Gradle Guides (https://guides.gradle.org)

Creating Multi-project Builds

Schalk Cronjé

Table of Contents

Multi-project builds helps with modularization. It allows a person to concentrate on one area of work in a larger project, while Gradle takes care of dependencies from other parts of the project.

What you'll build

You'll build a greeting app which also includes documentation. In the process you will create a Groovy-based library project, an Asciidoctor-based documentation project and a Java distributable command-line application. You will see how to connect these projects together to create a final product.

What you'll need

- About 21 minutes
- A text editor
- A command prompt
- The Java Development Kit (JDK), version 1.7 or higher
- A Gradle distribution (https://gradle.org/install), version 3.5 or better

Create a root project

The first step is to create a folder for the new project and add a Gradle Wrapper (https://docs.gradle.org/3.5/userguide/gradle_wrapper.html#sec:wrapper_generation) to the project. If you use the Build Init plugin (https://docs.gradle.org/3.5/userguide/build_init_plugin.html) then the necessary `settings.gradle` and `build.gradle` will also be added.

```
$ mkdir creating-multi-project-builds
$ cd creating-multi-project-builds
$ gradle init  1   2

:wrapper
:init

BUILD SUCCESSFUL
```

1 Use of `init` will create skeleton `build.gradle` and `settings.gradle` files which can be customized.

2 This allows a version of Gradle to be locked to a project, and afterwards you can use `./gradlew` (or `gradlew.bat` on Windows) instead of `gradle`.

Open `settings.gradle`. There will be a number of auto-generated comments which you can remove, leaving only:

settings.gradle

GROOVY
```
rootProject.name = 'creating-multi-project-builds'
```

Great! You are now ready to start development. (Remember to save your file).

Configure from above

In a multi-project you can use the top-level build file (also known as the root project) to configure as much commonality as possible, leaving sub-projects to customize only what is necessary for that subproject.

When using the `init` task with no parameters, Gradle generates a `build.gradle` file with a basic Java layout in a comment block. Open `build.gradle` and replace its contents with:

build.gradle

GROOVY
```
allprojects {
    repositories {
        jcenter()  1
    }
}
```

1 Add the JCenter repository to all projects.

The `allprojects` block is used to add configuration items that will apply to all sub-projects as well as the root project. In a similar fashion, the `subprojects` block can be used to add configurations items for all sub-projects only. You can use these two blocks as many times as you want in the root project.

Now set the version for each of the modules which you will be adding, via the `subproject` block in the top-level build file as follows

build.gradle

GROOVY

```groovy
subprojects {
    version = '1.0'
}
```

Add a Groovy library subproject

Create the directory for your library subproject.

```
$ mkdir greeting-library
```

Create a `build.gradle` and add the basic Groovy library project content.

(Don’t worry if you’ve never built a Groovy library before. The complete contents will be supplied here. If you are interested in the details, you might want to look at the Getting Started Guide for Building Groovy Libraries (https://guides.gradle.org/building-groovy-libraries) in the User Manual).

greeting-library/build.gradle

GROOVY

```
apply plugin : 'groovy'

dependencies {
    compile 'org.codehaus.groovy:groovy:2.4.10'

    testCompile 'org.spockframework:spock-core:1.0-groovy-2.4', {
        exclude module : 'groovy-all'
    }
}
```

Now edit `settings.gradle` *in the top-level project* to make the new Groovy library project part of the multi-project build.

settings.gradle

GROOVY

```
include 'greeting-library'
```

Finally, create the `src/main/groovy` folder under `greeting-library and add the package folder `greeter`.

```
$ mkdir -p src/main/groovy/greeter
$ mkdir -p src/test/groovy/greeter
```

Add a `GreetingFormatter` class to the `greeter` package in `src/main/groovy`.

greeting-library/src/main/groovy/greeter/GreetingFormatter.groovy

GROOVY

```
package greeter

import groovy.transform.CompileStatic

@CompileStatic
class GreetingFormatter {
    static String greeting(final String name) {
        "Hello, ${name.capitalize()}"
    }
}
```

Add a Spock Framework test called `GreetingFormatterSpec` in the `greeter` package under `src/test/groovy` .

greeting-library/src/test/groovy/greeter/GreetingFormatterSpec.groovy

GROOVY

```
package greeter

import spock.lang.Specification

class GreetingFormatterSpec extends Specification {

    def 'Creating a greeting'() {

        expect: 'The greeeting to be correctly capitalized'
        GreetingFormatter.greeting('gradlephant') == 'Hello, Gradlephant'

    }
}
```

Run `./gradlew build` from the top-level project directory.

```
$ ./gradlew build

:greeting-library:compileJava NO-SOURCE
:greeting-library:compileGroovy
:greeting-library:processResources NO-SOURCE
:greeting-library:classes
:greeting-library:jar
:greeting-library:assemble
:greeting-library:compileTestJava NO-SOURCE
:greeting-library:compileTestGroovy
:greeting-library:processTestResources NO-SOURCE
:greeting-library:testClasses
:greeting-library:test
:greeting-library:check
:greeting-library:build

BUILD SUCCESSFUL
```

Gradle automatically detected that there is a `build` task in `greeting-library` and executed it. This is one of the powerful features of a Gradle multi-project build. When tasks in sub-projects have the same names as those in the top-level project, then maintenance of the build will be easier, and Gradle is able to execute the same tasks in each project by specifying the common task name at the top level.

A single subproject does not truly make a multi-project build, however. Therefore the next step is to add a sub-project which will consume this library,

Add a Java application sub-project

Create a folder in the root project for the sub-project which will contain the application.

```
$ mkdir greeter
```

greeter/build.gradle

GROOVY

```
apply plugin : 'java'  1
apply plugin : 'application'  2
```

1 This is a Java project, so the Java plugin (https://docs.gradle.org/3.5/userguide/java_plugin.html) is required.

2 Add the Application plugin (https://docs.gradle.org/3.5/userguide/application_plugin.html) to make this a Java application.

The Application plugin (https://docs.gradle.org/3.5/userguide/application_plugin.html) allows you to bundle all of your applications JARs as well as all of their transitive dependencies into a single ZIP or TAR file. It will also add two startup scripts (one for UNIX-like operations systems and one for Windows) to the archive to make it easy for your users to run your application.

Once again, update `settings.gradle` to add the new project

settings.gradle

GROOVY

```
include 'greeter'
```

Now create a Java class file with a main function, which will consume the `Greeter` library from the `greeting-library` subproject.

```
$ mkdir -p greeter/src/main/java/greeter
```

greeter/src/main/java/greeter/Greeter.java

JAVA

```java
package greeter;

public class Greeter {
    public static void main(String[] args) {
        final String output = GreetingFormatter.greeting(args[0]);
        System.out.println(output);
    }
}
```

As the target is a Java application, you also need to tell Gradle the name of the class which is the entry point. Edit the `build.gradle` file again and assign the `mainClassName` property to Java class that contains the `main` method.

greeter/build.gradle

GROOVY

```groovy
mainClassName = 'greeter.Greeter'  1
```

1 Use `mainClassName` to set the entry point. (The assigned class must have a standard `main` method).

Run the build (which will fail, because we haven't resolved all dependencies yet).

```
$ ./gradlew build

...
.../Greeter.java:5: error: cannot find symbol
        final String output = GreetingFormatter.greeting(args[0]);
                              ^
  symbol:   variable GreetingFormatter
  location: class Greeter
1 error

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':greeter:compileJava'.
> Compilation failed; see the compiler error output for details.

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.
```

This is because the `greeter` project does not know where to find the `greeting-library`. Creating a collection of sub-projects does not automatically make their respective artifacts automatically available to other sub-projects - that would lead to very brittle projects. Gradle has a specific syntax to link the artifacts of one subproject to the dependencies of another sub-project. Edit the `build.gradle` script in the `greeter` sub-project again and add:

greeter/build.gradle

GROOVY

```
dependencies {
    compile project(':greeting-library') 1
}
```

1 Use the `project(NAME)` syntax to add the artifacts of one subproject to the dependencies of another subproject.

Run the build again, which should now succeed.

```
$ ./gradlew build

:greeting-library:compileJava NO-SOURCE
:greeting-library:compileGroovy UP-TO-DATE
:greeting-library:processResources NO-SOURCE
:greeting-library:classes UP-TO-DATE
:greeting-library:jar UP-TO-DATE
:greeter:compileJava
:greeter:compileGroovy NO-SOURCE
:greeter:processResources NO-SOURCE
:greeter:classes
:greeter:jar
:greeter:startScripts
:greeter:distTar
:greeter:distZip
:greeter:assemble
:greeter:compileTestJava NO-SOURCE
:greeter:compileTestGroovy
:greeter:processTestResources NO-SOURCE
:greeter:testClasses
:greeter:test
:greeter:check
:greeter:build
:greeting-library:assemble UP-TO-DATE
:greeting-library:compileTestJava NO-SOURCE
:greeting-library:compileTestGroovy UP-TO-DATE
:greeting-library:processTestResources NO-SOURCE
:greeting-library:testClasses UP-TO-DATE
:greeting-library:test UP-TO-DATE
:greeting-library:check UP-TO-DATE
:greeting-library:build UP-TO-DATE

BUILD SUCCESSFUL
```

Notice how each subproject is prefixed in the output, so that you know which task from which project is being executed. Also note that Gradle does not process all tasks from one subproject before moving onto another.

Add a test to ensure your code in the application itself works. As Spock Framework is a popular approach to testing Java code as well, create the test by first adding the Groovy plugin (https://docs.gradle.org/3.5/userguide/groovy_plugin.html) to the `build.gradle` script in the `greeter` sub-project. This requires the `groovy` plugin, and that includes the `java` plugin, so you can replace the word `java` with `groovy` as shown.

greeter/build.gradle

GROOVY

```
apply plugin : 'groovy'  1

// then, add the following testCompile dependency to the dependencies block:

dependencies {
    testCompile 'org.spockframework:spock-core:1.0-groovy-2.4', {
        exclude module : 'groovy-all'
    }
}
```

1 Having `groovy` plugin automatically applies the `java` plugin, so you could actually deleted the `pply plugin : 'java'` line. However for semantics it might be better to keep both as it indicates that you are primarily building a Java project.

Then add a test called `GreeterSpec` in the `greeter` package to the sub-project in the `src/test/groovy/greeter` directory (you'll have to create that directory if it does not already exist).

greeter/src/test/groovy/greeter/GreeterSpec.groovy

JAVA

```
package greeter

import spock.lang.Specification

class GreeterSpec extends Specification {

    def 'Calling the entry point'() {

        setup: 'Re-route standard out'
        def buf = new ByteArrayOutputStream(1024)
        System.out = new PrintStream(buf)

        when: 'The entrypoint is executed'
        Greeter.main('gradlephant')

        then: 'The correct greeting is output'
        buf.toString() == "Hello, Gradlephant\n"
    }
}
```

Instead of running the complete build again, just run the test inside the `greeter` subproject. The Gradle wrapper script `gradlew` only exists at top-level, so change directory to there first.

```
$ ./gradlew :greeter:test

:greeting-library:compileJava NO-SOURCE
:greeting-library:compileGroovy UP-TO-DATE
:greeting-library:processResources NO-SOURCE
:greeting-library:classes UP-TO-DATE
:greeting-library:jar UP-TO-DATE
:greeter:compileJava UP-TO-DATE
:greeter:compileGroovy NO-SOURCE
:greeter:processResources NO-SOURCE
:greeter:classes UP-TO-DATE
:greeter:compileTestJava NO-SOURCE
:greeter:compileTestGroovy UP-TO-DATE
:greeter:processTestResources NO-SOURCE
:greeter:testClasses UP-TO-DATE
:greeter:test UP-TO-DATE

BUILD SUCCESSFUL
```

It is possible to run any task in any sub-project from the top (or any other subproject), by using the format `:SUBPROJECT:TASK` as the task name. This removes the need to do the following alternative (which also works):

```
$ cd greeter
$ ../gradlew test  1
$ cd ..
```

1 The parent directory, which contains the generated Gradle wrapper script

If you were to spend a lot of time working in one sub-project, changing to that directory and running the build from there is normal. However, if you need to quickly run a task in a specific subproject, having the flexibility to specify the task by subproject path is very helpful.

Did you notice that the task executed was `:greeter:test`, but when running it Gradle first visited `greeting-library` to ensure that the dependencies were up to date? This is one more example how the powerful task graph implementation in Gradle saves you time.

Add documentation

It is considered good practice to create documentation for a software project. Although there a number of authoring methods to accomplish this goal, you will be using the very popular Asciidoctor (http://asciidoctor.org/) tool.

Start by adding the Asciidoctor plugin (https://plugins.gradle.org/plugin/org.asciidoctor.convert) to a plugins block (https://docs.gradle.org/3.5/userguide/plugins.html#sec:plugins_block) to the top of the `build.gradle` script in the root project.

build.gradle

GROOVY

```
plugins {
  id 'org.asciidoctor.convert' version '1.5.3' apply false  1
}
```

1 The use of `apply false` adds the plugin to the overall project, but does not add it to the root project.

Now another subproject folder, called `docs`, for the documentation.

```
$ mkdir docs
```

Create a `build.gradle` file in the `docs` folder with the following content:

docs/build.gradle

GROOVY

```
apply plugin : 'org.asciidoctor.convert'  1

asciidoctor {
    sources {
        include 'greeter.adoc'  2
    }
}

build.dependsOn 'asciidoctor'  3
```

1 Apply the Asciidoctor plugin to this subproject. This technique lets you selectively apply plugins to sub-projects while defining all of the plugins in the root project.

2 Tells the plugin to look for a document called `greeter.adoc` in the default source folder `src/docs/asciidoc`

3 Adds `asciidoctor` task into the build lifecycle so that if `build` is executed for the top-level project, then documentation will be built as well.

Add this subproject to `settings.gradle`.

settings.gradle

GROOVY

```
include 'docs'
```

Add a document called `greeter.adoc` in the Asciidoctor source folder, `src/docs/asciidoc`, in the `docs` sub-project. You'll need to generate that directory if it doesn't exist.

docs/src/docs/asciidoc/greeter.adoc

```
= Greeter Command-line Application

A simple application demonstrating the flexibility of a Gradle multi-project.

== Installation

Unpack the ZIP or TAR file in a suitable location

== Usage

[listing]
----
$ cd greeter-1.0
$ ./bin/greeter gradlephant

Hello, Gradlephant
----
```

Run `asciidoctor` task from the top-level project.

If you run the command `./gradlew tasks`, you will now see a task called `asciidoctor` in the "Documentation tasks" category.

```
$ ./gradlew asciidoctor

:docs:asciidoctor

BUILD SUCCESSFUL
```

> Did you notice that Gradle knew to only run this task in the `docs` subproject? This is because when you run a task from the level of a root project and this task does not exist in the root, then Gradle will run the task in each of the sub-projects where a task by that name exists.

The documentation artifact will appear in `docs/build/asciidoc/html5`. Feel free to open the the `greeter.html` file and inspect the output.

Include the documentation in the distribution archive

Documentation is useful when published, but usage documentation distributed with an application is very valuable for people trying out your application. Add this generated documentation to your distribution by updating task dependencies in the `build.gradle` script in the `greeter` sub-project.

greeter/build.gradle

```
distZip {
    from project(':docs').asciidoctor, {  1
        into "${project.name}-${version}"
    }
}
distTar {
    from project(':docs').asciidoctor, {
        into "${project.name}-${version}"
    }
}
```

1 Use `project(:NAME).TASKNAME` format to reference a task instance in another project.

Build from the top again, and this time the resulting archive will include the docs.

```
$ ./gradlew build

:docs:asciidoctor UP-TO-DATE
:docs:assemble UP-TO-DATE
:docs:check UP-TO-DATE
:docs:build UP-TO-DATE
:greeting-library:compileJava NO-SOURCE
:greeting-library:compileGroovy UP-TO-DATE
:greeting-library:processResources NO-SOURCE
:greeting-library:classes UP-TO-DATE
:greeting-library:jar UP-TO-DATE
:greeter:compileJava UP-TO-DATE
:greeter:compileGroovy NO-SOURCE
:greeter:processResources NO-SOURCE
:greeter:classes UP-TO-DATE
:greeter:jar UP-TO-DATE
:greeter:startScripts UP-TO-DATE
:greeter:distTar
:greeter:distZip
:greeter:assemble
:greeter:compileTestJava NO-SOURCE
:greeter:compileTestGroovy UP-TO-DATE
:greeter:processTestResources NO-SOURCE
:greeter:testClasses UP-TO-DATE
:greeter:test UP-TO-DATE
:greeter:check UP-TO-DATE
:greeter:build
:greeting-library:assemble UP-TO-DATE
:greeting-library:compileTestJava NO-SOURCE
:greeting-library:compileTestGroovy UP-TO-DATE
:greeting-library:processTestResources NO-SOURCE
:greeting-library:testClasses UP-TO-DATE
:greeting-library:test UP-TO-DATE
:greeting-library:check UP-TO-DATE
:greeting-library:build UP-TO-DATE

BUILD SUCCESSFUL
```

Notice how most tasks are still up to date, but both the `distZip` and `distTar` tasks have been re-run to include the documentation. If you unpack one of the archives (`greeter-1.0.zip` or `greeter-1.0.tar` in the `greeter/build/distributions` directory) you will see the documentation included in the `html5` folder.

> A strong feature of Gradle that sets it apart from some other build tools is how well it handles incremental building. It is not necessary to do a `./gradlew clean` before each build.

Refactor common build script code

At this point you might have noticed that you have common script code in both `greeting-library/build.gradle` and `greeter/build.gradle` . A key feature of Gradle is the ability to place such common build script code in the root project.

Edit `build.gradle` in the root project and add the following code

build.gradle

GROOVY

```groovy
configure(subprojects.findAll {it.name == 'greeter' || it.name == 'greeting-library'} ) {  1

    apply plugin : 'groovy'

    dependencies {
        testCompile 'org.spockframework:spock-core:1.0-groovy-2.4', {
            exclude module : 'groovy-all'
        }
    }
}
```

1 Use of `configure` along with a predicate closure allows for selective sub-projects to be configured. The predicate closure is passed a subproject whose name can be queried. In this case only sub-projects with specific name matches will be configured.

In the same time remove the following lines from `greeting-library/build.gradle`

greeting-library/build.gradle (Removed code)

```
apply plugin : 'groovy'

dependencies {
    testCompile 'org.spockframework:spock-core:1.0-groovy-2.4', {
        exclude module : 'groovy-all'
    }
}
```

also remove similar lines from `greeter/build.gradle`. In other words, remove the `apply plugin: groovy` line and the `spock` dependency from the `dependencies` block, but leave the `dependencies` block itself.

Re-run everything from the top-level to make sure it all still works.

```
$ ./gradlew clean build
```

Summary

By following the steps you have seen how to

- Create a modular software project by combining multiple sub-projects.
- Have one sub-project consume artifacts from another sub-project.
- Use a polyglot project with ease.
- Run similar named tasks in all sub-projects.
- Run tasks in specific sub-projects without changing to that subproject's folder.

- Refactor common sub-project settings into the root project.
- Selectively configure sub-projects from the root project.

Next Steps

- Remembering to type `./gradlew`, `../gradlew` or even `../../gradlew` can become an arduous task. If you are on a Unix-like operating system consider installing gdub (https://github.com/dougborg/gdub).
- Read about multi-project in the User Manual (https://docs.gradle.org/3.5/userguide/multi_project_builds.html)

Last updated 2017-08-09 21:13:57 UTC