically.

This feature is facilitated by reading the project files of your dependencies from the remote repositories specified. In general, all dependencies of those projects are used in your project, as are any that the project inherits from its parents, or from its dependencies, and so on.

There is no limit to the number of levels that dependencies can be gathered from. A problem arises only if a cyclic dependency is discovered.

With transitive dependencies, the graph of included libraries can quickly grow quite large.

Although transitive dependencies can implicitly include desired dependencies, it is a good practice to explicitly specify the dependencies you are directly using in your own source code. This best practice proves its value especially when the dependencies of your project changes their dependencies.

For example, assume that your project A specifies a dependency on another project B, and project B specifies a dependency on project C. If you are directly using components in project C, and you don't specify project C in your project A, it may cause build failure when project B suddenly updates/removes its dependency on project C.

Another reason to directly specify dependencies is that it provides better documentation for your project: one can learn more information by just reading the POM file in your project.

5.2.5. Plugins & Goals

Maven consists of a core engine which provides basic project-processing capabilities and build-process management, and a host of plugins which are used to execute the actual build tasks.

"Maven" is really just a core framework for a collection of *Maven Plugins*. In other words, plugins are where much of the real action is performed, plugins are used to: create jar files, create war files, compile code, unit test code, create project documentation, and on and on. Almost any action that you can think of performing on a project is implemented as a Maven plugin.

Plugins are the central feature of Maven that allow for the reuse of common build logic across multiple projects. They do this by executing an "action" (i.e. creating a JAR file or compiling unit tests) in the context of a project's description - the Project Object Model (POM). Plugin behavior can be customized through a set of unique parameters which are exposed by a description of each plugin goal (or Mojo).

One of the simplest plugins in Maven is the *Clean Plugin*. The Maven Clean plugin (*maven-clean-plugin*) is responsible for removing the target directory of a Maven project. When you run "*mvn clean*", Maven executes the "clean" goal as defined in the Clean plug-in, and the target directory is removed. The Clean plugin defines a parameter which can be used to customize plugin behavior, this parameter is called outputDirectory and it defaults to *${project.build.directory}*.

5.2.6. Lifecycle

5.2.6.1. Phases & Goals

Maven is based around the central concept of a *build lifecycle*. What this means is that the process for building and distributing a particular artifact (project) is clearly defined.

For the person building a project, this means that it is only necessary to learn a small set of commands to build any Maven project, and the POM will ensure they get the results they desired.

There are three built-in build lifecycles: default, clean and site. The default lifecycle handles your project deployment, the clean lifecycle handles project cleaning, while the site lifecycle handles the creation of your project's site documentation.

5.2.6.2. Maven default lifecycle (explain the most important phases)

The default lifecycle comprises of the following phases (for a complete list of the lifecycle phases, refer to the Lifecycle Reference):

- 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. These tests should not require the code be packaged or deployed
- package - take the compiled code and package it in its distributable format, such as a JAR.
- verify - run any checks on results of integration tests to ensure quality criteria are met
- install - install the package into the local repository, for use as a dependency in other projects locally
- deploy - done in the build environment, copies the final package to the remote repository for sharing with other developers and projects.

These lifecycle phases are executed sequentially to complete the default lifecycle. Given the lifecycle phases above, this means that when the default lifecycle is used, Maven will first validate the project, then will try to compile the sources, run those against the tests, package the binaries (e.g. jar), run integration tests against that package, verify the integration tests, install the verified package to the local repository, then deploy the installed package to a remote repository.

5.2.6.3. Maven Clean Lifecycle

Maven clean lifecycle phases are:

1. pre-clean
2. clean
3. post-clean

Calling one phase of the clean lifecycle results in the execution of all phases up to an including that phase. So, if we perform a "mvn clean", we will execute the pre-clean and the clean phases. If we perform a "mvn post-clean", we will execute the pre-clean, clean, and post-clean phases.

The maven "*clean:clean*" goal is typically bound to the clean phase. This goal 'cleans' the project's build (usually 'target') directory, which typically involves deleting old files.

5.2.7. Multi-module projects

A multi-module project is built from an aggregator POM that manages a group of submodules. In most cases, the aggregator is located in the project's root directory and must have packaging of type *pom*.

Now, the submodules are regular Maven projects, and they can be built separately or through the aggregator POM.

By building the project through the aggregator POM, each project that has packaging type different than pom will result in a built archive file.

The significant advantage of using this approach is that we may **reduce duplication**.

Let's say we have an application which consists of several modules, let it be a front-end module and a back-end module. Now, we work on both of them and change functionality which affects the two. In that case, without a specialized build tool, we'll have to build both components separately or write a script which would compile the code, run tests and show the results. Then, after we get even more modules in the project, it will become harder to manage and maintain.

Besides, in the real world, projects may need certain Maven plugins to perform various operations during build lifecycle, share dependencies and profiles or include other BOM projects.

Therefore, when leveraging multi-modules, we can **build our application's modules in a single command** and if the order matters, Maven will figure this out for us. Also, we can share a vast amount of configuration with other modules.

5.2.8. The Maven Wrapper (mvnw)

The Maven Wrapper is an easy way to ensure a user of your Maven build has everything necessary to run your Maven build. Why might this be necessary? Maven to date has been very stable for users, is available on most systems or is easy to procure: but with many of the recent changes in Maven it will be easier for users to have a fully encapsulated build setup provided by the project.

The easiest way to add a Maven wrapper to your project is to use the Maven Wrapper plugin. Just *cd* into your projects folder and call `mvn -N io.takari:maven:wrapper`. This call adds a *.mvn* folder containing a wrapper jar, some properties and so on. It also adds two scripts which will replace the *mvn* call via console. You just have to use these generated scripts instead of the mvn calls on your terminal. The scripts can be used the same way maven is used via command line. A clean and install call via Maven wrapper for example would look like: `./mvnw clean install`

5.3. Gradle

5.3.1. Convention over Configuration

Convention over configuration is a software engineering paradigm that allows a tool or framework to make an attempt at decreasing the number of decisions the user has to make without losing its flexibility. What does that mean for Gradle plugins? Gradle plugins can provide users with sensible defaults and standards (conventions) in a certain context. Let's take the Java plugin as an example.

- It defines the directory *src/main/java* as the default source directory for compilation.

- The output directory for compiled source code and other artifacts (like the JAR file) is build.

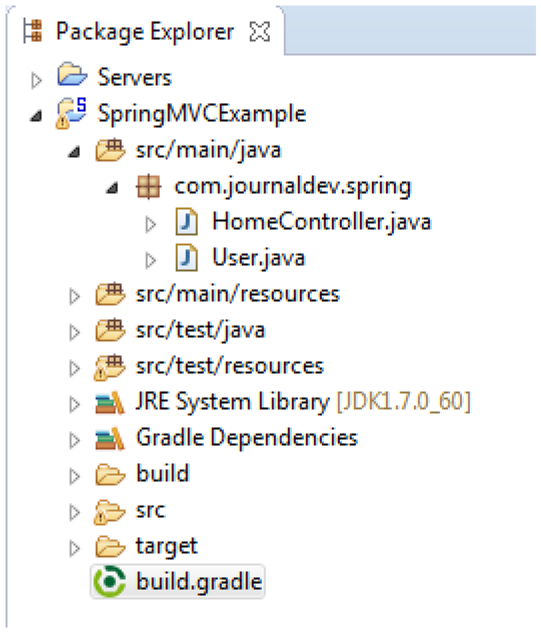
As long the user of the plugin does not prefer to use other conventions, no additional configuration is needed in the consuming build script. It simply works out-of-the-box. However, if the user prefers other standards, then the default conventions can be reconfigured. You get the best of both worlds.

5.3.2. Project Structure

Whenever you add a plugin to your build it assume a certain setup of your Java project (similar to Maven). Take a look into the following directory structure:

- *src/main/java* contains the Java source code
- *src/test/java* contains the Java tests
- *build.gradle*

If you follow this setup, the following build file is sufficient to compile, test, and bundle a Java project.



5.3.3. Easy Migration

5.3.3.1. From Maven

To migrate from a Maven project to a Gradle one it's easy and simple with some gradle commands. Later on you can enhance your build.gradle as per the project need. The steps are the following:

- Check out (or copy) your code into a new directory. This is not strictly necessary but it makes things cleaner.
- `cd` into the directory that contains your `pom.xml`
- Run the command: `run gradle init`. This will create a new gradle project creating a `build.gradle` file based on your `pom.xml`.
- Create a new IntelliJ project as follows: *File -> New -> Project From Existing Sources*. Select the `build.gradle` file you created in the previous step and choose the defaults

That's really all there is to it. A brand new Gradle project.

5.3.3.2. Support for Maven dependencies

If you want to include files you already have in your existing Maven .m2 repo add mavenLocal() as a dependency. Your dependencies get downloaded here (the equivalent of *.m2*):
.gradle/caches/modules-2/files-2.1

5.3.4. Project

5.3.4.1. Groovy

The Groovy plugin extends the Java plugin to add support for Groovy projects. It can deal with Groovy code, mixed Groovy and Java code, and even pure Java code (although we don't necessarily recommend to use it for the latter). The plugin supports *joint compilation*, which allows you to freely mix and match Groovy and Java code, with dependencies in both directions. For example, a Groovy class can extend a Java class that in turn extends a Groovy class. This makes it possible to use the best language for the job, and to rewrite any class in the other language if needed.

To use the Groovy plugin, include the following in your *build.gradle* file:

```
plugins {
    id 'groovy'
}
```

The Groovy plugin adds the following tasks to the project:

- compileGroovy — GroovyCompile
  Compiles production Groovy source files.

- compileTestGroovy — GroovyCompile
  Compiles test Groovy source files.

- compileSourceSetGroovy — GroovyCompile
  Compiles the given source set's Groovy source files.

- groovydoc — Groovydoc
  Generates API documentation for the production Groovy source files.

5.3.4.2. Groovy based DSL

DSL stands for Domain Specific Language. It is not a new concept; people have been creating DSLs for a long time now. The reason I am writing this article is to hopefully show you a couple tricks that you may not be aware of.

There is a larger definition of a domain specific language, however in the context of Groovy code, a DSL is a way of creating APIs that leverages Groovy's closures to create an easy way to build complex data. To understand how a DSL works, you must understand how closures work.

If you are familiar with Groovy at all, then you are also likely familiar with closures. Closures are the gateway to functional programming in Groovy. They represent executable pieces of code that may be passed as arguments to other methods.

```
void someMethod(Closure c) {
    println "Inside someMethod"
    c.call()
}
Closure closure = {
    println "I'm inside a closure"
}
someMethod(closure)

// The following will be output to the console
Inside someMethod
I'm inside a closure
```

Another example of how DSL can be implemented in Groovy:

```
class EmailDsl {
   String toText
   String fromText
   String body

   /**
   * This method accepts a closure which is essentially the DSL. Delegate
the
   * closure methods to
   * the DSL class so the calls can be processed
   */

   def static make(closure) {
      EmailDsl emailDsl = new EmailDsl()
      // any method called in closure will be delegated to the EmailDsl
class
      closure.delegate = emailDsl
      closure()
   }

   /**
   * Store the parameter as a variable and use it later to output a memo
   */

   def to(String toText) {
      this.toText = toText
   }

   def from(String fromText) {
      this.fromText = fromText
   }

   def body(String bodyText) {
      this.body = bodyText
   }
}

EmailDsl.make {
   to "Nirav Assar"
   from "Barack Obama"
   body "How are things? We are doing well. Take care"
}
```

When we run the above program, we will get the following result:

```
How are things? We are doing well. Take care
```

5.3.4.3. Artifact coordinates (naming conventions)

Gradle has several conventions around the naming of archives and where they are created based on the plugins your project uses. The main convention is provided by the Base Plugin, which defaults to creating archives in the $buildDir/distributions directory and typically uses archive names of the form *[projectName]-[version].[type]*.

The following example comes from a project named zipProject, hence the myZip task creates an archive named zipProject-1.0.zip:

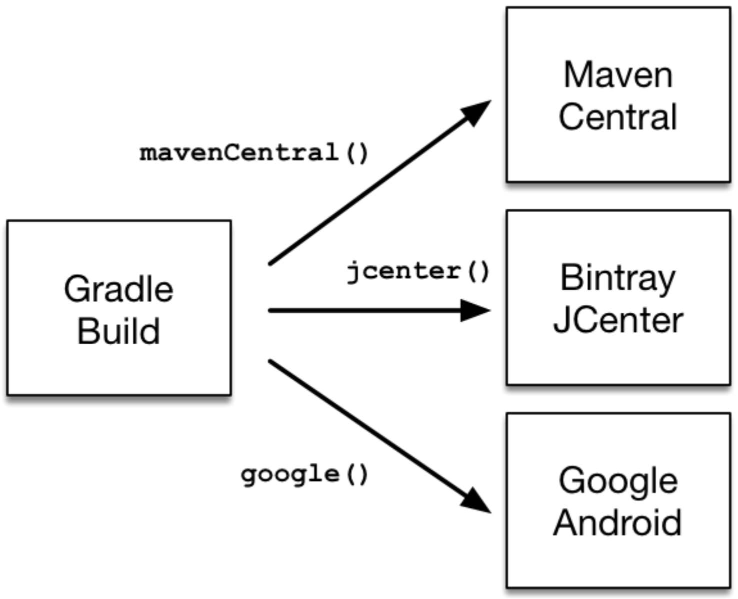
```
plugins {
    id 'base'
}
```

```
version = 1.0

task myZip(type: Zip) {
    from 'somedir'

    doLast {
        println archiveFileName.get()
        println relativePath(destinationDirectory)
        println relativePath(archiveFile)
    }
}
```

5.3.4.4. Supported Repositories

Gradle can resolve dependencies from one or many repositories based on Maven, Ivy or flat directory formats. Check out the full reference on all types of repositories for more information.

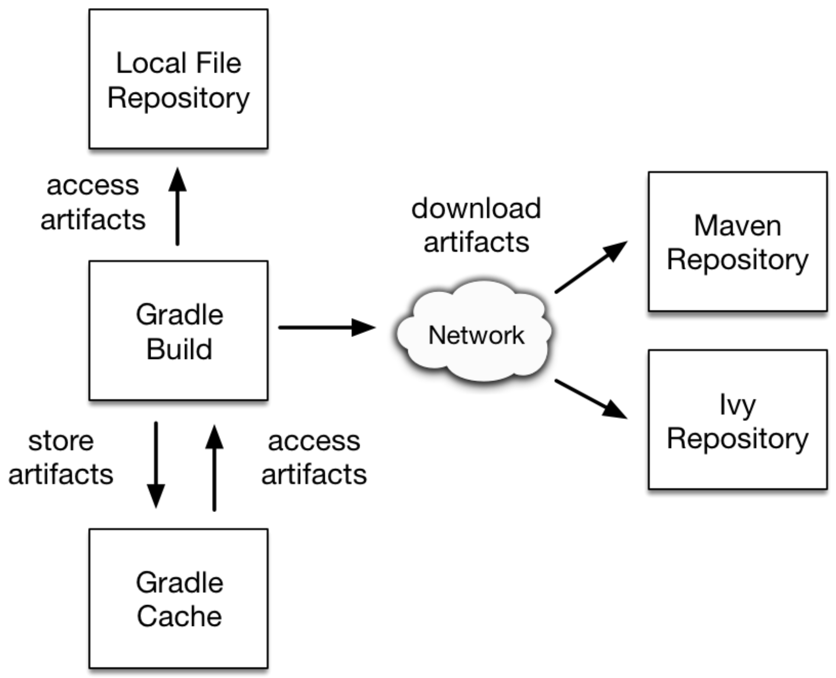
Organizations building software may want to leverage public binary repositories to download and consume open source dependencies. Popular public repositories include Maven Central, Bintray JCenter and the Google Android repository. Gradle provides built-in shortcut methods for the most widely-used repositories.



Under the covers Gradle resolves dependencies from the respective URL of the public repository defined by the shortcut method. All shortcut methods are available via the *RepositoryHandler* API. Alternatively, you can spell out the URL of the repository for more fine-grained control.

5.3.4.5. Transitive Dependencies

Gradle has built-in support for dependency management and lives up the task of fulfilling typical scenarios encountered in modern software projects. We'll explore the main concepts with the help of an example project. The illustration below should give you an rough overview on all the moving parts.



By default, Gradle dependencies are transitive. Transitive dependencies are described more in Maven Repositories, but for now, all you need to know is that one module may depend on other modules, and Gradle can discover those dependencies-of-dependencies when it resolves the declared dependency against a repository. This is almost always an enormous time-saver, but sometimes it can create problems. If you depend on version 1 of module A and version 2 of module B, and module A transitively depends on version 3 of module B, you may not want Gradle to resolve that final dependency. The wrong version of a JAR file might make it into your compile or runtime classpath.

5.3.4.6. Conflicts and exclusions

Managing dependencies in a project can be challenging. Gradle resolves version conflicts by picking the highest version of a module. Build scans and the dependency insight report are immensely helpful in identifying why a specific version was selected. If the resolution result is not satisfying (e.g. the selected version of a module is too high) or it fails (because you configured ResolutionStrategy.failOnVersionConflict()) you have the following possibilities to fix it.

- Configuring any dependency (transitive or not) as forced. This approach is useful if the dependency in conflict is a transitive dependency. See Enforcing a particular dependency version for examples.

- Configuring dependency resolution to prefer modules that are part of your build (transitive or not). This approach is useful if your build contains custom forks of modules (as part of multi-project builds or as include in composite builds). See ResolutionStrategy.preferProjectModules() for more information.

- Using dependency resolve rules for fine-grained control over the version selected for a particular dependency.

5.3.5. Plugins & Tasks

A Gradle plugin packages up reusable pieces of build logic, which can be used across many different projects and builds. Gradle allows you to implement your own plugins, so you can reuse your build logic, and share it with others.

You can implement a Gradle plugin in any language you like, provided the implementation ends up compiled as JVM bytecode. In our examples, we are going to use Groovy as the implementation language. Groovy, Java or Kotlin are all good choices as the language to use to implement a plugin, as the Gradle API has been designed to work well with these languages. In general, a plugin implemented using Java or Kotlin, which are statically typed, will perform better than the same plugin implemented using Groovy.

5.3.6. The Java Plugin tasks

The Java plugin adds Java compilation along with testing and bundling capabilities to a project. It serves as the basis for many of the other JVM language Gradle plugins. You can find a comprehensive introduction and overview to the Java Plugin in the Building Java Projects chapter.

To use the Java plugin, include the following in your build script:

```
plugins {
    id 'java'
}
```

The Java plugin adds a number of tasks to your project, as shown below:

- compileJava — JavaCompile
  Compiles production Java source files using the JDK compiler.

- processResources — Copy
  This is an aggregate task that just depends on other tasks. Other plugins may attach additional compilation tasks to it.

- compileTestJava — JavaCompile
  Compiles test Java source files using the JDK compiler.

- processTestResources — Copy
  Copies test resources into the test resources directory.

- testClasses
  This is an aggregate task that just depends on other tasks. Other plugins may attach additional test compilation tasks to it.

- jar — Jar
  Assembles the production JAR file, based on the classes and resources attached to the main source set.

- javadoc — Javadoc
  Generates API documentation for the production Java source using Javadoc.

- test — Test
  Runs the unit tests using JUnit or TestNG.

- uploadArchives — Upload
  Uploads artifacts in the archives configuration — including the production JAR file — to the configured repositories.

- clean — Delete
  Deletes the project build directory.

- cleanTaskName — Delete
  Deletes files created by the specified task. For example, cleanJar will delete the JAR file created by the jar task and cleanTest will delete the test results created by the test task

5.3.7. Multi-module projects

Although we can create a working application by using only one module, sometimes it is wiser to divide our application into multiple smaller modules. A multi-project build in gradle consists of one root project, and one or more subprojects that may also have subprojects.

5.3.8. The Gradle Wrapper (gradlew)

Gradle wrapper allows you to run a Gradle task without requiring that Gradle is installed on your system. Using the wrapper is the recommended way of executing any Gradle build.

When someone runs the build for the first time, they run *gradlew* which is an automatically generated shell script designed to download Gradle. As the build manager, you generate this file from your Gradle build so that you don't have to write instructions for installing Gradle.

In a nutshell you gain the following benefits:

- Standardizes a project on a given Gradle version, leading to more reliable and robust builds.

- Provisioning a new Gradle version to different users and execution environments (e.g. IDEs or Continuous Integration servers) is as simple as changing the Wrapper definition.

So how does it work? For a user there are typically three different workflows:

- You set up a new Gradle project and want to add the Wrapper to it.

- You want to run a project with the Wrapper that already provides it.

- You want to upgrade the Wrapper to a new version of Gradle.

6. Conclusion

In small projects, developers will often manually invoke the build process. This is not practical for larger projects, where it is very hard to keep track of what needs to be built, in what sequence and what dependencies there are in the building process. Using an automation tool allows the build process to be more consistent.

In the end, what you choose will depend primarily on what you need. Gradle is more powerful. However, there are times that you really do not need most of the features and functionalities it offers. Maven might be best for small projects, while Gradle is best for bigger projects. If you've been working with Maven and find that your project has outgrown it, it is possible to migrate from Maven to Gradle.