

COURS ANNEXE 02

GERER SON CODE AVEC GIT

Icam Site de Strasbourg-Europe

Mathias REHEISSER

2021-2022

Page laissée vide intentionnellement

TABLE DES MATIERES

1.	Introduction	5
2.	Installation de Git.....	7
3.	Le fonctionnement de Git	9
3.1.	Gestion des modifications	9
3.2.	Gestion des branches	14
4.	Interface graphique.....	19
4.1.	Installation de Git Graph et prise en main.....	19
4.2.	Commandes de base	21
4.3.	Gestion des conflits	25
5.	travail collaboratif à distance avec Git Hub	31
5.1.	Fonctionnement d'un dépôt en ligne	31
5.2.	Ajouter des collaborateurs à un projet privé.....	36
5.3.	Création de branche	38
5.4.	« Fetch » et « pull »	39

Page laissée vide intentionnellement

1. INTRODUCTION

Vous êtes en train de développer une application, mais vous avez besoin de vérifier une version précédente de votre code. Ou alors, vous souhaitez gérer efficacement vos versions de votre programme, mais entre celle qui tourne en production, celle en environnement de test, et celle en développement, on s’y perd facilement.

Si en plus vous développez à plusieurs, chacun à sa version avec ses modifications, et quand il s’agit de mettre en commun...c’est le bazar !

Heureusement, il existe un formidable outil pour nous aider à gérer les différentes versions d’un programme, il s’agit de Git !



Mais du coup, **qu’est-ce que Git ?**

Git est un logiciel de gestion de versions de code. Il nous permet d’avoir différentes versions d’un système qui forment ainsi des « branches », « branches » qu’on peut fusionner, séparer en plusieurs versions...etc.

Aussi, il nous permet d’avoir la chronologie des modifications, des indications concernant celles-ci...etc.

Associé à la plateforme en ligne GitHub, il nous permet de stocker toutes ces versions en ligne, et ainsi de travailler en équipe sur un même projet, chacun ayant sa propre version et pouvant récupérer les modifications des autres.

Page laissée vide intentionnellement

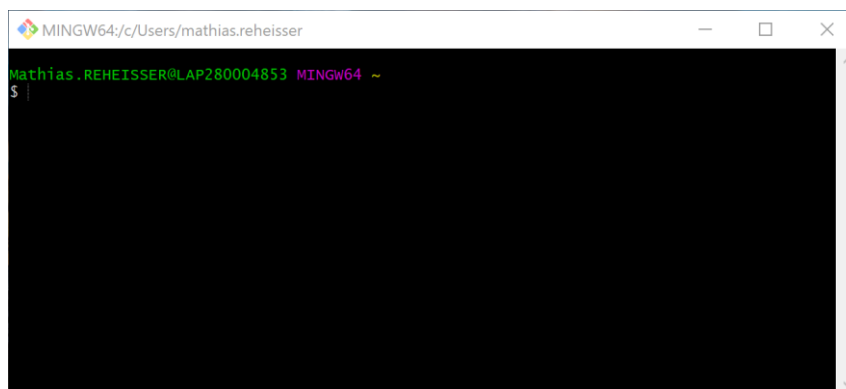
2. INSTALLATION DE GIT

Git est téléchargeable à l'adresse suivante :

<https://git-scm.com/>

Récupérez la dernière version et installez-la. Utilisez les versions recommandées et/ou par défaut pour les différentes options.

Une fois Git installé, vous devriez avoir la console suivante :



```
MINGW64: c:/Users/mathias.reheisser
Mathias.REHEISSER@LAP280004853 MINGW64 ~
$
```

« Hein ? Mais elle est où la jolie interface ? C'est quoi ce rectangle noir tout moche ? »

Alors, ce rectangle noir, ça s'appelle une « **console** », ou un « **invite de commande** ». Il permet une interface très simple pour exploiter un logiciel, en utilisant des « **commandes** », sans qu'on ait à développer une **interface graphique**. C'est une pratique courante dans le milieu de l'informatique.

« Oui mais ce n'est pas très intuitif. Il n'y a pas d'indication, de boutons sur lesquels cliquer... et puis c'est moche ! »

C'est vrai ! Les développeurs de Git ont préféré se concentrer sur les fonctionnalités de l'outil plutôt que sur une interface ergonomique. Ça n'a pas empêché d'autres ingénieurs qui pensent comme vous de développer leur propre interface. On y viendra un peu plus tard dans le chapitre 4, mais ça se mérite !

« Bon ok. J'espère juste que je ne vais pas devoir passer des heures à apprendre tout ça, ça ne me donne pas vraiment envie... »

Comment ça tu ne veux pas dédier du temps pour apprendre à utiliser ce formidable outil ?!? Je plaisante, je comprends ton désarroi et pas d'inquiétude, nous allons voir ensemble les éléments de bases et nous passerons vite à quelque chose de plus digérable. Ce passage sert surtout à comprendre ce qu'il y a à la base, et que si un jour vous développez un outil qui fait appel à Git, ça sera par ces commandes qu'il faudra passer !

Revenons-en à l'essentiel :

Cette console est celle utilisée par défaut par Git pour utiliser des lignes de commande.

En s'installant, Git a créé ce qu'on appelle une « variable d'environnement » à son nom dans Windows, c'est-à-dire qu'on peut appeler ses fonctionnalités associées depuis n'importe quelle application ou console.

Git nous propose toute une liste de commandes pour commencer à gérer nos projets. On peut les afficher en tapant la commande « Git ».

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~
$ git
usage: git [--version] [--help] [-C <path>] [-c <name>=<value>]
        [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
        [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
        [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
        [--super-prefix=<path>] [--config-env=<name>=<envvar>]
        <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
  restore Restore working tree files
  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
  diff    Show changes between commits, commit and working tree, etc
  grep    Print lines matching a pattern
  log     Show commit logs
  show    Show various types of objects
  status  Show the working tree status

grow, mark and tweak your common history
  branch  List, create, or delete branches
  commit  Record changes to the repository
  merge   Join two or more development histories together
  rebase  Reapply commits on top of another base tip
  reset   Reset current HEAD to the specified state
  switch  Switch branches
  tag     Create, list, delete or verify a tag object signed with GPG

collaborate (see also: git help workflows)
  fetch   Download objects and refs from another repository
  pull    Fetch from and integrate with another repository or a local branch
  push    Update remote refs along with associated objects

'git help -a' and 'git help -g' list available subcommands and some
concept guides. See 'git help <command>' or 'git help <concept>'
to read about a specific subcommand or concept.
See 'git help git' for an overview of the system.

Mathias.REHEISSER@LAP280004853 MINGW64 ~
```

Nous allons voir dans le prochain chapitre le fonctionnement de l'outil et la signification de ces commandes.

3. LE FONCTIONNEMENT DE GIT

Le fonctionnement de Git s’articule autour de deux grandes fonctionnalités :

- La possibilité de sauvegarder au fur et à mesure les modifications de son projet, fichier par fichier, et d’en conserver un historique des modifications, voire de revenir à un état précédent.
- La possibilité de générer différentes versions du projet, aussi appelées « branches », pour travailler sur des évolutions sans compromettre une version fonctionnelle antérieure, et basculer de l’une à l’autre.

3.1. GESTION DES MODIFICATIONS

Pour cette première fonctionnalité, Git part d’un **dépôt** (ou « **repository** » en anglais), qui est le dossier dans lequel se trouve l’ensemble des fichiers composants votre projet.

Ce **dépôt se trouve à un instant T dans une version A.**

Vous allez ensuite effectuer des modifications dans vos fichiers, en ajouter, en supprimer...etc. Lorsque vous estimez votre projet dans un nouvel état prêt à être sauvegarder, vous allez effectuer un « **commit** » (qui pourrait se traduire par « **engagement** » ou « **validation** »).

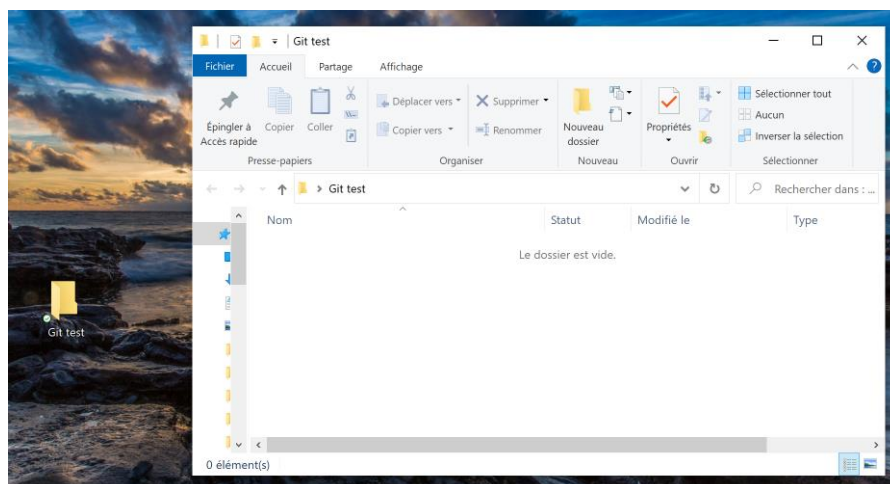
Ce commit permet de sauvegarder votre dépôt et toutes les modifications effectuées.

Vous aurez donc **une version B de votre dépôt, à un instant T+1.**

Git conservant l’ensemble des différentes modifications (grâce notamment à un fichier « .git » qui va venir se glisser dans votre dépôt), vous allez pouvoir si vous le souhaitez, visualiser les modifications entre deux versions, voire revenir à une précédente.

Essayons !

Créons un dossier « Git test » sur notre bureau :



Ouvrons la console Git, et commençons par la commande « cd » suivie du chemin du répertoire pour se placer dedans :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~ (master)
$ cd "C:\Users\mathias.reheisser\OneDrive - ECAM Strasbourg-Europe\Desktop\Git test"

Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ |
```

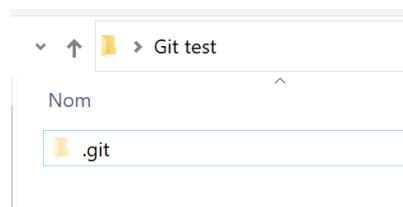
Ok, maintenant utilisons la commande « git init » pour en faire un dépôt git :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~ (master)
$ cd "C:\Users\mathias.reheisser\OneDrive - ECAM Strasbourg-Europe\Desktop\Git test"

Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git init
Initialized empty Git repository in C:/Users/mathias.reheisser/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test/.git/

Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ |
```

On peut voir qu'un dossier « .git » est apparu dans notre dossier de test :



Note : il est possible que ce soit un « dossier caché ». Pensez à activer leur affichage dans les options Windows

Ce dossier sert de référence à Git, on n'a pas à y toucher.

On peut désormais avoir des infos sur notre projet dans le dossier avec la commande « git status » :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git status
On branch master

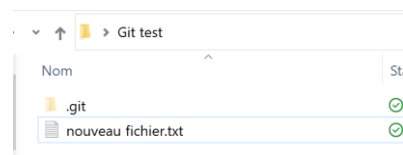
No commits yet

nothing to commit (create/copy files and use "git add" to track)

Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
```

On observe donc qu'on est sur la branche « master », la branche de base, qu'on n'a encore effectué aucun commit, et qu'on n'a aucune modification à commit. Ça fait sens puisque nous n'avons encore rien modifié.

Ajoutons un simple fichier texte dans notre dossier, et réitérons notre dernière commande :



```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git status
On branch master

No commits yet

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

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

Git a bien repéré notre nouveau fichier, mais n'en suit pas l'évolution. Il faut l'ajouter à sa liste des fichier « suivis » ou en anglais « **tracked** ». Pour ça, on a la commande « add ».

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git add nouveau\ fichier.txt

Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$
```

Astuce : utilisez la touche **tabulation** après « add » pour faire défiler les fichiers disponibles.

En suivant l'évolution, on s'aperçoit que le fichier est désormais pris en compte, mais que cet ajout doit encore être « commit » pour être sauvegardé dans l'historique Git.

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git status
On branch master

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   nouveau fichier.txt
```

Effectuons un premier commit. Un commit a besoin d'un titre qu'il faut ajouter en argument de la commande avec l'option « -m » :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git commit -m "premier commit"
[master (root-commit) e7a1f69] premier commit
Committer: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author


1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 nouveau fichier.txt
```

Notre « premier commit » a bien été effectué, on peut voir qu'on a un seul fichier qui a changé : notre fichier texte !

Si on fait un « git status », on est de retour à 0 :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git status
On branch master
nothing to commit, working tree clean
```

Comme notre fichier texte est désormais suivi, il suffit d'y faire la moindre modification pour que Git nous informe des changements :

 nouveau fichier.txt - Bloc-notes

Fichier Edition Format Affichage Aide

On rajoute cette ligne|

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git status
On branch master
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   nouveau fichier.txt

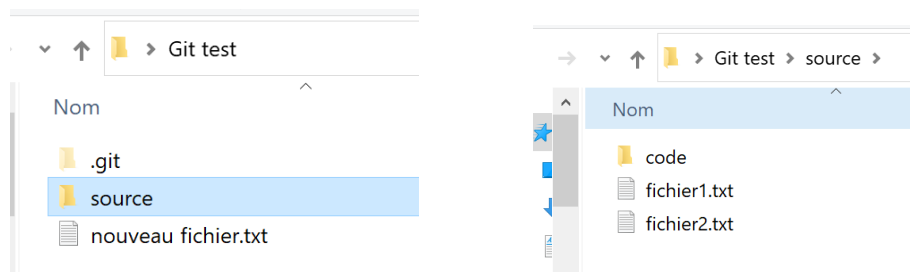
no changes added to commit (use "git add" and/or "git commit -a")
```

Git nous permet même d'avoir des infos sur les lignes qui ont été modifiées avec la commande « git diff *nom du fichier* » :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git diff "nouveau fichier.txt"
diff --git a/nouveau fichier.txt b/nouveau fichier.txt
index e69de29..40af54b 100644
--- a/nouveau fichier.txt
+++ b/nouveau fichier.txt
@@ -0,0 +1 @@
+On rajoute cette ligne
\ No newline at end of file
```

Bref, j'espère que vous commencez à comprendre le fonctionnement.

Evidemment, pour ne pas avoir à ajouter chaque fichier manuellement avec la commande « add », il suffit de l'effectuer sur un dossier source pour que chaque modification à l'intérieur de ce dossier, y compris l'ajout de fichier, soit prise en compte :



```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git add source
```

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git status
On branch master
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   source/fichier1.txt
    new file:   source/fichier2.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   nouveau fichier.txt
```

Pour tester le retour à une version antérieure, on va effectuer ce nouveau commit :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git commit -m "Ajout du dossier"
[master f2e63f2] Ajout du dossier
  Committer: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 source/fichier1.txt
create mode 100644 source/fichier2.txt
```

« Git log » nous donne l'historique des commits :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git log
commit f2e63f250338964d6f830a208d60a45be0160c76 (HEAD -> master)
Author: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
Date: Thu Mar 17 18:17:36 2022 +0100

    Ajout du dossier

commit e7a1f6958fee9f84caf65ce526e489632c9942d5
Author: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
Date: Thu Mar 17 17:26:39 2022 +0100

    premier commit

Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
```

Pour revenir à la version précédente (revenir au « commit » précédent donc), il convient d'utiliser la commande « revert ». Celle-ci propose plusieurs options qu'on peut afficher en la faisant suivre de « -h ».

Pour un retour simple au commit précédent, aucune option n'est nécessaire. Il faut néanmoins préciser quelle branche on souhaite annuler.

Actuellement nous n'avons pas créé de branche, nous nous situons donc sur la branche principale « master ».

Annulons notre dernier commit !

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git revert master
hint: Waiting for your editor to close the file...
[main 2022-03-18T12:51:45.172Z] window#load: attempt to load window (id: 1)
[main 2022-03-18T12:51:45.239Z] update#setState idle
[main 2022-03-18T12:51:45.330Z] ExtensionHostStarterWorker created
[main 2022-03-18T12:51:46.747Z] window#load: window reported ready (id: 1)
[main 2022-03-18T12:51:47.180Z] Starting extension host with pid 18884 (fork()) took 18 ms.
[main 2022-03-18T12:51:47.180Z] ExtensionHostStarterWorker.start() took 20 ms.
[main 2022-03-18T12:51:51.279Z] Waiting for extension host with pid 18884 to exit.
[main 2022-03-18T12:51:51.279Z] Extension host with pid 18884 exited with code: 0, signal: null.
[master 04f7f39] Revert "Ajout du dossier"
Committer: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

2 files changed, 0 insertions(+), 0 deletions(-)
delete mode 100644 source/fichier1.txt
delete mode 100644 source/fichier2.txt
```

Les logs dans la console nous indiquent que deux fichiers ont été modifiés : ça correspond effectivement aux fichiers ajoutés dans le commit :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git log
commit 04f7f3947843a9576248e310885c35c9fba9b747 (HEAD -> master)
Author: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
Date: Fri Mar 18 13:51:43 2022 +0100

    Revert "Ajout du dossier"

    This reverts commit f2e63f250338964d6f830a208d60a45be0160c76.

commit f2e63f250338964d6f830a208d60a45be0160c76
Author: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
Date: Thu Mar 17 18:17:36 2022 +0100

    Ajout du dossier

commit e7a1f6958fee9f84caf65ce526e489632c9942d5
Author: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
Date: Thu Mar 17 17:26:39 2022 +0100

    premier commit
```

Voilà pour l'essentiel concernant le suivi de l'évolution de nos projets ! À vous de découvrir les quelques sous-fonctionnalités qui vous permettront de gérer au mieux vos développements.

Il est désormais temps de s'intéresser à l'autre grande fonctionnalité : les branches !

3.2. GESTION DES BRANCHES

Imaginez que vous arrivez à un niveau de développement suffisamment stable de votre projet et que vous souhaitez la déployer au public. Aucun problème pour vous, vous créez une copie compilée du projet que vous envoyez vers vos serveurs.

Mais maintenant vous souhaitez travailler sur une fonctionnalité future, vous allez devoir effectuer des commits par-dessus cette version stable. Aie. Ça veut dire perdre cette version, ou alors il faut en faire une copie en local, mais niveau organisation...bof.

Heureusement, Git est là pour nous prêter main forte !

En effet, Git nous permet de créer des « branches » de développement, c'est-à-dire différentes versions indépendantes de votre projet, sur lesquelles vous pouvez effectuer des commits sans impacter les autres branches.

Vous pouvez avoir une branche pour les versions déployées au public, une pour les versions en environnement de test, une pour les développements, une pour des fonctionnalités spécifiques...etc.

Bien évidemment, vous pouvez sauter d'une branche à l'autre, en fusionner deux ensembles, récupérer les versions d'une branche dans une autre...ce qui permet de conserver les éléments dans une structure logique.



Un exemple de workflow entre différentes branches, permettant de suivre l'évolution d'un projet.

Commençons à utiliser des branches !

Notre projet est actuellement dans sa branche de base, aussi appelée branche « master ». En général, la branche master représente une version stable du projet. On va créer une branche de développement à partir de celle-ci.

Pour créer une branch, rien de plus simple, on utilise la commande « **git branch *nom de la nouvelle branche*** » :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git branch "dev"
```

On peut vérifier sa création avec « **git branch --all** » qui affiche toutes les branches :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git branch --all
dev
* master
```

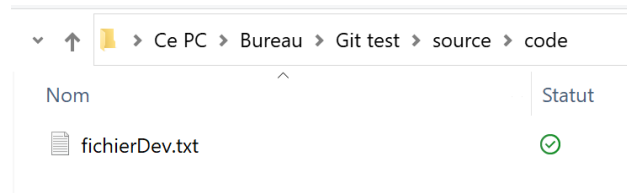
Positionnons-nous sur cette nouvelle branche avec la commande « **checkout** » :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git checkout dev
Switched to branch 'dev'

Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (dev)
$ |
```

On est bien passé sur la branche de développement, comme le précise l'indication entre parenthèses.

Créons un fichier dans cette branche :



« Git status » nous indique que le fichier a bien été créé :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (dev)
$ git status
On branch dev
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    new file:   source/code/fichierDev.txt
```

Nous pouvons faire un nouveau commit !

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (dev)
$ git commit -m "Commit de dev 01"
[dev 1916c9f] Commit de dev 01
  Committer: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
  Your name and email address were configured automatically based
  on your username and hostname. Please check that they are accurate.
  You can suppress this message by setting them explicitly. Run the
  following command and follow the instructions in your editor to edit
  your configuration file:
    git config --global --edit
  After doing this, you may fix the identity used for this commit with:
    git commit --amend --reset-author
  1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 source/code/fichierDev.txt

Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (dev)
$ git log
commit 1916c9f0740923d2f1e33ecec1d2cb5607a4027f (HEAD -> dev)
Author: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
Date:   Fri Mar 18 16:12:43 2022 +0100

    Commit de dev 01

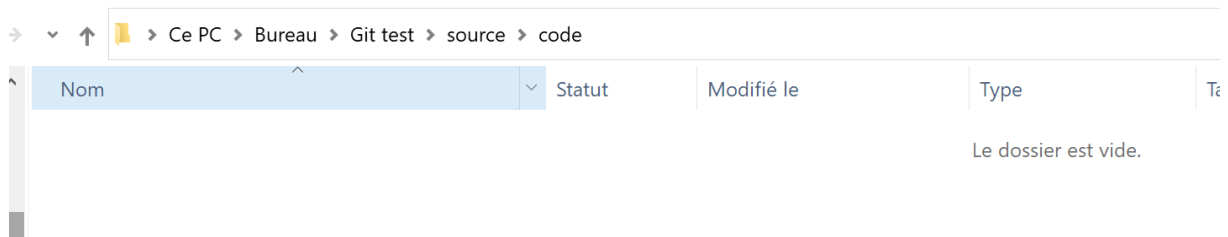
commit 6efbd8074bdb1574534ff9313405ae4966028f35 (master)
Author: Mathias REHEISSER <Mathias.REHEISSER@ECAM-STR.local>
Date:   Fri Mar 18 16:03:22 2022 +0100

    Premier Commit
```

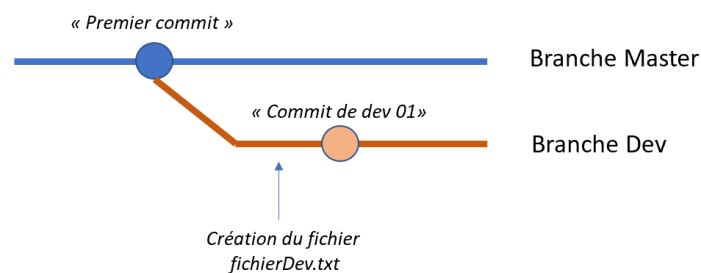
Dans les logs, on voit un premier commit effectué dans la branche master avant la création de la branche dev.

Si vous repassez dans la branche master, vous allez remarquer quelque chose au niveau du fichier qu'on vient de créer :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (dev)
$ git checkout master
Switched to branch 'master'
```

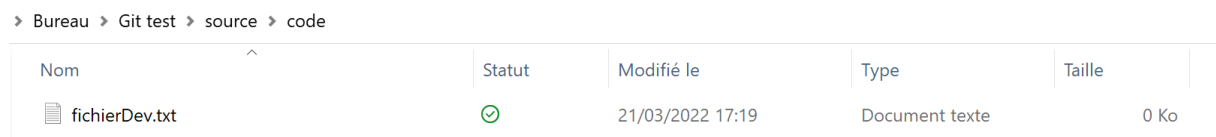


Le dossier est vide, c'est normal ! Le fichier a été créé dans la branche de dev, pas dans la branche master.



Git est en mesure de modifier les fichiers pour les passer d'une version A à une version B et inversement. En revenant dans la branche dev, le fichier réapparaît :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git checkout dev
Switched to branch 'dev'
```



C'est là tout l'intérêt des versions : on peut conserver une ou plusieurs versions stables d'un projet tout en travaillant sur de nouvelles fonctionnalités.

Admettons maintenant que vous avez travaillé sur la branche « dev », que vous estimiez avoir terminé les fonctionnalités souhaitées pour le prochain sprint et que vous avez une version stable. Vous aimeriez faire basculer ces nouveautés sur la version Master.

On appelle ça faire un « merge » : votre branche Master passe dans une nouvelle version avec les modifications apportées dans une autre branche.

Mergeons notre version avec le nouveau fichier dans la branche Master.

Pour ça, rien de plus simple : placez vous dans la branche Master, et utilisez la commande « git merge <name> », <name> étant le nom de la branche à merger :

```
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (dev)
$ git checkout master
Switched to branch 'master'

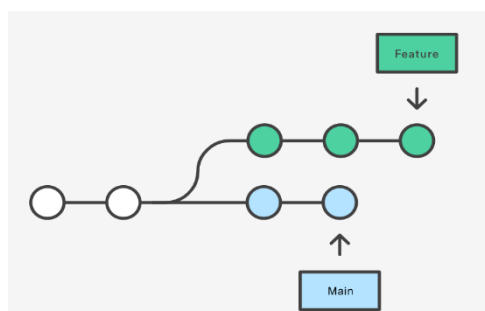
Mathias.REHEISSER@LAP280004853 MINGW64 ~/OneDrive - ECAM Strasbourg-Europe/Desktop/Git test (master)
$ git merge dev
Updating 6efbd80..1916c9f
Fast-forward
 source/code/fichierDev.txt | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 source/code/fichierDev.txt
```

Les logs nous indiquent qu'un fichier a bien été ajouté, et effectivement on le retrouve dans nos dossiers :

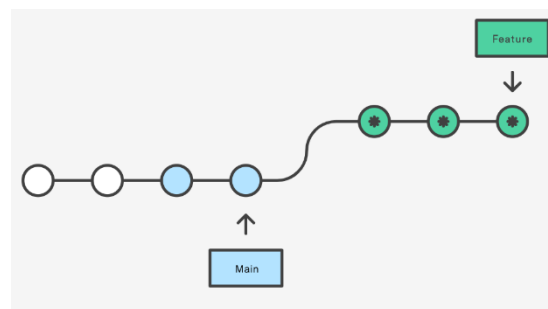
C > Bureau > Git test > source > code

Nom	Statut	Modifié le	Type	Taille
fichierDev.txt	✓	21/03/2022 17:37	Document texte	0 Ko

Voilà pour les branches ! Pour aller plus loin, vous pouvez aller consulter d'autres commandes, en particulier la commande « rebase », qui permet à une branche fille de récupérer les commits effectués sur la branche mère après sa création et ainsi de placer sa création sur la version la plus à jour de la branche mère :



Avant rebase



Après rebase

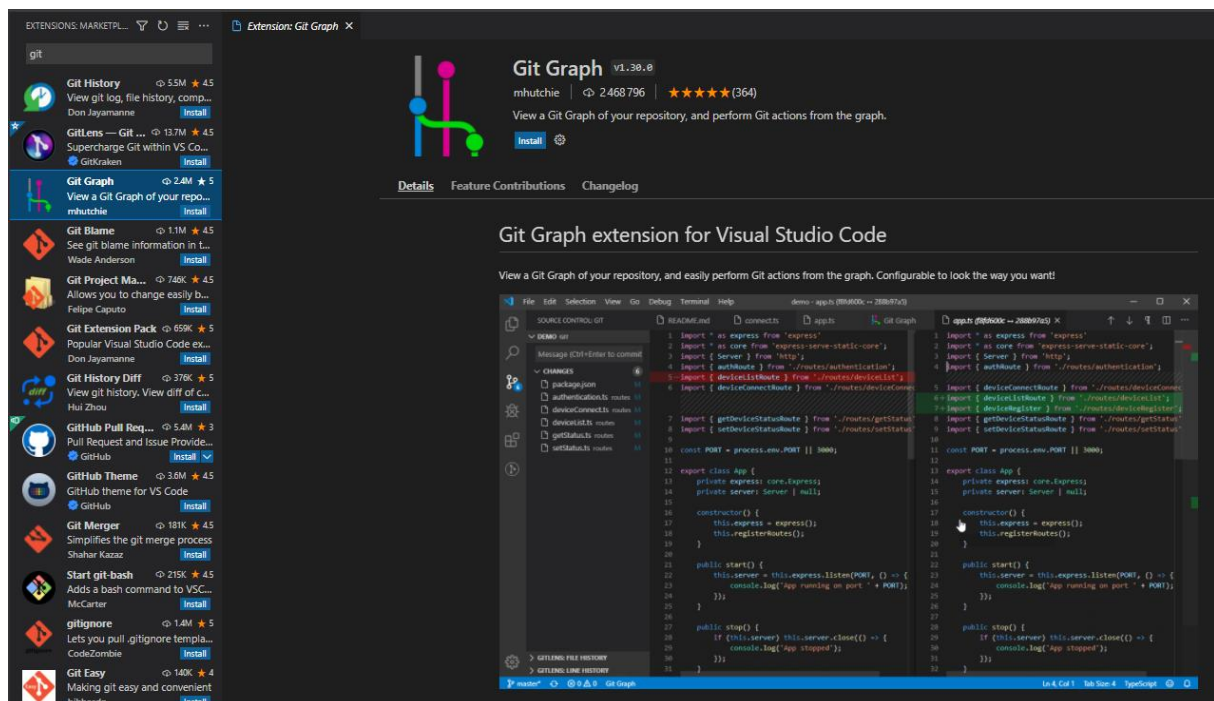
On a ainsi un historique plus propre et on anticipe certains conflits de fichiers potentiels.

4. INTERFACE GRAPHIQUE

Bon, maintenant que vous avez bien travaillé pour comprendre le fonctionnement de Git en lignes de commande, je vous propose de vous rendre le travail plus agréable avec l'utilisation d'outils graphiques qui permettent de rapidement visualiser les branches et les modifications.

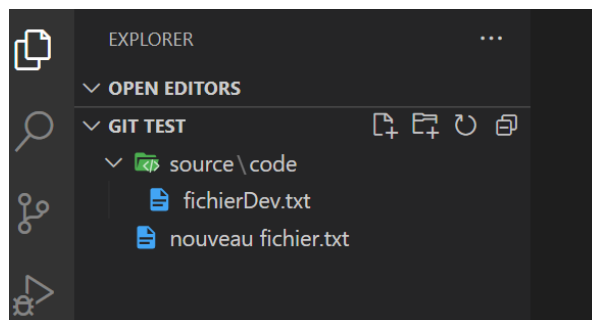
4.1. INSTALLATION DE GIT GRAPH ET PRISE EN MAIN

Il existe de nombreux outils graphiques pour Git. Aujourd'hui je vous propose d'utiliser l'extension « Git Graph » de Visual Studio Code :



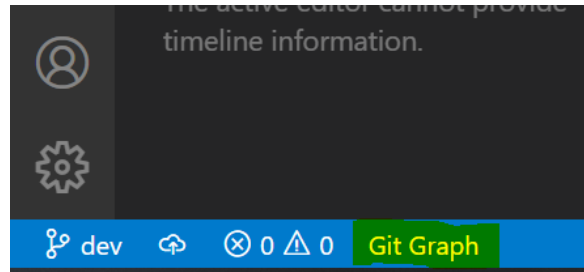
Cliquez sur « Install » et vous aurez l'extension.

Ouvrez votre dossier de projet pour afficher nos fichiers :

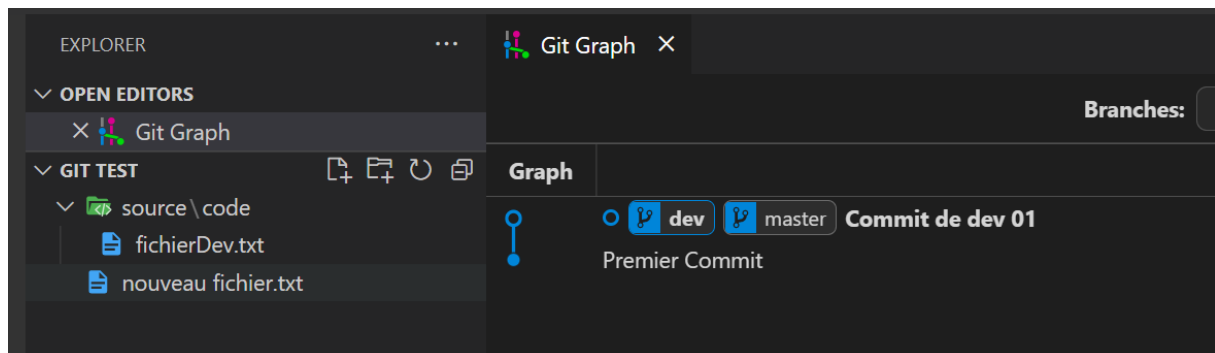


Vous remarquez que le dossier « .git » n'apparaît pas et c'est normal car c'est un « fichier caché » : il ne faudrait pas qu'il soit corrompu par une mauvaise manipulation. On perdrait tout historique du projet, ainsi que nos branches.

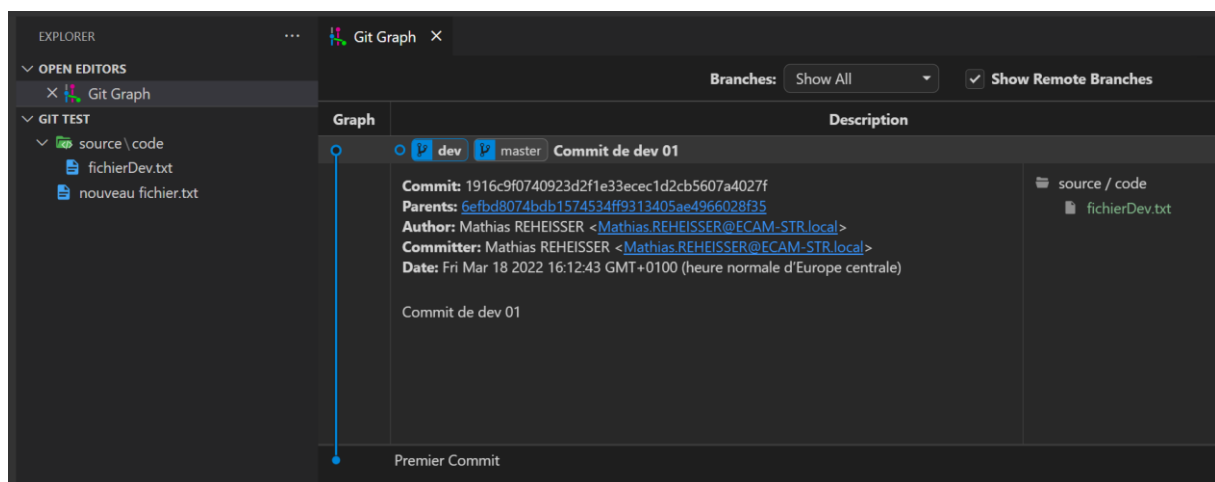
Pour ouvrir Git Graph, cliquez sur le bouton en bas à gauche de l'écran VSC :



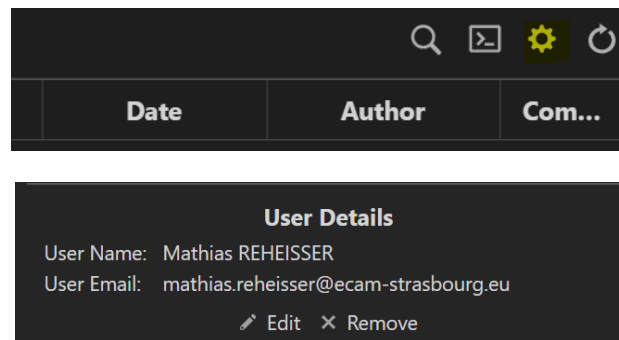
Vous avez l'écran Git Graph qui apparaît, et on peut déjà observer notre branche « Master » sur laquelle est venue se merger la branche de dev, et l'ensemble des commits associés :



En cliquant sur un commit, on obtient davantage d'informations, comme l'auteur du commit, l'identifiant de la version de la branche parente, la date, le message, et surtout : les fichiers qui ont été modifiés.



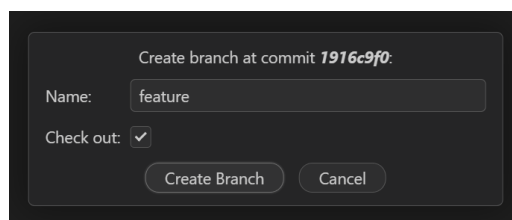
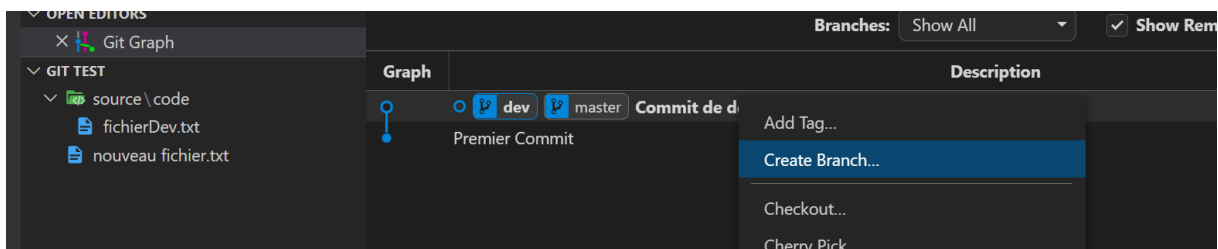
Git Graph a aussi besoin de connaître notre nom et notre mail. Allez dans les paramètres, et ajoutez vos informations :



Vous voilà prêts à travailler avec Git Graph !

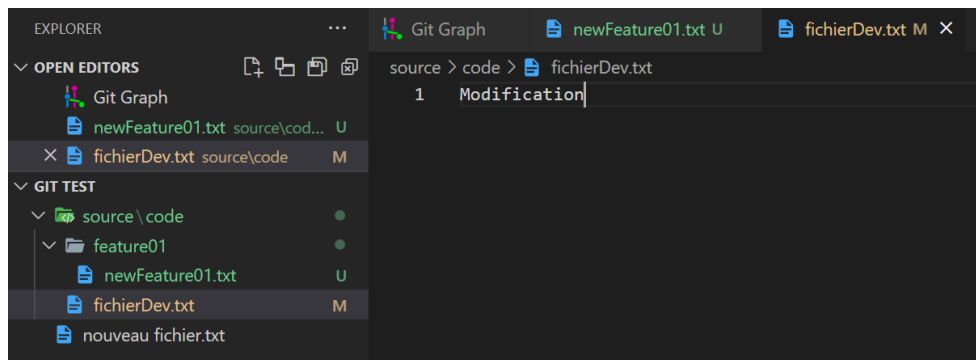
4.2. COMMANDES DE BASE

Créons une branche pour une nouvelle fonctionnalité. Appelons la « feature ». Sur Git Graph, il suffit de faire un clic droit, puis « Create Branch » :



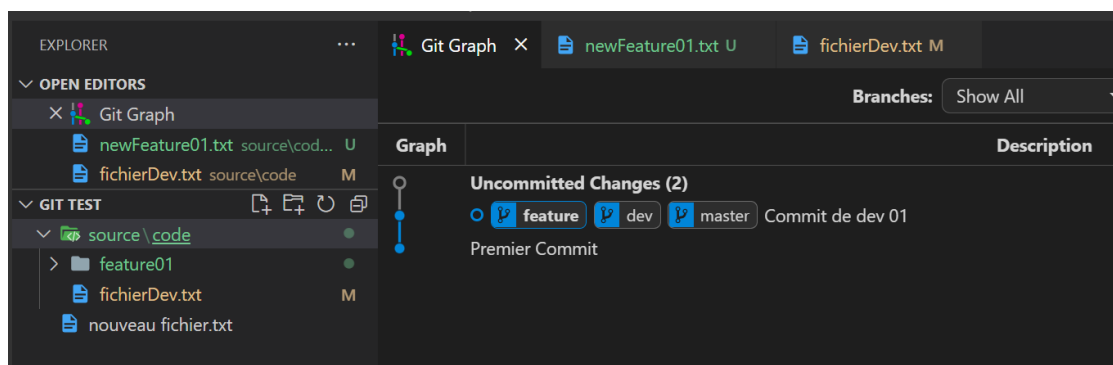
« Check Out » permet de se placer sur la branche à sa création.

Travaillons notre code en rajoutant un dossier pour la nouvelle fonctionnalité et modifions notre fichier « fichierDev.txt » :

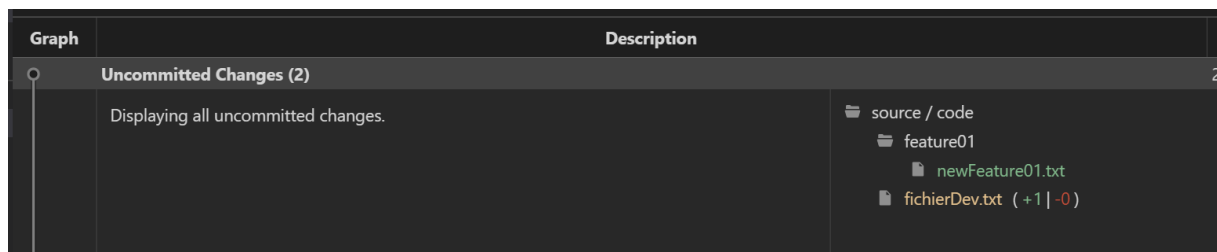


Grâce à Git Graph, vous pouvez voir les fichiers et dossiers qui ont été ajoutés et modifiés depuis le dernier commit directement dans l'arborescence.

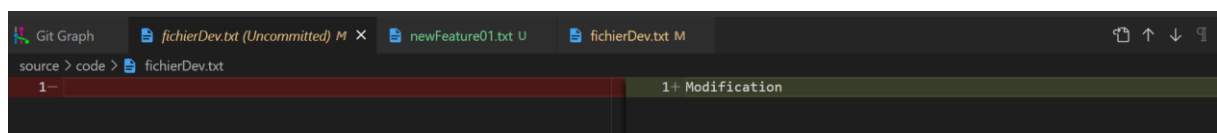
Retournons voir l'interface Git Graph :



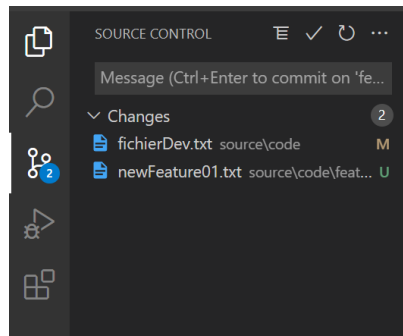
On peut voir que la branche a été créée et qu'il y a des changements en cours sur la branche qui n'ont pas été commit. En cliquant dessus, on peut voir les fichiers concernés :



On peut même voir les lignes modifiées dans les fichiers avec un clic droit, puis « view diff » sur le fichier :



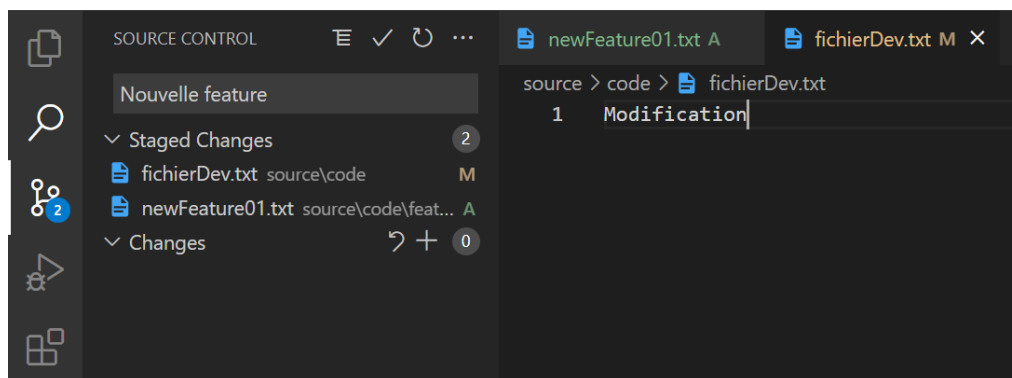
Pour effectuer un commit, il faut aller dans le menu « source control » à gauche de l'écran VSC. Vous y retrouvez la liste des éléments ajoutés ou modifiés.



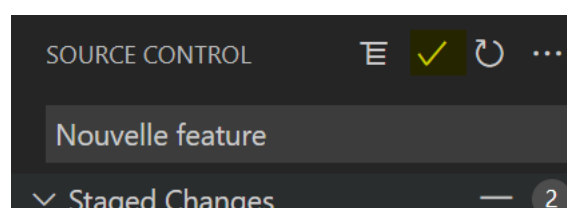
Avant de commit, il est important de valider les changements qu'on souhaite effectuer. C'est une étape intermédiaire facultative qui permet de valider seulement les fichiers qui nous intéressent.

Il est possible qu'on ait, dans le cadre des développements, modifié un fichier pour nous arranger mais qu'il n'est pas nécessaire de commit, ou simplement une modification faite par mégarde. On peut ainsi s'assurer qu'on ne valide que ce qui nous intéresse. On dit qu'on « stage » un fichier. (Oui le franglais est très présent avec Git...)

Pour cela, clic droit sur les fichiers en question, puis « **stage changes** ». Vous pouvez aussi stage l'ensemble des changements avec un clic droit sur le menu « Changes » :

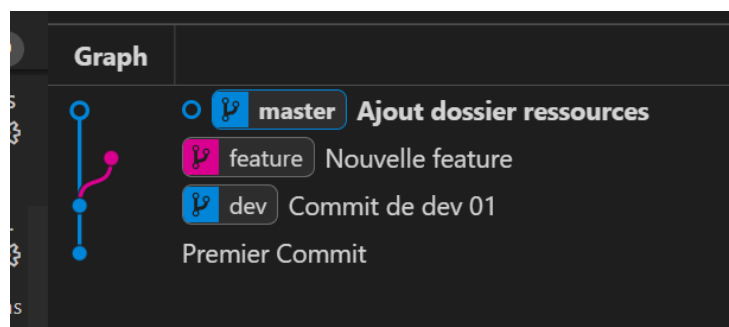
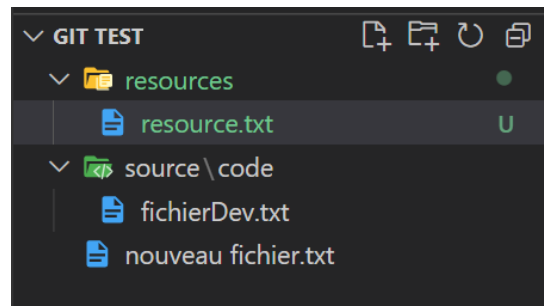


Enfin, Il suffit de mettre un message pour décrire le commit et de cliquer sur « Commit » :

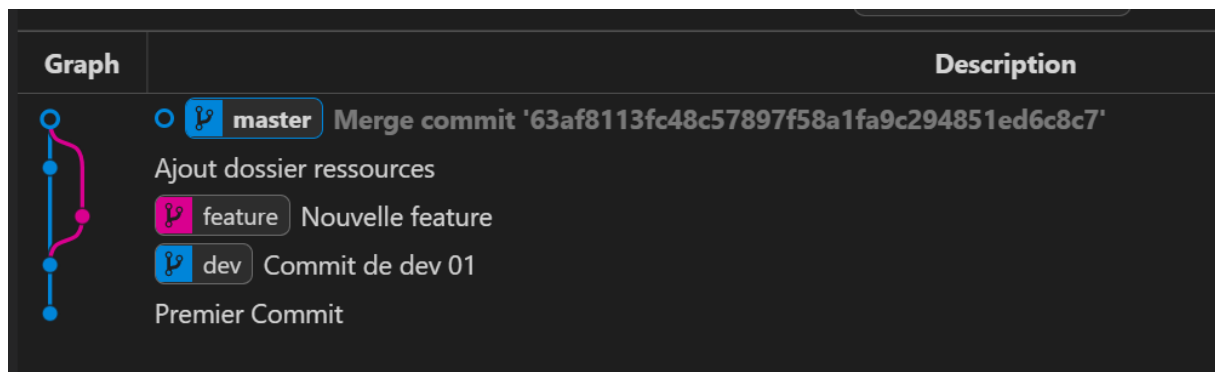


Graph	Description
	feature Nouvelle feature dev master Commit de dev 01 Premier Commit

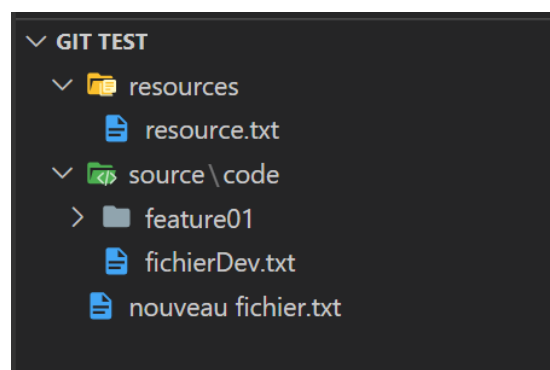
On a bien notre commit qui a été effectué ! Si on revient sur la branche master et qu'on effectue un commit sur celle-ci, on a, comme dans nos schémas, les branches qui se séparent :



Pour merge la branche feature dans la branche master, clic droit sur la branche feature, puis « **merge into current branch** ».



Le merge a bien été effectué, on retrouve notre dossier « feature01 » en plus du dossier « resources » :



4.3. GESTION DES CONFLITS

Dans le sous-chapitre précédent, nous avons effectué un merge. Celui-ci s'est bien passé car il n'y avait pas de fichier modifié en commun.

Mais que se passe-t-il si un fichier a été modifié différemment dans deux branches A et B et qu'on cherche à les merge ensemble ? Il y a un conflit. Git va nous informer qu'il faut traiter le cas pour effectuer le merge correctement.

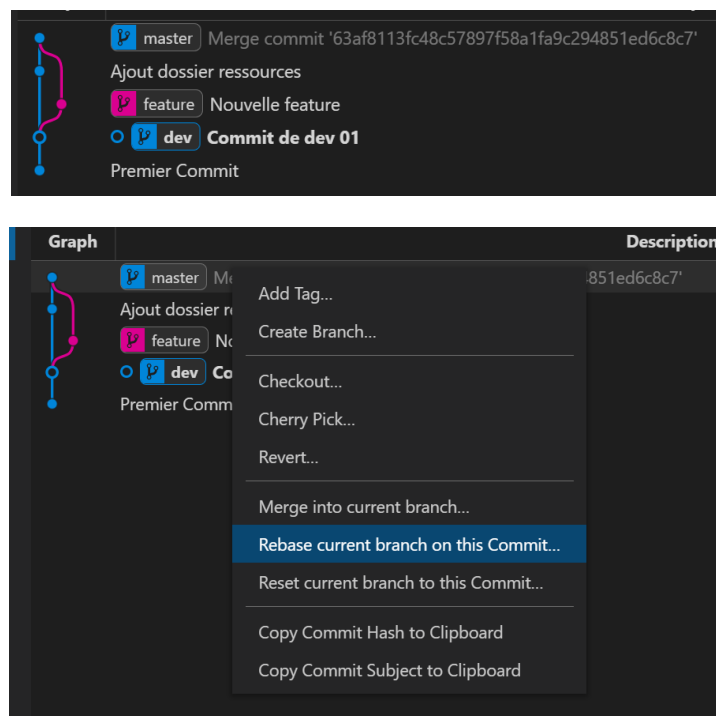
3 possibilités s'offrent à nous :

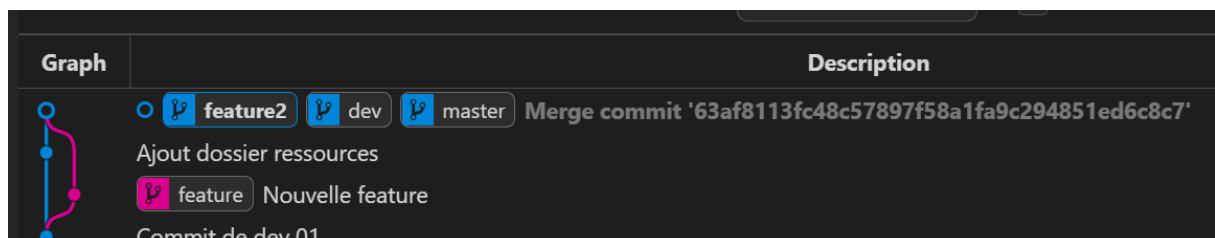
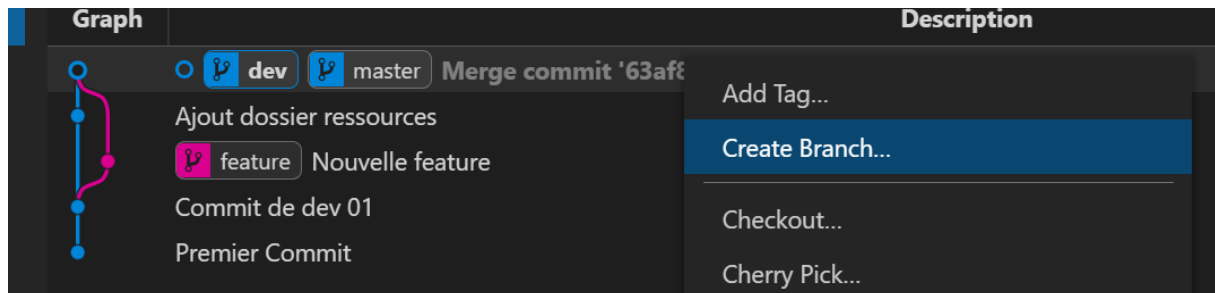
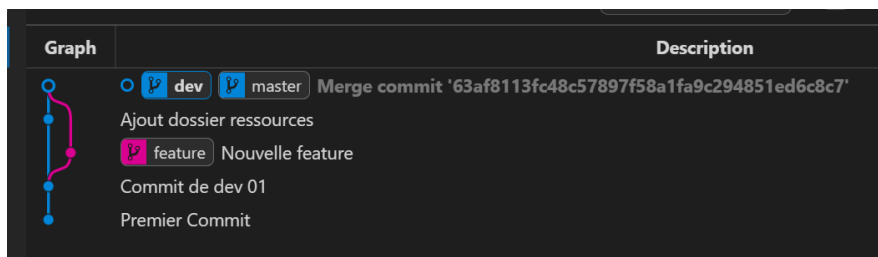
- Soit on garde la version A
- Soit on garde la version B
- Soit on garde les deux
- Soit on sélectionne les lignes à garder une à une

Testons ces possibilités.

Aussi, pour vous habituer à l'utilisation conventionnelle des branches, nous allons travailler avec la branche dev qui sert de référence à l'équipe de développement, et les branches features qui en découlent. La branche master doit rester une base stable qui n'évolue que lorsqu'un ensemble de développements a été effectué.

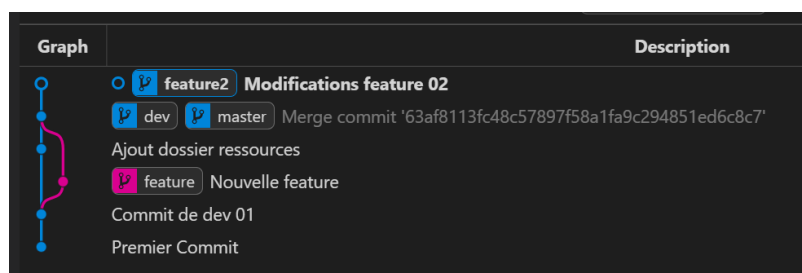
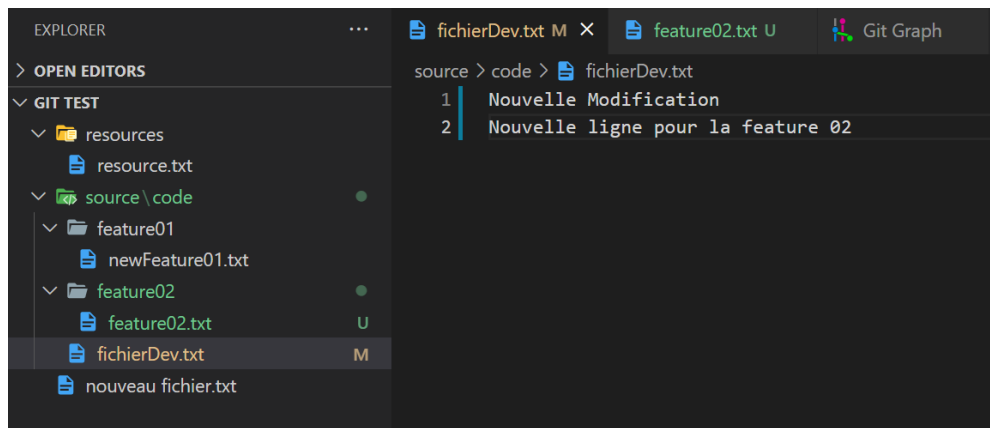
Mettons à jour la branche de dev avec un rebase, et créons à partir de celle-ci une une branche « feature2 » :



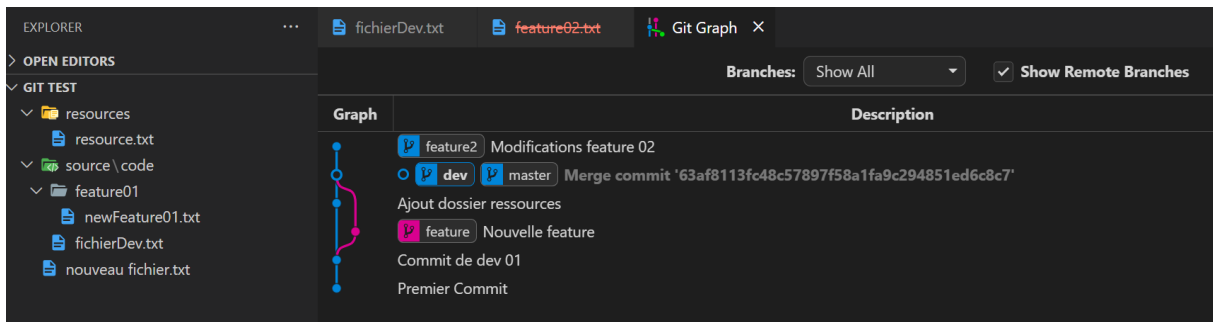


Parfait !

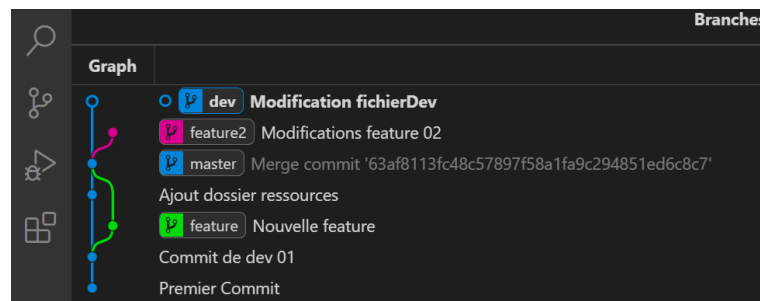
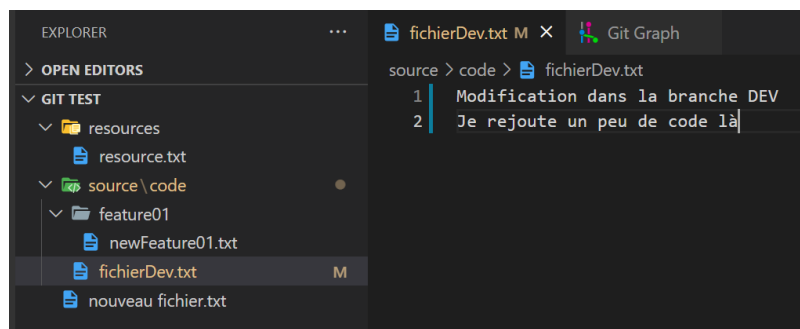
Créons un dossier « feature02 », et modifions légèrement le fichier « fichierDev.txt » :



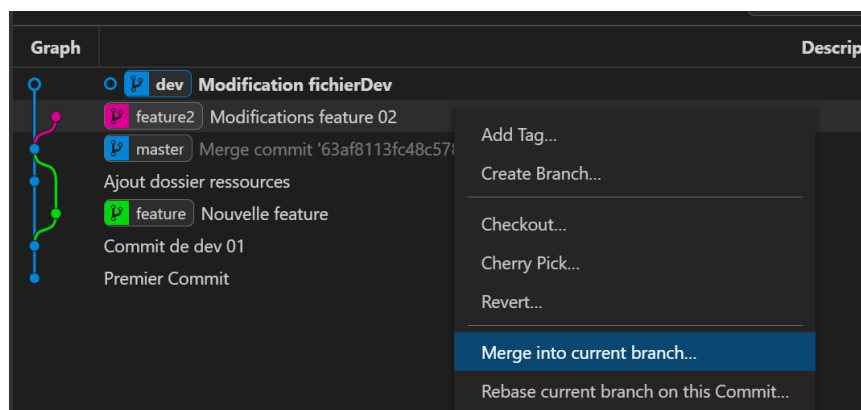
Ok, nous avons effectué notre commit pour la feature numéro 2. Nous allons repasser sur la branche de développements, mais avant de merge, nous allons aussi modifier le fichier « fichierDev.txt » :

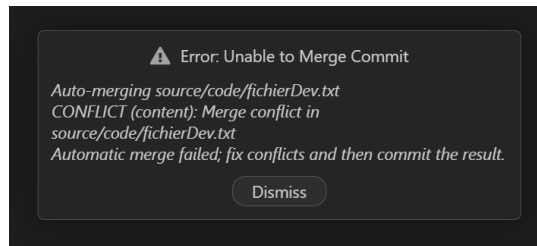


Note : Vous remarquez que le fichier « feature02.txt » est en rouge et barré dans la branche dev, et c'est normal puisque dans cette branche il n'existe pas. Vous pouvez le fermer.



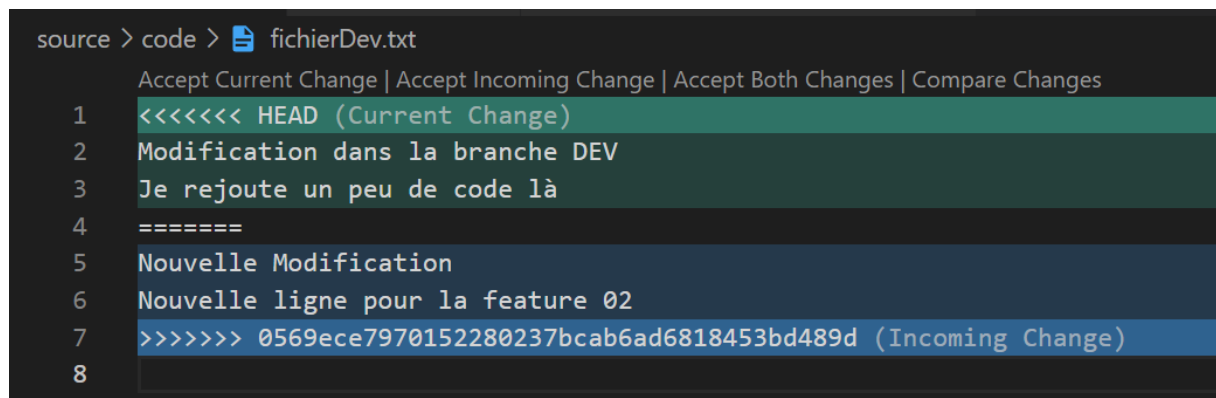
Ok, nous avons donc deux branches, chacune avec sa version du fichier « fichierDev.txt ». Voyons ce qu'il se passe si on souhaite merger la feature dans dev :





Un message d’erreur apparaît. « Le commit de merge n’a pas pu être effectué à cause du fichier « fichierDev.txt ». Il y a un conflit dans le contenu. »

Si on va dans le fichier en question, l’interface nous propose les deux versions :

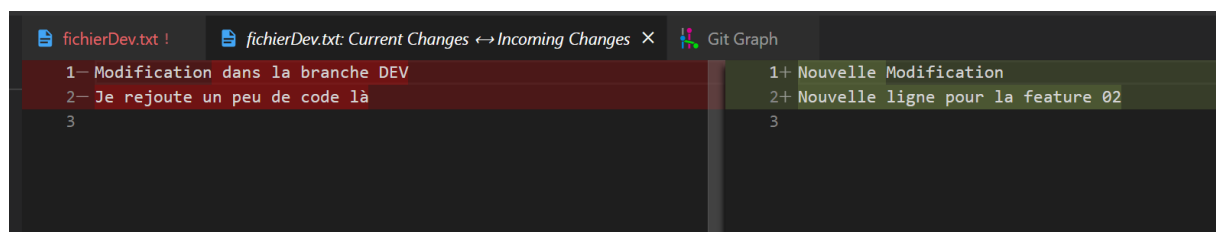


En vert, la version de la branche de base (ici dev), et en bleu, la version de la branche qui vient se merger (ici feature2). On peut alors retravailler le contenu pour résoudre le conflit.

De plus, 3 possibilités de résolution du conflit sont proposées :

- « Accept Current Change » : On garde la version Dev
- « Accept Incoming Change » : On garde la version Feature2
- « Accept Both Changes » : On garde l’ensemble, soit 4 lignes au final
- On a aussi « Compare Changes » qui permet de comparer les lignes une à une

Les trois premières possibilités offrent une résolution immédiate, nous allons donc nous pencher sur la quatrième. Celle-ci nous offre une vision plus détaillée des conflits, ligne par ligne :



L’idée est alors de réfléchir à quelle version garder, et si plusieurs personnes ont travaillé dessus de discuter sur les modifications en question.

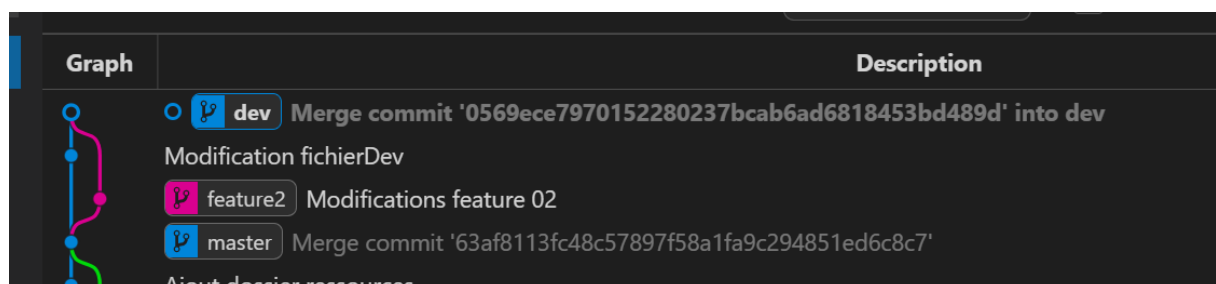
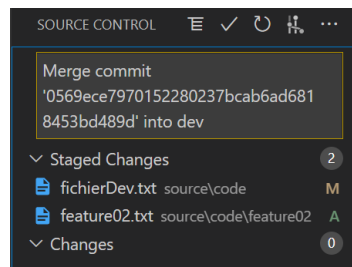
Ici, la décision est la suivante : la ligne 1 doit être celle de dev, et la ligne 2 celle de feature2. On peut donc revenir dans le fichier et faire nos modifications.

```
source > code > fichierDev.txt
Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes
1 <<<<<< HEAD (Current Change)
2 Modification dans la branche DEV
3 Je rejoute un peu de code là
4 =====
5 Nouvelle Modification
6 Nouvelle ligne pour la feature 02
7 >>>>>> 0569ece7970152280237bcab6ad6818453bd489d (Incoming Change)
8
```



```
fichierDev.txt ! x fichierDev.txt: Current Changes ↔ Incoming Changes Git Graph
source > code > fichierDev.txt
1 Modification dans la branche DEV
2 Nouvelle ligne pour la feature 02
```

On retente le commit :



Notre commit a finalement bien été effectué, et le conflit a été résolu !

Note : Dans l'idéal, on essaie d'éviter au maximum les conflits de fichiers quand on développe un projet. Bien évidemment, la réalité fait que ceux-ci arrivent toujours. Il s'agit donc de les limiter. Pour cela, deux principes de développement sont à respecter (les « bonnes pratiques ») :

- Une architecture de projet claire, définie et séparée en couches applicatives, ainsi que des revues de codes régulières avant chaque merge.
- Une stratégie de développement en amont : savoir comment on va implémenter telle ou telle fonctionnalité, et quel impact ça va avoir sur le reste du projet.

Page laissée vide intentionnellement

5. TRAVAIL COLLABORATIF A DISTANCE AVEC GIT HUB

Nous avons vu que Git permet de gérer son code dans différentes versions, mais tout en local. La question ici est donc de savoir comment plusieurs utilisateurs peuvent travailler sur le même projet en même temps

L'outil en ligne GitHub permet d'avoir un dépôt de travail commun, accessible en ligne !



L'idée est d'avoir votre dépôt (le fameux « repo », retournez au début du cours si la notion vous est déjà floue) en ligne, et chaque collaborateur peut en cloner une copie pour travailler dessus. Certains créateurs y mettent même leur projet en open-source et disponible au public !

Le responsable du dépôt peut configurer qui a accès au dépôt, qui peut en cloner une copie, publier dessus...etc.

Utilisé dans une équipe, plusieurs collaborateurs peuvent en extraire une branche et avoir accès à l'ensemble des différentes versions. Parfait pour la gestion d'un projet !

Ça fait rêver non ? Alors c'est parti !

5.1. FONCTIONNEMENT D'UN DEPOT EN LIGNE

Avant de pratiquer, un peu de théorie s'impose.

La première chose à savoir, c'est que le dépôt en ligne (dépôt « central », « distant » ou dépôt « origin ») ne fonctionne pas comme un dépôt local : vous ne pouvez pas travailler directement dessus, y créer des branches, passer d'une version à l'autre.

En effet, ce dépôt central sera dans une version, et si chacun commence à développer dessus, à se positionner dans des branches, les fichiers changeront constamment et il ne sera pas possible de travailler (imaginez que vous travaillez dans votre branche et sans prévenir votre collègue passe le dépôt dans la version de sa propre branche...vous risquez de perdre votre travail, sans compter les conflits...).

Pour résoudre cette problématique, voici la stratégie proposée par Git Hub :

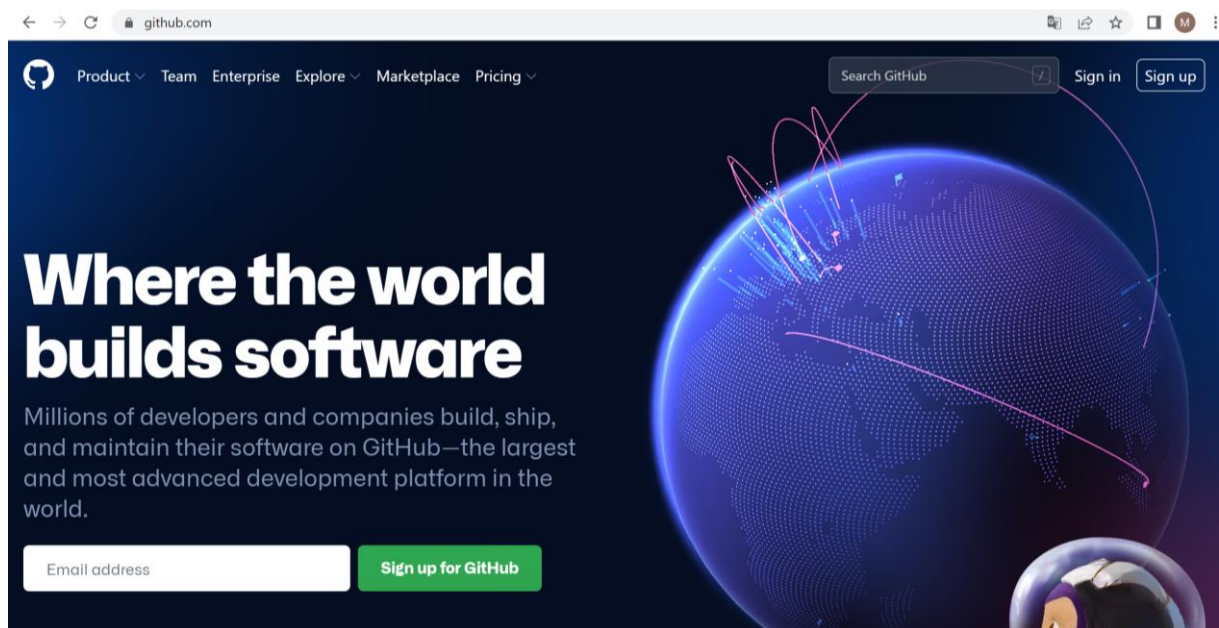
- Vous avez un dépôt git central sur Git Hub qui sert de **référence** à l'ensemble des équipes.
- Si vous souhaitez travailler sur le projet, vous allez devoir **copier** ce dépôt en local (on utilise plutôt le terme « **cloner** »).
- Vous travaillez en local comme d'habitude, rien de change à ce niveau-là, **mais les modifications ne sont effectives qu'en local, et donc pas effectives dans le dépôt central.**
- Lorsque vous avez terminé d'effectuer vos modifications sur votre branche en local, vous allez « **pousser** » la nouvelle version de votre branche sur le dépôt central en ligne pour la publier.

On a donc des nouvelles commandes à utiliser :

- **Clone** : pour copier un dépôt distant.
- **Fetch** : pour voir (sans les télécharger dans son dépôt local) les modifications effectuées sur le dépôt distant.
- **Pull** : pour télécharger la dernière version d'une branche distante en local.
- **Push** : pour charger nos modifications locales sur le dépôt distant.

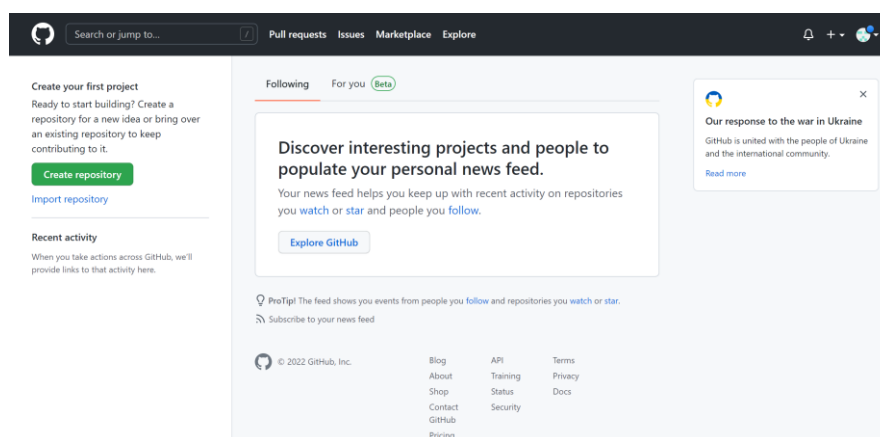
Bon, assez parlé ! Pratiquons un peu !

Rendez-vous sur le site github : <https://github.com/>



Note : Le site GitHub est une pépite de design numérique. N'hésitez pas à vous y balader ! L'animation du globe terrestre par exemple peut paraître seulement esthétique, mais tous les points et courbes que vous y voyez sont des opérations effectuées en temps réel sur les dépôts publics. Passez votre souris dessus pour vous en rendre compte.

Créez-vous un compte et connectez-vous pour arriver sur la page d'accueil.



GitHub propose en réalité beaucoup plus que du simple travail collaboratif. On peut y ajouter le suivi des évolutions, des projets open-sources, proposer des améliorations ou relever des problèmes sur des projets publics, un accès à un marketplace d'extension...etc.

Aujourd'hui, nous allons nous concentrer sur notre dépôt commun. Cliquez à gauche sur « Create Repository » :

The screenshot shows the 'Create a new repository' page on GitHub. At the top, it says 'Create a new repository' and 'A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)'. Below this, there are two input fields: 'Owner' with a dropdown menu showing 'MReheisser' and 'Repository name' with an empty text box. A hint below says 'Great repository names are short and memorable. Need inspiration? How about [furry-octo-fortnight?](#)'. There is a 'Description (optional)' text area. Below that, there are two radio buttons for visibility: 'Public' (selected) and 'Private'. The 'Public' option says 'Anyone on the internet can see this repository. You choose who can commit.' The 'Private' option says 'You choose who can see and commit to this repository.' Below this, there is a section 'Initialize this repository with:' with the instruction 'Skip this step if you're importing an existing repository.' There are three checkboxes: 'Add a README file' (with a note 'This is where you can write a long description for your project. [Learn more.](#)'), 'Add .gitignore' (with a note 'Choose which files not to track from a list of templates. [Learn more.](#)'), and 'Choose a license' (with a note 'A license tells others what they can and can't do with your code. [Learn more.](#)'). At the bottom is a green 'Create repository' button.

Pour créer votre dépôt, vous devez entrer :

- Un propriétaire
- Un nom de dépôt (par convention, en minuscule et mots séparés par des tirets)
- Une description de votre projet
- Une visibilité : private ne sera visible que par vous et votre équipe, tandis que public sera visible par tous
- Quelques options si souhaités (nous n'allons pas en inclure ici) : un fichier « Readme » pour expliquer votre projet, un fichier « .gitignore » pour des fichiers statiques qui ne sont jamais modifiés, une licence pour limiter l'exploitation du public...etc.

Puis cliquez sur « create repository ».

The screenshot shows the GitHub repository page for 'MReheisser / mon-super-projet'. The repository is marked as 'Private'. At the top, there are links for 'Pull requests', 'Issues', 'Marketplace', and 'Explore'. Below the repository name, there are buttons for 'Unwatch', 'Fork', and 'Star'. The main navigation bar includes 'Code', 'Issues', 'Pull requests', 'Actions', 'Projects', 'Security', 'Insights', and 'Settings'. The 'Code' tab is selected. Below the navigation bar, there is a 'Quick setup' section with the text 'Quick setup — if you've done this kind of thing before'. It offers three options: 'Set up in Desktop', 'HTTPS', and 'SSH'. The 'SSH' option is selected, showing the URL 'https://github.com/MReheisser/mon-super-projet.git'. Below this, there is a section titled '...or create a new repository on the command line' with a code block containing the following commands:

```
echo "# mon-super-projet" >> README.md
git init
git add README.md
git commit -m "first commit"
```

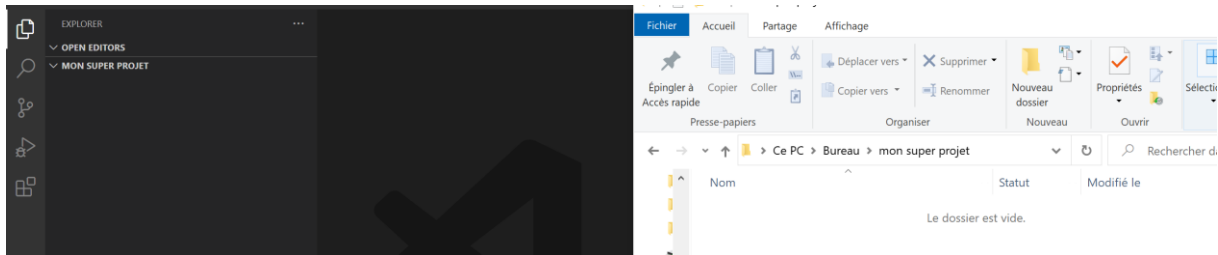
Cours annexe 2 – Gérer son code avec Git

Votre dépôt en ligne est créé ! Bon, il est vide, il va falloir y ajouter quelques éléments.

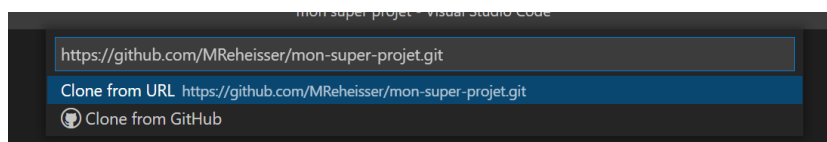
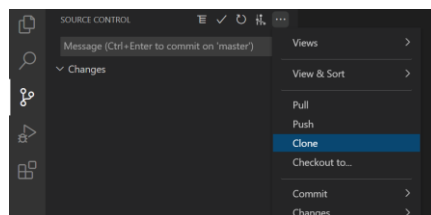
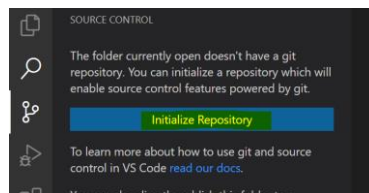
Comme dit précédemment, il faut que je clone ce dépôt sur mon poste en local. Pour cela, copiez le lien https proposé :



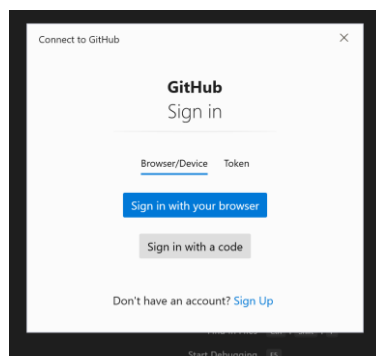
Allez maintenant dans Visual Studio Code, et ouvrez un dossier dans lequel vous allez cloner votre dépôt distant :



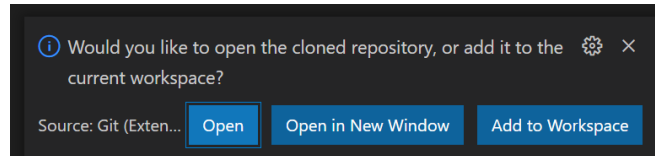
Allez dans le menu source control, initialisez un dépôt git, puis allez dans le menu explorer, puis « clone » et collez le lien obtenu précédemment dans Git :



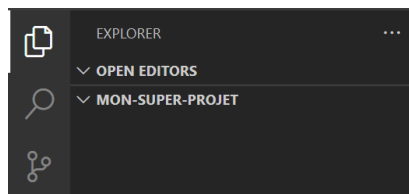
Pour un dépôt privé, une fenêtre GitHub vous demandera sûrement de vous identifier :



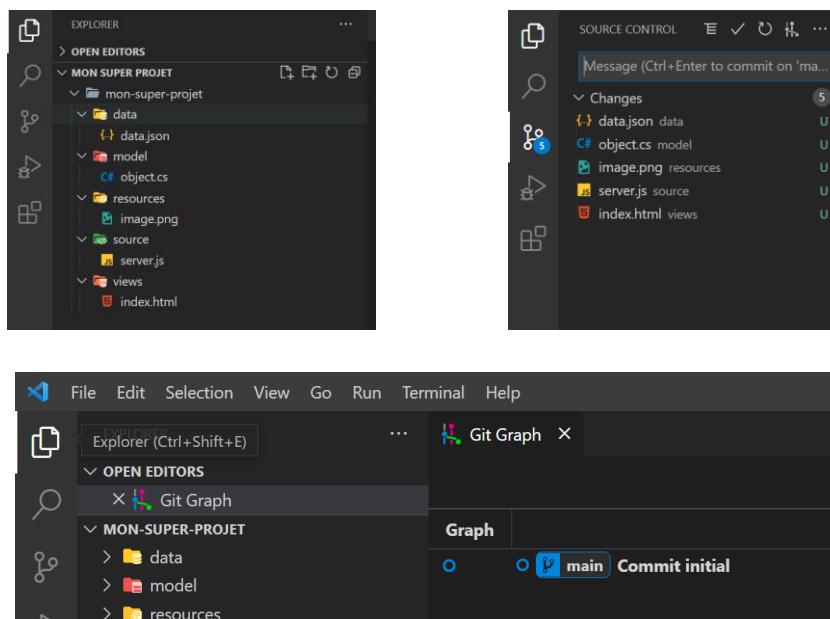
Identifiez-vous par la méthode de votre choix (le navigateur étant la solution la plus simple), et votre projet apparaîtra ! Une boîte de dialogue va s'ouvrir, cliquez sur « open » pour ouvrir le dossier cloné. Sinon, ouvrez le depuis « File > Open folder... ».



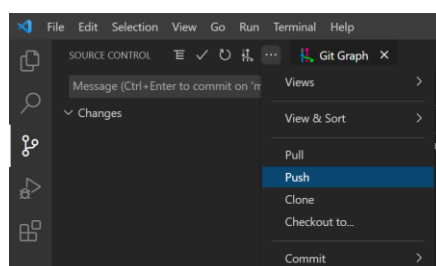
Vous vous retrouvez avec le clone du dossier en ligne :



Bon certes, il est vide. Remédions à ça. Ajoutez une base de fichiers à votre projet, puis effectuez un premier commit :



Votre commit est effectué en local, mais il faut maintenant le « pousser » sur le dépôt en ligne. Dans le menu de « source control », cliquez sur « push » :



Retournez sur la page de votre projet GitHub et hop !

The screenshot shows the GitHub interface for a repository named 'mon-super-projet' by user 'MReheisser'. The repository is marked as 'Private'. The navigation bar includes links for Code, Issues, Pull requests, Actions, Projects, Security, Insights, and Settings. Below the navigation bar, there are buttons for 'main' branch, '1 branch', and '0 tags', along with 'Go to file', 'Add file', and a green 'Code' button. The main content area displays a commit history table:

MReheisser Commit initial		6cf9537 18 hours ago	🕒 1 commit
data	Commit initial	18 hours ago	
model	Commit initial	18 hours ago	
resources	Commit initial	18 hours ago	
source	Commit initial	18 hours ago	
views	Commit initial	18 hours ago	

Votre commit a bien été push ! Youpi !

Vous êtes désormais capable de récupérer votre travail depuis n'importe quel accès à internet.

Il nous reste encore deux choses à voir :

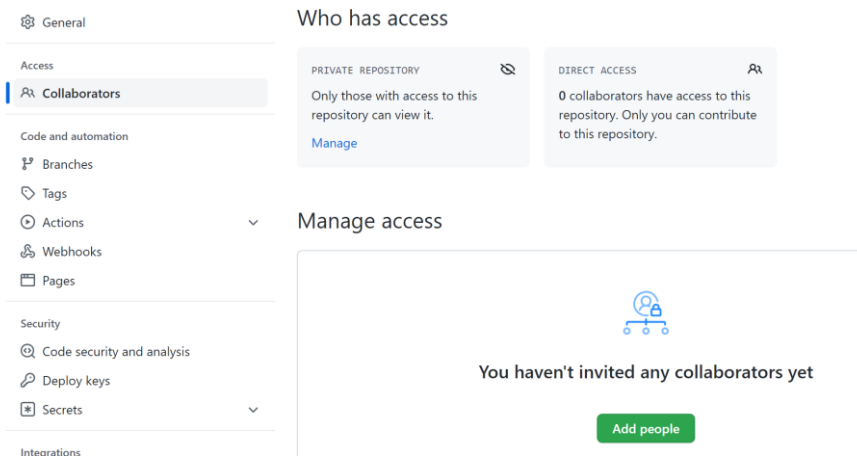
- L'ajout de participants au projet
- La création de branches

5.2. AJOUTER DES COLLABORATEURS A UN PROJET PRIVE

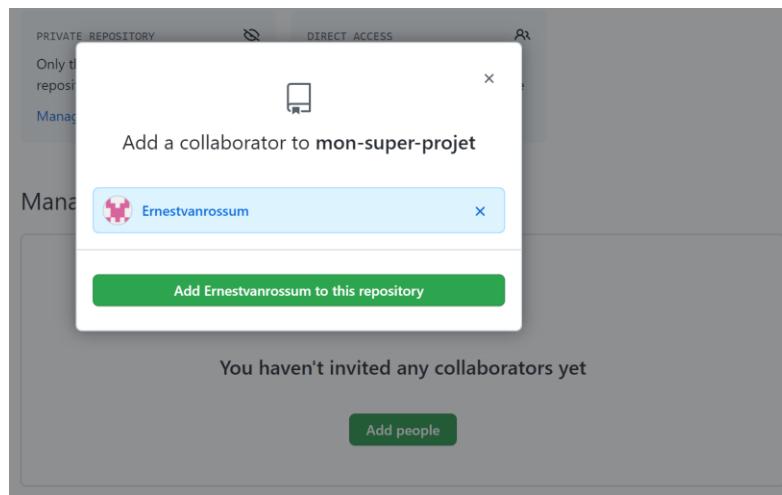
Les membres de votre équipe doivent impérativement disposer d'un compte GitHub pour pouvoir travailler sur le projet.

Pour les ajouter, allez (en tant que créateur du projet) dans « Settings » puis « Collaborators » :

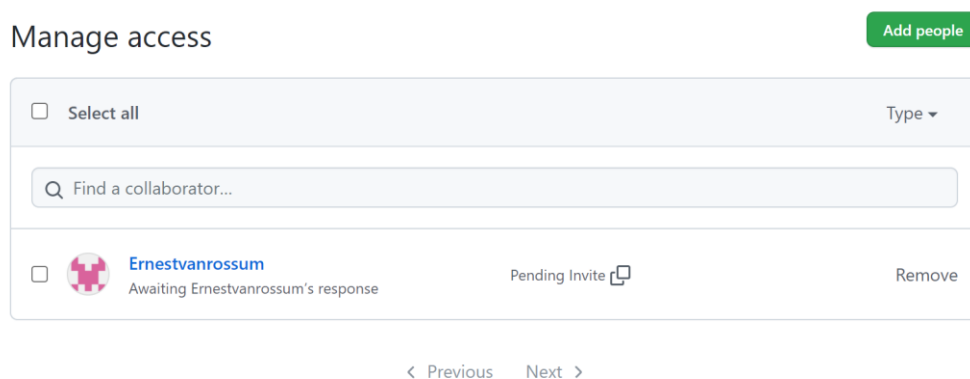
The screenshot shows the 'Settings' page for the repository 'mon-super-projet'. The 'General' tab is selected in the left sidebar. Under the 'Access' section, the 'Collaborators' option is highlighted with a yellow box. The main content area shows the 'General' settings, including the 'Repository name' field which contains 'mon-super-projet' and a 'Rename' button.



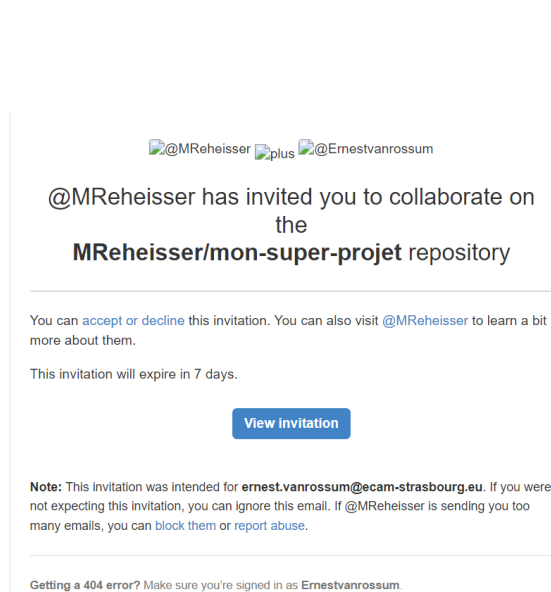
Cliquez simplement sur « Add people » pour rechercher et ajouter un utilisateur :



Manage access



Votre contact devrait recevoir l'invitation par mail, il n'a plus qu'à accepter pour pouvoir lui ou elle aussi travailler sur le projet !



@MReheisser has invited you to collaborate on the **MReheisser/mon-super-projet** repository

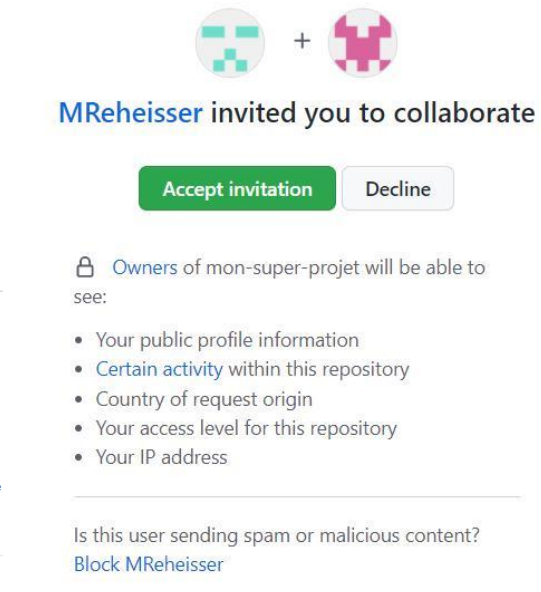
You can [accept](#) or [decline](#) this invitation. You can also visit [@MReheisser](#) to learn a bit more about them.

This invitation will expire in 7 days.

[View invitation](#)

Note: This invitation was intended for [ernest.vanrossum@ecam-strasbourg.eu](#). If you were not expecting this invitation, you can ignore this email. If [@MReheisser](#) is sending you too many emails, you can [block them](#) or [report abuse](#).

Getting a 404 error? Make sure you're signed in as Ernestvanrossum.



MReheisser invited you to collaborate

[Accept invitation](#) [Decline](#)

🔒 Owners of mon-super-projet will be able to see:


- Your public profile information
- [Certain activity](#) within this repository
- Country of request origin
- Your access level for this repository
- Your IP address

Is this user sending spam or malicious content?
[Block MReheisser](#)

Manage access

[Add people](#)

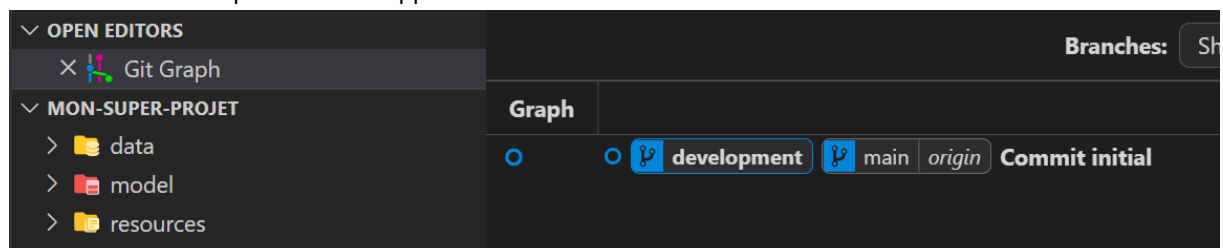
☐ Select all Type ▾

☐  **Ernestvanrossum**
Collaborator Remove

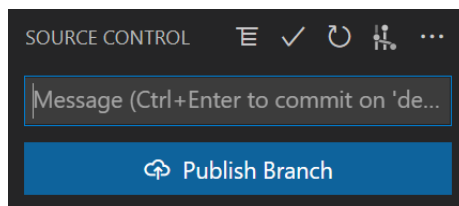
5.3. CREATION DE BRANCHE

La création de branche se fait exactement comme en local, sauf qu'il va falloir la « push » ensuite.

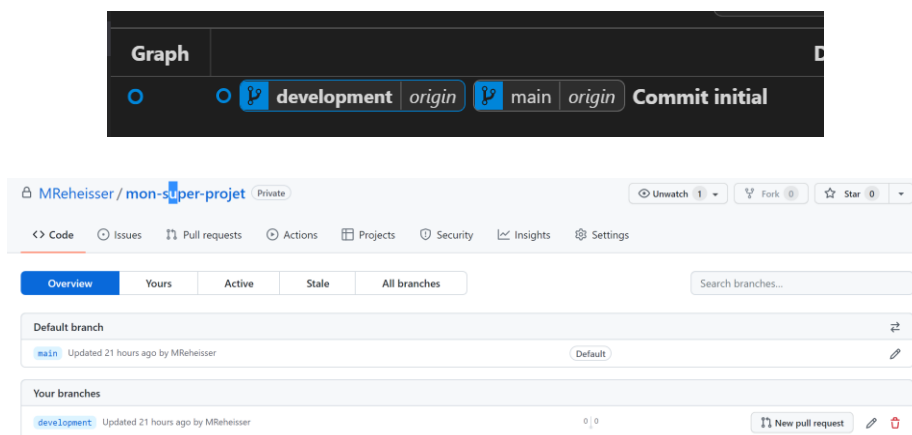
Créons une branche pour les développements :



Il suffit alors d'aller dans le menu de contrôle des sources, et de cliquer sur « Publish branch »



Si la publication s'est bien passée, elle s'affiche comme étant présente sur le dépôt « origin » et on la retrouve dans l'onglet « branch » de GitHub :



Sachez qu'en version payante, GitHub permet de limiter l'accès à certaines branches chez les différents collaborateurs en fonction de leur rôle.

5.4. « FETCH » ET « PULL »

Avant de conclure, laissez-moi vous parler des commandes « fetch » et « pull ». Ces deux commandes sont à connaître pour être efficace dans votre travail collaboratif.

La « fetch », ou « recherche », permet en local d'actualiser les infos concernant le dépôt distant. On obtient par exemple les infos sur les dernières branches créées, les commits qui ont été push, les versions...etc.

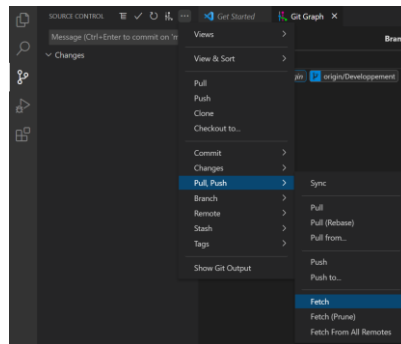
Cela permet aussi de vérifier si la branche mère sur laquelle notre propre branche se base a évolué ou non.

Imaginons qu'un collaborateur a créé sa propre branche basée sur la branche de développement. Actuellement, si vous ne faites pas de fetch, vous ne la verrez pas apparaître.

Prenons la situation suivante, où notre stagiaire a créé sa branche de développement avec un commit :

Default branch			
main	Updated 23 hours ago by MReheisser	Default	
Active branches			
Dev-feature-stagiaire	Updated 3 minutes ago by Stagiaire ECAM	0 2	New pull request edit delete
Developpement	Updated 23 hours ago by MReheisser	0 0	New pull request edit delete

Sans fetch, vous ne verrez pas les changements. Pour ça, allez dans le menu « source control », puis « Push/Pull > Fetch » :



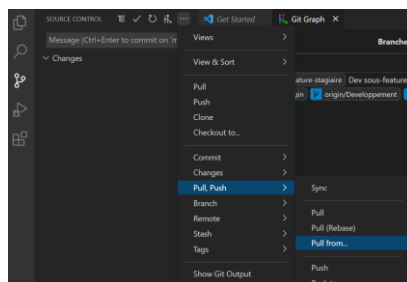
Graph	Description	Date	Author	Commit
origin/Dev-feature-stagiaire	Dev sous-feature 1	25 Mar 2022 17:42	Stagiaire ECAM	e25f45fe
main origin origin/Developpement origin/HEAD	Commit initial	24 Mar 2022 18:32	Mathias REHEISSER	6cf95370

On a bien les infos sur cette branche qui apparaissent (on le voit avec le préfixe « origin/ » devant le nom) ! On peut, si on le souhaite, checkout sur cette branche pour pouvoir travailler dessus.

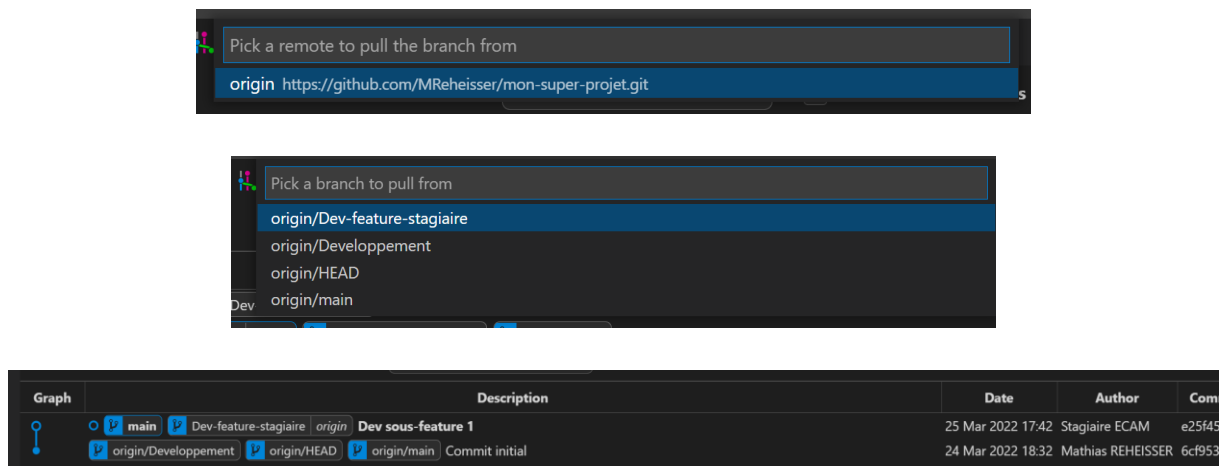
La « pull » quant à elle va venir, en plus d'exécuter un fetch, venir merge une branche dans la branche sur laquelle on est actuellement.

Un pull simple va permettre de récupérer les derniers commits effectués sur la branche.

Pour pull les modifications d'une autre branche, cliquez dans le menu sur « pull from... » (« pull » simple permet seulement de pull les derniers commits de notre branche en cours).



Dans la barre qui apparaît, entrez l'url du dépôt Git, et sélectionnez la branche qui vous intéresse pour pouvoir la merge :



Attention, cette fusion n'est présente qu'en local, il faut encore la push en ligne pour qu'elle soit réellement effective !

Cours annexe 2 – Gérer son code avec Git
Version 1.0

Mathias REHEISSER
2021-2022