Gradle Goodness

Experience Gradle through code snippets

Hubert A. Klein Ikkink

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 2024-03-23

Also By Hubert A. Klein Ikkink (mrhaki)

DataWeave Delight Notebook

Clojure Goodness Notebook

Ratpacked Notebook

Awesome Asciidoctor Notebook

Spocklight Notebook

Grails Goodness Notebook

Groovy Goodness Notebook

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

Contents

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.

Original post written on November 9, 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 post written on June 18, 2012

Enabling Preview Features For Java

Java introduced preview features in the language since Java 12. This features can be tried out by developers, but are still subject to change and can even be removed in a next release. By default the preview features are not enabled when we want to compile and run our Java code. We must explicitly specify that we want to use the preview feature to the Java compiler and Java runtime using the command-line argument `--enable-preview`. In Gradle we can customize our build file to enable preview features. We must customize tasks of type `JavaCompile` and pass `--enable-preview` to the compiler arguments. Also tasks of type `Test` and `JavaExec` must be customized where we need to add the JVM argument `--enable-preview`.

In the following Gradle build script written in Kotlin we have a Java project written with Java 15 where we reconfigure the tasks to enable preview features:

```
plugins {
    java
    application
}

repositories {
    mavenCentral()
}

dependencies {
    testImplementation("org.junit.jupiter:junit-jupiter-api:5.7.1")
    testRuntimeOnly("org.junit.jupiter:junit-jupiter-engine:5.7.1")
}

application {
    mainClass.set("mrhaki.Patterns")
}

tasks {
    val ENABLE_PREVIEW = "--enable-preview"

    // In our project we have the tasks compileJava and
    // compileTestJava that need to have the
    // --enable-preview compiler arguments.
    withType<JavaCompile>() {
        options.compilerArgs.add(ENABLE_PREVIEW)
```

```
        // Optionally we can show which preview feature we use.
        options.compilerArgs.add("-Xlint:preview")

        // Explicitly setting compiler option --release
        // is needed when we wouldn't set the
        // sourceCompatiblity and targetCompatibility
        // properties of the Java plugin extension.
        options.release.set(15)
    }

    // Test tasks need to have the JVM argument --enable-preview.
    withType<Test>() {
        useJUnitPlatform()
        jvmArgs.add(ENABLE_PREVIEW)
    }

    // JavaExec tasks need to have the JVM argument --enable-preview.
    withType<JavaExec>() {
        jvmArgs.add(ENABLE_PREVIEW)
    }
}
```

Written with Gradle 6.8.3

Original post written on March 5, 2021

Using Maven Toolchains Configuration For Gradle Java Toolchain Resolution

When we apply the Java plugin to our Gradle project we can configure which Java version we want to use for compiling our source code and running our tests using a toolchain configuration. The benefit of having a toolchain configuration is that we can use a different Java version for compiling and running our code than the Java version that is used by Gradle to execute the build. Gradle will look for that Java version on our local computer or download the correct version if it is not available. To search for a local Java installation Gradle will look for operating system specific locations, installations by package managers like SKDMAN! and Jabba, IntelliJ IDEA installations and Maven Toolchain specifications. Maven Toolchain specifications is an XML file describing the location of local Java installation. Each Java installation is described by a version and optional vendor it provides and the location of the installation. Maven uses this information to find the correct Java installation when the `maven-toolchain-plugin` is used in a Maven project. But Gradle can also utilize Maven Toolchain specifications to find local Java installations. This can be useful when we have to work on multiple projects where some use Maven and others use Gradle. We can place the Maven Toolchain specification file in our Maven home directory. This is also the default place where Gradle will look, but we can use a project property to override this location.

The following example shows a Maven toolchain configuration with three different Java versions:

```
<?xml version="1.0" encoding="UTF-8"?>
<toolchains>
    <toolchain>
        <type>jdk</type>
        <provides>
            <version>11</version>
            <vendor>Azul Zulu</vendor>
        </provides>
        <configuration>
            <jdkHome>C:/Users/mrhaki/tools/apps/zulu11-jdk/current</jdkHome>
        </configuration>
    </toolchain>
    <toolchain>
        <type>jdk</type>
        <provides>
            <version>17</version>
            <vendor>Azul Zulu</vendor>
        </provides>
        <configuration>
            <jdkHome>C:/Users/mrhaki/apps/zulu17-jdk/current</jdkHome>
        </configuration>
    </toolchain>
    <toolchain>
        <type>jdk</type>
        <provides>
            <version>21</version>
            <vendor>Azul Zulu</vendor>
        </provides>
        <configuration>
            <jdkHome>C:/Users/mrhaki/tools/apps/zulu-jdk/current</jdkHome>
        </configuration>
    </toolchain>
</toolchains>
```

In our Gradle build file we apply the java plugin and define in the toolchain configuration we want to use Java 17 for our builds:

```
// File: build.gradle.kts
plugins {
    java
}

java {
    toolchain {
        // Use Java 17 for building and running tests
        languageVersion = JavaLanguageVersion.of(17)
    }
}
```

We can now use the `javaToolchains` task to see the available Java installations:

```
$ ./gradlew javaToolchains

> Task :javaToolchains

 + Options
     | Auto-detection:     Enabled
     | Auto-download:      Enabled

  + Azul Zulu JDK 11.0.22+7-LTS
     | Location:           C:\Users\mrhaki\tools\apps\zulu11-jdk\current
     | Language Version:   11
     | Vendor:             Azul Zulu
     | Architecture:       amd64
     | Is JDK:             true
     | Detected by:        Maven Toolchains

 + Azul Zulu JDK 17.0.10+7-LTS
     | Location:           C:\Users\mrhaki\tools\apps\zulu17-jdk\current
     | Language Version:   17
     | Vendor:             Azul Zulu
     | Architecture:       amd64
     | Is JDK:             true
     | Detected by:        Current JVM

 + Azul Zulu JDK 21.0.2+13-LTS
     | Location:           C:\Users\mrhaki\tools\apps\zulu-jdk\current
     | Language Version:   21
     | Vendor:             Azul Zulu
     | Architecture:       amd64
     | Is JDK:             true
     | Detected by:        Maven Toolchains

BUILD SUCCESSFUL in 4s
1 actionable task: 1 executed
```

The command was run using Java 17 and we can see in the output that it is detected by Gradle as the current JVM. The Java installations for Java 11 and Java 21 are detected using Maven Toolchains.

If the location of the Maven Toolchain specification file is not in the default location, we can use the Gradle project property `org.gradle.java.installations.maven-toolchain-file` to specify a custom location. We can use it from the command line using the `-P` option or we can add it to `gradle.properties` in the project root directory.

```
$ ./gradlew javaToolchains -Porg.gradle.java.installations.maven-toolchain-file=C:/Users/mrhaki/tools/\
maven/toolchains.xml

...
```

Written with Gradle 8.5.

Original post written on February 2, 2024

Java Toolchain Configuration Using User Defined Java Locations

With the java plugin we can configure a so-called Java toolchain. The toolchain configuration is used to define which Java version needs to be used to compile and test our code in our project. The location of the Java version can be determined by Gradle automatically. Gradle will look at known locations based on the operating system, package managers, IntellIJ IDEA installations and Maven Toolchain configuration.

But we can also define the locations of our Java installations ourselves using the project property `org.gradle.java.installations.paths`. We provide the paths to the local Java installations as a comma separated list as value for this property. When we set this property we can also disable the Gradle toolchain detection mechanism, so only the Java installations we have defined ourselves are used. To disable the automatic detection we set the property `org.gradle.java.installations.auto-detect` to `false`. If we leave the value to the default value `true`, then the locations we set via `org.gradle.java.installations.paths` are added to the Java installations already found by Gradle.

The property `org.gradle.java.installations.paths` is a project property we can set via the command line, but we can also set it in the `gradle.properties` file in our `GRADLE_USER_HOME` directory. Then the values we define will be used by all Gradle builds on our machine.

In the following example `gradle.properties` file we define the locations of two Java installations and also disable the automatic detection of Java installations. We store this file in our `GRADLE_USER_HOME` directory.

```
# File: $GRADLE_USER_HOME/gradle.properties
# We define the locations of two Java installations on our computer.
org.gradle.java.installations.paths=C:/Users/mrhaki/tools/apps/zulu11-jdk/current,C:/Users/mrhaki/tool\
s/apps/zulu17-jdk/current

# We disable the automatic detection of Java installations by Gradle.
org.gradle.java.installations.auto-detect=false

# We also disable the automatic download of Java installations by Gradle.
org.gradle.java.installations.auto-download=false
```

We add the java plugin and configure our toolchain with the following Gradle build script:

```
// File: build.gradle.kts
plugins {
    java
}

java {
    toolchain {
        // We want to use Java 17 to compile, test and run our code.
        // Now it doesn't matter which Java version is used by Gradle itself.
        languageVersion = languageVersion.set(JavaLanguageVersion.of(17))
    }
}
```

We run the `javaToolchains` task to see the Java toolchain configuration:

```
$ ./gradlew javaToolchains

> Task :javaToolchains

 + Options
     | Auto-detection:     Disabled
     | Auto-download:      Disabled

 + Azul Zulu JDK 11.0.22+7-LTS
     | Location:           C:\Users\mrhaki\tools\apps\zulu11-jdk\current
     | Language Version:   11
     | Vendor:             Azul Zulu
     | Architecture:       amd64
     | Is JDK:             true
     | Detected by:        Gradle property 'org.gradle.java.installations.paths'

 + Azul Zulu JDK 17.0.10+7-LTS
     | Location:           C:\Users\mrhaki\tools\apps\zulu17-jdk\current
     | Language Version:   17
     | Vendor:             Azul Zulu
     | Architecture:       amd64
     | Is JDK:             true
     | Detected by:        Gradle property 'org.gradle.java.installations.paths'


BUILD SUCCESSFUL in 1s
1 actionable task: 1 executed
```

In the generated output we can see that Gradle detected the two Java installations we defined in the `gradle.properties` file using the Gradle property `org.gradle.java.installations.paths`.

Written with Gradle 8.5.

Original post written on February 3, 2024

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:

```
usePlugin '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 -t` and we get the following list of tasks:

```
------------------------------------------------------------
Root Project
------------------------------------------------------------
:assemble - Builds all Jar, War, Zip, and Tar archives.
   -> :jar
:build - Assembles and tests this project.
   -> :assemble, :check
:buildDependents - Assembles and tests this project and all projects that depend on it.
   -> :build
:buildNeeded - Assembles and tests this project and all projects it depends on.
   -> :build
:check - Runs all checks.
   -> :test
:classes - Assembles the main classes.
   -> :compileGroovy, :compileJava, :processResources
:clean - Deletes the build directory.
:compileGroovy - Compiles the main Groovy source.
   -> :compileJava
:compileJava - Compiles the main Java source.
:compileTestGroovy - Compiles the test Groovy source.
   -> :classes, :compileTestJava
:compileTestJava - Compiles the test Java source.
   -> :classes
:groovydoc - Generates the groovydoc for the source code.
:jar - Generates a jar archive with all the compiled classes.
   -> :classes
:javadoc - Generates the javadoc for the source code.
   -> :classes
:processResources - Processes the main resources.
:processTestResources - Processes the test resources.
:test - Runs the unit tests.
   -> :classes, :testClasses
:testClasses - Assembles the test classes.
```

```
   -> :compileTestGroovy, :compileTestJava, :processTestResources
rule - Pattern: build<ConfigurationName>: Builds the artifacts belonging to the configuration.
rule - Pattern: upload<ConfigurationName>Internal: Uploads the project artifacts of a configuration to\
 the internal Gradle repository.
rule - Pattern: upload<ConfigurationName>: Uploads the project artifacts of a configuration to a publi\
c Gradle repository.
```

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.
        convention.sourceSets.all."${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 {
    mavenCentral()  // Define Maven central repository to look for dependencies.
}

dependencies {
    groovy 'org.codehaus.groovy:groovy:1.6.5'  // group:name:version is a nice shortcut notation for d\
ependencies.
}
```

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));
    }
}
```

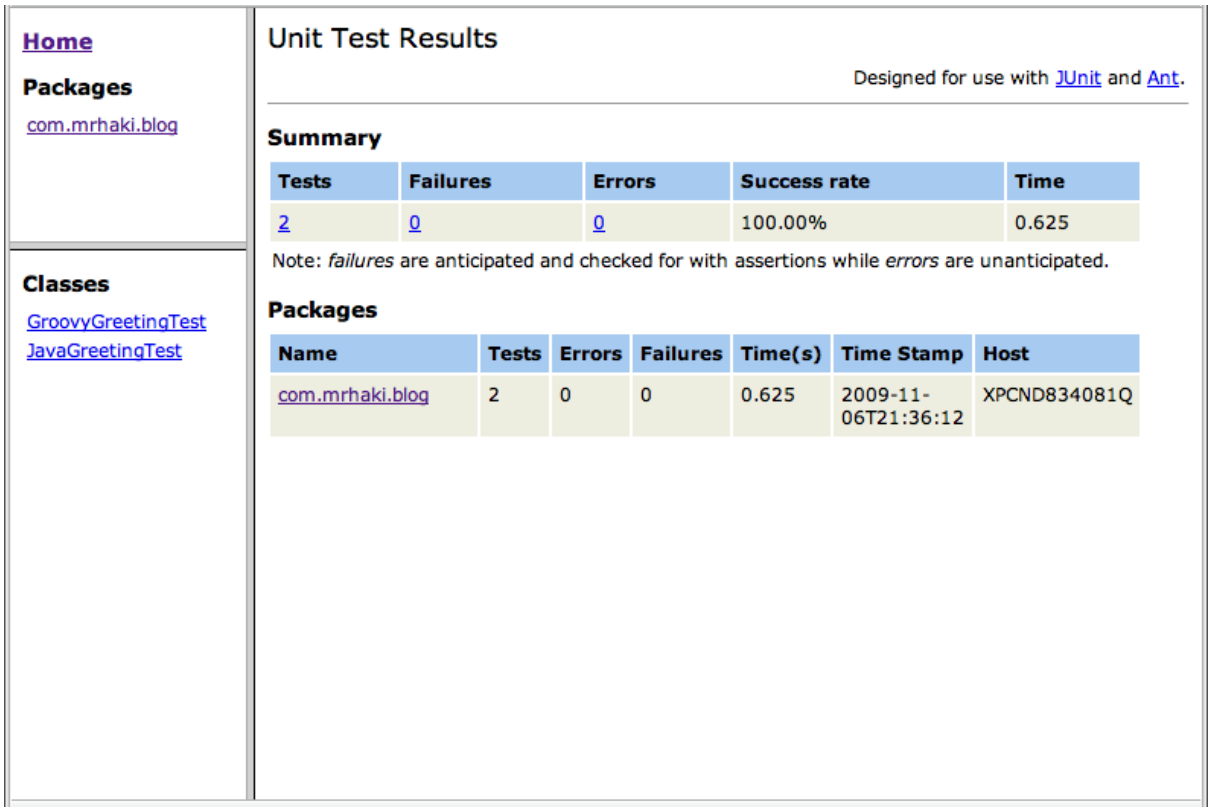
And we create a Groovy test class in `src/test/groovy/com/mrhaki/blog/GroovyGreetingTest.groovy`:

```
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:1.6.5'  // group:name:version is a nice shortcut notation for d\
ependencies.
    testCompile 'junit:junit:4.7'
}
```

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:

```
version = "1.0-${new Date().format('yyyyMMdd')"  // The script is all Groovy, so we make use of all me\
thods and features.
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-1.0-20091107.jar`. Here is the complete `build.gradle` with all changes we made:

```
usePlugin 'groovy'

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

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

dependencies {
    groovy 'org.codehaus.groovy:groovy:1.6.5'  // group:name:version is a nice shortcut notation for d\
ependencies.
```

```
    testCompile 'junit:junit:4.7'
}

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.
        convention.sourceSets.all."${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.

Written with Gradle 0.8.

Original post written on November 7, 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:

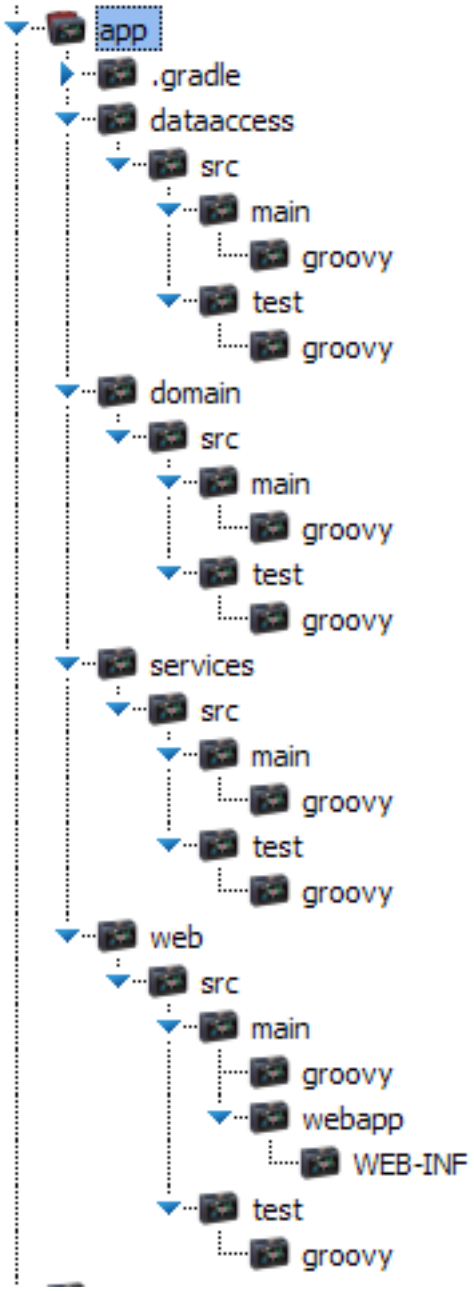
```
subprojects {
    usePlugin 'groovy'
    version = '1.0.0-SNAPSHOT'
    group = 'com.mrhaki.blog'
    configurations.compile.transitive = true  // Make sure transitive project dependencies are resolve\
d.

    repositories {
        mavenCentral()
    }

    dependencies {
        groovy 'org.codehaus.groovy:groovy:1.6.5'
    }

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

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

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

project(':web') {
    usePlugin 'jetty'  // jetty plugin extends war plugin, so we get all war plugin functionality as w\
ell.

    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'."
    }
}
```

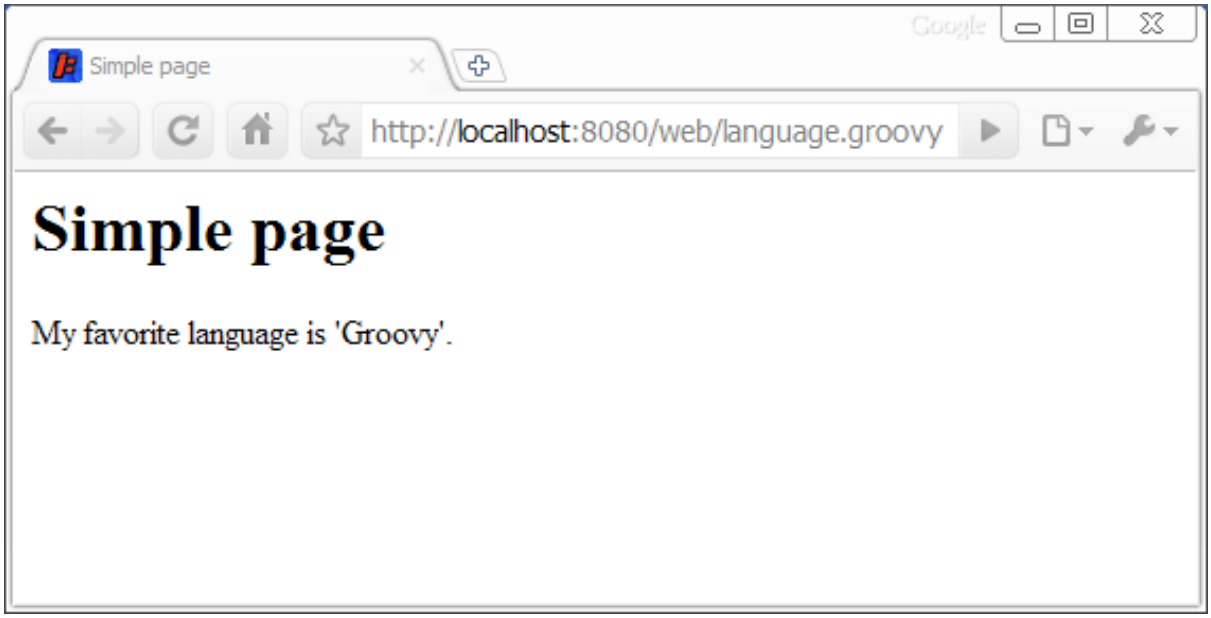
```
<?xml version="1.0" encoding="UTF-8"?>
<!-- File: app/web/src/main/webapp/WEB-INF/web.xml -->
<web-app>
    <servlet>
        <servlet-name>Groovy</servlet-name>
        <servlet-class>groovy.servlet.GroovyServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>Groovy</servlet-name>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

Written with Gradle 0.8.

Original post written on November 12, 2009

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.

Original 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 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 post written on September 21, 2015

Add Support For ”Scratch” Files To Java Project

When working on a Java project, we might want to have a place where we can just play around with the code we write. We need a ”scratch” file where we can access the Java classes we write in our main sourceset. The scratch file is actually a Java source file with a `main` method where we can create instances of the Java code we write and invoke methods on them. This gives back a fast feedback loop, and we can use it to play around with our Java classes without the need to write a test for it. It gives great flexiblity during development. We must make sure the scratch file will not be packed in the JAR file with our production code.

To support this in our Gradle build file we can add a new sourceset that can access all classes we write in the main sourceset. Also we want to have new configurations for this sourceset so we can add dependencies that are only used by our scratch file. And finally we want a new task to run our scratch file. By default our scratch file will not be part of the JAR file with the classes from the main sourceset.

In the following example build script we first define the common configuration for a Java project with a dependency on the Log4j2 library. Notice we use the toolchain feature of Gradle to use Java 15 to compile and run our Java code. Using the toolchain definition Gradle will look for a Java 15 JDK on our computer and if it cannot find one can even download it automatically.

Next we define a new sourceset `dev` so we can create a `Scratch.java` file in the directory `src/dev/java` and we define the compile and runtime classpath to be dependent on the main source set output. As a bonus we also can use the `src/dev/resources` directory for resource files we want to have in the classpath when we run our `Scratch.java` file.

If we want to define dependencies that are only used by our `Scratch` class file we must add extra configurations: `devImplementation` and `devRuntimeOnly`. These configurations extend from the `implementation` and `runtimeOnly` configurations added by the `java-library` plugin. So all dependencies needed by classes in the main sourceset will also be available in the configurations for the `dev` sourceset.

Finally, we add a new task `runDev` that executes the `main` method in the `Scratch.java` file in the `src/dev/java` directory.

```
// File: build.gradle.kts
plugins {
    `java-library`
}

repositories {
    mavenCentral()
}

dependencies {
    implementation(platform("org.apache.logging.log4j:log4j-bom:2.14.0"))
    implementation("org.apache.logging.log4j:log4j-api")
    implementation("org.apache.logging.log4j:log4j-core")
}

java {
    toolchain {
        languageVersion.set(JavaLanguageVersion.of(15))
    }
}

//-----------------------------------------------------------------------------
// Configure "dev" sourceset for running Scratch class
//-----------------------------------------------------------------------------

// Create new dev sourceset with a compile and runtime classpath dependency
// on the main sourceset. This allows us to use the classes we create in
// the main sourceset in our dev sourceset.
// The directories src/dev/java and src/dev/resources are recognized
// this sourceset.
val dev: SourceSet by sourceSets.creating {
    compileClasspath += sourceSets.main.get().output
    runtimeClasspath += sourceSets.main.get().output
}

// Create implementation and runtimeOnly configurations for the dev sourceset.
// These configurations can be used to define dependencies that only
// apply for the source files in the dev sourceset.
val devImplementation: Configuration by configurations.getting {
    extendsFrom(configurations.implementation.get())
}
val devRuntimeOnly: Configuration by configurations.getting {
    extendsFrom(configurations.runtimeOnly.get())
}

// Create a new task "runDev" that will run the compiled Scratch.java file
// in the root of src/dev/java. The classpath will contains all dependencies
// from the devImplementation and devRuntimeOnly configurations.
val runDev by tasks.registering(JavaExec::class) {
    description = "Run Scratch file."
    group = "dev"
    classpath = dev.runtimeClasspath
    mainClass.set("Scratch")
}

dependencies {
```

```
    // Here we add an extra dependency only for the dev sourceset.
    devImplementation("org.apache.commons:commons-lang3:3.12.0")
}
```

Now we have our build file with scratch file support so it is time to have some sample code.

First we create a simple Java file in our main sourceset together with a Log4j2 configuration properties file:

```
// File: src/main/java/mrhaki/Sample.java
package mrhaki;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Sample {
    private static Logger log = LogManager.getFormatterLogger(Sample.class);

    public String sayHello(String name) {
        log.info("sayHello(name=%s)", name);
        return "Hello %s".formatted(name);
    }
}
```

```
# File: src/main/resource/log4j2.properties
appender.console.type=Console
appender.console.name=STDOUT
appender.console.layout.type = PatternLayout
appender.console.layout.pattern = %m%n

rootLogger.level=ERROR
rootLogger.appenderRef.stdout.ref=STDOUT
```

To play around with our `Sample` class we add a scratch file and also an extra Log4j2 configuration properties file to change the configuration when we run our scratch file:

```
// File: src/dev/java/Scratch.java
import mrhaki.Sample;
import org.apache.commons.lang3.SystemUtils;
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class Scratch {
    private static Logger log = LogManager.getFormatterLogger(Scratch.class);

    public static void main(String[] args) {
        log.info("Running dev with Java %s.", SystemUtils.JAVA_VERSION);
        Sample sample = new Sample();
        sample.sayHello("mrhaki");
    }
}
```

```
# File: src/dev/resources/log4j2.properties
rootLogger.level=DEBUG
```

To execute our scratch file we invoke the `runDev` task from the command-line:

```
$ gw runDev

> Task :runDev
Running dev with Java 15.0.2.
sayHello(name=mrhaki)

BUILD SUCCESSFUL in 1s
5 actionable tasks: 5 executed
```

Written with Gradle 6.8.3.

Original post written on March 10, 2021

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 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 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 post written on February 10, 2016

Generate Javadoc In HTML5

Since Java 9 we can specify that the Javadoc output must be generated in HTML 5 instead of the default HTML 4. We need to pass the option `-html5` to the `javadoc` tool. To do this in Gradle we must add the option to the `javadoc` task configuration. We use the `addBooleanOption` method of the `options` property that is part of the `javadoc` task. We set the argument to `html5` and the value to `true`.

In the following example we reconfigure the `javadoc` task to make sure the generated Javadoc output is in HTML 5:

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

javadoc {
    options.addBooleanOption('html5', true)
}
```

The boolean option we added to the `options` property is not part of the Gradle check to see if a task is up to date. So if we would change the key `html5` to `html4`, because we want to get documentation in HTML 4, the task would be seen as up to date, because Gradle doesn't keep track of the change. We can change this by adding a property to the task `inputs` property, that contains the output format. Let's also add a new extension to `Javadoc` tasks to define our own DSL to set the output format.

We need to create an extension class and plugin to apply the extension to the `Javadoc` tasks. In the plugin we can also add support to help Gradle check to see if the task is up to date, based on the output format. In the following example we define an extension and plugin in our build file, but we could also place the classes in the `buildSrc` directory of our project.

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

javadoc {
    // New DSL to configure the task
    // added by the JavadocPlugin.
    output {
        html5 = true
    }
}

/**
 * Plugin to add the {@link JavadocOutputOptions} extension
 * to the Javadoc tasks.
 * <p>
 * Also make sure Gradle can check if the task needs
 * to rerun when the output format changes.
 */
class JavadocPlugin implements Plugin<Project&g;t {

    void apply(Project project) {
        project.tasks.withType(Javadoc) { Javadoc task ->
            // Create new extension for Javadoc task with the name "output".
            // Users can set output format to HTML 5 as:
            // javadoc {
            //     output {
            //         html5 = true
            //     }
            // }
            // or as HTML4:
            // javadoc {
            //     output {
            //         html4 = true
            //     }
            // }
            JavadocOutputOptions outputOptions =
                    task.extensions.create("output", JavadocOutputOptions)

            // After project evaluation we know what the
            // user has defined as output format using the
            // "output" configuration block.
            project.afterEvaluate {
                // We need to make sure the up-to-date check
                // is triggered when the output option changes.
                // If the value is not changed the task is up-to-date.
                task.inputs.property("output.html5", outputOptions.html5)

                // We add the boolean option html4 and html5
                // based on the user's value set via the
                // JavadocOutputOptions.
                task.options.addBooleanOption("html4", outputOptions.html4)
                task.options.addBooleanOption("html5", outputOptions.html5)
            }

        }
```

```
    }

}

/**
 * Extension for Javadoc tasks to define
 * if the output format must be HTML 4 or HTML 5.
 */
class JavadocOutputOptions {
    Boolean html4 = true
    Boolean html5 = !html4

    void setHtml4(boolean useHtml4) {
        html4 = useHtml4
        html5 = !html4
    }

    void setHtml5(boolean useHtml5) {
        html5 = useHtml5
        html4 = !html5
    }
}
```

Written with Gradle 4.10.2.

Original post written on November 14, 2018

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.

Original post written on November 3, 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.

Original post written on November 5, 2010

Use Groovy Ruleset File with Code Quality Plugin

The code-quality 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 `codeNarcConfigFileName`. The code-quality 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', 'code-quality'].each {
    apply plugin: it
}

repositories {
    mavenCentral()
}

dependencies {
    groovy group: 'org.codehaus.groovy', name: 'groovy', version: '1.7.6'
}

codeNarcConfigFileName = 'config/codenarc/rules.groovy'
```

Original post written on January 10, 2011

Don't Let CodeNarc Violations Fail the Build

With the code-quality 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: 'code-quality'

[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: 'code-quality'

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

Original post written on January 17, 2011