CHAPITRE 1

1

POURQUOI GIT ?

Il existe un outil de référence utilisé par Google, par Facebook, par Microsoft, par Amazon, par Netflix... et en réalité par la majorité des développeurs. C'est un outil incontournable si vous voulez devenir développeur — ou juste utiliser du code. Cet outil s'appelle “git”. Cet outil est plébiscité par 85% des développeurs. [1]

Nous allons passer ensemble en revue les commandes les plus courantes et voir ensemble comment les utiliser, et à quoi elles servent. De l’extérieur git peut sembler être une science obscure. Mais en réalité une fois que le fonctionnement de git est compris, il y a peu de choses à savoir. Ce qui explique pourquoi ce livret est court. Ensemble nous allons apprendre à manipuler Git, avec des exercices concrets et pratiques. Linus Torvarlds explique :

En général, la meilleure façon d'apprendre git est probablement de commencer par ne faire que des choses très basiques et de ne même pas regarder certaines des choses que vous pouvez faire jusqu'à ce que vous soyez familier et confiant avec les bases. [2]

C’est très exactement la méthode que nous allons suivre ensemble, et qui vous fournira les bases qui permettront d’avancer plus loin et d’utiliser git sans crainte. Et ces exercices, nous allons les effectuer pour la plupart depuis la ligne de commande. Pourquoi ?

Git est un outil (des outils, même), en ligne de commande. Et la ligne de commande est son environnement naturel, ce qui permet de voir le plus clairement son fonctionnement.

Si vous maitrisez git en ligne de commande, vous manipulerez facilement l’ensemble des sur-couches graphiques qui servent à piloter la ligne de commande (mais fournissant malheureusement moins de détails lorsqu’il y a des accrocs). Mais d'abord une question fondamentale avant de parler des commandes : à quoi sert Git ?

2
A QUOI SERT GIT ?

La philosophie des systèmes Linux, c'est de limiter la responsabilité des outils. Un outil ne doit faire qu'une seule chose. Git s'inscrit dans cette philosophie; sa seule responsabilité est le SCM ou "source contrôle management", la gestion de code source.

A quoi sert la gestion de code source ? Elle sert à suivre tout l'historique des modifications dans le code source : Qu'est-ce qui a été changée par qui, à quel endroit ?

Au cours d'un projet donné, des versions différentes du code se retrouvent à plusieurs endroits. Prenons l'exemple d'un site web sur lequel travaillent des développeurs. Quelles sont les différentes versions du code qui coexistent ?

Tout d'abord, il y a du code en *production* , c'est-à-dire : déployé sur le site disponible publiquement. Vis-à-vis de ce code, il faut pouvoir retourner en arrière si une fonctionnalité critique est cassée par une mise en ligne. Il faut pouvoir revenir à une version stable.

On peut aussi imaginer un environnement (c'est-à-dire: un ou des servers) où est déployé du code en train d'être testé en vue d'une prochaine mise en ligne.

À d'autres endroits — sur les ordinateurs des développeurs par exemple — se trouve du code sur lequel on apporte de nouvelles fonctionnalités. Et les développeurs doivent pouvoir travailler sans se marcher sur les pieds. Il y a même des cas où je veux pouvoir sur mon propre ordinateur pouvoir basculer d'une version du code à une autre.

Git permet de répondre à toutes ces problématiques. Avec Git, nous pouvons parcourir l'historique des modifications du code, et isoler le code de chacun dans des espaces de travail indépendants, au sein d'un même dossier. Grâce à ce qui dans Git s'appellent les branches.

Git permet de comprendre *qui* a fait quelles modifications, avec un petit message de la personne qui explique : "j'ai fait telle modification, pour telle raison". Git permet surtout de communiquer du code, de pouvoir transvaser du code d'un environnement de travail à un autre. Comment ? C'est justement l'objet de ce livret.

3
UN SYSTÈME DISTRIBUÉ, ET UN PEU D'HISTOIRE

Git n'est pas (et de loin) le premier outil de gestion de code source. Avant lui les développeurs utilisaient par exemple SourceSafe de Microsoft ou encore SVN de Subversion. Mais Git s'est imposé comme *la* référence. Pourquoi ?

Pour cela il nous faut faire un petit détour historique.

En l'an 2000 (au millénaire dernier, en somme), l'équipe de Linux se trouve confronté à des difficultés de gestion de code. Tous les gestionnaires de code source étaient calqués sur le modèle de l'entreprise. C'est-à-dire, avec une organisation centralisée et structurée, avec des utilisateurs répartis sur quelques lieux, sur des réseaux communs.

Mais le modèle de développement Open Source est tout autre : décentralisé, avec des développeurs sur toute la planète, qui travaillent de manière indépendante. Les systèmes de gestion de code source existants n'étaient pas adaptés.

Intervient alors une entreprise : BitMover. Elle propose aux développeurs Linux d'utiliser gracieusement son logiciel, BitKeeper. Ce logiciel avait pour avantage d'être distribué. Qu'est-ce que cela signifie ?

Un système centralisé réside dans une base de donnée principale qui fait autorité quand à ce qui appartient ou non dans le code. Dans un système distribué, à l'inverse, ce serveur central n'existe pas. Au contraire, chaque poste de développement comporte sa propre base de donnée du code, qui fait foi localement. Et ces bases de données communiquent du code entre eux, par exemple par le biais de serveurs, à la guise des développeurs. Cette

base de données locales est ce qu'on appelle un "repository" ou dépôt. C'est justement ce que propose Git.

Apprendre Git, c'est apprendre comment manipuler ces dépôts. Comment les créer, comment les copier, comment y enregistrer ses modifications en code, comment communiquer ces modifications entre serveurs. Ainsi nous allons voir ensemble :

- comment installer Git,
- comment créer un dépôt avec git init
- comment voir l'état de son dépôt avec git status
- comment sauvegarder son travail avec git add & git commit,
- comment créer une nouvelle branche avec git branch,
- comment basculer et du code d'une branche avec une autre avec git merge
- et finalement comment travailler sur un dépôt distant avec git clone, git remote, git fetch et git pull et git push.

Mais tout d'abord finissons notre histoire.

BitKeeper avait été mis gratuitement à disposition des développeurs de Linux (et d'autres logiciels de logiciel libre) à condition que ceux-ci ne contribuent pas à des logiciels concurrents.

Une des fonctionnalités les plus utiles d'un logiciel de gestion de code source, c'est celle qui permet de suivre les différentiels entre les différentes versions historiques du code. Or cette fonctionnalité était indisponible dans la version gratuite de BitKeeper. Un développeur a donc crée un logiciel permettant de lire ces données depuis la base de données locales. BitMoover a réagi en annonçant en avril 2005 suspendre les licences gratuites pour l'équipe de Linux.

L'équipe de Linux étant obligée de trouver une autre solution, l'instigateur de Linux, Linus Torvalds, en a développée une, en l'espace d'une dizaine de jours. Et c'est ainsi qu'est né Git, le gestionnaire de code source de référence.

4
INSTALLATION DE GIT

Voyons à présent comment installer Git. Avant toute chose, il faut vérifier s'il n'est pas déjà installé. Pour cela, il faut ouvrir un "terminal", c'est à dire l'interface qui permet de rentrer des lignes de commande (en Anglais, "Command Line Interface" ou CLI).

Sur Windows, tapez Windows + R, puis tapez "cmd" .

Sur Mac ou sous Linux, allez dans les applications et lancez celle qui s'appelle "Terminal".

(Si vous n'y arrivez pas, vous pouvez aussi utiliser le terminal qui se trouve dans un IDE tel que Visual Studio Code).

Une fois que le terminal est ouvert, nous allons vérifier si git est déjà installé ou non. Pour cela, il faut ouvrir un terminal et taper:

```
git --version
```

Si le terminal répond avec des chiffres (par exemple "git version 2.32.1"), bravo ! Git est déjà installé et vous pouvez passer à la suite. Si par contre le terminal vous indique qu'il ne connait pas git, nous allons l'installer ensemble.

Pour ce faire, nous allons nous rendre sur le site officiel de git, à savoir : https:// git- scm. com/ download/ Ici, en fonction de votre type d'ordinateur (et de système opératoire) suivez les instructions fournies par le site web. Je vous en fournis une synthèse ci-dessous.

SUR PC / WINDOWS

Le site de git propose de télécharger un installeur. (Cf. https:// git- scm. com/ download/win) Le plus souvent, il s'agira de la version 64bits. Téléchargez-le puis suivez les instructions. Si vous pouvez, je vous conseille de cocher la case qui propose d'installer les utilitaires Unix/Linux/Cygwin. Ils ne sont pas absolument nécessaires mais ça vous simplifiera bien des choses par la suite. Ils permettent notamment d'utiliser des utilitaires en ligne de commande qui vont nous être bien utiles.

SUR MAC / OSX

Le plus simple sur Mac / OSX est d'installer "homebrew" (via https:// brew.sh/) puis de lancer la commande:

```
brew install git
```

SUR LINUX

La façon d'installer dépendra de votre distribution de Linux. Par exemple sur Ubuntu ou Debian il faudra lancer :

```
sudo apt update
```

Puis

```
sudo apt install git
```

Sous d'autres distribution le mode opératoire sera différent, mais une petite recherche sur votre moteur de recherche préféré vous fournira la réponse facilement.

VERIFICATION DE L'INSTALLATION

Une fois que l'installation est faite, ré-ouvrez une fenêtre de terminal et réessayez la commande:

```
git --version
```

Il faut bien ouvrir un nouveau terminal pour que celui-ci puisse aller vérifier au lancement ce qui est installé. L'ancienne fenêtre ne percevra pas l'existence du nouveau venu.

CONFIGURATION DE L'UTILISATEUR SUR GIT

Maintenant que git est installé, il faut lui indiquer qui vous êtes. Ou plus exactement, quel nom il doit afficher lorsque vous écrivez des modifications dans la base de donnée. (Git ne va pas vérifier quoi que ce soit donc il peut s'agir de pseudonymes). Il faut simplement indiquer à Git le nom et l'email que vous souhaitez voir apparaître à côté de vos enregistrements. Pour cela nous allons utiliser la commande *git config* . Pour préciser le nom pour tout l'environnement de travail (quel que soit le dossier sur l'ordinateur), il faut taper (en l'adaptant à votre cas):

```
git config --global user.name "Votre Nom"
```

Pour l'email, de la même façon, il faut taper (en l'adaptant à votre cas) :

```
git config --global user.email "votre@email.com"
```

Voilà, vous êtes parés pour commencer à utiliser la commande git. Si vous le voulez bien, nous allons nous douter d'un outil supplémentaire pour simplifier les explications.

INSTALLATION DE VISUAL SOURCE CODE

Comme nous allons éditer du texte et manipuler la console simultanément, je conseille d'utiliser un outil qui permette de faire les deux simultanément. Pour cela, je conseille Visual Studio Code, un éditeur de code développé par Microsoft. Pour cela il suffit de se rendre à https:// code. visualstudio.com/ et d'y suivre les instructions de téléchargement et d'installation.

Une fois que vous avez lancé Visual Studio Code, vous pouvez ouvrir un terminal via le menu (Terminal / Nouveau Terminal).

5
CRÉER UN DÉPÔT (OU “REPOSITORY”) ET VOIR SON ÉTAT

Maintenant nous allons voir concrètement comment créer un dépôt, et commencer à travailler avec git. Commençons par créer un dossier du nom de “TestRepository” dans votre dossier utilisateur (celui qui a votre nom d’utilisateur sur votre ordinateur). Nous allons utiliser ce dossier pour apprendre ensemble. Il nous faut à présent ouvrir une interface en ligne de commande dans ce dossier.

Pour cela nous avons plusieurs options. Le plus simple, comme indiqué ci-dessus, est de lancer Visual Studio Code, d’ouvrir le dossier ainsi crée, puis de lancer un terminal dans Visual Studio Code.

NB: Si vous n’avez pas souhaité installer VS Code, vous pouvez vous rendre dans le dossier en ouvrant le terminal (qui s’ouvre par défaut dans votre dossier utilisateur) et en tapant :

```
cd TestReposiory
```

A présent commençons par vérifier l’état de ce dossier, du point de vue de git, en tapant:

```
git status
```

Git répondra alors que le dossier n’est pas géré par git:

```
fatal: not a git repository
```

Nous allons donc commencer par initialiser Git dans ce dossier. Pour cela, il suffit de taper notre première commande git:

```
git init
```

Si tout va bien, le terminal répond une ligne qui ressemble à ceci :

```
Initialized empty Git repository in [...]/TestRepository/.git/
```

Nous avons donc initialisé un “repository” (ou dépôt) vide dans ce dossier. A présent faisons la liste des fichiers qui sont présents dans ce dossier, en tapant ls -a (ou dir /a si vous êtes sous Windows et que vous n’avez pas installé les outils Linux). Le “a” que nous avons apposé signifie “en faisant abstraction des attributs”, et donc y compris les fichiers cachés. En faisant cela, nous voyons qu’a été créé un dossier appelé “.git”. (Le point devant en fait un dossier caché).

Ce dossier .git est l'endroit où on se gardent toutes les modifications du code — toute la base de données de git. La base des modifications est centralisée dans ce dossier, ce qui fait que si on veut enlever la gestion git d’un ensemble de fichiers de code il suffira de supprimer ce dossier à la racine du projet.

A présent dans ce dossier il n’y aura plus besoin de taper la commande “git init”. La configuration initiale a été réalisée et n’a pas besoin d’être répétée.

Voyons à présent l’état de notre dépôt. Pour cela nous allons utiliser une commande bien pratique lorsque nous travaillons dans un dépôt git. Cette commande permet de voir l'état du repos : y’a-t-il des fichiers en cours de d'édition ? Y-a-t’il des fichiers qui ont besoin d’être enregistrés dans git ?

Cette commande, c’est “git status”. Si nous la saisissons dans le terminal:

```
git status
```

Git nous répond trois choses :

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Regardons de plus près de quoi il en retourne. La première phrase nous indique sur quelle “branche” nous nous trouvons. (Si vous avez une version plus ancienne de git, il se peut que votre branche s’appelle “master” au lieu de “main”). Nous verrons sous peu ce qu’est une branche et à quoi elle peut bien servir.

La deuxième phrase nous indique qu’à date le dépôt ne comporte aucun “commit”, c’est-à-dire qu’aucun fichier n’y a encore été consigné. Et finalement git indique qu’il ne se trouve dans ce dossier aucun fichier qui pourrait être enregistré dans le dépôt.

6
UN PREMIER COMMIT, ET L'INDEX

Créons donc un fichier "README.md" dans ce dossier (par exemple par le biais de Visual Source Code). Dans ce fichier allons écrire "# Ceci est un readbe" (en utilisant un dièse qui dans la syntaxe markdown indique un titre), puis sauvegarder. A présent regardons ce que dit git, en tapant:

```
git status
```

Git répond

```
On branch main [...] No commits yet [...]

Untracked files: (use "git add <file>..." to include in what will be committed)

README.md
```

Git nous donne beaucoup d'informations utiles dans le statut. Tout d'abord, comme d'habitude, il nous indique sur quelle "branche" nous nous trouvons : la branche "main". Et nous n'avons toujours pas enregistré de commit. Par contre git nous indique qu'un fichier qui est pas suivi ("untracked files" signifie fichiers non suivis). Sans grande surprise, il s'agit de README.md, le fichier que nous venons de créer. Git nous indique même comment agir pour remédier à la situation: utiliser "git and". C'est donc ce que nous allons faire:

```
git add README.md
```

A présent vérifions de nouveau le statut :

```
git status
```

Git nous répond à présent :

```
Changes to be committed: [...] new file: README.md
```

Notre fichier a changé d'état. Il était indiqué comme étant "untracked", c'est à dire non suivi. Et à présent git nous dit que ce fichier fait partie des changements à "committer", c'est-à-dire, dans l'index. Et nous voyons là les trois états possibles d'un fichier (ou plus largement d'un changement) : non suivis, ajoutés à l'index ou enregistrés (ou committés).

FICHIERS NON SUIVIS (UNTRACKED)

Un fichier peut être inconnu de git, il est donc alors "**untracked** ". C'est ce qui se passe quand un développement est en cours. On est en train de taper du code. Et ce code n'est pour le moment pas traité par git. Git sait simplement qu'il existe des motifs sur les fichiers qu'il référence.

L'INDEX

Une fois que les modifications sont prêtes à être sauvegardées, vient le moment de préparer un commit. Pour ce faire, le développeur ajoute les différentes modifications d'une même tâche fichier par fichier (voire même modification par modification) dans l'index, par le biais de la commande "git add".

C'est quoi l'index ? L'index regroupe l'ensemble des modifications qu'on se prépare à sauvegarder. C'est en quelque sorte le brouillon de l'enregistrement (ou "commit" à venir). Une fois les modifications sélectionnées dans l'index, on le sauvegarde, et on crée donc un "commit".

LE (PREMIER) COMMIT

Le commit est un enregistrement dans la base de données de modifications, une étape dans l'état du code. Un commit se doit d'être dans la mesure du possible un ensemble cohérent. C'est notre cas, nous allons donc nous apprêter à l'enregistrer.

Ici, si jamais vous avez omis l'étape de configuration au moment de l'installation de git, l'outil va se plaindre qu'il ne peut pas aller plus loin, donc assurez vous de bien avoir préciser le nom et l'email de l'auteur des commits.

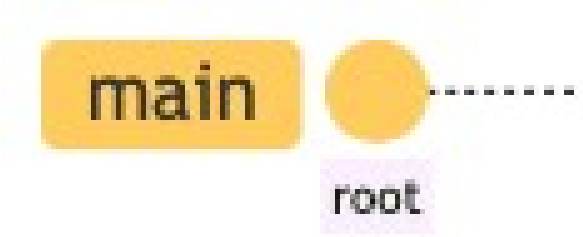
Pour sauvegarder les modifications contenues dans l'index dans un commit, c'est très simple en ligne de commande. Il suffit de lancer la commande:

```
git commit -m "Première version de Readme"
```

Comme vous avez pu voir, on a indiqué un message qui décrit les modifications contenues dans le commit. Ce petit message permet d'indiquer, lorsqu'on parcours l'historique des enregistrements, quelle était l'intention de cette modification : L'a-t-on a faite pour corriger un bug ? Ou pour implémenter telle fonctionnalité ? Ici le message indique juste qu'on crée le fichier Readme. Lorsqu'on lance le commit, git répond:

```
[main (root-commit) 3d36698] Première version de Readme

1 file changed, 1 insertion(+)

create mode 100644 README.md
```

Le message nous indique que nous sommes sur la branche main, et qu'on enregistre le premier commit (l'enregistrement racine ou "root commit"). "Root" signifie racine en anglais. Et ce premier commit va être la racine, le point de départ de l'ensemble de l'activité du dépôt, ce qu'on appelle *l'arbre* git.



Et puis git nous indique un code. Ici dans mon exemple, il s'agit de "*3d36698* ". Ce code est le début d'un "hash" avec l'algotythme SHA. C'est ce code qui identifie de manière unique le commit dans l'historique de git. C'est en quelque sorte l'identifiant du commit. Git nous indique qu'un fichier a été changé. Il y a une insertions puisque qu'on a écrit une ligne, et il a créé le fichier README.md.

Histoire de ne pas changer une équipe qui gagne, lançons la commande "git status". Git répond:

```
On branch main

nothing to commit, working tree clean
```

Autrement dit, nous sommes sur la branche main, et il n'y a pas de modification en attente.

ET UN DEUXIÈME COMMIT

Par contre je vois que j'ai commis une petite erreur. J'ai écrit "Ceci est un readbe", alors qu'*évidemment* je voulais écrire "Ceci est un readme." Comme c'est dommage ! Nous allons donc corriger cette faute (lourde de conséquences). Pour cela nous allons éditer le fichier README.md et corriger le texte. Si a présent nous faisons un "git status", git nous répond:

```
Changes not staged for commit:

  (use "git add <file>..." to update what will be
committed)
```

```
  (use "git restore <file>..." to discard changes
in working directory)

    modified:  README.md
```

Git nous informe qu'il perçoit une modification à l'intérieur du fichier. Voyons quelles sont ces différences, a l'aide de la commande git diff auquel on fournit le nom du fichier qui nous intéresse:

```
git diff README.md
```

Ici nous voyons la composition naturelle d'un commit ou d'un changement:

```
diff --git a/README.md b/README.md

index ecacfd6..c6bb546 100644

--- a/README.md

+++ b/README.md

@@ -1 +1 @@

-Ceci est un readbe

+Ceci est un readme
```

Le - indique une ligne qui va être supprimée, le « + » une ligne qui a été rajoutée : on enlève la ligne le "Ceci est un readbe" on rajoute la ligne "Ceci est un readme." On voit ici très précisément l'information qui va être enregistrée dans le commit : tel fichier a été impacté, telle ligne à telle position a été supprimée, telle ligne à telle position a été supprimée.

Si on effectue de nouveau le commit, on commence par ajouter le fichier modifié à l'index, puis on crée le commit avec le message indicatif :

```
git add ./README.md

git commit -m "Fix spelling mistake"
```

Git nous indique :

```
[main 8e16db9] Fix spelling mistake
```

```
1 file changed, 1 insertion(+), 1 deletion(-)
```

Entre les crochets carrés nous avons la branche en cours, et en deuxième nous avons le début du hashage SHA du commit (8e16db9…), puis vient le message et la portée des changements. A présent l'arbre du dépôt git comporte un noeud racine et un premier commit qui lui est rattaché:

Tout cela ne ressemble pas encore à un arbre, mais c'est l'occasion de voir ensemble ce que sont les branches.

7

LES BRANCHES

Pour commencer allons dans le terminal, faisons un “git status”. Nous voyons que nous sommes sur la branche main, la branche principale. Dans cette branche master nous avons README.md.

A présent imaginons que nous voulons développer une nouvelle fonctionnalité. On va appeler cette fonctionnalité "Fonctionalité 1", ou en anglais « feature1 ». Pour développer cette fonctionnalité, nous allons créer une version de notre code qui va servir à développer cette fonctionnalité, sans interférence venant du code d’autres développeurs. Cette version du code, cette copie protégée, c’est la branche. Pour la créer il existe plusieurs façons de procéder.

GIT BRANCH

Commençons par faire :

```
git branch feature1
```

Puis faisons un “git status”. Qu’est-ce qui a changé ?

```
On branch main

nothing to commit, working tree clean
```

Rien du tout. Parce que nous avons créé la branche mais nous n’avons pas basculé dessus. Comment changer de branche ? Avec la commande “git

checkout” :

```
git checkout feature1
```

Git répond :

```
Switched to branch 'feature1'
```

Si à présent nous faisons un “git status”, nous voyons que nous sommes à présent sur la branche “feature1”, mais qu’il n'y a rien à enregistrer puisque nous n’avons encore fait aucune modification.

```
On branch feature1

nothing to commit, working tree clean
```

Maintenant pour illustrer tout ceci, nous allons rajouter une ligne dans le fichier README. Ce n’est pas notre propos ici de coder, donc nous allons simplement rajouter une ligne, disant : “feature1”. Notre README ressemble à présent à ceci:

```
# Ceci est un readme

feature1
```

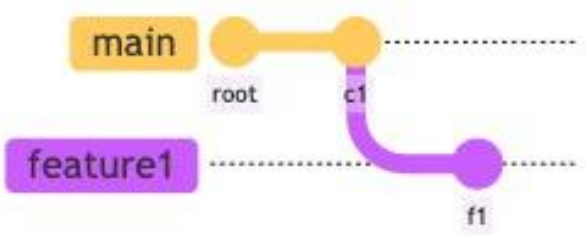
Si on fait un “git status” nous voyons sans surprise que nous sommes sur la branche “feature1” et qu’il y a des modifications à enregistrer. Pour les enregistrer vous savez à présent qu’il faut ajouter à l’index puis faire un commit avec un message:

```
git add ./README.md

git commit -m "Add feature1"
```

Git nous dit :

```
[feature1 9603786] Add feature1

1 file changed, 1 insertion(+)
```

Nous somme sur la branche “feature 1” et nous avons réalisé un premier commit. Victoire !



A présent changeons de branche, et retournons sur la branche main:

```
git checkout main
```

Si vous regardez le schéma ci-dessus, nous sommes passés de feature1 à main, donc du noeud “f1” au noeud “c1”. Si à présent nous ouvrons le fichier README nous voyons qu’il ne contient pas la modification feature 1 :

```
# Ceci est un readme
```

Retournons sur la branche feature 1 :

```
git checkout feature1
```

La modification a réapparu, le fichier contient à présent de nouveau :

```
# Ceci est un readme

feature1
```

Attention, nous sommes ici au coeur de la raison d’être de git. L’intérêt fondamental de travailler avec git, c’est précisemment cette capacité à changer le code de contexte. De pouvoir avoir du code en cours, et avec une commande, se retrouver dans un autre contexte où ce code en cours est sauvegardé de manière sécurisée mais n’est plus présent dans le code actif. Comme si on changeait de sauvegarde en cours. Et en réalité c’est précisément ce que nous faisons, puisque nous passons d’un commit (le dernier de la première branche) à un autre (le dernier de la deuxième branche).

Nous allons à présent créer une deuxième branche pour développer une deuxième fonctionnalité. Si vous vous souvenez, on avait crée une branche avec “git branch”, puis on avait changé de branche avec “git checkout”.

Mais il existe une commande qui regroupe les deux actions en une seule. Commençons par **retourner sur la branche master** , avant d'aller plus loin:

```
git checkout master.
```

A présent voyons la commande…

GIT CHECKOUT -B

Git est capable de créer une branche et de s'y rendre directement, à partir du code en cours. Commençons par vérifier que nous sommes bien sur la branche main avec un git status. Git devrait répondre:

```
On branch main

nothing to commit, working tree clean
```

A présent que nous sommes dans de bonnes conditions, voyons la commande. C'est très simple. Il suffit de lancer :

```
git checkout -b feature2
```

Si nous faisons un "git status", nous voyons que que nous sommes sur la nouvelle branche feature2. Il nous est également possible de voir quelles branches existent dans le dépôt, avec la commande:

```
git branch
```

La branche active est alors indiquée avec une astérisque :

```
  feature1
* feature2
  main
```

A présent, nous allons faire un commit sur cette branche. Si nous ouvrons le fichier README.md, il contient pour le moment le même contenu que la version sur main:

```
# Ceci est un readme
```

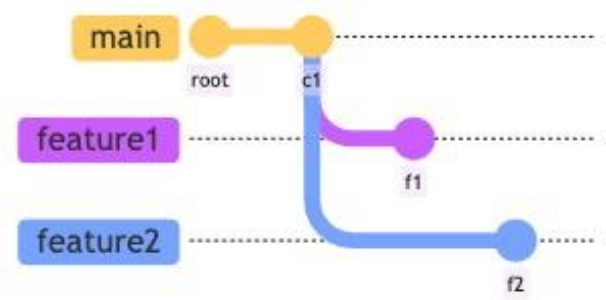
Nous allons de nouveau rajouter un bout de texte qui représente une fonctionnalité:

```
# Ceci est un readme

feature2
```

Les plus observants d'entre vous auront noté que sur la branche feature1, nous avions mis un texte au même endroit. Comme d'habitude nous l'ajoutons à l'index (git add ./README.md) et nous créons le premier enregistrement de la branche feature2 :

```
git commit -m "create feature 2"
```

Notre arbre git ressemble à quelque chose comme ceci :



Maintenant que nous avons vu comment :

- créer une branche avec git branch <nom_de_branche>
- changer de branche avec git checkout <nom_de_branche>
- créer et changer de branche avec git checkout -b <nom_de_branche>
- lister les branches avec git branch

Nous allons à présent simuler le travail de plusieurs développeurs, pour nous permettre de voir comment combiner ces branches pour pouvoir le publier.

8
COMBINER LES BRANCHES AVEC GIT MERGE

Commençons par nous rendre sur la branche main :

```
git checkout main
```

Si nous lisons le contenu de README.md il ne comporte toujours qu'une seule ligne :

```
# Ceci est un readme
```

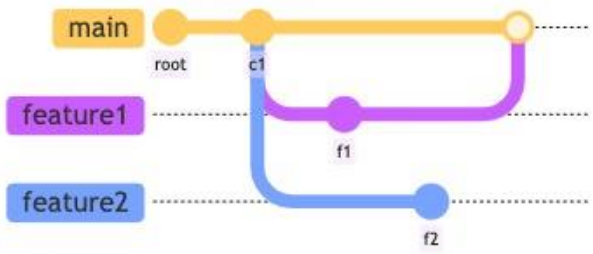
Nous allons à présent récupérer le contenu de la branche "feature1" pour l'inclure sur la branche "main". Pour cela, nous allons combiner (ou "merger") la branche feature1 vers la branche main. La commande git pour faire cela s'appelle "merge". On lui indique la branche que l'on souhaite combiner sur la branche actuelle:

```
git merge feature1
```

Git nous indique ici:

```
Updating 8e16db9..9603786
Fast-forward
README.md | 1 +
1 file changed, 1 insertion(+)
```

Ici, pas de difficulté particulière, le commit a été répercuté sur la branche main:



Si à présent nous ouvrons le README, nous voyons que son contenu correspond à celui de feature1.

A présent nous allons voir concrètement ce qui se produit couramment lorsque deux développeurs travaillent sur un même fichier. Et c'est ici qu'il est important de bien comprendre comment git fait le suivi des modifications. En effet, git ne fait pas d'analyse syntaxique. Il se limite à dire : telle modification se base sur tel commit précédent, il modifie tels fichiers en supprimant telles lignes et/ou en ajoutant telles lignes.

Et ici, dans notre cas, nous avons modifié la même ligne d'un même fichier (la deuxième ligne du fichier README.md, pour être précis). Nous pourrions juste faire "git merge feature2". Mais ici nous sommes en train de simuler le travail de développeurs sérieux et un développeur sérieux ne va pas simplement mettre son travail sur la branche principale (la branche "main" dans notre cas.)

Non, un développeur sérieux (et nous sommes des développeurs sérieux, n'est-ce pas ?) va commencer par combiner ce qui a été fait sur la branche principale vers sa branche de travail.

Nous allons donc faire :

```
git checkout feature2
```

```
git merge main
```

Et comme nous avons changé le fichier README.md à la fois sur main et sur feature2, git rencontre un conflit qu'il ne sait pas résoudre:

```
Auto-merging README.md
```

```
CONFLICT (content): Merge conflict in README.md
```

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

Que faire ? Et bien une fois de plus git nous indique directement la marche à suivre : “fix conflicts and then commit the result.” Résolvez les conflits puis enregistrez le résultat. Git status nous indique peu ou prou la même chose:

```
Unmerged paths:

  both modified:  README.md
```

Si nous ouvrons le fichier README.md nous voyons le texte suivant :

```
# Ceci est un readme

<<<<<<< HEAD

feature2

=======

feature1

>>>>>>> main
```

Ici git nous indique le conflit. Tout d’abord, comme nous pouvons voir, git utilise les lignes de <<<<<, de ===== et de >>>>>> pour indiquer la structure du conflit.

Plus exactement, nous avons <<<<<< HEAD. Head signifie ,pour Git, “là où nous en sommes actuellement dans le dépôt”, l’état actuel du code au moment de lancer la commande.

Et donc dans le code actuel, sur feature2, nous avions changé la deuxième ligne pour y mettre “feature2”. Puis nous avons une ligne de ======, et git nous indique l’alternative, le feature1 qui vient de la branche main.

Comment résoudre le conflit ? Et bien il nous faut éditer le fichier, et mettre la bonne version du code. Mais comment savoir quelle est la bonne version du code ?

En réalité la résolution des conflits peut avoir plusieurs issues, en fonction du code. On peut :

- Ne garder qu'une seule des alternatives
- Garder aucune des alternatives
- Créer une troisième version
- Garder les deux alternatives.

Ainsi on pourrait par exemple éditer le README.md pour qu'il contienne :

```
# Ceci est un readme

feature 1 & 2
```

Ou encore

```
# Ceci est un readme

feature1

feature2
```

L'important étant de supprimer les marqueurs de conflit et de laisser une version fonctionnelle du code. Ici pour un fichier README en markdown (c'est ce que signifie l'extension .md) il y a très peu de contraintes de syntaxe, donc toutes ces options sont valables. Ici, je propose que nous choisissions la deuxième (en éditant, donc, le fichier README pour qu'il corresponde à ce qui est au dessus). Puis que nous allons ajouter et enregistrer la modification :

```
git add README.md

git commit
```

Ici, notez bien que je n'ai pas précisé de message. Je n'ai pas besoin d'en spécifier puisqu'il s'agit d'un "merge". Dans mon cas, git ouvre un éditeur pour m'indiquer le message que lui a choisi :

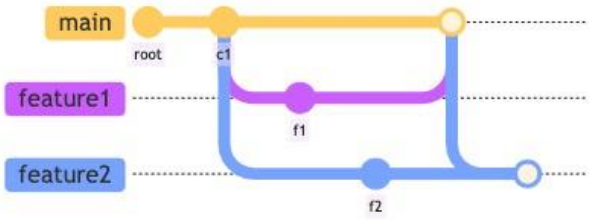
```
Merge branch 'main' into feature2
```

Il me suffit alors de fermer mon éditeur en sauvegardant (dans mon cas il s'agit de nano, et il me faut faire un CTRL+X pour sauvegarder et quitter, mais par défaut git va ouvrir un éditeur de type "vim", il vous faudra alors taper ":x!" pour lui indiquer que vous voulez sauvegarder et quitter).

A présent si je lance git status le conflit a disparu. Si je liste les dernières modifications avec git log, on me dit quelque chose qui ressemble à ceci (j'ai supprimé l'indication des dates et des auteurs pour simplifier la lecture) :

```
commit 4743f56dcbf640e172c7c0f6f58dbc8f83630453 (HEAD -> feature2)

Merge: 1414dd0 9603786

  Merge branch 'main' into feature2


commit 1414dd05b6f025375db312189f75c03a18beaa89

  create feature2


commit 96037867a2c3586eee429902f1d6f266c98765ff (main, feature1)

  Add feature1
```

A présent notre arbre git ressemble à ceci :



Il nous reste encore, pour finir notre travail, à basculer les modifications sur main. Mais avant cela il nous reste à voir comment gérer un cas d’erreur

9

METTRE DE CÔTÉ DES MODIFICATIONS (AVEC GIT STASH)

Imaginons que nous avions rajouté la suite de la feature2 dans le README.md (et éditons le fichier en fonction) :

```
# Ceci est un readme

feature1

feature2

feature2.1
```

A présent, essayons de basculer sur la branche main en faisant “git checkout main”. Git nous répond :

```
error: Your local changes to the following files would be overwritten by checkout:

  README.md

Please commit your changes or stash them before you switch branches.

Aborting
```

Ici git détecte que le fichier README comporte des modifications non sauvegardées qui ne sont pas présentes sur la branche main. Il refuse de ce fait de changer de branche (d’où le “aborting”).

En réalité, git ne va pas bloquer le changement de branche pour tous les changements. Mais ici git détecte que la dernière version de README

(enregistrée dans git) est différente sur les branches main et feature2. Passer sur la branche main aurait donc comme conséquence de perdre les modifications en cours. Git nous dit (en anglais): “Your local changes to the following files would be overwritten by checkout”. Vos changements locaux sur les fichiers suivants seraient écrasées par le changement de branche.

Ici nous avons deux options. Nous pouvons supprimer les modifications.

Il nous faut donc soit supprimer les modifications, soit les mettre de côté dans une réserve (c’est justement ce que signifie “stash” : une réserve). Nous allons donc réserver les modifications. Pour cela, rien de plus simple, il suffit de lancer la commande :

```
git stash
```

Git nous indique :

```
Saved working directory and index state WIP on feature2: 4743f56 Merge branch 'main' into feature2
```

Si nous regardons le fichier README.md, nous voyons:

```
# Ceci est un readme

feature1

feature2
```

Autrement dit, la modification en cours (la ligne “feature 2.1” que nous avons rajouté) a été sauvegardée séparément, et notre branche est à présent dans un état propre (sans modifications non sauvegardées).

A présent, retournons sur la branche “main” qui pour le moment ne comporte que les modifications provenant de “feature1”, en faisant :

```
git checkout main
```

La transition se passe bien à présent (puisque la branche ou nous étions n’as plus de modifications en cours). Si nous ouvrons le fichier README nous voyons qu’il ne comporte que les modifications provenant de "feature1”.

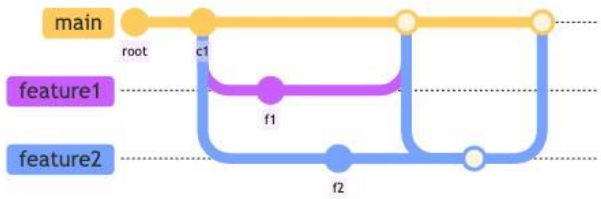
Pour récupérer les modifications venant de la branche feature2, nous allons donc faire :

```
git merge feature2
```

Si nous vérifions le contenu de README, nous voyons que le fichier comporte bien les modifications venant de la branche feature2 :

```
# Ceci est un readme

feature1

feature2
```

Notre arbre git ressemble donc à présent à quelque chose comme ceci :



… au détail près qu'il reste encore des modifications qui ont été mises de côté dans la réserve, le "stash". Pour lister les modifications qui ont été réservées, il faut lancer la commande :

```
git stash list
```

Git nous répond :

```
stash@{0}: WIP on feature2: 4743f56 Merge branch 'main' into feature2
```

Autrement dit, nous avons une modification (en position "0") dans la réserve.

Pour appliquer ce changement, rien de très compliqué. Il suffit de lancer:

```
git stash apply
```

Git nous informe alors avoir appliqué les changements qui étaient en réserve dans le stash:

```
On branch main
Changes not staged for commit: [...]
  modified: README.md

no changes added to commit
```

Si nous ouvrons le fichier README nous voyons que la ligne feature 2.1a bien été rajoutée. Par contre si nous demandons à git dans quel état il est, il nous informe (comme l'avait fait le stash) que les modifications n'ont pas été enregistrées. Comme si nous avions simplement modifié le fichier. Pour cela il faut faire un add et un commit.

Imaginons qu'au final nous ne voulons pas appliquer les modifications, comment pouvons nous annuler les modifications en cours ?