Rich Web Experience 2016

Learn to use Gradle (and understand it)

Brent Laster

Gradle Lab 1 - Creating and invoking a simple Gradle Build Script

1. Open up a command prompt (or bash shell if you prefer) and create a new "playpen" subdirectory on your disk.

    mkdir playpen

2. CD into that directory and invoke gradle to make sure your setup is correct.

    cd playpen

    gradle

3. Assuming that worked, open up a text editor and create a new file. (We will be naming this file **build.gradle**)

4. Enter the following in the editor to create a new task.

    task hello

5. Save the file as build.gradle (in your playpen directory) and then invoke gradle to see the lists of tasks.

    gradle tasks

    Does your new task show up in the list?

6. In the editor, add a couple of properties to your task.

    task hello {

        group = 'students'


```
        description = 'Displays a greeting to a group.'
}
```

Save your changes and run gradle to show the list of tasks. Notice where the description shows up now.

```
gradle tasks
```

7. Rename build.gradle to lab1.gradle and invoke the tasks option again. Don't forget to use the -b option to specify that our gradle build file is named something other than the default.

```
gradle -b lab1.gradle tasks
```

8. Add an action inside the closure to be done first with the doFirst API call. (Remember the file is now called lab1.gradle.)

```
task hello {
        group = 'students'
        description = 'Displays a greeting to a group.'
        doFirst { println 'Get Ready...' }
}
```

9. Save your changes and run the build script.

```
gradle -b lab1.gradle
```

Did your print output show up? Why not?

10. Now run the build script and invoke your task so the action occurs.

```
gradle -b lab1.gradle hello
```

Did your print output show up this time? Why?

Gradle Lab 2 - Understanding Gradle Phases

1. Start in your playpen directory again.

2. In your lab1.gradle file, add another step to be done last. (Add the line in **bold font** below.)

```
task hello {
        group = 'students'
        description = 'Displays a greeting to a group.'
        doFirst { println 'Get Ready...' }
        doLast { println 'Hello ' + group }
}
```

Which lines will be executed during the configuration phase and which during the execution phase?

3. Add steps outside the closure to add additional actions to do first and last -below the current task definition.

```
hello.doFirst {
        println 'Get Set...'
        println 'Go!...'
}

hello.doLast {
        println 'Thanks for attending'
}
```

4. Add one more step using the shortcut for the doLast action.

```
hello << {
    println 'Enjoy the workshop!'
}
```

5. Save your changes and invoke the hello task again. Take a moment and look at your file. What order do you think things will print out in?

gradle -b lab1.gradle hello

Did things print out in the order you expected? Why or why not?

6. Run the build script without invoking the task.

gradle -b lab1.gradle

Did your actions print out?

7. Add a configuration step (not an action) by adding two lines without the doFirst or doLast APIs. (Add the lines in **bold font** below.)

```
task hello {
    group = 'students'
    description = 'Displays a greeting to a group.'
    println 'Setting group to ' + group
    println 'This task ' + description
    doFirst { println 'Get Ready...' }
    doLast { println 'Hello ' + group }
}
```

8. Save your work and invoke the build script without the task.

gradle -b lab1.gradle

Note what prints out just after the invocation. Why?

9. Finally run the build script and invoke your task to see the full effects.

gradle -b lab1.gradle hello

Gradle Lab 3 - Plugins and Properties

1. In your playpen directory, clone the gradle-greetings project from github.

git clone https://github.com/brentlaster/gradle-greetings.git

2. cd into the new gradle-greetings directory. Browse around the structure if you like. Note that it used the gradle/maven standard organization of src/main/java.

cd gradle-greetings

3. At the top level directory (in gradle-greetings) Create a simple **build.gradle** file with just a comment.

// lab 3 gradle script

4. Invoke gradle to see what default tasks are available on an "empty" build script.

gradle tasks

5. Invoke gradle to see what properties are available on an "empty" build script.

gradle properties

6. Add the java plugin to your gradle build script by adding the following line in build.gradle.

apply plugin: 'java'

7. Invoke gradle to see what additional tasks are available on a "java" build script.

gradle tasks

8. Invoke gradle to see what additional properties are available on a "java" build script.

gradle properties

9. Now invoke gradle to run the clean task and the build task on the project.

gradle clean build

(Note the passing of multiple tasks on the command line.)

10. Note the output from the build and the UP-TO-DATE indicators next to some of the tasks that run.

Find the build directory tree and examine the structure underneath. Find the class and jar files.

11. Invoke the build task again and note the UP-TO-DATE indicators. Then run the command with the "quiet" option.

gradle build

gradle -q build

12. Do a clean and take a look at the directory tree - what was removed?

gradle clean

13. Let's override one of the properties in the script by setting it from a new variable.

Add the following lines in **build.gradle.**

myDir = '.\\output'

buildDir = myDir

14. Save the changes and run gradle again.

gradle build

Did the build succeed? what is the problem?

15. Try defining this from the gradle command line using the –P option to pass a value.

gradle -PmyDir='.\\output' build (Windows)

or

gradle -PmyDir='./output' build (Bash)

Did the build succeed? If so, why? What's the difference compared to the last step?

16. Remember that we can't define custom properties like this unless we put them (or they are implicitly put) in an ext block. Change the line for myDir = so that myDir is part of an ext block. (For simplicity we'll use the shorthand notation since there's only one new property.)

Change

myDir = '.\\output'

to

ext.myDir = '.\\output'

17. Run a clean build again.

gradle clean build

Did it succeed? Where is your output?

Gradle Lab 4 - Enhancing our Build Script and using the Gradle wrapper

1. We want to add some additional functionality in our build script. Let's start by setting up some default tasks. In this case, we'll make the clean and build tasks our defaults. In the **gradle-greetings** project directory, add the following line in your build.gradle file.

```
defaultTasks 'clean', 'build'
```

2. Save your changes and execute the script just by invoking 'gradle'. Look at the output - the lists of tasks that were run. Note where in the list the ":clean" task is for future reference.

```
gradle
```

3. We know that we always want to do a clean before a build. So what's another way we could specify this relationship? We can use the dependsOn task api call. How would you do this? In build.gradle, change the existing defaultTasks line to remove the clean task and then add a line so that build depends on clean.

```
defaultTasks 'build'

build.dependsOn clean
```

4. Save your changes and execute the script just by invoking 'gradle'. Look at the output - the lists of tasks that were run. Note where in the list the ":clean" task is. Is this where you expected it? Is this where it needs to be?

```
gradle
```

5. We want the clean task to be the first one processed. How do we make it the first one in the list? What should depend on it?

6. Change the dependsOn line so that the second task (:compileJava) depends on the clean task so clean will be run before it. Change the "build.dependsOn" line to the following in your build.gradle file.

compileJava.dependsOn clean

7. Save your changes and invoke gradle to make sure the tasks are in the order you would expect now.

gradle

8. Let's go ahead and add the gradle wrapper to this project.

9. Add the wrapper task (lines below) into your build.gradle file.

```
task wrapper (type :Wrapper) {
    gradleVersion = '2.12'
}
```

10. Save your changes. Now generate the wrapper files by running the wrapper task.

gradle wrapper

11. Take a look at the additional files that were generated. Invoke the script with the new wrapper.

ls

gradlew (may need ./gradlew)

Gradle Lab 5 - Dependencies and Publishing

1. In the gradle-greetings directory, switch to the branch named 'test'.

```
git checkout test
```

2. Let's go ahead and set some properties on the jar file we're creating - such as a version in the manifest. Add this code to extend the jar task.

```
version = '1.0'
jar {
    manifest {
        attributes 'Implementation-Title': 'Workshop Example',
'Implementation-Version': version
    }
}
```

3. In order to build our project, we need to be able to pull some dependencies from a repository.

In this case, we'll just use the Maven central repository to keep things simple. Add the repository reference into your script as below.

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

4. Now we need to add in the specific external dependencies we need. First, let's add in a specific commons collections one for the compile configuration. (You can just put this under the repositories section.)

dependencies {

compile group: 'commons-collections', name: 'commons-collections', version: '3.2'

}

5. Save your changes and run gradle (by itself) to see if everything works.

gradle

6. Did the build work? Look at the output and see if you can figure out why it didn't work.

7. It looks like it was trying to do some unit testing. What is the definition of the "build" task?

Invoke

gradle tasks

What two things does the build task say it does?

8. We're not quite ready for unit testing yet (we'll cover that in the next section). For now, we want to exclude the test task from being run. We could just call the assemble task, but let's specifically exclude test instead.

gradle clean build -x test

Did the build work now?

9. Our build works so we want to upload the artifacts to a repository (publication). First, we need to define a repository. For simplicity, we'll make this a flat directory repository - however it could have also been an ivy or maven one. Add the following repository definition in your build.gradle file.

```
repositories {
    flatDir {
        name 'myRepo'
        dirs 'repos'
    }
}
```

Here "flatDir" is the type, "name" is the name that gradle refers to it by, and "dirs" is the actual directory.

10. Now we need to add our publications repository to the repositories used in the uploadArchives task. Add the following in your build.gradle file.

```
uploadArchives {
    repositories {
        add project.repositories.myRepo
    }
}
```

11. Save your changes and run the uploadArchives task.

```
gradle uploadArchives
```

12. Switch into the repos directory to see what it published.

Gradle Lab 6 - Testing with Gradle

1. Make sure you are in your gradle-greetings project directory (and in the "test" branch). Take a look at the two test files in **src/test/java.**

2. In the gradle-greetings directory, edit the build.gradle file and add the following: (changes can be inside existing blocks).

```
dependencies {
        testCompile 'junit:junit:4.+'
}

test {
        testLogging {
                events 'started', 'passed'
    }
}
```

3. Run the tests and look at what output we get.

```
gradle test
```

4. Take a look at the test reports generated by Gradle. Open up the file **output/reports/tests/index.html** in a browser.

5. Click on the "**TestExample2.example2**" link. Take a look at the additional output provided.

6. In the gradle-greetings directory, let's build just the one failing test.

gradle -Dtest.single=TestExample2 test

Take a look at the output.

7. Add more detail in the build script to provide more informative error output.

Insert the line below inside the **testLogging { }** section (with the "events" line)

exceptionFormat = 'full'

Save your changes.

8. Execute the failing test again and take a look at what output you get this time.

gradle -Dtest.single=TestExample2 test

Gradle Lab 7 – Working with Source Sets

In this lab, we'll take a java project in an alternate source structure and look at how to make it build with Gradle.

1. Switch back to your playpen directory.

cd ~/playpen (Bash)

or

cd C:\Users\RETstudent\playpen (Windows)

2. In the *playpen* directory, clone the sas icons project down from gitgrid.

git clone https://github.com/brentlaster/gradle-greetings-sets

2. Once the project is cloned down, switch into the sasicons directory and browse around the project.

cd gradle-greetings-sets

3. Since we know that this is a java project and that gradle has easy support for java

through the java plugin, let's start out by creating a basic build file for java.

In the *gradle-greetings-sets* directory, create a **build.gradle** file with a line to include the java plugin.

apply plugin: 'java'

Save the file.

4. Take a look again to see what tasks we have available to us.

gradle tasks

5. Now, let's try a build.

gradle build

6. We didn't get any errors. But lots of things, such as compileJava, report *UP-TO-DATE*. That means that Gradle thought all inputs and output were the latest. That seems unusual since we would have expected it to produce new output the first time through. Take a look at what's in our sasicons area now. Do a directory listing.

ls or **dir**

We now have a *build* directory. Look around in that directory. Look in the *libs* directory under build.

We have a *gradle-greetings.jar* file there. What is the size of it?

7. Hmmm, the size of that file seems pretty small for a fully compiled java jar. Let's take a closer look at what the build actually did by turning on the info messages.

Go back to the gradle-greetings-sets directory (if not there). Then, do

gradle --info clean build

8. Scroll back up to the top of the output from the command and start looking down through the output. Notice a couple of things.

On the line in the *:clean* section that starts with *"Executing task ':clean'..."* notice the line that says

"Task has not declared any outputs."

On the line in the *:compileJava* section that starts with *"Skipping",* notice it says *"has no source files."*

On the line starting with *:jar*, notice that it says it completed.

9. What does the jar task do? Remind yourself by running

gradle help --task jar

10. So perhaps gradle actually didn't compile any java code and the jar task just created the small "empty" jar we saw as build output.

11. Let's verify this. Run a gradle clean to blow away the *output* directory.

gradle clean

12. Now run just the jar task.

gradle jar

13. Look in the build output directory and see if you have the same jar as before.

14. So, now we know that gradle isn't finding our source files. Why?

(Hint: Our source structure starts out as *"Source/Java..."* What is the standard source layout assumed by Gradle for the Java plugin?)

15. Let's have Gradle tell us where/what it thinks it should build by using the components task.

gradle components

16. Where does the output say it's expecting the source sets to be? And where are ours for this project?

17. To fix this, we need to tell Gradle how to map our source structure to what it expects. To do this, we can use a sourceSet in gradle.

Add the following to your *build.gradle* file (after the *apply plugin* line):

```
sourceSets {
        main {
                java {
                        srcDir 'Source/Java'
                }
        }
}
```

Note: Another way to write this would be *sourceSets.main.java.srcDir 'Source/Java'*

18. Run the components task again and note what source sets gradle knows about now.

```
gradle components
```

(Look for a new line in the "*Additional source sets*" section.)

19. Now, run the build again with a clean and the info flag.

```
gradle --info clean build
```

20. Take a look at the same lines of output as before - especially the *:compileJava* section. What's different?

21. Take a look at the *"build"* output directory. What's different from before? What's the size of *gradle-greetings.jar*?

Lab 8 - Using the continuous build feature, working with the daemon and using more command line options.

1. In our last lab, we built the *gradle-greetings-sets* project. Now, we're going to see how to do some further builds optimizations and profiling.

2. The continuous build feature of gradle is an incubating feature. Go ahead and start it up by starting Gradle with the --continuous or -t option. Note that we also need to pass it a task to execute.

gradle --continuous build

3. Take a look at the output and the last line. Gradle is waiting for us to update an input file.

4. Since Gradle is running in continuous build mode in this session, switch to another session to make file changes. (On Linux, we could have started this in the background as another option.)

5. In the second terminal session/command prompt, go to the *playpen/gradle-greetings-sets/Source/Java/com/demo/worskhop* directory.

cd ~/playpen/gradle-greetings-sets/Source/Java/com/demo/workshop

6. In that directory, modify the helloWorkshop.java file in some way. For example, you could change the text inside one of the *println* statements or change one of the comments. (If you are on a Linux system, you can also just *touch* the file.)

modify or touch helloWorkshop.java

7. Change back to the original terminal session/command prompt. You should now have output from another run of the build. (It may look like the same screen as before, but scan up and you should see text like "*Change detected, executing build…*". Further up, you'll see the previous place where it was waiting for a change.)

8. In the original session, stop the instance of Gradle that is running for the continuous build.

Ctrl-C

9. Let's see how much time we're spending on the build. We can do this by telling Gradle to profile our build. Run

gradle -profile clean build

10. Gradle puts the profile output into a nice html format. Take a look at this output by browsing out to the *build/reports/profile* directory and opening up the .html file there in a browser.

Click around on the *Summary, Configuration* and other buttons on the web page and look at the results.

Leave this page open in the browser.

11. We'd like to speed up this build. Gradle has a build daemon that can reduce the startup costs for gradle builds. Let's enable that.

Create a **gradle.properties** file in *c:\users\<userid>\.gradle* (~/.gradle in Linux shell) that contains this line:

org.gradle.daemon=true

Save the file making sure it is saved as **gradle.properties** (and not gradle.properties.txt or such) in the *.gradle* directory.

12. Switch back to the *playpen/gradle-greetings-sets* directory and start the daemon.

gradle --daemon

13. Now do a clean build with the profile option again.

gradle -profile clean build

14. Go to the *build/reports/profile* directory and open up the html file with the timestamp of the build you just ran. Compare the times reported in there against the profile from earlier. The daemon will now be active unless we stop it or it has 3 hours of inactivity.

Gradle Lab 9 - Multi-project builds

1. In your playpen directory, clone the *gradle-multi* project from github.

    git clone https://github.com/brentlaster/gradle-multi.git

2. cd into the new *gradle-multi* directory. Take a look at the contents of the *build.gradle* file. Look at the *setupProject* task. What do you think it does?

3. Let's get a list of tasks available to us.

    gradle tasks

    Read step 4!

4. We got a compile error with our build file because a project was missing. We need to define the various projects we'll be using for gradle.

5. Using an editor, create a new file named *settings.gradle* with the following content:

    include 'domain', 'dataaccess', 'services', 'web'

6. Save this file as **"settings.gradle"** in your *playpen\gradle-multi* directory.

7. Try running the task command again.

    gradle tasks

    Did it work this time? Why?

8. Let's let the *gradle setupProject* task do what is intended. Run the task.

```
gradle setupProject
```

Browse around the directory tree that was created. Why were these particular directories created? (Hint: Think about *settings.gradle* and note the "*groovy.srcDirs**" part of the *setupProject* task.

9. We now realize we need to define a *WEB-INF* directory for our webapp. We want to do this as part of the web project. But first we need the web project defined. Add the following code block at the end of your *build.gradle* file.

```
project(':web') {
}
```

10. Now we'll add the extra code for the *setupProject* task. Add the code below inside of the *web* project block you just created (between the **{}**).

```
setupProject << { task ->
    def webInfDir = new File(task.project.webAppDir, '/WEB-INF')
    println "Creating $webInfDir"
    webInfDir.mkdirs()
}
```

Save your file.

11. Now run the *setupProject* task again.

```
gradle setupProject
```

What was the output? Read the next step to fix this.

12. The message says that we are missing the property *'webAppDir'*. We need to define that for Gradle or pull it in from somewhere. In this case, we'll pull it in using the jetty plugin. Add the following line as the first one inside the *project(':web)* block (after the {).

apply plugin: 'jetty'

13. Save your *build.gradle* file and try the *setupProject* task again.

gradle setupProject

It should work this time.

14. Now we need the groovy source files for this project. These already exist in our Git repository in the branch named "*groovySource*". We'll switch there and grab those in a moment, but we need to do one thing first. We need to save our modified copy of *build.gradle* so that Git doesn't think we need to commit it when we switch branches.

15. We use the git *"stash"* function to save a copy of our modified *build.gradle*. Then we switch branches and restore our files from the stash. Run the following commands.

git stash

git checkout groovySource

git stash pop --quiet

16. You should now have all of the source files needed to build the projects. We just need a few final steps.

17. Bring in the groovy dependencies. Add the following lines in your *build.gradle* file where the comment "// add groovy dependency here" is.

```
dependencies {
        compile 'org.codehaus.groovy:groovy-all:2.4.4'
}
```

18. We want to configure the jetty port. Add the following line after the "*apply plugin: 'jetty'*" line (in the web project specification).

```
jettyRun.httpPort=9999
```

19. Finally, we need to tell Gradle that the web project is dependent on the services project. Note here that because we have *configurations.compile.transitive = true* at the beginning of our file, we only have to specify the *services* project, even though we also reference the *dataaccess* and *domain* projects. Add the following lines below the line from the previous step (below the *httpPort* line in the web project specification).

```
dependencies {
        compile project(':services')
}
```

20. Save your changes and build the projects.

```
gradle build
```

21. Invoke the *jettyRun* task to start up the webserver. (Note: This will run and not return.)

```
gradle jettyRun
```

22. Open a web browser and go to the address below to see the end result.

http://localhost:9999/web/course.groovy

Gradle Lab 10 - Optional: Using Gradle with Jenkins

1. If you already have a version of Jenkins installed and running, you can skip to step 4.

2. Open up a browser and navigate to http://www.jenkins-ci.org.

3. On the right-hand side of the page, in the Download Jenkins section, select the appropriate installer and go through the setup. It is preferable here not to use Jenkins 2.0.

4. Browse to http://localhost:8080 and make sure that Jenkins is up and running.

5. You will also need to install the Gradle plugin if you don't have it. This can be done through Manage Jenkins->Manage Plugins. Switch to the Available tab and in the Filter box (upper right) type in "Gradle". Once you find it, install it, and restart Jenkins if needed.

6. We want to create a new job. Click the "*New Item*" link on the left hand menu.

Enter "*Gradle Greetings*" for the name. Don't hit Enter - you still need to fill in the job type.

7. We're going to configure everything for our job, so it will be a "*Freestyle project*".

Select the first option

Freestyle project.

Select **OK**.

8. After the screen changes, note that you are now in the configuration screen for the Gradle Greetings job. We will just use our existing copy of the project instead of cloning again from Git. To tell Jenkins how to find our project, we need to configure a *Custom Workspace*. We'll use the same location we used in earlier labs.

Find the section of the page labeled "**Advanced Project Options**".

Click the button labeled "**Advanced…**" to expand this area.

Check the box titled "**Use custom workspace**".

Fill in the "**Directory**" field with the path to your gradle-greetings project:

(probable path on windows) **C:\users\<userid>\playpen\gradle-greetings**

9. Now, we need to configure the actual build part of our job.

Find the section of the page labeled "**Build**".

Select the "**Add build step**" button.

Select "**Invoke Gradle script**" from the drop-down list.

9. Note that Jenkins has added new fields to allow us to configure the Gradle build.

We'll just use the default Gradle version installed on the image – so just leave that as **"(Default)".**

(That will use the one in the system path.)

We don't need the Gradle wrapper for this project, so leave that unselected.

Enter some text in the "**Build step description**" field if you wish.

The "**Switches**" field can have command-line options we want to pass to Gradle. We don't have any for this job.

The "**Tasks**" field is used to tell Jenkins/Gradle what tasks we want to execute. This can be left empty if we have default tasks defined. In this case, we want to run the clean and build tasks but also skip the failing test task, so enter the following in that field:

clean build –x test

"Root Build script" can be used to define a custom location for the root script of a multi-project build if the root script is not in the default location. You can leave it empty.

The **"Build File"** field would be where we list the name of the build script if it is different from build.gradle. We will use the default (build.gradle) so you can leave it empty.

10. Click on the **"Save"** button at the bottom of the screen to save the job's configuration.

11. You should now be back on the job page. On the left side of that page should be a set of menu options.

Click **"Build Now"** to start building our job.

12. In the lower left part of the page, you'll see the job running in the "*Build History*" area. When completed, the ball indicator should be blue for success. Let's look at the console output from this job.

Click on the blue ball next to the latest run in the *Build History* area.

(Alternatively, you can click on the date/timestamp and then click on the "*Console Output*" link.)

13. Notice that on the left side of the page, we can see all of the Gradle tasks that have been executed.

14. To be able to look at the generated artifacts and test results, we have to add two post-build actions to the job's configuration.

Click on the **"Back to Project"** link.

On the project page, click on the **"Configure"** link.

In the **"Tasks"** field, remove the **–x test** portion. The field should now look like:

clean build

In the *Post-build Actions* section, click on the "**Add post-build action"**.

Select **"Archive the artifacts".**

Also select **"Publish Junit test results report".**

15. Since our artifacts from our builds are stored in the build/libs directory, fill in the values for the *Archive the artifacts* section as follows:

Files to archive: output/libs/*.jar

16. Fill in the value for the *Publish Junit test result report* section as below. Jenkins will complain about the area not existing. That's ok.

Test report XMLs: output/test-results/*.xml

17. Save your changes by clicking on the **Save** button.

18. Run the job again by clicking on **Build Now**. The build will fail. That's expected because one of our tests is designed to fail.

19. Click on the most recent date/time link in the *Build History* window to get to the output page. On the output page, notice that we have links to download the artifacts from the build. Test results are also shown. More test detail is available by clicking on the *Test Result* link.