



GIT & GITHUB

GÉREZ VOTRE CODE AVEC LE CONTRÔLE DE VERSION



Avant propos.....	4
Le controle de versions.....	5
Qu'est ce qu'un gestionnaire de versions?	6
La notion de dépôt.....	8
Dépôt local et dépôt distant.....	9
Git.....	11
GitHub	12
Le Terminal	13
Où trouver le terminal?.....	15
racine & Arborescence	17
Commandes de base pour naviguer	18
Commandes pour Manipuler	20
Raccourcis clavier	22
Installation de l'environnement	24
Installation de Git.....	25
Configuration de Git et aide	27
Creation d'un compte GitHub	29
GitHub dans la ligne de commandes	33
Premiers pas avec Git.....	35
Les Différents Etats de Git	36
Initialiser Git pour un projet	37
Mon Premier Commit.....	38
Second Commit.....	40
GitIgnore.....	44
Revenir en arrière	46
Conclusion.....	52
Github	53
Ajout de notre dépôt a GitHub.....	54
Ajouter un Readme	58
Créer un Dépôt directement sur GitHub	59
Cloner un dépôt distant	61

Commit sur GitHub depuis un dépôt cloné	62
PULL, « Tirer des modifications »	63
branches et travail en collaboration	65
Créer une branche et en changer sur Git.....	66
Notre première Merge	68
Fusionner des branches en local.....	70
Rebaser.....	71
Supprimer une branche	73
GERER LES CONFLITS DE CODE.....	74

AVANT PROPOS

Bonjour et bienvenue dans l'apprentissage de Git & GitHub. Si vous lisez ces lignes, je peux supposer que vous connaissez déjà au moins quelques notions de développement. Que ce soit le développement Web, mobile ou autre. Vous avez sûrement aussi déjà entendu parler du contrôle de versions de votre code. Si ce n'est pas encore le cas, pas de soucis nous allons voir ou revoir tout ceci ensemble.

En effet, lorsque l'on travaille seul ou en collaboration, savoir versionner son code, avoir la possibilité de revenir en arrière et travailler à plusieurs sans affecter le travail de son ou sa collègue est primordial.



Je me suis efforcé de rendre cet apprentissage accessible à tous, quel que soit la technologie que vous utilisez, le système d'exploitation sur lequel vous êtes mais aussi quel que soit votre niveau.

J'espère de tout cœur que cette formation conviendra au plus grand nombre et que vous en ressortirez avec un des bases solides. Vous pourrez ainsi dire que le contrôle de version avec Git & GitHub n'a plus aucun secret pour vous !

Matthieu.

LE CONTROLE DE VERSIONS

Dans ce chapitre nous allons parler du contrôle de version et comprendre un peu mieux l'utilité de cette façon de gérer notre code. Mais nous allons aussi présenter les outils que nous allons utiliser ici à savoir Git et GitHub.

Nous allons donc pas à pas voir:

- Qu'est ce que le contrôle de versions?
- La notion de dépôt, un mot clé très important et omniprésent tout au long de cet apprentissage.
- Qu'est ce que Git?
- Qu'est ce que GitHub?

A la fin de ce chapitre, nous serons à même de connaître les différences entre les outils utilisés ainsi que les raisons pour lesquelles nous avons décidé de gérer notre code, qu'il soit du code pour site web, applications mobiles ou autre...

QU'EST CE QU'UN GESTIONNAIRE DE VERSIONS?

Définition:

Le gestionnaire de versions est un programme ou logiciel qui a pour but de conserver un historique des modifications de versions de tous les fichiers sur plusieurs branches.

Il garde en mémoire les modifications, les raisons ainsi que le nom de l'auteur de ce changement.

Une personne travaillant seule garde ainsi l'historique des modifications de son projet.
(Dépôt)

Quand au travail en équipe, le gestionnaire fusionne les modifications des différentes personnes travaillant en même temps sur un même projet.

Les fonctionnalités clé:

Grâce à cette définition, nous pouvons ainsi noter les fonctionnalités clés du gestionnaire de version. Ce sont les principales raisons qui nous pousserons à utiliser cet outil. A savoir:

- Revenir à une version antérieure facilement.
- Travailler en collaboration sans avoir la crainte de supprimer le travail d'une autre personne.
- Annoter et valider chaque étape du projet. Une sorte de checklist.

On pourrait aussi ajouter à cela la faculté de travailler en remote (télé travail) plus facilement. Et nous avons bien vu au cours des dernières années, que ce soit voulu ou pas, le télé travail a largement été utilisé.

Bon à savoir: Dans la plupart de vos entretiens d'embauche, le recruteur sera amené à vous demander si vous savez utiliser le gestionnaire de versions.

Exemple en solo:

Imaginons que nous avons un projet avec du code. Cela peut être un site, une application ou autre. C'est la version 1. Nous souhaitons ensuite y ajouter des fonctionnalités pour l'améliorer. Nous créons donc une version 2. Mais pour une raison ou une autre :bug, fonctionnalité inutile, travail effacé... Nous souhaitons retourner à la version 1.

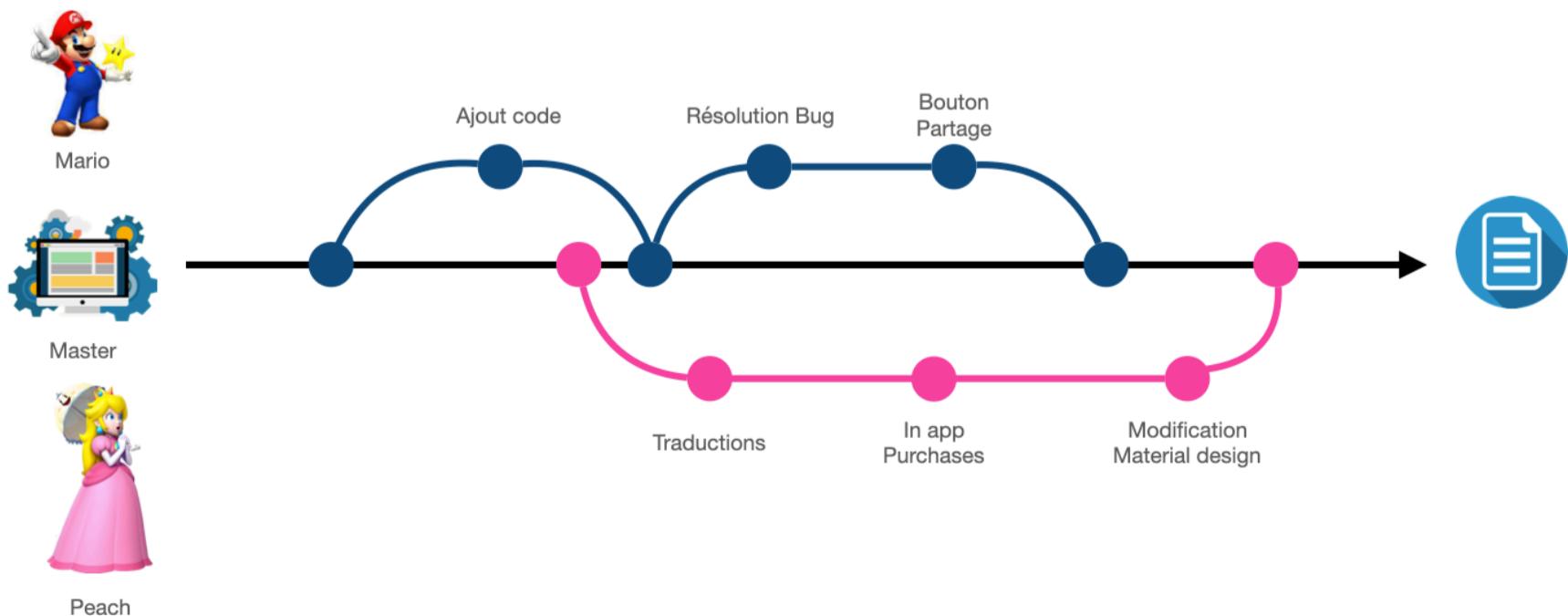
C'est là qu'entre en jeu le gestionnaire de versions. Il nous permettra dans ce cas précis de revenir en arrière dans l'historique.

Pour le travail en équipe:

Lorsque l'on travaille en équipe, grâce au gestionnaire de version, il est possible de travailler sur le même projet, en s'attelant à une tache précise sans avoir la crainte de supprimer le travail de son collègue. Pour cela, on utilisera des fonctionnalités comme:

- Les dépôts distants
- Les branches
- Les merges (fusion)

Exemple en image du travail en groupe:



LA NOTION DE DÉPÔT

Lorsque l'on travaille avec un logiciel de contrôle de versions, que ce soit Git, GitHub ou autre, la notion de dépôt est un élément du vocabulaire essentiel et indispensable.

On parle de dépôt pour désigner un « entrepôt » virtuel du projet. Il permet d'enregistrer et stocker les différentes versions du code et d'y accéder lorsque nous en avons besoin.

Pour créer un dépôt on peut:

- Créer un dossier vide pour générer un nouveau lieu de stockage.
- Cloner un dépôt pour accéder à un dépôt distant. Par exemple lorsque l'on veut travailler en local sur un dépôt distant.

A noter que dans les 2 cas, nous créons un nouveau dépôt. Chaque copie du code (ou clonage) correspond à un nouveau dépôt dans lequel sera aussi conservé l'historique des modifications. En effet, Git possède comme caractéristique d'être décentralisé. Tout n'est pas forcément stocké à un endroit unique.

Vous verrez aussi le mot **repository**. C'est la traduction anglaise de dépôt. Ne soyez donc pas étonné si on vous parle de l'un ou de l'autre. Ils signifient la même chose.

DÉPÔT LOCAL ET DÉPÔT DISTANT

Vous avez sûrement remarqué que dans les pages précédentes, une distinction a été faite entre dépôt local et dépôt distant.

Le dépôt Local:

C'est le fameux « entrepôt » ou dossier virtuel dans lequel vous allez effectuer les modifications et enregistrer les versions de votre code en local. **Local signifie directement sur votre machine.** Vous pourrez ainsi y accéder facilement et travailler rapidement sur votre projet.

Prenez votre code comme un Lego que vous avez récemment acheté:

Lorsque vous ouvrez la boîte, et que vous vous lancez dans sa construction, vous créez votre dépôt.

Chaque étape de sa construction dans le manuel correspond à une version du montage.



Imaginez donc que pour chaque étape de votre code, à l'image du lego, vous enregistriez sur git une nouvelle version. Ainsi si une étape n'est pas respectée ou que cela génère une erreur, vous pouvez revenir à tout moment en arrière à l'étape voulue.

Ps: L'exemple fonctionne aussi avec les meubles en Kit.

Ainsi votre code ou votre jouet sera versionné selon les étapes du plan défini et stocké dans le dépôt local.

Le dépôt distant:

Quand au dépôt distant, **il n'est pas stocké sur la machine mais sur internet ou sur un réseau local.** Son but principal est de garder un historique et un contrôle de version délocalisé et accessible à vos collaborateurs, ou à vous-même bien sûr.

Pour une personne travaillant en solo, il est très utile car il va garder une copie de votre travail. Si vous versez du café sur votre machine, que toutes vos données locales sont perdues, une copie de votre dépôt en distant vous permettra de retrouver aisément votre travail.

Lorsque vous travaillez en collaboration, chaque personne aura une copie (clone) en local pour effectuer les modifications et fusionnera (merge) le projet distant pour y ajouter uniquement ce qu'il a modifié sans écraser les changements des autres personnes.



Reprenons l'exemple du Lego et passons au niveau supérieur: Lego Master, le jeu télévisé. Des groupes de candidats s'affrontent pour créer des structures de briques selon l'énoncé du présentateur.

Prenons le plateau de création comme un dépôt distant. Chaque personne d'une même équipe y clone le projet et y ajoute des améliorations. Une fois l'amélioration validée, elle est intégrée au plateau sans qu'elle n'ait altéré ou remplacé le travail en cours de son équipier.

Cela paraît plus clair?

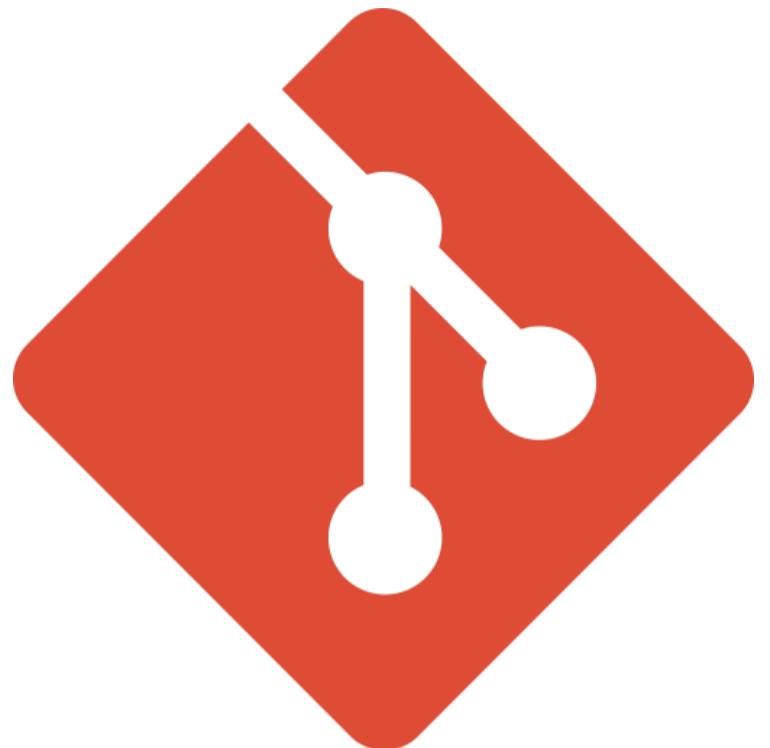
A noter aussi que dans les dépôt distants, nous avons 2 types de dépôts en accès: Public ou Privé.

- Le public sera accessible à tous et n'importe qui pourra collaborer.
- Pour le privé, vous déciderez qui pourra avoir accès au projet. Vos collaborateurs, clients ou managers...

GIT

Git est un logiciel de gestion de versions décentralisé libre et gratuit. Il a été créé en 2005 par l'auteur du noyau Linux: Linus Torwalds.

Fait amusant, Git n'est autre qu'un mot d'argot britannique qui signifie « connard ». Il a été nommé ainsi par son créateur car de nombreuses personnes le trouvaient imbu de sa personne et égocentrique. Un Git en quelque sorte. A savoir que Linus Torwalds a tendance à nommer ses projets en fonction de sa propre personne. Git donc, mais ne trouvez vous pas que Linux, une autre de ses créations ressemble étrangement à son prénom?
(Linus)



Dès le début des années 2010, Git est le logiciel de contrôle de versions le plus populaire et le plus utilisé dans le monde. Des dizaines de millions de développeurs utilisent cet outil, que ce soit sous Mac, Windows et bien évidemment Linux. D'ailleurs, pour les utilisateurs de Mac et les développeurs iOS avec Xcode, git est automatiquement installé sur la machine avec l'installation des lignes de commande. Git est aussi le système de base de GitHub, dont nous allons parler plus tard qui lui est le plus grand hébergeur de code sur le web.

Git ne repose pas sur un serveur centralisé, mais il utilise un système de connexion peer to peer. Le code informatique développé est stocké non seulement sur l'ordinateur de chaque contributeur du projet, mais il peut également l'être sur un serveur dédié. C'est un outil de bas niveau, qui se veut simple et performant, dont la principale tâche est de gérer l'évolution du contenu d'une arborescence d'un projet.

La décentralisation de Git a beaucoup apporté au développement des logiciels libres, puisque le besoin de demander un compte sur un dépôt centralisé devient obsolète. Il suffit ainsi de cloner un projet pour commencer à travailler dessus. L'historique du projet suit naturellement. Chacun peut ainsi proposer sa contribution au dépôt principal.

GITHUB

Github est un service web d'hébergement et de gestion de versions et de développement de logiciels lancé en 2008. Il utilise le logiciel de gestion de versions Git.



Se voulant accès sur un modèle typé réseau social, son nom est composé de Git, le contrôle de version open source et Hub qui signifie noyau et fait référence en anglais aux réseaux sociaux.

On peut ainsi déposer ses dépôts, contribuer sur d'autres dépôts, suivre des personnes et aussi aimer (star) certains dépôts.

Pour chaque dépôt, en plus du contrôle de versions. On peut aussi entre autres, suivre les bugs, demander des fonctionnalités, suivre les tâches, créer un Wiki, un readme et une page web.

GitHub propose plusieurs tarifications selon les besoins. Un service gratuit pour les particuliers, les logiciels libres, les projets collaboratifs mais aussi des comptes payants pour les entreprises ne voulant pas partager leur code avec tous.

Grâce à ces fonctionnalités, le site est devenu le plus important dépôt de code au monde, utilisé comme dépôt public de projets libres ou dépôt privé d'entreprises.

En 2018, GitHub est acquis par Microsoft. Ceci lui a valu un bad buzz et quelques développeurs se sont ainsi tournés vers leurs concurrents de peur de voir Microsoft rendre ce service payant et moins libre qu'auparavant. Cependant, GitHub est resté le plus populaire des hébergeurs de dépôt centralisés.

GitHub possède plusieurs alternatives qui reprennent les éléments qui ont fait son succès comme par exemple:

- GitLab
- BitBucket

LE TERMINAL

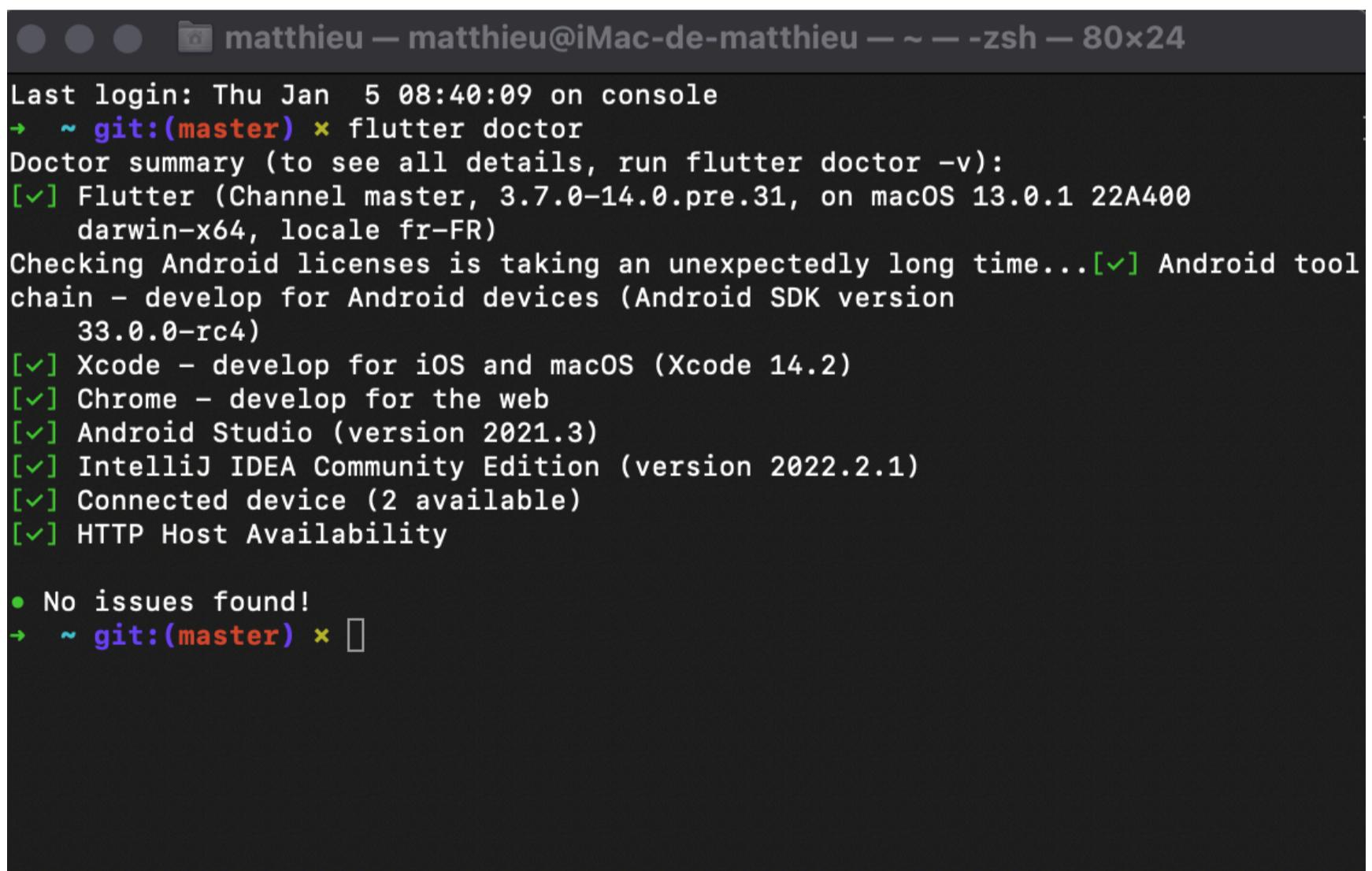
Avant de se lancer dans le contrôle de version, nous devons maîtriser un logiciel qui nous permettra d'utiliser Git, à savoir le Terminal.

Définition:

C'est un système de commandes en mode texte sans interface graphique qui ne s'utilise qu'à l'aide du clavier.

Bien qu'il ne soit pas visuellement attrayant, il permet d'effectuer rapidement de nombreuses opérations comme la création de dossier, de fichiers, leur suppression mais pour nous il sera bien utile dans l'utilisation de git.

Voici un exemple de mon terminal sous macOS lorsque je vérifie si tout est bien installé pour le développement avec Flutter:



```
matthieu — matthieu@iMac-de-matthieu — ~ — -zsh — 80x24
Last login: Thu Jan  5 08:40:09 on console
→ ~ git:(master) ✘ flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel master, 3.7.0-14.0.pre.31, on macOS 13.0.1 22A400
    darwin-x64, locale fr-FR)
Checking Android licenses is taking an unexpectedly long time...[✓] Android tool
chain - develop for Android devices (Android SDK version
    33.0.0-rc4)
[✓] Xcode - develop for iOS and macOS (Xcode 14.2)
[✓] Chrome - develop for the web
[✓] Android Studio (version 2021.3)
[✓] IntelliJ IDEA Community Edition (version 2022.2.1)
[✓] Connected device (2 available)
[✓] HTTP Host Availability

• No issues found!
→ ~ git:(master) ✘
```

Cela nous fait remonter dans le temps lorsque les PC utilisaient Dos n'est pas?

Avantages et Inconvénients:

- +: Rapide
- +: Certains programmes ne sont accessibles que via le terminal
- +: Utile pour réaliser des scripts
- +: Rend certaines tâches beaucoup plus simple une fois l'outil pris en main
- : Austère
- : Il faut au minimum connaître son système d'exploitation pour ne pas se retrouver bloqué
- : Utilise un lexique spécifique pour effectuer ses tâches.

Bien que cette interface ne soit pas la plus attrayante, elle est relativement simple à utiliser et nous allons voir ceci dans ce chapitre.

OU TROUVER LE TERMINAL?

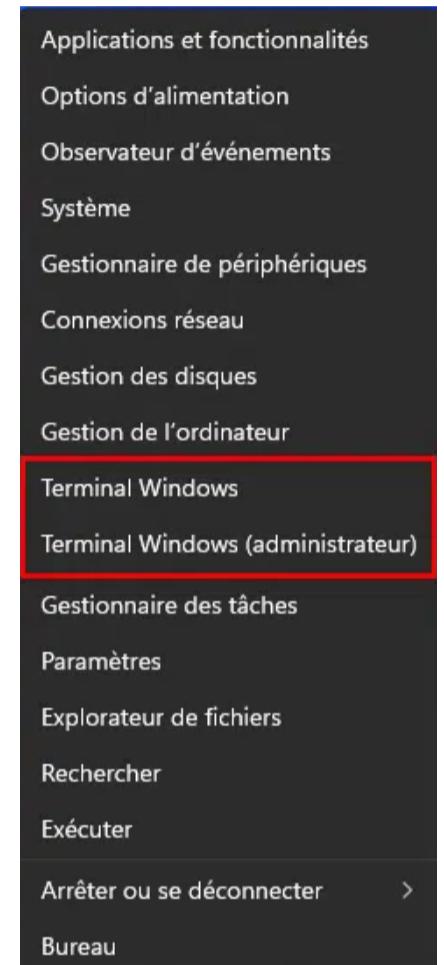
Le terminal est présent sur toutes vos machines. Que ce soit Windows, Linux ou bien macOS. Pour les trouver et les ouvrir, c'est relativement simple, cependant nous allons voir ensemble comment y accéder:

Sur Windows 11:

Nous commençons par le système d'exploitation le plus utilisé: Windows. Pour trouver le terminal. Le plus simple est d'utiliser le menu lien rapide. Pour ceci, il y a le raccourci clavier Windows + X ou clic droit sur le menu démarrer. Le Terminal Windows sera un des onglets proposés. Vous pouvez l'utiliser en version simple ou avec les droits d'administrateur.

Vous pouvez aussi dans le menu démarrer entrer une recherche Terminal Windows. Dans ce cas, pour l'utiliser en tant qu'administrateur vous devrez faire clic droit -> Plus -> Exécuter en tant qu'administrateur.

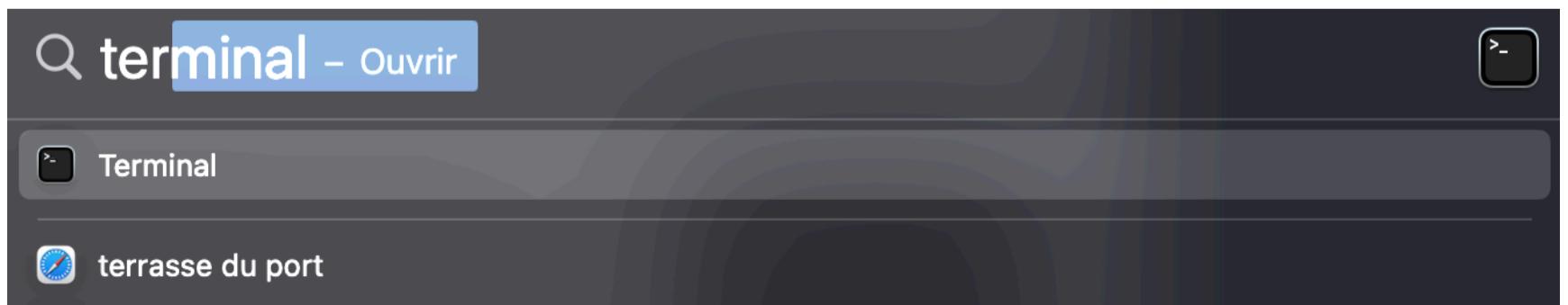
Il existe bien d'autres chemins mais j'ai listé ici les 2 plus rapides.



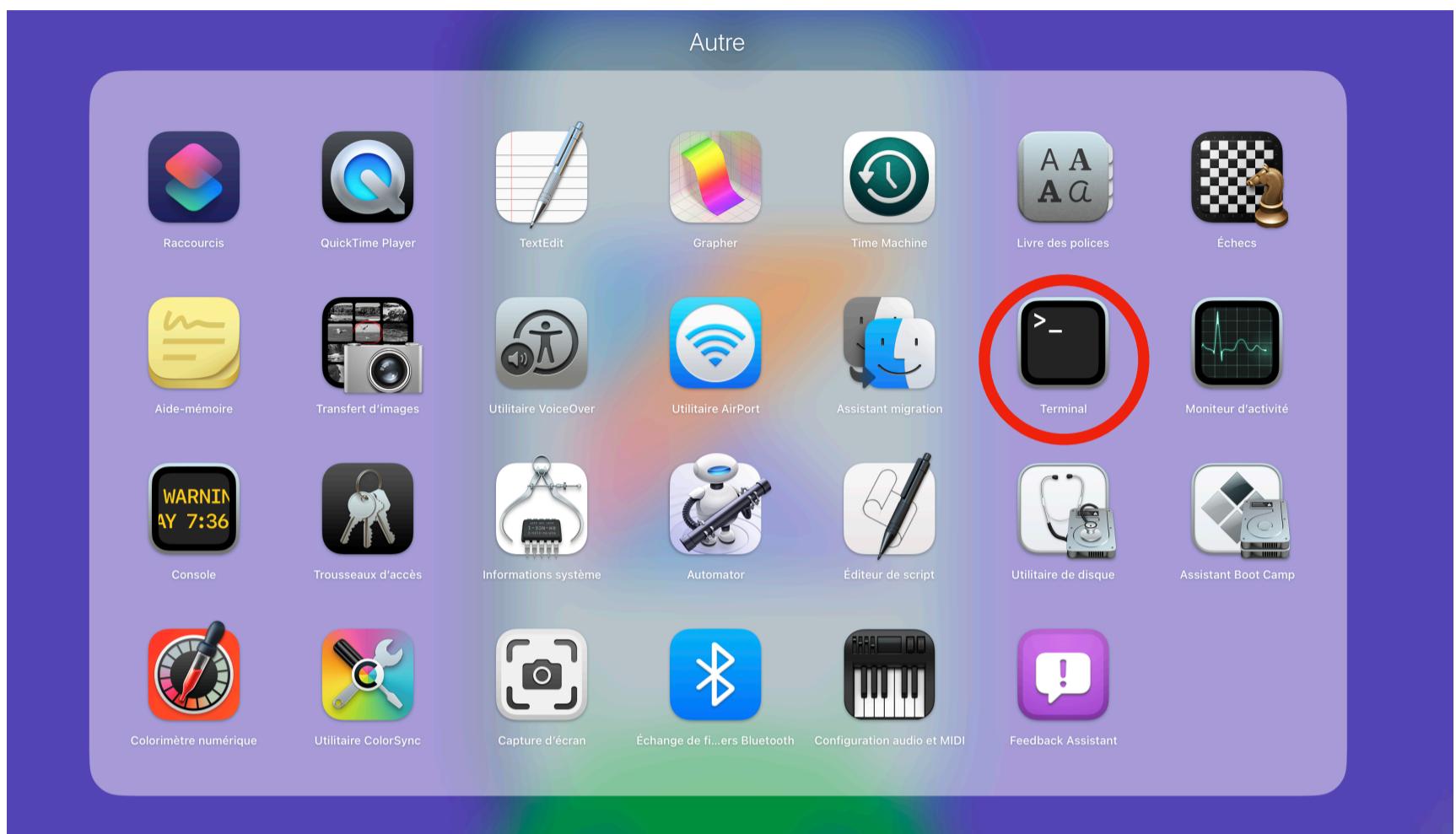
Sur macOS:

Passons maintenant au Mac. Là aussi nous allons voir le raccourci: **Command + la barre d'espace pour ouvrir la recherche spotlight.**

Puis entrez terminal dans la barre de recherche. Choisissez terminal pour lancer l'application.



Vous pouvez aussi aller dans le Launchpad, puis le dossier autre et y retrouver le terminal.



Sur Linux:

Ici le raccourci est encore plus simple sur Ubuntu et ses déclinaisons: Il suffit de taper sur son clavier: CTRL + ALT + T et voilà! Le terminal s'ouvre !

Depuis le menu contextuel, (clic droit sur le bureau), cliquez sur ouvrir dans un terminal.

Nous avons tous ouvert notre terminal, il est temps de passer aux choses sérieuses !

RACINE & ARBORESCENCE

Pour se déplacer au sein de votre machine et effectuer des actions, par exemple trouver un dossier, en créer un autre ou encore créer un dépôt Git, notre terminal doit connaître le chemin vers lequel se déplacer.

Le premier élément à prendre en compte est la racine! C'est le point de départ de l'arborescence de vos fichiers. Et selon le système d'exploitation que vous utilisez, il sera différent:

- / Pour macOS et Linux
- C:\ pour Windows

Une fois entré dans la racine, nous pourrons aller un peu plus loin selon l'arborescence de notre disque:

Ex sur Mac: /Users/Matthieu/Documents

Ex sur Linux: /home/Gerard/Documents

Ex sur Windows: C:\Users\Bernard\Documents

Pour savoir où vous vous trouvez tapez pwd dans votre terminal puis entrée et vous aurez la réponse.



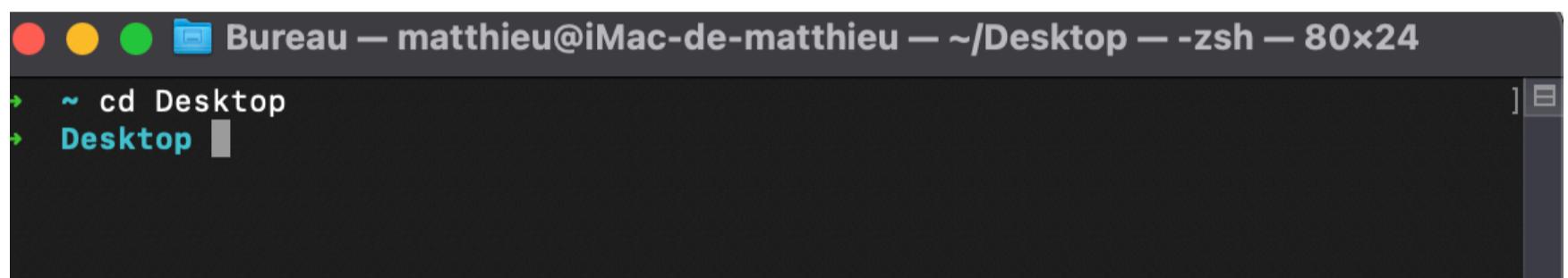
A screenshot of a macOS terminal window. The title bar says "matthieu — matthieu@iMac-de-matthieu". The command line shows the user is in their home directory: ~ git:(master) x pwd /Users/matthieu → ~ git:(master) x

COMMANDES DE BASE POUR NAVIGUER

cd: Change Directory.

Si vous tapez cd simplement, vous vous retrouverez dans le répertoire racine de l'utilisateur.

Si vous voulez passer aux dossiers suivant l'arborescence, vous tapez cd suivi du nom du dossier:



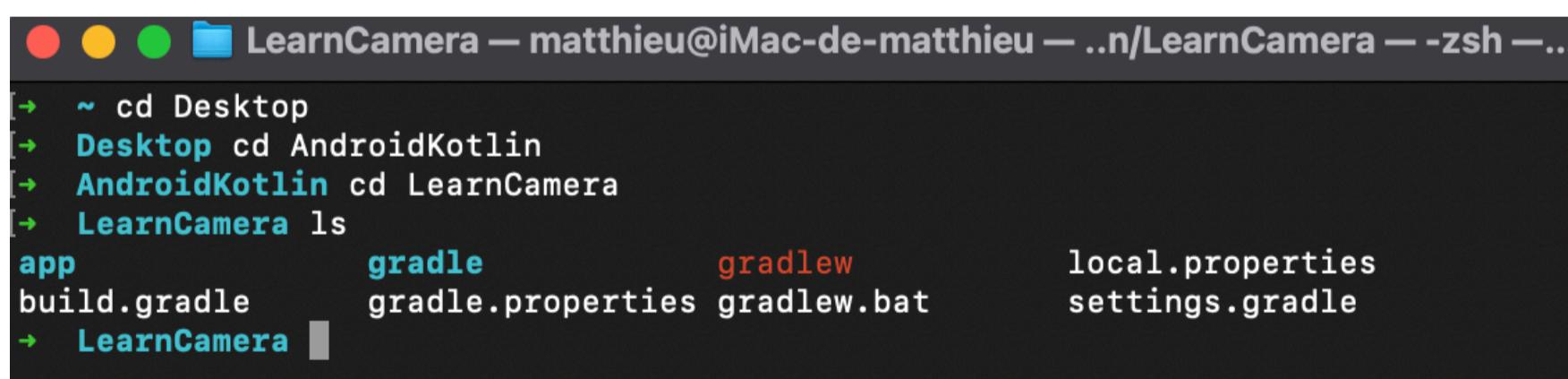
```
Bureau — matthieu@iMac-de-matthieu — ~/Desktop — -zsh — 80x24
→ ~ cd Desktop
→ Desktop
```

On peut voir que ici je suis passé au dossier Desktop, à savoir mon bureau

cd - vous permettra de revenir dans le dossier précédent, alors que cd -- vous amènera tout comme cd seul à la racine de l'utilisateur.

ls: abréviation de List.

Il permet comme son nom l'indique de lister tous les dossiers et fichiers présents à l'endroit où vous vous trouvez.

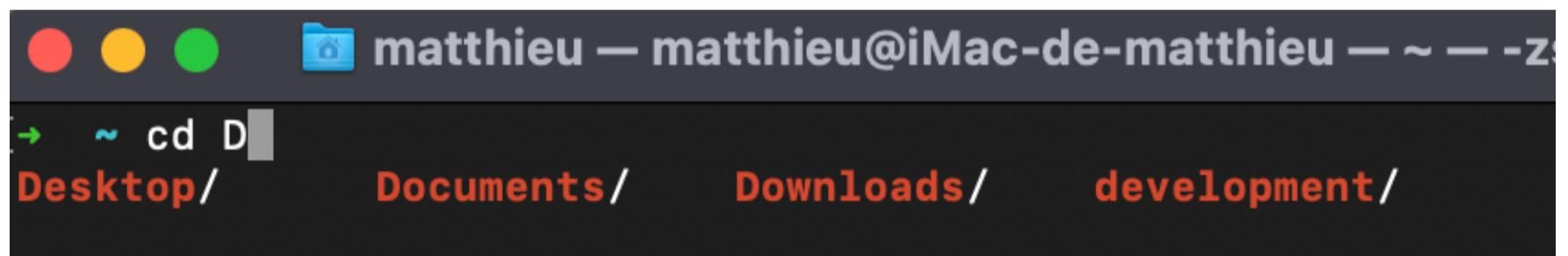


```
LearnCamera — matthieu@iMac-de-matthieu — ..n/LearnCamera — -zsh — ...
→ ~ cd Desktop
→ Desktop cd AndroidKotlin
→ AndroidKotlin cd LearnCamera
→ LearnCamera ls
app           gradle          gradlew      local.properties
build.gradle   gradle.properties gradlew.bat  settings.gradle
→ LearnCamera
```

J'ai navigué dans un dossier Android Kotlin, puis dans un dossier LearnCamera, qui est une application montrée dans une formation, et j'ai listé avec ls tout le contenu de ce dossier. On peut ainsi y retrouver tous les fichiers et dossiers présents.

Tabulation:

Ce n'est pas en soi une commande mais une autocompletion qui nous permet utilisé avec cd de au lieu de tout réécrire de pré remplir notre chemin. Si il y a plusieurs possibilités, ces dernières sont proposées.

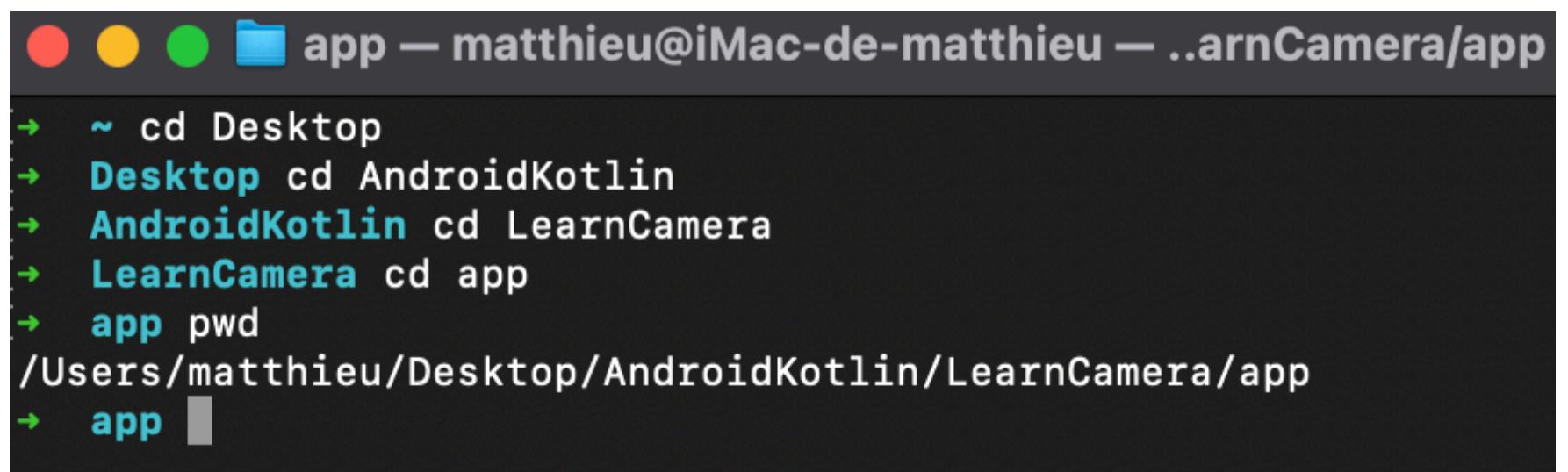


A screenshot of a macOS terminal window. The title bar shows the user's name and session information. The command line shows the user has typed 'cd D' and is pressing Tab to complete it. The terminal lists four possible completions: 'Desktop', 'Documents', 'Downloads', and 'development'. The 'Desktop' option is highlighted in red, indicating it is the selected choice.

Si par exemple ici je commence à taper cd d puis que j'appuie sur Tab, le terminal me propose ces choix. Evidemment nous n'avons pas la même arborescence et vous aurez des propositions différentes.

pwd: Print Working Directory.

On va ici imprimer sur notre terminal notre chemin. Ce qui peut bien être utile à certains moments soit pour copier notre chemin, soit pour se retrouver. Le terminal est comme le petit Poucet, il a semé des petits cailloux pour retrouver la sortie parmi le dédale que peut parfois être notre ordinateur.

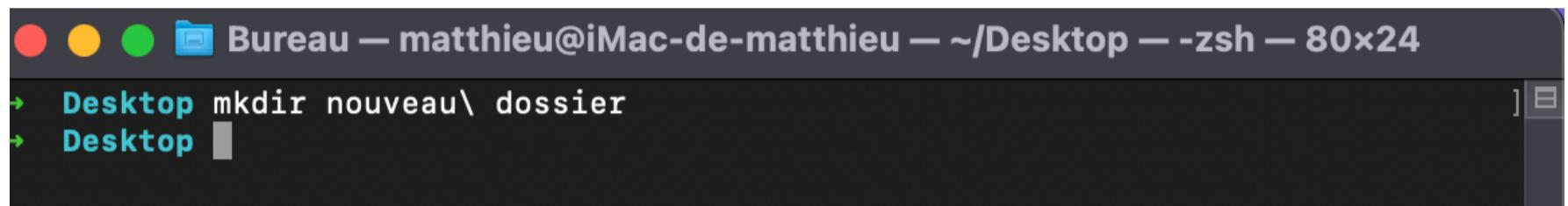


A screenshot of a macOS terminal window. The title bar shows the user's name and session information. The command line shows the user has typed 'pwd' and is pressing Enter to execute the command. The terminal displays the full path '/Users/matthieu/Desktop/AndroidKotlin/LearnCamera/app' followed by a new line character and a cursor at the end of the line.

COMMANDES POUR MANIPULER

mkdir: Make Directory

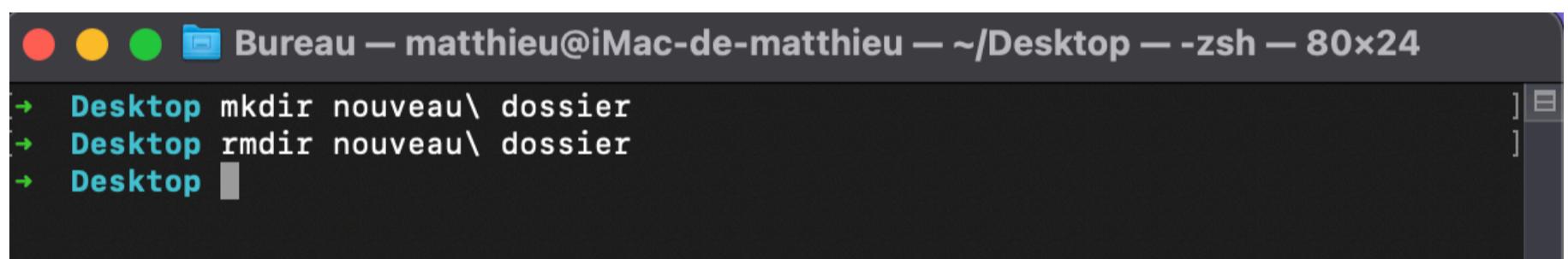
En français, cela se traduit par créer un dossier. Ici nous pouvons directement via le terminal créer un dossier en utilisant mkdir puis le nom du dossier. Si on veut un espace dans le nom, il nous suffit d'ajouter un slash \ avant l'espace. Sinon vous créerez 2 dossiers.



```
● ● ● ⌂ Bureau — matthieu@iMac-de-matthieu — ~/Desktop — -zsh — 80x24
→ Desktop mkdir nouveau\ dossier
→ Desktop
```

rmdir: Remove Directory.

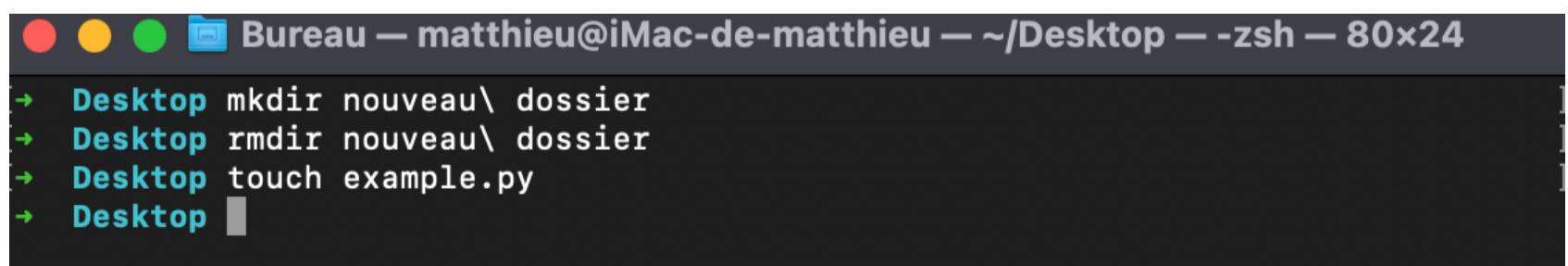
Si on peut créer un dossier, on peut le supprimer. Il suffit ainsi de modifier mkdir par rmdir. Et hop! Le dossier a disparu.



```
● ● ● ⌂ Bureau — matthieu@iMac-de-matthieu — ~/Desktop — -zsh — 80x24
→ Desktop mkdir nouveau\ dossier
→ Desktop rmdir nouveau\ dossier
→ Desktop
```

touch: Créer un fichier

Alors la fonction à la base pour touch n'était pas la création de fichier, bien qu'elle en soit devenu sa principale utilité. En effet, elle a été créée pour modifier le timestamp, une valeur de temps sur un dossier. Nous l'utiliserons ici pour sa faculté à créer un fichier. Pour ceci, il

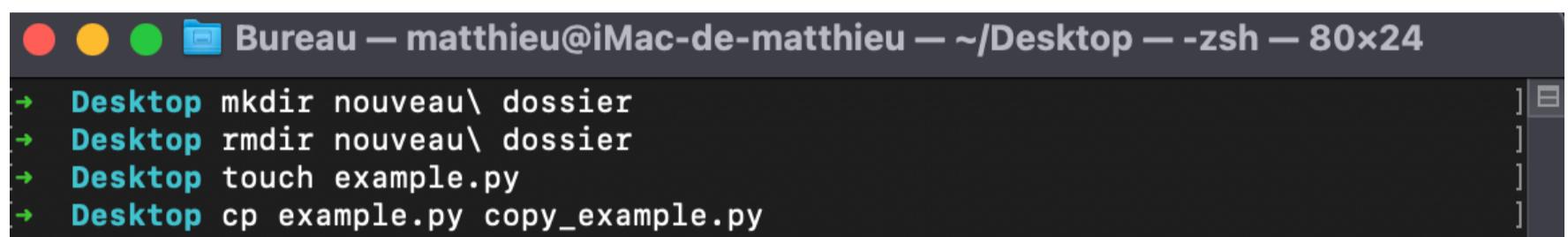


```
● ● ● ⌂ Bureau — matthieu@iMac-de-matthieu — ~/Desktop — -zsh — 80x24
→ Desktop mkdir nouveau\ dossier
→ Desktop rmdir nouveau\ dossier
→ Desktop touch example.py
→ Desktop
```

faut le mot clé touch suivi du nom et du format du fichier. Ici nous créons un fichier python nommé example.

cp: Copy

Pour copier un fichier, nous utilisons cp suivi du nom du fichier existant (source) et enfin le nom du fichier à créer (destination).



```
● ● ● └ Bureau — matthieu@iMac-de-matthieu — ~/Desktop — -zsh — 80x24
→ Desktop mkdir nouveau\ dossier
→ Desktop rmdir nouveau\ dossier
→ Desktop touch example.py
→ Desktop cp example.py copy_example.py
```

mv: Move

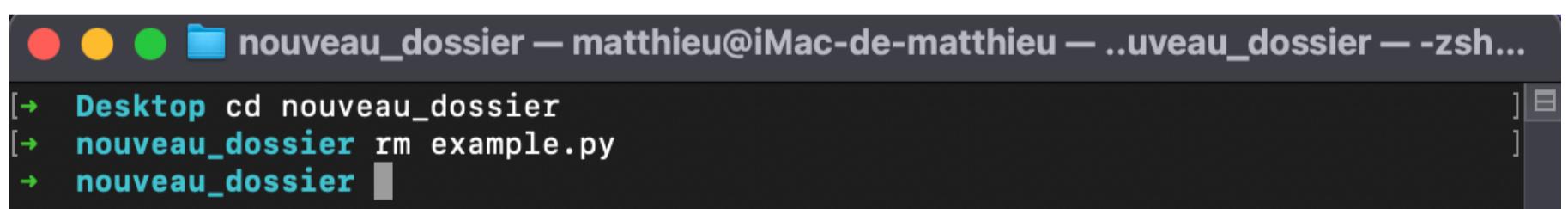
Change l'emplacement du fichier ou dossier. Mv suivi du nom de l'élément à déplacer suivi de sa destination. Si la destination n'existe pas, elle sera créée. Ici j'ai créé un nouveau dossier et j'y ai déplacé example.py



```
● ● ● └ Bureau — matthieu@iMac-de-matthieu — ~/Desktop — -zsh — 80x24
→ Desktop mkdir nouveau_dossier
→ Desktop mv example.py nouveau_dossier
→ Desktop
```

rm: Remove

Enfin nous pouvons supprimer le fichier. Nous lui avons fait bien des misères, il est temps de s'en séparer. Il fonctionne comme le rmdir. Nous avons donc rm suivi du nom du fichier à supprimer.



```
● ● ● └ nouveau_dossier — matthieu@iMac-de-matthieu — ..veau_dossier — -zsh...
[→ Desktop cd nouveau_dossier
[→ nouveau_dossier rm example.py
→ nouveau_dossier
```

A noter que si le dossier n'est pas vide, nous devrons utiliser rm -r puis le nom du dossier.

RACCOURCIS CLAVIER

Passons maintenant aux raccourci clavier, car comme nous ne nous servons pas ou peu de la souris, ces derniers seront importants pour pouvoir se servir au mieux de notre terminal. J'ai listé uniquement les raccourcis qui me semblent importants et nécessaires pour une utilisation du Terminal. Ces derniers sont utilisés sur tous les systèmes d'exploitation. Il en existe bien d'autres que vous pouvez trouver facilement grâce à une simple recherche sur Internet.

Pour certains, vous utilisez macOS et donc avez un clavier un peu différent:

- CTRL sera Control
- Option sera ALT

Copier, Couper et Coller:

- **CTRL + L** : Effacer l'intégralité du contenu du terminal
- **CTRL + K**: Coupe tout ce qui se trouve sur la ligne après le curseur.
- **CTRL + U** : Coupe tout ce qui se trouve sur la ligne avant le curseur.
- **CTRL + W** : Coupe le mot précédent le curseur.
- **CTRL + Y**: Coller
- **ALT + D**: Coupe le mot suivant le curseur (nécessite d'activer option comme touche Méta pour macOS)

Modifier:

- **CTRL + T** : inverse les deux caractères avant le curseur
- **CTRL + _** : annule la dernière modification

Naviguer:

- **CTRL + A:** Curseur au début de ligne
- **CTRL + E:** Curseur en fin de ligne
- **CTRL + B:** Curseur au début du mot
- **ALT + F:** Curseur en fin de mot

Commandes:

- **CTRL + C:** Pour arrêter une commande en cours d'exécution.
- **CTRL + Z :** Mettre en pause une exécution de programme
- **fg:** Reprendre une exécution de programme mis en pause
- **CTRL + D:** Quitter le terminal

INSTALLATION DE L'ENVIRONNEMENT

Dans ce chapitre nous allons commencer à entrer dans le vif du sujet en installant et configurant tout ce dont nous avons besoin pour pouvoir utiliser Git & Github.

Cette section sera donc une mise en place des outils suivants:

- Installation de Git
- Configuration de Git
- Découverte du site GitHub
- Création d'un compte GitHub. Nous choisirons le plan gratuit.
- Tour de notre compte GitHub.

Si tout ceci est déjà configuré chez vous, vous pouvez passer au chapitre suivant ou revoir cette mise en place.

Je vous sent impatients alors c'est parti !

INSTALLATION DE GIT

Nous allons ensemble voir comment installer Git sur votre machine. Pour ceci, je vais tout d'abord lister des liens qui vous seront bien utiles lors de l'installation:

- Le site officiel de Git: <https://git-scm.com/>. Dès la page d'accueil, il pourra voir le système d'exploitation que vous utilisez et ainsi vous orienter vers la meilleure façon d'installer votre nouveau compagnon
- La page d'installation de Git: <https://git-scm.com/book/fr/v2/Démarrage-rapide-Installation-de-Git>. Ici seront listées toutes les façons de faire pour installer Git.

Comme nous savons nous servir du terminal, je vous invite tout d'abord à voir si une version de git est présente sur votre machine. Pour ceci, tapez dans le terminal git -- version



```
matthieu ~ % git --version
git version 2.39.0
```

Si rien ne se passe ou que vous avez un message command not found, cela signifie que Git n'est pas présent. Si comme moi il est présent, comparez la version indiquée avec celle présente sur le site Git. Si celle ci n'est pas la plus récente, suivez les indications selon votre Os.

Mettre à jour Git:

- **Pour MacOS:** Entrez dans le terminal **brew update** puis **brew upgrade git**
- **Pour windows:** Entrez dans le terminal **git update-git-for-windows**
- **Pour Linux:** Entrez dans le terminal **sudo apt-get update** puis **sudo apt-get install git**

Maintenant passons aux installations pour ceux qui n'ont pas encore Git.

Installation sur MacOS:

Pour installer Git sur Mac, vous aurez tout d'abord besoin d'installer Brew <https://brew.sh/>. C'est le gestionnaire de paquets pour macOS.

Pour ceci tapez dans votre terminal: `/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"` » et laissez-vous guider tout au long d'installation.

Une fois brew installé, tapez encore dans votre terminal **brew install git**. Et si cela est demandé, choisissez les paramètres par défaut.

Et voilà! Vous pouvez désormais revenir au `git -- version` pour vérifier si git est bien installé et à jour.

Il existe une façon beaucoup plus simple qui est l'intallation des lignes de commandes via Xcode, mais cet outil ne sera utile que si vous utilisez Xcode. En effet ce logiciel est assez volumineux.

Installation sur Linux:

Pour ceux qui sont sur Linux, vous le savez, il existe de nombreuses distributions. Pour installer git, allez sur ce lien et entrez dans le terminal la distribution qui est la vôtre: <https://git-scm.com/download/linux>

Installation sur Windows:

Pour Windows. Allez sur la page principale de git : <https://git-scm.com/> et choisissez le lien de téléchargement selon votre système d'exploitation.

Un fichier .exe sera téléchargé.

Suivez pas à la pas l'installation et choisissez les réglages par défaut. Ils conviennent parfaitement pour une utilisation simple et efficace de Git.

Lancez ensuite votre terminal et faites un `git -- version` histoire de vous assurer que tout a bien été installé?

CONFIGURATION DE GIT ET AIDE

Maintenant que tout le monde a bien installé Git. Nous allons configurer et utiliser la commande d'aide de Git.

En effet, il y aura à coup sûr à un moment du développement, une commande que vous aurez sur le bout de la langue. Pour la retrouver il suffit de taper dans le terminal git -- help

Et vous pourrez ainsi obtenir toutes les informations sur les commandes possibles. Voyez-vous le -- version dès la seconde ligne?

```
matthieu — matthieu@iMac-de-mathieu — ~ — zsh — 102x48

→ ~ git --help
usage : git [--version] [-h | --help] [-C <chemin>] [-c <nom>=<valeur>]
          [--exec-path[=<chemin>]] [--html-path] [--man-path] [--info-path]
          [-p | --paginate | -P | --no-pager] [--no-replace-objects] [--bare]
          [--git-dir=<chemin>] [--work-tree=<chemin>] [--namespace=<nom>]
          [--super-prefix=<chemin>] [--config-env=<nom>=<variable-d-environnement>]
          <commande> [<args>]

Ci-dessous les commandes Git habituelles dans diverses situations :

démarrer une zone de travail (voir aussi : git help tutorial)
  clone      Cloner un dépôt dans un nouveau répertoire
  init       Créer un dépôt Git vide ou réinitialiser un existant

travailler sur la modification actuelle (voir aussi : git help revisions)
  add        Ajouter le contenu de fichiers dans l'index
  mv         Déplacer ou renommer un fichier, un répertoire, ou un lien symbolique
  restore    Restaurer les fichiers l'arbre de travail
  rm         Supprimer des fichiers de la copie de travail et de l'index

examiner l'historique et l'état (voir aussi : git help revisions)
  bisect    Trouver par recherche binaire la modification qui a introduit un bogue
  diff      Afficher les changements entre les validations, entre validation et copie de travail, etc
  grep      Afficher les lignes correspondant à un motif
  log       Afficher l'historique des validations
  show      Afficher différents types d'objets
  status    Afficher l'état de la copie de travail

agrandir, marquer et modifier votre historique
  branch   Lister, créer ou supprimer des branches
  commit   Enregistrer les modifications dans le dépôt
  merge    Fusionner deux ou plusieurs historiques de développement ensemble
  rebase   Réapplication des commits sur le sommet de l'autre base
  reset   Réinitialiser la HEAD courante à l'état spécifié
  switch   Basculer de branche
  tag     Créer, lister, supprimer ou vérifier un objet d'étiquette signé avec GPG

collaborer (voir aussi : git help workflows)
  fetch   Télécharger les objets et références depuis un autre dépôt
  pull    Rapatrier et intégrer un autre dépôt ou une branche locale
  push    Mettre à jour les références distantes ainsi que les objets associés

'git help -a' et 'git help -g' listent les sous-commandes disponibles et
quelques concepts. Voir 'git help <commande>' ou 'git help <concept>'
pour en lire plus à propos d'une commande spécifique ou d'un concept.
Voir 'git help git' pour un survol du système.
```

Lorsque vous aurez besoin d'aide rapide, vous saurez où chercher!

Passons maintenant à la configuration de Git. La configuration sera notre identité, à savoir notre nom et notre adresse mail:

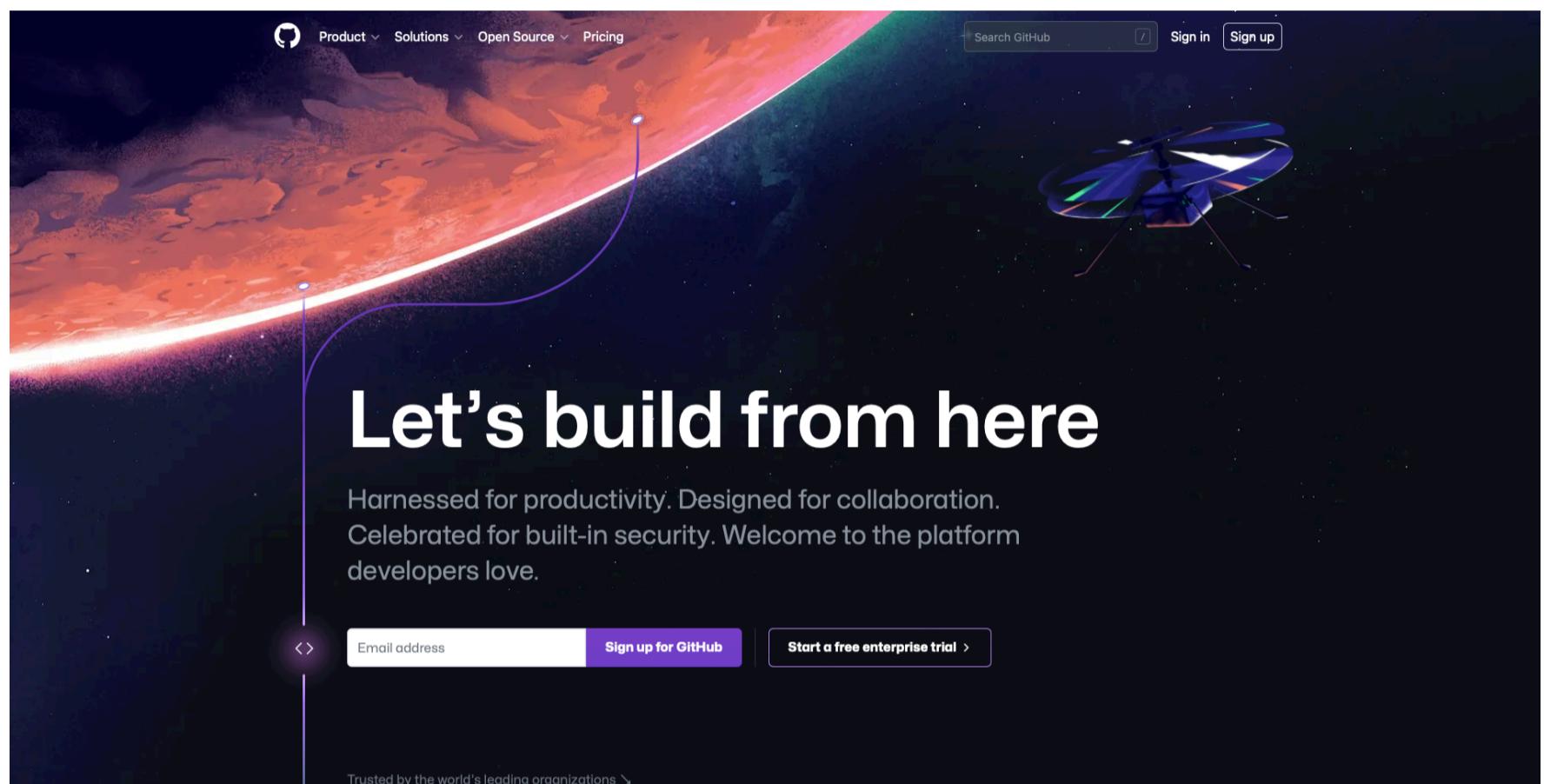
```
→ ~ git config --global user.name = "Matthieu Codabee"
→ ~ git config --global user.email matthieu@codabee.com
```

J'ai ici pu configurer mon nom et mon adresse mail grâce à ces commandes. Ce sera très important car il servira de nom dans les modifications mais aussi servira de validation.

CREATION D'UN COMPTE GITHUB

Lorsque nous avions parlé des dépôts locaux, nous avions vu ce qu'était Github. Il est désormais temps d'aller sur ce fameux site et y créer un compte !

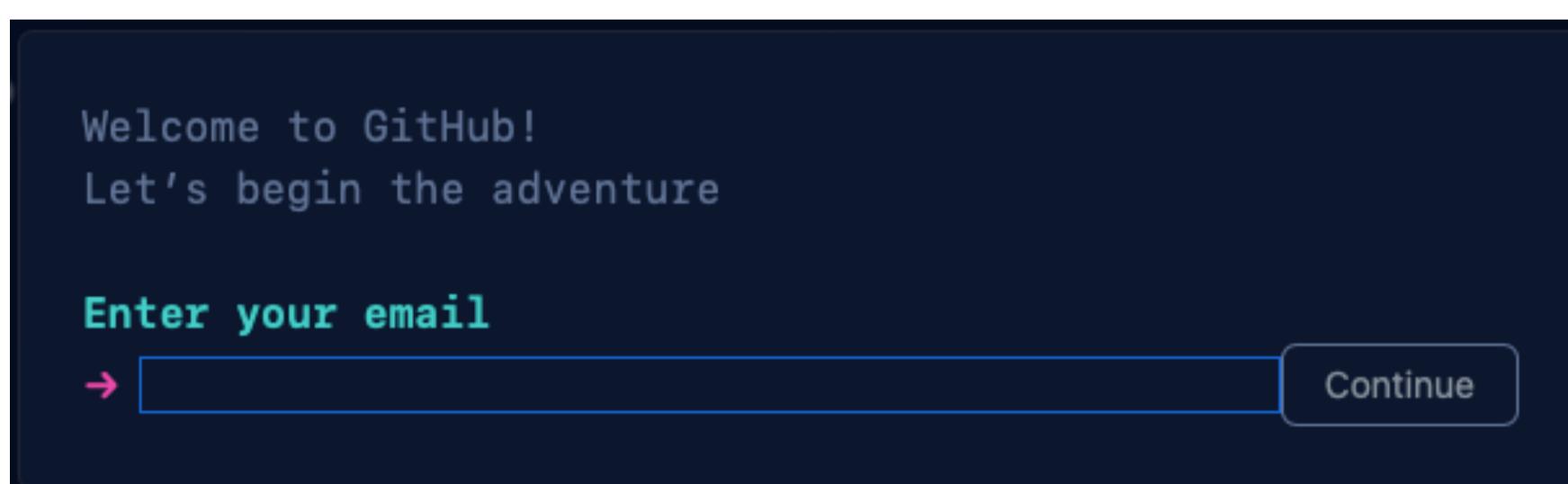
Pour ceci, ouvrez votre navigateur préféré et rendez vous sur : <https://github.com/>



L'affichage pourrait être différent. En effet, ils mettent régulièrement à jour leur page d'accueil mais le fonctionnement reste le même!

Allez sur le bouton Sign up en haut de page à droite et cliquez dessus.

Il ne reste qu'à suivre les instructions pour créer un compte



Welcome to GitHub!
Let's begin the adventure

Enter your email

✓

Create a password

✓

Enter a username

→

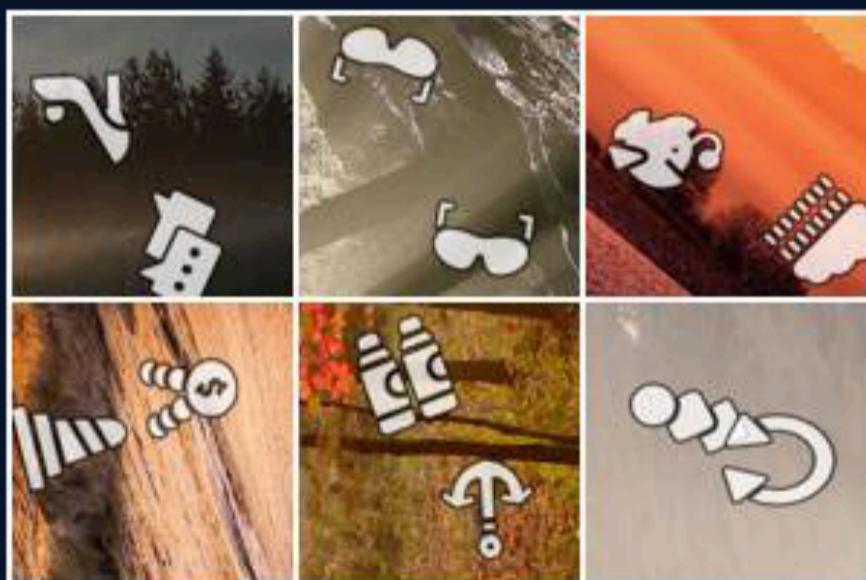
Continue

Vérifiez ensuite si vous n'êtes pas un robot.

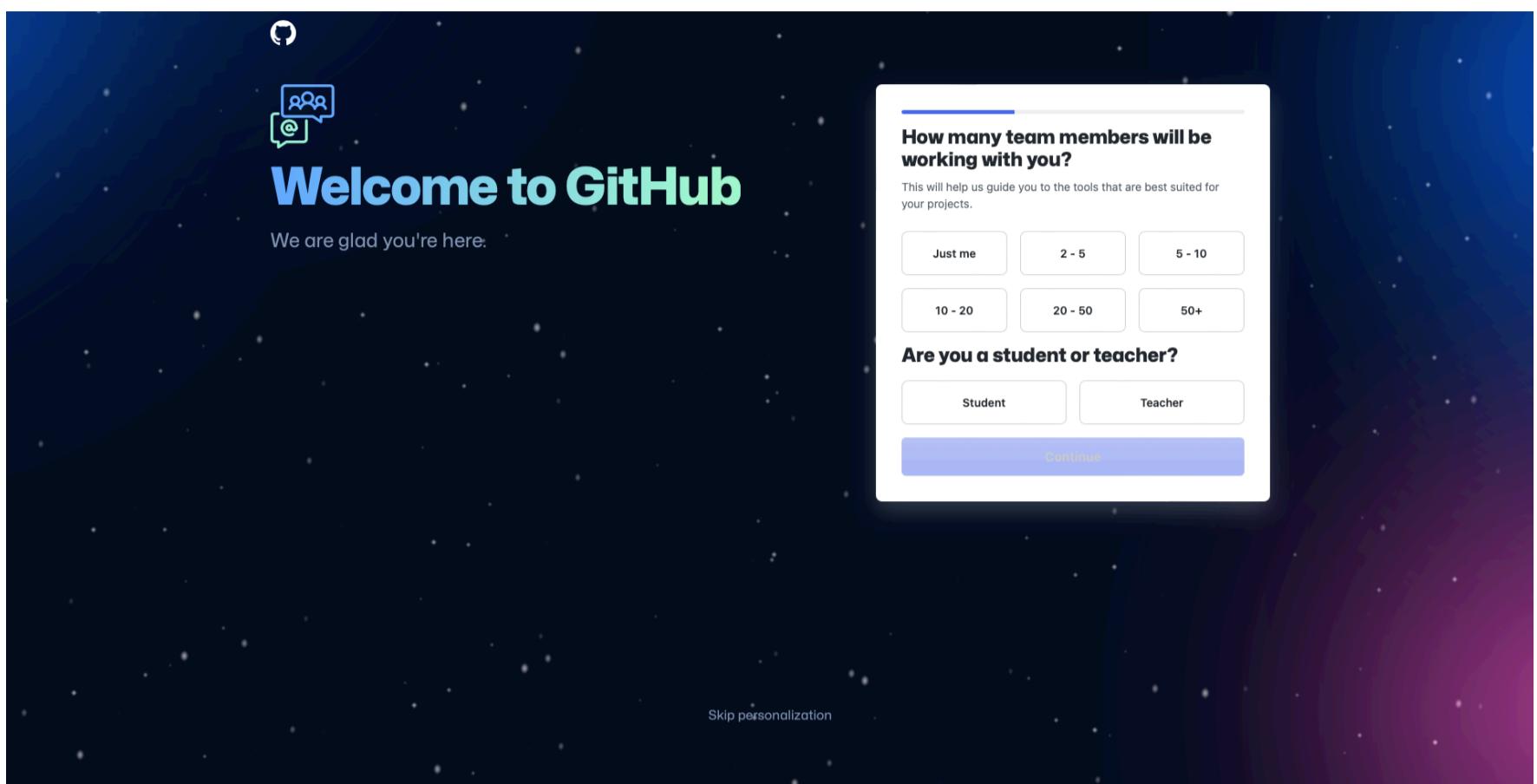
Et enfin un code de validation vous sera envoyé.

Verify your account

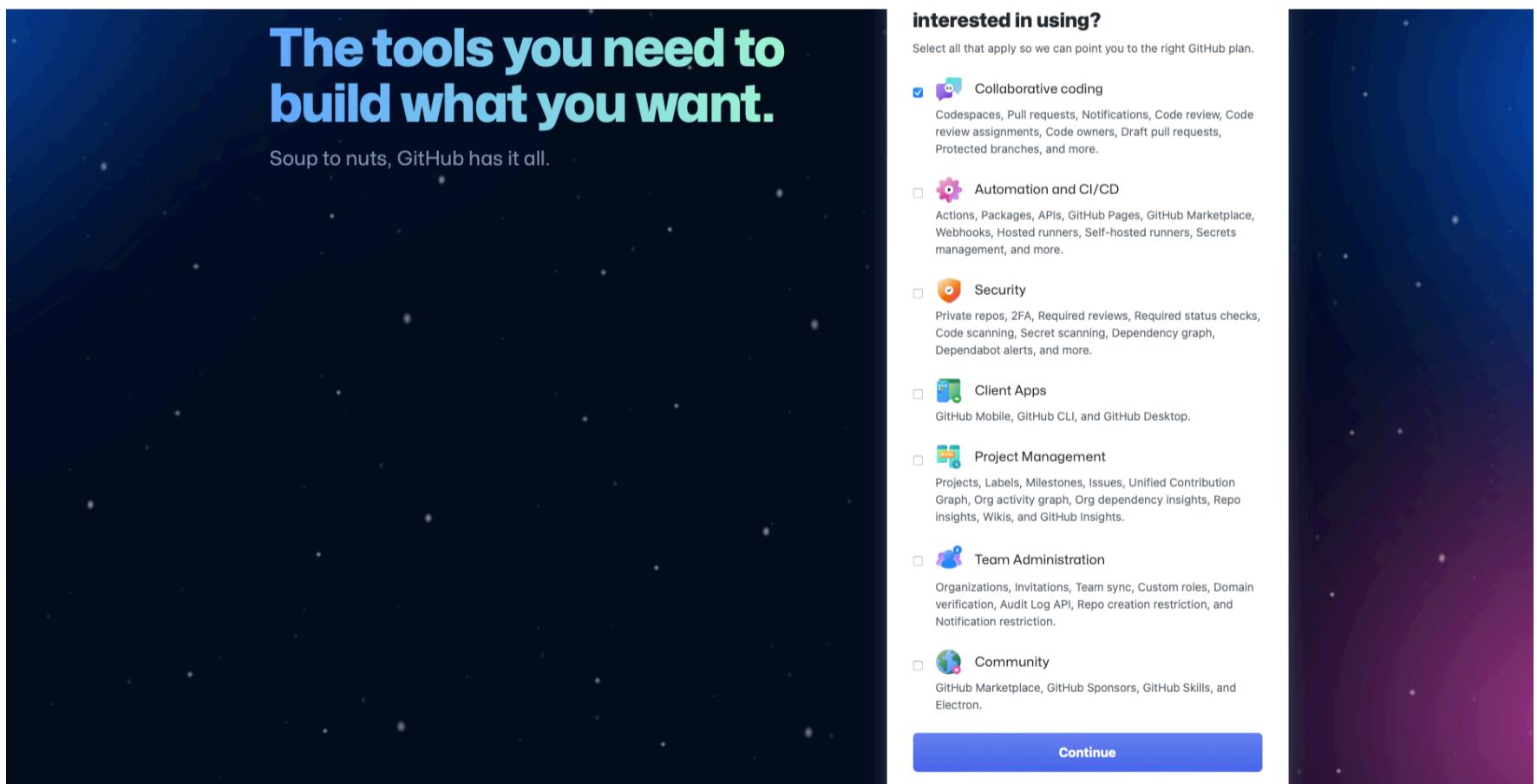
Choisissez une case qui montre deux objets identiques.



Suivront ensuite plusieurs étapes pour déterminer quel projet vous va le mieux



Choisissez just me et student



Puis cochez Collaborative coding. C'est ce que l'on apprend ici. Vous pouvez évidemment cocher d'autres cases qui vous intéresseraient. Cela n'a pas vraiment d'importance, c'est pour déterminer quel plan vous convient le mieux.

Learn to ship software like a pro.

GitHub gives students free access to the best developer tools so they can learn by doing.

The screenshot shows the GitHub pricing page for students. On the left, the 'Free' plan is listed with the following features:

- Unlimited public/private repositories
- 2,000 CI/CD minutes/month (Free for public repositories)
- 500MB of Packages storage (Free for public repositories)
- 120 core-hours of Codespaces compute
- 15GB of Codespaces storage
- Community support

On the right, 'Get additional student benefits' is shown for GitHub Pro, which includes:

- Protect your branches
- Draft pull requests
- Pages and Wikis
- 3,000 CI/CD minutes/month (Free for public repositories)
- 2GB of Packages storage (Free for public repositories)
- 180 core-hours of Codespaces compute
- 20GB of Codespaces storage
- Web-based support

Below these are sections for the GitHub Student Developer Pack and GitHub Campus Expert training.

Dans notre cas nous allons choisir le plan gratuit. Cliquez donc sur Continue for free

The screenshot shows the GitHub dashboard. At the top, there are tabs for 'Following', 'For you' (Beta), and 'Explore'. A central box says 'Discover interesting projects and people to populate your personal news feed.' Below it, there's a 'ProTip!' about the news feed and a link to 'Explore GitHub'. To the right, there's a 'Start coding instantly with GitHub Codespaces' section with a 'Get started' button. At the bottom, there's a 'Latest changes' section listing recent updates from GitHub, such as 'Secret scanning emits audit log events for custom pattern push protection enablement' (12 hours ago) and 'GitHub Issues – January 5th update' (17 hours ago).

Le Dashboard est créé, vous pouvez désormais voir votre profil en cliquant en haut à droite, puis profile.

Vous pouvez ainsi modifier votre profil et y entrer vos données personnelles comme le nom, le lieu ou une photo de profil.

GITHUB DANS LA LIGNE DE COMMANDES

Maintenant que nous avons créé notre compte GitHub est que Git est configuré, il faudrait pouvoir communiquer et s'authentifier avec notre compte directement la ligne de commande, c'est à dire le Terminal.

Alors il existe plusieurs façons de faire: Via un Token, une clé SSH et via le package Github CLI dans la ligne de commandes. Il y a des avantages et des inconvénients avec chaque façon de faire. Vous pouvez évidemment trouver des informations sur internet pour chaque méthode.

Personnellement, je vais vous montrer la méthode avec le CLI pour les raisons suivantes:

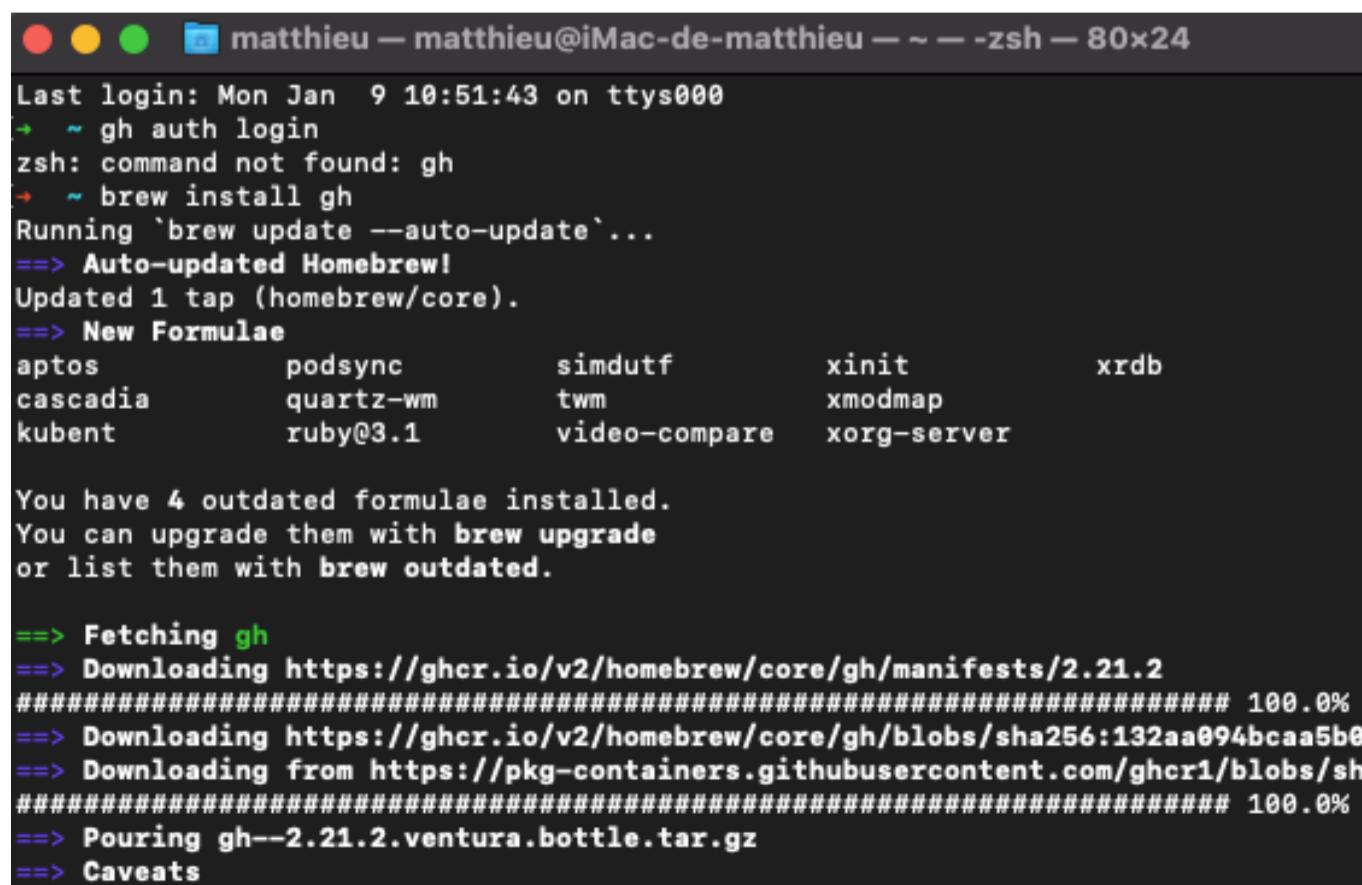
Sur macOS, je ne sais pas si l'erreur est la même sur les autres OS, avec le SSH je reçois l'erreur suivante: **Support for password authentication was removed. Please use a personal access token instead.**

Quand aux Token ce sont des éléments qui ont une durée de validité.

C'est pourquoi je vais passer sur la 3eme solution. Cela aura plus de simplicité d'utilisation.

Nous allons donc nous rendre sur le dépôt GitHub de ce package et l'installer: <https://github.com/cli/cli#installation>. Chaque système d'exploitation possède ses commandes pour installer le package.

Une fois l'installation, effectuée, nous devrons entrer la commande gh auth login



```
matthieu — matthieu@iMac-de-matthieu — ~ — zsh — 80x24
Last login: Mon Jan  9 10:51:43 on ttys000
~ $ gh auth login
zsh: command not found: gh
~ $ brew install gh
Running `brew update --auto-update`...
==> Auto-updated Homebrew!
Updated 1 tap (homebrew/core).
==> New Formulae
aptos      podsnc      simdutf      xinit       xrdb
cascadia   quartz-wm   twm          xmodmap
kubent     ruby@3.1    video-compare xorg-server

You have 4 outdated formulae installed.
You can upgrade them with brew upgrade
or list them with brew outdated.

==> Fetching gh
==> Downloading https://ghcr.io/v2/homebrew/core/gh/manifests/2.21.2
#####
==> Downloading https://ghcr.io/v2/homebrew/core/gh/blobs/sha256:132aa094bc当地 100.0%
==> Downloading from https://pkg-containers.githubusercontent.com/ghcr1/blobs/sh
#####
==> Pouring gh--2.21.2.ventura.bottle.tar.gz
==> Caveats
```

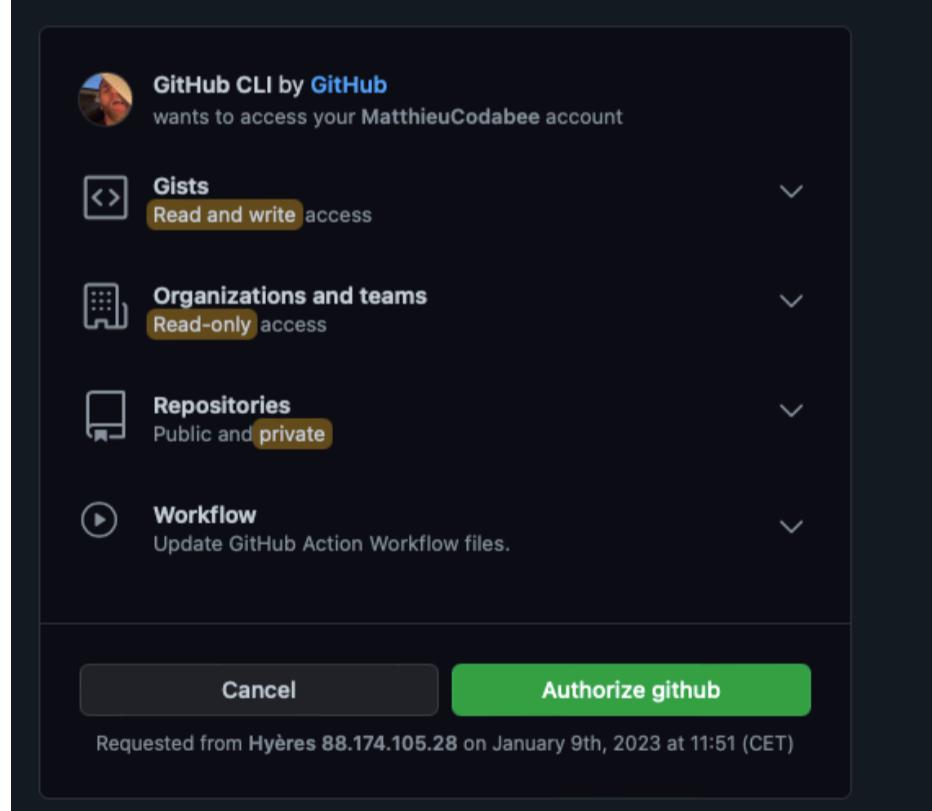
Des informations seront demandées ainsi que l'accès au navigateur pour s'authentifier directement via GitHub.

Vous pouvez d'ailleurs voir un code d'authentification dans la seconde capture d'écran.

matthieu — matthieu@iMac-de-matthieu — ~ — zsh — 80x24

```
==> Pouring gh--2.21.2.ventura.bottle.tar.gz
==> Caveats
zsh completions have been installed to:
  /usr/local/share/zsh/site-functions
==> Summary
🍺 /usr/local/Cellar/gh/2.21.2: 152 files, 40.1MB
==> Running `brew cleanup gh`...
Disable this behaviour by setting HOMEBREW_NO_INSTALL_CLEANUP.
Hide these hints with HOMEBREW_NO_ENV_HINTS (see `man brew`).
→ ~ hit auth login
zsh: command not found: hit
→ ~ gh auth login
? What account do you want to log into? GitHub.com
? What is your preferred protocol for Git operations? HTTPS
? Authenticate Git with your GitHub credentials? Yes
? How would you like to authenticate GitHub CLI? Login with a web browser

! First copy your one-time code: 554D-A62B
Press Enter to open github.com in your browser...
✓ Authentication complete.
- gh config set -h github.com git_protocol https
✓ Configured git protocol
✓ Logged in as MatthieuCodabee
→ ~ █
```



Et voilà ! Tout est prêt, passons aux choses sérieuses !

A noter que si vous voulez passer par une autre méthode libre à vous !

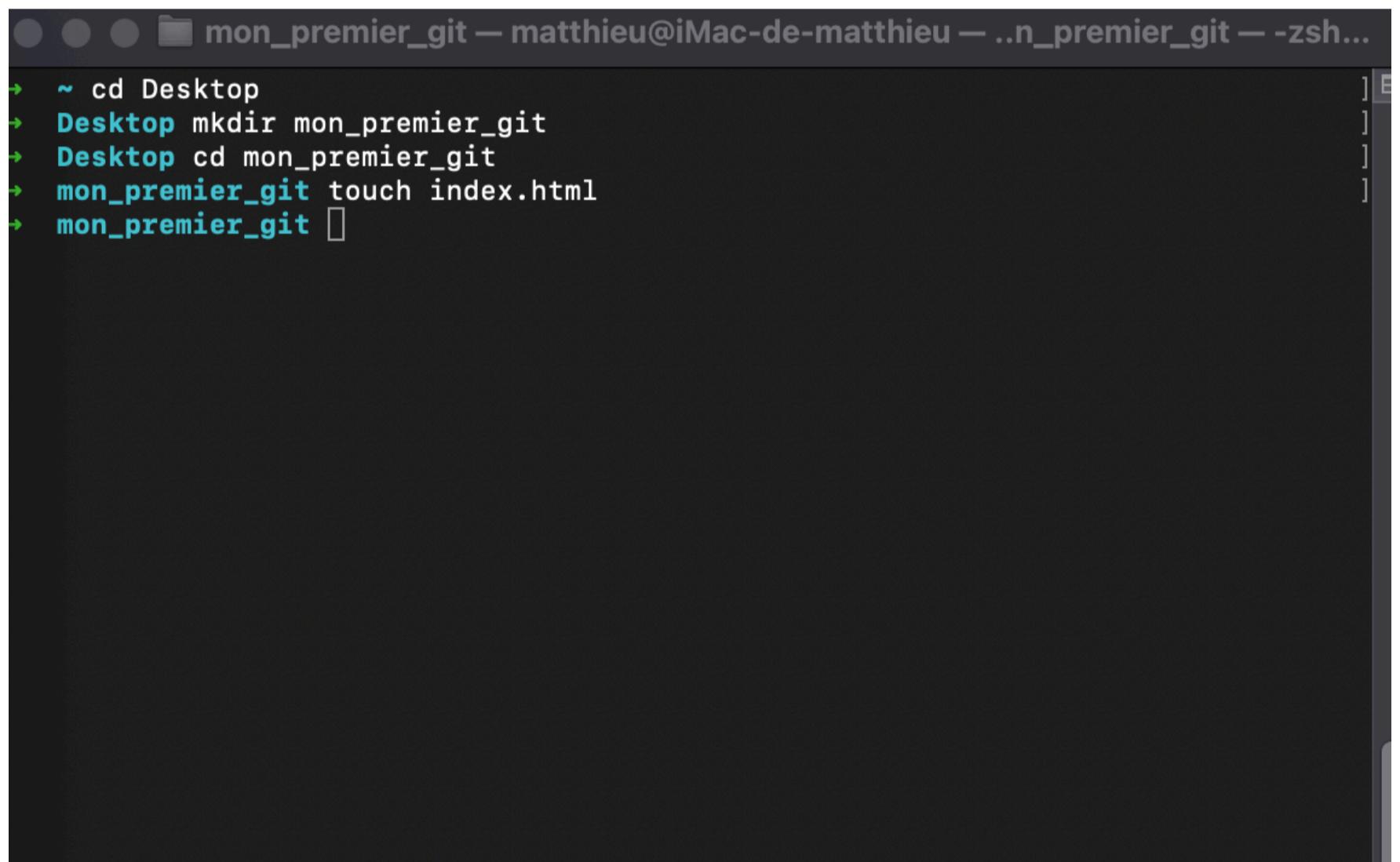
SSH: <https://docs.github.com/fr/authentication/connecting-to-github-with-ssh/adding-a-new-ssh-key-to-your-github-account>

Token: <https://docs.github.com/fr/authentication/keeping-your-account-and-data-secure/creating-a-personal-access-token>

PREMIERS PAS AVEC GIT

Avant de commencer avec Git, nous avons besoin de créer un projet que nous devrons créer. Et si nous en profitons pour utiliser notre Terminal et ainsi créer un projet html? Je tiens à rassurer ceux qui ne connaissent pas html, j'ai choisi ce language (j'entends déjà dire Matthieu HTML n'est pas un language) car c'est peut-être celui que le plus de monde connaît.

Bref, ouvrons notre Terminal et créons un dossier mon_premier_git sur le bureau avec en son sein un fichier html index.html



```
mon_premier_git — matthieu@iMac-de-matthieu — ..n_premier_git — -zsh...
→ ~ cd Desktop
→ Desktop mkdir mon_premier_git
→ Desktop cd mon_premier_git
→ mon_premier_git touch index.html
→ mon_premier_git [ ]
```

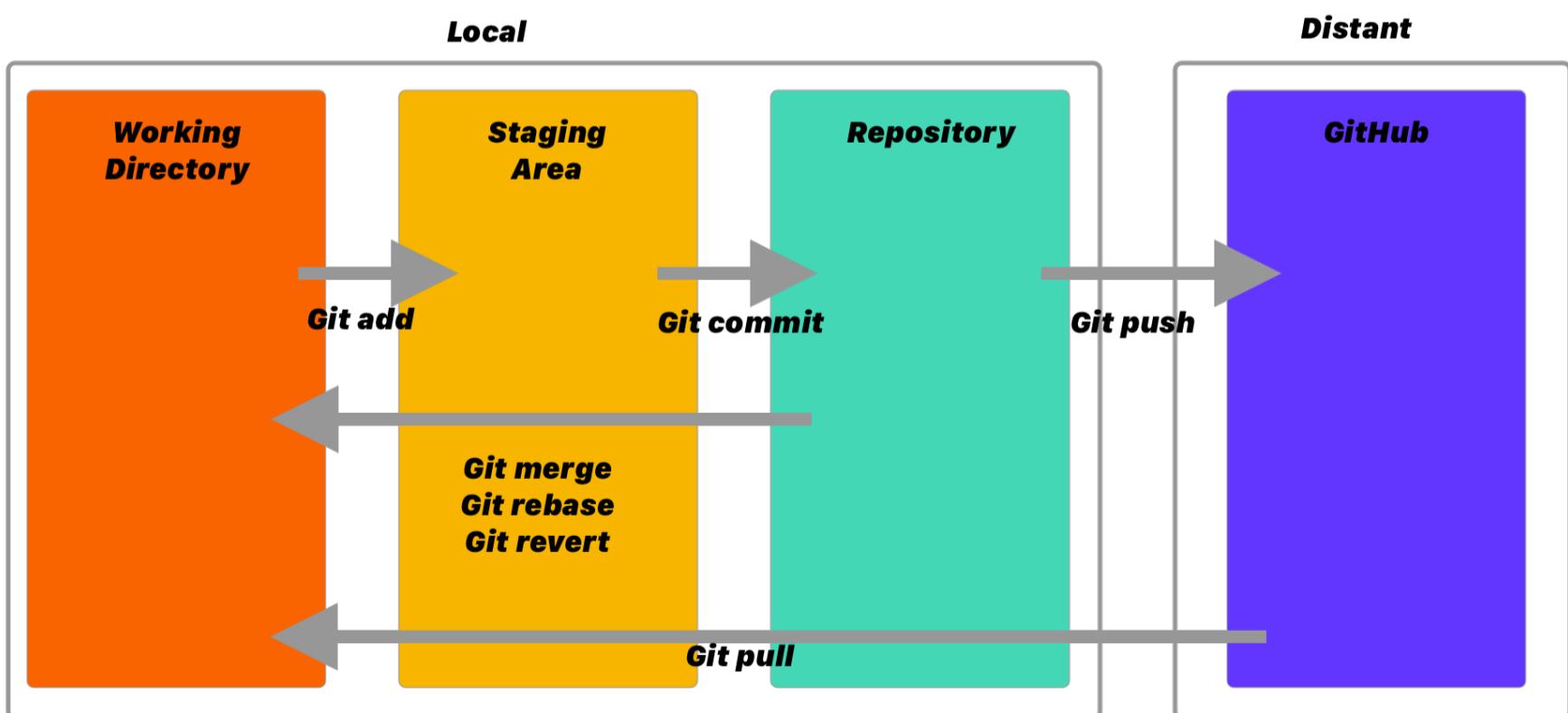
Pour ma part, lors de l'édition du code, j'utiliserai Visual Studio Code. Si vous êtes habitués à utiliser un autre éditeur, n'hésitez pas à choisir celui qui vous convient le mieux. En fait je choisis Visual Studio Code car il y a un plugin html preview qui permet de voir le code en temps réel sur une page web et je trouve cela plutôt pratique. Je resterai évidemment sur le Terminal pour les actions avec Git.

LES DIFFÉRENTS ÉTATS DE GIT

Lors d'un contrôle de versions, nous serons amenés à passer par plusieurs états:

- Le Working directory: C'est le dossier sur lequel on travaille. Il est en local, directement sur la machine. Le dossier `mon_premier_git` que l'on a créé juste avant par exemple correspond à ce Working directory
- Le Staging Area: Il peut aussi être appelé index. Il est l'emplacement de représentation des fichiers et dossiers modifiés qui sont amenés à être ajoutés au projet. Ces fichiers passent dans le Staging Area avec la commande `git add`
- Le Repository: Notre fameux dépôt. Une fois la commande `git commit` effectuée, nous envoyons une nouvelle version à notre dépôt.
- Le distant est aussi un repository, ici je l'ai nommé GitHub pour bien faire la distinction entre notre dépôt local et distant. On utilisera `git push` et `git pull` pour pousser un dépôt vers le distant depuis le local ou tirer (obtenir) un dépôt distant vers le local.

Type Les Différents états de Git



INITIALISER GIT POUR UN PROJET

Désormais nous voyons comment fonctionne git, ses différentes étapes via les états. Il est donc temps de reprendre notre dossier et d'y initialiser Git.

Pour initialisé git, rien de plus simple:

1. Rendez-vous dans le dossier que vous souhaiter rendre versionnable
2. Entrez la commande git init dans le terminal

```
[→ mon_premier_git git init
astuce: Utilisation de 'master' comme nom de la branche initiale. Le nom de la b
ranche
astuce: par défaut peut changer. Pour configurer le nom de la branche initiale
astuce: pour tous les nouveaux dépôts, et supprimer cet avertissement, lancez :
astuce:
astuce:      git config --global init.defaultBranch <nom>
astuce:
astuce: Les noms les plus utilisés à la place de 'master' sont 'main', 'trunk' e
t
astuce: 'development'. La branche nouvellement créée peut être renommée avec :
astuce:
astuce:      git branch -m <nom>
Dépôt Git vide initialisé dans /Users/matthieu/Desktop/mon_premier_git/.git/]
```

3. Un petit ls -a pour voir ce qui a été ajouté !

```
[→ mon_premier_git git:(master) ✘ ls -a
.          ..          .git        index.html
```

Remarquez ici que un dossier caché a été ajouté. C'est un dossier git. C'est donc un succès. Nous avons bien ajouté Git à notre projet.

MON PREMIER COMMIT

Il est désormais temps de faire notre Commit et de passer à travers les états locaux de Git.
Pour ceci, nous allons procéder par plusieurs étapes:

1. Vérifier le statut des modifications avec git status
2. Ajouter les fichiers que l'on veut faire passer vers la staging Area avec git add suivi soit du nom du fichier, soit de -A pour désigner tous les fichiers.
3. Faire un commit pour passer au repository et ajouter les modifications à l'historique. Pour ceci nous ferons git commit -m suivi entre guillemets du nom du commit. De manière générale, le premier commit se nomme toujours first commit.
4. Optionnellement, on peut faire un git status à chaque étape pour voir si tout a bien été ajouté.

```
→ mon_premier_git git:(master) ✘ git status
Sur la branche master

Aucun commit

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    index.html

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (utilisez
"git add" pour les suivre)
→ mon_premier_git git:(master) ✘ git add index.html
→ mon_premier_git git:(master) ✘ git status
Sur la branche master

Aucun commit

Modifications qui seront validées :
  (utilisez "git rm --cached <fichier>..." pour désindexer)
    nouveau fichier : index.html

→ mon_premier_git git:(master) ✘ git commit -m "first Commit"
[master (commit racine) 8636c7d] first Commit
  1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 index.html
→ mon_premier_git git:(master) git status
Sur la branche master
rien à valider, la copie de travail est propre
→ mon_premier_git git:(master)
```

5. Ensuite faire un git log. Appuyez sur Q pour sortir du git log

```
commit 8636c7d3192d03a56f95e916b1f818d05b37e3e8 (HEAD -> master)
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:00:42 2023 +0100

  first Commit
(END)
```

Que pouvons nous noter ici?

- A chaque étape grâce au git status les fichiers ont changé de couleur.
- Les fichiers à ajouter vers le staging sont rouge, une fois ajoutés, ils sont verts et indexés. On peut voir écrit pour les désindexer. C'est pourquoi le staging est aussi appelé index.
- Une fois le commit fait, nous voyons qu'il est écrit rien à valider, le travail est propre.
- Dans notre log, nous voyons le numéro du commit, qui nous sera utile pour revenir en arrière, le nom de l'auteur, la date ainsi que le nom first commit.

Il y a aussi la notation « master » qui apparaît à plusieurs reprises. Il s'agit ici de la branche utilisée. Par défaut, la branche principale est la branche master. Mais nous y reviendrons évidemment plus tard lorsque nous devrons changer de branches.

Ca y est vous avez réussi votre premier commit. Vous avez versionné pour la première fois votre code. Vous pouvez vous féliciter car la première étape dans la maîtrise de Git est acquise.

Mais comme tout projet, il va y avoir plusieurs « versions » et de nombreux ajouts dans le code. C'est pourquoi dès la prochaine étape, nous allons ajouter du code dans le html, pourquoi pas d'autres fichiers et réaliser ainsi plusieurs commits.

SECOND COMMIT

Nous allons commencer doucement dans notre projet et y insérer le tout début d'un code Html, à savoir le head.

Ps: Vous pouvez noter que Visual Studio Code dialogue directement avec Git. Mais c'est une autre histoire.

```
1 <!DOCTYPE html>
2 <html lang="fr">
3   <head>
4     <meta charset="utf-8">
5     <meta name="viewport" content="width=device-width">
6     <title>Mon super projet</title>
7   </head>
8 </html>      You, il y a 1 seconde • Uncommitted changes
```

Retournons donc dans le terminal et créons un nouveau commit avec comme nom ajout du head. Essayez de le faire par vous même !

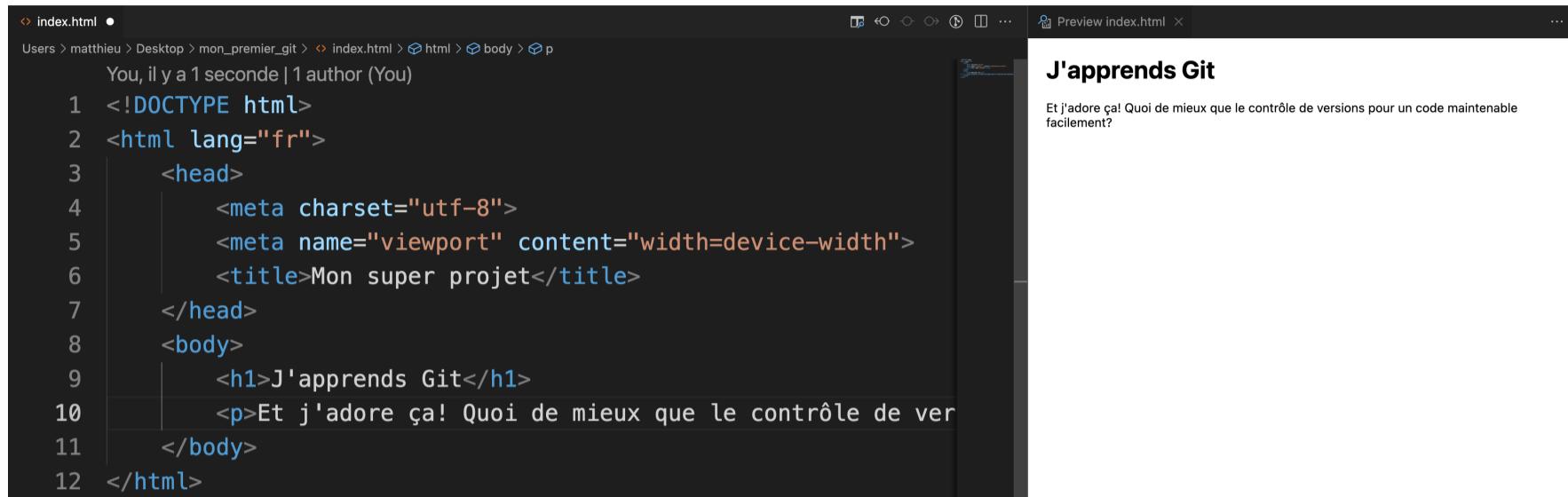
```
[→ mon_premier_git git:(master) git status
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :          index.html

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
[→ mon_premier_git git:(master) ✘ git add -A
[→ mon_premier_git git:(master) ✘ git status
Sur la branche master
Modifications qui seront validées :
  (utilisez "git restore --staged <fichier>..." pour désindexer)
    modifié :          index.html

[→ mon_premier_git git:(master) ✘ git commit -m "ajout du head"
[master 4ddb9bb] ajout du head
  1 file changed, 8 insertions(+)
[→ mon_premier_git git:(master) git status
Sur la branche master
rien à valider, la copie de travail est propre
[→ mon_premier_git git:(master) ]
```

Facile? Ou avez-vous eu besoin de revoir les étapes?

Allez, ajoutons maintenant un titre dans notre body du html pour pouvoir à nouveau faire un commit.



The screenshot shows a code editor with the file 'index.html' open. The code contains a title and some explanatory text. On the right, there is a preview window showing the rendered HTML with the title 'J'apprends Git' and the text 'Et j'adore ça! Quoi de mieux que le contrôle de versions pour un code maintenable facilement?'. The status bar at the bottom indicates 'You, il y a 1 seconde | 1 author (You)'.

```
You, il y a 1 seconde | 1 author (You)
1  <!DOCTYPE html>
2  <html lang="fr">
3      <head>
4          <meta charset="utf-8">
5          <meta name="viewport" content="width=device-width">
6          <title>Mon super projet</title>
7      </head>
8      <body>
9          <h1>J'apprends Git</h1>
10         <p>Et j'adore ça! Quoi de mieux que le contrôle de ver
11     </body>
12 </html>
```

En version sans status cette fois

```
[→ mon_premier_git git:(master) ✘ git add -A
[→ mon_premier_git git:(master) ✘ git commit -m "ajout d'un titre et de texte"
[master 13bf0c3] ajout d'un titre et de texte
 1 file changed, 4 insertions(+)
→ mon_premier_git git:(master) ]
```

Et voici notre historique avec le git log

```
commit 13bf0c37a96a7f24afae8f652fa18ba8f1ac43f0 (HEAD -> master)
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:33:09 2023 +0100

    ajout d'un titre et de texte

commit 4ddb9bbe10768aff23967d6ca22d1cb2f7c61463
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:26:07 2023 +0100

    ajout du head

commit 8636c7d3192d03a56f95e916b1f818d05b37e3e8
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:00:42 2023 +0100

    first Commit
(END)
```

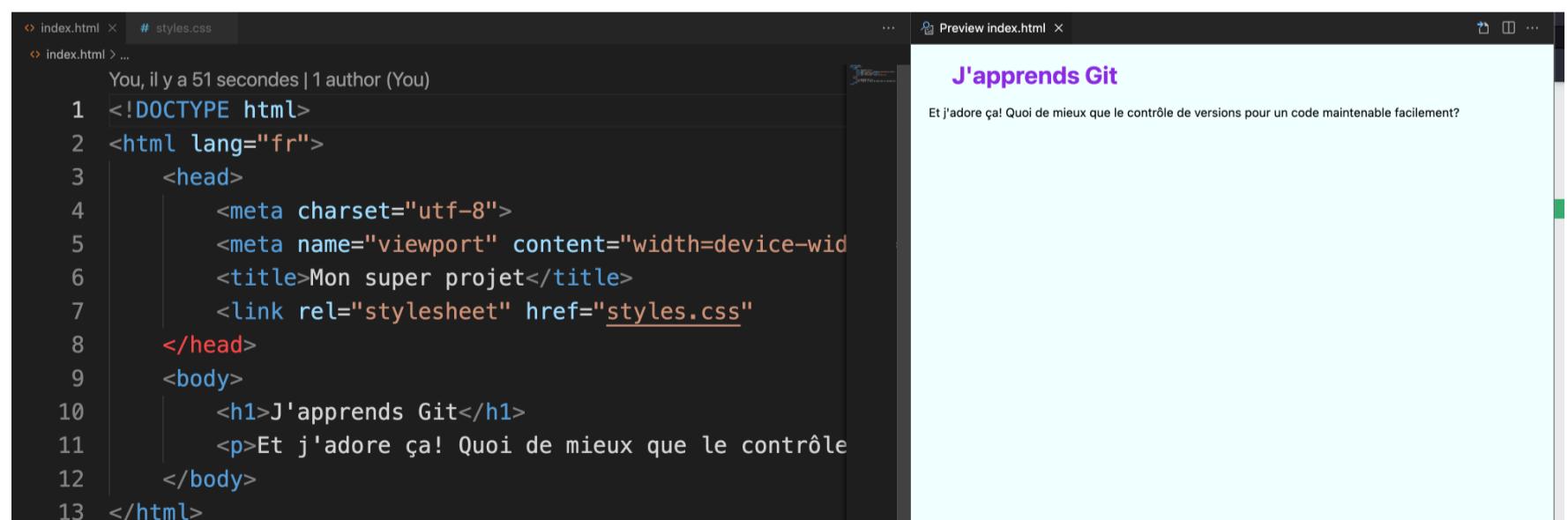
Mais qu'en est il pour l'ajout de nouveaux fichiers dans notre dossier? On va ajouter un fichier CSS !

Première étape, créer un fichier css. Nous le nommerons styles.css Avec le terminal c'est hyper simple, on entre : touch styles.css.

Ensuite dans mon éditeur de texte, j'ouvre ce fichier et j'y ajoute ce code:

```
1 body {  
2     background-color: #azure;  
3 }  
4 h1 {  
5     color: #blueviolet;  
6     margin-left: 25px;  
7 }
```

Puis je lie mon fichier CSS à mon fichier html. Ok, ce sont vraiment des bases de HTML et CSS, mais ce n'est pas le sujet de cet apprentissage et je souhaite rendre ceci accessible à tous.



The screenshot shows a code editor with two tabs: 'index.html' and '# styles.css'. The 'index.html' tab contains the following HTML code:

```
You, il y a 51 secondes | 1 author (You)  
1 <!DOCTYPE html>  
2 <html lang="fr">  
3     <head>  
4         <meta charset="utf-8">  
5         <meta name="viewport" content="width=device-width, height=device-height, initial-scale=1, shrink-to-fit=no">  
6         <title>Mon super projet</title>  
7         <link rel="stylesheet" href="styles.css">  
8     </head>  
9     <body>  
10        <h1>J'apprends Git</h1>  
11        <p>Et j'adore ça! Quoi de mieux que le contrôle de versions pour un code maintenable facilement?  
12    </body>  
13 </html>
```

The '# styles.css' tab contains the CSS code shown earlier in the slide.

The preview window shows the rendered HTML with the heading 'J'apprends Git' and the explanatory text below it.

Maintenant revenons à notre Git et faisons un git status:

```
[→ mon_premier_git git:(master) ✘ git status
Sur la branche master
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :           index.html

Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    styles.css

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
→ mon_premier_git git:(master) ✘ ]
```

Nous pouvons voir clairement qu'il a suivi des modifications dans le fichier html mais la création d'un fichier non suivi qui est notre css.

Ajoutons tous ceci. Et voila! Vous avez pu suivre un nouveau fichier dans votre code.

```
[→ mon_premier_git git:(master) ✘ git add -A
[→ mon_premier_git git:(master) ✘ git commit -m "ajout de styles"
[master 45202cd] ajout de styles
  2 files changed, 8 insertions(+)
  create mode 100644 styles.css ]
```

Afin de maîtriser au mieux ces bases, je vous invite à faire autant de commits que vous voulez. Continuez tant que ces quelques commandes ne vous sont pas totalement familières.

GITIGNORE

Il arrivera certainement à un moment ou un autre de votre développement que vous ayez un ou plusieurs fichiers que vous ne souhaitez pas ajouter à git. Prenons par exemple une application utilisant une API. Cette dernière possède une clé que vous ne souhaitez pas partager et garder uniquement pour vous, hors de git.

Pour ceci il existe le gitignore.

Commençons par créez un fichier texte avec votre clé api. Nous allons l'appeler key.txt

```
[→ mon_premier_git git:(master) touch key.txt]
```

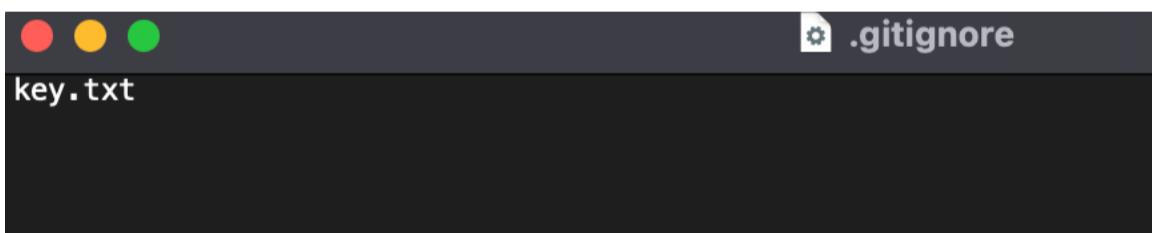
Lors d'un git status elle apparaitra ! Et elle apparaitra tant que vous ne l'aurez pas ajouté. L'exclure à chaque commit peut être un peu contraignant.

```
[→ mon_premier_git git:(master) × git status
Sur la branche master
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    key.txt

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (utilisez
"git add" pour les suivre)
→ mon_premier_git git:(master) × ]
```

Nous allons donc créer un fichier .gitignore et y ajouter à l'intérieur le nom de notre fichier:

```
[→ mon_premier_git git:(master) × touch .gitignore
[→ mon_premier_git git:(master) × open .gitignore
[→ mon_premier_git git:(master) × ]
```



A noter la commande open pour ouvrir le fichier en question et y ajouter les modifications.

Dès lors si je fais un git status, le fichier key.txt est ignoré.

Je pense tout de même à faire un commit pour que nous puissions garder une trace dans l'historique de ce fameux .gitignore.

```
[→ mon_premier_git git:(master) ✘ git status
Sur la branche master
Fichiers non suivis:
  (utilisez "git add <fichier>..." pour inclure dans ce qui sera validé)
    .gitignore

aucune modification ajoutée à la validation mais des fichiers non suivis sont présents (utilisez
"git add" pour les suivre)
[→ mon_premier_git git:(master) ✘ git add .gitignore
[→ mon_premier_git git:(master) ✘ git commit -m "gitignore"
[master 899b8f6] gitignore
  1 file changed, 1 insertion(+)
  create mode 100644 .gitignore
[→ mon_premier_git git:(master) git status
Sur la branche master
rien à valider, la copie de travail est propre
→ mon_premier_git git:(master) ]
```

Nous avons vu ici comment ignorer un fichier simple en donnant son nom. Mais nous pouvons aussi ignorer tous les fichiers d'une même extension:

Par exemple tous les fichiers texte: `*.txt`

Nous pouvons ignorer les fichiers textes d'un répertoire enfant:

`monDossierEnfant/**/*.txt`

Nous pouvons contourner la règle en disant que l'on veut ignorer tous les fichiers enfants texte sauf certains

`monDossierEnfant/**/*.txt`

`!monDossierEnfant/nomDuFichier.txt`

Voici donc pour ignorer un ou plusieurs fichiers. Maintenant voyons comment revenir en arrière dans notre historique?

REVENIR EN ARRIÈRE

Comme je vous l'ai expliqué à plusieurs reprises, git nous offre la possibilité de revenir en arrière dans notre historique. C'est en effet une des raisons pour lesquelles vous avez voulu apprendre git non?

Alors pour ceci il existe plusieurs façons de faire et plusieurs méthodes qui auront chacune un effet différent sur notre code:

- Checkout
- Revert
- Reset

Checkout

Cette commande possède plusieurs fonctions:

- Revenir sur un commit
- Revenir sur un fichier spécifique par rapport à un commit
- Changer de branche

Nous allons ici voir comment utiliser checkout pour revenir sur un commit. Lorsque nous faisons git checkout « numéro du commit », cela transforme tous les fichiers pour reproduire l'état du commit demandé en Working Directory. Cette commande nous déplace de master vers detached head. C'est à dire que vous revenez mais en tant que simple observateur. Vous voyez donc le projet dans l'état dans lequel il était au moment voulu. Vous avez la possibilité de voir ainsi les vieux commits et revenir à l'instant présent à tout moment.

Voyons cela en pratique, nous allons choisir un numéro de commit parmi notre log et l'observer:

Je vais choisir le commit ajout d'un titre et de texte.

```
→ mon_premier_git git:(master) git log  
→ mon_premier_git git:(master) |
```

```

commit 899b8f6f082bc3272523fc981ce15184fe1b0580 (HEAD -> master)
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 15:00:34 2023 +0100

gitignore

commit 45202cde0d955db7fdeb543ddb9e730ee2f7e3af
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:46:19 2023 +0100

ajout de styles

commit 13bf0c37a96a7f24afae8f652fa18ba8f1ac43f0
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:33:09 2023 +0100

ajout d'un titre et de texte

commit 4ddb9bbe10768aff23967d6ca22d1cb2f7c61463
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:26:07 2023 +0100

ajout du head

commit 8636c7d3192d03a56f95e916b1f818d05b37e3e8
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:00:42 2023 +0100

first Commit
(END) []

```

```

[→ mon_premier_git git:(master) git checkout 13bf0c37a96a7f24afae8f652fa18ba8f1ac43f0
Note : basculement sur '13bf0c37a96a7f24afae8f652fa18ba8f1ac43f0'.

Vous êtes dans l'état « HEAD détachée ». Vous pouvez visiter, faire des modifications expérimentales et les valider. Il vous suffit de faire un autre basculement pour abandonner les commits que vous faites dans cet état sans impacter les autres branches

Si vous voulez créer une nouvelle branche pour conserver les commits que vous créez, il vous suffit d'utiliser l'option -c de la commande switch comme ceci :

git switch -c <nom-de-la-nouvelle-branche>

Ou annuler cette opération avec :

git switch -

Désactivez ce conseil en renseignant la variable de configuration advice.detachedHead à false

HEAD est maintenant sur 13bf0c3 ajout d'un titre et de texte
[→ mon_premier_git git:(13bf0c3) × ]

```

Voila ce que nous obtenons

Et dans notre projet ça donne quoi?

```
You, il y a 1 heure | 1 author (You)
1 <!DOCTYPE html>
2 <html lang="fr">
3     <head>
4         <meta charset="utf-8">
5         <meta name="viewport" c
6             <title>Mon super projet
7         </head>
8         <body>
9             <h1>J'apprends Git</h1>
10            <p>Et j'adore ça! Quoi
11        </body>
12    </html>
```

Le style a disparu ! Vous pouvez même remarquer que le styles.css a été supprimé !

Maintenant revenons au présent, ou plus précisément sur notre branche principale en faisant git checkout master. Et comme par magie, mon projet revient à la normale.

```
[→ mon_premier_git git:(13bf0c3) ✘ git checkout master
La position précédente de HEAD était sur 13bf0c3 ajout d'un titre et de texte
Basculement sur la branche 'master'
→ mon_premier_git git:(master) ]
```

Revert.

Ici nous allons inverser un commit. Prenez le commit de votre choix et inversez le.

Attention ! **Ne revenez pas ensuite sur vos pas avec un revert**. Le revert fusionne les changements automatiquement sur un même fichier! Nous utiliserons la dernière méthode pour annuler le revert.

```
[→ mon_premier_git git:(master) git revert 13bf0c37a96a7f24afae8f652fa18ba8f1ac43f0 ]
```

Vous pouvez noter que une fois la commande revert appelée vous aurez une fenêtre Vim qui s'ouvre pour noter des changements. Quelques notions de Vim:

- I pour commencer à écrire
- Escape pour finir l'écriture

- :x pour quitter et sauvegarder

Vous êtes revenu à un point précis de l'historique.

Vous n'avez plus le style sur votre projet.

Et avec un git log?

```
Revert "ajout d'un titre et de texte"

This reverts commit 13bf0c37a96a7f24afae8f652fa18ba8f1ac43f0.

# Veuillez saisir le message de validation pour vos modifications. Les lignes
# commençant par '#' seront ignorées, et un message vide abandonne la validation.
#
# Sur la branche master
# Modifications qui seront validées :
#     modifié :           index.html
#
```

```
commit 569685bfd798f247bf785985605b49e24a18f81d (HEAD -> master)
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 16:45:48 2023 +0100

    Revert "ajout d'un titre et de texte"

    This reverts commit 13bf0c37a96a7f24afae8f652fa18ba8f1ac43f0.

commit d8d08aa0e53181bd7e7ffd962d23c9916395bed2
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 16:41:12 2023 +0100

    gitignore

commit 45202cde0d955db7fdeb543ddb9e730ee2f7e3af
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:46:19 2023 +0100

    ajout de styles

commit 13bf0c37a96a7f24afae8f652fa18ba8f1ac43f0
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:33:09 2023 +0100

    ajout d'un titre et de texte

commit 4ddb9bbe10768aff23967d6ca22d1cb2f7c61463
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:26:07 2023 +0100

    ajout du head

commit 8636c7d3192d03a56f95e916b1f818d05b37e3e8
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:00:42 2023 +0100

    first Commit
(END)
```

Reset

Le reset a pour fonction de supprimer TOUT les commits précédents. Cette action ne se fait que lorsque vous

Êtes totalement certains de vouloir supprimer ce que vous venez de faire!

Par exemple ici nous voulons supprimer le revert:

Nous prenons donc le dernier commit que nous voulons avoir dans l'historique et nous faisons un git reset:

```
[→ mon_premier_git git:(master) git reset d8d08aa0e53181bd7e7ffd962d23c9916395be]
d2
Modifications non indexées après reset :
M      index.html
```

Voilà ce que nous dit désormais le git log: Le git reset a disparu !

```
commit d8d08aa0e53181bd7e7ffd962d23c9916395bed2 (HEAD -> master)
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 16:41:12 2023 +0100

    gitignore

commit 45202cde0d955db7fdeb543ddb9e730ee2f7e3af
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:46:19 2023 +0100

    ajout de styles

commit 13bf0c37a96a7f24afae8f652fa18ba8f1ac43f0
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:33:09 2023 +0100

    ajout d'un titre et de texte

commit 4ddb9bbe10768aff23967d6ca22d1cb2f7c61463
Author: = <matthieu@codabee.com>
Date:   Fri Jan 6 14:26:07 2023 +0100

    ajout du head
:
```

Et si nous faisons un git status?

Nous voyons que le index.html a changé !

Pour voir les différences, nous allons ici faire un git diff:

```
diff --git a/index.html b/index.html
index 2721b4d..2c2a3f0 100644
--- a/index.html
+++ b/index.html
@@ -6,8 +6,4 @@
     <title>Mon super projet</title>
     <link rel="stylesheet" href="styles.css">
 </head>
- <body>
-     <h1>J'apprends Git</h1>
-     <p>Et j'adore ça! Quoi de mieux que le contrôle de versions pour un cod
e maintenable facilement?</p>
- </body>
</html>
\ No newline at end of file
(END)
```

Cet différence est due au fait que le git revert avait modifié le fichier html.

Etant donné que nous ne voulons pas que ces modifier ceci, nous allons utiliser le git Restore pour ne pas voir ces modifications (dans notre cas suppression) apparaitre.

```
→ mon_premier_git git:(master) ✘ git restore index.html
→ mon_premier_git git:(master) █
```

Et voila!

CONCLUSION

Dans ce chapitre, nous avons pu toucher aux bases de Git et effectuer nos premiers commits. Nous avons ici pu travailler en local et nous amuser à faire plusieurs modifications. Vous êtes désormais prêts à utiliser git en solo et en local sur votre machine. Vous pouvez vous féliciter car c'est une grosse avancée dans l'apprentissage !

Voici un petit récapitulatif de ce que nous avons appris, et des fonctionnalités de base:

- Git init pour initialiser un dépôt
- Git statut pour voir le statut des changements du dépôt
- Git add pour ajouter des fichiers à indexer
- Git commit pour pousser une modification du staging vers le repo.
- Git log pour voir l'historique
- Git checkout pour voir en tant qu'observateur l'historique
- Git revert pour inverser un commit
- Git reset pour annuler un commit
- Git diff pour visualiser les différences
- Git restore pour restaurer un fichier.
- Le fichier .gitignore

Cela fait pas mal de choses non?

Maintenant on va retourner sur GitHub pour travailler avec un dépôt distant !

GITHUB

Maintenant que nous connaissons les bases de Git, il est temps de passer en distant avec GitHub !

Nous allons donc voir pas à pas:

- Créer un dépôt distant et y ajouter notre code local
- Créer un dépôt distant et tirer le code distant vers le local
- Faire des commîts sur GitHub
- Récupérer les modifications

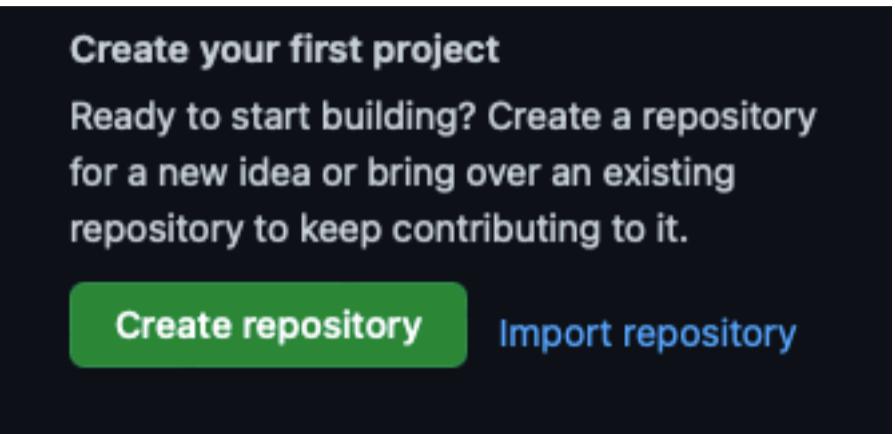
Et bien d'autres choses.

C'est parti!

AJOUT DE NOTRE DÉPÔT A GITHUB

Il est désormais temps de créer notre premier dépôt. Si aucun n'est déjà ajouté. Et cela devrait être le cas, nous avons ici un bouton pour créer un repository.

Nous allons donc cliquer dessus.



S'ouvre ensuite une seconde page où nous devrons entrer les informations essentielles pour la création d'un dépôt:

- Owner: Le propriétaire du dossier. C'est vous
- Repository name: Le nom du dépôt
- Public ou private. Nous allons choisir public pour qu'il soit accessible à tous
- Add a readme. Nous n'allons pas l'ajouter, nous le ferons plus tard
- Add .gitignore. Nous l'avons déjà dans notre projet
- Choose a licence. Selon votre projet, une license d'utilisation conviendra, pour tester, nous allons choisir none.

Nous pouvons désormais créer notre dépôt distant.

Quick setup — if you've done this kind of thing before

[Set up in Desktop] or [HTTPS] [SSH] https://github.com/MatthieuCodabee/mon_premier_git.git []

Get started by creating a new file or uploading an existing file. We recommend every repository include a `README`, `LICENSE`, and `.gitignore`.

...or create a new repository on the command line

```
echo "# mon_premier_git" >> README.md  
git init  
git add README.md  
git commit -m "first commit"  
git branch -M main  
git remote add origin https://github.com/MatthieuCodabee/mon_premier_git.git  
git push -u origin main
```

...or push an existing repository from the command line

```
git remote add origin https://github.com/MatthieuCodabee/mon_premier_git.git  
git branch -M main  
git push -u origin main
```

...or import code from another repository

You can initialize this repository with code from a Subversion, Mercurial, or TFS project.

[Import code](#) []

Une fois ce dépôt distant créé, nous avons tout d'abord le lien vers notre dépôt puis plusieurs possibilités:

Créer un dépôt depuis le terminal, pousser un dépôt existant, ou importer depuis un autre dépôt distant. Comme nous avons créé notre dépôt en local, nous allons pouvoir l'ajouter.

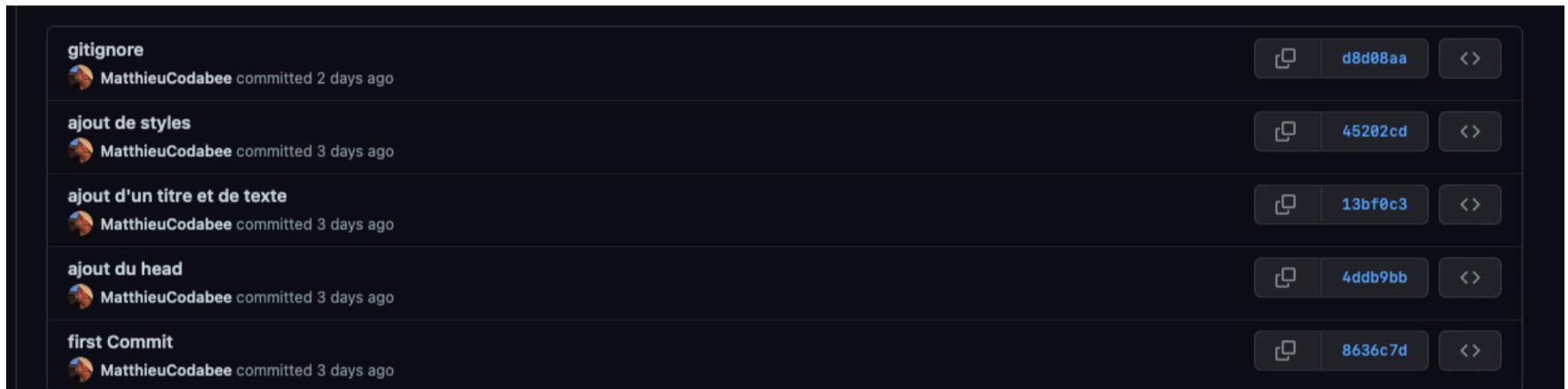
```
[→ mon_premier_git git:(master) git remote add origin https://github.com/MatthieuCodabee/mon_premier_git.git  
[→ mon_premier_git git:(master) git branch -M main  
[→ mon_premier_git git:(main) git push -u origin main
```

Les étapes sont notées une par une

La première ligne signifie que nous ajoutons depuis le dépôt distant (remote) une origine qui est le lien vers notre dépôt.

```
→ mon_premier_git git:(main) git push -u origin main
[Énumération des objets: 17, fait.
Décompte des objets: 100% (17/17), fait.
Compression par delta en utilisant jusqu'à 6 fils d'exécution
Compression des objets: 100% (12/12), fait.
Écriture des objets: 100% (17/17), 1.50 Kio | 768.00 Kio/s, fait.
Total 17 (delta 3), réutilisés 0 (delta 0), réutilisés du pack 0
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/MatthieuCodabee/mon_premier_git.git
 * [new branch]      main -> main
la branche 'main' est paramétrée pour suivre 'origin/main'.
```

La seconde: Nous allons nous déplacer vers la branche Main. La branche principale. A notre disposition que la branche principale est Main sur GitHub mais Master sur Git



Notre dépôt local est donc copié sur GitHub en distant.

```
[→ mon_premier_git git:(main) git status
Sur la branche main
Votre branche est à jour avec 'origin/main'.

Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :       styles.css

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
[→ mon_premier_git git:(main) ✘ git add styles.css
[→ mon_premier_git git:(main) ✘ git commit -m "ajout de couleur pour le texte"
[main 4045a68] ajout de couleur pour le texte
  1 file changed, 4 insertions(+)
[→ mon_premier_git git:(main) git push -u origin main
Énumération des objets: 5, fait.
Décompte des objets: 100% (5/5), fait.
Compression par delta en utilisant jusqu'à 6 fils d'exécution
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (3/3), 313 octets | 313.00 Kio/s, fait.
Total 3 (delta 2), réutilisés 0 (delta 0), réutilisés du pack 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/MatthieuCodabee/mon_premier_git.git
  d8d08aa..4045a68  main -> main
la branche 'main' est paramétrée pour suivre 'origin/main'.
→ mon_premier_git git:(main)
```

Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? [Import a repository](#).

Owner *



MatthieuCodabee

Repository name *

mon_premier_git



Great repository names are short and memorable. Need inspiration? How about [fictional-barnacle](#)?

Description (optional)

Public

Anyone on the internet can see this repository. You choose who can commit.

Private

You choose who can see and commit to this repository.

Initialize this repository with:

Skip this step if you're importing an existing repository.

Add a README file

This is where you can write a long description for your project. [Learn more](#).

Add .gitignore

Choose which files not to track from a list of templates. [Learn more](#).

.gitignore template: None

Choose a license

A license tells others what they can and can't do with your code. [Learn more](#).

License: None

You are creating a public repository in your personal account.

Create repository

Maintenant si nous faisons un rafraîchissement de la page web GitHub notre projet est bel est bien présent, avec l'historique des commits !

Maintenant, dès lors que vous devrez pousser un commit vers GitHub, vous taperez dans le terminal git pus -u origin main pour pousser sur la branche principale. Allez modifiez votre code et faites un push

AJOUTER UN README

Lorsque nous travaillons avec GitHub, il est fortement conseillé d'ajouter un fichier Readme.md.

Ce fichier Readme.md est un fichier au format Markdown, il permettra à tous ceux qui vont voir votre dépôt d'avoir en quelque sorte une notice d'utilisation, une page d'accueil ou encore une présentation du produit.

Ici nous allons tout d'abord l'ajouter en local pour le pousser vers GitHub.

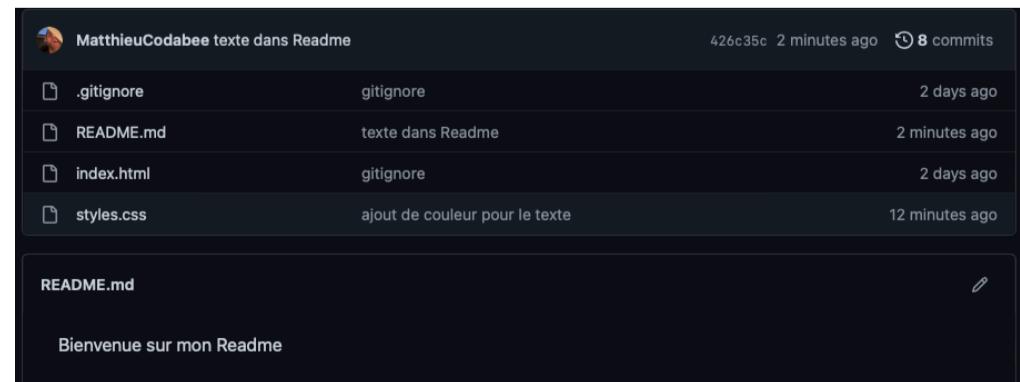
```
[→ mon_premier_git git:(main) touch README.md]
```

Puis, nous allons faire un commit et pousser tout ceci vers GitHub

```
[→ mon_premier_git git:(main) ✘ git add -A
[→ mon_premier_git git:(main) ✘ git commit -m "ajout du Readme"
[main 7603f3b] ajout du Readme
 1 file changed, 0 insertions(+), 0 deletions(-)
  create mode 100644 README.md
[→ mon_premier_git git:(main) git push -u origin main
Énumération des objets: 4, fait.
Décompte des objets: 100% (4/4), fait.
Compression par delta en utilisant jusqu'à 6 fils d'exécution
Compression des objets: 100% (2/2), fait.
Écriture des objets: 100% (3/3), 339 octets | 339.00 Kio/s, fait.
Total 3 (delta 0), réutilisés 0 (delta 0), réutilisés du pack 0
To https://github.com/MatthieuCodabee/mon_premier_git.git
 4045a68..7603f3b main -> main
la branche 'main' est paramétrée pour suivre 'origin/main'.
→ mon_premier_git git:(main) ]
```

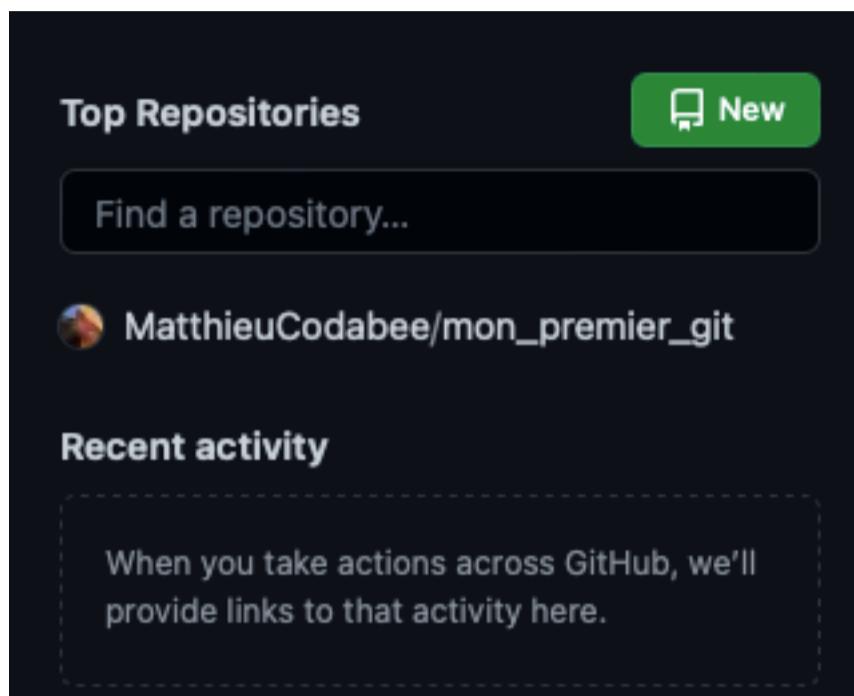
Pour l'instant, Vous ne voyez pas la différence, mais si on édite le Readme, alors la:

- J'ouvre depuis le terminal le fichier avec la commande open README.md
- J'écris Bienvenue sur mon Readme
- J'ajoute les modifications au Staging avec la commande git add
- Je fais un commit « texte dans Readme »
- ... Et Voila! Nous avons notre Readme. Evidemment, il ne fait rien de spécial, mais nous l'avons ajouté et modifié



CRÉER UN DÉPÔT DIRECTEMENT SUR GITHUB

Maintenant, nous allons inverser l'ordre et créer directement le projet sur GitHub. Nous verrons ensuite comment importer ce projet en Local.



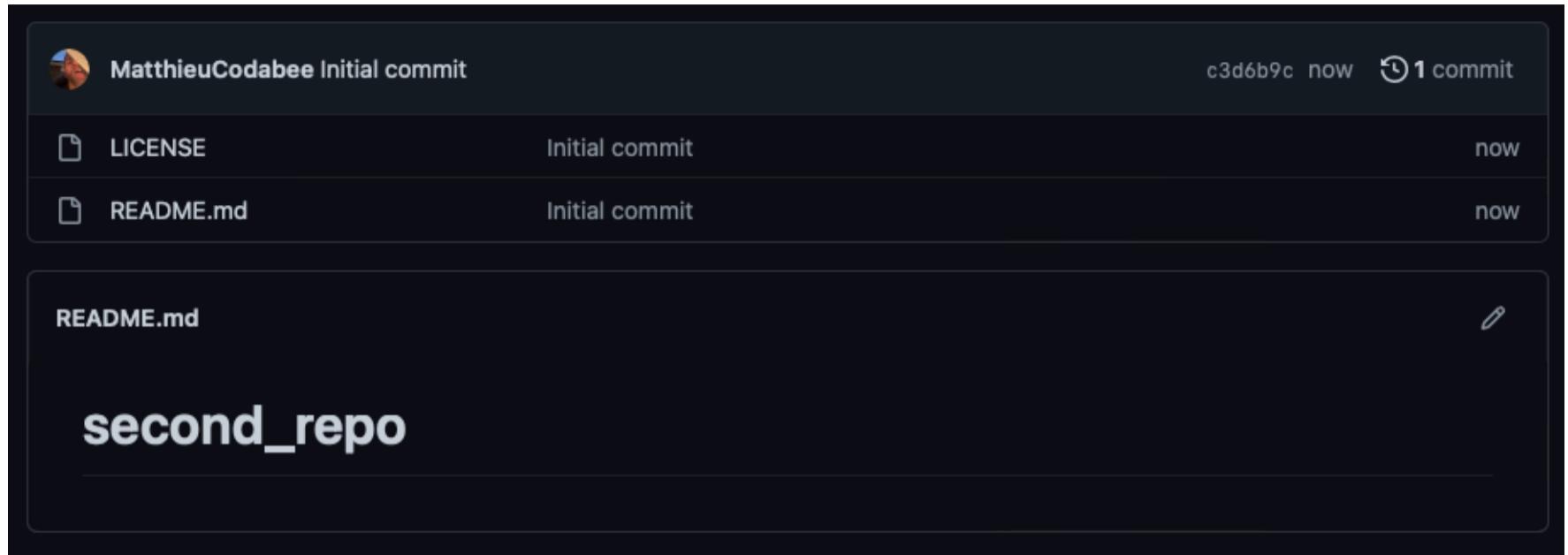
Maintenant qu'un dépôt a déjà été créé, nous avons le bouton New pour en créer un nouveau ! Appuyons dessus et créons tout ceci !

La fenêtre de création sera la même que précédemment, nous allons donner le nom du projet second_repo et il sera public. Nous y ajouterons aussi un readme et une license.

A screenshot of the GitHub "Create repository" form. The form has several fields:

- "Owner *": A dropdown menu showing "MatthieuCodabee".
- "Repository name *": An input field containing "second_repo" with a green checkmark icon.
- "Description (optional)": A text area with a placeholder message.
- "Visibility": A radio button group with "Public" selected (indicated by a blue dot) and "Private" as an option.
- "Initialize this repository with":
 - A checkbox for "Add a README file" which is checked.
 - A note explaining that this is where you can write a long description for your project.
- "Add .gitignore": A note explaining that you can choose which files not to track from a list of templates.
- "Choose a license": A note explaining that a license tells others what they can and can't do with your code.
- A note at the bottom stating: "This will set `main` as the default branch. Change the default name in your [settings](#)".
- An informational note: "You are creating a public repository in your personal account."
- A large green "Create repository" button at the bottom.

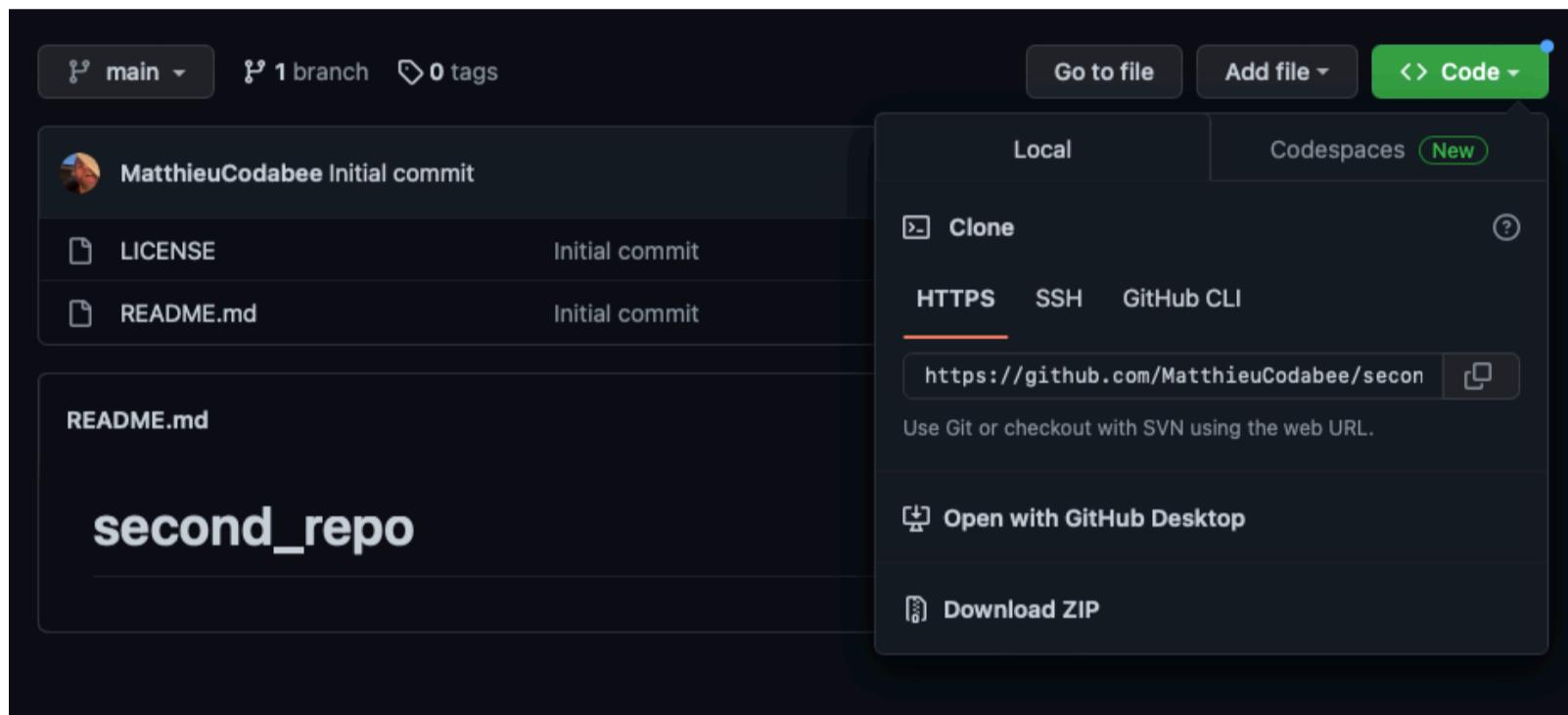
Voila ce que cela donne:



CLONER UN DÉPÔT DISTANT

Maintenant, nous allons vouloir obtenir ce projet en local. On va faire tout ceci directement dans le terminal

1. Déterminez un emplacement où stocker le dépôt: Idéalement au sein d'un dossier dédié
2. Effectuez la commande git clone suivi de l'url du dépôt. L'url se trouve sur le bouton code, puis https



Et c'est tout ! Hyper simple non ?

```
[→ ~ cd Desktop
[→ Desktop mkdir second_repo
[→ Desktop git clone https://github.com/MatthieuCodabee/second_repo.git
Clonage dans 'second_repo'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Réception d'objets: 100% (4/4), fait.
→ Desktop ]]
```

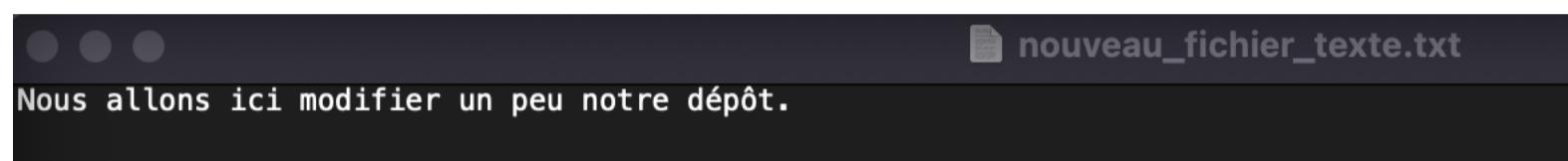
A noter que vous pouvez faire ceci avec n'importe quel dépôt existant sur GitHub pour peu qu'il soit public ou qu'il soit privé mais que vous ayez l'accès.

COMMIT SUR GITHUB DEPUIS UN DÉPÔT CLONÉ

C'est génial, nous avons notre dépôt cloné, nous pouvons désormais le modifier. Etant donné que nous sommes le créateur du dépôt, nous n'avons pas à ajouter de remote.

Je vais donc ajouter un fichier texte simple, puis le modifier

```
[→ second_repo git:(main) touch nouveau_fichier_texte.txt  
[→ second_repo git:(main) × open nouveau_fichier_texte.txt
```



Maintenant, ajoutons le commit et faisons un push

```
[→ second_repo git:(main) × git add -A  
[→ second_repo git:(main) × git commit -m "ajout d'un fichier texte  
[→ second_repo git:(main) git push -u origin main  
Énumération des objets: 4, fait.  
Décompte des objets: 100% (4/4), fait.  
Compression par delta en utilisant jusqu'à 6 fils d'exécution  
Compression des objets: 100% (2/2), fait.  
Écriture des objets: 100% (3/3), 369 octets | 369.00 Kio/s, fait.  
Total 3 (delta 0), réutilisés 0 (delta 0), réutilisés du pack 0  
To https://github.com/MatthieuCodabee/second_repo.git  
  c3d6b9c..62dce44  main -> main  
la branche 'main' est paramétrée pour suivre 'origin/main'.  
→ second_repo git:(main) '
```

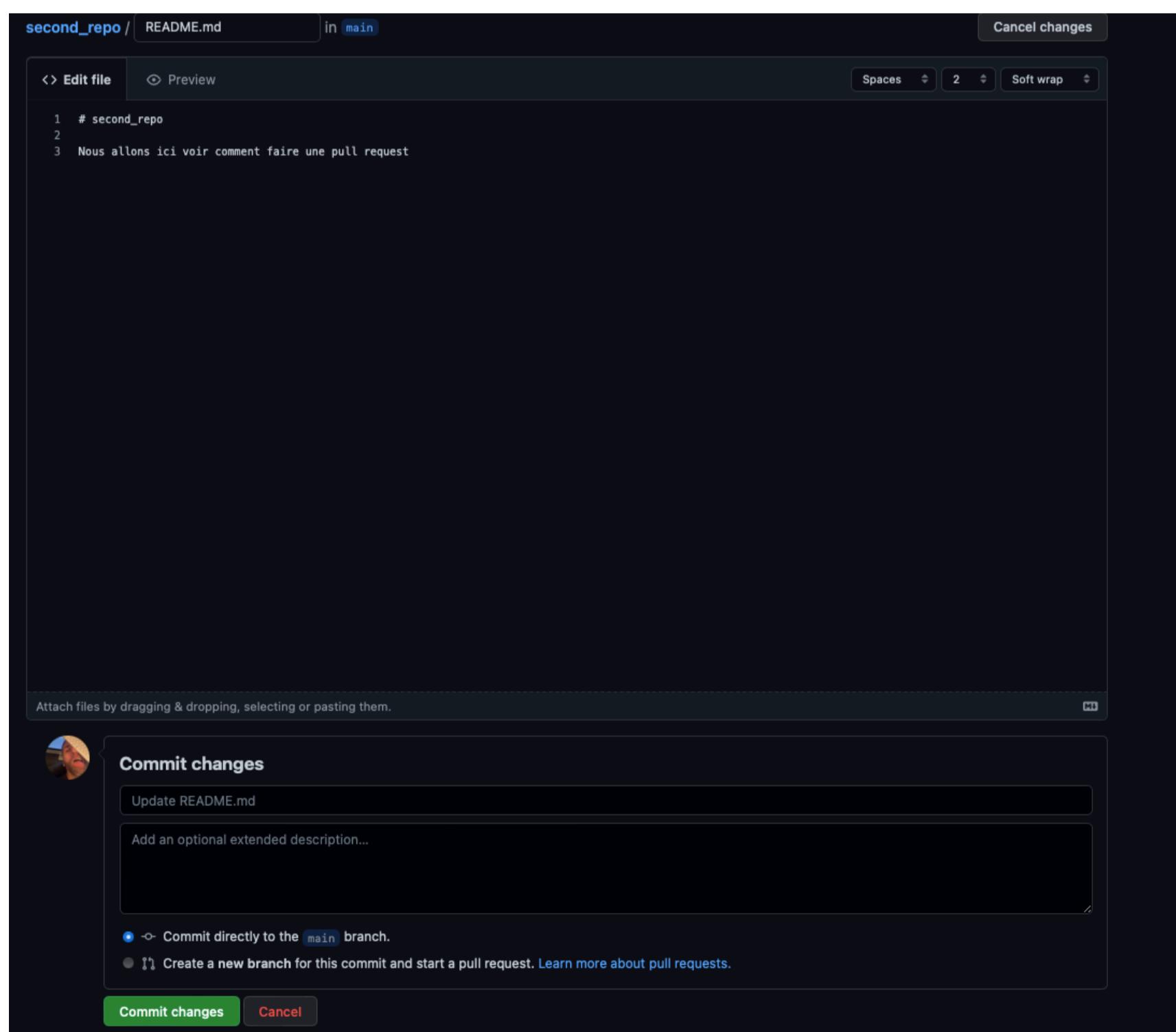
Et voilà le travail ! Notre projet est mis à jour en distant, nous avons pu ajouter et mettre à jour des modifications sur notre travail.

PULL, « TIRER DES MODIFICATIONS »

Il est possible que même pour un projet simple, vous soyez amenés à travailler sur plusieurs machines en même temps. Par exemple, votre ordinateur de bureau mais aussi votre ordinateur perso pour le télétravail !

Le concept de pull request entrera donc en jeu. Cela signifie que au contraire du push où vous poussez des modifications, vous voudrez tirer vers le local des modifications qui sont sur le dépôt distant.

Pour l'exemple, nous allons simplement modifier le Readme directement sur Git. Pour ceci, appuyez sur le crayon en haut à droite du readme et changez le texte. Validez ensuite les changements.



Vous convenez que votre projet en local n'est donc plus à jour !

Il suffit donc de faire un git pull pour obtenir la nouvelle version du dépôt !

```
→ second_repo git:(main) git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Dépaquetage des objets: 100% (3/3), 767 octets | 383.00 Kio/s, fait.
Depuis https://github.com/MatthieuCodabee/second_repo
  62dce44..cf3fc87 main      -> origin/main
Mise à jour 62dce44..cf3fc87
Fast-forward
 README.md | 4 +-+-
 1 file changed, 3 insertions(+), 1 deletion(-)
    . . . . .
```

A noter que le git pull est une commande qui a la fois fait un git Fetch et un git merge. Nous verrons ces concept dans les prochains chapitres, notamment lorsque nous parlerons des systèmes de branches.

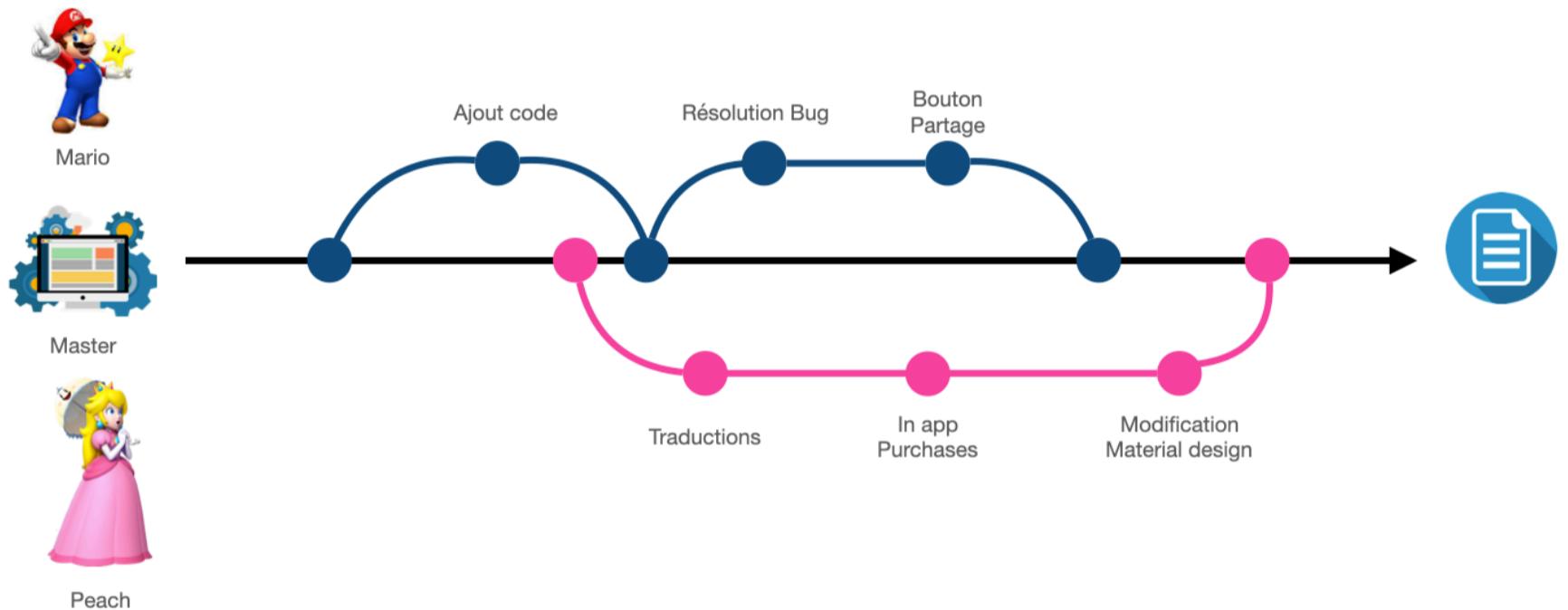
D'ailleurs, si on voyait les branches dès à présent?

BRANCHES ET TRAVAIL EN COLLABORATION

Nous allons ici parler du systèmes de branches. Reprenons le graphique que nous avions plus tôt, lors de notre introduction. Nous avions notre application Mario.

La branche Master était la branche principale. Peach et Mario avaient chacun une branche pour que chacun puisse faire des modifications sans altérer les changements de l'autre.

Chacun pourra ainsi travailler de son côté sans se soucier du code de l'autre.



Nous allons ici dans un premier temps voir comment on peut changer de branche et passer de l'une à l'autre sans aucun soucis.

Puis nous allons voir comment fusionner des branches et ainsi gérer si incompatibilité il y a.

CRÉER UNE BRANCHE ET EN CHANGER SUR GIT

Nous allons reprendre notre premier projet. Celui que nous avions créé pour les bases de Git, à savoir le dépôt mon_premier_git.

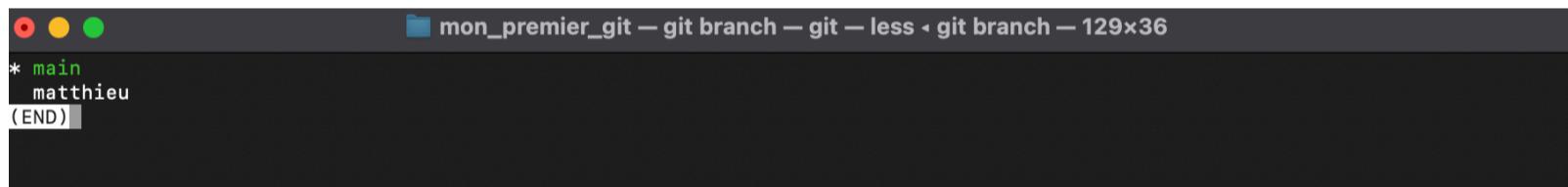
Je vais donc créer une branche Matthieu et m'y rendre avec la commande checkout

```
→ Desktop cd mon_premier_git
→ mon_premier_git git:(main) git branch matthieu
→ mon_premier_git git:(main) git checkout matthieu
Basculement sur la branche 'matthieu'
→ mon_premier_git git:(matthieu) █
```

Comme vous pouvez le voir, j'ai bien changé de branche et j'ai pu m'y rendre. On pourrait revenir en arrière et revenir sur le main

```
→ mon_premier_git git:(matthieu) git checkout main
Basculement sur la branche 'main'
Votre branche est à jour avec 'origin/main'.
→ mon_premier_git git:(main) █
```

Ok, tout ceci est assez simple, maintenant on va taper git branch, nous pourrons ainsi voir les différentes branches disponibles. On peut même voir sur quelle branche nous sommes actuellement:



```
mon_premier_git — git branch — git — less ↵ git branch — 129x36
* main
  matthieu
(END) █
```

Maintenant nous allons effectuer des modifications sur la branche matthieu (ou le nom de votre branche créée) et nous allons modifier le readme. Vous écrirez ce que vous voudrez dans le Readme

```
→ mon_premier_git git:(main) git checkout matthieu
Basculement sur la branche 'matthieu'
→ mon_premier_git git:(matthieu) open README.MD
→ mon_premier_git git:(matthieu) git status
Sur la branche matthieu
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :      README.md

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
→ mon_premier_git git:(matthieu) × git add -A
→ mon_premier_git git:(matthieu) × git commit -m "update Readme"
[matthieu 2793ad9] update Readme
  1 file changed, 1 insertion(+), 1 deletion(-)
```

Nous avons ici effectué des modifications sur la branche matthieu en Local. Nous allons désormais ajouter une branche en distant (remote) et ensuite pousser les changements sur

```
→ mon_premier_git git:(matthieu) git remote add matthieu https://github.com/MatthieuCodabee/mon_premier_git.git
→ mon_premier_git git:(matthieu) git push -u origin matthieu
Énumération des objets: 5, fait.
Décompte des objets: 100% (5/5), fait.
Compression par delta en utilisant jusqu'à 6 fils d'exécution
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (3/3), 303 octets | 303.00 Kio/s, fait.
Total 3 (delta 1), réutilisés 0 (delta 0), réutilisés du pack 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'matthieu' on GitHub by visiting:
remote:     https://github.com/MatthieuCodabee/mon_premier_git/pull/new/matthieu
remote:
To https://github.com/MatthieuCodabee/mon_premier_git.git
 * [new branch]      matthieu -> matthieu
```

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base: main ▾ ← compare: matthieu ✓ Able to merge. These branches can be automatically merged.

update Readme

Write Preview

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Create pull request

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

-o- 1 commit

1 file changed

1 contributor

Commits on Jan 9, 2023

update Readme

MatthieuCodabee committed 11 minutes ago

Showing 1 changed file with 1 addition and 1 deletion.

README.md

@@ -1,2 +1,2 @@
1 Bienvenue sur mon Readme
2 -
2 + C'est genial d'utiliser Git

cette même branche.

update Readme #1

MatthieuCodabee wants to merge 1 commit into `main` from `matthieu`

Conversation 0 Commits 1 Checks 0 Files changed 1 +1 -1

MatthieuCodabee commented now
No description provided.

update Readme 2793ad9

Add more commits by pushing to the `matthieu` branch on [MatthieuCodabee/mon_premier_git](#).

Require approval from specific reviewers before merging
Branch protection rules ensure specific people approve pull requests before they're merged. [Add rule](#)

Continuous integration has not been set up
GitHub Actions and several other apps can be used to automatically catch bugs and enforce style.

This branch has no conflicts with the base branch
Merging can be performed automatically.

Merge pull request You can also open this in GitHub Desktop or view command line instructions.

Write Preview

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

[Close pull request](#) [Comment](#)

Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).
ProTip! Add `.patch` or `.diff` to the end of URLs for Git's plaintext views.

Reviewers
No reviews
Still in progress? Convert to draft

Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Successfully merging this pull request may close these issues.
None yet

Notifications
Customize [Unsubscribe](#)
You're receiving notifications because you're watching this repository.

1 participant

[Lock conversation](#)

NOTRE PREMIERE MERGE

Pull request successfully merged and closed
You're all set—the `matthieu` branch can be safely deleted.

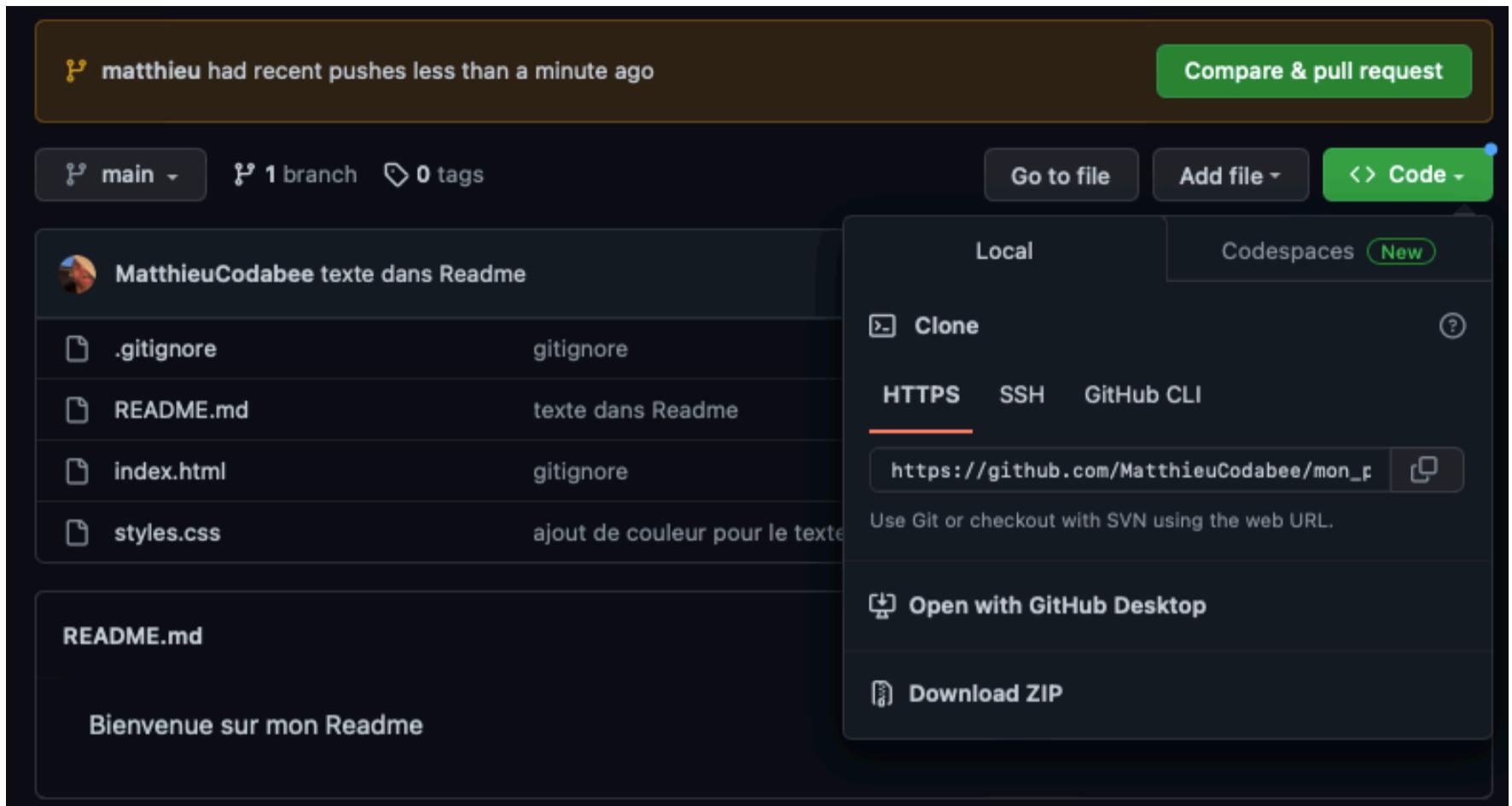
Delete branch

Allons désormais voir sur GitHub ce qu'il s'est passé:

La branche `matthieu` possède des push récents. Allons donc voir ceci.

Une pull request a été créée, montrant les différences ainsi que les erreurs potentielles d'écrasement. En rouge ce qui a été supprimé, en vert ce qui a été ajouté.

Si vous voulez l'ajouter, il suffit de cliquer sur Create a pull request. Faisons le donc !



Elle laisse apparaître une autre page avec d'autres informations comme des notifications de conflits, ou des commentaires à laisser. Nous allons donc faire cette Merge en cliquant sur Merge pull request.

Mais qu'est -ce que cette « Merge ». C'est une fusion des codes entre 2 branches.

Une fois la Marge faite, il est désormais possible de supprimer la branche. Mais nous n'allons pas le faire, nous en auront besoin.

Pour que notre projet en local soit à jour, le mieux est de revenir sur la branche main et de faire un git pull.

```
[→ mon_premier_git git:(matthieu) git checkout main
Basculement sur la branche 'main'.
Votre branche est à jour avec 'origin/main'.
[→ mon_premier_git git:(main) git pull
Depuis https://github.com/MatthieuCodabee/mon_premier_git
  426c35c..2af686f  main      -> origin/main
Mise à jour 426c35c..2af686f
Fast-forward
 README.md | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
[→ mon_premier_git git:(main) open README.md
[→ mon_premier_git git:(main) git status
Sur la branche main
Votre branche est à jour avec 'origin/main'.
rien à valider, la copie de travail est propre]
```

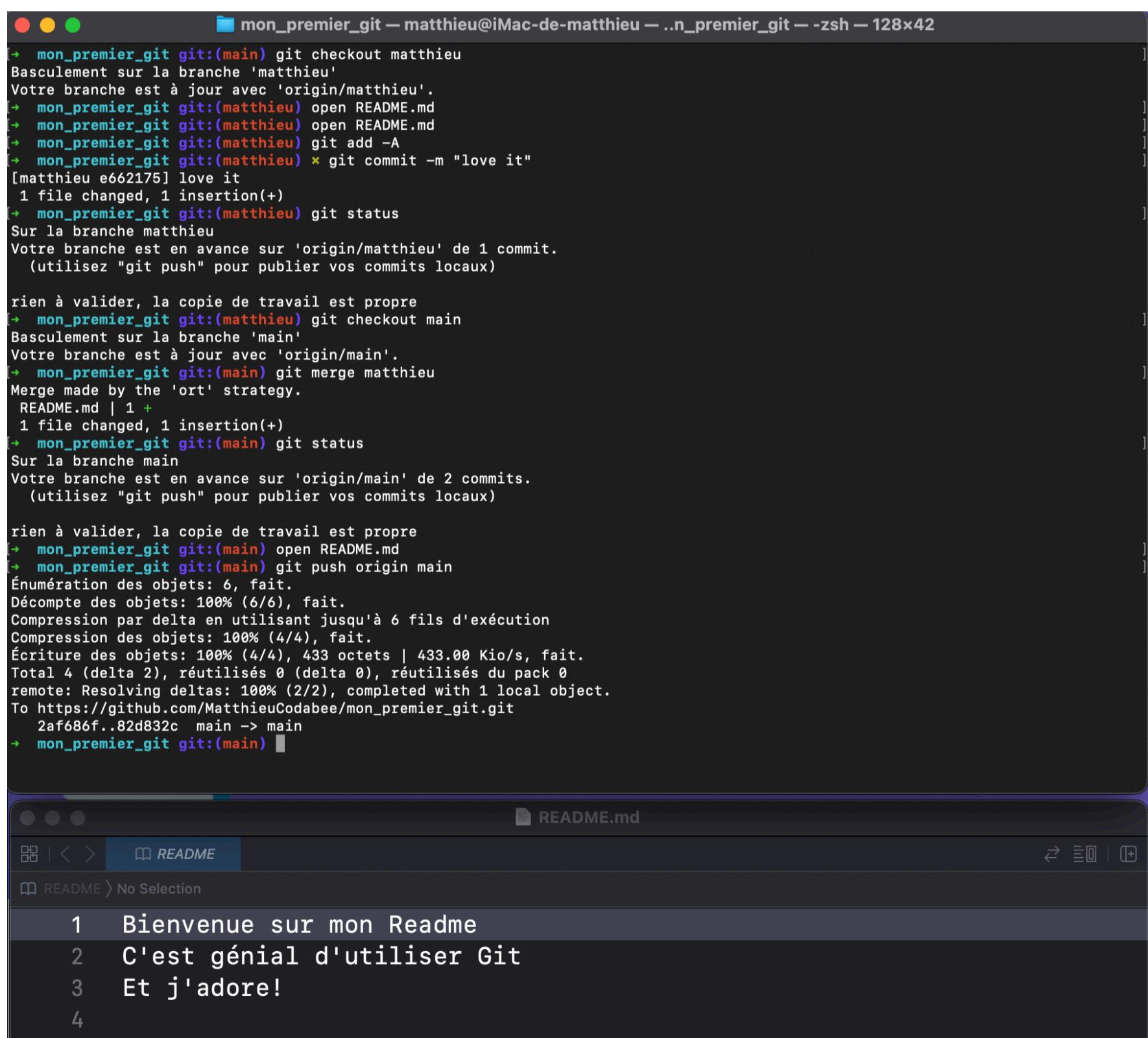
FUSIONNER DES BRANCHES EN LOCAL

Maintenant que nous avons vu ce qu'il se passait sur GitHub, et si nous faisons un changement sur notre branche en local uniquement ?

Vous pouvez désormais lire dans le terminal ce que nous avons fait.

Vous pouvez aussi noter la relation entre git et GitHub qui voient les push à faire pour être à jour.

Cependant, nous noterons aussi l'absence de visuel sur les changements effectués. C'est dommage.



The terminal window shows the following command history:

```
mon_premier_git git:(main) git checkout matthieu
Basculement sur la branche 'matthieu'
Votre branche est à jour avec 'origin/matthieu'.
mon_premier_git git:(matthieu) open README.md
mon_premier_git git:(matthieu) open README.md
mon_premier_git git:(matthieu) git add -A
mon_premier_git git:(matthieu) git commit -m "love it"
[matthieu e662175] love it
1 file changed, 1 insertion(+)
mon_premier_git git:(matthieu) git status
Sur la branche matthieu
Votre branche est en avance sur 'origin/matthieu' de 1 commit.
  (utilisez "git push" pour publier vos commits locaux)

rien à valider, la copie de travail est propre
mon_premier_git git:(matthieu) git checkout main
Basculement sur la branche 'main'
Votre branche est à jour avec 'origin/main'.
mon_premier_git git:(main) git merge matthieu
Merge made by the 'ort' strategy.
 README.md | 1 +
1 file changed, 1 insertion(+)
mon_premier_git git:(main) git status
Sur la branche main
Votre branche est en avance sur 'origin/main' de 2 commits.
  (utilisez "git push" pour publier vos commits locaux)

rien à valider, la copie de travail est propre
mon_premier_git git:(main) open README.md
mon_premier_git git:(main) git push origin main
Énumération des objets: 6, fait.
Décompte des objets: 100% (6/6), fait.
Compression par delta en utilisant jusqu'à 6 fils d'exécution
Compression des objets: 100% (4/4), fait.
Écriture des objets: 100% (4/4), 433 octets | 433.00 Kio/s, fait.
Total 4 (delta 2), réutilisés 0 (delta 0), réutilisés du pack 0
remote: Resolving deltas: 100% (2/2), completed with 1 local object.
To https://github.com/MatthieuCodabee/mon_premier_git.git
  2af686f..82d832c main -> main
mon_premier_git git:(main) |
```

The Finder window below shows the contents of the README.md file:

```
README.md
README
No Selection

1 Bienvenue sur mon Readme
2 C'est génial d'utiliser Git
3 Et j'adore!
4
```

REBASE

Lorsque vous travaillez avec plusieurs branches, il est parfois compliqué de s'y retrouver avec tous ces commits. Pour cela il existe la commande git rebase qui va nous permettre de fusionner les branches pour avoir un historique à plat.

Voici notre historique du main ainsi que celui de matthieu. Nous pouvons voir les commits en ligne ici mais nous avons plusieurs branches. En faisant un rebase nous avons un historique linéaire

```
commit 82d832c1e7482f4f8650f5e76f20d1359a6cba5c (HEAD -> main, origin/main, matthieu/main)
Merge: 2af686f e662175
Author: = <matthieu@codabee.com>
Date:   Mon Jan 9 14:57:47 2023 +0100

    Merge branch 'matthieu'

commit e6621756cc68ffa07af83742c8e780cb70e8d822 (matthieu)
Author: = <matthieu@codabee.com>
Date:   Mon Jan 9 14:57:17 2023 +0100

    love it

commit 2af686f1c8db31f07a3312a0dc9616a0bdcdc2c5
Merge: 426c35c 2793ad9
Author: Matthieu <122023361+MatthieuCodabee@users.noreply.github.com>
Date:   Mon Jan 9 14:44:21 2023 +0100

    Merge pull request #1 from MatthieuCodabee/matthieu

    update Readme

commit 2793ad9eb0c5289766b360d42477fa70701fb7cb (origin/matthieu, matthieu/matthieu)
Author: = <matthieu@codabee.com>
Date:   Mon Jan 9 14:29:01 2023 +0100

    update Readme

commit 426c35c4b29ab7dc7ecab3aa9cae11afc4386764
Author: = <matthieu@codabee.com>
Date:   Mon Jan 9 13:05:38 2023 +0100

    texte dans Readme

commit 7603f3b5bdb1b9ad40ce8325165aed1c3207a3cd
Author: = <matthieu@codabee.com>
Date:   Mon Jan 9 13:00:36 2023 +0100

    ajout du Readme

commit 4045a68ce43330625727e8105eef0af93c9c34bb
Author: = <matthieu@codabee.com>
Date:   Mon Jan 9 12:55:02 2023 +0100

    ajout de couleur pour le texte

commit d8d08aa0e53181bd7e7ffd962d23c9916395bed2
Author: = <matthieu@codabee.com>
:
```

Désormais nous ne voyons plus les différentes branches et tout l'historique est clairement sur le main. Ainsi pour des longues branches ou pour des soucis de fusion future, tout sera extrêmement clair et précis.

```
commit 82d832c1e7482f4f8650f5e76f20d1359a6cba5c (HEAD -> matthieu, origin/main, matthieu/main, main)
Merge: 2af686f e662175
Author: = <matthieu@codabee.com>
Date:   Mon Jan 9 14:57:47 2023 +0100

    Merge branch 'matthieu'

commit e6621756cc68ffa07af83742c8e780cb70e8d822
Author: = <matthieu@codabee.com>
Date:   Mon Jan 9 14:57:17 2023 +0100

    love it

commit 2af686f1c8db31f07a3312a0dc9616a0bdcdc2c5
Merge: 426c35c 2793ad9
Author: Matthieu <122023361+MatthieuCodabee@users.noreply.github.com>
Date:   Mon Jan 9 14:44:21 2023 +0100

    Merge pull request #1 from MatthieuCodabee/matthieu

    update Readme
```

SUPPRIMER UNE BRANCHE

Une fois que le travail est fini sur une branche et que vous n'en avez plus l'utilité, il est possible de la supprimer.

Supprimons Matthieu par exemple. En faisant un git branch , on obtient 2 branches: main et matthieu.

Maintenant, avec la commande git branch -d nomDeLaBranche, nous pouvons la supprimer.

```
[→ mon_premier_git git:(main) git branch -d matthieu
avertissement : branche 'matthieu' non supprimée car elle n'a pas été fusionnée dans
    'refs/remotes/origin/matthieu', même si elle est fusionnée dans HEAD.
erreur : La branche 'matthieu' n'est pas totalement fusionnée.
Si vous souhaitez réellement la supprimer, lancez 'git branch -D matthieu'.
```

Si la branche n'est pas totalement mergée, il faudra remplacer le -d par -D.

```
[→ mon_premier_git git:(main) git branch -D matthieu
Branche matthieu supprimée (précédemment 82d832c).
```

Attention, cette action de suppression entraîne la suppression de tout ce qui n'a pas été commit.

GERER LES CONFLITS DE CODE

Vous connaissez Git sur le bout des doigts désormais, vous gérez GitHub et les branches avec Brio, et si on simulait un travail en collaboration?

Pour ceci, nous allons travailler sur plusieurs branches pour simuler ce que plusieurs personnes pourraient faire ainsi que les conflits que cela pourrait créer.

Nous avons déjà vu pas mal de choses, comme les merges, les branches, etc... Ici ce sera plutôt un cas pratique.

Nous allons reprendre le projet mon_premier_git et:

- Modifier sur GitHub
- Modifier 2 branches différentes
- Gérer les conflits de code
- Corriger des erreurs
- Mettre de coté des modifications

La première chose à faire ici est de créer 2 branches, que nous appellerons brancheA et brancheB. En faisant un git branch, nous les voyons clairement apparaître en plus du main.

```
[→ mon_premier_git git:(main) git branch brancheA
[→ mon_premier_git git:(main) git branch brancheB
[→ mon_premier_git git:(main) ]
```

Désormais ajoutons ces branches en remote Et c'est parti pour les modifications!

```
[→ mon_premier_git git:(main) git remote add brancheA https://github.com/MatthieuCodabee/mon_premier_git.git
[→ mon_premier_git git:(main) git remote add brancheB https://github.com/MatthieuCodabee/mon_premier_git.git
[→ mon_premier_git git:(main) ]
```

Imaginons que brancheA et branche B modifient le Readme toutes les 2 sur la même ligne.

Une fois les 2 modifiées et poussés, nous allons sur GitHub.

Je vais me focaliser ici sur GitHub car lorsque l'on travaille en collaboration, les fichiers sont fusionnées (merge) sur GitHub et non en local. Car nous ne travaillons pas sur la même machine.

```

→ mon_premier_git git:(main) git checkout brancheA
Basculement sur la branche 'brancheA'
→ mon_premier_git git:(brancheA) open README.md
→ mon_premier_git git:(brancheA) git status
Sur la branche brancheA
Modifications qui ne seront pas validées :
  (utilisez "git add <fichier>..." pour mettre à jour ce qui sera validé)
  (utilisez "git restore <fichier>..." pour annuler les modifications dans le répertoire de travail)
    modifié :      README.md

aucune modification n'a été ajoutée à la validation (utilisez "git add" ou "git commit -a")
→ mon_premier_git git:(brancheA) × git add -A
→ mon_premier_git git:(brancheA) × git commit -m "readme branche A"
[brancheA 48f0b09] readme branche A
  1 file changed, 1 insertion(+)
→ mon_premier_git git:(brancheA) git push -u origin brancheA
Énumération des objets: 5, fait.
Décompte des objets: 100% (5/5), fait.
Compression par delta en utilisant jusqu'à 6 fils d'exécution
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (3/3), 351 octets | 351.00 Kio/s, fait.
Total 3 (delta 1), réutilisés 0 (delta 0), réutilisés du pack 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'brancheA' on GitHub by visiting:
remote:     https://github.com/MatthieuCodabee/mon_premier_git/pull/new/brancheA
remote:
To https://github.com/MatthieuCodabee/mon_premier_git.git
 * [new branch]      brancheA -> brancheA
la branche 'brancheA' est paramétrée pour suivre 'origin/brancheA'.

```

```

→ mon_premier_git git:(brancheA) git checkout brancheB
Basculement sur la branche 'brancheB'
→ mon_premier_git git:(brancheB) openReadme.md
zsh: command not found: openReadme.md
→ mon_premier_git git:(brancheB) open README.md
→ mon_premier_git git:(brancheB) git add -A
→ mon_premier_git git:(brancheB) × git commit -m "readme brancheB"
[brancheB af6a56f] readme brancheB
  1 file changed, 2 insertions(+)
→ mon_premier_git git:(brancheB) git push -u origin brancheB
Énumération des objets: 5, fait.
Décompte des objets: 100% (5/5), fait.
Compression par delta en utilisant jusqu'à 6 fils d'exécution
Compression des objets: 100% (3/3), fait.
Écriture des objets: 100% (3/3), 344 octets | 344.00 Kio/s, fait.
Total 3 (delta 1), réutilisés 0 (delta 0), réutilisés du pack 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'brancheB' on GitHub by visiting:
remote:     https://github.com/MatthieuCodabee/mon_premier_git/pull/new/brancheB
remote:
To https://github.com/MatthieuCodabee/mon_premier_git.git
 * [new branch]      brancheB -> brancheB
la branche 'brancheB' est paramétrée pour suivre 'origin/brancheB'.
→ mon_premier_git git:(brancheB) []

```

readme branche A

Write Preview

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Create pull request

① Remember, contributions to this repository should follow our [GitHub Community Guidelines](#).

-o- 1 commit

Commits on Jan 9, 2023

readme branche A
MatthieuCodabee committed 5 minutes ago

Showing 1 changed file with 1 addition and 0 deletions.

Split Unified

v 1 README.md

... ... @@ -1,3 +1,4 @@

1 1 Bienvenue sur mon Readme

2 2 C'est génial d'utiliser Git

3 3 Et j'adore!

4 + Nous allons voir comment travailler en collaboration

Reviewers
No reviews

Assignees
No one—assign yourself

Labels
None yet

Projects
None yet

Milestone
No milestone

Development
Use [Closing keywords](#) in the description to automatically close issues

Helpful resources
[GitHub Community Guidelines](#)

1 contributor

branche A, aucun problème n'est décelé, en effet, nous n'avons pas encore mergé la branche B, qui a modifié la même ligne

Par contre, la seconde branche nous montre un message en rouge

Mais continuons tout de même, allons au bout de la potentielle erreur !

Pour ceci nous allons utiliser le Web editor. Il va ainsi nous montrer visuellement le code qui entre en conflit.

readme brancheB #3

I'm Open MatthieuCodabee wants to merge 1 commit into main from brancheB

Conversation 0 Commits 1 Checks 0 Files changed 1

MatthieuCodabee commented now
No description provided.

readme brancheB af6a56f

This branch has conflicts that must be resolved
Use the [web editor](#) or the [command line](#) to resolve conflicts.
Conflicting files

MERGE PULL REQUEST You can also open this in GitHub Desktop or view command line instructions.

Write Preview

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Close pull request Comment

Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also [compare across forks](#).

base: main ← compare: brancheB × Can't automatically merge. Don't worry, you can still create the pull request.

The screenshot shows a GitHub interface for resolving a conflict in a README.md file. The file contains the following text:

```
1 Bienvenue sur mon Readme
2 C'est génial d'utiliser Git
3 Et j'adore!
4 <<<<< brancheB
5 Nous travaillons sur le même fichier
6
7 =====
8 Nous allons voir comment travailler en collaboration
9 >>>>> main
10
```

Line 4 is highlighted with a red box, indicating it's a conflict from branch B. The GitHub UI includes buttons for '1 conflict', 'Prev ^', 'Next v', and 'Mark as resolved'.

Ici cela donne:

<<<<<< brancheB

Nous travaillons sur le même fichier

=====

Nous allons voir comment travailler en collaboration

>>>>>> main

Nous voyons bien le conflit entre les 2 branches. Nous pouvons ainsi supprimer soit uniquement les commentaires si nous voulons garder le tout, soit supprimer le code que nous ne voulons pas

Ensuite, il suffit d'appuyer marked as resolved et le tour est joué.

On pourrait évidemment faire de même pour tous les fichiers comparer les conflits. Vous pouvez essayer de changer le fichier html à 2 endroits en même temps par exemple !