CHAPITRE 2

1

CHANGER LA VERSION DU FICHIER

ANNULER LES CHANGEMENTS

Nous avons plusieurs façons de changer la version du fichier, en fonction de ce que nous voulons faire. Imaginons dans un premier temps que nous souhaitons simplement annuler toutes les modifications en cours qui n'ont pas été enregistrées. (Il s'agit donc de récupérer la version enregistrée).

Pour cela nous allons utiliser git checkout sur un fichier :

```
git checkout ./README.md
```

Git nous dit :

```
Updated 1 path from the index
```

Et si nous ouvrons le README.md, les modifications (le feature 2.1) a été enlevé. Le fichier a été retourné à son état enregistré dans le dépôt.

RÉCUPÉRER LA VERSION D'UNE AUTRE BRANCHE

Nous pouvons aussi récupérer le fichier tel qu'il est dans une branche. Pour cela nous allons utiliser la commande checkout ainsi:

```
git checkout feature1 -- ./README.md
```

Si nous ouvrons le fichier, il contient le contenu de la branche feature1:

```
# Ceci est un readme

feature1
```

Pourtant si nous demandons à git de nous indiquer son état (avec git status) il nous informe que :

```
On branch main

Changes to be committed: modified: README.md
```

Autrement dit, ici, la modification du fichier README.md est déjà ajoutée à l'index, prête à être enregistrée.

RETIRER UN FICHIER DE L'INDEX

Ici, pour remettre README.md à l'état enregistré, il faut commencer par retirer les modifications de l'index, en faisant :

```
git restore --staged ./README.md
```

Si nous faisons un git status nous voyons que la modification n'est plus dans l'index mais simplement en état "non suivi" … même si le contenu du fichier lui-même n'a pas changé. (C'est l'état de l'enregistrement de la modification qui a changé, pas le fichier lui-même).

Et pour annuler ces changements, on refait :

```
git checkout README.md
```

Le fichier est alors rétabli dans la version enregistrée sur main, et comporte donc à la fois feature1 et feature2.

2
NETTOYER LA RÉSERVE ET AUTRES MANOEUVRES

Un dernier petit détail : si nous faisons un git stash list nous voyons que la modification réservée est encore enregistrée dans la liste.

Nous avons ici deux options.

Si nous voulons appliquer un élément de la réserve *et* le sortir de la liste, il faut faire git stash popau lieu de git stash apply, puisque la deuxième commande applique la modification *sans pour autant la supprimer de la réserve* , alors que git pop applique la modification, et si aucune erreur n'a été constatée, la supprime de la réserve.

Nous pouvons aussi supprimer un élément de la réserve sans l'appliquer, en utilisant git stash drop.

Par ailleurs, en tout ce qui concerne la réserve, nous pouvons spécifier quel élément de la réserve est concerné. (Pour le moment notre réserve n'a qu'un seul élément donc la question est plus simple).

Par exemple, comme notre élément est en position 0, nous pourrions faire :

```
git stash drop 0
```

Ou inversement :

```
git stash pop 0
```

Maintenant voyons comment partager son code entre plusieurs développeurs. Pour cela nous allons passer par des déports distants, en

utilisant GitHub.

3

GITHUB : CRÉATION D'UN COMPTE & SSH

Le premier pas consiste à créer son compte sur GitHub. Une fois le compte crée, il y a plusieurs possibilités pour configurer votre poste de travail pour qu'il communique bien avec GitHub.

Celui que je préfère — et qui colle le mieux avec la ligne de commande — c'est de mettre en place une authentification par le biais le clefs SSH. La documentation de GitHub est très complète à ce sujet (mais en anglais), et se trouve à l'adresse : https:// docs. github. com/ en/ authentication

Voici en synthèse ce qu'il faut faire :

VÉRIFIER S'IL EXISTE DÉJÀ UNE CLEF SSH

Pour cela, il faut tout d'abord se rendre dans son dossier personnel (qui s'ouvre par défaut lorsqu'on ouvrer un terminal). Il se trouve à un chemin qui ressemble à "/Users/<nom>" ou "/Utilisateurs/<nom>". C'est le dossier qui contient le bureau, vos documents et ainsi de suite.

La première chose à voir c'est de vérifier si le dossier .ssh(avec un point devant) existe dans votre dossier utilisateur. S'il n'existe pas, vous n'avez pas encore de clef ssh. Il vous faudra donc créer le dossier et passer à l'étape suivante.

Si vous avez bien ce dossier il faut vérifier s'il contient des fichiers. Pour cela il faut lancer le terminal, ou (si vous êtes sous Windows et n'avez pas

installé les outils Linux au moment de l'installation de Git) le Git Bash.

Puis il faut taper :

```
ls -al ~/.ssh
```

Ceci permettre de lister l'ensemble des clefs SSH. Si vous voyez apparaitre id_rsa.pub, ou id_ecdsa.pub, ou id_ed25519.pub, ce sont des fichiers qui peuvent servir à l'authentification avec GitHub, et vous pouvez passer à l'étape qui permet de rajouter les clefs SSH locales sur GitHub. Si le dossier ne contient aucun de ces fichiers, ou si vous souhaitez créer une nouvelle clef… nous allons voir ensemble comment faire

CRÉER UNE NOUVELLE CLEF SSH

Pour créer une nouvelle clef, il faut taper (en remplaçant l'email par celui que vous utilisez sur GitHub) :

```
ssh-keygen -t ed25519 -C "votre@email.com"
```

Si jamais cette commande ne fonctionne pas (parce que votre système ne supporte pas l'algorithme Ed25519), vous pouvez taper :

```
ssh-keygen -t rsa -b 4096 -C "votre@email.com"
```

Si tout va bien, le système vous répond:

```
> Generating public/private algorithm key pair.
```

Ici, il va vous demander de saisir un mot de passe pour votre clef. Je vous conseille de laisser le mot de passe vide, ça vous simplifiera la vie par la suite. (Si vous avez des besoin de sécurité renforcée c'est possible de saisir un mot de passe et de le sécuriser via une clef physique, mais ça dépasse le cadre de ce livret et le SSH sans mot de passe est suffisant pour la plupart des cas).

A présent il faut informer votre OS que vous avez une clef SSH que vous voulez utiliser. Pour cela il faut commencer par lancer l'agent SSH en tapant:

```
eval "$(ssh-agent -s)"
```

(En fonction de votre OS vous aurez peut-être besoin d'ajouter un "sudo" devant la commande)

Attention, si vous êtes sur MacOS, il faut vérifier que votre fichier de configuration contienne ceci (où le "IdentityFile" correspond à la clef que vous avez générée) :

```
Host *

AddKeysToAgent yes

IdentityFile ~/.ssh/id_ed25519
```

A présent nous allons ajouter la clef SSH à l'agent SSH local, en tapant :

```
ssh-add ~/.ssh/id_ed25519
```

Si vous tombez sur un cas particulier, la documentation de GitHub est très fournie à ce sujet !

METTRE LA CLEF SSH SUR GITHUB

A présent nous sommes presque arrivés au but : il nous faut rajouter la clef publique (c'est le sens de l'extension ".pub") à notre profil GitHub. Pour cela, rien de plus simple; il suffit de naviguer vers le dossier .ssh et de copier-coller le contenu de la clef publique (le fichier .pub avec le même nom que la clef privée), et de le renseigner sur Github.

Pour cela, il faut cliquer sur la photo de profil sur GitHub puis sur les réglages (ou "Settings"). Sur la partie accès ("Access", en Anglais) il faut rentrer dans la partie intitulée SSH and GPG Keys, et cliquer sur le bouton qui permet d'ajouter une clef SSH (New SSH Key), donner un titre à la clef et copier le contenu dans le champ prévu à cet effet. Et puis enregistrer le tout.

Et voilà, nous sommes prêts à utiliser GitHub !

4
UTILISATION DE DÉPÔT DISTANT AVEC GITHUB

CRÉER & CONNECTER UN NOUVEAU DÉPÔT

Nous allons commencer par crée un nouveau dépôt (ou repository), que nous allons appeler “TestRespository”. Ici nous n’allons **pas créer de README ou de .gitignore ni de licence** parce que nous voulons connecter notre dépôt local à ce dépôt sur GitHub.

GitHub nous indique alors l’adresse de notre dépôt. Faites bien attention à choisir l’option SSH, puis copiez l’adresse du dépôt.

A présent, rendons nous dans notre dépôt local, et exécutons la commande :

```
git remote add origin <addresse du dépôt>
```

Ici à la place de l’adresse du dépôt il faut coller l’adresse venant de GitHub.

A présent nous allons simplement pousser notre dépôt vers GitHub :

```
git push
```

Ici git nous répond :

```
fatal: The current branch main has no upstream branch.

To push the current branch and set the remote as upstream, use:
```

```
git push --set-upstream origin main
```

Pour traduire : notre branche (main) n'est pas connectée avec une branche de GitHub. Il faut indiquer à git, sur notre ordinateur, vers quelle branche il doit chercher à pousser le code. Comme souvent, il suffit de suivre les indications de git, et ici même copier coller la ligne que git nous indique et l'exécuter :

```
git push --set-upstream origin main
```

Git répond alors quelque chose comme :

```
Enumerating objects: 15, done.

Counting objects: 100% (15/15), done.

Delta compression using up to 8 threads

Compressing objects: 100% (5/5), done.

Writing objects: 100% (15/15), 1.19 KiB | 1.19 MiB/s, done.

Total 15 (delta 1), reused 0 (delta 0), pack-reused 0

remote: Resolving deltas: 100% (1/1), done.

To github.com:Gosev/TestRepository.git

* [new branch] main -> main

Branch 'main' set up to track remote branch 'main' from 'origin'.
```

Git a réussi à envoyer le code de notre dépôt sur GitHub. Si nous allons sur GitHub, et que nous ouvrons le README, nous voyons qu'il est en effet mis à jour avec notre contenu. Pour demander à git quels sont les dépôts distants, nous allons faire :

```
git remote -v
```

Git nous répond :

```
origingit@github.com:Gosev/TestRepository.git (fetch)

origingit@github.com:Gosev/TestRepository.git (push)
```

Où “Gosev” est remplacé par votre nom d’utilisateur.

Imaginons à présent qu’un deuxième développeur travaille sur le même code (ça demande visiblement beaucoup de ressources pour éditer un fichier texte…).

CLONER UN DÉPÔT EXISTANT

Pour cela nous allons clone le dépôt (présent sur GitHub) vers un dépôt local. Pour cela, nous allons sortir du dossier courant du dépôt :

```
cd ..
```

(Ou juste ouvrir un nouveau terminal dans le dossier utilisateur).

Puis nous allons exécuter :

```
git clone <addresse du dépôt> depot2
```

Par exemple dans mon cas :

```
git clone git@github.com:Gosev/TestRepository.git depot2
```

Git nous informe :

```
Cloning into 'depot2'...

remote: Enumerating objects: 15, done.

remote: Counting objects: 100% (15/15), done.

remote: Compressing objects: 100% (4/4), done.

Receiving objects: 100% (15/15), done.
```

```
Resolving deltas: 100% (1/1), done.

remote: Total 15 (delta 1), reused 15 (delta 1), pack-reused 0
```

Si nous ouvrons le dossier “depot2”, nous voyons qu’il contient le même fichier README. Editons le pour y ajouter une ligne :

```
# Ceci est un readme

feature1

feature2

feature3
```

Puis nous allons ajouter le README à l’index, faire un commit puis lancer :

```
git push
```

Git nous répond:

```
To github.com:Gosev/TestRepository.git

4743f56..239602a main -> main
```

Si je regarde le contenu sur GitHub je vois que le “feature3” y a bien été rajouté. A présent retournons dans le premier dossier avec notre dépôt d’origine. Le fichier README ne contient pas la modification. Mais si nous faisons :

```
git pull
```

Git récupère la modification qui avait été faite dans le dossier depot2. Ainsi nous voyons comment GitHub nous permet de partager du code entre différents développeurs. A présent, voyons ce qui se passe … quand ça se passe mal.

GESTION DES CONFLITS

Nous allons éditer le contenu du README du premier dépôt pour y faire apparaitre :

```
# Ceci est un readme

feature1

feature2

feature3


featureA
```

Puis nous faisons :

```
git commit -a -m "add featureA"

git push
```

La première ligne permet de faire le “git add” de tous les fichiers suivis, et d’enregistrer ces modifications d’un seul coup. Généralement ce n’est pas une bonne idée puisqu’on risque d’embarquer des modifications qui n’ont rien à voir avec le travail en cours, mais ici la portée des modifications est réduite.

La deuxième ligne envoie le code sur le serveur. Si nous vérifions l’état de GitHub nous voyons que la branche main y est à jour avec notre code. Si nous faisons git status, l’outil nous répond que tout est à jour:

```
On branch main

Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

A présent rendons nous dans depot2. Et nous allons commencer par créer une modification sur README.md :

```
# Ceci est un readme

feature1
```

```
feature2
feature3


featureB
```

Puis nous faisons :

```
git commit -a -m "add featureB"
```

Commençons à présent par faire un git status. Git nous répond :

```
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)
nothing to commit, working tree clean
```

Ici, nous allons donc faire “git push”, comme git nous le recommande. Et là… c’est le drame. Git nous répond :

```
! [rejected] main -> main (fetch first)
error: failed to push some refs to 'github.com:Gosev/TestRepository.git'
```

Mais si nous demandons à git son état en relançant un git status, le message n’a pas changé :

```
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
(use "git push" to publish your local commits)
nothing to commit, working tree clean
```

Ici nous voyons que git n’est par défaut pas conscient de l’état du serveur distant (du “remote”). Il nous faut donc d’abord récupérer l’état du serveur

distant, qui a été mis à jour par un tiers (enfin en l'occurence par nous-mêmes via un autre dossier). Pour ce faire, il faut lancer la commande :

```
git fetch
```

Git nous répond alors :

```
remote: Enumerating objects: 5, done. remote:
Counting objects: 100% (5/5), done. remote:
Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 266 bytes | 88.00 KiB/s, done.
From github.com:Gosev/TestRepository
  239602a..7197c70 main -> origin/main
```

Et à présent lançons de nouveau git status. Cette fois-ci le statut correspond mieux à ce que nous savons être vrai :

```
On branch main
Your branch and 'origin/main' have diverged,
and have 1 and 1 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)
nothing to commit, working tree clean
```

Mais si nous ouvrons le README, il n'a pas changé. Pourquoi ? En réalité, tout se passe comme si nous avions récupéré localement une branche "origin/main", sur laquelle nous ne pouvons pas nous rendre directement, et qui contient les modifications qui nous intéressent. (Toutefois si on fait un

“git branch” on voit bien que cette branche n’existe pas localement). Comment récupérer ces modifications ?

En réalisant un merge de la branche origin/main vers notre branche main.

Et il y a une commande git qui réalise à la fois le “git fetch” et le “git merge” d’une branche distante. Nous allons l’exécuter maintenant :

```
git pull
```

Ici, si tout se passe normalement, git nous répond avec un long laïus sur la méthodologie à utiliser pour combiner les branches, puis nous indique :

```
Auto-merging  README.md

CONFLICT (content): Merge conflict in README.md

Automatic merge failed; fix conflicts and then commit the result.
```

Commençons par lui indiquer nos préférences pour le git pull (la différence entre “rebase” et “merge” sort du champ de ce voire même de ce livret):

```
git config pull.rebase false
```

Et à présent ouvrons le fichier README. Nous voyons qu’il y a effectivement un conflit :

```
# Ceci est un readme

feature1

feature2

feature3

<<<<<<< HEAD

featureB

=======

featureA
```

```
>>>>>>> 7197c70c46c40d013161119ef32c52956eaad766
```

Comme auparavent, le block intitulé “HEAD” représente les modifications de la branche en cours, et l’autre block représente les modifications entrantes. Ici pas d’indication de branche, mais le hash SHA d’un enregistrement (d’un commit).

Il nous faut éditer le fichier pour garder produite le résultat qui nous semble être le bon, par exemple :

```
# Ceci est un readme

feature1

feature2

feature3

featureA

featureB
```

A présent nous allons ajouter ce fichier à l’index pour résoudre le conflit.

```
git add ./README.md
```

Si nous vérifions le statut du dépôt, git nous indique (notamment) “All conflicts fixed but you are still merging. (use "git commit" to conclude merge)”. Et nous allons donc suivre ses recommandations en faisant :

```
git commit
```

Il faut ici sauvegarder le message de commit (en tapant “:x” ou “:x!” pour forcer la sortie si vous avez une interface de type vi / vim). Un git status nous informe que nous avons deux modifications d’avance sur le dépôt distant. Nous allons donc les envoyer sur le serveur, comme nous avions tenté de le faire précédemment (mais nous n’avions pas encore fait “git pull” alors ça avait échoué) :

```
git push
```

A présent l’opération se passe bien. Si nous retournons dans le dossier d’origine, et que nous ouvrons le README.md, il n’a pas les modifications

que nous avions faites. Nous allons donc les récupérer avec :

```
git pull
```

Et ce faisant, nous avons récupéré les modifications dans ce dossier. Et surtout nous avons vu comment récupérer le statut d’un dépôt distant avec “git fetch”, comment gérer un conflit dans une récupération, et comment plusieurs développeurs peuvent travailler de concert sur un fichier commun.