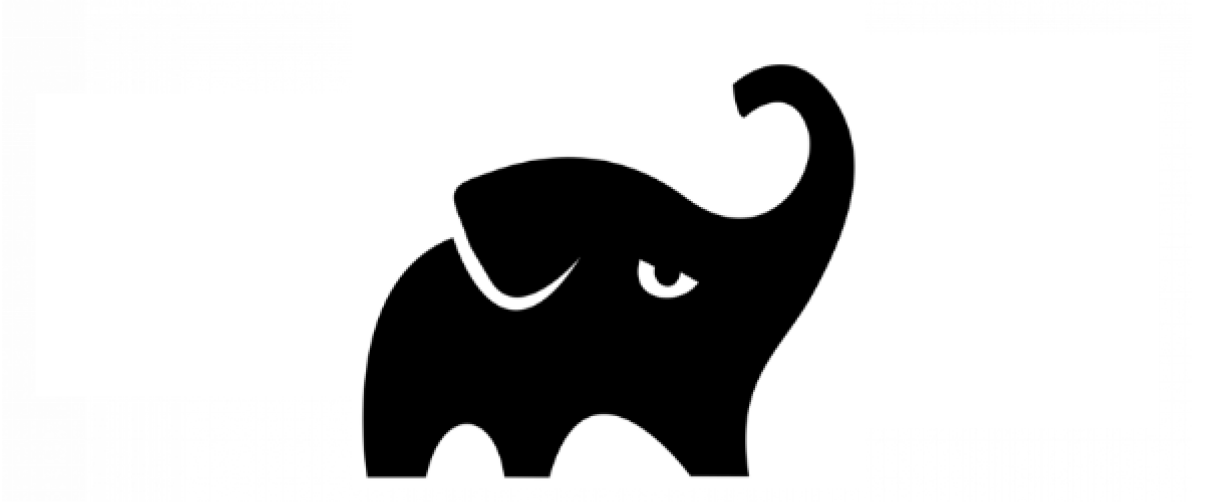

Gradle

Victor Herrero Cazurro

Contenidos

1. Introducción

Es una herramienta de construcción como Maven o ANT, basada en un DSL Groovy.

La documentacion del DSL se encuentra aquí y una guia de usuario aquí

En gradle se definen principalmente tres tipos de objetos

- Projects
- Task
- Settings

Los proyectos gestionados con **Gradle**, se basan en la configuración definida en los ficheros **.gradle**, por defecto el fichero de configuración se llama **build.gradle**, aunque esto se puede cambiar, no es recomendable por favorecer la legibilidad.

A estos ficheros, se les denomina ficheros de **Script**, y están escritos en un DSL de **groovy**

2. Instalacion

Descargar la distribucion de aquí

Definir la variable de entorno **GRADLE_HOME** con la siguiente ruta **<directorio de descompresion de gradle>**

Añadir a la variable de entorno **PATH** la siguiente ruta **GRADLE_HOME/bin**

2.1. IDEs

Gradle se puede emplear desde varios IDEs, incluidos los mas habituales: eclipse, netbeans, sts, intellij, …¬

Para eclipse el plugin recomendado, que habrá que añadir, es **BuildShip Gradle Integration**, aunque es un pulgin que solo ofrece **resaltado de sintaxis** y **gestor de tareas gradle** integrado.

[Gradle eclipse plugin] | *Gradle_eclipse_plugin.png*

Para las ultimas versiones de eclipse (> Neon), se recomienda **EGradle**, que incluye tambien **content asssit**.

3. Primer proyecto

Una vez que se han definido las variable de entorno, se puede crear un nuevo proyecto con el comando

```
> gradle init
```

Este comando crea la estructura de ficheros necesaria para un proyecto gradle.

Se puede definir el tipo de proyecto a crear, pudiendose asi emplear distintas plantillas

- java-application
- java-library
- scala-library
- groovy-library
- basic

```
> gradle init --type java-library
```

3.1. Tareas por defecto

En cualquier proyecto Gradle por defecto se definen una serie de **task**, algunas son

- **init** ¬ Crea un nuevo proyecto gradle.
- **dependencies** ¬ Muestra las dependencias del proyecto.
- **help** ¬ Muestra la ayuda.
- **projects** ¬ Muestra los subproyectos del proyecto principal.
- **properties** ¬ Muetra las propiedades del proyecto.
- **tasks** ¬ lista las tareas disponibles.

Para ejecutarlas, desde la linea de comandos, se hace referencia al comando **gradle** y a la tarea o conjunto de tareas deseadas

```
> gradle tasks

> gradle tasks projects
```

Con el primer comando se obtiene una salida similar a esta

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

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

Help tasks
----------
buildEnvironment - Displays all buildscript dependencies declared in
root project 'HolaMundo'.
components - Displays the components produced by root project
'HolaMundo'. [incubating]
dependencies - Displays all dependencies declared in root project
'HolaMundo'.
dependencyInsight - Displays the insight into a specific dependency in
root project 'HolaMundo'.
dependentComponents - Displays the dependent components of components
in root project 'HolaMundo'. [incubating]
help - Displays a help message.
model - Displays the configuration model of root project 'HolaMundo'.
[incubating]
projects - Displays the sub-projects of root project 'HolaMundo'.
properties - Displays the properties of root project 'HolaMundo'.
tasks - Displays the tasks runnable from root project 'HolaMundo'.
```

Si se lanza el siguiente comando

```
> gradle properties
```

Se obtiene una salida similar a esta

```
------------------------------------------------------------
```

```
Root project
------------------------------------------------------------

allprojects: [root project 'HolaMundo']
ant: org.gradle.api.internal.project.DefaultAntBuilder@57513a9
antBuilderFactory:
org.gradle.api.internal.project.DefaultAntBuilderFactory@1b87129c
artifacts:
org.gradle.api.internal.artifacts.dsl.DefaultArtifactHandler_Decorated@
662d785b
asDynamicObject: DynamicObject for root project 'HolaMundo'
baseClassLoaderScope:
org.gradle.api.internal.initialization.DefaultClassLoaderScope@4cedcfe0
buildDir: D:\GitLab\Gradle\workspace\HolaMundo\build
buildFile: D:\GitLab\Gradle\workspace\HolaMundo\build.gradle
buildPath: :
buildScriptSource:
org.gradle.groovy.scripts.TextResourceScriptSource@61a8658f
buildscript:
org.gradle.api.internal.initialization.DefaultScriptHandler@421888d4
childProjects: {}
class: class org.gradle.api.internal.project.DefaultProject_Decorated
classLoaderScope:
org.gradle.api.internal.initialization.DefaultClassLoaderScope@4ffadf6c
components: SoftwareComponentInternal set
configurationActions:
org.gradle.configuration.project.DefaultProjectConfigurationActionConta
iner@6dca9bbc
configurationTargetIdentifier:
org.gradle.configuration.ConfigurationTargetIdentifier$1@4f6c0f97
configurations: configuration container
convention: org.gradle.api.internal.plugins.DefaultConvention@49347086
defaultTasks: []
deferredProjectConfiguration:
org.gradle.api.internal.project.DeferredProjectConfiguration@502cc773
dependencies:
org.gradle.api.internal.artifacts.dsl.dependencies.DefaultDependencyHan
dler_Decorated@6902a1b1
dependencyLocking:
org.gradle.internal.locking.DefaultDependencyLockingHandler_Decorated@4
63d0bb1
depth: 0
description: null
displayName: root project 'HolaMundo'
ext:
```

```
org.gradle.api.internal.plugins.DefaultExtraPropertiesExtension@2c67f3b
0
extensions: org.gradle.api.internal.plugins.DefaultConvention@49347086
fileOperations:
org.gradle.api.internal.file.DefaultFileOperations@1a46ae10
fileResolver: org.gradle.api.internal.file.BaseDirFileResolver@66359bc0
gradle: build 'HolaMundo'
group:
identityPath: :
inheritedScope:
org.gradle.api.internal.ExtensibleDynamicObject$InheritedDynamicObject@
48fb6434
layout: org.gradle.api.internal.file.DefaultProjectLayout@5b92e4e8
logger:
org.gradle.internal.logging.slf4j.OutputEventListenerBackedLogger@531f7
239
logging:
org.gradle.internal.logging.services.DefaultLoggingManager@740be7a4
modelRegistry:
org.gradle.model.internal.registry.DefaultModelRegistry@3eaa997e
modelSchemaStore:
org.gradle.model.internal.manage.schema.extract.DefaultModelSchemaStore
@1ffb6e4d
module: org.gradle.api.internal.artifacts.ProjectBackedModule@43ff359b
name: HolaMundo
normalization:
org.gradle.normalization.internal.DefaultInputNormalizationHandler_Deco
rated@2de3805b
objects: org.gradle.api.internal.model.DefaultObjectFactory@611a84ab
parent: null
parentIdentifier: null
path: :
pluginManager:
org.gradle.api.internal.plugins.DefaultPluginManager_Decorated@4a0769eb
plugins: [org.gradle.api.plugins.HelpTasksPlugin@63a2e7d1]
processOperations:
org.gradle.api.internal.file.DefaultFileOperations@1a46ae10
project: root project 'HolaMundo'
projectConfigurator:
org.gradle.api.internal.project.BuildOperationCrossProjectConfigurator@
200c78bc
projectDir: D:\GitLab\Gradle\workspace\HolaMundo
projectEvaluationBroadcaster: ProjectEvaluationListener broadcast
projectEvaluator:
org.gradle.configuration.project.LifecycleProjectEvaluator@115cd5be
```

```
projectPath: :
projectRegistry:
org.gradle.api.internal.project.DefaultProjectRegistry@245634e2
properties: {...}
providers:
org.gradle.api.internal.provider.DefaultProviderFactory@7e342f70
repositories: repository container
resourceLoader:
org.gradle.internal.resource.transfer.DefaultUriTextResourceLoader@55ef
49b2
resources:
org.gradle.api.internal.resources.DefaultResourceHandler@16e5022b
rootDir: D:\GitLab\Gradle\workspace\HolaMundo
rootProject: root project 'HolaMundo'
script: false
scriptHandlerFactory:
org.gradle.api.internal.initialization.DefaultScriptHandlerFactory@61e3
d46c
scriptPluginFactory:
org.gradle.configuration.ScriptPluginFactorySelector@74871d96
serviceRegistryFactory:
org.gradle.internal.service.scopes.ProjectScopeServices$4@5b8a686d
services: ProjectScopeServices
standardOutputCapture:
org.gradle.internal.logging.services.DefaultLoggingManager@740be7a4
state: project state 'EXECUTED'
status: release
subprojects: []
tasks: task set
version: unspecified
```

3.2. Establecer properties

Es habitual redefinir una serie de properties para cada proyecto:

- description
- displayName
- mame
- version

Para ello se ha de asignar un valor a la propiedad en el fichero **build.gradle**

```
version = '0.0.1-SNAPSHOT'
```

3.3. Comentarios

Se emplea la misma sintaxis que en groovy o en java

```
//
/*
*/
```

3.4. Aplicar naturaleza eclipse al proyecto

Si se desea trabajar con un IDE como eclipse, se pueden generar los ficheros propios del IDE eclipse, añadiendo el plugin **eclipse**.

```
apply plugin: 'eclipse'
```

Y lanzando el comando

```
> gradle eclipse
```

3.5. Propiedades de Ejecucion

Se puede definir un fichero **gradle.properties**, que defina constantes para la ejecución, estas propiedades se dice que son del proyecto.

```
propiedad=valor
```

Para emplearlas en el fichero **build.gradle** unicamente es necesario hacer referencia al nombre de la propiedad

```
println propiedad
```

Se pueden definir propiedades de sistema

```
systemProp.sistema=valor
```

Y accederlas desde **build.gradle** con

```
println System.properties['systema']
```

NOTE	Ojo con definir propiedades con el mismo nombre que propiedades de la tarea

Si se desea pasar el valor de una propiedad por linea de comandos, se ha de emplear alguna de las siguientes sintaxis

- **-P** para parametros del proyecto
- **-D** para parametros del sistema, con esta ultima opcion, tambien se pueden pasar parametros de proyecto seguiendo el nombrado **org.gradle.project.<propiedad>**

```
> gradle task -PcommandLineProjectProp=commandLineProjectPropValue
-Dorg.gradle.project.systemProjectProp=systemPropertyValue
```

Si se desea acceder a otro fichero de configuracion, se ha de declarar dicho fichero como objeto properties dentro del script

```
def props = new Properties()
file("build.properties").withInputStream { props.load(it) }

task printProps {
    doFirst {
        println props.getProperty("application.name")
        println props.getProperty("project.name")
    }
}
```

Se pueden definir propiedades del proyecto a traves de variables de entorno, siguiendo la sintaxis **ORG_GRADLE_PROJECT_<propiedad>**

```
ORG_GRADLE_PROJECT_foo=bar
```

La misma propiedad definida en el fichero **gradle.properties** seria

```
org.gradle.project.foo=bar
```

4. Logging

Se puede configurar con que nivel de granularidad se quiere obtener la traza, los niveles son:

- ERROR - Error messages
- QUIET - Important information messages
- WARNING - Warning messages
- LIFECYCLE - Progress information messages
- INFO - Information messages
- DEBUG - Debug messages

Para configurar el nivel de log, se puede optar por dos formas:

- Pasando un parametro a la ejecucion
  - ¬ <sin parametro> - LIFECYCLE y superiores
  - ¬ -q or --quiet - QUIET y superiores
  - ¬ -w or --warn - WARN y superiores
  - ¬ -i or --info - INFO y superiores
  - ¬ -d or --debug - DEBUG y superiores (es decir todos)
- Definiendo en el fichero **gradle.properties** la propiedad **org.gradle.logging.level** con alguno de los siguientes valores: quiet, warn, lifecycle, info, debug.

Para hacer uso del log en las tareas definidas, basta con emplear el objeto **logger** definido por **gradle**

```
logger.quiet('An info log message which is always logged.')
logger.error('An error log message.')
logger.warn('A warning log message.')
logger.lifecycle('A lifecycle info log message.')
logger.info('An info log message.')
logger.debug('A debug log message.')
logger.trace('A trace log message.')
```

5. Ciclo de vida del Build

Gradle define tres fases en el ciclo de vida del Build

- **Initialization** ¬ Como se soporta la construccion de multiples proyectos, en esta fase, se determina que proyectos participan del build. Creando una instancia de **Project** por cada uno de ellos.
- **Configuration** ¬ En esta fase los scripts de todos los proyectos son ejecutados, con lo que se acaban configurando los objetos que participan del Build, como los **Repository**, y todas las **Task** disponibles.
- **Execution** ¬ En esta fase se determina el conjunto de **Task** involucradas en la ejecucion del Build, el orden de ejecución de dichas **Task** y se ejecutan.

5.1. Inicializacion

Durante esta fase, se ejecutan los scripts definidos en el fichero **settings.gradle**, que genera un objeto **Settings**

Dado el fichero settings.gradle

```
println 'This is executed during the initialization phase.'
```

y el fichero build.gradle

```
println 'This is executed during the configuration phase.'

task configured {
    println 'This is also executed during the configuration phase.'
}

task test {
    doLast {
        println 'This is executed during the execution phase.'
    }
}

task testBoth {
    doFirst {
      println 'This is executed first during the execution phase.'
    }
    doLast {
      println 'This is executed last during the execution phase.'
    }
    println 'This is executed during the configuration phase as well.'
}
```

Ejecutando la tarea

```
> gradle test testBoth
```

Se obtiene la siguiente traza.

```
This is executed during the initialization phase.
This is executed during the configuration phase.
This is also executed during the configuration phase.
This is executed during the configuration phase as well.
:test
This is executed during the execution phase.
:testBoth
This is executed first during the execution phase.
This is executed last during the execution phase.

BUILD SUCCESSFUL

Total time: 1 secs
```

6. Proyecto Java

Se basa en la aplicacion del plugin **java**, uno de los plugins mas empleados.

Para añadirlo

```
apply plugin: 'java'
```

6.1. Tareas

Este plugin añade tareas para poder procesar proyectos java (build).

- **assemble** ¬ Genera el jar del proyecto.
- **build** ¬ Genera el jar del proecto y lo prueba.
- **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 main classes.
- **clean** ¬ Deletes the build directory.
- **jar** ¬ Assembles a jar archive containing the main classes.
- **testClasses** ¬ Assembles test classes.

Para chequear el estado del proyecto (verification)

- **test** ¬ ejecuta la tarea de test
- **check** ¬ es la ultima tarea de la fase verificacion, existe para poder garantizar que todas las tareas de verificacion se ejecutan

Y generar la documentacion (documentation)

- **javadoc**

Estas tareas estan enlazadas entre sí, formando un **Ciclo de Vida** para proyectos java.

[javaPluginTasksDiagram] | *javaPluginTasksDiagram.png*

6.2. Estructura del Proyecto

El plugin asume que se empleará la siguiente estructura de proyecto

- **src/main/java** - Directorio que contiene los ficheros java del proyecto
- **src/main/resources** - Directorio que contiene los ficheros no compilables (resources) del proyecto
- **src/test/java** - Directorio que contiene los ficheros java para probar el proyecto
- **src/test/resources** - Directorio que contiene los ficheros no compilables (resources) para probar el proyecto
- **src/<sourceSet>/java** - Directorio que contiene los ficheros java para el **SourceSet** indicado
- **src/<sourceSet>/resources** - Directorio que contiene los ficheros no compilables (resources) para el **SourceSet** indicado

Se puede cambiar esta estructura con la siguiente configuracion

```
sourceSets {
    main {
        java {
            srcDirs = ['src/java']
        }
        resources {
            srcDirs = ['src/resources']
        }
    }
}
```

6.3. Dependencias

Permiten enriquecer el classpath del proyecto con librerias de terceros.

```
dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
    testCompile group: 'org.seleniumhq.selenium', name: 'selenium-
java', version: '2.53.0'
}
```

Otro formato de definicion de las dependencias sería

```
dependencies {
    compile 'org.hibernate:hibernate-core:3.6.7.Final'
}
```

Se puede indicar que se desea la ultima version empleando el operador **+**

```
dependencies {
    compile 'org.hibernate:hibernate-core:+'
}
```

6.3.1. Repositorios

Siempre que se vaya a trabajar con dependecias es necesario definir repositorios desde donde poder realizar la descarga. Los mas habituales son:

- jcenter
- mavenCentral

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

Se pueden definir repostiorios particulares

```
maven {
    url "http://localhost:8081/nexus/content/repositories/central/"
}
```

Y definir autenticacion para acceder a ellos

```
maven {
    url "http://localhost:8081/nexus/content/repositories/releases/"
    credentials {
        username = 'admin'
        password = project.credentials.nexus
    }
}
```

6.3.2. Ambitos

Para este tipo e proyectos se definen los siguientes ambitos

- **compile(Deprecated)** ¬ Compile time dependencies. Superseded by implementation.
- **implementation extends compile** ¬ Implementation only dependencies.
- **compileOnly** ¬ Compile time only dependencies, not used at runtime.
- **compileClasspath extends compile, compileOnly, implementation** ¬ Compile classpath, used when compiling source. Used by task compileJava.
- **annotationProcessor** ¬ Annotation processors used during compilation.
- **runtime(Deprecated) extends compile** ¬ Runtime dependencies. Superseded by runtimeOnly.
- **runtimeOnly** ¬ Runtime only dependencies.
- **runtimeClasspath extends runtimeOnly, runtime, implementation** ¬ Runtime classpath contains elements of the implementation, as well as runtime only elements.
- **testCompile(Deprecated) extends compile** ¬ Additional dependencies for compiling tests. Superseded by testImplementation.
- **testImplementation extends testCompile, implementation** ¬ Implementation only dependencies for tests.
- **testCompileOnly** ¬ Additional dependencies only for compiling tests, not used at runtime.
- **testCompileClasspath extends testCompile, testCompileOnly, testImplementation** ¬ Test compile classpath, used when compiling test sources. Used by task compileTestJava.

- **testRuntime(Deprecated) extends runtime, testCompile** ¬ Additional dependencies for running tests only. Used by task test. Superseded by testRuntimeOnly.
- **testRuntimeOnly extends runtimeOnly** ¬ Runtime only dependencies for running tests. Used by task test.
- **testRuntimeClasspath extends testRuntimeOnly, testRuntime, testImplementation** ¬ Runtime classpath for running tests.
- **archives** ¬ Artifacts (e.g. jars) produced by this project. Used by tasks uploadArchives.
- **default extends runtime** ¬ The default configuration used by a project dependency on this project. Contains the artifacts and dependencies required by this project at runtime.

6.4. Jar name

Se puede configurar el nombre del jar que se genera con las propiedades **archivesBaseName** y **version**

```
archivesBaseName = "miProyecto"
version = '0.0.1-SNAPSHOT'
```

6.5. Java Version

Se puede definir la version de java a emplear

```
sourceCompatibility = 1.6
targetCompatibility = 1.6
```

6.6. SourceSet

Define un conjunto de recursos del proyecto.

Por defecto se definen **main** y **test**.

Un ejemplo habitual de su uso, es crear un espacio particular para albergar las pruebas de integracion.

```
sourceSets {
    integrationTest {
        java {
            compileClasspath += main.output + test.output
            runtimeClasspath += main.output + test.output
            srcDir file('src/integration-test/java')
        }
        resources.srcDir file('src/integration-test/resources')
    }
}
```

Al definir un nuevo **sourceSet**, se definen por defecto dos **ambitos** (configurations) para gestionar las depedencias a emplear al trabajar con el **SourceSet**.

Para el ejemplo anterior serian **integrationTestCompile** y **integrationTestRuntime**.

Estos **ambitos** se pueden redefinir indicando algun tipo de herencia, es decir reutilizar otras dependencias definidas para otro ambito en el ambito definido, en este caso es interesante que hereden de los ambitos de **test**

```
configurations {
    integrationTestCompile.extendsFrom testCompile
    integrationTestRuntime.extendsFrom testRuntime
}
```

Una vez definido los ambitos, se pueden añadir depedencias con dichos ambitos

```
dependencies {
    integrationTestCompile 'junit:junit:4.11'
}
```

Las dependencias añadidas a un ambito, permiten componer los classpath a emplear por las tareas, en este caso para ejecutar los **test de integracion**.

```
task integrationTest(type: Test) {
    testClassesDirs = sourceSets.integrationTest.output.classesDirs
    classpath = sourceSets.integrationTest.runtimeClasspath
}
```

Se puede definir relaciones entre la nueva **Task** y las existentes, de tal forma que se

la haga participe de por ejemplo el **ciclo de vida java**. En el caso de los test de integracion, es interesante incluir la tarea despues de ejecutar los **test** y antes de ejecutar **check**

```
check.dependsOn integrationTest
integrationTest.mustRunAfter test
```

6.7. Selenium

Si se desean realizar pruebas de integracion puede ser interesante trabajar con un framework como selenium, que permite automatizar la interaccion con los navegadores.

Para incorporarlo al proyecto basta con añadir la dependencia

```
repositories {
    mavenCentral()
}

dependencies {
    testCompile group: 'junit', name: 'junit', version: '4.11'
    testCompile group: 'org.seleniumhq.selenium', name: 'selenium-
java', version: '2.53.0'
}
```

Un ejemplo de uso d el framework seria

```
public class SeleniumTest {
    private WebDriver driver;

    @BeforeClass
    public static void setUp() {
        System.setProperty("webdriver.chrome.driver",
                "D:\\utilidades\\Selenium\\chromedriver\\chromedriver-
2.40.exe");
    }

    @Test
    public void testChromeDriver() {
        driver = new ChromeDriver();
        driver.manage().window().maximize();
        driver.get("http://localhost:8080/");
        String titulo = driver.getTitle();
        assertNotNull(titulo);
        assertEquals("Apache Tomcat/8.5.9", titulo);
    }

    @After
    public void closeBrowsers() throws Exception {
        driver.quit();
    }
}
```

7. Proyecto Java Web

Se basa en la aplicacion del plugin **war**, uno de los plugins mas empleados.

Para añadirlo

```
apply plugin: 'war'
```

7.1. Tareas

Añade una unica tarea **war**.

7.2. Estructura del Proyecto

Por defecto toma el directorio **src/main/webapp** como el directorio de contenido web.

7.3. Ambitos

Se añaden dos nuevos ambitos (configurations) **providedCompile** y **providedRuntime**, para indicar librerias que serán provistas por el servidor donde se despliegue.

Permite generar un **war** en vez de un **jar** con el proyecto java, con la tarea **build**

```
gradle build
```

Define una serie de propiedades entre las cuales se encuentra el directorio donde se genera en **war**, que es **archivePath**

8. Task

La pieza más pequeña que compone un script de gradle es un **task**, una tarea.

Para definir un **task**, se emplea la palabra reservada para el DSL **task**

```
task helloWorld {
    doLast {
        println 'Hello world!'
    }
}
```

Existen otras sintaxis validas como

```
task(helloWorld) {
    doLast {
        println 'Hello world!'
    }
}
```

Estas tareas no se muestran por defecto con el comando **task**, ya que se asignan al

grupo por defecto **other**, se pueden mostrar todos los grupos añadiendo un parametro

```
> gradle task --all
```

O asignandolas a un grupo

```
task(helloWorld) {
    group 'hello'
    doLast {
        println 'Hello world!'
    }
}
```

Para ejecutar la nueva tarea basta con ejecutar el comando

```
> gradle helloWorld
```

8.1. Acciones (ciclo de vida de la tarea)

Las task de gradle, están formadas por acciones como **doLast**, que es la última acción que se ejecutará asociada al **task**, y ademas es la accion por defecto

```
task(helloWorld) {
    doLast {
        println 'Hello world!'
    }
}
```

Al ser la accion por defecto, tiene una sintaxis reducida

```
task helloWorld << {
    println 'Hello world!'
}
```

Existen más acciones como:

- dofirst

- onlyIf
- mustRunAfter
- hasProperty

8.2. Propiedades

Se pueden añadir más propiedades a las tareas, empleando la propiedad **ext**.

```
task myTask {
    ext.myProperty = "myValue"

    doLast {
        println myProperty
    }
}
```

8.3. Ciclo de Vida del Proyecto

En Gradle, no se define un ciclo de vida como tal, sino que se puede definir dependencias de ejecucion entre las tareas, de tal forma que se supedite la ejecucion de una tarea particular a la ejecucion de otra, basta con establecer la propiedad **dependsOn** de la tarea que tiene que supeditar su ejecución a otra.

```
task myTask{
    dependsOn help
}
```

Tambien se puede emplear la sintaxis

```
task hello {
    doLast {
        println 'Hello world!'
    }
}
task intro(dependsOn: hello) {
    doLast {
        println "I'm Gradle"
    }
}
```

Si la tarea proviene de un plugin, es decir no se tiene acceso a su codigo fuente, basta con hacer referencia al nombre de la tarea.

Ejemplo en el que se supedita la ejecucion de la tarea de Liquibase **update**, *a la ejecucion de la tarea* **help**

```
update.dependsOn test
```

Las tareas se ejecutan por completo, no se pueden anidar.

8.4. Herencia en las Tareas

Se pueden definir nuevas tareas, que hereden la configuracion de la tarea de un tipo predefinido

```
task hello(type: GreetingTask)
```

Para el ejemplo anterior, tambien se proporciona la siguiente clase que define la tarea padre.

```
class GreetingTask extends DefaultTask {
    @TaskAction
    def greet() {
        println 'hello from GreetingTask'
    }
}
```

Se pueden definir tareas basandose en la clase **DefaultTask**

Un ejemplo de como heredar la tarea de **copy** que proporciona el API

```
task(copy, type: Copy) {
    from(file('srcDir'))
    into(buildDir)
}
```

8.5. Ejecucion de tareas multiple

Se puede asignar a los métodos de las tareas, varios bloques de código, ejecutandose todos, en el orden en el que se asignen

Así pues con la siguiente definición

```
task hello {
    doLast {
        println 'Hello Earth'
    }
}
hello.doFirst {
    println 'Hello Venus'
}
hello.doLast {
    println 'Hello Mars'
}
hello {
    doLast {
        println 'Hello Jupiter'
    }
}
```

Y lanzando la tarea

```
>gradle hello
```

Se obtiene la salida

```
Hello Venus
Hello Earth
Hello Mars
Hello Jupiter
```

8.6. Ejecutando comandos del SO

Para ejecutar comandos del SO, se porporcionan el tipo de tarea **Exec**

```
task startTomcat(type:Exec) {
    group 'docker'
    executable 'docker'
    args 'run','-d', '--name', 'dockerTomcat','--rm','-p',
'8080:8080','-v', buildDir.getAbsolutePath() + '\\libs' +
':/usr/local/tomcat/webapps', 'tomcat:8.5-alpine'
}
```

8.7. Ejecutando Test

Para la ejecucion e Test, se porporcionan el tipo de tarea **Test**

```
task integrationTest(type: Test) {
    testClassesDirs = sourceSets.integrationTest.output.classesDirs
    classpath = sourceSets.integrationTest.runtimeClasspath
}
```

9. Plugins

Los plugins son proyectos independientes que ofrecen un conjunto de **task** a otros proyectos.

Para usar un plugin, hay que referenciarlo desde el script.

```
apply plugin: <nombre del plugin>
```

9.1. Plugins de terceros

Para añadir un plugin definido por terceros, es necesario incluir el jar en el ambito de la interpretacion del **script**, para ello se define la seccion **buildScript**, que permite configurar este ambito.

Seccion del **script** que permite definir las necesidades que tiene Gradle para ejecutar el **script**, es decir, es la configuracion para poder ejecutar el **build.gradle**

Configuracion para el uso del plugin Gretty

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'org.akhikhl.gretty:gretty:2.0.0'
    }
}
apply plugin: 'org.akhikhl.gretty'
```

9.2. Custom Plugins

Para crear plugins basta con definir una nueva clase que herede de **Plugin<Project>** dentro del **build.gradle**

```
class GreetingPlugin implements Plugin<Project> {
    void apply(Project project) {
        project.task('hello') {
            doLast {
                println "Hello from the GreetingPlugin"
            }
        }
    }
}
```

Esta herencia, proporciona un método **apply**, que permite asociar al **project** nuevas **task**.

Para emplear el plugin desde el mismo script que lo define, basta con añadir

```
apply plugin: GreetingPlugin
```

Una vez se aplica el nuevo **plugin** las **task** definidas, están disponiles y se puede invocar dicha tarea

```
gradle -q hello
```

9.2.1. Configuracion Custom Plugins

Para poder configurar un Plugin, lo normal es definir una nueva clase con los atributos que se deseen definir para la configuracion

```
class DockerTomcatPluginExtension {

  String containerName = 'tomcat8-test'
    String imageName = 'tomcat:8.5-alpine'
    String localPort = '8080'
    String deployDir

  DockerTomcatPluginExtension(Project project) {
      deployDir = project.buildDir.getAbsolutePath() + '\\libs'
    }
}
```

Y asignarla al Plugin como una **extension**

```
class DockerTomcatPlugin implements Plugin<Project> {

    void apply(Project project) {

      def extension = project.extensions.create('dockerTomcat',
DockerTomcatPluginExtension, project)

      project.task('stopTomcat', type:Exec) {
        group 'docker'
        executable 'docker'
      args 'stop', extension.containerName
    }

    project.task('startTomcat', type:Exec) {
      group 'docker'
      executable 'docker'
        args 'run','-d', '--name', extension.containerName,'--rm','-p',
extension.localPort + ':8080','-v', extension.deployDir +
':/usr/local/tomcat/webapps', extension.imageName
        /*
        -d --> Demonio, se arranca el contenedor liberando el proceso
de arranque.
        --name <nombre> --> Establece el nombre del contenedor
        --rm --> Indica que cuando se pare el contenedor se elimine
        -P <portLocal>:<portContainer> --> Establece el mapeo de
puertos
        -v <directorioLocal>:<directorioContainer> --> Establece el
mapeo de volumenes
        */
    }
    }
}
```

Una vez hecho, basta con crear una instancia de la clase, con los valores de los campos que se desee.

```
apply plugin: 'war'

repositories {
    mavenCentral()
}

apply plugin: DockerTomcatPlugin

dockerTomcat {
    containerName = 'tomcat8-test'
    imageName = 'tomcat:8.5-alpine'
    localPort = '8080'
    deployDir = war.archivePath
}
```

10. Otros Plugins

10.1. Jetty Plugin (Deprecated en Gradle 4.0)

Para añadirlo

```
apply plugin: 'jetty'
```

Permite arrancar un jetty y desplegar la aplicacion web

```
gradle jettyRunWar
```

10.2. Gretty Plugin

Sustituto del plugin de Jetty, para emplearlo hay que añadir la siguiente configuracion

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'org.akhikhl.gretty:gretty:2.0.0'
    }
}
apply plugin: 'org.akhikhl.gretty'
```

o bien

```
plugins {
    id 'org.akhikhl.gretty' version '2.0.0'
}
```

Se añaden multitud de **tasks**, entre las que puede destacar **appRun**.

Por defecto se despliega la aplicacion en un jetty9, pero se puede cambiar indicando la propiedad **servletContainer**, donde los valores posibles son: jetty7, jetty8, jetty9, jetty93 (jdk 8 only), jetty94 (jdk 8 only), tomcat7 y tomcat8.

Si se desea ejecutar test de integracion sobre el despliegue en el servidor que gestiona **Gretty**, unicamente hay que indicar cual es la tarea que ejecuta dichos test.

```
gretty {
  integrationTestTask = 'integrationTest'
}

task integrationTest {
  group 'verification'
  doLast {
    println 'Ejecutando los test de integracion'
  }
}
```

10.3. Liquibase Plugin

Permite interaccionar con una base de datos con el objetivo de prepararla para su uso, en general en pruebas de integracion.

Se ha declarar como dependencia de la fase de construccion

```
buildscript {
    repositories {
        mavenCentral()
    }
    dependencies {
        classpath "org.liquibase:liquibase-gradle-plugin:1.2.4"
    }
}
apply plugin: 'org.liquibase.gradle'
```

Y configurar:

- El fichero donde se iran generando los cambios que se producen en la base de datos.
- La conexion a la base de datos.

```
liquibase {
  activities {
    main {
      changeLogFile "$rootDir/changelog.xml"
      url 'jdbc:derby://localhost:1527/facturas;create=true'
      username 'user'
      password 'user'
    }
  }
  runList = 'main'
}
```

El uso del plugin será el siguiente

- Si la base de datos existe:
  - ¬ Lanzar el comando **generateChangelog**, que genera el fichero configurado **changelog.xml** y las tablas **DATABASECHANGELOG** y **DATABASECHANGELOGLOCK** en el esquema al que se conecta Liquibase.
  - ¬ Lanzar el comando **changelogSync**, que actualiza la tabla **DATABASECHANGELOG** con las entradas definidas en el fichero **changelog.xml**.

- Si la base de datos no existe:
  - ¬ Crear el fichero **changelog.xml** manualmente
  - ¬ Lanzar el comando **update** que aplica los cambios indicados en el fichero en la base de datos, controlando no aplicar aquellos que ya han sido aplicados.

10.4. Maven

Plugin que permite realizar tareas de **Maven** dentro de un script de **Gradle**, el caso mas habitual de uso, es la publicacion del proyecto como artefacto en un repositorio, para ello

```
apply plugin: 'maven'

apply plugin: 'java'

uploadArchives {
    repositories {
        mavenDeployer {
            repository(url:
"http://localhost:8081/nexus/content/repositories/releases/"){
              authentication(userName: "admin", password: "admin123")
            }
            pom.version = '0.0.1'
            pom.groupId = 'com.ejemplo.gradle'
            pom.artifactId = 'miProyecto'
        }
    }
}
```

Para realizar la plubicacion hay que ejecutar

```
> gradle uploadArchive
```

NOTE

Para ejecutar Nexus en un contenedor de Docker, se puede lanzar el siguiente comando

```
docker run -d -p 8081:8081 --name nexus
sonatype/nexus:oss
```

El servidor será accesible en la url

```
http://localhost:8081/nexus/#welcome
```

Siendo el usuario y password, **admin** y **admin123**

10.5. Credentials

Permite encriptar passwords, para posteriormente emplearlos en el script.

```
buildscript {
    repositories {
        jcenter()
    }
    dependencies {
        classpath 'nu.studer:gradle-credentials-plugin:1.0.4'
    }
}

apply plugin: 'nu.studer.credentials'
```

Una vez definido el plugin, se puede encriptar el password con el comando

```
> gradle addCredentials --key nexus --value admin123
```

Donde **key** es el identificador del password para su recuperacion y **value** el valor del password.

Y empelarlo en el script a traves del objeto **credentials**, haciendo referencia a la *key definida.

```
repository(url:
"http://localhost:8081/nexus/content/repositories/releases/"){
  authentication(userName: "admin", password: credentials.nexus)
}
```

11. MultiProyecto

Se pueden crear proyectos independientes, cada uno con su script de gradle, o se pueden establecer relaciones entre ellos, lo que se denomina un multiproyecto.

Para ello por defecto se supone que los proyectos estaran definidos en forma de arbol de directorios

```
**Proyecto
******Subproyecto
```

Cada uno con su propio **build.gradle**

Para relacionarlos, lo primero es añadir el **settings.gradle** en el principal, donde se hace referencia a los otros.

```
include 'project1', 'project2:child', 'project3:child1'
```

El método **include**, recibe como parametros paths de proyectos relativos al directorio donde se encuentra el fichero **settings.gradle**, el formato cambia las **/** o **** por **:** para estandarizar los distintos SO, por lo que 'services:api' equivale a '<root dir>/services/api'.

Solo es necesario indicar los proyectos finales, no hace falta indicar los intermedios, por lo que si se tienen distintos **build.gradle** definidos en un arbol de carpetas, al incluir **services:hotels:api**, se incluiran los proyectos: **services**, **services:hotels** y **services:hotels:api**.

Una vez definida la relación, en el **build.gradle** del principal, se pueden definir configuraciones comunes empleando **allprojects**

```
allprojects {
    apply plugin : 'java'
}
```

A partir de esa configuracion, cuando se lancen tareas configuradas como comunes, sobre el principal, se lanzaran sobre todos los subproyectos tambien, si ses desea lanzar la tarea sobre un subproyecto concreto, se puede emplear la sintaxis

```
> gradle project2:build
```

Si solo se desea afectar a los subproyectos, se dispone de **subprojects**

```
subprojects {
    apply plugin : 'java'
}
```

Y si se quiere afectar a un proyecto en concreto

```
project(':api') {

}
```

Esta ultima configuracion equivale a definir en cada pryeco su propio **build.gradle**

Se pueden definir dependencias entre proyectos, de forma abreviada

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

12. Otras Funcionalidades

12.1. Deteccion del SO

Gradle porporciona un API de utilidad que permite conocer el SO en el que se ejecuta el script.

```
import org.gradle.internal.os.OperatingSystem;

task detect {
    doLast {
        if(OperatingSystem.current().isMacOsX())
            println("Mac")
        if(OperatingSystem.current().isLinux())
            println("Linux")
        if(OperatingSystem.current().isWindows())
            println("Windows")
    }
}
```

12.2. Perfiles

No se proporciona una funcionalidad particular para activar configuraciones, pero dado que se puede programar, basta con implementar un **if** para importar distintos ficheros de configuracion.

Contenido del **build.gradle**

```
if (!hasProperty('buildProfile')) ext.buildProfile = 'default'

apply from: 'build-' + buildProfile + '.gradle'

task saludo << {
    println mensaje
}
```

Contenido del **build-default.gradle**

```
ext.mensaje = "Default"
```

Contenido del **build-production.gradle**

```
ext.mensaje = "Produccion"
```

Si se ejecuta el comando

```
> gradle saludo
```

Se obtiene como resultado

```
Default
```

Si se ejecuta el comando

```
> gradle saludo -PbuildProfile=production
```

Se obtiene como resultado

```
Produccion
```