Git

**Table des matières**

[Prérequis formateur 3](#__RefHeading___Toc823_3892772214)

[Création d’un personal token sur github 4](#__RefHeading___Toc893_1392315544)

[NOTE formateur : la procédure est susceptible de changer avec le temps, vérifier en amont du cours et la rectifier si besoin. 4](#__RefHeading___Toc895_1392315544)

[Voir fichier « Git / Create Personal Token » 4](#__RefHeading___Toc897_1392315544)

[Pourquoi Git ? 5](#__RefHeading___Toc513_3892772214)

[Le repository 6](#__RefHeading___Toc515_3892772214)

[Un commit 7](#__RefHeading___Toc517_3892772214)

[Local et remote server 8](#__RefHeading___Toc519_3892772214)

[Configurer git 9](#__RefHeading___Toc521_3892772214)

[Configurer l’auteur des commit 9](#__RefHeading___Toc523_3892772214)

[Configurer le cache des credentials 9](#__RefHeading___Toc525_3892772214)

[Configurer le comportement par défaut du pull et du merge 9](#__RefHeading___Toc899_1392315544)

[Afficher la configuration 9](#__RefHeading___Toc527_3892772214)

[Cloner un repo existant 10](#__RefHeading___Toc529_3892772214)

[Voir l’historique des commits 11](#__RefHeading___Toc531_3892772214)

[Créer / Modifier / Supprimer un fichier 12](#__RefHeading___Toc533_3892772214)

[Voir le statut des fichiers 13](#__RefHeading___Toc535_3892772214)

[Préparer un commit 14](#__RefHeading___Toc537_3892772214)

[Créer un commit 16](#__RefHeading___Toc539_3892772214)

[Envoyer les changements sur le serveur distant 18](#__RefHeading___Toc541_3892772214)

[Récupérer les changements depuis le serveur distant 20](#__RefHeading___Toc543_3892772214)

[Les branches 23](#__RefHeading___Toc545_3892772214)

[Créer une branche 24](#__RefHeading___Toc547_3892772214)

[Aller sur une branche existante 25](#__RefHeading___Toc549_3892772214)

[Codons un peu 26](#__RefHeading___Toc551_3892772214)

[Fusionner une branche dans une autre 27](#__RefHeading___Toc553_3892772214)

[Comment revenir en arrière avec git 29](#__RefHeading___Toc555_3892772214)

[Enlever les fichiers ajoutés au prochain commit 29](#__RefHeading___Toc829_3892772214)

[Supprimer toutes les modifications de code 30](#__RefHeading___Toc858_3892772214)

[Revenir définitivement à un commit 31](#__RefHeading___Toc852_3892772214)

[Avec git, rien n’est perdu 33](#__RefHeading___Toc854_3892772214)

[Gitignore 34](#__RefHeading___Toc901_1392315544)

[Dans nos projets, certaines choses ne doivent aucunement être commités 34](#__RefHeading___Toc903_1392315544)

[- Les mots de passe (.env) 34](#__RefHeading___Toc905_1392315544)

[- Les dépendances (node\_modules, vendor) 34](#__RefHeading___Toc907_1392315544)

[- Les builds (dist) 34](#__RefHeading___Toc909_1392315544)

[Pour dire à git d’ignorer ces fichiers / dossiers, on met à la racine du projet un fichier appelé « .gitignore ». Sans oublier le point devant. 34](#__RefHeading___Toc911_1392315544)

[A l’intérieur, on y met les fichiers / dossiers concernés 34](#__RefHeading___Toc913_1392315544)

[Gérer les versions 35](#__RefHeading___Toc915_1392315544)

[Gérer les versions avec git 36](#__RefHeading___Toc943_1392315544)

[Pour aller plus loin 37](#__RefHeading___Toc856_3892772214)

# Prérequis formateur

Ce cours est à destination des apprenants n’ayant jamais utilisé cet outil.

Il est interactif, c’est à dire que les étudiants pratiquent avec le formateur.

Il sera réalisé avec github.

Le formateur devra créer dans son environnement github un repo « mon-projet-git » de visibilité public qui servira de fil rouge, avec le contenu du zip « mon-projet-git.zip ». Il sera partagé avec les apprenants (à la manière d’une équipe de travail).

Il devra y ajouter tous les apprenants de la session : pour se faire aller sur le repo créé > Paramètres (Settings) > Collaboration (Collaborators).

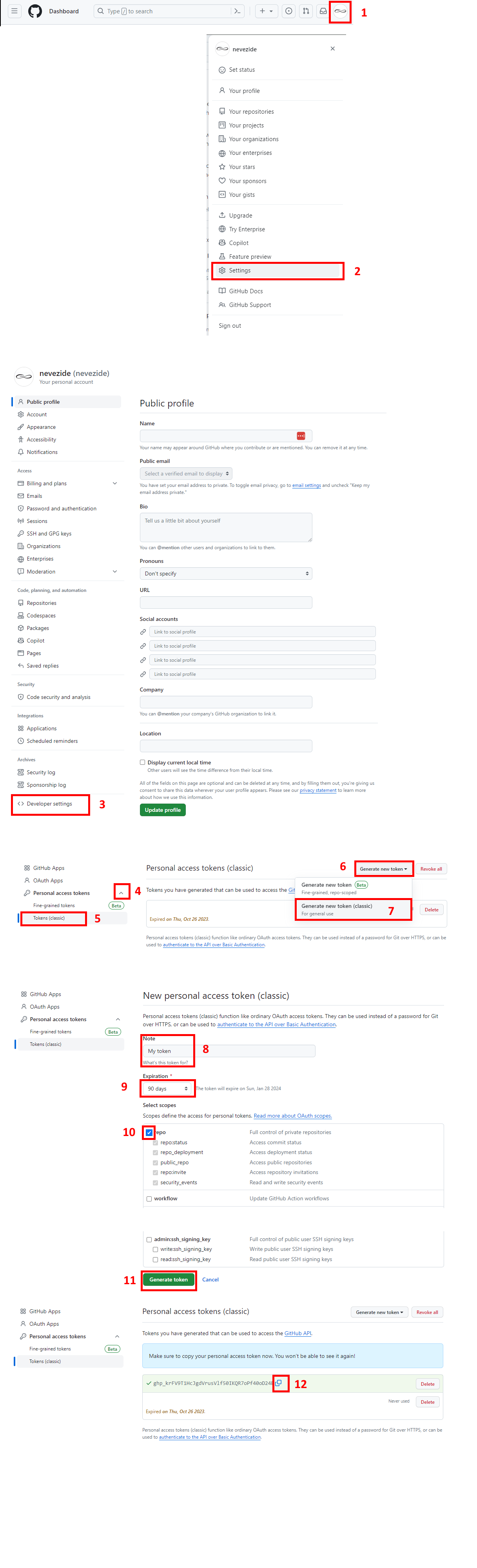
Enfin, github n’autorisant plus l’utilisation des mots de passe lors des interactions avec le serveur, un personal token devra être créé ([voir procédure de création au prochain chapitre](#_toc197))

Timing : 0.5J pour les bases + 0.5J pour les branches / revenir en arrière

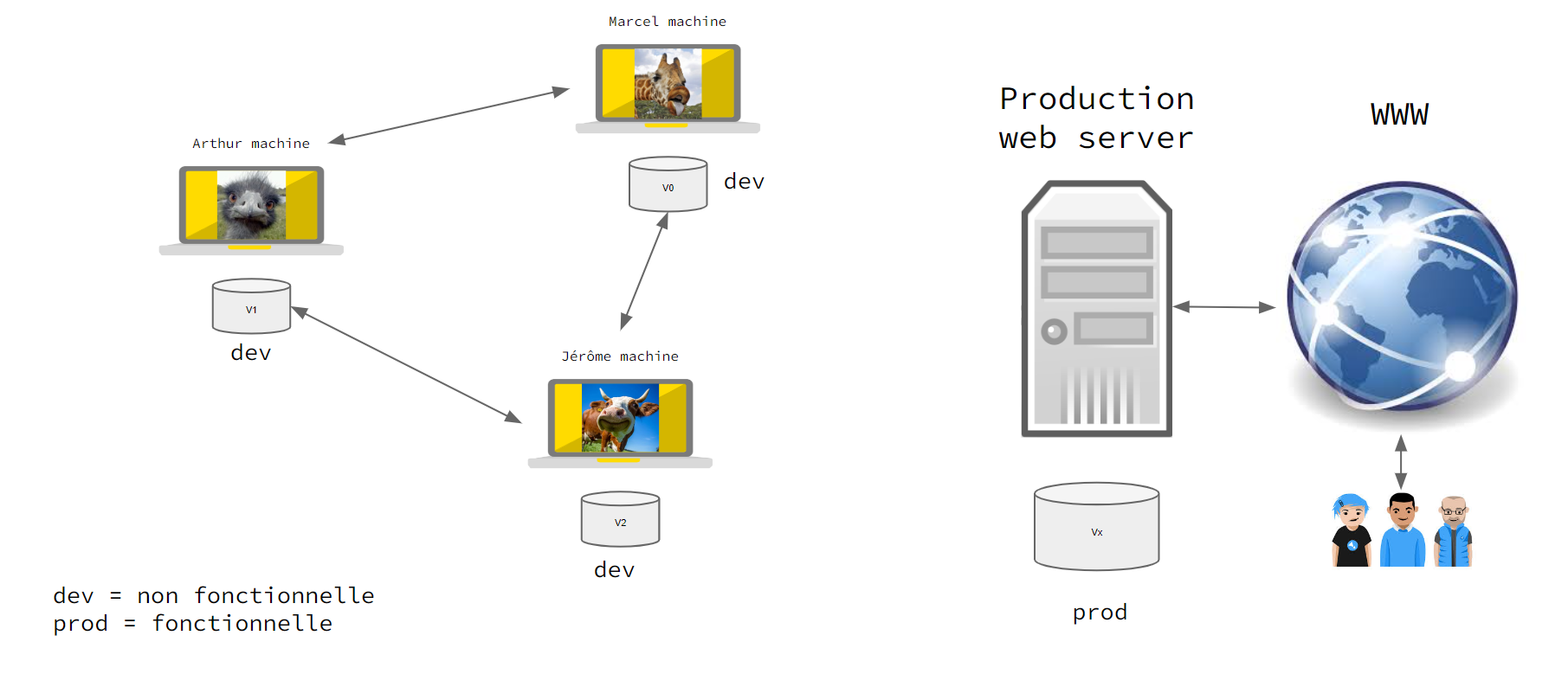
Pour tout complément d’information, manque de clarté, problème avec ce cours, envoyez un courriel à son auteur Fabrice VALLEE : nevezide44@gmail.com

# Création d’un personal token sur github

NOTE formateur : la procédure est susceptible de changer avec le temps, vérifier en amont du cours et la rectifier si besoin.

Voir fichier « Git / Create Personal Token »

# Pourquoi Git ?



* Sauvegarder son code dans un endroit sécurisé, sur un serveur distant
* Centraliser et partager le code
* Historisation des changements (avec leur auteur) et revenir en arrière si besoin :)
* Séparer le code en cours de développement de celui prêt à mettre en production
* …

# Le repository



Un repo git est une base de données qui contient :

* Les versions de chaque fichier dans leur intégralité (même si seulement un mot a été modifié)
* Les versions de l’arborescence

Un repo git est une base de données où :

* Tout est historisé : ajout / modifications et suppressions de fichiers et dossiers
* Rien n’est supprimé physiquement, on peut donc toujours revenir en arrière.

# Un commit

Un commit est comme une capture d’écran du code.

L’état de chaque fichier modifié est intégralement copié.

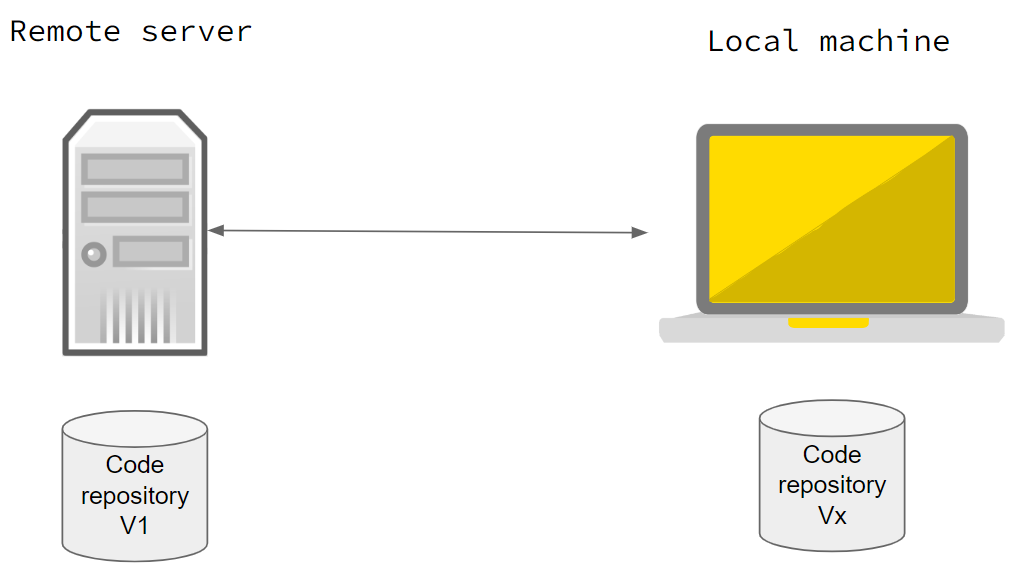
Il contient 3 éléments :

* L’arborescence des dossiers où sont placés les fichiers
* Le contenu des fichiers dans leur intégralité
* Un hash qui sert d’identifiant unique du commit. Ce hash est calculé à partir du contenu du commit.

**Commit et push régulièrement**

*au moins 1 fois par jour, le soir avant de partir*

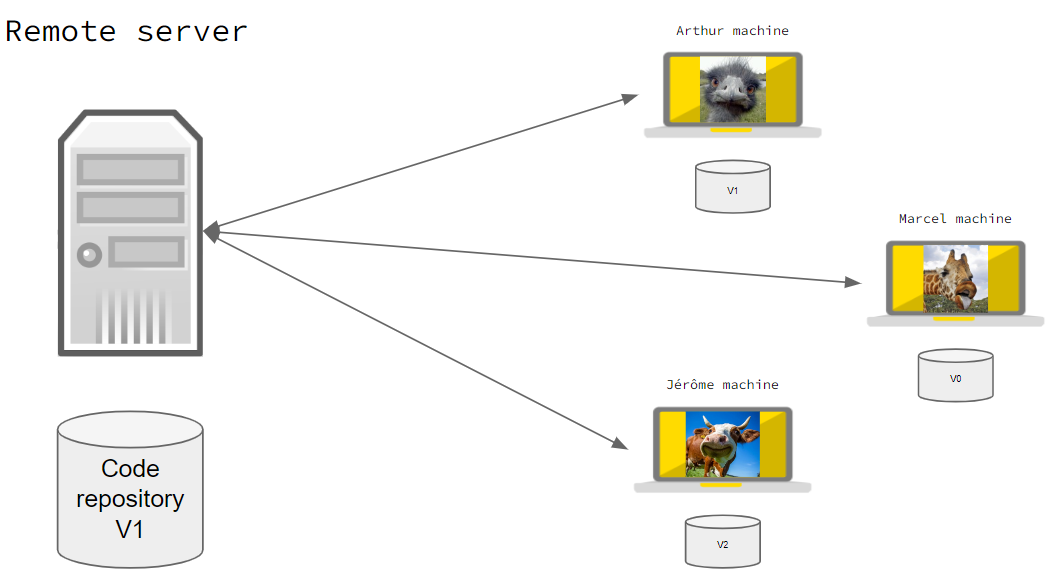
# Local et remote server



On ne travaille pas directement sur le code du serveur distant.

Avant de commencer à coder, le développeur va faire une copie du repo depuis le serveur sur sa machine.

Puis, lorsqu’on créé un commit, il est créé dans le repo local, cad sur la machine du développeur.



Ainsi, chaque développeur a une version différente du repo.

# Configurer git

**git config --global <param name> <value>**

## Configurer l’auteur des commit

Saisir les 2 commandes suivantes :

**git config --global user.name "fabrice.vallee"**

**git config --global user.email "fabrice.vallee@3wa.io"**



## Configurer le cache des credentials

Configure le stockage temporaire des credentials (login / mot de passe) durant 15min et ainsi éviter de les saisir à chaque interaction avec le serveur.

Saisir la commande suivante :

**git config --global credential.helper cache**

## Configurer le comportement par défaut du pull et du merge

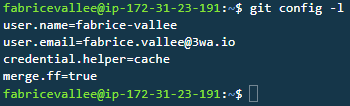
On verra plus loin ce que c’est.

**git config --global merge.ff true**

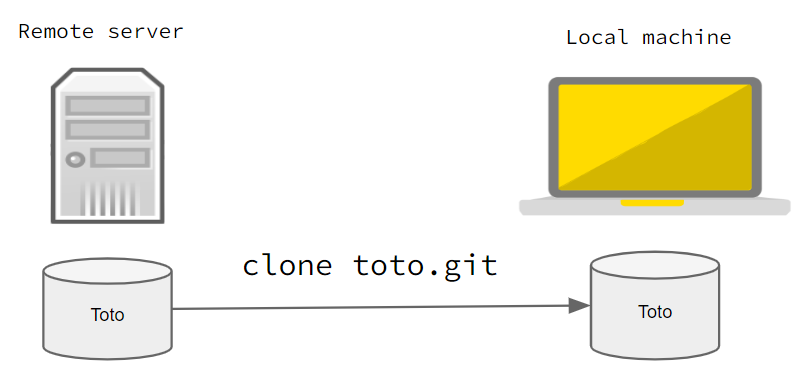
**git config --global pull.rebase false**

## Afficher la configuration

**git config -l**



# Cloner un repo existant

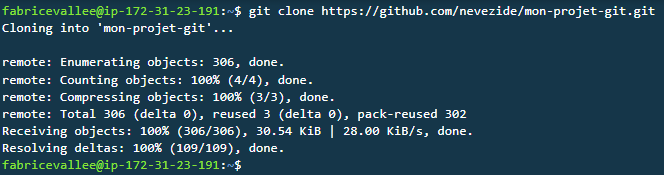


**git clone <nom repo>**

Pour pouvoir travailler sur le code, il faut faire une copie du repo depuis le serveur distant sur la machine du développeur.

Aller dans le dossier «**sites**» : **cd sites**

Saisir la commande : **git clone https://github.com/nevezide/mon-projet-git.git**



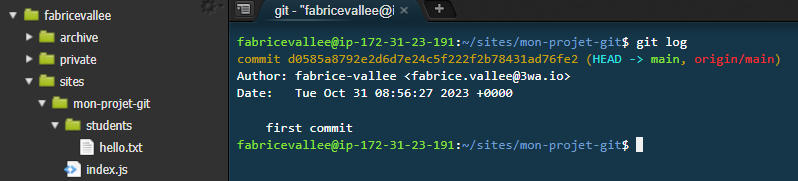
=> Observer la copie « mon-projet-git »

# Voir l’historique des commits

**git log**

Aller dans le dossier du projet : **cd mon-projet-git**

Saisir : **git log**



=> Le premier commit apparaît avec son auteur, la date de création et le message associé

=> HEAD représente le pointeur, l’endroit où se base notre code.

On peut se promener de commit en commit et ainsi voir le code de chacun d’eux, pratique !

# Créer / Modifier / Supprimer un fichier

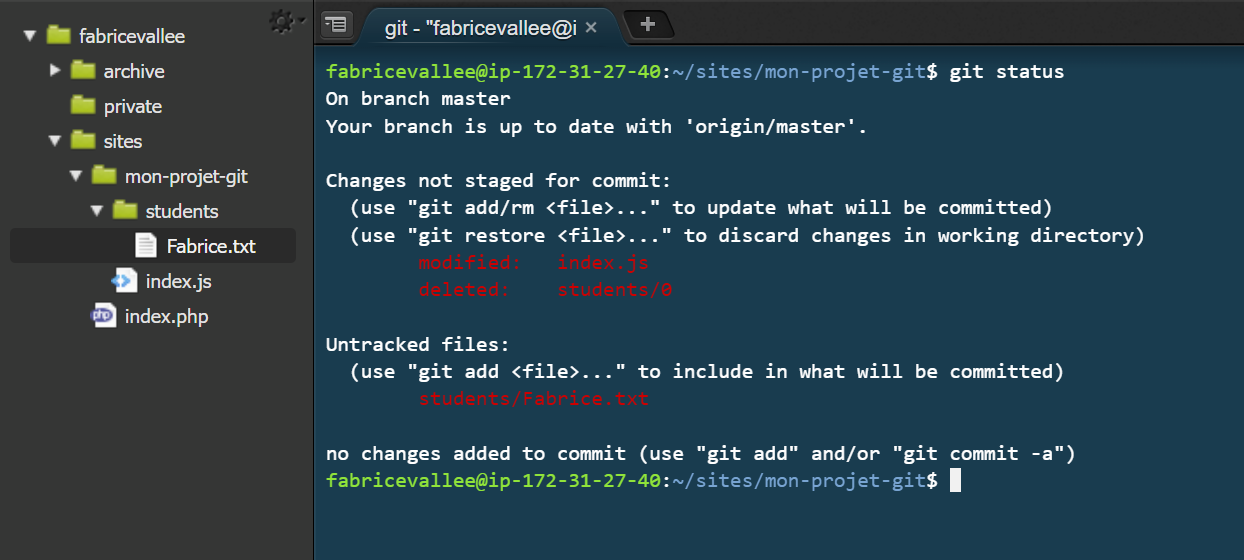
Réaliser les tâches suivantes :

* **Supprimer** le fichier « students/hello.txt »
* **Ajouter** dans « students » un fichier avec leur nom (sans accents) et ce qu’ils veulent dans le contenu.
* **Modifier** le fichier « index.js » en ajoutant le nom de leur fichier dans le tableau « studentsFiles »

# Voir le statut des fichiers

**git status**

Saisir : git status



=> On voit les fichiers modifiés dans « modified »

=> On voit les fichiers supprimés dans « deleted »

=> On voit les nouveaux fichiers dans « Untracked files »

=> On voit « Changes not staged for commit », qu’est ce que cela veut dire ?

On voit ça maintenant !

# Préparer un commit

Un commit est une version de notre code.

Avant de le créer, on va spécifier quel(s) fichier(s) on veut y mettre.

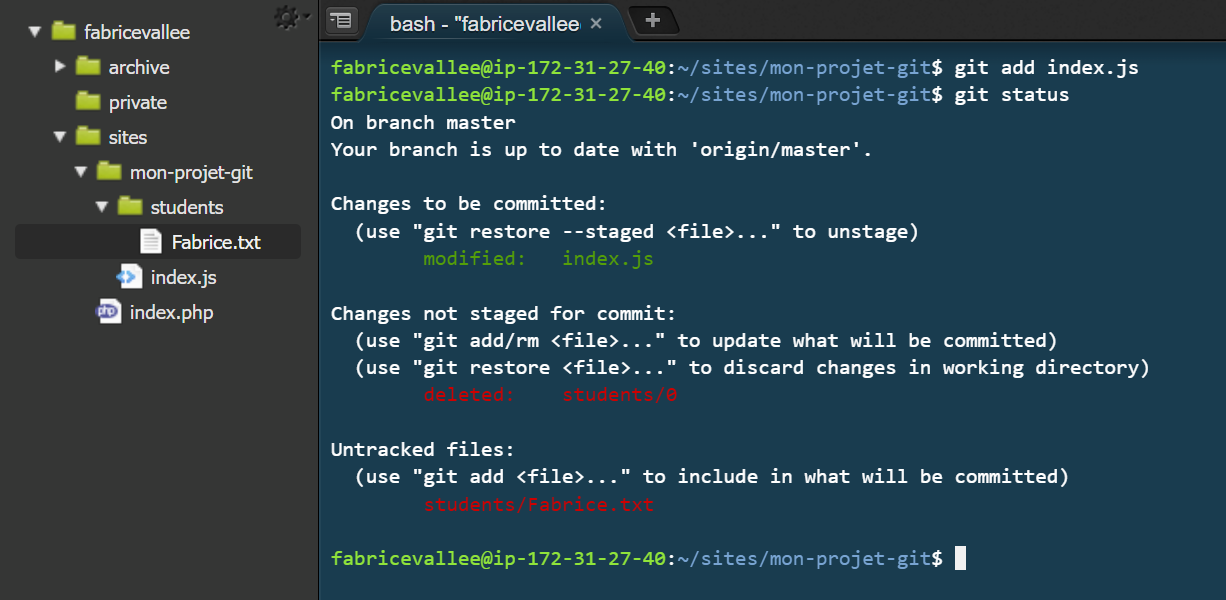
Tout à l’heure, on a vu « Changes not staged for commit » et « Untracked files ».

Ces messages signifient que ces fichiers ne sont pas spécifiés pour être dans le prochain commit.

**git add fichier1 fichier2 ...**

Saisir la commande : **git add index.js**

Puis **git status** pour voir sa prise en compte



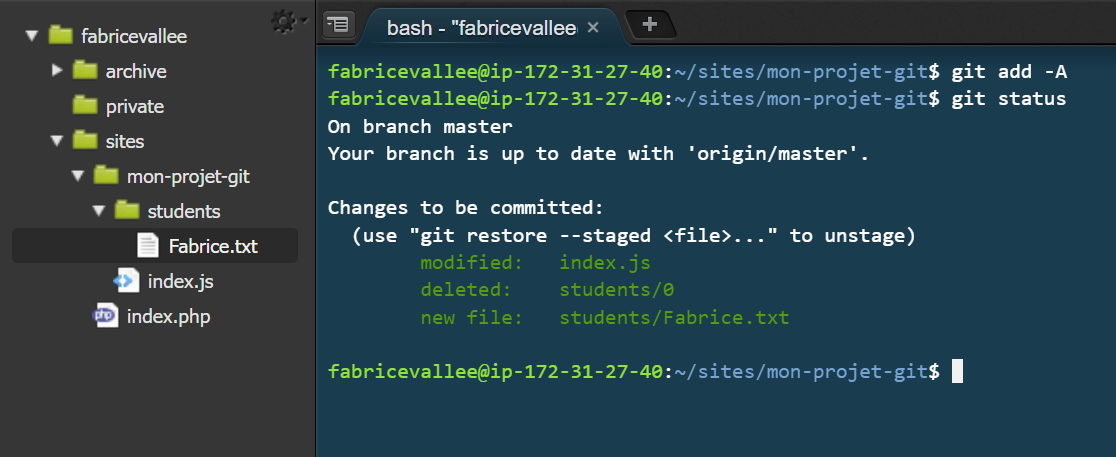
=> On peut voir que **index.js** est dans « Changes to be committed ».

Mettre les fichiers un à un peu être très chronophage, aussi on peut mettre tout un dossier avec **git add <nom dossier>**

Également, on peut mettre tous les fichiers avec **git add -A**

Saisir : **git add -A**

Puis pour vérifier : **git status**



=> On voit que tous les fichiers ont été mis dans le prochain commit.

Il reste plus qu’à créer le commit !

# Créer un commit

**git commit -m"message de commit"**

Un message de commit sert à savoir ce qu’il y a dans le commit à posteriori.

Aussi il est important qu’il soit clair et concis.

Imaginez qu’il soit noyé dans une centaines de commit et comporte un bug, difficile de le trouver si son message n’est pas clair.

C’est pour cela qu’une convention est née : <https://grafikart.fr/tutoriels/nommage-commit-1009>

Le message est formaté comme suit : «   **type ( portée ) : objet**»

Exemple : « feat(Books) : User can rend a book if is not already rent »

**type**

Les plus utilisés sont :

* **fix** : pour un correctif
* **feat** : pour une feature (ajout de fonctionnalité)
* **hotfix** : correctif critique en production
* **docs** : ajout de documentation / commentaires ds code
* **refactor** : refactorisation du code

**portée**

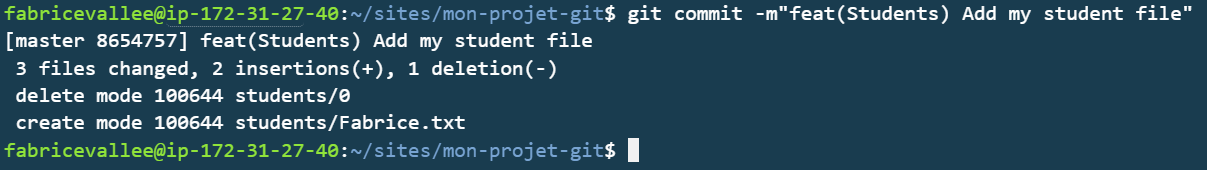
Partie du code impactée par le commit, par exemple :

* Un domaine fonctionnel (ex : Books)
* Un contrôleur, une vue
* Une librairie
* Un module

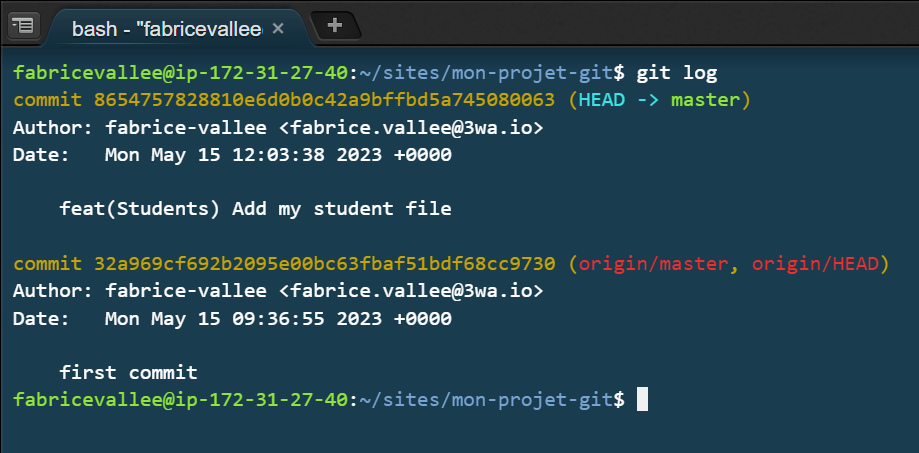
**objet**

La raison du commit, pourquoi je fais ce commit.

Saisir la commande : **git commit -m"feat(Students) Add my student file"**



Puis **git log** pour vérifier que le commit est bien créé



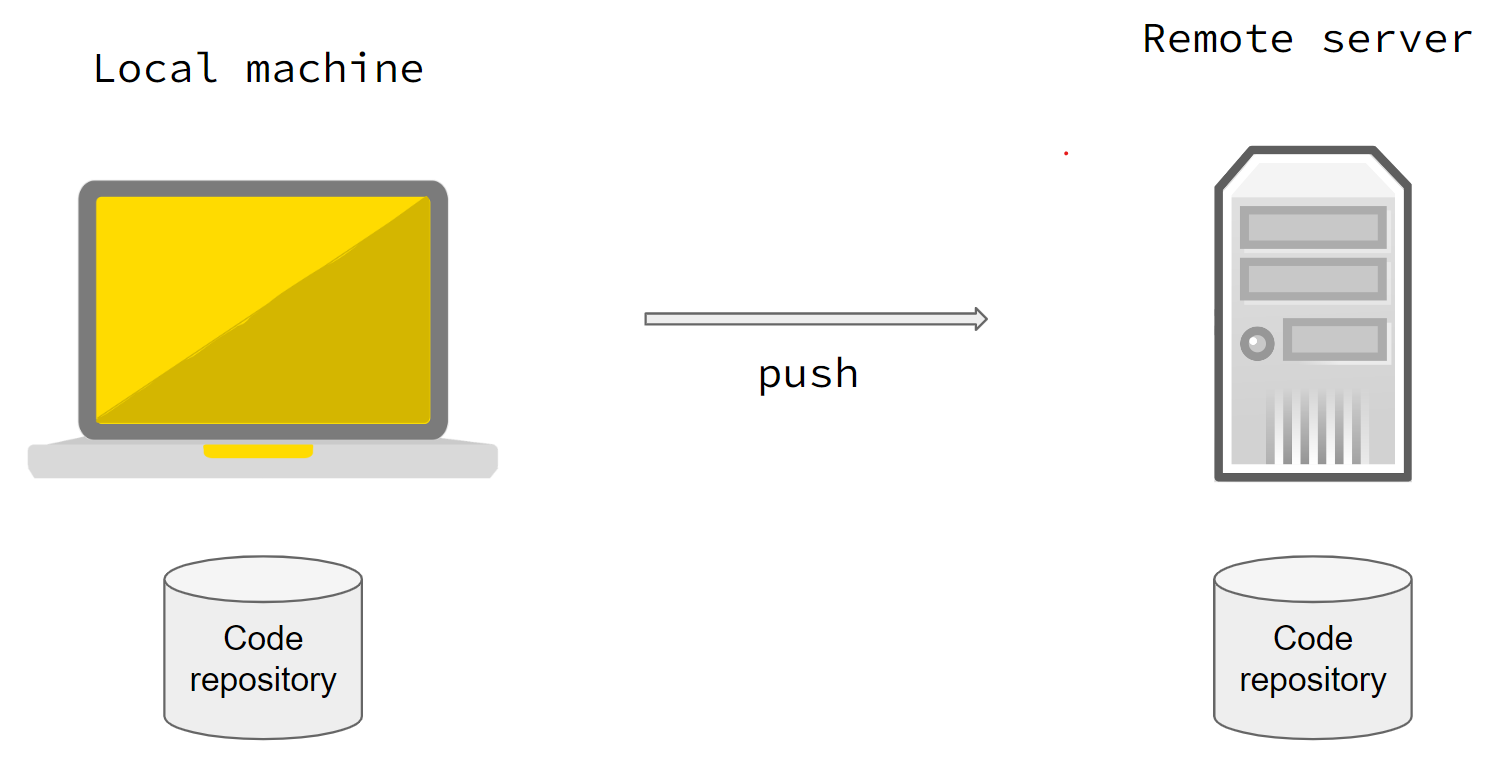
Si on saisi **git status**



=> On voit le message « nothing to commit, working tree clean »

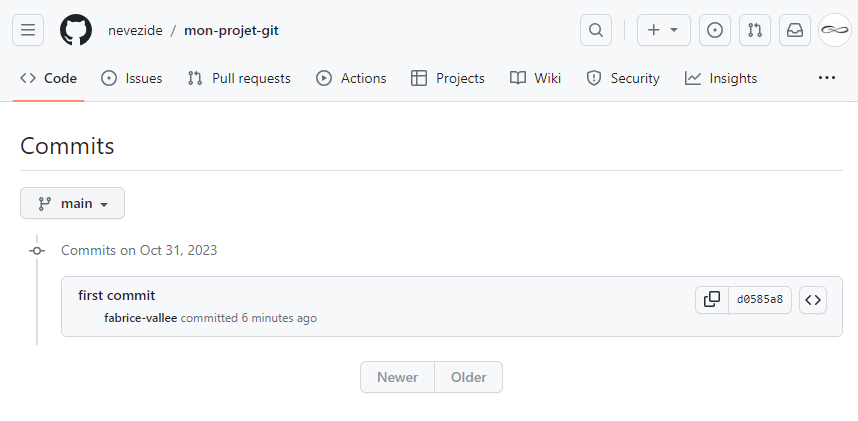
# Envoyer les changements sur le serveur distant

**git push**



Pour le moment le commit qu’on a créé est en local sur ma machine.

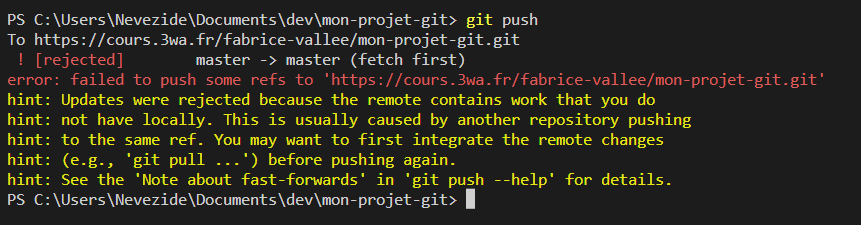
Si on regarde la liste des commit du repo sur le serveur



Pour l’envoyer sur le serveur, saisir : **git push**



A part le premier qui aura push, vous allez voir l’erreur suivante :

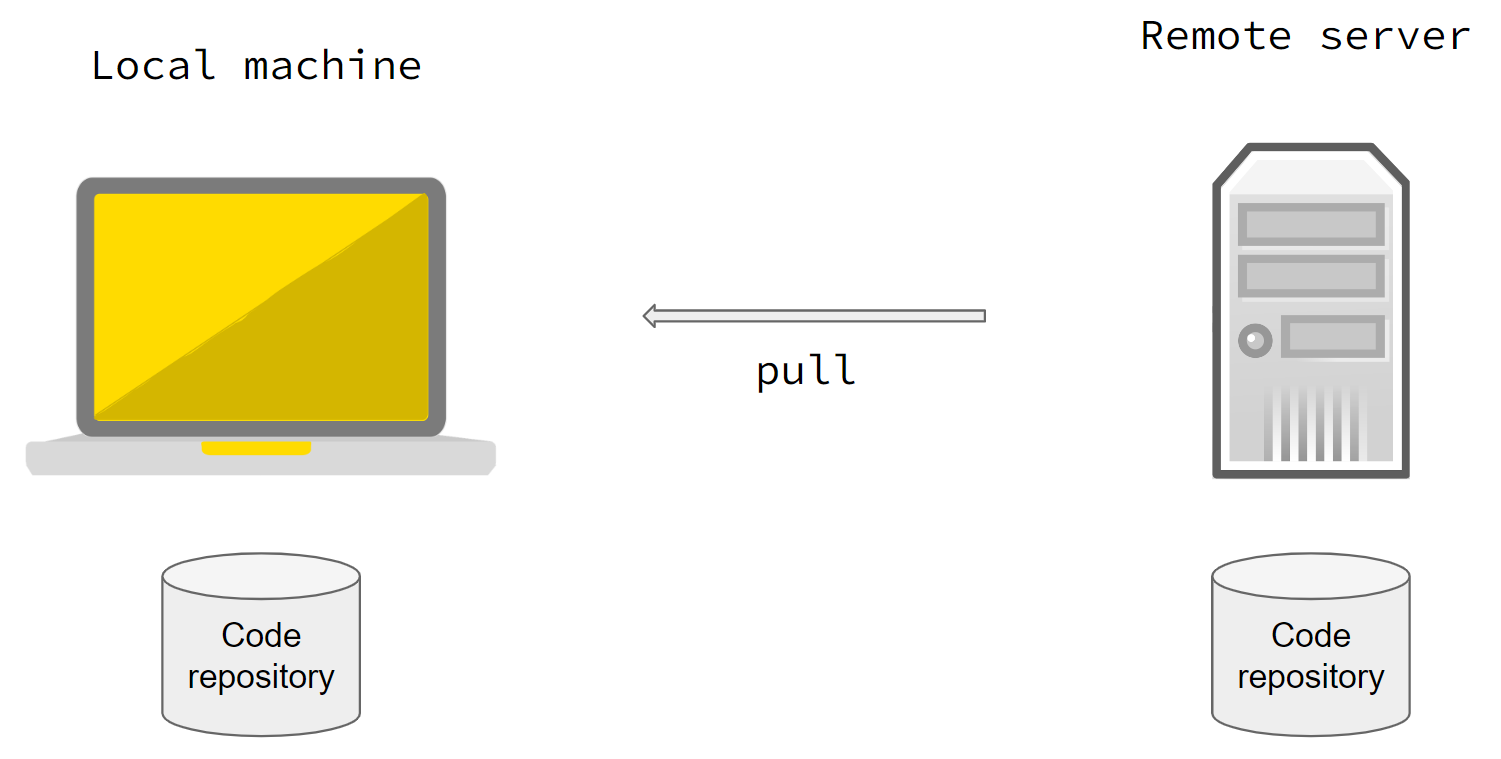


Git nous dit qu’il y a eu de nouveaux changements sur le serveur et nous demande de les récupérer avant de pouvoir envoyer les nôtres.

Nous allons maintenant voir comment faire

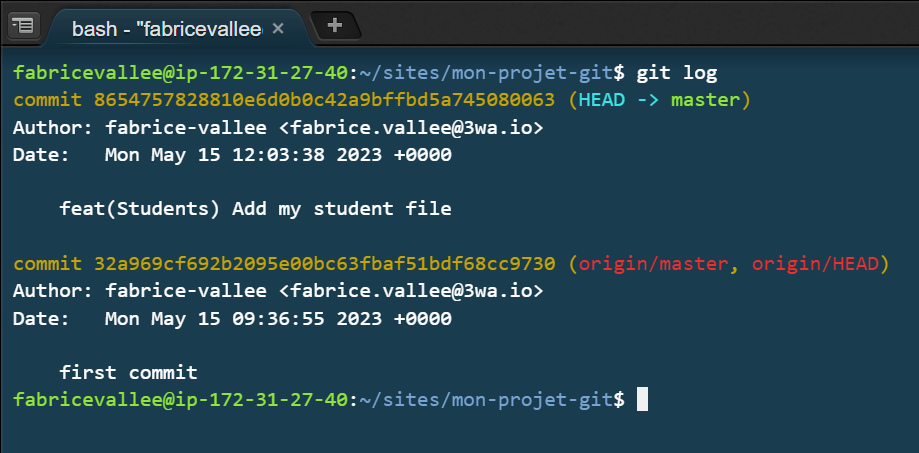
# Récupérer les changements depuis le serveur distant

**git pull**



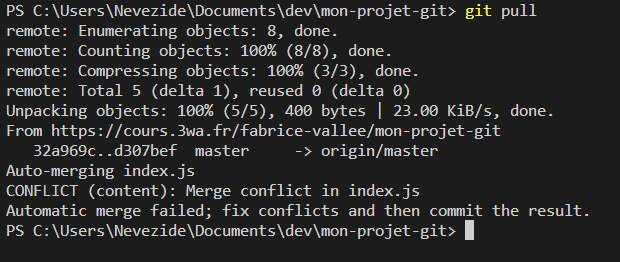
Permet de récupérer les commits envoyés par les autres développeurs.

Vérifions où on en est avec **git log**



Maintenant, récupérons les changements, avec **git pull**

ATTENTION : seul une personne (par exemple le formateur) exécute la commande **git pull** pour le moment.



On peut voir qu’il y a un conflit dans **index.js** qu’il n’a pas pu résoudre automatiquement.

Et oui, étant donné qu’on a tous travaillé dans la même portion de code sur le fichier **index.js**, git ne sait pas s’il doit ajouter toutes nos lignes ou en garder qu’une seule.

NOTE Formateur : l’exercice suivant peut prendre du temps dû aux comportements pas toujours prévisible de git. Faites-le si vous êtes à l’aise avec la gestion des conflits de GIT.

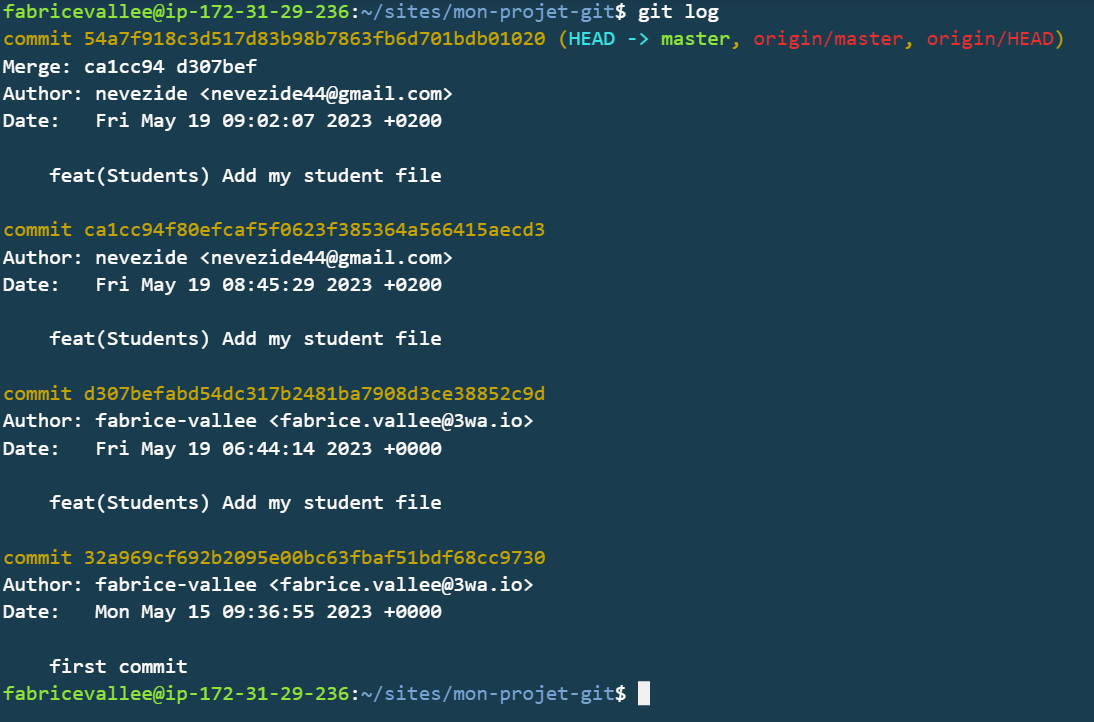
Nous allons devoir nous organiser pour créer tour à tour notre commit.

Pour chacun d’entre vous, à tour de rôle,

* Saisir dans le terminal **git pull**
* Garder toutes les lignes de index.js séparées par des virgules
* Restage le code (avec **git add -A**)
* Si git le demande, refaire un **git commit -m"resolve conflicts"**
* Saisir **git push**

Une fois qu’on a tous push notre commit, on fait tous ensemble un **git pull** pour récupérer les commits de tout le monde

On recheck avec **git log**

****

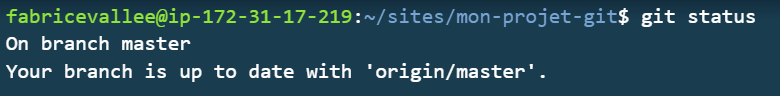
=> On voit les commit des autres étudiants

Si maintenant on regarde la liste des commits sur le repo du serveur

=> On voit que les commits de tout le monde ont bien été envoyés

# Les branches

Si on fait un **git status**, on remarque la ligne « On branch master »



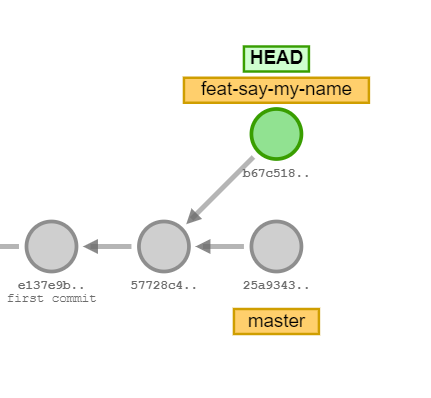
«**master**» est le nom de la branche par défaut *(parfois elle s’appelle aussi «****main****»)*

Dans un repo partagé, cette branche **est réservée pour mettre en production.**

Tout code sur cette branche est fonctionnel et prêt à partir en production.

**⚠️ ⚠️ ⚠️ ON NE COMMIT JAMAIS son code SUR MASTER / MAIN ⚠️ ⚠️ ⚠️**

Pour travailler, on créé une branche et une fois que le code de cette branche est testé et validé, on la fusionne sur master.



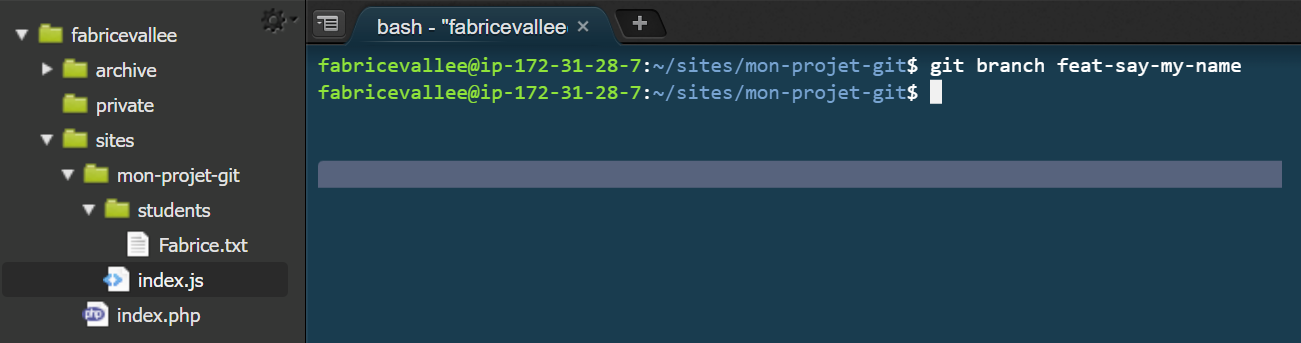
Une branche se réfère toujours à un commit (présent dans une autre branche, à défaut master/main).

## Créer une branche

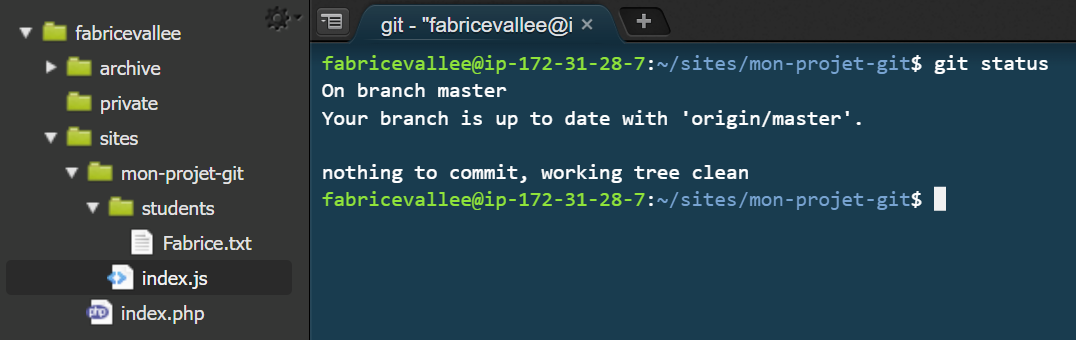
**git branch <nom branche>**

Saisir : **git branch feat-say-<my-name>**  avec ton nom à la place de <my-name>

NOTE formateur : Chaque étudiant créé sa propre branche avec son nom



Si on refait **git status**



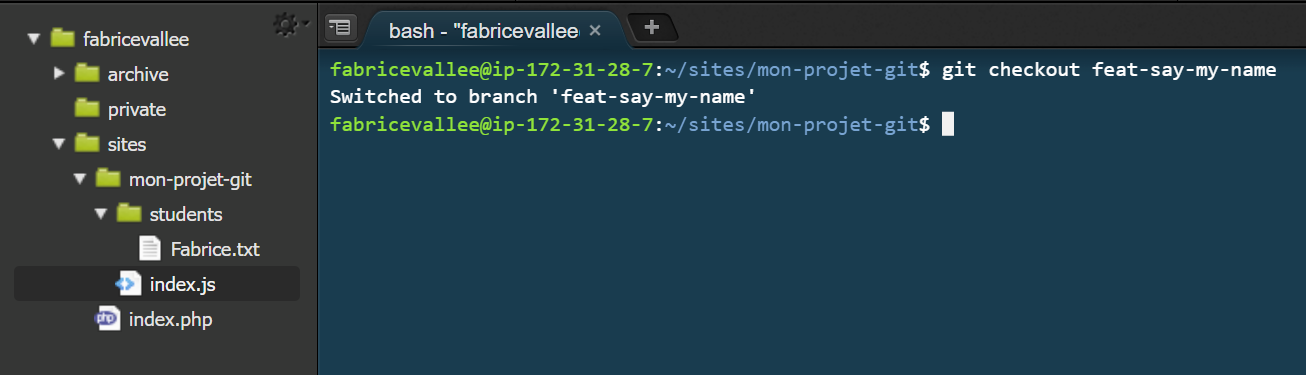
=> On remarque qu’on est toujours sur la branche master/main

## Aller sur une branche existante

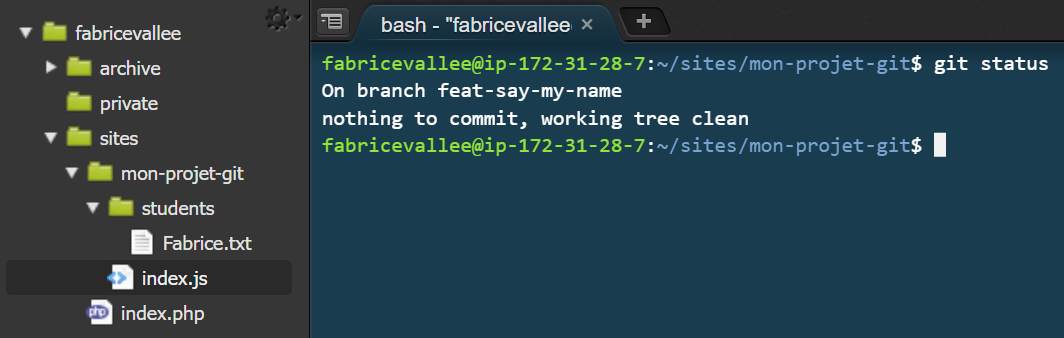
***ou nouvellement créée***

**git checkout <nom branche>**

Saisir : **git checkout feat-say-<my-name>**



Vérifions avec **git status**



=> Cette fois-ci on est bien sur la branche **feat-say-<my-name>**

## Codons un peu

Nous allons créer un commit sur cette branche.

Modifier le fichier que vous avez créé auparavant, puis créer le commit (avec votre nom) et le pousser sur le serveur.

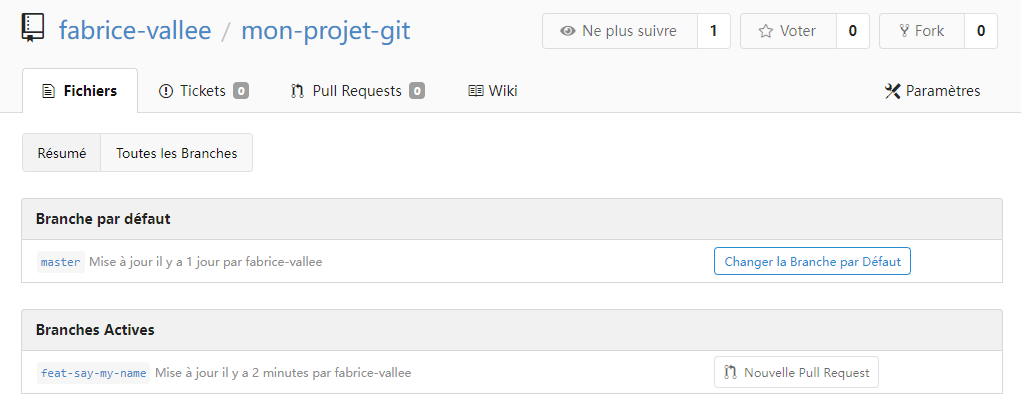
**RAPPEL des commandes :**

**git add -A**

**git commit -m"feat(Students) Update <nom>"**

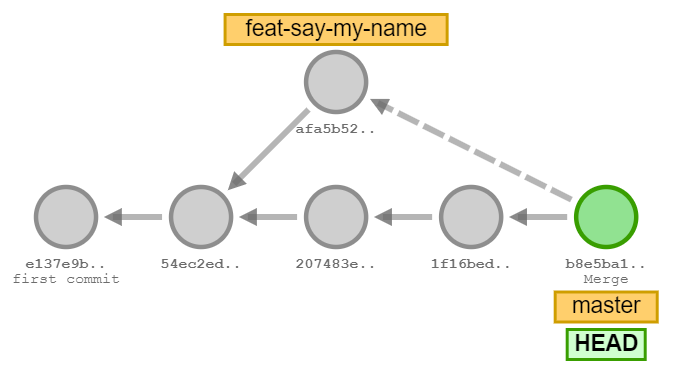
**git push**

Regarder dans le repo du serveur les différentes branches ainsi créées.



Maintenant nous allons fusionner tout ça dans master !

## Fusionner une branche dans une autre



**git merge <nom branche>**

La branche va être fusionnée dans la branche courante.

Du coup, il faut au préalable aller dans la branche de destination.

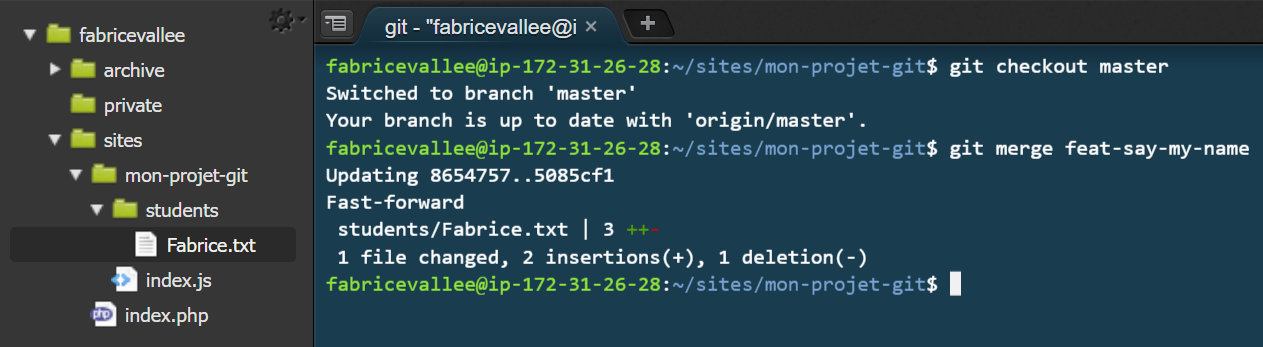
Saisir les commandes suivantes, chacun son tour (pour éviter les conflits) :

**git checkout master**

**git pull**

**git merge feat-say-<my-name>** (avec le nom de la branche de chacun)

**git push**



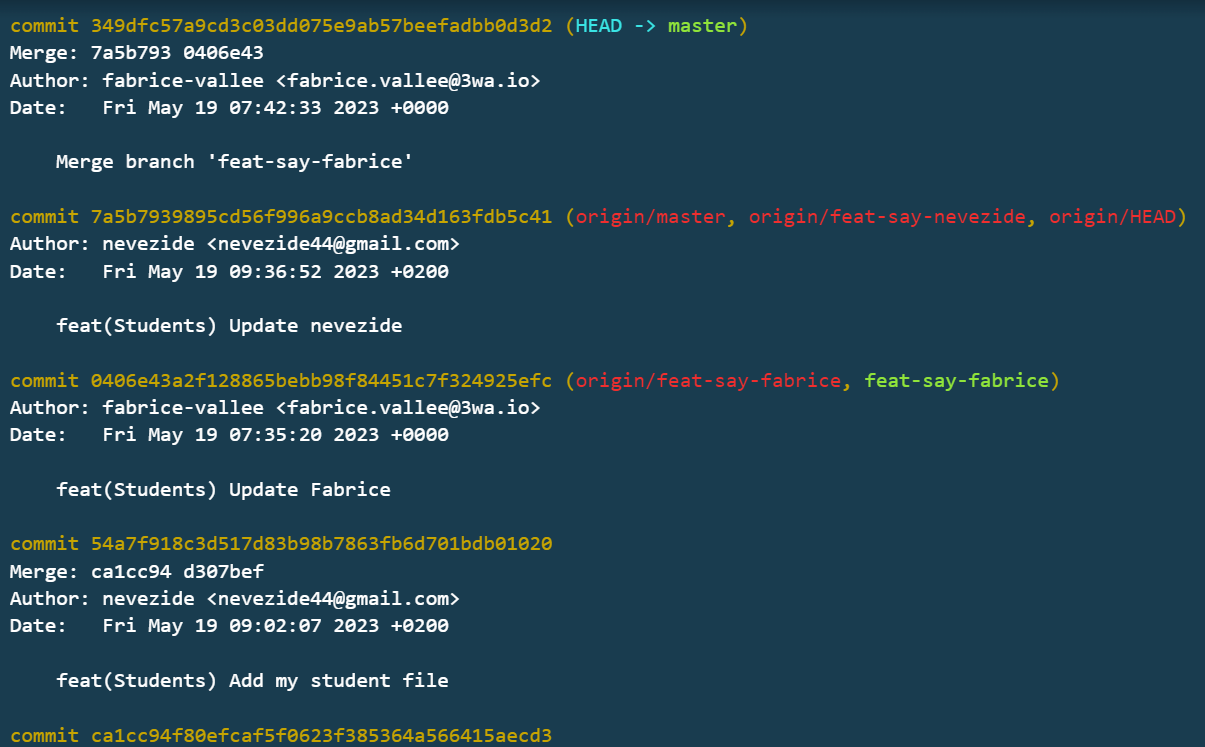
=> On peut voir qu’il fait un fast-forward et non un merge la première fois.

Un fast-forward va seulement déplacer le pointeur (HEAD) de la branche master sur le dernier commit de la branche fusionnée.

=> Pour les fois suivantes, git va créer un nouveau commit qui va contenir l’historique de la branche (cad l’ordre dans lequel les commits ont été créés). Cela permet de revenir en arrière facilement s’il y a un problème.

Un fichier invitant à saisir le message du commit de merge s’ouvre. Enregistrer sans modifier.

Maintenant regardons **git log**



=> On voit les commits issus de la branche mergée.

A côté des commits, on voit quelle branche le référence

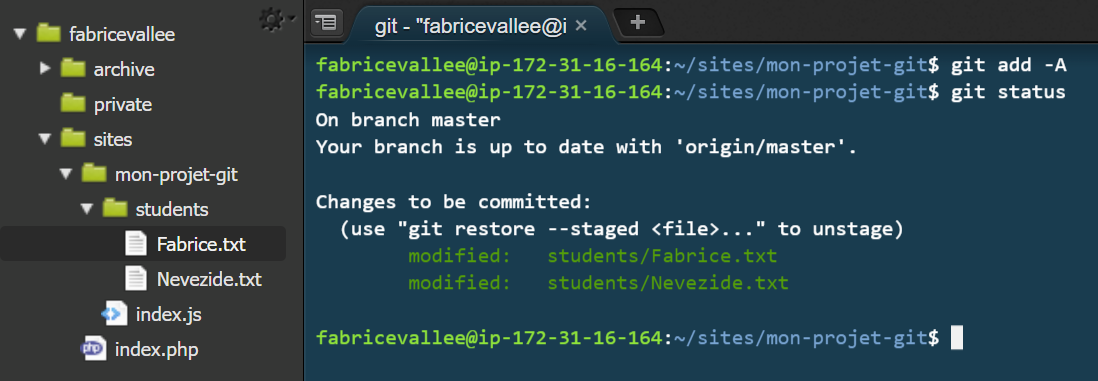
Regardons l’historique des commits sur le serveur

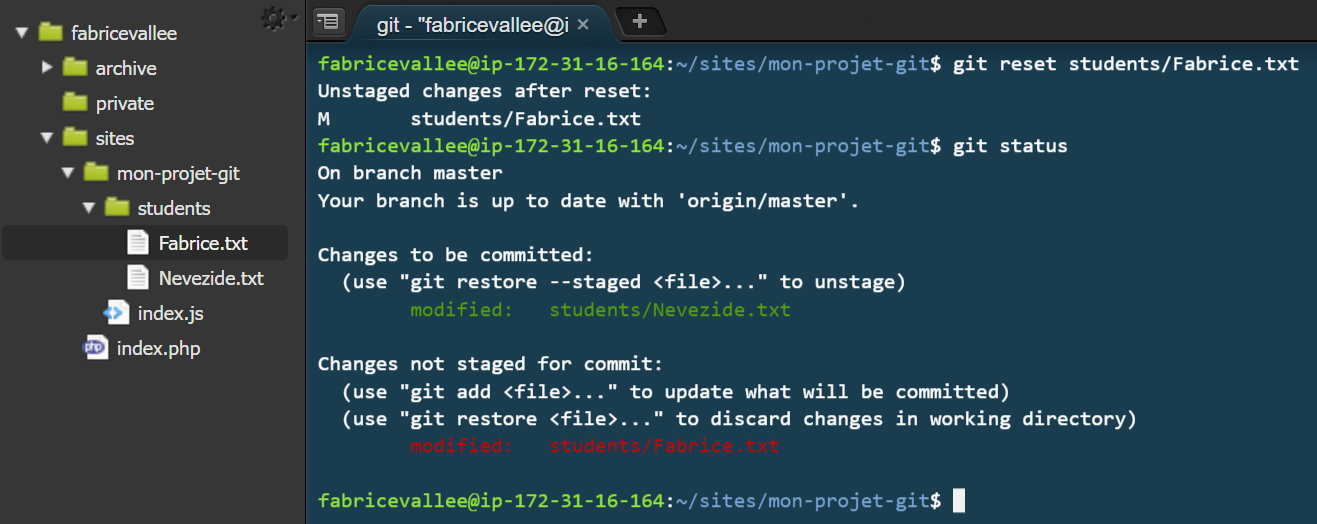
=> On voit les commits sur la branche master du serveur

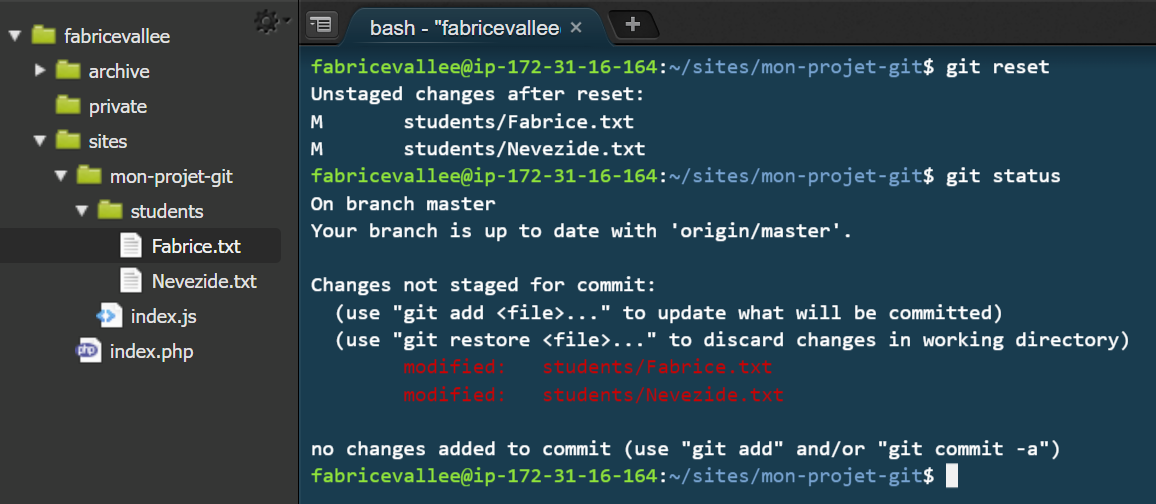
# Comment revenir en arrière avec git

## Enlever les fichiers ajoutés au prochain commit

* 1. **git reset**
  2. Enlève un, plusieurs ou tous les fichiers du prochain commit (unstage)
  3. Pour illustrer cela, modifions plusieurs fichiers.
  4. Les ajouter pour le prochain commit : **git add -A**

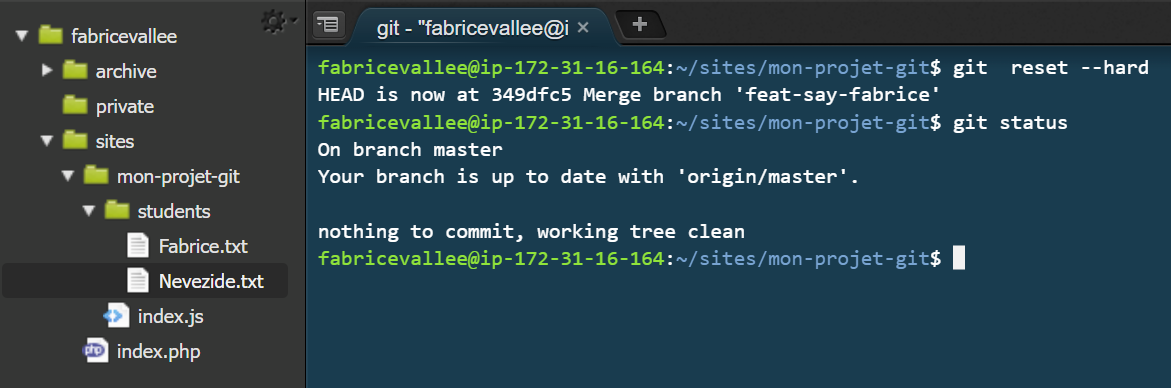
****

* 1. Oups, je veux revenir en arrière : c’est à dire enlever les fichiers que j’ai ajoutés.
  2. Saisir **git reset students/Fabrice.txt**
  3. ****
  4. On peut aussi enlever tout les fichiers avec **git reset** (sans spécifier de fichiers)

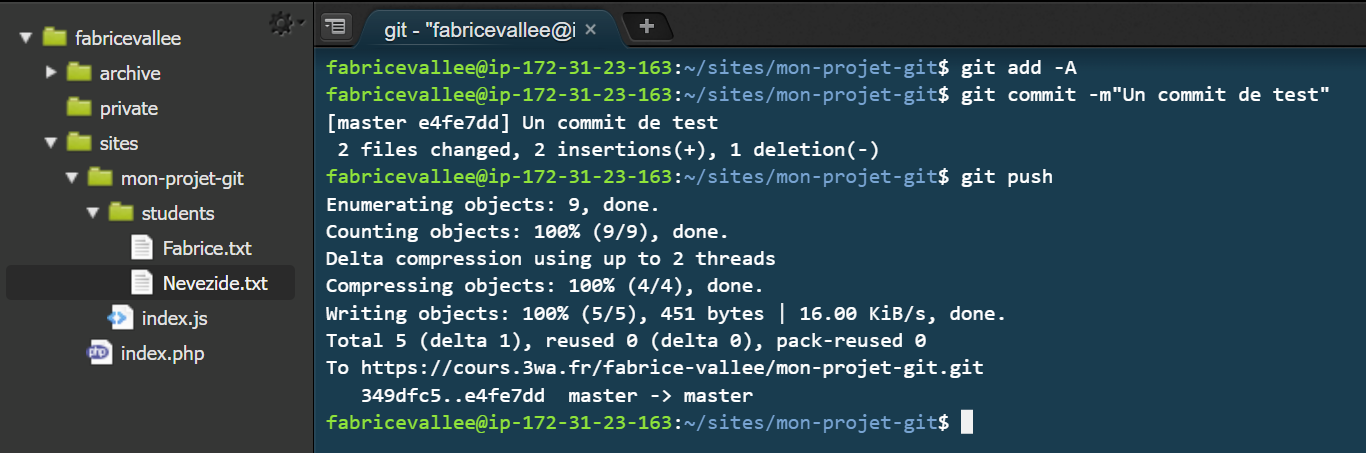


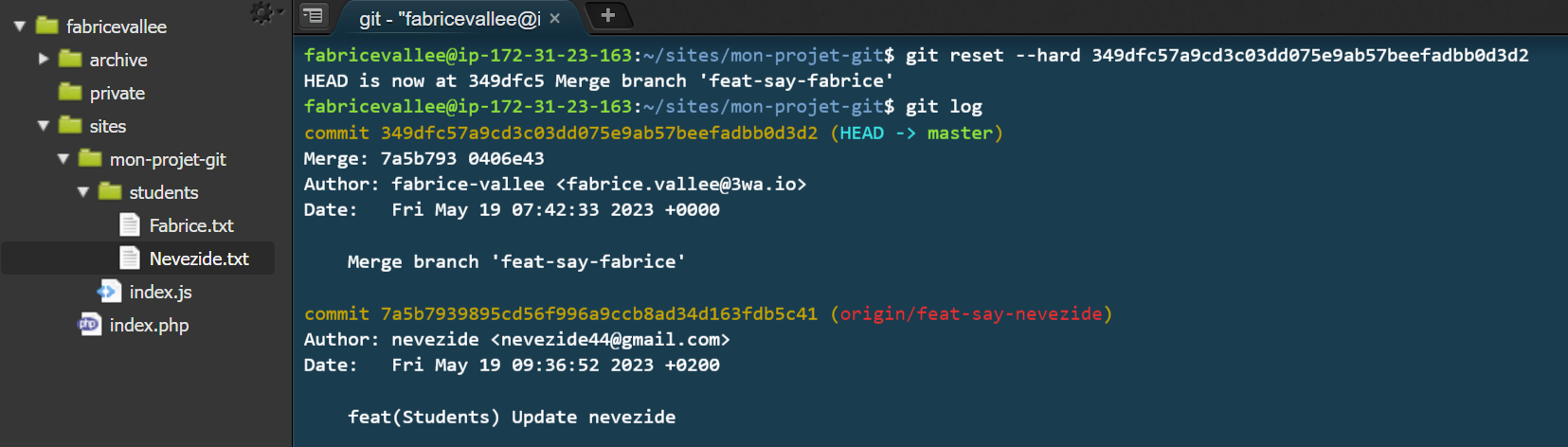
## Supprimer toutes les modifications de code

* 1. Saisir **git reset --hard**

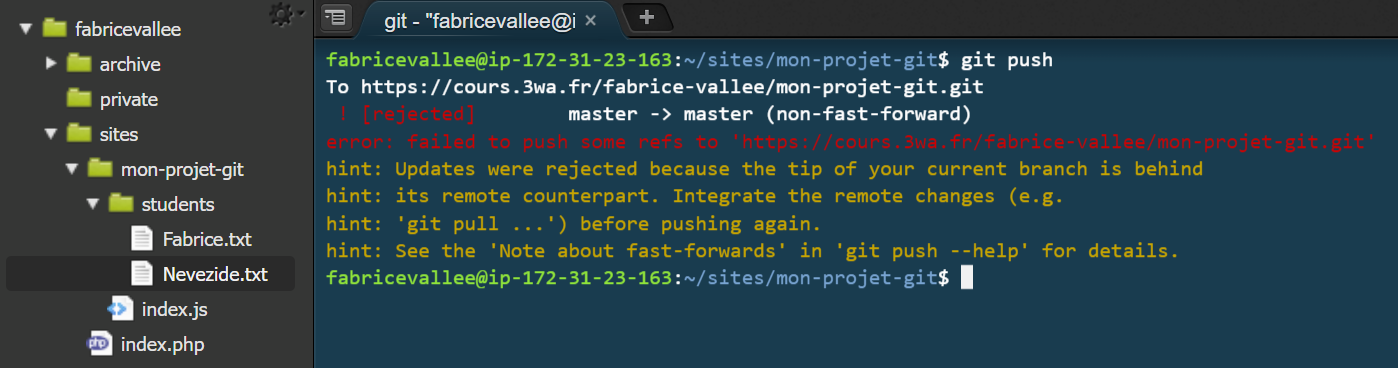


## Revenir définitivement à un commit

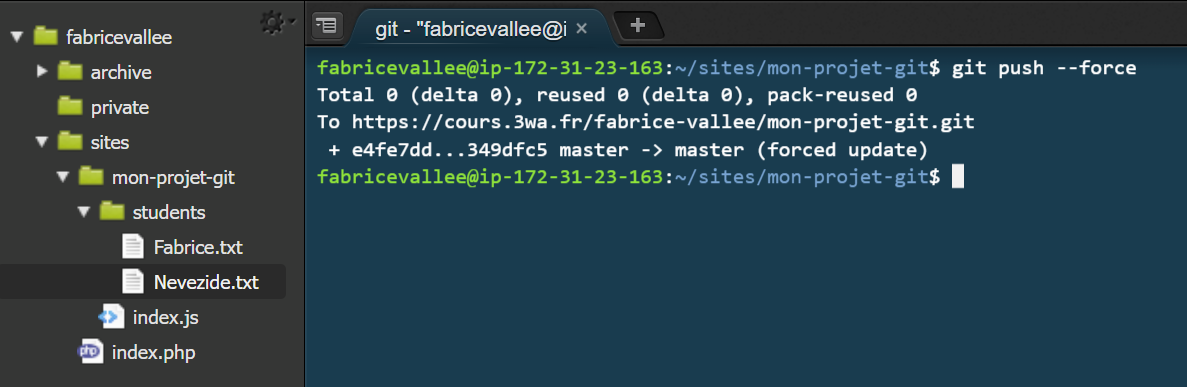
* 1. Créons un nouveau commit.
  2. Puis, on l’envoie sur le serveur avec **git push**
  3. Maintenant, affichons l’historique des commits : **git log**
  4. ****
  5. Maintenant, je souhaite revenir au commit précédent « Merge branch ... »
  6. Saisir : **git reset --hard <hash du commit>**



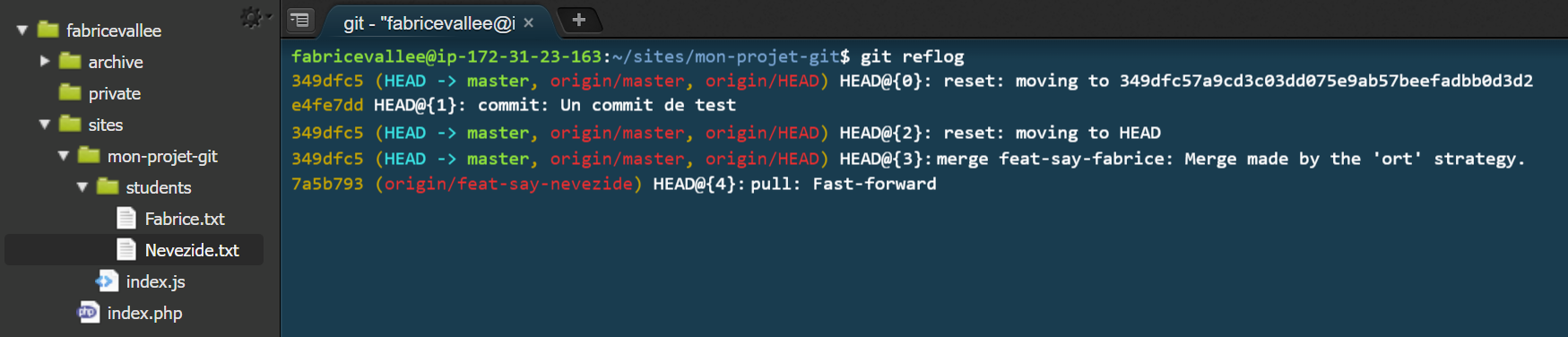
* 1. On peut voir que le commit « Un commit de test » n’est plus sur la branche.
  2. Maintenant, faisons un **git push**

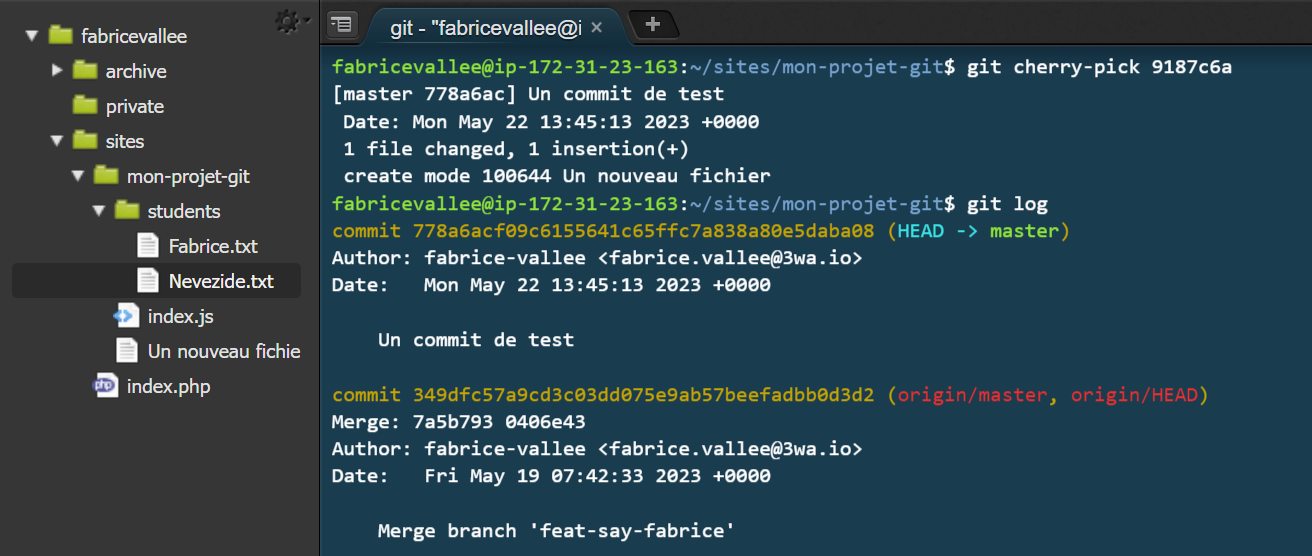
****

* 1. => Le push fail car l’historique des commits est différent entre la version locale et celle du serveur.
  2. Pour contourner le check, on peut forcer, c’est à dire dire écraser la version de l’historique des commits du serveur avec la mienne.
  3. ATTENTION : bien s’assurer qu’il n’y a pas de nouveaux commits sur le serveur entre temps, car ils disparaîtront.
  4. Saisir : **git push --force**



# Avec git, rien n’est perdu

1. Avec git, rien n’est supprimé, on peut toujours retrouver ce qu’on a perdu.
2. Saisir **git reflog**
3. ****
4. => Ce log affiche l’historique de manipulation des commits, branches, …
5. On voit le moment où on est revenu en arrière et juste avant, nous avons le moment où on a créé le commit.
6. Pour le récupérer il suffit de saisir : **git cherry-pick <hash du commit>**

****

1. => Git va l’ajouter à la suite des commits de la branche courante.
2. Le tour est joué, on a récupéré notre commit !

# Gitignore

Dans nos projets, certaines choses ne doivent aucunement être commités

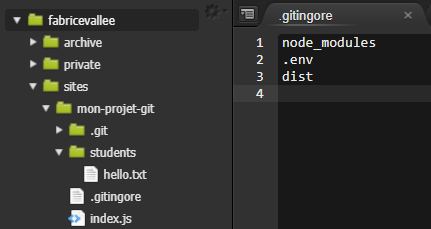
- Les mots de passe (.env)

- Les dépendances (node\_modules, vendor)

- Les builds (dist)

Pour dire à git d’ignorer ces fichiers / dossiers, on met à la racine du projet un fichier appelé « .gitignore ». Sans oublier le point devant.

A l’intérieur, on y met les fichiers / dossiers concernés



# Gérer les versions

Une version est un livrable, une sortie officielle diffusée aux clients.

Elle est matérialisée par deux choses : le code source et le numéro de version (souvent précédé de la lettre v).

Le numéro de version est codé comme suit :

Version majeure . Version mineure . Numéro de patch

**La version majeure** est une version dans laquelle du code a été modifié / supprimé. Cela indique que la mise à jour va introduire des bugs.

Une adaptation sera nécessaire pour corriger ce qui utilise le code du livrable.

Elle contient des nouvelles fonctionnalités majeures, nécessitant une refonte totale ou partielle de code.

**La version mineure** est une version dans laquelle du code a été ajouté uniquement. Cela indique qu’il n’y a aucun risque de bugs. Elle contient des nouvelles fonctionnalités et des correctifs.

**Le numéro de patch** indique une version dans laquelle uniquement des correctifs ont été apportés (corrections de bugs, correction de failles de sécurité). Il n’y a aucune nouveauté fonctionnelle contrairement aux changements de version.

Prenons l’exemple v2.5.12

Le **2** correspond à la 2ème version majeure (il y a eu 1 refonte de code)

Le **5** à la 5ème version mineure (il y a eu 5 versions d’ajouts de fonctionnalités)

Le **12** au 12ème patch (il y a eu 12 correctifs apportés)

## Gérer les versions avec git

Pour se faire, on utilise les tags.

Un tag est une étiquette qui est mise sur un commit.

Pour voir les tags existants, saisir **git tag**



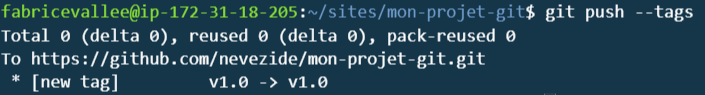
Pour créer un tag de version , saisir par exemple **git tag v1.0**



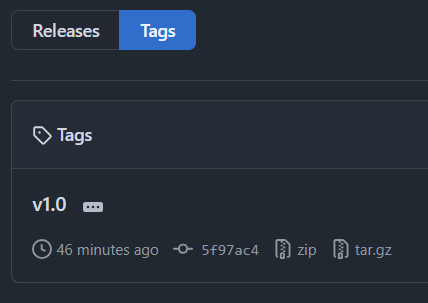
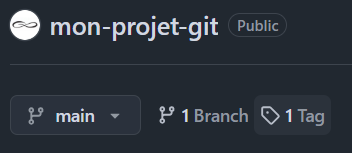
Les tags sont créés sur la machine locale

Pour les envoyer sur le serveur, il faut ajouter quelque chose lors du push

Git push --tags



Maintenant, on peut voir le tag sur github



On voit ici le commit associé au tag.

# Pour aller plus loin

1. Documentation officielle

[**https://git-scm.com/**](https://git-scm.com/)

Sandbox git

[**https://git-school.github.io/visualizing-git/**](https://git-school.github.io/visualizing-git/)