Git et Github

1/ Découvrir Git et Git/hub Page 2
- Gestionnaire de versions
- Différence entre Git et GitHub

2/ Saisir l'utilité des dépôts distant sur GitHub Page 2

3/ Démarrer son projet avec GitHub Page 3

4/ Travailler depuis son dépôt local Git Page 4
- Appréhender le fonctionnement de git
- Initialiser un dépôt

5/ Appréhender le système de branches Page 7
- Voir les branches
- Créer une branche
- Changer de branche
- Fusionner les branches

6/ Travailler avec un dépôt Git distant, GitHub Page 10
- Récupérer un repository
- Mettre à jour un dépôt localement
- Collaborer avec GitHub (pull request)
- Demander la relecture de son code (code review)

7/ Corriger nos erreurs sur un dépôt local Page 15
- Supprimer une branche
- J'ai modifié la branche principale
- J'ai modifié la branche après avoir fait un commit
- Le message de mon commit est erroné
- J'ai oublié un fichier dans mon dernier commit

8/ Corriger nos erreurs sur un dossier distant Page 18
- Annuler son commit publique
- Les 3 types de reset
- Les conflits
- J'ai ajouté un mauvais fichier dans le commit

9/ Journalisation, corrigé un commit raté Page 12
- Git reflog
- Git blame
- Git cherry-pick

10/ Récapitulation des principales commandes Git Page 22

1/ Qu’est-ce qu’un gestionnaire de versions ?

Un gestionnaire de versions est un programme qui permet aux développeurs de conserver un historique des modifications et des versions de tous leurs fichiers.
L’action de contrôler les versions est aussi appelée "versioning" en anglais, vous pourrez entendre les deux termes.

Cet outil a donc trois grandes fonctionnalités :

1/ Revenir à une version précédente de votre code en cas de problème.

2/ Suivre l’évolution de votre code étape par étape.

3/ Travailler à plusieurs sans risquer de supprimer les modifications des autres collaborateurs.

Git est de loin le système de contrôle de versions le plus largement utilisé aujourd'hui.

Git et GitHub sont deux choses différentes :

Git est un gestionnaire de versions. Vous l’utiliserez pour créer un dépôt local et gérer les versions de vos fichiers.

GitHub est un service en ligne qui va héberger votre dépôt. Dans ce cas, on parle de **dépôt distant** puisqu’il n’est pas stocké sur votre machine.

En résumé :

- Un gestionnaire de versions permet aux développeurs de conserver un historique des modifications et des versions de tous leurs fichiers.
- Git est un gestionnaire de versions tandis que GitHub est un service en ligne qui héberge les dépôts Git. On parle alors de dépôt distant.

2/ Saisir l'utilité des dépôts distants sur GitHub

- Un dépôt est comme un dossier qui conserve un historique des versions et des modifications d’un projet. Il est essentiel pour travailler en équipe ou collaborer à un projet open source.
- Un dépôt local est l’endroit où l’on stocke, sur sa machine, une copie d’un projet, ses différentes versions et l’historique des modifications.

- Un dépôt distant est une version dématérialisée du dépôt local, que ce soit sur Internet ou sur un réseau. Il permet de centraliser le travail des développeurs dans un projet collectif.
- Il existe plusieurs services en ligne pour héberger un dépôt distant, GitHub étant l'un des plus populaires.

3/ Démarrer son projet avec GitHub

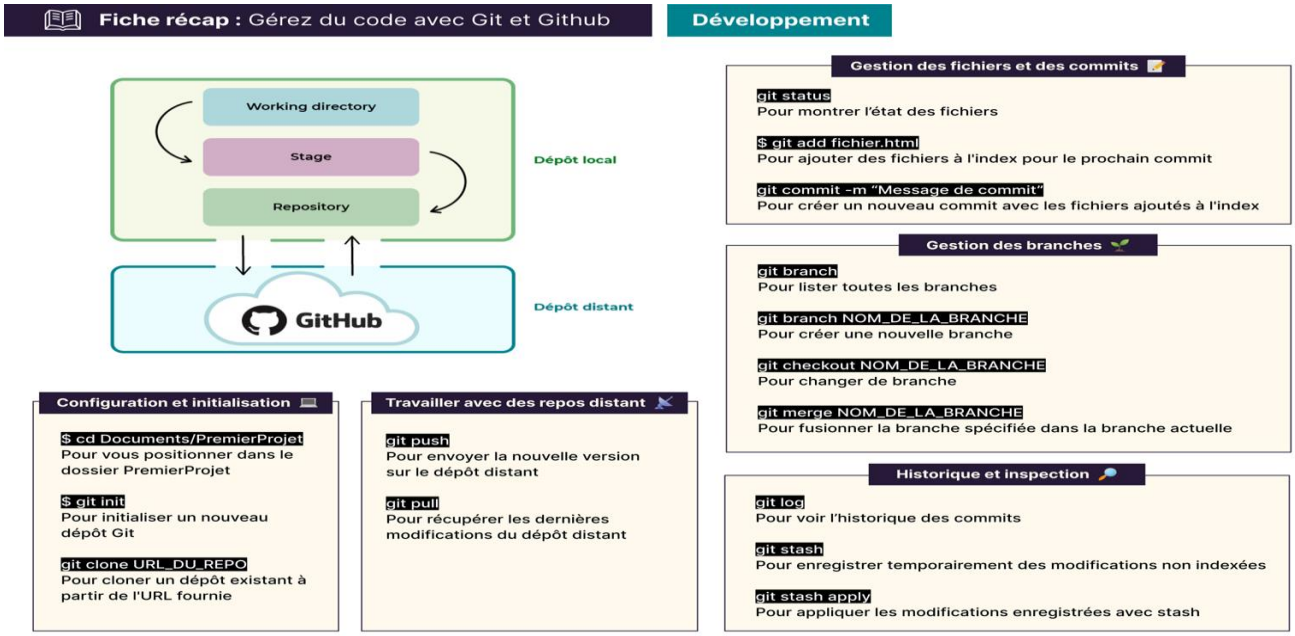
- Pour démarrer un projet, vous devez créer votre compte GitHub.
- Pour mettre votre projet sur GitHub, vous devez créer un repository.
- Vous pouvez suivre vos différents projets facilement grâce au tableau de bord.

Git sur votre ordinateur

- Pour installer Git, vous devez télécharger et configurer Git sur votre ordinateur.
- Pour initialiser un dépôt Git, vous pouvez soit créer un dépôt local vide, soit cloner un dépôt distant.
- Git init permet d'initialiser un projet Git.

4/ Travaillez depuis votre dépôt local Git

Appréhendez le fonctionnement de Git



Ce schéma représente le fonctionnement de Git. Il est composé de 3 zones qui forment le dépôt local, et du dépôt distant GitHub.

Regardons plus en détail les différentes zones du dépôt local.

Le Working directory

Cette zone correspond au dossier du projet sur votre ordinateur.

Le Stage ou index

Cette zone est un intermédiaire entre le working directory et le repository. Elle représente tous les fichiers modifiés que vous souhaitez voir apparaître dans votre prochaine version de code.

Le Repository

Lorsque l'on crée de nouvelles versions d'un projet, c'est dans cette zone qu'elles sont stockées.

Les 3 zones sont donc présentes dans votre ordinateur, en local.

En-dessous, vous trouvez le repository **GitHub**, c'est-à-dire votre dépôt distant.

Initialisez un dépôt

Aller dans son dossier et faire un git init

c:/user/Cyril/Document/MonProjet

$ git init

On créé nos fichiers, par exemple index.html et style.css

Je peux maintenant indexer mes fichiers choisis :

$ git add index.html style.css

ou pour indexer tous mes fichiers :

$ git add .

Maintenant que nos fichiers modifiés sont indexés, vous pouvez créer une version, c'est-à-dire archiver le projet en l'état. Pour ce faire, utilisez la commande "git commit" :

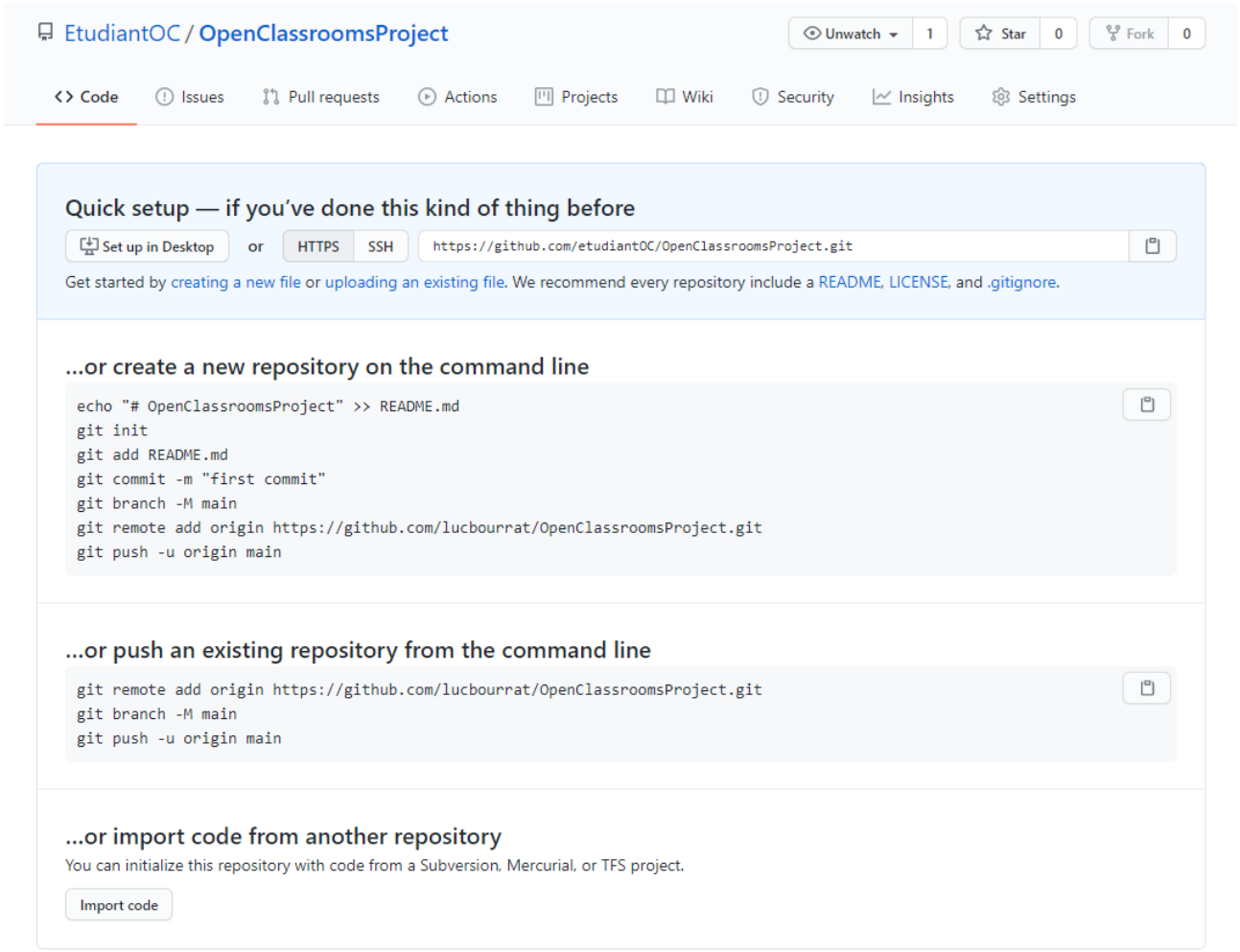
$ git commit -m ‘’Ajout des fichier html et css’’

“Ajout des fichiers html et css” est le message rattaché au commit grâce à l'argument "-m".

Envoyez votre commit sur le dépôt distant avec la commande **git push**

Pour commencer, vous allez devoir **“relier” votre dépôt local au dépôt distant** que vous avez créé sur GitHub précédemment. Pour cela :

- Sur votre page d’accueil GitHub cliquer sur new
- Donner un nom et cliquer sur Create repository



Maintenant, retournons sur Git Bash, vérifiez que vous êtes bien dans votre repository et tapez la commande suivante afin de relier votre dépôt local a votre gitHub:

$ git remote add origin (puis l'adresse de votre dossier git hub que vous trouver en haut a droite de la photo)

Changer sa branche en main

$ git branch -M main

Ça y est ! Vous avez relié le dépôt local au dépôt distant. Vous pouvez donc envoyer des commits du repository vers le dépôt distant GitHub en utilisant la commande suivante :

$ git push -u origin main

En résumé

- git add permet d'ajouter des fichiers dans l'index, qui est une zone intermédiaire dans laquelle stocker les fichiers modifiés.

- git commit permet de créer une nouvelle version avec les fichiers situés dans l'index.
- git commit -m permet de créer une nouvelle version et de préciser le message rattaché au commit.
- git push permet d'envoyer les modifications faites en local vers un dépôt distant.

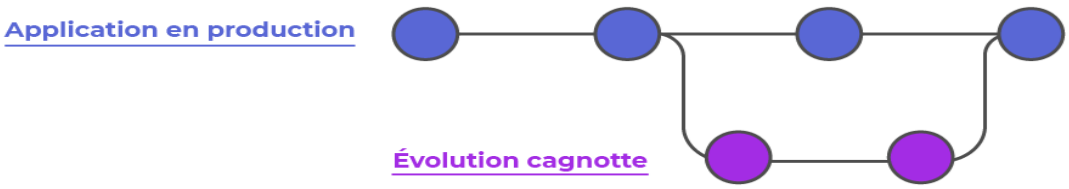
5/ Appréhendez le système de branches

Le principal atout de Git est son système de branches. C'est sur ces branches que repose toute la magie de Git !

Les différentes branches correspondent à des copies de votre code principal à un instant T, où vous pourrez tester toutes vos idées les plus folles sans que cela impacte votre code principal.

Sous Git, la branche principale est appelée la **branche main**, ou **master**

La branche principale (main ou master) portera l'intégralité des modifications effectuées. Le but n'est donc pas de réaliser les modifications directement sur cette branche, mais de les réaliser sur d'autres branches, et après divers tests, de les intégrer sur la branche principale.



On dit qu'un bon développeur est toujours fainéant ! En fait, un bon développeur trouvera toujours une technique simple pour faire le travail à sa place. Eh bien, Git est l'outil idéal dans ce cas. Il va créer une branche virtuelle, mémoriser tous vos changements, et seulement quand vous le souhaitez, les ajouter à votre application principale. Il va vérifier s'il n'y a pas de conflits avec d'autres fusions, et hop, le tour est joué !

Pour connaître les branches présentes dans notre projet, il faut taper la ligne de commande :

git branch

Dans un premier temps, vous n'aurez que :

git branch

*** main**

Et c'est normal. L'étoile signifie que c'est la branche sur laquelle vous vous situez, et que c'est sur celle-ci qu'actuellement vous réalisez vos modifications.

Je vous conseille fortement de créer une branche si votre modification est lourde, compliquée, ou si elle risque d'avoir des impacts importants sur votre projet.

Création d'une nouvelle branche (ex : cagnotte) :

git branch cagnotte

Maintenant si je fais :

git branch

*** main**

cagnotte

Pour basculer de branche, nous allons utiliser :

git checkout cagnotte

et maintenant :

git branch

main

*** cagnotte**

La branche va fonctionner comme un dossier virtuel. Avec **git checkout**, on va être téléporté dans le dossier virtuel Cagnotte. On reste dans le dossier principal physiquement, mais virtuellement nous sommes passés dans un monde parallèle !

Vous pouvez désormais réaliser votre évolution sans toucher à la branche main qui abrite votre code principal fonctionnel. Vous pouvez re-basculer si besoin à tout moment sur la branche main, sans impacter les modifications de la branche Cagnotte.

Vous avez réalisé des évolutions sur la branche Cagnotte, il faut maintenant demander à Git de les enregistrer, de créer une nouvelle version du projet comprenant les évolutions réalisées sur la branche Cagnotte.

Vous devez créer un "commit" grâce à la commande :

$ git commit -m ‘’Réalisation de la partie cagnotte’’

Avec la commande git commit, nous avons enregistré des modifications en local, uniquement. Il ne vous reste plus qu’à envoyer les modifications réalisées sur le dépôt distant grâce à la commande git push.

Fusionnez votre travail avec la commande git merge

À présent, il faut intégrer l'évolution réalisée dans la branche "cagnotte" à la branche principale "main". Pour cela, vous devez utiliser la commande "git merge".

Cette commande doit s'utiliser à partir de la branche dans laquelle nous voulons apporter les évolutions. Dans notre cas, la commande s’effectuera donc dans la branche main. Pour y retourner, utilisez la commande :

$ git checkout main

Maintenant que vous êtes dans votre branche principale, vous pouvez fusionner votre branche "cagnotte" à celle-ci grâce à la commande suivante :

$ git merge cagnotte

En résumé

- Une branche est une “copie” d’un projet sur laquelle on opère des modifications de code.
- La branche main (ou anciennement master) est la branche principale d’un projet.
- git checkout permet de basculer d’une branche à une autre.
- git merge permet de fusionner deux branches.

6/ Travaillez avec un dépôt distant

Imaginons que vous deviez travailler sur un projet avec des amis. Ces derniers ont créé le repository sur GitHub. Il est temps pour vous de récupérer le code pour apporter vos modifications :

Tout d'abord, vous allez récupérer l'URL du dépôt distant : cela se passe sur GitHub !

Cliquez sur le bouton "Code" pour copier le dépôt

git clone (puis coller l'URL récupéré sur GitHub)

Mettez à jour le dépôt en local

Imaginons que durant la semaine, un de vos amis ait ajouté des modifications sur la branche main et que vous souhaitiez les récupérer. Comment faire ?

On s'assure d'être dans le bon dossier (dépôt) puis on fait :

$ git pull origin main

Vous devriez maintenant avoir l'ensemble des fichiers et dossiers du repository à jour dans votre répertoire courant.

Collaborez sur GitHub

Commençons par vérifier les différentes branches du projet : vos collègues travaillent peut-être sur de nouvelles fonctionnalités. Pour nous en assurer, nous pouvons utiliser la commande avec

$ git branch

Nous avons vu que **git merge** nous permettait de fusionner les modifications de notre branche avec la branche principale.

Mais dans un contexte professionnel, c'est un peu plus compliqué ! 😀 Lorsque vous travaillez en équipe sur un repository, la branche principale est souvent bloquée. Vous ne pouvez pas pusher directement votre code sans qu'il soit vérifié. Vous ne pouvez donc pas fusionner vos modifications vous-même !

On fait ce qu'on appelle une **pull request** !

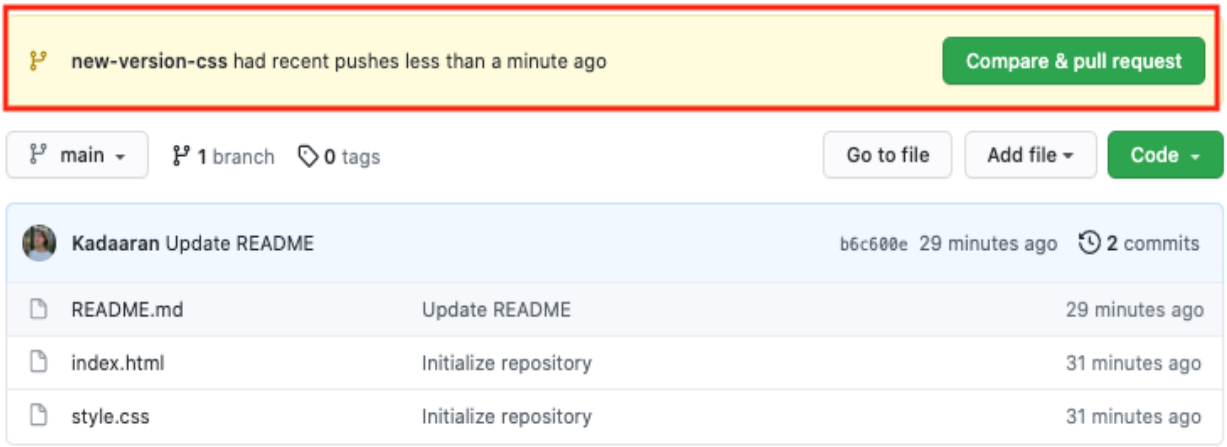
Une **pull request**, ou *demande de pull*, en français, est une fonctionnalité de GitHub qui permet de demander aux propriétaires d'un repository l'autorisation de fusionner nos changements sur la branche principale ou toute autre branche sur laquelle on souhaite apporter nos modifications.

Donc si nous créons une pull request, nous avons au préalable :

1. Créé une nouvelle branche.
2. Envoyé votre code sur cette même branche.

Lorsque ces deux conditions sont remplies, un bandeau apparaît à l'écran pour vous suggérer de créer une pull request.

Rendez-vous sur la page GitHub :

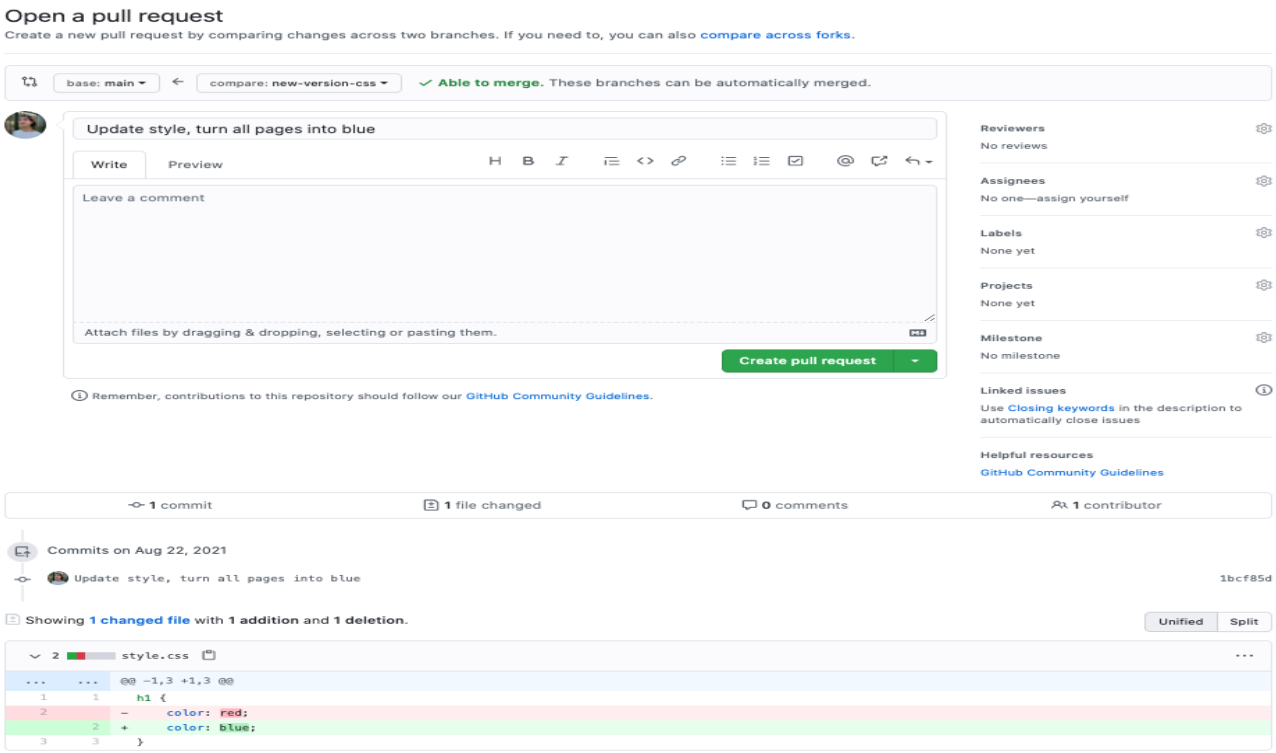


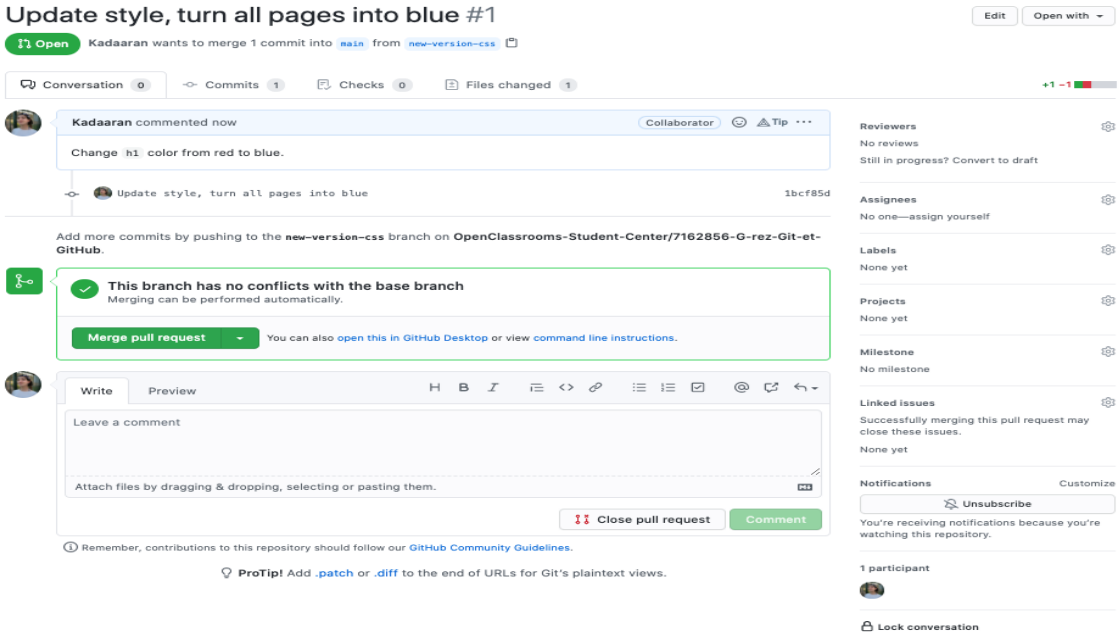
Ajoutez un commentaire pour expliquer les raisons de vos modifications.

Ici, votre modification consiste à changer la couleur d'une balise. Je vous conseille donc de commenter : "Change h1 color from red to blue".

GitHub indique les modifications effectuées par un code couleur. Les lignes en rouge indiquent une suppression, et les lignes vertes une addition. Ici on voit bien qu'il y eu un changement sur une ligne, l'attribut red a été supprimé, et a été remplacé par blue.

- Cliquez sur **Create pull request** pour valider la pull request.
- Et voilà une belle pull request prête à l'emploi :





Notre **pull request** est créée et peut être fusionnée avec la branche main en cliquant sur le bouton “**Merge Pull Request**”.

Résumé pour faire une Pull Request

Etape 1 : Assurez- vous d’être à jour avec la branche principale

Avant de commencer, récupérez les dernières mises à jour de la branche principale (généralement main ou master).

$ git checkout main

$ git pull origin main

Etape 2 : Créez une nouvelle branche pour votre pull request et se positionner sur cette branche

Créez une branche distinct pour vos modifications

$ git branch nom_de-votre_branche

$ git checkout nom_de-votre_branche

Ou pour aller plus vite en créant notre nouvelle branche et se positionner directement dessus

$ git checkout -b nom_de-votre_branche

Etape 3 : Ajouter et valider vos changements

Ajoutez vos fichiers modifiés ou nouveaux, puis validez les changements.

$ git add .

$ git commit -m’’Description de vos modifications’’

Etape 4 : Pousser votre branche vers GitHub

Poussez votre branche locale vers le dépôt distant sur GitHub.

$ git push origin nom_de-votre_brance

Etape 5 : Créez la pull request sur GitHub

Rendez-vous sur la page du dépôt sur GitHub. Vous devriez voir un bouton "Compare & pull request" à côté de votre branche récemment poussée. Cliquez dessus.

Etape 6 : Soumettez la pull request

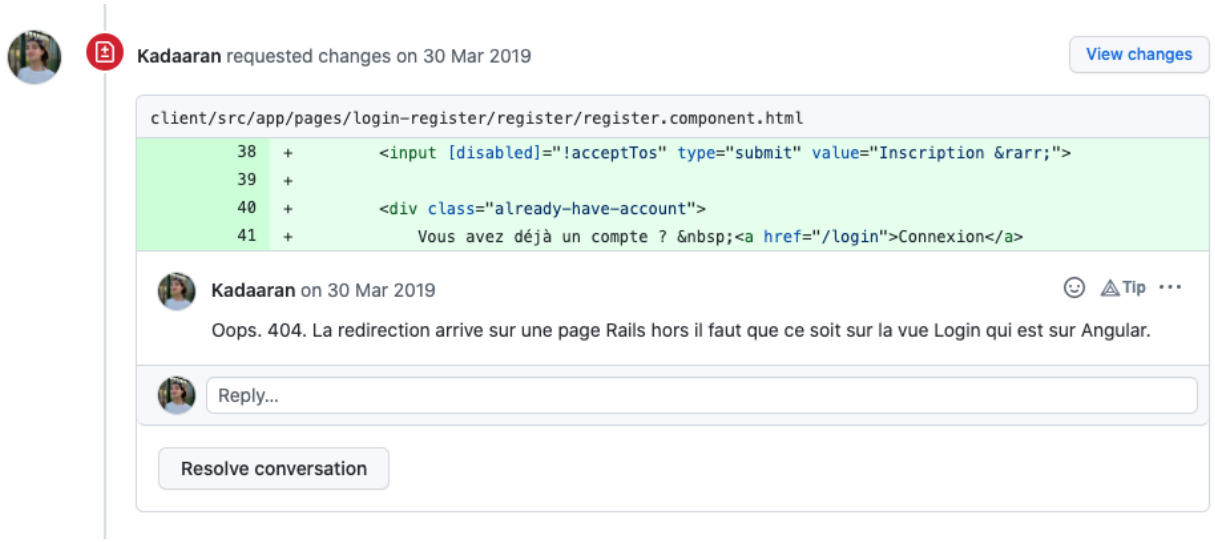
Ajoutez une description détaillée de vos modifications, puis soumettez la pull request.

Etape 7 : Attendez la révision et la fusion

Les propriétaires du projet examineront votre Pull Request. Ils peuvent poser des questions ou demander des modifications supplémentaires. Une fois approuvée, elle peut être fusionnée dans la branche principale.

Demandez une relecture de code

Sur des projets d'envergure, il peut arriver que votre code ne puisse être fusionné sur la branche principale sans être relu et validé par d'autres membres du projet. C'est ce qu'on appelle une Code Review, ou revue de code, en français. Cela permet de prévenir les erreurs éventuelles, de discuter sur un choix, une prise de position ou même de poser des questions.



En résumé

- Sur GitHub, nous pouvons récupérer l'URL d'un dépôt distant.
- git clone permet de copier en local un dépôt distant.
- git remote add permet de lier un dépôt à un "nom court", pour une plus grande facilité d'utilisation.
- git pull permet de dupliquer un dépôt GitHub en local.
- Une Pull Request permet de demander à fusionner votre code sur la branche principale.

7/ Corriger nos erreurs sur un dépôt local

Git est un outil merveilleux, mais on a vite fait de créer une branche alors qu'on ne le souhaitait pas, de modifier la branche principale ou encore d'oublier des fichiers dans ses commits. Mais ne vous inquiétez pas, nous allons voir ensemble que toutes ces petites erreurs ne sont pas difficiles à corriger avec les bonnes techniques.

Grâce à la ligne de commande ls -la, vous pouvez faire apparaître les dossiers cachés.

Un des avantages majeurs de Git réside dans l'aspect local des travaux réalisés : un dépôt Git gère son cycle de vie localement, indépendamment de la connectivité avec son dépôt distant. Tout se passe directement sur votre ordinateur.

J'ai créé une branche que je n'aurais pas dû créer

Pour la supprimer, on reste ou on se met sur la branche principale puis :

$ git branche -d brancheTest (nom de la branche à supprimer)

$ git branche -D brancheTest (si la branche n'est pas vide, utiliser -D)

J'ai modifié la branche principale

L'erreur est humaine et il peut arriver de modifier une branche principale par erreur... Dans ce cas, ne paniquez pas !

Si vous avez modifié votre branche principale (main ou master) avant de créer votre branche et que vous n'avez pas fait le commit, ce n'est pas bien grave. Il vous suffit de faire une **remise ou un stash** en anglais.

La **remise, ou stash**, permet de mettre vos modifications de côté, les ranger, le temps de créer votre nouvelle branche et d'appliquer cette remise sur la nouvelle branche.

Vous pouvez à tout moment voir l'état dans lequel sont vos fichiers, c'est-à-dire voir les changements qui ont été indexés ou ceux qui ne l'ont pas été, avec la commande suivante :

$ git status

Créez un stash avec la commande suivante :

$ git stash

Assurez-vous que votre branche principale soit de nouveau propre, en faisant un nouveau $ git status

Et finalement, vous pouvez appliquer le stash pour :

- Récupérer les modifications que vous avez rangées dans le stash.
- Appliquer ces modifications sur votre nouvelle branche.

$ git stash apply

Cette commande va appliquer le dernier stash qui a été fait.

Si pour une raison ou une autre, vous avez créé plusieurs stash, et que le dernier n'est pas celui que vous souhaitez appliquer, pas de panique, il est possible d'en appliquer un autre..

En premier lieu, regardez la liste de vos stash avec la commande suivante :

$ git stash list

Cette commande va vous retourner un "tableau" des stash avec des identifiants du style :

stash@{0}: WIP on main: 51928dd création de la branche master

Il suffira alors d'appeler la commande git stash en indiquant l'identifiant.

$ git stash apply stash@{0}

Et voilà, le tour est joué !

J'ai modifié la branche après avoir fait un commit

Maintenant, admettons que vous ayez réalisé vos modifications et qu'en plus vous ayez fait le commit. Le cas est plus complexe, puisque vous avez enregistré vos modifications sur la branche principale, alors que vous ne deviez pas.

Pour réparer cette erreur, vous devez analyser vos derniers commits avec la fonction **git log**. Vous allez alors récupérer l'identifiant du commit que l'on appelle couramment le hash.

Par défaut, git log va vous lister par ordre chronologique inversé tous vos commits réalisés.

$ git log

commit d257e1523274bf232a2ccce16383b3ff2cddc60b (HEAD -> main)

Author: Cyril Pholoppe <cyril.pholoppe@gmail.com>
Date: Fri Nov 24 14:29:13 2023 +0100

    test

Maintenant que vous disposez de votre identifiant, gardez-le bien de côté. Vérifiez que vous êtes sur votre branche principale et réalisez la commande suivante :
$ git reset --hard HEAD^

Cette ligne de commande va supprimer de la branche principale notre dernier commit. Le HEAD^ indique que c'est bien le dernier commit que nous voulons supprimer. L'historique sera changé, les fichiers seront supprimés.

Je crée une nouvelle branches

git branch brancheCommit

Pour appliquer ce commit sur la nouvelle branche je fais $ git reset avec l'identifiant (SHA) du commit en question.

Il n'est pas nécessaire d'écrire l'identifiant (SHA) en entier. Seuls les **8 premiers caractères** sont nécessaires.

Git reset --hard

$ git reset --hard d257e152

HEAD is now at d257e1 création de la branche master
Et voilà, le tour est joué !

Le message de mon commit est erroné

Lorsque l'on travaille sur un projet avec Git, il est très important de marquer correctement les modifications effectuées dans le message descriptif. Cependant, si vous faites une erreur dans l'un de vos messages de commit, il est possible de changer le message après coup.

L'exécution de cette commande, lorsqu'aucun élément n'est encore modifié, vous permet de modifier le message du commit précédent sans modifier son instantané. L'option -m permet de transmettre le nouveau message.

$ git commit --amend "Mon nouveau message"

je vérifie avec git log

$ git log

commit d257e1523274bf232a2ccce16383b3ff2cddc60b (HEAD -> brancheCommit)

Author: Cyril Pholoppe <cyril.pholoppe@gmail.com>
Date: Fri Nov 24 13:48:09 2023 +0100

    Mon nouveau message

J'ai oublié un fichier dans mon dernier commit

$ git add fichierOublier.txt

$ git commit --amend --no-edit

[brancheCommit 5cc654c] Mon nouveau message
Date: Fri Nov 24 13:48:09 2023 +0100
2 files changed, 0 insertions(+), 0 deletions(-)
create mode 100644 PremierFichier.text
create mode 100644 fichierOublier.txt

Votre fichier a été ajouté à votre commit et grâce à la commande –no-edit que vous avez ajoutée, vous n'avez pas modifié le message du commit.

`git commit --amend`

vous permet de sélectionner le dernier commit afin d'y ajouter de nouveaux changements en attente. Vous pouvez ajouter ou supprimer des changements afin de les appliquer avec `commit --amend`

En résumé

- git branch -d permet de supprimer une branche.
- git status permet de voir l'état des fichiers.
- git stash enregistre les modifications non indexées pour une utilisation ultérieure.
- git log affiche l'historique des commits réalisés sur la branche courante.
- git reset --hard HEAD^ permet de réinitialiser l'index et le répertoire de travail à l'état du dernier commit.
- git commit --amend permet de sélectionner le dernier commit pour y effectuer des modifications.

8/ Corrigez vos erreurs sur votre dépôt distant

La journée a été difficile et par mégarde vous avez pushé des fichiers erronés.

Il est possible d'annuler son commit public avec la commande **git revert**. L'opération revert annule un commit en créant un nouveau commit. C'est une méthode sûre pour annuler des changements, car elle ne risque pas de réécrire l'historique du commit.

$ git revert HEAD^

Nous avons maintenant reverté notre dernier commit public et cela a créé un nouveau commit d'annulation. Cette commande n'a donc aucun impact sur l'historique ! Par conséquent, il vaut mieux utiliser

git revert : pour annuler des changements apportés à une **branche publique**

et **git reset** pour faire de même, mais sur une **branche privée**.

Gardez à l'esprit que **git revert** sert à annuler des changements commités, tandis que **git reset HEAD** permet d'annuler des changements non commités.

Toutefois, attention, **git revert** peut écraser vos fichiers dans votre répertoire de travail, il vous sera donc demandé de commiter vos modifications ou de les remiser.

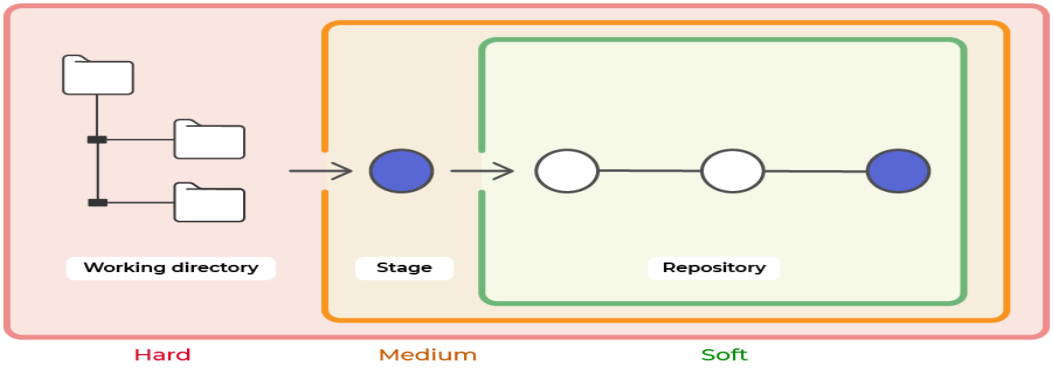
Utilisez git reset

Imaginez que votre client vous demande une nouvelle fonctionnalité ; vous travaillez dessus toute la journée et le lendemain, finalement, il change d'avis. Catastrophe !

Vous avez perdu une journée à développer une fonctionnalité pour rien, mais en plus il faut que vous trouviez le moyen de revenir en arrière ! Heureusement, notre ami Git arrive à notre rescousse avec la commande **git reset** !

Les trois types de réinitialisation de Git

a commande **git reset** est un outil complexe et polyvalent pour annuler les changements. Elle peut être appelée de trois façons différentes, qui correspondent aux arguments de ligne de commande **--soft, --mixed et --hard**.



Git reset --hard.

Si vous voulez exécuter cette commande, vérifiez 5 fois avant de la lancer et soyez sûr de vous à 200 %.

git reset notreCommitCible --hard

Cette commande permet de revenir à n'importe quel commit mais en oubliant absolument tout ce qu'il s'est passé après ! Quand je dis tout, c'est TOUT !

Git reset –mixed

va permettre de revenir juste après votre dernier commit ou le commit spécifié, sans supprimer vos modifications en cours. Il permet aussi, dans le cas de fichiers indexés mais pas encore commités, de désindexer les fichiers.

Si rien n'est spécifié après git reset, par défaut il exécutera un git reset –mixed HEAD~

Le HEAD est un pointeur, une référence sur votre position actuelle dans votre répertoire de travail Git.

Git reset –soft

Cette commande permet de se placer sur un commit spécifique afin de voir le code à un instant donné, ou de créer une branche partant d'un ancien commit. Elle ne supprime aucun fichier, aucun commit, et ne crée pas de HEAD détaché.

Oups, j'ai des conflits !

Nous avons vu comment fusionner des branches, en utilisant un exemple assez simple où tout s'est bien terminé. Malheureusement, il arrive parfois, même souvent, que cela ne se passe pas aussi bien, et que des conflits apparaissent.

Ce conflit, vous allez devoir le résoudre en ouvrant le fichier avec votre éditeur habituel, VSCode par exemple.

Maintenant que vous avez résolu le conflit, il vous reste à le dire à Git !

Git add leBonFichier git commit

Git va détecter que vous avez résolu les conflits et va vous proposer un message de commit.

J'ai ajouté le mauvais fichier au commit

Avec Git nous avons une super fonction qui va remonter le temps. La commande **git revert** vous permet de revenir à l'état précédent, tout en faisant un deuxième commit. Au lieu de supprimer le commit de l'historique du projet, elle détermine comment annuler les changements introduits par le commit et ajoute un nouveau commit avec le contenu ainsi obtenu. Vous allez donc revenir à l'état précédent mais avec un nouveau commit. Ainsi, Git ne perd pas l'historique, lequel est important pour l'intégrité de votre historique de révision et pour une collaboration fiable.

Quelle est la différence entre git reset et git revert ?

Git reset va revenir à l'état précédent sans créer un nouveau commit, alors que

git revert a créer un nouveau commit

En résumé

- git revert HEAD^ permet d'annuler un commit en créant un nouveau commit.

- git reset est une commande puissante. Elle peut être appliquée de 3 façons différentes (--soft; --mixed; --hard).
- La commande git merge produit un conflit si une même ligne a été modifiée plusieurs fois. Dans ce cas, il faut indiquer à Git quelle ligne conserver.
- git reset permet de revenir à l'état précédent sans créer un nouveau commit.
- git revert permet de revenir à l'état précédent en créant un nouveau commit.

9/ Journalisation, corriger un commit raté

Les techniques de journalisation de Git ont été prévues pour répondre à ce genre de situation.

La **journalisation,** ou *history*, en anglais, désigne l'enregistrement dans un fichier ou une base de données de tous les événements affectant une application. Le journal (en anglais *log file* ou plus simplement *log*), désigne alors le fichier contenant ces enregistrements.

Un trou de mémoire ? git reflog !

Va loguer les commits ainsi que toutes les autres actions que vous avez pu faire en local : vos modifications de messages, vos merges, vos resets, enfin tout, quoi .
Pour revenir à une action donnée, on prend les 8 premiers caractères de son SHA (identifiant) et on fait :

git reflog

git checkout e789e7c

Qui s’est amusé dans mon dépôt ? git blame

Si vous découvrez un bug dans votre projet, vous allez avoir besoin d'identifier son origine et de savoir qui a modifié chaque ligne du code.

permet d’examiner le contenu d’un fichier ligne par ligne et de déterminer la date à laquelle chaque ligne a été modifiée, et le nom de l’auteur des modifications

$ git blame monfichier.php

git blame va afficher pour chaque ligne modifiée :

- son ID ;
- l'auteur ;
- l'horodatage ;
- le numéro de la ligne ;
- le contenu de la ligne.

Il me faut ce commit ! Vite git cherry-pick

Lorsque vous travaillez avec une équipe de développeurs sur un projet de moyenne à grande taille, la gestion des modifications entre plusieurs branches de Git peut devenir une tâche complexe. Parfois, vous ne voulez pas fusionner une branche entière dans une autre et vous n'avez besoin que de choisir un ou deux commits spécifiques. Ce processus s'appelle cherry-pick !

Ici, nous allons prendre les deux commits ayant pour SHA d356940 et de966d4, et nous les ajoutons à la branche principale sans pour autant les enlever de votre branche actuelle. Nous les dupliquons !

$ git cherry-pick d356940 de966d4

En résumé

- git log affiche l'historique des commits réalisés sur la branche courante.
- git reflog est identique à git log. Cette commande affiche également toutes les actions réalisées en local.
- git checkout un_identifiant_SHA-1 permet de revenir à une action donnée.
- git blame permet de savoir qui a réalisé telle modification dans un fichier, à quelle date, ligne par ligne.
- git cherry-pick un_identifiant_SHA-1 un_autre_identifiant_SHA-1 permet de sélectionner un commit et de l'appliquer sur la branche actuelle.

Un petit bonus

Si la branche a été push il y a un moment, elle devient stale, c'est-à-dire qu'elle est considérée comme inactive et GitHub ne la retourne plus dans la liste. Pour résoudre ce problème, lancez la commande suivante :

$ git branch -r

10/ Récapitulation des principales commandes Git

git init : initialiser dans dossier git en local

git clone : cloner un repository de GitHub

git add : indexer

git add remote (URL) : relier notre dépôt à notre GitHub

git commit -m ‘’monMessage’’ : comitter

git push -u orgin main : envoyer son commit sur GitHub

git pull -u origin main : mettre à jour son dossier local

git branch : voir les branches

git branch maBranche : créer une branche

git checkout : changer de branche

git merge : fusionner les branches

git branch -d : supprime une branche vide

git branch -D : supprime une branche non vide

git status : permet de voir l'état des fichiers

git stash : mettre les informations de cotés

git stash apply : apllique le dernier stash

git stash list : affiche le tableau des stash

git stash apply stash@{0} : afficher le stash du tableau indiqué par numéro, là exemple 0

git log : voir tout les commits

git commit --amend : permet de sélectionner le dernier commit pour y effectuer des modifications.

git commit --amend ‘’nouveauMessage’’ : permet de modifier le message de notre commiter

git reset --hard HEAD^ : permet de réinitialiser l'index et le répertoire de travail à l'état du dernier commit.

git commit --amend –no-edit : ajoute un fichier oublié à mon dernier commit

git revert HEAD^ permet d'annuler un commit en créant un nouveau commit.

git reflog : est identique à git log. Cette commande affiche également toutes les actions réalisées en local.

git blame : permet de savoir qui a réalisé telle modification dans un fichier, à quelle date, ligne par ligne.

git cherry-pick : git cherry-pick un_identifiant_SHA-1 un_autre_identifiant_SHA-1 permet de sélectionner un commit et de l'appliquer sur la branche actuelle

git reset : permet de revenir à l'état précédent sans créer un nouveau commit.

git revert :permet de revenir à l'état précédent en créant un nouveau commit.