Gradle Goodness

NOTEBOOK

Hubert A. Klein Ikkink

HA.
KI

Gradle Goodness Notebook

Experience Gradle through code snippets

Hubert A. Klein Ikkink (mrhaki)

This book is for sale at http://leanpub.com/gradle-goodness-notebook

This version was published on 2016-11-21

Also By Hubert A. Klein Ikkink (mrhaki)

Groovy Goodness Notebook

Grails Goodness Notebook

Spocklight Notebook

Awesome Asciidoctor Notebook

Ratpacked Notebook

This book is dedicated to my lovely family. I love you.

Contents

About Me

I am born in 1973 and live in Tilburg, the Netherlands, with my beautiful wife and three gorgeous children. I am also known as mrhaki, which is simply the initials of his name prepended by mr. The following Groovy snippets shows how the alias comes together:

```
['Hubert', 'Alexander', 'Klein', 'Ikkink'].inject('mr') { alias, name ->
    alias += name[0].toLowerCase()
}
```

(How cool is Groovy that we can express this in a simple code sample ;-))

I studied Information Systems and Management at the Tilburg University. After finishing my studies I started to work at a company which specialized in knowledge-based software. There I started writing my first Java software (yes, an applet!) in 1996. Over the years my focus switched from applets, to servlets, to Java Enterprise Edition applications, to Spring-based software.

In 2008 I wanted to have fun again when writing software. The larger projects I was working on were more about writing configuration XML files, tuning performance and less about real development. So I started to look around and noticed Groovy as a good language to learn about. I could still use existing Java code, libraries, and my Groovy classes in Java. The learning curve isn't steep and to support my learning phase I wrote down interesting Groovy facts in my blog with the title *Groovy Goodness*. I post small articles with a lot of code samples to understand how to use Groovy. Since November 2011 I am also a DZone Most Valuable Blogger (MVB); DZone also posts my blog items on their site.

I have spoken at the Gr8Conf Europe and US editions about Groovy, Gradle, Grails and Asciidoctor topics. Other conferences where I talked are Greach in Madrid, Spain, JavaLand in Germany and JFall in The Netherlands.

I work for a company called JDriven in the Netherlands. JDriven focuses on technologies that simplify and improve development of enterprise applications. Employees of JDriven have years of experience with Java and related technologies and are all eager to learn about new technologies. I work on projects using Grails and Java combined with Groovy and Gradle.

Introduction

When I started to learn about Gradle I wrote done little code snippets with features of Gradle I found interesting. To access my notes from different locations I wrote the snippets with a short explanation in a blog: Messages from mrhaki. I labeled the post as *Gradle Goodness*, because I thought this is good stuff, and that is how the *Gradle Goodness* series began.

A while ago I bundled all my blog Groovy Goodness blog posts in a book published at Leanpub. Leanpub is very easy to use and I could use Markdown to write the content, which I really liked as a developer. So it felt natural to also bundle the Grails Goodness blog posts at Leanpub.

In this book the blog posts are bundled and categorized into sections. Within each section blog posts that cover similar features are grouped. The book is intended to browse through the subjects. You should be able to just open the book at a random page and learn more about Gradle. Maybe pick it up once in a while and learn a bit more about known and lesser known features of Gradle.

I hope you will enjoy reading the book and it will help you with learning about Gradle, so you can apply all the goodness in your projects.

Gradle versions

Because the blog posts are written in the last 5 years, they also cover older versions of Gradle. At the end of each blog post we can see which version of Gradle was used to write the post. Most tips are still useable in newer versions of Gradle, but some could also be out-dated and only applicable to the designated version. Please keep this in mind when you read an article.

Code samples

Code samples in the book can contain a backslash (\) at the end of the line to denote the line actually continues. This character is not part of the code, but for better formatting of the code blocks in the book the line is split. If you want to use the code you must keep this in mind.

Java and Groovy

Set Java Version Compatibility

We can use the properties *sourceCompatibility* and *targetCompatibility* provided by the Java plugin to define the Java version compatibility for compiling sources. The value of these properties is a `JavaVersion` enum constant, a String value or a Number. If the value is a String or Number we can even leave out the 1. portion for Java 1.5 and 1.6. So we can just use 5 or '6' for example.

We can even use our own custom classes as long as we override the `toString()` method and return a String value that is valid as a Java version.

```
apply plugin: 'java'

sourceCompatibility = 1.6 // or '1.6', '6', 6,
                          // JavaVersion.VERSION_1_6,
                          // new Compatibility('Java 6')

class Compatibility {
    String version

    Compatibility(String versionValue) {
        def matcher = (versionValue =~ /Java (\d)/)
        version = matcher[0][1]
    }

    String toString() { version }
}
```

Written with Gradle 0.9-rc-2.

Original blog post written on November 09, 2010.

Set Java Compiler Encoding

If we want to set an explicit encoding for the Java compiler in Gradle we can use the `options.encoding` property. For example we could add the following line to our Gradle build file to change the encoding for the `compileJava` task:

```
apply plugin: 'java'

compileJava.options.encoding = 'UTF-8'
```

To set the encoding property on all compile tasks in our project we can use the `withType()` method on the `TaskContainer` to find all tasks of type `Compile`. Then we can set the encoding in the configuration closure:

```
apply plugin: 'java'

tasks.withType(Compile) {
    options.encoding = 'UTF-8'
}
```

Written with Gradle 1.0.

Original blog post written on June 18, 2012.

Using Gradle for a Mixed Java and Groovy Project

Gradle is a build system to build software projects. Gradle supports convention over configuration, build-in takss and dependency support. We write a build script in Groovy (!) to define our Gradle build. This means we can use all available Groovy (and Java) stuff we want, like control structures, classes and methods. In this post we see how we can use Gradle to build a very simple mixed Java and Groovy project.

To get started we must first have installed Gradle on our computers. We can read the manual to see how to do that. To check Gradle is installed correctly and we can run build script we type `$ gradle -v` at our shell prompt and we must get a result with the versions of Java, Groovy, operating system, Ivy and more.

It is time to create our Groovy/Java project. We create a new directory `mixed-project`:

```
$ mkdir mixed-project
$ cd mixed-project
$ touch build.gradle
```

Gradle uses plugins to define taks and conventions for certain type of projects. The plugins are distributed with Gradle and not (yet) downloaded from the internet. One of the plugins is the Groovy plugin. This plugin is extended from the Java plugin, so if we use the Groovy plugin we also have all functionality of the Java plugin. And that is exactly what we need for our project. The plugin provides a set of tasks like compile, build, assemble, clean and a directory structure convention. The plugin assumes we save our source files in `src/main/java` and `src/main/groovy` for example. The structure is similar to Maven conventions. As a matter of fact Gradle also has a Maven plugin that add Maven tasks like build, install to our build. For now we just need the Groovy plugin, so we open the file `build.gradle` in a text editor and add the following line:

```
apply plugin: 'groovy'
```

To see the lists of tasks we can now execute by just including this one line we return to our shell and type `$ gradle tasks` and we get the following list of tasks:

```
:tasks

------------------------------------------------------------
All tasks runnable from root project
------------------------------------------------------------

Build tasks
-----------
assemble - Assembles the outputs of this project.
build - Assembles and tests this project.
buildDependents - Assembles and tests this project and all projects that depend on it.
buildNeeded - Assembles and tests this project and all projects it depends on.
classes - Assembles classes 'main'.
clean - Deletes the build directory.
jar - Assembles a jar archive containing the main classes.
testClasses - Assembles classes 'test'.

Build Setup tasks
-----------------
init - Initializes a new Gradle build. [incubating]
wrapper - Generates Gradle wrapper files. [incubating]

Documentation tasks
-------------------
groovydoc - Generates Groovydoc API documentation for the main source code.
javadoc - Generates Javadoc API documentation for the main source code.

Help tasks
----------
components - Displays the components produced by root project 'mixed-project'. [incubating]
dependencies - Displays all dependencies declared in root project 'mixed-project'.
dependencyInsight - Displays the insight into a specific dependency in root project 'mixed-projec\
t'.
help - Displays a help message.
projects - Displays the sub-projects of root project 'mixed-project'.
properties - Displays the properties of root project 'mixed-project'.
tasks - Displays the tasks runnable from root project 'mixed-project'.

Verification tasks
------------------
check - Runs all checks.
test - Runs the unit tests.

Rules
-----
Pattern: clean<TaskName>: Cleans the output files of a task.
Pattern: build<ConfigurationName>: Assembles the artifacts of a configuration.
Pattern: upload<ConfigurationName>: Assembles and uploads the artifacts belonging to a configurat\
ion.

To see all tasks and more detail, run with --all.

BUILD SUCCESSFUL
```

We don't have any code yet in our project so we don't have any task to run right now, but

it is good to know all these tasks can be executed once we have our code. Okay, we have to create our directory structure according to the conventions to make it all work without to much configuration. We can do this all by hand but we can also use a trick described in the Gradle cookbook. We add a new task to our `build.gradle` file to create all necessary directories for us.

```
task initProject(description: 'Initialize project directory structure.') << {
    // Default package to be created in each src dir.
    def defaultPackage = 'com/mrhaki/blog'

    ['java', 'groovy', 'resources'].each {
        // convention.sourceSets contains the directory structure
        // for our Groovy project. So we use this struture
        // and make a directory for each node.
        sourceSets*."${it}".srcDirs*.each { dir ->
            def newDir = new File(dir, defaultPackage)
            logger.info "Creating directory $newDir"  // gradle -i shows this message.
            newDir.mkdirs()  // Create dir.
        }
    }
}
```

At the command prompt we type `$ gradle initProject` and the complete source directory struture is now created. Let's add some Java and Groovy source files to our project. We keep it very simple, because this post is about Gradle and not so much about Java and Groovy. We create a Java interface in `src/main/java/com/mrhaki/blog/GreetingService.java`:

```
package com.mrhaki.blog;

public interface GreetingService {
    String greet(final String name);
}
```

We provide a Java implementation for this interface in `src/main/java/com/mrhaki/blog/-JavaGreetingImpl.java`:

```
package com.mrhaki.blog;

public class JavaGreetingImpl implements GreetingService {
    public String greet(final String name) {
        return "Hello " + (name != null ? name : "stranger") + ". Greeting from Java.";
    }
}
```

And a Groovy implementation in `src/main/groovy/com/mrhaki/blog/GroovyGreetingImpl.groovy`:

```
package com.mrhaki.blog

class GroovyGreetingImpl implements GreetingService {
    String greet(String name) {
        "Hello ${name ?: 'stranger'}. Greeting from Groovy"
    }
}
```

We have learned Gradle uses Groovy to define and execute the build script. But this bundled Groovy is not available for our project. We can choose which version of Groovy we want and don't have to rely on the version that is shipped with Gradle. We must define a dependency to the Groovy library version we want to use in our project in `build.gradle`. So we must add the following lines to the `build.gradle` file:

```
repositories {
    // Define Maven central repository to look for dependencies.
    mavenCentral()
}

dependencies {
    // group:name:version is a nice shortcut notation for dependencies.
    compile 'org.codehaus.groovy:groovy:2.3.9'
}
```

In our shell we type `$ gradle compileJava compileGroovy` to compile the source files we just created. If we didn't make any typos we should see the message `BUILD SUCCESSFUL` at the command prompt. Let's add some test classes to our project to test our simple implementations. We create `src/test/java/com/mrhaki/blog/JavaGreetingTest.java`:

```
package com.mrhaki.blog;

import static org.junit.Assert.*;
import org.junit.Test;

public class JavaGreetingTest {
    final GreetingService service = new JavaGreetingImpl();

    @Test public void testGreet() {
        assertEquals("Hello mrhaki. Greeting from Java.", service.greet("mrhaki"));
    }

    @Test public void testGreetNull() {
        assertEquals("Hello stranger. Greeting from Java.", service.greet(null));
    }
}
```

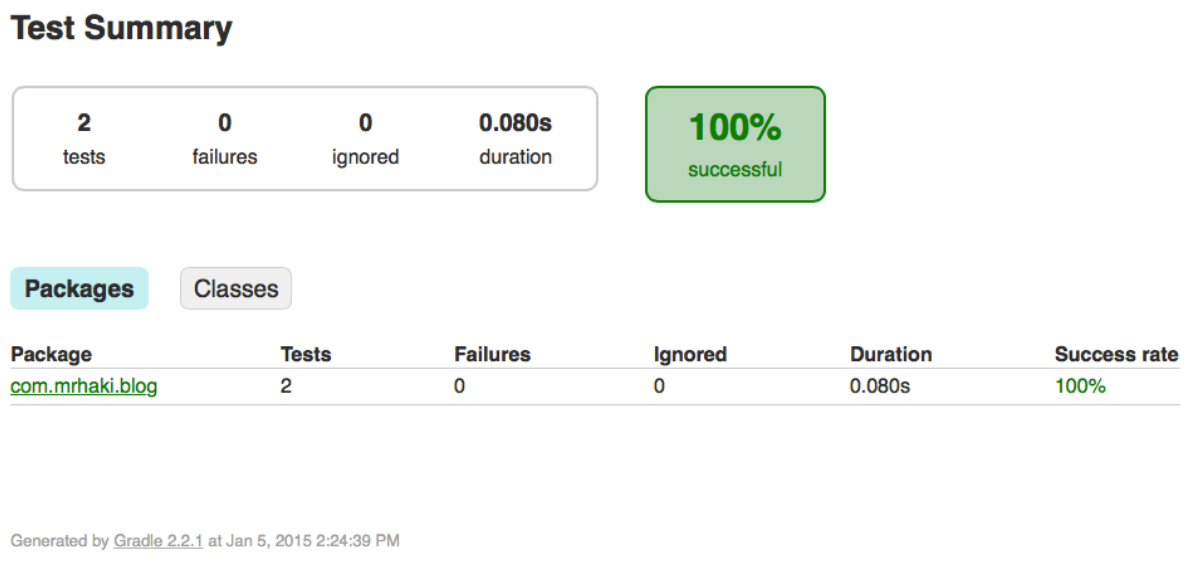
And we create a Groovy test class in `src/test/groovy/com/mrhaki/blog/GroovyGreetingTest.groovy`:

```
package com.mrhaki.blog

import static org.junit.Assert.*
import org.junit.Test

public class JavaGreetingTest {
    final GreetingService service = new JavaGreetingImpl()

    @Test public void testGreet() {
        assertEquals("Hello mrhaki. Greeting from Java.", service.greet("mrhaki"))
    }

    @Test public void testGreetNull() {
        assertEquals("Hello stranger. Greeting from Java.", service.greet(null))
    }
}
```

We add a dependency to `build.gradle` for JUnit:

```
dependencies {
    groovy 'org.codehaus.groovy:groovy:2.3.9'  // group:name:version is a nice shortcut notation \
for dependencies.
    testCompile 'junit:junit:4.11'
}
```

We return to the command prompt and type `$ gradle test`. Gradle compiles the code and runs the JUnit tests. The results of the test are stored in `build/reports/tests`. We can see the results in a web browser if we open `index.html`:



Let's leave the coding part for now. It it time to package our code. We can use `$ gradle build` to create a JAR file with the compiled classes from the `src/main` directories. But first we make a change to `build.gradle` to include a version number. If we run `$ gradle -r` we get an overview of all properties for our project. Among them is the version property. We can set a value for the version property in the `build.gradle` file. We also set the basename for the archive:

```
// The script is all Groovy, so we make use of all methods and features.
version = "1.0-${new Date().format('yyyyMMdd')"
archivesBaseName = 'greeting'
```

We return to the command prompt and type `$ gradle build`. Gradle runs and if all is successful we see in `build/libs` the file `greeting-1.0-20150105.jar`. Here is the complete `build.gradle` with all changes we made:

```
apply plugin: 'groovy'

version = "1.0-${new Date().format('yyyyMMdd')}" // The script is all Groovy, so we make use of a\
ll methods and features.
archivesBaseName = 'greeting'

repositories {
    mavenCentral()  // Define Maven central repository to look for dependencies.
}

dependencies {
    compile 'org.codehaus.groovy:groovy:2.3.9'  // group:name:version is a nice shortcut notation\
 for dependencies.
    testCompile 'junit:junit:4.11'
}

task initProject(description: 'Initialize project directory structure.') << {
    // Default package to be created in each src dir.
    def defaultPackage = 'com/mrhaki/blog'

    ['java', 'groovy', 'resources'].each {
        // convention.sourceSets contains the directory structure
        // for our Groovy project. So we use this struture
        // and make a directory for each node.
        sourceSets*."${it}".srcDirs.each { dirs ->
            dirs.each { dir ->
                def newDir = new File(dir, defaultPackage)
                logger.info "Creating directory $newDir"  // gradle -i shows this message.
                newDir.mkdirs()  // Create dir.
            }
        }
    }
}
```

We learned how we can start a new project from scratch and with little coding get a compiled and tested archive wth our code. In future blog posts we learn more about Gradle, for example the multi-project support.

Original written with Gradle 0.8 and re-written with Gradle 2.2.1.

Original blog post written on November 07, 2009.

A Groovy Multi-project with Gradle

Gradle is a flexible build system that uses Groovy for build scripts. In this post we create a very simple application demonstrating a multi-project build. We create a Groovy web

application with very simple domain, dataaccess, services and web projects. The sample is not to demonstrate Groovy but to show the multi-project build support in Gradle.

We start by creating a new application directory `app` and create two files `settings.gradle` and `build.gradle`:

```
$ mkdir app
$ cd app
$ touch settings.gradle
$ touch build.gradle
```

We open the file `settings.gradle` in a text editor. With the `include` method we define the subprojects for the application:

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

Next we open `build.gradle` in the text editor. This build file is our main build file for the application. We can define all settings for the subprojects in this file:

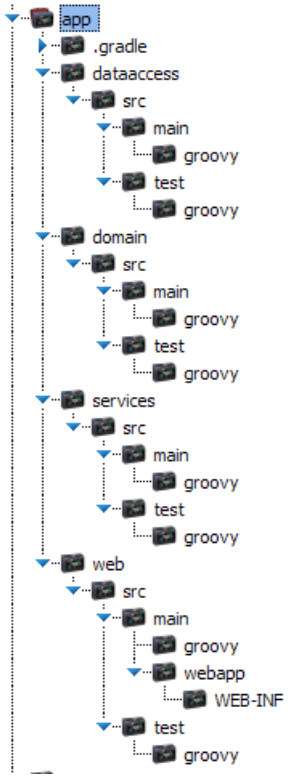
```
subprojects {
    apply plugin: 'groovy'
    version = '1.0.0-SNAPSHOT'
    group = 'com.mrhaki.blog'

    // Make sure transitive project dependencies are resolved.
    configurations.compile.transitive = true

    repositories {
        mavenCentral()
    }

    dependencies {
        coimpile 'org.codehaus.groovy:groovy:2.3.9'
    }

    task initProject(description: 'Initialize project') << { task ->
        task.project.sourceSets*.groovy.srcDirs*.each {
            println "Create $it"
            it.mkdirs()
        }
    }
}

project(':dataaccess') {
    dependencies {
        compile project(':domain')
    }
}

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

```
}

project(':web') {
    // jetty plugin extends war plugin,
    // so we get all war plugin functionality as well.
    apply plugin: 'jetty'

    dependencies {
        compile project(':services')
        // Because configurations.compile.transitive = true
        // we only have to specify services project,
        // although we also reference dataaccess and domain projects.
    }

    // Add extra code to initProject task.
    initProject << { task ->
        def webInfDir = new File(task.project.webAppDir, '/WEB-INF')
        println "Create $webInfDir"
        webInfDir.mkdirs()
    }
}
```

The `subprojects` method accepts a closure and here we define common settings for all subprojects. The `project` method allows us to fine tune the definiton of a subproject. For each project we define project dependencies between the different projects for the compile configuration. This is a very powerful feature of Gradle, we define the project dependency and Gradle will make sure the dependent project is first build before the project that needs it. This even works if we invoke a build command from a subproject. For example if we run `gradle build` from the `web` project, all dependent projects are build first.

We also create a new task `initProject` for all subprojects. This task creates the Groovy source directories. In the `web` project we add an extra statement to the task to create the `src/main/webapp/WEB-INF` directory. This shows we can change a task definition in a specific subproject.

Okay it is time to let Gradle create our directories: `$ gradle initProject`. After the script is finished we have a new directory structure:



It is time to add some files to the different projects. As promised we keep it very, very simple. We define a domain class `Language`, a class in dataaccess to get a list of `Language` objects, a services class to filter out the Groovy language and a web component to get the name property for the `Language` object and a Groovlet to show it in the web browser. Finally we add a `web.xml` so we can execute the Groovlet.

```
// File: app/domain/src/main/groovy/com/mrhaki/blog/domain/Language.groovy
package com.mrhaki.blog.domain

class Language {
    String name
}
```

```
    // File: app/dataaccess/src/main/groovy/com/mrhaki/blog/data/LanguageDao.groovy
package com.mrhaki.blog.data

import com.mrhaki.blog.domain.Language

class LanguageDao {
    List findAll() {
        [new Language(name: 'Java'), new Language(name: 'Groovy'), new Language(name: 'Scala')]
    }
}
```

```
// File: app/services/src/main/groovy/com/mrhaki/blog/service/LanguageService.groovy
package com.mrhaki.blog.service

import com.mrhaki.blog.domain.Language
import com.mrhaki.blog.data.LanguageDao

class LanguageService {
    def dao = new LanguageDao()

    Language findGroovy() {
       dao.findAll().find { it.name == 'Groovy' }
    }
}
```

```
// File: app/web/src/main/groovy/com/mrhaki/blog/web/LanguageHelper.groovy
package com.mrhaki.blog.web

import com.mrhaki.blog.service.LanguageService

class LanguageHelper {
    def service = new LanguageService()

    String getGroovyValue() {
        service.findGroovy()?.name ?: 'Groovy language not found'
    }
}
```

```
// File: app/web/src/main/webapp/language.groovy
import com.mrhaki.blog.web.LanguageHelper

def helper = new LanguageHelper()

html.html {
    head {
        title "Simple page"
    }
    body {
        h1 "Simple page"
        p "My favorite language is '$helper.groovyValue'."
    }
}
```

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- File: app/web/src/main/webapp/WEB-INF/web.xml -->
<web-app>
    <servlet>
        <servlet-name>Groovy</servlet-name>
        <servlet-class>groovy.servlet.GroovyServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Groovy</servlet-name>
        <url-pattern>*.groovy</url-pattern>
    </servlet-mapping>
</web-app>
```

We have created all the files and it is time to see the result. Thanks to the Jetty plugin we only have to invoke the `jettyRun` tasks and all files (and dependent projects) are compiled and processed:

```
$ cd web
$ gradle jettyRun
:domain:compileJava
:domain:compileGroovy
:domain:processResources
:domain:classes
:domain:jar
:domain:uploadDefaultInternal
:dataaccess:compileJava
:dataaccess:compileGroovy
:dataaccess:processResources
:dataaccess:classes
:dataaccess:jar
:dataaccess:uploadDefaultInternal
:services:compileJava
:services:compileGroovy
:services:processResources
:services:classes
:services:jar
:services:uploadDefaultInternal
:web:compileJava
:web:compileGroovy
:web:processResources
:web:classes
:web:jettyRun
```

We open a web browser and go to *http://localhost:8080/web/language.groovy* and get a simple web page with the results of all our labour:

This concludes this blog about the multi-project support of Gradle. What we need to remember is Gradle is great in resolving dependencies between projects. If one project dependents on another we don't have to worry about first compiling the dependent project, Gradle does this for us. We can define tasks for each project, but still fine tune a task for a specific project. Also we have a certain freedom about the project structure, as long as we define the needed projects in the `settings.gradle` all will be fine. Also we only need one `build.gradle` (but can be more per project if we want) to configure all projects.

Original blog post written with Gradle 0.8 and re-written with Gradle 2.2.1.

Original blog post written on November 12, 2009.

Shortcut Notation for Dependencies

In a Gradle build script we define dependencies with the `dependencies` method. We can use a map notation or the short string notation. The string notation is the following format: *group:name:version*.

```
dependencies {
    compile group: 'commons-lang', name: 'commons-lang', version: '2.4'  // Map notation.
    compile(
        [group: 'commons-lang', name: 'commons-lang', version: '2.4'],
        [group: 'commons-io', name: 'commons-io', version: '1.4']
    )  // Multiple map notation.
    compile 'commons-lang:commons-lang:2.4', 'commons-io:commons-io:1.4'  // String notation.
}
```

Written with Gradle 0.8.

Original blog post written on November 17, 2009.

Display Dependencies for a Gradle Build

We can see the dependencies defined in our project by using the `dependencies` tasks. We get an overview of the dependencies for each configuration in our Gradle build.

```
// File: build.gradle
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    compile 'org.springframework:spring-webmvc:2.5.6'
}


$ gradle dependencies
:dependencies

------------------------------------------------------------
Root project
------------------------------------------------------------

archives - Configuration for archive artifacts.
No dependencies

compile - Compile classpath for source set 'main'.
\--- org.springframework:spring-webmvc:2.5.6
     +--- commons-logging:commons-logging:1.1.1
     +--- org.springframework:spring-beans:2.5.6
     |    +--- commons-logging:commons-logging:1.1.1
     |    \--- org.springframework:spring-core:2.5.6
     |         \--- commons-logging:commons-logging:1.1.1
     +--- org.springframework:spring-context:2.5.6
     |    +--- aopalliance:aopalliance:1.0
     |    +--- commons-logging:commons-logging:1.1.1
     |    +--- org.springframework:spring-beans:2.5.6 (*)
     |    \--- org.springframework:spring-core:2.5.6 (*)
     +--- org.springframework:spring-context-support:2.5.6
     |    +--- aopalliance:aopalliance:1.0
     |    +--- commons-logging:commons-logging:1.1.1
     |    +--- org.springframework:spring-beans:2.5.6 (*)
     |    +--- org.springframework:spring-context:2.5.6 (*)
     |    \--- org.springframework:spring-core:2.5.6 (*)
     +--- org.springframework:spring-core:2.5.6 (*)
     \--- org.springframework:spring-web:2.5.6
          +--- commons-logging:commons-logging:1.1.1
          +--- org.springframework:spring-beans:2.5.6 (*)
          +--- org.springframework:spring-context:2.5.6 (*)
          \--- org.springframework:spring-core:2.5.6 (*)

default - Configuration for default artifacts.
\--- org.springframework:spring-webmvc:2.5.6
     +--- commons-logging:commons-logging:1.1.1
     +--- org.springframework:spring-beans:2.5.6
     |    +--- commons-logging:commons-logging:1.1.1
     |    \--- org.springframework:spring-core:2.5.6
     |         \--- commons-logging:commons-logging:1.1.1
     +--- org.springframework:spring-context:2.5.6
```

```
     |    +--- aopalliance:aopalliance:1.0
     |    +--- commons-logging:commons-logging:1.1.1
     |    +--- org.springframework:spring-beans:2.5.6 (*)
     |    \--- org.springframework:spring-core:2.5.6 (*)
     +--- org.springframework:spring-context-support:2.5.6
     |    +--- aopalliance:aopalliance:1.0
     |    +--- commons-logging:commons-logging:1.1.1
     |    +--- org.springframework:spring-beans:2.5.6 (*)
     |    +--- org.springframework:spring-context:2.5.6 (*)
     |    \--- org.springframework:spring-core:2.5.6 (*)
     +--- org.springframework:spring-core:2.5.6 (*)
     \--- org.springframework:spring-web:2.5.6
          +--- commons-logging:commons-logging:1.1.1
          +--- org.springframework:spring-beans:2.5.6 (*)
          +--- org.springframework:spring-context:2.5.6 (*)
          \--- org.springframework:spring-core:2.5.6 (*)

runtime - Runtime classpath for source set 'main'.
\--- org.springframework:spring-webmvc:2.5.6
     +--- commons-logging:commons-logging:1.1.1
     +--- org.springframework:spring-beans:2.5.6
     |    +--- commons-logging:commons-logging:1.1.1
     |    \--- org.springframework:spring-core:2.5.6
     |         \--- commons-logging:commons-logging:1.1.1
     +--- org.springframework:spring-context:2.5.6
     |    +--- aopalliance:aopalliance:1.0
     |    +--- commons-logging:commons-logging:1.1.1
     |    +--- org.springframework:spring-beans:2.5.6 (*)
     |    \--- org.springframework:spring-core:2.5.6 (*)
     +--- org.springframework:spring-context-support:2.5.6
     |    +--- aopalliance:aopalliance:1.0
     |    +--- commons-logging:commons-logging:1.1.1
     |    +--- org.springframework:spring-beans:2.5.6 (*)
     |    +--- org.springframework:spring-context:2.5.6 (*)
     |    \--- org.springframework:spring-core:2.5.6 (*)
     +--- org.springframework:spring-core:2.5.6 (*)
     \--- org.springframework:spring-web:2.5.6
          +--- commons-logging:commons-logging:1.1.1
          +--- org.springframework:spring-beans:2.5.6 (*)
          +--- org.springframework:spring-context:2.5.6 (*)
          \--- org.springframework:spring-core:2.5.6 (*)

testCompile - Compile classpath for source set 'test'.
\--- org.springframework:spring-webmvc:2.5.6
     +--- commons-logging:commons-logging:1.1.1
     +--- org.springframework:spring-beans:2.5.6
     |    +--- commons-logging:commons-logging:1.1.1
     |    \--- org.springframework:spring-core:2.5.6
     |         \--- commons-logging:commons-logging:1.1.1
     +--- org.springframework:spring-context:2.5.6
     |    +--- aopalliance:aopalliance:1.0
     |    +--- commons-logging:commons-logging:1.1.1
     |    +--- org.springframework:spring-beans:2.5.6 (*)
     |    \--- org.springframework:spring-core:2.5.6 (*)
     +--- org.springframework:spring-context-support:2.5.6
```

```
     |    +--- aopalliance:aopalliance:1.0
     |    +--- commons-logging:commons-logging:1.1.1
     |    +--- org.springframework:spring-beans:2.5.6 (*)
     |    +--- org.springframework:spring-context:2.5.6 (*)
     |    \--- org.springframework:spring-core:2.5.6 (*)
     +--- org.springframework:spring-core:2.5.6 (*)
     \--- org.springframework:spring-web:2.5.6
          +--- commons-logging:commons-logging:1.1.1
          +--- org.springframework:spring-beans:2.5.6 (*)
          +--- org.springframework:spring-context:2.5.6 (*)
          \--- org.springframework:spring-core:2.5.6 (*)

testRuntime - Runtime classpath for source set 'test'.
\--- org.springframework:spring-webmvc:2.5.6
     +--- commons-logging:commons-logging:1.1.1
     +--- org.springframework:spring-beans:2.5.6
     |    +--- commons-logging:commons-logging:1.1.1
     |    \--- org.springframework:spring-core:2.5.6
     |         \--- commons-logging:commons-logging:1.1.1
     +--- org.springframework:spring-context:2.5.6
     |    +--- aopalliance:aopalliance:1.0
     |    +--- commons-logging:commons-logging:1.1.1
     |    +--- org.springframework:spring-beans:2.5.6 (*)
     |    \--- org.springframework:spring-core:2.5.6 (*)
     +--- org.springframework:spring-context-support:2.5.6
     |    +--- aopalliance:aopalliance:1.0
     |    +--- commons-logging:commons-logging:1.1.1
     |    +--- org.springframework:spring-beans:2.5.6 (*)
     |    +--- org.springframework:spring-context:2.5.6 (*)
     |    \--- org.springframework:spring-core:2.5.6 (*)
     +--- org.springframework:spring-core:2.5.6 (*)
     \--- org.springframework:spring-web:2.5.6
          +--- commons-logging:commons-logging:1.1.1
          +--- org.springframework:spring-beans:2.5.6 (*)
          +--- org.springframework:spring-context:2.5.6 (*)
          \--- org.springframework:spring-core:2.5.6 (*)

(*) - dependencies omitted (listed previously)

BUILD SUCCESSFUL

Total time: 3.669 secs
$
```

Another way is to add the plugin `project-report` to our project. We now get the the task `dependencyReport` in our project. When we run `$ gradle dependencyReport` we get a text file `build/reports/project/dependencies.txt` with the same overview of the dependencies.

Written with Gradle 0.8 and re-written with Gradle 2.2.1.

Original blog post written on November 21, 2009.

Getting More Dependency Insight

In most of our projects we have dependencies on other code, like libraries or other projects. Gradle has a nice DSL to define dependencies. Dependencies are grouped in dependency configurations. These configuration can be created by ourselves or added via a plugin. Once we have defined our dependencies we get a nice overview of all dependencies in our project with the `dependencies` task. We can add the optional argument `--configuration` to only see dependencies for the given configuration. But we can even check for a specific dependency where it is used, any transitive dependencies and how the version is resolved.

In the following sample build we define a compile dependency on Spring Boot and SLF4J API. The SLF4J API is also a transitive dependency for the Spring Boot dependency, so we can see how the `dependencyInsight` tasks shows a version conflict.

```
apply plugin: 'java'

// Set Bintray JCenter as repository.
repositories.jcenter()

dependencies {
    // Set dependency for Spring Boot
    compile "org.springframework.boot:spring-boot-starter-web:1.1.5.RELEASE"

    // Set dependency for SLF4J with conflicting version.
    compile 'org.slf4j:slf4j-api:1.7.1'
}
```

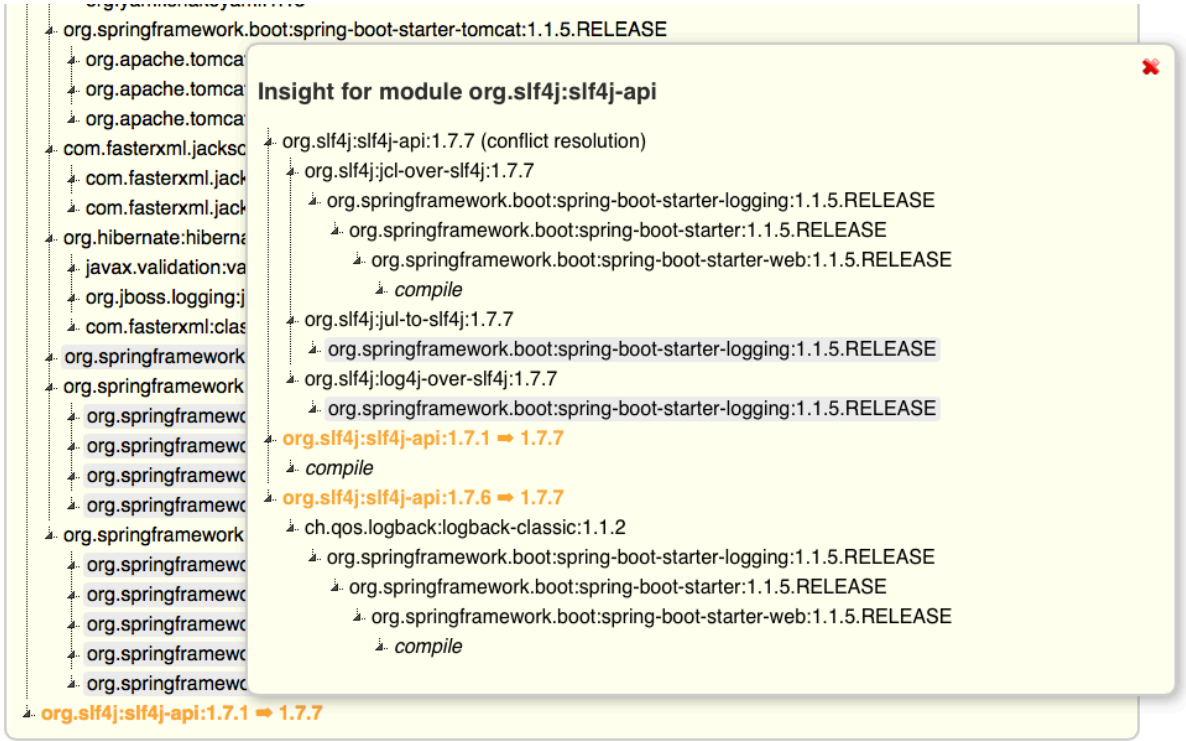
Now let's run the `dependencyInsight` task for the dependency SLF4J API in the compile configuration:

```
$ gradle -q dependencyInsight --configuration compile --dependency slf4j-api
org.slf4j:slf4j-api:1.7.7 (conflict resolution)
+--- org.slf4j:jcl-over-slf4j:1.7.7
|    \--- org.springframework.boot:spring-boot-starter-logging:1.1.5.RELEASE
|         \--- org.springframework.boot:spring-boot-starter:1.1.5.RELEASE
|              \--- org.springframework.boot:spring-boot-starter-web:1.1.5.RELEASE
|                   \--- compile
+--- org.slf4j:jul-to-slf4j:1.7.7
|    \--- org.springframework.boot:spring-boot-starter-logging:1.1.5.RELEASE (*)
\--- org.slf4j:log4j-over-slf4j:1.7.7
     \--- org.springframework.boot:spring-boot-starter-logging:1.1.5.RELEASE (*)

org.slf4j:slf4j-api:1.7.1 -> 1.7.7
\--- compile

org.slf4j:slf4j-api:1.7.6 -> 1.7.7
\--- ch.qos.logback:logback-classic:1.1.2
     \--- org.springframework.boot:spring-boot-starter-logging:1.1.5.RELEASE
          \--- org.springframework.boot:spring-boot-starter:1.1.5.RELEASE
               \--- org.springframework.boot:spring-boot-starter-web:1.1.5.RELEASE
                    \--- compile

(*) - dependencies omitted (listed previously)
```

In the output we can see slf4j-api is referenced three times, once as a transitive dependency for jcl-over-slf4j, jul-to-slf4j and log4j-over-slf4j, once as transitive dependency for logback-classic and once as a direct dependency for the compile configuration. We also see the version is bumped to 1.7.7 where necessary, because the transitive dependency of jcl-over-slf4j defines the newest version.

The value we use for the `--dependency` option is used to do partial matching in the group, name or version properties of the dependencies. For example to see an insight in all dependencies with logging we can invoke `$ gradle dependencyInsight --dependency logging`.

We can also get an HTML report page with an overview of all dependencies. To get dependency insight we must click on the desired dependency from the HTML page and we get a similar output as via the command-line. First we must add the project-report plugin to our project. Next we invoke the `dependencyReport` task. When the task is finished we can open `build/reports/project/dependencies/index.html` in our web browser. When we navigate to the compile configuration and click on the slf4j-api dependency we get the following output:



Written with Gradle 2.0.

Original blog post written on August 11, 2014.

Exclude Transitive Dependency from All Configurations

We can exclude transitive dependencies easily from specific configurations. To exclude them from all configurations we can use Groovy's spread-dot operator and invoke the `exclude()` method on each configuration. We can only define the group, module or both as arguments for the `exclude()` method.

The following part of a build file shows how we can exclude a dependency from all configurations:

```
...
configurations {
    all*.exclude group: 'xml-apis', module: 'xmlParserAPIs'
}

// Equivalent to:
configurations {
    all.collect { configuration ->
        configuration.exclude group: 'xml-apis', module: 'xmlParserAPIs'
    }
}
...
```

Written with Gradle 1.2.

Original blog post written on October 23, 2012.

Adding Dependencies Only for Packaging to War

My colleague, Tom Wetjens, wrote a blog post Package-only dependencies in Maven. He showed a Maven solution when we want to include dependencies in the WAR file, which are not used in any other scopes. In this blog post we will see how we solve this in Gradle.

Suppose we use the SLF4J Logging API in our project. We use the API as a compile dependency, because our code uses this API. But in our test runtime we want to use the SLF4J Simple implementation of this API. And in our WAR file we want to include the Logback implementation of the API. The Logback dependency is only needed to be included in the WAR file and shouldn't exist in any other dependency configuration.

We first add the War plugin to our project. The `war` task uses the `runtime` dependency configuration to determine which files are added to the `WEB-INF/lib` directory in our WAR file. We add a new dependency configuration `warLib` that extends the `runtime` configuration in our project.

```
apply plugin: 'war'

repositories.jcenter()

configurations {
    // Create new dependency configuration
    // for dependencies to be added in
    // WAR file.
    warLib.extendsFrom runtime
}

dependencies {
    // API dependency for Slf4j.
    compile 'org.slf4j:slf4j-api:1.7.7'
```

```
    testCompile 'junit:junit:4.11'

    // Slf4j implementation used for tests.
    testRuntime 'org.slf4j:slf4j-simple:1.7.7'

    // Slf4j implementation to be packaged
    // in WAR file.
    warLib 'ch.qos.logback:logback-classic:1.1.2'
}

war {
    // Add warLib dependency configuration
    classpath configurations.warLib

    // We remove all duplicate files
    // with this assignment.
    // geFiles() method return a unique
    // set of File objects, removing
    // any duplicates from configurations
    // added by classpath() method.
    classpath = classpath.files
}
```

We can now run the `build` task and we get a WAR file with the following contents:

```
$ gradle build
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:war
:assemble
:compileTestJava
:processTestResources UP-TO-DATE
:testClasses
:test
:check
:build

BUILD SUCCESSFUL

Total time: 6.18 secs
$ jar tvf build/libs/package-only-dep-example.war
     0 Fri Sep 19 05:59:54 CEST 2014 META-INF/
    25 Fri Sep 19 05:59:54 CEST 2014 META-INF/MANIFEST.MF
     0 Fri Sep 19 05:59:54 CEST 2014 WEB-INF/
     0 Fri Sep 19 05:59:54 CEST 2014 WEB-INF/lib/
 29257 Thu Sep 18 14:36:24 CEST 2014 WEB-INF/lib/slf4j-api-1.7.7.jar
270750 Thu Sep 18 14:36:24 CEST 2014 WEB-INF/lib/logback-classic-1.1.2.jar
427729 Thu Sep 18 14:36:26 CEST 2014 WEB-INF/lib/logback-core-1.1.2.jar
   115 Wed Sep 03 09:24:40 CEST 2014 WEB-INF/web.xml
```

Also when we run the `dependencies` task we can see how the implementations of the SLF4J API relate to the dependency configurations:

```
$ gradle dependencies
:dependencies

------------------------------------------------------------
Root project
------------------------------------------------------------

archives - Configuration for archive artifacts.
No dependencies

compile - Compile classpath for source set 'main'.
\--- org.slf4j:slf4j-api:1.7.7

default - Configuration for default artifacts.
\--- org.slf4j:slf4j-api:1.7.7

providedCompile - Additional compile classpath for libraries that should not be part of the WAR a\
rchive.
No dependencies

providedRuntime - Additional runtime classpath for libraries that should not be part of the WAR a\
rchive.
No dependencies

runtime - Runtime classpath for source set 'main'.
\--- org.slf4j:slf4j-api:1.7.7

testCompile - Compile classpath for source set 'test'.
+--- org.slf4j:slf4j-api:1.7.7
\--- junit:junit:4.11
     \--- org.hamcrest:hamcrest-core:1.3

testRuntime - Runtime classpath for source set 'test'.
+--- org.slf4j:slf4j-api:1.7.7
+--- junit:junit:4.11
|    \--- org.hamcrest:hamcrest-core:1.3
\--- org.slf4j:slf4j-simple:1.7.7
     \--- org.slf4j:slf4j-api:1.7.7

warLib
+--- org.slf4j:slf4j-api:1.7.7
\--- ch.qos.logback:logback-classic:1.1.2
     +--- ch.qos.logback:logback-core:1.1.2
     \--- org.slf4j:slf4j-api:1.7.6 -> 1.7.7

(*) - dependencies omitted (listed previously)

BUILD SUCCESSFUL

Total time: 6.274 secs
```

Code written with Gradle 2.1.

Original blog post written on September 19, 2014.

Run Java Application From Build Script

Gradle has a special task to run a Java class from the build script: `org.gradle.api.tasks.JavaExec`. We can for example create a new task of type `JavaExec` and use a closure to configure the task. We can set the main class, classpath, arguments, JVM arguments and more to run the application.

Gradle also has the `javaexec()` method available as part of a Gradle project. This means we can invoke `javaexec()` directly from the build script and use the same closure to configure the Java application that we want to invoke.

Suppose we have a simple Java application:

```
// File: src/main/java/com/mrhaki/java/Simple.java
package com.mrhaki.java;

public class Simple {

    public static void main(String[] args) {
        System.out.println(System.getProperty("simple.message") + args[0] + " from Simple.");
    }

}
```

And we have the following Gradle build file to run `Simple`:

```
// File: build.gradle
apply plugin: 'java'

task(runSimple, dependsOn: 'classes', type: JavaExec) {
    main = 'com.mrhaki.java.Simple'
    classpath = sourceSets.main.runtimeClasspath
    args 'mrhaki'
    systemProperty 'simple.message', 'Hello '
}

defaultTasks 'runSimple'

// javaexec() method also available for direct invocation
// javaexec {
//    main = 'com.mrhaki.java.Simple'
//    classpath = sourceSets.main.runtimeClasspath
//    args 'mrhaki'
//    systemProperty 'simple.message', 'Hello '
// }
```

We can execute our Gradle build script and get the following output:

```
$ gradle
:compileJava
:processResources
:classes
:runSimple
Hello mrhaki from Simple.

BUILD SUCCESSFUL

Total time: 4.525 secs
```

Written with Gradle 0.9-rc-1.

Original blog post written on September 24, 2010.

Running Java Applications from External Dependency

With Gradle we can execute Java applications using the `JavaExec` task or the `javaexec()` method. If we want to run Java code from an external dependency we must first pull in the dependency with the Java application code. The best way to do this is to create a new dependency configuration. When we configure a task with type `JavaExec` we can set the classpath to the external dependency. Notice we cannot use the `buildscript{}` script block to set the classpath. A `JavaExec` task will fork a new Java process so any classpath settings via `buildscript{}` are ignored.

In the following example build script we want to execute the Java class `org.apache.cxf.tools.wsdlto.WSDL` from Apache CXF to generate Java classes from a given WSDL. We define a new dependency configuration with the name `cxf` and use it to assign the CXF dependencies to it. We use the `classpath` property of the `JavaExec` task to assign the configuration dependency.

```
// File: build.gradle

// Base plugin for task rule clean<task>
apply plugin: 'base'

repositories.mavenCentral()

// New configuration for CXF dependencies.
configurations { cxf }

ext {
    // CXF version.
    cxfVersion = '2.6.2'

    // Artifacts for CXF dependency.
    cxfArtifacts = [
        'cxf-tools-wsdlto-frontend-jaxws',
        'cxf-tools-wsdlto-databinding-jaxb',
        'cxf-tools-common',
        'cxf-tools-wsdlto-core'
    ]
}
```

```
dependencies {
    // Assign CXF dependencies to configuration.
    cxfArtifacts.each { artifact ->
        cxf "org.apache.cxf:$artifact:$cxfVersion"
    }
}

// Custom task to generate Java classes
// from WSDL.
task wsdl2java(type: JavaExec) {
    ext {
        wsdlFile = 'src/wsdl/service-contract.wsdl'
        outputDir = file("$buildDir/generated/cxf")
    }

    inputs.file file(wsdlFile)
    outputs.dir outputDir

    // Main Java class to invoke.
    main = 'org.apache.cxf.tools.wsdlto.WSDLToJava'

    // Set classpath to dependencies assigned
    // to the cxf configuration.
    classpath = configurations.cxf

    // Arguments to be passed to WSDLToJava.
    args '-d', outputDir
    args '-client'
    args '-verbose'
    args '-validate'
    args wsdlFile
}
```

Code written with Gradle 1.2.

Original blog post written on October 22, 2012.

Pass Java System Properties To Java Tasks

Gradle is of course a great build tool for Java related projects. If we have tasks in our projects that need to execute a Java application we can use the `JavaExec` task. When we need to pass Java system properties to the Java application we can set the `systemProperties` property of the `JavaExec` task. We can assign a value to the `systemProperties` property or use the method `systemProperties` that will add the properties to the existing properties already assigned. Now if we want to define the system properties from the command-line when we run Gradle we must pass along the properties to the task. Therefore we must reconfigure a `JavaExec` task and assign `System.properties` to the `systemProperties` property.

In the following build script we reconfigure all `JavaExec` tasks in the project. We use the `systemProperties` method and use the value `System.properties`. This means any system properties from the command-line are passed on to the `JavaExec` task.

```
apply plugin: 'groovy'
apply plugin: 'application'

mainClassName = 'com.mrhaki.sample.Application'

repositories.jcenter()

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

// The run task added by the application plugin
// is also of type JavaExec.
tasks.withType(JavaExec) {
    // Assign all Java system properties from
    // the command line to the JavaExec task.
    systemProperties System.properties
}
```

We write a simple Groovy application that uses a Java system property `app.greeting` to print a message to the console:

```
// File: src/main/groovy/com/mrhaki/sample/Application.groovy
package com.mrhaki.sample

println "Hello ${System.properties['app.greeting']}"
```

Now when we execute the `run` task (of type `JavaExec`) and define the Java system property `app.greeting` in our command it is used by the application:

```
$ gradle -Dapp.greeting=Gradle! -q run
Hello Gradle!
```

Written with Gradle 2.7.

Original blog post written on September 21, 2015.

Running Groovy Scripts as Application

In a previous post we learned how to run a Java application in a Gradle project. The Java source file with a `main` method is part of the project and we use the `JavaExec` task to run the Java code. We can use the same `JavaExec` task to run a Groovy script file.

A Groovy script file doesn't have an explicit `main` method, but it is added when we compile the script file. The name of the script file is also the name of the generated class, so we use that name for the `main` property of the `JavaExec` task. Let's first create simple Groovy script file to display the current date. We can pass an extra argument with the date format we wan't to use.

```
// File: src/main/groovy/com/mrhaki/CurrentDate.groovy
package com.mrhaki

// If an argument is passed we assume it is the
// date format we want to use.
// Default format is dd-MM-yyyy.
final String dateFormat = args ? args[0] : 'dd-MM-yyyy'

// Output formatted current date and time.
println "Current date and time: ${new Date().format(dateFormat)}"
```

Our Gradle build file contains the task `runScript` of type `JavaExec`. We rely on the Groovy libraries included with Gradle, because we use `localGroovy()` as a compile dependency. Of course we can change this to refer to another Groovy version if we want to using the group, name and version notation together with a valid repository.

```
// File: build.gradle
apply plugin: 'groovy'

dependencies {
    compile localGroovy()
}

task runScript(type: JavaExec) {
    description 'Run Groovy script'

    // Set main property to name of Groovy script class.
    main = 'com.mrhaki.CurrentDate'

    // Set classpath for running the Groovy script.
    classpath = sourceSets.main.runtimeClasspath

    if (project.hasProperty('custom')) {
        // Pass command-line argument to script.
        args project.getProperty('custom')
    }
}

defaultTasks 'runScript'
```

We can run the script with or without the project property `custom` and we see the changes in the output:

```
$ gradle -q
Current date and time: 29-09-2014
$ gradle -q -Pcustom=yyyyMMdd
Current date and time: 20140929
$ gradle -q -Pcustom=yyyy
Current date and time: 2014
```

Code written with Gradle 2.1.

Original blog post written on September 29, 2014.

Alter Start Scripts from Application Plugin

For Java or Groovy projects we can use the application plugin in Gradle to run and package our application. The plugin adds for example the `startScripts` task which creates OS specific scripts to run the project as a JVM application. This task is then used again by the `installDist` that installs the application, and `distZip` and `distTar` tasks that create a distributable archive of the application. The `startScripts` tasks has the properties `unixScript` and `windowsScript` that are the actual OS specific script files to run the application. We can use these properties to change the contents of the files.

In the following sample we add the directory `configuration` to the `CLASSPATH` definition:

```
...
startScripts {

    // Support closures to add an additional element to
    // CLASSPATH definition in the start script files.
    def configureClasspathVar = { findClasspath, pathSeparator, line ->

        // Looking for the line that starts with either CLASSPATH=
        // or set CLASSPATH=, defined by the findClasspath closure argument.
        line = line.replaceAll(~/^${findClasspath}=.*$/) { original ->

            // Get original line and append it
            // with the configuration directory.
            // Use specified path separator, which is different
            // for Windows or Unix systems.
            original += "${pathSeparator}configuration"
        }

    }

    def configureUnixClasspath = configureClasspathVar.curry('CLASSPATH', ':')
    def configureWindowsClasspath = configureClasspathVar.curry('set CLASSPATH', ';')

    // The default script content is generated and
    // with the doLast method we can still alter
    // the contents before the complete task ends.
    doLast {

        // Alter the start script for Unix systems.
        unixScript.text =
            unixScript
                .readLines()
                .collect(configureUnixClasspath)
                .join('\n')

        // Alter the start script for Windows systems.
        windowsScript.text =
            windowsScript
                .readLines()
                .collect(configureWindowsClasspath)
                .join('\r\n')
```

```
    }

}
...
```

This post was inspired by the Gradle build file I saw at the Gaiden project.

Written with Gradle 2.3.

Original blog post written on April 19, 2015.

Running Groovy Scripts Like From Groovy Command Line

In a previous post we have seen how to execute a Groovy script in our source directories. But what if we want to use the Groovy command line to execute a Groovy script? Suppose we want to evaluate a small Groovy script expressed by a String value, that we normally would invoke like `$ groovy -e "println 'Hello Groovy!'"`. Or we want to use the command line option `-l` to start Groovy in listening mode with a script to handle requests. We can achieve this by creating a task with type `JavaExec` or by using the Gradle `javaexec` method. We must set the Java main class to `groovy.ui.Main` which is the class that is used for running the Groovy command line.

In the following sample build file we create a new task `runGroovyScript` of type `JavaExec`. We also create a new dependency configuration `groovyScript` so we can use a separate class path for running our Groovy scripts.

```
// File: build.gradle

repositories {
    jcenter()
}

// Add new configuration for
// dependencies needed to run
// Groovy command line scripts.
configurations {
    groovyScript
}

dependencies {
    // Set Groovy dependency so
    // groovy.ui.GroovyMain can be found.
    groovyScript localGroovy()
    // Or be specific for a version:
    //groovyScript "org.codehaus.groovy:groovy-all:2.4.5"
}

// New task to run Groovy command line
// with arguments.
task runGroovyScript(type: JavaExec) {

    // Set class path used for running
    // Groovy command line.
```

```
    classpath = configurations.groovyScript

    // Main class that runs the Groovy
    // command line.
    main = 'groovy.ui.GroovyMain'

    // Pass command line arguments.
    args '-e', "println 'Hello Gradle!'"

}
```

We can run the task `runGroovyScript` and we see the output of our small Groovy script `println 'Hello Gradle!'`:

```
$ gradle runGroovyScript
:runGroovyScript
Hello Gradle!

BUILD SUCCESSFUL

Total time: 1.265 secs
$
```

Let's write another task where we use the simple HTTP server from the Groovy examples to start a HTTP server with Gradle. This can be useful if we have a project with static HTML files and want to serve them via a web server:

```
// File: build.gradle

repositories {
    jcenter()
}

configurations {
    groovyScript
}

dependencies {
    groovyScript localGroovy()
}

task runHttpServer(type: JavaExec) {
    classpath = configurations.groovyScript
    main = 'groovy.ui.GroovyMain'

    // Start Groovy in listening mode on
    // port 8001.
    args '-l', '8001'

    // Run simple HTTP server.
    args '-e', '''\
// init variable is true before
// the first client request, so
```

```
// the following code is executed once.
if (init) {
    headers = [:]
    binaryTypes = ["gif","jpg","png"]
    mimeTypes = [
        "css" : "text/css",
        "gif" : "image/gif",
        "htm" : "text/html",
        "html": "text/html",
        "jpg" : "image/jpeg",
        "png" : "image/png"
    ]
    baseDir = System.properties['baseDir'] ?: '.'
}

// parse the request
if (line.toLowerCase().startsWith("get")) {
    content = line.tokenize()[1]
} else {
    def h = line.tokenize(":")
    headers[h[0]] = h[1]
}

// all done, now process request
if (line.size() == 0) {
    processRequest()
    return "success"
}

def processRequest() {
    if (content.indexOf("..") < 0) { //simplistic security
        // simple file browser rooted from current dir
        def file = new File(new File(baseDir), content)
        if (file.isDirectory()) {
            printDirectoryListing(file)
        } else {
            extension = content.substring(content.lastIndexOf(".") + 1)
            printHeaders(mimeTypes.get(extension,"text/plain"))

            if (binaryTypes.contains(extension)) {
                socket.outputStream.write(file.readBytes())
            } else {
                println(file.text)
            }
        }
    }
}

def printDirectoryListing(dir) {
    printHeaders("text/html")
    println "<html><head></head><body>"
    for (file in dir.list().toList().sort()) {
        // special case for root document
        if ("/" == content) {
            content = ""
```

```
        }
        println "<li><a href='${content}/${file}'>${file}</a></li>"
    }
    println "</body></html>"
}

def printHeaders(mimeType) {
    println "HTTP/1.0 200 OK"
    println "Content-Type: ${mimeType}"
    println ""
}
    '''

    // Script is configurable via Java
    // system properties. Here we set
    // the property baseDir as the base
    // directory for serving static files.
    systemProperty 'baseDir', 'src/main/resources'
}
```

We can run the task `runHttpServer` from the command line and open the page http://localhost:8001/index in our web browser. If there is a file `index.html` in the directory `src/main/resources` it is shown in the browser.

```
$ gradle runGroovyScript
:runHttpServer
groovy is listening on port 8001
> Building 0% > :runHttpServer
```

Written with Gradle 2.11.

Original blog post written on February 10, 2016.

Running a Single Test

We can run test code with Gradle using the `test` task that is added by the Java plugin. By default all tests found in the project are executed. If we want to run a single test we can use the Java system property `test.single` with the name of the test. Actually the pattern for the system property is `_taskName_.single`. The `taskName` is the name of the task of type `Test` in our project. We will see how we can use this in our builds.

First we create a simple `build.gradle` file to run our tests:

```
// File: build.gradle
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    testCompile 'junit:junit:[4,)'
}

test {
    testLogging {
        // Show that tests are run in the command-line output
        events 'started', 'passed'
    }
}
```

Next we create two test classes with each a single test method, just to demonstrate we can invoke them as single test later on.

```
// File: src/test/java/com/mrhaki/gradle/SampleTest.java
package com.mrhaki.gradle;

import static org.junit.Assert.*;
import org.junit.*;

public class SampleTest {

    @Test public void sample() {
        assertEquals("Gradle is gr8", "Gradle is gr8");
    }

}



// File: src/test/java/com/mrhaki/gradle/AnotherSampleTest.java
package com.mrhaki.gradle;

import static org.junit.Assert.*;
import org.junit.*;

public class AnotherSampleTest {

    @Test public void anotherSample() {
        assertEquals("Gradle is great", "Gradle is great");
    }
}
```

To only run the `SampleTest` we must invoke the `test` task from the command-line with the Java system property `-Dtest.single=Sample`:

```
$ gradle -Dtest.single=Sample test
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava
:processTestResources UP-TO-DATE
:testClasses
:test

com.mrhaki.gradle.SampleTest > sample STARTED

com.mrhaki.gradle.SampleTest > sample PASSED

BUILD SUCCESSFUL

Total time: 11.404 secs
```

Notice only one test is executed now. Gradle will get the value *Sample* and uses it in the following pattern ***/<Java system property value=Sample>*.class* to find the test class. So we don't have to type the full package and class name of our single test class. To only invoke the `AnotherSampleTest` test class we run the `test` task with a different value for the Java systme property:

```
$ gradle -Dtest.single=AnotherSample test
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test

com.mrhaki.gradle.AnotherSampleTest > anotherSample STARTED

com.mrhaki.gradle.AnotherSampleTest > anotherSample PASSED

BUILD SUCCESSFUL

Total time: 5.62 secs
```

We can also use a pattern for the Java system property to run multiple tests that apply to the pattern. For example we can use **Sample* to run both `SampleTest` and `AnotherSampleTest`:

```
$ gradle -Dtest.single=*Sample test
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test

com.mrhaki.gradle.AnotherSampleTest > anotherSample STARTED

com.mrhaki.gradle.AnotherSampleTest > anotherSample PASSED

com.mrhaki.gradle.SampleTest > sample STARTED

com.mrhaki.gradle.SampleTest > sample PASSED

BUILD SUCCESSFUL

Total time: 5.605 secs
```

To show the Java system property also works for other tasks of type `Test` we add a new task to our `build.gradle` file. We name the task `sampleTest` and include our tests. We also apply the same `testLogging` now to all tasks with type `Test` so we can see the output.

```
// File: build.gradle
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    testCompile 'junit:junit:[4,)'
}

task sampleTest(type: Test, dependsOn: testClasses) {
    include '**/*Sample*'
}

tasks.withType(Test) {
    testLogging {
        events 'started', 'passed'
    }
}
```

Next we want to run only the `SampleTest` class, but now we use the Java system property `-DsampleTest.single=S*`:

```
$ gradle -DsampleTest.single=S* sampleTest
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:sampleTest

com.mrhaki.gradle.SampleTest > sample STARTED

com.mrhaki.gradle.SampleTest > sample PASSED

BUILD SUCCESSFUL

Total time: 10.677 secs
```

Code written with Gradle 1.6

Original blog post written on May 13, 2013.

Running Tests in Parallel

Once we apply the Java plugin we can run our tests with the `test` task. Normally each test is run sequentially, but we can also run tests in parallel. This speeds up the test task considerably especially with a lot of tests. We set the property `maxParallelForks` to a number greater than 1 to enable parallel tests.

We can also set the additional property `forkEvery` on the test task. With this property we define how many tests should run in a parallel test fork.

In the following build script we first create a lot of tests with the `createTests` task and then we can run them with the `test` task. We can pass the properties `maxParallelForks` and `forkEvery` to play around and see what happens. Of course we can also use hard coded values in our build script for the `test` task properties.

```
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    testCompile 'junit:junit:4.8.2'
}

test {
    if (project.hasProperty('maxParallelForks'))
        maxParallelForks = project.maxParallelForks as int
    if (project.hasProperty('forkEvery'))
        forkEvery = project.forkEvery as int
}
```

```
packages = 10
tests = 30

task createTests << {
    (0..packages).each { packageCounter ->
        def packageName = "gradletest${packageCounter}"
        (0..tests).each { classCounter ->
            def testClassName = "Gradle${classCounter}Test"
            copy {
                from 'src/templates'
                into 'src/test/java'
                expand([packageName: packageName, testClassName: testClassName])
                rename '(.*).java', packageName + '/' + testClassName + '.java'
                include 'SampleTest.java'
            }
        }
    }
}


// File: src/templates/SampleTest.java
package ${packageName};

import org.junit.Test;
import static org.junit.Assert.*;

public class ${testClassName} {

    @Test
    public void testString() throws Exception {
        Thread.sleep(200);
        assertEquals("mrhaki", "mr" + "haki");
    }
}
```

So first we create the tests: `$ gradle createTests`. And then we can experiment with different options:

```
$ gradle clean test
...
BUILD SUCCESSFUL

Total time: 1 mins 33.942 secs

$ gradle clean test -PmaxParallelForks=10
...
BUILD SUCCESSFUL

Total time: 36.68 secs

$ gradle clean test -PmaxParallelForks=4 -PforkEvery=25
...
BUILD SUCCESSFUL

Total time: 56.066 secs
```

Written with Gradle 0.9-rc-2.

Original blog post written on November 10, 2010.

Running All Tests From One Package

If we have a Gradle task of type `Test` we can use a filter on the command line when we invoke the task. We define a filter using the `--tests` option. If for example we want to run all tests from a single package we must define the package name as value for the `--tests` option. It is good to define the filter between quotes, so it is interpreted as is, without any shell interference.

If we configure the `test` task to output the test class name we can see that which tests are executed. In the following snippet we reconfigure the `test` task:

```
...
test {
    beforeTest { descriptor ->
       logger.lifecycle("Running test: $descriptor.className")
    }
}
...
```

Suppose we have a project and we execute all tests:

```
$ gradle test
...
:test
Running test: mrhaki.gradle.model.CourseSpec
Running test: mrhaki.gradle.model.TeacherSpec
Running test: mrhaki.gradle.service.CourseServiceSpec
...
$
```

To only run the tests in the `mrhaki.gradle.model` package we use the following command:

```
$ gradle test --tests "mrhaki.gradle.model.*"
...
:test
Running test: mrhaki.gradle.model.CourseSpec
Running test: mrhaki.gradle.model.TeacherSpec
...
$
```

We can even filter on a single test to be executed:

```
$ gradle test --tests "mrhaki.gradle.model.CourseSpec"
...
:test
Running test: mrhaki.gradle.model.CourseSpec
...
$
```

If we only want to run a single method of a test we can specify the method name as a filter:

```
$ gradle test --tests "mrhaki.gradle.model.CourseSpec.create new Course as immutable"
...
:test
Running test: mrhaki.gradle.model.CourseSpec
...
$
```

Written with Gradle 2.13.

Original blog post written on June 14, 2016.

Show Standard Out or Error Output from Tests

We use the `Test` task in Gradle to run tests. If we use the `System.out.println` or `System.err.println` methods in our test we don't see the output when we execute the tests. We can customize the test task to show any output send to standard out or error in the Gradle output.

First we show our test class written with Spock, but it could also be a JUnit or TestNG test:

```
// File: src/test/groovy/com/mrhaki/gradle/SampleSpec.groovy
package com.mrhaki.gradle

import spock.lang.*

class SampleSpec extends Specification {

    def "check that Gradle is Gr8"() {
        when:
        def value = 'Gradle is great!'

        then:
        // Include a println statement, so
        // we have output to show.
        println "Value = [$value]"
        value == 'Gradle is great!'
    }

}
```

Now we write a simple Gradle build file which can execute our test:

```
// File: build.gradle
apply plugin: 'groovy' // Adds test task

repositories.jcenter()

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.7'
    testCompile 'org.spockframework:spock-core:0.7-groovy-2.0'
}
```

Let's run the `test` task from the command line and look at the output:

```
$ gradle test
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava
:compileTestGroovy
:processTestResources UP-TO-DATE
:testClasses
:test

BUILD SUCCESSFUL

Total time: 7.022 secs
$
```

Well at least our test is successful, but we don't see the output of our `println` method invocation in the test. We customize the `test` task and add the `testLogging` method with a configuration closure. In the closure we set the property `showStandardStreams` to the value `true`. Alternatively we can set the `events` property or use the `events` method with the values `standard_out` and `standard_err` to achieve the same result. In the next build file we use the `showStandardStreams` property:

```
// File: build.gradle
apply plugin: 'groovy' // Adds test task

repositories.jcenter()

dependencies {
    compile 'org.codehaus.groovy:groovy-all:2.3.7'
    testCompile 'org.spockframework:spock-core:0.7-groovy-2.0'
}

test {
    testLogging {
        // Make sure output from
        // standard out or error is shown
        // in Gradle output.
        showStandardStreams = true

        // Or we use events method:
```

```
        // events 'standard_out', 'standard_error'

        // Or set property events:
        // events = ['standard_out', 'standard_error']

        // Instead of string values we can
        // use enum values:
        // events org.gradle.api.tasks.testing.logging.TestLogEvent.STANDARD_OUT,
        //        org.gradle.api.tasks.testing.logging.TestLogEvent.STANDARD_ERROR,
    }
}
```

We re-run the `test` task from the command line and look at the output to see the result from the `println` method:

```
$ gradle test
:compileJava UP-TO-DATE
:compileGroovy UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava
:compileTestGroovy
:processTestResources UP-TO-DATE
:testClasses
:test

com.mrhaki.gradle.SampleSpec > check that Gradle is Gr8 STANDARD_OUT
    Value = [Gradle is great!]

BUILD SUCCESSFUL

Total time: 8.716 secs
$
```

Written with Gradle 2.1.

Original blog post written on October 15, 2014.

Show More Information About Failed Tests

Running tests in Gradle is easy. Normally if one of the tests fails the build fails as well. But we don't see immediately in the command-line output why a test fails. We must first open the generated HTML test report. But there are other ways as well.

First we create the following sample Gradle build file:

```
// File: build.gradle
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    testCompile 'junit:junit:[4,)'
}
```

And we use the following sample JUnit test class. Notice this test will always fail, which is what we want in this case.

```
// File: src/test/java/com/mrhaki/gradle/SampleTest.java
package com.mrhaki.gradle;

import org.junit.*;

public class SampleTest {

    @Test public void sample() {
        Assert.assertEquals("Gradle is gr8", "Gradle is great");
    }

}
```

To run our test we execute the `test` task. If we run the task we see in the output on which line the test fails, but we don't see the assertion why it went wrong:

```
$ gradle test
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava
:processTestResources UP-TO-DATE
:testClasses
:test

com.mrhaki.gradle.SampleTest > sample FAILED
    org.junit.ComparisonFailure at SampleTest.java:8

1 test completed, 1 failed
:test FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':test'.
> There were failing tests. See the report at: file:///Users/mrhaki/Projects/mrhaki.com/blog/post\
s/samples/gradle/testlogging/build/reports/tests/index.html

* Try:
```

```
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get mor\
e log output.

BUILD FAILED

Total time: 4.904 secs
```

We can run the `test` task again, but now set the Gradle logging level to info with the command-line option `-i` or `--info`. Now we get the assertion about what went wrong in the output:

```
$ gradle test -i
Starting Build
Settings evaluated using empty settings script.
Projects loaded. Root project using build file
...
Successfully started process 'Gradle Worker 1'
Gradle Worker 1 executing tests.
Gradle Worker 1 finished executing tests.

com.mrhaki.gradle.SampleTest > sample FAILED
    org.junit.ComparisonFailure: expected:<gradle is gr[8]> but was:<gradle is gr[eat]>
        at org.junit.Assert.assertEquals(Assert.java:115)
        at org.junit.Assert.assertEquals(Assert.java:144)
        at com.mrhaki.gradle.SampleTest.sample(SampleTest.java:8)
Process 'Gradle Worker 1' finished with exit value 0 (state: SUCCEEDED)

1 test completed, 1 failed
Finished generating test XML results (0.025 secs)
Generating HTML test report...
Finished generating test html results (0.027 secs)
:test FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':test'.
> There were failing tests. See the report at: file:///Users/mrhaki/Projects/mrhaki.com/blog/post\
s/samples/gradle/testlogging/build/reports/tests/index.html

* Try:
Run with --stacktrace option to get the stack trace. Run with --debug option to get more log outp\
ut.

BUILD FAILED

Total time: 5.117 secs
```

But this still generates a lot of noise. It is better to customize the test logging by configuring the `test` task. We can configure the logging on different levels. To get the information about the failure we want we only have to change the `exceptionFormat` property and set the value to `full`. Our Gradle build file now looks like this:

```
// File: build.gradle
apply plugin: 'java'

repositories {
    mavenCentral()
}

dependencies {
    testCompile 'junit:junit:[4,)'
}

test {
    testLogging {
        exceptionFormat = 'full'
    }
}
```

We can re-run the `test` task and use the normal logging level, but this time we also get the reason why our test fails, without the extra noise:

```
$ gradle test
:compileJava UP-TO-DATE
:processResources UP-TO-DATE
:classes UP-TO-DATE
:compileTestJava UP-TO-DATE
:processTestResources UP-TO-DATE
:testClasses UP-TO-DATE
:test

com.mrhaki.gradle.SampleTest > sample FAILED
    org.junit.ComparisonFailure: expected:<gradle is gr[8]> but was:<gradle is gr[eat]>
        at org.junit.Assert.assertEquals(Assert.java:115)
        at org.junit.Assert.assertEquals(Assert.java:144)
        at com.mrhaki.gradle.SampleTest.sample(SampleTest.java:8)

1 test completed, 1 failed
:test FAILED

FAILURE: Build failed with an exception.

* What went wrong:
Execution failed for task ':test'.
> There were failing tests. See the report at: file:///Users/mrhaki/Projects/mrhaki.com/blog/post\
s/samples/gradle/testlogging/build/reports/tests/index.html

* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get mor\
e log output.

BUILD FAILED

Total time: 5.906 secs
```

Sample written with Gradle 1.6

Original blog post written on May 10, 2013.

Create JAR Artifact with Test Code for Java Project

Today, during my Gradle session, someone asked how to create a JAR file with the compiled test classes and test resources. I couldn't get the task syntax right at that moment, so when I was at home I had to find out how we can create that JAR file. And it turned out to be very simple:

```
apply plugin: 'java'

task testJar(type: Jar) {
    classifier = 'tests'
    from sourceSets.test.classes
}
```

The magic is in the `from` method where we use `sourceSets.test.classes`. Because we use `sourceSets.test.classes` Gradle knows the task `testClasses` needs to be executed first before the JAR file can be created. And of course the `assemble` task will pick up this new task of type Jar automatically.

When we run the build we get the following output:

```
$ gradle assemble
:compileJava
:processResources
:classes
:jar
:compileTestJava
:processTestResources
:testClasses
:testJar
:assemble
```

Written with Gradle 0.9-rc-2.

Original blog post written on November 03, 2010.

Add Filtering to ProcessResources Tasks

When we apply the Java plugin (or any dependent plugin like the Groovy plugin) we get new tasks in our project to copy resources from the source directory to the classes directory. So if we have a file `app.properties` in the directory `src/main/resources` we can run the task `$ gradle processResources` and the file is copied to `build/classes/main/app.properties`. But what if we want to apply for example some filtering while the file is copied? Or if we want to rename the file? How we can configure the `processResources` task?

The task itself is just an implementation of the `Copy` task. This means we can use all the configuration options from the `Copy` task. And that includes filtering and renaming the files. So we need to find all tasks in our project that copy resources and then add for example filtering to the configuration. The following build script shows how we can do this:

```
import org.apache.tools.ant.filters.*

apply plugin: 'java'

version = '1.0-DEVELOPMENT'

afterEvaluate {
    configure(allProcessResourcesTasks()) {
        filter(ReplaceTokens,
               tokens: [version: project.version,
                        gradleVersion: project.gradle.gradleVersion])
    }
}

def allProcessResourcesTasks() {
    sourceSets.all.processResourcesTaskName.collect {
        tasks[it]
    }
}
```

Let's create the following two files in our project directory:

```
# src/main/resources/app.properties
appversion=@version@


# src/test/resources/test.properties
gradleVersion=@gradleVersion@
```

We can now execute the build and look at the contents of the copied property files:

```
$ gradle build
:compileJava
:processResources
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

BUILD SUCCESSFUL

Total time: 4.905 secs

$ cat build/classes/main/app.properties
appversion=1.0-DEVELOPMENT
$ cat build/classes/test/test.properties
gradleVersion=0.9-rc-2
```

Written with Gradle 0.9-rc-2.

Original blog post written on November 05, 2010.

Use Groovy Ruleset File with Code Quality Plugin

The *codenarc* plugin supports CodeNarc for Groovy projects. The default configuration file is XML based with the name `codenarc.xml` and must be placed in the directory `config/codenarc`. But CodeNarc also supports a Groovy DSL for writing configuration files. Suppose we have a configuration file with the name `rules.groovy` and we put it in the directory `config/codenarc`. In our `build.gradle` file we reference this file with the property `configFile` inside a `codenarc` configuration block. The codenarc plugin will pass this value on to CodeNarc and the rules defined in our Groovy ruleset file are used.

```
// File: config/codenarc/rules.groovy

ruleset {
    description 'Rules Sample Groovy Gradle Project'

    ruleset('rulesets/basic.xml')
    ruleset('rulesets/braces.xml')
    ruleset('rulesets/exceptions.xml')
    ruleset('rulesets/imports.xml')
    ruleset('rulesets/logging.xml') {
        'Println' priority: 1
        'PrintStackTrace' priority: 1
    }
    ruleset('rulesets/naming.xml')
    ruleset('rulesets/unnecessary.xml')
    ruleset('rulesets/unused.xml')
}


// File: build.gradle
['groovy', 'codenarc'].each {
    apply plugin: it
}

repositories {
    mavenCentral()
}

dependencies {
    compile group: 'org.codehaus.groovy', name: 'groovy', version: '2.3.9'
}

codenarc {
    configFile = file('config/codenarc/rules.groovy')
}
```

Written with Gradle 0.9.1 and re-written with Gradle 2.2.1.

Original blog post written on January 10, 2011.

Don't Let CodeNarc Violations Fail the Build

With the codenarc plugin for Groovy project we use CodeNarc to check our code. By default we are not allowed to have any violations in our code, because if there is a violation

the Gradle build will stop with a failure. If we don't want to our build to fail, because of code violations, we can set the property `ignoreFailures` to `true` for the CodeNarc task.

The code-quality plugin adds two CodeNarc tasks to our project: codenarcMain and codenarcTest. We can simply set the property `ignoreFailures` for these tasks:

```
apply plugin: 'codenarc'

[codenarcMain, codenarcTest]*.ignoreFailures = true
```

We can also search for all tasks of type CodeNarc and set the `ignoreFailures` property. This is useful if we added new tasks of type CodeNarc to our project and want to change the property of all these tasks:

```
apply plugin: 'codenarc'

tasks.withType(CodeNarc).allTasks { codeNarcTask ->
    codeNarcTask.ignoreFailures = true
}
```

Written with Gradle 0.9.1 and re-written with Gradle 2.2.1.

Original blog post written on January 17, 2011.