***La gestion de données :***

La gestion de données est ce qui permet de gérer les données d’une appli ou un site que l’on créé sur le temps.

Par exemple, si on veut revenir en arrière ou gérer deux versions différentes en même temps, il faut toujours garder une trace des versions antérieures pour gérer les problèmes.

Il est possible de faire ça en local en faisant un document avec chaque semaine de dev, mais c’est contraignant dans la mesure ou il est difficilement possible de savoir ce que contient telle semaine et ce qui a changé d’une semaine à l’autre.

Il est possible de gérer ça via un serveur, mais si celui-ci est down, les utilisateurs se retrouvent bloqués (les devs qui codent). En revanche, si le serveur marche, on sait qui a modifié quoi et quand.

Enfin, il est possible de gérer ça via un serv qui va en même temps envoyer l’intégralité du fichier à tous les utilisateurs pour pouvoir continuer le travail même en cas de crash serveur. Cette méthode permet également de savoir qui a fait quoi et quand.

Le logiciel le plus utilisé est ***git.***

***A savoir :***

Un fichier staged est prêt à être intégré à la base de donné mais c’est le commit qui l’intègrera.

**Git config --global alias.raccourci « chose qu’on veut raccourcir sans le mot git »** permet de créer un alias.

Il est possible de créer des raccourcis directement dans le système. Il suffit de faire cd pour arriver dans le système.

**Touch .bash\_profile** pour créer le dossier qui contiendra les raccourcis.

**Vi .bash\_profile** pour rentrer les alias directement depuis git

Appuyer sur **i** pour écrire.

**Alias raccourci=’la formule git’** par exemple **alias c=’clear’** ou alias **gita=’git add .’**

Appuyer sur **echap** et faire **:x** pour enregistrer.

Il suffit juste de relancer git pour que ça soit effectif.

***Les commandes les plus utilisées sur Git :***

**Mkdir :** créer un répertoire. (créé un dossier, inutile s’il est déjà existant.)

**Cd  nomdedossier:** changer de répertoire. ( par exemple passer du :C au bureau ou à un fichier précis. Cd tout seul mène au dossier système.

Pour revenir au dossier parent (par exemple d’un dossier sur le bureau) on met **cd ..**

le .. fonctionne comme en HTML et CSS et permet de revenir au parent.

**Git init :** dit que le répertoire contiendra notre projet. (créer un nouveau projet)

**Ls :** lister les fichiers. Dans le dossier.

**Cat nomdefichier:** afficher le contenu

**Pwd**:afficher le nom du répertoire courrant.

**Touch nomdefichier :** créer un fichier.

**Git clone URL** : créé une copie exacte du fichier ciblé par l’URL pour pouvoir travailler dessus. (reprendre un projet existant).

**Git status :** permet de savoir ce qu’il y a dans le fichier actuellement. S’il dit untracked, ça signifie que git ne suit pas ce fichier.

***Configurer git pour qu’il nous reconnaisse en tant qu’utilisateur sur un projet. S’il y a plusieurs utilisateurs, on sait grâce à ça qui a modifié quoi et quand. Il faut faire :***

**Git config --global user.name « antoine » :** configurer le nom

**Git config --global user.email « leinhou@gmail.com» :** configurer l’email.

**Git config --list**: permet de vérifier ce qui a été ajouté dans la configuration globale.

**Git add nomdefichier :** pour ajouter un fichier dans la base de données git.

**Git add \* :** permet de tout ajouter (le plus efficace).

**Git add \*.\*** : permet d’ajouter les fichiers qui ont une extension (utile sous Linux).

**Git add .** envoie tout le répertoire courrant.mgit sta

**Git show nomdefichier :** montre ce qu’il y a dans le fichier, sa date de création, son auteur, ses dernières modifs… C’est un cat en plus complet.

**Git commit :** permet de faire une sauvegarde du fichier actuel et mettre un commentaire pour dire ce qui a été affiché. Quand on est entré dans l’éditeur de commentaire, il faut taper **i** pour rentrer le commentaire **et echap :wq** pour sauvegarder.

**Git log :** permet de montrer qui a modifié quoi à quelle heure et les commentaires.

**Git log -- oneline** : permet de voir juste les commentaires.

**Git commit –m  « commentaire »** : pour commenter directement sur git sans passer par l’éditeur, mais le commentaire doit rentrer sur la ligne donc être court.

**Git commit –a –m « commentaire »** fait la maj du fichier et le commentaire en même temps.

**Git diff :** montre les modifications entre mon fichier et le fichier qui a été modifié. Cette fonction montre la diff entre le répertoire de travail et ce qui est staged.

**Git diff –cached :** c’est la diff entre ce qui est staged et ce qui est enregistré dans la base de données. Pour q’un fichier staged entre dans la base de données il faut forcément le commenter.

**.gitignore (fichier à ignorer ou alors \*type de fichier pour ignorer les fichiers qui font partie de ce type)** permet d’ignorer les fichiers qu’on ne veut pas mettre à jour quand on fait git add. Il faut créer le .gitignore faisant la commande touche .gitignore pour le créer et surtout ne pas ignorer ce fichier.

**Git config –global core.editor « chemin vers vscode avec les / doublés » :** pour écrire les commentaires avec vscode.

**Git rm nomdefichier** permet d’effacer un fichier cd

62014-06-11@STA6100339 MINGW64 ~

$ pwd

/c/Users/62014-06-11

62014-06-11@STA6100339 MINGW64 ~

$ cd desktop

62014-06-11@STA6100339 MINGW64 ~/desktop

$ cd test

62014-06-11@STA6100339 MINGW64 ~/desktop/test

$ mkdir test

62014-06-11@STA6100339 MINGW64 ~/desktop/test

$ git init

Initialized empty Git repository in C:/Users/62014-06-11/Desktop/test/.git/

62014-06-11@STA6100339 MINGW64 ~/desktop/test (master)

$

***Récupérer des fichiers :***

Si on voit une diff sur un fichier qu’on a pas encore add (on dit qu’il est « non staged ») , pour récupérer la version précédente il faut faire :

**Git checkout nomdefichier**

Le checkout permet de revenir à la version précédente après par exemple un git reset signature.

**Git reset HEAD nomdefichier :** permet de permet de revenir au fichier avant celui qui a été add (donc staged) mais pas encore commit. Une fois cette étape faite, il suffit de refaire un git checkout nomdefichier pour revenir au fichier précédent. Cette commande permet de « destage » un fichier.

***Pour récupérer un commit précédent :***

**Git reset signature(appelé SH-1) nomdefichier** pour repointer cette version précédente.

**Git checkout nomdefichier** pour récupérer la version précédente.

**Les interfaces graphiques de git :**

**Git gui &** dans git permet d’avoir une interface qui permet d’observer les fichiers pas encore commit pour voir les modifications et les commit précédents afin de les gérer au mieux.

**Gitk &** permet de voir les différentes versions avec le contenu, voir celles qui n’ont pas encore été commit et l’ordre et ceux qui ont modifié dans le temps. C’est un git log avec un got diff et un git show intégré mais plus facile d’accès. On peut démarrer git gui depuis gitk depuis le menu file si on veut commit un add.

**Gitk --all** **&** permet de montrer un gitk avec les branches.

En faisant **gitk nomdebranche &** on peut voir uniquement le chemin suivi par une branche. Ça marche aussi pour le master.

***Donner un tag à un commit :***

Un tag est une référence, un nom, que l’on donne à un commit.

Pour donner un nom à un commit : **git tag –a nomdutag –m « commentaire descriptif »**

Pour enlever le nom à un commit : **git** **tag –d nomdutag**

**Pour créer une branche :**

Une branche est une version parallèle d’un commit. Par exemple, si l’on veut développer deux versions différentes à partir d’un commit, on fera un dev principal sur le master et on va créer une branche pour dev une version secondaire. C’est utile par exemple pour une version d’un site que l’on veut aussi mettre à jour sur mobile.

**Git branch nomdelabranche** pour la créer.

**Git checkout nomdelabranche** pour passer sur la branche

**Git checkout master** pour revenir au master

**Git checkout -b nomdelabranche** créé la branche et passe directement sur la nouvelle branche.

**Git branch** montre toutes les branches existantes. Celle marquée d’une étoile est celle active.

**Git branch –v** montre le dernier commit de chaque branche.

**Git branche –m nomdelabranche** permet de renommer la branche actuelle. La c’est la branche qui est renommée, pas le commit comme le ferait un tag.

**Pour delete une branche :**

**Git branch –d nomdelabranche**

Une branche va alterner les fichiers dans le dossier. Par exemple, le fichier file1 de la branche ne sera pas le même que celui du master. Si on créé un autre fichier dans le dossier et qu’on l’ajoute à la branche, si on retourne sur le master on ne le verra pas. Au final, c’est comme si c’était deux dossiers différents. Le dossier qui n’est pas utilisé est en réalité stocké dans la base de donnée. Les deux fichiers file1 sont indépendants les uns des autres.

**Pour fusionner des branches :**

Depuis la branche principale que l’on veut conserver on fait.

**Git merge « nomdelabranche » :** le nom de branche est celui de celle que l’on veut fusionner avec la branche principale. S’il y a des fichiers différents, tout se passera bien et dans ce cas la git proposera un commit.

S’il y a deux fichiers identiques : git ne peut pas choisir pour nous ce qu’il faut conserver. Il faut aller dans le fichier et delete ce que l’on ne veut pas garder. Il faut également supprimer les = qui séparent ce que contient les deux branches et les chevrons (<<< et >>>) qui montrent à quelle branche appartient le contenu. On peut même ajouter un nouveau contenu***. Il faut juste faire un git add . et un commit après la modification pour que la fusion soit effective.***

***Le temps que le commit final n’a pas été validé, le fichier est en cours de merge.***

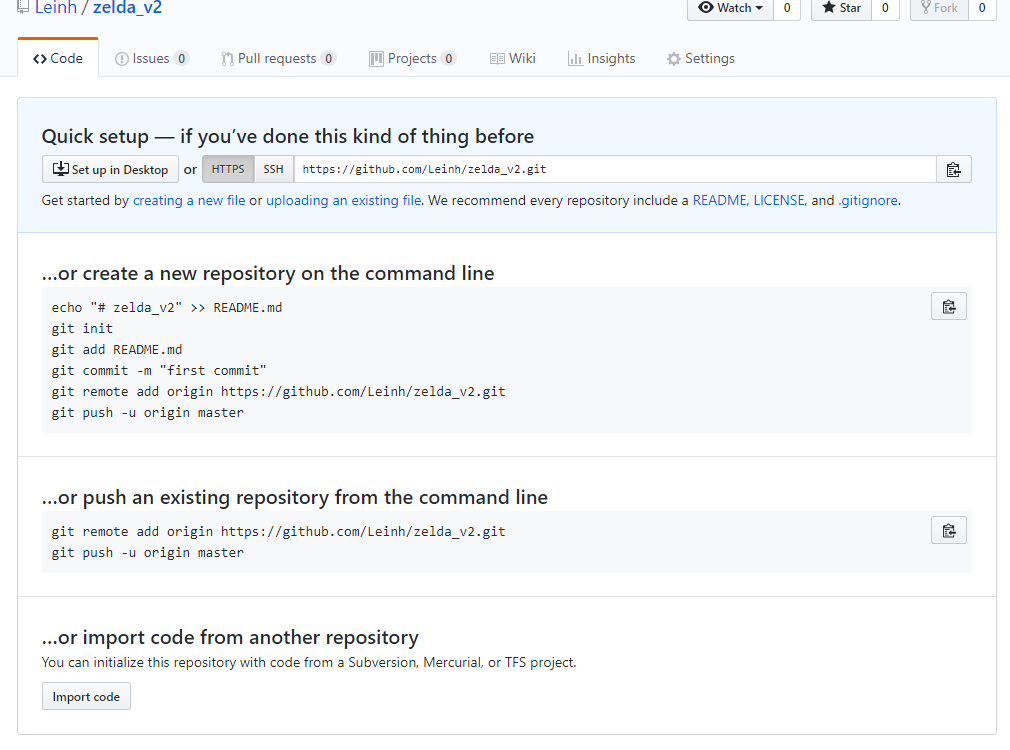
**Pour annuler un merge en cours :**

**Git reset --Hard HEAD**

Tant que le commit n’a pas été validé, les fichiers différents seront merge. Annuler un merge annulera le conflit entre les fichiers communs mais retirera les fichiers différents du dossier auquel ils n’appartiennent pas. C’est utile pour revenir en arrière si on se rend compte qu’il est trop tôt pour merge afin de corriger par avance le plus de conflits possible.

Un merge créé un nouveau commit, donc une nouvelle version. Il faut prendre ça en compte pour ne pas créer de versions inutiles.

**Pour lier un git à internet afin de le reprendre par exemple chez-soi ou travailler de partout :**



Si on a un dossier ou on a fait git init pour le lier à git, on peut directement commencer par l’étape **git remote add origine url**

Une fois passé l’étape du **git push –u origin master** , il faut entrer ses coordonnées sur le login qui est apparu dans la barre des tâches et le git va s’envoyer.

**Pour récupérer un git la première fois :**

**Git clone url.**

**Pour envoyer sur la data base un git déjà partagé :**

**Git push origin master (ou nomdelabranche si on bosse sur une branche)**

**Pour DL la maj d’un git partagé sur internet :**

**Git pull origin master**

**Pour vérifier si le git est bien relié à un lien :**

**Git remote –v**

On peut remote un git sur un autre serveur.

Il suffit de remplacer le origin par le nom d’un serveur créé préalablement comme par exemple :

**Git remote add exemplenomdeserveur url**

Ensuite, pour envoyer l’update vers le serveur, il suffira de mettre :

**Git push exemplenomdeserveur master**

Origin restera le nom du serveur principal. Ça permet d’envoyer une version publique vers un serveur en commun par exemple et une version privée sur son serveur personnel.

**Git revert HEAD** permet d’annuler le dernier commit enregistré pour revenir à la version d’avant le dernier commit.

**Git rebase master** *depuis une branche* permet d’avoir les branches dans le master pour ceux qui préfèrent avoir une seule ligne. Les fichiers ne changent pas, uniquement la visibilité change. Visuellement, la branche apparaîtra après les fichiers du master.

**Git fetch origin master** fait comme le pull mais dl un autre fichier identique pour qu’on puisse voir les modifs avant de les fusionner avec un pull. Le git pull va merger le local avec ce qui est stocké sur git hub. Git fetch ne fusionnera rien.