

Programmation Objet pour les sciences du vivant

Gestion Git et Github

2A UC5 AgroParisTech

UFR d'Informatique
Département MMIP

Fatma Chamekh

2021-2022



1 Introduction :

Un gestionnaire de versions est un programme qui permet aux développeurs de conserver un historique des modifications et des versions de tous leurs fichiers. Le gestionnaire de version permet de garder en mémoire chaque modification pour chaque fichier (pourquoi, quand, qui).

Cet action de versioning peut être mise en place en cas de :

- Un développeur (se) qui travaille seul afin de garder l'historique des modifications et surtout revenir à une version précédente facilement.
- Une équipe de développeurs (ses) peut travailler simultanément sur un même fichier. Grâce au gestionnaire des versions, les modifications apportées par les membres de l'équipe seront fusionnées. En plus, le risque de suppression de par erreur, des changements apportés sur le fichier, est écarté.

Il existe plusieurs outils de gestion des versions : DARCS, SVK, visual source safe, GIT ...etc. L'outil le plus utilisé est GIT. Ce dernier fera l'objet du présent tutoriel.

Pour résumer, un gestionnaire de versions offre trois principales fonctionnalités :

- Suivre l'évolution du code étape par étape.
- Naviguer entre les versions du code afin de gérer d'éventuels problèmes.
- Travailler en équipe sans risque de perte des modifications des collaborateurs.
- Gérer les conflits d'édition des fichiers.

Afin de mieux ancrer l'intérêt de l'usage d'un outil de gestion des versions, nous allons exposer le cas d'Alice et Bob, étudiants en 2 UCA. Ils suivent le module programmation objet pour la science du vivant. Alice et Bob décident de travailler la série des exercices de révision proposée au cours de la deuxième séance. Les deux étudiants se partagent les exercices et décident de travailler chacun de son côté. Des points d'étape réguliers sont prévus afin de gérer les erreurs/réconcilier les idées. Ils décident de créer un dossier sur Drive afin de mettre en commun leurs travaux respectifs.

Alice et Bob travaillent sur le premier exercice de la série révision du cours UC5. Alice a commencé à coder et à échanger son code avec Bon afin de terminer le code et le corrigé.

```

mydate =datetime.datetime.today()
year = mydate.year

#Renseigner lesinfos utilisateur
answer1 = input("Comment vous appelez-vous ? ")
answer2 = input("Quel âge avez-vous ? ")
answer2 = int(str(answer2))

answer3= year - answer2 + 20
if (answer3 < 2020):
    answer4 = "et vous avez eu 20 ans en "
else:
    answer4 = "et vous en aurez 20 en "
print(answer1 + " vous avez " + str(answer2) + " ans " + answer4 + str(answer3))

```

Alice envoie son code à Bob car elle rencontre des erreurs de compilation. De son côté Bob a corrigé les erreurs et renvoyé le code à Alice. Deux jours après, Bob a essayé de compiler le code mais ceci ne fonctionne pas. Le problème ? Sans s'en rendre compte, Alice a écrasé le code de Bob en effectuant ses modifications. Bob n'a pas copié son travail en local, il a donc codé pendant un mois pour rien car il lui est impossible de récupérer son travail.

Tout cela aurait pu être évité avec le gestionnaire de versions !

Si Alice et Bob avaient initialisé Git pour leur projet, ils auraient pu modifier leurs fichiers, envoyer et recevoir les mises à jour à tout moment, sans risque d'écraser les modifications de l'autre. Les modifications de dernière minute n'auraient donc pas eu d'impact sur la production finale !

1.2 Quels différences entre GIT, Github et GITLAB

Git et GitHub sont deux choses différentes.

Git est un gestionnaire de versions. Vous l'utilisez pour créer un dépôt local et gérer les versions de vos fichiers.

GitHub est un service en ligne qui va héberger votre dépôt. Dans ce cas, on parle de **dépôt distant** puisqu'il n'est pas stocké sur votre machine.

De même, **GitLab** offre aussi un service en ligne pour héberger le dépôt du code. Cependant, **GitHub** est une solution moins « prête à l'emploi » que GitLab, qui offre plutôt aux développeurs la possibilité d'implémenter librement des applications et des intégrations via la place de marché GitHub.

C'est un peu confus ? Pas de panique, prenons un exemple pour illustrer cela.

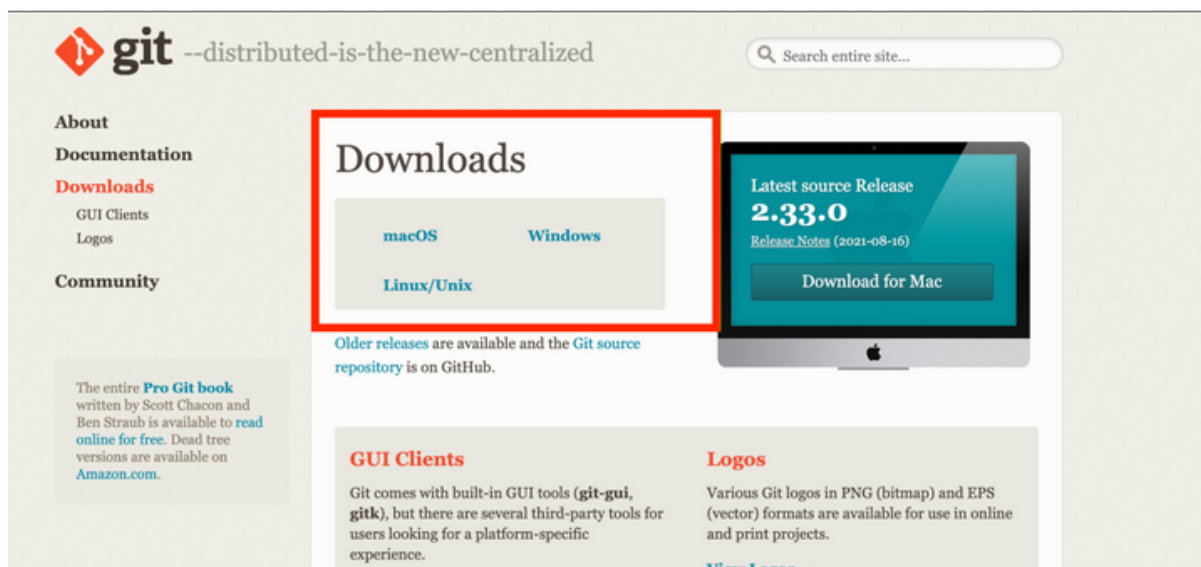
Imaginez que vous participiez à un dîner familial sous thème de tarte. Chez vous, vous avez préparé une pâte brisée et vous l'avez ramenée chez vos parents pour la conserver au frigo. Cette pâte pourra être utilisée telle qu'elle est ou modifier en ajoutant d'autres ingrédients.

Eh bien, c'est la même chose pour Git et GitHub. Git est la pâte que vous avez réalisée chez vous, et GitHub est le frigo, dans lequel, est conservée où elle peut être modifiée par les autres ou distribuée.

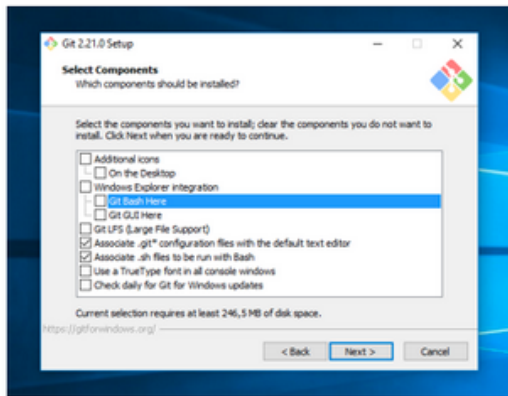
3 Installation GIT

Entrons dans le vif du sujet et passons immédiatement à l'installation du logiciel Git. La façon la plus simple d'installer Git est de télécharger la dernière version sur le site officiel <https://git-scm.com/downloads>, d'ouvrir le fichier téléchargé et de suivre les instructions à l'écran en laissant toutes les valeurs par défaut.

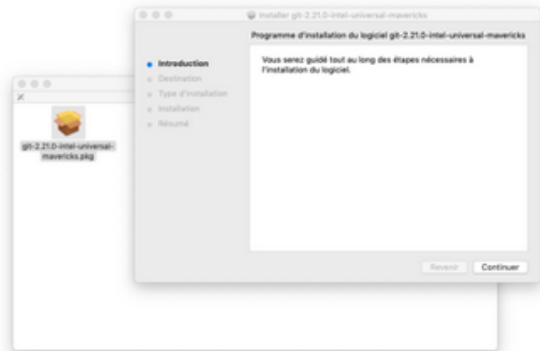
Si vous êtes sous Windows, téléchargez plutôt la version de Git présente sur <https://gitforwindows.org/>. Cette version inclut un outil permettant d'émuler le comportement de Bash (le langage utilisé par Mac et Linux) et donc d'avoir accès aux mêmes commandes que moi.



Exécutez le fichier que vous venez de télécharger. Appuyez sur Suivant à chaque fenêtre puis sur **Installer**. Lors de l'installation, laissez toutes les options par défaut, elles conviennent bien.



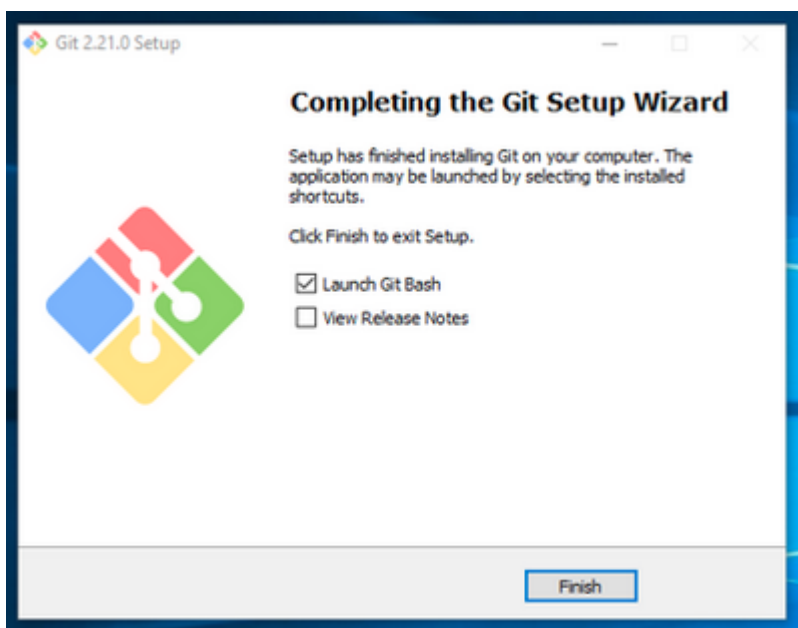
Sous Windows



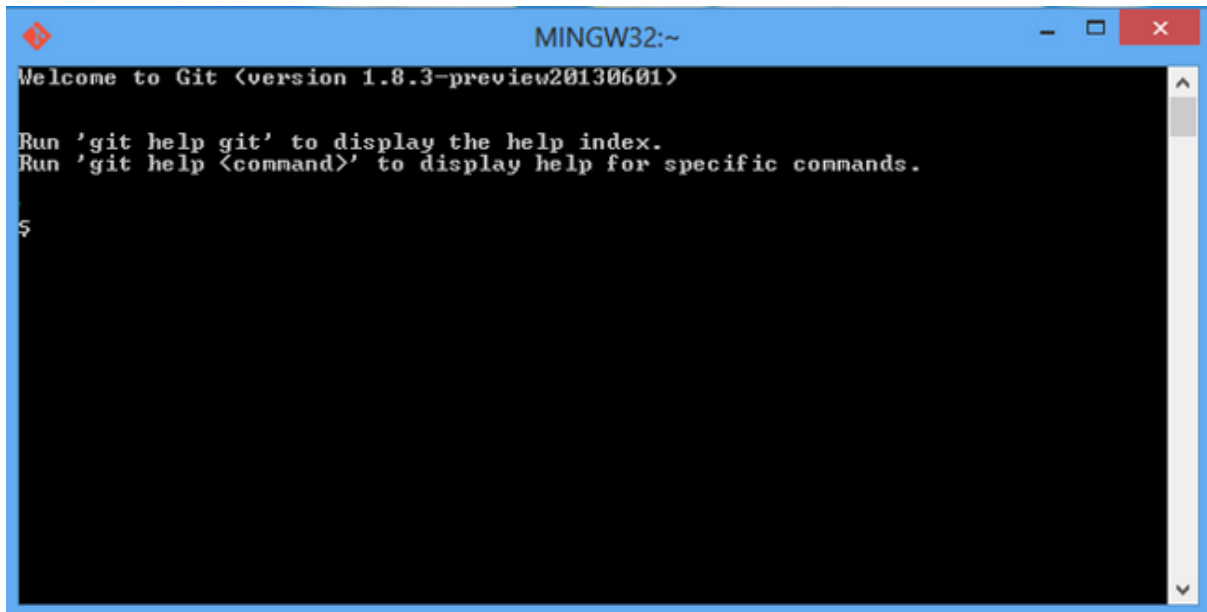
Sous Mac

Lancez l'installation du programme

Si vous êtes sous Windows : cochez ensuite Launch Git Bash. Pour les utilisateurs de Mac ou Linux, votre terminal suffira amplement.



Git Bash est l'interface permettant d'utiliser Git en ligne de commande. Git Bash se lance.:



```
MINGW32:~
Welcome to Git (version 1.8.3-preview20130601)

Run 'git help git' to display the help index.
Run 'git help <command>' to display help for specific commands.

$
```

Fenêtre Git Bash

4 Paramétrage de GIT :

Une fois Git installé, nous allons paramétrer le logiciel afin d'enregistrer certaines données pour ne pas avoir à les fournir à nouveau plus tard. Afin de réaliser cet étape, nous devons travailler, impérativement, en ligne de commande pour les trois systèmes d'exploitations (Linux, windows et mac). Il faut ouvrir le terminal/cmd.

Nous allons notamment ici renseigner un nom d'utilisateur et une adresse mail que Git devra utiliser ensuite.

Pour faire cela, nous allons utiliser notre première commande Git qui est la commande git config. Cette commande permet de de voir et modifier les variables de configuration qui contrôlent tous les aspects de l'apparence et du comportement de Git.

Nous allons également passer une option --global à notre commande. Les options permettent de personnaliser le comportement par défaut de certaines commandes. Ici, l'option --global va nous permettre d'indiquer à Git que le nom d'utilisateur et l'adresse mail renseignés doivent être utilisés globalement (c'est-à-dire pour tout projet Git).

On va donc taper les commandes suivantes : git config --global user.name "Fatma Chamekh" et git config --global user.email fatma.chamekh@agroparistech.fr à la suite pour renseigner un nom et une adresse email. Bien évidemment, utilisez votre propre nom et votre propre adresse email.

```
fatma@mmip-HP-EliteBook-840-G2:~$ git --version
git version 2.25.1
fatma@mmip-HP-EliteBook-840-G2:~$ git config --global user.name "Fatma Chamekh"
fatma@mmip-HP-EliteBook-840-G2:~$ git config --global user.email fatma.chamekh@agroparistech.fr
fatma@mmip-HP-EliteBook-840-G2:~$
```

Pour vous assurer que vos informations ont bien été enregistrées, vous pouvez taper `git config user.name` et `git config user.email`. Les informations entrées devraient être renvoyées.

```
fatma@mmip-HP-EliteBook-840-G2:~$ git config user.name
Fatma Chamekh
fatma@mmip-HP-EliteBook-840-G2:~$ git config user.email
fatma.chamekh@agroparistech.fr
fatma@mmip-HP-EliteBook-840-G2:~$
```

Voilà tout pour la configuration. Notez que certaines commandes Git ouvrent un éditeur, notamment celles qui vous demandent de saisir du texte. L'éditeur par défaut utilisé par Git est l'éditeur système qui est généralement Vi ou Vim.

J'utiliserai l'éditeur par défaut pour ce cours. Si vous souhaitez configurer un éditeur différent, vous pouvez entrer la commande `git config --global core.editor nom_de_votre_editeur`.

Nous allons présenter le fonctionnement général de Git et définir des éléments de vocabulaire qui vont nous être utiles par la suite.

5 Fonctionnement général de Git

Dans cette leçon, nous allons présenter le fonctionnement général de Git et définir des éléments de vocabulaire qui vont nous être utiles par la suite.

Démarrer un dépôt Git

Un "dépôt" correspond à la copie et à l'importation de l'ensemble des fichiers d'un projet dans Git. Il existe deux façons de créer un dépôt Git :

- On peut importer un répertoire déjà existant dans Git ;

- On peut cloner un dépôt Git déjà existant.

Nous allons voir comment faire cela dans la suite de ce cours. Avant cela, je pense qu'il est bon de comprendre comment Git conçoit la gestion des informations ainsi que le fonctionnement général de Git.

La gestion des informations selon Git

La façon dont Git considère les données est assez différente de la plupart des autres systèmes de gestion de version.

Git pense les données à la manière d'un flux d'instantanés ou "snapshots". Grosso modo, à chaque fois qu'on va valider ou enregistrer l'état d'un projet dans Git, il va prendre un instantané du contenu de l'espace de travail à ce moment et va enregistrer une référence à cet instantané pour qu'on puisse y accéder par la suite.

Chaque instantané est stocké dans une base de donnée locale, c'est-à-dire une base de donnée située sur notre propre machine.

Le fait que l'on dispose de l'historique complet d'un projet localement fait que la grande majorité des opérations de Git peuvent être réalisées localement, c'est-à-dire sans avoir à être connecté à un serveur central distant. Cela rend les opérations beaucoup plus rapides et le travail de manière générale beaucoup plus agréable.

Les états des fichiers

Comment Git fait-il pour suivre les modifications sur les fichiers d'un projet ? Pour comprendre cela, il faut savoir qu'un fichier peut avoir deux grands états dans Git : il peut être sous suivi de version ou non suivi.

Un fichier possède l'état "suivi" si il appartenait au dernier instantané capturé par Git, c'est-à-dire si il est enregistré en base. Tout fichier qui n'appartenait pas au dernier instantané et qui n'a pas été indexé est "non suivi".

Lorsqu'on démarre un dépôt Git en important un répertoire déjà existant depuis notre machine, les fichiers sont au départ tous non suivis. On va donc déjà devoir demander à Git de les indexer et de les valider (les enregistrer en base). Lorsqu'on clone un dépôt Git déjà existant, c'est différent puisqu'on copie tout l'historique du projet et donc les fichiers sont tous déjà suivis par défaut.

Ensuite, chaque fichier suivi peut avoir l'un de ces trois états :

- Modifié ("modified") ;
- Indexé ("staged") ;
- Validé ("committed").

Lors du démarrage d'un dépôt Git à partir d'un dépôt local, on demande à Git de valider l'ensemble des fichiers du projet. Un fichier est "validé" lorsqu'il est stocké dans la base de données locale. Lors du clonage d'un dépôt déjà existant, les fichiers sont enregistrés par défaut en base et ils sont donc validés par défaut.

Ensuite, lorsqu'on va travailler sur notre projet, on va certainement ajouter de nouveaux fichiers ou modifier des fichiers existants. Les fichiers modifiés vont être considérés comme "modifiés" par Git tandis que les nouveaux fichiers vont être "non suivis". Un fichier modifié est considéré comme "modifié" par Git tant qu'il n'a pas été indexé.

On dit qu'on "indexe" un fichier lorsqu'on indique à Git que le fichier modifié ou que le nouveau fichier doit faire partie du prochain instantané dans sa version actuelle.

Enfin, lorsqu'on demande à Git de prendre l'instantané, c'est-à-dire lorsqu'on lui demande d'enregistrer en base l'état du projet actuel (c'est-à-dire l'ensemble des fichiers indexés et non modifiés), les fichiers faisant partie de l'instantané sont à nouveau considérés comme "validés" et le cycle peut recommencer.

Les zones de travail

Les états de fichiers sont liés à des zones de travail dans Git. En fonction de son état, un fichier va pouvoir apparaître dans telle ou telle zone de travail. Tout projet Git est composé de trois sections : le répertoire de travail (working tree), la zone d'index (staging area) et le répertoire Git (repository).

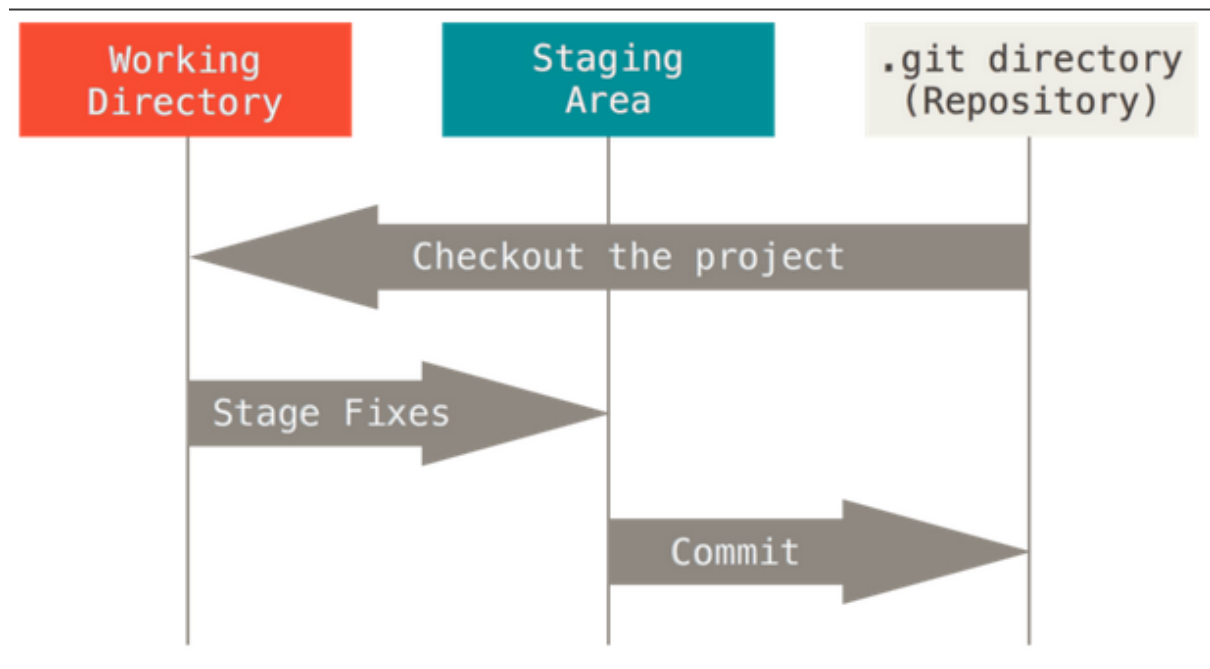
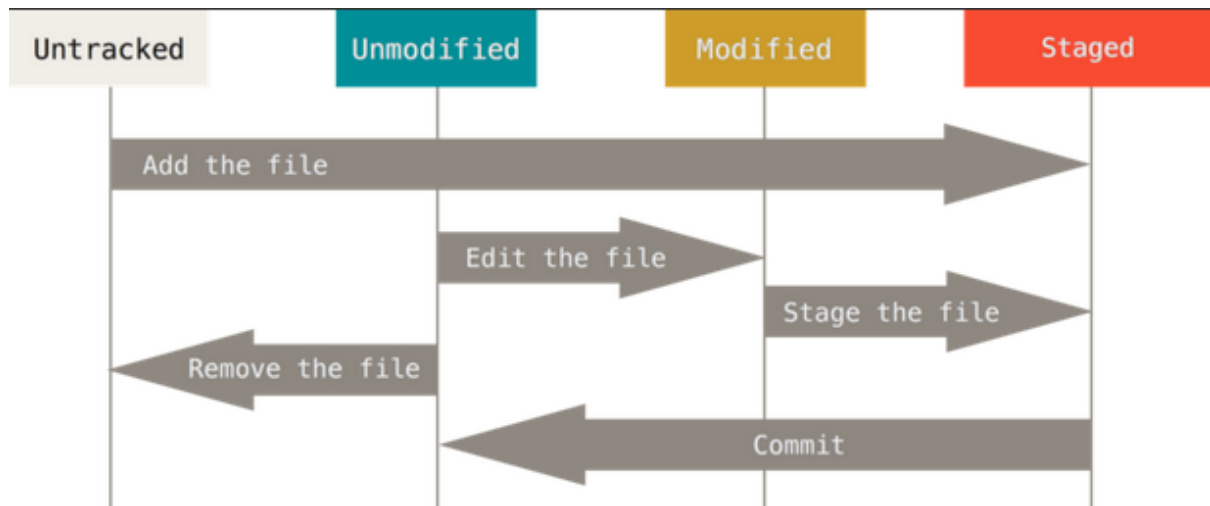
Le répertoire de travail ou "working tree" correspond à une extraction unique ("checkout") d'une version du projet. Les fichiers sont extraits de la base de données compressée située dans le répertoire Git et sont placés sur le disque afin qu'on puisse les utiliser ou les modifier.

La zone d'index ou "staging area" correspond à un simple fichier, généralement situé dans le répertoire Git, qui stocke les informations concernant ce qui fera partie du prochain instantané ou du prochain "commit".

Le répertoire Git est l'endroit où Git stocke les méta-données et la base de données des objets de votre projet. C'est la partie principale ou le coeur de Git.

Le processus de travail va ainsi être le suivant : nous allons travailler sur nos fichiers dans le répertoire de travail. Lorsqu'on modifie ou crée un fichier, on peut ensuite choisir de l'indexer. Tant qu'un fichier n'est pas indexé, il possède l'état modifié ou est non suivi si c'est un nouveau fichier. Dès qu'il est indexé i.e que son nom est ajouté à la zone d'index, il possède l'état indexé. Finalement, on va valider ("commit") la version indexée de nos fichiers pour les ajouter au répertoire Git.

Pour retenir ces informations, vous pouvez vous aider des schémas ci-dessous (source : git-scm.com).



Démarrer un dépôt Git

Un “dépôt” correspond à la copie et à l’importation de l’ensemble des fichiers d’un projet dans Git. Comme évoqué précédemment, Il existe deux façons de créer un dépôt Git :

- On peut initialiser un dépôt Git à partir d’un répertoire déjà existant, soit cloner un dépôt Git déjà existant.
- On peut cloner un dépôt Git déjà existant.

Nous allons voir comment faire cela dans la suite de ce cours. Avant cela, je pense qu’il est bon de comprendre comment Git conçoit la gestion des informations ainsi que le fonctionnement général de Git.

Créer un dépôt Git à partir d'un répertoire existant

Lorsqu'on démarre avec Git, on a souvent déjà des projets en cours stockés localement sur notre machine ou sur serveur distant et pour lesquels on aimerait implémenter un système de gestion de version.

Dans ce cas là, nous allons pouvoir importer l'ensemble des ressources d'un projet dans Git. Pour la suite de cette leçon, je vais créer un répertoire "projet-git" qui se trouve sur mon bureau et qui contient deux fichiers texte vides "fichier1.txt" et "README.txt". Ce répertoire va me servir de base pour les exemples qui vont suivre (ce sera le répertoire importé).

Je vous invite à créer le même répertoire sur votre machine. Vous pouvez le faire soit à la main, soit en utilisant la ligne de commandes comme ci-dessous (attention, toute mon installation et mon système sont en anglais, il est possible que vous ayez à remplacer "desktop" par "bureau" entre autres) :

```
fatma@mmip-HP-EliteBook-840-G2:~$ cd Bureau
fatma@mmip-HP-EliteBook-840-G2:~/Bureau$ mkdir projet-git
fatma@mmip-HP-EliteBook-840-G2:~/Bureau$ cd projet-git
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ touch fichier1.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ touch README.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ ls
fichier1.txt  README.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$
```

La commande `cd` sert à se déplacer dans un répertoire. Dès qu'on est sur le bureau, on utilise `mkdir` pour créer un répertoire vide qu'on appelle "projet-git". On se place dans ce répertoire et on crée deux fichiers texte grâce à la commande Bash `touch`. On utilise enfin `ls` pour afficher le contenu du répertoire et s'assurer que tout a bien fonctionné.

Pour initialiser un dépôt Git, on utilise ensuite la commande `git init` comme ci-dessous. Cela crée un sous répertoire `.git` qui contient un ensemble de fichiers qui vont permettre à un dépôt Git de fonctionner.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git init
Dépôt Git vide initialisé dans /home/fatma/Bureau/projet-git/.git/
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$
```

Lorsqu'on utilise `git init`, Git nous renvoie un message en nous informant que le dépôt Git a bien été initialisé et qu'il est vide. C'est tout à fait normal puisque nous n'avons encore versionné aucun fichier (nous n'avons ajouté aucun fichier du répertoire en base).

On peut utiliser ici la commande `git status` pour déterminer l'état des fichiers de notre répertoire. Cette commande est extrêmement utile et c'est une de celles que j'utilise le plus personnellement :

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master

Aucun commit

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    README.txt
    fichier1.txt

aucune modification ajoutée à la validation mais des fichiers non suivis sont pr
ésents (utilisez "git add" pour les suivre)
```

Ici, git status nous informe que notre projet possède deux fichiers qui ne sont pas sous suivi de version ("untracked") et qui sont les fichiers "README.txt" et "fichier1.txt". Il nous dit aussi qu'aucun fichier n'a été validé ("commit") en base pour le moment ni ajouté pour validation. La commande git status nous informe également sur la branche sur laquelle on se trouve ("master" ici). Nous reparlerons des branches plus tard.

L'étape suivante va donc ici être d'indexer nos fichiers afin qu'ils puissent ensuite être validés, c'est-à-dire ajoutés en base et qu'on puisse ainsi avoir un premier historique de version.

Pour indexer des fichiers, on utilise la commande git add. On peut lui passer un nom de fichier pour indexer le fichier en question, le nom d'un répertoire pour indexer tous les fichiers du répertoire d'un coup ou encore un "fileglob" pour ajouter tous les fichiers correspondant au schéma fourni.

Les fileglobs utilisent les extensions de chemin de fichier. Grosso-modo, cela signifie que certains caractères comme * et ? vont posséder une signification spéciale et nous permettre de créer des schémas de correspondances. Le caractère * par exemple correspond à n'importe quel caractère. Lorsque j'écris git add *.txt, je demande finalement à Git d'ajouter à l'index tous les fichiers du projet qui possèdent une extension .txt, quel que soit leur nom.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git add *.txt
```

Si on relance une commande git status, on obtient les informations suivantes :

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master

Aucun commit

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)
    nouveau fichier : README.txt
    nouveau fichier : fichier1.txt
```

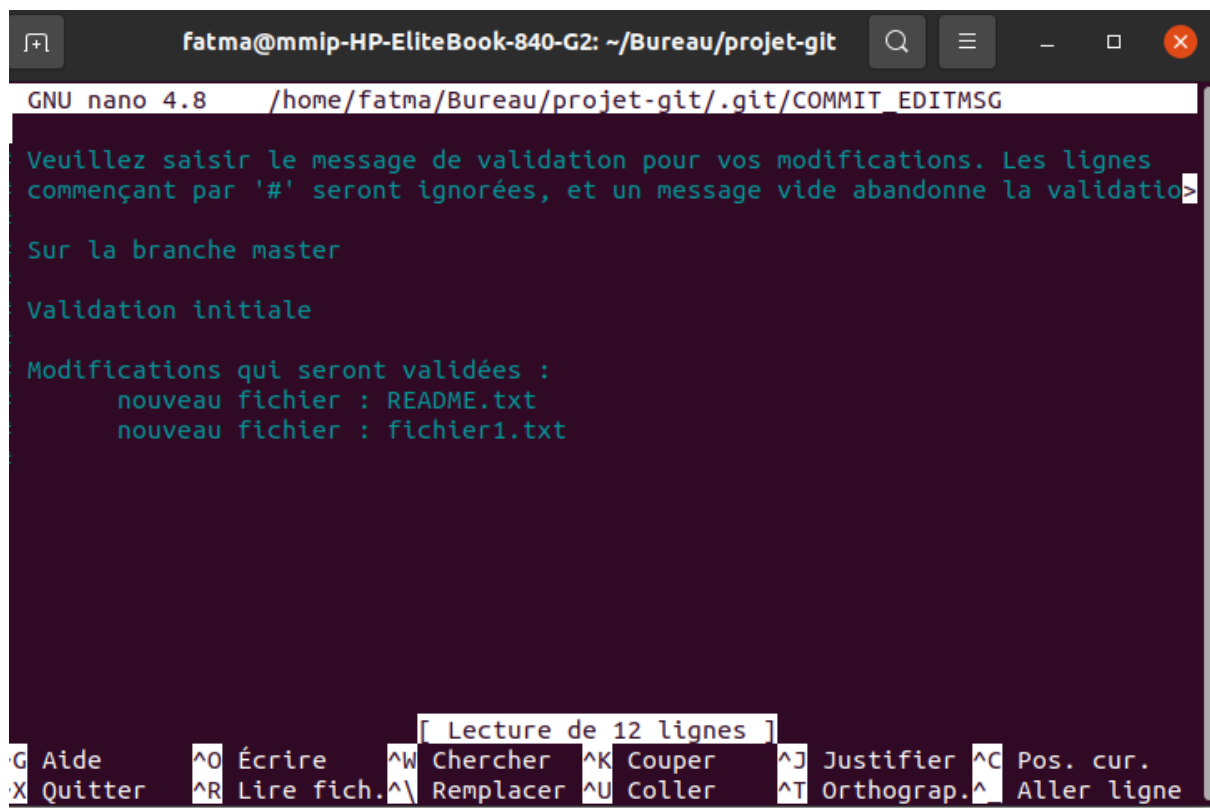
git status nous dit qu'on a maintenant deux nouveaux fichiers ajoutés à l'index. La commande git add permet en fait de faire plusieurs choses : elle permet d'indexer des fichiers déjà sous suivi de version et de placer sous suivi des fichiers non suivi (en plus de les indexer).

Ici, on est certains que nos deux nouveaux fichiers ont bien été ajoutés à l'index puisqu'ils apparaissent dans la section "changes to be committed" ("modifications à valider").

Pour valider ces fichiers et les ajouter en base, on va maintenant utiliser la commande git commit comme cela :

```
fatma@mmip-HP-EliteBook-840-G2: ~/Bureau/projet-git$ git commit
```

Lorsqu'on utilise git commit sans argument, une nouvelle fenêtre s'ouvre en utilisant l'éditeur par défaut qui est dans la majorité des cas VIM.



The screenshot shows a terminal window titled "fatma@mmip-HP-EliteBook-840-G2: ~/Bureau/projet-git". Inside, the GNU nano 4.8 editor is open at the file "/home/fatma/Bureau/projet-git/.git/COMMIT_EDITMSG". The editor displays the following text:

```
Veuillez saisir le message de validation pour vos modifications. Les lignes commençant par '#' seront ignorées, et un message vide abandonne la validation>
```

Below this, it indicates "Sur la branche master" and "Validation initiale". It then lists the modifications to be validated:

```
Modifications qui seront validées :
nouveau fichier : README.txt
nouveau fichier : fichier1.txt
```

At the bottom of the terminal, a status bar shows "Lecture de 12 lignes" and a list of nano editor shortcuts: G Aide, X Quitter, ^O Écrire, ^R Lire fich., ^W Chercher, ^\ Remplacer, ^K Couper, ^U Coller, ^J Justifier, ^T Orthograp., ^C Pos. cur., and ^_ Aller ligne.

Ici, on nous demande d'ajouter un message avec notre commit. Bien documenter chaque commit permet aux auteurs et aux différents contributeurs à un projet de rapidement comprendre les modifications et de pouvoir les valider. C'est une part essentielle de Git. Ici, j'ajoute simplement le message "Version initiale du projet".

Une fois le message entré, si votre éditeur est bien VIM, il faut appuyer sur la touche "esc" pour sortir du mode insertion puis taper :wq et entrée pour valider et quitter ou :x et entrée ou tout simplement ZZ.

Une fois sorti de VIM, un message s'affiche avec des informations sur le commit effectué.

On nous informe ici qu'on se situe sur la branche master, qu'il s'agit du premier commit (root-commit) et on nous donne sa somme de contrôle (4ed866e) qui permet de l'identifier de manière unique. On nous dit également que deux fichiers ont été modifiés et que 0 lignes ont été ajoutées ou supprimées dans ces fichiers.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master

Aucun commit

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)
    nouveau fichier : README.txt
    nouveau fichier : fichier1.txt
```

Git nous informe désormais qu'il n'y a plus aucun fichier à vider, ce qui signifie que tous les fichiers sont sous suivi de version et sont enregistrés en base et qu'aucune modification n'a été apportée à ces fichiers depuis le dernier commit.

Cloner un dépôt Git

La deuxième façon de démarrer un dépôt Git est de cloner localement un dépôt Git déjà existant. Pour cela, on va utiliser la commande Git clone.

En pratique, dans la grande majorité des cas, nous clonons des dépôts Git distants, c'est-à-dire des dépôts hébergés sur serveur distants pour pouvoir contribuer à ces projets.

Cependant, nous pouvons également cloner des dépôts locaux. Nous parlerons des dépôts distants et apprendrons à les cloner lorsqu'on abordera GitHub. Pour le moment, contentons nous d'essayer de cloner notre dépôt "projet-git" tout juste créé.

Pour cela, on va se placer sur le bureau. Comme je suis pour le moment situé dans mon répertoire "projet-git", j'utilise la commande Bash `cd ..` pour atteindre le répertoire parent (c'est-à-dire mon bureau).

J'utilise ensuite la commande `git clone` en lui passant d'abord le chemin complet du projet à cloner (qui correspond à son nom dans notre cas puisque le répertoire du projet est également sur le bureau) puis le chemin complet du clone qui doit être créé. On va choisir de créer le clone sur le bureau par simplicité et on va donc simplement passer un nom à nouveau. Appelons le clone "projet-git-2" par exemple comme ceci :

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ cd ..
fatma@mmip-HP-EliteBook-840-G2:~/Bureau$ git clone projet-git projet-git2
Clonage dans 'projet-git2'...
warning: Vous semblez avoir cloné un dépôt vide.
fait.
fatma@mmip-HP-EliteBook-840-G2:~/Bureau$
```

On peut se déplacer dans le répertoire du projet que je viens de créer en utilisant `cd` et effectuer un ultime `git status` pour s'assurer de l'état des fichiers du répertoire :

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau$ cd projet-git2
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git2$ git status
Sur la branche master

Aucun commit

rien à valider (créez/copiez des fichiers et utilisez "git add" pour les suivre)
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git2$
```

Comme vous pouvez le voir, le dépôt a bien été cloné puisque les fichiers du répertoire `projet-git-2` sont déjà tous sous suivi de version et sont stockés en base.

6 Modifier un dépôt GIT

Ajouter ou modifier des fichiers dans un projet et actualiser notre dépôt Git

A ce niveau, nous avons donc d'un côté notre projet contenant un ensemble de fichiers et de ressources sur lesquelles on travaille ainsi qu'un dépôt Git qui sert à gérer les différentes versions de notre projet.

En continuant à travailler sur notre projet, nous allons être amenés à ajouter, modifier, voire supprimer des fichiers. On va indiquer tous ces changements à Git pour qu'il conserve un historique des versions auquel on pourra ensuite accéder pour revenir à un état précédent du projet (dans le cas où une modification entraîne un bogue par exemple ou n'amène pas le résultat souhaité).

A chaque fois qu'on souhaite enregistrer une modification de fichier ou un ajout de fichier dans le dépôt Git, on va devoir utiliser les commandes `git add` et `git commit` comme on a pu le faire dans la leçon précédente.

J'attire ici votre attention sur un point important : le commit (la validation / l'enregistrement en base de données) d'un fichier se basera sur l'état de ce fichier au moment du `git add`.

Cela signifie que si vous effectuez une commande `git add` sur un fichier puis modifiez à nouveau le fichier puis effectuez un `git commit`, c'est le fichier dans son état au moment du dernier `git add` qui sera validé et les dernières modifications ne seront donc pas enregistrées dans le dépôt Git.

Si vous souhaitez enregistrer la dernière version d'un fichier, pensez donc bien toujours à effectuer un `git add` juste avant un `git commit`. Pour mettre en un coup les fichiers modifiés et déjà sous suivi dans la zone d'index puis pour les valider, vous pouvez également utiliser `git commit` avec une option `-a` comme ceci : `git commit -a`. Cela vous dispense d'avoir à taper `git add`.

Supprimer un fichier d'un projet et / ou l'exclure du suivi de version Git

Concernant la suppression de fichiers, il existe plusieurs situations possibles en fonction de ce que vous souhaitez réellement faire : voulez vous simplement exclure un fichier du suivi de version mais le conserver dans votre projet ou également le supprimer du projet ?

Avant tout, notez que simplement supprimer un fichier de votre dossier avec une commande Bash rm par exemple ne suffira pas pour que Git oublie ce fichier : il apparaîtra dans chaque git status dans une section "changes not stages for commit" (modifications qui ne seront pas validées).

On peut s'en assurer en faisant le test. Pour cela, commençons par ajouter un nouveau fichier dans votre répertoire "projet-git". On peut faire cela avec une commande Bash touch.

```
fatma@mmip-HP-EliteBook-840-G2:~$ cd Bureau
fatma@mmip-HP-EliteBook-840-G2:~/Bureau$ cd projet-git
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ touch test.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master

Aucun commit

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)
    nouveau fichier : README.txt
    nouveau fichier : fichier1.txt

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    test.txt

fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$
```

Comme le fichier vient tout juste d'être ajouté, on va devoir l'indexer pour qu'il soit sous suivi Git puis le valider.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git add test.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git commit -m 'Ajout du fichier test.txt'
[master (commit racine) 8fdb7a] Ajout du fichier test.txt
 3 files changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.txt
 create mode 100644 fichier1.txt
 create mode 100644 test.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$
```


ici, on utilise la commande git commit avec une option -m qui me permet de renseigner directement le message lié à mon commit plutôt que de devoir le faire dans VIM.

Le fichier est désormais sous suivi et la dernière version est enregistrée en base. Essayons maintenant de l'effacer avec une commande rm par exemple et tapons un nouveau git status :

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ rm test.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add/rm <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le ré
pertoire de travail)
    supprimé :      test.txt

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git
commit -a")
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$
```

Comme vous pouvez le voir, Git continue de suivre le fichier et la suppression simple du fichier ne sera pas validée comme changement par Git.

Pour supprimer un fichier et l'exclure du suivi de version, nous allons utiliser la commande git rm (et non pas simplement une commande Bash rm).

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ touch test.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
rien à valider, la copie de travail est propre
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git rm test.txt
rm 'test.txt'
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
    supprimé :      test.txt

fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git commit -m
error: switch `m' a besoin d'une valeur
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git commit -m 'suppression d
e test.txt projet + suivi'
[master 5fe47cd] suppression de test.txt projet + suivi
 1 file changed, 0 insertions(+), 0 deletions(-)
 delete mode 100644 test.txt
```

Pour simplement exclure un fichier du suivi Git mais le conserver dans le projet, on va utiliser la même commande git rm mais avec cette fois-ci une option --cached.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ touch fichier2.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git add fichier2.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git commit -m 'suppression d
e fichier2.txt projet + suivi'
[master 49b4890] suppression de fichier2.txt projet + suivi
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 fichier2.txt
```

```
fatna@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git rm --cached fichier2.txt
rm 'fichier2.txt'
fatna@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
    supprimé :      fichier2.txt

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    fichier2.txt
```

Ici, le fichier a bien été exclu du suivi Git mais existe toujours dans notre projet. On va ensuite pouvoir modifier ce fichier (lui ajouter du texte par exemple) comme n'importe quel fichier et Git ne se préoccupera pas des modifications.

```
fatna@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ echo 'Ajout de texte'>> fichier2.txt
fatna@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ cat fichier2.txt
Ajout de texte
fatna@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
    supprimé :      fichier2.txt

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    fichier2.txt
fatna@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$
```

Retirer un fichier de la zone d'index de Git

Le contenu de la zone d'index est ce qui sera proposé lors du prochain commit. Imaginons qu'on ait ajouté un fichier à la zone d'index par erreur. Pour retirer un fichier de l'index, on peut utiliser `git reset HEAD nom-du-fichier`.

A la différence de `git rm`, le fichier continuera d'être suivi par Git. Seulement, le fichier dans sa version actuelle va être retiré de l'index et ne fera donc pas partie du prochain commit.

Regardez plutôt l'exemple ci-dessous :

```

fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git add fichier2.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git commit -m
error: switch `m' a besoin d'une valeur
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git commit -m 'sauvegarde fichier2'
[master 0cc69ef] sauvegarde fichier2
1 file changed, 1 insertion(+)
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git add fichier2.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
rien à valider, la copie de travail est propre
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ echo 'modification du fichier'
modification du fichier
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ cat fichier2.txt
Ajout de texte
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
rien à valider, la copie de travail est propre
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git reset HEAD fichier2.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ echo 'modification du fichier' >> fichier2.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git reset HEAD fichier2.txt
Modifications non indexées après reset :
M    fichier2.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :      fichier2.txt

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$

```

Ici, on ajoute à nouveau le fichier fichier2.txt à l'index et on le passe donc sous suivi de version. On valide cela avec git commit puis on modifie le contenu de notre fichier et on ajoute la dernière version de notre fichier à la zone d'index pour qu'il fasse partie du prochain commit. Finalement, on change d'idée et on veut retirer cette version de la zone d'index. On fait cela avec git reset HEAD fichier2.txt.

Empêcher l'indexation de certains fichiers dans Git

Lorsqu'on dispose d'un projet et qu'on souhaite utiliser Git pour effectuer un suivi de version, il est courant qu'on souhaite exclure certains fichiers du suivi de version comme certains fichiers générés automatiquement, des fichiers de configuration, des fichiers sensibles, etc.

On peut informer Git des fichiers qu'on ne souhaite pas indexer en créant un fichier .gitignore et en ajoutant les différents fichiers qu'on souhaite ignorer. Notez qu'on peut également mentionner des schémas de noms pour exclure tous les fichiers correspondant à ce schéma et qu'on peut même exclure le contenu entier d'un répertoire en écrivant le chemin du répertoire suivi d'un slash.

Renommer un fichier dans Git

On peut également renommer un fichier de notre projet depuis Git en utilisant cette fois-ci une commande git mv ancien-nom-fichier nouveau-nom-fichier.

On peut par exemple renommer votre fichier "README.txt" en "LISEZMOI.txt" de la manière suivante :

```
fatma@mmip-HP-EliteBook-840-G2:~$ cd Bureau
fatma@mmip-HP-EliteBook-840-G2:~/Bureau$ cd projet-git
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git mv README.txt LISEZMOI.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
    renommé :      README.txt -> LISEZMOI.txt

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :      fichier2.txt

fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$
```

Le fichier a bien été renommé dans notre répertoire et le changement est prêt à être validé dans le prochain commit.

Consulter l'historique des modifications Git

La manière la plus simple de consulter l'historique des modifications Git est d'utiliser la commande `git log`. Cette commande affiche la liste des commits réalisés du plus récent au plus ancien. Par défaut, chaque commit est affiché avec sa somme de contrôle SHA-1, le nom et l'e-mail de l'auteur, la date et le message du commit.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git log
commit 0cc69ef43b1ebf5f8478201b97047e2d47b2dd94 (HEAD -> master)
Author: Fatma Chamekh <fatma.chamekh@agroparistech.fr>
Date:   Wed Mar 9 08:47:16 2022 +0100

    sauvegarde fichier2

commit 49b489056401c9ea0aa76fca9626116c9469b52a
Author: Fatma Chamekh <fatma.chamekh@agroparistech.fr>
Date:   Tue Mar 8 17:04:25 2022 +0100

    suppression de fichier2.txt projet + suivi

commit 5fe47cd15f73d8ce2bceedb1029a0a202b6bf9a3
Author: Fatma Chamekh <fatma.chamekh@agroparistech.fr>
Date:   Tue Mar 8 16:50:09 2022 +0100

    suppression de test.txt projet + suivi

commit 8fdb7e7a11f0a798c63e1ae726ecac69d223c8059
Author: Fatma Chamekh <fatma.chamekh@agroparistech.fr>
Date:   Tue Mar 8 15:23:32 2022 +0100

    Ajout du fichier test.txt
```

La commande git log supporte également de nombreuses options. Certaines vont pouvoir être très utiles comme par exemple les options -p, --pretty, --since ou --author.

Utiliser git log -p permet d'afficher explicitement les différences introduites entre chaque validation.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git log -p
commit 0cc69ef43b1ebf5f8478201b97047e2d47b2dd94 (HEAD -> master)
Author: Fatma Chamekh <fatma.chamekh@agroparistech.fr>
Date:   Wed Mar 9 08:47:16 2022 +0100

    sauvegarde fichier2

diff --git a/fichier2.txt b/fichier2.txt
index e69de29..27e7cf9 100644
--- a/fichier2.txt
+++ b/fichier2.txt
@@ -0,0 +1 @@
+Ajout de texte

commit 49b489056401c9ea0aa76fca9626116c9469b52a
Author: Fatma Chamekh <fatma.chamekh@agroparistech.fr>
Date:   Tue Mar 8 17:04:25 2022 +0100

    suppression de fichier2.txt projet + suivi

diff --git a/fichier2.txt b/fichier2.txt
new file mode 100644
index 0000000..e69de29
```

L'option --pretty permet, avec sa valeur oneline, d'afficher chaque commit sur une seule ligne ce qui peut faciliter la lecture dans le cas où de nombreux commits ont été réalisés.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git log --pretty=oneline
0cc69ef43b1ebf5f8478201b97047e2d47b2dd94 (HEAD -> master) sauvegarde fichier2
49b489056401c9ea0aa76fca9626116c9469b52a suppression de fichier2.txt projet + suivi
5fe47cd15f73d8ce2bceedb1029a0a202b6bf9a3 suppression de test.txt projet + suivi
8fdbbe7a11f0a798c63e1ae726ecac69d223c8059 Ajout du fichier test.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$
```

L'option --since permet de n'afficher que les modifications depuis une certaine date (on peut lui fournir différents formats de date comme 2.weeks ou 2019-10-10 par exemple).

L'option --author permet de n'afficher que les commits d'un auteur en particulier.

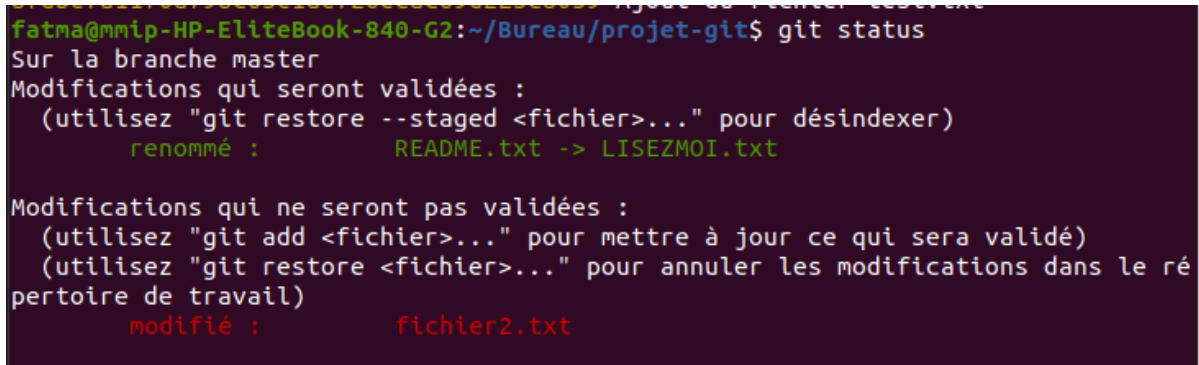
Écraser et remplacer un commit

Parfois, on voudra annuler une validation (un commit), notamment lorsque la validation a été faite en oubliant des fichiers ou sur les mauvaises versions de fichiers.

La façon la plus simple d'écraser un commit est d'utiliser la commande `git commit` avec l'option `--amend`. Cela va pousser un nouveau commit qui va remplacer le précédent en l'écrasant.

Par exemple, dans notre projet, on peut imaginer qu'on souhaite commit les changements effectués dans la leçon précédente sur le fichier `README.txt` et qu'on souhaite également réintégrer le fichier `fichier2.txt` dans l'index.

Pour cela, on effectue un `git commit` :



```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
    renommé :      README.txt -> LISEZMOI.txt

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :      fichier2.txt
```

Ici, on s'aperçoit après coup qu'on a oublié de réintégrer le fichier `fichier2.txt` à l'index. On peut utiliser une commande `git add` puis `git commit --amend` pour remplacer le commit précédent :

Annuler des modifications apportées à un fichier

L'un des principaux intérêts d'utiliser un logiciel de gestion de version est de pouvoir "roll back", c'est-à-dire de pouvoir revenir à un état antérieur enregistré d'un projet.

Après un commit, on va continuer à travailler sur nos fichiers et à les modifier. Parfois, certaines modifications ne vont pas apporter les comportements espérés et on voudra revenir à l'état du fichier du dernier instantané Git (c'est-à-dire au dernier état enregistré). On va pouvoir faire cela avec la commande générale `git checkout -- nom-du-fichier` ou la nouvelle commande spécialisée `git restore`.

Imaginons qu'on modifie le texte de notre fichier `LISEZMOI.txt` avec une commande `echo` et `>>`.


```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ echo 'blablabla' >>LISEZMOI.TXT
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
    renommé :      README.txt -> LISEZMOI.txt

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :      fichier2.txt

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    LISEZMOI.TXT
```

On se rend ensuite compte que ce texte ne convient pas et on souhaite revenir à l'état du fichier tel qu'il a été enregistré pour la dernière fois dans Git. Pour cela, on utilise simplement `git restore LISEZMOI.txt`.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git restore LISEZMOI.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ cat LISEZMOI.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
    renommé :      README.txt -> LISEZMOI.txt

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :      fichier2.txt

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    LISEZMOI.TXT
```

7 La Notion de Branche

Créer une branche, c'est en quelque sorte comme créer une "copie" de votre projet pour développer et tester de nouvelles fonctionnalités sans impacter le projet de base.

Dans la plupart des systèmes de contrôle de version, justement, une copie physique de la totalité du répertoire de travail est effectuée, ce qui rend la création de branches contraignante et en fait une opération lourde.

Git a une approche totalement différente des branches qui rend la création de nouvelles branches et la fusion de branche très facile à réaliser. Une branche, dans Git, est simplement un pointeur vers un commit (une branche n'est qu'un simple fichier contenant.

Concrètement, lorsqu'on effectue un commit, Git stocke en fait un objet commit qui contient les nom et prénom de l'auteur du commit, le message renseigné lors de la création du commit ainsi qu'un pointeur vers l'instantané du contenu indexé et des pointeurs vers le ou les commits précédant directement le commit courant.

Un pointeur est un objet qui contient l'adresse d'une donnée stockée quelque part. On peut utiliser le pointeur pour accéder à la donnée en question.

La branche par défaut dans Git s'appelle master. Cette branche master va se déplacer automatiquement à chaque nouveau commit pour pointer sur le dernier commit effectué tant qu'on reste sur cette branche.

Notez que la branche master n'est pas une branche spéciale pour Git : elle est traitée de la même façon que les autres branches. L'idée est que lorsqu'on tape une commande git init, une branche est automatiquement créée et que le nom donné à cette branche par Git par défaut est "master". On pourrait très bien la renommer ensuite mais ça ne présente généralement aucun intérêt.

En résumé, une branche est un pointeur vers un commit en particulier. Un commit est un objet qui contient un pointeur vers l'instantané du contenu indexé ainsi que des pointeurs vers le ou les commits le précédant directement.

Ainsi, créer une nouvelle branche dans Git crée simplement un nouveau pointeur plutôt que de recopier l'intégralité du répertoire de travail.

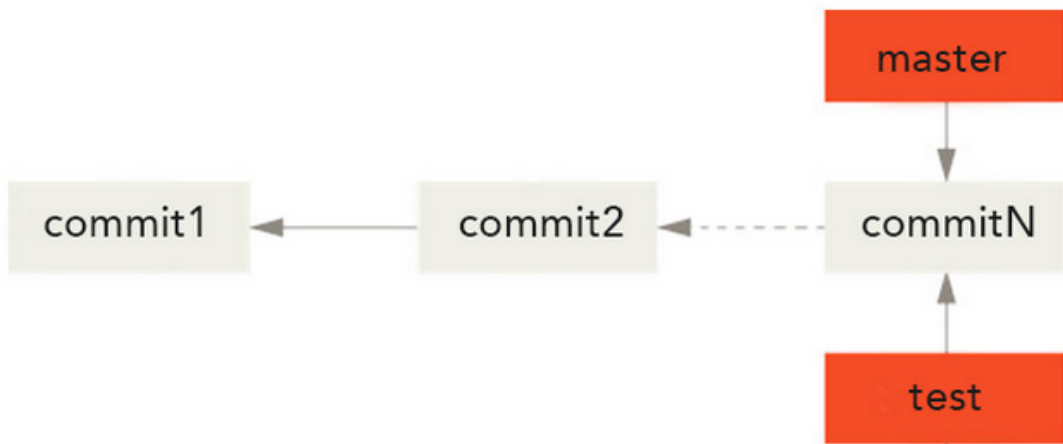
Créer une nouvelle branche

Pour créer une nouvelle branche, on utilise la commande git branch nom-de-la-branche. Cette syntaxe va créer un nouveau pointeur vers le dernier commit effectué (le commit courant). A ce stade, vous allez donc avoir deux branches (deux pointeurs) vers le dernier commit : la branche master et la branche tout juste créée.

```
branch
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git branch test
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
    renommé :      README.txt -> LISEZMOI.txt

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le ré
  pertoire de travail)
    modifié :      fichier2.txt

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    LISEZMOI.TXT
```

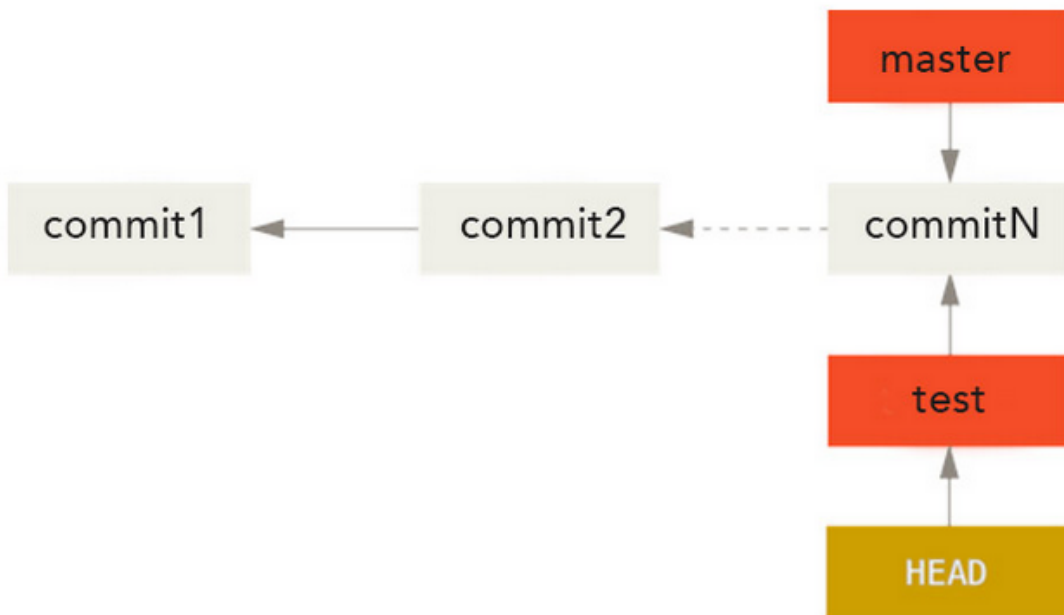
Pour déterminer quel pointeur vous utilisez, c'est-à-dire sur quelle branche vous vous trouvez, Git utilise un autre pointeur spécial appelé HEAD. HEAD pointe sur la branche master par défaut. Notez que la commande git branch permet de créer une nouvelle branche mais ne déplace pas HEAD.

Nous allons donc devoir déplacer explicitement HEAD pour indiquer à Git qu'on souhaite basculer sur une autre branche. On utilise pour cela la commande git checkout suivi du nom de la branche sur laquelle on souhaite basculer.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git checkout test
A    LISEZMOI.txt
D    README.txt
M    fichier2.txt
Basculement sur la branche 'test'
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche test
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
    renommé :      README.txt -> LISEZMOI.txt

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le ré
  pertoire de travail)
    modifié :      fichier2.txt

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    LISEZMOI.TXT
```



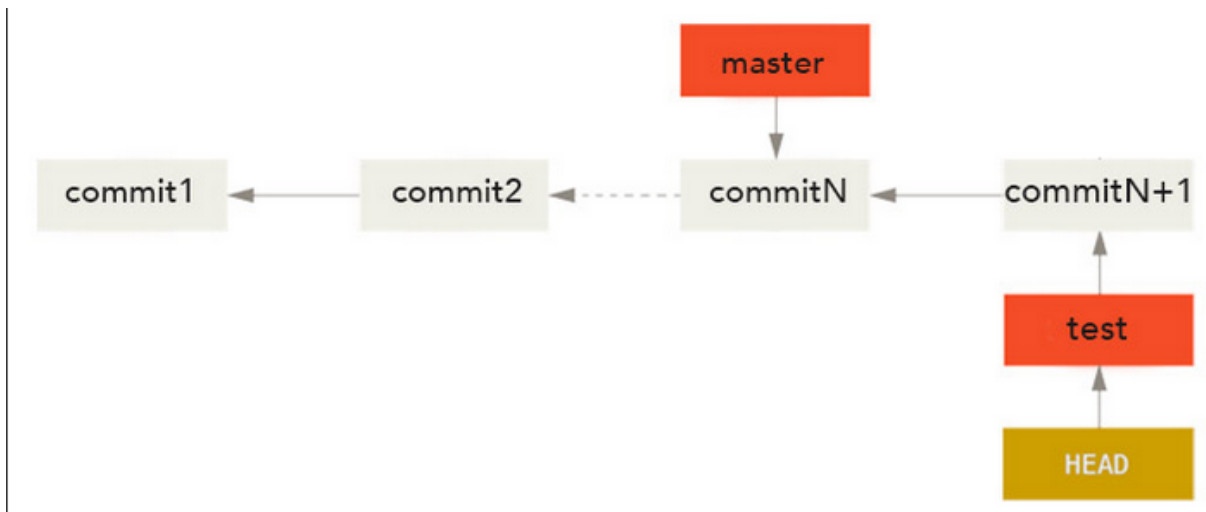
Note : On peut également utiliser `git checkout -b` pour créer une branche et basculer immédiatement dessus. Cela est l'équivalent d'utiliser `git branch` puis `git checkout`.

HEAD pointe maintenant vers notre branche test. Si on effectue un nouveau commit, la branche test va avancer automatiquement tandis que master va toujours pointer sur le commit précédent. C'est en effet la branche sur laquelle on se situe lors d'un commit qui va pointer sur ce commit.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ touch fichier3.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git add fichier3.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git commit -m "nouveau fichier ajouté"
[test 700b9a1] nouveau fichier ajouté
 2 files changed, 0 insertions(+), 0 deletions(-)
 rename README.txt => LISEZMOI.txt (100%)
 create mode 100644 fichier3.txt
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche test
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
       modifié :      fichier2.txt

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
       LISEZMOI.TXT

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
```



Basculer entre les branches

On peut revenir sur notre branche master en tapant à nouveau une commande git checkout master. Cela remplace le pointeur HEAD sur la branche master et restaure le répertoire de travail dans l'état de l'instantané pointé par le commit sur lequel pointe master.

```
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git checkout master
git: 'checkout' n'est pas une commande git. Voir 'git --help'.

La commande la plus ressemblante est
checkout
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$ git status
Sur la branche test
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :      fichier2.txt

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    LISEZMOI.TXT

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
fatma@mmip-HP-EliteBook-840-G2:~/Bureau/projet-git$
```


On va donc développer nos fonctionnalités sur des branches connexes et les tester jusqu'à ce qu'on soit sûrs qu'il n'y a pas de problème et on va finalement réintégrer ces fonctionnalités développées au sein de notre ligne de développement principale.

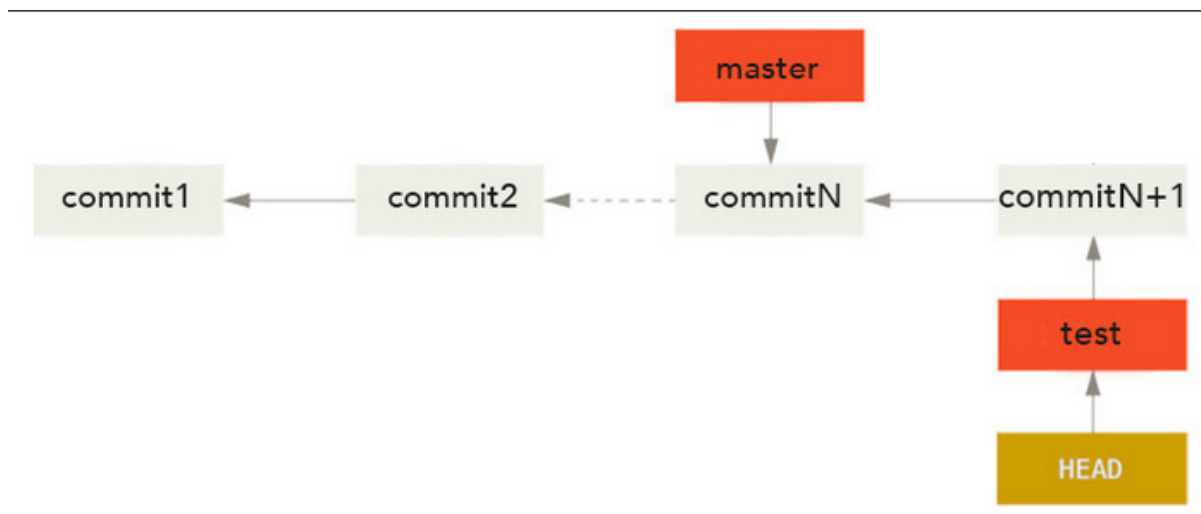
Pour faire cela, il va falloir rapatrier le contenu des branches créées dans la branche principale. On peut faire ça de deux manières avec Git : en fusionnant les branches.

Fusionner des branches

Commençons par nous concentrer sur la fusion de branches.

Dans la leçon précédente, on avait fini avec deux branches master et test divergentes. On parle de divergence car les deux branches possèdent un ancêtre commun mais pointent chacune vers de nouveaux commits qui peuvent correspondre à des modifications différentes d'un même fichier du projet.

Revenons un peu en arrière pour commencer avec un cas plus simple et imaginons que notre projet soit dans cet état :



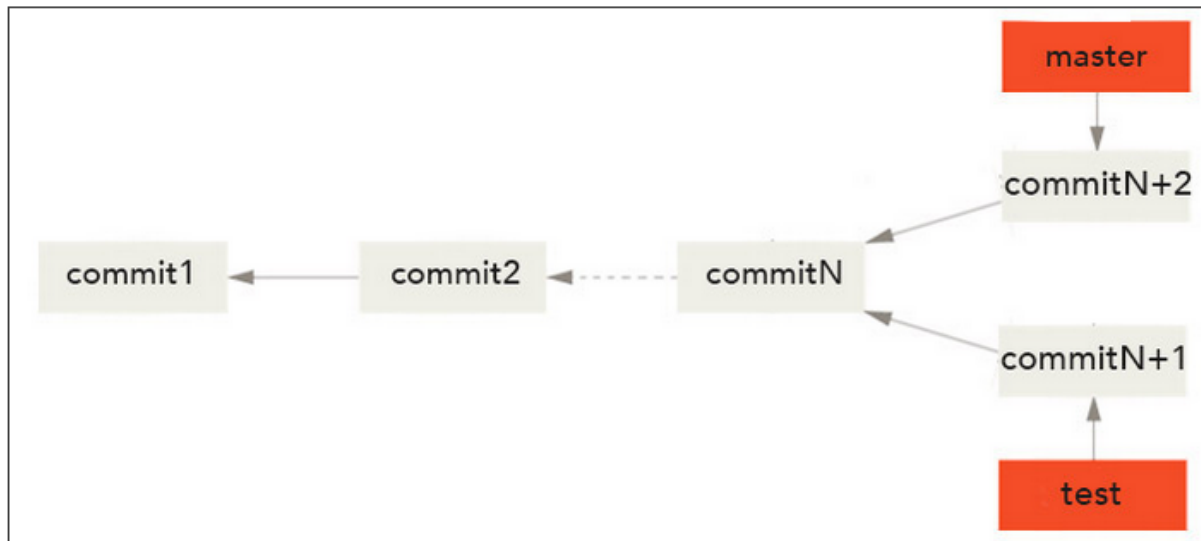
Ici, on a une branche **test** qui pointe sur un commit **commitN+1** et une branche **master** qui pointe sur un commit **commitN**. **commitN** est l'ancêtre direct de **commitN+1** et il n'y a donc pas de problème de divergence.

Pour fusionner nos deux branches, on va se placer sur **master** avec une commande `git checkout` puis taper une commande `git merge` avec le nom de la branche qu'on souhaite fusionner avec **master**.

Dans ce cas, "fusionner" nos deux branches revient finalement à faire avancer **master** au niveau du commit pointé par **test**. C'est exactement ce que fait Git et c'est ce qu'on appelle un "fast forward" (avance rapide).

Il ne nous reste alors plus qu'à effacer notre branche **test**. On peut faire cela en utilisant la commande `git branch -d`.

Reprenons maintenant la situation précédente avec deux branches dont les historiques divergent. On peut représenter cette situation comme cela :



Pour fusionner deux branches ici on va à nouveau se placer dans la branche dans laquelle on souhaite fusionner puis effectuer un `git merge`.

Ici, comme la situation est plus complexe, il me semble intéressant d'expliquer comment Git fait pour fusionner les branches. Dans ce cas, Git réalise une fusion en utilisant 3 sources : le dernier commit commun aux deux branches et le dernier commit de chaque branche.

Cette fois-ci, plutôt que de simplement faire un `fast forward`, Git crée automatiquement un nouvel instantané dont le contenu est le résultat de la fusion ainsi qu'un commit qui pointe sur cet instantané. Ce commit s'appelle un commit de fusion et est spécial puisqu'il possède plusieurs parents.

Notez que dans le cas d'une fusion à trois sources, il se peut qu'il y ait des conflits. Cela va être notamment le cas si une même partie d'un fichier a été modifiée de différentes manières dans les différentes branches. Dans ce cas, lors de la fusion, Git nous alertera du conflit et nous demandera de le résoudre avant de terminer la fusion des branches.

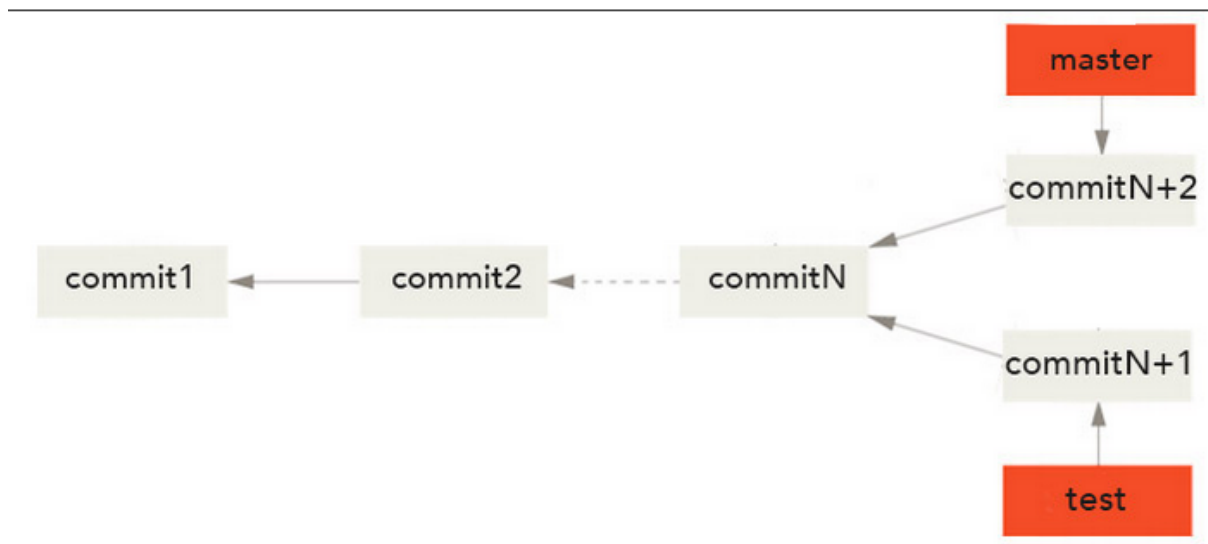
On peut utiliser une commande `git status` pour voir précisément quels fichiers sont à l'origine du conflit. Imaginons par exemple que nos deux branches possèdent un fichier `LISEZMOI.txt` et que les deux fichiers `LISEZMOI.txt` possèdent des textes différents. Git va automatiquement "fusionner" les contenus des deux fichiers en un seul qui va en fait contenir les textes des deux fichiers de base à la suite l'un de l'autre avec des indicateurs de séparation.

On peut alors ouvrir le fichier à la main et choisir ce qu'on conserve (en supprimant les parties qui ne nous intéressent pas par exemple). Dès qu'on a terminé l'édition, on va taper une commande `git add` pour marquer le conflit comme résolu. On n'aura alors plus qu'à effectuer un `git commit` pour terminer le commit de fusion.

Rebaser

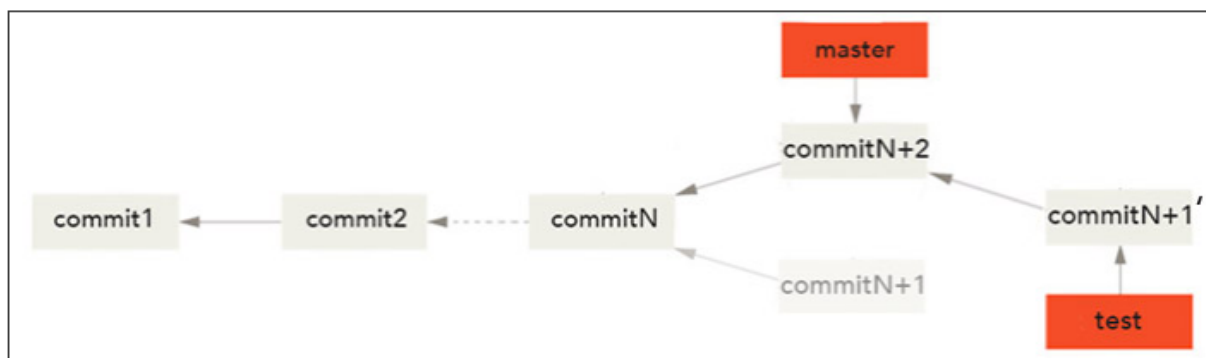
Git nous fournit deux moyens de rapatrier le travail effectué sur une branche vers une autre : on peut fusionner ou rebaser. Nous allons maintenant nous intéresser au rebasage, comprendre les différences entre la fusion et le rebasage et voir dans quelle situation utiliser une opération plutôt qu'une autre.

Reprenons notre exemple précédent avec nos deux branches divergentes.



Plutôt que d'effectuer une fusion à trois sources, on va pouvoir rebaser les modifications validées dans commitN+1 dans la branche master. On utilise la commande `git rebase` pour récupérer les modifications validées sur une branche et les rejouer sur une autre.

Dans ce cas, Git part à nouveau du dernier commit commun aux deux branches (l'ancêtre commun le plus récent) puis récupère les modifications effectuées sur la branche qu'on souhaite rapatrier et les applique sur la branche vers laquelle on souhaite rebaser notre travail dans l'ordre dans lequel elles ont été introduites.



Le résultat final est le même qu'avec une fusion mais l'historique est plus clair puisque toutes les modifications apparaissent en série même si elles ont eu lieu en parallèle. Rebaser rejoue les modifications d'une ligne de commits sur une autre dans l'ordre d'apparition, alors que la fusion joint et fusionne les deux têtes.

Gardez cependant à l'esprit que rebaser équivaut à supprimer des commits existants pour en créer de nouveaux (qui sont similaires de par leur contenu mais qui sont bien des entités différentes). Pour cette raison, vous ne devez jamais rebaser des commits qui ont déjà été poussés sur un dépôt public.

En effet, imaginons la situation suivante :

1. Vous poussez des commits sur un dépôt public ;
2. Des personnes récupèrent ces commits et se basent dessus pour travailler ;
3. Vous utilisez un git rebase pour "réécrire" ces commits et les poussez à nouveau.

Dans ce cas, des problèmes vont se poser puisque les gens qui ont travaillé à partir des commits de départ ne vont pas les retrouver dans le projet si il veulent récupérer les mises à jour et lorsqu'ils vont pousser leur modification sur le dépôt public les commits effacés vont être réintroduits ce qui va créer un historique très confus et potentiellement des conflits.

Jusqu'à présent, nous avons utilisé Git pour mettre en place un système de gestion de version local. En pratique, vous allez souvent être amené à travailler à plusieurs sur un même projet.

Pour collaborer sur un projet et mettre en place un système de gestion de version, vous allez devoir utiliser des dépôts distants c'est-à-dire des dépôts hébergés sur serveur distant (serveurs accessibles via Internet ou via votre réseau d'entreprise).

La gestion de dépôts distants est plus complexe que la gestion d'un dépôt local puisqu'il va falloir gérer les permissions des différents utilisateurs, définir quelles modifications doivent être adoptées ou pas, etc.

Dans le cas où on travaille à plusieurs sur un dépôt distant, nous n'allons jamais directement modifier le projet distant. Nous allons plutôt cloner le dépôt distant localement (sur ordinateur donc), effectuer nos modifications puis les pousser vers le dépôt distant. Ces modifications pourront ensuite être acceptées ou pas.

Cloner, afficher et renommer un dépôt distant

Pour cloner un dépôt distant, nous allons pouvoir utiliser la commande git clone. Une fois que vous disposez d'un dépôt distant cloné localement, vous pouvez utiliser la commande git remote pour afficher la liste des serveurs distants des dépôts.

Le nom donné par défaut par Git à un serveur à partir duquel on a effectué un clonage est origin. On peut également utiliser git remote -v pour afficher les URLs stockées.

Pour définir un nom personnalisé pour un dépôt distant, on peut utiliser la commande git remote add suivi du nom choisi suivi de l'URL du dépôt. Pour renommer ensuite notre référence, on pourra utiliser git remote rename [ancien-nom] [nouveau-nom].

Récupérer des données depuis un dépôt distant

La commande `git fetch [nom-distant]` va nous permettre de récupérer toutes les données d'un dépôt distant qu'on ne possède pas déjà. Cela permet de récupérer les ajouts du projet distant et d'avoir une version à jour du projet.

Notez que `fetch` tire les données dans votre dépôt local mais sous sa propre branche ce qui signifie que ces données ne sont pas fusionnées avec vos autres travaux et que votre copie de travail n'est pas modifiée. Il faudra donc fusionner explicitement les données tirées par `fetch` par la suite.

Dans le cas où on a créé une branche sur notre projet pour suivre les avancées d'une branche distante, on peut également utiliser `git pull` qui va récupérer les données d'une branche distante et les fusionner automatiquement dans notre branche locale.

Pousser des modifications vers un dépôt distant

Une fois qu'on a terminé nos modifications localement, on va les pousser vers le dépôt distant. On utilise pour cela la commande `git push [nom-distant] [nom-de-branche]`. Si on souhaite par exemple pousser les modifications faites sur notre branche `master` vers le serveur `origin`, on écrira `git push origin master`.

Notez que cette commande ne fonctionnera que si vous avez cloné depuis un serveur sur lequel vous avez des droits d'accès en écriture et si personne n'a poussé dans l'intervalle. Cela évite des conflits et d'effacer certaines modifications dans le cas où deux personnes auraient modifié différemment les mêmes fichiers par exemple.

Obtenir des informations sur un dépôt distant

Pour obtenir des informations sur un dépôt distant, on peut utiliser la commande `git remote show [nom-du-depot]`.

Cette commande peut notamment fournir la liste des URL pour le dépôt distant ainsi que la liste des branches distantes suivies, les branches poussées automatiquement avec une commande `git push`, les branches qui seront fusionnées avec un `git pull`, etc.

Retirer un dépôt distant

Pour retirer un dépôt distant, on utilisera la commande `git remote rm [nom-du-depot]`.

8 La plateforme GitHub

GitHub est la plus grande plateforme d'hébergement de projets Git au monde. Vous serez probablement amené à travailler avec GitHub et il est donc important de comprendre comment ce service fonctionne.

Commencez déjà par noter que GitHub est un outil gratuit pour héberger du code open source, et propose également des plans payants pour les projets de code privés.

Pour utiliser GitHub, il suffit de créer un compte gratuitement sur le site <https://github.com>.

Le grand intérêt de GitHub est de faciliter la collaboration à une échelle planétaire sur les projets : n'importe qui va pouvoir récupérer des projets et y contribuer (sauf si le propriétaire du projet ne le permet pas bien entendu).

Sur GitHub, nous allons en effet notamment pouvoir cloner des projets (dépôts) publics, dupliquer ("fork") des projets ou encore contribuer à des projets en proposant des modifications ("pull request").

De GitHub à Git et inversement

GitHub permet de contribuer simplement à des projets ou de laisser les gens contribuer à ces propres projets.

La plupart des gens qui utilisent GitHub vont cependant souvent préférer travailler hors ligne (en local; sur leur machine) plutôt que de devoir être constamment connecté à GitHub et de devoir passer par cette plateforme pour effectuer toutes les opérations.

Pour travailler localement, il suffit de cloner un projet après l'avoir forké. On va ensuite pouvoir effectuer nos différentes modifications sur notre machine.

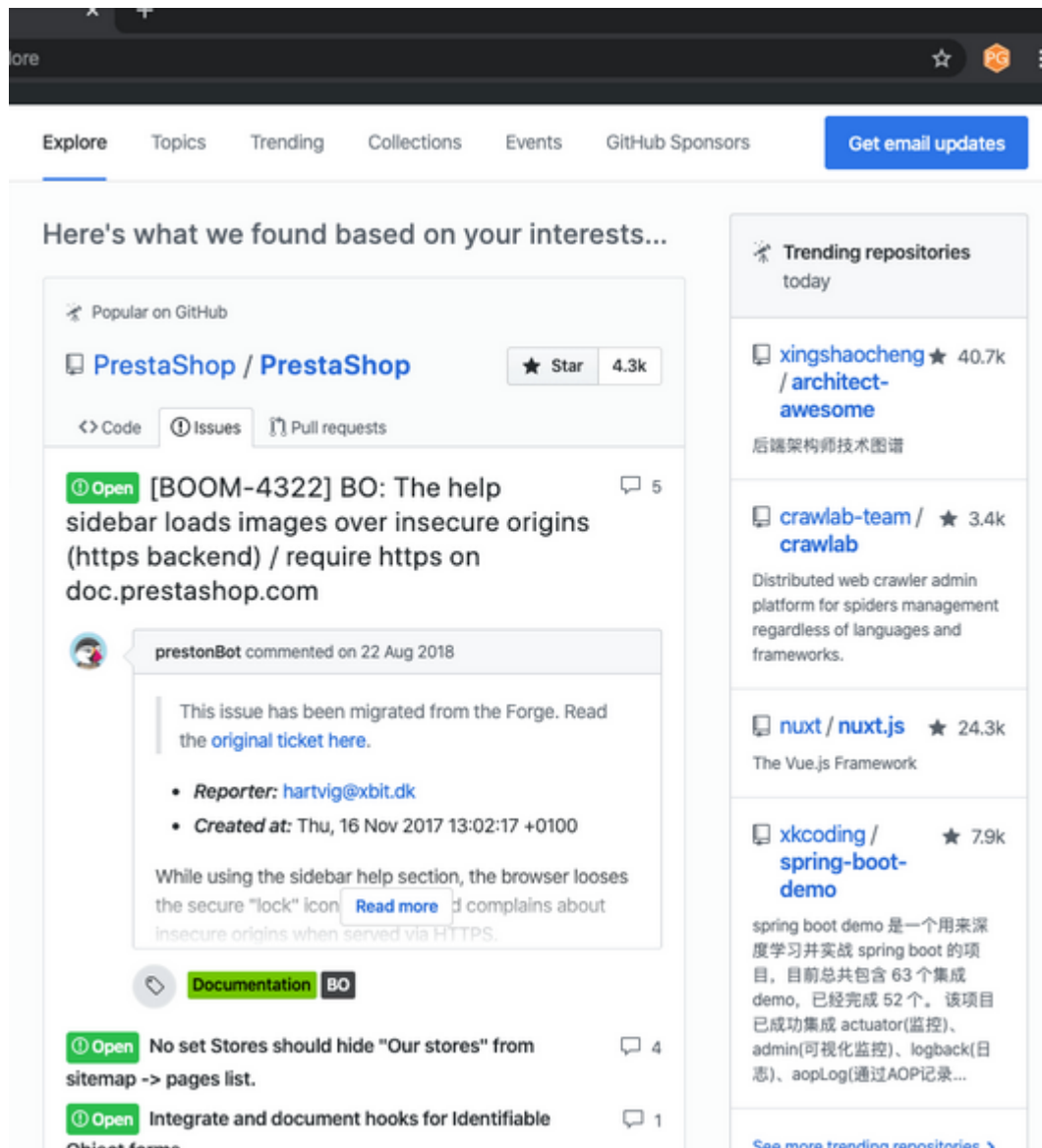
Pour synchroniser les modifications faites depuis notre machine avec notre dépôt distant (dépôt GitHub), il suffit de faire un git commit depuis le dépôt local et de push (envoyer) les modifications sur le dépôt GitHub à l'aide de la commande `git push [nom-distant] [nom-local]`. Taper `git push origin master` par exemple revient à envoyer les modifications situées dans ma branche master vers origin.

Pour récupérer en local les dernières modifications du dépôt GitHub, on va utiliser la commande `git pull [nom-distant] [nom-local]`.

Contribuer à un projet avec GitHub ou le copier

Sur GitHub, nous allons pouvoir contribuer aux projets publics d'autres personnes ou créer nos propres dépôts publics afin que d'autres personnes contribuent à nos propres projets. Commençons déjà par nous familiariser avec l'aspect contributeur de GitHub.

GitHub est une gigantesque plateforme collaborative qui héberge des dépôts Git. Pour rechercher des dépôts auxquels contribuer ou pour rechercher des fonctionnalités intéressantes qu'on aimerait récupérer, on peut aller dans l'onglet "explore" ou chercher un dépôt en particulier grâce à la barre de recherche en haut du site.



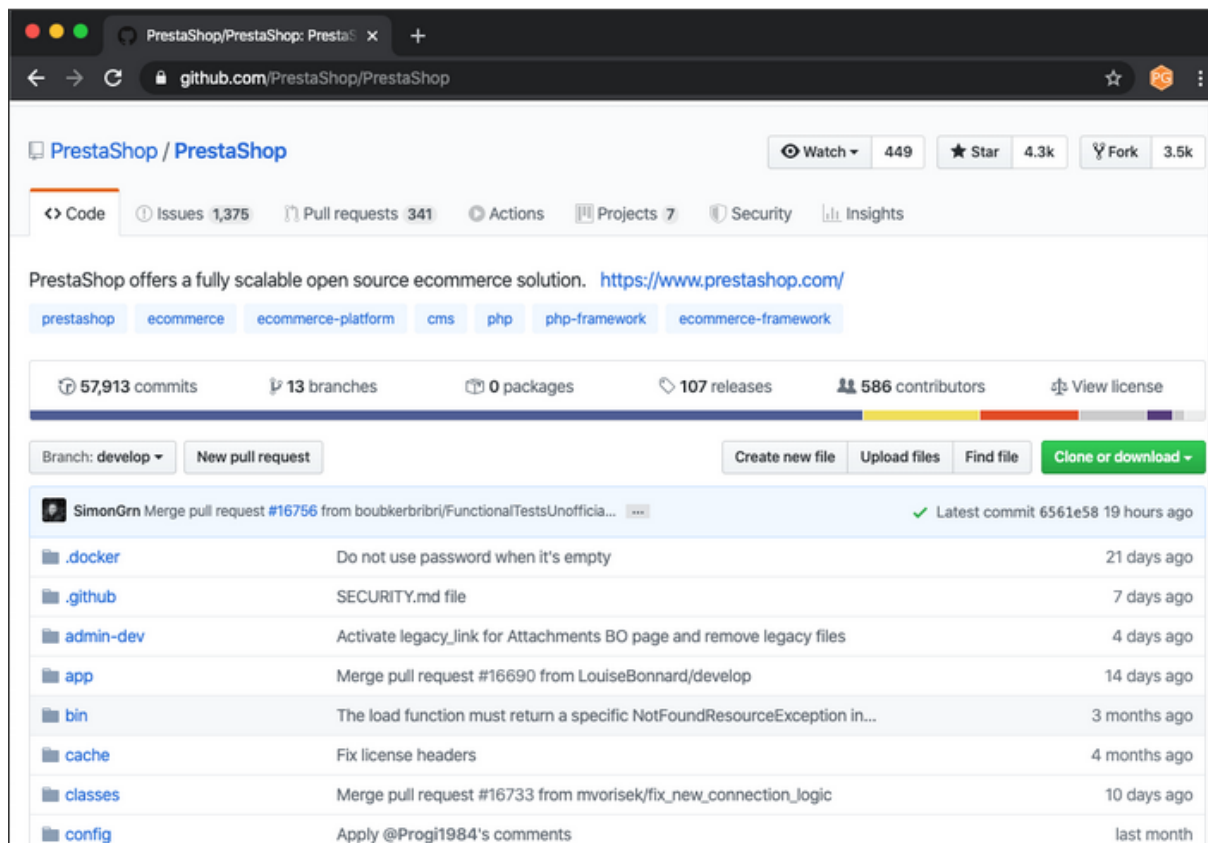
Les étapes pour contribuer à un projet (le cycle de travail) vont toujours être les mêmes :

1. On copie un projet sur notre espace GitHub ;
2. On crée une branche thématique à partir de master ;
3. On effectue nos modifications / améliorations au projet ;
4. On pousse ensuite la branche locale vers le projet qu'on a copié sur GitHub et on ouvre une requête de tirage depuis GitHub ;
5. Le propriétaire du projet choisit alors de refuser nos modifications ou de les fusionner dans le projet d'origine ;
6. On met à jour la version du projet en récupérant les dernières modifications à partir du projet d'origine.

Copier un dépôt : clone vs fork

Pour copier un dépôt (repository) GitHub sur nos machines, il suffit d'utiliser l'option "clone URL" de GitHub pour récupérer le lien du repo puis d'utiliser la commande `git clone [URL]` dans notre console.

On peut également utiliser l'option "fork" de GitHub. Un fork est une copie d'un dépôt distant qui nous permet d'effectuer des modifications sans affecter le projet original.



La différence entre un fork et un clone est que lorsqu'on fork une connexion existe entre notre fork (notre copie) et le projet original. Cela permet notamment de pouvoir très simplement contribuer au projet original en utilisant des pull requests, c'est-à-dire en poussant nos modifications vers le dépôt distant afin qu'elles puissent être examinées par l'auteur du projet original.

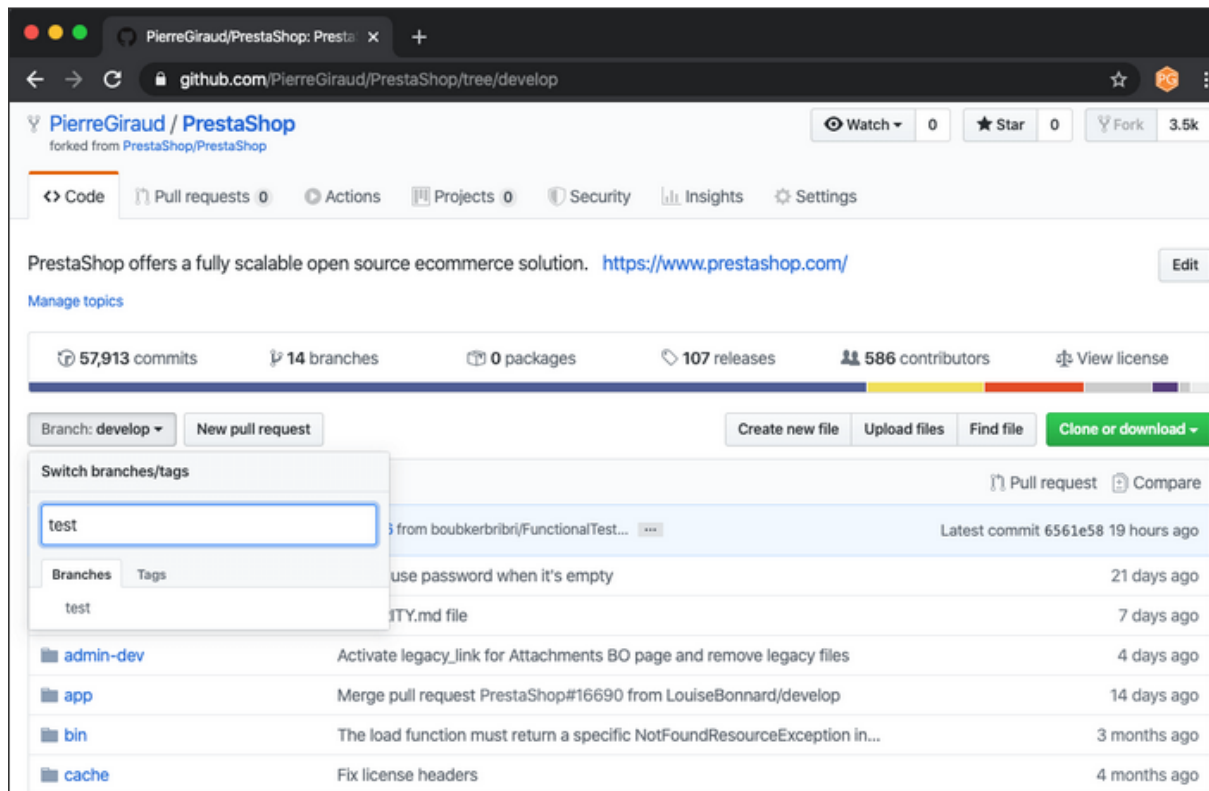
Lorsqu'on clone un projet, on ne va pas pouvoir ensuite récupérer les changements à partir du projet d'origine ni contribuer à ce projet à moins que le propriétaire du projet d'origine ne nous accorde des droits spéciaux (privilèges).

Le fork est une fonctionnalité très utile puisqu'elle permet à n'importe quelle personne de pouvoir dupliquer un projet et de contribuer à ce projet tout en garantissant à l'auteur du projet l'intégrité du projet original car ce sera à lui de valider ou pas les différentes pull requests (requêtes de tirage) des contributeurs.

Créer une branche thématique à partir de la branche principale

Pour contribuer à un projet, on va donc très souvent le copier en le forkant. Cela crée une copie du projet dans notre espace GitHub. On va ensuite créer une branche thématique et effectuer nos modifications.

Pour cela, une fois sur la page du projet forké dans notre espace personnel Git, on va cliquer sur le bouton de liste "branch" et ajouter un nom pour créer une nouvelle branche.



Rappelez vous qu'on utilise les branches pour expérimenter et apporter des modifications sans polluer notre branche principale (master par défaut). Lorsqu'on crée une branche à partir de la branche master, on effectue une copie (un instantané ou snapshot) de master telle qu'elle était à ce moment-là. Si quelqu'un d'autre apporte des modifications à la branche principale pendant qu'on travaille sur notre branche, il faudra qu'on récupère ces mises à jour.

Note importante : On peut également bien évidemment cloner le projet localement (sur notre machine) afin de pouvoir travailler dessus hors connexion puis renvoyer ensuite les modifications vers notre projet forké. Dans cette partie, je vais cependant me concentrer sur ce qu'il est possible de faire depuis GitHub seulement.

Effectuer des modifications au projet

Pour modifier un fichier, on va cliquer sur le fichier en question puis sur l'icône en forme de crayon à droite. Cela ouvre un éditeur.

Branch: test ▾ PrestaShop / README.md Find file Copy path

Roman3349 Update Travis CI status image 1c4d0c7 on 19 Oct

20 contributors

148 lines (96 sloc) 8.28 KB Raw Blame History

About PrestaShop

build passing chat on gitter

PrestaShop is an Open Source e-commerce web application, committed to providing the best shopping cart experience for both merchants and customers. It is written in PHP, is highly customizable, supports all the major payment services, is translated in many languages and localized for many countries, has a fully responsive design (both front and back office), etc. [See all the available features.](#)

Code Pull requests 0 Actions Projects 0 Security Insights Settings

PrestaShop / README.md Cancel

Edit file Preview changes Spaces 2 Soft wrap

```
112 Themes and modules can be obtained (and sold!) on \[PrestaShop Addons\]\[addons\], the official marketplace for PrestaShop.
113
114
115
116 Community forums
117 -----
118
119 You can discuss about e-commerce, help other merchants and get help, and contribute to improving PrestaShop together with
the PrestaShop community on \[the PrestaShop forums\]\[forums\].
```

On peut alors modifier les fichiers puis commit nos modifications dès qu'on les juge satisfaisantes. Pour cela, il suffit de renseigner un message de commit et de cliquer sur le bouton "commit changes" en bas de la page.

Commit changes

blablabla

Add an optional extended description...

☒ Commit directly to the test branch.

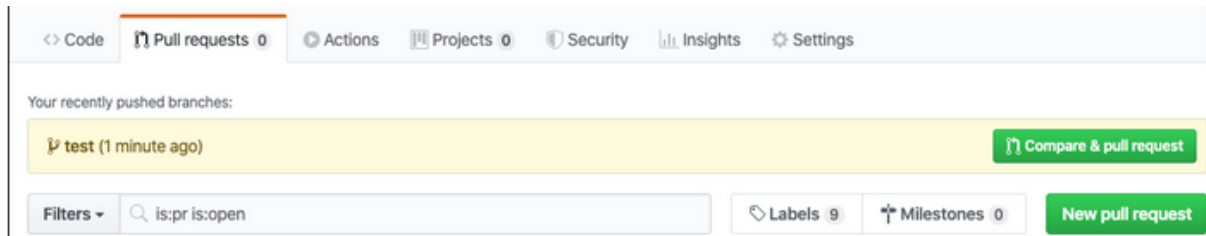
☐ Create a new branch for this commit and start a pull request. [Learn more about pull requests.](#)

Commit changes Cancel

Pousser nos modifications : pull request

Une fois qu'on a terminé nos modifications, on va pouvoir les proposer. Pour cela, on va ouvrir une requête de tirage ou "pull request".

Pour effectuer un pull request, on clique sur l'onglet "pull requests" puis sur "compare & pull request" ou sur "new pull request".



Effectuer une requête de tirage correspond à demander à quelqu'un d'examiner et d'extraire votre contribution et de la fusionner dans sa branche. Les demandes d'extraction montrent des différences de contenu des deux branches. Les modifications, ajouts et soustractions sont affichés en vert et en rouge.

Suite à un pull request, le propriétaire du projet examine notre contribution et une discussion peut s'engager si il souhaite qu'on effectue d'autres modifications.

Fusionner notre pull request

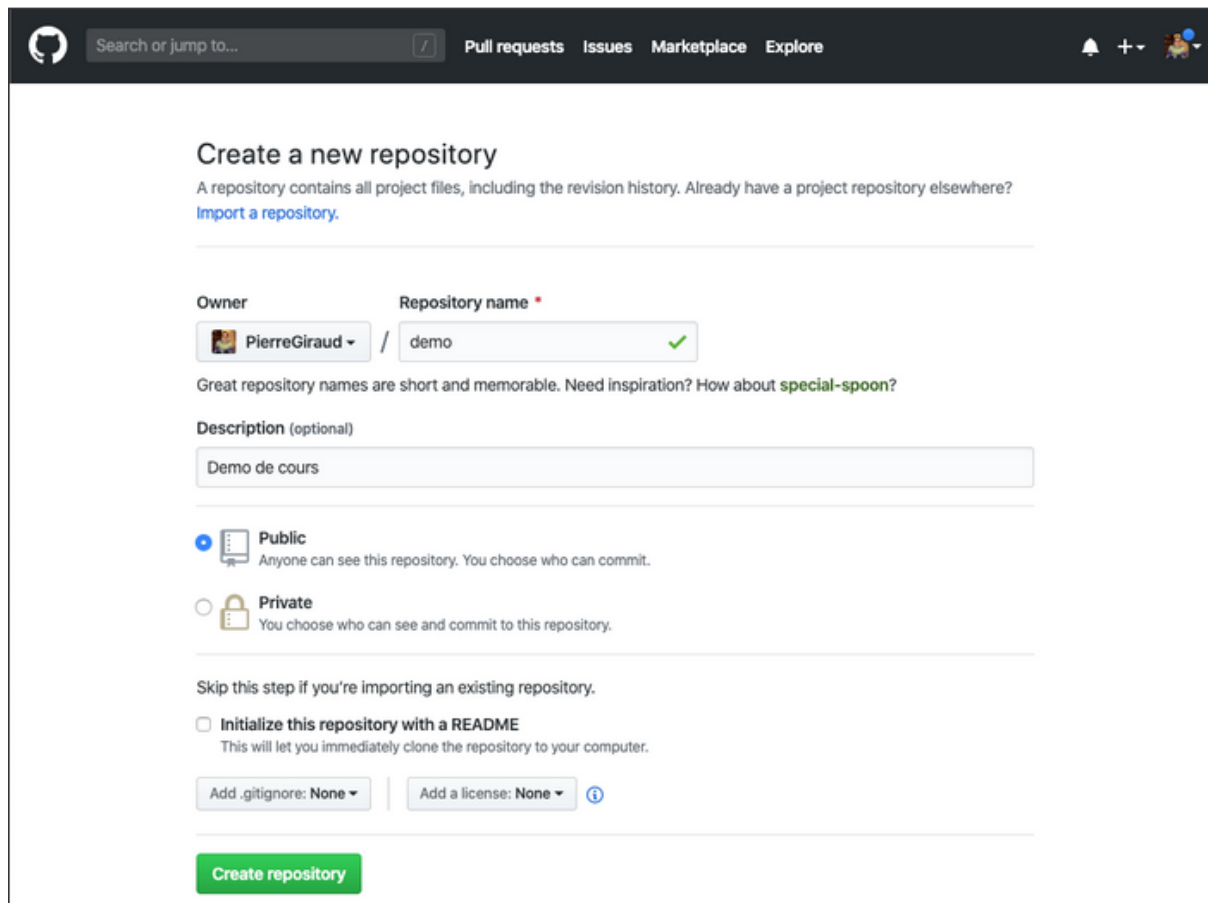
La dernière étape du processus de travail correspond à fusionner notre branche thématique dans notre branche principale afin d'incorporer les changements effectués et d'avoir une branche principale à jour.

Pour cela, on peut cliquer sur le bouton "merge request" suite à notre pull request.

Créer un dépôt GitHub


La deuxième face de GitHub correspond à la création de dépôt sur GitHub afin que des gens collaborent dessus. Pour créer un nouveau dépôt sur GitHub, il suffit de cliquer sur l'icône "+" située en haut à droite puis sur "new repository". Une page s'ouvre vous permettant de créer un nouveau dépôt.

Note : vous pouvez également importer un dépôt en cliquant sur "import repository".



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository.](#)

Owner:  PierreGiraud / Repository name:

Great repository names are short and memorable. Need inspiration? How about **special-spoon**?

Description (optional):

☒ **Public**
Anyone can see this repository. You choose who can commit.

☐ **Private**
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☐ **Initialize this repository with a README**
This will let you immediately clone the repository to your computer.

Add .gitignore: **None** | Add a license: **None** ⓘ

Create repository

Sur la page de création de dépôt, vous devez renseigner un nom et une description pour le dépôt. Vous avez également une option « Initialise with a README », qui vous permet de cloner votre dépôt sur votre machine. Cette option est à cocher uniquement dans le cas où vous n'avez pas encore créé le dépôt en question sur votre machine.

9 Gérer des dépôts distant

Jusqu'à présent, nous avons utilisé Git /Github pour mettre en place un système de gestion de version local. En pratique, vous allez souvent être amené à travailler à plusieurs sur un même projet.

Pour collaborer sur un projet et mettre en place un système de gestion de version, vous allez devoir utiliser des dépôts distants c'est-à-dire des dépôts hébergés sur serveur distant (serveurs accessibles via Internet ou via votre réseau d'entreprise).

La gestion de dépôts distants est plus complexe que la gestion d'un dépôt local puisqu'il va falloir gérer les permissions des différents utilisateurs, définir quelles modifications doivent être adoptées ou pas, etc.

Dans le cas où on travaille à plusieurs sur un dépôt distant, nous n'allons jamais directement modifier le projet distant. Nous allons plutôt cloner le dépôt distant localement (sur notre

ordinateur donc), effectuer nos modifications puis les pousser vers le dépôt distant. Ces modifications pourront ensuite être acceptées ou pas.

Cloner, afficher et renommer un dépôt distant

Pour cloner un dépôt distant, nous allons pouvoir utiliser la commande `git clone`. Une fois que vous disposez d'un dépôt distant cloné localement, vous pouvez utiliser la commande `git remote` pour afficher la liste des serveurs distants des dépôts.

Le nom donné par défaut par Git à un serveur à partir duquel on a effectué un clonage est `origin`. On peut également utiliser `git remote -v` pour afficher les URLs stockées.

Pour définir un nom personnalisé pour un dépôt distant, on peut utiliser la commande `git remote add` suivi du nom choisi suivi de l'URL du dépôt. Pour renommer ensuite notre référence, on pourra utiliser `git remote rename [ancien-nom] [nouveau-nom]`.

Récupérer des données depuis un dépôt distant

La commande `git fetch [nom-distant]` va nous permettre de récupérer toutes les données d'un dépôt distant qu'on ne possède pas déjà. Cela permet de récupérer les ajouts du projet distant et d'avoir une version à jour du projet.

Notez que `fetch` tire les données dans votre dépôt local mais sous sa propre branche ce qui signifie que ces données ne sont pas fusionnées avec vos autres travaux et que votre copie de travail n'est pas modifiée. Il faudra donc fusionner explicitement les données tirées par `fetch` par la suite.

Dans le cas où on a créé une branche sur notre projet pour suivre les avancées d'une branche distante, on peut également utiliser `git pull` qui va récupérer les données d'une branche distante et les fusionner automatiquement dans notre branche locale.

Pousser des modifications vers un dépôt distant

Une fois qu'on a terminé nos modifications localement, on va les pousser vers le dépôt distant. On utilise pour cela la commande `git push [nom-distant] [nom-de-branche]`. Si on souhaite par exemple pousser les modifications faites sur notre branche `master` vers le serveur original, on écrira `git push origin master`.

Notez que cette commande ne fonctionnera que si vous avez cloné depuis un serveur sur lequel vous avez des droits d'accès en écriture et si personne n'a poussé dans l'intervalle. Cela évite des conflits et d'effacer certaines modifications dans le cas où deux personnes auraient modifié différemment les mêmes fichiers par exemple.

Obtenir des informations sur un dépôt distant

Pour obtenir des informations sur un dépôt distant, on peut utiliser la commande `git remote show [nom-du-depot]`.

Cette commande peut notamment fournir la liste des URL pour le dépôt distant ainsi que la liste des branches distantes suivies, les branches poussées automatiquement avec une commande `git push`, les branches qui seront fusionnées avec un `git pull`, etc.

Retirer un dépôt distant

Pour retirer un dépôt distant, on utilisera la commande `git remote rm [nom-du-depot]`.

10 Gérer les dépôts distants

GitHub est la plus grande plateforme d'hébergement de projets Git au monde. Vous serez probablement amené à travailler avec GitHub et il est donc important de comprendre comment ce service fonctionne.

Commencez déjà par noter que GitHub est un outil gratuit pour héberger du code open source, et propose également des plans payants pour les projets de code privés.

Pour utiliser GitHub, il suffit de créer un compte gratuitement sur le site <https://github.com>.

Le grand intérêt de GitHub est de faciliter la collaboration à une échelle planétaire sur les projets : n'importe qui va pouvoir récupérer des projets et y contribuer (sauf si le propriétaire du projet ne le permet pas bien entendu).

Sur GitHub, nous allons en effet notamment pouvoir cloner des projets (dépôts) publics, dupliquer ("fork") des projets ou encore contribuer à des projets en proposant des modifications ("pull request").

