Gradle: build, dependency management, wrapper and testing

In these notes, extracts from the Gradle documentation are summarised around four topics:

- Build lifecycle
- Dependency management
- Gradle wrapper
- Testing Java applications

Build lifecycle

In Gradle you can define tasks and dependencies between tasks. Gradle guarantees that these tasks are executed in the order of their dependencies, and that each task is executed only once. These tasks form a Directed Acyclic Graph. There are build tools that build up such a dependency graph as they execute their tasks. Gradle builds the complete dependency graph before any task is executed. This lies at the heart of Gradle and makes many things possible which would not be possible otherwise.

Your build scripts configure this dependency graph. Therefore they are strictly speaking build configuration scripts.

A Gradle build has three distinct phases:

- **Initialization**: Gradle supports single and multi-project builds. During the initialization phase, Gradle determines which projects are going to take part in the build, and creates a Project instance for each of these projects.
- **Configuration**: During this phase the project objects are configured. The build scripts of all projects which are part of the build are executed.
- **Execution**: Gradle determines the subset of the tasks, created and configured during the configuration phase, to be executed. The subset is determined by the task name arguments passed to the gradle command and the current directory. Gradle then executes each of the selected tasks.

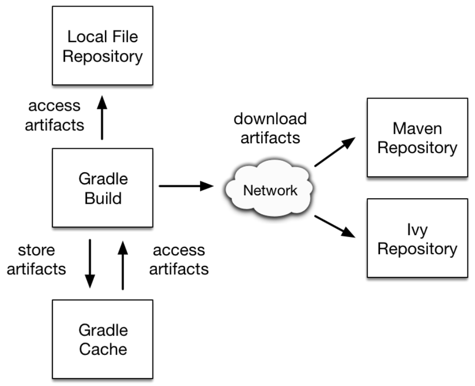
For more information about these phases check :books: the documentation.

Beside the build script files, Gradle defines a settings file. The settings file is determined by Gradle via a naming convention. The default name for this file is `settings.gradle`. The settings file is executed during the initialization phase. A multi-project build must have a `settings.gradle` file in the root project of the multi-project hierarchy. However, for a single-project build, a settings file is optional.

Dependency Management

When we develop software, we normally either reuse functionality that is available in the form of libraries, or build a a modularised system out of smaller components. **Dependency management** is a technique for declaring dependencies between components and libraries in a system.

In the following, we will use an example extracted from :books: documentation on dependency management:



The example project builds Java source code. Some of the Java source files import classes from Google Guava, a open-source library providing a wealth of utility functionality. In addition to Guava, the project needs the JUnit libraries for compiling and executing test code.

Guava and JUnit represent the dependencies of this project. A build script developer can declare dependencies for different scopes e.g. just for compilation of source code or for executing tests. In Gradle, the scope of a dependency is called a **configuration**.

Often times dependencies come in the form of modules. You'll need to tell Gradle where to find those modules so they can be consumed by the build. The location for storing modules is called a **repository**. By declaring repositories for a build, Gradle will know how to find and retrieve modules. Repositories can come in different forms: as local directory or a remote repository. The reference on repository types provides a broad coverage on this topic.

At runtime, Gradle will locate the declared dependencies if needed for operating a specific task. The dependencies might need to be downloaded from a remote repository, retrieved from a local directory or requires another project to be built in a multi-project setting. This process is called **dependency resolution**.

Once resolved, the resolution mechanism stores the underlying files of a dependency in a local cache, also referred to as the **dependency cache**. Future builds reuse the files stored in the cache to avoid unnecessary network calls.

Modules can provide additional metadata. Metadata is the data that describes the module in more detail e.g. the coordinates for finding it in a repository, information about the project, or its authors. As part of the metadata, a module can define that other modules are needed for it to work properly. For example, the JUnit 5 platform module also requires the platform commons module. Gradle automatically resolves those additional modules, so called **transitive dependencies**.

Concepts

- **Configuration**: A configuration is a named set of dependencies grouped together for a specific goal. For example the implementation configuration represents the set of dependencies required to compile a project. Configurations provide access to the underlying, resolved modules and their artifacts.
- A **dependency** is a pointer to another piece of software required to build, test or run a module. For more information, see Declaring Dependencies.
- **Module**: A piece of software that evolves over time e.g. Google Guava. Every module has a name. Each release of a module is optimally represented by a module version. For convenient consumption, modules can be hosted in a repository.
- **Module metadata**: Releases of a module can provide metadata. Metadata is the data that describes the module in more detail e.g. the coordinates for locating it in a repository, information about the project or required transitive dependencies. In Maven the metadata file is called .pom, in Ivy it is called ivy.xml.
- **Module version**: A module version represents a distinct set of changes of a released module. For example 18.0 represents the version of the module with the coordinates com.google:guava:18.0. In practice there's no limitation to the scheme of the module version. Timestamps, numbers, special suffixes like -GA are all allowed identifiers. The most widely-used versioning strategy is semantic versioning.
- **Repository**: A repository hosts a set of modules, each of which may provide one or many releases indicated by a module version. The repository can be based on a binary repository product (e.g. Artifactory or Nexus) or a directory structure in the filesystem. For more information, see Declaring Repositories.
- **Transitive dependencies**: A module can have dependencies on other modules to work properly, so-called transitive dependencies. Releases of a module hosted on a repository can provide metadata to declare those transitive dependencies. By default, Gradle resolves transitive dependencies automatically but it can be configured.

Advanced concepts, like configuration management, dependency constraints, resolution rules, transitive dependency management, have been skipped. Feel free to explore them in :books: the documentation for more information.

Declaring repositories

Gradle can resolve dependencies from one or many repositories based on Maven, Ivy or flat directory formats.

- Declaring a publicly-available repository: 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. To declare Maven Central as repository, add this code to your build script:

```
repositories {
    mavenCentral()
}
```

- Declaring a custom repository by URL: Most enterprise projects set up a binary repository available only within an intranet. In-house repositories enable teams to publish internal binaries, setup user management and security measure and ensure uptime and availability. Specifying a custom URL is also helpful if you want to declare a less popular, but publicly-available repository. Add the following code to declare an in-house repository for your build reachable through a custom URL.

```
repositories {
    maven {
        url "http://repo.mycompany.com/maven2"
    }
}
```

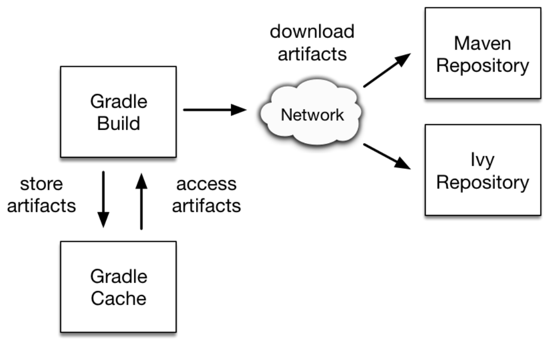
- Declaring multiple repositories: You can define more than one repository for resolving dependencies. Declaring multiple repositories is helpful if some dependencies are only available in one repository but not the other. You can mix any type of repository.

```
repositories {
    jcenter()
    maven {
        url "https://maven.springframework.org/release"
    }
    maven {
        url "https://maven.restlet.com"
    }
}
```

Gradle takes your dependency declarations and repository definitions and attempts to download all of your dependencies by a process called **dependency resolution**. The order of declaration determines how Gradle will check for dependencies at runtime. If Gradle finds a module descriptor in a particular repository, it will attempt to download all of the artifacts for that module from the same repository. You can learn more about the inner workings of :books: Gradle's resolution mechanism.

Declaring dependencies

Modern software projects rarely build code in isolation. Projects reference modules for the purpose of reusing existing and proven functionality. Upon resolution, selected versions of modules are downloaded from dedicated repositories and stored in the dependency cache to avoid unnecessary network traffic.



Gradle builds can declare dependencies on modules hosted in repositories, files and other Gradle projects. A typical example for such a library in a Java project is the Spring framework, which we will start using in the following sprint. The following code snippet declares a compile-time dependency on the Spring web module by its coordinates: `org.springframework:spring-web:5.0.2.RELEASE`. Gradle resolves the module including its transitive dependencies from the Maven Central repository and uses it to compile Java source code. The version attribute of the dependency coordinates points to a concrete version indicating that the underlying artifacts do not change over time. The use of concrete versions ensure reproducibility for the aspect of dependency resolution.

```
apply plugin: 'java-library'

repositories {
    mavenCentral()
}

dependencies {
    implementation 'org.springframework:spring-web:5.0.2.RELEASE'
}
```

A Gradle project can define other types of repositories hosting modules. You can learn more about the syntax and API in the section on declaring repositories, above. Refer to the chapter on the Java Plugin for a deep dive on declaring dependencies for a Java project.

For more information on how to declare either dependencies without a concrete version (e.g. `implementation 'org.springframework:spring-web'`) or dynamic dependencies (e.g. `implementation 'org.springframework:spring-web:5.+'`), explore the :books: documentation.

Gradle wrapper

The recommended way to execute any Gradle build is with the help of the Gradle Wrapper (in short just "Wrapper"). The Wrapper is a script that invokes a declared version of Gradle, downloading it beforehand if necessary. As a result, developers can get up and running with a Gradle project quickly without having to follow manual installation processes saving your company time and money.

In a nutshell you gain the following benefits:

- Standardizes a project on a given Gradle version, leading to more reliable and robust builds. Whoever downloads the project will use the same gradle version.
- Provisioning a new Gradle version to different users and execution environment (e.g. IDEs or Continuous Integration servers) is as simple as changing the Wrapper definition.

To create a Gradle project with a wrapper, create it with `Gradle Buildship` from the IDE and select `Gradle wrapper`.

From that point onwards, you can use `./gradlew` (`gradlew.bat` in Windows) instead of `gradle` for executing tasks from your command line. This is not even necessary from the `Gradle Tasks` view in the IDE.

Upgrading the wrapper version can be done as explained in the :books: documentation.

Gradle for testing Java Projects

Testing on the JVM is a rich subject matter. There are many different testing libraries and frameworks, as well as many different types of test. All need to be part of the build, whether they are executed frequently or infrequently. This section is a brief introduction to automating testing of Java applications with Gradle, which is explained in more detail in the :books: documentation.

Test execution

All JVM testing revolves around a single task type: Test. This runs a collection of test cases using any supported test library — JUnit, JUnit Platform or TestNG — and collates the results. You can then turn those results into a report.

When you're using a JVM language plugin — such as the Java Plugin — you will automatically get the following:

- A dedicated test source set for unit tests
- A test task of type Test that runs those unit tests

All you need to do in most cases is configure the appropriate compilation and runtime dependencies and add any necessary configuration to the test task. The following example shows a simple setup that uses JUnit 4.x:

```
dependencies {
    testImplementation 'junit:junit:4.12'
}
```

By default, Gradle will run all tests that it detects, which it does by inspecting the compiled test classes. This detection uses different criteria depending on the test framework used. For JUnit, Gradle scans for both JUnit 3 and 4 test classes. A class is considered to be a JUnit test if it:

- Ultimately inherits from `TestCase` or `GroovyTestCase`
- Is annotated with `@RunWith`
- Contains a method annotated with `@Test` or a super class does

Test filtering

It's a common requirement to run subsets of a test suite, such as when you're fixing a bug or developing a new test case. Gradle provides two mechanisms to do this:

- Filtering (the preferred option)
- Test inclusion/exclusion

See more information on test filtering in the :books: documentation.

Test reporting

The `Test` task generates the following results by default:

- An HTML test report under `build/reports/tests/test`
- XML test results in a format compatible with the Ant JUnit report task — one that is supported by many other tools, such as CI servers - under `build/test-results/test`

- An efficient binary format of the results used by the Test task to generate the other formats

In most cases, you'll work with the standard HTML report, which automatically includes the results from all your Test tasks, even the ones you explicitly add to the build yourself. For example, if you add a Test task for integration tests, the report will include the results of both the unit tests and the integration tests if both tasks are run.

For configuring test reporting, check the :books: documentation

Documentation

- Dependency management in Gradle
- Gradle wrapper
- Testing in Java and JVM projects: includes detailed information on configuration and other topics not mentioned above (test grouping, integration testing, skipping tests…)

References

References