Cours : GIT - GITHUB

I. Introduction

1. GitHub Kézako ?

- GitHub est un service web d'hébergement et de gestion de développement de logiciels, utilisant le logiciel de gestion de versions **Git**.
- Le nom **GitHub** est composé du mot :
  - « git » : référence à un système de contrôle de version open-source.
  - « hub » faisant référence au réseau social bâti autour du système Git.
- **GitHub** est juste le plus grand espace de stockage de travaux collaboratifs dans le monde.
- **GitHub** est aussi un réseau social comme Facebook.
- On crée un profil, on y dépose des projets à partager et on se connecte avec d'autres utilisateurs en suivant leurs comptes.

2. Git kizako ?

- **Git** est le logiciel qui fait tourner le cœur de **GitHub**.
- **Git** est un logiciel de contrôle de version ou **VCS** (**Version Control System** – système de gestion de version).
- il gère les modifications d'un projet sans écraser n'importe quelle partie du projet.
- **Pourquoi utiliser quelque chose comme Git ?**
  - Supposons que vous mettiez à jour avec un collègue des pages sur le même site web. Vous faites des modifications, vous les sauvegardez et les versez sur le site. À

ce stade, tout va bien. Le problème survient quand votre collègue travaille sur la même page que vous en même temps. L'un de vous va voir son travail écrasé.

- Une application de contrôle de version comme Git empêche ça d'arriver.
- Vous et votre collègue pouvez chacun de votre côté verser vos révisions sur la même page, et Git sauvegardera deux copies.
- Plus tard, vous pourrez fusionner vos modifications sans perdre le travail dans le processus. Vous pouvez même revenir en arrière à tout moment, parce que Git conserve une "copie instantanée" de tous les changements produits.

- Le problème avec **`Git`** est qu'il est vieux, si vieux que nous devons utiliser la ligne de commande.
  - Pour éviter la ligne de commande, on utilise GitHub.
- **`GitHub`** facilite l'utilisation de **`Git`** sur deux points :
  - En téléchargeant le logiciel GitHub sur votre ordinateur, une interface visuelle est fournit pour vous aider à gérer localement vos projets avec les contrôles de version.
  - créer un compte sur https://github.com/ emmène vos projets sur le web avec des contrôles de versions, et leur attache des fonctionnalités de réseaux sociaux.
- On peut naviguer sur les projets d'autres utilisateurs de **`GitHub`**, télécharger des copies pour nous afin de les modifier, apprendre ou les enrichir.
- D'autres utilisateurs peuvent faire de même avec vos projets publics, repérer vos erreurs et vous suggérer des corrections.
- De toute façon, aucune donnée ne se perd parce que Git sauvegarde un "instantané" de chaque modification.

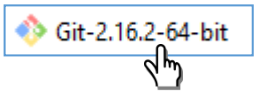
  - Il est possible d'utiliser GitHub sans apprendre Git mais si on veut comprendre comment fonctionne GitHub, on doit apprendre et connaitre Git.

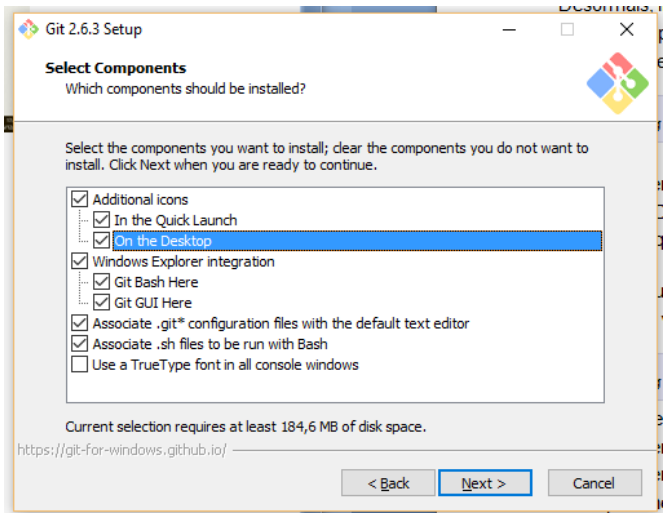
II. Téléchargement et Installation de GIT

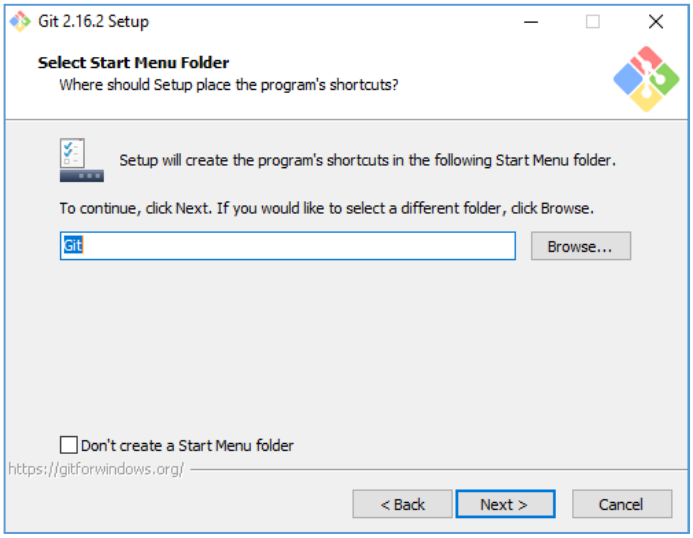
- Pour pouvoir travailler sur un projet sur notre ordinateur local, on doit télécharger et installer git en local :

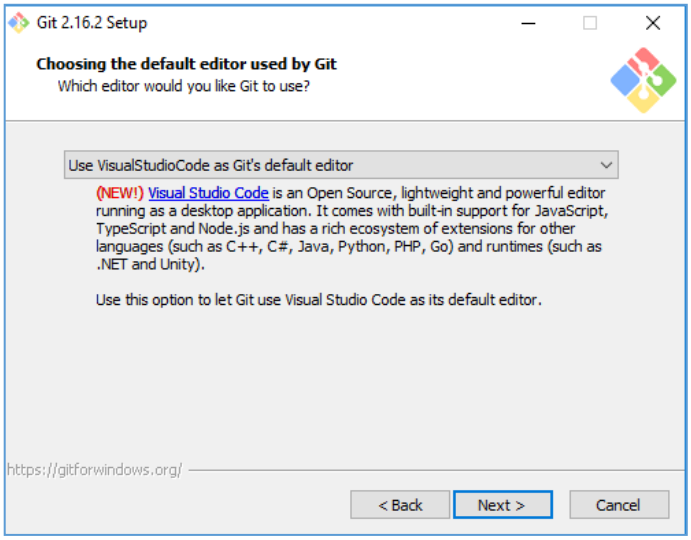
  - On va télécharger GIT sur le site : https://git-scm.com/downloads

  - On va maintenant l'installer :

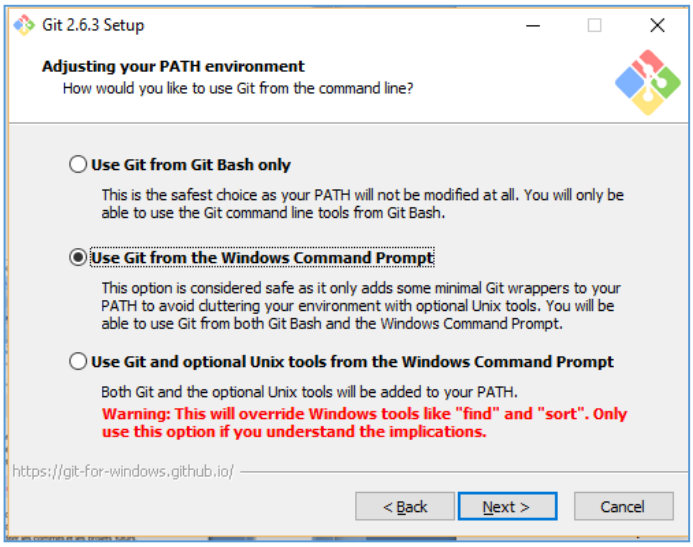


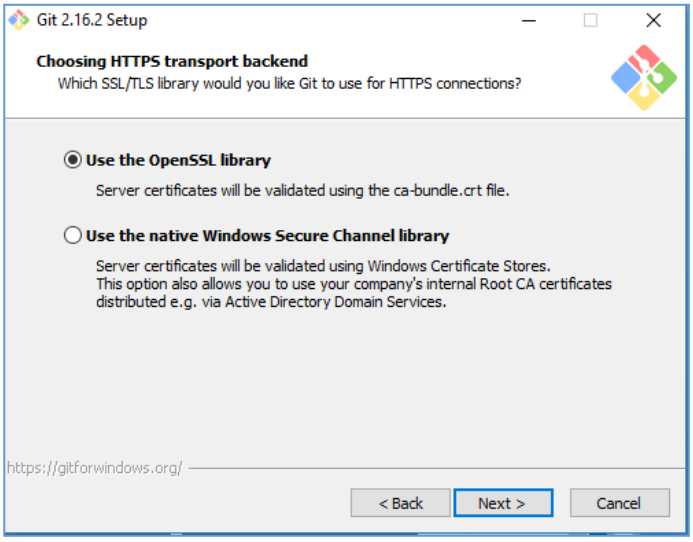


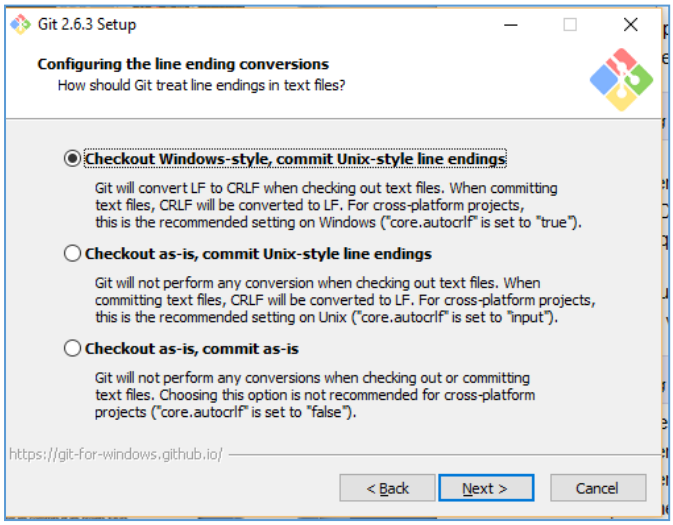


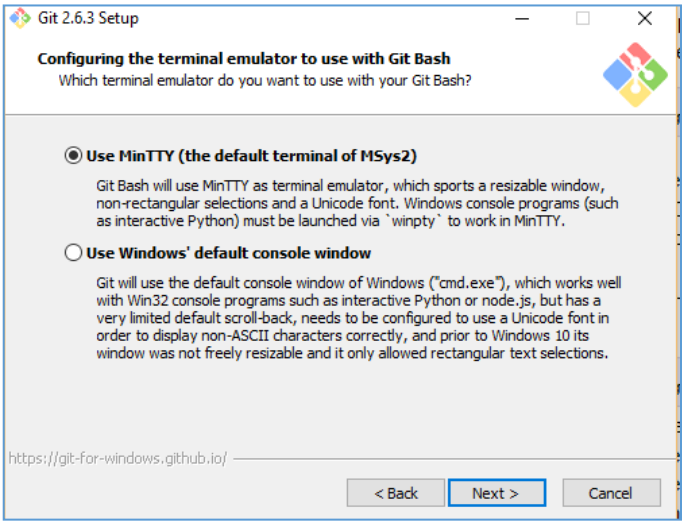


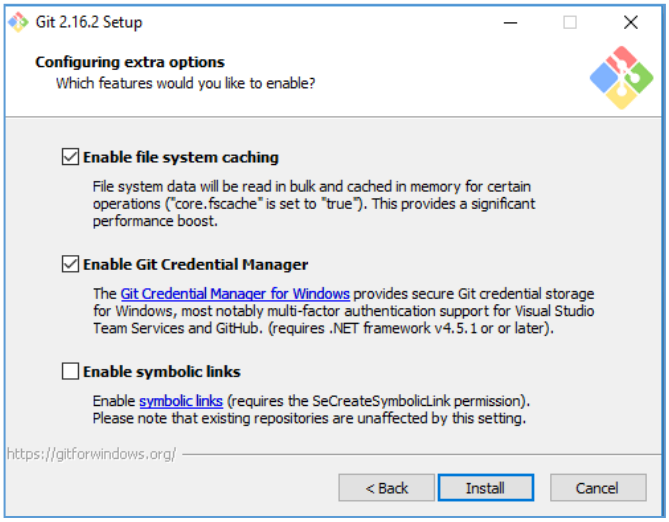
- On choisit NotePad++

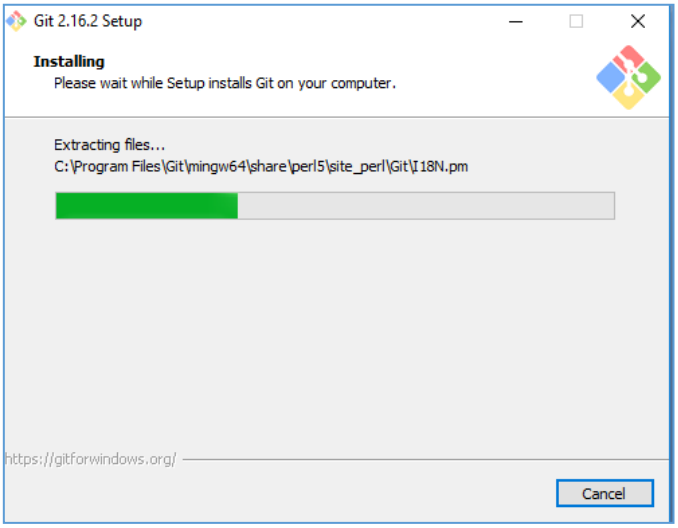


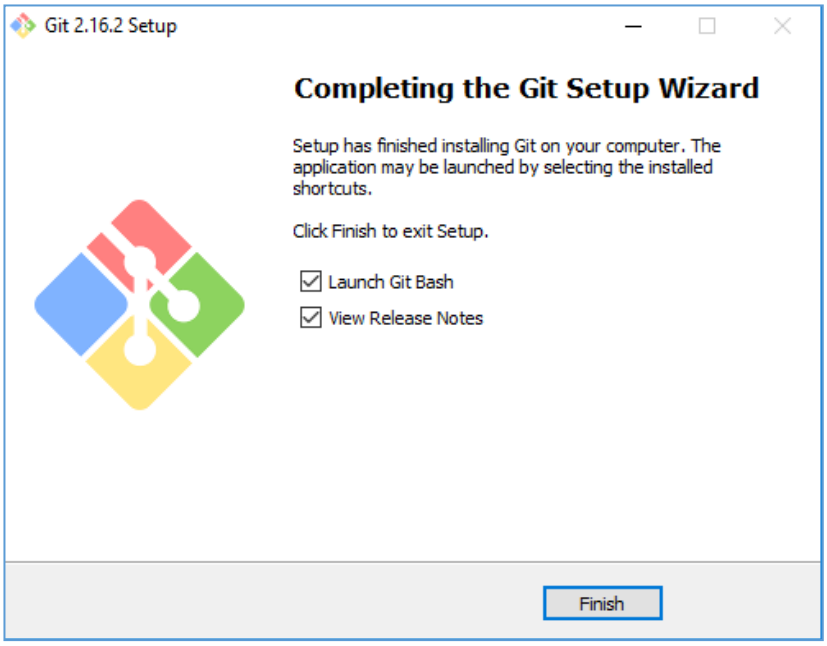




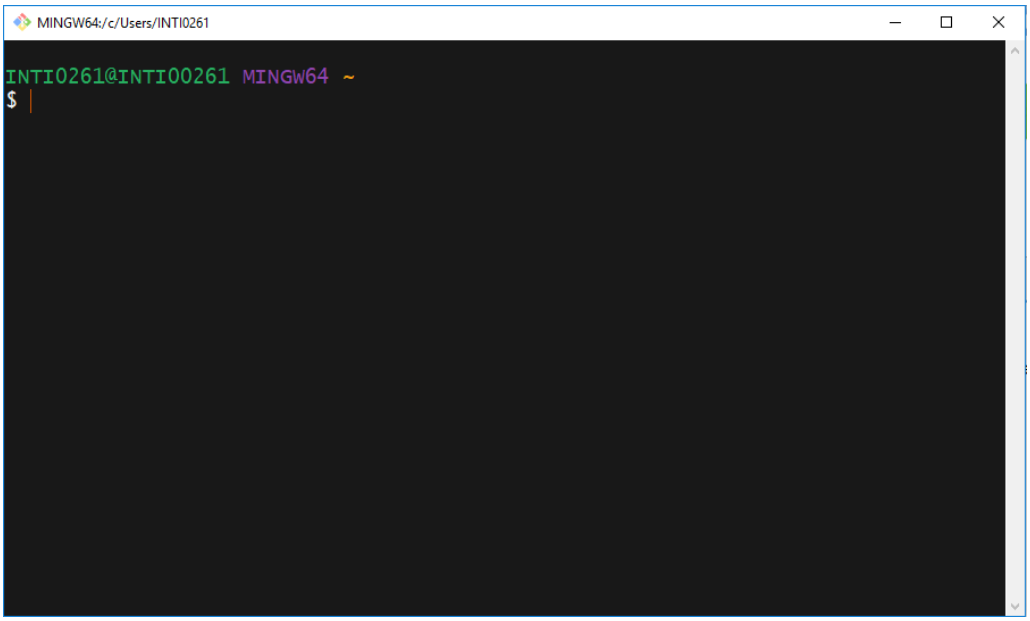








- La console de ligne de commande est lacée :



➡ On peut la démarrer avec l'app Git Bash dans le menu démarrer

III. Débuter avec GIT

1. Indexer les changements

- Nous allons maintenant commencer à manipuler git :
  - On va commencer par une 1ere commande : **`git`**
    - Git me ressort une documentation, ce qui veut dire que Git est correctement installé et on peut l'utiliser.

```
MINGW64:/c/Users/INTI0261
INTI0261@INTI00261 MINGW64 ~
$ git
usage: git [--version] [--help] [-C <path>] [-c name=value]
           [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
           [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
           [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
           <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
   clone      Clone a repository into a new directory
   init       Create an empty Git repository or reinitialize an existing one

work on the current change (see also: git help everyday)
   add        Add file contents to the index
   mv         Move or rename a file, a directory, or a symlink
   reset      Reset current HEAD to the specified state
   rm         Remove files from the working tree and from the index

examine the history and state (see also: git help revisions)
   bisect     Use binary search to find the commit that introduced a bug
   grep       Print lines matching a pattern
```

  - 2 commandes de base :

```
git --version
```

Pour voire la version du Git installée

```
git --help
```

Qui va nous renvoyer à la documentation de base

```
git --help
```

Si on la précède d'un nom de commande, on aura une explication sur l'utilisation de la commande :

```
git commit --help ou git commit -h (affiche dans la console)
```

- La commande **`config`** nous permet de configurer git :

```
git config -h
```

```
MINGW64:/c/Users/INTI0261

INTI0261@INTI00261 MINGW64 ~
$ git config -h
usage: git config [<options>]

Config file location
    --global              use global config file
    --system              use system config file
    --local               use repository config file
    -f, --file <file>     use given config file
    --blob <blob-id>      read config from given blob object
```

- La commande **`config`** va nous permettre de définir le comportement de git pour tous les utilisateurs **`--global`**, pour mon poste de travail **`--system`** ou uniquement pour le dépôt ou on est.

- Nous on va définir nos variable au global puisqu'on seul à travailler sur le poste, et on va définir le username :

```
git config --global user.name "eladib dhiaeddine"
```

- On réalise la même opération pour le mail :

```
git config --global user.email "eladibdhiaeddine@gmail.com"
```

- C'est les 2 notions fondamentales que Git à besoin pour nous identifier en tant que développeur et donner une adresse en cas ou un autre développeur veut nous contacter.

- Notre configuration Git est terminée, il sait qui on est, il sait nous contacter.

- On peut commencer à l'utiliser.

Pour définir l'éditeur de texte à utiliser par défaut :

```
git config --global core.editor "Notepad++"
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git config --global core.editor "Notepad++"
```

- On va initialiser notre premier projet :
  - On crée un répertoire **`coursGit`** :

```
INTI0261@INTI00261 MINGW64 ~
$ mkdir ~/coursGIT
```

`[mkdir]` : "make directory".

`[~/]` : Nous assure que nous construisons le dépôt tout en haut de la structure du fichier De notre ordinateur, au lieu d'un répertoire coincé dans quelque autre répertoire qui serait plus difficile à retrouver.

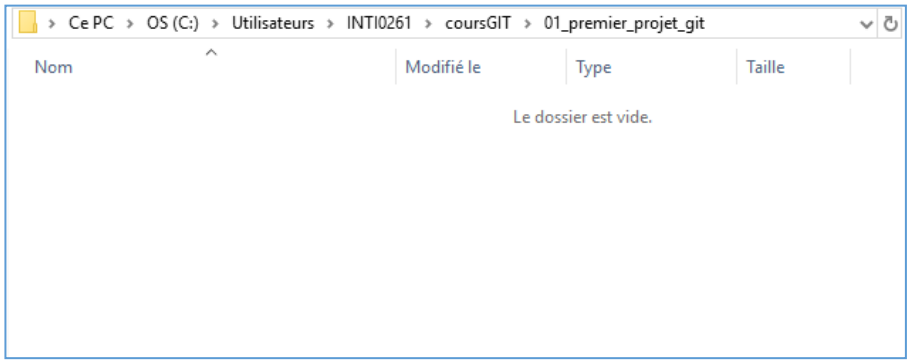
le répertoire est crée sur : `[C:\Users\dheladib\coursGit]`

- On accède au répertoire, on crée un autre 01_premier_projet_git et on y accède :

```
INTI0261@INTI00261 MINGW64 ~
$ cd ~/coursGIT
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT
$ mkdir 01_premier_projet_git

INTI0261@INTI00261 MINGW64 ~/coursGIT
$ cd 01_premier_projet_git/

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git
$
```

- On obtient un répertoire vide :



- On va lancer la commande **`git init`**:
  - Cette commande nous permet d'initialiser un dépôt et d'être prêt à travailler avec git.

```
git init
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git
$ git init
Initialized empty Git repository in C:/Users/INTI0261/coursGIT/01_premier_projet
_git/.git/

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$
```

- git nous dit qu'il a initialisé un depot vide à cet emplacement.

- Si on se rende dans notre répertoire, on a un dossier caché **`.git`** qui contient toutes les informations utiles à la mécanique de git, pour stocker des informations sur le projet lui-même

```
git init nomRepertoire
```

Aura pour effet de créer le répertoire **`nomRepertoire`** et d'initialiser un dépôt git dans ce répertoire.

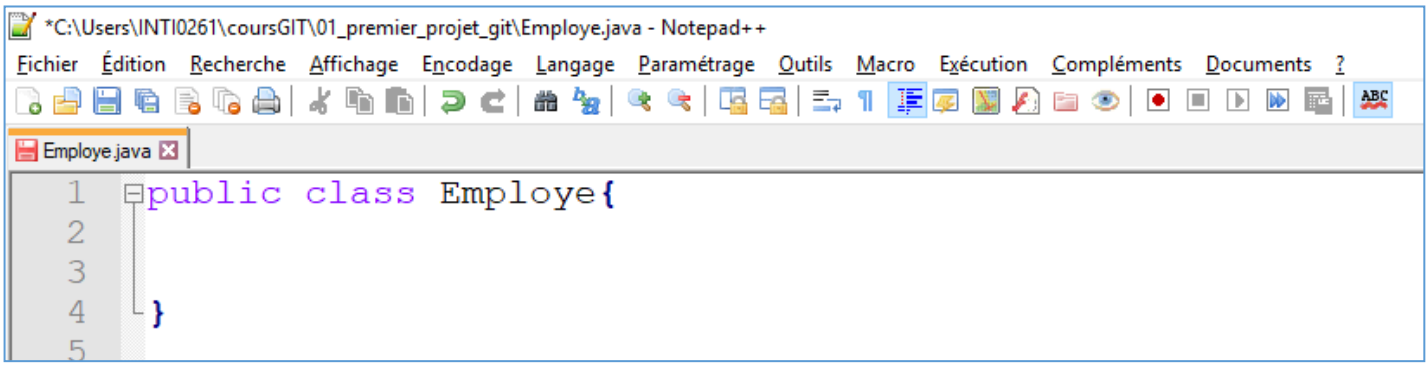
- Le projet est maintenant initialisé, git va commencer à surveiller le système de fichiers et va être à l'écoute des différentes commandes qu'on va lui donner.

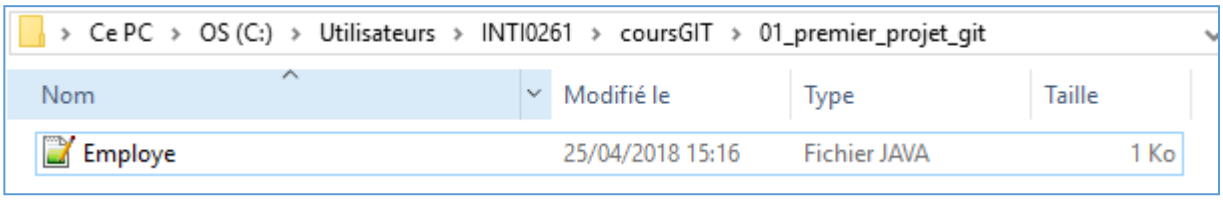
➡ Notre dépôt est prêt à recevoir les codes source.

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$
```

`master` : le nom de la branche sur laquelle on est entrain de travailler. On reviendra dessus après.

- Maintenant, on créer un fichier source java **`Employe.java`** avec NotePad++ :





- On a finis de travailler sur notre première version du projet, on la testée, elle est stable, elle fonctionne correctement et on souhaite la versionner.

- On tape **`ls`** et on voit bien notre fichier source :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ ls
Employe.java

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$
```

- Qu'en est il au niveau de GIT ?

  Il ya une commande récurrente qu'on va utiliser massivement qui s'appelle **`git status`**

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master

No commits yet

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

        Employe.java

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

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ |
```

- La commande nous renseigne sur le status actuel du dépôt :
  - Git nous informe qu'on est sur la branche initiale
  - On est sur le commit initial, ce qui veut dire que le référentiel de version est complètement vide
  - On a des fichiers qui sont dit **`Untracked`** **:** ils ne sont pas référencés ou indexés par **`git`**

- La particularité de Git est qu'il fait appel à un système à étape pour gérer ses fichiers.

- En tant que tel ici, le fichier **`Employe.java`** n'est pas connu du référentiel de git et git ne prendra l'initiative de versionner ce fichier la.

- On est de notre responsabilité à nous développeurs d'enregistrer manuellement ce fichier auprès de git et de lui dire : git, ce fichier doit être versionné et tu dois le surveiller et mettre par la suite les outils pour le tracker de version en version.

  - Donc pour ça, on va devoir utiliser la commande **`git add`** **:**

```
git add Employe.java
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git add Employe.java
```

- on ajoute au référentiel de git le fichier source.
  - On vérifie avec **`git status`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master

No commits yet

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

        new file:   Employe.java
```

- On est toujours sur la branche master

- Qu'est ce qu'il sera commité : Commité veut dire, qu'est ce qu'il va être versionné la prochaine fois qu'on va exécuter un commit, la prochaine fois qu'on va enregistrer au sens git du terme, une nouvelle révision, une nouvelle version.

- Ici Git nous informe qu'on déclarera qu'une nouvelle version doit être enregistrée, commitée, on aura un nouveau fichier qui s'appel Employe.java qui va être enregistré.

`git status` nous explique ce qui va se passer lors de notre prochaine révision :

- Le fait d'ajouter à la main de nombreux fichiers peut vite s'avérer rébarbatif surtout si on a de nombreux dossiers, sous dossiers, …etc

- Dans ce cas la, on a un raccourcis :

```
git add .
```

qui va ajouter tous les fichiers qui ne sont pas encore indexés dans le référentiel git.

Exemple :

- On ajoute 2 fichiers *readme.md* | *test.log* avec la commande **touch** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ touch test.log

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master

No commits yet

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

        new file:   Employe.java

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

        readme.md
        test.log
```

- il nous dit qu'il ya 2 fichiers qui sont pas indexés par git

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git add .

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master

No commits yet

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

        new file:   Employe.java
        new file:   readme.md
        new file:   test.log
```

- Git référence tous les fichiers.
- Ça peut être une manière assez pratique mais relativement dangereuse pour ajouter tous les fichiers qui ne sont pas encore référencés.
- Pour plus de sécurité on peut faire **`git add -p`** qui ajoute les fichiers d'une manière interactive et nous poser la question à chaque fois oui non pour l'indexation.

2. Commiter les changements

- Maintenant que nos nouveaux fichiers sont indexés, connus de Git, on va devoir activer le processus de révision et donc marquer dans Git une nouvelle version, une nouvelle révision qui va enregistrer, faire une photocopie à l'instant T de nos fichiers et nous permettre de revenir en arrière ou de consulter notre historique à postériori.

- Pour ça, on utilise la commande **`commit`**.

- Le commit, c'est cette action de prendre le répertoire de travail et de faire une copie carbone que l'on va conserver que l'on appellera révision ou version.

  Bien sure dans Git, il est de notre responsabilité en tant que développeur d'informer Git quel fichier il va devoir s'occuper et de quel fichier modifié il va devoir enregistrer dans sa prochaine révision.

  Ici Git se contente de mettre dans sa prochaine révision ce qu'on lui a dit d y mettre, donc les nouveaux fichiers, les nouveaux changements.

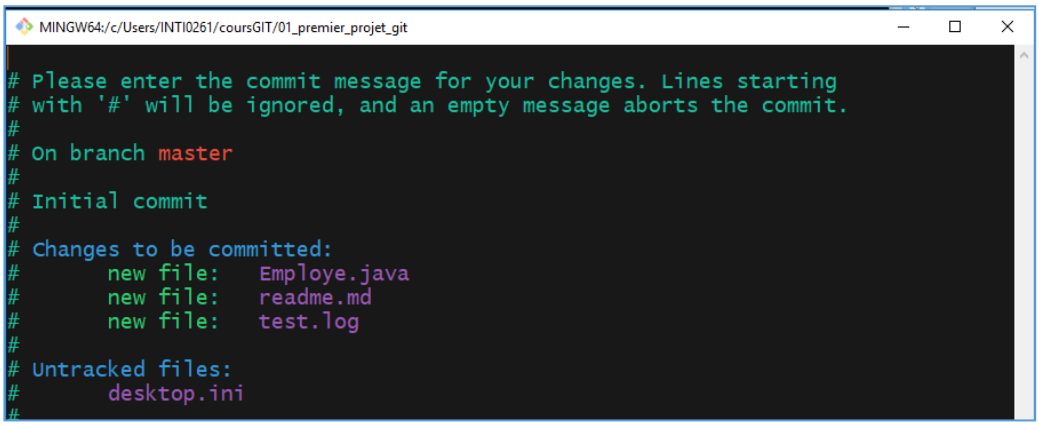
- Si on tape :

```
git commit
```

- Git ouvre l'éditeur par défaut, ici **vi** et nous demande d'ajouter un commentaire au commit.

- Un commit n'a pas de sens pour Git s'il n'est pas commenté.

- Le commentaire, c'est quelque chose de très succin qui va expliquer ce qu'on a fait.



On ajoute un commentaire :

```
MINGW64:/c/Users/INTI0261/coursGIT/01_premier_projet_git
création du projet, prmière implémtation de la classe Employe
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
#
# On branch master
#
# Initial commit
```

- On voit ici que la coloration syntaxique change à partir du 150eme caractère.
- Git va considère qu'un message d'un commit trop long est un problème.
- Si un message est trop long, certainement il y aurait pu y avoir 2 commits.
- Puisque c'est l'une des particularités de GIT, étant donner qu'on n'est pas en interaction direct avec le serveur distant, on va pouvoir nous développeur faire autant de commit que l'on veut et choisir le moment ou on va synchroniser avec les autres.
- C'est-à-dire ici, on peut dire à chaque fois qu'on fait une modification importante, on implémente une fonction, on corrige un bug, on va faire un commit et on peut faire des commit toutes les 2 minutes par exemple.

  L'ensemble des commits sera poussés ensuite vis-à-vis des autres développeurs.

  Etant donné qu'on est en local, il n ya pas de risque suite à un commit, un développeur récupère une version à moitié implémentée, c'est uniquement vis-à-vis de notre référentiel local.
- Ici par exemple, on aurait du passer par 2 commits, le 1er pour la création du projet et le 2eme pour l'implémentation de la classe Employe

  - Une fois qu'on terminer d'écrire le commentaire, on sauvegarde : Echap -> (shit+zz)

```
[master (root-commit) b789086] création du projet, prmière implémtation de la cl
asse Employe.
 3 files changed, 5 insertions(+)
 create mode 100644 Employe.java
 create mode 100644 readme.md
 create mode 100644 test.log
```

- Git nous informe que
  - 3 fichiers on été modifiés
  - qu'il y a eu 5 lignes d'insérées
  - et qu'il a crée 3 nouveaux objets à surveiller (Employe.java, readme.md et test.log)

    - si on fait un **git status** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

- On a plus rien puisque au sens Git, une fois que le dépôt local à été mis à jour, pour lui il y a pas eu de nouveau changement, les changements on été historisés, donc on peut passer à la suite.

  - Maintenant, On va continuer notre développement et ajouter du code à la classe Employe et on sauvegarde :

```
Employe.java
1  public class Employe{
2
3      private String nom;
4      private String prenom;
5      private String mail;
6
7  }
```

  - On fait un **git status** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ 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:   Employe.java

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

- Il nous informe qu'il ya un fichier comme avant mais un fichier qui a été modifié.

- Notre fichier été précédemment à l'état validé

- Ici le fichier est en l'état modifié (modified)

- Pour autant, si on fait un commit dans cet état la, il se passera rien, puisque avant de pouvoir faire un commit, il faudra qu'on valide nos modifications, qu'on passe nos fichiers à l'état indexés pour qu'ils puissent être embarqués dans la prochaine révision.

  - Dans ce cas on utilise la même commande utilisée précédemment **git add**

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git add Employe.java
```

- Ça va avoir comme effet de passer le fichier de l'état modifié à l'état indexé (du rouge au vert)

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Employe.java
```

- Git nous informe que lors du prochain commit, il va ajouter les modifications de ce nouveau fichier la.

Une notion fondamentale

- ✘ Il est de notre responsabilité de développeur d'inclure à la main qu'on veut versionner.
- ✘ Il est également de notre responsabilité d'inclure à la main toutes les modifications qu'on veut embarquer dans la révision.

  - Maintenant on va faire un **`commit`** et plutôt que de passer par l'éditeur pour ajouter le commentaire, on l'ajoute dans la commande avec **`-m`**

```
git commit -m "ajout des champs de la classe Employe"
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git commit -m "ajout des champs de la classe Employe"
[master 02be8fd] ajout des champs de la classe Employe
 1 file changed, 4 insertions(+), 1 deletion(-)
```

- Git nous fait le comte rendu du commit :

- Il nous informe qu'un fichier à été modifié,
- Qu'il ya 4 lignes de modifiée
- Qu'il y a 1 ligne de supprimée

- Ici on voit quelque chose d'intéressant :

```
[master 02be8fd] ajout des champs de la classe Employe
```

- Le message du commit
- et le hash, l'identifiant unique qui nous permettre de remonter l'historique si on veut comparer dans 6 mois la version actuelle avec celle crée à l'époque.

- Une dernière petite astuce : si on remodifie notre classe Employe

```java
Employe.java
public class Employe{

    private String nom;
    private String prenom;
    private String mail;
    private String telephone;

}
```

- Le **`git status`** nous informe qu'un fichier à été modifié mais pas indexé :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ 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:   Employe.java

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

- Si on fait un **`commit`**, il ne se passera rien.

- On peut néanmoins indiquer au commit lors de l'ajout du commentaire avec l'option **`-a`** pour indiquer que tous les fichiers modifiés vont être passé à l'état indexés avant de lancer le commit

```
git commit -am "Ajout du champ telephone à la classe Employe"
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git commit -am "Ajout du champ telephone à la classe Employe"
[master 1de078b] Ajout du champ telephone à la classe Employe
 1 file changed, 1 insertion(+)
```

- Malgré le fait qu'on n'a pas fait le **`git add`**, il a comme même indexé nos modifications et embarquer dans sa révision les modifications qu'on a fait au fichier **Employe.java**

  - Si on fait un **`git status`** : effectivement rien à commiter et le répertoire de travail est nettoyé, ce qui veut dire que tous les fichiers son à l'état validé.

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Attention à l'option **`-a`** :

- Ça ne fait que référencer et indexer les fichiers qui ont été modifiés.
- Tous les nouveaux fichiers qui ont été crée ne seront pas inclus avec l'option **`-a`** et il faudra faire une commande **`git add`** à la main.

Pour résumer le processus du commit

- ✓ Il est nécessaire de faire un **`git add`**, donc d'indexer à la main tous les nouveaux fichiers qu'on va inclure dans un commit ou dans une révision.
- ✓ D'indexer à la main tous les fichiers qui ont été modifiés.
- ✓ Et pour le reste, on valide, ça crée une révision, on passe à la suivante, …etc

3. Ignorer les fichiers

- Il peut être très utile dans un projet d'indiquer à Git d'ignorer systématiquement certains fichiers.

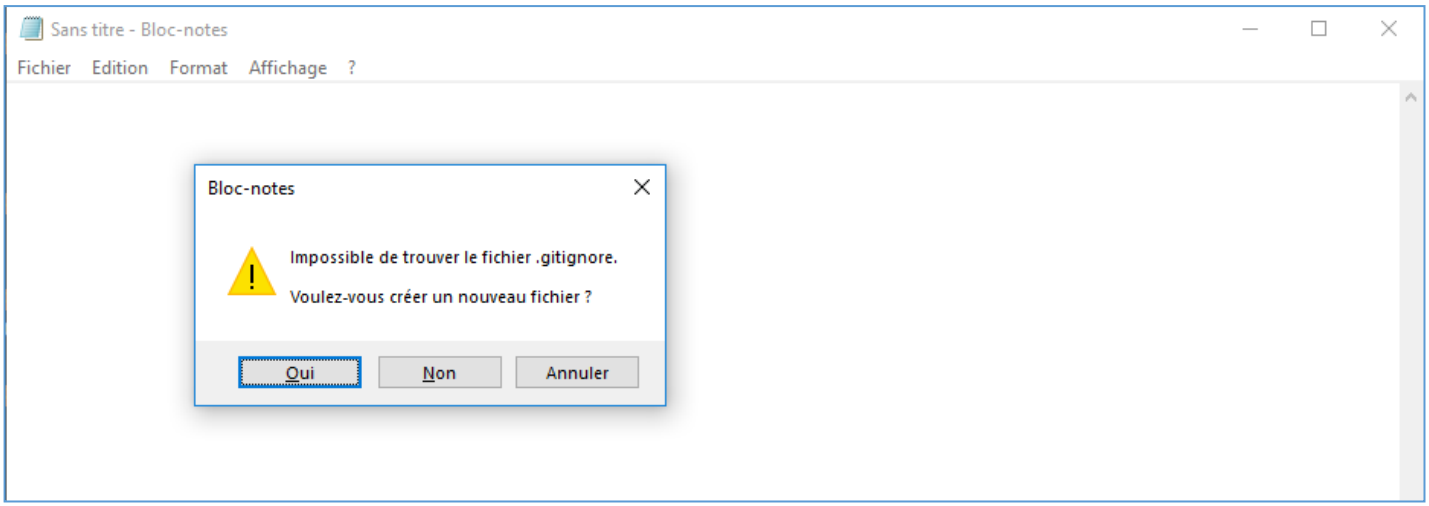
- Ça peut être des fichiers de log, des fichiers de configuration par ce que ça peut être dangereux de laisser une configuration à titre d'exemple des mots de passe à la BDD.

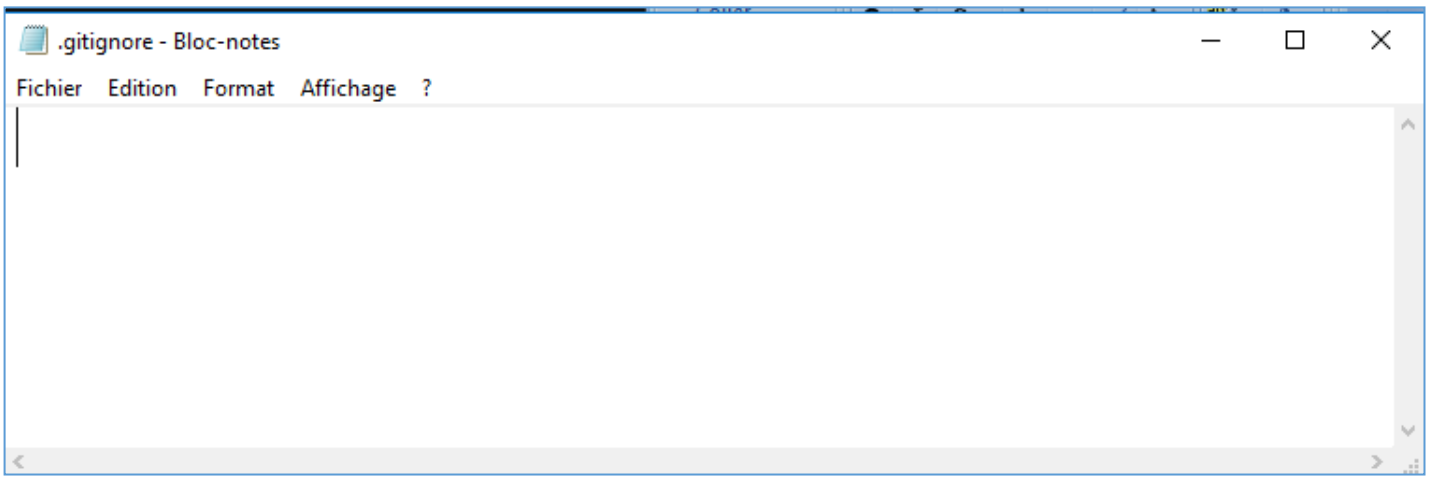
- Git nous met à disposition un système de gitignore qui va nous permettre d'ignorer certains fichiers de sorte qu'il ne soit jamais indexés et jamais enregistrés par Git.

- Pour mettre en place ce mécanisme de gitignore, il est simplement nécessaire d'écrire à la racine du dépôt un fichier qui va s'appeler sombrement **.gitignore**

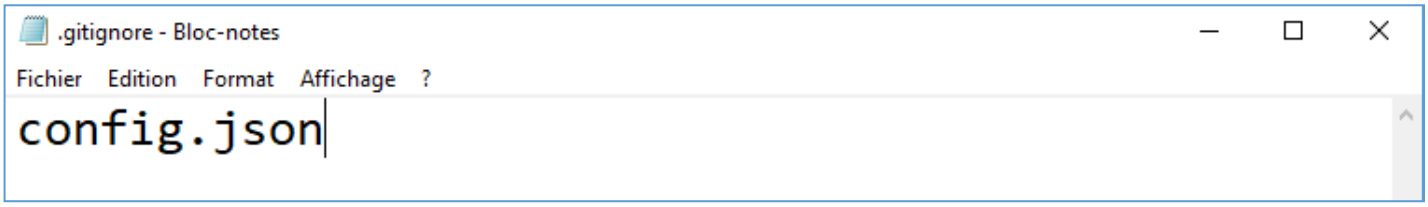
  - Dans la console on crée ce fichier avec notepad:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ notepad .gitignore
```

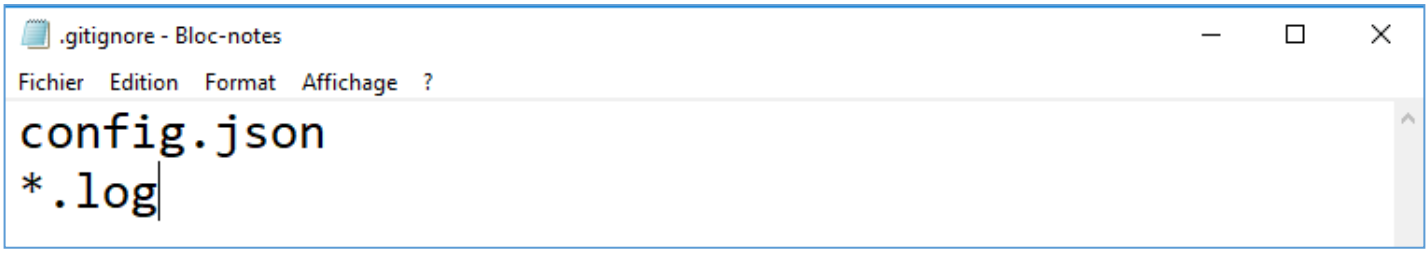




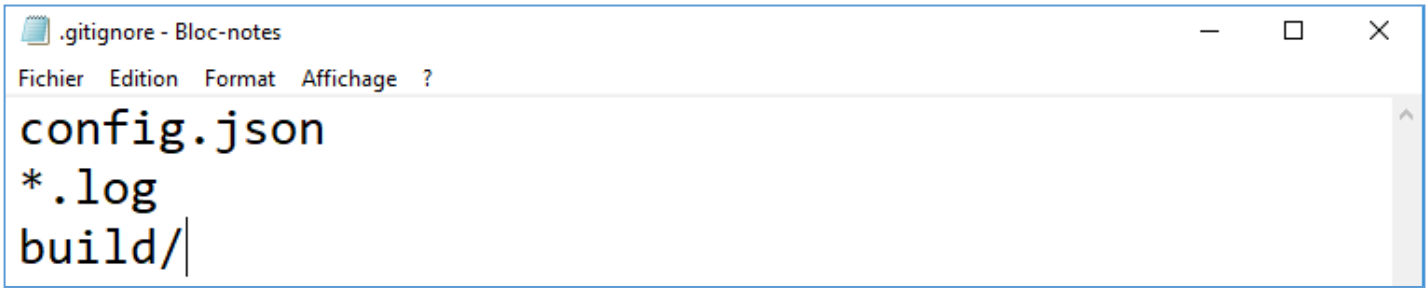
- Dans ce fichier, on va pouvoir déterminer quels sont les fichiers ignorés.
- Par exemple, si on veut ignorer le fichier **`config.json`**, on l'ajoute :



- Par exemple, on peut ignorer tous les fichiers se terminant par .log :

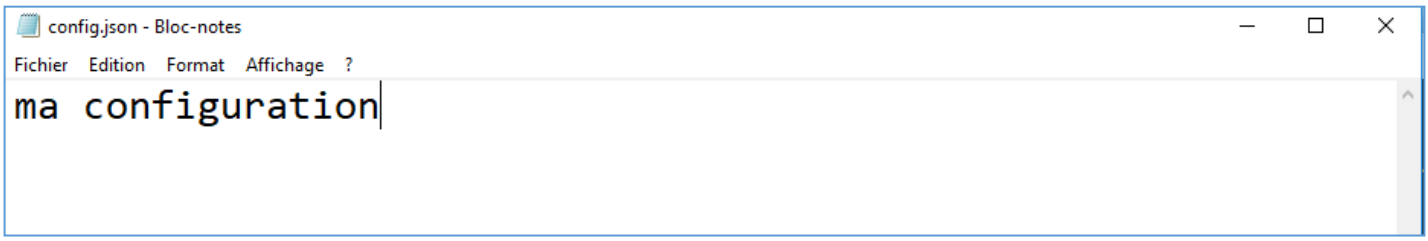


- Par exemple, on peut ignorer tous le répertoire **`build`** et ses fichiers :



- Dans la console, on crée le fichier **`config.json`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ notepad config.json
```



- On fait un **`git status`**:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

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

- Git nous informe qu'il y a un nouveau fichier qui n'est pas tracké, c'est le **`.gitignore`** lui-même qui considéré comme un nœud, comme un fichier standard aux yeux de git.

- Par contre notre fichier **`config.json`** est splendidement ignoré.

  - Si on fait un **`ls`** pour afficher le contenu, on a un fichier **`test.log`**

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ ls
config.json  desktop.ini  Employe.java  readme.md  test.log
```

  - Si on crée un nouveau fichier **`test2.log`**

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ touch test2.log
```

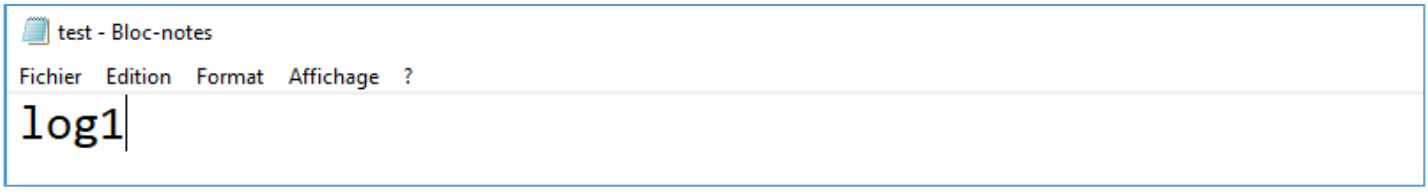
- Il va être ignoré par Git

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        .gitignore

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

- Attention néanmoins, le fichier **`test.log`** qui a été déjà référencé auparavant reste référencé par Git :

  Si on modifie le fichier :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ notepad test.log
```



- Etant donné qu'il a été référencé à la main, il va bien être traité comme un fichier <u>modifié</u> :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ 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.log

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

        .gitignore

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

Dans ce cas, pas de panique, on va voire la commande git **rm** avec l'option **--cached** qui veut supprimer du cache le fichier **test.log**

```
git rm --cached test.log
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git rm --cached test.log
```

Apres, on va que le fichier est présent sur notre système local :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ ls
config.json  desktop.ini  Employe.java  readme.md  test.log  test2.log
```

Par contre, à travers d'un **git status**, on nous informe que le fichier test.log est sortie du référentiel de version :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        deleted:    test.log

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

        .gitignore
```

- Pour terminer notre séquence, on va ajouter le fichier **.gitignore** au référentiel avec un **git add:**

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git add .gitignore
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   .gitignore
        deleted:    test.log
```

- Puis on commit le fichier **.gitignore** avec un message pour valider nos modifications:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git commit -m "Ajout du fichier .gitignore"
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git commit -m "Ajout du fichier .gitignore"
[master b8576b6] Ajout du fichier .gitignore
 2 files changed, 3 insertions(+)
 create mode 100644 .gitignore
 delete mode 100644 test.log
```

4. Récupérer une version d'un fichier

- Nous avons maintenant tous les outils qui nous permettent d'ajouter des nouveaux fichiers, de nouvelles modifications à notre référentiel et d'historiser ces modifications à travers des révisions, des versions lorsque l'on commit.

- Pour récupérer un commit, on peut récupérer une version d'un fichier précis, Git met à notre disposition la commande au nom trompeur **`git checkout`**.

- Cette commande permet de remplacer dans notre espace de travail un fichier par sa révision à un instant T.

- On a la commande **`git log`** qui permet de consulter tous les logs de notre dépôt local.

```
git log
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git log
commit b8576b674bb98250118511ca241ccbc2c398de8f (HEAD -> master)
Author: mezine younes <mezineyounes@gmail.com>
Date:   Wed Apr 25 19:27:55 2018 +0200

    Ajout du fichier .gitignore

commit 1de078b7f48c773cb261988e1474ad73bd92aa6d
Author: mezine younes <mezineyounes@gmail.com>
Date:   Wed Apr 25 17:45:03 2018 +0200

    Ajout du champ telephone à la classe Employe

commit 02be8fd8bf82f88f5227d374f48029d99990fc22
Author: mezine younes <mezineyounes@gmail.com>
Date:   Wed Apr 25 17:25:31 2018 +0200

    ajout des champs de la classe Employe

commit e6af4087c771d3b7b7e0b74eff2aa4ef7652dc06
Author: mezine younes <mezineyounes@gmail.com>
Date:   Wed Apr 25 16:55:54 2018 +0200

    fichier inconnu du bataillon.

commit b789086a11175e84dc91712d7e4581caad4bbde4
Author: mezine younes <mezineyounes@gmail.com>
Date:   Wed Apr 25 16:25:49 2018 +0200

    création du projet, prmière implémtation de la classe Employe.
```

- Ici on voit nos différents commits : le 1er commit de création (en bas).

- On va disposer de la commande **`git checkout`** qui va nous permettre soit de préciser un fichier soit de préciser toute une révision pour remettre notre projet dans ce contexte la :

  - Par exemple, si on veut repartir dans notre contexte initial :

```
commit b789086a11175e84dc91712d7e4581caad4bbde4
Author: mezine younes <mezineyounes@gmail.com>
Date:   Wed Apr 25 16:25:49 2018 +0200

    création du projet, prmière implémtation de la classe Employe.
```

- On fait un **`checkout`** de cette révision :

```
git checkout b789086a11175e84dc91712d7e4581caad4bbde4
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git checkout b789086a11175e84dc91712d7e4581caad4bbde4
Note: checking out 'b789086a11175e84dc91712d7e4581caad4bbde4'.

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 b789086 création du projet, prmière implémtation de la classe Employe.
```

- Git nous dit différentes choses :

  - Le HEAD, la tête de la branche est désormais la b789086 qui correspond bien à notre premier commit.

  - On est dans un état détaché '**`detached HEAD`**'. étant donné qu'on est plus sur la version connue mais une version intermédiaire.

    Si on modifie les fichiers de la version intermédiaire, on les commit, ils ne vont pas venir s'insérer entre les 2 révisions mais ils vont donner lieux à un autre référentiel à côté qui n'a rien à voire.

    Effectivement, on peut consulter une version précédente mais on ne peut pas la modifier en tant que telle puisqu'on ne peut pas revenir dans le passé, effectuer des modifications et repartir dans le futur et bénéficier de ces modifications.

    La, c'est un avertissement Git qui nous dit : en consultation, il y aura pas de problèmes mais si on fait des modifications, il va y avoir des problèmes.

- Ici si on fait un **`ls`** pour afficher le contenu :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b789086...))
$ ls
config.json  Employe.java  readme.md  test.log  test2.log
```

- On voit que config.json et les fichiers .log qui sont normalement exclus de Git grâce au gitignore sont bien présents.

- Effectivement, Git ne prend la responsabilité de modifier que les fichiers qu'il gère.

- Ces fichiers étant exclus du référentiel, lorsqu'on fait un checkout, Git ne touche pas à ces fichiers.ils était pas présents lors du commit initial et ils ne sont pas gérer par lui donc il n y touche pas.

- Ici si on voit que la mention **`master`** est remplacée par le numéro de commit **`((b789086))`**.

  Si on affiche notre fichier source, il va être dans son état initial :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b789086...))
$ cat Employe.java
public class Employe{



}
```

- Ici si on veut partir sur le dernier commit :

```
commit b8576b674bb98250118511ca241ccbc2c398de8f (HEAD -> master)
Author: mezine younes <mezineyounes@gmail.com>
Date:   Wed Apr 25 19:27:55 2018 +0200

    Ajout du fichier .gitignore
```

```
git checkout b8576b674bb98250118511ca241ccbc2c398de8f
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b789086...))
$ git checkout  b8576b674bb98250118511ca241ccbc2c398de8f
Previous HEAD position was b789086 création du projet, prmière implémtation de la classe Employe.
HEAD is now at b8576b6 Ajout du fichier .gitignore
```

- Maintenant, on est sur la dernière version de notre projet et on peut repartir sur un mode de développement.

- Attention : si on fait un **`ls`**

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ ls
config.json  desktop.ini  Employe.java  readme.md  test2.log
```

- Si on modifie par exemple **`config.json`**, on aurait toujours bénéficié de ces modifications étant donner que Git n y touche pas puisqu'ils ne font pas partie de son référentiel.

- Ici le fichier **`test.log`** à disparue puisque dans cette révision, on a indiqué qu'il devrait être supprimé du référentiel.

- Alors par rapport à l'historique, on peut tomber sur plusieurs problèmes au quotidien :

  - On crée un nouveau fichier **`Client.java`**

Client.java

```
1  public class Client{
2
3
4  }
```

    - Il fait bien partie de notre répertoire :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ ls
Client.java  config.json  desktop.ini  Employe.java  readme.md  test2.log
```

    - Le **`git status`** nous indique que le fichier n'est pas référencé, qui n'est pas connu de lui :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git status
HEAD detached at b8576b6
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Client.java

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

    - Donc on va devoir faire un **`git add`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git add Client.java
```

- Le **`git add`** nous indique qu'on a un nouveau fichier qui va faire partie du commit:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git status
HEAD detached at b8576b6
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        new file:   Client.java
```

- Mais on se rend compte de notre erreur, on s'est un peut emballer, on n'a pas terminé de travailler sur ce fichier la et on ne veut pas qu'il fasse partie du prochain commit mais néanmoins il est déjà indexé.

- Comment est ce qu'on peut procéder ?

  - On a une commande **`git reset`** qui nous permettre de ressortir les fichiers indexés et les remettre à l'état initial :

```
git reset
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git reset

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git status
HEAD detached at b8576b6
Untracked files:
  (use "git add <file>..." to include in what will be committed)

        Client.java

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

- Ici on retourne dans la configuration avant le **`git add`**

✘ Donc si on a ajouté un fichier qu'il ne fallait pas et si on a indexés des modifications qui n'était terminées, un **`git reset`** peut nous sauver.

- On a une version un peut plus radical du git reset qui est le :

```
git reset --hard
```

- qui va non seulement repasser les fichiers indexés à l'état précèdent mais qui en plus va écraser les modifications qu'on a faites depuis le dernier commit .

- On me modifie Employe.java :

```
Employe.java
1  public class Employe{
2
3      private String nom;
4      private String prenom;
5      private String mail;
6      private String telephone;
7
8      private String adresse;
9
10 }
```

- A l'aide d'un **`git status`**, on va bien que notre fichier à subi des changements :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git status
HEAD detached at b8576b6
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:   Employe.java

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

        Client.java

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

- Si on fait un **`reset hard`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git reset --hard
HEAD is now at b8576b6 Ajout du fichier .gitignore
```

- Ça a fait revenir notre **`Employe.java`** tel qu'il était au dernier commit.

✘ C'est quelque chose qui peut être très utile notamment lorsque dans une modification on se rend compte qu'elle n'apporte satisfaction, non seulement on va commiter ces fichiers mais on va revenir à l'état initial de ce fichier comme il l'était au dernier commit.

- On va voire quelques commandes qui vont nous permettre de nettoyer notre espace de travail.

  - Si on fait un **`ls`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ ls
Client.java  config.json  desktop.ini  Employe.java  readme.md  test2.log
```

- on constate qu'on à des fichiers qui sont dans le gitignore (config.json et test2.log) que Git va ignorer et on a le fichier Client.java qu'on n'a pas souhaité commiter.

- On voudrait pouvoir les nettoyer de notre espace de travail puisqu'ils ne sont pas représentatifs de ce qui se passe sur mon référentiel.

  - Pour nettoyer notre espace de travail, on a la commande:

```
git clean
```

- Qui nous permet au choix de nettoyer les fichiers qui ne font pas partie de référentiel ou tous les fichiers du gitignore

    - La commande **`git clean -n`** va nous expliquer ce que la commande va faire si on l'exécutait :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git clean -n
Would remove Client.java
```

- Elle supprimera *Client.java* qui ne fait pas partie du gitignore mais qui n'est pas indexé

- Pour supprimer tous les fichiers qui sont pas référencé par Git, on utilise la commande **`git clean -f`** :

```
git clean -f
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git clean -f
Removing Client.java
```

- On est plus à l'image de ce qui se passe dans notre dépôt.

- Pour supprimer tous les dossiers avec leurs fichiers qui sont pas référencé par Git, on utilise la commande **`git clean -df`** :

```
git clean -df
```

- Le problème ici est que **`git clean -n`** ne prend les fichiers du gitignore.

- L'option **`-x`** supprime les fichiers du gitignore de notre espace de travail

```
git clean -xn
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git clean -xn
Would remove config.json
Would remove test2.log
```

- Si on veut nettoyer complètement notre espace de travail et qu'il ne soit pas polluer par des fichiers qui sont ignorés par Git, on combine **`-x`** pour gitignore, **`-d`** pour les répertoires et **`-f`** pour les fichiers :

```
git clean -xdf
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git clean -xdf
Removing config.json
Removing test2.log
```

- Ici il nous supprime tous les fichiers qui sont dans le gitignore et tous les fichiers qui ne sont pas indexés par Git.

- Et Si on fait un **`ls`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ ls
desktop.ini  Employe.java  readme.md
```

- On a désormais un espace de travail qui est complètement nettoyé puisqu'on a plus que les 2 fichiers indexés par le référentiel, qui sont surveillés par lui et qui font partie de notre projet.

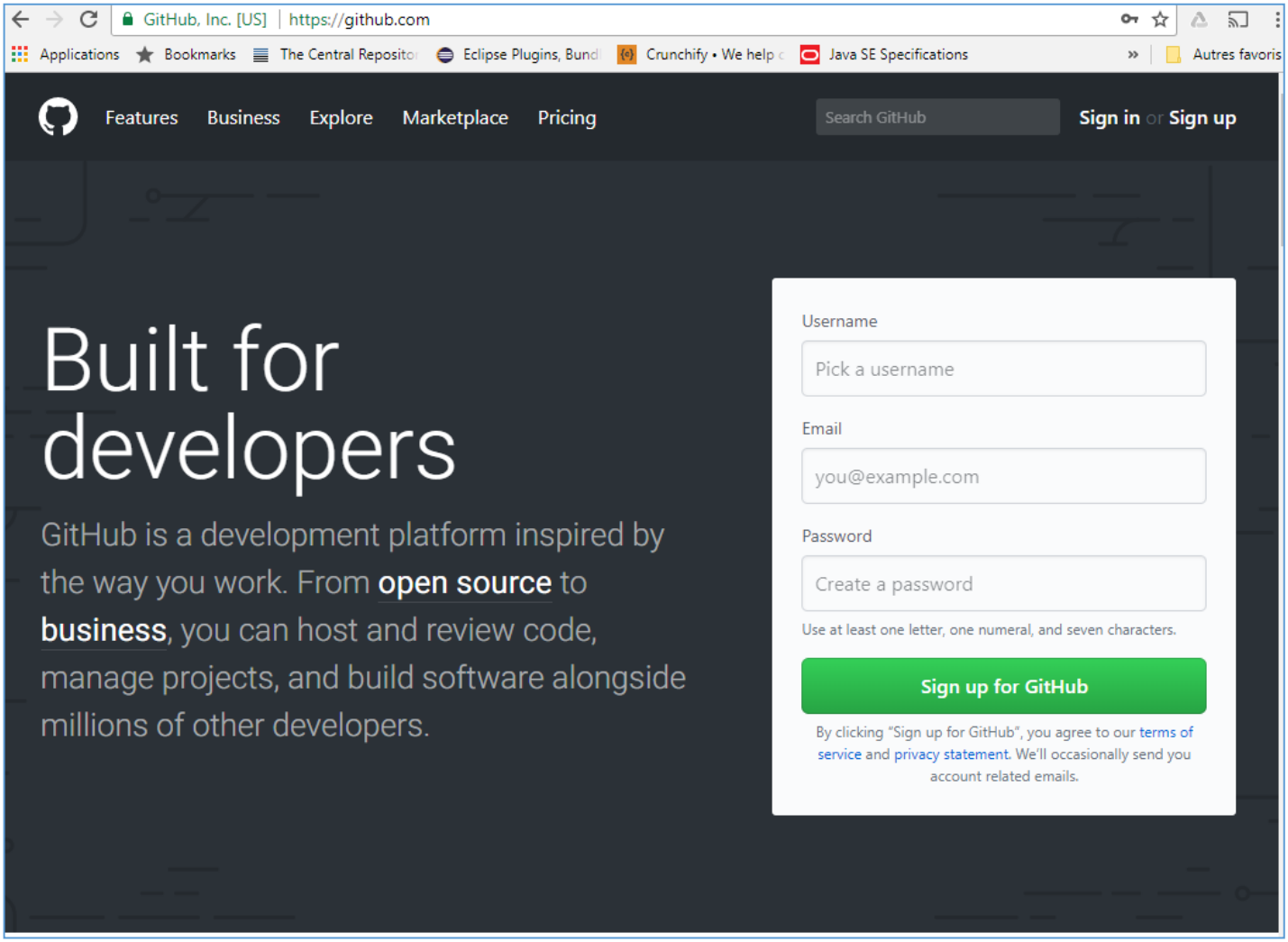
IV. Interaction avec un dépôt distant

1. Présentation de GitHub

- GitHub est la plateforme en ligne la plus utilisée, la plus plébiscitée par la communauté libre open-source à travers le monde.

- C'est une plateforme gratuite qui permet à n'importe quel développeur, amateur, professionnel, expert, d'ouvrir un dépôt en son nom, d y déclarer des projets, de travailler sur ces projets en collaboration avec d'autres développeurs et mettre son code à disposition de tous.

- On accède à GitHub sur le site : https://github.com/

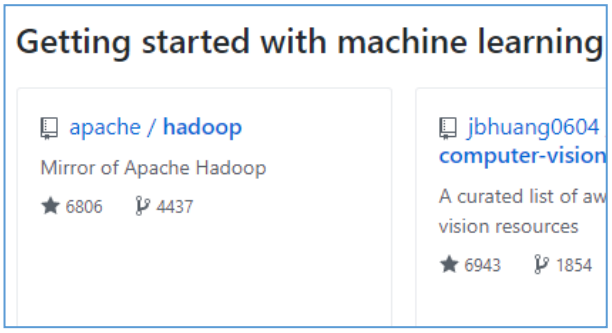


- C'est une interface très épurée ou on va procéder à une inscription gratuite qui vous permet :
    - De disposer d'un pseudo

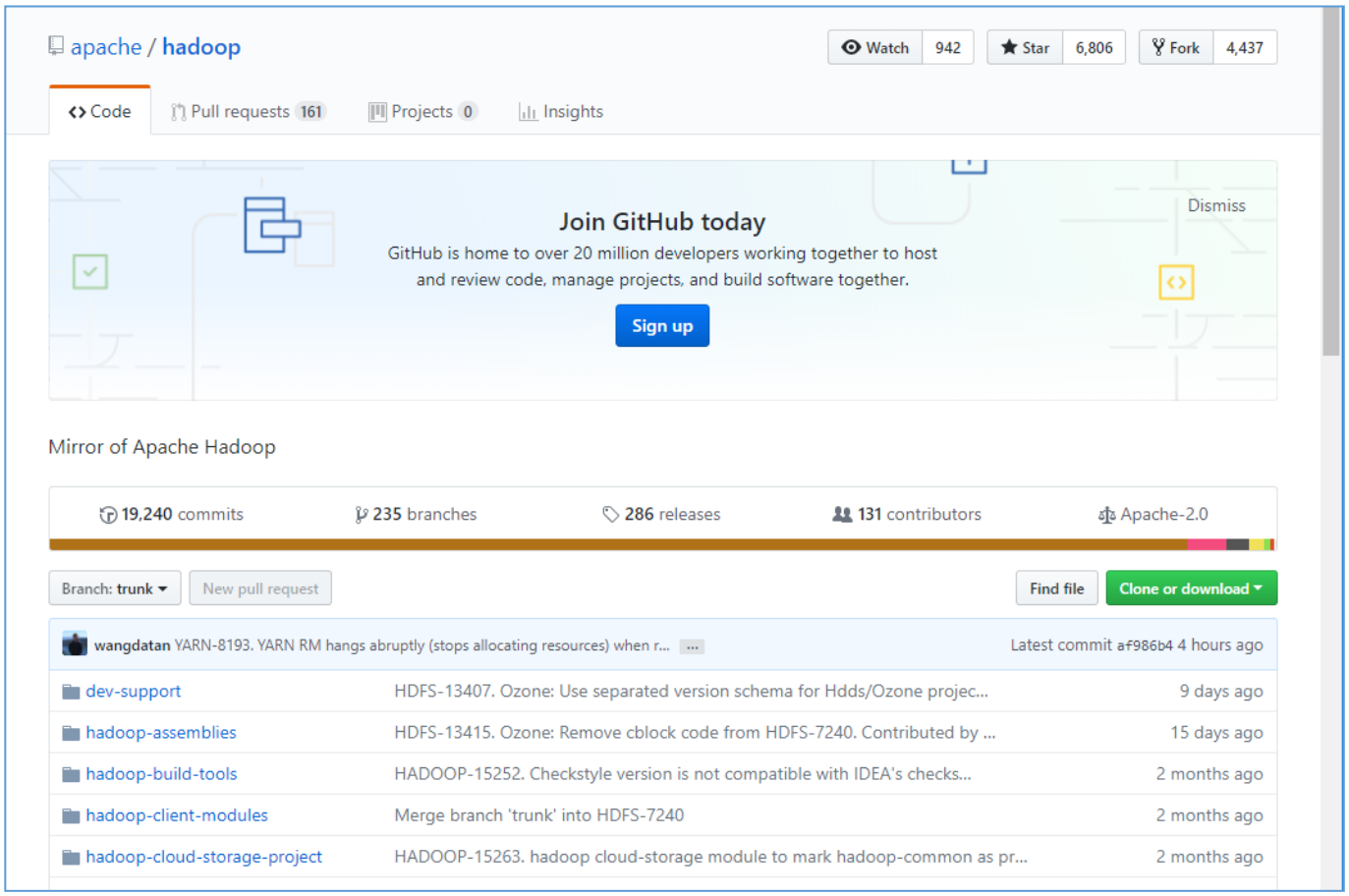
- de créer nos propres dépôts
- de souscrire à des dépôts existants pour collaborer et apporter de nouvelles fonctionnalités
- effectuer des corrections sur des projets de d'autres participants
- simplement pour faire de la veille, ça veut dire de suivre un dépôt particulier pour une technologie qui nous plait ou pour un plugin que nous utilisons dans un projet et qui va nous tenir au courant des dernières évolutions.

- Prenons un exemple d'un dépôt quelque conque :

  - Sur l'onglet **`Explore`** -> on choisit : **`apache/hadoop`**



  - On accède au dépôt **`hadoop`** d'apache :



- Un dépôt est toujours constitué de la même manière :

  - On a l'onglet **<>Code** qui contient le code source du projet.

    On peut le télécharger ou le rapatrier dans un dépôt local avec **Clone or download.**

    On peut voire tous les commits qui ont été fait(**commits**).

    On peut voire les branches.

    Les **releases** ou les sorties qui ont été faite par l'équipe.

    Le nom des **contributeurs** soit pour une virgule soit pour tout le projet à la vie du projet

  - Le système **Issues** permet de dialoguer dans la communauté pour par exemple remonter des problèmes ou suggérer de nouvelles fonctionnalités.

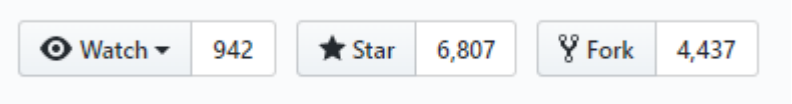
  - **Pull Requests** qu'on verra plu tard

  - **Projects** : pour gérer l'organisation dans l'équipe.

  - **Insights** : un système de graphe pour voire l'activité du typo au fur et à mesure du temps.

- C'est un outil très complet qui va nous permettre d'initialiser des projets, de participer ou de suivre d'autres projets

  - On a ici 3 notions :



Watch : pour surveiller ce projet

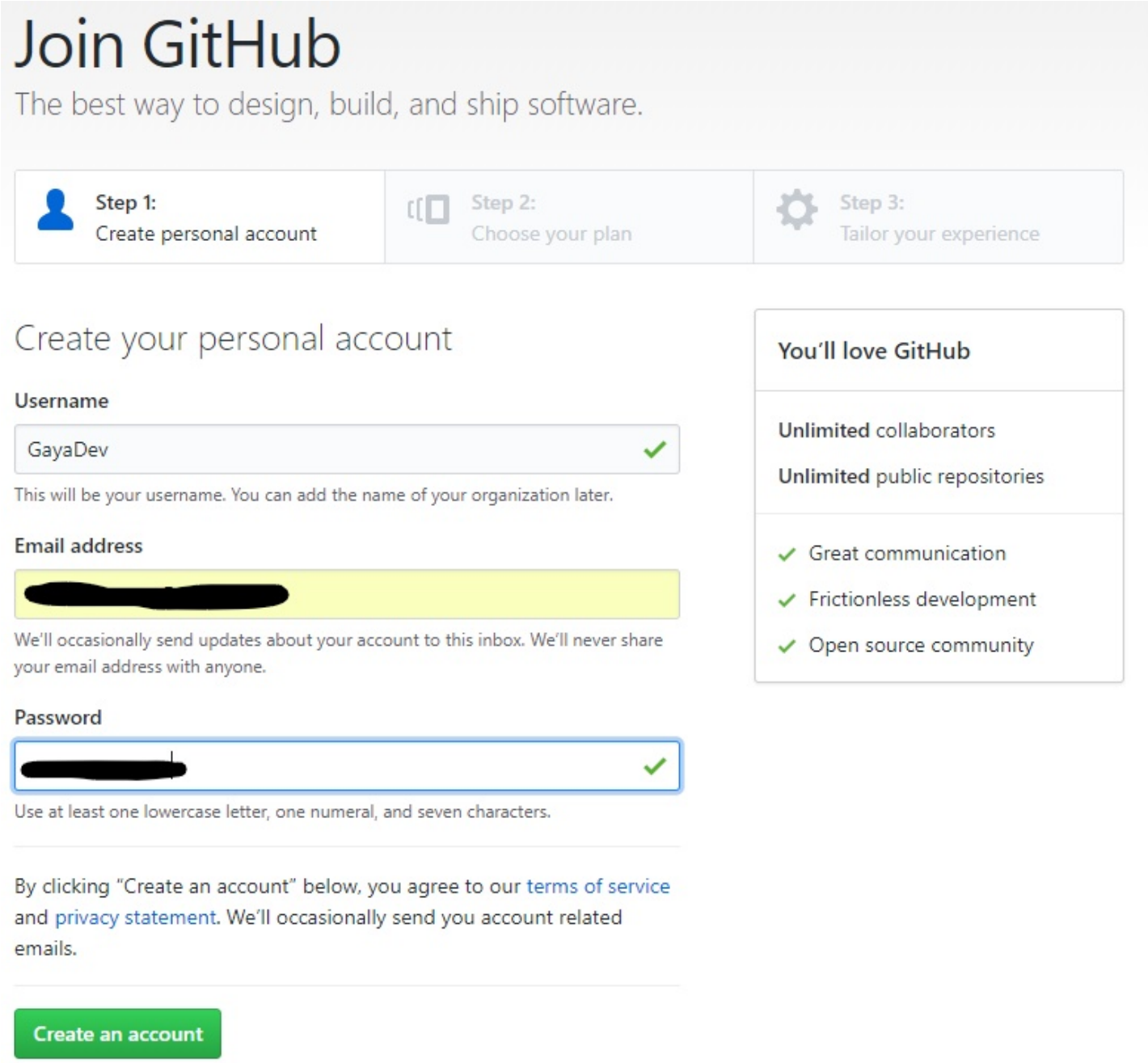
Star : permet d'indiquer un intérêt ou de valider la pertinence d'un projet

Fork : permet de cloner le projet dans notre dépôt à nous pour pouvoir l'enrichir, apporter des modifications au projet.

- ✘ Ce qui peut être dérangeant avec GitHub est que notre code source est accessible à tous.
- ✘ tous le monde peut le commenter et suggérer des choses et tous le monde peut copier notre code pour l'enrichir de son côté , et éventuellement pour partir dans une direction différente puis le pousser vers nous pour améliorer notre propre plugin.

- On va créer un compte sur GitHub avec le bouton **`Sign up`**:

  Password : IntiFormation2019



Welcome to GitHub

You've taken your first step into a larger world, **@GayaDev**.

Completed — Set up a personal account
Step 2: Choose your plan
Step 3: Tailor your experience

Choose your personal plan

- Unlimited public repositories for free.
- **Unlimited private repositories** for $7/month. (view in EUR)

Don't worry, you can cancel or upgrade at any time.

Both plans include:

- Collaborative code review
- Issue tracking
- Open source community
- Unlimited public repositories
- Join any organization

Help me set up an organization next
Organizations are separate from personal accounts and are best suited for businesses who need to manage permissions for many employees.
Learn more about organizations

Send me updates on GitHub news, offers, and events
Unsubscribe anytime in your email preferences. Learn more

Continue

Welcome to GitHub

You'll find endless opportunities to learn, code, and create, **@GayaDev**.

Completed — Set up a personal account
Step 2: Choose your plan
Step 3: Tailor your experience

How would you describe your level of programming experience?

- Very experienced
- Somewhat experienced
- Totally new to programming

What do you plan to use GitHub for? (check all that apply)

- School projects
- Research
- Design
- Development
- Project Management
- Other (please specify)

Which is closest to how you would describe yourself?

- I'm a professional
- I'm a student
- I'm a hobbyist
- Other (please specify)

What are you interested in?

e.g. tutorials, android, ruby, web-development, machine-learning, open-source

Submit skip this step

Search GitHub — Pull requests — Issues — Marketplace — Explore

Learn Git and GitHub without any code!

Using the Hello World guide, you'll create a repository, start a branch, write comments, and open a pull request.

Read the guide — Start a project

© 2018 GitHub, Inc. Terms Privacy Security Status Help — Contact GitHub API Training Shop Blog About

Search GitHub — Pull requests — Issues — Marketplace — Explore

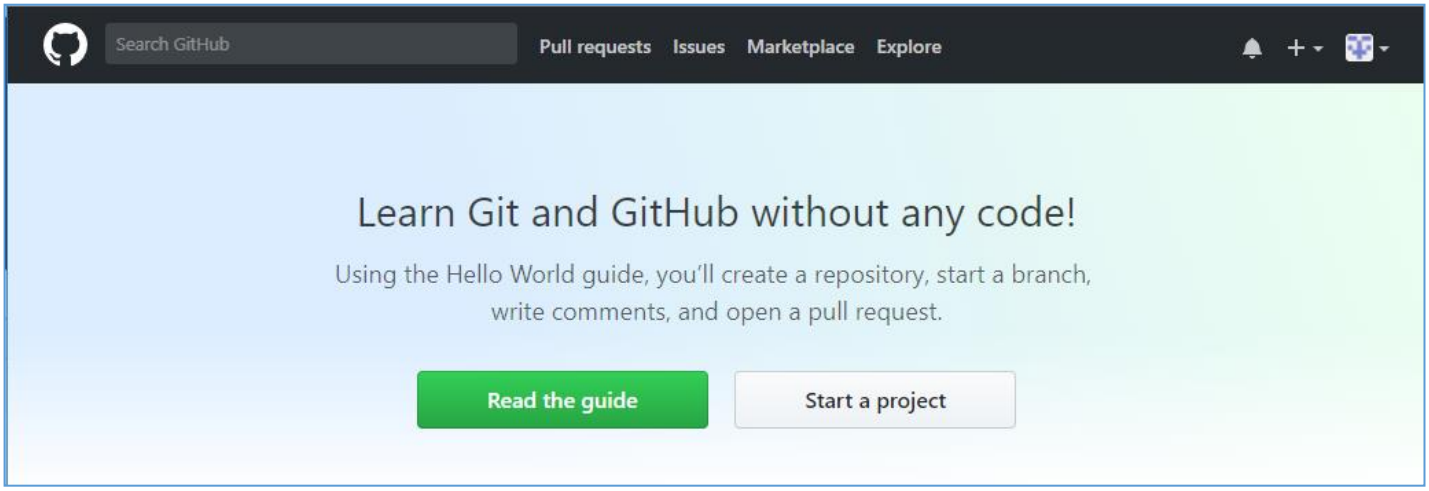
Please verify your email address

Before you can contribute on GitHub, we need you to verify your email address.
An email containing verification instructions was sent to [redacted].

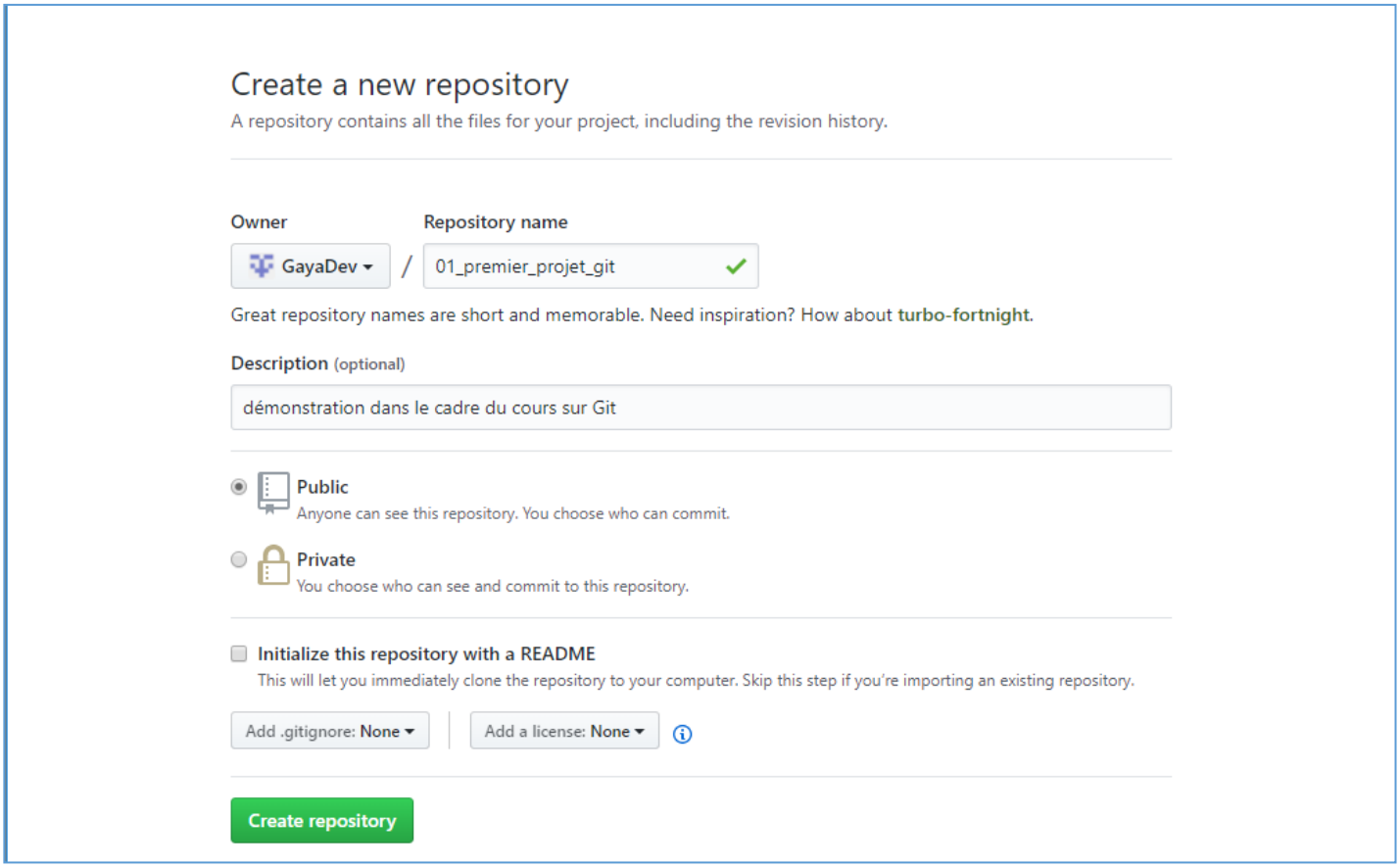
Didn't get the email? Resend verification email or change your email settings.

2. Créer un projet sur GitHub

- Une fois connecté à notre profil, on arrive à cette page d'accueil :



- Pour créer un projet ou un dépôt sur GitHub, on clique sur **« Start a project »**



- On nous demande qui le possesseur du dépôt avec **Owner.** On peut aiguiller les dépôts en fonction de leurs pertinence, projet du weekend, projet professionnel, …etc.

- On doit indiquer un nom dans **`Repository name`** qui doit être unique dans notre organisation.

- On ajouter une description dans **`Description`**

- On a un choix de pouvoir afficher notre dépôt en publique avec la case **`Public`**. C'est de l'open-source, tous le monde le verra, lire notre code et peut y contribuer. C'est gratuit.

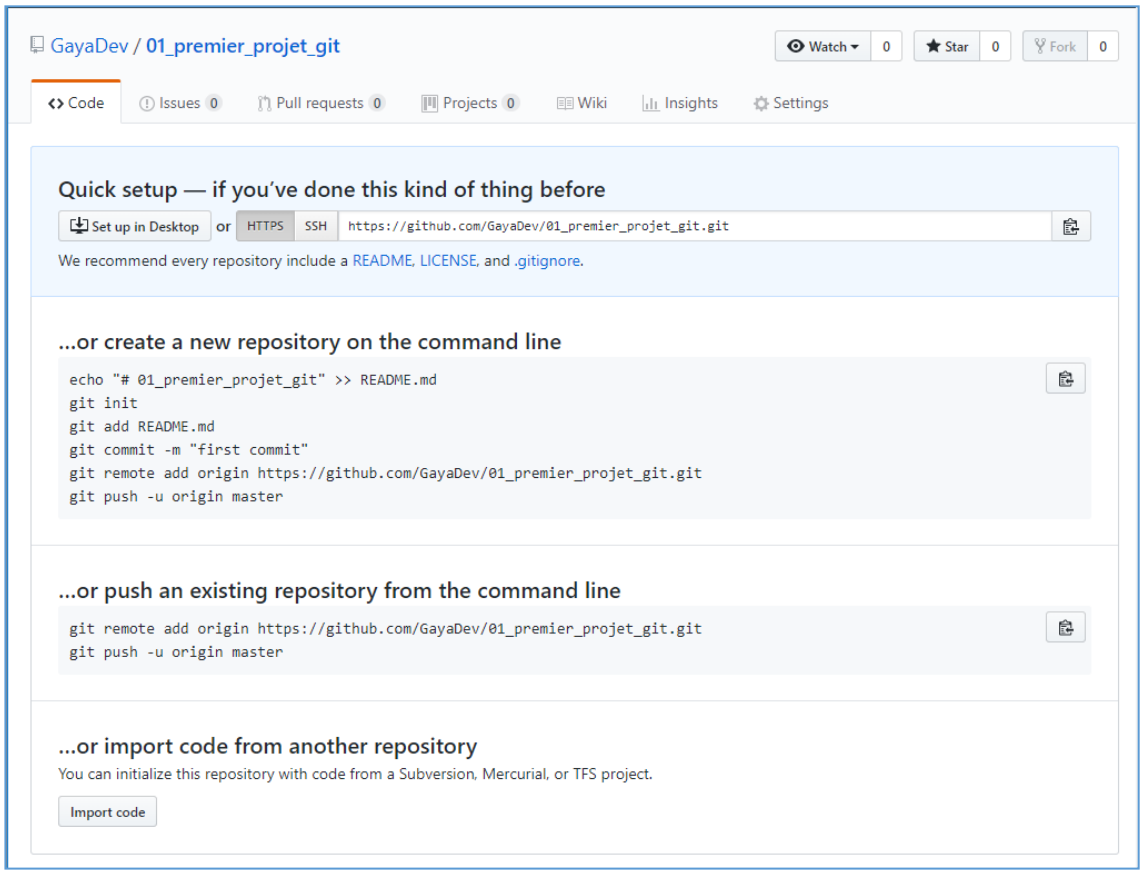
  On a la même chose en privée avec la case **`Private`**. Si on ne veut pas que notre code soit consulté par tous le monde, néanmoins il faudra souscrire un abonnement GitHub

- On peut initialiser notre dépôt avec un **`readme`**

- On peut ajouter un **`gitignore`** en sélectionnant le langage.

- On veut spécifier les instructions sur la licence open-source qu'on utilise. Il y a un vaste système de licence open-source qu'on peut mettre en place sur notre projet.

  - Une fois qu'on a tout réglé, on clique sur **`Create Repository`** pour créer le dépôt :



- Notre dépôt est bien crée, ça veut dire que sur GitHub.

- La page nous expose plusieurs instructions qui vont permettre de pousser une version depuis notre dépôt local vers GitHub et qui va permettre à d'autres développeurs de le lire et de l'enrichir.

3. Ajouter un dépôt distant

- Maintenant qu'on a mis en place un dépôt distant sur GitHub, il est temps de le synchroniser avec le dépôt local que nous avons crée et sur lequel on a déjà fait plusieurs commit.

- On reprend notre dépôt local et la console avec un git status :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git status
HEAD detached at b8576b6
nothing to commit, working tree clean
```

  - Il nous informe qu'on est sur une version détachée à part de notre workflow.

  - Donc, on va devoir nous synchroniser avec notre depot local pour pouvoir envoyer des choses sur le depot distant

    - On va faire un **`checkout`** sur la branche principale qui est master :

```
git checkout master
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git ((b8576b6...))
$ git checkout master
Switched to branch 'master'

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$
```

  - Et un git status nous indique qu'on est bien sur la branche principale :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

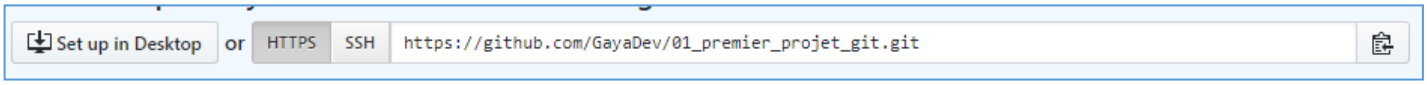
- Ensuite, on va ajouter un depot distant avec la commande:

```
git remote
```

- La commande toute seule nous renvoie les dépôts distants auxquels on est abonnés. Pour le moment, on n'a aucun abonnement.

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git remote

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$
```

- Pour ajouter un dépôt distant, on va utiliser
  - L'option **add**
  - suivis du nom du dépôt référentiel qui est **origin** (l'origine des sources). On pousse les sources vers lui, Plu tard, ça sera le référentiel pour tous les développeurs
  - suivis de l'URL du dépôt distant qu'on peut récupérer sur GitHub




```
git remote add origin https://github.com/GayaDev/01_premier_projet_git.git
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git remote add origin https://github.com/GayaDev/01_premier_projet_git.git
```

- Maintenant si on tape **`git remote`**, on verra le dépôt distant origin apparaitre dans la liste :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git remote
origin
```

- On peut avoir plus d'infos avec **`git remote -v`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git remote -v
origin  https://github.com/GayaDev/01_premier_projet_git.git (fetch)
origin  https://github.com/GayaDev/01_premier_projet_git.git (push)
```

- Il nous explique 2 notions qu'on n'a pas encore vu, le fetch et le push, qui vont nous permettre de récupérer des choses du dépôt distant (**fetch**) et de pousser des choses vers le dépôt (**push**)

  - Si on veut supprimer un depot, on peut utiliser la commande :

```
git remote rm origin
```

  - Si on veut renommer un depôt, on peut utiliser la commande :

```
git remote rename origin history
```

- origin : ancien nom | history : nouveau nom

  - Même si se n'est pas recommandé par ce que ça amène de la confusion, on peut avoir plusieurs dépôts distants à al fois.

4. Pousser des versions vers le dépôt distant

- Maintenant on va voire comment pousser les modifications faites dans le dépôt local vers le dépôt distant.

- On utilise le mot pousser par ce que La commande associée est :

```
git push
```

- Ça va nous permettre de synchroniser les révisions local avec le distant.

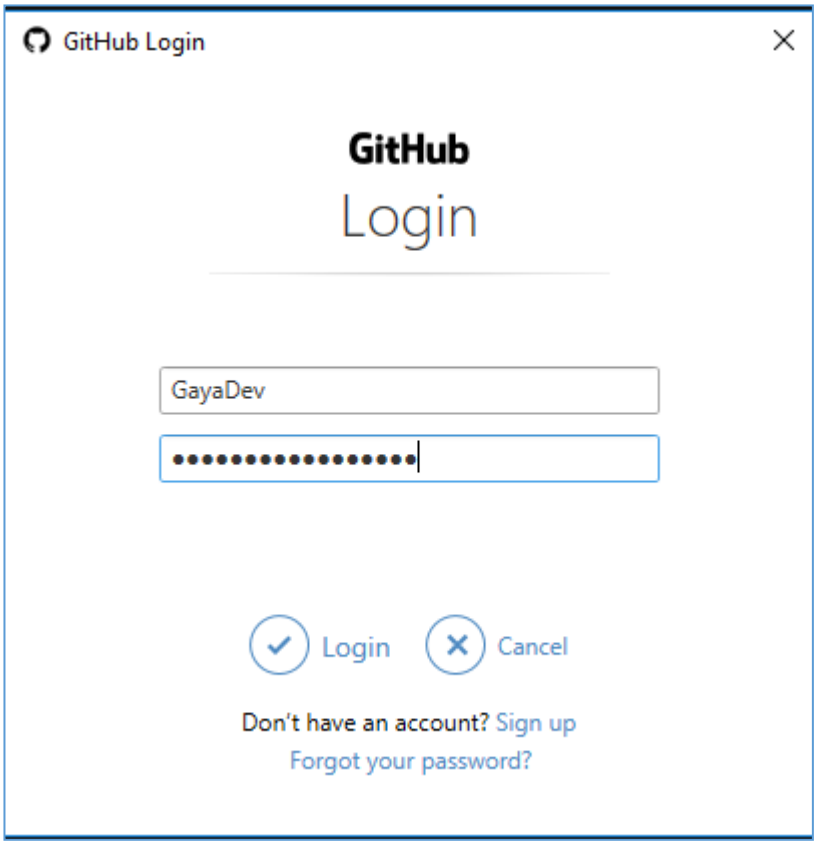
  - Cette commande prend 2 paramètres :

    - Le 1er est le nom du distant vers lequel on va pousser, ici **`origin`**
    - Le 2eme est la branche qu'on veut pousser, ici on est sur **`master`**

```
git push origin master
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git push origin master
```

On saisi notre les identifiants GitHub :

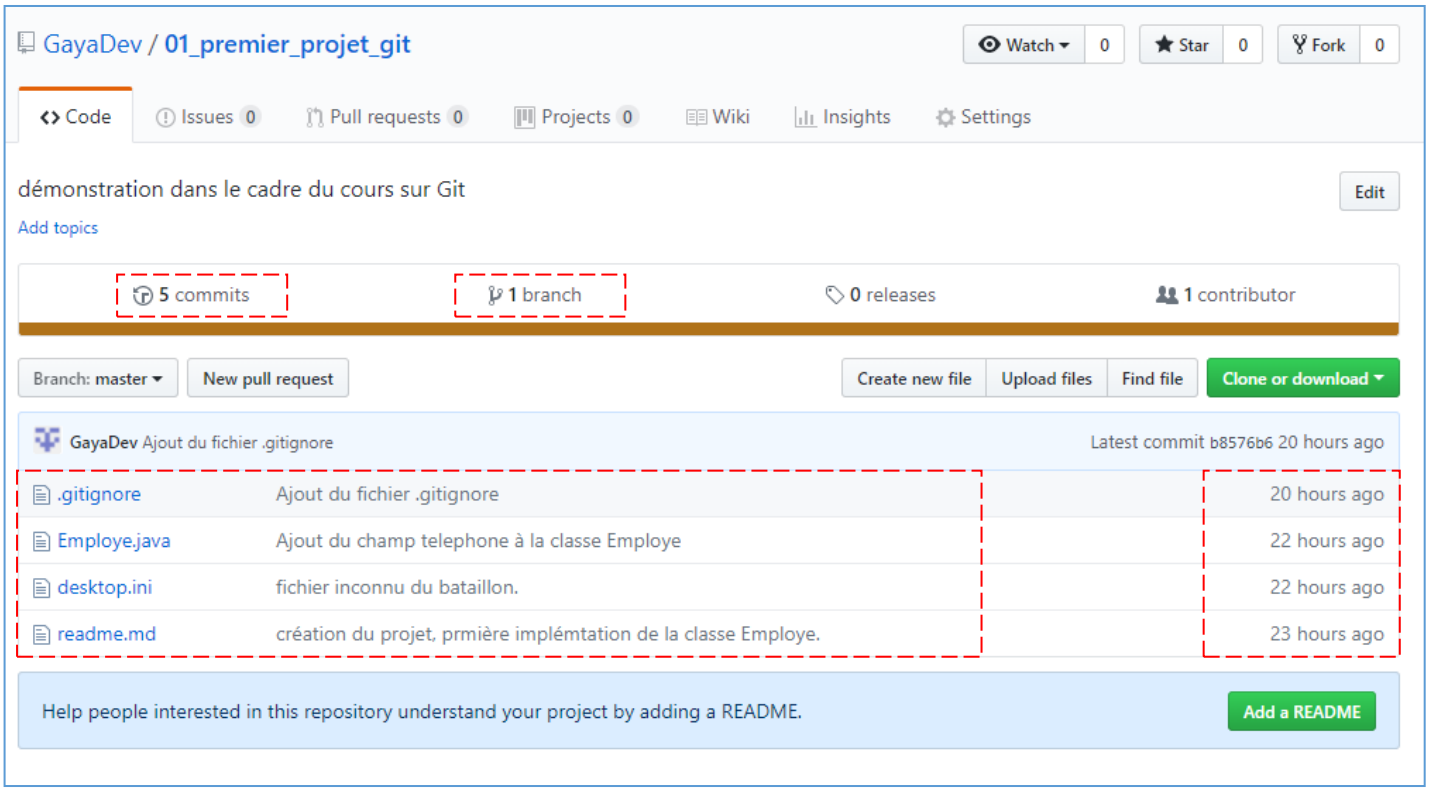


- Git commence la synchronisation :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git push origin master
Counting objects: 16, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (12/12), done.
Writing objects: 100% (16/16), 1.41 KiB | 720.00 KiB/s, done.
Total 16 (delta 5), reused 0 (delta 0)
remote: Resolving deltas: 100% (5/5), done.
To https://github.com/GayaDev/01_premier_projet_git.git
 * [new branch]      master -> master
```

- Il compte le nombre d'objets qui vont être synchronisé
- Il va les compresser pour des économies de bande passante
- Il les envois
- Au final, il nous dit, que sur le dépôt https://github.com/GayaDev/01_premier_projet_git.git , il a poussé une nouvelle branche appelé **`master`** (il a gardé le même nom dans le local)

- Maintenant si on retourne sur GitHub et on rafraichi la page, on retrouve le projet :

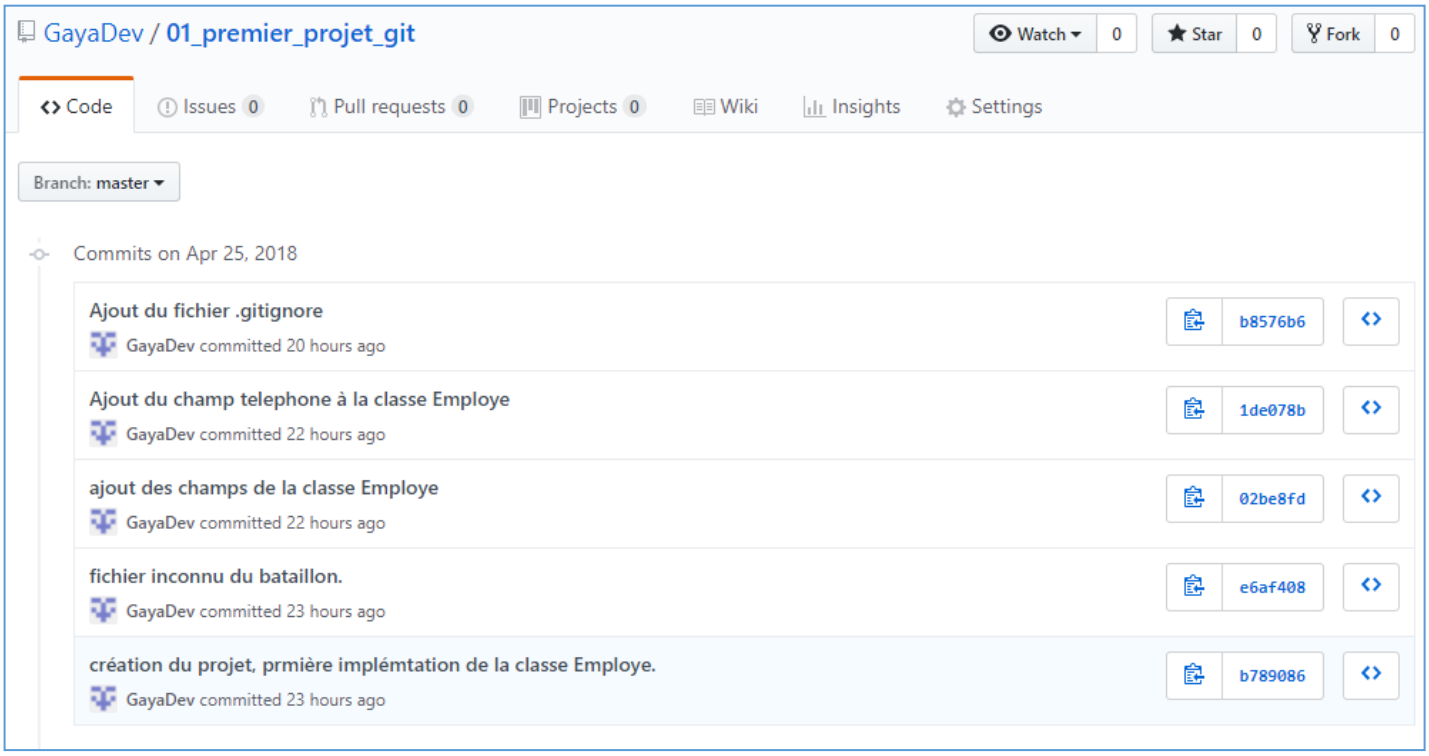


- On retrouve les 5 commits (**`5 commits`**)
- On retrouve la branche (**`1 branch`**)
- On retrouve notre code source
- Ce qui est intéressant de constater avec Git, c'est que tout ce qu'on fait en local va être synchronisé à un instant T et l'eurodatage, la temporisation de notre projet va être conservée.

  Le classe **`Employe.java`**, on la modifié il y a 22 heures. Lorsqu'on synchronise, pour le dépôt distant, c'est simplement le fait de prendre les informations, de les croire sur parole et de les intégrer.

  On peut très bien travailler dans le train par exemple pendant 3 heures, on fait des commits successifs et en arrivant à la gare de destination, on accroche un réseau wifi, on fait notre push, il va synchroniser non seulement l'historique des commits toute en gardant l'eurodatage et les commentaires.

- On peut voire les commentaires en cliquant sur le lien **5 commits** :



- On a bien les commentaires avec leurs numéros de hash et l'eurodatage.

- On garde l'historique qu'on a monté en local et c'est vraiment une synchronisation avec le serveur distant.

- Du côté Git avec un git status :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

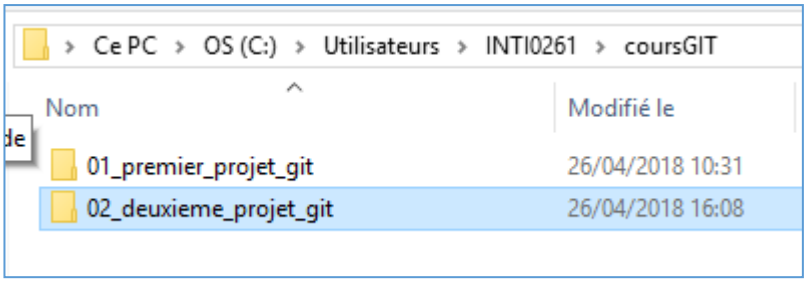
- Rien n'a changé, on est toujours sur la branche **master** et il n y a rien à commiter.

5. Récupérer des versions depuis le dépôt distant

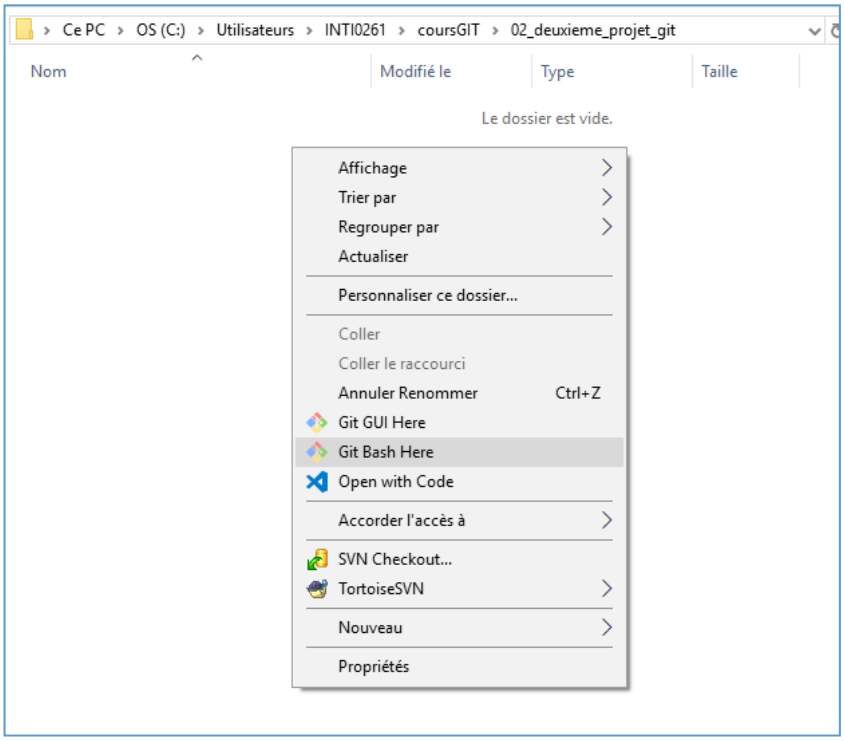
- Nous allons nous mettre dans la peau d'un autre développeur qui souhaitera contribuer à notre projet.

- Ce développeur n'est pas à l'initiative du projet. Il arrive sur GitHub, il voit le projet, il s y intéresse et il voudra synchroniser tout ce qui se passe sur GitHub avec son dépôt local et créer un par la même occasion.

- On va créer un nouveau dossier vierge **`02_deuxieme_projet_git`**



  - On va ouvrir une 2eme console **`git Bash`** pour ce projet :

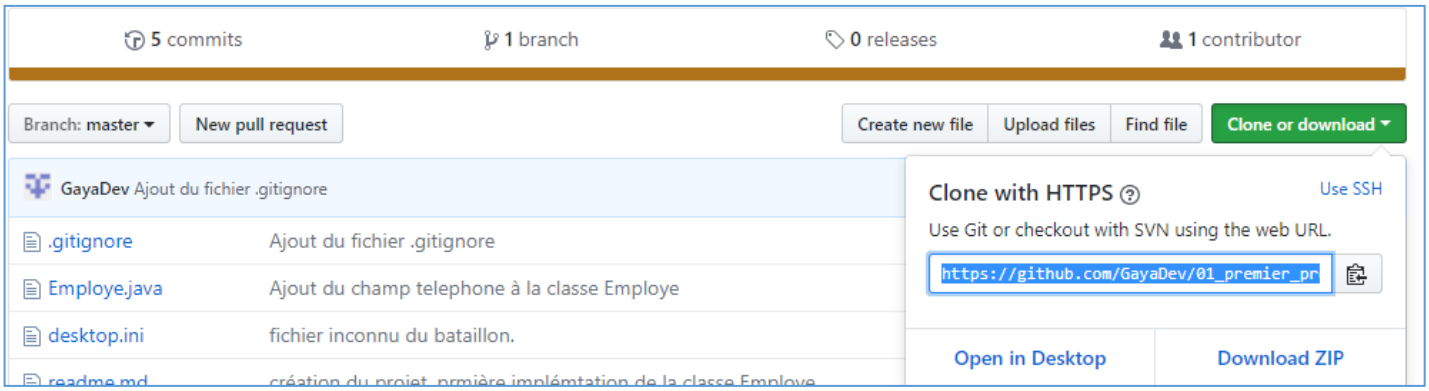


- La 1ere commande à connaitre lorsqu'on n'est pas à l'initiative d'un projet est :

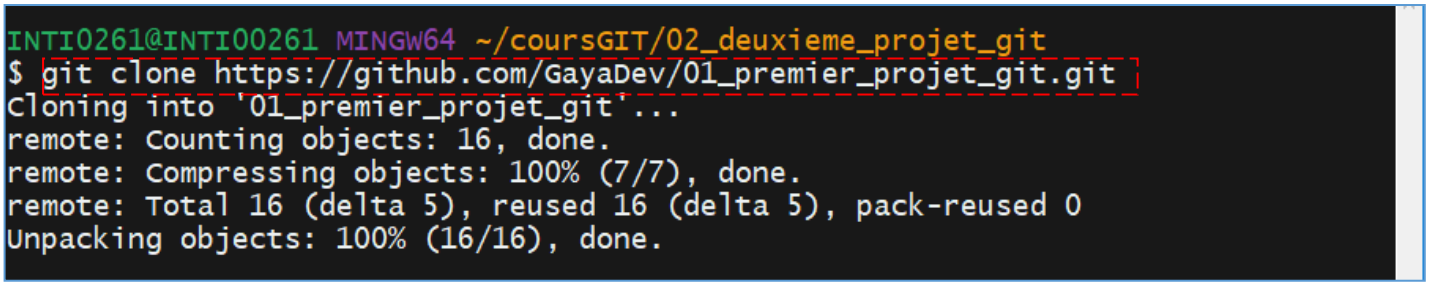
```
git clone
```

- cette commande nous permet d'aller s'abonner à un dépôt distant, de copier l'intégralité du dépôt et son historique et de le ramener chez nous en tant que dépôt local.

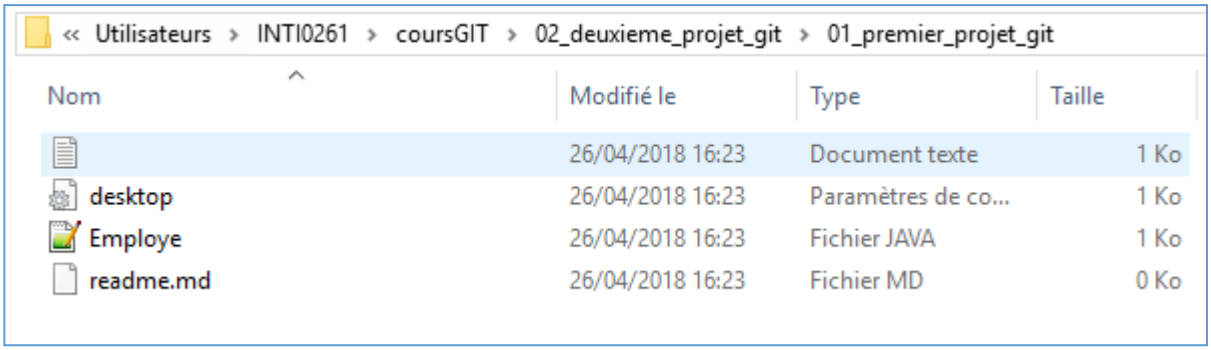
  - Cette commande prend l'adresse du dépôt distant qu'on récupère sur GitHub :




```
git clone  https://github.com/GayaDev/01_premier_projet_git.git
```



  - Ici, il récupère le projet, le dépôt et tout l'historique depuis **`GitHub`** et il le met à notre disposition dans le répertoire :



- On se rend dans ce répertoire avec un **cd**:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git
$ cd 01_premier_projet_git/

INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ |
```

- Un **`git status`** nous informe que :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

- c'est un dépôt git
- qu'on est sur la branche **`master`**
- cette branche est bien à jour par rapport à la branche de l'origine
- il y a rien à commiter
- notre espace de travail est propre.

- Donc la, on peut récupérer les sources pour éventuellement commencer à travailler dessus :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_git
 (master)
$ ls
desktop.ini  Employe.java  readme.md
```

- De l'autre côté (**`01_premier_projet_git`**), le développeur effectue un changement de la classe **`Employe`**

Employe.java

```java
public class Employe{

    private String nom;
    private String prenom;
    private String mail;
    private String telephone;
    private String adresse;

    private int age;

}
```

- A l'aide d'un **`git status`**, il constate que le fichier **`Employe`** est passé de l'état validé à l'état modifié :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ 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:   Employe.java

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

- Il l'indexe avec un **`add`** et vérifie avec un **`git status`**:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git add Employe.java

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

        modified:   Employe.java
```

- Il le commit avec un message :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git commit -m "Ajout du champ age à la classe Employe"
[master fe6914b] Ajout du champ age à la classe Employe
 1 file changed, 2 insertions(+)
```

- Il pousse ces modifications vers **`GitHub`** pour les rendre publique avec un **`push`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git push origin master
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git push origin master
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 343 bytes | 343.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/GayaDev/01_premier_projet_git.git
   b8576b6..fe6914b  master -> master
```

- On pousse de nouvelles choses vers le dépôt distant sur GitHub

- De l'autre côté (**`02_deuxieme_projet_git`**), notre nouveau collaborateur :

  - Un **`git status`** l'informe de rien :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_git
 (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

- Effectivement pour lui la branche est up-to-date dans le sens ou si il ne demande pas la synchronisation manuelle avec le dépôt distant, Git ne le fera pas pour lui.

  - ✘ Il faut comprendre que dans notre Git local, on a 2 espaces :

    - L'**`espace de travail`** qui est le dossier qu'on a dans notre explorateur et qui nous permet de travailler sur nos sources.

    - Et on a **`le dépôt local`** qui est simplement un dépôt, ça veut dire un ensemble de révision, de branches, de tags et autres mais strictement local.

  - ✘ Si on veut synchroniser notre travail avec un dépôt distant et récupérer des modifications, il va avoir 2 étapes :

    - La 1ere : synchroniser le dépôt local avec le dépôt distant

    - La 2eme : synchroniser le dépôt local avec notre working directory.

- Pour synchroniser le dépôt local avec le dépôt distant, on utilise la commande **fetch** :

```
git fetch origin master
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_git
 (master)
$ git fetch origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 3 (delta 2), reused 3 (delta 2), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/GayaDev/01_premier_projet_git
 * branch            master     -> FETCH_HEAD
   b8576b6..fe6914b  master     -> origin/master
```

- Il récupère depuis GitHub certaines choses : il est passé de la révision **b8576b6** à la version **fe6914b** sur la branche origin master.

- Les deux sont synchronisés, pour autant, s'il regarde son fichier source **Employe.java** avec un **cat**, il n'a pas la modification :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_git
 (master)
$ cat Employe.java
public class Employe{

        private String nom;
        private String prenom;
        private String mail;
        private String telephone;

}
```

- Par ce que son espace de travail, ce qu'il voit dans son navigateur n'a pas été altéré.

- Un **git status** l'informe que sa branche, son espace de travail n'est plus **up to date** mais en retard (**behind = derrière**) par rapport à origin master d'un commit :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_git
 (master)
$ git status
On branch master
Your branch is behind 'origin/master' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

- La plusieurs possibilité lui sont offertes :

    - Soit son espace de travail est clair et il va pouvoir du coup faire ce qu'on appel un **merge**, c'est-à-dire prendre les modifications depuis la dernière fois qu'il a synchronisé et les appliquer a son espace de travail et tout va bien se passer.

    - Ou il était entrain de travailler, et du coup il y a des conflits et il va devoir gérer ces conflits à la main et faire des choix entre ce qu'il va garder et ce qu'il ne va pas garder ou ce qu'il va fusionner.

- ✘ Ici Git lui explique que ça peut être **fast-forwarded**, ça veut dire que le **merge** peut se faire d'une manière transparente.

- Donc il va pouvoir **merger**, c'est-à-dire prendre les modifications de son dépôt local et les pousser dans son working directory pour en bénéficier.

    - Pour ça, il lance la commande **merge** pour merger son espace de travail avec autre chose qui est origin master :

```
git merge origin master
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_git
 (master)
$ git merge origin master
Updating b8576b6..fe6914b
Fast-forward
 Employe.java | 2 ++
 1 file changed, 2 insertions(+)
```

- Ici Git passe de la révision **b8576b6**, celle qui la récupéré lors du git clone à la révision **fe6914b**, la nouvelle révision qui a été poussée par son contributeur principale.

- Et Git informe que **Employe.java** à été altéré.

- Un autre **`git status`** l'informe qu'ile est bien synchroniser. Son répertoire local est synchronisé avec son espace de travail tout en sachant que son répertoire local est synchronisé avec le dépôt distant.

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_git
 (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

- Ici, s'il fait un **`cat`**, on affiche le code source de la classe **`Employe.java`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_git
 (master)
$ cat Employe.java
public class Employe{

        private String nom;
        private String prenom;
        private String mail;
        private String telephone;

        private String adresse;

}
```

- On voit bien qu'il bénéficie de la dernière amélioration.

- Donc c'est un véritable point d'attention quand on est sur notre espace de travail :
  - les **`commit`** nous permet de passer de l'espace de travail au depot local.
  - Les **`push`** vont nous permettre de passer du dépôt local au dépôt distant
  - Les **`fetch`** vont nous permettre de récupérer du depot distant de nouvelles informations et les mettre dans le depot local
  - Les **`merge`** vont nous permettre de récupérer depuis notre depot local des modifications et les impacter sur notre répertoire de travail.

- Il existe un raccourci concernant les 2 dernières étapes avec **`git pull`** qui est l'association d'un **`fetch`** et d'un **`merge`** :

  - Les répertoires distants et locaux vont synchronisés
  - Le répertoire local et le working directory vont synchronisés

```
git pull
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_git
 (master)
$ git pull origin master
From https://github.com/GayaDev/01_premier_projet_git
 * branch            master     -> FETCH_HEAD
Already up to date.
```

- A pour effet de faire un fetch et si des modifications ont été récupérées, il va faire un merge

- Dans la <u>réalité</u>, on se sert quasi exclusivement de **`git pull`** <u>et non pas</u> de git **`fetch`** et git **`merge`**

- Néanmoins attention : lorsqu'on fait un **`git pull`** étant donner que notre espace de travail va être synchronisé avec le répertoire local, On doit être sure qu'aucune modification est en cours sinon le conflit sera systématique.

V. Système des branches

1. Présentation de systèmes de branches

- Nous allons maintenant découvrir le système de branches qui est très souvent au cœur de la stratégie de gestion de version dans un projet.

- Ce système va nous permettre de suivre le cycle de vie de l'application, d'apporter des modifications successive tout en gardant suffisamment de souplesse pour revenir et corriger par exemple une version qui est en production, tout ça au sein du même dépôt sans qu'il y est de collision en tout ces mondes.

- Avec Git, on développe toujours en local et on finit par pousser les modifications sur le dépôt distant, c'est la techno qui nous l'impose.

- Maintenant, comment organiser les branches dans notre projet ?

- Une branche, c'est simplement une divergence dans le développement

- Par exemple, on a commencé de travailler sur une version v0.1. cette version est constituée d'un tas de choses, notamment d'un lot de fonctionnalités, métier ou technique par exemple, on peut déclarer en ce moment la une branche.

- Ce qui fait qu'il va y avoir 2mondes : le monde de la branche **`master`** sur lequel il ne se passera grand-chose et le monde de la branche Lot1.

- Potentiellement, on pouvait continuer à travailler sur la branche master et travailler sur la branche Lot1 qui ne se voit pas. C'est vraiment 2 mondes séparés, c'est comme si on cloné le projet. Néanmoins ce qui est intéressent, c'est ce que, on travailler sur Lot1 qui

est séparée, les commit et les push sont bien séparés d'une branche à l'autre. Et à un moment donné on va pouvoir se rejoindre à la branche master, et tout ce qui a été fait dans Lot1, une fois que c'est finis de développer, testé et stabilisé, on va pouvoir le merger, c'est le terme, sur la branche principale et du coup la branche principale rattrapera son retard sur la branche Lot1 et bénéficiera des dernières fonctionnalités.

- Ce qui fait potentiellement, on peut avoir plusieurs branches en même temps (Lot1, Lot2, Lot3,…etc), chaque développé dans son coin et une fois que chaque Lot est stabilisé, on les rassemble au sein de la même branche et on relance une compagne de tests pour vérifier que les lots communique bien entre eux.

- Cette nécessité d'avoir des branches, nécessité opérationnelle, va nous permettre de pouvoir gérer plusieurs versions simultanées, de faire des expérimentations, porter des gros changements sans s'impacter les uns les autres.

- L'organisation type est une organisation par versions :

  - Sur notre branche master, on va avoir que des versions qui vont être validées et vont testées (version 0.1, version 0.2, version 1.0, …etc)

  - A chaque fois qu'on veut développer quelque chose, on va donc tirer une branche puis on développe, on commit, on push, …etc à plusieurs développeurs par exemple.

  - Et seulement une fois que ce développement sera stabilisé, on va l'envoyer directement sur notre branche master, par exemple en version 0.2.

  - Donc une branche par version de manière à bien isoler tous les développements. Ce qui va nous permettre de bien isoler, de bien segmenter notre organisation de manière à développer des fonctionnalités les unes à côté des autres et pas une espèce de brouhaha ou tout se mélangerais.

Pour résumer :

- Une branche par version.

  La branche principale est la branche **`master`** qui va stocker tout ce qui est stable dans le projet et on va tagguer chaque commit pour indiquer par exemple, ça c'est la v0.1, etc et d'éviter d'utiliser les numéros de commit comme on la vu portant des noms pas très porteurs d'informations (**`b789086`**).

- On tire une branche par lot ou par fonctionnalité de manière à isoler le développement et à les intégrer si seulement ils sont stable.

Exemple :

- Je suis seul développeur sur un projet, ceci dit, je n'est pas besoin de branches.
- J'ai la v1.0 qui part en production.
- Je tire une branche et je commence à travailler sur la v1.1 ou il y a plusieurs fonctionnalités.
- Mon chef de projet m'appel et me dit, attention, sur la v1.0, il y a un bug à corriger d'une manière urgente.
- Pas de problèmes, je suis entrain de travailler sur la version 1.1, je fais un commit de mon travail.
- Je repasse sur la branche master, je tire une branche pour la correction, je corrige, je réintègre cette branche en version 1.0.1et je pars travailler sur la version 1.1.

- Donc ça m'apporte énormément de souplesse en tant que développeur même si je suis tout seul puisque j'isole vraiment les versions et les lots.

- Et au cas ou je doit revenir sur un problème, je peut switcher mon espace de travail pour passer sur la branche a, sur la b, sur la c, travailler un peut sur la v1.0, l'après midi me mettre à la v1.1 et puis en soirée passer sur le prototype de la version v2.

2. Créer une branche

- D'une manière plus concrète, on va voir comment créer une branche, travailler dessus et changer pour passer d'une branche à l'autre

- Pour créer une branche, Git nous propose la commande branch suivi d'un nom :

```
git branch v2.0
```

  - On se met sur le 1ere projet : **`01_premier_projet_git`** et on crée la branche v2.0 :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git branch v2.0

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ |
```

- On va bien ici qu'on reste sur la branche **`master`**

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git status
On branch master
nothing to commit, working tree clean
```

  - Pour passer et switcher vers la branche v2.0, on utilise la commande :

```
git checkout v2.0
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git checkout v2.0
Switched to branch 'v2.0'

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.0)
$ |
```

- **Moralité** : Si on fait des modifications, elles n'affecteront pas la branche **`master`**.

- On a un raccourci pour créer une branche et passer dessus :

```
git checkout v2.0 -b v2.1
```

  - je suis dans la branche **v2.0** et je crée la branche **v2.1** et je passe dessus.

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.0)
$ git checkout v2.0 -b v2.1
Switched to a new branch 'v2.1'

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.1)
$ |
```

- Maintenant on à 3 branches dans le projet, la branche **master**, la **v2.0** et la **v2.1**
  - Pour passer et switcher de l'une à l'autre, on utilise checkout :

```
git checkout master
```

- On switch vers la branche **master**

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.1)
$ git checkout master
Switched to branch 'master'

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ |
```

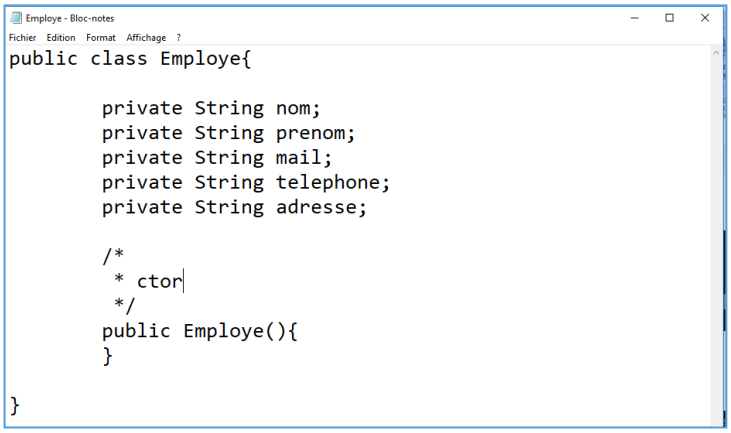
- On va travailler sur la branche **v2.0** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git checkout v2.0
Switched to branch 'v2.0'

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.0)
$ |
```

- On va modifier notre fichier source :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.0)
$ notepad Employe.java
```

- On ajoute le constructeur :



- On sauvegarde et on ferme le fichier pour avoir la main dans la console Git Bach

- Le **`git status`** nous avertis qu'un fichier à été modifié :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.0)
$ git status
On branch v2.0
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:   Employe.java

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

- On ajoute pour indexer le fichier avec **`git add`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.0)
$ git add Employe.java
```

- On fait un **`commit`** avec un message :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.0)
$ git commit -m "Ajout du constructeur par défaut de la classe Employe"
[v2.0 609fa28] Ajout du constructeur par défaut de la classe Employe
 1 file changed, 6 insertions(+), 1 deletion(-)
```

- On refait un **`git status`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.0)
$ git status
On branch v2.0
nothing to commit, working tree clean
```

- On affiche le code source avec **`cat`** pour voire les modifications :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.0)
$ cat Employe.java
public class Employe{

        private String nom;
        private String prenom;
        private String mail;
        private String telephone;
        private String adresse;

        /*
         * ctor
         */
        public Employe(){
        }

}
```

- On va se réaligner sur la branche **`master`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (v2.0)
$ git checkout master
Switched to branch 'master'

INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$
```

- On affiche le code source de la classe **`Employe`** avec **`cat`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ cat Employe.java
public class Employe{

        private String nom;
        private String prenom;
        private String mail;
        private String telephone;

        private String adresse;

}
```

- On a bien une divergence de notre développement.

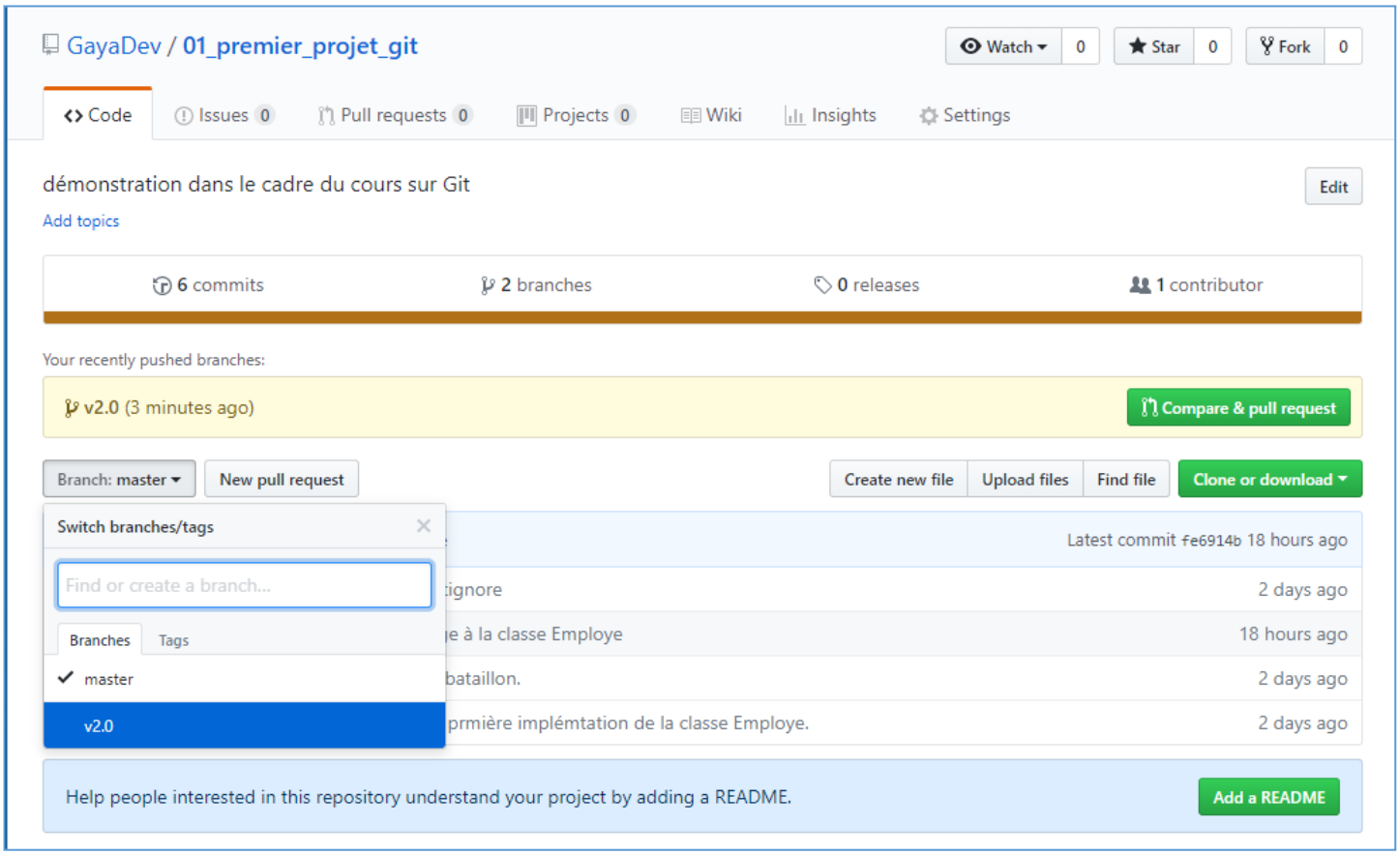
- On a bien 2 mondes ou 2 branches bien séparés, qui contiennent leurs propres versions, leurs propres commits, leurs propres codes sources et qui vont suivre sans instructions contraires, donc sans merge, un cycle de vie complètement séparé avec des fonctionnalités différentes, … etc

- On va maintenant pousser avec un **`push`** le contenu de la branche **`v2.0`** vers le dépôt distant **`origin:`**

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git push origin v2.0
Counting objects: 3, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 407 bytes | 407.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/GayaDev/01_premier_projet_git.git
 * [new branch]      v2.0 -> v2.0
```

- Tout se passe bien : il y a une nouvelle branche qui a été crée sur le distant

  - Si on se rend sur **`GitHub`** (rafraichir):



- On va bien qu'il ya 2 branches master et v2.0.

- Ce qui est encore une fois important: on fait passer au dépôt distant que ce qu'on souhaite.

  Si la branche v2.1 est uniquement à usage interne par exemple, on est obligé de la pousser.

---

3. Fusionner deux branches

- Maintenant qu'on a crée plusieurs branches qui contiennent une version spécifique et dédiées du projet.
- On va apprendre à rassembler ces branches, à les merger pour refusionner le développement et pour repartir sur une base commune.
- Ici, on a 2 branches :
  - une branche master qui a un temps de retard sur la branche v2.0
  - et une branche v2.0 sur laquelle on a fait une évolution.
- Pour **`merger une branche`**, c'est très simple :
  - On se positionne sur la branche de destination, ici **`master`**
  - Et on va engager un **`merge`**

```
git merge v2.0
```

> merge le contenu de la branche **`v2.0`** dans la branche **`master`**

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git merge v2.0
Updating fe6914b..609fa28
Fast-forward
 Employe.java | 7 ++++++-
 1 file changed, 6 insertions(+), 1 deletion(-)
```

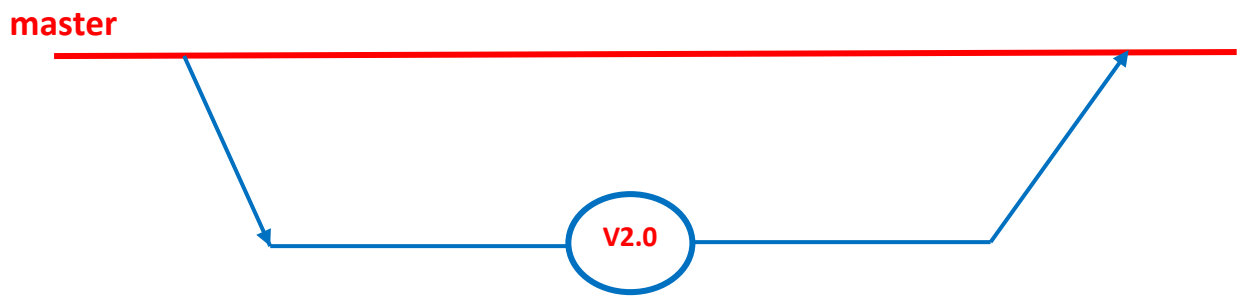
- Etant donné que le **`master`** n'a pas subi de modifications depuis que la branche a été faite, il y a un **`Fast-forward`** qui a été fait, ça veut dire qu'il n y a pas de questions à se poser.
- de toute façon le master n'a pas bougé, la version v2.0 à bougée et on réintègre nos modifications.

- Ici, si on regarde notre **`Emplye.java`** avec un **`cat`** :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ cat Employe.java
public class Employe{

        private String nom;
        private String prenom;
        private String mail;
        private String telephone;
        private String adresse;

        /*
         * ctor
         */
        public Employe(){
        }

}
```

- La classe Employe.java à bénéficiée des modifications.

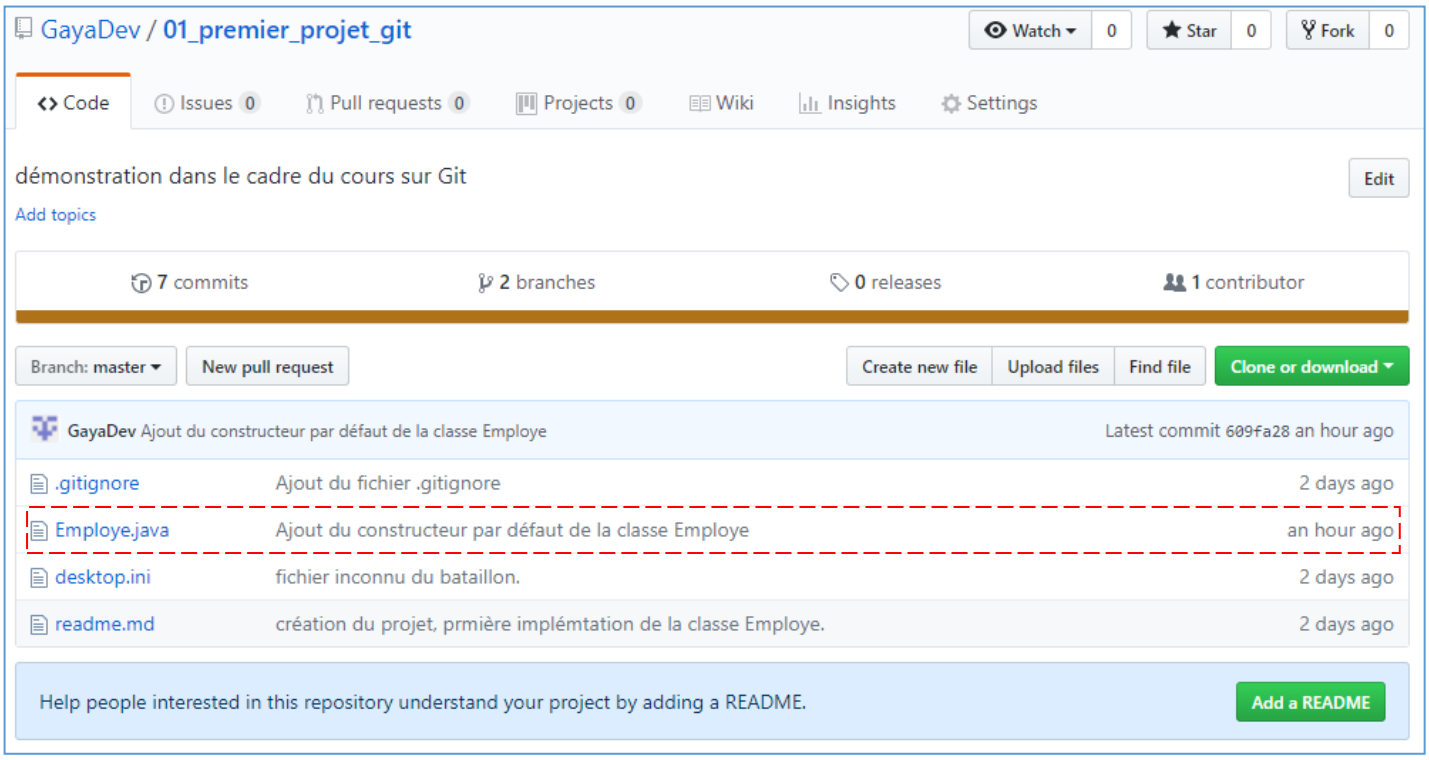
- Donc on a eu notre master.
- La branche v2.0 est partie dans son coin faire ses évolutions
- Par contre on a rassemblé le contenu de v2.0 avec le master
- Et maintenant notre master est bien aligné avec la v2.0 et bénéficie des mêmes fonctionnalités



- Si maintenant on fait un **`push`** vers le distant :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git push origin master
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/GayaDev/01_premier_projet_git.git
   fe6914b..609fa28  master -> master
```

  - Si on rend sur **`GitHub`** :



- Notre branche master à bien qui été faite il y a 1 heure pourtant on l'a fait sur la branche v2.0 mais qui a été reporté directement sur la branche master.

- Dernière chose sur le merge :

  - Lorsqu'on fait un merge, c'est généralement pour indiquer qu'on rejoint le développement et c'est quelque chose d'important. C'est certainement une nouvelle version ou autres.

  - Dans ce cas la, on peut ajouter le concept de **`tag`** qui va nous permettre de <u>positionner une étiquette sur ce commit</u> la pour dire que ça, c'est ma version v2.0

- Donc ici, on va créer un tag :
    - avec la commande **tag**
    - suivi du nom avec l'option **-a**
    - suivi du commentaire avec l'option **-m**
    - A la fin on indique sur quelle branche, par exemple le **master**

```
git tag -a 2.0 -m "mise en production de la 2.0"  master
```

Attention :

- Ne pas utiliser le même nom pour le tag et la branche.
- ça peut créer une confusion et Git ne sera pas faire le choix entre les deux.

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git tag -a 2.0 -m "mise en production de la 2.0" master
```

- Donc la maintenant on a bien un tag.

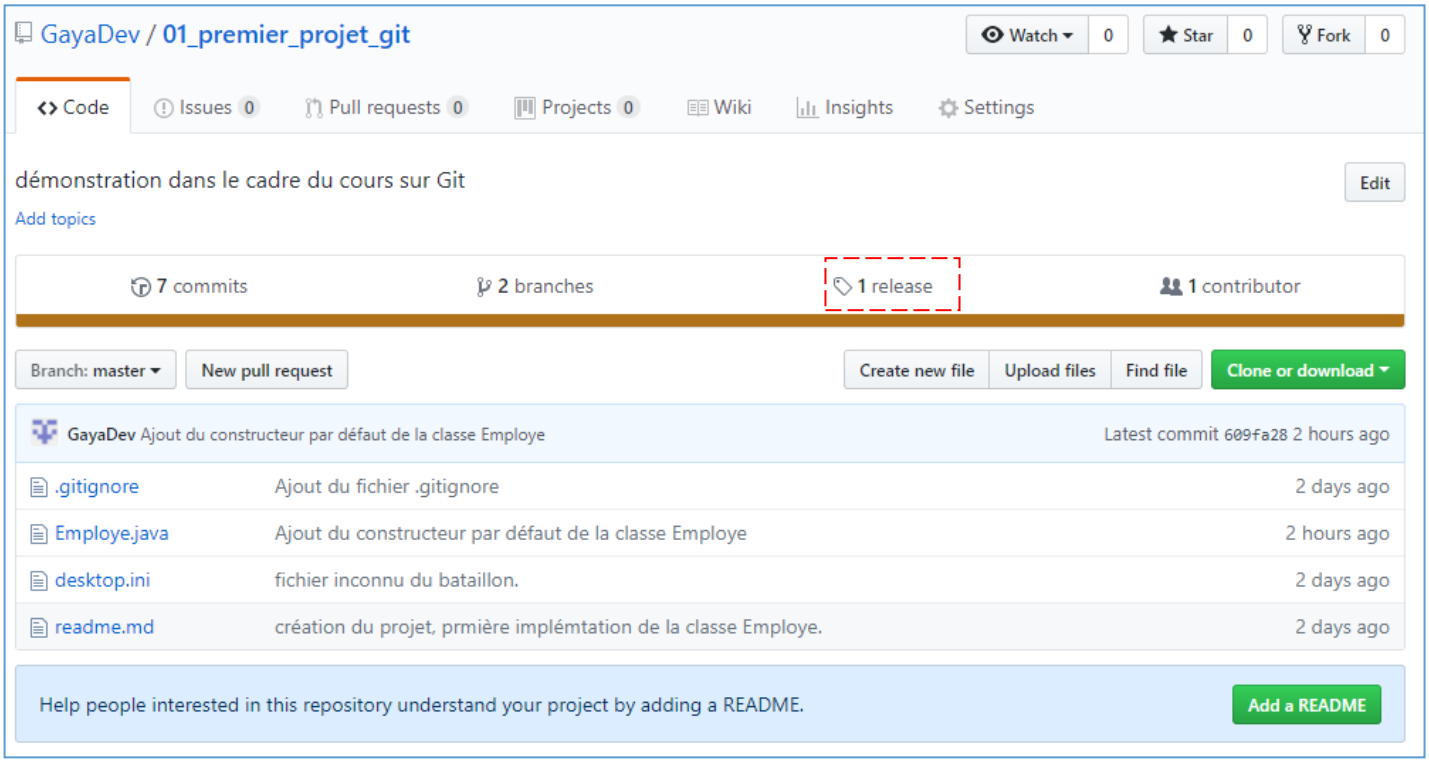
- Attention, il est à noter que si on fait un **git push** pour pousser dans le distant, il ne se passera absolument rien.

  Il faut ajouter l'option **--tags** qui va permettre de pousser en même que le code source et les révisions, les tags :

```
git push origin master --tags
```

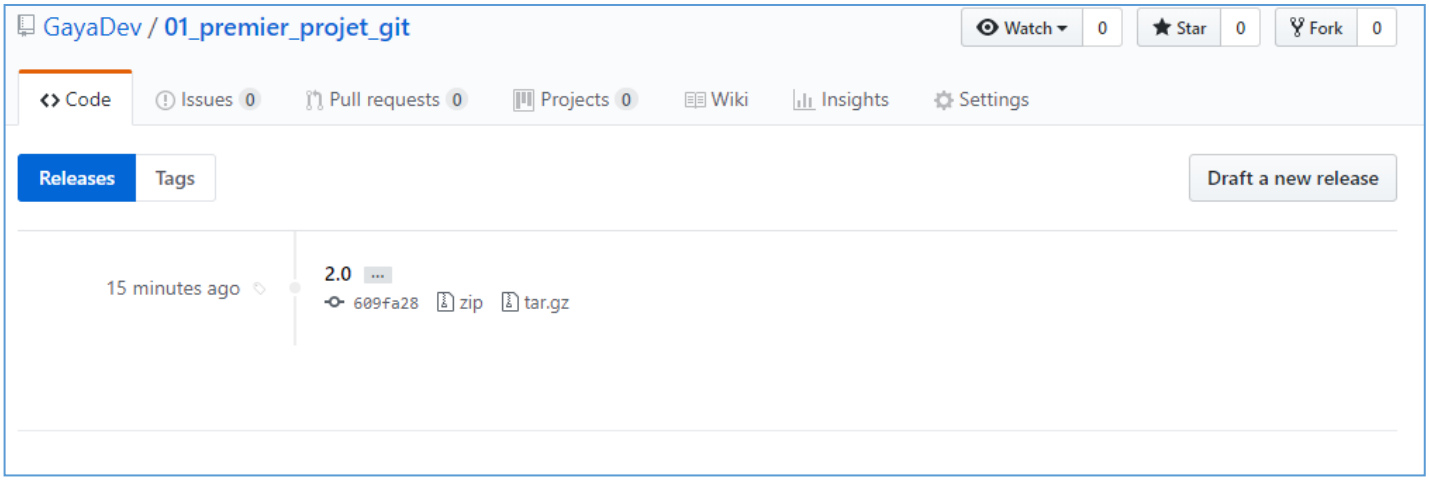
```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git push origin master --tags
Counting objects: 1, done.
Writing objects: 100% (1/1), 169 bytes | 169.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To https://github.com/GayaDev/01_premier_projet_git.git
 * [new tag]         2.0 -> 2.0
```

- Si on rend sur **`GitHub`** :



- On va bien qu'on a un **`release`**, un tag qui est la **2.0**

- Et ce qui est pratique pour l'utilisateur, ça peut être consultable directement ou être téléchargeable en zip ou gz :



4. Gérer les conflits

- Jusqu'à maintenant nos 2 développeurs, nos 2 contributeurs à notre projet ont eu le bon gout de ne pas développer en même temps des fonctionnalités.

- Donc tous les phénomènes de push, de pull, de merge, de création de branche et de fusion n'ont eu absolument aucune incidence en termes de conflits.

- Maintenant nous allons provoquer un conflit entre notre 1er développeur (**`01_premier_projet_git`**) qui a fait sa branche de 2.0 et qui a rapporté quelque chose sur le master et notre 2eme développeur (**`02_deuxieme_projet_git`**) qui est resté sur la version d'il y a peut près une petite heure.

- Sur le projet : **`02_deuxieme_projet_git`**

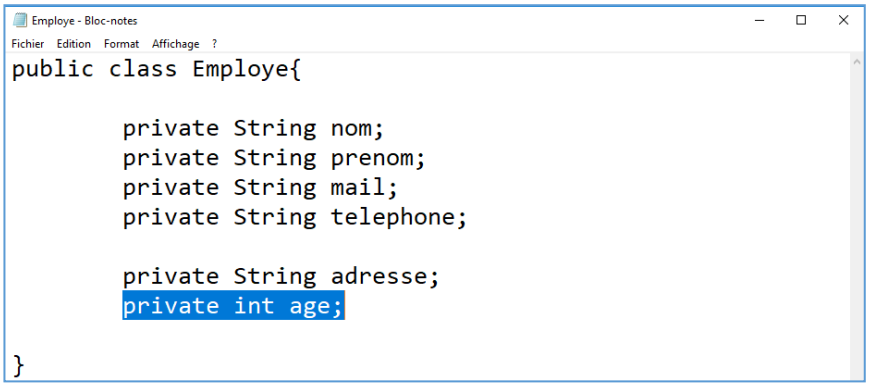
  - Si on fait un **`git status`** :

    - Moi en tant que développeur, tout est bon. Je n'est pas vu que le 1er développeur a fait des modifications, donc je continu à travailler.

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

  - Ici, pour corser un peut l'exercice, on va modifier le code source de la classe **Employe.java** et ajouter le champ age:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ notepad Employe.java
```



- Un git status m'informe de la modification dans le fichier:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ git status
On branch master
Your branch is up to date with 'origin/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:   Employe.java

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

- Je dois l'indexer avec add :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ git add Employe.java
```

- Je fais mon commit avec un message:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ git commit -m "Ajout du champ age à la classe Employe"
[master f910b50] Ajout du champ age à la classe Employe
 1 file changed, 1 insertion(+)
```

- Très bien, mon git status m'informe que tout va bien:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

  - Par contre, il nous dit que je suis derrière la branche origin/master
  - J'étais up-to-date et la je suis derrière.

- Donc la attention, je vais engager un **`pull`** en me disant que toute façon, on a collaboré sur le même projet mais ça me choque pas, je sais que je ne suis pas tout seul :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ git pull origin master
remote: Counting objects: 3, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 3 (delta 1), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/GayaDev/01_premier_projet_git
 * branch            master     -> FETCH_HEAD
   fe6914b..609fa28  master     -> origin/master
Auto-merging Employe.java
CONFLICT (content): Merge conflict in Employe.java
Automatic merge failed; fix conflicts and then commit the result.
```

- Et la contrairement aux fois précédentes, le **`pull`** qui consiste en un fetch, synchronisation entre dépôt distant et depot local et le merge, synchronisation entre le depot local et mon espace de travail,
  M'indique qu'il ya un conflit dans le fichier source Employe.java

```
CONFLICT (content): Merge conflict in Employe.java
```

  - Et ce conflit, il ne peut pas prendre de décision, on a fait impacter la même ligne.

  - Il nous indique qu'il faudra le résoudre pour qu'on puisse faire quelque chose.

  - On voit bien ici que la branche à changée. On s'est détachée de la ligne de vie master et on est à côté, complètement instable.

    Il va falloir faire quelque chose pour pouvoir raccrocher les wagons

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master|MERGING)
$
```

- Si je regarde dans mon depot, il se passe rien de particulier :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master|MERGING)
$ ls
desktop.ini  Employe.java  readme.md
```

- Par contre si j'ouvre Employe.java :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master|MERGING)
$ notepad Employe.java
```

```
public class Employe{

        private String nom;
        private String prenom;
        private String mail;
        private String telephone;
        private String adresse;
<<<<<<< HEAD
        private int age;
=======

        /*
         * ctor
         */
        public Employe(){
        }
>>>>>>> 609fa28f7c435f477d99b8c25974fe115a6b90a7

}
```

- Mon code est complètement altéré et notamment on me dit plusieurs choses :

  - Dans le HEAD, ma version de tête a moi, la version de mon espace de travail, j'ai la ligne « private int age »

```
<<<<<<< HEAD
        private int age;
=======
```

  - Et depuis le dépôt local qui a été synchronisé avec le dépôt distant et qui porte le numéro de commit **609fa28f7c435f477d99b8c25974fe115a6b90a7** contient les lignes :

```
        /*
         * ctor
         */
        public Employe(){
        }
```

- Et il me dit je ne sais pas quoi faire ?

- Rassurez-vous ici, pour bien comprendre le phénomène on se glisse dans le fichier source et on va résoudre le conflit à la main.

  Dans la pratique, on a des outils intégrés aux IDEs qui nous permettent de régler ces conflits d'une façon beaucoup plus visuelle.

- Ici, il y a une décision à prendre entre ce qui se passe chez moi et ce qui s'est passé en local.
  - Et par exemple, je peux prendre la décision de régler le conflit à la main :
    - Je mixe un petit peut tout le monde, je garde les deux :

```
public class Employe{

        private String nom;
        private String prenom;
        private String mail;
        private String telephone;
        private String adresse;
        private int age;

        /*
         * ctor
         */
        public Employe(){
        }
}
```

- Et ça vous voyez bien que seul un développeur peut le faire.
- Le programme ne peut pas prendre l'initiative de dire que je vais mixer es deux.

- J'ai finis de résoudre mon conflit mais je suis toujours en merge :

  - Un git status me dit qu'il y a des choses en état de merge.

  - Et je dois en sortir pour pouvoir continuer de travailler

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master|MERGING)
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)

        both modified:   Employe.java

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

- D'une manière toute a fait classique, je dois faire un **`add`** dessus :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master|MERGING)
$ git add Employe.java
```

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master|MERGING)
$ git status
On branch master
Your branch and 'origin/master' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:

        modified:   Employe.java
```

- On termine par un **`commit`** pour me sortir dans cette situation:

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master|MERGING)
$ git commit -m "résolution du conflit"
[master 13faae0] résolution du conflit

INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$
```

- Ici on va bien qu'on raccroche le wagon de nouveau avec la branche master.

- Un **`git status`** me dit que tout va bien :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

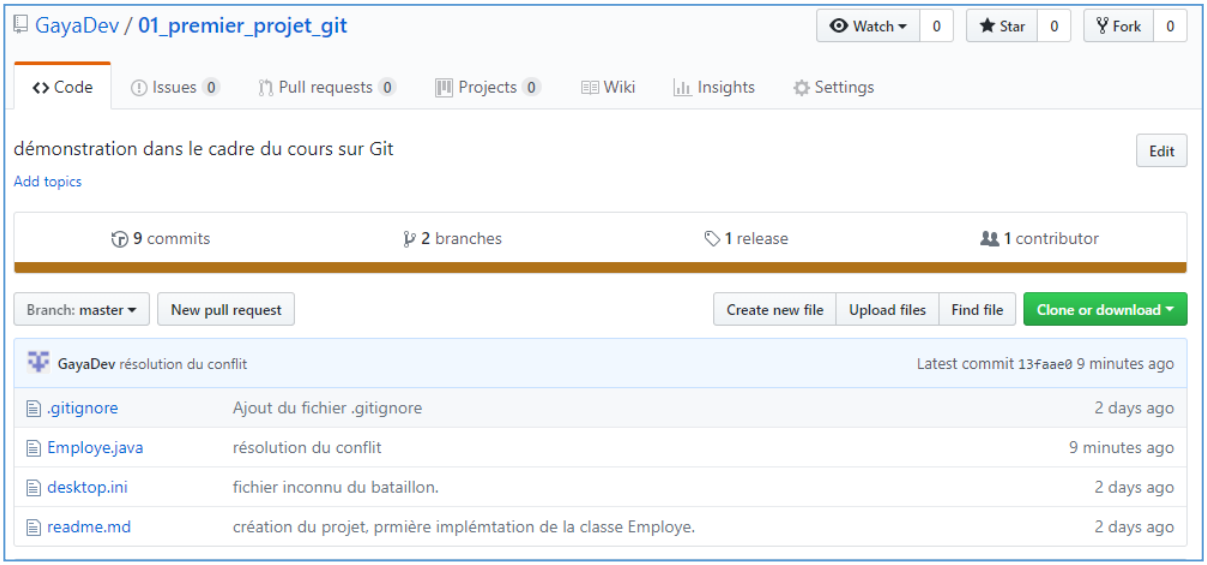
- Je vais pouvoir enchainer avec un **`git push`** pour pousser vers le distant :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ git push origin master
Counting objects: 6, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 619 bytes | 619.00 KiB/s, done.
Total 6 (delta 4), reused 0 (delta 0)
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
To https://github.com/GayaDev/01_premier_projet_git.git
   609fa28..13faae0  master -> master
```

- Un **`git status`** m'indique que la ou le temps j'avais une avance (**`ahead`** dans le précédent get status) sur le dépôt distant, maintenant tout est propre :

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/02_deuxieme_projet_git/01_premier_projet_g
it (master)
$ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
```

- Dans **`GitHub:`**



- Notre **Emplye.java** à bénéficié d'un nouveau commit de notre 2eme contributeur.

- Et si on regarde le code source, on voit bien les changements suite au règlement du conflit :

Branch: master ▾ 01_premier_projet_git / Employe.java Find file Copy path

GayaDev résolution du conflit 13faae0 13 minutes ago

1 contributor

17 lines (13 sloc) | 236 Bytes Raw Blame History

```
1  public class Employe{
2
3          private String nom;
4          private String prenom;
5          private String mail;
6          private String telephone;
7          private String adresse;
8          private int age;
9
10         /*
11          * ctor
12          */
13         public Employe(){
14         }
15 }
16
```

- Si on se remet dans le contexte de notre 1er contributeur (projet : `01_premier_projet_git`)
  - Il suffit de faire un **`pull`** pour récupérer les dernières modifications

```
INTI0261@INTI00261 MINGW64 ~/coursGIT/01_premier_projet_git (master)
$ git pull origin master
remote: Counting objects: 6, done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 6 (delta 4), reused 6 (delta 4), pack-reused 0
Unpacking objects: 100% (6/6), done.
From https://github.com/GayaDev/01_premier_projet_git
 * branch            master     -> FETCH_HEAD
   609fa28..13faae0  master     -> origin/master
error: Your local changes to the following files would be overwritten by merge:
        Employe.java
Please commit your changes or stash them before you merge.
Aborting
Updating 609fa28..13faae0
```

On doit avoir ce résultat :

```
cedric@DESKTOP-5JRMCAA MINGW64 ~/Desktop/Vidéo GIT (master)
$ git pull origin master
From https://github.com/cmillauriaux/demonstration-video-eni
 * branch            master     -> FETCH_HEAD
Updating 51ee5c9..b95a4ff
Fast-forward
 source.js | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Conclusion

- On l'a vu, avec GIT une fois qu'on maitrise bien les concepts et qu'on a apprivoisé les outils, ça se résume finalement à 4 ou 5 commandes différentes.

- Néanmoins ça demande comme même une gymnastique de l'esprit pour savoir comment situer le working directory, comment bien situer le dépôt local, situer le dépôt distant, comment collaborer intelligemment et comment créer des branches et les fusionner.

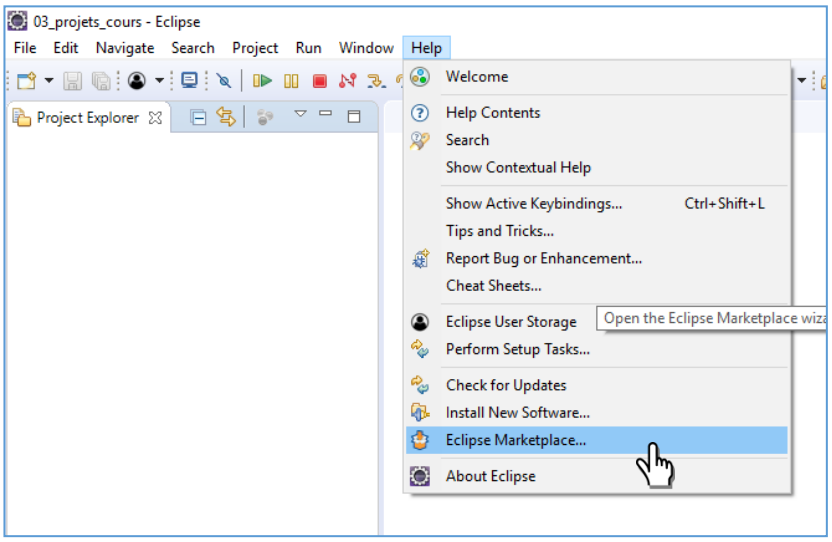
- Néanmoins une fois que toutes ces notions sont bien apprivoisées, l'utilisation de Git au quotidien est quelque chose de très intuitif et très facile toute en sachant qu'ici pour la bonne compréhension de Git, nous avons utilisé uniquement les lignes de commandes.

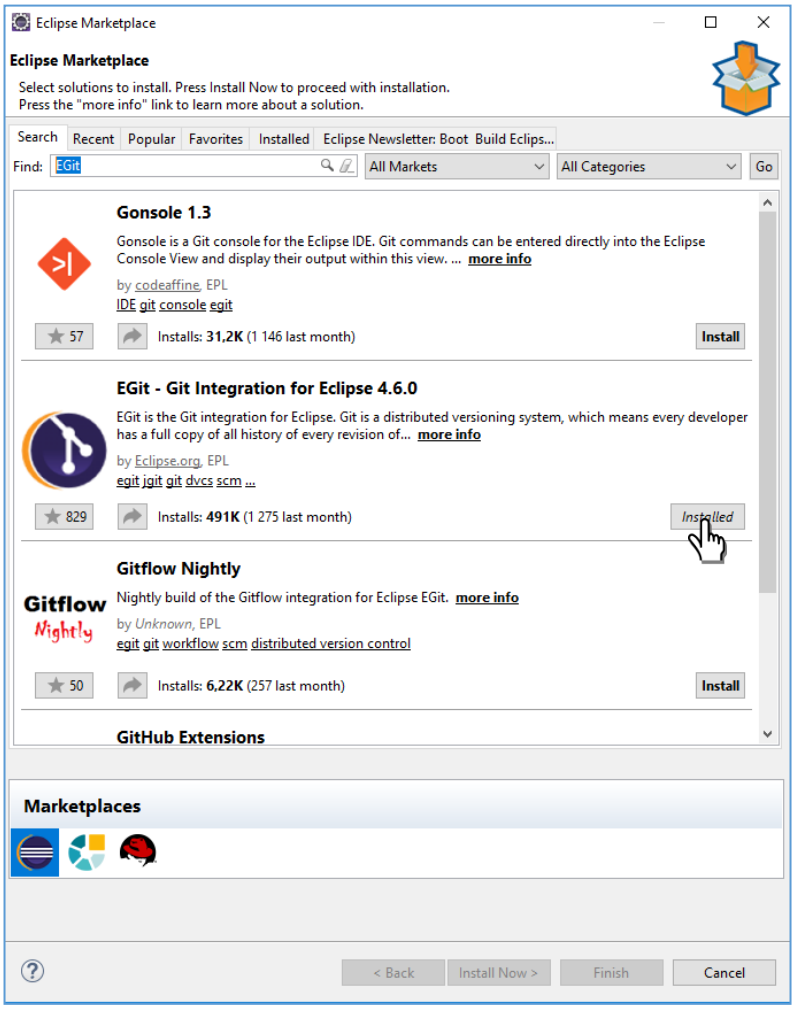
- Bien sure tous les IDEs moderne type Eclipse propose des assistants qui vont nous permettre de régler automatiquement la majorité des problèmes et réaliser la majorité des opérations d'une manière toute a fait naturelle et qui nous rendrons la main uniquement en cas de conflit par exemple.

VI. Intégration de Git et GitHub avec Eclipse

1.installation du plugin EGit sur Eclipse :

- Pour l'intégration de Git dans Eclipse, on doit installer le plugin **EGit** :

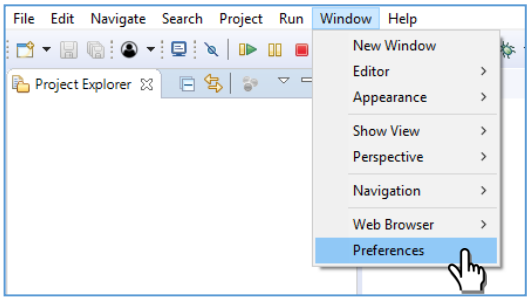




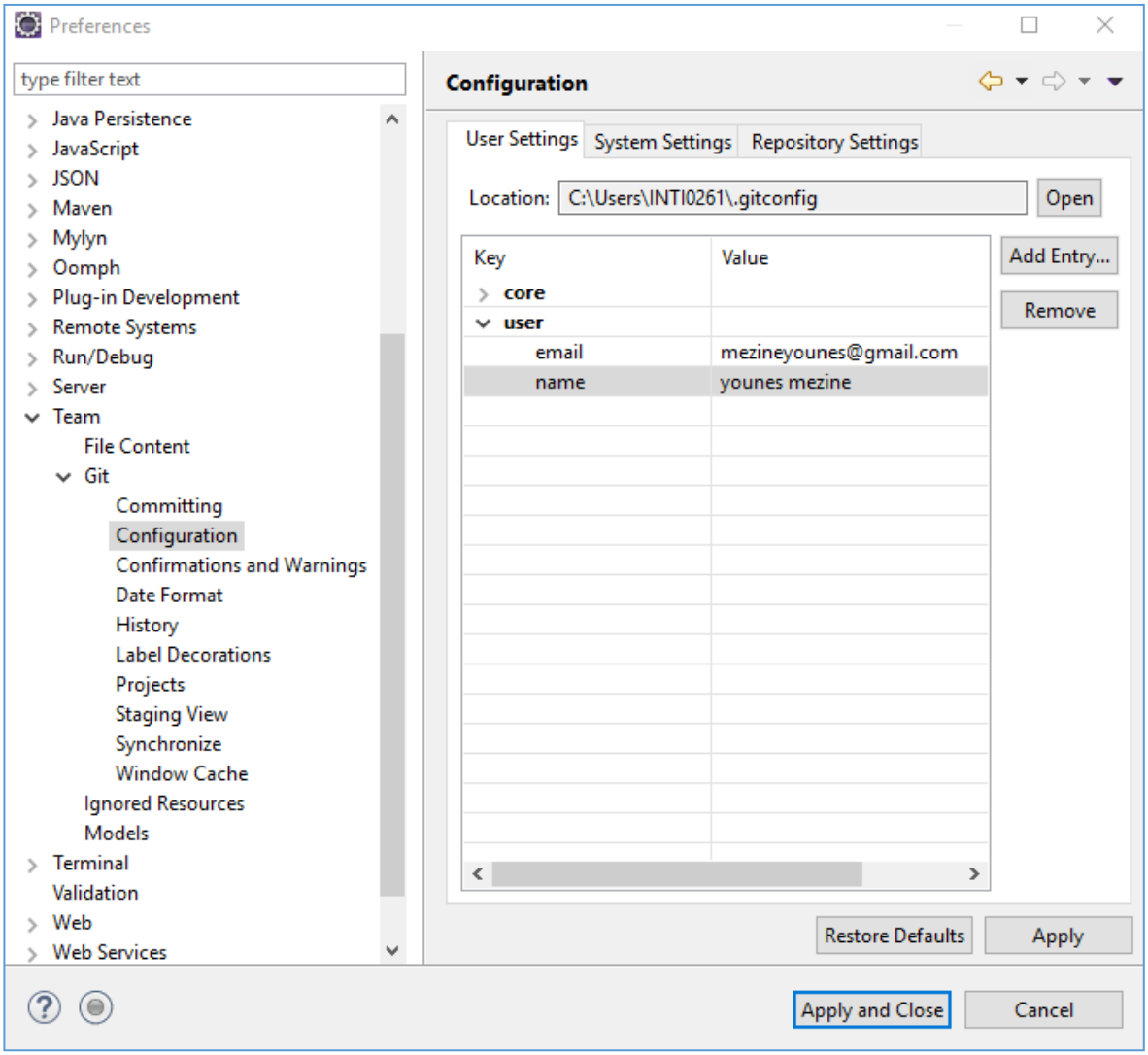
2. EGit Configuration

- Pour la configuration, on doit fournir à Git un **`username`** pour qu'il puisse nous identifier en tant que développeur et un **`mail`** en cas ou un autre développeur veut nous contacter.

- Pour la mise en place de la configuration :
  - On passe par : **`Window`** => **`Preferences`**

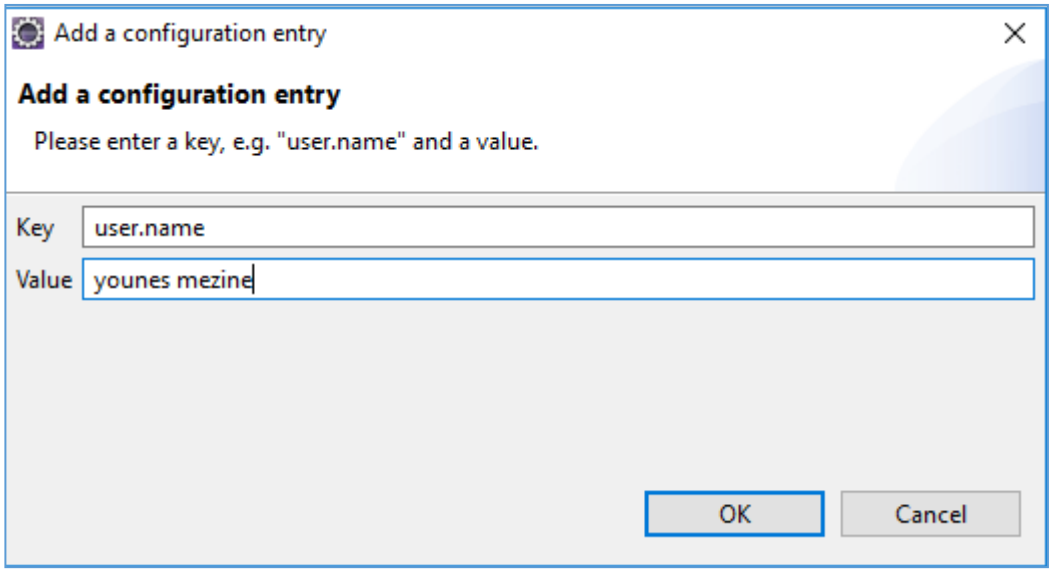


  - Puis : **`Team`** => **`Git`** => **`Configuration`**

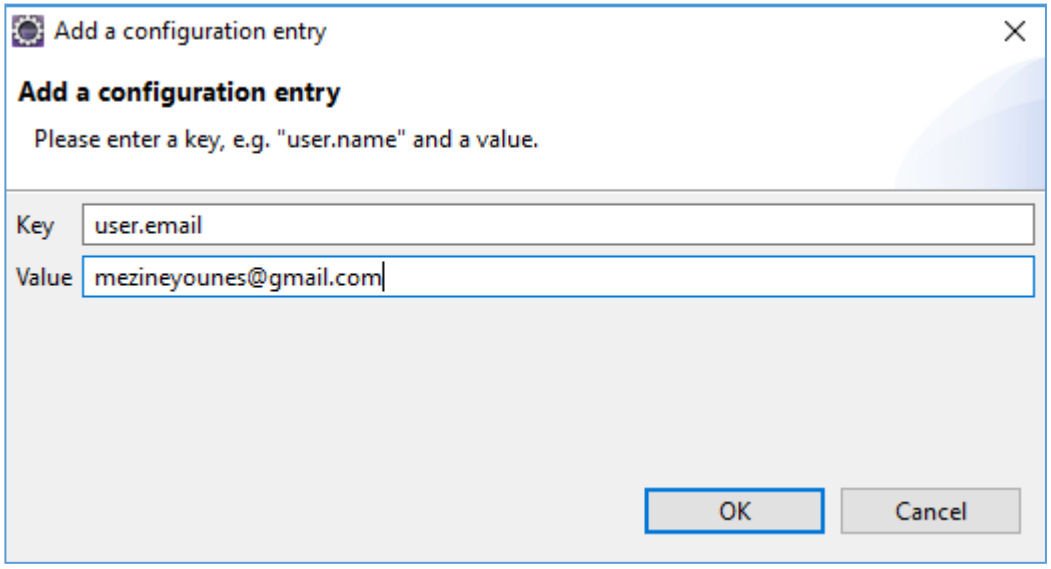


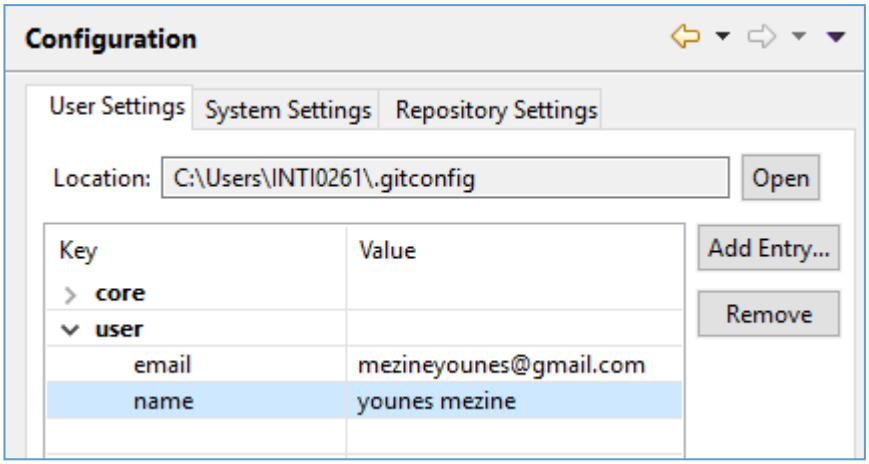
- Puis : => **`Add Entry`**

  - saisir comme Key : **`user.name`** et la valeur



  - refaire la procédure pour la clé : **`user.email`** et sa valeur



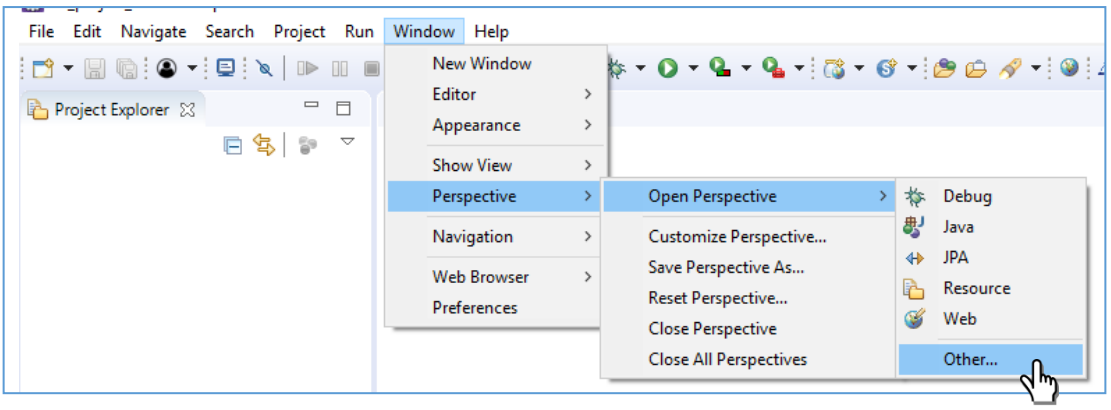


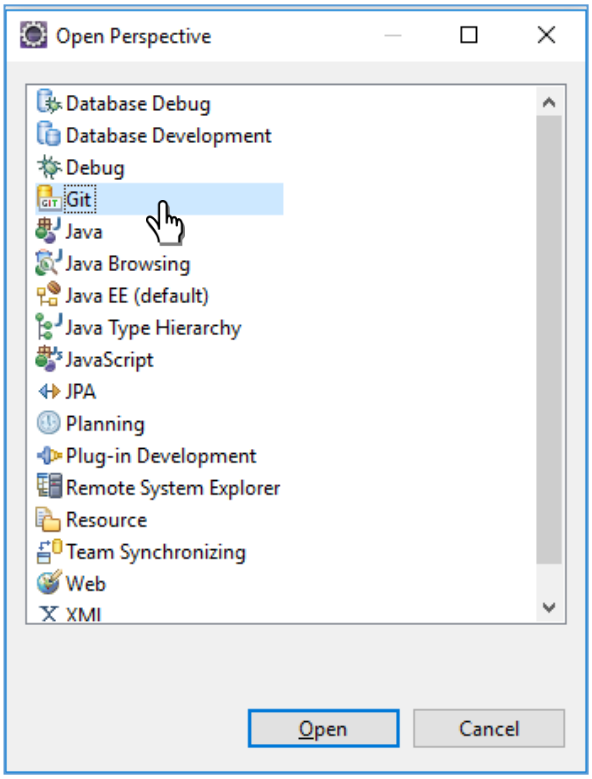
- Notre configuration Git est terminée, il sait qui on est, il sait nous contacter.
- On peut commencer à l'utiliser.

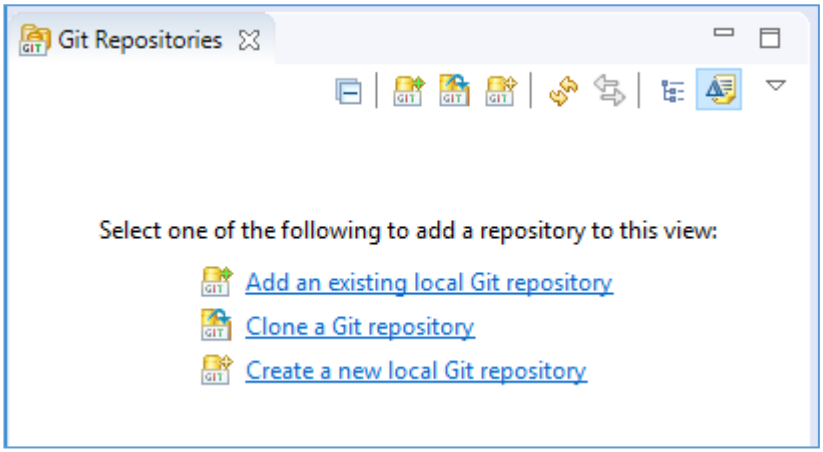
3.Création d'un dépôt Locale (Local Repository)

- Pour ajouter un dépôt local avec Eclipse, on va utiliser la perspective Git.

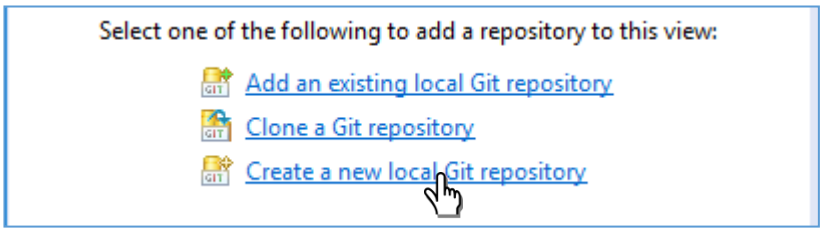
- Pour y accéder :

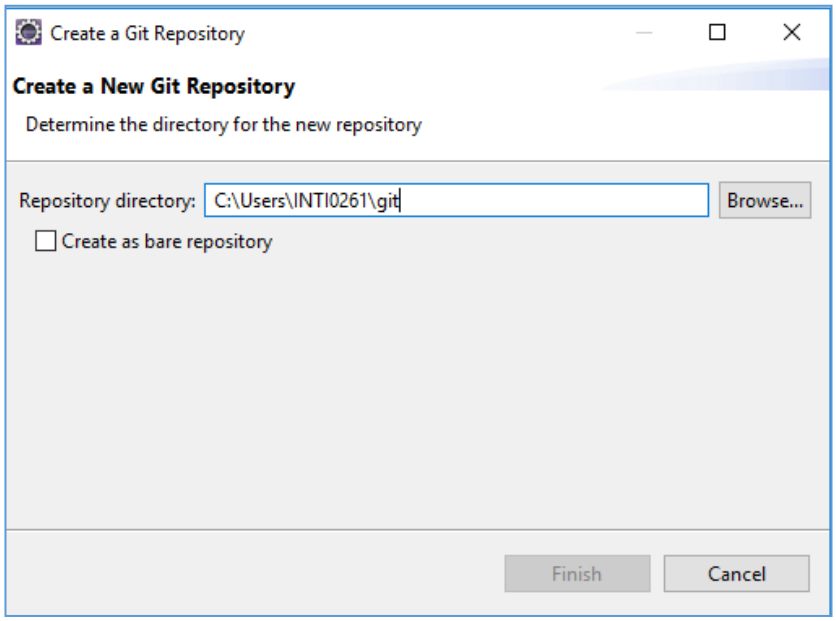






- Pour créer un nouveau dépôt local, on utilise le lien dans la perspective:

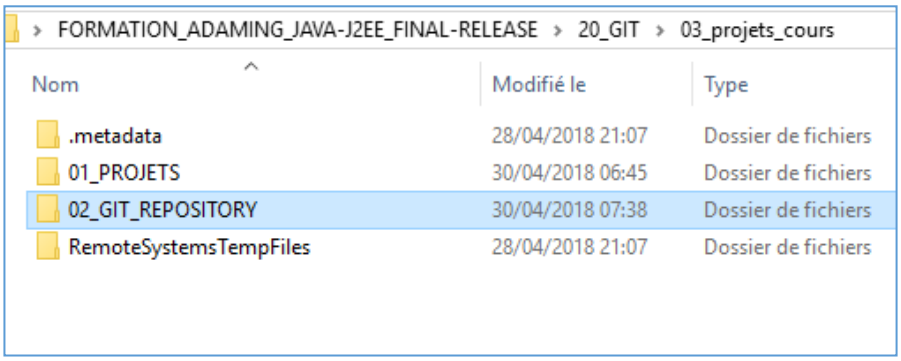




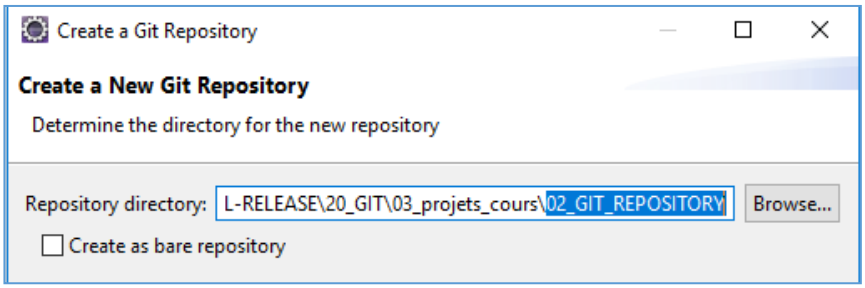
- L'emplacement par défaut est = **`C:\Users\INTI0261\git`**

- On peut spécifier notre propre emplacement avec **`Browse`**:

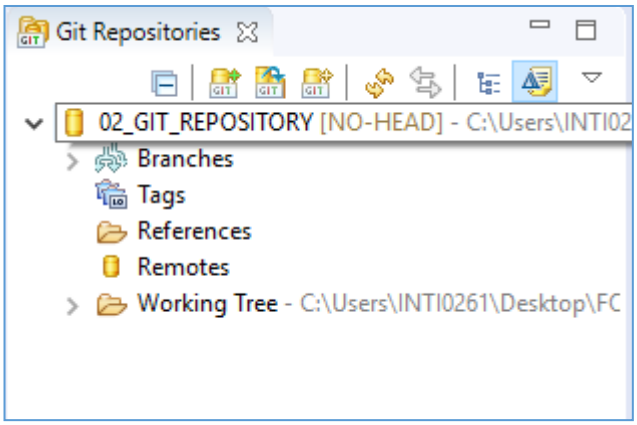
  - On crée le dossier pour le dépôt :



  - On le pointe dans Eclipse :

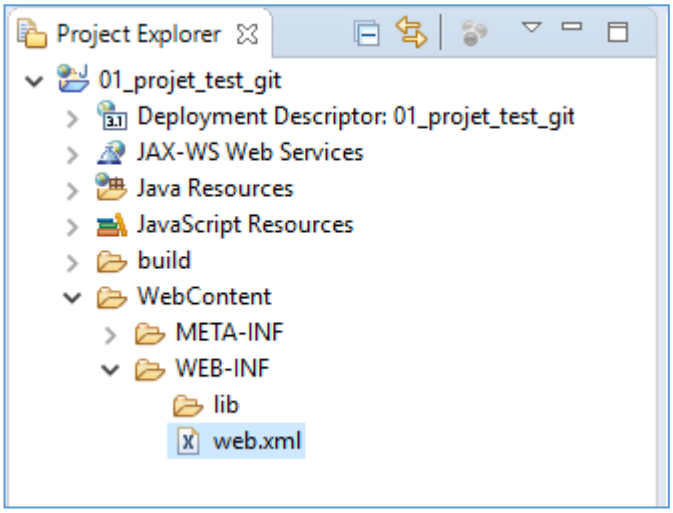


- Le dépôt est affiché dans Eclipse :

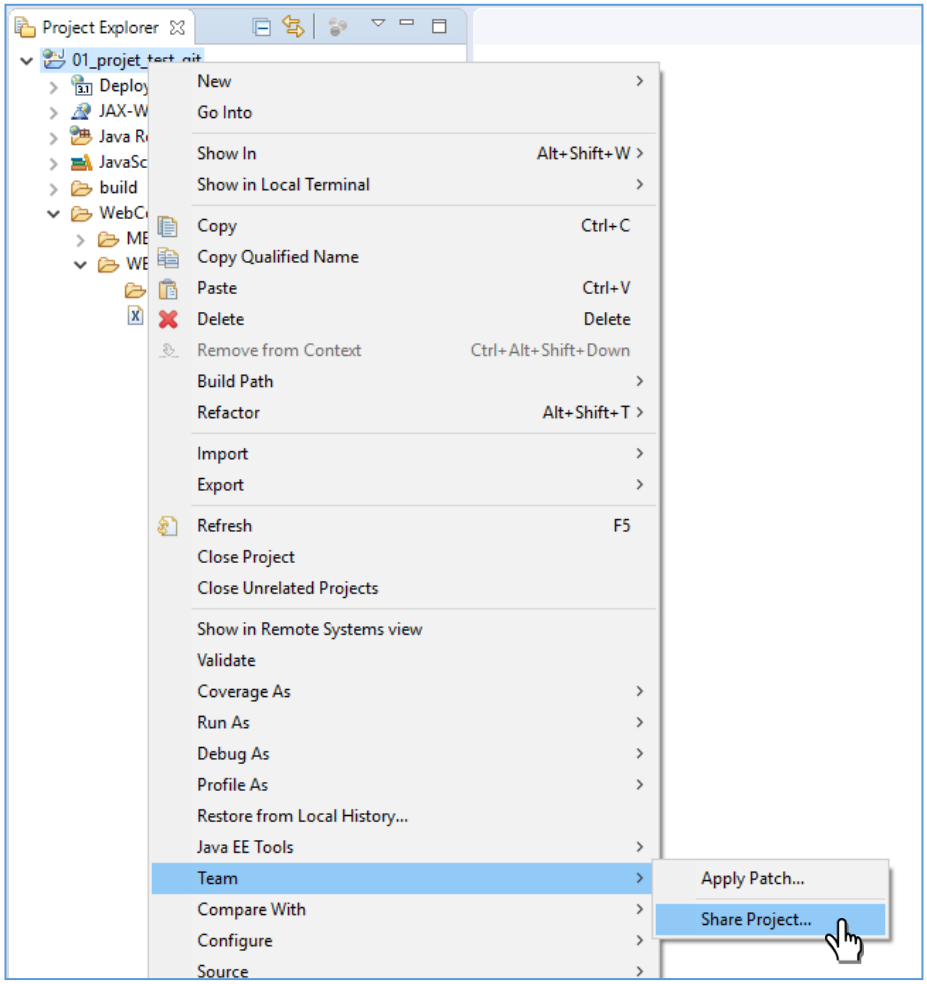


4. Création d'un projet versionné par Git

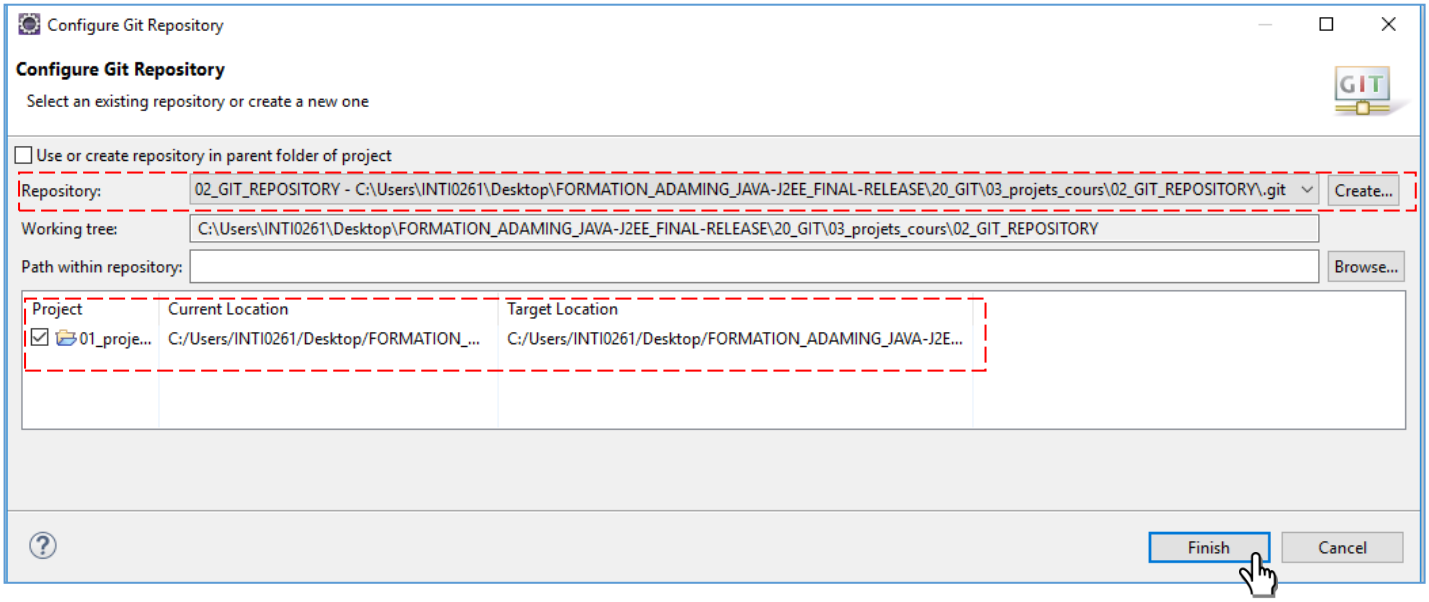
- On va ajouter une application web : **01_projet_test_git**



- Maintenant, On va demander à Git de surveiller notre projet :
  - Pour ça, on va demander à Git d'indexer le projet et l'ajouter au dépôt local :
    - On fait un click droit sur le projet :

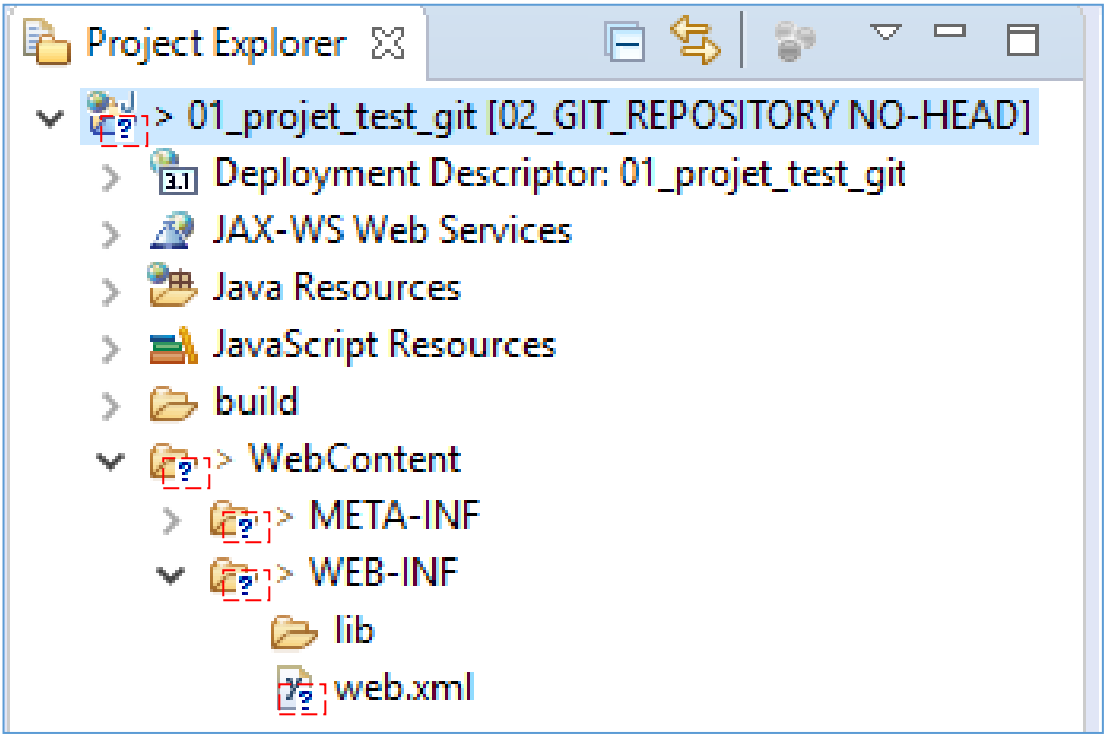


- Dans la liste déroulante **`Repository`**, on pointe le dépôt qu'on a crée :



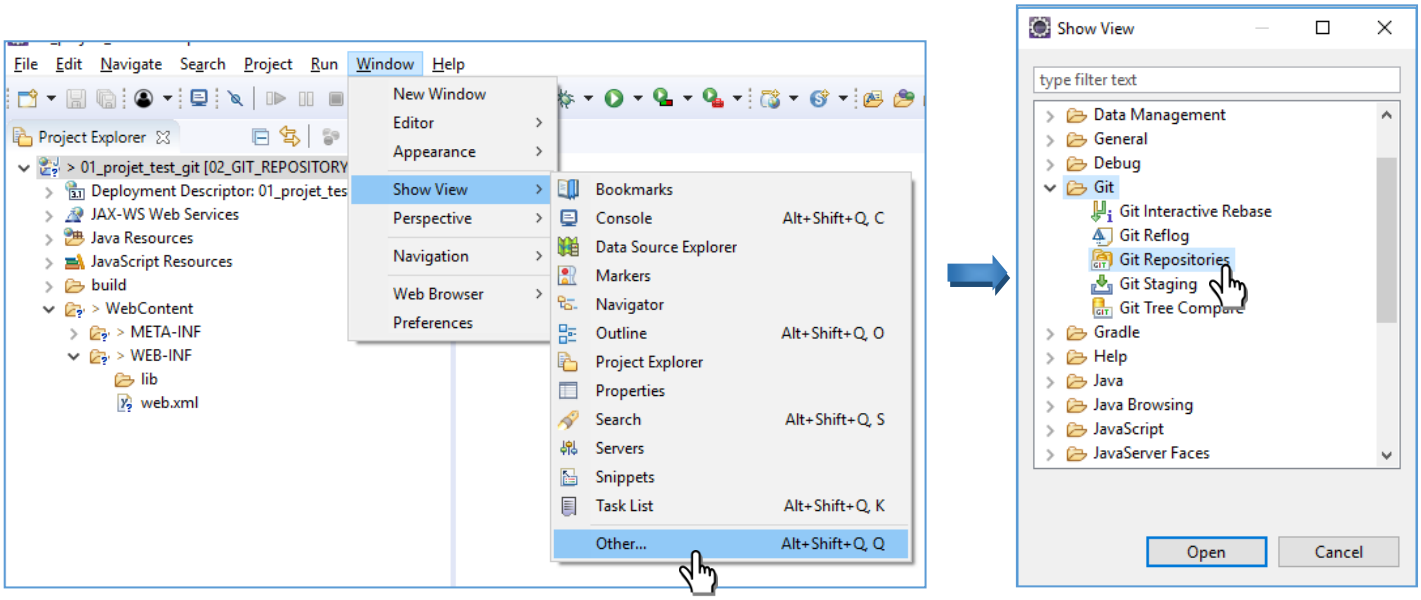
- L'icone [**?**] est ajouté sur le nom du projet indiquant que notre projet est indexé par Git :

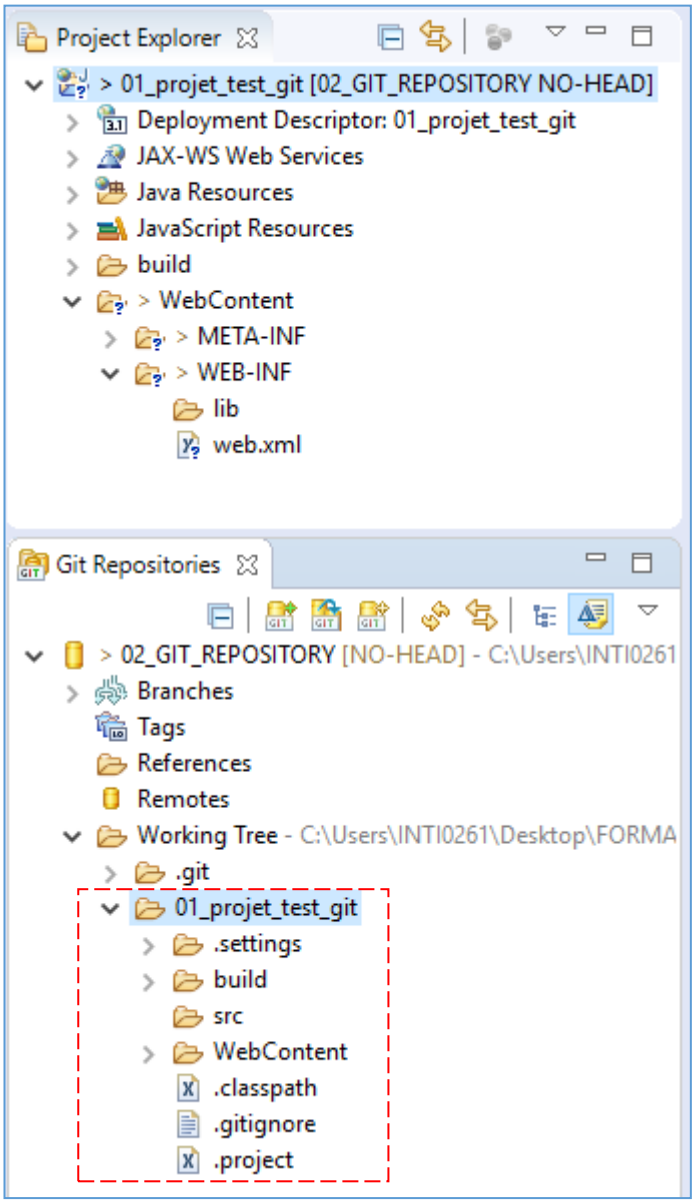
untracked (non pisté- non surveillé) : chaque fichier est identifié par Git mais pas encore pris en charge.



- ✘ Le projet est dorénavant géré par le système de contrôle de versions Git, mais les fichiers existants ne sont pas encore considérés comme versionnés : ils sont décorés avec un point d'interrogation.
- ✘ Le projet est maintenant initialisé, git va commencer à surveiller le système de fichiers et va être à l'écoute des différentes commandes qu'on va lui donner.
- ✘ Notre dépôt est prêt à recevoir les codes source.

- On peut ajouter la vue Git Repositories à la perspective pour afficher le dépôt local :

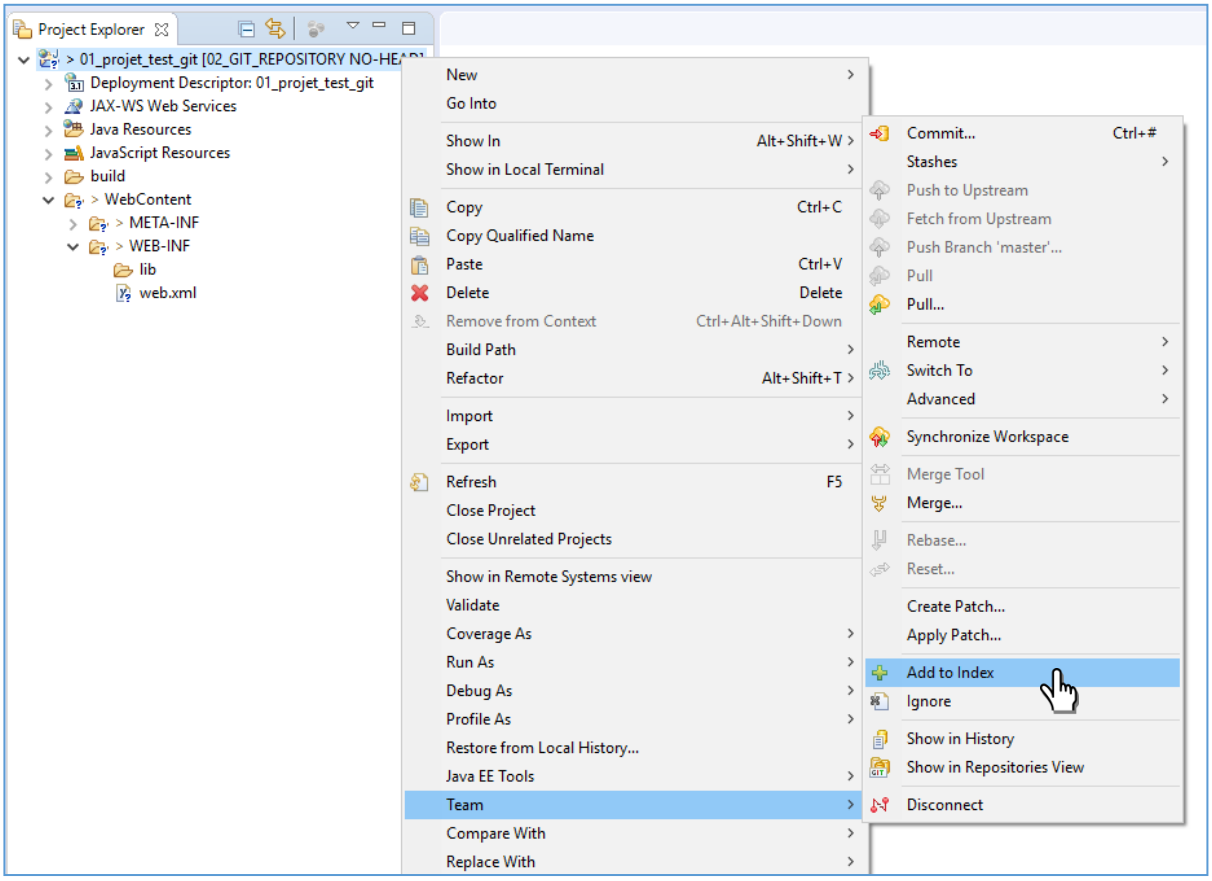


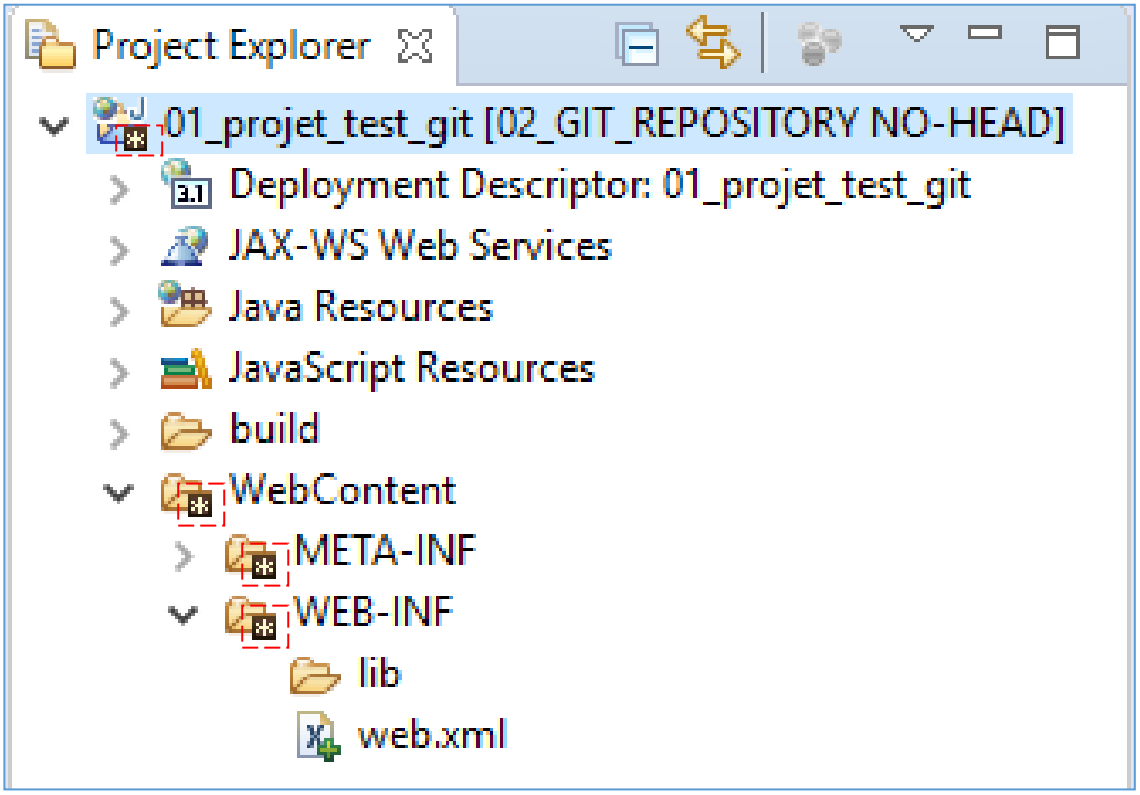


Tracker les fichiers du projet

- Pour que Git puisse tracker le projet ainsi que ses fichiers et les considérés comme versionnés, on :

```
click droit sur le projet  => Team => Add to Index
```



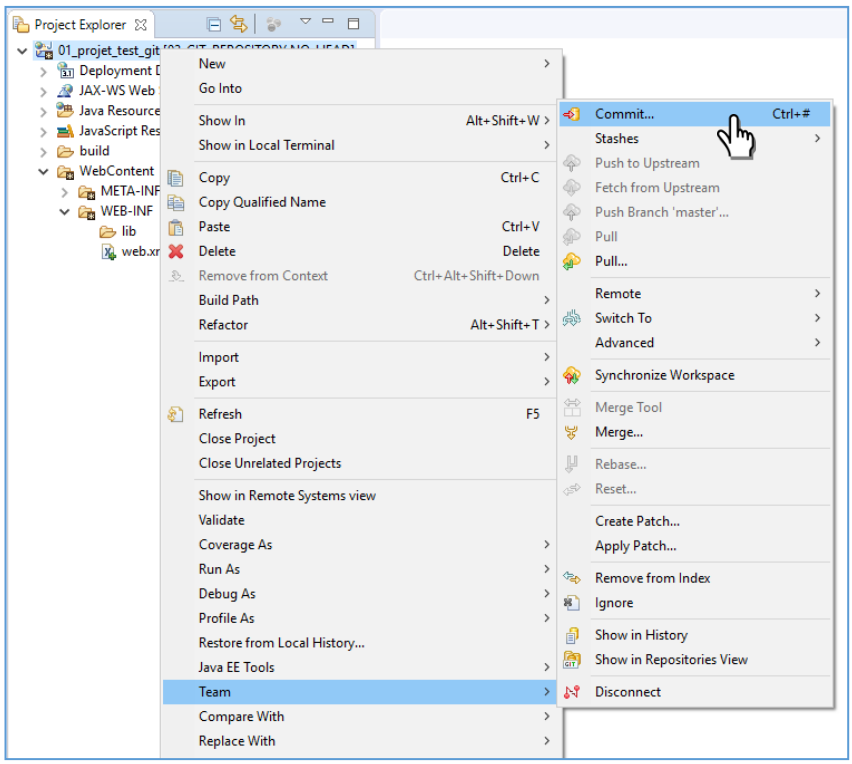


- Les dossiers sont maintenant marqués par un **astérisque** sur fond noir : ils sont détectés comme faisant partie d'une modification par Git.
- Les fichiers sont marqués par un signe plus [+]

Commiter les changements

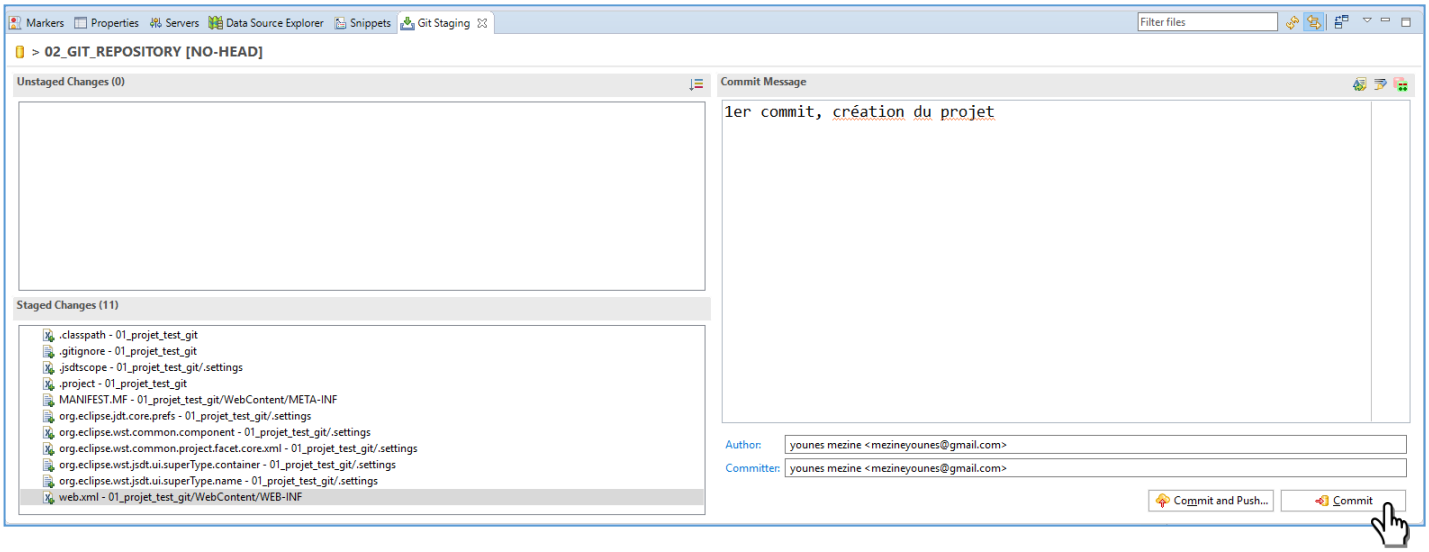
- Pour soumettre les modifications du projet dans le dépôt local, on doit effectuer un **`commit`** :

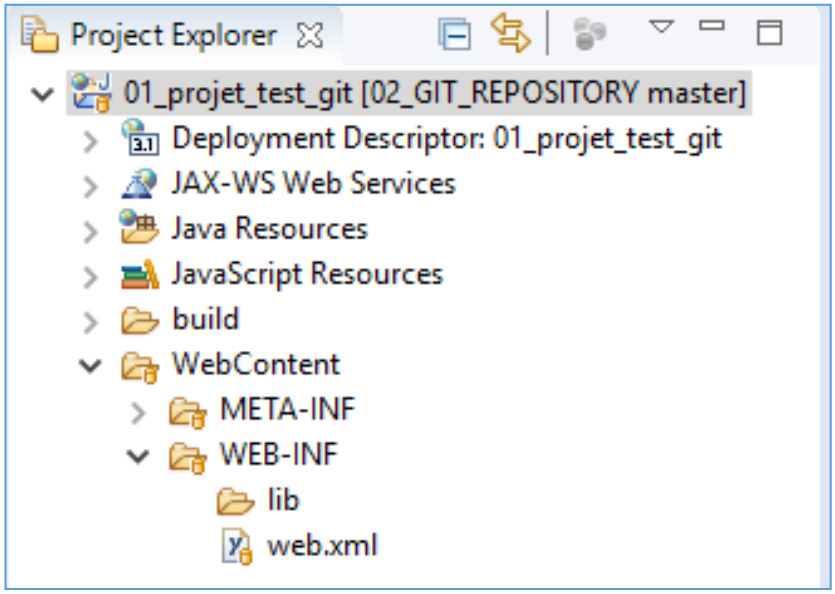
  - Pour ça, on fait : `click droit sur le projet => Team => commit`



  - La vue **`Git Staging`** s'affiche pour pouvoir effectuer le **`commit`** :

    - On ajoute un message pour la soumission des modifications (le message est obligatoire dans le système Git), et on clique sur **`Commit`**.

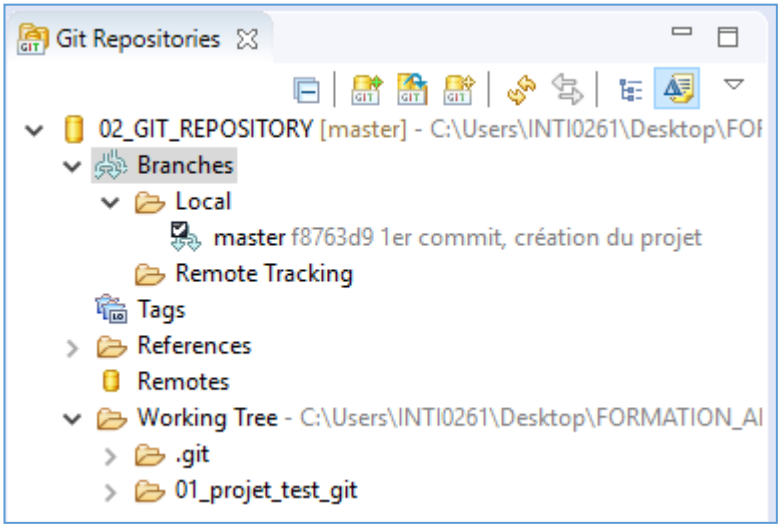




✘ Les fichiers ont été versionnés (commités).

✘ ils sont maintenant décorés avec un petit cylindre jaune.

- Dans la vue Git, on voit apparaitre sur la branche principale **`master`** **(**sur laquelle on est entrain de travailler**)** , le commit avec son hash (identifiant) et le message :



Ajout de fichiers source au projet

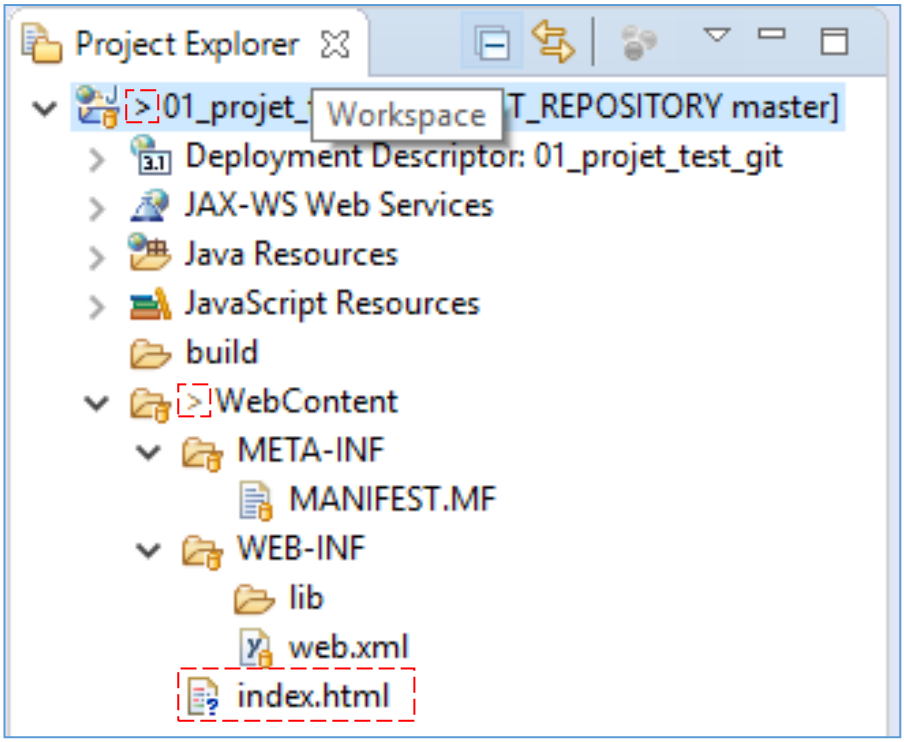
- On va commencer notre développement.
  - On va ajouter la page d'accueil **`index.html`** :

```
*index.html
1  <!DOCTYPE html>
2  <html>
3  <head>
4  <meta charset="ISO-8859-1">
5  <title>page d'accueil</title>
6  </head>
7  <body>
8      <h1>Page d'accueil</h1>
9  </body>
10 </html>
```

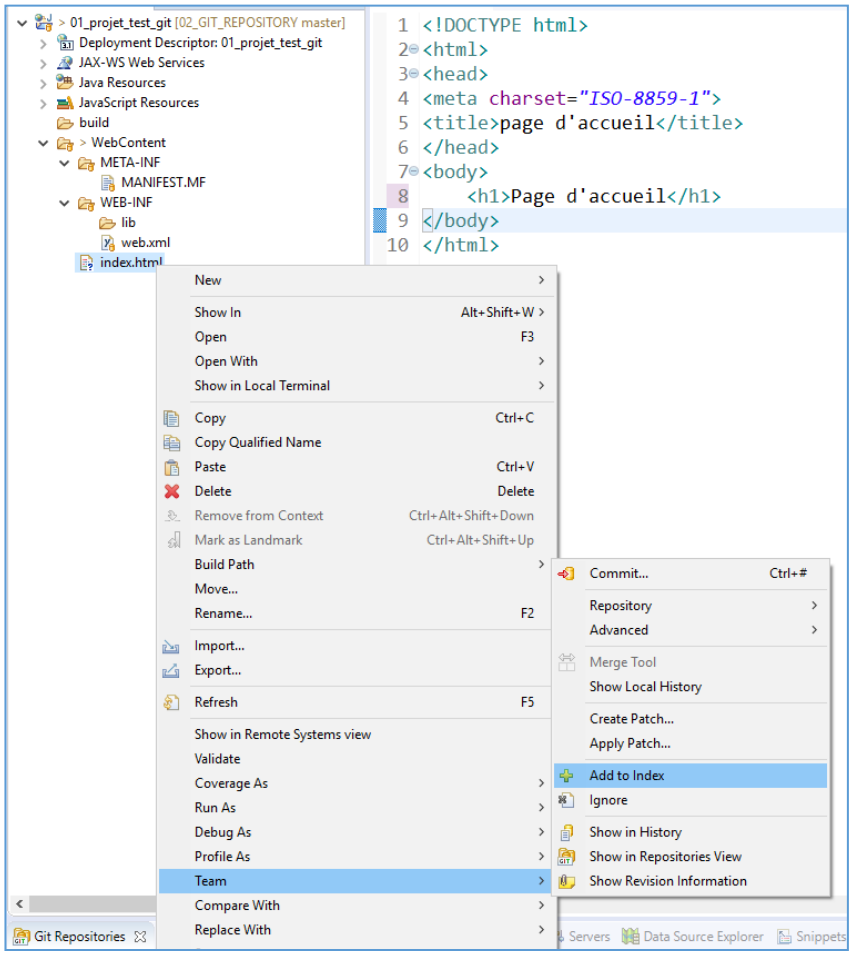
- A l'ajout de la page **`index.html`** :

- Le projet et Les dossiers modifiés (**`Web Content`**) sont marqués par le signe `chevron` **>** devant le nom.

- Le fichier index.html est marqué par le signe **?**



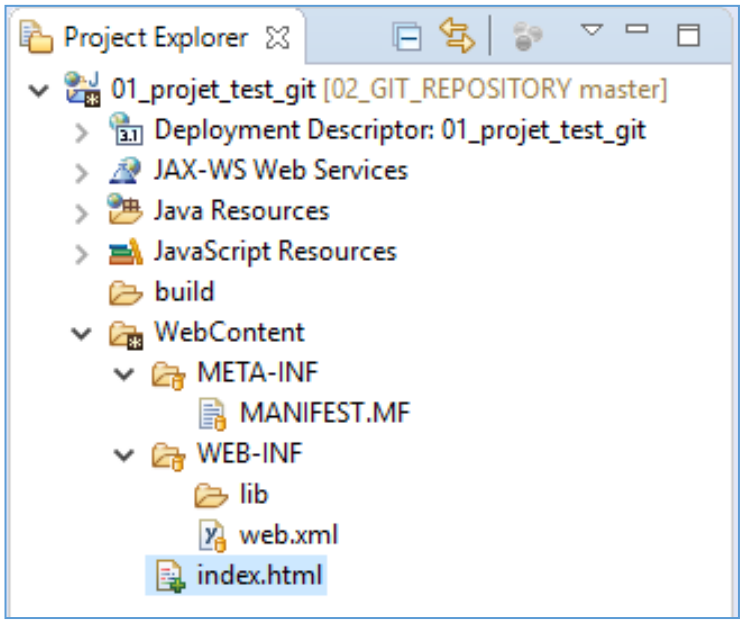
- On va maintenant ajouter la page à Git :



  - La page est marqué avec un [**+**] :

- Le projet et Les dossiers modifiés (**`Web Content`**) sont marqués par un **`astérisque`**.

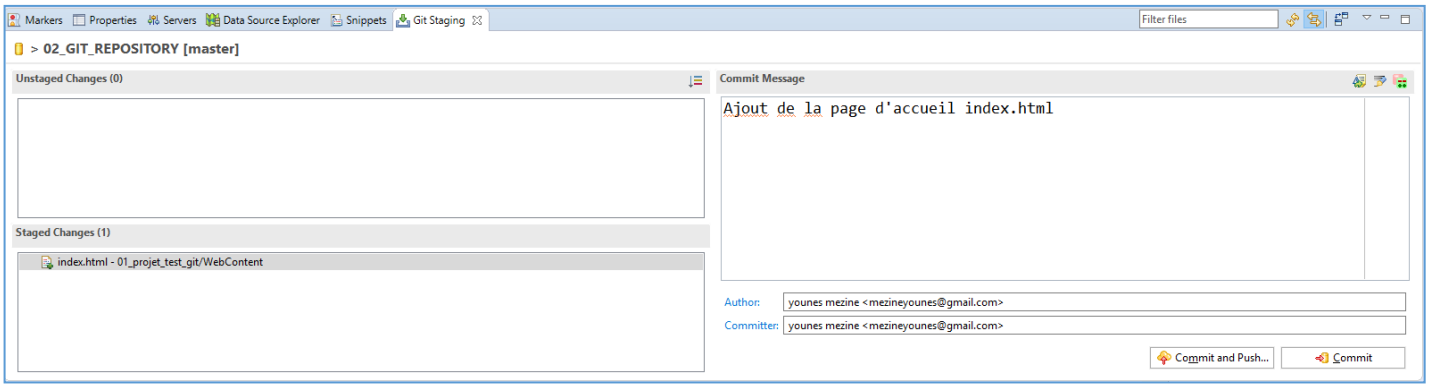
- La page est marqué avec un [**+**]



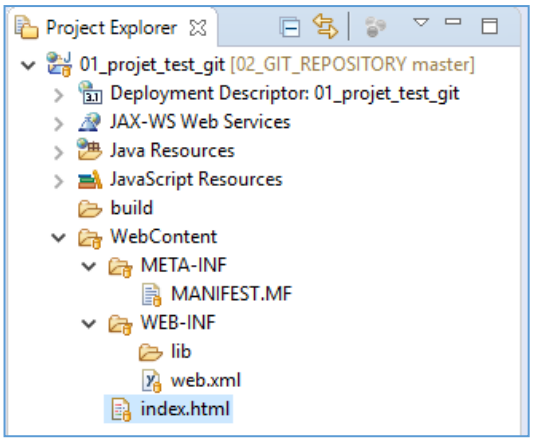
- On est prêt à commiter nos modifications à savoir la page **`index.html`** :

  - On fait un : `click droit sur le projet => Team => commit`

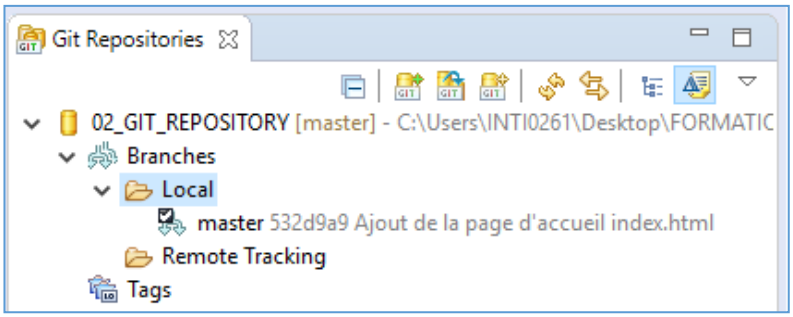
  - On ajoute un message pour la soumission des modifications (le message est obligatoire dans le système Git), et on clique sur **`Commit`**.



- Le fichier est versionné (commité).
- Il est maintenant décoré avec un petit cylindre jaune.



  - Dans la vue Git, on voit apparaitre sur la branche principale **`master`** **(**sur laquelle on est entrain de travailler**)** , le commit avec son hash (identifiant) et le message :



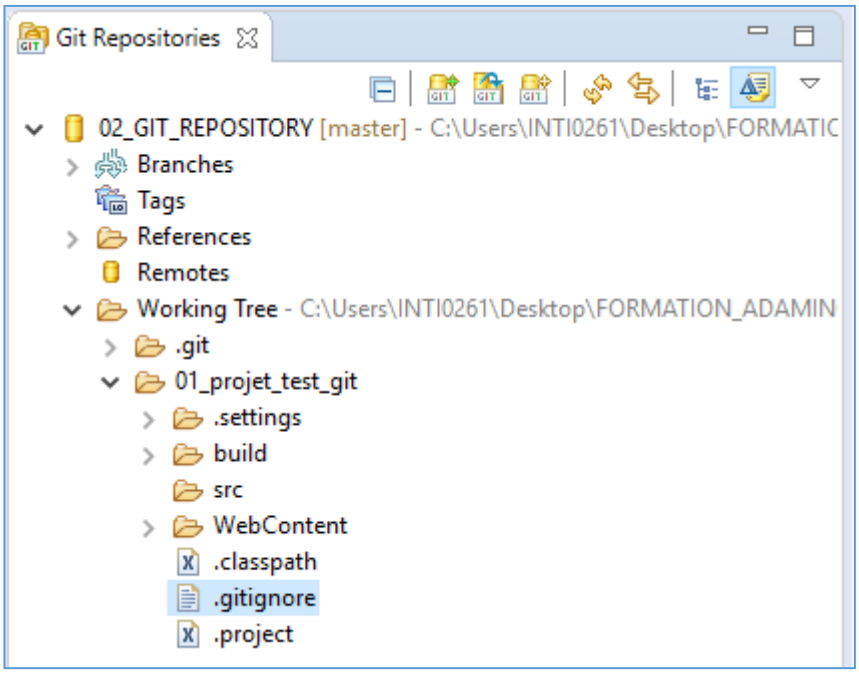
Ignorer les fichiers

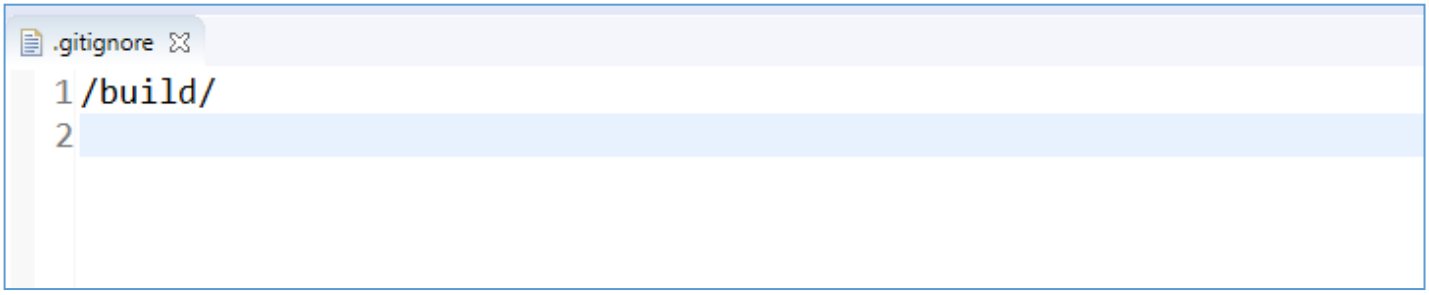
- Il peut être très utile dans un projet d'indiquer à Git d'ignorer systématiquement certains fichiers.

- Ça peut être des fichiers de log, des fichiers de configuration par ce que ça peut être dangereux de laisser une configuration à titre d'exemple des mots de passe à la BDD.

- Git nous met à disposition un système de gitignore qui va nous permettre d'ignorer certains fichiers de sorte qu'il ne soit jamais indexés et jamais enregistrés par Git.

- Pour mettre en place ce mécanisme de gitignore, il est simplement nécessaire d'écrire à la racine du dépôt un fichier qui va s'appeler sombrement **.gitignore**

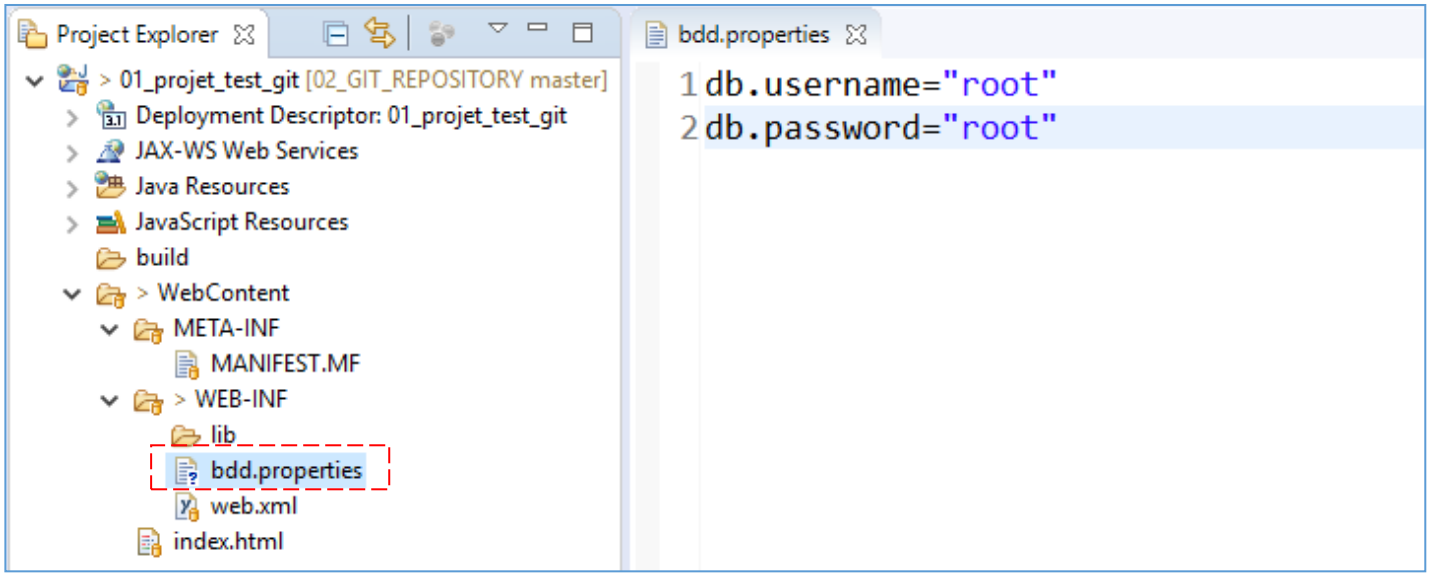




- Dans ce fichier, on va pouvoir déterminer quels sont les fichiers ignorés.

Exemple :

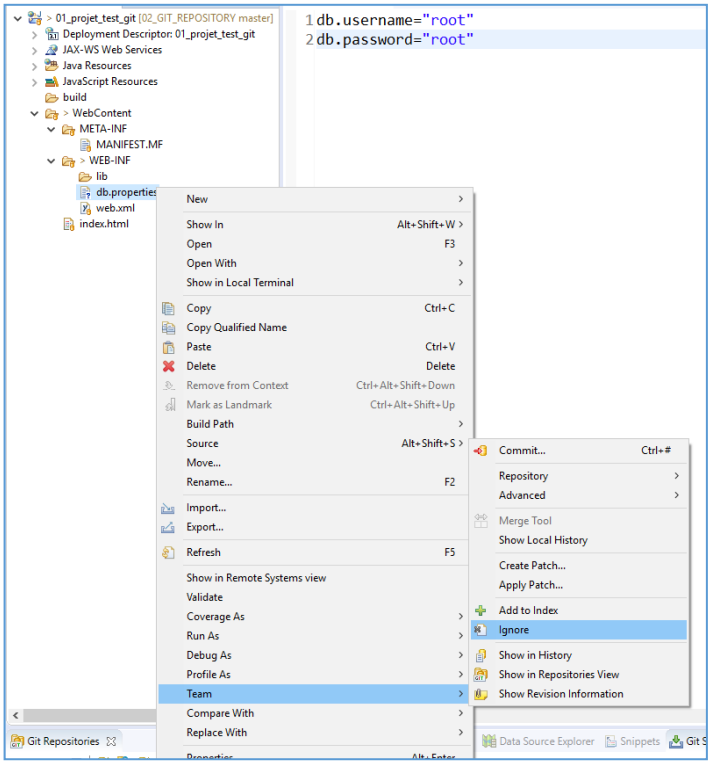
- On va ajouter un fichier de type properties **`bdd.properties`** contenant les informations pour se connecter a la BDD.

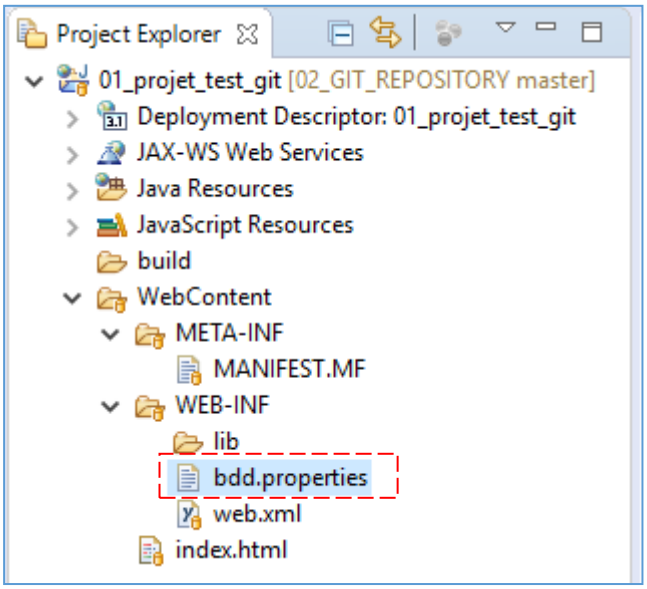


- Ce fichier est très sensible et ça peut être dangereux de laisser cette configuration à la portée de tous le monde.

- On décide d'indiquer à Git d'ignorer systématiquement ce fichier la.

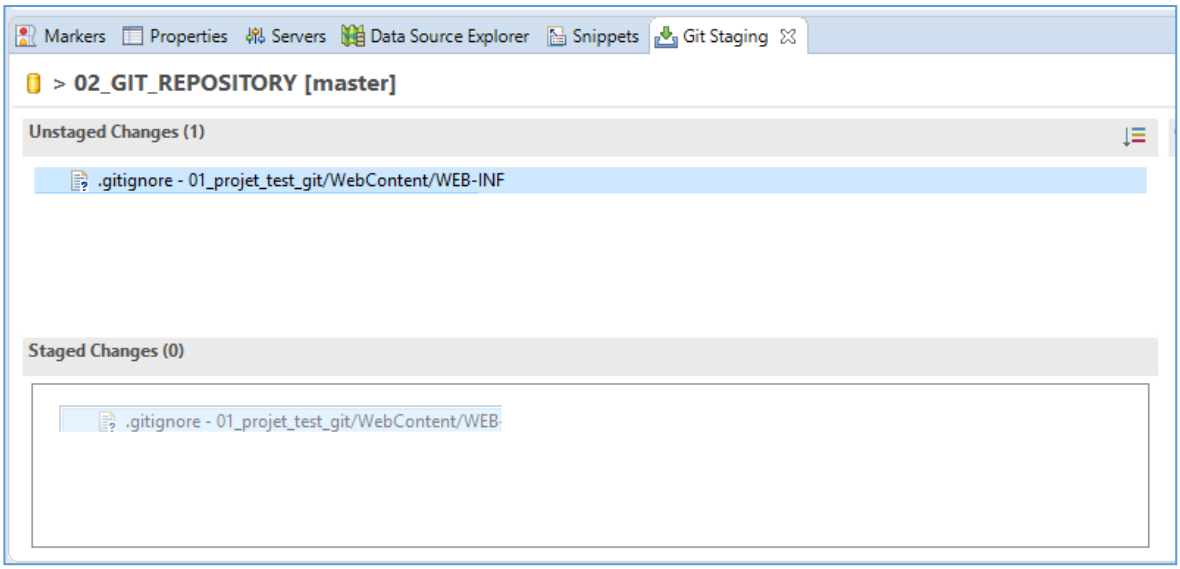
  - Pour se faire, on fait un : `click droit sur le projet => Team => ignore`



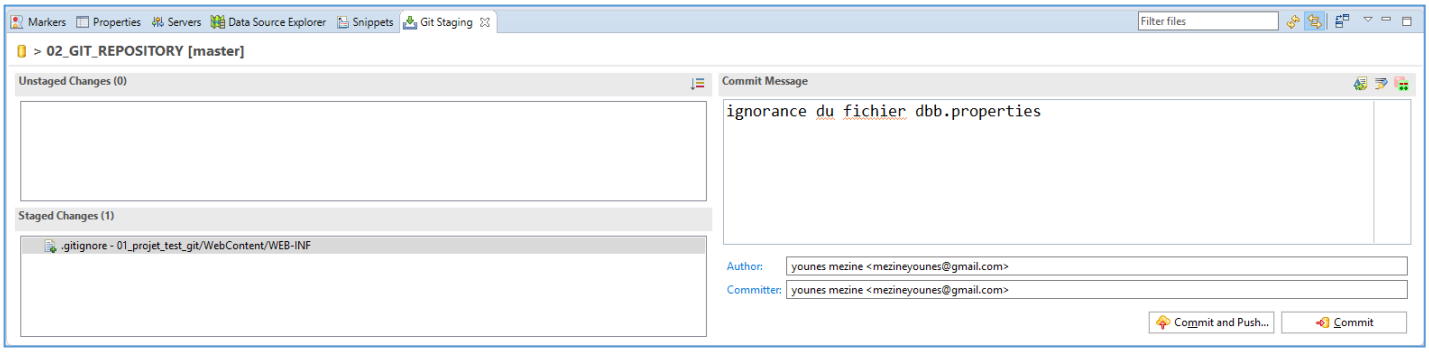


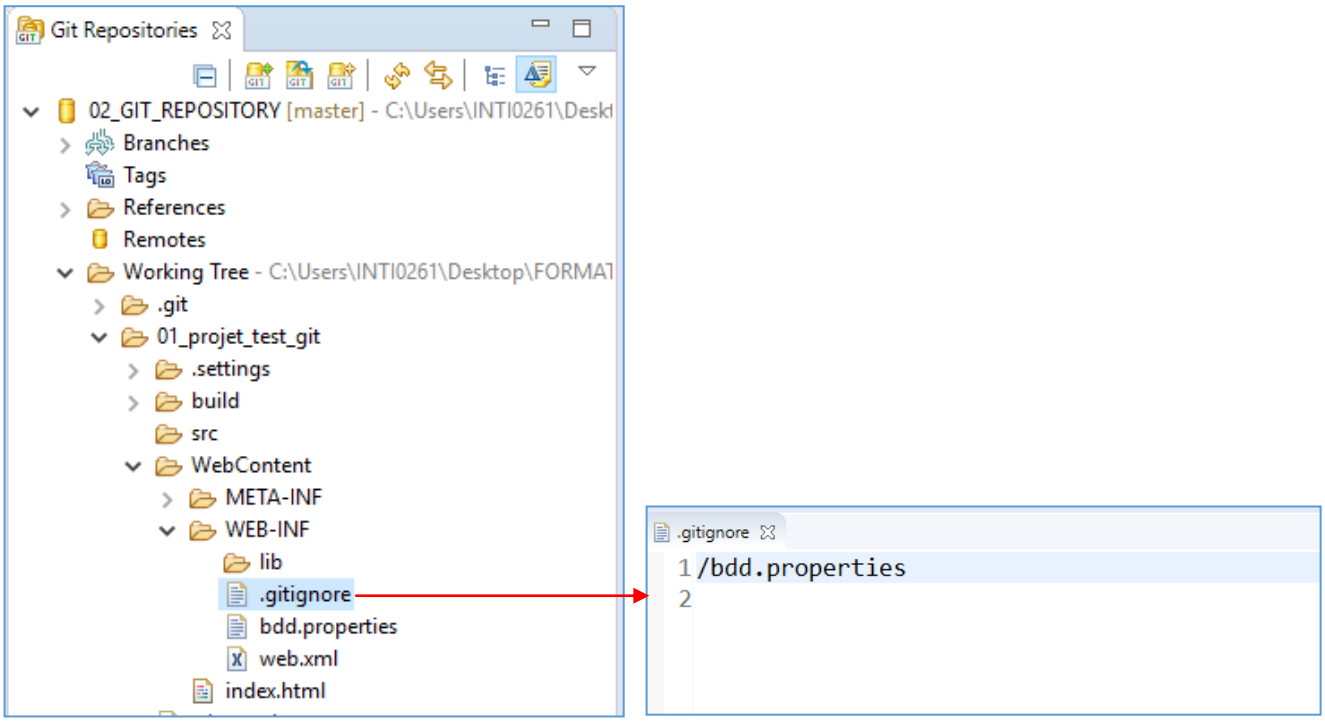
- Dans la vue **`Git Staging`**, Eclipse nous donne la main pour le **`commit`** :

  - On drag-and-drop le fichier **`.gitignore`** dans **`Staged Changes`** :



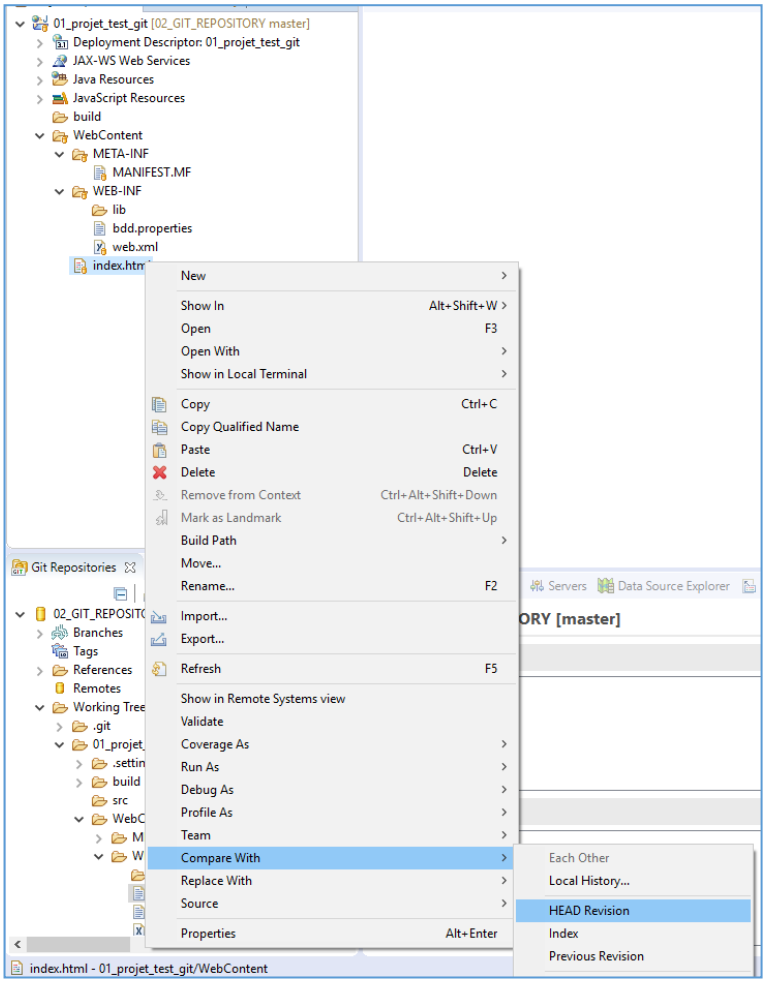
- On ajoute le message et on lance le commit :

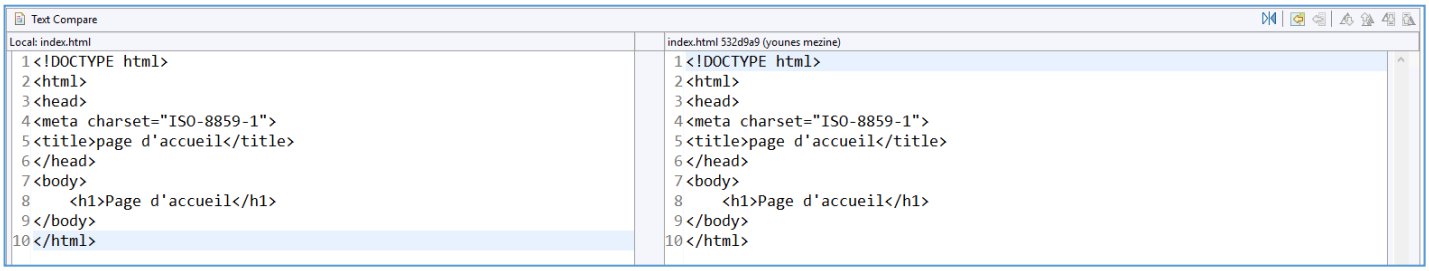




Annuler des changements

- Si on souhaite annuler les modifications, il existe deux options :
  - On peut comparer chaque fichier qu'on veut changer avec la révision `HEAD` (HEAD revision) et annuler tout ou une partie des changements effectués.

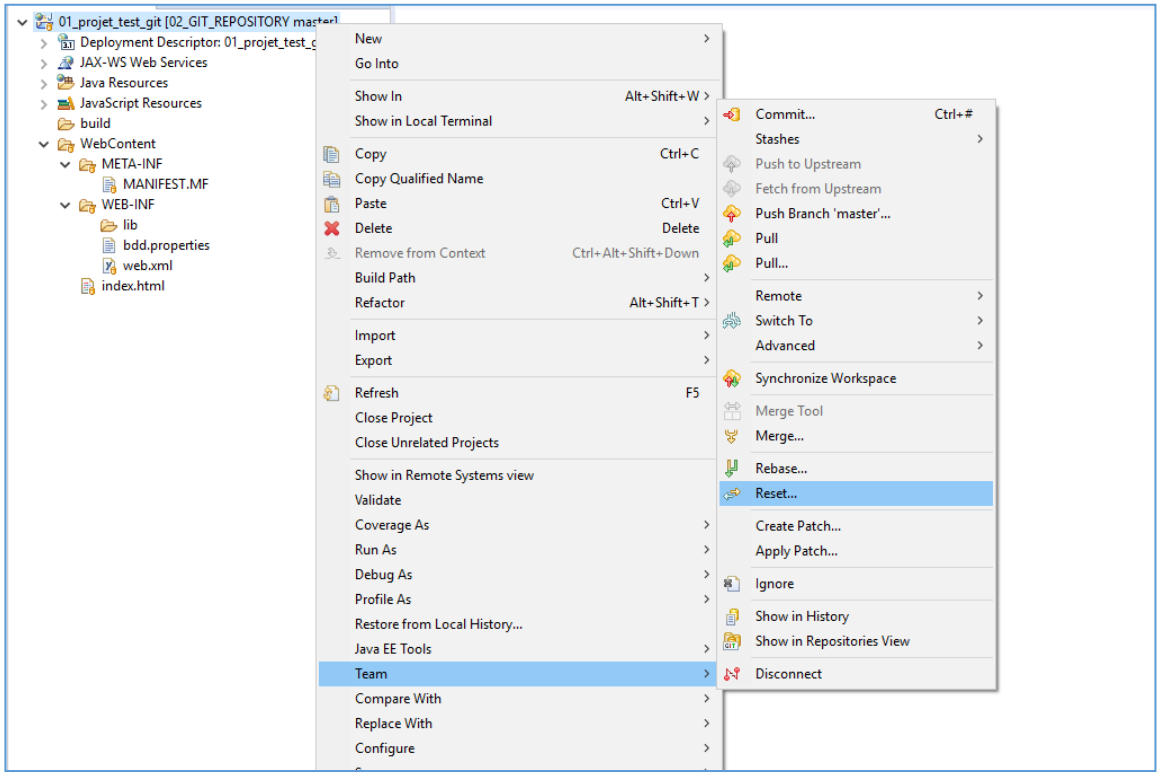




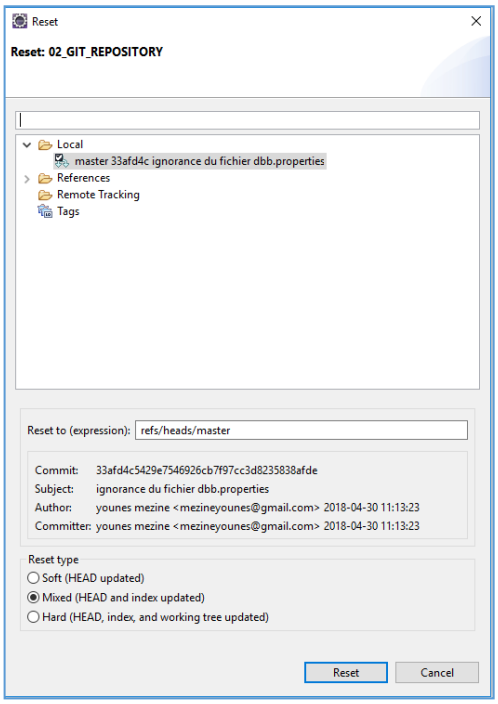
Right click the file you want to revert and select *Compare With => HEAD Revision*. This will open a comparison with the HEAD Revision, highlighting any changes done. If you want to completely revert your file, hit the *Copy All Non-Conflicting Changes from Right to Left*-button in the Java Source Compare toolbar. If you only want to revert several lines, select each line individually and hit the *Copy Current Change from Right to Left*button (in the toolbar) for each line. To complete the Revert operation, you will have to save either the comparison or your local copy of the file.

- Deuxièmement, vous pouvez réinitialiser votre projet, ce qui entraînera la restauration de tout changement dans le répertoire de travail.

```
click droit sur le projet => Team=> reset
```



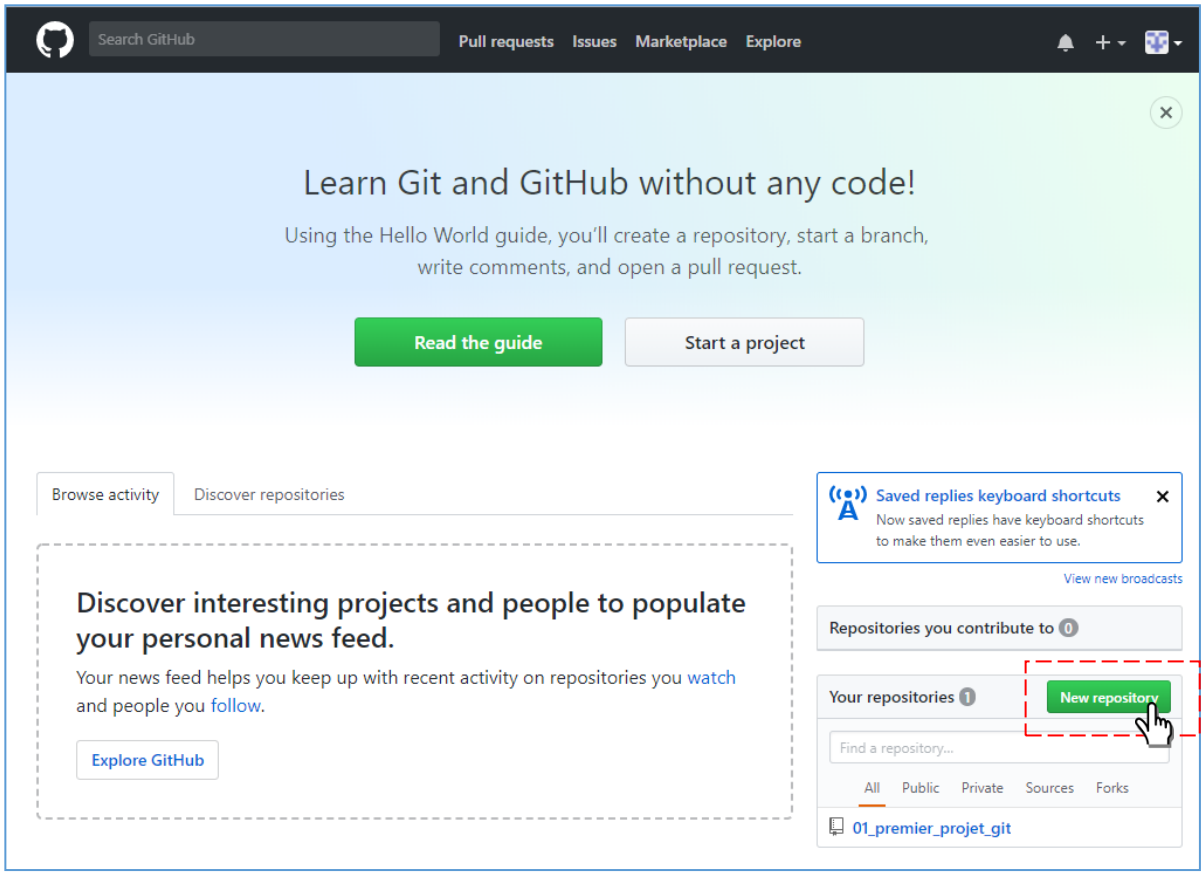
To reset all changes made to your project, right click the project node and navigate to*Team => Reset*…. Select the branch you want to reset to (if you haven't created any other branches, there will be just one) and choose *Hard* as a reset type. By confirming this operation, all changes will be reset to this branch's last commit, including all changes done in the workspace (and index, more on that in the section "index"). Be careful with this option as all changes in your workspace will be lost.

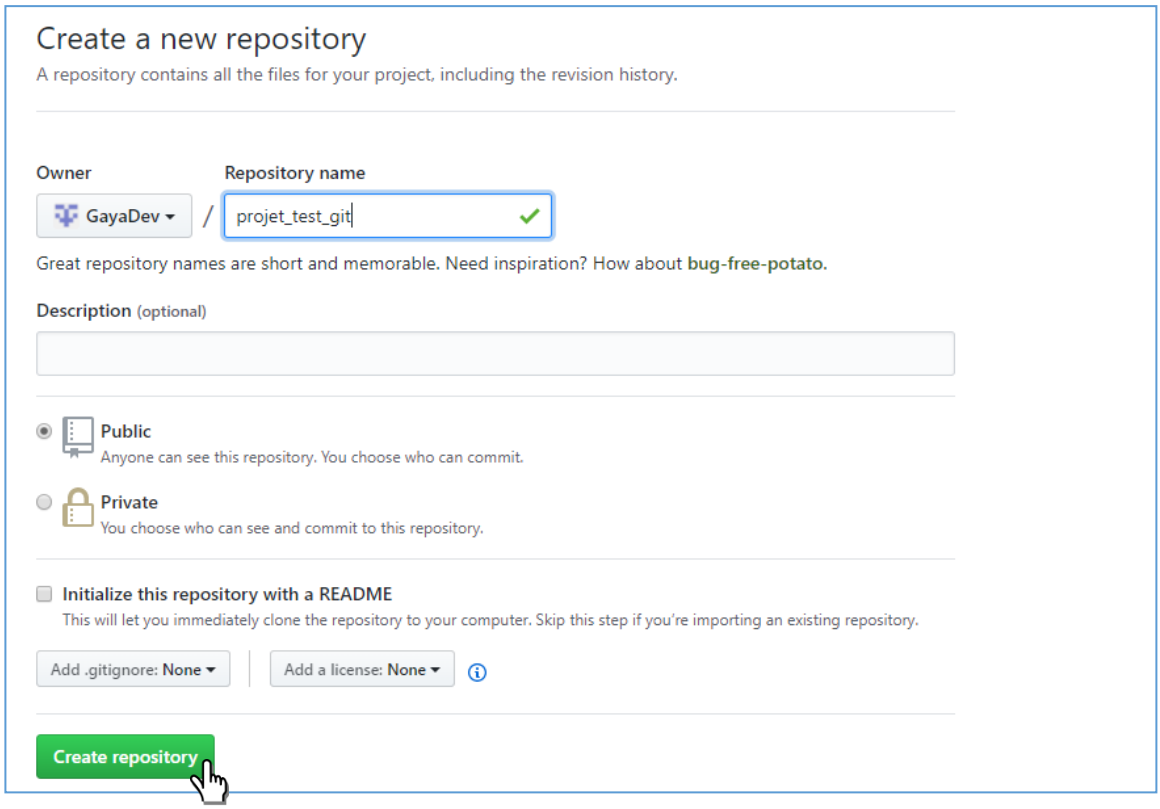


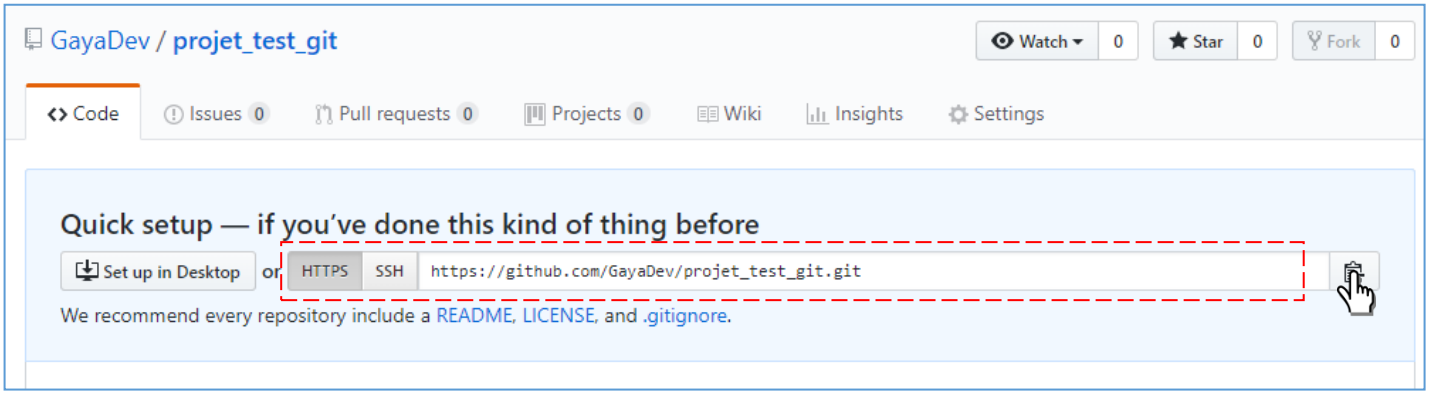
5.Pousser un projet vers le dépôt distant GitHub

- Maintenant on va voire comment pousser les modifications faites dans le dépôt local vers le dépôt distant GitHub.

  - On crée un dépôt sur GitHub :

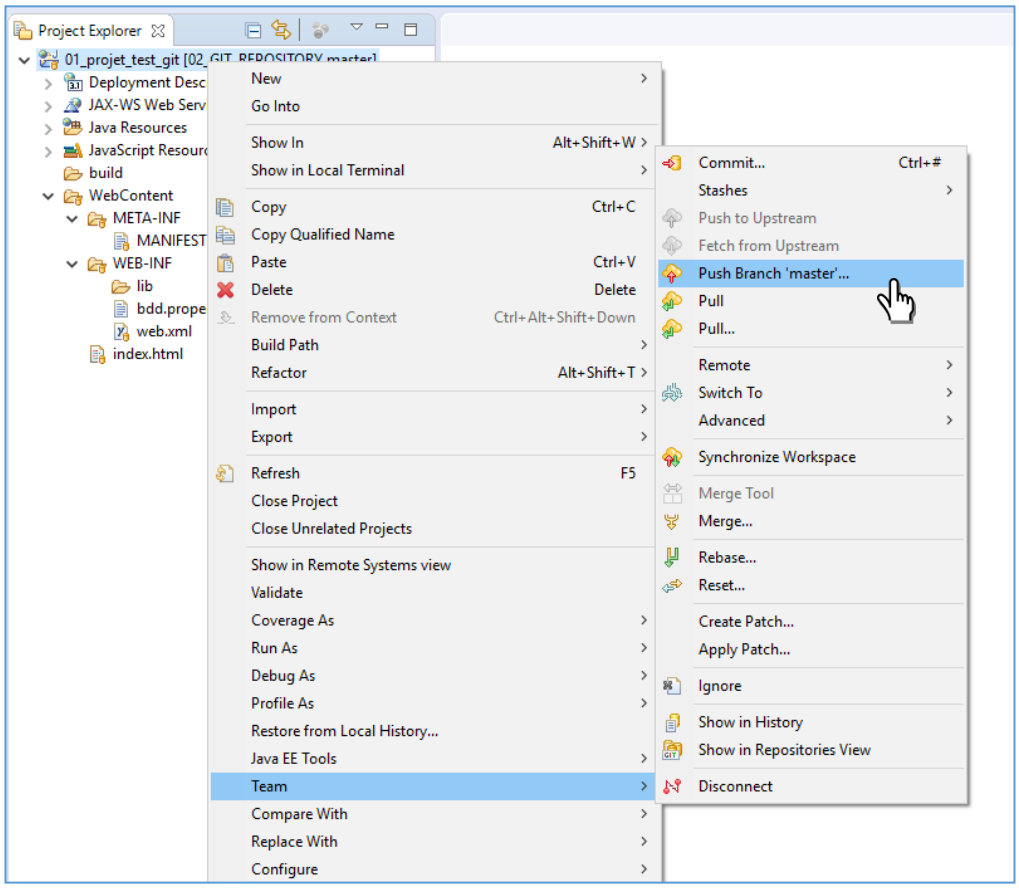




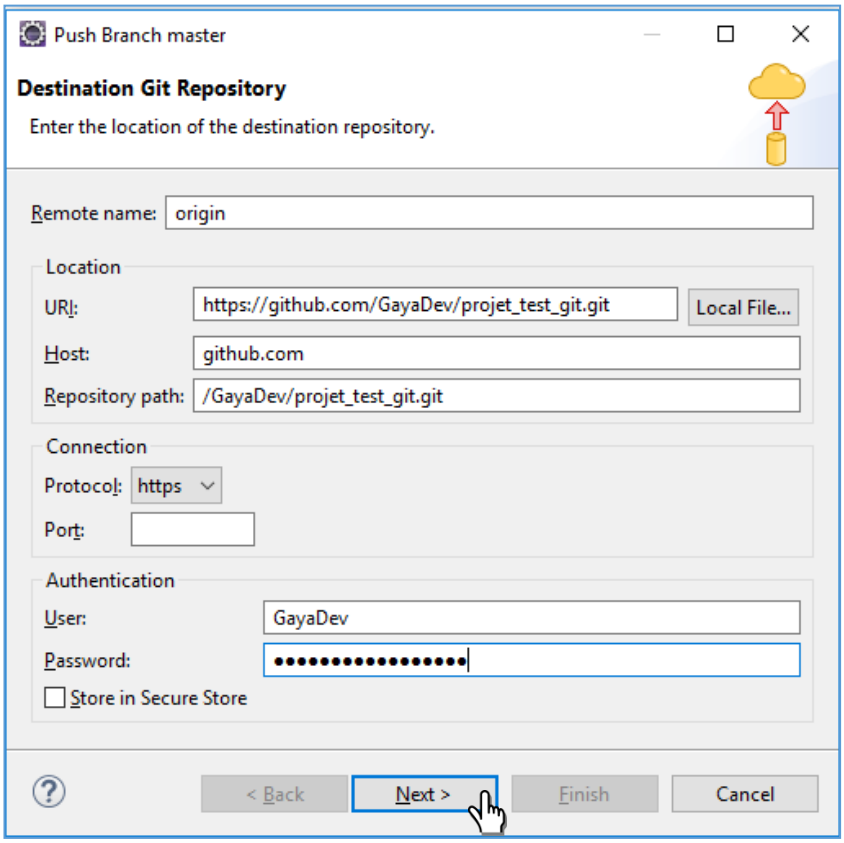


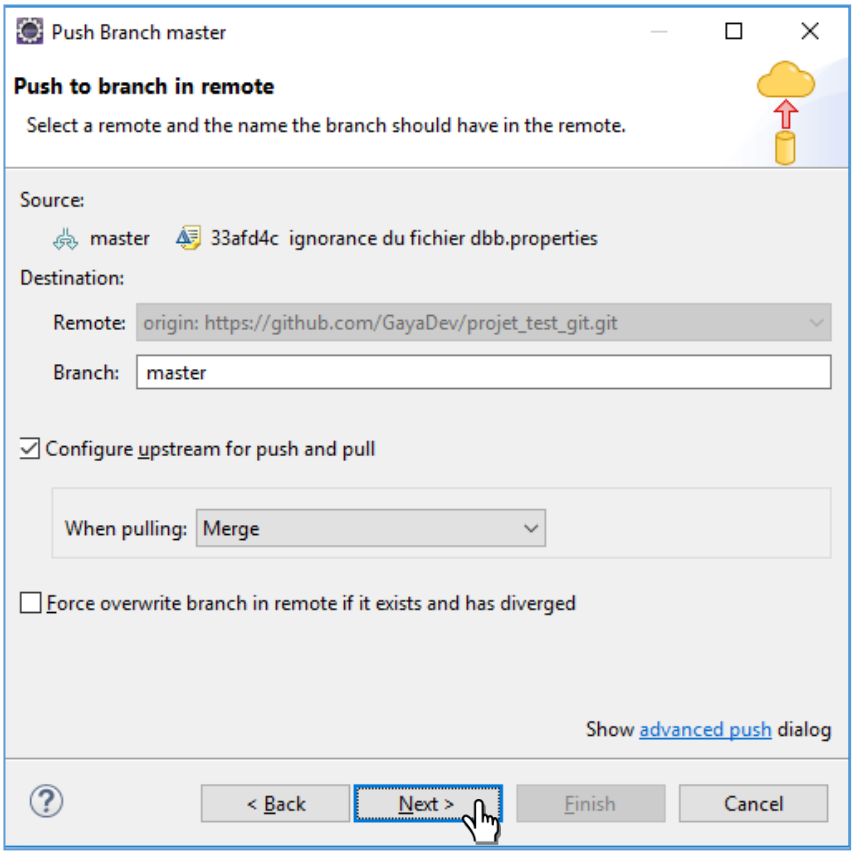
- On récupère l'URL de notre dépôt distant GitHub :

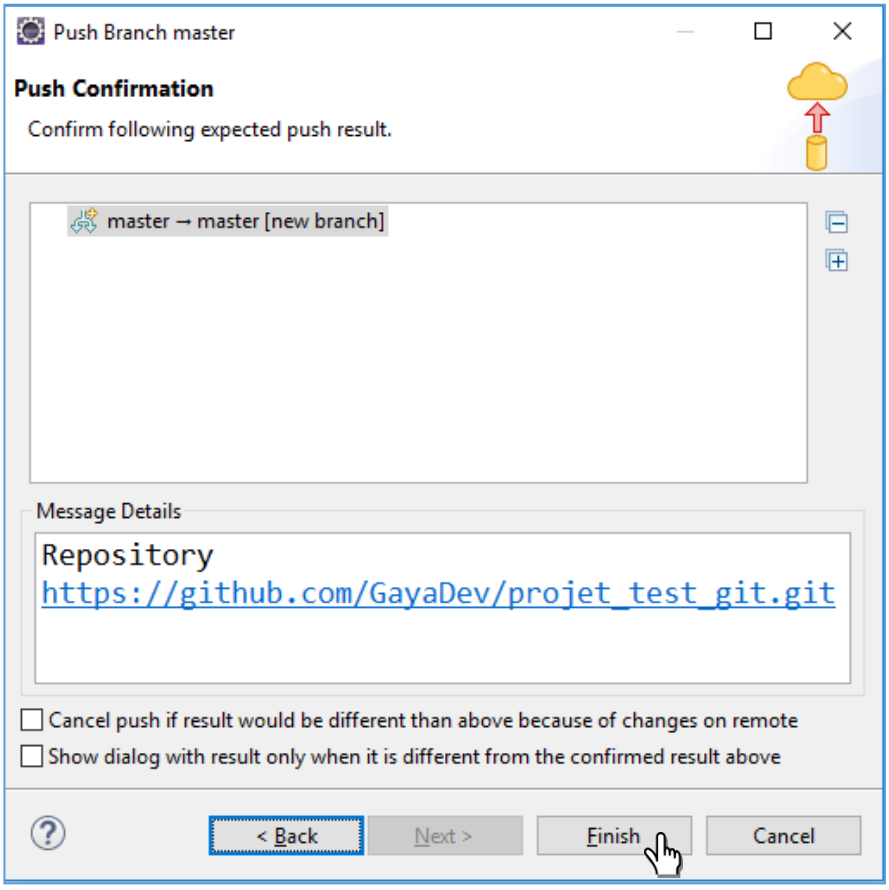
- On revient sur Eclipse :
  - On fait : `click droit sur le projet => Team`
    `=> Push  Branch master`



  - Dans **URL**, on saisi l'URL du dépôt GitHub puis le **User** et **Password** du compte de GitHub:

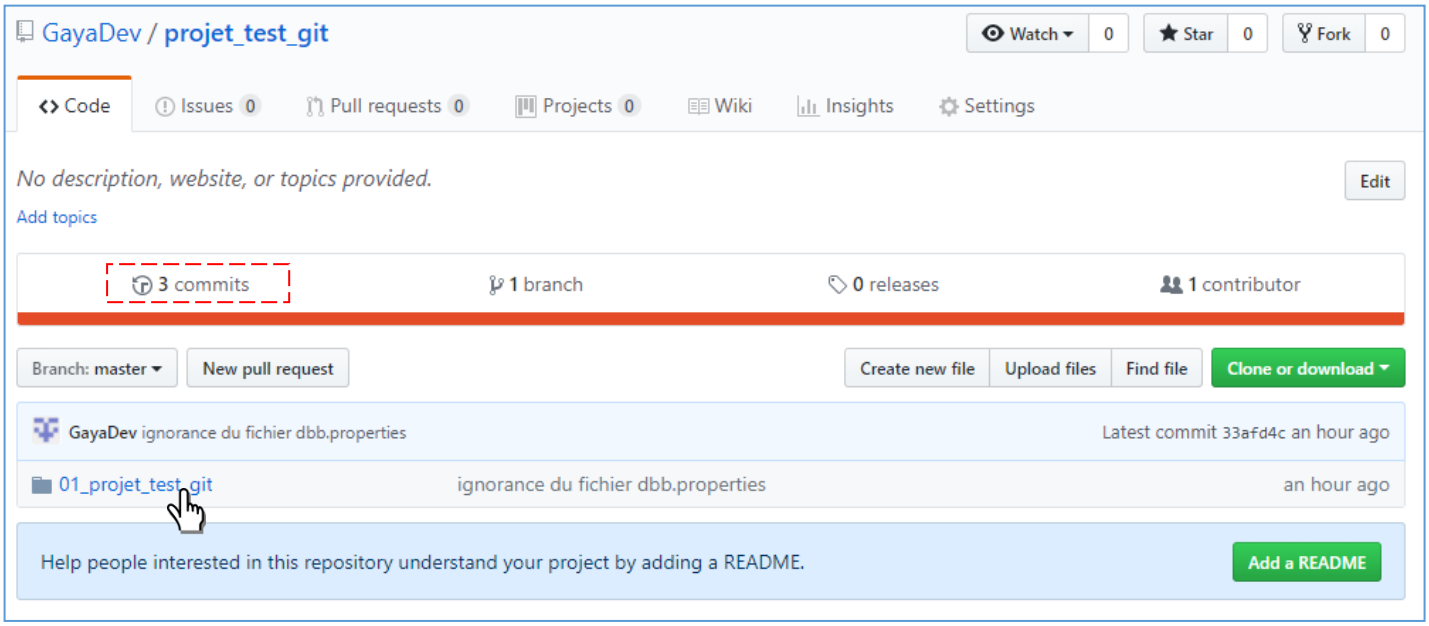

Push Branch master
Destination Git Repository
Enter the location of the destination repository.
Remote name: origin
Location
URI: https://github.com/GayaDev/projet_test_git.git
Local File...
Host: github.com
Repository path: /GayaDev/projet_test_git.git
Connection
Protocol: https
Port:
Authentication
User: GayaDev
Password:
Store in Secure Store
< Back
Next >
Finish
Cancel

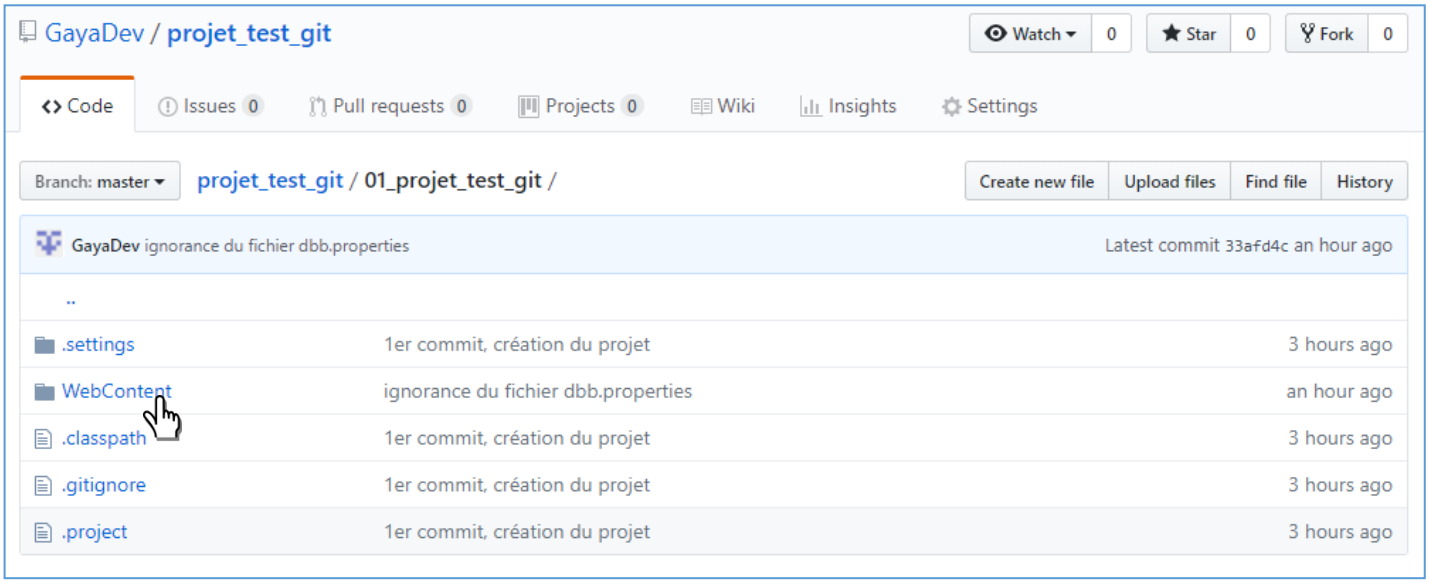

Push Branch master
Push to branch in remote
Select a remote and the name the branch should have in the remote.
Source:
master
33afd4c ignorance du fichier dbb.properties
Destination:
Remote: origin: https://github.com/GayaDev/projet_test_git.git
Branch: master
Configure upstream for push and pull
When pulling: Merge
Force overwrite branch in remote if it exists and has diverged
Show advanced push dialog
< Back
Next >
Finish
Cancel

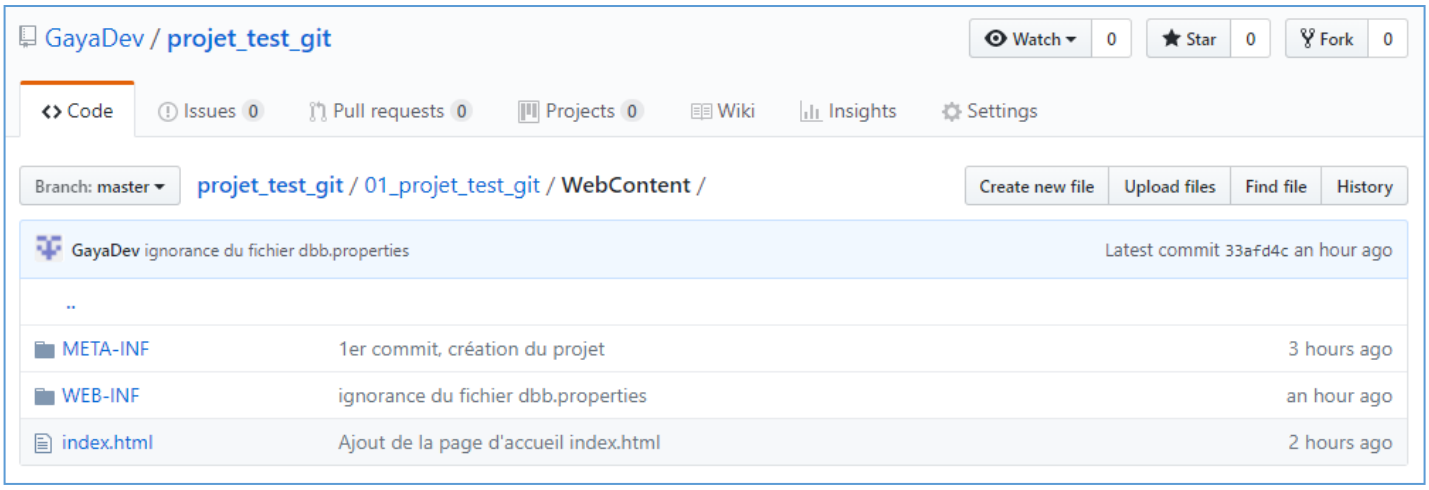

Push Branch master
Push Confirmation
Confirm following expected push result.
master → master [new branch]
Message Details
Repository
https://github.com/GayaDev/projet_test_git.git
Cancel push if result would be different than above because of changes on remote
Show dialog with result only when it is different from the confirmed result above
< Back
Next >
Finish
Cancel

Login
Repository
https://github.com/GayaDev/projet_test_git.git
User
GayaDev
Password
Store in Secure Store
OK
Cancel

- Si on regarde sur GitHub, on a bien notre projet avec les différentes commits:

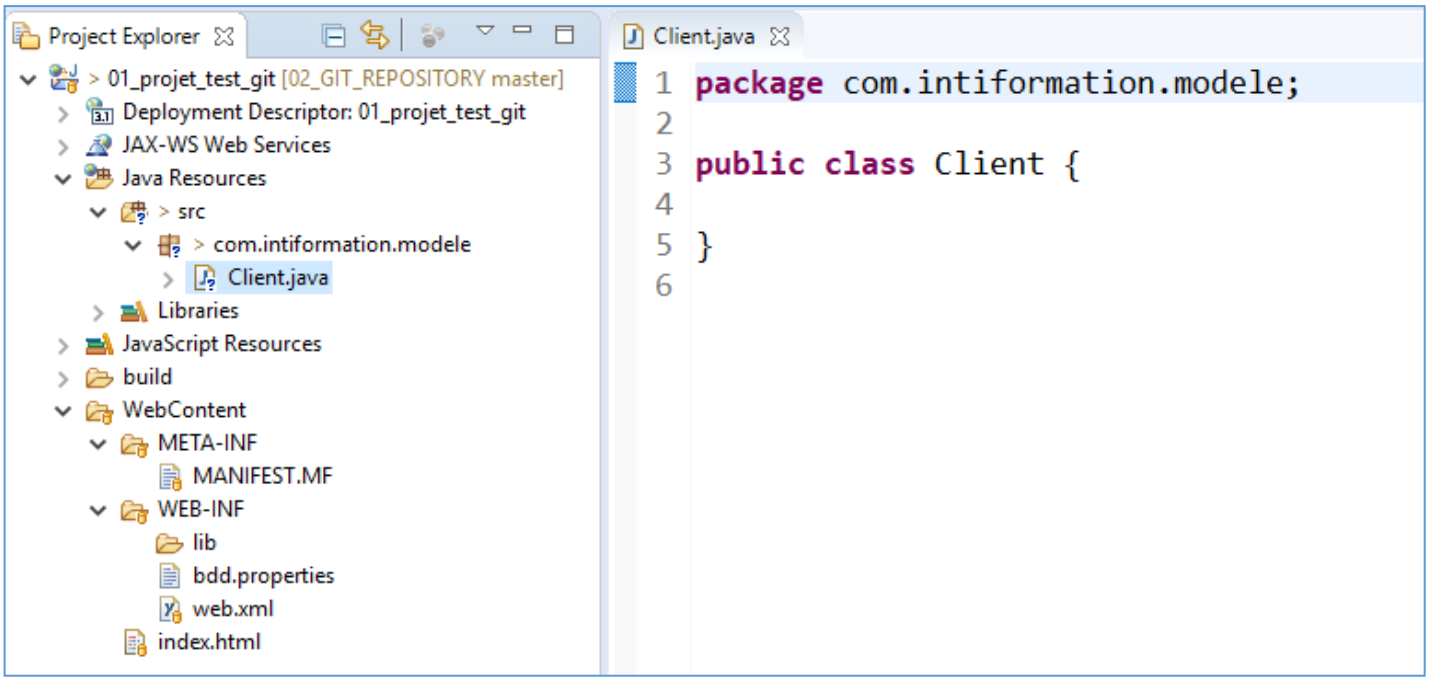




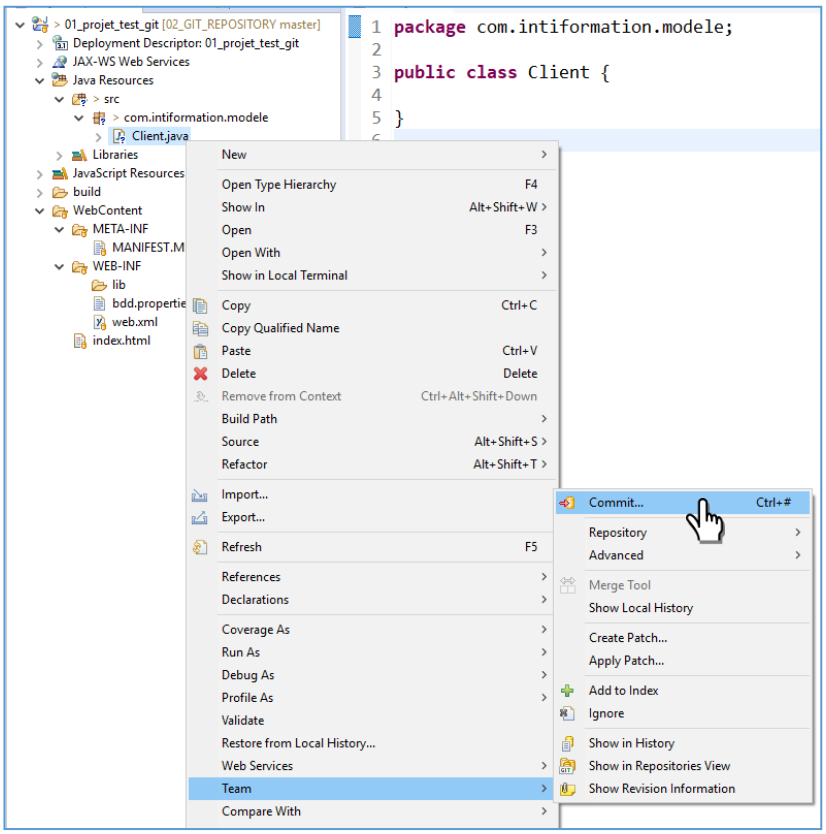


Ajout de modifications au projet Eclipse

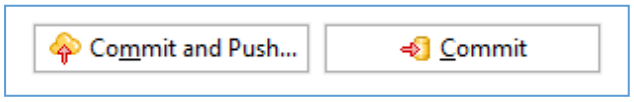
- On ajoute la classe **`Client`** au projet :

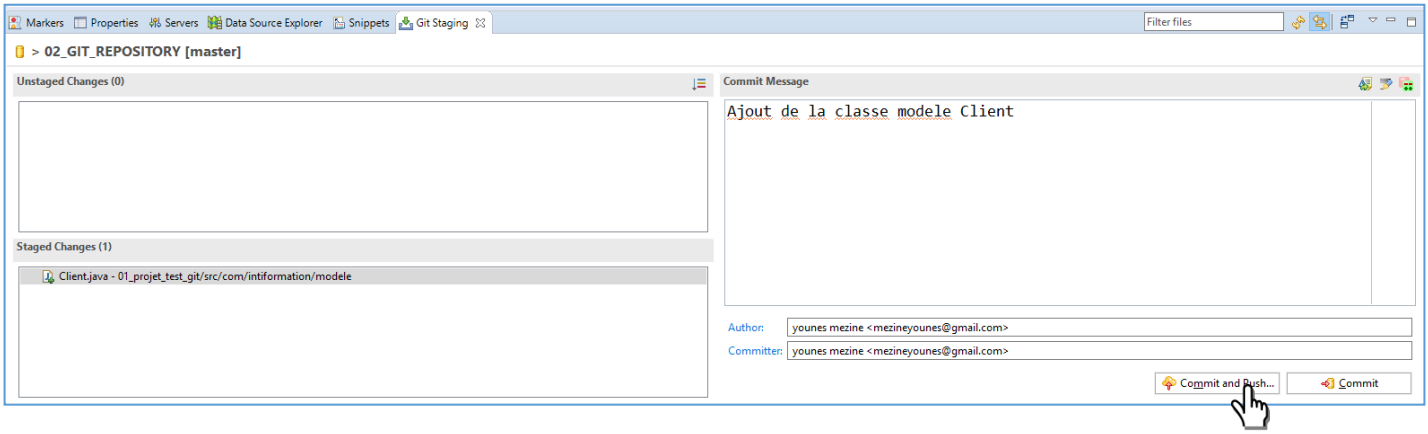


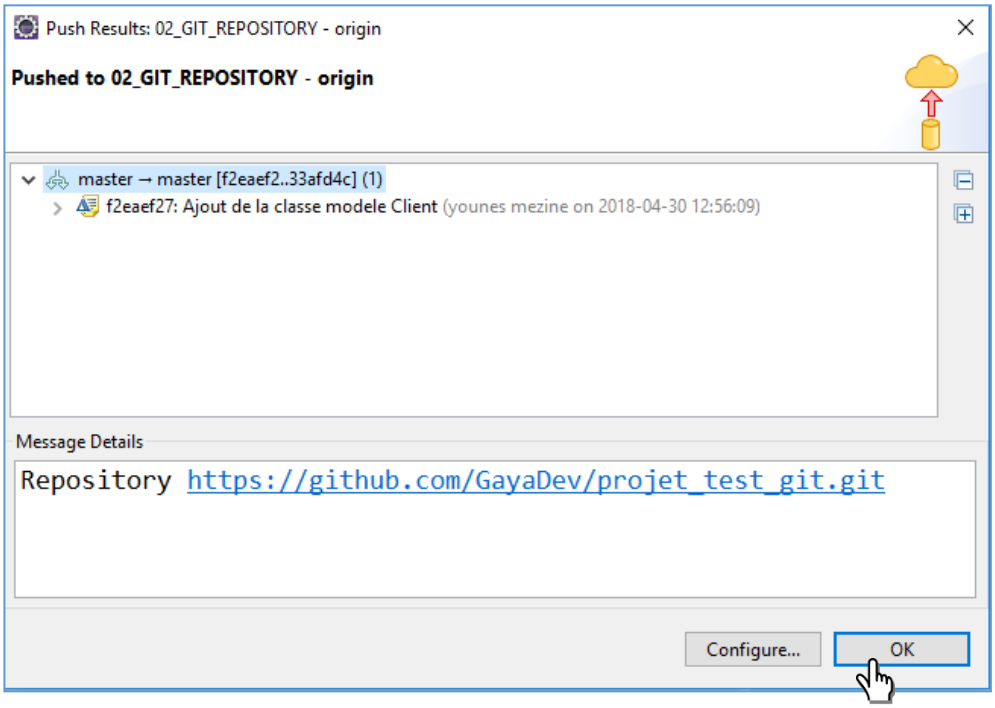
  - On peut faire un commit :



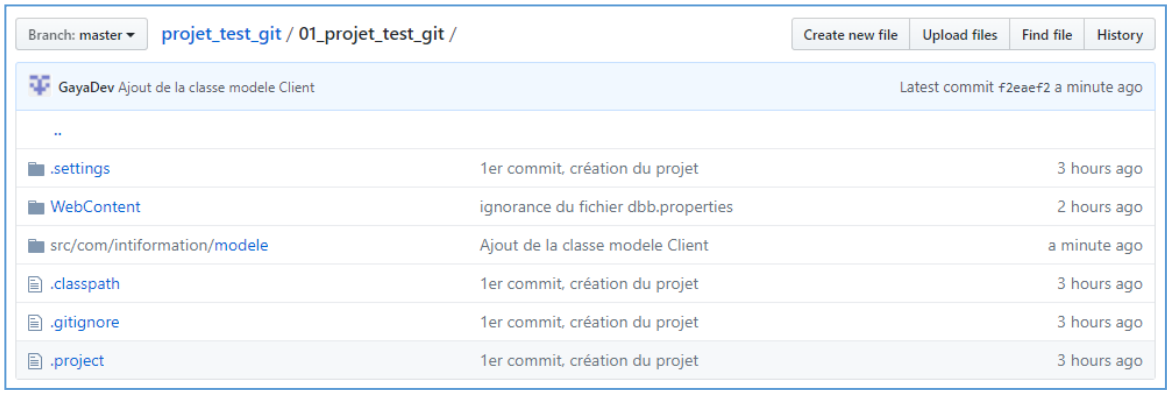
- On a la possibilité de faire un commit vers notre dépôt local et pousser en même temps les modifications vers notre dépôt distant GitHub avec le bouton







- Sur GitHub, on a bien notre dernier commit :

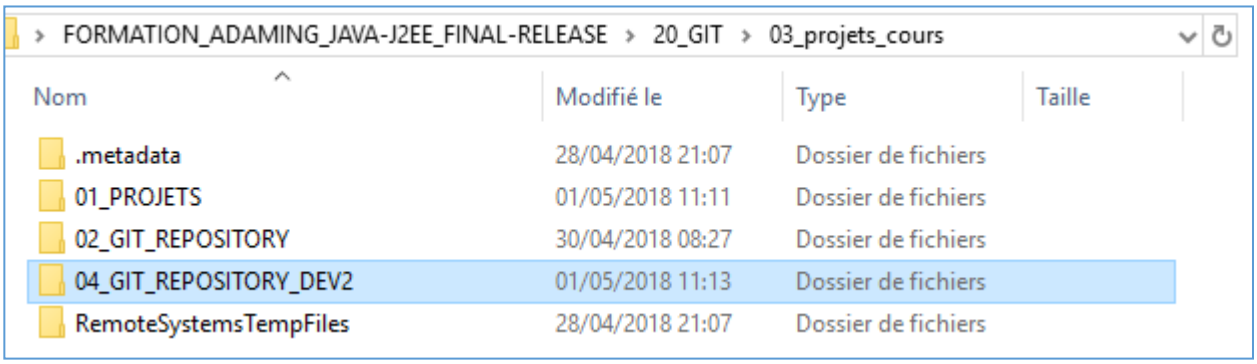


6. Récupérer des versions depuis le dépôt distant GitHub

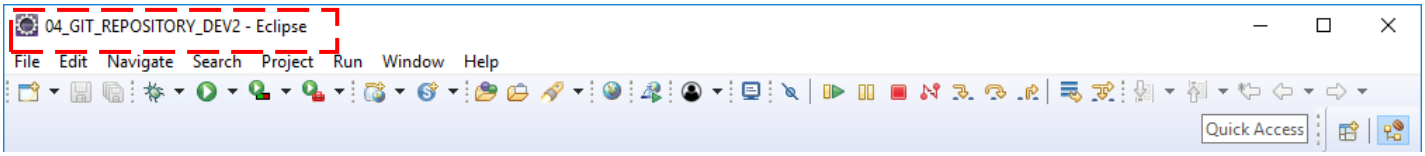
- Nous allons nous mettre dans la peau d'un autre développeur qui souhaitera contribuer à notre projet.

- Ce développeur n'est pas à l'initiative du projet. Il arrive sur GitHub, il voit le projet, il s y intéresse et il voudra synchroniser tout ce qui se passe sur GitHub avec son dépôt local et créer un par la même occasion.

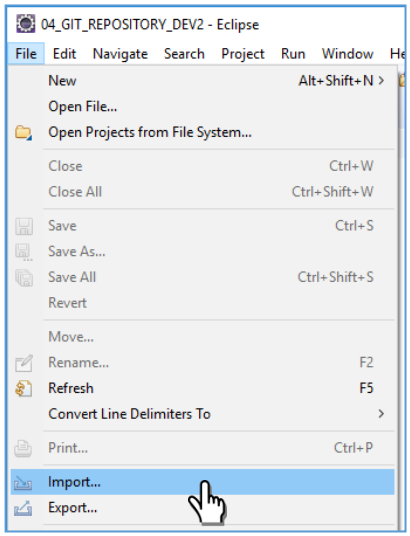
- D'abord, le développeur crée un dossier pour son dépôt local **`04_GIT_REPOSITORY_DEV2`**

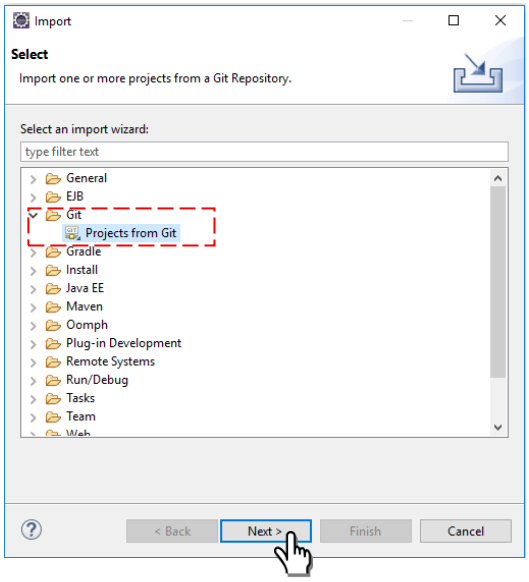


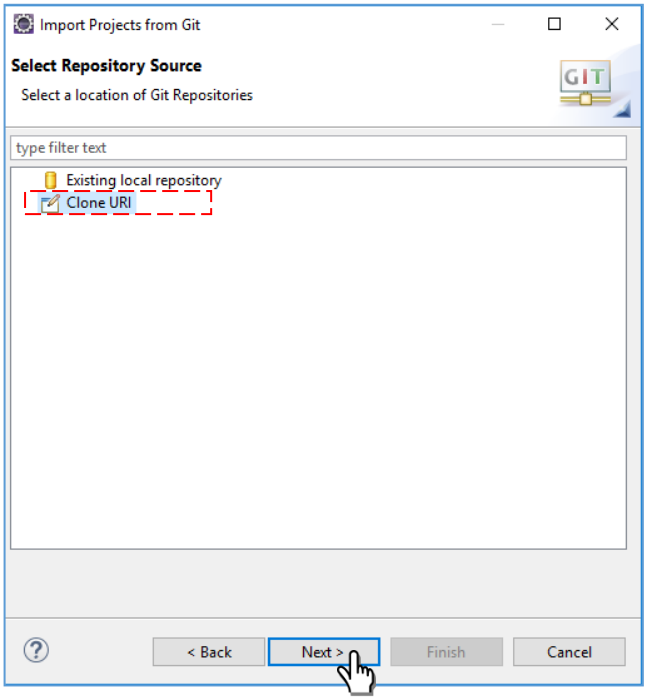
- Il ouvre un autre Workspace dans **`Eclipse`** :

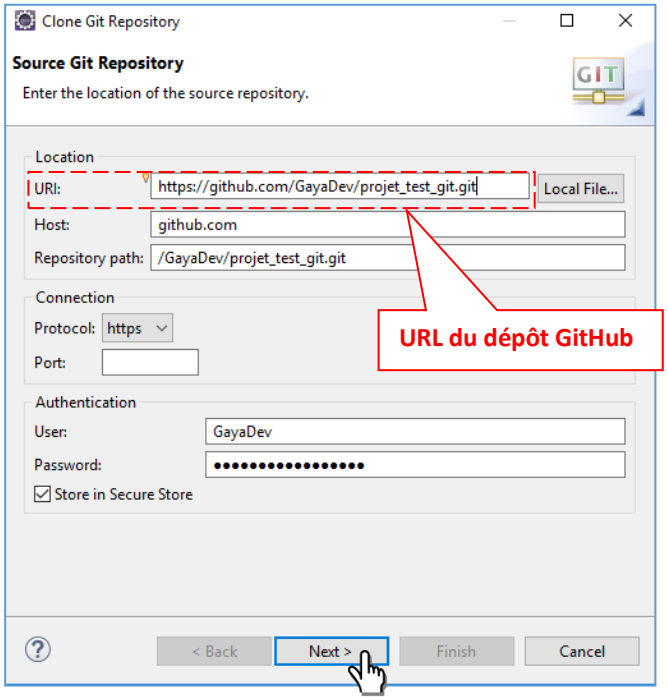


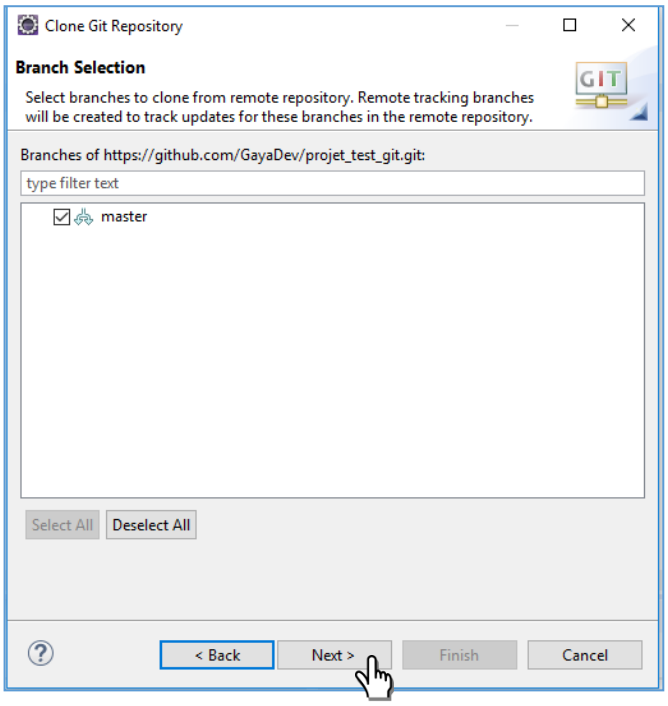
- Dans **`Eclipse`**, il va aller dans : `File > Import > Git > Projects from Git`

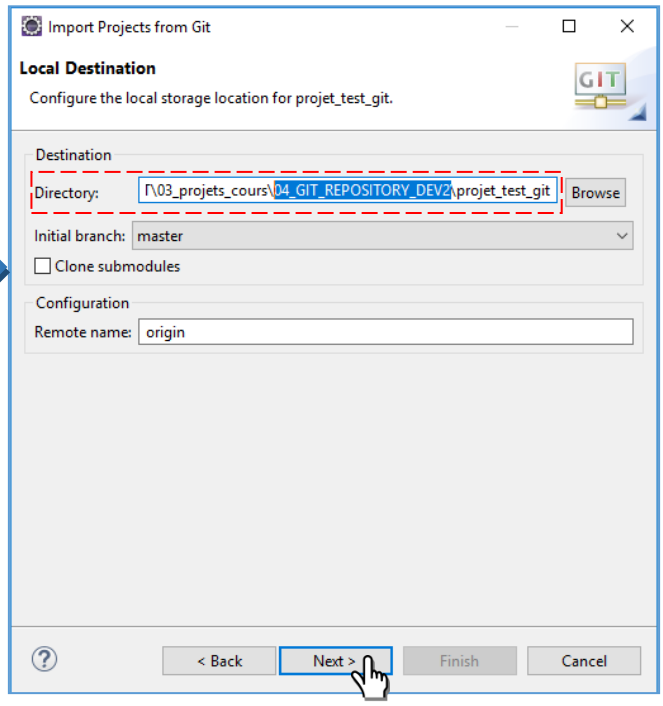


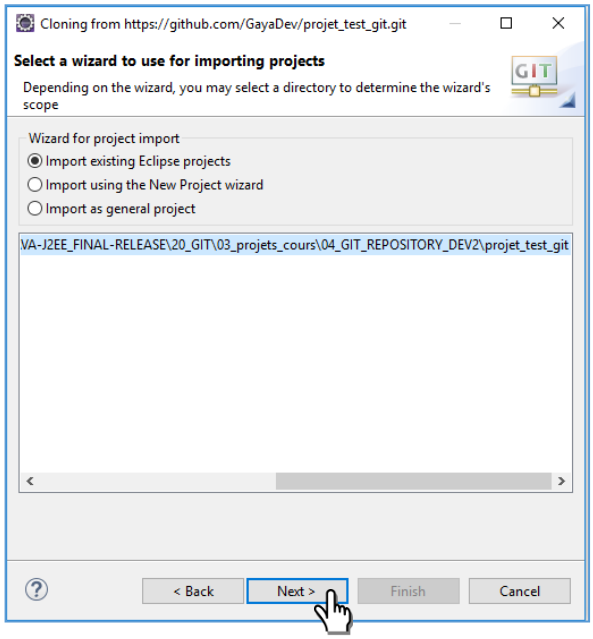


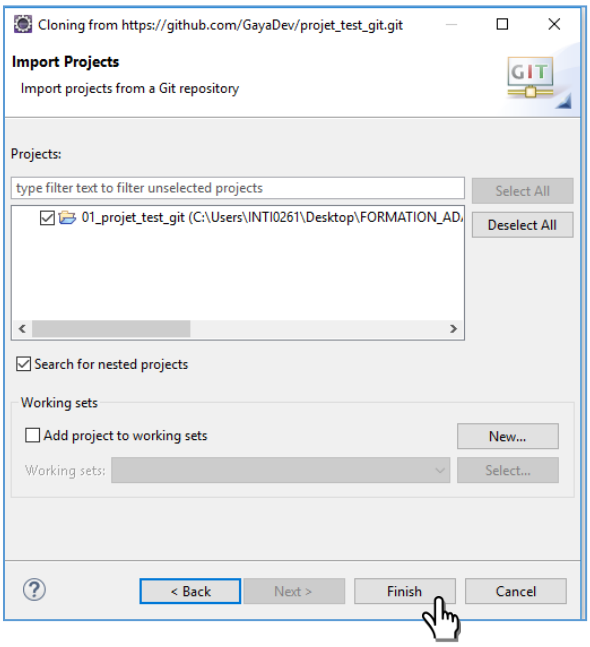

Import Projects from Git
Select Repository Source
Select a location of Git Repositories
type filter text
Existing local repository
Clone URI
< Back
Next >
Finish
Cancel


Clone Git Repository
Source Git Repository
Enter the location of the source repository.
Location
URI:
https://github.com/GayaDev/projet_test_git.git
Local File...
Host:
github.com
Repository path:
/GayaDev/projet_test_git.git
Connection
Protocol:
https
Port:
Authentication
User:
GayaDev
Password:
Store in Secure Store
URL du dépôt GitHub
< Back
Next >
Finish
Cancel

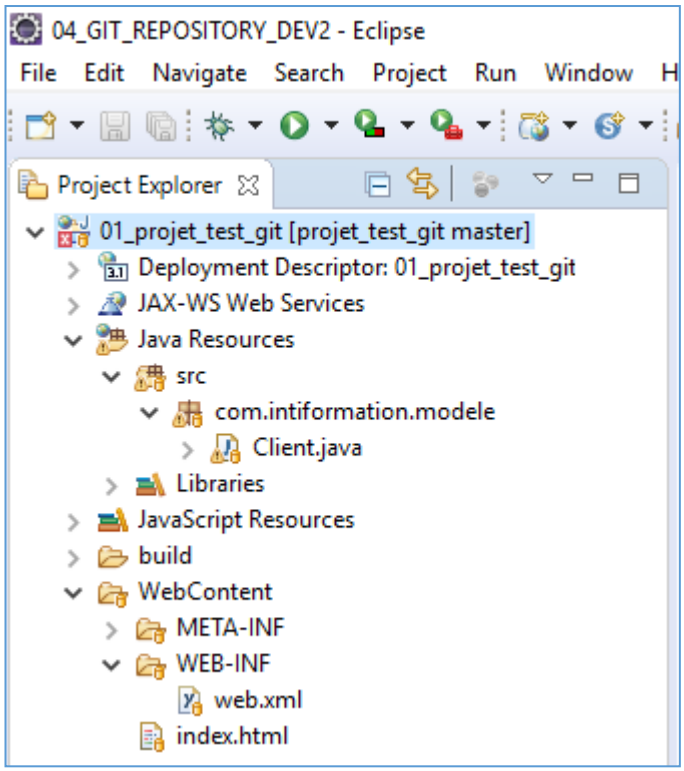

Clone Git Repository
Branch Selection
Select branches to clone from remote repository. Remote tracking branches will be created to track updates for these branches in the remote repository.
Branches of https://github.com/GayaDev/projet_test_git.git:
type filter text
master
Select All
Deselect All
< Back
Next >
Finish
Cancel


Import Projects from Git
Local Destination
Configure the local storage location for projet_test_git.
Destination
Directory:
I:\03_projets_cours\04_GIT_REPOSITORY_DEV2\projet_test_git
Browse
Initial branch:
master
Clone submodules
Configuration
Remote name:
origin
< Back
Next >
Finish
Cancel


Cloning from https://github.com/GayaDev/projet_test_git.git
Select a wizard to use for importing projects
Depending on the wizard, you may select a directory to determine the wizard's scope
Wizard for project import
Import existing Eclipse projects
Import using the New Project wizard
Import as general project
VA-J2EE_FINAL-RELEASE\20_GIT\03_projets_cours\04_GIT_REPOSITORY_DEV2\projet_test_git
< Back
Next >
Finish
Cancel

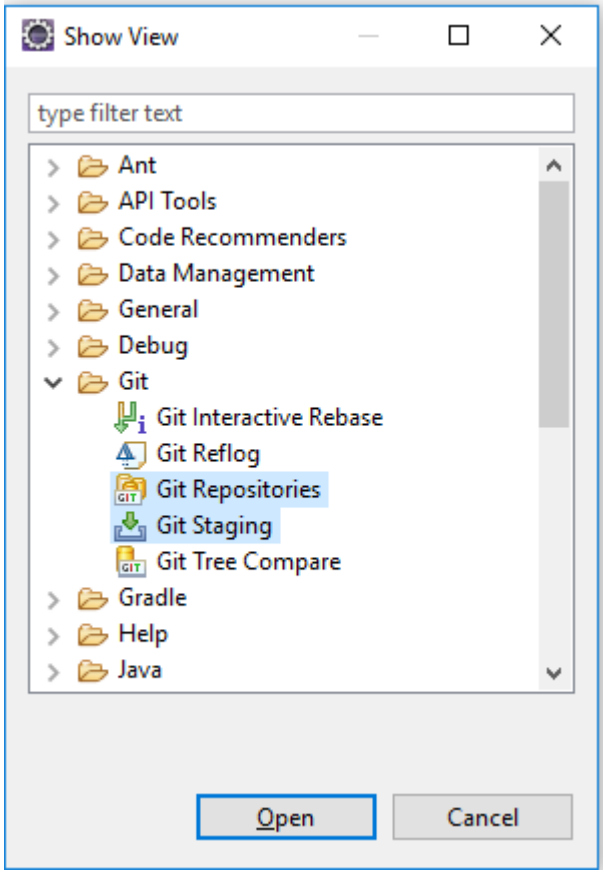

Cloning from https://github.com/GayaDev/projet_test_git.git
Import Projects
Import projects from a Git repository
Projects:
type filter text to filter unselected projects
Select All
Deselect All
01_projet_test_git (C:\Users\INTI0261\Desktop\FORMATION_AD
Search for nested projects
Working sets
Add project to working sets
New...
Working sets:
Select...
< Back
Next >
Finish
Cancel

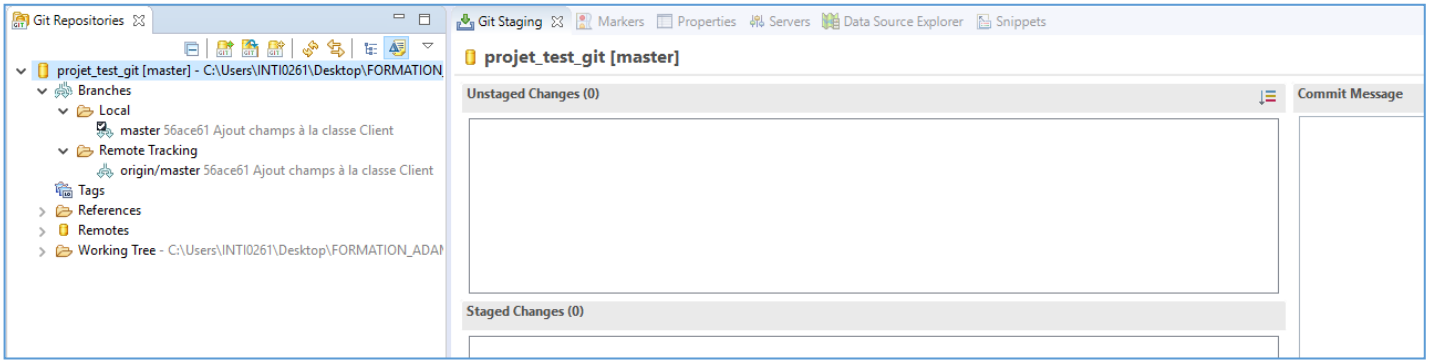
- Il a le projet dans son Workspace :



- Il va maintenant afficher les vues pour travailler avec Git :

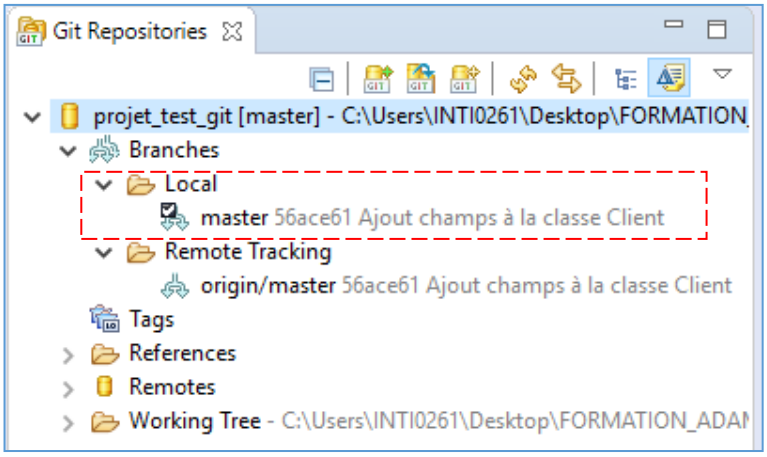
`Window > Show View > Other > Git >`



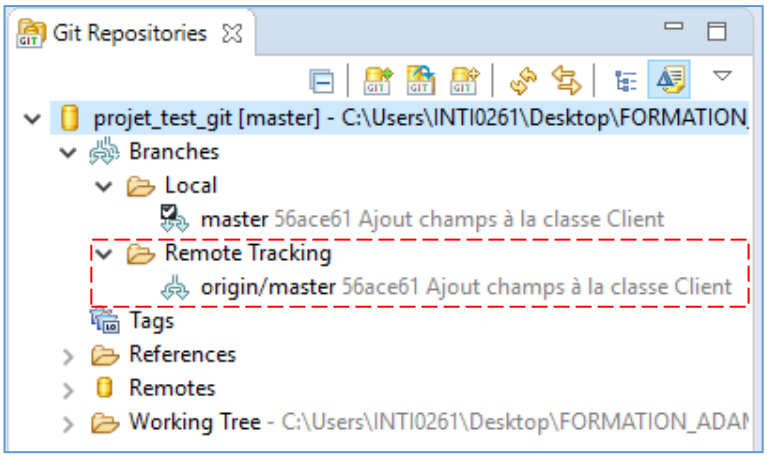


- Dans la vue **`Git Repositories`** :

  - Dans le dossier **`local`**, il a La branche locale **`master`** sur laquelle il va travailler :

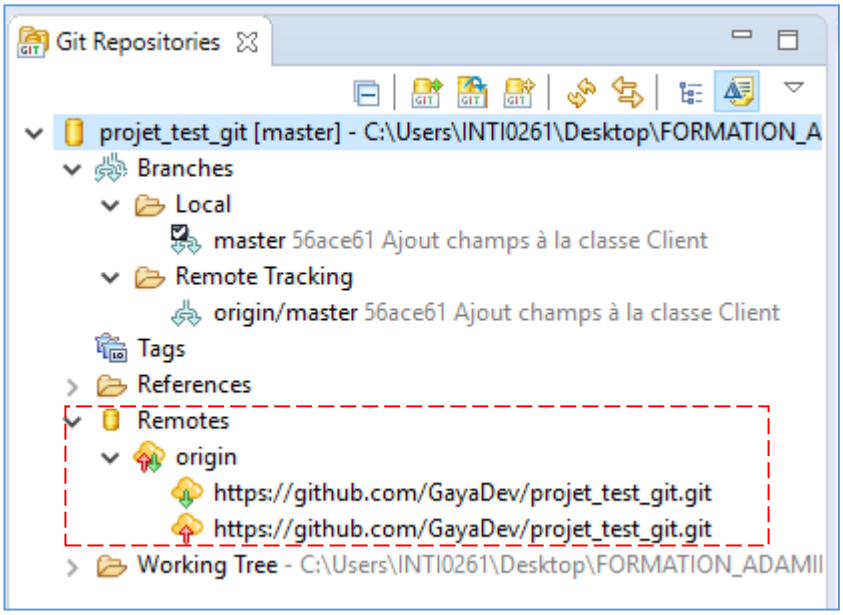


  - Dans le dossier **`Remote Tracking`**, il a La branche distante **`master`** du dépôt distant GitHub:



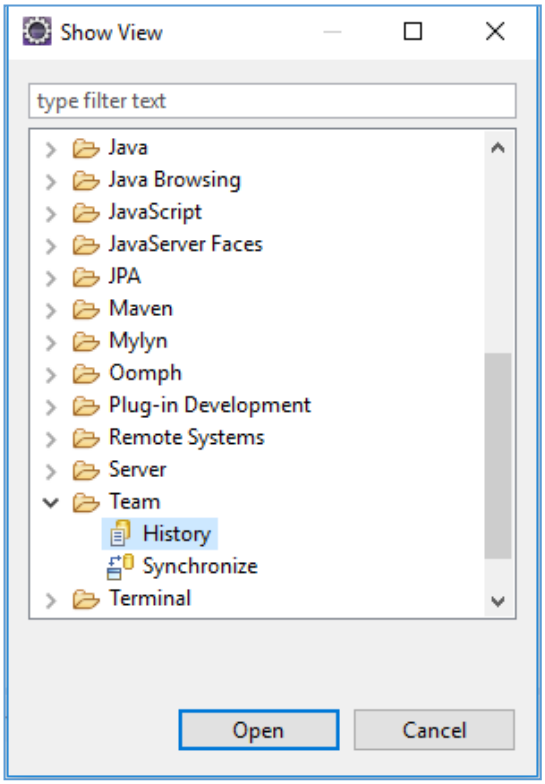
  - Les changements dans la branche locale master vont être poussés vers la branche distante master (Remote Tracking).

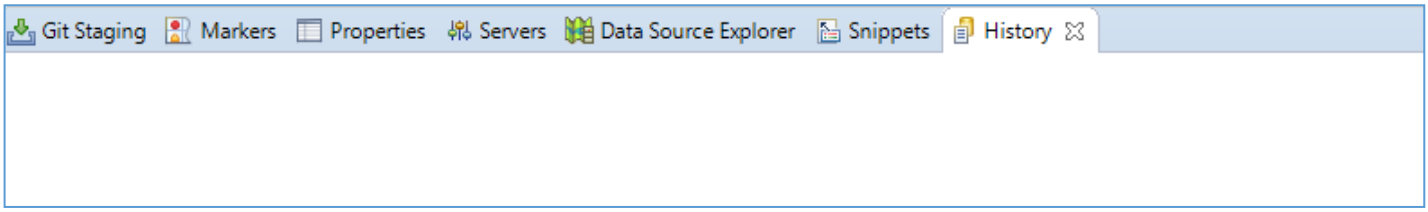
- Dans le dossier **`Remotes`**, il a l'adresse du dépôt GitHub :



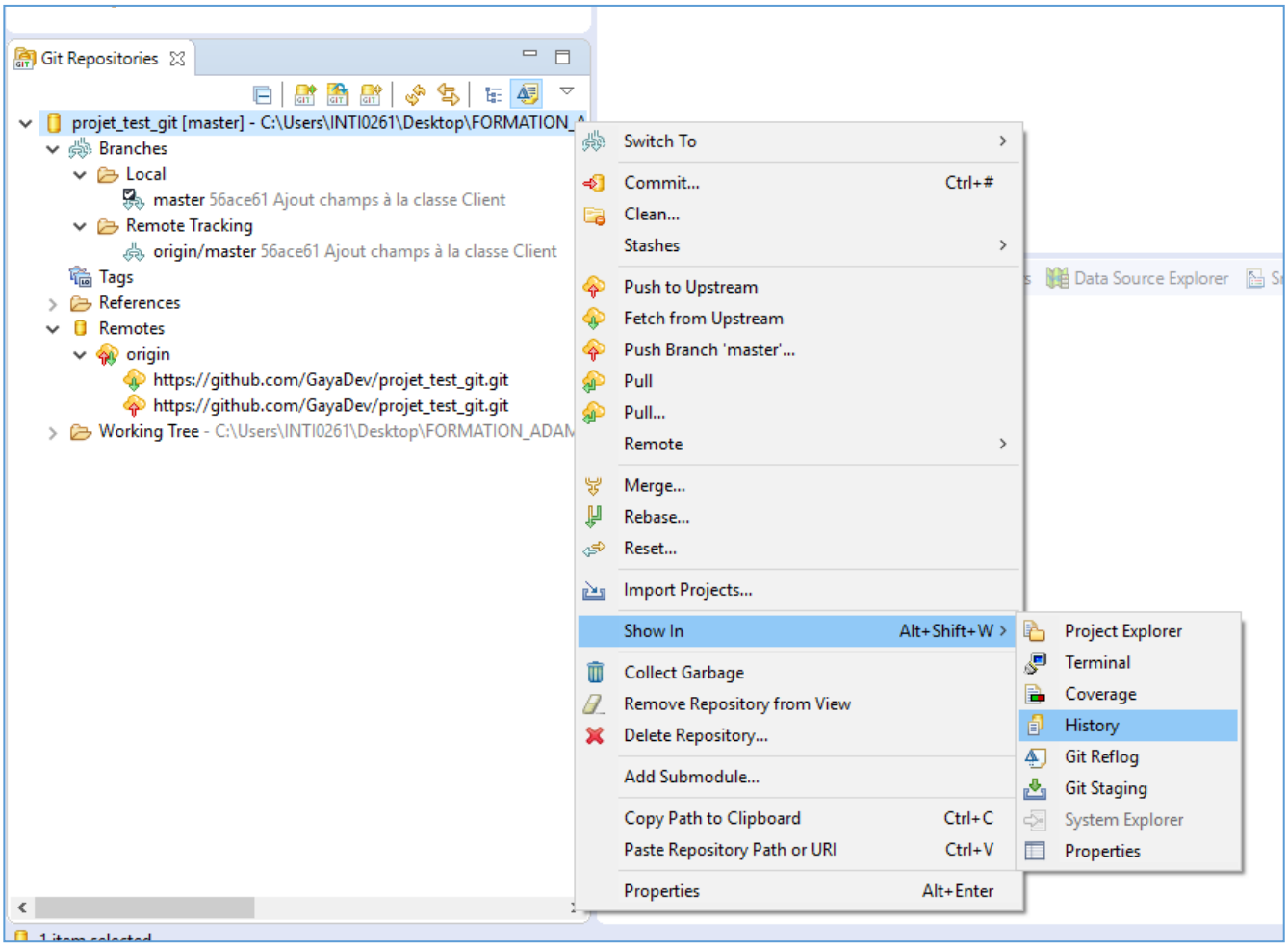
- S'il veut savoir l'historique du projet, il a la vue **`History`** :

`Window > Show View > Other > Team > History`

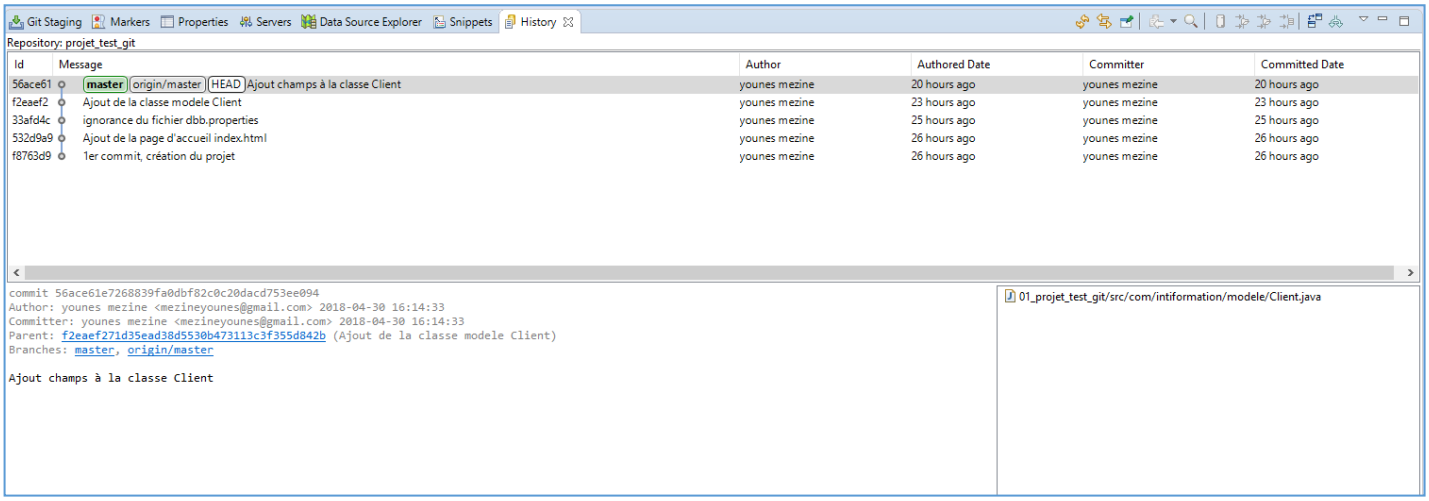




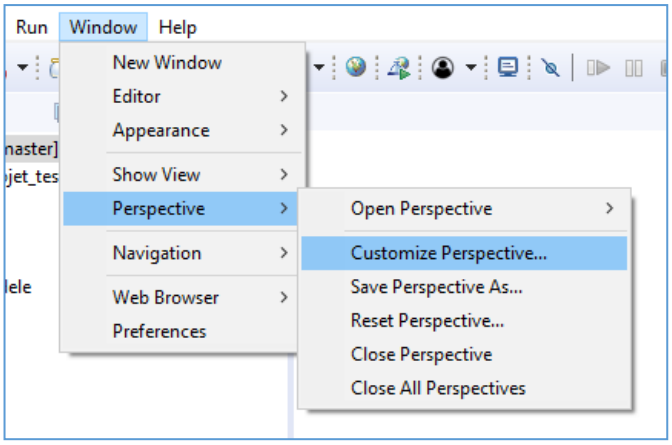
- Dans voire l'historique du projet et les différents commits :

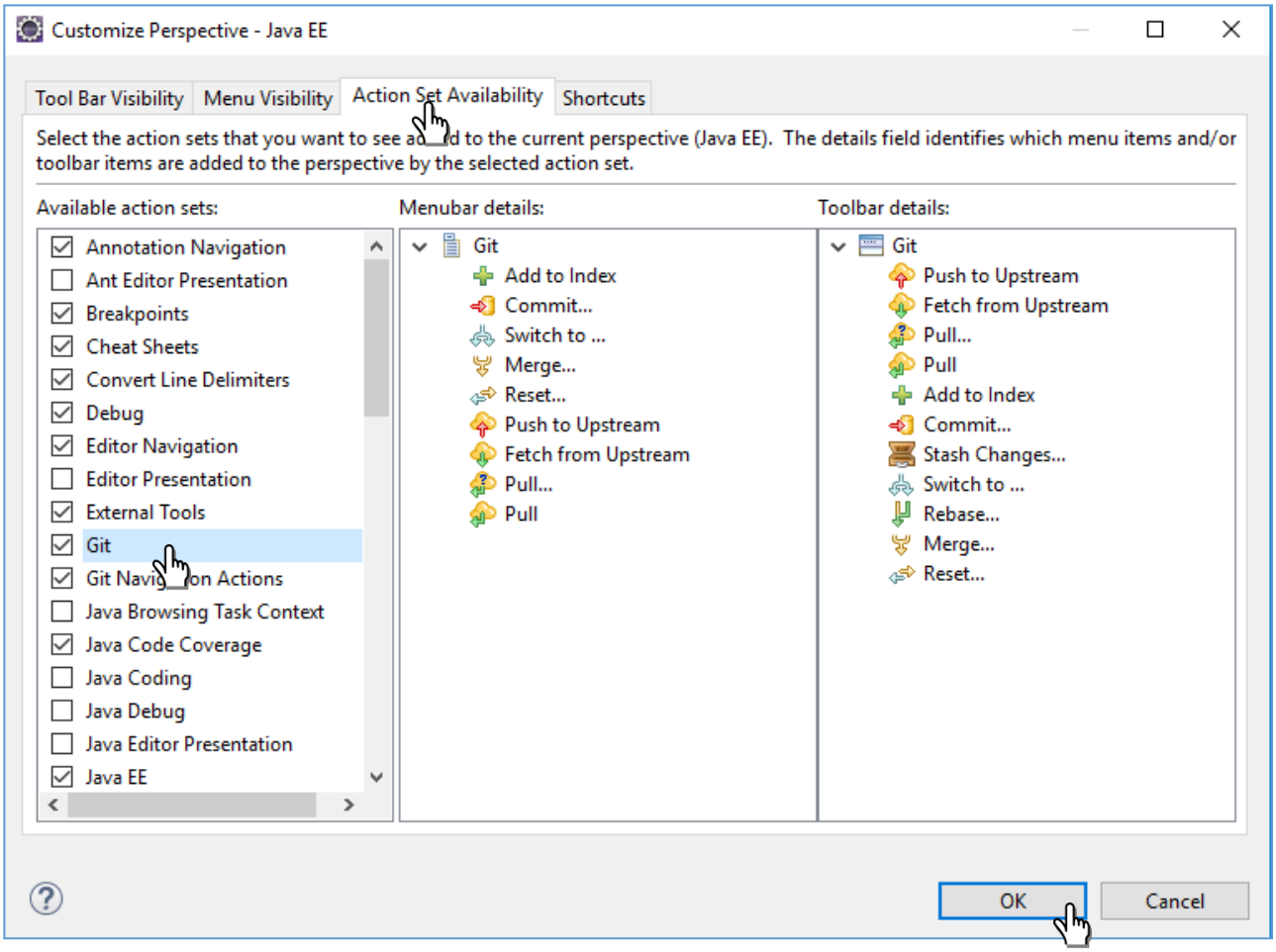


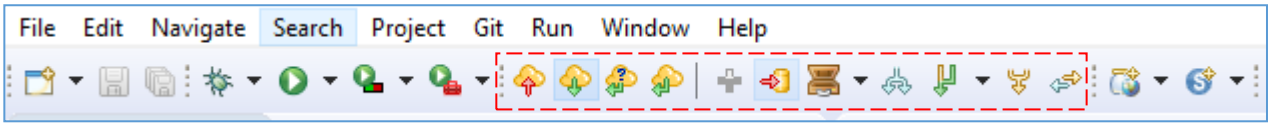
- L'historique s'affiche dans la vue **`History`**

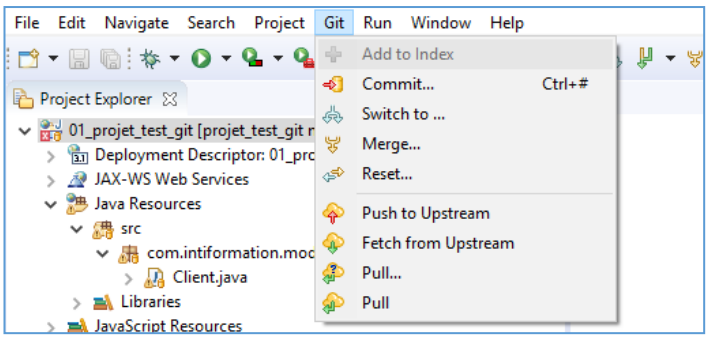


- il ajoute la **`Tools Bar`** (barre d'outils) de Git :







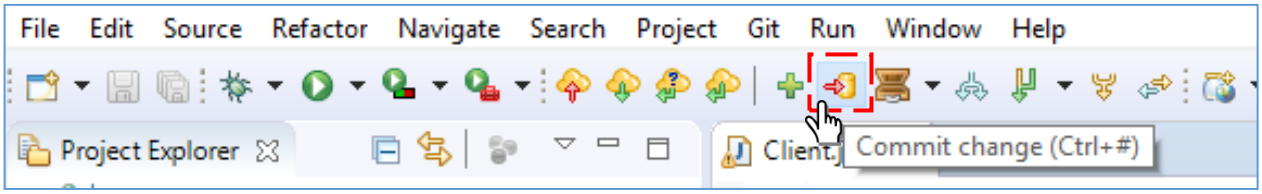


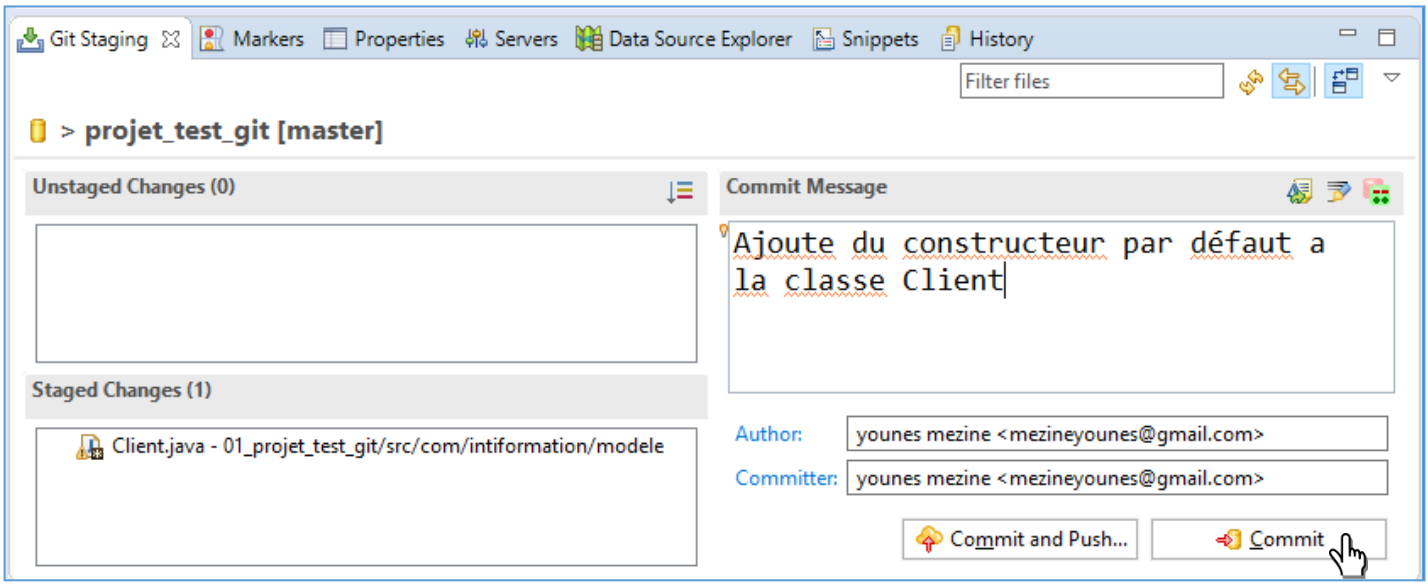
- Maintenant il peut commencer à développer le projet :

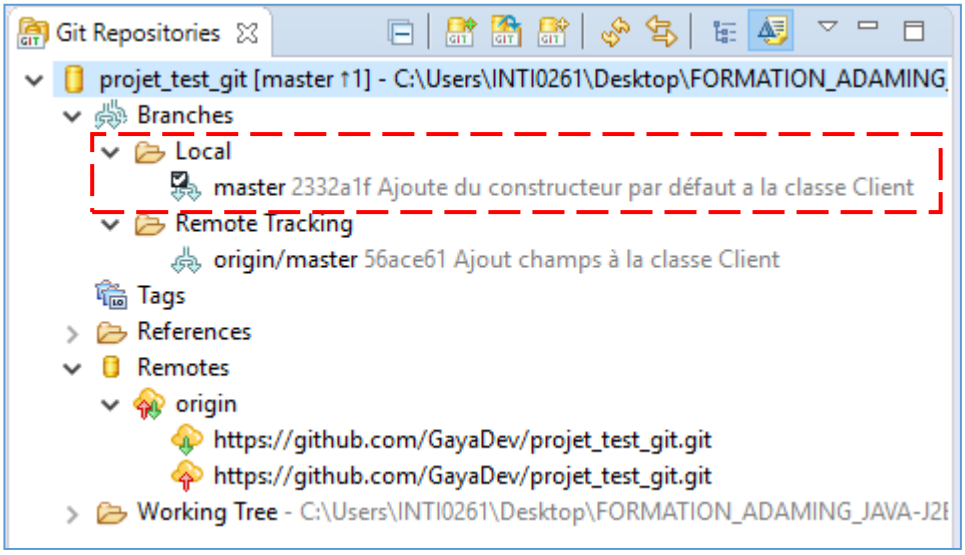
  - Il ajoute un constructeur à la classe Client :

```
Client.java
1  package com.intiformation.modele;
2
3  public class Client {
4
5      private int id;
6      private String nom;
7
8      /**
9       * ctor par défaut
10      */
11     public Client() {
12
13     }
14 }
15
```

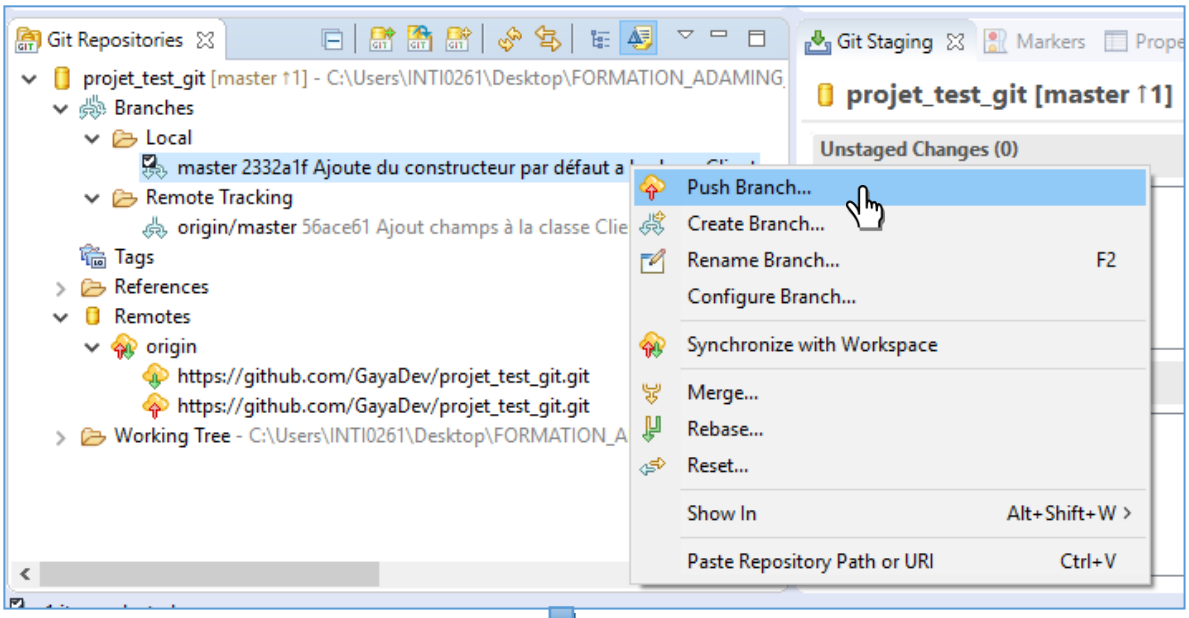
- Il va commiter les changements :

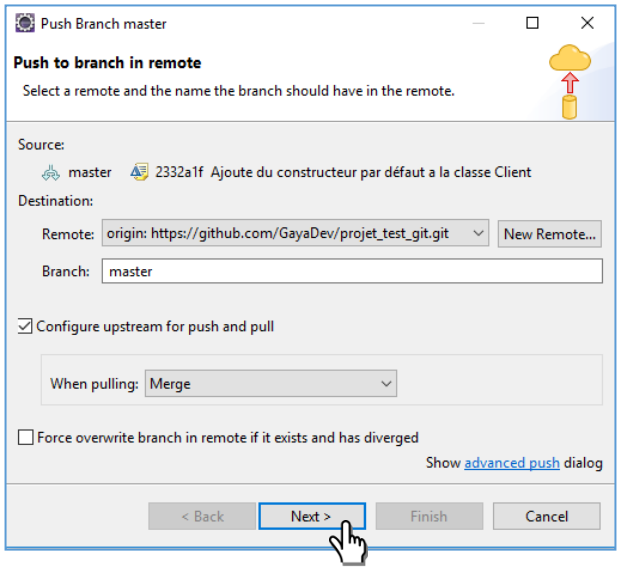


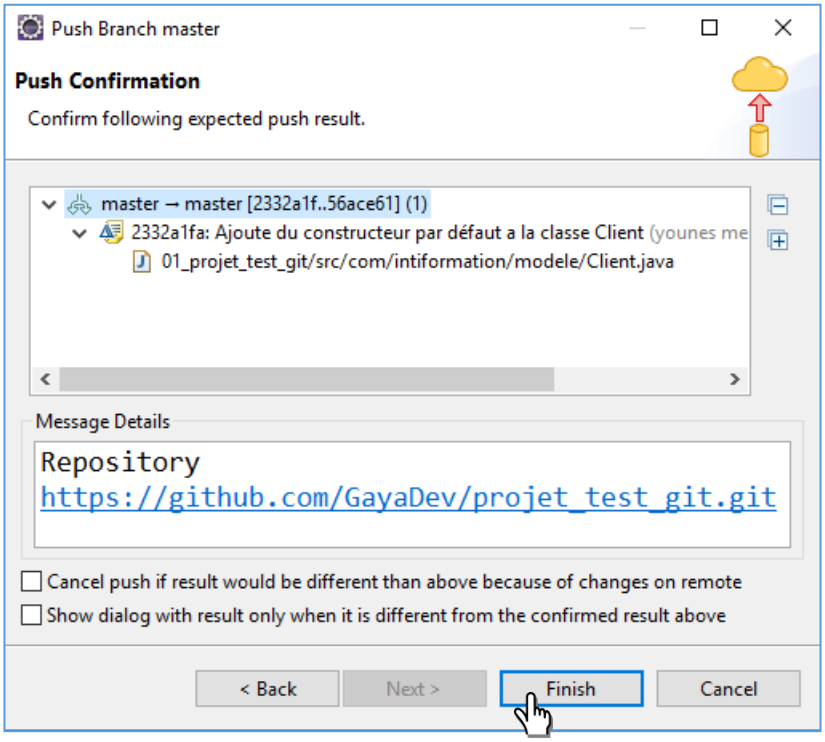


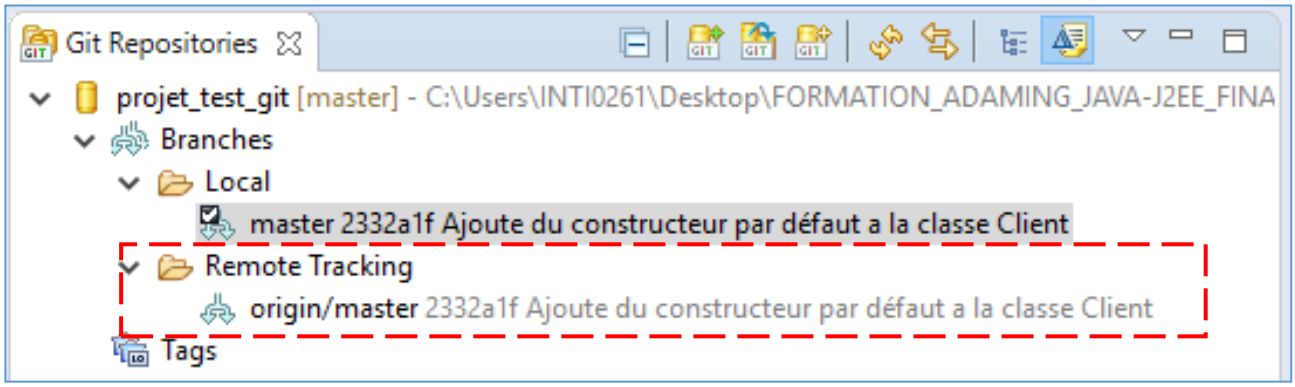


- Il peut pousser (**`push`**) les changements vers la branche master du dépôt distant avec un <u>click droit sur la branche master locale</u> :



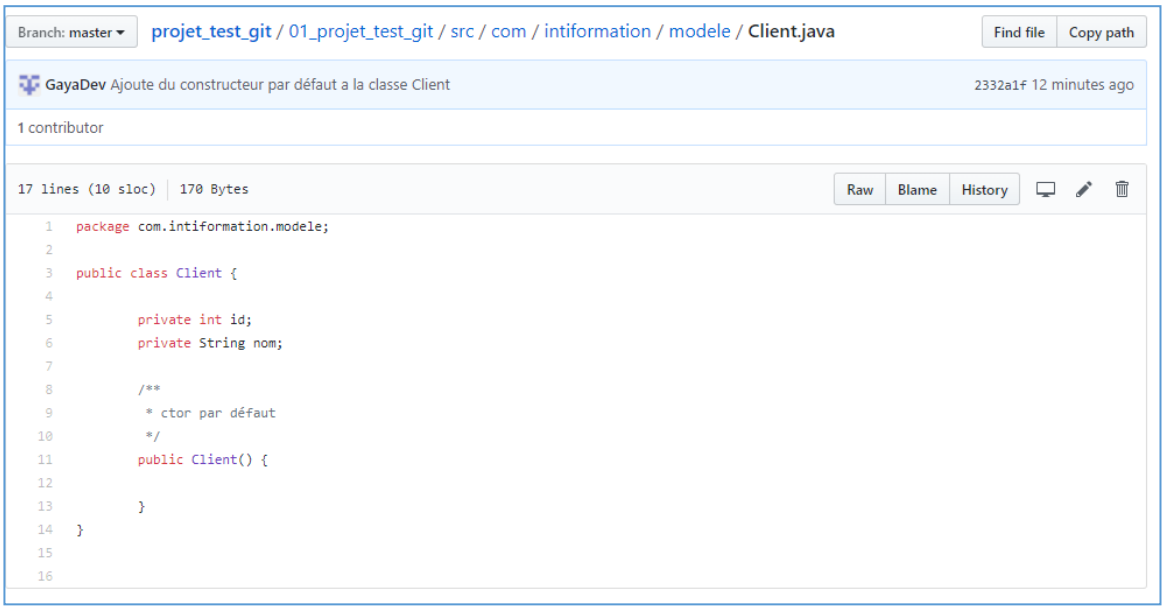






- **Les changements sont bien poussés vers la branche master distante avec le hash du commit.**

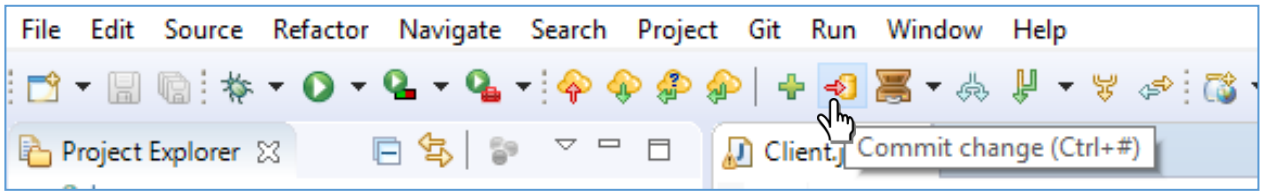
GitHub

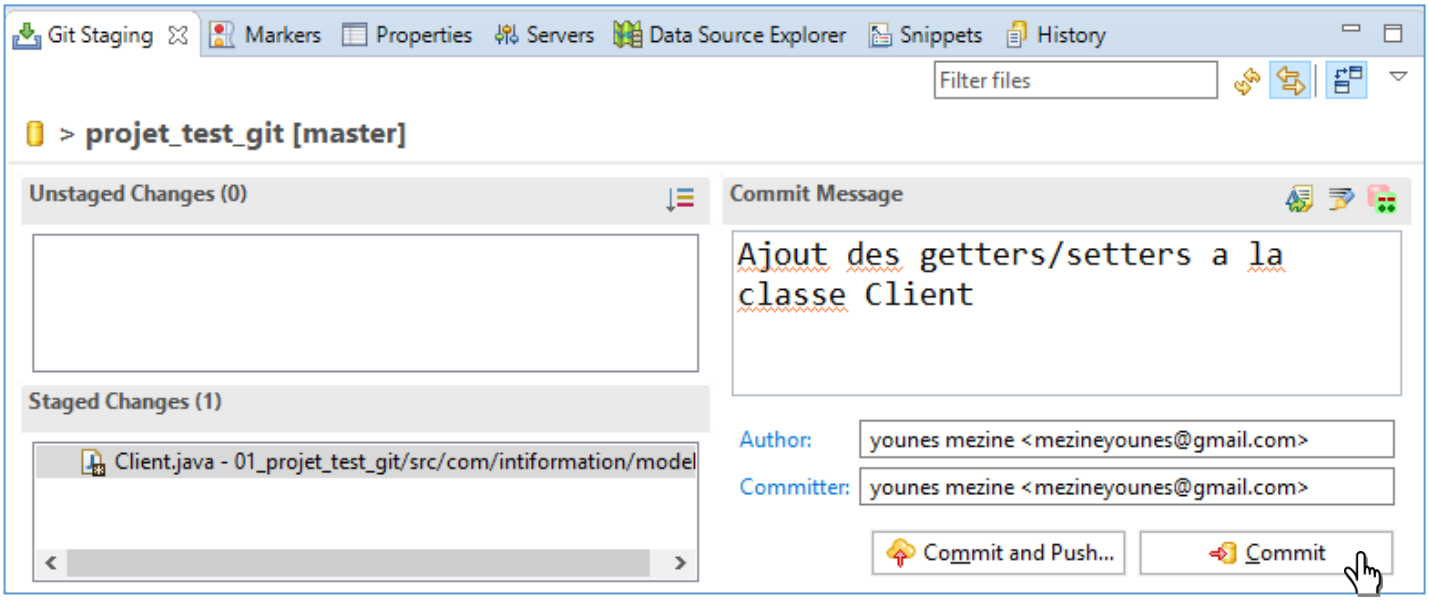


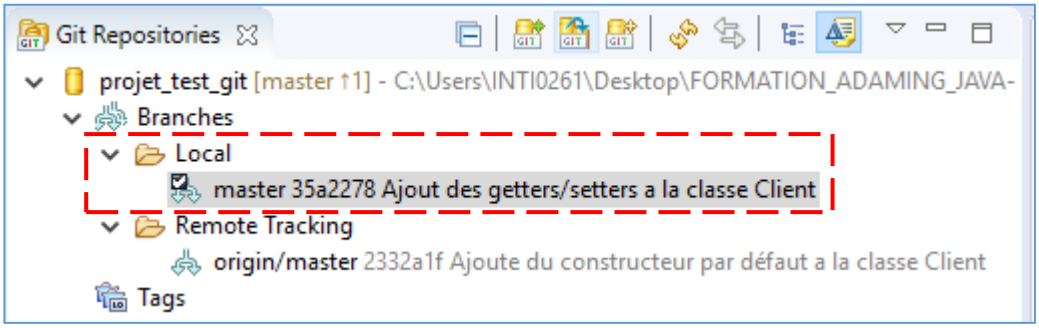
- Il ajoute un des getters/setters à la classe Client :

```java
*Client.java
1  package com.intiformation.modele;
2
3  public class Client {
4
5      private int id;
6      private String nom;
7
8      /**
9       * ctor par défaut
10      */
11     public Client() {
12
13     }
14
15     public int getId() {
16         return id;
17     }
18
19     public void setId(int id) {
20         this.id = id;
21     }
22
23     public String getNom() {
24         return nom;
25     }
26
27     public void setNom(String nom) {
28         this.nom = nom;
29     }
30
31 }
```

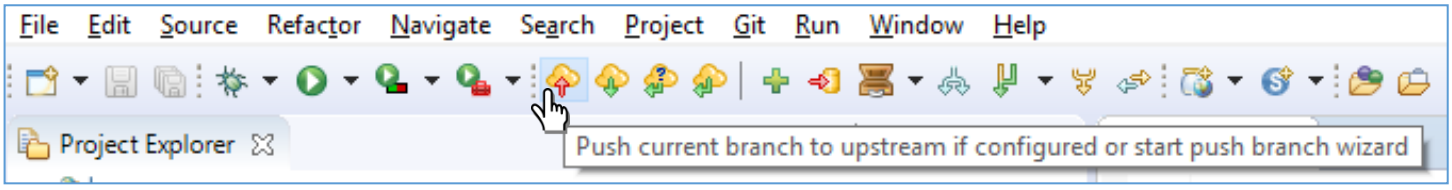
- Il commit les changements :

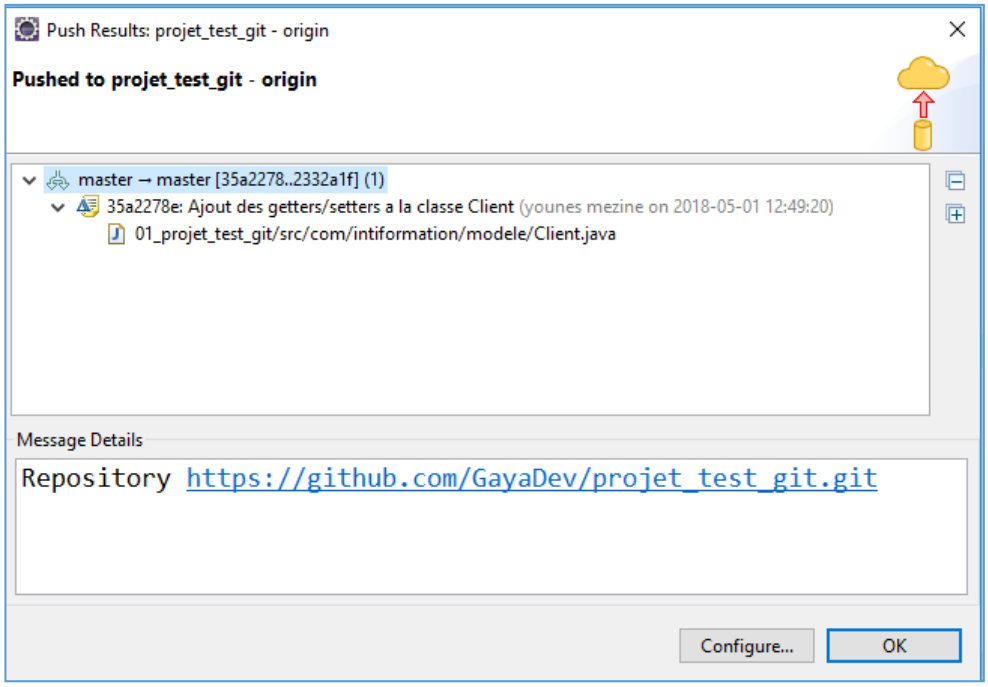


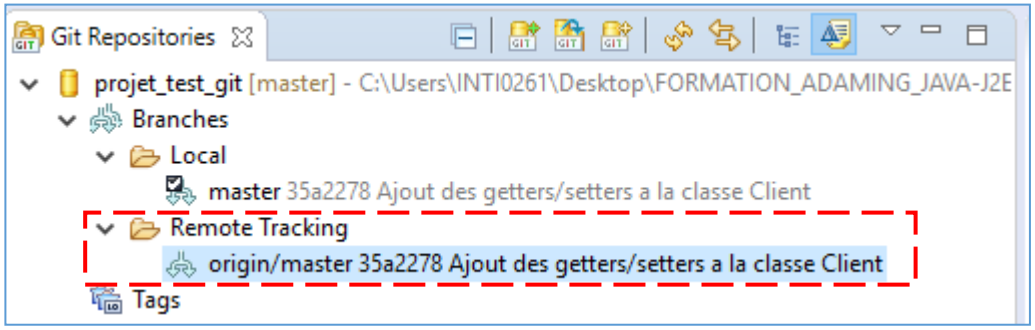




- Il peut pousser (**`push`**) les changements vers la branche master du dépôt distant avec un <u>click droit sur la branche master locale</u> :







7. créer une branche

Voire : http://www.cs.sfu.ca/CourseCentral/276/bfraser/videolist.html

- Changes in Branches: The standard workflow of making changes in topic branches, merging, and pushing.

Video YouTube : https://www.youtube.com/watch?v=rblGZRWqFVI

8. résolution de conflits

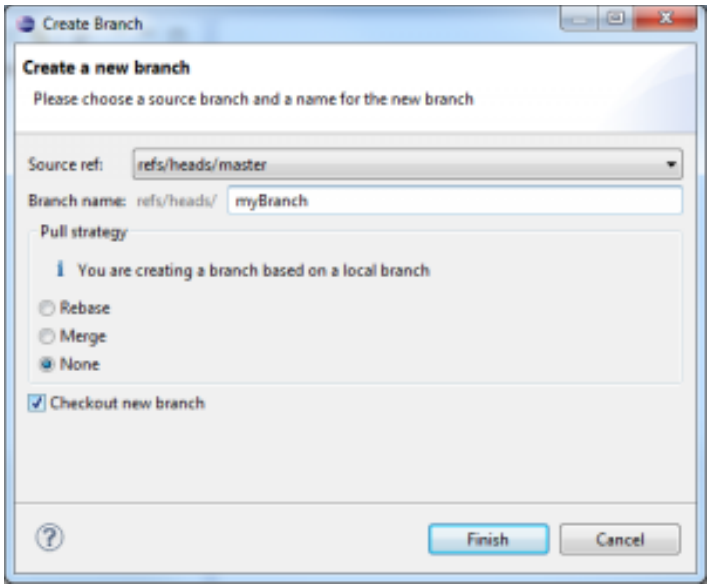
Voire : http://www.cs.sfu.ca/CourseCentral/276/bfraser/videolist.html

Video YouTube : https://www.youtube.com/watch?v=KfeqnernMmE

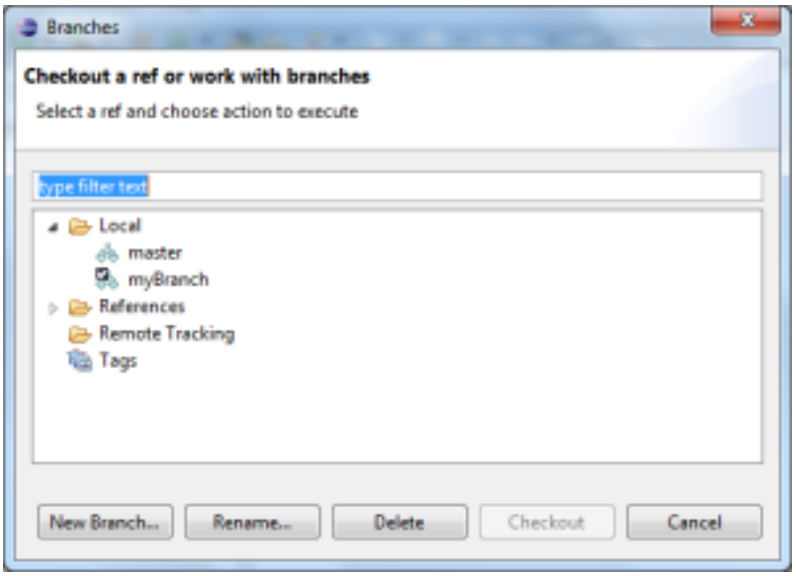
Remote Git Server and Merging:Git & Eclipse

Creating Branches

To create a new branch in your repository, right click a shared project and navigate to*Team => Switch to => New Branch…* from the context menu. Select the branch you want to create a new branch from, hit *New branch* and enter a name for the new branch.



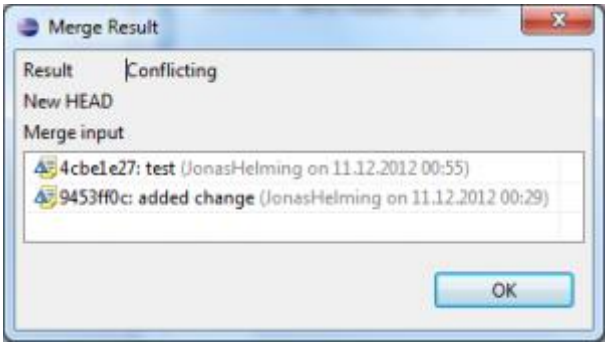
The new branch should appear in the branch selection window. If you would like to checkout the newly created branch, select it and click *Checkout*.



Merge

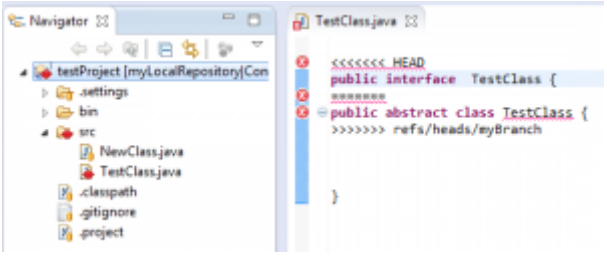
To merge one branch into another, you will have to checkout the branch you want to merge with. Right click the project node and navigate to *Team => Merge…* . Select any branch (other than the checked out branch) and hit *Merge*.

The merge will execute and a window will pop-up with the results. The possible results are *Already-up-to-date, Fast-forward, Merged, Conflicting, Failed*. A *Conflicting* result (see image below) will leave the merge process incomplete. You will have to resolve the conflicts (see next section). A *Failed* result may occur when there are already conflicting changes in the working directory.



Resolving Conflicts

If your merge resulted in conflicts (note the red symbols on the file icons), you will have to resolve these manually. Open the conflicting files and scroll to the conflicting changes marked with "<<<<<<<".



After you are finished the manual part of the merge, you will have to tell Git that the conflicts are resolved. To do so, *Add* the files and *Commit* to complete your merge.

Fetch and Pull

When cloning remote repositories, Git creates copies of the branches as local branches and as remote branches. A *Fetch* operation will update the remote branches only. To update your local branches as well, you will have to perform a *Merge* operation after fetching. The operation *Pull* combines *Fetch* and *Merge*. To perform a *Fetch*, select *Team => Fetch From…* from the project's context menu. Enter the repository you want to fetch branches from. (If you cloned this repository, the remote branch will be selected as default.) In the following window you will have to select what you want to fetch. As default, all branches are selected. The result of the *Fetch*-operation will be shown in a final confirmation window. Follow the same steps to apply a *Pull*.

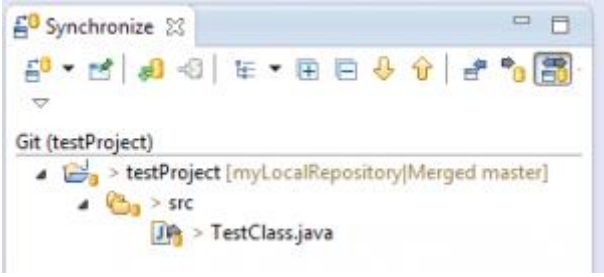
Push

Local changes made to your local branches can be pushed to remote repositories causing a merge from your branches into the branches of the remote repository (X pulls from Y is the same as Y pushes to X). The *Push* wizard is pretty much the same as the *Fetch* wizard. First, right click the project node and navigate to *Team=> Push…* . Enter the repository you want to push your branches to (the default for this will be the same as the*Fetch* default if you didn't configure a *Push* default) and hit *Next*. Choose the branches you want to push or click *Add all branches spec* if you want to push all branches. You can also select branches you want to delete from the remote repository. If you are done hit*Finish*. A final window will show the results of the *Push*.

Synchronize

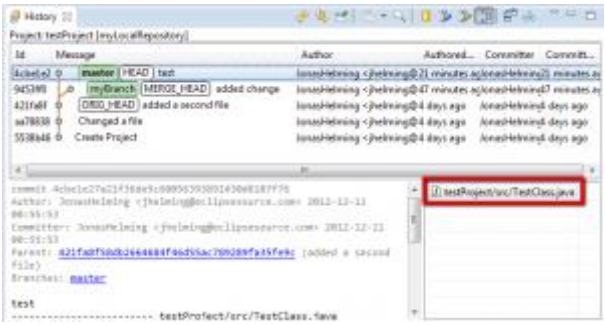
Comparisons between your workspace and the local repository or between the current branch and others and are done via the *Synchronize* operation. If you right click Team => Sychronize Workspace, your local workspace will be compared with the current branch showing uncommitted changes. If you select *Team => Advanced => Synchronize…* . , you can select other branches to compare your current branch with. In this case you can also include local uncommitted changes.

To compare the branches you may want to switch to the Synchronizing perspective, where you can get a more detailed view of several changes. Here is an example of a*Synchronize* operation in the Synchronizing perspective:



History View

To show any shared file's history, right click it and select *Team => Show in History*. This will open the History View, giving an overview of the commits and allowing you to perform several actions (compare, creating branches/tags, reset…). Every commit you select comes up with a revision comment and revision details. The revision comment (bottom left corner) includes parents, children, commit message and changes whereas the revision details (bottom right corner) name the changed files and the actions performed upon them (A=ADD, M=MODIFY, D=DELETE). Selecting a file in the revision details will scroll the revision comment to the changes to that file.



Creating Patches

Any patch can only include one commit that is from a parent to its child. To create a patch you have to open the History View first (see previous section). Right click a commit you want to create a patch for (this must be a child with exactly one parent) and select*Create Patch…*.

Select either *Clipboard* or *File* and hit *Next* and click *Finish*. The resulting patch can be applied to the parent commit via *Team => Apply Patch….*

Repository View

The repository view is useful when working with branches/tags and executing operations on them, as well as handling remote repositories and getting an overview of all your repositories. To open this view, select *Team => Show in Repositories View* from any file's context menu.

Additional Information

Icon Decorations/Signs

ignored: The repository treats these files as if they were non-existent (e.g. the bin-directory by default). Add a .gitignore file or *Team => Ignore* to ignore a file.

untracked: Any file known, but not yet recorded. To track a file, add it or select the *Show untracked files*-option in the commit-wizard and commit it directly.

tracked: Any file known to and recorded by the repository.

added: Any file known to the repository, but not yet committed. Perform a *Commit* to change this file's status to tracked.

removed: Any file that should be removed from the repository. For this icon to appear*Team => Untrack* has to be performed. By deleting the file from the workspace, the file will disappear (and therefore no icon will appear). However, it will still be removed from the repository with the next commit.

> *dirty*: Any tracked file with changes that have not yet been added to the index.

staged: Any tracked file with changes that are already included in the index.

> *partially-staged:* Any tracked file with changes, where some changes are already included in the index, and others that are not yet added.

> *conflicted*: Any file where the merge result caused a conflict. Resolve the conflicts and perform an *Add* operation to change this file's status.

assume-valid: Any modifications won't be checked by Git. This option can be activated via *Team => Assume unchanged*. However, it can only be turned off via the command line. Performing a *Reset* operation resets this status as well.

Index

The index, sometimes referred to as staging area, is an area between the working directory and the repository. Any change made to any file will change this file's status to*dirty* (see above). Any *dirty* file can be added to the index with an *Add* operation. The file's status changes to *staged*. You can compare files to the index and reset the index without resetting the workspace. In the original Git, files had to be added to the index before performing a *Commit* operation. This is not necessary in EGit, as *Team => Commit* allows you to commit unstaged changes.

Reset Types

You can reset your current branch to any other branch, tag or commit you want. Right click any commit in the History View and select *Reset*. There are three options available:

Soft:

The current branch's tip will point to this branch/tag/commit. Changes in the index and working directory, however, won't be reset.

Mixed:

Same as a soft reset, only that the current index will be replaced by the selected branch/tag/commit's index. The working directory stays unchanged.

Hard:

All changes will be reverted to the selected branch/tag/commit. Uncommitted changes will be lost, therefore this operation has to be confirmed.

Useful pages

- This tutorial is available now as a downloadable PDF.
- To create your own remote repositories and perform operations on them, you might want to register at https://github.com. As long as your repository is public, github is free.

- A tutorial with more information on certain options and actions:http://wiki.eclipse.org/EGit/User_Guide

Share it

Commandes Spécifiques Git :

`git init:`

Initialise un nouveau dépôt Git. Jusqu'à ce que vous exécutiez cette commande dans un dépôt ou répertoire, c'est simplement un répertoire normal. Seulement après avoir entré cette commande, il accepte les commandes Git qui suivent.

`git config:`

raccourci de "configurer," ceci est tout particulièrement utile quand vous paramétrez Git pour la première fois.

`git help:`

Oublié une commande ? Saisissez-ça dans la ligne de commande pour ramener les 21 commandes les plus courues de Git. Vous pouvez aussi être plus spécifique et saisir "git help init" ou tout autre terme pour voir comment utiliser et configurer une commande spécifique git.

`git status:`

Vérifie le statut de votre Repository. Voir quels fichiers sont à l'intérieur, quels sont les changements ayant besoin d'être gardés en mémoire, et sur quelle branche du repository vous êtes en train de travailler.

`git add:`

Ceci n'ajoute pas de nouveaux fichiers dans votre dépôt. Au lieu de cela, cela porte de nouveaux fichiers à l'attention de Git. Après que vous ayez ajouté des fichiers, ils sont inclus dans les "instantanés" du dépôt Git.

`git commit:`

la commande la plus importante de Git. Après avoir fait toute sorte de modification, vous saisissez ça afin de prendre un "instantané" du dépôt. Généralement cela s'écrit sous la forme git commit -m "Message ici". Le -m indique que la section suivante de la commande devrait être lue comme un message.

`git branch:`

Vous travaillez avec plusieurs collaborateurs et voulez produire des modifications de votre côté ? Cette commande vous permet de construire une nouvelle branche, ou une timeline de commits, de modifications et d'ajouts de fichiers qui sont complètement les vôtres. Votre titre

va après la commande. Si vous vouliez créer une nouvelle branche appelée “chats”, vous saisiriez git branch chats.

`git checkout:`

Permet littéralement de rapatrier un dépôt dans lequel vous n’êtes pas. C’est une commande de navigation qui vous laisse migrer vers le répertoire que vous voulez rapatrier. Vous pouvez utiliser cette commande sous la forme `git checkout` *master pour rapatrier la branche master, ou git checkout chats pour rapatrier une autre branche.*

`git merge:`

Quand vous avez terminé de travailler sur une branche, vous pouvez fusionner vos modifications vers la branche master, qui est visible pour tous les collaborateurs. git merge cats prendrait toutes les modifications que vous avez faites sur la branche “cats” et les ajoutera à la branche master.

`git push:`

Si vous travaillez sur votre ordinateur local, et voulez que vos commits soient visibles aussi sur GitHub, vous “push”ez les modifications vers GitHub avec cette commande.

`git pull:`

Si vous travaillez sur votre ordinateur local, et su vous voulez la version la plus à jour de votre repository pour travailler dessus, vous “pull”ez (tirez) les modifications provenant de Github avec cette commande.

Les Ressources Git :

Glossaire Git pour les Nuls

- `git init` : Initialise un dépôt git dans un répertoire vide.
- `git status` : Affiche le statut des choses dans le répertoire suivi
- `git add <NOM DU FICHIER A SUIVRE>` : Ajoute un fichier à suivre dans la zone d'attente
- `git commit -m "entrez votre message"` : Consigner (déclarer) toute modification.
- `git remote add origin [adresse URL GitHub ici]` : Connecte votre dépôt local avec GitHub.
- `git push -u origin nom_branche` : Pousse les modifications vers GitHub
- `git checkout` : Littéralement "check out" (retirer, rapatrier) une branche.
- `git -b <NomBranche>` : Crée une nouvelle branche dans votre dépôt.
- `git log`: affiche un journal des modifications au dépôt
- `git clone <URL PROVENANT DE GITHUB>` : clone un projet de votre système à partir de github
- `git branch <nom_branche>` : crée une copie de la branche master appelée
- `git checkout <branch_name>` : bascule vers nom_branche comme branche de travail
- `git checkout master` : bascule la branche vers master
- `git branch -a` : affiche les branches existantes pour le dépôt particulier

Autres ressources :

Git - le petit guide: http://rogerdudler.github.io/git-guide/index.fr.html

Code school : https://www.codeschool.com/courses/try-git

GitHub Guides: https://www.youtube.com/user/GitHubGuides

Git Références : http://gitref.org/

Pro Git : http://git-scm.com/book/fr/v1