Moteur de production (gradle)

L'objectif est d'automatiser les actions pour produire un logiciel, gérer les dépendances, les détecter et adapter la production du logiciel à la platforme. Gradle est un système qui permet d'automatiser tout ça.

Cela permet donc de rendre un projet indépendant de l'environement de développement de la personne qui écrit le code, car Gradle va automatiquement gérer toutes les dépendences nécessaires.

Historique des moteurs de production

Année	Programme	Langage	Avantage
1977	make	C et autres	
2000	Apache Ant	Java	
2004	Maven	Java	Plus complet que Apache Ant
2008	Gradle	Java (et autres comme Kotlin)	Plus complet que Maven

Que font Gradle et Maven

- La compilation
- Le packaging (jar, war, etc)
- Gestion des dépendences
- Génération de la documentation
- Gestionnaire de sources
- Accès au depot des gestinnaires des dépendences
- Le déploiement en différents environements (test, développement, production, etc)

Qu’est ce que Gradle

Gradle est un moteur de production tournant sur la JVM (Java Virtual Machine). Un moteur de production sert à automatiser les étapes de construction d’un projet. Un moteur de production est nécessaire car le faire à la main serait source d’erreur, très lent et complexe. Gradle au fur et à mesure du temps est devenu très important dans l’écosystème Java.

Pour aller plus loin que les informations du cours, on peut aller consulter la documentation officielle de Gradle.

Gestion simplifiée des dépendences

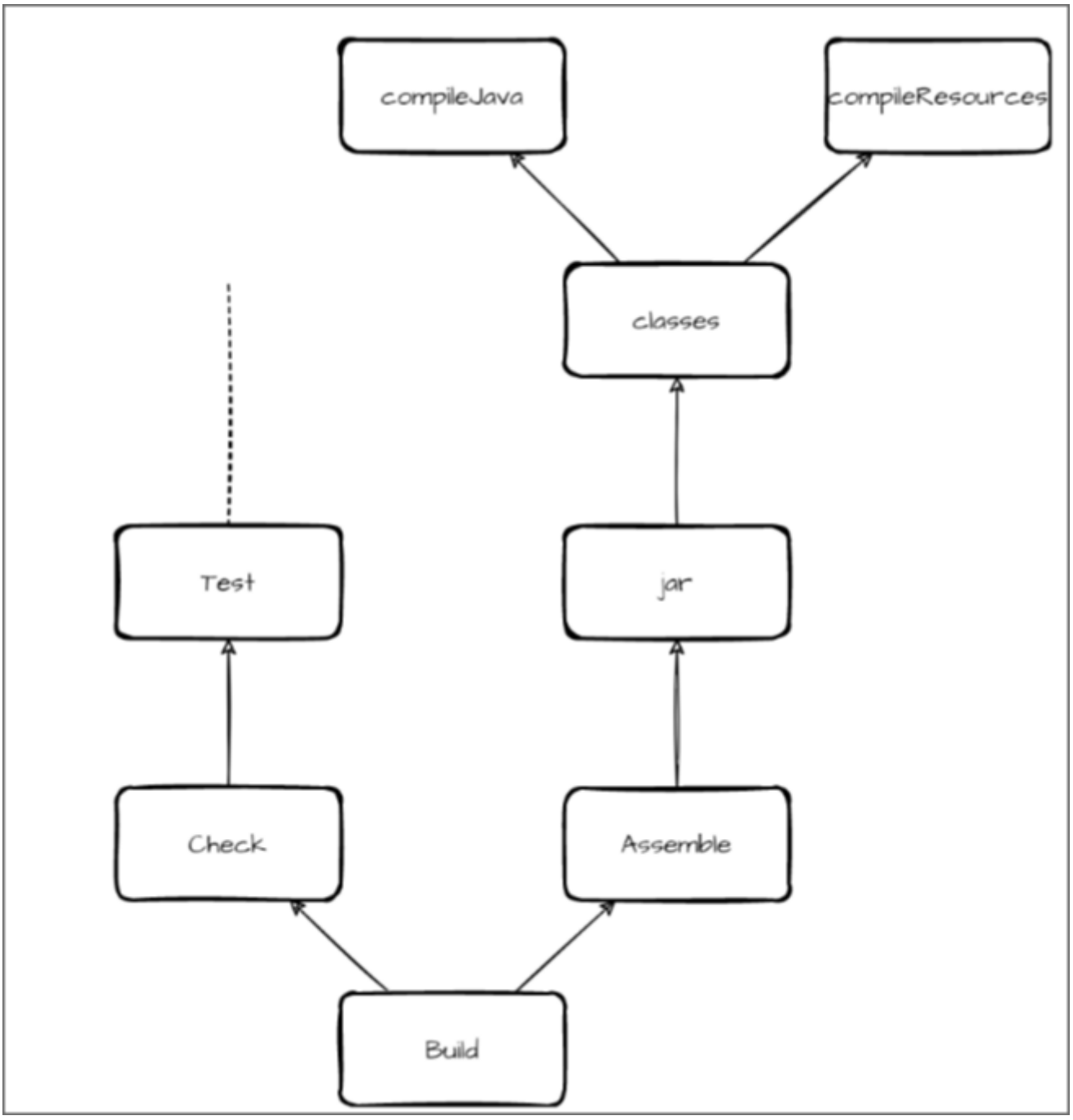
Gradle permet par exemple une gestion simplifiée des dépendances qui permet de simplement installer beaucoup de dépendences depuis les répertoires Maven (un autre moteur de production).

On peut en savoir plus en consultant le site MVN repository.

Les tâches Gradle

Gradle fonctionne sur base de tâches. Chaque tâche permet d’obtenir des sorites (fichiers ou répertoires mis à jour) à partir d’entrées (fichiers, répertoires, configuration, etc).

Les tâches peuvent en inclure d’autres, par exemple la tâche `build` d’un projet Java standard inclus la tache `check` qui va effectuer les tests, et la tâche `assemble` qui va générer l’exécutable.



Système de plugin

De base Gradle est très simple et peu utilisable en pratique. Pour rendre Gradle vraiment utile il faut y intégrer des plugins, comme le plugin `java` par exemple qui permet de compiler des applications Java. Ce système rends donc Gradle très flexible car il peut être beaucoup étendu et n'importe qui peut en créer des plugins

Processus d'une constructoin de projet Gradle

1. L'**initiation** met en place l'environement et détermine les projets qui le compose

2. La **configuration** construit et configure le DAG des tâches (l’arbre qui définit les liens entre les tâches) et définit les tâches à exécuter pour accomplir la tache initiale
3. L’**exécution** exécute les tâches identifiées

La seule phase dans laquelle on intervient est la phase de configuration.

Avantage de Gradle

- Plus rapide
- Plus flexible (pas limité à Java, contrairement à Maven)
- Fichier de config et plus simple
- Gestion des dépendances plus complète
- Indépendant
- Multilangage

Configuration de Gradle

- `settings.gradle` qui est à la racine du projet et contient le nom du projet ainsi que l'ensemble des sous-projets
- `build.gradle` est dans chaque (sous-)projet et contient l'ensemble des éléments utile pour compiler le projet (version de Java, dépendance, , tests, PMD, etc)

Liste des librairies

On peut rechercher les libraries sur le site *MVN Repository*.

Qu'est ce qu'une librairie

On ne s'amuse pas à réinventer la roue dès que l'on veut faire un programme. Du coup on va réutiliser des composants qui ont déjà été fait par d'autres personnes. Une librarie c'est exactement ça, c'est une collection d'outils qui permettent de programmer plus facilement et plus rapidement sans devoir réinveter les même choses en boucle.

Identification d'une dépendence/librarie

Il y a 3 éléments qui identifie une dépendence :

- Le *group id* qui défini le groupe des librairies
- Le *artifact id* qui défini la dépendence dans le groupe
- La *version* de la dépendence

Il est important de vérifier la version des dépendences qui ne sont pas toujours compatibles entre elles et il faut éviter de prendre des versions trop vieilles.

Installation de Gradle

Suivez les instructions sur le site officiel de Gradle.

Création d'un projet et autopsie

Pour créer un projet vous pouvez simplement aller créer un dossier vide, puis entrer dedans avec votre terminal et lancer la commande : `gradle init`

Après quoi Gradle va vous poser certaines questions, pour l'exemple voici ce que vous devez répondre :

- Project type ? → 2 (application)
- Implementation language ? → 3 (java)
- Build script DSL ? 2 (groovy)
- Test framework ? 4 (JUnit Jupiter)

Pour toutes les autres questions, faites juste ENTER pour choisir l'option par défault

Autopsie des tâches

Une fois cela fait nous pouvons ensuite avoir une liste des tâches possibles. Pour cela il suffit d'exécuter `./gradlew tasks`.

Ainsi on voit que si on fait `./gradlew run` ça va lancer le projet (par défault ça va afficher Hello World) ou encore si on fait `./gradlew check` ça va effectuer tous les tests.

Autopsie de la structure des fichiers

```
.
├── app
│   ├── build.gradle
│   └── src
│       ├── main
│       │   ├── java
│       │   │   └── hello
│       │   │       └── App.java
│       │   └── resources
│       └── test
│           ├── java
│           │   └── hello
│           │       └── AppTest.java
│           └── resources
├── .gitattributes
├── .gitignore
├── .gradle
│   └── file-system.probe
├── gradle
│   └── wrapper
│       ├── gradle-wrapper.jar
│       └── gradle-wrapper.properties
├── gradlew
├── gradlew.bat
└── settings.gradle

13 directories, 11 files
```

- `app/` est le dossier qui contient tout ce qui est relatif à votre projet (en particulier le code source)
  - `app/build.gradle` contient les informations sur le build du projet
  - `app/src/` contient le code source de l'application
    - `app/src/main/` et `app/src/test/` contiennent les deux partie du projet (main pour le code principal et test pour les test unitaires), ces dossiers sont ensuite divisé par langage de programmation (ici Java uniquement), puis par package. Par exemple le App.java se situe dans le package Java test.gradle.
- `.gradle/` est le cache du projet
- `gradle/` contient le *wrapper* de Gradle, c'est grâce à cela que va être automatiquement téléchargée la version Gradle appropriée si on viendrait à essayer de lancer le projet sans avoir Gradle installé ou sans avoir la bonne version de Gradle installée pour le projet.

- `gradlew` et `gradlew.bat` sont les scripts (un pour Linux/macOS et un pour Windows) qui vont exécuter le wrapper gradle pour télécharger la bonne version automatiquement. Il est recommander de les utiliser à la place de la commande `gradle` même si les deux fonctionnent en vérité.
- `settings.gradle` défini quels sont les projets qui font partie du dossier ainsi que d'éventuels autres plugins.
- `.gitignore` défini les fichiers à ignorer dans git
- `.gitattributes` défini les préférences du projet pour git

Autopsie des fichiers de configuration

Vous pouvez maintenant aller ouvrir les fichiers `app/build.gradle` et `settings.gradle` pour découvrir ce qu'il se passe à l'intérieur.

Build.gradle

Le fichier `build.gradle` indique :

- La liste des plugins (ici, uniquement "application")
- La sources des dépendances
- Liste des dépendances
- La définition de la version de java pour paramétrer le plugin "application"
- La définition de la classe principale de l'application (ici test.gradle.App)
- La configuration de la tache de test

Settings.gradle

Le fichier `settings.gradle` indique :

- Une liste de plugins, ici il y a uniquement un plugin servant à télécharger automatiquement des versions du JDK
- Le nom du projet global
- La liste des sous projets (ici il n'y en a qu'un c'est `app`)

Intégration avec Eclipse

Importer un projet Gradle

On peut par exemple importer notre projet Gradle créé plus tot dans Eclipse. Pour cela on peut aller dans Eclipse > File > Import > Gradle > Existing Gradle Project. Ensuite on peut sélectioner le dossier de notre projet Gradle. Enfin dans les “Import Options”, on peut cocher la case “Override workspace settings” et définir le Java home à la localisation du JDK aproprié. Une fois cela fait on peut cliquer sur “Finish” pour importer le projet. Pour être sur que tout a bien charger on peut faire un clic droit sur le projet puis aller dans Gradle puis dans “Refresh Gradle project”

Lancer une tâche Gradle

Pour lancer une tache ou voir la liste des tâches on peut aller dans l’onglet “Gradle tasks”, si l’onglet n’est pas affiché on peut l’afficher en faisant Window > Show view > Other > Gradle Tasks.

Ensuite il suffit de cliquer sur les tâches que l’on veut exécuter, on peut ensuite voir son résultat dans l’onglet Console.

A savoir qu’Eclipse génère tout un tas de fichiers qui ne sont pas intéressants à ajouter dans Git, il vaut donc mieux les ajouter au `.gitignore`

```
# Backupfiles from mergetools #
*.orig

# Java Class Files #
*.class

# Package Files #
*.jar
*.war
*.ear
*.zip

# Eclipse #
.project
.settings
.classpath
bin
```

Ajouts de plugins et dépendences supplémentaires

Il y a certain plugins Gradle que l'on est obligé d'avoir pour l'activité intégrative. Il faut en installer 2 :

- PMD qui fournit des rapports sur les potentielles faiblesses de notre code
- JaCoCo qui fournit un rapport sur le code coverage (la couverture de code couverte par les tests)

PMD (dans Gradle)

Pour installer le plugin PMD dans Gradle, on va tout d'abord l'ajouter dans la liste des plugins du `build.gradle` du dossier `app`.

```
plugins {
    id 'application'
    id 'pmd' // ← Ajout de cette ligne
}
```

Ensuite on peut ajouter le fichier ruleset de pmd présent sur l'espace de cours dans le projet. Une fois cela fait, on peut modifier de nouveau le build.gradle pour y ajouter la configuration de PMD :

```
pmd {
    ruleSets = []
    ruleSetFiles = files("pmd-ruleset.xml") // ← mettre le chemin de fichier relatif vers le fichier PMD ici
    maxFailures = 15 // Défini la limite acceptable d'erreurs PMD avant de stopper le build
}
```

Une fois cela fait PMD va afficher des erreurs dans la console lors du build si l'une de ses règles n'est pas respectée. Et au delà 5 erreurs, PMD va faire stopper le build.

PMD génère aussi un rapport dans le dossier `app/build/reports/pmd/`.

PMD dans Eclipse

Pour avoir les erreurs affichées directement dans l'IDE quand on écrit le code on peut activer le plugin Eclipse comme en B1.

Installation

Pour cela on peut aller dans Help > Eclipse Marketplace > eclipse-pmd > Install.

Ensuite il faut accepter la license et autoriser toutes les sources du plugin quand demandé.

Une fois le plugin installé, il va vous demander de redémarrer Eclipse.

Configuration

Ensuite pour l'activer sur notre projet, on peut aller dans les propriétés du projet (clic droit sur le projet > Properties) puis aller dans l'onglet "PMD" et choisir l'option "Enable PMD for this project".

Une fois cela fait on peut cliquer sur Add > Project > Browse puis sélectionner le fichier ruleset et cliquer sur Finish.

Une fois cela fait on a maintenant les alertes directement dans le code.

Jacoco

Pareil que PMD on doit d'abord ajouter Jacoco dans la liste des plugins :

```
plugins {
    id 'application'
    id 'pmd'
    id 'jacoco' // ← Ajout de cette ligne
}
```

Ensuite on peut le configurer, ici on va le configurer comme dit dans sa documentation en le faisant exécuter à chaque test (un report Jacoco sera ainsi généré à chaque fois que les tests seront effectués) :

```
test {
    finalizedBy jacocoTestReport // report is always generated after tests run
}


jacocoTestReport {
    dependsOn test // tests are required to run before generating the report
}
```

Une fois cela terminé, on peut maintenant lancer la tâche `check` ou `test` et un report jacoco devrait être généré dans le dossier `app/build/reports/jacoco`

Gestion des dépendences

Chaque dépendence dans Gradle a une certaine portée, ainsi certaines dépendencent ont une portée uniquement sur les tests unitaires et d'autres sont nécessaires pour le code principal.

Par exemple si on veut installer la dépdance `Apache Commons Text`. On peut aller rechercher le nom de la dépendance sur le site MVN repository. Ensuite on peut cliquer sur la version qui nous intéresse (ici 1.10.0) puis cliquer sur l'onget "Gradle (short)" pour savoir ce que nous devons ajouter dans la section `dependencies`

Par défault le site MVN repository va proposer de l'installer pour `implementation`, cependant on pourrait très bien choisir `api` ou `testImplementation` :

- `implementation` est la portée requise pour compiler la source de production du projet qui ne font pas partie de l'API exposée par le projet.
- `api` est la portée requise pour compiler la source de production du projet qui font partie de l'API exposée par le projet
- `testImplementation` est la portée requise pour compiler et exécuter la source de test du projet. Par exemple, le projet a décidé d'écrire le code de test avec le cadre de test JUnit.

Vous pouvez trouver plus d'information sur ce sujet sur la page consacrée à la gestion de dépendences Java de Gradle.

Projets modulaires avec Gradle

Gradle permet de décomposer le code en différents modules ce qui permet de maintenir le code plus facilement, ainsi que de le rendre plus robuste et portable.

- Pour l'activer on peut créer un projet avec `gradle init` et activer l'option pour y créer des sous-projets.
- Modifier le projet pour y ajouter des sous-projets

Lorsque l'on crée un projet modulaire avec gradle init, la librairie de test automatiquement choisie est JUnit Jupyter

Structure des projets modulaires

Les projets modulaires ont une structure plus complexe que les projets non-modulaire car ils ont des sous-projets supplémentaires que `app`. Par défault quand on init un projet Gradle avec des sous-projets, on va avoir les dossiers `app`, `buildSrc`, `list` et `utilities` qui vont être créés.

- Le dossier `app` par convention sert toujours pour le code principal.
- Le dossier `buildSrc` sert à créer des plugins ou tâches Gradle spécifiques
- Les dossiers `list` et `utilities` sont juste là à titres d'exemples. Vous pouvez par exemple avoir un sous-projet pour une librarie utilisée par d'autres sous projets, ou encore un sous-projet pour une application CLI et une autre pour une application Android.

Revision #4
Created 18 September 2023 16:40:39 by SnowCode
Updated 23 September 2023 08:30:23 by SnowCode