Module git

Utiliser un gestionnaire de versions

Table des matières

A. Qu’est-ce que git ?

Git est un logiciel créé en 2005 par Linus Thorvalds (l’homme à l’origine du noyau de Linux).

Git est un **gestionnaire de versions**. Autrement dit, il gère des versions de notre code. Cela signifie qu’il ne gère pas le code, mais simplement les différences qu’on y apporte. Sa principale utilité, c’est donc de garder un **historique** de l’avancée de notre code.

Ceci permet donc d’avoir un œil sur tout ce qui s’est passé pendant le développement, de pouvoir revenir dessus, de gérer des historiques alternatifs également.

Mais il est également très utilisé pour **mutualiser** le code entre plusieurs développeurs, et pour permettre à ceux-ci de **collaborer** sur un même projet.

Git est un service dit **décentralisé** : il permet de gérer un dépôt disponible à plusieurs endroits. Personne n’est réellement propriétaire du service.

B. Git vs GitHub

Il faut distinguer git, un logiciel de gestion de versions, de GitHub, un hébergeur spécialisé dans l'hébergement de dépôts git.

Ainsi, git est un **logiciel**, que l'on utilisera donc sur **notre** machine. Puis, lorsqu'on souhaitera sauvegarder ou mutualiser son code, on l'hébergera sur GitHub.

Il est également à noter que GitHub n'est pas le seul hébergeur spécialisé dans les dépôts git. Il existe aussi GitLab, BitBucket, et tout un tas d'autres sites.

On peut tout à fait, aussi, héberger sur son propre serveur son dépôt git. Il s'agira d'installer la version serveur du logiciel git, et d'utiliser son serveur comme hébergeur. (Mais nous ne le verrons pas dans ce cours)

C. Installation

Git étant un logiciel, il nous faut donc l'installer sur nos machines.

1. Windows

Sous Windows, la solution la plus couramment utilisée est Git for Windows. Si vous cliquez sur le bouton "Download", cela vous permettra d'installer git, mais également un terminal qui pourra vous être utile par la suite pour exécuter diverses commandes liées à git.

Pour l'installation de base, acceptez les termes d'utilisation et faites "suivant" sans changer la configuration de base.

2. MacOS

Sous MacOS, l'installation peut être faite de plusieurs manières. Toutes sont listées sur cette page de la documentation git.

3. Linux

Sous Linux, l'installation se fait via la commande `sudo apt install git-all`, ou `sudo dnf install git-all` (selon votre version de Linux).

4. Tester son installation

Pour tester votre installation, ouvrez un nouveau terminal et tapez la commande `git version`. Si vous avez une version de git qui s'affiche, votre installation est correctement faite. Sinon, le problème le plus courant est que git n'est pas dans votre variable d'environnement PATH (cherchez sur Google comment y remédier).

D. Les concepts de base

I. Initialisation

Avant d'utiliser git pour un projet, en local, il faut lui indiquer qu'il doit s'en occuper. Cela se fait grâce à la commande `git init`, qui initie un projet git dans **le répertoire courant**.

II. Paramétrage de git

Git a besoin de connaître 2 choses pour pouvoir **signer** notre travail : le nom d'utilisateur et l'email. Il est important de les renseigner, et on peut le faire d'une manière globale sur notre machine grâce aux commandes :

```
git config --global user.name "<nom d'utilisateur>"
git config --global user.email "<email>"
```

Ainsi, les différents travaux réalisés apparaîtront sous l'identité renseignée.

III. Les remotes

Les **remotes** désignent les hôtes distants qui hébergent notre dépôt git ("remote" = "distant" en anglais). Il va, par exemple, s'agir d'un dépôt GitHub. On peut gérer les remotes grâce à la commande `git remote`, et notamment :

- Ajouter un remote : `git remote add <nom du remote> <url du remote>`
- Supprimer un remote : `git remote rm <nom du remote>`
- Lister les remotes : `git remote -v`

Il est important de se rendre compte que git va pouvoir faire la jonction entre notre travail **local** et l'hébergeur **distant**. Souvent, le dépôt distant s'appelle *origin*. Cela signifie qu'il sert souvent de **référence**, puisqu'il est partagé entre tous les membres de l'équipe.

IV. Le clonage de dépôt

Lorsqu'on dispose déjà d'un dépôt distant, et qu'on souhaite le récupérer en local, on va pouvoir le cloner. Cela se fait via la commande `git clone <url du dépôt> <nom du dossier>`. Cela effectue plusieurs opérations :

- Créer un dossier (`<nom du dossier>`)
- Télécharge dans le dossier créé le contenu du dépôt distant
- Initialiser un dépôt git local
- Configurer le remote pour pointer vers `<url du dépôt>`

V. Lister les modification

Une version, pour git, correspond à un ensemble de **modifications** apportées à l'intérieur du dossier géré par git. Une modification peut consister en 3 choses : une création, une

suppression, ou bien une modification, que ce soit d'un fichier ou bien du contenu d'un fichier.

Pour créer une nouvelle version, il faut indiquer à git quelles modifications sont concernées par la version. On peut voir l'ensemble des modifications qui ont été effectuées **depuis la dernière version** avec la commande `git status`.

VI. Créer des commits

Dès lors que l'on connaît l'ensemble des modifications effectuées, on peut indiquer à git qu'on souhaite les inclure dans notre prochaine version. Cela se fait via la commande `git add <quelque chose>`. Quelque chose peut être le nom d'un fichier, ou bien d'un dossier, ou même une *wildcard* (une chaîne de caractères qui désigne plusieurs fichiers d'un coup, par exemple *.html pour dire "tous les fichiers qui finissent par .html").

Une version de code, pour git, s'appelle un **commit**. Dès que l'on a indiqué les modifications à prendre en compte pour la prochaine version (pour le prochain commit), on peut créer le commit avec `git commit`. Cela va avoir pour effet de sauvegarder nos modifications.

En règle générale, on aime bien mettre un petit message pour indiquer l'utilité des modifications qui ont été effectuées. On peut pour cela utiliser une *option* avec la commande: `git commit -m "<message>"`.

Un commit est lié à une chaîne de caractères, qu'on appelle **somme de contrôle**, et qui correspond au chiffrement par la méthode **SHA-1** des fichiers qui composent le commit. Ce qui est pratique, avec cette somme, c'est qu'elle permet de vérifier l'intégrité du commit : est-ce que quelque chose a été modifié ou non.

Cette somme de contrôle sert également d'**identifiant** au commit. C'est son nom, en quelque sorte.

VII. Push, pull, fetch

Tous les commits que nous créons sont créés en **local**. Tant que nous ne lui indiquons pas de le faire, git ne synchronise par le remote avec notre local.

Il va y avoir 3 commandes pour gérer la synchronisation avec le remote.

a. Push

La commande `git push` **pousse** (envoie) les données locales vers le remote. On peut explicitement lui dire où envoyer les données avec la version plus complète de la commande `git push <remote> <branche>` (on voit plus loin ce qu'est une branche). Par défaut, `git push` envoie les données vers *origin*.

Il peut arriver, la première fois que l'on fait un push, que celui-ci ne fonctionne pas car "il n'existe pas d'équivalent du travail local sur le serveur distant". Git va alors nous donner une commande à exécuter pour créer cette équivalence, qui ressemblera à :

```
git push --set-upstream origin master
```

b. Pull

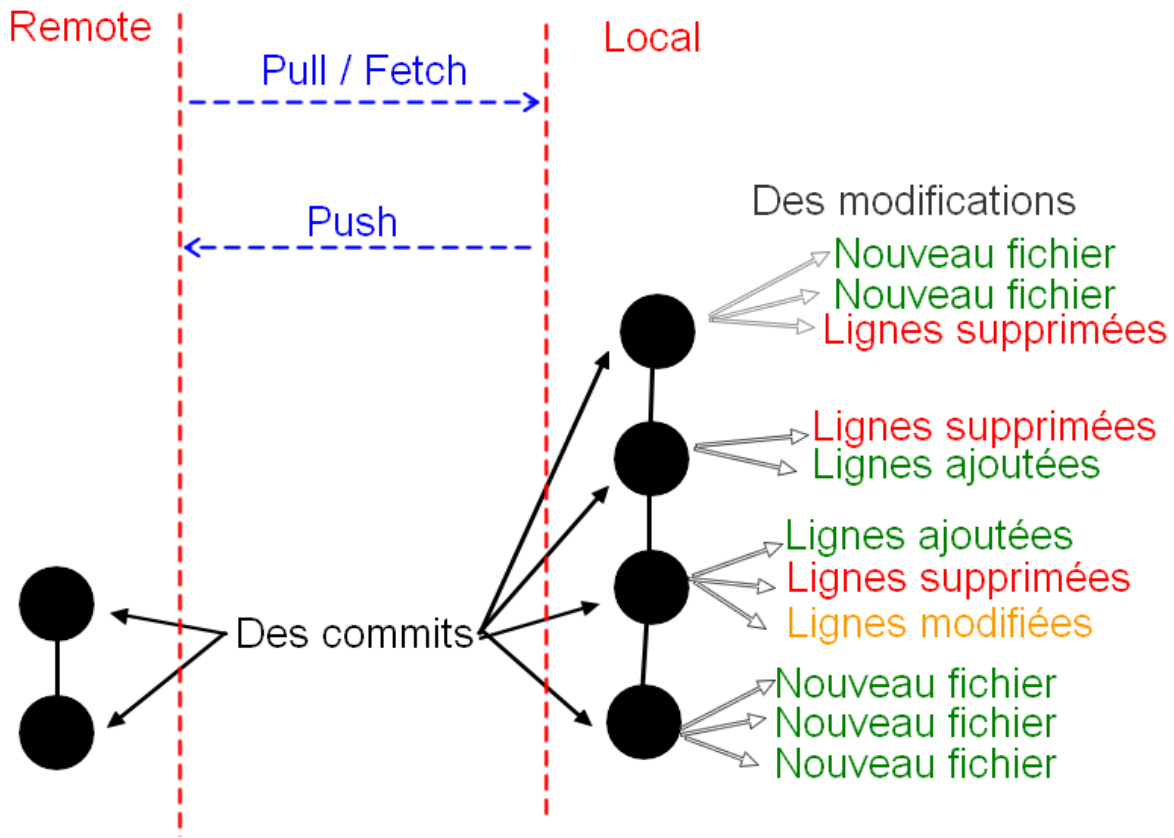
La commande git pull **tire** (récupère) les données du remote vers le dépôt local. C'est en quelque sorte la commande inverse de `git pull`. Tout comme `git push`, on peut explicitement lui dire où envoyer les données avec la version plus complète de la commande `git pull <remote> <branche>` (on voit plus loin ce qu'est une branche). Par défaut, `git pull` récupère les données depuis *origin*.

Lorsqu'on fait un `git pull`, les versions du serveur distant et celles qui sont locales sont *fusionnées*, autrement dit tous les changements s'ajoutent. (Cela peut entraîner des **conflits** si deux versions d'un même fichier existent.)

c. Fetch

La commande `git fetch` est équivalente à la commande `git pull`, à la différence près qu'elle ne fusionne pas les versions. Aussi, il faudra effectuer la fusion *à la main*, à l'aide de la commande `git merge` (voir plus loin).

VIII. Résumé des notions vues



E. Utilisation avancée

On a vu l'utilisation de base, qui se focalise principalement sur l'ajout de version **linéaire** : on a construit un "fil de versions" qui avance en ligne droite. On va voir qu'on peut également quitter ce schéma linéaire et utiliser différentes **branches** pour s'organiser comme bon nous semble.

I. Jouer avec les branches

Il existe une commande multipotente : `git branch`. Elle permet de créer, lister ou supprimer des branches.

Utilisée telle quelle, elle liste toutes les branches existantes.

Pour créer une branche, on l'utilise comme suit : `git branch <nom de la branche>`.

Cela crée une branche **à partir du commit courant**. Dès lors, les commits que nous faisons sont envoyés sur cette branche. Nous avons donc "bifurqué" par rapport à la branche de base, que l'on appelle *branche principale*, *master* ou encore *main*.

Si on rajoute l'option `-b`, après la création de la branche, le curseur est positionné dessus : `git branch -b <nom de la branche>`.

Pour supprimer une branche, on utilise l'option `-D` : `git branch -D <nom de la branche>`.

II. Se déplacer dans l'historique

Pour se déplacer dans l'arbre git de notre projet, on peut utiliser `git checkout`. Cette commande est ambivalente, et fonctionne pour les commits comme pour les branches : `git checkout <branche>` ou `git checkout <commit>`.

Cette commande "déplace le curseur" à l'endroit désiré et remet donc l'espace de travail à l'état décrit par la version sur laquelle on arrive.

III. Fusionner une branche

Si on possède plusieurs branches pour travailler, on a souvent besoin après coup de les **fusionner** pour réunir toutes les modifications sur la même branche, et ainsi avoir un projet qui avance.

Pour ce faire, il existe la commande `git merge <branche>`. Cela fusionne la branche désignée **dans la branche courante**.

S'il existe des conflits (s'il existe deux versions d'un même fichier), git va nous l'indiquer et les afficher dans les fichiers dans lesquels il y a des conflits. Il va afficher, dans le fichier directement, quelque chose qui ressemblera à ce qui suit :

```
<<<<<<< yours:sample.txt
Ici se trouve une version du fichier concerné.
Ceci sur la branche courante...
=======
```

```
Ici se trouve la version de la branche entrante (celle qu'on cherche à faire fusionner).
>>>>>>> theirs:sample.txt
```

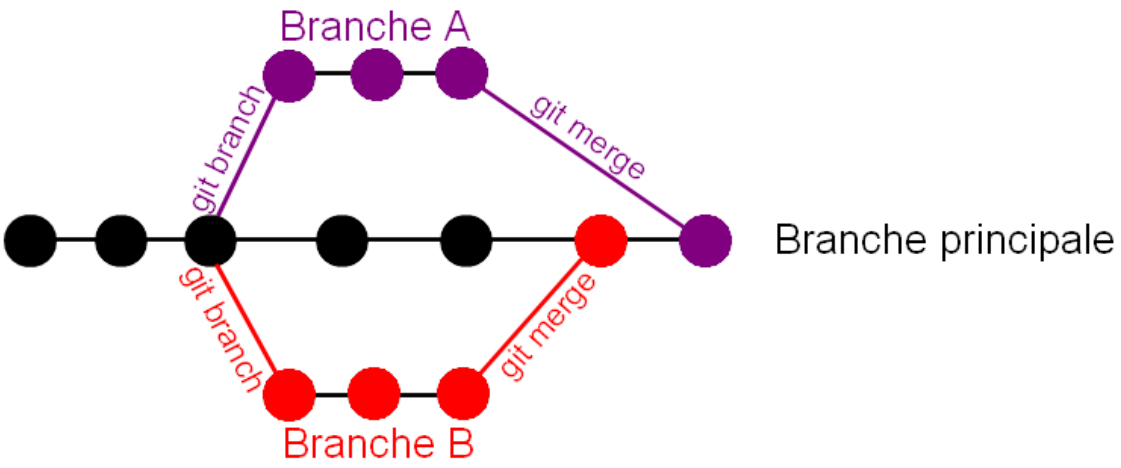
On peut retrouver les deux versions entre les chevrons (`<<<<<<<` et `>>>>>>>`), séparées par les `=======`.

IV. Le gitignore

On peut souhaiter que git ne s'occupe pas de certains fichiers de notre projet si, par exemple, ces fichiers contiennent des informations sensibles (mots de passe, clefs de chiffrement, …). On peut également souhaiter retirer des fichiers trop volumineux et qui peuvent être retrouvés facilement (souvent le cas des *vendor* (les plugins et librairies externes), des images, vidéos, et autres assets).

Pour ce faire, on peut créer un (ou plusieurs) fichier(s) `.gitignore`. Tous les fichiers, tous les dossiers, toutes les *wildcards* listés dans ce fichier seront ignorés par git.

V. Résumé des notions vues

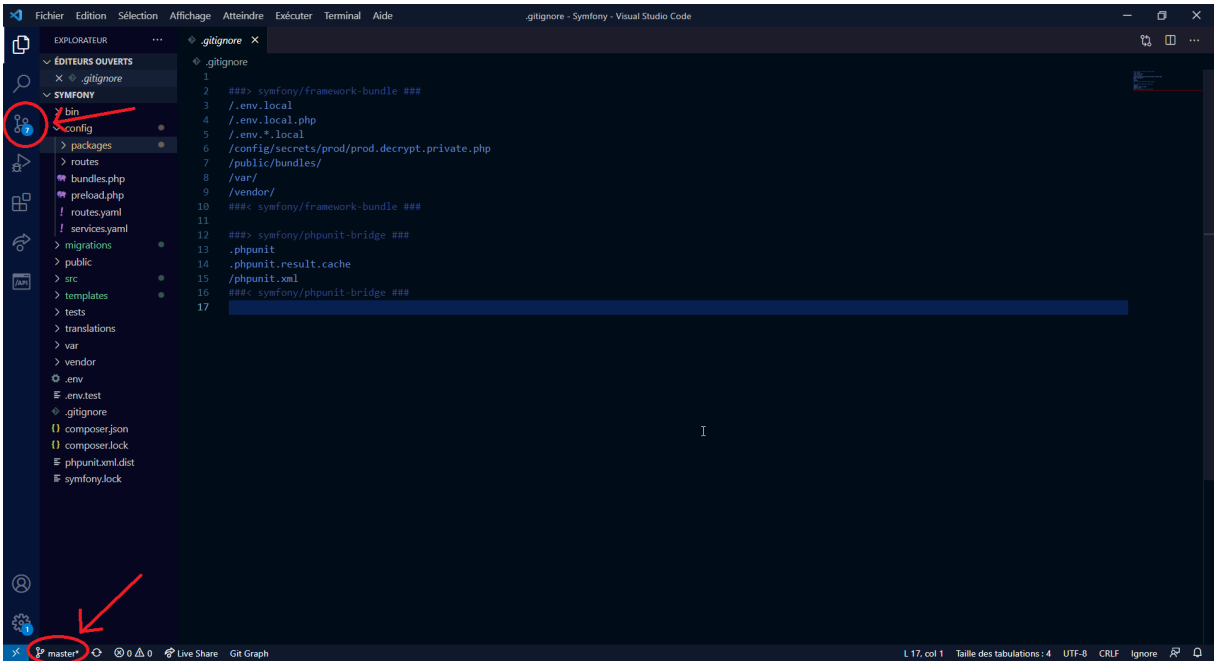


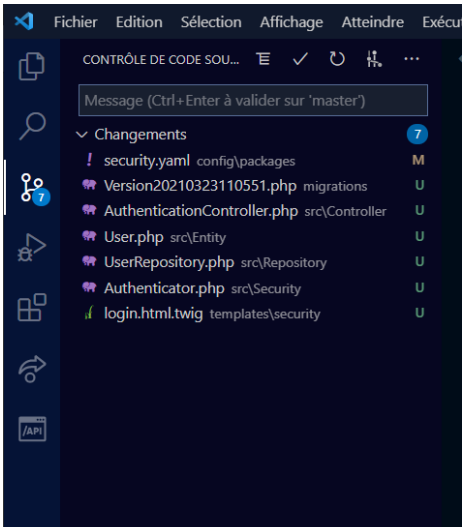
F. Sur VS Code

Toutes les commandes que vous avez vues étaient des commandes à réaliser dans un terminal. Mais VS Code intègre git dans son sein, et propose une utilisation plus aisée, avec une interface graphique dédiée.

On peut d'ores-et-déjà observer que, quand VS Code détecte qu'il se trouve dans un dossier géré par git, il affiche des couleurs sur les fichiers et les lignes. Ces couleurs sont : vert (quelque chose a été créé ou ajouté) ; orange (quelque chose a été modifié) ; rouge (quelque chose a été supprimé).

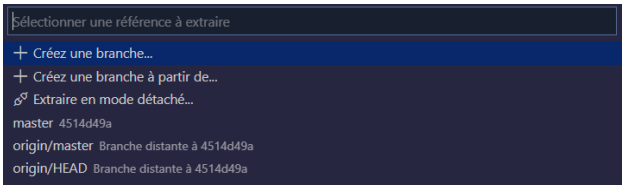
Il y a également (de base) 2 zones distinctes pour git. Une à gauche des fichiers, et l'autre en bas à gauche de la fenêtre, avec le nom de la branche.





← La zone de gauche ouvre l'onglet git, qui permet de gérer les fichiers modifiés et d'exécuter des commandes git **relatives aux commits**. (Des commandes supplémentaires, comme le push, pull, fetch, … sont cachées dans le menu "...")

La zone du bas-gauche ouvre un menu qui permet de gérer ce qui est **relatif aux branches**. ↓



On peut aussi rajouter une extension, Git Graph, qui nous permet d'avoir une **interface graphique** représentant notre arbre git. Elle vient se placer en bas de la fenêtre et on y accède en cliquant dessus.

G. Workflow recommandé avec git

Le workflow (la “manière de travailler”) recommandé avec git, pour une utilisation en équipe, est le suivant :

1. Un membre de l’équipe initialiser le dépôt distant (sur GitHub, par exemple)
2. Un membre de l’équipe clone et initialise le projet sur la branche principale (en mettant les fichiers communs, par exemple une stack Symfony si le projet est sous Symfony)
3. Le membre crée un fichier README.md à la racine du projet
4. Le membre commit puis push cette initialisation
5. Tous les membres de l’équipe clonent le projet
6. Chaque membre fait une nouvelle branche pour travailler sur **une** fonctionnalité spécifique
7. Lorsqu’un membre a terminé son travail sur sa branche :
    a. Il teste son code (tout doit fonctionner comme prévu)
    b. Il commit tous ses fichiers et les push
    c. Il ouvre une Pull Request sur GitHub (Merge Request sur GitLab) pour proposer une fusion avec la branche principale
    d. Il résout les éventuels conflits de la Pull Request
    e. Les autres membres peuvent alors :
        i. Relire la Pull Request pour vérifier que le code fourni correspond aux exigences de qualité du projet
        ii. Valider la Pull Request (ceci a pour effet de fusionner la branche sur la branche principale)
8. Le membre qui a fusionné sa branche peut :
    a. Se remettre sur la branche principale
    b. Pull
    c. Refaire une branche pour travailler sur une autre fonctionnalité
9. Les autres membres ne devront pas oublier de récupérer les changements effectués sur la branche principale (en faisant un fetch ou un pull)

Ce workflow présente plusieurs avantages :

- il évite que les membres de l’équipe ne travaillent sur la même chose (ce qui créerait des conflits de versions)
- il permet de relire le code pour le valider
- il permet d’avoir une branche principale qui ne contient **que** du code “propre” (puisque validé)
- il permet de bien différencier chaque fonctionnalité puisque chacune a sa propre branche

Liens utiles

- Une documentation de git : https://git-scm.com/book/fr/v2 (en français)
- OpenClassrooms : Gérez du code avec Git et GitHub (en français)
- OpenClassrooms : Utilisez Git et GitHub pour vos projets de développement (en français)
- Le dossier contenant toutes les ressources de ce module : Google Drive
- Youtube : [Cours Github] Apprendre Github de zéro : versionner son travail (en français)
- Une *cheat sheet* GitHub : Image

Glossaire

Glossaire organisé par ordre de rencontre :

- git
- gestionnaire de versions
- version
- mutualiser
- décentralisé
- GitHub
- dépôt
- remote
- cloner
- commit
- branche
- fusionner
- onglet git
- workflow