MASART Raphaël

B1-Git - Partie 1 : Création du dépôt et *commits*

* Ouvrir un terminal (terminal *Git Bash* à privilégier)

Une image contenant texte, capture d’écran, multimédia, logiciel

Description générée automatiquement

* Créer, en ligne de commande, un répertoire tp-git

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Se déplacer dans le répertoire

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement

* Vérifier qu’on est dans le bon répertoire en affichant le chemin du répertoire courant

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement

* Initialiser un dépôt Git

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Lister tous les fichiers du répertoire (y compris les fichiers cachés) pour s’assurer que le répertoire .git à été créé

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement

* Ouvrir ce répertoire sous VS Code

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

* Exécuter git status et copier/coller la sortie

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Créer le fichier fichier1.md avec un contenu quelconque et l’enregistrer (vous pouvez utiliser VS Code pour créer et éditer des fichiers)
  + Attention, il s’agit bien de créer un nouveau fichier, et non un répertoire. Il en sera de même tout au long de ce TP.

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran, Police, nombre

Description générée automatiquement

* Créer le fichier fichier2.md avec un contenu quelconque et l’enregistrer

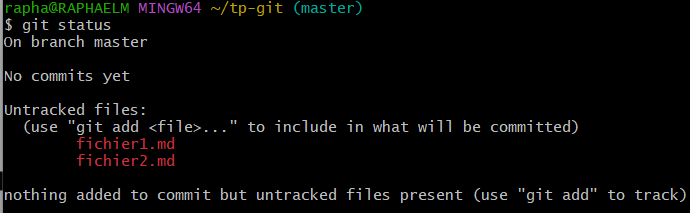
Une image contenant texte, Police, capture d’écran

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Exécuter git status et copier/coller la sortie



* Ajouter fichier1.md à l’index de Git (zone de *Staging*)



* Exécuter git status et copier/coller la sortie

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Créer un *commit* avec pour message : “Ajout de fichier1.md”

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* ***VALIDATION PROF01***
* Exécuter git status et copier/coller la sortie

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* Modifier fichier1.md et enregistrer

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Exécuter git status et copier/coller la sortie

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Ajouter fichier1.md et fichier2.md à la zone de *Staging*

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* Créer un *commit* “Ajout de fichier2.md et modification de fichier1.md”

Une image contenant texte, capture d’écran, Police

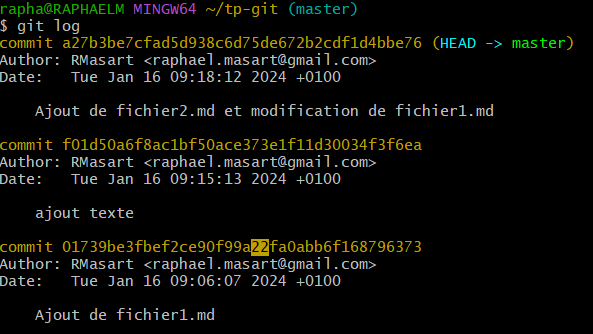
Description générée automatiquement

* Exécuter git status et copier/coller la sortie

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* Copier/coller l’ID des deux premiers *commits* (utiliser log) :
  + ID *commit* 1 : f01d50a6f8ac1bf50ace373e1f11d30034f3f6ea
  + ID *commit* 2 : a27b3be7cfad5d938c6d75de672b2cdf1d4bbe76



* Que signifie qu’un fichier est “*tracked*” ou “*untracked*“ ?

C’est lorsqu’un fichier et suivi / surveiller par git

* Pourquoi doit-on passer les fichiers par la zone de *Staging* (l’index) avant de les committer ?

Cela permet que git bash reconnait le fichier car sinon il peut le voir mais ne peut pas le commit

* ***VALIDATION PROF02***

B1-Git - Partie 2 : Gestion de branches

*Cette partie est à faire sur le même dépôt que la partie précédente. C’est la suite.*

* Créer une branche fonctionnalite1



* Lister les branches

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* Se déplacer sur la branche fonctionnalite1

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Lister les branches

Une image contenant texte, capture d’écran, Police, Graphique

Description générée automatiquement

* Que représente l’étoile à côté des noms des branches ?

Cela représente ou se situe l’utilisateur

* Créer un nouveau fichier fichier3.md

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Modifier le fichier fichier2.md
  + Comment utiliser VS Code pour qu’il nous montre les différences entre l’ancienne version de fichier2.md et la version courante que l’on vient d’éditer ?

Dans la zone ou l’on peut commit, on peut remarquer la modification

Une image contenant Logiciel multimédia, logiciel, Logiciel de graphisme, capture d’écran

Description générée automatiquement

* Committer ces deux modifications : “Fonctionnalité 1 - première phase”

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Créer un nouveau fichier fichier4.md

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement

* Modifier de nouveau le fichier fichier2.md

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Committer ces deux modifications : “Fonctionnalité 1 - terminée”

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* ***VALIDATION PROF03***
* Afficher la liste des fichiers du répertoire

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Se déplacer sur la branche master



* Afficher la liste des fichiers du répertoire

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement

* Pourquoi les deux sorties sont-elles différentes ? Les fichiers ont-ils disparus ?

Car on n’est pas dans la même branche donc les fichiers ajouter dans une branche ne modifie pas les autres

* Créer une nouvelle branche fonctionnalite2



* + Cette branche ne va pas avoir toutes les données incluses dans fonctionnalite1. Pourquoi ?

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement

Car la branche fonctionnalite2 a été créer dans la branche master et à récupérer ses fichiers (dossier parent-enfant)

* + Qu’aurait-il fallu faire si on avait souhaité démarrer la branche fonctionnalite2 en intégrant les modifications récentes de fonctionnalite1 ?

Il fallait créer la branche en se déplaçant dans fonctionnalite1

* Se déplacer sur la nouvelle branche fonctionnalite2

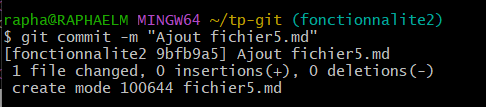
Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* Créer un nouveau fichier fichier5.md



* Faire un *commit* intégrant cette ajout : “Ajout fichier5.md”



* Entrer la commande git log --oneline --decorate --graph --all pour visualiser, sur le terminal, le graphe des *commits* sur toutes les branches

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* + Noter la « déviation » entre les deux branches, à partir de la branche master (schématisée sous forme de traits)
  + l’option --all permet de visualiser toutes les branches, pas seulement celle sur laquelle on est
  + l’option --oneline affiche les *commits* sur une seule ligne
  + l’option --graph affiche le log sous forme de graphe
  + (utilisez si besoin les touches haut/bas pour naviguer dans la sortie de cette commande et Q pour quitter)
* Installer l’extension VS Code *Git Graph* et visualiser le graphe actuel des *commits* à l’aide de cette extension
  + Sur cette représentation, que représente les points ?

Les points représentent tous les commits effectuer

* + Comment voit-on sur quelle branche on est actuellement ?

Les branches sont représenter par des lignes de couleurs avec chaque commits en points

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, logiciel, Logiciel multimédia, capture d’écran

Description générée automatiquement

* ***VALIDATION PROF04***

Partie 3 - Fusionner des branches

*Cette partie est à faire sur le même dépôt que la partie précédente. C’est la suite.*

* On considère que la branche originale (master ou main) est la branche d’intégration, c’est-à-dire celle qui va contenir l’historique de toutes les modifications développées au fur et à mesure dans les branches annexes
* On va maintenant fusionner la branche fonctionnalite1, qui est terminée, avec la branche d’intégration
* Se déplacer sur la branche master
  + Noter le changement dans l’onglet *Git Graph*

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Fusionner avec la branche fonctionnalite1

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Noter le changement dans l’onglet *Git Graph*. Que signifie la mention *Fast-forward* indiquée par la sortie de la commande ?

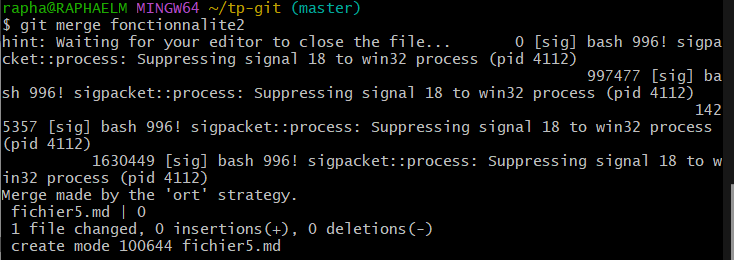
Cela veut dire qu’il ajoute, change et supprime des fichiers de manière a ce que master récupère fonctionnalite1

* On veut maintenant fusionner fonctionnalite2 dans la branche d’intégration (master)
* Effectuer cette fusion

Ouverture de Vi :

* i pour passer en mode *insertion* (le mode par défaut est *commande*, dans lequel vous ne pouvez pas écritre de texte)
* Esc (touche Echap du clavier) pour revenir au mode *commande*
* :wq! pour enregistrer et quitter (***w****rite* et ***q****uit*)
* :q! pour quitter sans enregistrerUne image contenant texte, Appareils électroniques, capture d’écran, logiciel

  Description générée automatiquement



* Noter le changement dans l’onglet *Git Graph*. Que signifie la mention *Merge made by the … strategy* indiquée par la sortie de la commande ?
* Quelle est la différence fondamentale avec la fusion précédente ?

Vi nous a bloquer pour ajouter un message commit nécessaire pour la fusion

* Créer une nouvelle branche fonctionnalite3, se déplacer dessus, et modifier le fichier fichier1.md en y ajoutant une ligne de texte. Committer : “Modification fichier1 pour fonctionnalité 3”

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* + Comment utiliser *Git Graph* pour qu’il nous montre les différences entre l’ancienne version de fichier1.md et la version courante que l’on vient de committer ?

Une image contenant texte, logiciel, Logiciel multimédia, capture d’écran

Description générée automatiquement

* Repartir sur master, et modifier fichier1.md en y ajoutant aussi une ligne (différente de celle qu’on a ajoutée sur l’autre branche) ; ajouter à l’index et *commit*

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement

Une image contenant texte, Police, capture d’écran, nombre

Description générée automatiquement

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Tenter de fusionner la branche fonctionnalite3 avec master
  + Que se passe-t-il et pourquoi ?

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Une image contenant texte, capture d’écran, logiciel, Police

Description générée automatiquement

On tente de fusionner mais il y a un conflit avec les fichiers

* Lancer un git status

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* + Que doit-on faire si on veut annuler la fusion en cours ? (**ne pas lancer la commande**)

On fait une marche arrière avec git merge --abort

* On veut résoudre le conflit. Plusieurs possibilités :
  + Conserver uniquement les modifications faites dans fonctionnalite3
  + Conserver uniquement les modifications faites dans master
  + Conserver les deux modifications
  + Supprimer les deux modifications ou remanier sensiblement le fichier pour les intégrer correctement
* Git nous laisse totalement la main et ne va pas essayer d’imposer l’un de ces choix pour nous, ni nous assister dans l’application automatique de l’un d’entre eux : il faut examiner le(s) fichier(s) en conflit et éditer nous-mêmes
* Ouvrir le fichier en question sous VS Code
  + La chaîne <<<<<<<<<< marque le début du conflit
  + La chaîne >>>>>>>>>> marque la fin du conflit
  + La chaîne ========== sépare les deux versions

Une image contenant texte, capture d’écran, logiciel, Logiciel multimédia

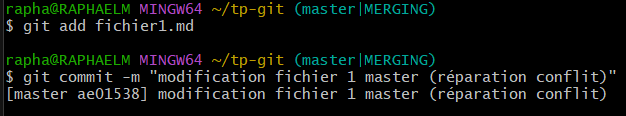
Description générée automatiquement

* Éditer le fichier pour faire en sorte d’intégrer les deux modifications ; à la fin de l’édition :
  + Il ne doit plus y avoir de marques quelconques en dehors des ajouts fonctionnels originaux, c’est-à-dire pas de <<<<<<<<<<, ni de mentions de nom de branche, etc. : vous rendez le fichier tel qu’il doit apparaître dans le *commit* de fusion, avec les conflits résolus manuellement
  + Sauvegarder

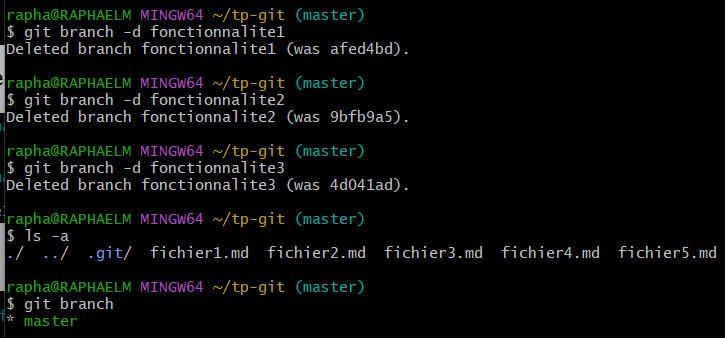
Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Ajouter les modification à l’index et committer



* NB : parfois, plusieurs fichiers sont en conflit ; le processus est identique, il faut juste résoudre les conflits sur tous les fichiers
* NB : les conflits de fusion sont fréquents lorsqu’on travaille en collaboration (plusieurs personnes vont travailler sur le même fichier pour remplir deux fonctionnalités différentes)
* Les branches créées n’ont plus de raison d’exister
  + Elles avaient pour but de créer une déviation afin de travailler sur des fonctionnalités individuelles, sans interférer avec le travail des autres, avant de 300les fusionner dans la branche d’intégration
  + On va vouloir nettoyer le dépôt en les supprimant
  + Cela ne va bien sûr pas supprimer tous les *commits* qui y sont associés
  + Attention cependant d’éviter en général de supprimer une branche qui n’a pas encore été intégrée à la branche d’intégration, sauf on souhaite vraiment abandonner le développement de cette branche
  + Ne pas réutiliser une branche qui a déjà été intégrée pour démarrer une nouvelle piste : toujours utiliser une nouvelle branche
  + Nouvelle tâche ? => nouvelle branche à partir d’un *commit* de la branche d’intégration (en général le plus récent)
  + Tâche terminée ? => fusion dans la branche d’intégration et suppression de la branche
* Supprimer les trois branches fonctionnalitex (attention : on ne peut pas supprimer une branche sur laquelle on est)



Partie 4 - Travailler avec un dépôt distant (*remote*)

* Faire un *fork* du dépôt https://github.com/rose-line/sio1-2024-java-grpb (pas par commande Git, c’est sur l’interface GitHub)
* Cloner localement votre *fork* (**ne pas cloner le dépôt original**)

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* + Quelle est la différence entre un *fork* et un clonage ?

Un clone permet de faire un copier-coller en local d’un fichier du cloud au dossier local alors qu’un fork et un copier-coller mais en ligne qui permet de modifier un fichier commun

* + Indiquer dans quelles circonstances on voudrait *forker* et/ou cloner un dépôt

Cloner pour avoir le dossier en local et travailler seul dessus

Forker pour travailler sur le dossier et pouvoir l’envoyer plus facilement sur un groupe en ligne

* Modifier le fichier README.md à la racine du dépôt en ajoutant une ligne quelconque

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* On veut maintenant envoyer cette modification vers le dépôt distant
  + Il faut d’abord faire un *add/commit* local

Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement



Une image contenant texte, Police, capture d’écran, ligne

Description générée automatiquement

* + Puis utiliser la commande qui « pousse » les modifs sur le dépôt GitHub (*push*)



* + Vérifier directement sur GitHub que le *push* a bien fonctionné



* Trouver la commande qui affiche le nom du ou des dépôt(s) distant(s) relié(s) avec le dépôt local : cela permet de savoir si le dépôt courant est synchronisé avec un dépôt en ligne ou non

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

Git remote -v

* On va faire un *merge* en local puis *push* :
  + Créer une branche locale bugfix1, se déplacer dessus, créer un nouveau fichier ok.java à la racine du dépôt

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* + Ajouter ok.java à l’index et faire un *commit*

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* + Retourner sur master, créer le fichier ajout.java, ajouter à l’index et committer

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* + Fusionner la branche bugfix1 dans la branche master

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* + Afficher le log des *commits* ; noter les emplacements des trois branches différentes, en local et en remote

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* + Faire un *push*

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* + Refaire un affichage du log ; origin/master a bougé : que représente cette branche ?

Cette branche représente le dépôt distant (quand on publie le fichier, c’est lui qui bouge)

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* Étudier le résultat sur GitHub, en examinant *commits* et branches (bouton *drop-down* sur la page du dépôt pour voir les branches) : qu’est-ce qui est différent de la version locale ?

Ce qui est différent de la version locale a git et que l’on peut voir les commits effectuer au fil du temps

Une image contenant texte, logiciel, capture d’écran

Description générée automatiquement

* Le bug est corrigé et intégré ; que doit-on faire de la branche bugfix1 maintenant ?

On doit supprimer la branche bugfix1

* ***VALIDATION PROF06***
* Supposons que l’on veuille effectivement publier sur le *remote* une branche sur laquelle on travaille (pour sauvegarde ou pour que d’autres puissent l’utiliser)
  + Créer une nouvelle branche partage



* + Aller sur la branche

Une image contenant texte, Police, capture d’écran, Graphique

Description générée automatiquement

* + Ajouter un fichier partage.md



* + L’inclure dans l’index



* + Faire un *commit*

Une image contenant texte, Police, capture d’écran

Description générée automatiquement

* + *Push*

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* + Que se passe-t-il ?
  + Exécuter la bonne commande pour sauvegarder la branche sur le remote

Une image contenant texte, capture d’écran, Police

Description générée automatiquement

* + Vérifier sur GitHub que la branche apparaît bien
* La branche partage n’a pas été fusionnée avant le *push* ; on va utiliser un autre moyen offert par GitHub pour fusionner une branche en *remote* : la ***Pull Request***
  + La *Pull Request* est très utilisée en collaboration : elle permet à l’intégrateur du projet d’examiner les demandes de *merge* au niveau du *remote* avant de les accepter (ou non)
* Sur GitHub, cliquer sur le bouton *Compare & pull request*, qui apparaît directement sur la page du dépôt depuis le dernier *push* qui a ajouté la branche (voir ci-dessus).
* À gauche, la branche d’intégration (« *base* », qui reçoit le *merge*) ; à droite, la branche à fusionner (« *compare* »)
  + On doit fusionner partage dans master
  + Attention, il faut bien choisir le repo de base qui est sur votre propre compte (pas le compte du dépôt d’origine que vous avez *forké*)
  + On peut ajouter d’autres informations : titre et commentaire de la *pull request*, et même upload de fichiers annexes, liens éventuels… Également, on peut ajouter des labels pour étiquetter la *pull request* et lui affecter un *reviewer*, qui va officiellement être en charge
  + Valider en cliquant sur le bouton *Create pull request*
* La page résultante informe sur les branches source et cible, sur les *commits* concernés, les fichiers qui ont été modifiés…
* On peut aussi y démarrer une conversation notamment entre l’émetteur de la *pull request* et l’intégrateur
* On voit que l’intégrateur (possesseur du compte cible) peut accepter la *pull request* en cliquant sur le bouton *Merge pull request* (ou refuser en cliquant sur *Close pull request*).
* En discutant de la *pull request*, on se rend compte que certaines choses devraient être modifiées
  + Repartir en local pour effectuer une modification sur partage.md et ajouter precision.md
  + *Commit* des deux modifs
  + *Push*
  + Observer la *pull request* : que s’est-il passé ?
  + Finalement, faire le *merge* sur la page de la *pull request*
  + Noter que l’interface nous propose alors de supprimer la branche devenue inutile ; supprimer la branche
* Dans un contexte de travail en collaboration sur un même dépôt, donner un *workflow* (façon de travailler) possible qui va permettre à tous les intervenants de viser des ajouts à la branche d’intégration, d’en discuter, et ceci sans danger pour la branche d’intégration, avant que finalement l’intégrateur (probablement propriétaire du dépot) accepte les changements.
* ***VALIDATION PROF07***