

# Quelques outils de base

## Table des matières



.....	4
1 – A quoi sert ant ? .....	4
2 – Installation .....	4
3 – Usage de base.....	5
3.1 – Exemple minimal .....	5
3.2 – Compilation Java .....	6
4 – Utilisation dans Eclipse .....	8



.....	10
1 – A quoi sert maven ? .....	10
2 – Installation .....	10
3 – Usage minimal .....	11
3.1 – Un projet comparable à l'exemple Ant précédent.....	11
3.2 – Créer un projet simple avec Maven .....	12
4 – Concepts utilisés dans les builds Maven .....	12
4.1 – Plugins Maven et buts ("goals") .....	13
4.2 – Cycles et phases ("lifecycle") .....	13
4.3 - Coordonnées .....	14
4.4 - Repositories.....	15
4.5 – Gestion des dépendances .....	15
5 – Utilisation dans Eclipse .....	16
5.1 – Création d'un projet Maven simple.....	16
5.2 – Exemple d'un projet Spring Maven dans Eclipse .....	18
6 – Aller plus loin avec maven.....	18



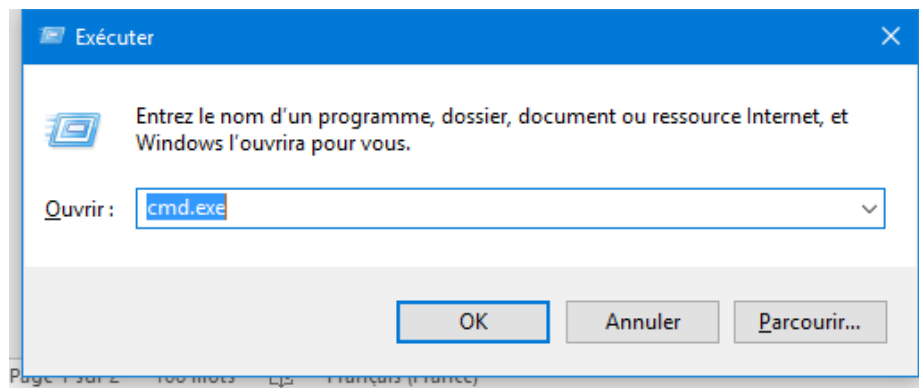
.....	19
1 – A quoi sert git ? .....	19
2 – Installation .....	20

3 – Usage de base.....	24
4 – Utilisation dans Eclipse .....	35
5 – Utilisation distante .....	37
5.1 – Principe .....	37
5.2 – Exemple de mise en œuvre .....	38
5.3 – Et Bitbucket / Github ? .....	40
6 – Aller plus loin avec Git .....	40
IV – Autres outils utiles.....	41
1 – Outils graphiques du JDK : JVisual VM (jvisualvm.exe) .....	41
2 – Editeurs de code .....	41
2.1 - SublimeText.....	41
2.2 – VSCode.....	42

On va examiner ici quelques outils fréquemment rencontrés dans l'atelier logiciel des développeurs : Ant, Maven, Git et quelques autres.

**Note** : la plupart des outils vus ci-dessous s'exécutent en ligne de commandes, ou bien dans une console PowerShell.

Pour ouvrir une console de ligne de commandes sous Windows 7 ou 10 on fera simplement : touche windows + R, et on tapera cmd.exe.



Pour qu'une commande puisse s'exécuter en ligne de commande, elle doit être vue par Windows et se trouver 'dans le path'. Il faut simplement pour cela, ajouter le répertoire où elle se trouve dans la variable d'environnement PATH (panneau de configuration, Systeme, Paramètres systèmes avancés, variables d'environnement, et on y édite la variable utilisateur PATH.



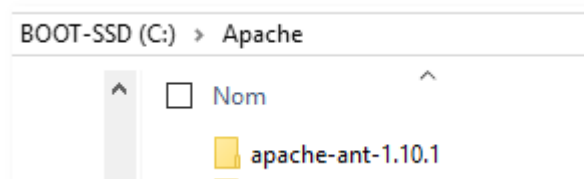
## 1 – A quoi sert ant ?

Ant (littéralement, en anglais, 'insecte') est un '*petit outil fait pour construire de gros projets*'. Ant permet donc d'effectuer les tâches typiques, utiles au développeur : compilation, exécution, manipulation de fichiers (copie, suppression ...), création de fichiers jar etc.

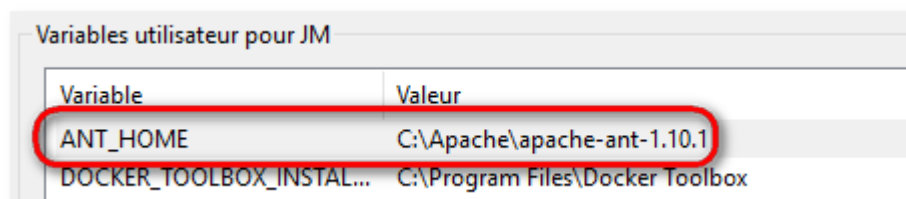
## 2 – Installation

On peut télécharger ant sur le site Apache <http://ant.apache.org/bindownload.cgi> et sa documentation se trouve sur <http://ant.apache.org/manual/index.html>.  
On va dans ces exemples utiliser la version 1.10.

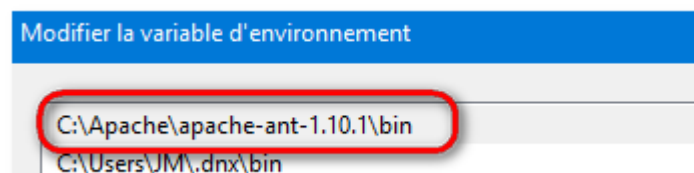
On décompresse le fichier zip téléchargé (par exemple `apache-ant-1.10.1-bin.zip`) dans son dossier :



On doit ensuite créer la variable d'environnement `ANT_HOME` et la faire pointer vers ce dossier :



On va, enfin ajouter le répertoire bin de l'installation au PATH :



L'ouverture d'un terminal et la frappe de la commande ant doit nous donner :

```
PS C:\Apache\test_ant> ant
Buildfile: build.xml does not exist!
Build failed
PS C:\Apache\test_ant>
```

### 3 – Usage de base

La syntaxe de la commande ant est simplement :

```
ant <options> <cible>
```

ant tente alors d'exécuter la tâche cible définie dans le fichier de configuration (par défaut `build.xml`), en utilisant les options précisées.

Si on tape simplement la commande `ant`, comme ci-dessus, il tente de construire la *cible par défaut* de `build.xml` (qui doit exister, sous peine de l'erreur vue ci-dessus).

#### 3.1 – Exemple minimal

Un exemple minimal est la simple copie d'un fichier texte (que l'on va créer), sous un autre nom, dans le même répertoire. Pour cela :

1. On crée un répertoire de travail (par exemple `test_ant`).
2. On y ajoute un fichier texte, par exemple `acopier.txt`, avec un contenu quelconque.
3. On crée un fichier `build.xml` structuré ainsi :

```
build.xml
<?xml version="1.0" encoding="UTF-8"?>
<project name="mon projet" default="copie" basedir=".>
  <property name="fichierDestination" value="destination.txt" />
  <target name="copie" description="Copie d'un fichier">
    <copy file="acopier.txt" tofile="${fichierDestination}" />
  </target>
</project>
```

L'exécution de la commande ant donne désormais :

```
PS C:\Apache\test_ant> ant
Buildfile: C:\Apache\test_ant\build.xml

copie:
  [copy] Copying 1 file to C:\Apache\test_ant
BUILD SUCCESSFUL
Total time: 0 seconds
```

Cet exemple montre donc :

1. La structure générale d'un fichier de build ant.
2. Comment définir une cible par défaut et un répertoire de travail.
3. Comment définir une propriété et l'utiliser dans les tâches de build, grâce à `<property>`.
4. La définition d'une cible grâce à `<target>`.
5. L'utilisation d'une tâche ant pour réaliser notre cible (ici `<copy>`).

Ce fichier fait également apparaître **les concepts de base ant** :

- La notion de **projet ant** (balise racine) : il s'agit de l'ensemble des tâches décrites par un fichier de configuration donné.
- La notion de **cible** (balise 'target') ant, qui est un but à atteindre à travers l'exécution d'une ou plusieurs tâches ant. Une cible a un *nom* (ici 'copie') et peut être spécifiée au lancement de ant, sur la ligne de commande. Il existe éventuellement une cible *par défaut*, définie au niveau du projet, par l'attribut 'default' de la balise 'project' : c'est celle qui sera exécutée si *aucun nom de cible n'est saisi sur la ligne de commande*.
- La **liste des tâches ant** nécessaires pour la bonne obtention d'une cible, ainsi que leur(s) paramètre(s) sont placées entre les balises `<target></target>`. Ant possède nombreuses tâches standard (voir la documentation) et des tâches spécifiques peuvent être créées en Java.
- La possibilité de définir des **propriétés** (balise `<property>`), des espèces de variables utilisable à plusieurs endroits dans le fichier de build ant, via la syntaxe `${nom_prop}`.

### 3.2 – Compilation Java

On va, maintenant, dans cet exemple, compiler deux classes java, et créer un fichier jar les contenant.

On place dans un dossier src/test les deux classes Java `Principale.java` et `Voiture.java` (toutes les deux membres du package test).

Elles contiennent :

```
package test;

public class Principale {

    public static void main(String[] args) {
        Voiture v = new Voiture();
        v.demarrer();
    }
}
```

```

package test;

public class Voiture {

    public void demarrer() {
        System.out.println("vroom vroom, je démarre...");
    }
}

```

Puis on modifie le fichier `build.xml` pour qu'il contienne :

```

<?xml version="1.0" encoding="UTF-8"?>
<project name="mon projet" default="copie" basedir=".">
    <property name="fichierDestination" value="destination.txt" />
    <target name="copie" description="Copie d'un fichier">
        <copy file="acopier.txt" tofile="${fichierDestination}" />
    </target>
    <target name="compilation">
        <javac srcdir="./src" destdir="./bin" includeAntRuntime="false" />
    </target>
    <target name="archive" depends="compilation">
        <jar destfile="appli.jar" >
            <fileset dir="./bin" />
            <manifest>
                <attribute name="Main-Class" value="test.Principale" />
            </manifest>
        </jar>
    </target>
</project>

```

On lance la création du jar par la commande :

```
ant archive
```

Qui nous donne :

```

PS C:\Apache\test_ant> ant archive
Buildfile: C:\Apache\test_ant\build.xml

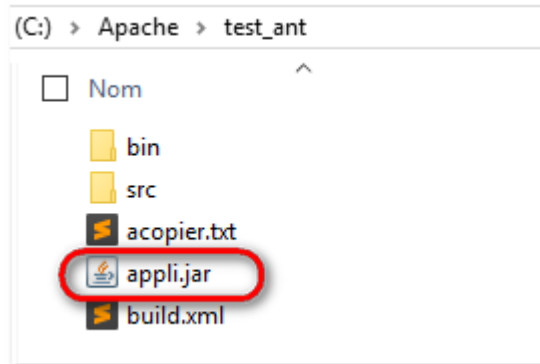
compilation:
[javac] Compiling 2 source files to C:\Apache\test_ant\bin

archive:
[jar] Building jar: C:\Apache\test_ant\appli.jar

BUILD SUCCESSFUL

```

On observe la construction du jar :



Ensuite, l'exécution du jar exécutable créé par la commande :

```
java -jar appli.jar
```

Donne :

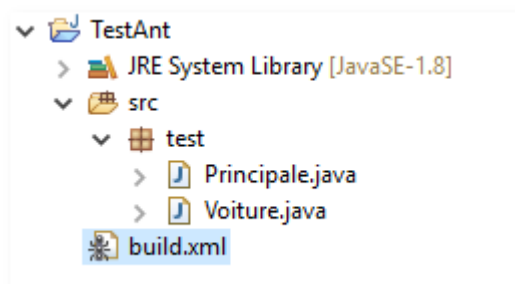
```
PS C:\Apache\test_ant> java -jar appli.jar
vroom vroom, je démarre...
```

#### 4 – Utilisation dans Eclipse

Les détails de l'utilisation de Ant dans Eclipse se trouvent dans la rubrique d'aide "Workbench User Guide / Tasks / Running Ant buildfiles".

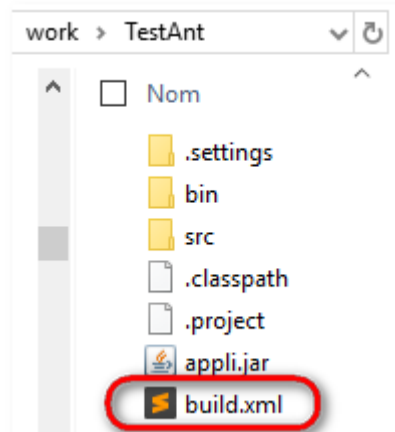
En résumé :

1. On crée un projet Java Eclipse (par exemple "TestAnt")
2. On y place les deux fichiers .java précédant, dans leur package test.
3. On copie le fichier build.xml dans le dossier du projet. Vu depuis l'explorateur de packages :

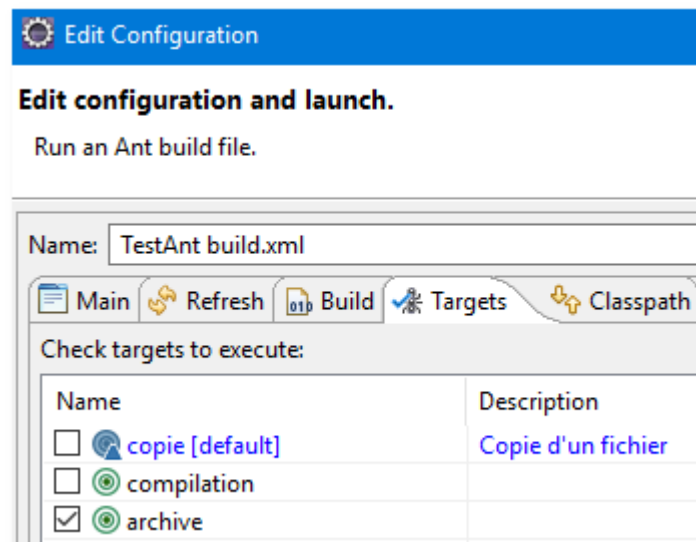


4. Et vu depuis l'explorateur Windows :





5. On effectue un clic droit sur le fichier build.xml et on sélectionne Run as / Ant Build...  
On prend bien l'option avec les 3 points, ce qui nous permet de choisir la cible :



6. On coche archive, qui est la cible qui nous intéresse et on lance le build. Le jar est alors créé.

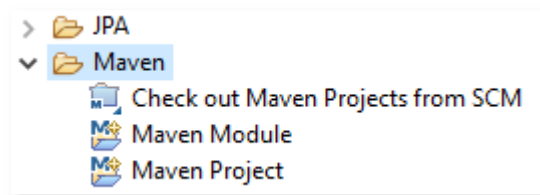


## 1 – A quoi sert maven ?

Maven est le successeur de Ant et est particulièrement puissant et nettement plus complexe. Il va pouvoir construire nos projets :

1. En étant piloté par un fichier XML (pom.xml, pour 'Project Object Model'), aussi simple que possible, et privilégiant la "convention plutôt que la configuration".
2. En identifiant et téléchargeant les dépendances du projet depuis des repositories distants, et en les maintenant dans un repository local.
3. En possédant des gabarits de projet courant (les "archétypes maven") permettant de simplifier les builds.
4. En utilisant des plugins, lui permettant de s'interfacer avec de nombreux autres outils (ex : exécuter les tests unitaires comme étape d'un processus de build, être piloté par des outils d'intégration continue comme Jenkins etc.)

Maven est supporté par tous les grands IDEs Java comme Eclipse, Netbeans et IntelliJ qui savent construire les projets basés sur Maven. Dans Eclipse :



Il est à l'heure actuelle fréquemment utilisé pour construire les projets complexes mettant en œuvre de nombreuses dépendances (projets Spring, notamment).

Maven semble progressivement laisser la place à un outil plus récent, Gradle, reprenant les grands concepts de Maven mais *plus souple car scriptable via le langage Groovy* (un Java simplifié, générant du bytecode, et donc exécutable grâce à la JVM).

## 2 – Installation

On commence par télécharger le fichier zippé de la distribution binaire de maven (un fichier de nom `apache-maven-3.x.x-bin.zip`) sur le site <https://maven.apache.org>.

On le dézippe à l'emplacement souhaité, en prenant garde à ce que la variable d'environnement `JAVA_HOME` soit bien définie et initialisé vers le répertoire d'un JDK de version 1.7 ou plus récent.

On veille également à mettre dans le `PATH` le répertoire bin de l'installation de maven contenant le script `mvn.bat`.

Ensuite, l'ouverture d'une fenêtre console doit nous donner quelque chose comme :

```
C:\Users\JM>mvn -v
Apache Maven 3.2.1 (ea8b2b07643dbb1b84b6d16e1f08391b666bc1e9; 2014-02-14T18:37:52+01:00)
Maven home: C:\Apache\apache-maven-3.2.1
Java version: 1.8.0_144, vendor: Oracle Corporation
Java home: C:\Program Files\Java\jdk1.8.0_144\jre
Default locale: fr_FR, platform encoding: Cp1252
OS name: "windows 10", version: "10.0", arch: "amd64", family: "dos"
```

### 3 – Usage minimal

#### 3.1 – Un projet comparable à l'exemple Ant précédent

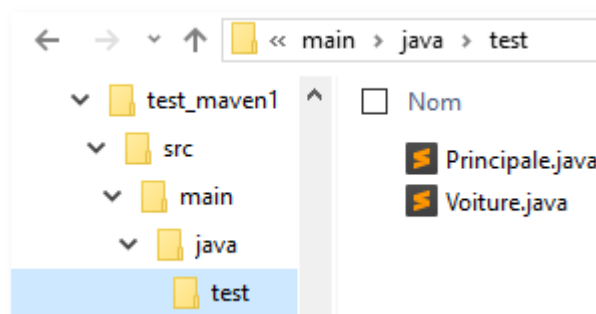
Pour utiliser à minima Maven, on va créer un répertoire `test_maven1`, dans lequel on va placer :

1. Un fichier `pom.xml`, contenant les lignes suivantes :

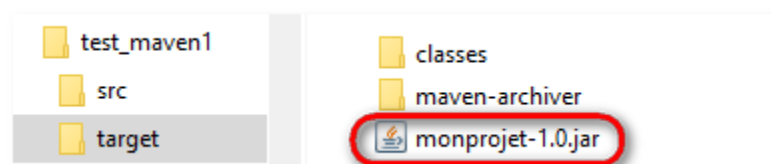
```
pom.xml
<project>
  <modelVersion>4.0.0</modelVersion>
  <groupId>fr.arati.demo1</groupId>
  <artifactId>monprojet</artifactId>
  <version>1.0</version>
</project>
```

2. Les fichiers Java utilisés précédemment avec l'exemple Ant (`Principale.java` et `Voiture.java`, tous deux dans le package `test`), rangé dans l'arborescence :

<rép. base > \ src \ main \ java :



3. On ouvre une console dans le répertoire `test_maven1`, et on lance la commande : `mvn install`. On y observe le téléchargement et la construction d'un **dépôt local**, placé dans le dossier `<home>\.m2` ainsi que la construction du projet. L'affichage nous indique la compilation des sources, et un 'BUILD SUCCESS'. L'examen du répertoire montre la création du dossier `target` contenant le jar généré :



4. Il suffit de se placer dans le répertoire `target` et de taper dans la console :

```
PS C:\Apache\test_maven1> cd target
PS C:\Apache\test_maven1\target> java -cp monprojet-1.0.jar test.Principale
vroom vroom, je démarre...
PS C:\Apache\test_maven1\target>
```

Ce petit exemple montre que Maven est **nettement plus déclaratif que Ant**. Plutôt que d'expliquer comment parvenir au but, on conforme la structure de notre répertoire au format attendu par Maven, et on laisse s'appliquer le traitement par défaut. Dans le fichier `pom.xml` aucune définition de build n'a été indiquée : le comportement par défaut s'est donc appliqué : compilation et création d'un jar.

Ant est plus 'procédural', et possède moins de comportement par défaut ("convention over configuration").

### 3.2 – Créer un projet simple avec Maven

Maven utilise un certain nombre de conventions que nos projets devront suivre, s'ils veulent bénéficier de la technique de build de Maven. Pour l'observer :

1. On crée un dossier `test_maven2`, on y ouvre une console et on y tape :  
`mvn archetype:generate`
2. Le type de projet (archétype `quickstart`) est choisi par défaut (no 1381), on appuie simplement sur Entrée.
3. Puis on choisit 6 (version 1.1) comme version de cet archétype de projet.
4. On prend '`fr.arati.demo`' comme `groupId` et '`demo2`' comme `artifactId` (par exemple).
5. On conserve la version proposée.
6. On valide l'ensemble et le projet est généré.

On observe :

- La structure des dossiers générés (code java, tests unitaires).
- Le contenu du `pom.xml`, particulièrement les dépendances (ici `jUnit 3.8.1`).

On fait le build de ce projet, en se plaçant dans le dossier du nom de l'`artifactId` (`demo2`, ici) et on lance : `mvn install` qui produit le build du projet, et **exécute les tests unitaires** :

```
-----
T E S T S
-----
Running fr.arati.demo.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0
```

On observe la génération effectuée dans le dossier `target`.

## 4 – Concepts utilisés dans les builds Maven

#### 4.1 – Plugins Maven et buts ("goals")

Maven contient de nombreux plugins, et chaque plugin définit des buts. On lance une tâche de build en utilisant la syntaxe :

```
mvn <plugin>:<but>           Ex:mvn archetype:generate
```

Comme plugins on peut citer le plugin Compiler, Jar, Archetype etc. De nouveaux plugins peuvent également être développés en Java (ou d'autres langages comme Ruby ou Groovy).

Un but est une tâche élémentaire de build, comme `Compiler:compile`, qui compile les sources Java.

Le cœur de Maven a simplement pour rôle de lancer des plugins, et de leur faire exécuter des buts.

#### 4.2 – Cycles et phases ("lifecycle")

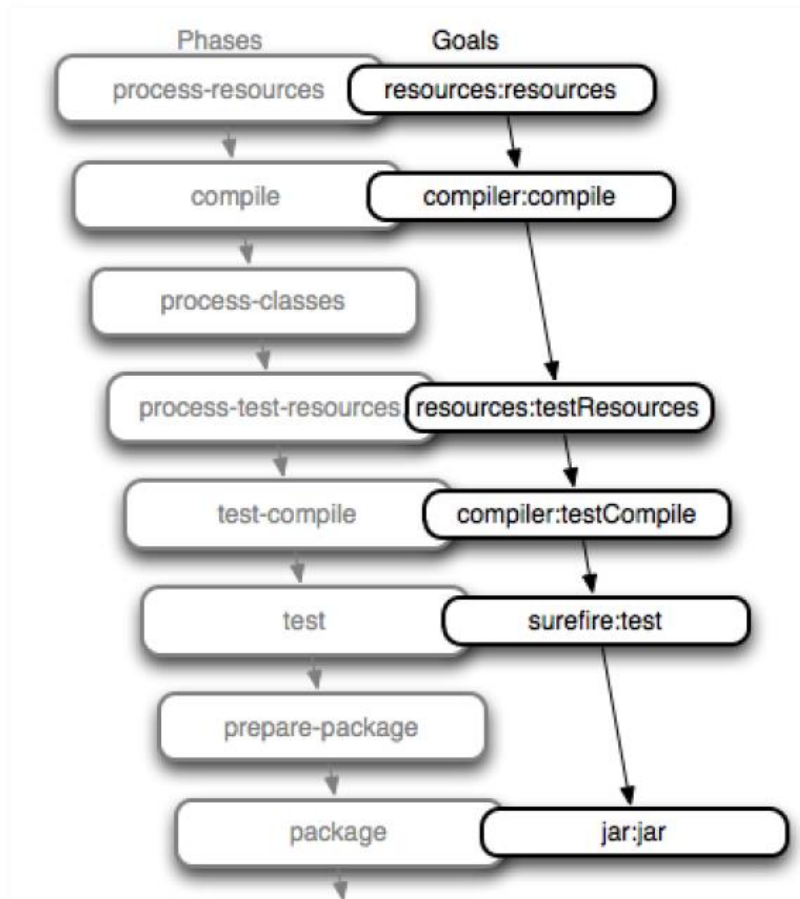
Si on observe un cycle de build (ici le `mvn install` précédant) on constate qu'une série de **phases** sont exécutées : `resources:resources`, `compiler:compile`, `resources:testresources` etc.

```
[INFO] -----
[INFO] Building demo2 1.0-SNAPSHOT
[INFO] -----
[INFO] --- maven-resources-plugin:2.6:resources (default-resources) @ demo2 ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\Apache\test_maven2\demo2\src\main\resources
[INFO] --- maven-compiler-plugin:2.5.1:compile (default-compile) @ demo2 ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- maven-resources-plugin:2.6:testResources (default-testResources) @ demo2 ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory C:\Apache\test_maven2\demo2\src\test\resources
[INFO] --- maven-compiler-plugin:2.5.1:testCompile (default-testCompile) @ demo2 ---
[INFO] Nothing to compile - all classes are up to date
[INFO] --- maven-surefire-plugin:2.12.4:test (default-test) @ demo2 ---
[INFO] Surefire report directory: C:\Apache\test_maven2\demo2\target\surefire-reports

T E S T S
-----
Running fr.arati.demo.AppTest
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0 sec
Results :
Tests run: 1, Failures: 0, Errors: 0, Skipped: 0

[INFO] --- maven-jar-plugin:2.4:jar (default-jar) @ demo2 ---
[INFO] -----
```

Chacune de ces phases est associée à un but, et la correspondance, dans notre cas apparaît dans la figure suivante :



La **construction des projets maven utilise toujours des phases**, qui sont plus ou moins les mêmes d'un projet à l'autre, certaines étant plus utilisées. Citons : **validate, compile, test, package, install, deploy** etc.

Pour lancer une phase maven on tape simplement : `mvn <phase>` (ex : `mvn install`). On lance **plus fréquemment Maven via un nom de phase** qu'avec un but de plugin.

#### 4.3 - Coordonnées

Le fichier de configuration `pom.xml` contient une série d'identifiants uniques appelées **coordonnées d'un projet**, qui sont :

- **groupId** : le groupe, l'équipe, la société ou l'organisation à qui appartient le projet. L'usage veut que, comme les packages Java, on utilise des noms de domaines inversés. Ex : `fr.arati.demo`.
- **artifactId** : un identificateur unique représentant le nom du projet.
- **version** : la version spécifique d'un projet. En cours de développement le no de version est suivi du mot `SNAPSHOT`.

Le mode de packaging peut être aussi précisé et est, par défaut, **jar** pour les projets Java :

```
<groupId>fr.arati.demo</groupId>
<artifactId>demo2</artifactId>
<version>1.0-SNAPSHOT</version>
<packaging>jar</packaging>
```

#### 4.4 - Repositories

Maven va, lorsque c'est nécessaire, se connecter sur un ou plusieurs repositories pour télécharger des plugins ou des jars nécessaires. Il les stocke dans un **repository local**, dans le répertoire `<home>\.m2`.

Par défaut le repository distant de maven est `http://repo1.maven.org/maven2/`.

#### 4.5 – Gestion des dépendances

C'est un des rôles essentiels de Maven : retrouver les dépendances d'un projet.

Par exemple, le fichier `pom.xml` du dossier `dependances`, retrouve les dépendances du framework Spring 4.3.2 et de quelques librairies associées :

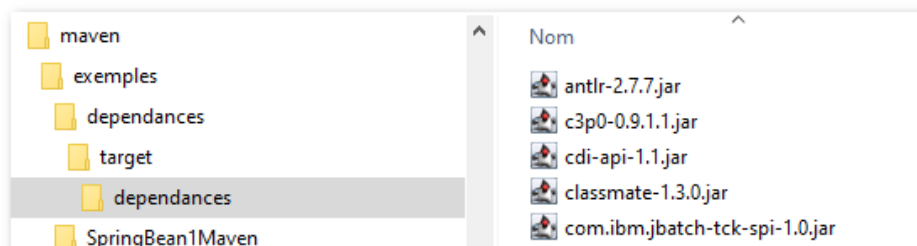
```
<dependencies>
  <!-- spring core -->
  <dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-context</artifactId>
    <version>4.3.2.RELEASE</version>
  </dependency>
  <!-- hibernate -->
</dependencies>
```

La seconde partie identifie le plugin recherchant les dépendances, la phase de build (`generate-resources`) et le dossier de sortie (`target\dependances`) :

```
<plugins>
  <plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-dependency-plugin</artifactId>
    <version>2.8</version>
    <executions>
      <execution>
        <id>download-dependencies</id>
        <phase>generate-resources</phase>
        <goals>
          <goal>copy-dependencies</goal>
        </goals>
        <configuration>
          <outputDirectory> ${project.build.directory}/dependances </outputDirectory>
        </configuration>
      </execution>
    </executions>
  </plugin>
</plugins>
```

On lance cette copie par la commande `mvn generate-resources` (nom de la phase).

On observe que 50 jars ont bien été téléchargé depuis le repository Maven et placé dans le dossier 'dependances' :



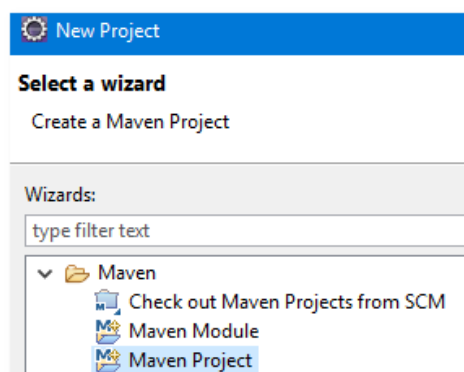
## 5 – Utilisation dans Eclipse

### 5.1 – Création d'un projet Maven simple

Il est possible d'utiliser, de créer et d'importer des projets Maven dans Eclipse, comme dans les autres IDEs. Eclipse est distribué avec un maven "embarqué", mais une version externe peut être aussi utilisée.

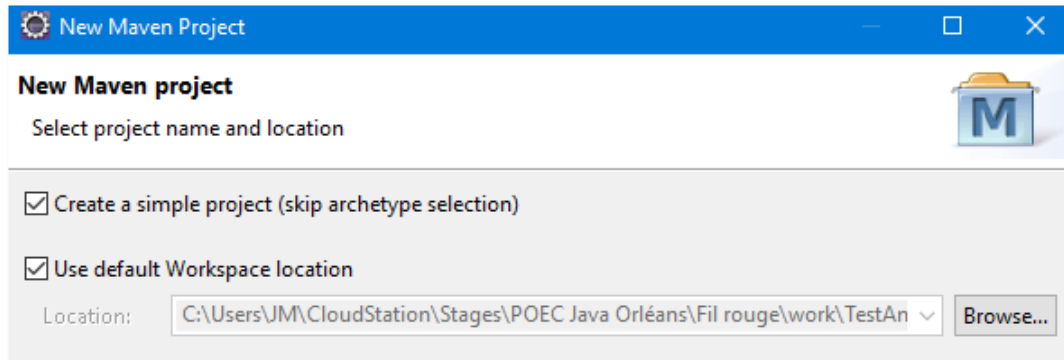
Créons un projet Eclipse Maven simple :

1. On commence par créer un nouveau projet, et dans le wizard, on choisit un projet Maven :

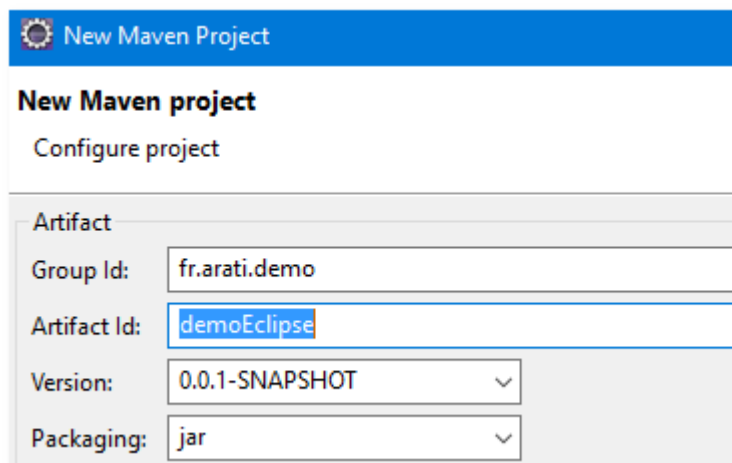




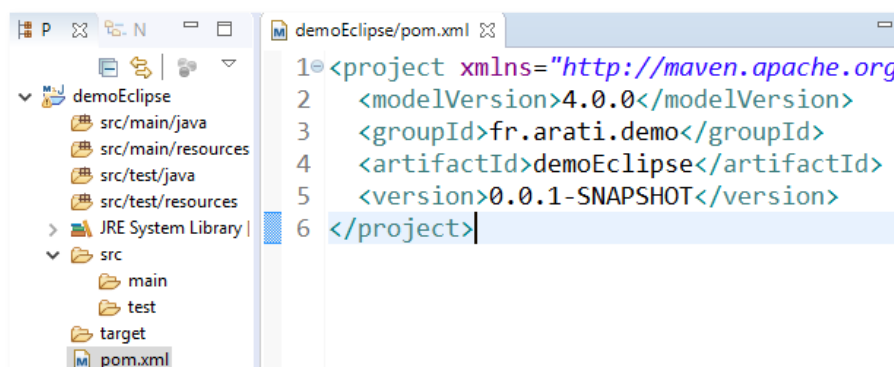
2. La seconde étape consiste à choisir un archétype de projet, ou bien, si on saute cette étape, à utiliser un format de projet "simple" (non Java EE, Spring etc.). On va utiliser un projet simple :



3. On renseigne ensuite les "coordonnées maven" du projet ainsi que quelques autres informations (packaging, notamment) :



4. On clique ensuite sur Finish pour obtenir notre projet. On retrouve alors l'organisation des répertoires typique des projets maven :



5. Etrangement cependant, Eclipse nous a configuré ce projet pour utiliser Java 5 (ancienne version) ! Pour utiliser la version actuelle JDK 8, on doit ajouter à pom.xml les instructions suivantes :

```

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www
<modelVersion>4.0.0</modelVersion>
<groupId>fr.arati.demo</groupId>
<artifactId>demoEclipse</artifactId>
<version>0.0.1-SNAPSHOT</version>

<properties>
  <maven.compiler.source>1.8</maven.compiler.source>
  <maven.compiler.target>1.8</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>

</project>

```

6. Pour les faire prendre en compte par Eclipse on doit faire la mise à jour du projet (clic droit sur le projet, option Maven / Update project...).
7. Pour exécuter le build du projet, on ajoute une classe Java minimale, dans un package, dans le dossier src/main/java et on choisit (clic droit sur le projet) l'option Run As / Maven build... Le dialogue suivant apparait et on y définit le but :

Base directory:

Goals:

Profiles:

User settings:

Workspace... File System... Variables...

Pour changer la version maven utilisée, on peut aller dans Window / Preferences / Maven / Installations et ajouter une installation externe de Maven plutôt que la version embarquée dans Eclipse.

## 5.2 – Exemple d'un projet Spring Maven dans Eclipse

Comme exemple d'un projet Eclipse plus réaliste, on examinera un projet Spring / Maven dans Eclipse.

## 6 – Aller plus loin avec maven

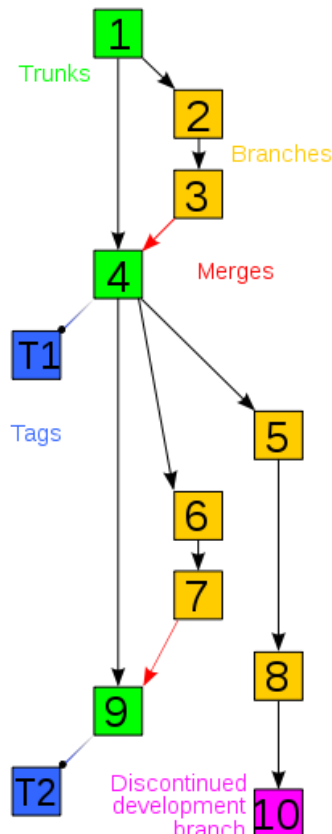
D'innombrables tutoriels de Maven existent sur le web, et plusieurs livres ont été écrits sur lui. Le développement de Maven, a été largement financé par la société Sonatype, qui a également écrit des livres gratuits très complets, comme le "Maven The reference guide", fourni avec ce document.

Le site <http://maven.apache.org> reste le site de référence de Maven.



## 1 – A quoi sert git ?

Git est un outil de SCM, pour "Source Control Management", où "Outil de Gestion de Code Source", ou encore, plus simplement un outil de "Gestion de versions" (voir Wikipédia [https://fr.wikipedia.org/wiki/Gestion\\_de\\_versions](https://fr.wikipedia.org/wiki/Gestion_de_versions), pour une vue d'ensemble des concepts de ces outils (versions, dépôts, branches, commits etc.). Une illustration de cet article :



Il s'agit d'un outil gratuit, permettant le travail en mode autonome (sans serveur), mais également avec un dépôt central, partagé, "consolidant" et regroupant de multiples dépôts décentralisés. Il serait, suivant Wikipédia, utilisé actuellement par plus de 12 millions de personnes...

Il a été développé pour les besoins du développement Linux par la communauté autour de Linus Torvald et est actuellement assez largement utilisé dans le monde du développement Open Source.

Il est une alternative à de nombreux autres outils comme CVS, SVN, Mercurial, ClearCase, Team Foundation Server de Microsoft etc.

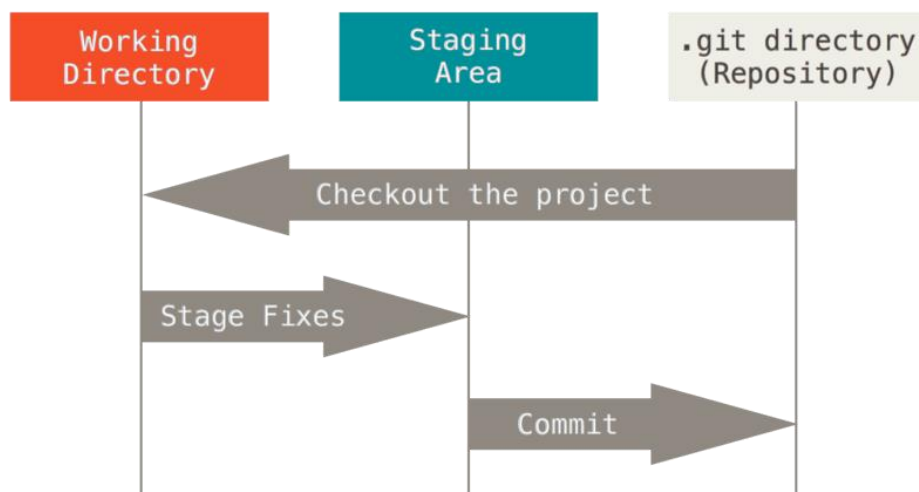
Il est conforme à l'esprit du monde Linux : un environnement console, de multiples commandes, une grande puissance mais un usage parfois un peu cryptique, et une courbe d'apprentissage assez sérieuse...

Enfin, de nombreux outils ont été créés autour de Git : des interfaces graphiques (comme

SourceTree par exemple), des plugins d'IDE (pour Eclipse comme EGit, Netbeans etc.), Web comme GitHub, GitLab, Bitbucket, Gerrit etc.

Pour travailler efficacement avec Git, il faut se souvenir (voir le livre officiel en ligne, Progit, à l'URL <https://git-scm.com/book/en/v2/Getting-Started-Git-Basics>) qu'il considère qu'un fichier peut se trouver dans les états :

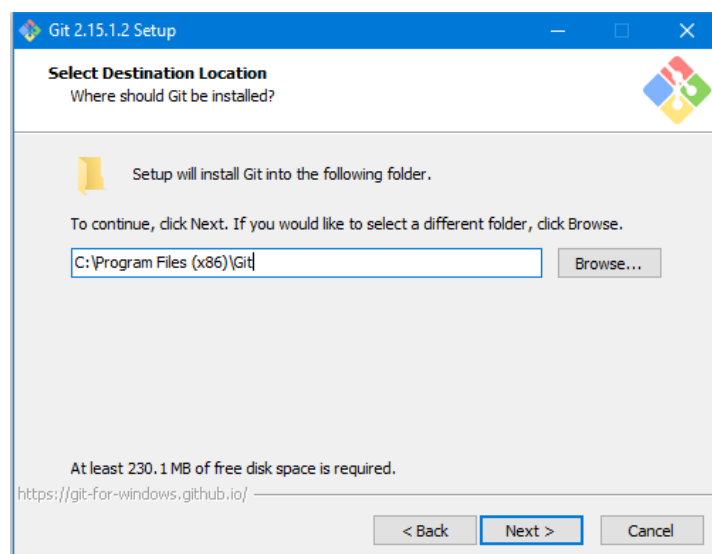
- **Non surveillé** par git (« **untracked** »).
- **Surveillé** (placé dans la zone de 'Staging', de transit) mais **non enregistré** par Git. On utilise la commande `add` pour cela.
- **Enregistré dans le repository** ('committed').



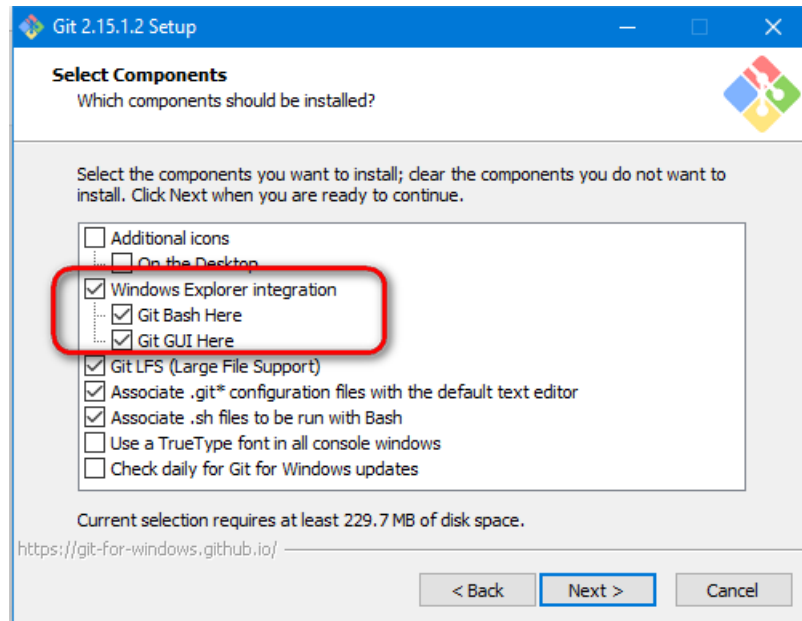
Diverses commandes Git permettent de le faire passer entre tous ces états.

## 2 – Installation

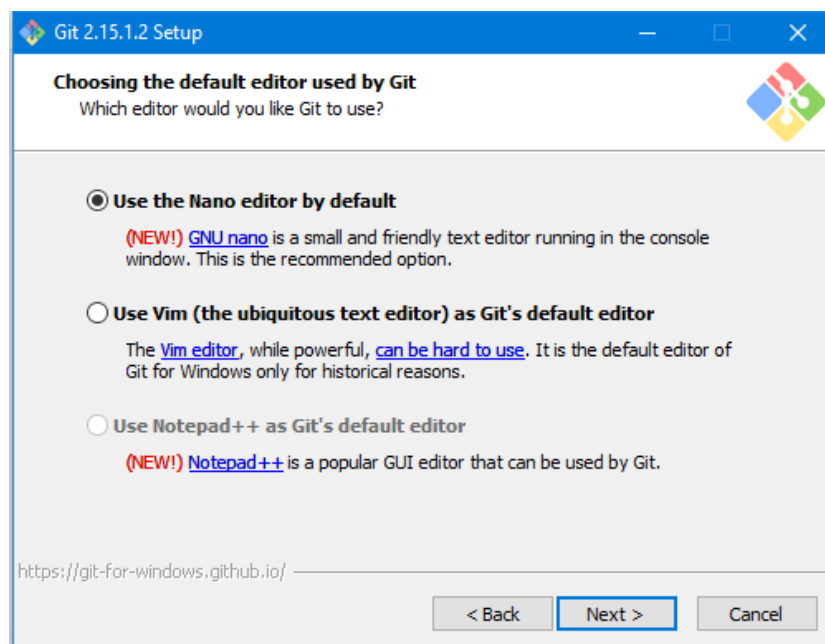
On le télécharge sur le site <https://git-scm.com/downloads> et le lancement de l'installateur se fait sans problème :



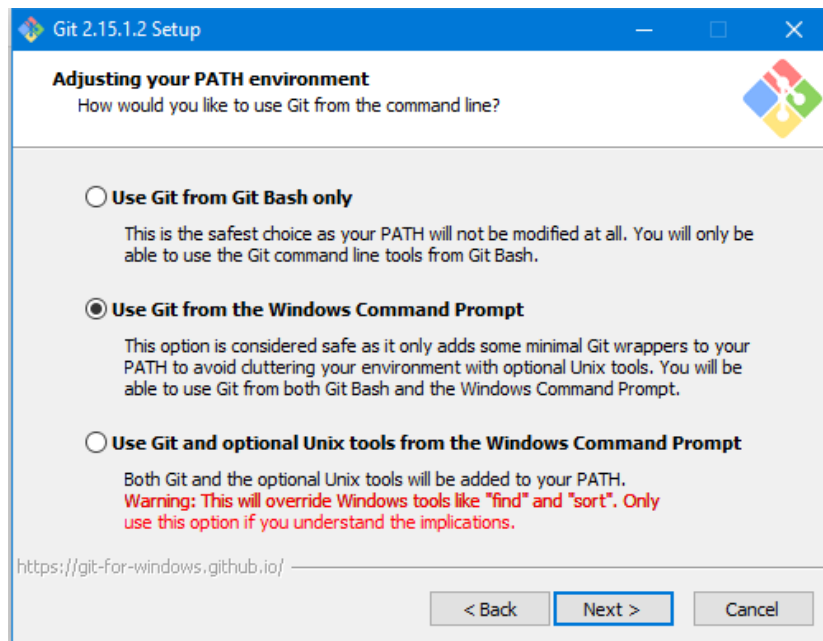
Dans les options suivantes on peut demander l'intégration à l'explorateur Windows (cochées par défaut) :



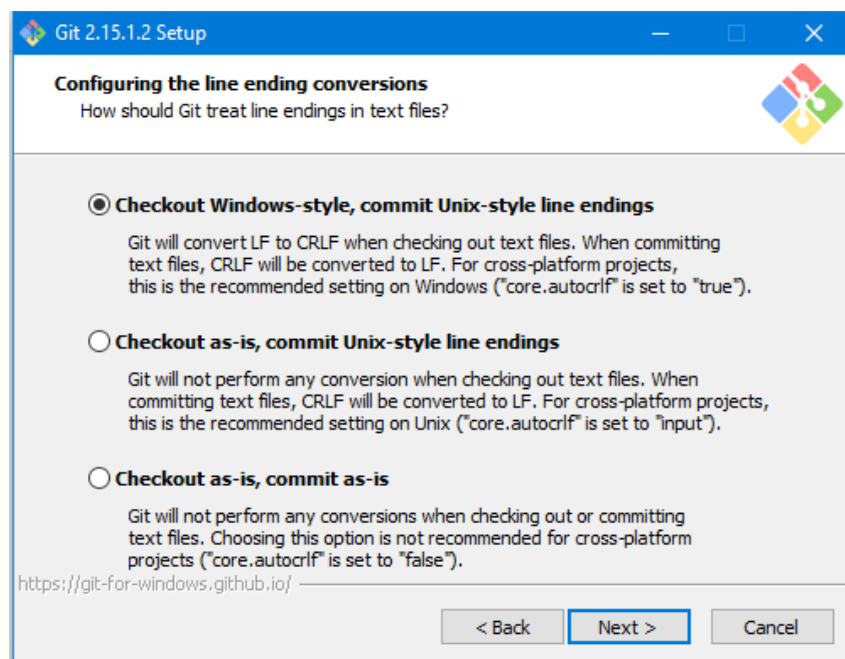
Après la création d'un raccourci dans le menu de démarrage, Git nous propose de choisir un éditeur par défaut (dont Notepad++ s'il est installé sur la machine) :



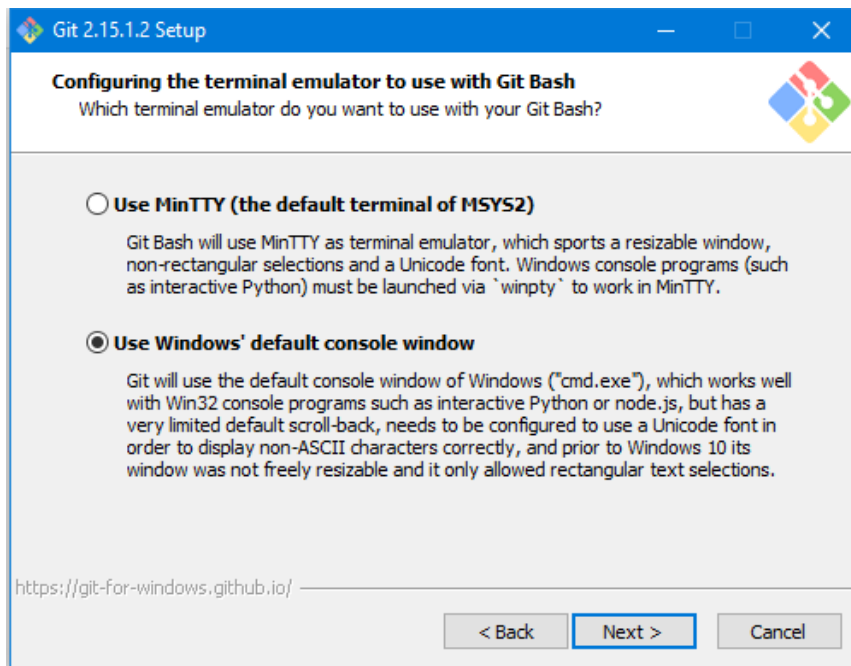
Ensuite Git permet de choisir le type d'intégration de la ligne de commande (bash, cmd ou Linux) ajout de commandes Linux à cmd.exe) :



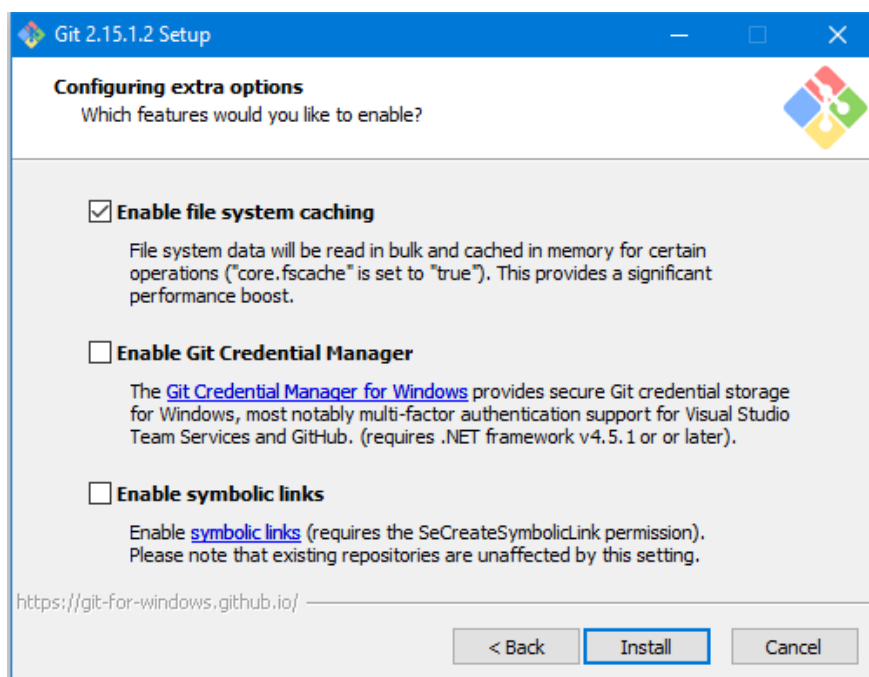
On choisit ensuite le type de connexion SSH que l'on souhaite utiliser (OpenSSH et putty, typiquement) et le type de délimiteur de ligne utilisé :



Enfin on demande l'utilisation de la console Windows comme terminal :



Et finalement :



L'installation s'exécute alors normalement. Pour vérifier que l'installation est correcte et que la commande git est disponible, on ouvre une invite de commande et on tape `git help`, la liste des commandes git doit alors s'afficher.

Dans le cas contraire on doit placer manuellement le chemin du dossier d'installation de Git dans la variable d'environnement PATH et réessayer.

Et Github, dans tout cela ?

GitHub est distinct de Git.

Github est un site payant (gratuit dans certaines conditions) hébergeant de nombreux projets Open Source. Il est possible d'obtenir une copie de ces projets en « clonant » les repositories Git qu'il héberge.

Si une entreprise souhaite fabriquer un équivalent de GitHub pour héberger ses propres projets en interne, elle peut utiliser GitLab, par exemple.

Noter que GitHub a été récemment racheté par Microsoft.

Pour cloner un repository GitHub on pourra utiliser une commande comme :

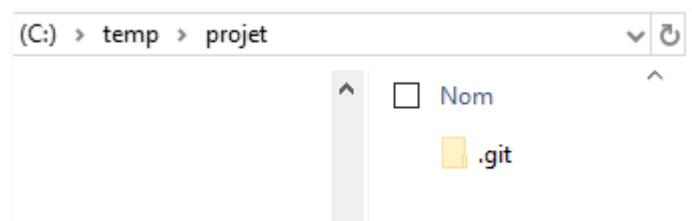
```
git clone https://github.com/libgit2/libgit2 monlibgit2
```

### 3 – Usage de base

Les quelques étapes suivantes illustrent l'utilisation de base, simplifiée, de git.

#### 1. Création d'un dossier de travail et initialisation d'un dossier par git en utilisant la commande `git init`.

- On commence par créer un répertoire projet, qui va contenir les fichiers à placer sous le contrôle de git et on y ouvre une console (classique ou powershell). On commence par y taper simplement `git` pour vérifier que la commande y est bien disponible.
- On tape ensuite la commande `git help init` pour consulter l'aide de la commande `init`.
- On y tape la commande `git init` pour initialiser le dossier projet. On y note la présence, après cela, d'un dossier caché `.git`, qui peut devenir très volumineux et **qui contient le repository lui-même** (il n'y a pas de base de données centralisée, dans git) :



#### 2. Vérification de la configuration courante de git par `git config --list`, modification de quelques valeurs (`user.name` et `user.email`). On y repère les valeurs `user.name` et `user.email`, nécessaire pour identifier l'auteur des commits :



```

filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
pack.packsizelimit=2g
core.editor=nano.exe
core.editor='C:/Program Files/Sublime Text
user.name=Jean Michel
user.email=arati@wanadoo.fr
core.repositoryformatversion=0

```

(on tapera sur 'q', pour quitter la commande)

Si les valeurs ne sont pas définies, on doit le faire avant toute autre opération. Elles peuvent être définies globalement, pour tous les repositories (et donc stockées dans le fichier `.gitconfig` du répertoire home) , ou localement, juste pour ce dossier (par défaut) :

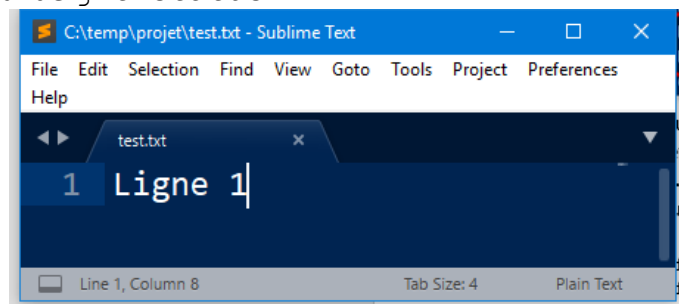
```

git config --global user.name "paul dupont"
git config --global user.email "pdupont@gmail.com"
git config --list

```

### 3. Premier commit d'un fichier.

- Ajout d'un fichier texte (ici `test.txt`) dans le dossier et examen de son état par la commande `git status`.



La commande `git status` donne :

```

PS C:\temp\projet> git status
On branch master

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    test.txt

nothing added to commit but untracked files present (use "git add" to track)

```

- On le passe en mode 'indexé' ('staged') par la commande `git add`, et on examine son statut par `git status` :

```

git add text.txt
git status

```

```

PS C:\temp\projet> git add test.txt
PS C:\temp\projet> git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)

        new file:   test.txt

```

- Enfin on le commit par `git commit`, en ajoutant un message, en testant le statut courant, et en utilisant la commande `git log` pour voir l'historique des commits :

```

git commit -m "Premier commit"
git status
git log

```

```

PS C:\temp\projet> git commit -m "Premier commit"
[master (root-commit) b778ea3] Premier commit
1 file changed, 1 insertion(+)
 create mode 100644 test.txt
PS C:\temp\projet> git status
On branch master
nothing to commit, working tree clean
PS C:\temp\projet> git log
commit b778ea3c6f559a2351cff9e4ccdc3aac49626e79 (HEAD -> master)
Author: Jean Michel <arati@wanadoo.fr>
Date:   Tue Oct 3 15:35:02 2017 +0200

    Premier commit

```

- Ou la version sur une ligne, plus compacte de `git log --oneline` :

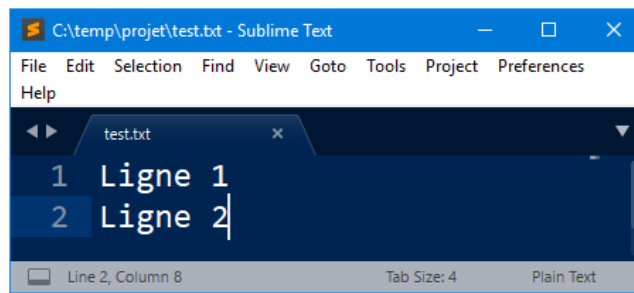
```

PS C:\temp\projet> git help log
PS C:\temp\projet> git log --oneline
b778ea3 (HEAD -> master) Premier commit

```

Le commit est désigné par un code SHA-1, dont seuls les 7 premiers caractères (ici b778ea3) suffisent pour le désigner. **La première version de test.txt est donc enregistrée dans le repository.**

- On modifie ensuite ce fichier, on observe son statut :  
On commence par ajouter une seconde ligne à test.txt, et on l'enregistre :



La commande `git status` nous montre que les modifications sont bien vues :

```
PS C:\temp\projet> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

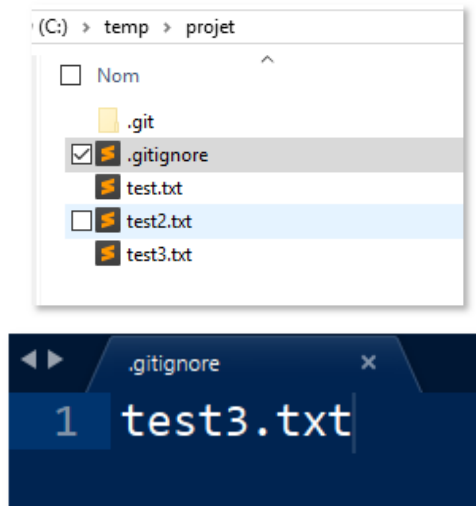
- On le 'commite' à nouveau (on l'ajoute sans avoir besoin d'appeler add, via l'option -a de commit) :  
`git commit -a -m "Second commit"`  
`git log --oneline`

```
PS C:\temp\projet> git log --oneline
abb8524 (HEAD -> master) Second commit
b778ea3 Premier commit
```

Les deux commits effectués sont bien visibles. On aurait pu travailler avec des caractères génériques pour ajouter plusieurs fichiers d'un coup, si nous en avions plus d'un :

```
add *.txt # tous les fichiers texte
add .     # tous les fichiers du répertoire courant
```

- Pour éviter de placer dans le repository des fichiers produits par la compilation, des fichiers de log ou tout fichier non souhaité on peut utiliser un fichier `.gitignore` listant les exclusions.
  - On ajoute deux autres fichiers texte au dossier projet : `test2.txt` (contenant juste la ligne `test2`), et `test3.txt` (contenant juste la ligne `test3`), par exemple et on crée un fichier `.gitignore`, contenant sur chaque ligne, la liste des fichiers à ignorer :



- Si on fait un `git status`, on observe que le fichier `test3.txt` a bien été ignoré :

```
PS C:\temp\projet> git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore
        test2.txt

nothing added to commit but untracked files present (use "git add" to track)
```

- Puis on ajoute une troisième ligne à `test.txt`, on ajoute tous les fichiers `.txt` à la zone surveillée et on fait un commit et un log.  
`git add *.txt`  
`git commit -m "Troisieme commit"`  
`git log --oneline`

On obtient :

```
PS C:\temp\projet> git log --oneline
bb4499a (HEAD -> master) Troisieme commit
abb8524 Second commit
b778ea3 Premier commit
```

En cas d'erreur dans le message du dernier commit, on peut toujours le modifier par :  
`git commit --amend`

Qui ouvre l'éditeur choisi lors de l'installation pour éditer le message.

5. La commande `log` a de nombreuses options de formatage. Par exemple :  
`git log -n 2` # juste les deux derniers commits  
`git log -p -n 2` # fait en plus un diff entre les versions
6. L'utilisation de la commande `git diff` permet de voir les différences de la version actuelle avec un commit spécifique, désigné par son code SHA abrégé, (ou encore de voir les différences entre deux commits) :

```

PS C:\temp\projet> git diff b778ea3
diff --git a/test.txt b/test.txt
index a692012..d7cd8eb 100644
--- a/test.txt
+++ b/test.txt
@@ -1,1,2 @@
-Ligne 1
\ No newline at end of file
+Ligne 1
+Ligne 2
\ No newline at end of file

```

On aurait pu utiliser : `git diff abb8524 b778ea3`

7. **Revenir dans le passé** : utilisation de la commande `checkout`. Elle permet de retrouver des fichiers dans l'état des commits précédents (ou bien de changer de branche, voir plus loin).

Si je décide de revenir au premier commit, via `checkout`, par exemple :

```
git checkout b778ea3
```

On obtient alors :

```

PS C:\temp\projet> git log --oneline
bb4499a (HEAD -> master) Troisieme commit
abb8524 Second commit
b778ea3 Premier commit
PS C:\temp\projet> git checkout b778ea3
Note: checking out 'b778ea3'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

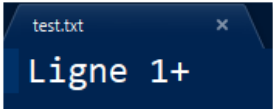
  git checkout -b <new-branch-name>

HEAD is now at b778ea3... Premier commit

```

On constate que les fichiers sont bien revenus dans l'état où on les avait laissés et que `test2.txt` a disparu, car il ne faisait pas parti de ce commit.

Maintenant que faire ? Si on souhaite modifier `test.txt` (en ajoutant '+' à la seule ligne qu'il contient, par exemple) :



A screenshot of a text editor window titled "test.txt". The window shows a single line of text: "Ligne 1+".

Et que l'on fait un commit de notre modification, puis une tentative de retour à l'état courant de la branche maitre :

```
git commit -a -m "Commit possible ?"  
git checkout master  
git log --oneline
```

On obtient :

```
PS C:\temp\projet> git commit -a -m "Commit possible"  
[detached HEAD 042a249] Commit possible  
1 file changed, 1 insertion(+), 1 deletion(-)  
PS C:\temp\projet> git checkout master  
Warning: you are leaving 1 commit behind, not connected to  
any of your branches:  
  
042a249 Commit possible  
  
If you want to keep it by creating a new branch, this may be a good time  
to do so with:  
  
git branch <new-branch-name> 042a249  
  
Switched to branch 'master'  
PS C:\temp\projet> git log --oneline  
bb4499a (HEAD -> master) Troisieme commit  
abb8524 Second commit  
b778ea3 Premier commit
```

Le commit "Commit possible ?" n'est pas associé à la branche maitresse, et ne peut pas y être conservé. Le seul moyen pour cela aurait été de créer une nouvelle branche. On constate le retour du fichier qui avait disparu (test2.txt) et des 3 lignes dans test.txt.

Dans ce cas checkout nous a donc permis **d'explorer le passé et de retrouver les versions précédentes**, mais la branche maitresse n'est pas vraiment revenue au commit demandé, et les commits suivants n'ont pas été abandonnés.

Cette commande *si elle s'applique à des fichiers spécifiques, permet de retrouver une version antérieure, de la modifier et de l'intégrer au prochain commit* :

La séquence suivante montre cela :

```

PS C:\temp\projet> git checkout b778ea3 test.txt
PS C:\temp\projet> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   test.txt

Untracked files:
  (use "git add <file>..." to include in what will be committed)

    .gitignore

PS C:\temp\projet> git commit -a -m "Retour et modification de test.txt"
[master df67270] Retour et modification de test.txt
1 file changed, 1 insertion(+), 3 deletions(-)
PS C:\temp\projet> git log --oneline
df67270 (HEAD -> master) Retour et modification de test.txt
bb4499a Troisieme commit
abb8524 Second commit
b778ea3 Premier commit

```

On a ajouté le nom d'un fichier (test.txt) à la commande git checkout et nous l'avons récupéré, dans son état ancien, prêt à être modifié. Il a été ensuite rajouté correctement, via un commit, à la branche maitresse.

#### 8. Revenir dans le passé : utilisation de reset

Cette commande permet vraiment de revenir à une étape précédente du projet en modifiant l'historique et est donc assez dangereuse en cas de mauvaise utilisation, car on peut perdre des données...

Elle sert en fait à **plusieurs choses** :

- Vider le cache de la zone surveillée, sans affecter les fichiers. Pour les associer au prochain commit, il faut à nouveau faire un add. Dans ce cas, on ne revient pas à un commit précédant.
- Revenir à un commit précédant (en modifiant donc l'historique), en gardant les modifications actuelles en cache (mode "soft"), revenir à un commit précédant, en gardant les modifications actuelles, mais hors cache (mode "mixte", il faudra faire un add avant un nouveau commit) ou en perdant les modifications actuelles et en revenant strictement à l'état du commit choisi (mode "hard").

Pour la mettre en pratique dans son premier usage (vidage du cache), toujours dans un répertoire avec 3 commits (par exemple) :

```

PS C:\temp\test_git> git log --oneline
9fd4560 (HEAD -> master) troisieme commit
45a4f83 second commit
5339cb9 premier commit

```

Si on modifie un fichier test.txt, on observe :



```

PS C:\temp\test_git> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")

```

Utiliser la commande `git reset` produit bien le vidage du cache :

```

PS C:\temp\test_git> git reset
Unstaged changes after reset:
M       test.txt
PS C:\temp\test_git> git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")

```

Pour la mettre en pratique dans ses usages plus dangereux, on peut, dans le même dossier tenter de revenir au premier commit, en mode soft (par exemple) :

```

PS C:\temp\test_git> git log --oneline
9fd4560 (HEAD -> master) troisieme commit
45a4f83 second commit
5339cb9 premier commit
PS C:\temp\test_git> git reset 5339cb9 --soft
PS C:\temp\test_git> git log --oneline
5339cb9 (HEAD -> master) premier commit
PS C:\temp\test_git> git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   test.txt

```

Ou, après avoir committées les modifications en attente, même le retour à l'état exact après le premier commit :

```

PS C:\temp\test_git> git commit -m "apres le reset"
[master 2029855] apres le reset
1 file changed, 3 insertions(+), 1 deletion(-)
PS C:\temp\test_git> git log --oneline
2029855 (HEAD -> master) apres le reset
5339cb9 premier commit
PS C:\temp\test_git> git reset 5339cb9 --hard
HEAD is now at 5339cb9 premier commit

```

## 9. Création d'une branche et travail sur la nouvelle branche



Une branche permet d'explorer des variantes ou des solutions nouvelles, sans affecter la branche principale. On utilise les variantes de la commande `branch` pour cela :

```
PS C:\temp\test_git> # liste les branches
PS C:\temp\test_git> git branch
* master
PS C:\temp\test_git> # creation d'une branche
PS C:\temp\test_git> git branch "branche"
PS C:\temp\test_git> git branch
branche
* master
PS C:\temp\test_git> # travaille sur la branche 'branche'
PS C:\temp\test_git> git checkout "branche"
Switched to branch 'branche'
PS C:\temp\test_git> git branch
* branche
master
PS C:\temp\test_git> # premier commit sur la branche
PS C:\temp\test_git> git add --all
PS C:\temp\test_git> git commit -m "premier commit sur branche"
[branche 53a997b] premier commit sur branche
1 file changed, 2 insertions(+), 1 deletion(-)
PS C:\temp\test_git> git log --oneline
53a997b (HEAD -> branche) premier commit sur branche
5339cb9 (master) premier commit
PS C:\temp\test_git> # renommer la branche courante en "mabranche"
PS C:\temp\test_git> git branch -m "mabranche"
PS C:\temp\test_git> git branch
* mabranche
master
```

Si on revient sur la branche master, et que l'on y fait un commit :

```
PS C:\temp\test_git> git checkout master
Switched to branch 'master'
PS C:\temp\test_git> git commit -a -m "commit apres branche"
[master 75b2e38] commit apres branche
1 file changed, 1 insertion(+), 1 deletion(-)
```

On observe alors (ungit) :



## 10. Fusionner une branche avec le tronc principal

Après avoir travaillé sur une branche, on peut vouloir la fusionner sur le tronc et en récupérer les nouveautés et ensuite la supprimer. On devra :

- Se placer sur le tronc : `git checkout master`
- Demander la fusion : `git merge mabranche`

L'algorithme de la fusion sera alors choisi automatiquement par git :

- 'fast forward' : si aucun commit n'a eu lieu sur la branche principale, l'historique de la branche secondaire sera ajouté à celui de la branche principale.
- '3-way merge' : si des commits ont eu lieu sur la branche principale entre temps, une fusion plus complexe doit avoir lieu.

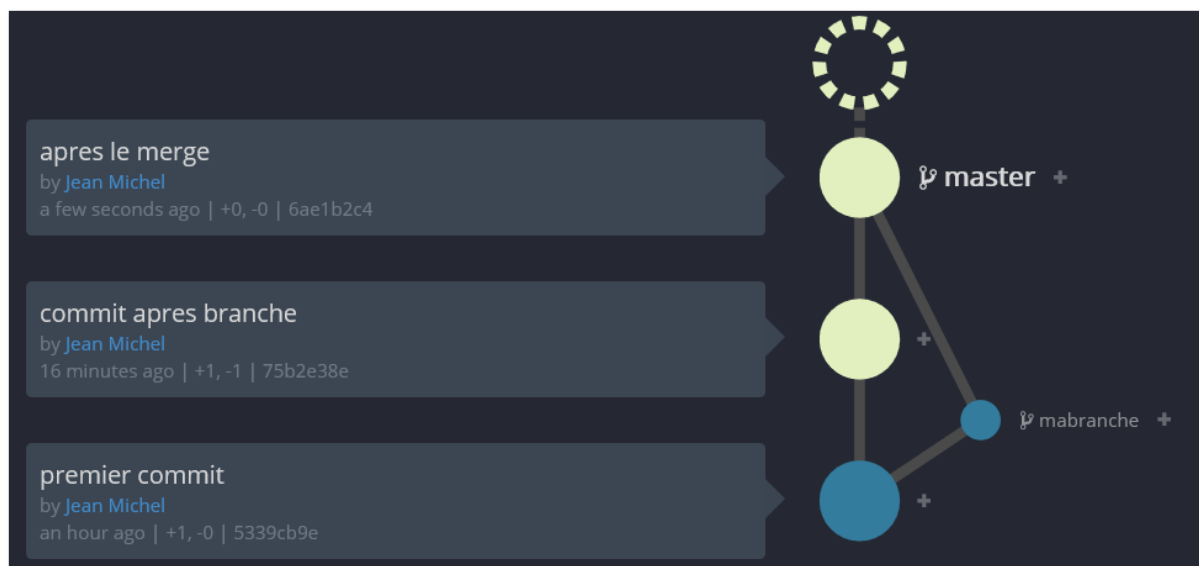
Au moment de la fusion des conflits peuvent donc apparaître et ils doivent être réglés à la main :

```
PS C:\temp\test_git> git branch
mabranche
* master
PS C:\temp\test_git> git merge mabranche
Auto-merging test.txt
CONFLICT (content): Merge conflict in test.txt
Automatic merge failed; fix conflicts and then commit the result.
```

Après résolution dans un éditeur des conflits marqués dans les fichiers par git, si on fait :

```
PS C:\temp\test_git> git commit -a -m "apres le merge"
[master 6ae1b2c] apres le merge
```

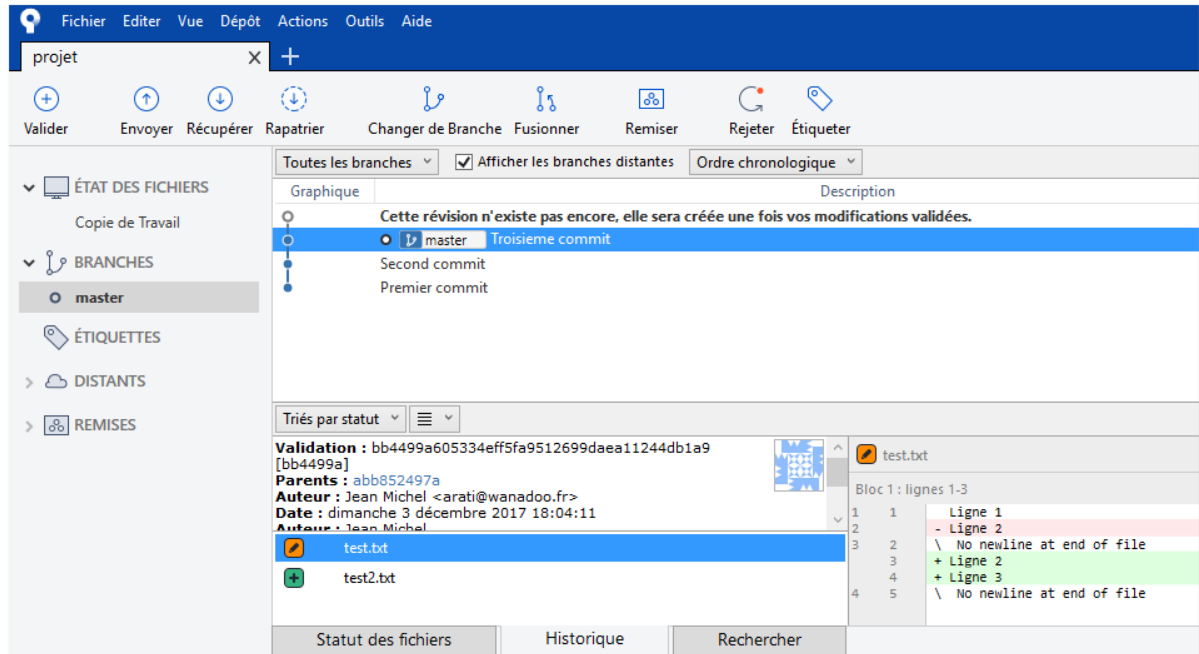
On observe dans ungit :



Après la fusion, on supprimera la branche par :

```
PS C:\temp\test_git> git branch -d mabranche
Deleted branch mabranche (was 53a997b).
```

## 11. Examen du repository avec un outil visuel comme ungit, ou SourceTree.



### En résumé :

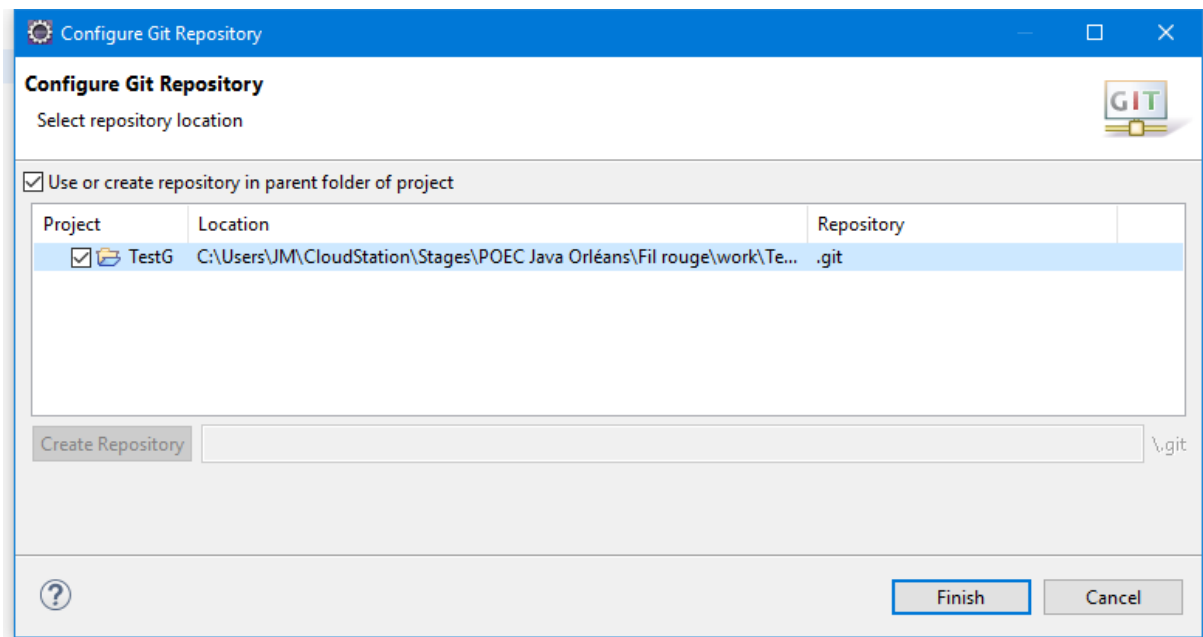
- **git init** : initialise un dépôt git dans un dossier
- **git add** : ajoute des modifications ou des nouveaux fichiers, dans la zone de 'staging'.
- **git status** : affiche le statut de l'état courant de la branche.
- **git log** : affiche l'historique des divers commits, avec de (très) nombreuses options.
- **git commit** : ajoute les modifications en attente au dépôt.
- **git diff** : affiche les différences entre fichiers entre des commits.
- **git checkout** : retrouve l'état de fichiers dans un commit précédant ou change de branche.
- **git revert** : annule les actions d'un commit passé.
- **git reset** : remet le projet dans l'état d'un commit précédent.
- **git branch** : crée une nouvelle branche.
- **git merge** : fusionne une branche au tronc principal.

Comme autres commandes utiles de git, citons `stash`, qui met de côté les dernières modifications du repository et revient au dernier commit, et `revert` qui annule les actions d'un commit précédent.

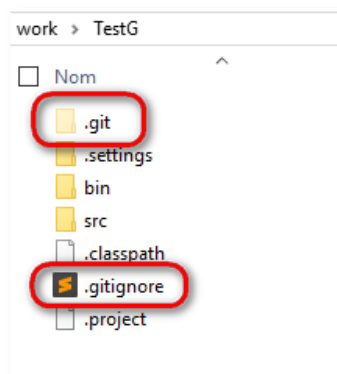
## 4 – Utilisation dans Eclipse

Eclipse contient le plugin EGit, donnant accès aux fonctionnalités de git, avec une représentation graphique commode.

Pour l'utiliser, il faut d'abord savoir qu'il n'est pas souhaitable de créer de dépôt git dans le workspace d'Eclipse, pour des questions de performances : le nombre de fichiers du workspace peut devenir énorme et ralentir de nombreuses opérations de git. Cependant, pour des questions de simplicité **on va procéder ainsi pour cet exemple**. On effectue un clic droit sur le projet (ici TestG), on choisit Team/Share... et on obtient (après avoir coché la case "Use or create repository in parent folder of project" et cliqué sur "Create Repository" :

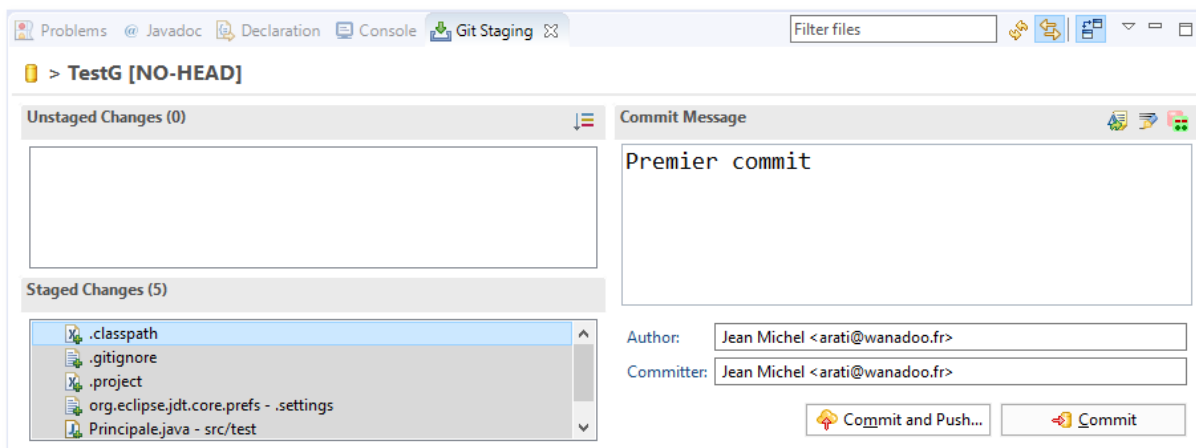


On clique ensuite sur Finish. En observant le dossier du projet on constate qu'il contient les éléments caractéristiques d'un dépôt git :

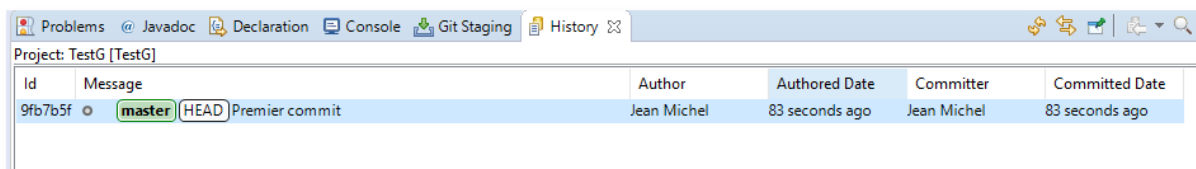


Le repository étant créé les options du menu Team sont nettement plus étoffées et on y reconnaît les grandes commandes de git : Commit, Stashes, Pull etc.

Si on souhaite faire un premier commit, on se place sur le projet et on sélectionne l'option "Add to index", ce qui ajoute les fichiers non 'trackés' en zone "staged", puis on choisit l'option Commit et on obtient :



Après avoir saisi un message on presse sur le bouton Commit.  
Si on choisit ensuite l'option "Show in history", l'historique apparait :



La documentation d'EGit, présente dans l'aide d'Eclipse détaille toutes les opérations de base de Git à travers cette interface.

## 5 – Utilisation distante

Il est possible de se connecter, via git, à un **dépôt distant** qui peut être un autre dossier local, être situé sur un serveur du même réseau, ou encore sur l'Internet, comme GitHub ou Bitbucket.

Cela permet de partager un projet entre plusieurs développeurs, chacun pouvant y apporter ses modifications et les publier dans un repository centralisé.

### 5.1 – Principe

Les étapes de mise en œuvre sont :

1. La création du dépôt distant via la commande `git init -bare`, qui initialise un dépôt, sans qu'il ne soit un répertoire de travail. On ne peut pas l'utiliser comme un dépôt Git normal, mais seulement y faire des pushes, des pulls ou le cloner.
2. Depuis un dépôt git local "classique", avec des fichiers de projet, des commits, des branches etc., on va l'ajouter aux dépôts distants "vus" par ce dépôt local. On utilise la commande `git remote` pour cela.
3. Lorsqu'on le souhaitera, on pourra publier, le dépôt local vers le dépôt distant, en faisant une commande `git push`. On peut choisir la branche à pousser.

4. Inversement, on peut récupérer le contenu du projet présent sur le dépôt distant vers le dépôt courant, en faisant un `git pull`.
5. Enfin, si on souhaite créer un nouveau dépôt local, on peut cloner le dépôt distant vers ce dépôt local, créé à la volée, en faisant un `git clone`.

## 5.2 – Exemple de mise en œuvre

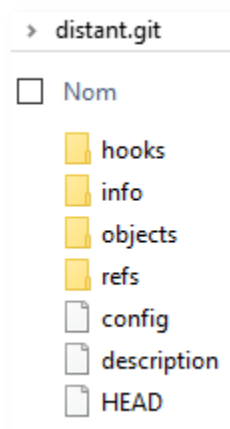
1. On commence par créer un dépôt git local avec 2 ou 3 commits d'un fichier `test.txt`. Appelons ce dépôt `test-git` :

```
PS C:\temp\test-git> git log --oneline
86487e5 (HEAD -> master) Troisieme commit
75bb9bf Second commit
8e70c72 Premier commit
```

2. Ensuite, on crée un nouveau dossier appelé `distant.git` (c'est la tradition d'appeler les dossiers de dépôts 'bare' avec l'extension `.git`), on s'y déplace, et on y lance la commande `git init --bare` :

```
PS C:\temp\distant.git> git init --bare
Initialized empty Git repository in C:/temp/distant.git/
PS C:\temp\distant.git>
```

On constate que les fichiers/dossiers présents d'habitude dans le dossier `.git` se trouvent placés directement dans ce répertoire :



3. De retour dans notre dépôt local `test-git`, on y définit `distant.git` comme dépôt remote, de nom d'alias `depot-distant` :

```
PS C:\temp\test-git> git remote -v
PS C:\temp\test-git> git remote add depot-distant C:\temp\distant.git
PS C:\temp\test-git> git remote -v
depot-distant C:\temp\distant.git (fetch)
depot-distant C:\temp\distant.git (push)
```

La commande `git remote -v` servant à lister les dépôts distants définis pour ce repository.

4. On peut alors **pousser le dépôt local vers le distant** :

```
PS C:\temp\test-git> git push depot-distant master
Counting objects: 9, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (9/9), 666 bytes | 166.00 KiB/s, done.
Total 9 (delta 0), reused 0 (delta 0)
To C:\temp\distant.git
* [new branch]      master -> master
PS C:\temp\test-git> git branch -r
depot-distant/master
```

La commande `git push` reçoit l'alias du dépôt distant et la branche à pousser. La commande `git branch -r` liste les branches distantes.

5. Si dans le dépôt distant, on fait après cela un `git log --oneline` on obtient bien notre historique :

```
PS C:\temp\distant.git> git log --oneline
86487e5 (HEAD -> master) Troisieme commit
75bb9bf Second commit
8e70c72 Premier commit
```

6. Si on crée une autre branche locale, on peut la pousser vers le dépôt distant avec la même commande, et comme on peut supprimer cette branche localement, on pourra la supprimer à distance :

```
PS C:\temp\test-git> git branch mabranche
PS C:\temp\test-git> git branch
mabranche
* master
PS C:\temp\test-git> git push depot-distant mabranche
Total 0 (delta 0), reused 0 (delta 0)
To C:\temp\distant.git
* [new branch]      mabranche -> mabranche
PS C:\temp\test-git> git branch -r
depot-distant/mabranche
depot-distant/master
PS C:\temp\test-git> git branch -d mabranche
Deleted branch mabranche (was 86487e5).
PS C:\temp\test-git> git push depot-distant --delete mabranche
To C:\temp\distant.git
- [deleted]          mabranche
PS C:\temp\test-git> git branch -r
depot-distant/master
```

- `git branch mabranche`, crée la branche 'mabranche' localement.
- `git push depot-distant mabranche` pousse la branche locale vers le dépôt distant.
- `git branch -d mabranche` supprime la branche locale 'mabranche'.
- `git push depot-distant --delete mabranche` supprime la branche distante 'mabranche'.

7. Inversement, si quelqu'un d'autre avait poussé sur le repository distant ses modifications, on aurait dû utiliser la commande pull pour les récupérer :

```
PS C:\temp\test-git> git pull depot-distant master
From C:\temp\distant
 * branch          master      -> FETCH_HEAD
Already up to date.
PS C:\temp\test-git>
```

8. La commande **git clone** permet, quant à elle, de **cloner un dépôt distant dans un nouveau dossier local, créé lors du clonage** :

```
PS C:\temp> git clone C:\temp\distant.git clone-depot
Cloning into 'clone-depot'...
done.
PS C:\temp> cd clone-depot
PS C:\temp\clone-depot> git remote -v
origin C:\temp\distant.git (fetch)
origin C:\temp\distant.git (push)
PS C:\temp\clone-depot> git log --oneline
86487e5 (HEAD -> master, origin/master, origin/HEAD) Troisieme commit
75bb9bf Second commit
8e70c72 Premier commit
```

La commande `git clone` reçoit **le chemin du dépôt distant à cloner** (ici `c:\temp\distant.git`) et **le nouveau répertoire à créer contenant le clone** (ici `clone-depot`). On observe que, dans le nouveau clone, **un alias appelé 'origin' du dépôt distant a été créé**, et que **tout l'historique est bien là**.

En général on ajoutera à la ligne de commande du clone le paramètre `--depth` où `n` vaudra le nombre de commits que l'on souhaite récupérer.

### 5.3 – Et Bitbucket / Github ?

Ce sont des services en lignes (avec des politiques tarifaires différentes) permettant de publier des dépôts git de projets publiques et privés et d'en permettre la collaboration à l'échelle globale.

Gitlab offre un service similaire, à l'intérieur d'une entreprise.

## 6 – Aller plus loin avec Git

Il existe d'innombrables tutoriaux sur le Web concernant Git et même un livre complet gratuit, "Pro Git 3" traduit en français (!). On le trouvera à <https://git-scm.com/book/fr/v2>.

En ce qui concerne les tutoriels, citons (entre autres !) :

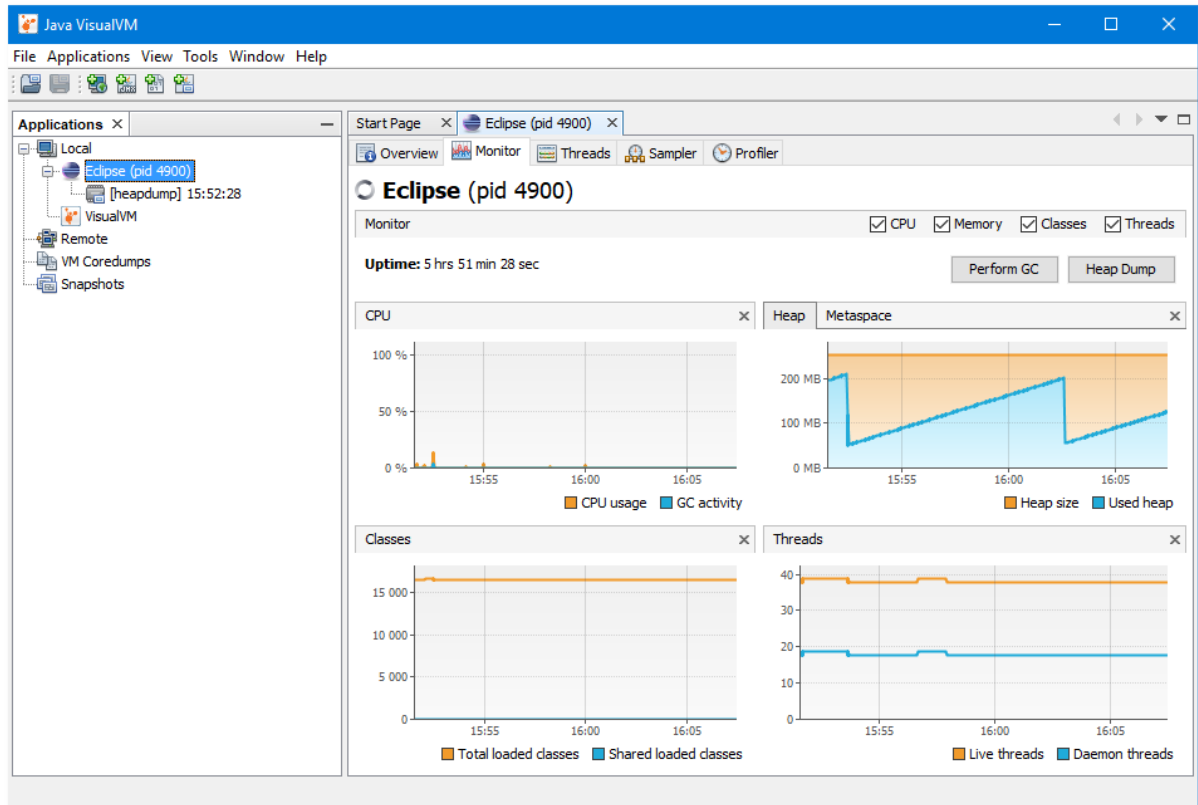
- Celui d'atlassian : <https://www.atlassian.com/git/tutorials>.
- Celui d'open classroom : <https://openclassrooms.com/courses/gerez-vos-codes-source-avec-git>



## IV – Autres outils utiles

### 1 – Outils graphiques du JDK : JVisual VM (jvisualvm.exe)

Cet outil permet un monitoring graphique de l'activité d'une ou plusieurs JVM



Il possède une vingtaine de plugin téléchargeables (option Tools / Plugins... et onglet Available Plugins) dont un analyseur d'activité du garbage collecteur (VisualGC).

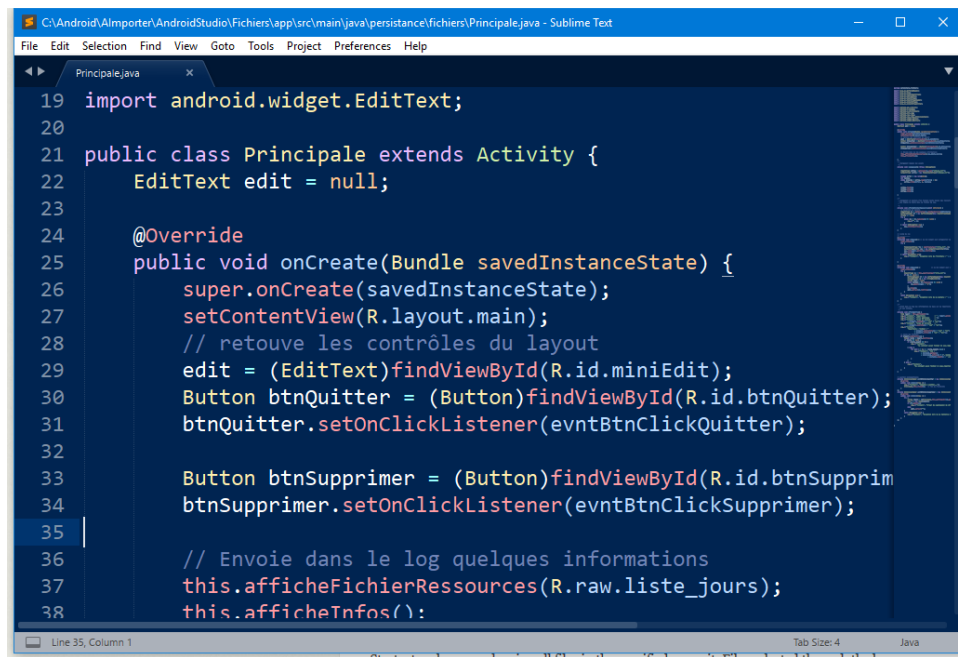
JM Doudoux a fourni une exploration détaillée de JVisualVM à

<https://www.jmdoudoux.fr/java/dej/chap-outils-jdk.htm>.

### 2 – Editeurs de code

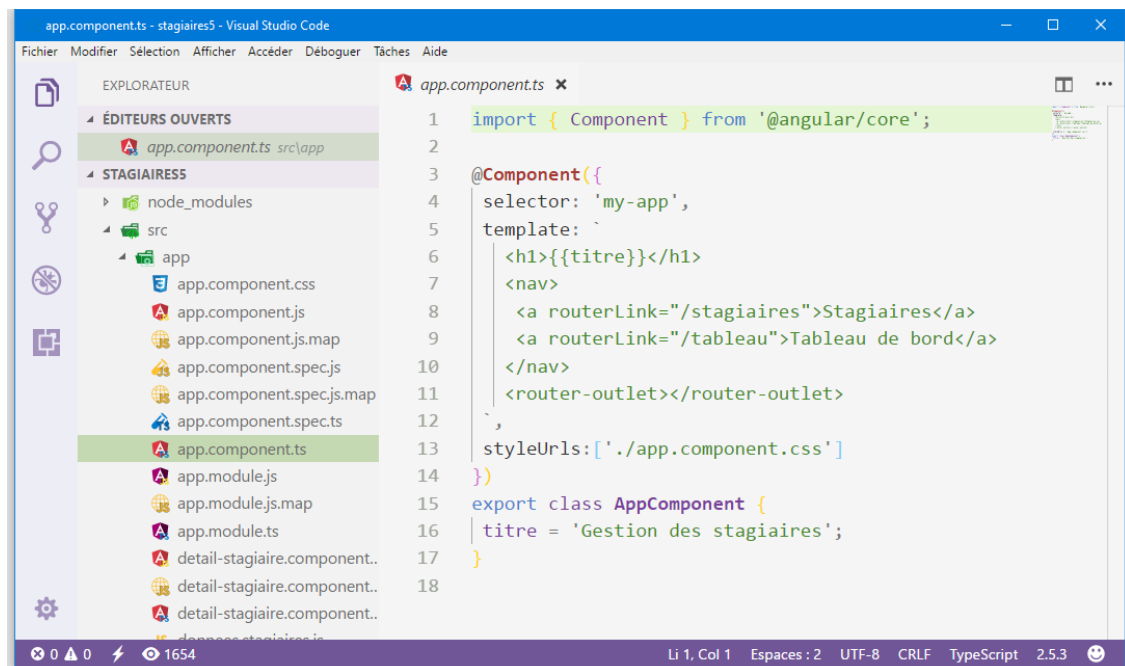
Un éditeur de code est un outil indispensable au développeur, en plus d'un IDE. IL en existe de très nombreux gratuits. Sous Windows, en plus du Notepad, notoirement insuffisant, on pourra trouver Notepad++, SublimeText, Atom et de nombreux autres. On va présenter rapidement ici Sublime Text et VSCode.

#### 2.1 - SublimeText



Excellent éditeur généraliste multiplateformes, théoriquement payant, mais utilisable gratuitement sans limitation de durée. Très rapide et efficace.

## 2.2 – VSCode



Editeur de code gratuit et efficace réalisé par Microsoft, multiplateformes (Windows, Linux, Mac) et supportant de nombreux langages et types de développements via des plugins.

Cet éditeur est très commode pour les développement Node / TypeScript et Angular. Il supporte de nombreux autres langages (Java, C#, Python etc.).