ATTSW Exam: Gradle come sostituto di Maven

Gabriele Puliti - 5300140 - *gabriele.puliti@stud.unifi.it*

Marzo 2018

Indice

1 Introduzione: Gradle

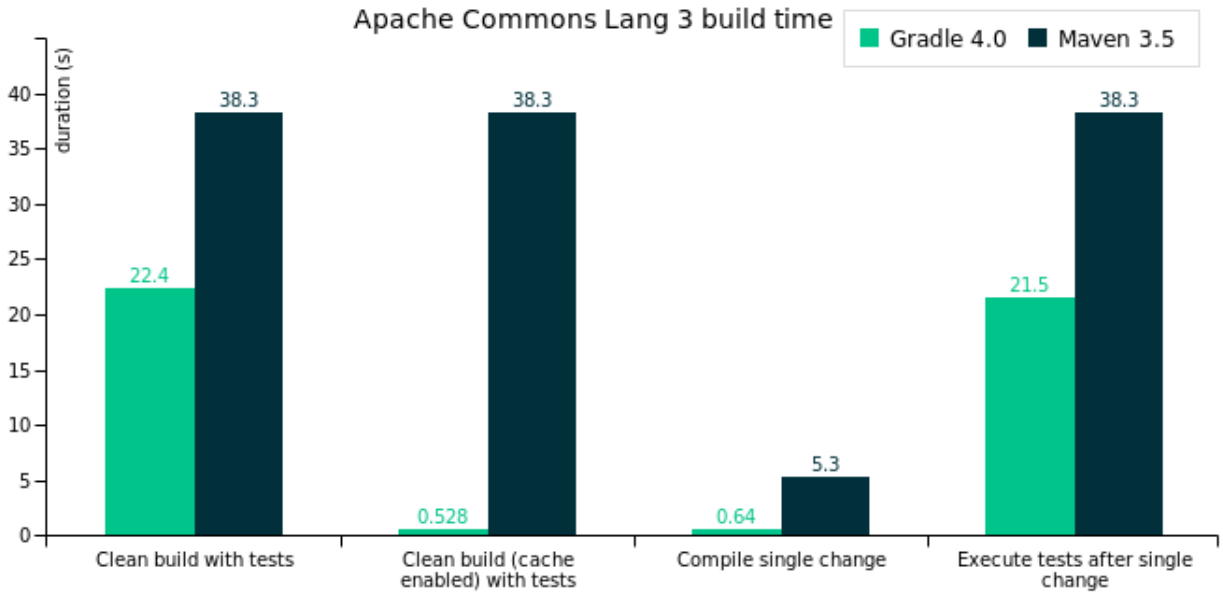
Gradle è un progetto open source che fornisce un tool di build automation, che può essere un ottimo sostituto di Maven. Offre un modello in grado di sostenere l'intero ciclo di vita dello sviluppo del software ed è stato progettato per supportare build automation attraverso più linguaggi e piattaforme. Nel nostro caso considereremo questo tool per lo sviluppo di software Java.

1.1 Differenze tra Gradle e Maven

Ci sono molte differenze tra questi due tools: flessibilità, performance, gestione delle dipendenze e molto altro. La configurazione di Gradle in un progetto ha una convenzione molto più facile e comprensibile rispetto alla tediosa e a volte impossibile configurazione del pom.xml di Maven, anche se entrambi usano dei metodi di miglioramento della velocità di esecuzione delle build. Grandle usufruisce di:

- **Incrementality:** evitando il lavoro di monitoraggio dei task di I/O eseguendo solo il necessario e quando possibile processare solo i files che sono cambiati;
- **Build Cache:** utilizza un sistema di cache riusando gli outputs di altre build Gradle con gli stessi inputs;
- **Deamon:** sfrutta un long-lived process che mantiene tutte le informazioni in memoria.

Queste 3 caratteristiche rendono Gradle molto veloce, ad esempio una build Gradle con Maven verrebbe completata con un tempo 3 volte maggiore. Tutto questo è anche possibile grazie a un sistema di esecuzioni parallele di task e intra-task.



(La relazione è stata scritta per SO Linux, ma i concetti generali valgono per tutti)

1.2 Installazione

L'istallazione di Gradle può essere fatta in più modi: tramite installazione manuale o utilizzando un package manager (tutte le informazioni possono essere trovate in **questo link**). Personalmente consiglio l'utilizzo del software development kit manager **SDKMAN!** che non solo permette l'installazione molto facilitata di Gradle, ma anche della JVM e di tanti altri tools. L'installazione si basa su 2 semplici comandi:

```
$ curl -s "https://get.sdkman.io" | bash

$ source "$HOME/.sdkman/bin/sdkman-init.sh"
```

A questo punto se tutto è andato a buon fine SDKMAN! è stato installato correttamente, è possibile verificarlo digitando il comando su terminale:

```
$ sdk version
```

l'output risultante dovrebbe essere qualcosa del tipo:

```
SDKMAN 5.5.15+284
```

Ora è possibile procedere con l'installazione di Gradle. Prima di tutto visualizziamo la lista delle versioni di Gradle:

```
$ sdk list gradle
```

L'output corrispondente sarà:

```
================================================================================
Available Gradle Versions
================================================================================
     4.6-rc-2            4.3.1               3.5                 2.2.1
     4.6-rc-1            4.3-rc-4            3.4.1               2.2
     4.6                 4.3-rc-3            3.4                 2.14.1
     4.5.1               4.3-rc-2            3.3                 2.14
     4.5-rc-2            4.3-rc-1            3.2.1               2.13
     4.5-rc-1            4.3                 3.2                 2.12
     4.5                 4.2.1               3.1                 2.11
     4.4.1               4.2-rc-2            3.0                 2.10
     4.4-rc-6            4.2-rc-1            2.9                 2.1
     4.4-rc-5            4.2                 2.8                 2.0
     4.4-rc-4            4.1                 2.7                 1.9
     4.4-rc-3            4.0.2               2.6                 1.8
     4.4-rc-2            4.0.1               2.5                 1.7
     4.4-rc-1            4.0                 2.4                 1.6
     4.4                 3.5.1               2.3                 1.5

================================================================================
+ - local version
* - installed
> - currently in use
================================================================================
```

La versione che vogliamo installare è quella più recente che in questo caso è la 4.6, possiamo quindi eseguire il comando:

```
$ sdk install gradle 4.6
```

appena il download e l'installazione sarà finita possiamo verificare il completamento tramite:

```
$ gradle -v
```

che non solo stamperà su terminale la versione di Gradle, ma anche:

- Groovy (linguaggio di programmazione usato per scrivere i file di configurazione)
- Ant (software usato per le build delle Java applications)

- Java Virtual Machine
- sistema operativo in uso

se l'output ha queste informazioni allora Gradle è stato completamente installato. SDKMAN! si preoccupa anche di creare la variabile $GRADLE_HOME che è possibile visualizzare con il comando

```
$ echo $GRADLE_HOME
```

Se ci sono errori di tipo Java, i problemi possono essere:

- Gradle non riesce a trovare la jdk, problema risolvibile installando java con sdkman con il comando

  ```
  $ sdk install java <versione>
  ```

- Java è aggiornato alla versione 9 o superiori (infatti attualmente Gradle non è aggiornato per versioni superiori alla 8), basterà fare un downgrade ad una versione precedente (possibile farlo anche tramite SDKMAN!).

In entrambi i casi sarà necessario anche comunicare al sistema la versione da usare:

```
$ sdk dafault java <versione_installata>
```

per essere sicuri che è stata installata la giusta versione di java possiamo controllare gli outputs dei seguenti comandi:

- `$ echo $JAVA_HOME`
- `$ java -version`

il primo comando dovrà restituire in output il giusto percorso della JVM installata, il secondo serve a controllare la versione java attualmente in uso.

2 Tasks & Task Dependencies

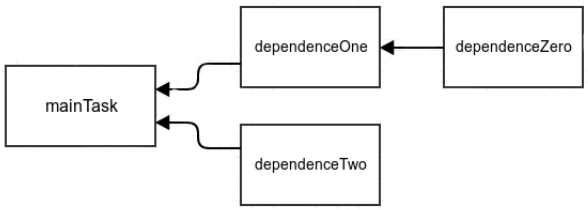
2.1 Configurazione del build.gradle

Come in Maven ci sono i goals, in Gradle ci sono i tasks ognuno dei quali ha il suo scopo definito nella sua implementazione. L'implementazione dei tasks viene fatta in un file di configurazione solitamente nominato build.gradle, che non è altro che uno script in linguaggio Groovy. Creaiamo quindi una cartella in cui inserire la nostra configurazione di gradle e creiamo il file build.gradle in cui andremo a inserire:

```
description = 'Example of Task'

task dependenceZero {
        description = 'Build Dependence Zero'
        doFirst {
                println 'First Zero'
        }
        doLast {
                println 'Last Zero'
        }
}

task dependenceOne(dependsOn: [dependenceZero]) {
        description = 'Build Dependence One'
        doFirst {
                println 'First One'
        }
        doLast {
                println 'Last One'
        }
}

task dependenceTwo {
        description = 'Build Dependence Two'
        doFirst {
                println 'First Two'
        }
        doLast {
                println 'Last Two'
        }
}

task mainTask(dependsOn: [dependenceOne, dependenceTwo]) {
        description = 'Build Main Task'
        doFirst {
                println 'First MainTask'
        }
        doLast {
                println 'Last MainTask'
        }
}
```

In questa build abbiamo definito 4 task: dependenceZero, dependeceOne, dependenceTwo e mainTask. Nella definizione del task può essere usata la parola DEPENDSON per indicare che il task definito dipende da uno o più task. Nel caso di dependenceOne abbiamo una sola dipendenza che è dependenceZero, nel caso invece di taskMain si hanno 2 dipendenze che sono dependenceOne e dependenceTwo. Possiamo notare che

si è data una descrizione sia dei tasks che della build, questo non serve nella pratica ma è buona norma dare sempre una spiegazione sia della build che dei nuovi task che si creano. All'interno dei tasks si nota che ci sono definite delle azioni: doFirst e doLast, quando sarà eseguita la build di un task verrà eseguita prima doFirst e infine doLast. Con la configurazione precedente abbiamo creato un albero delle dipendenze di questo tipo:



Le builds di gradle vengono eseguite usando il comando da terminale `$ gradle taskName`, per l'esempio è possibile quindi eseguire le builds:

- `$ gradle dependenceZero`
- `$ gradle dependenceOne`
- `$ gradle dependenceTwo`
- `$ gradle mainTask`

Ma è anche possibile eseguire più task contemporaneamente, per esempio:

- `$ gradle dependenceZero mainTask`
- `$ gradle dependenceOne dependenceTwo`
- `$ gradle dependenceOne dependenceTwo mainTask`

Considerando che il mainTask è dipendente da dependenceOne e dependenceTwo, l'ultimo esempio non aggiunge niente di più alla build dato che verrebbero comunque eseguiti i 2 tasks. Se eseguiamo infatti

```
$ gradle mainTask
```

e poi

```
$ gradle dependenceOne dependenceTwo mainTask
```

otterremo il solito output, che è il seguende:

```
> Task :dependenceZero
First Zero
Last Zero

> Task :dependenceOne
First One
Last One

> Task :dependenceTwo
First Two
Last Two

> Task :mainTask
First MainTask
```

```
Last MainTask


BUILD SUCCESSFUL in 0s
4 actionable tasks: 4 executed
```

2.2 Approfondimenti

Andiamo ad approfondire le azioni che è possibile fare tramite il terminale.

2.2.1 Abbreviazione dei nomi

È possibile abbreviare il nome del task da eseguire stando però attenti ad identificarlo unicamente, per esempio se volessi eseguire il task **dependenceTwo** potrei farlo semplicemente con il comando:

```
$ gradle depTw
```

considerando i task creati precedentemente notiamo che il task è univocamente identificato.

2.2.2 Escludere i task

È possibile escludere un task di una build, aggiungendo come argomento il task da escludere preceduto da -x:

```
$ gradle <task_da_eseguire> -x <task_da_escludere>
```

questo viene usato al fine di eliminare un task inutile per lo scopo della build che abbiamo intenzione di eseguire. Riprendendo l'output di

```
$ gradle mainTask
```

notiamo che vengono eseguiti tutti i tasks definiti nella build.gradle (a pagina 2), se volessimo escludere dependenceOne dalla build allora dovremo eseguire:

```
$ gradle mainTask -x dependenceOne
```

Otteniamo in questo modo in output:

```
> Task :dependenceTwo
First Two
Last Two

> Task :mainTask
First MainTask
Last MainTask


BUILD SUCCESSFUL in 0s
2 actionable tasks: 2 executed
```

Possiamo notare che non verrà eseguito nemmeno il task dependenceZero perchè è una dipendenza del task dependenceOne.

2.2.3 Selezionare la build da eseguire

Consideriamo che esista in una subdirectory chiamata subdir una build chiamata subbild.gradle, partendo dalla directory source è possibile eseguire questa build eseguendo il comando:

```
$ gradle -b subdir/subbuild.gradle <task_da_eseguire>
```

Questa particolare funzione serve soprattutto ai progetti multi-builds, in cui è necessario avere a disposizione più di una build di riferimento.

2.2.4 Forzare l'esecuzione di un task

A causa della Gradle cache è possibile che un task o più di uno non vengano eseguiti perchè marcati come UP-TO-DATE (anche se dalla versione Gradle 4.0 non viene più mostrato in output), in questo caso è possibile forzarne l'esecuzione con:

```
$ gradle --rerun-tasks <tasks_da_eseguire>
```

2.2.5 Continuare la build quando si verifica un errore

Se durante una build un task fallisce, Gradle di default interromperà l'esecuzione e farà fallire anche la build. Questo permette alla build di completare velocemente, ma il fallimento anticipato della build potrebbe nascondere altri problemi che possono presentarsi in altri tasks. A volte è quindi necessario imporre ad una build di gradle di continuare nonostante il fallimento di uno o più tasks, questo è possibile usando l'opzione `--continue`:

```
$ gradle <tasks_da_eseguire> --continue
```

In questo modo verranno eseguiti tutti i tasks e solo al completamento della build saranno resi noti gli errori.

2.2.6 Ottenere informazioni generali

Per visualizzare una lista dei principali tasks eseguibili è possibile eseguire il task

```
$ gradle tasks
```

l'output di questa build sarà:

```
> Task :tasks

------------------------------------------------------------
All tasks runnable from root project - Example of Task
------------------------------------------------------------

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

To see all tasks and more detail, run gradle tasks --all

To see more detail about a task, run gradle help --task <task>
```

```
BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

come dice l'output, per visualizzare la lista di tutti i tasks eseguibili nel nostro project è necessario eseguire la build del task

```
$ gradle tasks --all
```

noteremo che in questo caso verranno visualizzati anche i tasks che abbiamo precedentemente creato (dependenceZero, dependenceOne, dependenceTwo, mainTask con le relative descrizioni):

```
> Task :tasks

------------------------------------------------------------
All tasks runnable from root project - Example of Task
------------------------------------------------------------

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

Other tasks
-----------
dependenceOne - Build Dependence One
dependenceTwo - Build Dependence Two
dependenceZero - Build Dependence Zero
mainTask - Build Main Task


BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Se invece vogliamo informazioni più specifiche riguardo un singolo task la build da fare è

```
$ gradle help --task <nome_del_task>
```

per esempio eseguiamo:

```
$ gradle help --task mainTask
```

otterremo una descrizione specifica del task mainTask:

```
> Task :help
Detailed task information for mainTask

Path
     :mainTask

Type
     Task (org.gradle.api.Task)

Description
     Build Main Task

Group
     -
```

2.2.7 Build scan

Una funzione molto interessante di Gradle è la possibilità di poter pubblicare la propria build, questo permette di avere un report completo e condivisibile. Per utilizzare questa funzionalità è necessario aggiungere alla build di un task l'opzione `--scan`:

```
$ gradle <task_da_eseguire> --scan
```

Al completamento della build del task verrà richiesto di accettare i termini di uso di questo servizio. Una volta accettati verrà fornito un link alla build pubblicata in cui sarà richiesta una mail di riferimento per confermare la pubblicazione della build. Prendendo come esempio eseguiamo il comando:

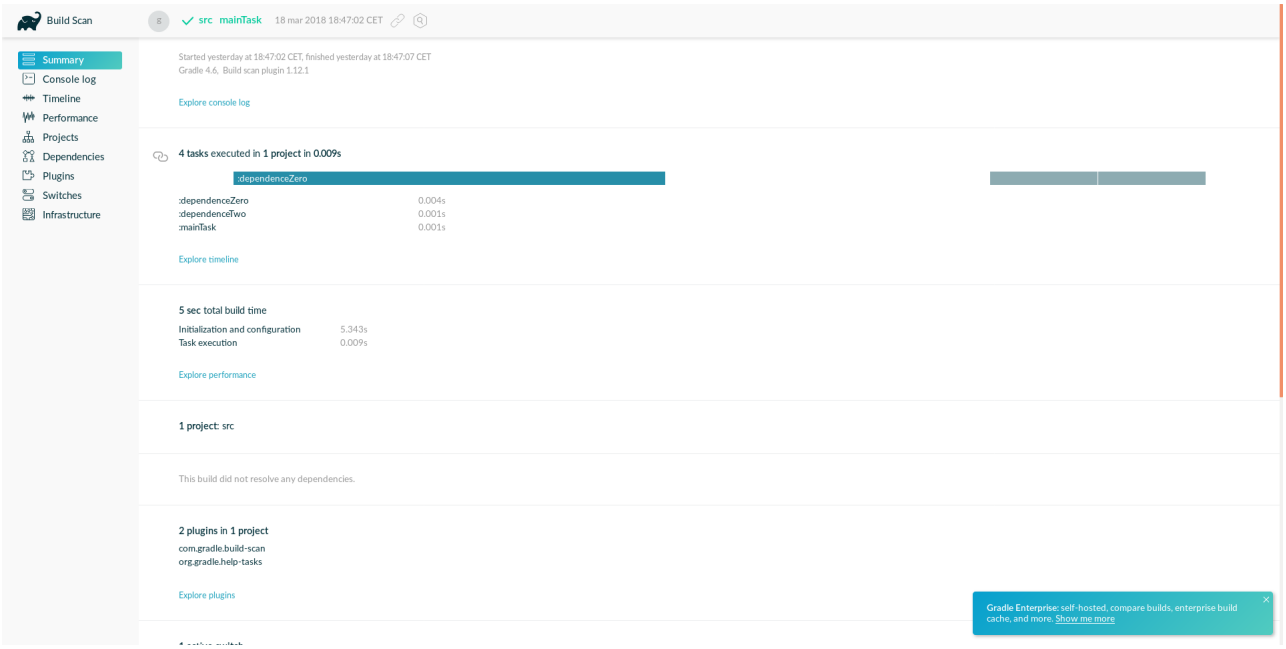
```
$ gradle mainTask --scan
```

l'output risultante sarà:

```
> Task :dependenceZero
First Zero
Last Zero

> Task :dependenceOne
First One
Last One

> Task :dependenceTwo
First Two
Last Two

> Task :mainTask
First MainTask
Last MainTask


BUILD SUCCESSFUL in 1s
4 actionable tasks: 4 executed

Publishing a build scan to scans.gradle.com requires accepting the Gradle Terms of Service
defined at https://gradle.com/terms-of-service. Do you accept these terms? [yes, no]
yes
Gradle Terms of Service accepted.
```

```
Publishing build scan...
https://scans.gradle.com/s/qcc4vkuegibig
```

cliccando sul sito e seguendo le indicazioni, il risultato finale sarà:



2.3 Tutorial

Il tutorial di seguito è possibile anche trovarlo al link:
GITHUB.COM/WABRI/ATTSW_EXAM/BLOB/MASTER/GRADLE.EXAMPLE/FIRST/.

1. Creare una cartella gradle.example/first
2. All'interno della nuova cartella creare il file build.gradle contenente:

```
description = 'Example of Task'

task dependenceZero {
        description = 'Build Dependence Zero'
        doFirst {
                println 'First Zero'
        }
        doLast {
                println 'Last Zero'
        }
}

task dependenceOne(dependsOn: [dependenceZero]) {
        description = 'Build Dependence One'
        doFirst {
                println 'First One'
        }
        doLast {
                println 'Last One'
        }
}
```

```
task dependenceTwo {
        description = 'Build Dependence Two'
        doFirst {
                println 'First Two'
        }
        doLast {
                println 'Last Two'
        }
}

task mainTask(dependsOn: [dependenceOne, dependenceTwo]) {
        description = 'Build Main Task'
        doFirst {
                println 'First MainTask'
        }
        doLast {
                println 'Last MainTask'
        }
}
```

3. Eseguire la build:

```
$ gradle mainTask
```

4. Eseguire la build multi-tasks:

```
$ gradle dependenceZero dependenceTwo
```

5. Eseguire la build usando una abbreviazione:

```
$ gradle maTa
```

6. Eseguire la build precedente escludendo il task dependenceOne:

```
$ gradle mainTask -x dependenceOne
```

7. Creare una build differente in una subdirectory rispetto alla posizione iniziale:

```
description = 'Sub directory'

task subMainTask {
        description = 'Sub Build Main Task'
        doFirst {
                println 'First MainTask'
        }
        doLast {
                println 'Last MainTask'
        }
}
```

8. Eseguire il task subMainTask della build appena creata partendo dalla directory root:

```
$ gradle -b subdir/build.gradle suMT
```

9. Forzare l'esecuzione di un task marcato come UP-TO-DATE:

```
$ gradle --rerun-tasks maTa
```

10. Ottenere la lista dei tasks di default:

```
$ gradle tasks
```

11. Ottenere la lista di tutti i tasks:

```
$ gradle tasks --all
```

12. Eseguire il comando:

```
$ gradle help --task mainTask
```

13. Pubblicare la build del task mainTask:

```
$ gradle mainTask --scan
```

3 Project, Wrapper & Deamon

Potete trovare un tutorial guidato a questo link:
github.com/Wabri/ATTSW_Exam/tree/master/gradle.example/second

3.1 Creazione di un nuovo progetto Gradle

Creare un progetto Gradle è molto semplice sfruttando direttamente il task di default init. Creiamo una cartella in cui eseguiremo da terminale il comando:

```
$ gradle init
```

Notiamo che nella directory sono stati creati 4 file e 1 cartella:

- `build.gradle` e `settings.gradle`: sono i file di configurazione
- `gradlew`, `gradlew.bat` e la directory `gradle`: sono i file corrispondenti al wrapper

Abbiamo già trattato il file di configurazione build.gradle (pagina 1). Il file settings.gradle è anch'esso uno script Groovy dove vengono indicati quali progetti parteciperanno alla build. Questo task crea un progetto di default, ma è possibile essere più specifici in quanto il task init con l'opzione `--type` può assumere come argomento una tipologia di progetto. Assumiamo che il progetto che vogliamo creare sia una applicazione java, il comando da eseguire sarà:

```
$ gradle init --type java-application
```

Rispetto al comando precedente verrà creata una directory `src` in più che sarà specifica per il linguaggio java:

- `main/java/App.java`: che è un file preimpostato il cui contenutò sarà un semplice main che stamperà il consueto Hello World!
- `test/java/AppTest.java`: in cui viene testato il metodo contenuto nella classe App.java

Spieghiamo a questo punto i file del wrapper precedentemente indicati.

3.2 Wrapper

Molto spesso prima di poter usufruire di uno strumento di sviluppo è necessaria una installazione. Gradle mette a disposizione uno script che permette di usare tutte le sue funzionalità evitando di installare Gradle su tutte le macchine di sviluppo, questo strumento viene chiamato Gradle Wrapper. Se in un progetto è stato settato il Wrapper è possibile eseguire le builds sostituendo il comando `gradle` con il comando `./gradlew` (se si lavora con sistema operativo windows il comando è `./gradlew.bat`). Se più persone lavorano a un progetto può capitare che ci siano differenze tra le versioni di uno strumento, nel caso del wrapper non è possibile sbagliare perchè la sua versione è insita durante la sua creazione o durante il suo upgrade (o downgrade). Quindi è sempre consigliato l'uso del wrapper e lasciare tutte le sue informazioni anche nella repository del VCS usato. Per creare il wrapper in un progetto è necessario eseguire il comando:

```
$ gradle wrapper
```

Il comando creerà 4 files:

- **gradlew**: script shell per eseguire il wrapper in sistemi Unix
- **gradlew.bat**: file batch per eseguire il wrapper in sistemi Windows
- **gradle/wrapper/gradle-wrapper.properties**: file di configurazione delle proprietà del Wrapper
- **gradle/wrapper/gradle-wrapper.jar**: contiene il codice per scaricare le distribuzioni Gradle

Questi sono i file di cui ha bisogno il wrapper per poter essere usato. Ovviamente Quando il wrapper viene creato la sua versione sarà quella di Gradle attualmente installato sulla macchina, è possibile specificare in vari modi quale versione usare:

1. eseguire il solito comando con l'aggiunta dell'argomento `--gradle-version` con il numero della versione:

   ```
   $ gradle wrapper --gradle-version <numero_versione>
   ```

   oppure se già inserito il wrapper:

   ```
   $ ./gradlew wrapper --gradle-version <numero_versione>
   ```

   per esempio se volessimo passare dalla versione attuale alla versione 2.0 basterà eseguire il comando:

   ```
   $ ./gradlew wrapper --gradle-version 2.0
   ```

   dopo aver eseguito il download della versione, l'output corrispondente sarà:

   ```
   ------------------------------------------------------------
   Gradle 2.0
   ------------------------------------------------------------
   Build time:   2014-07-01 07:45:34 UTC
   Build number: none
   Revision:     b6ead6fa452dfdadec484059191eb641d817226c
   Groovy:       2.3.3
   Ant:          Apache Ant(TM) version 1.9.3 compiled on December 23 2013
   JVM:          1.8.0_161 (Oracle Corporation 25.161-b12)
   OS:           Linux 4.13.0-37-generic amd64
   ```

   (il task wrapper non esisteva fino alla versione 3.0, eseguire quindi questo task con versioni precedenti risulterebbe in un fallimento della build).

2. modificare direttamente il file gradle-wrapper.properties in cui ci sarà:

   ```
   distributionUrl=https\://services.gradle.org/distributions/gradle-4.4.1-bin.zip
   ```

   che è il tipo di distribuzione usata attualmente dal wrapper. Per passare alla versione 2.0 possiamo modificare questa riga con:

   ```
   distributionUrl=https\://services.gradle.org/distributions/gradle-2.0-bin.zip
   ```

   eseguendo poi un qualsiasi comando la versione sarà aggiornata.

3. infine è possibile specificarlo direttamente modificando il file build.gradle aggiungendo un task chiamato wrapper che estenderà la classe Wrapper:

   ```
   task wrapper(type: Wrapper) {
       gradleVersion = '2.0'
   }
   ```

   in questo modo viene effettivamente fatto un override del task wrapper. A questo punto per aggiornare alla versione indicata basterà eseguire il comando:

   ```
   $ ./gradlew wrapper
   ```

In ogni caso possiamo visualizzare la versione usata dal wrapper con il comando:

```
$ ./gradlew --version
```

Il wrapper è altamente configurabile sia come proprietà sia come versionamento. Per esempio riprendendo il punto 3 della lista precedente, se non si vuole specificare tutte le volte il tipo di distribuzione voluta è possibile inserire un altro campo all'interno del task wrapper `distributionType` a cui assegneremo `Wrapper.DistributionType.ALL`:

```
task wrapper(type: Wrapper) {
    gradleVersion = '4.6'
    distributionType = Wrapper.DistributionType.ALL
}
```

In questo modo verrà scaricata tutta la distribuzione e non solo i file binari.

3.3 Deamon

Gradle viene eseguito sulla Java virtual machine (JVM) e usa librerie di supporto che necessitano una inizializzazione, entrambi allungano il processo di esecuzione iniziale della build allungando i tempi di attesa. Questo problema viene risolto usando un Daemon che mantiene le informazioni della build in background velocizzando le esecuzioni successive alla prima, mantenendo le informazioni in memoria pronte all'uso. Una build di gradle è possibile eseguirla con o senza il daemon, indicando nelle proprietà di Gradle quando usarlo e se usarlo. Il daemon permette non solo di evitare l'avviamento della JVM, ma ha anche un sistema di cache in cui sono immagazzinati: struttura del progetto, files, tasks e molto altro. Ovviamente se il progetto viene eseguito in contenitori temporanei, tipo un server di continuous integration (CI), è sconsigliato l'uso del daemon in quanto questi non riutilizzano lo stesso processo ma ne creano uno nuovo, quindi l'uso del deamon non solo è inutile ma ridurrà anche le prestazioni dato che dovrà rieseguire il daemon di gradle e ricreare la cache. Per controllare i processi daemon attivi sulla macchina basterà eseguire il comando:

```
$ gradle --status
```

che restituirà il pid, lo stato e la versione di gradle usata dal deamon. Ad esempio se abbiamo un progetto in cui è usata la versione di gradle 3.0 avremo un risultato di questo tipo:

```
   PID STATUS   INFO
 16463 IDLE     3.0
```

Come già detto è possibile disabilitare il processo deamon, per farlo è necessario modificare il campo `org.gradle.deamon` con l'assegnazione a false. Le proprietà si trovano seguendo il percorso `$HOME/.gradle/gradle.properties`, se il file non esiste basterà crearlo, a questo punto inseriamo in coda al file:

```
org.gradle.deamon=false
```

Ora tutte le volte che eseguiamo una build non verrà riattivato il deamon. Esistono 2 opzioni che permettono di indicare se usare o no il daemon specificatamente su una build:

- `--no-daemon`, che indica di non usare il daemon per questa build
- `--daemon`, che invece indica di usare il daemon per questa build

Spesso viene usato questo metodo che risulta essere molto più chiaro soprattutto se la build viene condivisa. Se volessimo stoppare un daemon attivo è possibile farlo con l'opzione `--stop` seguito dal PID del daemon da stoppare, per esempio se volessimo stoppare un daemon con PID=12812 eseguiremo il comando:

```
$ gradle --stop 12812
```

Il risultato sarà:

```
Stopping Daemon(s)
1 Daemon stopped
```

Se abbiamo più daemon attivi e volessimo stopparli tutti è possibile eseguire il comando:

```
$ gradle --stop
```

Evitando quindi di eseguire una cascata di comandi tutti uguali con PID diverso. Questo stopperà tutti i daemon attivi aventi la stessa versione di gradle usato per eseguire il comando.

3.4 Tutorial

Il tutorial di seguito è possibile anche trovarlo al link:
GITHUB.COM/WABRI/ATTSW_EXAM/BLOB/MASTER/GRADLE.EXAMPLE/SECOND/.

1. Creare una cartella gradle.example/second
2. Eseguire la build:

   ```
   $ gradle init --type java-application
   ```

3. Usare il wrapper per controllare la versione attualmente in uso dal progetto:

   ```
   ./gradlew --version
   ```

4. Cambiare la versione del wrapper alla 2.0:

   ```
   $ ./gradlew wrapper --gradle-version 2.0
   ```

5. Controllare se la versione del wrapper è stata cambiata:

   ```
   $ ./gradlew --version
   ```

6. Fare l'upgrade alla 3.0 del wrapper usando le properties, modificando il campo `distribuitionUrl`:

   ```
   distributionUrl=https\://services.gradle.org/distributions/gradle-3.0-bin.zip
   ```

7. Controllare se la versione del wrapper è stata cambiata
8. Modificare il build.gradle per impostare la versione del wrapper alla 4.6:

   ```
   task wrapper(type: Wrapper) {
       gradleVersion = '4.6'
   }
   ```

9. La versione non sarà modificata fintanto che non sarà eseguito il task `wrapper`, eseguire quindi la build:

   ```
   $ ./gradlew wrapper
   ```

10. Controllare se la versione del wrapper è stata cambiata
11. Impostare All come tipo di distribuzione da usare per il wrapper, aggiungere quindi il campo `distributionType`:

    ```
    task  wrapper(type: Wrapper) {
        gradleVersion ='4.6'
        distributionType = Wrapper.DistributionType.ALL
    }
    ```

12. Eseguire la build del task `wrapper` per aggiornare la distribuzione usata dal wrapper
13. Visualizzare lo stato attuale dei daemon attualmente in esecuzione:

    ```
    $ ./gradlew --status
    ```

14. Eseguire il task `test` usando il daemon:

    ```
    $ ./gradlew --daemon test
    ```

    (Notare che se viene rieseguita questa build il tempo di esecuzione risulta essere nullo)

15. Eseguire lo stesso task precedente senza usare il daemon:

    ```
    $ ./gradlew --no-daemon test
    ```

    (Notare che non usando il daemon il tempo di esecuzione non sarà nullo)

16. Stoppare il daemon attivo attualmente usato:

    ```
    $ ./gradlew --stop <PID>
    ```

4 Dependency Management

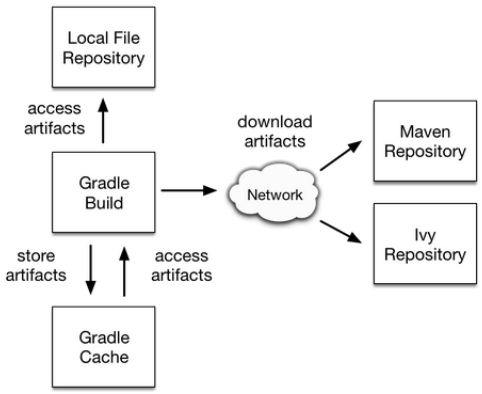
Una delle parti più importanti di uno strumento di questo tipo è la gestione delle dipendenze che si divide in 2 parti: incoming files e outgoing files. Gradle ha bisogno di conoscere di cosa il nostro progetto ha bisogno per poter essere compilato ed eseguito le così dette dipendenze (dipendencies) che in questo caso sono gli incoming files. Gli outgoing files sono invece tutto ciò che il progetto produce, definite pubblicazioni (pubblications). Le dipendenze vengono specificate in forma di modules, è quindi necessario indicare dove si trovano questi modules in modo tale che Gradle li possa scaricare e impostare per il progetto che stiamo sviluppando. La posizione dove è possibile trovare i modules è definita repository, è necessario quindi dichiarare le repositories usate per le dipendenze volute. Ci sono 2 tipi di repository:

1. esterne, in questo caso la repository si trova in un server online adibito alla raccolta di modules
2. interne, la repository è una cartella locale al progetto

Per quanto riguarda quelle esterne Gradle si occuperà di scaricarle. E' possibile che alcune dipendenze vengano usate in più progetti, gradle mantiene quindi una cache locale, chiamata dependency cache, in cui salverà i modules già scaricati in modo da evitare di effettuare il download ad ogni build. Possiamo quindi immaginare che il ciclo di risoluzione delle dipendenze esterne sarà questo:

1. ricerca delle dipendenze esterne nella cache locale
2. se non si trovano nella cache locale si controlla se esistono nelle repository specificate
3. se sono state trovate, vengono scaricate e inserite nella cache locale

Nell'immagine è possibile vedere il percorso specifico che effettua la dependency resolution di Gradle:



Analizziamo ora il caso di un progetto java.

4.1 Java Dependency Management

Prima di tutto è necessario indicare in che linguaggio il nostro progetto viene rilasciato (consideriamo d'ora in poi solo il caso di Java), per farlo aggiungiamo in testa al file build.gradle:

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

A questo punto per poter usufruire di una dipendenza è necessario specificare da dove Gradle deve andare a prenderla, dobbiamo quindi indicare il repository remoto di riferimento. Se per esempio vogliamo che il nostro repository di riferimento sia Maven allora dobbiamo aggiungere al build.gradle:

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

In questo modo tutte le dipendenze che andremo a indicare successivamente saranno riferimenti alle pubblicazioni su Maven Central. La dichiarazione delle dipendenze deve essere inserita nel tag `dependencies` nel build.gradle file. Per esempio vogliamo avere junit 4.12 come dipendenza al nostro progetto Gradle allora dobbiamo aggiungere:

```
dependencies {
    testImplementation group: 'junit', name: 'junit', version: '4.12'
}
```

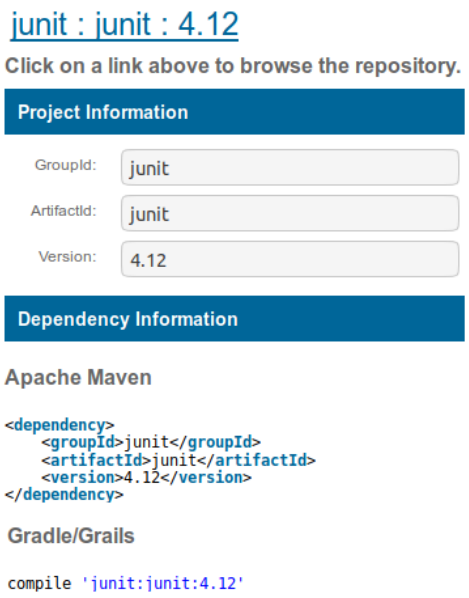
Osserviamo che nella dichiarazione ci sono 4 diversi indicatori:

- `testImplementation` indica la configurazione della dipendenza, in questo caso sarà importata durante l'implementazione dei test;
- `group, name, version` corrispondono rispettivamente al groupId (nome del team o della società che ha sviluppato il modulo), artifactId (nome effettivo del modulo) e al version (versione del modulo) definiti su Maven.

Esiste un modo molto più diretto per indicare una dipendenza:

```
dependencies {
    testImplementation 'junit:junit:4.12'
}
```

Ha lo stesso significato precedente ma ha una forma più compatta, forma che adotta anche la documentazione Maven:



Spesso in un progetto non è necessario specificare il numero dell'aggiornamento della versione, ma basta la versione più aggiornata, questo è possibile indicarlo a gradle con un `+` subito dopo la versione voluta:

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

In questo modo quando eseguiremo il task `dependencies` gradle si assicurerà che la versione in uso di quella dipendenza specifica sia l'ultima rilasciata. Possiamo notare ora la differenza sostanziale della configurazione delle dipendenze tra il pom.xml di Maven e il build.gradle di Gradle. A questo punto per scaricare le dipendenze si deve eseguire il comando `dependencies` il cui output restituirà una lista di tutte le configurazioni con le relative dipendenze associate (non mostro tutto l'output perchè è molto corposo):

```
> Task :dependencies

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

[...]

compile - Dependencies for source set 'main' (deprecated, use 'implementation ' instead).
No dependencies

[...]

default - Configuration for default artifacts.
No dependencies

[...]

runtime - Runtime dependencies for source set 'main' (deprecated, use 'runtimeOnly ' instead).
No dependencies

[...]

testCompileClasspath - Compile classpath for source set 'test'.
\--- junit:junit:4.+ -> 4.12
     \--- org.hamcrest:hamcrest-core:1.3

testCompileOnly - Compile only dependencies for source set 'test'.
No dependencies

testImplementation - Implementation only dependencies for source set 'test'. (n)
\--- junit:junit:4.+ (n)

testRuntime - Runtime dependencies for source set 'test' (deprecated, use 'testRuntimeOnly ' instead).
No dependencies

testRuntimeClasspath - Runtime classpath of source set 'test'.
\--- junit:junit:4.+ -> 4.12
     \--- org.hamcrest:hamcrest-core:1.3

[...]


BUILD SUCCESSFUL in 0s
1 actionable task: 1 executed
```

Come è possibile notare dall'output esistono molte configurazioni associabili a una dipendenza: compile, default, runtime, testImplementation, e così via, ognuno dei ha uno scopo ben preciso. Possiamo dividere le configurazioni in 3 scopi principali:

1. implementation
2. api
3. testImplementation