Instalación

Gradle es una aplicación portable (sin instalación). Basta con bajarse la aplicación y descomprimir. Luego para mayor facilidad reflejar el .exe en el path del sistema (si estamos en windows).

Para comprobar la instalación podemos hacer un

gradle –v

gradle wrapper

gradle tiene un wrapper que permite que los desarrolladores que comienzan a trabajar en un proyecto, no tienen por qué tener instalada la versión de gradle necesaria. Dicha versión se instalará automáticamente, para ello será necesario incluir los ficheros gradle en el sistema de control de versiones.

Para que pueda instalarse la versión adecuada de gradle, un desarrollador del equipo, tendrá que incluir las siguientes líneas en el fichero build.grade:

task createWrapper(type: Wrapper) {

gradleVersion = '1.0-milestone-4'

}

Cuando compartas tu proyecto solo será necesario que en lugar de usar el comando gradle, usen gradlew, si no tienen la versión de Gradle, el Wrapper lo descargará de internet y lo instalara (solo descomprime un ZIP en tu directorio home) para que sea usado en tu proyecto.

**Ficheros a excluir en el control de versiones (git):**

build/

.gradle/

tarjet/

.project

.settings/

Automatización de proyectos web

Para construir un proyecto web, en el build.gradle necesitamos incorporar el plugin war. De esta manera ya no necesitamos el plugin java, esto se debe a que el plugin war deriva del plugin java. Gradle asume que la parte client está en la carpeta src/java/webapp, tal y como lo hace maven.

Un plugin interesante, es jetty, incorporando este plugin y ejecutando gradlew jettyRun, se levantará la aplicación web. Para parar jetty, podemos lanzar la tarea jettyStop desde otro terminal.

El plugin jetty no estará en la versión gradle 4.0, ver la opción de utilizar el plugin gretty.

El objeto Project de una construcción (build.gradle

Podemos destacar los siguientes métodos en el objeto Project:

project.apply(plugin:’java’)

project.repositories({...})

project.dependencies({...})

Los métodos repositories y dependencies tienen como parámetros un closure.

Para ver si un proyecto tiene definida una propiedad podemos utilizar “boolean hasProperty(String propertyName)”.

Las propiedades definidas por usuario son las denominadas propiedades extra. Se definen:

ext.mip = 10

podemos hacer referencia a ella de las siguientes maneras:

print Project.ext.mip

print project.mip

print project.propertie(mip)

Dependencias

Tipos de dependencias:

1. compile: Este tipo lo añade el plugin java. Se requiere para la compilación de fuentes y en el caso de establecer plugin war, las compilaciones (clases) se copiarán al path WEB-INF/lib. Dentro de este tipo son populares los groups de spring o hibernte.
2. runtime: Este tipo lo añade el plugin java. No son requeridos para compilar pero si necesitadas en run time por las clases que sí son compiladas durante el proceso de construcción. Por tanto son clases que no necesitan estar en el classpath de compilación. Dentro de este tipo de dependencia están los drivers JDBC, por ejemplo mysql:msql-connector-java:5.1.37.
3. tescompile: Este tipo lo añade el plugin java. Son necesarias para compilar lo tests y no son requeridas por main. Si se establece el plugin war, las clases no serán incluidas en el war. Son groups de este tipo paquetes como JUnit, TestNG o Mokito.
4. testruntime: Este tipo es soportado por la plugin java. Es necesario en el run-time de las compilaciones para test.
5. providedCompile: Este tipo es soportado por el plugin war. Son necesarias para compilar pero no se incluyen en el war porque sabemos que ya están disponibles por el servidor web.
6. providedRuntime: Tipo soportado por el plugin war. Es igual que el tipo runtime, siendo la diferencia que no se incorporan estas clases en el jar.

Tareas

**La tarea B depende de la tarea A:**

task tB (DependOn: ‘tA’) {

doLast {

println ‘Soy A’

}

task tA {

doLast {

println ‘Soy B’

}

**tareas dinámicas:**

4.times { counter ->

task "task$counter" {

doLast {

println "I'm task number $counter"

}

}

}

Métodos para crear objetos task son:

Task task(String name)

Task task(String name, Closure configureClosure)

Task task(Map<String, ?> args, String name)

Task task(Map<String, ?> args, String name, Closure configureClosure)

Podemos importar una tarea en el build con:

apply from: 'task.gradle', siendo task.gradle el fichero que contiene la tarea.

Manipulación de tareas ya definidas:

Dependencias:

En el ejemplo anterior podemos modificar task0, haciéndola dependiente de la tarea 2 y 3: task0.dependsOn task2, task3. Así, si ejecutamos la tarea0, nos aparecerá en pantalla:

I´m task number 2

I´m task number 3

I´m task number 0

Comportamiento:

task hello {

doLast {

println 'Hello Earth'

}

}

hello.doFirst {

println 'Hello Venus'

}

hello.doLast {

println 'Hello Mars'

}

hello {

doLast {

println 'Hello Jupiter'

}

}

*> gradle -q hello*

Hello Venus

Hello Earth

Hello Mars

Hello Jupiter

task saludo {

print ' que tal '

}

saludo.doLast {

println ' como estas ' + project.properties['nombre'] + '?'

}

saludo.doFirst {

print 'Buenos dias,'

}

saludo.onlyIf{

project.properties['nombre']

Si ejecutamos gradle saludo, utiliza la propiedad nombre con el valor definido en el fichero gradle.properties. También podemos añadir y establecer una propiedad al proyecto desde la línea de comando con gradle –Pnombre=pepe, el resultado sería:

D:\TestGradle\proyectos\Test1>gradle -Pnombre=pepe saludo

que tal :saludo

Buenos dias, como estas pepe?

BUILD SUCCESSFUL

Total time: 1.489 secs

Secuenciación de tareas:

**task** taskTest1 {

println "tastTest1 en cuerpo"

doLast {println "tastTest1 en doLast" }

doFirst {println "tastTest1 en doFirst" }

}

**task** taskTest2 (dependsOn: 'taskTest1') {

println "taskTest2 en cuerpo"

doLast {println "taskTest2 en doLast" }

doFirst {println "taskTest2 en doFirst" }

}

**task** taskTest3 {

println "taskTest3 en cuerpo"

doLast {println "taskTest3 en doLast" }

doFirst {println "taskTest3 en doFirst" }

}

taskTest1.finalizedBy taskTest3

con la ejecución gradlew taskTest2, vemos:

D:\TestGradle\proyectos\smallNotes>gradlew taskTest2

tastTest1 en cuerpo

taskTest2 en cuerpo

taskTest3 en cuerpo

:taskTest1

tastTest1 en doFirst

tastTest1 en doLast

:taskTest3

taskTest3 en doFirst

taskTest3 en doLast

:taskTest2

taskTest2 en doFirst

taskTest2 en doLast

Múltiples proyectos

En el fichero settings.gradle, que debe incluirse en la raíz del proyecto, se especifican los proyectos tratados en la construcción. Gradle crea un objeto setting a partir de éste.

include ':repository', ':services', ':web-app'

**multiModulo**

**├── repository**

**│ └── ...**

**├── services**

**│ └── ...**

**├── web-app**

**│ └── ...**

**├── build.gradle**

**└── settings.gradle**

Gradle permite controlar todos los proyectos desde el fichero build.gradle, también admite que cada proyecto tenga su propio build.gradle. También permite que el proyecto root tenga su build y que cada uno de los subproyectos tenga en suyo.

Las tareas definidas en cada subproyecto no son visibles a los demás. Para hacer que una tarea que visibles a todos, haremos:

allprojects {

task whoami {

doLast {

println "I am ${project.name}"

}

}

}

para ejecutar una tarea de un módulo exclusivamente:

gradle :repository:whoami.

Comentar que es distinto ejecutar desde la raíz del proyecto gradle whoami a gradle :whoami. En el primer caso se ejecutan la mencionada tarea desde todos los módulos y con la segunda sólo la tarea del módulo:

D:\TestGradle\proyectos\multiModulo>gradle -q :whoami

I am multiModulo

D:\TestGradle\proyectos\multiModulo>gradle -q whoami

I am multiModulo

I am repository

I am services

I am web-app

Para añadir tareas a los subproyectos:

subprojects {

apply plugin : ‘java’

}

El proyecto raíz no contendrá la tarea ‘java’ es únicamente para las subtareas.

Dependencias entre subproyectos

Podríamos establecer que el proyecto services dependa del proyecto repository, para definir esta dependencia utilizamos el método project en el build.gradle del módulo raíz:

project(':services') {

dependencies {

compile project(':repository')

}

}

De forma similar, en el build.gradle del proyecto web-app, podemos especificar que este proyecto (web-app) depende del proyecto o módulo de servicios:

dependencies {

compile Project (‘:services’)

}

Migración de proyectos maven a gradle

La convención de maven sobre la ubicación de los fuentes coinciden con la de grade, lo cual facilita bastante el trabajo de conversión. src/main/java, src/test/java

Los plugins en maven se definen de la siguiente manera:

<plugin>

<artifactId>pluginName</artifactId>

<version>2.3.2</version>

</plugin>

y en gradle:

apply plugin: '<plugin name>'

Repositorios:

En maven:

<repositories>

<repository>

<id>repository\_1</id>

<name>custom Name</name>

<url> http://companylocalrepository.org </url>

</repository>

</repositories>

In Gradle, you can mention the repository using the following syntax:

repositories {

maven {

url <http://companylocalrepository.org>

}

}

Con gradle init, podemos transformar el proyecto a gradle desde maven.

Un proyecto que contine la carpeta src y el pom.xml, quedará así después del gradle init:

drwxr-xr-x 1 aflucena 1049089 0 ene 30 16:34 ./

drwxr-xr-x 1 aflucena 1049089 0 ene 30 16:33 ../

drwxr-xr-x 1 aflucena 1049089 0 ene 30 16:34 .gradle/

-rw-r--r-- 1 aflucena 1049089 736 ene 30 16:34 build.gradle

drwxr-xr-x 1 aflucena 1049089 0 ene 30 16:34 gradle/

-rwxr-xr-x 1 aflucena 1049089 5299 ene 30 16:34 gradlew\*

-rw-r--r-- 1 aflucena 1049089 2260 ene 30 16:34 gradlew.bat

-rw-r--r-- 1 aflucena 1049089 1534 may 24 2017 pom.xml

-rw-r--r-- 1 aflucena 1049089 38 ene 30 16:34 settings.gradle

drwxr-xr-x 1 aflucena 1049089 0 may 24 2017 src/

Publicación de artefactos

El plugin java tiene un artefacto por defecto para realizar el uplodad, se trata del archivo jar. De forma análoga el plugin war tiene el artefacto por defecto war. El usuario tiene la posibilidad de definir los artefactos y hacer el upload por medio de la tarea uploadArchive.

Ejemplo publicación sobre repositorio maven:

En el fichero build\_uploadArtifact.gradle:

apply plugin: 'java'

version=1.0

repositories {

mavenCentral()

} dependencies {

compile ('log4j:log4j:1.2.16')

}

uploadArchives {

repositories {

maven {

credentials {

username "user1"

password "user1"

}

url "http://company.private.repo"

}

}

}

y para ejecutarlo:

gradle –b uploadArtifact.gradle uploadArchives

Es este segundo ejemplo definimos el artefacto de subida:

apply plugin: 'java'

archivesBaseName="JavaProject" // to customize Jar Name

version=1.0

repositories {

mavenCentral()

}

def customFile= file('configurations.xml')

task customZip(type: Zip) {

from 'src'

}

artifacts {

archives customFile

archives customZip

}

uploadArchives {

repositories {

flatDir {dirs "./tempRepo"}

}

}

Lo que se está haciendo es manipular la tarea uploadArchive del plugin java

Gradle y la integración continua (CI)

Jenkins también está basado en plugin, con lo que integraremos gradle en Jenkins instalando su plugin.

Pruebas de integración

Dado que las pruebas de integración son costosas, conviene separarlas de las pruebas unitarias que tendríamos que hacer después de cada compilación, en el libro “Gradle essentials”, nos propone la siguiente estructura y posterior ejecución de una tarea predefinida:

IntegrationSample

src

integrationTest

java

main

java

test

java

tarea:

apply plugin: 'java'

sourceSets {

integrationTest {

java.srcDir file('src/integrationTest/java')

resources.srcDir file('src/integrationTest/resources') // to add theresources

}

}

task runIntegrationTest(type: Test) {

testClassesDir = sourceSets.integrationTest.output.classesDir

classpath = sourceSets.integrationTest.runtimeClasspath

}

Construcción de plugins

Dentro del propio script gradle es posible construir un plugin en groovy, implementando la interfaz plugin en una clase groovy.

class CustomPlugin implements Plugin<Project> {

void apply(Project project) {

project.task('task1') << {

println "Sample task1 in custom plugin"

}

project.task('task2') << {

println "Sample task2 in custom plugin"

}

}

}

task2.dependsOn task1

Esta forma de implementar un plugin no es conveniente pues estamos mezclado la lógica de construcción con la lógica de negocio o de cliente.

Es recomendable sacar la construcción de plugin a:

Al directorio buildSrc, que cuelga del directorio raíz del proyecto.

Plugin independientes.

El wrapper de gradle

Con el wrapper de gradle se evitan problemas debidos a distintas versiones de gradle entre los miembros del equipo de desarrolladores o en general para los que importen tu proyecto. Para ponerlo en práctica, hay que seguir los pasos:

1) En el build.gradle incorporar:

task createWrapper (type:wraper){

gradleVersion = 3.3// con la que se está haciendo el desarrollo

}

La tarea se puede llamar como se quiera.

2) Ejecutar esa tarea: gradle createWrapper.

Con esto se creará la carpeta gradle/, que tendremos que incorporar en el repositorio, para que los que clonan el proyecto puedan disponer de él.

También crea los archivos:

gradlew. Shell script para sistemas que usen una variante de UN\*X

gradlew.bat. Archivo por lotes para Windows

En el repositorio del proyecto, tendremos que incluir estos tres, la carpeta gradle y los archivox gradlew y gradlew.bat.

3) El desarrollador que ha descargado del proyecto del repositorio, trabajará con el bat gradlew.bat y automáticamente se descargará la versión indicada en el build.grade, es nuestro caso la 3.3. Por tanto el nuevo desarrollador, puede no tener gradle instalado, simplemente con el primer comando gradlew que ejecute, se descargará la versión requerida.